Final Report

Fernando Gabriel Gutiérrez Madrigal A01424790

Inés Alejandro García Mosqueda A00834571

Oliver Josel Hernández Rebollar A01641922

Paul Enrique Alonso Ramírez A01634608

Design of Advanced Embedded Systems

Group 601

Tecnológico de Monterrey, Campus Guadalajara

December 6, 2023

Table of Contents

Problem Statement

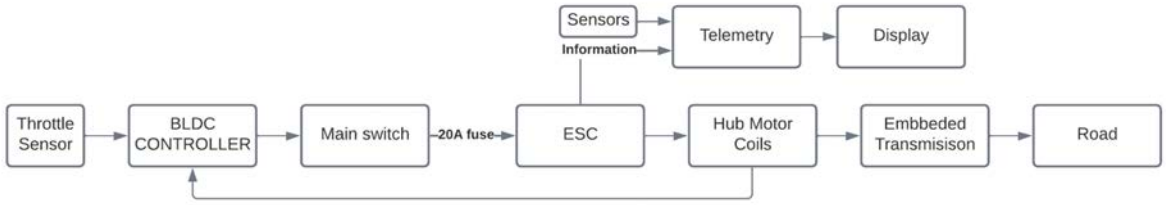
Telemetry is a new technology that helps the user of a vehicle with sensing physical variables and providing feedback for a better handling of the vehicle as well as preventing accidents. A telemetry system in formula SAE is important due to the conditions of the race and the pilot experience, the only problem with that situation is the manufacture of the car, measuring from the most basic variables such as speed, temperature and acceleration to even the pressure at the tires and temperature of each of the tires and breaks, even the compression of the suspension. Another important part of this systems is the intercommunication of the devices. The objective is to implement a telemetry system very close to what can be found at a formula SAE car or a commercial car implemented in an ebike with different variables sensed and intercommunication of the devices using CAN protocol.

Project Management

System Design and Requirements

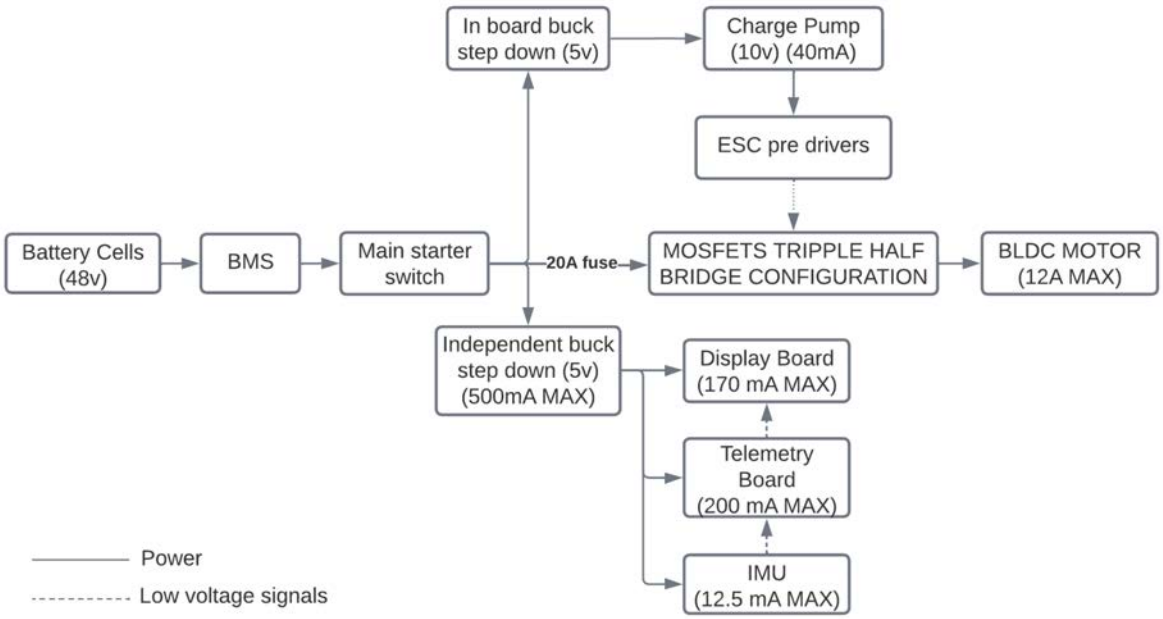
Basic Control and energy flow diagram

The throttle sensor inputs variable power to the motor. The chosen hub motor, with an internal planetary transmission, converts electrical energy into motion by energizing coils, spinning the wheel, and transferring mechanical energy to the road. The motor returns the hall sensors signals to the controller.



Basic system components block diagram

The system diagram includes key components: the battery cells provide 48v to the battery pack's BMS, which operates within a 46v to 52v range, cutting off power outside this range. A 20A fuse before the ESC's power stage prevents overloads, protecting motor coils and other components like mosfets and predrivers.



Product Features Description

	Feature	Description	Nice to have or must have
1	Motor commutation control	The essential part of the project, for the movement of the vehicle is the motor commutation part, the commutation algorithm will lead the microcontroller to write the appropriate signals to the MOSFETS based on the motor position.	Must Have
2	Fault state detection and correction	As a safety measure sensing the values of the current and voltage as well as the acceleration measured by each ax, the motor is driving more than the battery's rated voltage or current, the system will stop, as well, if IMU presents values greater than the average reference it will assume the user has had an accident.	Must Have
3	Speed measurement	Using one of the hall sensors as an encoder, the speed will be determined by measuring the RPMs and calculate the speed converting the RPMs to kmph based on wheel size.	Must Have
5	Control Power Management	Throttle potentiometer will give a reference for the microcontroller to give a determined power by taking the amount of voltage measured at the	Must Have

		throttle signal pin but the system still needs an algorithm to manage the power delivered to the motor and follow a reference, as well, it was a requirement as class outlines established, so the implementation of a basic control algorithm is needed.	
6	CAN communication	One microcontroller is capable of many tasks, nevertheless it will not be able to handle the screen rendering, sense and process other variables, then CAN communication for different microcontroller is planned to use, as well, research showed that CAN protocol will simply interconnection of the microcontrollers physically, since it only requires 2 cables and 2 resistors at the most basic configuration. This is a very common protocol in the automotive industry where a fast, reliable and physical communication with low limitations is required.	Nice to have
7	DSP implementation	As a part of the class outline DSP was mentioned to be included at the project, the plan of the implementation in the project is for a filter for the signals of the IMU, a signal filter was already included at the hall sensors signals part, (please go to page 60 for more information about the filter implemented), IMU will form part of the fault detection to prevent accidents and stop the system if user experiments an accident like falling or hitting something.	Must Have
8	GPS velocity	Speed measured by the previous mentioned method with the hall sensors as an encoder, is a method for a fast and reliable speed measurement, but it is not the real speed, a columnist from the wall street journal and many research papers find the same conclusion, that mechanical losses and variables involved in the car movement will make the speed determined by the rpms less accurate than the gps speed. GPS measurement is difficult, since it requires to measure the coordinates based on certain time units and finding the displacement of the user in that gap.	Nice to have
9	TPMS information	Tire Pressure Management System (TPMS) is becoming a very common and useful technology for cars, having the tire pressure all time without the necessity of stopping to measure it is a new safety measure so unexpected punctures or low	Nice to have

		tyre situations will not convert into a destroyed tyre and prevent the car to get out of control due to the low pressure, then it was considered a good variable to be sensed to prevent accidents, since tpms works with wireless protocols, and the most accessible sensors had no standard protocols, then team decided to establish it as a nice to have.	
10	RF Communication	Radio Frequence communication between the car and a station is considered necessary to measure the vehicle performance from a station and see the stats it gets, so experiments and testing is not limited to get the information from the on board computers directly and it can be accessed on real time, although this feature is considered a nice to have since the system will still perform the same if it is or not implemented.	Nice to have

Planning

Parts Requirements Planning

Product/Part	Requirements	Priority	Estimated Price
Motor	In order to move the user with a desired speed of 40 kmph, the recommended rating for the motor was 48v 750w at least, hub motor is considered in order to facilitate the assembly.	Very high	$$$
Battery	Due to the motor specs and desired 40km range the recommended battery is at least a 48v nominal battery pack with 10Ah of capacity and a discharge rate of 30A, so BMS will not be limited in high current demand situations, it will also help to help to not force the BMS to work at its maximum capacity.	Very high	$$$$
CAN module	Around 1mbps of speed for the transceptor and capable of working at 5v so step down and board voltage regulator will be capable of powering it with no problem.	Normal	$
TFT display	User must be able to see the information displayed in the TFT even with daylight and it should be able to display basic information at	High	$$

	least the current speed and the temperature, then the minimum size for the display would be 2.4in and minimum brightness,		
IMU	At least a 3 axis IMU should be acquired in order to provide information of the sudden movements the user makes as well as having a small form factor and being able to work with 3.3v.	Normal	$$$
ESC	The ESC mush have at least the capability of handling the power for our motor, being 48v 750w, and ensure the correct motion of the motor as well as protection for the battery.	Very High	$$$
TPMS	TPMS should be able to measure the bike's tire pressure being around 25 to 45 psi and work with RF signals so the information given can be decoded.	Below Normal	$$
RF modules	RF modules should support a minimum speed of 1 Mbps and operate efficiently at 5V, ensuring seamless integration with the vehicle's power system.	Below Normal	$$
Current Sensor	Current sensor must be able to read at least 15A maximum in order to ensure a good operating range as well as being non invasive, so system will have no losses from it. It should also deliver a 0 to 5v signal to the MCU.	High	$
Voltage Sensor	Voltage sensor must be able to stand at least 52v, since battery cells work at 52v when they are at their maximum capacity, it should deliver as well a 0 to 5v signal with less than 20mA current to the MCU ADC.	Normal	$
Temperature Sensors	Temperature sensors should be able to measure from 10 to even 70 degrees Celsius, which is the maximum temperature brakes should get, they should not be sensitive to vibrations and have a small form factor.	Below Normal	$

Priority order Symbology:
Very High: Criticall for the project core functionalities and or features
High: Registered in the course outlines as requirements
Normal: Non registered in the course outlines, helpful for the core functions and features
Below Normal: Non critical for the core functions and non listed as requirement in the outline

Estimated Price Symbology:
$$$$: Pricey, not acquirable with own team resources (Above 3000 MXN)
$$$: Pricey, Still Acquirable by the team (up to 2 units) (Below 3000 MXN, above 800 MXN)
$$: Average, (Acquirable (up to 4 units) (Below 800 MXN, above 200 MXN)
$: Inexpensive, Acquirable with no difficulties (Below 200 MXN)

Tasks Description

PCB Design: Develop the design of the schematic and layout for the PCB to use in the project
PCB Manufacturing: Place the order at the manufacture site and wait for the deliver
Adapter PCB design: design of the adapter PCB so testing can start as soon as possible
Adapter PCB manufacture: Due to time constraints the adapter manufacturer was determined to be in a fast prototyping pcb router.
BLDC commutation code implementation: write and flash the code for the phase commutation in order to move the motor properly.
Basic cabling and test bench setting for the BLDC commutation code: manufacture and do the cable management for a test bench for the BLDC motor in order to test it without any risk.
BLDC commutation code validation: Test and verify the motor movement as well as power consumption in order to validate the code implemented for the commutation sequence.
Sensor data acquisition code implementation: connect and enable individually the available sensors in order to test their functionality and verify the utility of the data delivered as well as limiting it only to the necessary data.
Validation of the sensors data: Confirm the reliability of the data delivered by the sensors

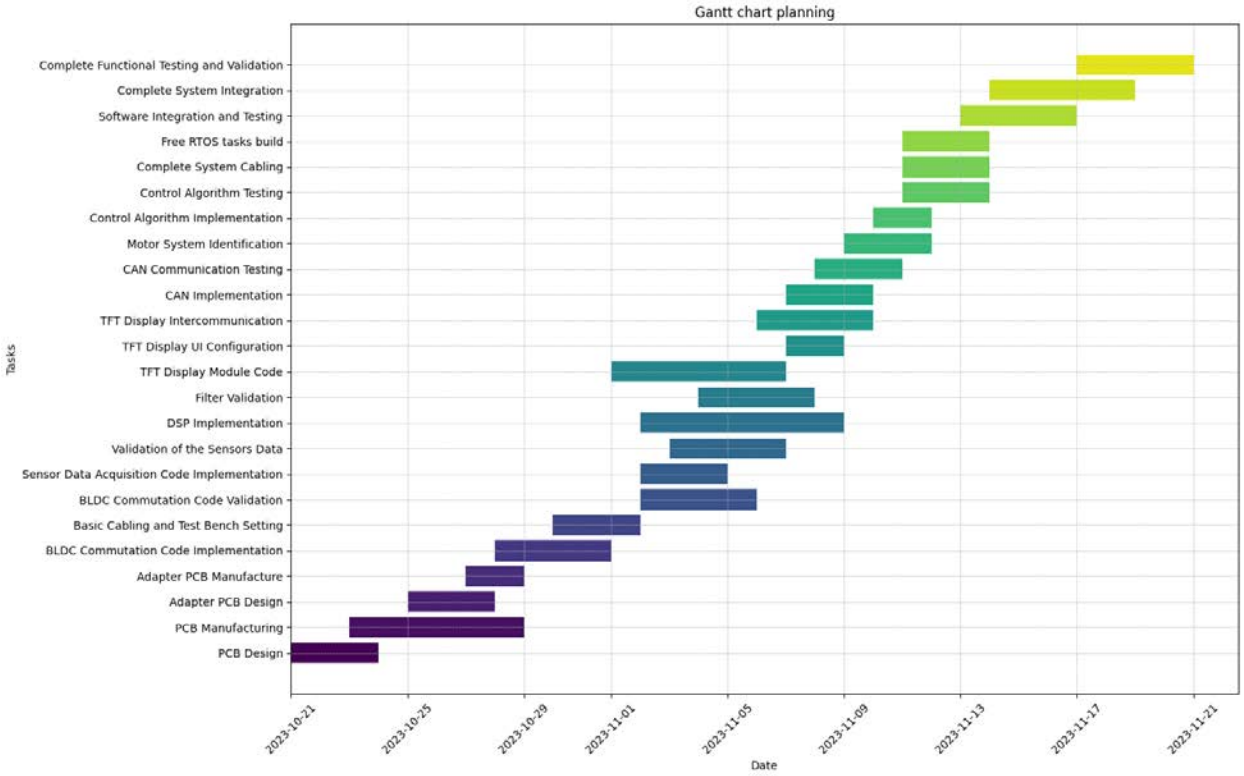
- **Speed:** MCU decodes correct speed with small variations and has no floating states
- **Voltage:** MCU reads the voltage that is being consumed at that node
- **Current:** MCU reads a current at the range established
- **IMU:** IMU is retrieving the acceleration at 3 different axes and it detects the difference between the common vibrations and a sudden shake.

DSP implementation: Filter theoretical design and coefficients retreive to write the code implementation and deploy it to the development board.
Filter validation: Test and validate the signals given the designed and implemented filter, changes if necessary must be done.
TFT display module code: Writing code for the TFT display module in order to turn on the display and be able to communicate with the user properly.
TFT display IU configuration: The graphical elements on the tft display are placed for a better user experience so user is actually able to see the information even though the distance and conditions of the environment.
TFT display intercommunication and interfacing: Establishing intercommunication for the TFT display will ensure the flow of information across different parts of the system to the one that will communicate with the user.
CAN implementation: This task consists in wire the CAN modules and configure them so we use the desired speeds and microcontrollers are able to exchange information.
CAN communication testing and validation: Testing the CAN communication, ensuring every signal is clear and every message reaches its destination without hindrance.

Motor System identification: Identifying the motor system consists in gathering essential information to understand and optimize the motor's performance.
Control algorithm implementation: Implementing the control algorithm with the brief background we already have by using one of the most common control algorithms to provide a better performance and consistency of the motor.
Control algorithm testing and validation: Validate the control algorithm is behaving normally and that it will not make sudden and aggressive changes, in this task the objective is also to tune the constants and coefficients for the algorithm so it behaves even better.
complete system cabling: Completing the system cabling, plugging everything and making the necessary splices when needed, every cable plays a crucial role in the integrity and functionality of the entire system.
Free RTOS tasks build: Consider each of the necessary tasks in terms of software to plan and develop the tasks for the implementation of free RTOS in the MCU
Software integration and testing: Integrate all of the tasks and routines for the application and ensure that none of them is making the others to fail or make the MCU to be overloaded.
Complete system integration: Once software and hardware is validated by parts, the integration will consists into plug everything and test they are receiving the correct signals and all modules as well as tasks are performing as intended and not causing any side effects into another routine of the system, as well checking if hardware design is no limiting any device and prevent any possible failure due to overheating or excess of current consumption.
Complete functional testing and validation: In order to ensure the system's functionality and reliability a complete and functional test must be performed with possible fail scenarios, where system must behave with secure responses as well as shut down in fault case, the objective is not to do an intense stress situation for the system but to ensure that it will not represent a risk in certain situations for the user.

Software Documentation: https://github.com/Ineso1/Conversion-de-bici-electrica-
Control algorithm Implementation: https://github.com/Ineso1/Conversion-de-bici-electrica-/blob/main/BLDC_Controller_RTOS_V1/Core/Src/freertos.c

Gantt Diagram



Team decided to start with the hardware part, since that platform is the one that would let the team to successfully and easily test the software, due to the time constraints and delivery times it was necessary to make the adapter pcb, original plan took the vast majority of the time to develop the main code, and test, tft display modulo took longer to configure due to the available libraries and bit shifting, some tasks were decided to be distributed in parallel to distribute the team into the tasks and use the time in a better way.

Contingency Actions

The original gantt chart took was planned for the best case where the ordered stm32 bluepills PCBs arrived on time, since they took longer to arrive the team had to go for an alternative plan, at least to start writing code and be able to test it, that is why the Adapter PCB manufacture was done, the adapter PCB helped with adapting the pinout of the raspberry pi pico to the stm 32 nucleo 64 board, which now was able to be used with the available controller board.
CAN implementation took more days than the original plan, it represented a problem since the workload increased, but solved by dedicating more daily hours to the project, since rtos implementation took less than expected, the team were able to finish the project in time.

Hardware design

Sensors Implemented

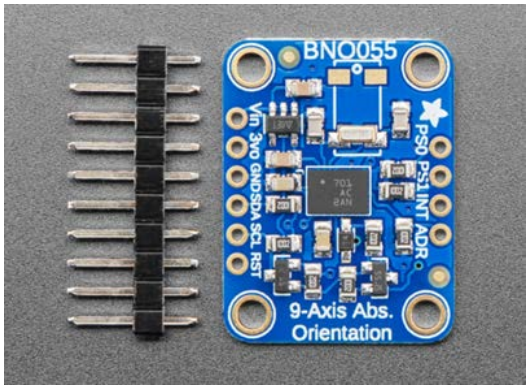
Sensors implemented:
IMU Sensor (BNo055 Adafruit) - This sensor, which uses I2C communication, is typically used for motion tracking. It combines accelerometer, gyroscope, and magnetometer data to determine orientation and movement.
CAN Communication Module (MCP2515) - This module facilitates communication over a Controller Area Network (CAN), often used in automotive and industrial applications for data transmission between various microcontrollers and devices.
Infrared Temperature Sensor (Hiletgo GY-906 MLX90614ESF) - Using I2C communication, this sensor measures temperature without direct contact by detecting infrared radiation emitted by objects.
Overvoltage Sensor with Voltage Divider Component (0/48V to 0/5V mapping with 2.2k and 47k resistors) - This setup is used for analog reading of voltages. It scales down high voltages (up to 48V) to a lower range (0 to 5V) to be safely read by sensors or microcontrollers.
Current Sensor (INA 181) - Also for analog reading, this sensor measures electrical current passing through a conductor.
Buck Step-Down Converter (K78U05-500R3) - This converter reduces voltage from a higher level (48V) down to a lower level (5VDC), useful for powering devices that require lower voltages.
Charge Pump (C1240A with an array of 1uF capacitors) - This circuit is used to increase or decrease voltage using capacitors. It's often used in low-power applications.
TPMS Pressure Sensor (ANGGREK Tire Pressure Monitoring System, TPMS Monitor C240) - This sensor is part of a system that monitors the air pressure inside pneumatic tires on various types of vehicles**.**

Electronic Components Specifications:

Sensor de presión TPMS (ANGGREK Sistema de Monitoreo de Presión de Neumáticos, TPMS Monitor C240)

Especificación:
Material: ABS
Tipo: digitales
Modelo: C-240
Frecuencia de trabajo: 433.9200MHz
Voltaje de trabajo: 5V
Corriente de trabajo (corriente estática): ≤30uA
Corriente de trabajo (corriente dinámica): ≤8mA
Ambiente de trabajo: -40 ℃ - + 85 ℃
Rango de detección de presión: 0~99 psi, 0~5 bar
Vida útil de la batería: Sensor externo de 3 años

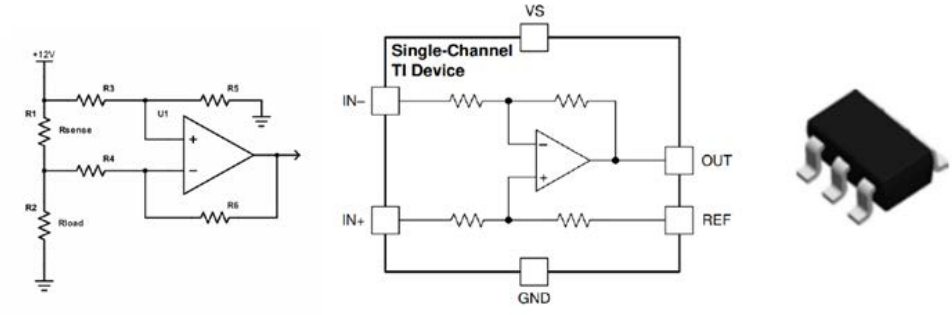
IMU Sensor (BNO055 Adafruit) - I2C Communication



Key Features: This sensor is a fusion of an accelerometer, gyroscope, magnetometer, and an onboard microcontroller. Its compact dimensions are 3.8mm x 5.2mm x 1.1mm. It offers various power management modes including normal, low power, and suspend, making it suitable for diverse applications. The sensor operates effectively in a wide temperature range from -40°C to +85°C. It features digital bidirectional I2C and UART interfaces for versatile communication options.
Additional Details: The accelerometer section provides a range of ±2g/±4g/±8g/±16g with adjustable low-pass filter bandwidths ranging from 1kHz to less than 8Hz. It also supports motion-triggered interrupt signal generation for advanced motion detection. The gyroscope offers switchable ranges from ±125°/s to ±2000°/s, with low-pass filter bandwidths adjustable between 523Hz and 12Hz. The digital interface section includes comprehensive information on

the I2C, UART, and HID over I2C protocols. The packaging takes into account environmental safety considerations, including halogen content.

Current Sensor (INA 181) - Analog Reading



Key Features: This sensor boasts a common-mode voltage range from -0.2 V to +26 V and features a high bandwidth of 350 kHz. It is capable of bidirectional current sensing and offers gain options ranging from 20 V/V to 200 V/V. It operates on a power supply range of 2.7-V to 5.5-V and is available in various package options, suiting different design requirements.
Additional Details: The specifications include comprehensive information on Absolute Maximum Ratings, ESD Ratings, and Thermal Information, providing critical details for safe and effective usage. The pin configuration section details the functions of pins for different package types, aiding in proper integration and connection in circuit designs.

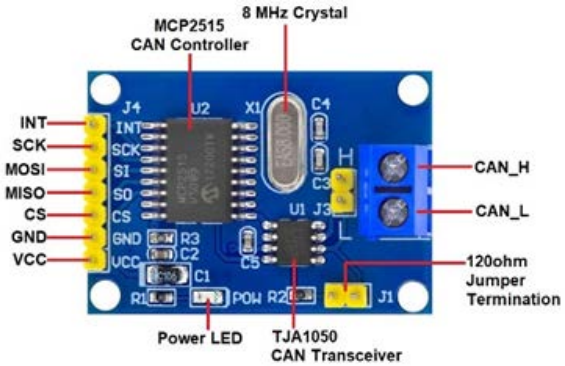
Buck Step Down Converter 48/ to 5 VDC (K78U05-500R3)

Key Features: This voltage converter supports an input voltage range up to 10:1 and demonstrates high efficiency, up to 93%. It operates within an ambient temperature range of -40°C to 85°C. The converter is also equipped with output short-circuit protection and is compatible with the K78XX series, ensuring reliability and versatility in various applications.
Additional Details: The performance section includes data on transient response deviation, recovery time, and details on short-circuit protection with continuous self-recovery features. The dimensions and layout information are provided, offering guidance on the physical integration of the converter in different design layouts.

Infrared Temperature Sensor (hiletgo GY-906 MLX90614ESF) - I2C Communication

Key Features: This compact and cost-effective sensor is easy to integrate into various systems. It comes factory calibrated for sensor temperatures ranging from -40…+125 ˚C and object temperature measurements from -70…+380 ˚C. The sensor provides high accuracy (±0.5°C) in the 0..+50°C range and has a measurement resolution of 0.02°C. It also features a customizable PWM output for continuous temperature reading and is available in both 3V and 5V versions.
Additional Details: The sensor is an Infra Red thermometer designed for non-contact temperature measurements. It integrates a low noise amplifier, a 17-bit ADC, and a DSP unit, ensuring high accuracy and resolution. The sensor is factory calibrated with digital PWM and SMBus output options. Electrical specifications include maximum ratings for supply voltage, operational and storage temperature ranges, and ESD sensitivity. The pin description section details the functions of pins for the MLX90614 family, including VSS, SCL/Vz, PWM/SDA, VDD.

CAN Communication Module (MCP2515)



Key Features: This module implements CAN V2.0B at a speed of 1 Mb/s. It features two receive buffers with prioritized message storage, six 29-bit filters, and two 29-bit masks. Additionally, it

includes three transmit buffers with prioritization and abort features, enhancing its utility in complex communication systems.
Additional Details: The module functions as a stand-alone CAN controller, capable of transmitting and receiving both standard and extended data and remote frames. It interfaces with microcontrollers via an SPI connection. The package types and pinout description provide detailed information on various package types and the functions of individual pins, aiding in its application in a wide range of electronic designs.

Actuator selection:

YJ-1500W electric bicycle motor

This motor is a gearless, brushless model with a power of 1500W and a nominal voltage of 48V, capable of reaching a maximum speed of 55 km/h. Here are some key aspects and considerations for its configuration:
Type of Actuator and Configuration:The motor is a brushless, gearless actuator. These motors typically require specific electronic controllers.For configuration, digital communication is likely to be used, possibly through SPI (Serial Peripheral Interface) or I2C (Inter-Integrated Circuit). These protocols allow for the adjustment of parameters such as speed, torque, and possibly aspects of energy efficiency.It's important to check if the motor includes an integrated controller or if an external one is required. If an external controller is needed, it must be compatible with the mentioned protocols.
Adjustable Parameters: Speed (RPM) and torque can be adjusted through the controller. This is crucial to adapt the motor's performance to different terrain and load conditions.
In some systems, the motor's response to acceleration and energy regeneration during braking can also be adjusted.
Circumstances for Modifying Parameters:
Terrain change: higher torque for uphill or greater efficiency for flat terrains.
Load variation: adjust power delivery according to the weight carried.
User preferences: for a smoother or more sporty driving experience.
Technical Aspects and Compatibility:
The motor is designed for city or mountain bikes, but not for bikes with wide tires ("fat bikes").
Compatible with V/Disc brake systems and various wheel sizes (16" to 29").

It has an IP54 waterproof rating, suitable for moderately wet conditions.

Fault Diagnostic Systems

Electrical Fault Diagnostics: Focuses on detecting anomalies in electrical systems such as short circuits, overloads, voltage fluctuations, and insulation failures. Sensors and data analysis software play a crucial role here, enabling real-time monitoring and identification of abnormal patterns.

Mechanical Fault Diagnostics: Includes detecting problems in moving parts like bearings and gears. Abnormal vibrations, increased temperature, and noise are common indicators of mechanical failures. Techniques like vibration analysis and thermography are used to diagnose these issues.

Pressure Fault Diagnostics: Applicable in hydraulic and pneumatic systems where inadequate pressure can indicate leaks or blockages. Pressure sensors are used to continuously monitor and alert about deviations from normal ranges.

Liquid System Fault Diagnostics: Involves detecting leaks, liquid contamination, or issues in pumps. Flow and liquid quality sensors help identify these issues.

Humidity and Other Environmental Factors Diagnostics: In some equipment, excessive moisture or adverse environmental conditions can cause failures. Humidity sensors and other environmental devices can help monitor and control these conditions.

Component Selection and Design

Overvoltage and Undervoltage Protection:

Voltage regulators and voltage dividers are selected to protect against extreme fluctuations.

In this case, a voltage divider was applied as it functions in a way that: Overvoltage Detection: When the input voltage exceeds a pre-set limit, the system detects this overvoltage condition. This is achieved through a detection circuit that monitors the output voltage of the voltage divider.

Undervoltage Detection: Similarly, if the input voltage falls below a minimum threshold, the system detects an undervoltage condition. This is also done through a monitoring circuit that compares the output voltage with a predefined reference level.

Actuation: In case of detecting overvoltage or undervoltage, the regulator acts to protect connected devices. This may involve disconnecting the load, activating a bypass circuit, or adjusting the resistance in the voltage divider to change the output voltage.

Voltage detection circuits are crucial for real-time monitoring and adjustment.

High and Low Current Protection:

Fuses and circuit breakers are used to protect against overcurrents.

Current sensors like the INA 181 provide accurate measurements to monitor performance and prevent failures.

Current Detection: The INA181 measures the voltage difference across a current sensing (shunt) resistor placed in the circuit's current line being monitored.

Signal Amplification: The voltage signal proportional to the current flowing through the shunt resistor is too small to be processed directly. The INA181 amplifies this signal to make it more manageable and easily readable by control systems or microcontrollers.

Current to Voltage Conversion: The INA181 converts the current flowing through the shunt resistor into a proportional output voltage. This allows for easy integration with digital systems and microcontrollers.

Overcurrent Protection: The device can be configured to trigger an alarm or interruption when the current exceeds a predefined threshold, providing effective protection against overcurrents.

Low Current Detection: Similarly, the INA181 can detect low current conditions, which is useful in applications where maintaining a certain level of current is critical.

High Precision and Low Power Consumption: The INA181 is known for its high precision in current measurement and low energy consumption, making it suitable for a wide range of applications, including portable systems and battery-powered devices.

Interface and Applications: The sensor offers a simple interface that can be easily connected to ADCs (analog-to-digital converters) of microcontrollers to monitor current in real time, useful in battery management systems, motor control, and power system protection.

Protection Components:

Current Sensor (INA 181): Key for monitoring load and preventing failures.

Charge Pump (C1240A): Maintains a constant and stable supply.

Voltage Divider: Used to handle voltage spikes and protect the system.

Current Sensor (INA 181):

Features: The INA181 is a current sense amplifier designed for cost-optimized applications. It can handle common-mode voltages from -0.2 V to +26 V, regardless of the supply voltage. This device integrates a matched gain resistor network in fixed gain options of 20 V/V, 50 V/V, 100 V/V, or 200 V/V, minimizing gain error and reducing temperature drift.

Expected Use: Suitable for bidirectionally monitoring the voltage drop across current sense resistors over a wide range of common-mode voltages.

Design: Can operate with a power supply from 2.7 V to 5.5 V, and the single-channel model INA181 has a maximum supply current of 260 µA.

Charge Pump (C1240A):

Features and Design: Unfortunately, I am unable to find detailed information on the C1240A charge pump. Typically, charge pumps are used in electronic circuits to generate voltages higher than the original power supply voltage, but without specific information on the C1240A, it is difficult to provide precise details.

Voltage Divider:

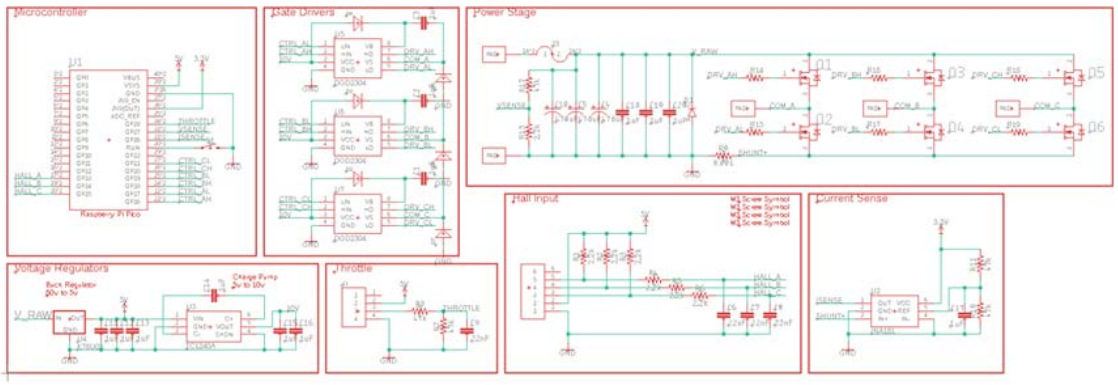
Features: A voltage divider is a simple circuit made of series resistors. The output voltage of a voltage divider is a fixed fraction of the input voltage, and this ratio is determined by two resistors.

The specific choice of resistors of **47kΩ and 2.2kΩ** is based on the desired voltage reduction ratio and the input characteristics of the microcontroller's ADC.

Expected Use: Used to reduce the voltage to a level that is manageable and safe for other circuit components.

Design: The output is calculated by scaling the input voltage by the ratio of the resistors: the lower resistor divided by the sum of the resistors.

Implementation in the Circuit All sensors and modules implemented for the safety of the same project can be implemented in the circuit as follows:



BLDC Controller PCB

Electronic components

The electronic design of a Brushless DC (BLDC) motor controller combines various electronic components and circuits to manage and control the motor's operation effectively. The primary components include a half-bridge configuration, predrivers, capacitors, sensors, and filtering mechanisms, each fulfilling a specific role within the system. This section outlines the electronic composition of the BLDC controller, detailing the purpose and functionality of each component.

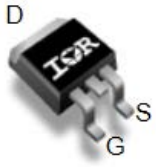
Half-Bridge Configuration

The half-bridge configuration is central to the process of electronic commutation in BLDC motors. Each half-bridge consists of two power transistors, typically MOSFETs or IGBTs, that are arranged in a series configuration across the power supply. These transistors alternate in their conduction to control the direction and magnitude of the current flowing through the motor's windings, thus creating the rotating magnetic field necessary for motor rotation.

In operation, the half-bridge functions by turning the high-side and low-side switches on and off in a complementary fashion. When the high-side MOSFET is activated, it allows current to flow from the power supply through the motor coil to the ground, which is facilitated by the low-side MOSFET being off. Conversely, activating the low-side MOSFET allows current to flow in the opposite direction when the high-side MOSFET is off. This controlled switching is synchronized with the motor's rotor position, which is detected by the Hall sensors.

Adequate thermal management is another critical aspect of the half-bridge configuration. The transistors in the half-bridge dissipate heat during operation, particularly at higher switching frequencies and load currents. The design of the motor controller must, therefore, include provisions for heat sinking and airflow to ensure that the temperature remains within safe operating limits.

For this design we chose the IRF135 because it is a high-power N-channel MOSFET known for its efficiency and robustness, making it suitable for high-frequency and high-power applications.



D^2-Pak
IRF135S203

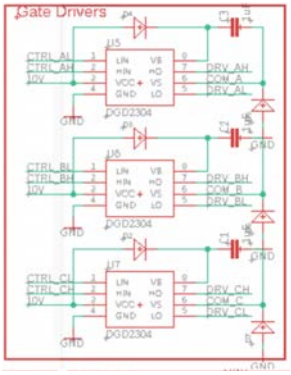
Pre-drivers

The predrivers act as intermediaries between the low-power control signals from the microcontroller and the high-power gates of the MOSFETs. They ensure that the signals are adequately amplified and shaped, providing the precise timing required for efficient motor control. Predrivers often include built-in safety features like overvoltage and overcurrent protection, contributing to the system's overall robustness.

Predrivers are designed to be easily integrated with various microcontrollers, offering flexibility in their control interfaces. Whether through standard GPIO signals or specialized communication protocols, predrivers can accommodate different control schemes to suit a wide range of applications.

For this application we decided to use the **IR2101** which is a high-voltage, high-speed power MOSFET and IGBT driver with independent high and low side referenced output channels.

8-Lead SOIC
IR2101S/IR2102S

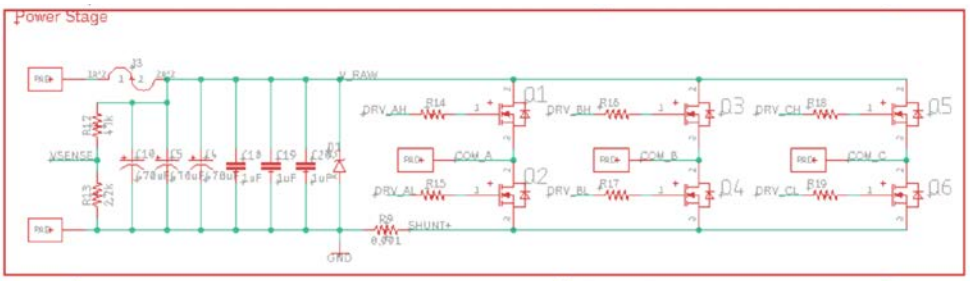


Capacitors

Electrolytic capacitors are placed between the source voltage of the motor phases and the ground to stabilize the power supply and absorb current spikes during switching events. Additionally, ceramic decoupling capacitors are strategically positioned near the power pins of the microcontroller and other sensitive components to mitigate voltage transients and noise, ensuring stable operation.

- **470uF Electrolytic capacitors**:
    - Energy Storage: They act as reservoirs of energy, providing bursts of current when the motor switches phases and requires more power.
    - Voltage Smoothing: These capacitors help smooth out the voltage ripples and spikes that occur due to the rapid switching of the MOSFETs in the half-bridge, thus protecting sensitive components from voltage fluctuations.
    - Reducing Voltage Droop: Under high current demands, these capacitors reduce the voltage droop, ensuring consistent power delivery to the motor.
- **1uF Ceramic capacitors**

- Decoupling: These capacitors are placed close to the power pins of microcontrollers and other ICs, providing a local charge reservoir to counteract transient voltage drops and spikes during rapid current changes.
- Noise Filtering: Ceramic capacitors are effective in filtering out high-frequency noise from the power supply, thereby protecting the logic circuits from electromagnetic interference.



Voltage Division for Voltage Sensing

A voltage divider circuit is utilized to step down the motor's voltage to a level that is compatible with the microcontroller's ADC inputs. This allows the controller to monitor the motor's voltage in real-time, which is essential for feedback and protective measures.

The specific choice of resistors of **47kΩ and 2.2kΩ** is based on the desired voltage reduction ratio and the input characteristics of the microcontroller's ADC.

The voltage divider works on the principle that the voltage drop across a resistor in a series circuit is proportional to its resistance. The formula for the output voltage (V_out) at the node between the two resistors is given by

$$V_{out} = V_{in} \, x \, \frac{R_{lower}}{R_{total}}$$

where:

- V_{in} is the input voltage from the motor.
- R_{lower} is the resistance of the lower resistor (2.2kΩ).
- R_{total} is the total resistance (47kΩ + 2.2kΩ).

Current Sensor

The current sensor, implemented using an operational amplifier (op-amp), measures the motor's current flow. This information is crucial for performance monitoring and the implementation of protective strategies, such as preventing overcurrent conditions that could lead to hardware damage or system failure.

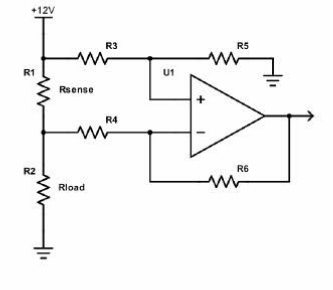
For current sense was implemented the **INA181**, which is a bidirectional, zero-drift current-sense amplifier that offers several key features beneficial for BLDC motor control:

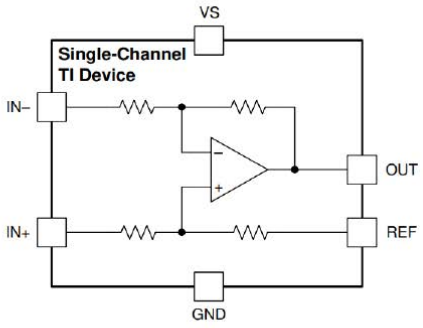
- High Accuracy: The INA181 provides precise current measurement, essential for accurate motor control and feedback.
- Low Offset Voltage: Its zero-drift architecture ensures minimal offset voltage, which is crucial for low-current sensing accuracy.
- Wide Common-Mode Voltage Range: This allows the INA181 to operate over a wide range of input voltages, suitable for varied BLDC motor applications.
- Bidirectional Sensing: The ability to sense current in both directions is beneficial for BLDC motors, where the current direction changes with the motor's operation.

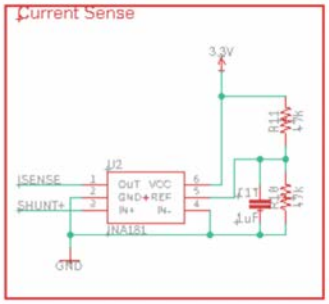
The INA181 is typically placed in series with a shunt resistor through which the motor current flows. The voltage across the shunt resistor is proportional to the current, which the INA181 amplifies and outputs as a measurable signal.

The current data provided by the INA181 is vital for implementing overcurrent protection mechanisms. By detecting excessive current, the motor controller can take preventive actions, such as reducing the power or shutting down the motor to prevent damage.

Accurate current measurement enables the BLDC controller to operate the motor more efficiently. It can optimize the power usage based on the current demand, thereby improving the overall energy efficiency of the system.







Filtering and Signal Conditioning

Low-pass passive filters are integrated into the system for both the Hall sensors and the throttle input. These filters remove high-frequency noise, ensuring that the signals fed to the microcontroller are clean and stable. Pull-up resistors on the Hall sensors provide the necessary logic levels and improve signal integrity.

- Halls Filtering with **2.2kΩ Resistors and 22nF Capacitors**
    - Purpose: These filters are designed to remove high-frequency noise from sensor signals, ensuring that the signals fed into the microcontroller are stable and free from transient spikes that could lead to erroneous readings.
    - Operation: The 2.2kΩ resistor and the 22nF capacitor work together to create a filter with a specific cut-off frequency. This cut-off frequency is calculated based on the resistor and capacitor values and is chosen to allow the desired signal frequencies (typically lower frequencies) while attenuating higher frequencies that constitute noise.
- **2.2kΩ Pull-Up Resistors** on Hall Sensor Outputs
    - Function: Pull-up resistors ensure that the Hall sensor outputs have a well-defined high state, particularly important for Hall sensors that have open-collector or open-drain outputs.
    - Stabilization: These resistors help in stabilizing the signal and prevent floating inputs, which could lead to unpredictable behavior in the microcontroller's readings.
- **Two 47kΩ Resistors**: These resistors are configured in parallel with each other. This arrangement forms part of the voltage divider network that scales down the throttle input voltage to a level suitable for the microcontroller's ADC.
    - Voltage Division: The parallel configuration of the 47kΩ resistors effectively divides the voltage from the throttle sensor, ensuring that the ADC receives a signal within its operating range.
- **22nF Ceramic Capacitor**: This capacitor is connected in parallel with one of the 47kΩ resistors. It serves as a low-pass filter, smoothing out high-frequency noise and transient fluctuations in the throttle signal.
    - Noise Filtering: The 22nF ceramic capacitor acts as a filter, attenuating high-frequency components that might be present in the throttle signal. This filtering is crucial to avoid erratic motor behavior caused by electrical noise.
    - Signal Stability: The combination of resistors and capacitor stabilizes the throttle signal, providing a smooth and consistent input to the controller. This stability is particularly important for achieving a linear and predictable response to throttle adjustments.

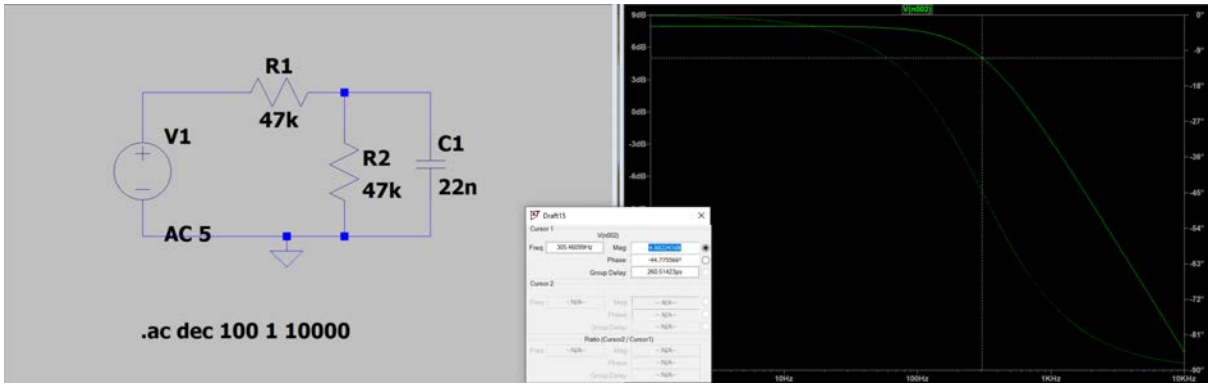
The cut-off frequency (f_c) for a passive low-pass RC (Resistor-Capacitor) filter is given by the formula:

$$f_c = \frac{1}{2\pi RC}$$

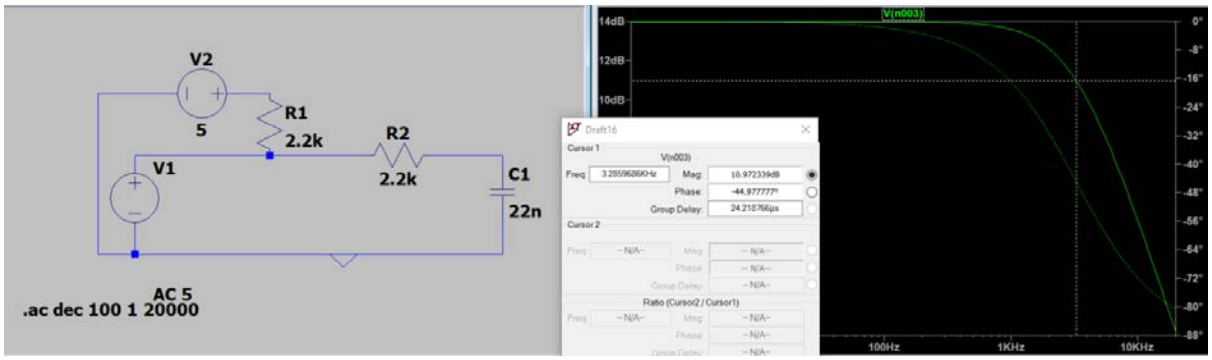
where:

- R is the resistance in ohms (Ω).
- C is the capacitance in farads (F).
- 2π (approximately 6.283) is a constant derived from the properties of a circle.

Throttle low pass filter simulation



Halls low pass filter simulation

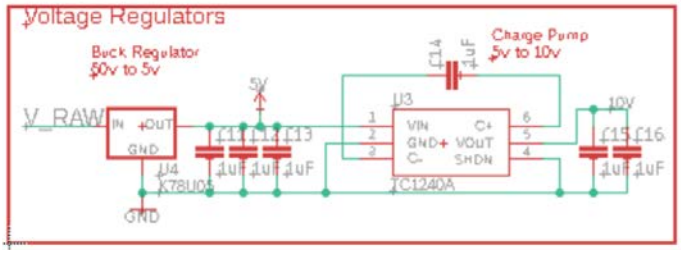


Power Regulators

The design includes power management, incorporating a buck regulator to step down voltage from 50V to 5V and a charge pump to boost voltage from 5V to 15V. These components, along with the use of 1uF ceramic decoupling capacitors, ensure stable and efficient power distribution within the system.

- **Buck Regulator from 50V to 5V**
  - The buck regulator efficiently reduces the 50V supply down to 5V, which is a more suitable level for powering low-voltage electronics like microcontrollers and sensors.

    - By using a buck (step-down) converter, the design achieves high efficiency, minimizing energy loss during voltage conversion.
    - The 5V output from the buck regulator is used to power critical components of the system, including the microcontroller, throttle sensor, and Hall sensors.
- **Charge Pump from 5V to 15V**
    - The charge pump circuit boosts the 5V output from the buck regulator up to 15V, providing a higher voltage supply where needed.
    - Charge pumps are known for their simplicity and compactness, making them a suitable choice for space-constrained applications.
    - The 15V output is specifically used to power the predrivers (e.g., Infineon IR2101). This higher voltage is necessary for effectively driving the gates of the high-power MOSFETs in the half-bridge.
- **Decoupling with 1uF Ceramic Capacitors**
    - Decoupling capacitors are essential for stabilizing the power supply to individual components. They store and release energy quickly, thereby smoothing out transient voltage spikes and dips.
    - The 1uF ceramic capacitors, placed close to the power inputs of the microcontroller, predrivers, and other sensitive components, help maintain a steady voltage level, crucial for reliable operation.
    - By mitigating voltage fluctuations and providing a local energy reservoir, decoupling capacitors enhance the overall stability and performance of the system.
    - They are particularly effective in filtering out high-frequency noise, improving the electromagnetic compatibility (EMC) of the device.



Additional Components

Standard components for typical applications include voltage regulators to provide stable power to the microcontroller and other logic circuits. Protection diodes are also employed to safeguard against reverse voltage and overvoltage conditions, which are common during motor operation due to back EMF and other inductive effects.

Microcontroller

The microcontroller executes the control algorithms essential for BLDC motor operation. These algorithms involve reading sensor inputs (like Hall sensors and throttle position), processing these inputs, and generating appropriate control signals for the motor phases.

- Raspberry pi pico
- Stm32 Bluepill
- Stm32 Nucleo-F401RE

Challenges in PCB Design

In the development of a Brushless DC (BLDC) motor controller, challenges such as logistical constraints and software tool limitations often arise. Our team's experience in designing a Printed Circuit Board (PCB) for the STM32 Bluepill microcontroller encapsulates these challenges.

Initially, we faced a significant logistical challenge: the PCB designed for the STM32 Bluepill, ordered from China, would not arrive in time due to shipping delays. This setback threatened to derail our project timeline, necessitating an immediate and innovative solution.

In response, we pivoted to using an adapter compatible with our existing instrumentation. This decision was a practical compromise, allowing us to proceed with the project without the custom-designed PCB. It exemplified our team's adaptability in the face of unforeseen challenges.

Our initial plan was to modify the PCB design using Altium Designer. However, we discovered that the original design was created in Eagle. This realization prompted a strategic shift in our approach, leading us to migrate the project to Eagle. The transition was not just a change in software but a learning curve in adapting to a different design environment.

Upon immersing ourselves in Eagle, we found it to be more intuitive and user-friendly, particularly suited to the scope of our project. Eagle provided the necessary tools and features for efficient PCB design, challenging our initial preference for Altium Designer.

Through this process, we honed our skills in various aspects of PCB design. We learned to fine-tune properties like vias, drills, and pads, and used international standard calculators to optimize component dimensions. Additionally, we explored Eagle's bitmap integration feature, allowing us to personalize our PCB design.

A significant part of our learning journey was understanding the importance of comprehensive documentation. We ensured that our PCB design was well-documented, facilitating ease of adoption and modification by others. Moreover, we adhered to IPC standards and best practices, reinforcing the importance of these guidelines in PCB design, irrespective of the software used.

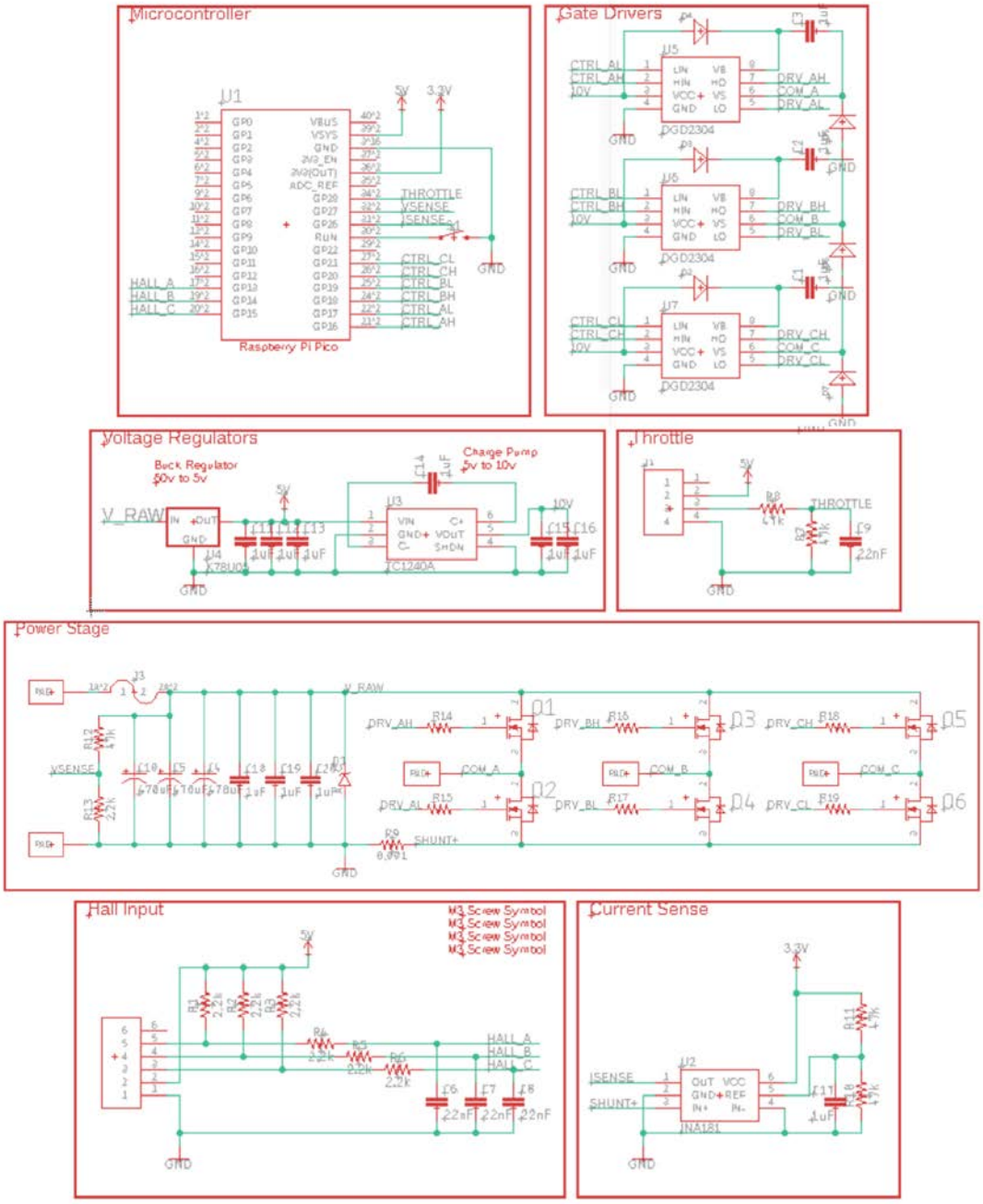
Our experience led us to an important realization: while development platforms significantly influence the design experience, they should not restrict the application of good engineering practices. Flexibility and adaptability in using different software tools are crucial skills in electronics design.

Our initial design centered around the Raspberry Pi Pico we identified the need for a microcontroller with specific features that the STM32 Bluepill offered, but we had STM32 Nucleo-F401RE available to the moment, since the Bluepill design did not arrived in time we chose to use the Nucleo-F401RE board

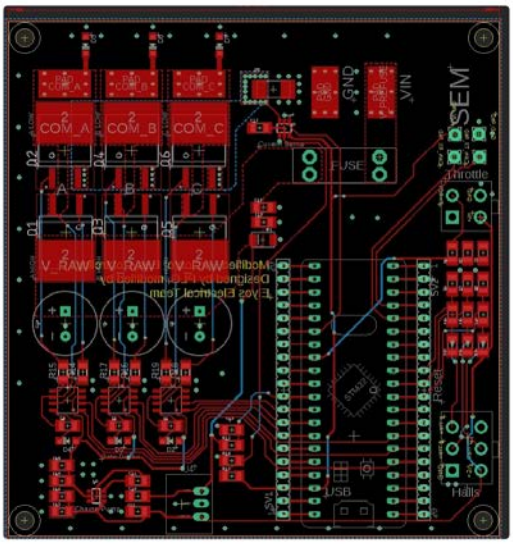
One of the primary challenges was ensuring compatibility between the STM32 and the existing design. This task involved careful consideration of pin configurations, power requirements, and communication protocols.

Pcbs Schematics and Layouts

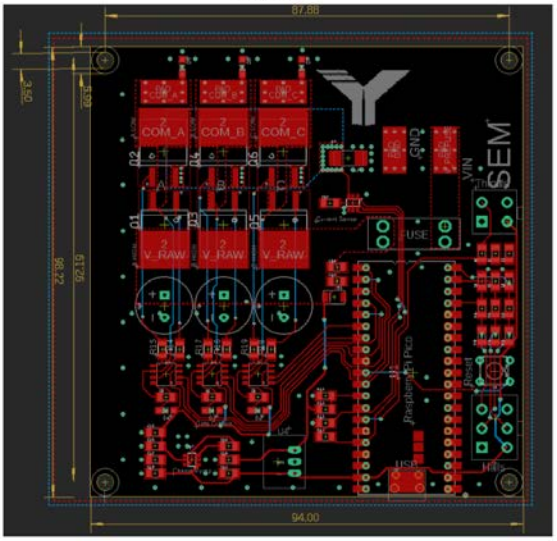
Raspberry pi pico schematic


Microcontroller
U1
Raspberry Pi Pico
THROTTLE
VSENSE
ISENSE
CTRL_CL
CTRL_CH
CTRL_BL
CTRL_BH
CTRL_AL
CTRL_AH
HALL_A
HALL_B
HALL_C
Gate Drivers
DGD2304
DRV_AH
DRV_AL
DRV_BH
DRV_BL
DRV_CH
DRV_CL
COM_A
COM_B
COM_C
Voltage Regulators
Buck Regulator
50v to 5v
Charge Pump
5v to 10v
V_RAW
TC1240A
Throttle
Power Stage
SHUNT+
Hall Input
M3_Screw Symbol
Current Sense
INA181

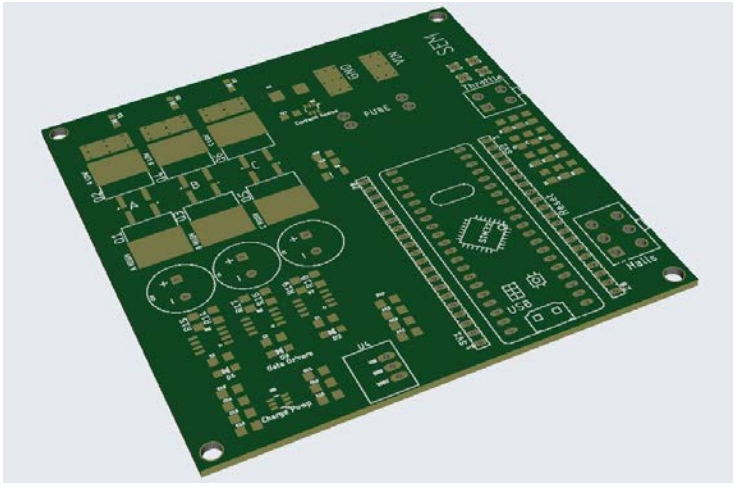
Stm32 bluepill layout



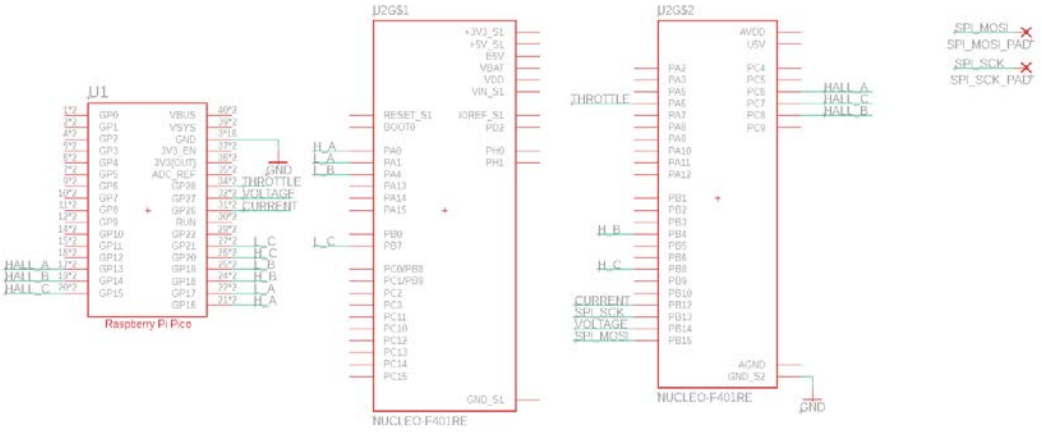
Raspberry pi pico layout



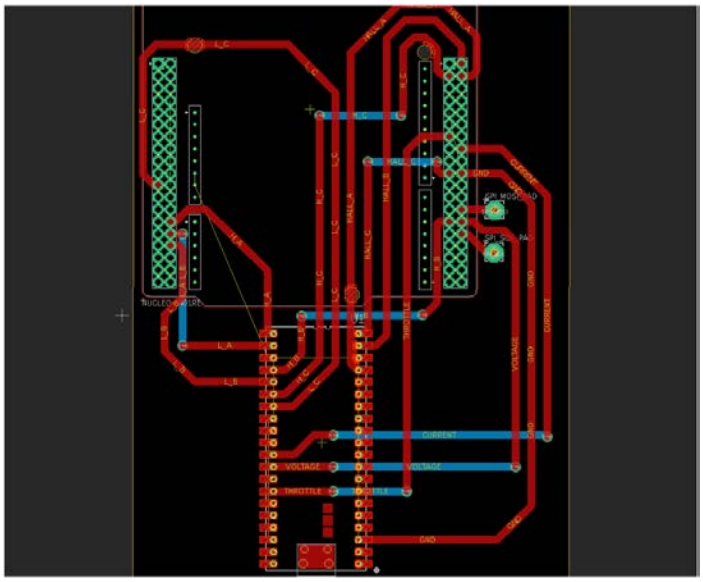
PCB render



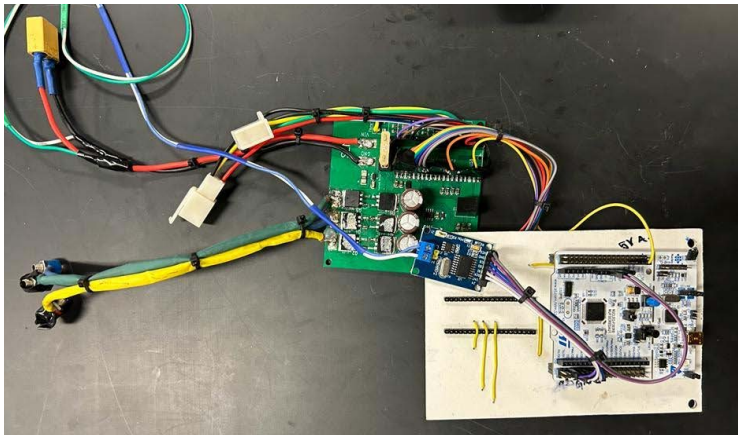
NUCLEO F401RE schematic



NUCLEO F401RE shield layout



Raspberry board with NUCLEO F401RE shield

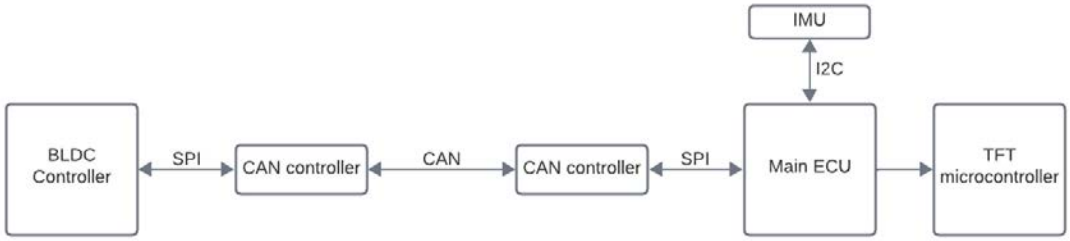


Embedded Software Development

Protocols used

In the realm of embedded systems, the choice and implementation of communication protocols play a pivotal role in defining the system's efficiency, reliability, and scalability. This application utilizes I2C for interfacing with the Inertial Measurement Unit (IMU), SPI for the CAN transceivers, and UART for communication with the TFT controller.

Protocol Application Diagram



I2C Communication

Purpose: The I2C protocol is used for communication between the main ECU and the Inertial Measurement Unit (IMU).

Functionality: This protocol facilitates the transfer of motion-related data, such as acceleration and orientation, which are critical for the ECU's decision-making processes.

Integration: The integration of I2C allows for efficient and real-time data acquisition from the IMU, enabling the ECU to make timely decisions based on the vehicle's physical state.

SPI Communication

Role of SPI-CAN: The SPI protocol is employed to interface with the CAN transceivers. This setup is critical for communicating key parameters like the angular velocity, current, and voltage of the BLDC controller to the main ECU.

Data Transmission: Through SPI, the ECU receives detailed and real-time information about the motor's performance, which is essential for monitoring and control purposes.

System Integration: The use of SPI for CAN communication ensures high-speed data transfer and reliability, which are paramount in vehicular systems where rapid data processing is required.

UART Communication

Communication with TFT: The UART protocol is utilized for communication between the main ECU and the TFT microcontroller.

Display and Control: This communication is vital for displaying system statuses, alerts, and other critical information on the TFT display such as the current velocity of the bike.

Efficiency and Reliability: UART provides a reliable and straightforward communication channel, ensuring that the user interface remains responsive and up-to-date with the system's state.

System Communication

Inter-Module Communication: The combination of these protocols facilitates a cohesive and integrated system where each component can effectively communicate and operate in unison.

Safety and Control: The real-time data from the IMU and BLDC controller, combined with the user interface on the TFT display, allows for comprehensive monitoring and control of the electric bike system. This integration is crucial for ensuring the safety and efficiency of the vehicle.

Future Scalability: The chosen communication protocols offer scalability, allowing for future enhancements or integration of additional components without significant redesign.

Utilization of DMA in Communication Protocols

Traditional polling methods for communication protocols, while straightforward, can significantly hamper system efficiency and responsiveness. Polling continuously checks the status of a peripheral device, leading to increased CPU workload and slower response times, especially problematic in high-frequency data environments like those in power electronics controllers. This inefficiency is particularly pronounced in systems requiring real-time data processing, such as our Telemetry ECU. To address these challenges, Direct Memory Access (DMA) is employed in conjunction with communication protocols like I2C, SPI, and UART. DMA enhances data transfer processes, enabling the system to manage high-volume data streams more effectively and with reduced CPU overhead

Transition from Embedded CAN to SPI-to-CAN Module

In the development of our project, a significant shift occurred in the approach to CAN (Controller Area Network) communication. Initially, the project employed the embedded CAN controller and OpenCAN, a decision rooted in leveraging the microcontroller's native capabilities. However, due to several challenges, the team transitioned to using a SPI-to-CAN module.

Initial Approach: Embedded CAN Controller and OpenCAN

- **Complex Integration**: Utilizing the microcontroller's built-in CAN module presented integration complexities. The need for a compatible transceiver added to the hardware requirements.
- **Protocol Challenges**: The CAN protocol's complexity, especially in configuring and managing the network, posed significant challenges. It required a deep understanding of the protocol's intricacies.
- **Analytical Tools Requirement**: The effective implementation of CAN demanded specialized tools like a protocol analyzer, essential for debugging and network analysis. The project, however, faced constraints in accessing such tools.
- **Project Timeline Constraints**: The project was operating under a tight deadline. The time required to overcome the learning curve and implementation hurdles associated with CAN was becoming a critical factor.

Transition to SPI-to-CAN Module

- **Simplifying Communication**: The decision to switch to a SPI-to-CAN module was driven by the need to simplify the communication setup. SPI (Serial Peripheral Interface) offered a more straightforward implementation approach compared to CAN.
- **Overcoming Transceiver Limitations**: The SPI-to-CAN module eliminated the need for a separate transceiver compatible with the microcontroller's CAN module, simplifying the hardware setup.
- **Ease of Use**: SPI's relative simplicity over CAN made it a more feasible option within the project's timeframe. It required less specialized knowledge, allowing the team to focus on other critical aspects of the project.
- **Rapid Integration and Testing**: The SPI-to-CAN module enabled faster integration and testing, crucial for adhering to the project's tight schedule. This approach facilitated quicker iterations and troubleshooting.

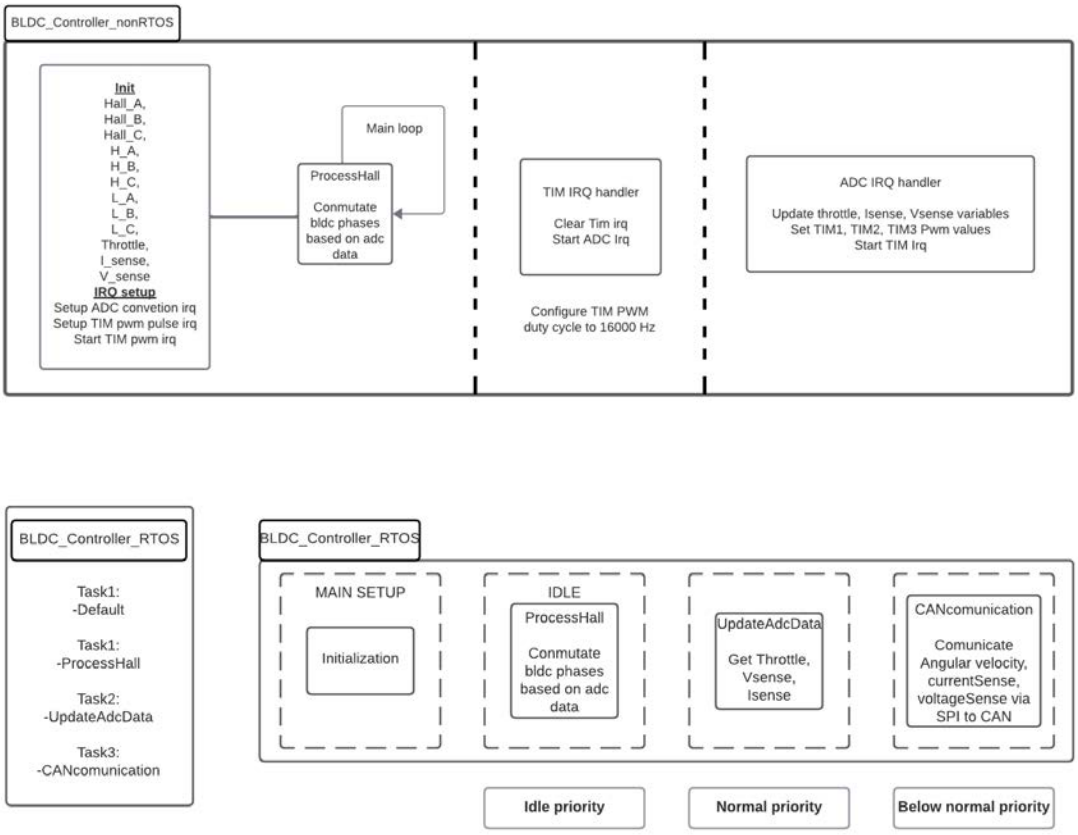
This transition from using the embedded CAN controller and OpenCAN to implementing a SPI-to-CAN module was a strategic decision influenced by hardware limitations, protocol complexity, tool availability, and project deadlines.

Logic behind BLDC Controller

The commutation process involves switching the current between the motor's phases in a sequence that aligns with the position of the rotor to create continuous rotational motion.The BLDC controller manages the motor's operation by controlling the power delivered to the motor phases. The core of the BLDC controller's logic lies in its PWM control. Operating at a frequency of 17 kHz, the PWM signals ensure that the motor operates efficiently and with minimal noise which is a critical factor in these applications. The software logic intricately controls the high and low sides of the motor phases, ensuring that power delivery to the motor is both efficient and safe. This precise control is essential for achieving the desired motor performance, be it speed and torque.

The controller constantly reads inputs from the Hall sensors to determine the rotor's position, which is vital for the accurate commutation of the motor. The throttle position is read using an

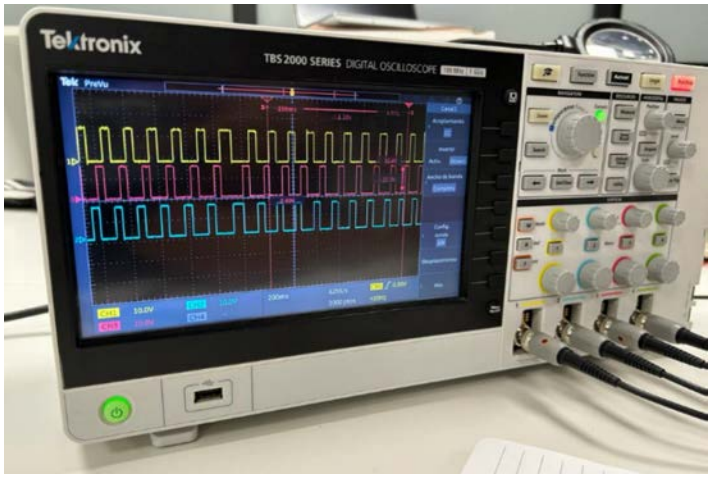
Analog-to-Digital Converter (ADC), allowing the controller to respond to the user's input promptly. Furthermore, current and voltage sensors provide crucial feedback on the motor's operating conditions, enabling the controller to make adjustments for efficiency and safety, such as implementing overcurrent protection.



Half-Bridge Communication Sequence

This sequence relies on the correct timing and order of activation to create a rotating magnetic field that interacts with the permanent magnets in the motor's rotor.

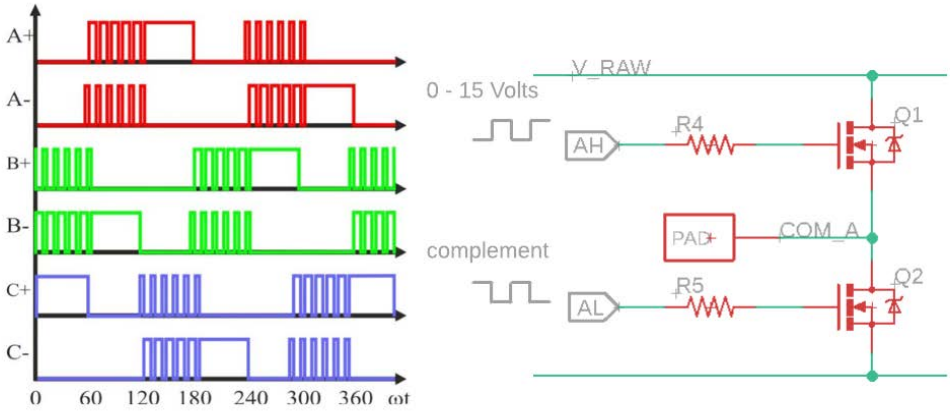
The commutation sequence begins with feedback from the Hall sensors, which provide information about the rotor's position. Each sensor outputs a high or low signal that corresponds to the magnetic pole it faces. The combination of these signals gives a binary representation of the rotor's position relative to the stator.



Based on the Hall sensor feedback, the controller determines which coils in the stator to energize to push or pull the rotor to the next position. The energization sequence is designed to maintain the rotor's motion in a particular direction, typically clockwise or counterclockwise, depending on the desired outcome.

In the controller, the Pulse Width Modulation (PWM) of the signals is crucial for speed control. The duty cycle of the PWM signals is adjusted according to the throttle input, which determines the motor's speed.

The commutation sequence is a continuous loop that repeats as long as the motor is in operation. The precision with which the controller executes this sequence directly affects the motor's performance

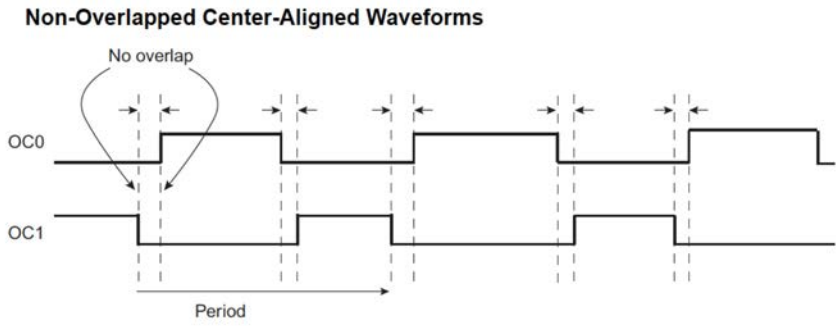


Half-Bridge Protection

In the context of setting the phases to the MOSFETs. Complementary PWM is implemented to prevent the perilous possibility of both MOSFETs conducting simultaneously, an event known as "shoot-through." This condition would create a direct short circuit, potentially damaging the system. Complementary PWM introduces a critical 'dead time' between the deactivation of one MOSFET and the activation of the other, ensuring that they never conduct at the same time. This approach is critical for the safe operation of the motor and the preservation of the controller's integrity.

Center-aligned PWM is another sophisticated feature of BLDC motor control, chosen for its symmetrical signal pattern. This strategy minimizes electromagnetic interference—a significant advantage in automotive applications where EMI can affect other critical systems. Moreover, it improves efficiency by reducing switching losses and provides a more uniform thermal distribution across the switching components.

The synthesis of the half-bridge configuration, complementary PWM, center-aligned PWM, and the predriver's role culminates in a BLDC motor controller that is efficient, reliable, and suited to the demands of automotive applications.



BLDC Controller RTOS version

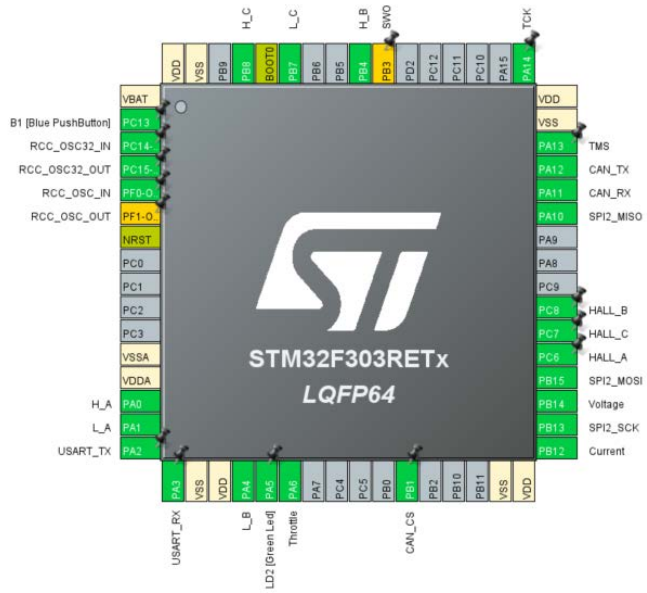
https://github.com/Ineso1/Conversion-de-bici-electrica-/blob/main/BLDC_Controller_RTOS_V1/Core/Src/freertos.c

The code incorporates FreeRTOS for real-time task management, spanning from hardware initialization to advanced communication through CAN. The controller is designed to provide precise control of the BLDC motor, along with efficient monitoring of critical parameters such as voltage, current, and rotor position.

BLDC Controller Initial Setup and Peripheral Configurations

The main.c file outlines a systematic approach to initializing the BLDC controller's microcontroller unit (MCU) and configuring various peripherals.

- **MCU and System Initialization**
  - **HAL (Hardware Abstraction Layer) Initialization**: The HAL_Init() function resets all peripherals, initializes the Flash interface, and configures the Systick timer.
  - **System Clock Configuration**: In SystemClock_Config(), the system clock is set up to use the High-Speed Internal (HSI) oscillator with a Phase-Locked Loop (PLL) multiplier of 9. This ensures that the MCU operates at an optimal frequency for motor control tasks.
- **Peripheral Configuration**
  - **Peripheral Initialization**: The code initializes a range of peripherals including GPIO, DMA, USART2, ADC2, ADC4, TIM2, TIM3, TIM4, CAN, TIM15, SPI2, and TIM6. These peripherals are important for tasks such as sensor data acquisition, communication, and motor control.
- **FreeRTOS Integration**
  - **RTOS Object Initialization**: The function MX_FREERTOS_Init() initializes objects like tasks and queues in the FreeRTOS environment, setting up the framework for multitasking.
  - **RTOS Scheduler Start**: The osKernelStart() function launches the FreeRTOS scheduler, which manages task execution based on priority and timing.
- **BLDC-Specific Configurations**
  - CAN Interface: The CANSPI_Initialize() function configures the Controller Area Network (CAN) interface, essential for communication with other system components.



Code Functions

BLDC Initialization and Configuration (`init_bldc()`)

- **Parameters and Configurations**
  - **Current and Voltage Scaling**
    - Defined to convert ADC readings into actual current and voltage values.
  - **PWM Settings and Throttle Limits**
    - Include PWM frequency (`F_PWM`), maximum duty cycle (`DUTY_CYCLE_MAX`), and throttle limits (`THROTTLE_LOW`, `THROTTLE_HIGH`).
- **Hardware Initialization**
  - **PWM**
    - Timer configuration and PWM start-up for motor control.
  - **ADC**
    - Activation of ADC for sensor and control signal readings.

C/C++

```
// Proportions
CURRENT_SCALING = 3.3 / 0.001 / 20 / 4096 * 1000;
VOLTAGE_SCALING = 3.3 / 4096 * (47 + 2.2) / 2.2 * 1000;

 // Parameter
HALL_OVERSAMPLE = 8;
HALL_IDENTIFY_DUTY_CYCLE = 40;
F_PWM = 16000;
DUTY_CYCLE_MAX = 255;

// Current cutoff
FULL_SCALE_CURRENT_MA = 30000;

// Throttle limits
THROTTLE_LOW = 210;
THROTTLE_HIGH = 1740;

adc_channel_count = sizeof(adc_dma_result)/sizeof(adc_dma_result[0]);
adc_conv_complete_flag = 0;


__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, 255);
__HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, 0);
__HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, 255);
__HAL_TIM_SET_COMPARE(&htim4, TIM_CHANNEL_3, 0);
__HAL_TIM_SET_COMPARE(&htim4, TIM_CHANNEL_2, 255);
```

```
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_3);
HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_2);
HAL_ADC_Start(&hadc2);
```

Hall Sensor Reading (`get_halls()`)

- **Rotor Position Detection**
  - Hall sensors are read to determine the current position of the BLDC motor rotor.
- **Hall Signal Processing**
  - Multiple sampling is performed, and a threshold is applied to determine logical states.
  - Debounce for Hall sensors reading

```C/C++
unsigned int get_halls(void){
	unsigned int hallCounts[] = {0, 0, 0};

	// Read all the Hall pins repeatedly and tally the results
	for (unsigned int i = 0; i < HALL_OVERSAMPLE; i++) {
		hallCounts[0] += HAL_GPIO_ReadPin(HALL_A_GPIO_Port, HALL_A_Pin);
		hallCounts[1] += HAL_GPIO_ReadPin(HALL_B_GPIO_Port, HALL_B_Pin);
		hallCounts[2] += HAL_GPIO_ReadPin(HALL_C_GPIO_Port, HALL_C_Pin);
	}
	unsigned int hall = 0;
	// If votes >= threshold, set the corresponding bit to 1
	if (hallCounts[0] > HALL_OVERSAMPLE / 2)
		hall |= (1 << 0);
	if (hallCounts[1] > HALL_OVERSAMPLE / 2)
		hall |= (1 << 1);
	if (hallCounts[2] > HALL_OVERSAMPLE / 2)
		hall |= (1 << 2);
	char message[50];
	snprintf(message, sizeof(message), "Hall Value: %u\r\n", hall);
```

```
	HAL_UART_Transmit(&huart2, (uint8_t *)message, strlen(message),
HAL_MAX_DELAY);
	*/
	return hall;
}
```

Motor Phase Control (`write_pd_table()`, `writePhases()`)

- **Phase Handling**
  - **Phase Selection**
    - Based on the combination of Hall sensor states.
  - **PWM Control**
    - Adjustment of duty cycles to control motor speed and direction.

```
C/C++
void write_pd_table(unsigned int halls, unsigned int duty){

	if(duty >= 257){
		duty = 0;
	}
	if(duty < 27){
		throttle_pwm = 0;
		duty = 0;
		halls = 8;
	}
	unsigned int complement = 255 - duty - 6;

	switch(halls){
	    case 1: // Case 001
		writePhases(duty, 0, 0, complement, 0, 255);
		break;
	    case 2: // Case 010
		writePhases(0, 0, duty, 0, 255, complement);
		break;
	    case 3: // Case 011
		writePhases(duty, 0, 0, complement, 255, 0);
		break;
	    case 4: // Case 100
		writePhases(0, duty, 0, 255, complement, 0);
		break;
```

```
            case 5: // Case 101
                writePhases(0, duty, 0, 0, complement, 255);
                break;
            case 6: // Case 110
                writePhases(0, 0, duty, 255, 0, complement);
                break;
            default: // Case 000 or error
                writePhases(0, 0, 0, 255, 255, 255);
        }
}

void writePhases(uint16_t ah, uint16_t bh, uint16_t ch, uint16_t al, uint16_t
bl, uint16_t cl){
        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, ah);
        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, 255 - al);
        __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, bh);
        __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, 255 - bl);
        __HAL_TIM_SET_COMPARE(&htim4, TIM_CHANNEL_3, ch);
        __HAL_TIM_SET_COMPARE(&htim4, TIM_CHANNEL_2, 255 - cl);
}
```

Throttle, Voltage and Current Reading (**`read_throttle()`, `read_voltage()`, `read_current()`**)

- **Signal Processing**
  - **`read_throttle()`:**
    - Reading and processing of the throttle signal.
  - **`read_voltage()`:**
    - Conversion of ADC reading to actual voltage.
  - **`read_current()`:**
    - Similar to `read_voltage()`, but applied to current.

```
C/C++
void read_throttle(void){
        unsigned int throttle_adc = HAL_ADC_GetValue(&hadc2);

        throttle_adc = (throttle_adc - THROTTLE_LOW) * 255;
        if(throttle_adc < 0){
                throttle_adc *= -1;
        }
```

```
        throttle_adc = throttle_adc / (THROTTLE_HIGH - THROTTLE_LOW);
        throttle_pwm = throttle_adc;

          if (throttle_adc >= 255) // Bound the output between 0 and 255
               throttle_pwm = 255;

          if (throttle_adc <= 0)
               throttle_pwm = 0;
}

void read_voltage(void){
        int voltage_adc = HAL_ADC_GetValue(&hadc4);
        voltage_mv = voltage_adc * CURRENT_SCALING;
}

void read_current(void){
        int current_adc = HAL_ADC_GetValue(&hadc4);
        current_ma = current_adc * VOLTAGE_SCALING;
}
```

FreeRTOS Thread Management

- **Tasks and Functions**
  - **`StartDefaultHandle`:**
    - Manages the Idle function of conmutation
  - **`StartPwmHandle`:**
    - Manages the PWM signal.
  - **`StartAdcHandle`:**
    - Conducts reading of throttle values and sensors.
  - **`StartCanHandler`:**
    - Setup and transmission of messages for interaction with other systems on a CAN network.
    - Responsible for CAN communication.

```
C/C++
void StartDefaultTask(void const * argument)
{
uint8_t nextState[] = {5, 3, 1, 6, 4, 2};
for(;;)
 {
```

```
        for(unsigned int j = 0; j<2; j++){
                unsigned int hall = get_halls();
                write_pd_table(hall, HALL_IDENTIFY_DUTY_CYCLE);
                osDelay(1);
                write_pd_table(nextState[hall - 1], HALL_IDENTIFY_DUTY_CYCLE);
        }
        osDelay(100);
 }
}

void StartPwmHandle(void const * argument)
{
 uint8_t nextState[] = {5, 3, 1, 6, 4, 2};
 for(;;)
 {
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        for(unsigned int j = 0; j < 160; j++)
        {
                unsigned int hall = get_halls();
                write_pd_table(hall, throttle_pwm);
                osDelay(1);
                write_pd_table(nextState[hall - 1], HALL_IDENTIFY_DUTY_CYCLE);
        }
 }
}

void StartAdcHandle(void const * argument)
{
 for(;;)
 {
        read_throttle();
        read_current();
        unsigned int halls = get_halls();
        write_pd_table(halls, throttle_pwm);
        osDelay(7);
 }
}

void StartCanHandler(void const * argument)
{
  for(;;)
 {
        txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
        txMessage.frame.id = 0x0A;
```

```
        txMessage.frame.dlc = 8;
        txMessage.frame.data0 = throttle_pwm;
        txMessage.frame.data1 = current_ma;
        txMessage.frame.data2 = 0;
        txMessage.frame.data3 = 0;
        txMessage.frame.data4 = 0;
        txMessage.frame.data5 = 0;
        txMessage.frame.data6 = 0;
        txMessage.frame.data7 = 0;
        CANSPI_Transmit(&txMessage);
        osDelay(50);
 }
}
```

BLDC Controller nonRTOS

https://github.com/Ineso1/Conversion-de-bici-electrica-/blob/main/BLDC_Controller_interruption_V1/Core/Src/bldc_controller.cpp

In the evolution of the Brushless DC (BLDC) motor controller's firmware from a non-Real-Time Operating System (non-RTOS) to an RTOS-based implementation, a significant change was made in the way the controller handles Pulse Width Modulation (PWM) and Analog-to-Digital Converter (ADC) interrupts.

IRQ Handlers

PWM and ADC Interrupts (`pwm_irq()`, `adc_irq()`):

- Manages interrupts from PWM and ADC peripherals, ensuring real-time responsiveness for critical motor control tasks.

```
C/C++
void pwm_irq(struct Bldc* bldc) {
        HAL_ADC_Start_IT(bldc->THROTTLE_ADC);
        HAL_ADC_Start(bldc->I_SENSE_ADC);
        __HAL_TIM_CLEAR_FLAG(bldc->PWM_A, TIM_FLAG_UPDATE);
}

void adc_irq(struct Bldc* bldc) {
  HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        read_throttle(bldc);
```

```
        read_voltage(bldc);
        read_current(bldc);

        unsigned int halls = get_halls(bldc);

        write_pd_table(bldc, halls, bldc->throttle_pwm);
        __HAL_ADC_RESET_HANDLE_STATE(bldc->THROTTLE_ADC);
        HAL_TIM_PWM_Stop(bldc->PWM_A, TIM_CHANNEL_2);
        HAL_TIM_PWM_Start_IT(bldc->PWM_A, TIM_CHANNEL_2);
}
```

BLDC Controller RTOS vs nonRTOS

The significant transition from a non-Real-Time Operating System (RTOS) implementation to a FreeRTOS-based system resulted in the development of a Brushless DC (BLDC) motor controller using an STM32 microcontroller to achieve communication. The initial implementation faced challenges with communication efficiency and motor control stability, leading to the decision to implement FreeRTOS. This transition aimed to enhance task management, communication handling, and motor control precision.

Key Change in Interrupt Handling

- **Non-RTOS Version**
    - In the non-RTOS version of the BLDC controller, the PWM and ADC interrupts were closely linked. Each time the PWM interrupt was triggered, it directly set off the ADC interrupt.
    - This approach ensured that every PWM cycle (which controls the motor speed and torque) was immediately followed by an ADC update, which refreshed global variables reliant on ADC readings, such as throttle position, motor current, and voltage.
    - However, this method tied the ADC updates to the PWM cycle, limiting flexibility and potentially affecting the system's responsiveness to rapidly changing motor conditions.
- **RTOS Version**
    - In contrast, the RTOS-based implementation decouples the PWM and ADC functionalities by handling them within separate tasks.
    - The ADC values are updated periodically in their dedicated task, with the frequency of updates determined by the task's delay (osDelay). This allows for a more controlled and potentially more frequent updating of ADC values, independent of the PWM cycle.

    - The PWM IRQ still plays a crucial role but is now primarily focused on motor control without the additional responsibility of triggering ADC updates. This separation can lead to more efficient task management and better utilization of the MCU's capabilities.
- **Implications of the Change**
    - Improved Task Management: By leveraging the multitasking capabilities of RTOS, the firmware can handle motor control and sensor readings more efficiently. Tasks can be prioritized and scheduled based on their criticality and necessity.
    - Enhanced Flexibility: Decoupling ADC updates from the PWM cycle provides greater flexibility in how sensor data is processed and used. This can be advantageous in dynamic environments where motor conditions change rapidly.
    - Increased Responsiveness: Independent ADC tasking allows for more frequent or strategically timed sensor readings, which can improve the controller's responsiveness to changes in motor load, throttle position, etc.

Overview of the initial implementation

Challenges and Limitations
Inefficient Communication: Utilizing polling for communication led to potential delays and less responsive data handling.
Motor Control Instability: Absence of a real-time task scheduler resulted in less precise motor control, contributing to vibration issues.

Rationale for Transition to FreeRTOS

Key Considerations

Enhanced Task Management: FreeRTOS facilitates organized and prioritized task handling, essential for complex embedded systems.
Responsive Communication: Transitioning to interrupt-driven communication improves the efficiency and speed of data processing.
Improved Motor Control: Real-time task scheduling in FreeRTOS enables more stable and smooth motor operation.

FreeRTOS to BLDC Controller Evaluation

Strategy and Approach

- Task Prioritization: Critical tasks, such as motor control, are assigned higher priorities to ensure timely execution.
- Interrupt Handling: Careful design of interrupt handling to maintain task timing and system responsiveness.
- Optimized Communication Protocol: Moving from polling to interrupt-based communication to enhance data handling efficiency.
- Resource Management: Efficient use of microcontroller resources to balance task execution and system performance.

Testing and Results: Methodology and Observations

The transition to FreeRTOS successfully addressed the initial challenges, particularly in improving motor control stability and communication efficiency. The system demonstrated enhanced performance, with more precise control and responsive data handling. This change lays a robust foundation for further development and optimization of the controller, paving the way for its application in more advanced and demanding environments.

The testing began with a close examination of task execution. The execution order, timing, and priority handling of tasks were scrutinized to ensure that critical tasks, especially those related to motor control, were receiving appropriate priority and executing within the expected time frames. This was pivotal in confirming that the real-time capabilities of FreeRTOS were being effectively utilized.

Alongside task execution, the system's response to interrupts was meticulously tested. This involved assessing the interrupt latency to ensure that it was within acceptable limits and verifying that the interrupt service routines (ISRs) were not adversely affecting the performance of high-priority tasks. This step was crucial to maintain the real-time responsiveness of the system.

Communication protocols also underwent thorough testing. The shift from polling to interrupt-driven communication was a significant change, and its implementation was rigorously tested. This included stress-testing the system with high volumes of data to ensure that communication remained efficient and reliable even under heavy loads.

A central aspect of this transition was the improvement of motor control, specifically targeting the vibration issues identified in the non-RTOS version. To this end, extensive vibration analysis was conducted. The motor was operated under various speeds and load conditions, and the vibration intensities were measured and compared before and after the implementation of FreeRTOS. This comparison was instrumental in quantifying the improvements in motor stability.

Control precision was another critical area of focus. The motor’s responsiveness to control inputs and its operational stability under varying conditions were evaluated. This assessment helped in determining the enhancements in the precision of motor control brought about by the FreeRTOS implementation.

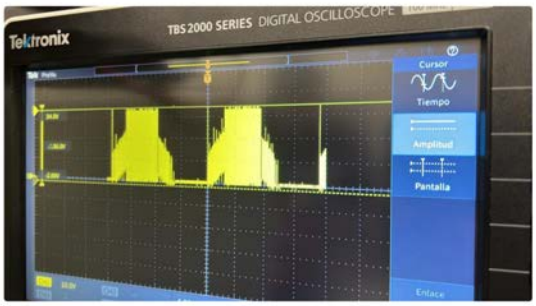
Through these comprehensive testing procedures, the project team systematically evaluated and validated the enhancements brought about by the FreeRTOS implementation. The results indicated a significant improvement in motor control stability, a reduction in vibrations, and enhanced communication efficiency. These outcomes collectively confirmed the success of transitioning to a real-time operating system, laying a solid foundation for the system's future development and optimization.

Summary of Achievements

The transition to FreeRTOS in the BLDC controller project necessitated a robust and comprehensive testing strategy to ensure the reliability and effectiveness of the system. The methodology focused on validating various aspects of the FreeRTOS implementation, ranging from task execution to communication protocols.

Phase voltage and back EMF appliying comunication protocols

nonRTOS version



RTOS version

Control

BLDC Control Implementation

https://github.com/Ineso1/Conversion-de-bici-electrica-/tree/main/BLDC_Controller_RTOS_Characterization

Motor Characterization

The primary goal of motor characterization was to map the relationship between throttle input and motor output, particularly focusing on the motor's angular velocity. This analysis is essential for optimizing the control algorithms and ensuring that the motor performs efficiently and reliably across a range of operating conditions.

Using the BLDC controller's software, we applied varying pulses to the throttle. These variations were designed to simulate a wide range of real-world operating scenarios, from gentle accelerations to rapid speed changes.

The controller's firmware was programmed to collect data on key parameters:

- **Throttle Position**: Measured using the ADC, this input was crucial in determining how the motor's speed control responded to user commands.
- **Angular Velocity**: Calculated based on the frequency of Hall sensor signal changes, providing a direct measure of the motor's response to throttle adjustments.

Velocity Calculation

The velocity of the motor is calculated based on the change in Hall sensor states over time. The system records the time at each Hall sensor state change. Using these timestamps, it calculates

the motor's angular velocity by determining the time elapsed between consecutive Hall state changes.

```
C/C++
unsigned int get_halls(void){
    // Read Hall sensor states and tally the results
    // ...
    return hall;
}

unsigned int hall = get_halls();
uint32_t current_time = HAL_GetTick();
uint32_t time_elapsed = current_time - last_pwm_interrupt_time;

// Calculate angular velocity
angular_velocity = (60000 / (time_elapsed)) * (hall - previous_hall_state);
```

Data Transmission

- **UART, FreeRTOS and DMA Configuration**

For efficient data transmission, the system uses UART in conjunction with DMA and FreeRTOS. DMA allows the transfer of data from memory to the UART transmit buffer without burdening the CPU, enabling continuous data flow while freeing the CPU to handle other tasks allowing the FreeRTOS scheduler to manage other tasks more effectively.

- **´info´ Function**

The info function is designed to send motor performance data, including angular velocity, to the user. This function formats the data into a string and transmits it via UART using DMA, ensuring real-time data communication.

```
C/C++
void DataTransmissionTask(void const * argument){
    while(1){
        info();
        osDelay(20);
    }
}

void info(void){
  // Toggle a GPIO pin for indication
  HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
```

```
    // Prepare data message
    char message[50];
    snprintf(message, sizeof(message), "%d,", throttle_pwm);

    // Transmit throttle data
    HAL_UART_Transmit_DMA(&huart2, (uint8_t*)message, strlen(message));

    // Prepare and transmit angular velocity data
    snprintf(message, sizeof(message), "%d\n", angular_velocity);
    HAL_UART_Transmit_DMA(&huart2, (uint8_t*)message, strlen(message));
}
```

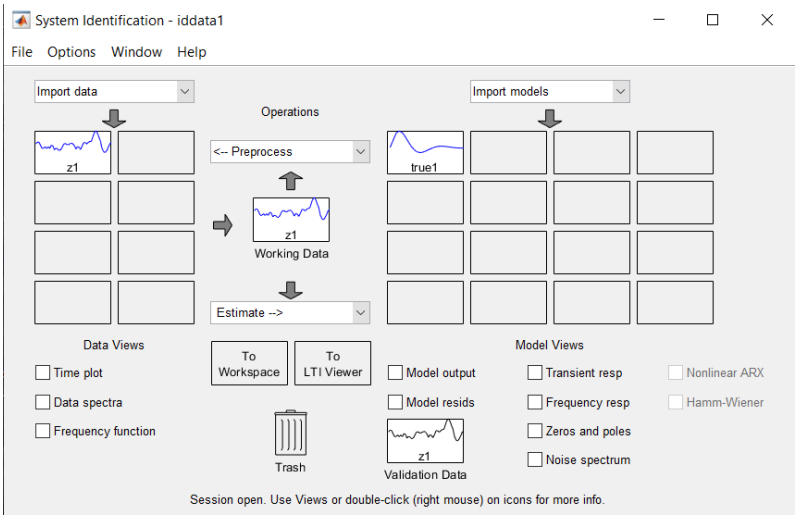
The info function transmitted the collected data in real-time via UART, allowing for immediate observation and analysis. This approach enabled us to monitor the motor's performance as we varied the throttle input.

We observed a clear correlation between throttle input and motor speed. Variations in throttle position resulted in corresponding changes in the motor's angular velocity, confirming the effectiveness of our control algorithm.

The characterization process also revealed the motor's sensitivity to rapid throttle changes, providing insights into the motor's dynamic behavior under sudden load variations.

Identification and Model Derivation

Leveraging Simulink’s System Identification Toolbox, we embarked on estimating a model that best represented the motor's dynamics. The toolbox provided a suite of candidate model structures, including Auto-Regressive with eXogenous input (ARX), state-space models, and transfer functions. By applying these structures to our processed data, we identified a model that accurately captured the motor's characteristics with respect to the input commands.



The transfer function model derived from system identification encapsulates the BLDC motor's dynamic response to various inputs. This model serves as the foundation for designing our PID controller by providing a mathematical framework that represents the motor's physical behavior

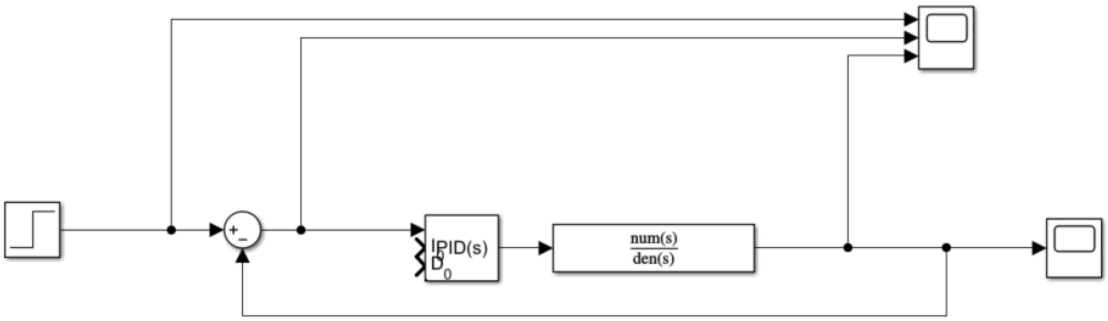
Angular velocity PID Controller

The PID controller, a widely used feedback control tool, was tailored to our motor's model within Simulink. The PID block within Simulink offers an intuitive interface for integrating the transfer function and implementing the PID algorithm. Simulink's PID tuning tool was employed to automate the process of adjusting the PID parameters, the proportional gain (Kp), integral gain (Ki), and derivative gain (Kd). This tool adjusts the parameters to achieve desired response characteristics such as minimal overshoot, quick settling time, and stable steady-state error.

Initially, we approached the PID control design from a continuous-time perspective. This method allowed us to utilize classical control theory and Simulink's PID tuning tools to ascertain optimal PID coefficients (Kp, Ki, Kd) based on the motor's transfer function.

Upon determining the optimal continuous-time PID parameters, the next step involved converting these into a discrete-time format suitable for implementation on a digital microcontroller. This conversion was achieved using the Zero-Order Hold (ZOH) method, which assumes that the control signals hold their value between sampling intervals. ZOH is a common and effective approach for discretizing control laws while preserving system stability and performance characteristics.

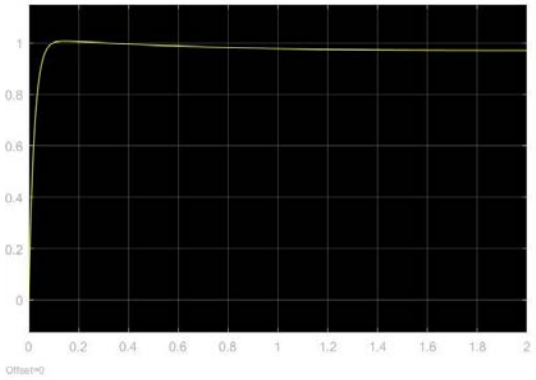
The resulting discrete-time PID coefficients were then used to formulate difference equations. These equations serve as the algorithmic backbone for the PID control in the digital domain, effectively replacing the differential equations used in continuous-time control with recursive relations that can be computed at each sampling interval.



Proportional (P): 1.10402607342959

Integral (I): 0.296077397975466

Derivative (D): 0.914690937280454



Derivation of Difference Equation for PID

The implementation of Proportional-Integral-Derivative (PID) control necessitates the translation of continuous-time control theory into discrete-time algorithms. This process involves deriving the difference equation for the PID controller.

The PID control law in the time domain is given by:

$$u(t) = K_p e(t) + K_I \int e(t)\,dt + K_d \frac{de(t)}{dt}$$

The next step involves applying the Laplace transform to convert the PID controller to the frequency domain. Additionally, considering the discrete nature of digital systems, a zero-order holder (ZOH) is introduced. The transfer functions for the ZOH and the PID controller are as follows:

$$\left[\frac{1-e^{T_s s}}{1}\left(\frac{K_D s^2+K_P s+K_I}{s}\right)\right]$$

To accommodate the discrete implementation, we apply the z-transform:

$$\mathcal{Z}\left[\frac{1-e^{T_s s}}{1}\left(\frac{K_D s^2+K_P s+K_I}{s}\right)\right]$$

The final step involves rearranging the expression to separate the control variable CV and the error signal E on different sides of the equation:

$$\frac{CV(z)}{E(z)} = K_P + K_I T_s \frac{1}{z-1} + \frac{K_D}{T_s}\frac{z-1}{z}$$

$$CV(z) - CV(z)z^{-1} = \left(K_P + \frac{K_D}{T_s}\right)E(z) + \left(-K_P + K_I T_s - 2\frac{K_D}{T_s}\right)E(z)z^{-1} + \frac{K_D}{T_s}E(z)z^{-2}$$

Using known z-transform formulas and inverse z-transform, we derive the difference equation form of the PID controller:

$$cv(n) = cv(n-1) + \left(K_p + \frac{K_d}{T_s}\right)e(n) + \left(-K_p + K_i T_s - 2\frac{K_d}{T_s}\right)e(n-1) + \frac{K_d}{T_s}e(n-2)$$

Control Block Integration

https://github.com/Ineso1/Conversion-de-bici-electrica-/blob/main/BLDC_Controller_RTOS_V1/Core/Src/freertos.c

The discrete-time PID difference equations were coded into the firmware of the microcontroller, forming the basis for the real-time control loop. This implementation accounted for the microcontroller's sampling rate and computational constraints, ensuring that the control loop operates with the necessary precision and responsiveness.

```C/C++
error = targetVelocity - (currentVelocity);
 errores[1] = errores[0];
 errores[2] = errores[1];
 errores[0] = error;
 prevPwm = motorVelocity;

 float kp = 1.10;
```

```
float ki = 0.29;
float kd = 0.91;
float a = (kp + kd/0.02) * errores[0];
float b = (-kp + (ki*0.02) - (2*kd/0.02)) * errores[1];
float c = (kd/0.02) * errores[2];

int pwmSpeed = prevPwm + a + b + c;
motorSpeed = pwmSpeed;
```

Bike Telemetry ECU

https://github.com/Ineso1/Conversion-de-bici-electrica-/blob/main/Telemetry_ECU_RTOS_V1/Core/Src/freertos.c

The Telemetry ECU is designed to centralize the data acquisition and communication tasks in the electric bike system. It interfaces with an Inertial Measurement Unit (IMU) for motion tracking, communicates with a TFT microcontroller via UART for display and user interaction, and handles CAN communication through SPI for broader system integration.

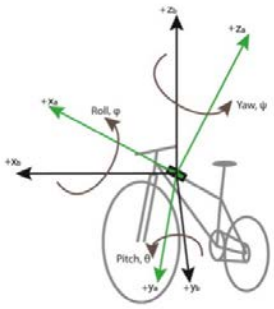
The decision to implement an RTOS was driven by the need for efficient multitasking, real-time responsiveness, and enhanced system stability, particularly crucial in dynamic environments.

As the Telemetry ECU is positioned at the bike's center of mass it ensures that the IMU can effectively measure the stability information, providing a true representation of the bike's dynamics.

The IMU data is processed using a DSP algorithm from the ARM Cortex Microcontroller Software Interface Standard (CMSIS) DSP library. This advanced filtering technique is crucial for removing noise from the IMU signals, ensuring that the data is both accurate and reliable.

The IMU plays a multifunctional role in tracking various aspects of the bike's motion, including:

- Velocity Measurement: By comparing the IMU velocity readings with the motor's angular velocity, the system can validate the motor's performance and the bike's speed.
- Orientation Metrics: The IMU measures pitch, roll, and yaw, which are vital for understanding the bike's orientation in three-dimensional space.
- Force Analysis: The IMU aids in measuring G-forces, which are indicative of various maneuvers such as acceleration, braking, and navigating turns, thus providing insights into the centrifugal forces at play.



Safety Features

The comprehensive data from the IMU can be used to implement advanced safety features, such as automatic braking or stability assistance, to protect the rider during critical maneuvers.

Incorporating an Inertial Measurement Unit (IMU) into our telemetry system required a reliable method for data acquisition. To facilitate this, we employed a specialized library that allowed us to interface with the IMU using the Inter-Integrated Circuit (I2C) protocol.

G Force Calculation

G-force can be calculated as the ratio of the measured acceleration to the standard gravity (9.81 m/s^2).

Roll and Pitch Calculation

yaw, roll, and pitch typically require a 3D orientation sensor (like a gyroscope) and possibly a magnetometer. But they can be estimated from the accelerometer data under static conditions or slow movements using trigonometric relationships. Yaw cannot be calculated from the accelerometer alone.

Velocity Estimation

Velocity can be estimated by integrating the acceleration over time. However, this simple integration may not be accurate due to sensor noise and bias.

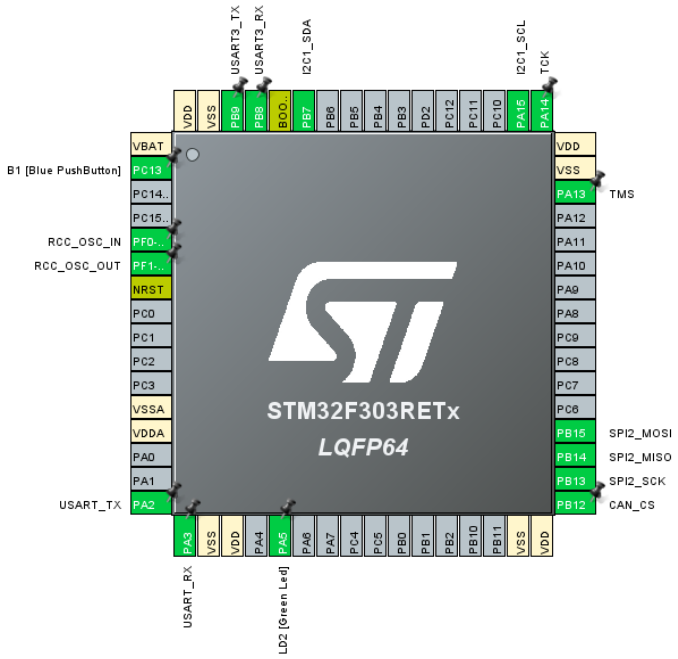
ECU Initial Setup and Peripheral Configurations

The main.c file begins with the standard inclusion of necessary headers and initialization of various peripherals. The code is well-structured, following a clear sequence from the basic initialization of the microcontroller to the configuration of communication interfaces and the FreeRTOS environment.

- **Microcontroller and Peripheral Initialization**

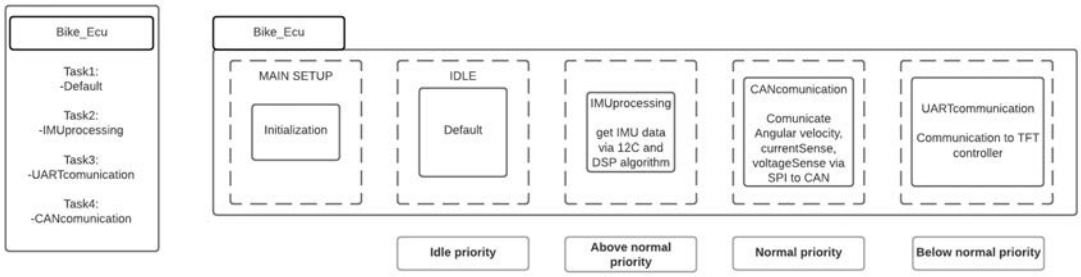
- **HAL Initialization**: The code begins with HAL_Init(), which resets all peripherals, initializes the Flash interface and the Systick.
- **System Clock Configuration**: SystemClock_Config() configures the system clock. The clock is set to use the High-Speed Internal (HSI) oscillator with a PLL (Phase-Locked Loop) multiplier of 9, ensuring the microcontroller operates at the desired frequency for optimal performance.
- **Peripheral Initialization**: Peripherals like GPIO, DMA, USART2, SPI2, I2C1, TIM17, and USART3 are initialized. This setup is important for the various communication and control tasks the ECU will perform.

- **FreeRTOS Environment Setup**
  - **RTOS Object Initialization**: MX_FREERTOS_Init() is called to initialize FreeRTOS objects, such as tasks, queues, and semaphores. This step is important for setting up the multitasking environment.
  - **Starting the RTOS Scheduler**: osKernelStart() is invoked to start the RTOS scheduler, handing over control to FreeRTOS to manage task execution.
- **ECU-Specific Configurations**
  - **CAN Interface Initialization**: CANSPI_Initialize() sets up the CAN interface, for network-based communication within the bike's electronic system.
  - **IMU Sensor Initialization**: BNO055_Init_I2C() initializes the BNO055 IMU sensor over the I2C bus.



Code Functions

The purpose of these functions is to handle the tasks of fetching accelerometer data, computing G-forces, estimating roll and pitch, integrating acceleration for velocity estimation, and transmitting the calculated data.



IMU Data getter (`IMU_acceleration()`)

- This function will be responsible for fetching the raw acceleration data from the IMU.
- Fetches the raw data.

Accelerometer processing (`IMU_dsp()`)

- This function will apply a CMSIS IIR filter to the raw acceleration data to remove noise and smoothen the signal.
- Processes the data using digital signal processing techniques.

Data conversion (`acceleration_data_conversion()`)

  - This function will convert the processed acceleration data into meaningful metrics like roll, pitch, G-forces, and velocity.
  - Converts the processed data into roll, pitch, G-forces, and velocity, then prepares a message string.

```
C/C++
/* Global definition of variables fo IMU */

uint8_t       imu_readings[IMU_NUMBER_OF_BYTES];
int16_t       accel_data[3];
float         acc_x, acc_y, acc_z;
int           acc_x_int, acc_y_int, acc_z_int;
const float32_t firCoeffs[NUM_TAPS] = {
   -0.0162988415008265,
   -0.0709271864972703,
   0.0225851291371347,
   0.0371278894358235,
   -0.0134217682863348,
   -0.0589846533854996,
   0.0145099553645701,
   0.103203330310147,
   -0.0148779327102912,
```

```
    -0.317339878494921,
    0.515006915991497,
    -0.317339878494921,
    -0.0148779327102912,
    0.103203330310147,
    0.0145099553645701,
    -0.0589846533854996,
    -0.0134217682863348,
    0.0371278894358235,
    0.0225851291371347,
    -0.0709271864972703,
    -0.0162988415008265
};

float32_t firState[NUM_TAPS + BLOCK_SIZE - 1];  // Filter state buffer
arm_fir_instance_f32 FIR_Filter;
float32_t inputBuffer[BLOCK_SIZE];
float32_t outputBuffer[BLOCK_SIZE];

//On the MX_FREERTOS_Init() add the init of the filter instance
arm_fir_init_f32(&FIR_Filter, NUM_TAPS, (float32_t*)firCoeffs, firState,
BLOCK_SIZE);

/* IMU Functions */
void IMU_acceleration(float *acc_x, float *acc_y, float *acc_z) {
  uint8_t imu_readings[IMU_NUMBER_OF_BYTES];
  GetAccelData(&hi2c1, imu_readings);

  int16_t accel_data[3];
  accel_data[0] = ((int16_t)(imu_readings[1] << 8) | imu_readings[0]);
  accel_data[1] = ((int16_t)(imu_readings[3] << 8) | imu_readings[2]);
  accel_data[2] = ((int16_t)(imu_readings[5] << 8) | imu_readings[4]);

  *acc_x = ((float)accel_data[0]) / 100.0f;
  *acc_y = ((float)accel_data[1]) / 100.0f;
  *acc_z = ((float)accel_data[2]) / 100.0f;
}

void IMU_dsp(float *acc_x, float *acc_y, float *acc_z) {
    arm_fir_f32(&FIR_Filter, inputBuffer, outputBuffer, BLOCK_SIZE);
}

void acceleration_data_conversion(float acc_x, float acc_y, float acc_z, char
*message) {
  // Calculate G-forces, roll, pitch, and velocity
```

```
  float g_force_x = acc_x / 9.81f;
  float g_force_y = acc_y / 9.81f;
  float g_force_z = acc_z / 9.81f;

  float roll = atan2(acc_y, acc_z) * 180.0f / PI;
  float pitch = atan2(-acc_x, sqrt(acc_y * acc_y + acc_z * acc_z)) * 180.0f / PI;

  static float velocity_x = 0.0f, velocity_y = 0.0f, velocity_z = 0.0f;
  velocity_x += acc_x * SAMPLE_TIME;
  velocity_y += acc_y * SAMPLE_TIME;
  velocity_z += acc_z * SAMPLE_TIME;

  sprintf(message, "GX: %.2f GY: %.2f GZ: %.2f Roll: %.2f Pitch: %.2f VX: %.2f
VY: %.2f VZ: %.2f\r\n",
      g_force_x, g_force_y, g_force_z,
      roll, pitch,
      velocity_x, velocity_y, velocity_z);
}
```

Code Tasks

Default Task (`StartDefaultTask`)

- A placeholder task, potentially for future expansions or low-priority background operations.
- Currently, it executes an infinite loop with minimal operation (osDelay(1)), indicating a low overhead presence in the system.

```
C/C++
void StartDefaultTask(void const * argument)
{
 for(;;)
 {
       int v = 0;
       char message[50];
       snprintf(message, sizeof(message), "Default: %u\r\n", v);
       HAL_UART_Transmit(&huart2, (uint8_t *)message, strlen(message),
HAL_MAX_DELAY);
       osDelay(1);
 }
}
```

CAN Reception Task (`StartRecepcionCAN`)

- Handles the reception of CAN messages, crucial for system-wide communication in the bike's electronic system.
- Processes receive CAN messages and transmit responses, effectively making the bike ECU an active node in the CAN network.
- Demonstrates the use of UART for debugging or monitoring CAN traffic.

```C/C++
void StartRecepcionCAN(void const * argument)
{
 char message[50];
 for(;;)
 {
       if(CANSPI_Receive(&rxMessage))
       {
              txMessage.frame.idType = rxMessage.frame.idType;
              txMessage.frame.id = rxMessage.frame.id;
              txMessage.frame.dlc = rxMessage.frame.dlc;
              txMessage.frame.data0++;
              txMessage.frame.data1 = rxMessage.frame.data1;
              txMessage.frame.data2 = rxMessage.frame.data2;
              txMessage.frame.data3 = rxMessage.frame.data3;
              txMessage.frame.data4 = rxMessage.frame.data4;
              txMessage.frame.data5 = rxMessage.frame.data5;
              txMessage.frame.data6 = rxMessage.frame.data6;
              txMessage.frame.data7 = rxMessage.frame.data7;
              CANSPI_Transmit(&txMessage);
              snprintf(message, sizeof(message),
                     "CAN ID: %lu, DLC: %u, Data: %u %u %u %u %u %u %u %u\r\n",
              rxMessage.frame.id, rxMessage.frame.dlc,
              rxMessage.frame.data0, rxMessage.frame.data1,
rxMessage.frame.data2,
              rxMessage.frame.data3, rxMessage.frame.data4,
rxMessage.frame.data5,
              rxMessage.frame.data6, rxMessage.frame.data7);
              HAL_UART_Transmit_DMA(&huart2, (uint8_t *)message, strlen(message),
HAL_MAX_DELAY);
       }
       osDelay(100);
 }
}
```

IMU Processing Task (`StartIMUprocess`)

- Acquires and processes data from the Inertial Measurement Unit (IMU) via I2C, providing vital information about the bike's motion and orientation.
- Converts raw accelerometer data into human-readable formats and transmits it using UART, which can be crucial for real-time monitoring or data logging purposes.

```
C/C++
void StartIMUprocess(void const * argument) {
       // Keeps track of the current position in the buffer
       static size_t bufferIndex = 0;
 for(;;)
 {
       IMU_acceleration();
       // Store the current sample in the buffer
       inputBuffer[bufferIndex] = acc_z;
       // Move to the next position in the buffer
       bufferIndex++;
       if (bufferIndex >= BLOCK_SIZE) {
              bufferIndex = 0; // Reset buffer index
              IMU_dsp();
              acceleration_data_conversion(acc_x, acc_y,
outputBuffer[BLOCK_SIZE], message); //Sends the data
       }
       osDelay(2); //Keep i mind the number of samplings for the filter
  }
}
```

UART Communication Task (`StartUARTcom`)

- Manages UART communication, potentially for user interface or external communication purposes.
- This task seems to be set up for testing or as a template for future development, as indicated by the static transmission of a test value.

```
C/C++
void StartUARTcom(void const * argument)
{
 for(;;)
 {
       char message[50];
       snprintf(message, sizeof(message), "%u", rxMessage.frame.data0);
```

```
        HAL_UART_Transmit_DMA(&huart3, (uint8_t *)message, strlen(message),
HAL_MAX_DELAY);
        osDelay(200);
 }
}
```

Real-Time Operating System (RTOS) Advantages

The implementation of FreeRTOS in the Bike ECU offers several advantages:

- Efficient Multitasking: FreeRTOS allows the ECU to handle multiple operations concurrently, such as CAN communication and IMU data processing, without interference.
- Priority-Based Task Management: Each task can be assigned a priority, ensuring that more critical tasks (like CAN message processing) are allocated more CPU time compared to less critical tasks.
- Modularity and Scalability: The structure of the code facilitates easy expansion and modification, allowing new functionalities to be added as separate tasks without disrupting existing operations.

TFT Controller

https://github.com/Ineso1/Conversion-de-bici-electrica-/tree/main/TFT_Controller

A key feature of the TFT controller implementation is the ability to read data from the UART buffer and display the velocity on the TFT screen. This functionality enhances user interaction by providing real-time velocity information

TFT Controller Initial Setup and Configuration

The initialization process in the main.c file demonstrates a structured approach to preparing the microcontroller and peripherals necessary for the TFT controller's operation.

MCU and System Initialization

- **HAL Initialization**: Utilizing HAL_Init() to reset peripherals, initialize the Flash interface, and set up the Systick timer, laying a foundation for stable operations.
- **System Clock Configuration**: In SystemClock_Config(), the system clock employs the High-Speed Internal (HSI) oscillator with a Phase-Locked Loop (PLL) multiplier of 9, providing an optimal operational frequency.

Peripheral Configuration

- **Peripheral Setup**: Initializations for GPIO, USART2, and TIM2 are performed. These peripherals are vital for the TFT controller’s interface and timing control.

TFT Initialization:

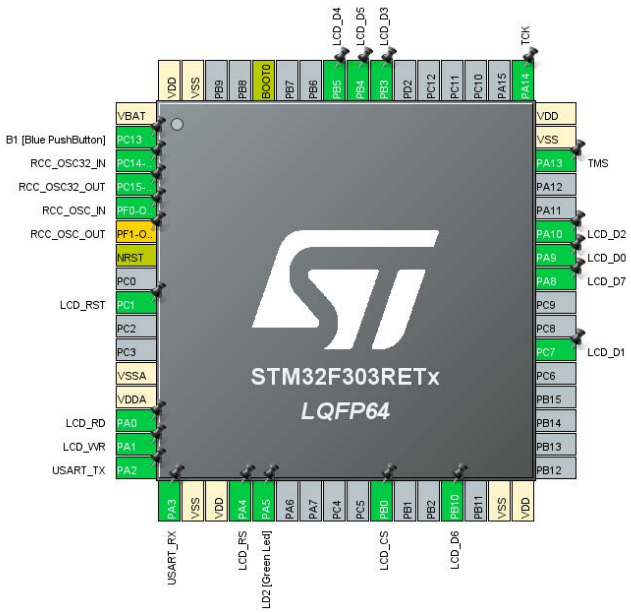
- The readID() function reads the TFT display ID, which is essential for proper initialization.
- tft_init(ID) then initializes the TFT display with the identified parameters, ensuring compatibility and optimal display settings.

Display Configuration and Testing:

- setRotation(0) and fillScreen(BLACK) configure the display orientation and background.
- Various test functions like testFillScreen(), testLines(), testFastLines(), etc., are employed to validate the display's functionality.

Custom Display Operations:

- Custom messages and designs are displayed using functions like printnewtstr(), showcasing the capability to render text and graphics on the TFT screen.



Code Functions

Display Velocity onTFT (`DisplayVelocityOnTFT`)

- takes the processed velocity data and displays it on the TFT screen. This function bridges the gap between raw data processing and user interface.

Process UART Data (`ProcessUARTData()`)

- is responsible for interpreting and converting the raw data received from the UART buffer into a meaningful format. In this case, it processes velocity data, which is essential for real-time monitoring and control.

```C/C++
int main(void) {
       MX_USART2_UART_Init();
       uint8_t uartBuffer[MAX_UART_BUFFER_SIZE];
       uint16_t velocity = 0;
       tft_init(readID());
       setRotation(0);
       fillScreen(BLACK);

  while (1) {
    // Check if UART data is received
    if (HAL_UART_Receive(&huart2, uartBuffer, MAX_UART_BUFFER_SIZE, HAL_MAX_DELAY)
== HAL_OK) {
      ProcessUARTData();
      DisplayVelocityOnTFT(velocity);
    }
  }
}

void ProcessUARTData(void) {
    velocity = atoi((char *)uartBuffer); // Convert string to integer
}

void DisplayVelocityOnTFT(uint16_t velocity) {
  char displayStr[20];
  sprintf(displayStr, "Velocity: %d", velocity);

  // Clear previous data and display new velocity
  fillScreen(BLACK);
  setTextColor(WHITE, BLACK);
  setTextSize(2);
  drawString(10, 10, displayStr, &Font16, WHITE, BLACK);
}
```

Fault Diagnosis

A) Rationale for Implementing a Digital Filter

- Noise Reduction: In an environment with electrical noise or fluctuating signals, a digital filter is essential to extract accurate data from sensors.
- Signal Smoothing: For applications like motor control or telemetry, it's crucial to have smooth, reliable data inputs. A digital filter helps in attenuating unwanted signal components.
- System Stability: By filtering out high-frequency noise, the system can avoid erroneous interpretations, thereby enhancing operational stability.

B) Filter Requirements

- Cut-off Frequency: The filter's cut-off frequency is selected based on the nature of the signal and the noise. For instance, if the system primarily deals with signals below 1 kHz, a filter with a cut-off frequency just above this range can effectively reduce higher-frequency noise.
- Filter Type: Depending on the application, a low-pass, high-pass, band-pass, or band-stop filter may be chosen. In most cases, a low-pass filter suffices for eliminating high-frequency noise.

C) Mathematical Modelling and Simulation

- Design Approach: The filter design starts with mathematical modelling, typically using tools like MATLAB or Python for simulation. The Butterworth, Chebyshev, or Bessel filter models are commonly used due to their predictable characteristics.
- Simulation: The simulated filter is tested with inputs mimicking real-world signals and noise, allowing for adjustments in filter parameters to achieve desired performance.

D) Integration with Application and Hardware

- Usage Conditions: The filter is primarily used in scenarios where signal integrity is critical, like reading sensor data or during high electrical noise conditions.
- Frequency of Use: The usage frequency depends on the application's dynamics. For instance, in a motor control system, the filter might be applied continuously to the feedback loop.
- Hardware Interaction: The filter interacts with the ADC to process raw sensor data and with communication interfaces (like UART) to ensure clean data transmission.

E) C Implementation and FreeRTOS Integration

- C Implementation: The filter algorithm is implemented in C, considering the MCU's computational capabilities. Functions for filter initialization, data processing, and real-time adjustments are developed.
- RTOS Task Integration: In a FreeRTOS environment, the filter operates as a dedicated task, ensuring timely execution. Task priorities are set based on the filter's criticality to the system's performance.

F) Validation and Integration Testing

- Module Testing: Initial validation is performed in a controlled environment using known signal inputs. The filter's output is measured against expected results to ensure accuracy.
- Integration Testing: Once integrated into the system, the filter's performance is tested under various operational scenarios, including extreme conditions, to validate its effectiveness and reliability.
- Continuous Monitoring: Post-deployment, the system logs and diagnostic tools are used for ongoing performance monitoring, ensuring the filter maintains its efficacy over time.

Test and Validation

After architecting the application, the team planned the execution phase, focusing on task distribution and timeline management.

Tasks were delegated strategically to leverage each member's unique skills.
Collaboration played a key role in critical components like hardware integration and algorithm development. The team committed to daily meetings to assess progress, tackle challenges, and refine plans. A validation plan, involving unit testing and peer reviews, was implemented for individual tasks. This approach facilitated the early detection and resolution of issues. Following the validation of individual components, integration checks were conducted to confirm that the application functioned as planned.

The first iteration of the application underwent functional testing, validating core functionalities against predefined requirements. Feedback from these tests was important in making refinements, paving the way for subsequent iterations.
In the next phase, the control algorithm, initially developed and tested in a simulated environment, was ported into the application. This process involved adapting the code to fit the target hardware environment and ensuring compatibility with other system components. The integrated system was then rigorously tested to verify the control algorithm's functionality. Various operational scenarios, such as random stimuli to the bike system, were simulated to test the control algorithm's effectiveness. Key performance indicators were used to measure the algorithm's efficiency and reliability.

The filtering module, developed and tested in a simulated environment, was adapted for the application using STM32 DSP libraries. Special attention was given to ensure that the filtering module was compatible with data acquisition and processing components. The module underwent data integrity testing to ensure accurate processing and filtering of data. The impact of the filtering module on system performance and response time was assessed and optimized.

The fault diagnosis module, conceptualized and simulated initially, was also integrated into the application. Equipped with error-handling mechanisms, the module effectively managed and diagnosed system faults, utilizing tools like the GDB debugger for troubleshooting. To test the effectiveness of the fault diagnosis module, the system was subjected to various simulated fault conditions. Tests were conducted to assess the system's ability to recover from faults and maintain operational stability.

As a summary:

6.1 First Iterations of the Application

Project Execution Planning

Load Partitioning and Distribution

- Team Roles: Responsibilities were split based on expertise. While the primary responsibility for software development lay with the software (SW) team member, tasks were distributed to utilize the strengths of each team member.

- Collaboration: Critical components like hardware integration and algorithm development saw collaborative efforts, ensuring a blend of perspectives and skills.

Meeting Schedule and Task Validation

- Regular Meetings: The team convened regularly to assess progress, discuss challenges, and adjust plans as needed. (Everyday)
- Validation Plan: For individual tasks, a validation plan involving unit testing and peer review was implemented. This ensured early detection and resolution of issues.
- Integration Checks: After validating individual components, integration checks were conducted to ensure the application worked as planned.

Validation of First Iteration

- Functional Testing: The first iteration underwent functional testing to validate core functionalities against requirements.
- Feedback Loop: Feedback from these tests led to refinements, setting the stage for subsequent iterations.

6.2 Control Algorithm Integration and Testing

Porting and Integration

- Simulation to Application: The control algorithm, initially simulated, was ported into the application. This involved adapting the code for the target hardware environment and ensuring compatibility with other system components.
- Integration Testing: Once integrated, the system was tested to verify that the control algorithm functioned as expected within the application context.

Validation Process

- Test Scenarios: Various operational scenarios were simulated to test the control algorithm's effectiveness like random stimulus to the bike system.
- Performance Metrics: Key performance indicators were used to measure the algorithm's efficiency and reliability.

6.3 Filtering Module Integration and Testing

Integration of Filtering Module

- Code Adaptation: The filtering module, developed and tested in a simulated environment, was adapted for the application using the stm32 DSP libraries.
- Compatibility Checks: Special attention was given to ensure that the filtering module was compatible with data acquisition and processing components.

Validation of Filtering Module

- Data Integrity Testing: Tests were conducted to ensure that the filtering module accurately processed and filtered data.
- System Impact Assessment: The impact of the filtering module on system performance and response time was assessed and optimized.

6.4 Fault Diagnosis Integration and Testing

Integration of Fault Diagnosis

- Code Conversion: The fault diagnosis module, initially conceptualized and simulated, was converted for integration into the application.
- Error Handling Mechanisms: The module was equipped with error handling mechanisms to manage and diagnose system faults effectively using the gdb debugger as a tool for troubleshooting.

Validation and Testing

- Fault Simulation: The system was subjected to various fault conditions to test the effectiveness of the fault diagnosis module.
- System Recovery: Tests were conducted to assess the system's ability to recover from faults and maintain operational stability.

Digital Signal Processing

Using a digital filter for an accelerometer is crucial for processing and extracting meaningful information from the raw accelerometer data. Accelerometers measure acceleration, and the raw data often contains high-frequency noise and disturbances that can obscure the actual signals of interest. Digital filters help enhance the quality and accuracy of the accelerometer data by addressing several key aspects like noise reduction because digital filters can effectively reduce high-frequency noise present in accelerometer readings. This noise may originate from various sources such as vibrations, electronic interference, or external disturbances. By employing a filter, we can smooth out the data, making it easier to identify and analyze the true acceleration patterns. Likewise, signal conditioning with filters allowed us to tailor the characteristics of the accelerometer signal to meet specific requirements. This involves adjusting the filter parameters to emphasize certain frequency components while attenuating others. Signal conditioning is essential for highlighting relevant features and removing unwanted elements. Another thing that filters help us to solve is anti-aliasing, mainly, because when sampling analog signals to obtain digital data, there's a risk of aliasing if the signal contains frequencies higher than half the sampling rate. Digital filters can incorporate anti-aliasing

techniques to prevent aliasing artifacts by attenuating high-frequency components before sampling. Digital filters allow to configure a desired frequency band adjustment. More specifically, different applications may require focusing on specific frequency bands of the accelerometer signal. Digital filters enable us to isolate and analyze signals within a particular frequency range, enhancing the ability to extract relevant information related to the behavior or motion being monitored. Basically, utilizing a digital filter for an accelerometer is essential for improving data quality, reducing noise, and tailoring the signal to meet specific analysis requirements. It plays a crucial role in extracting meaningful information, enhancing the accuracy of measurements, and making the accelerometer data more suitable for a wide range of applications.

Defining the requirements of a filter is a complicated task. Since, there are several common requirements and considerations that apply to filters. We investigated the accounted for criteria like frequency response. Particularly, Passband and Stopband Characteristics, we had to specify the desired frequency range for passing signals (passband) and attenuating signals (stopband). Our filter should have the ability to achieve the desired attenuation in the stopband while allowing the signals of interest to pass through the passband with minimal distortion. Another criteria is the filter type which are either low-pass, high-pass, band-pass, or band-stop. For our application we went for a high-pass filter because we intend to remove noise from gentle motions as described. Likewise, an important component is the order and complexity, for instance, the filter order because the order of the filter determines its complexity and directly affects its ability to shape the frequency response. Higher-order filters provide steeper roll-off in the transition between the passband and stopband but may introduce more phase distortion. Since, we are using a large signal it should not affect much, we verify this later experimentally. Arguably the most important component is the stability of the filter. We had to ensure that the filter is stable under all operating conditions. Accounting for the fact an unstable filter can lead to unpredictable and undesirable behavior, potentially causing numerical issues or

instability in real-time applications. We implemented our filter using a well tested numerical library, so this was not much of an issue. Passband ripple and stopband attenuation are common parameters that specify the maximum allowable ripple in the passband and the minimum required attenuation in the stopband. These parameters are critical for achieving the desired signal quality and noise reduction. Luckily, MATLAB handled this for us. A time critical application like ours must account for computational complexity. We had to consider the computational resources available for implementing our filter. The microprocessor is designed to support digital signal processing so this was not a problem at all. 20 coefficients were recommended at design time, so this is what we end up choosing. Adaptability is recommended and we would’ve liked to implement it. It is believed it could have been of value because filters that can be adjusted in real-time based on changing conditions or requirements offer greater flexibility. We conclude that when designing or selecting a filter, it's essential to carefully balance these requirements based on the specific needs to achieve the desired filtering performance.

In crafting frequency-selective filters, the intended characteristics of the filter are outlined in the frequency domain, specifying the desired magnitude and phase response. Throughout the filter design process, the coefficients of a causal FIR or IIR filter are determined to closely match the specified frequency response. The choice between FIR and IIR filter types hinges on the nature of the problem at hand and the specifications of the desired frequency response (Proakis). In practical applications, FIR filters find use in scenarios where a linear-phase characteristic is imperative within the filter's passband. If a linear-phase characteristic is not necessary, either an IIR filter or an FIR filter may be employed (Proakis). Modern advancements in computer software programs greatly facilitate the design of FIR and IIR digital filters (Proakis).

Filters represent a crucial category of LTI systems. While the term "frequency-selective filter" initially implies a system passing specific frequency components while rejecting others, in a broader sense, any system altering certain frequencies in relation to others is considered a filter (Oppenheim). Designing discrete-time filters involves determining the parameters of a

transfer function or difference equation to approximate a desired impulse or frequency response within specified tolerances. FIR filter design entails polynomial approximation, with distinct techniques for these classes. Notably, FIR design techniques in continuous time only emerged after gaining significance in practical systems, unlike IIR filters. The predominant approaches for designing FIR filters involve the application of windowing techniques (Oppenheim).

Design of Linear-Phase FIR Filters Using Windows

In this method we begin with the desired frequency response specification $H_d(\omega)$ and determine the corresponding unit sample response $h_d(n)$. Indeed, $h_d(n)$ is related to $H_d(\omega)$ by the Fourier transform relation (Proakis).

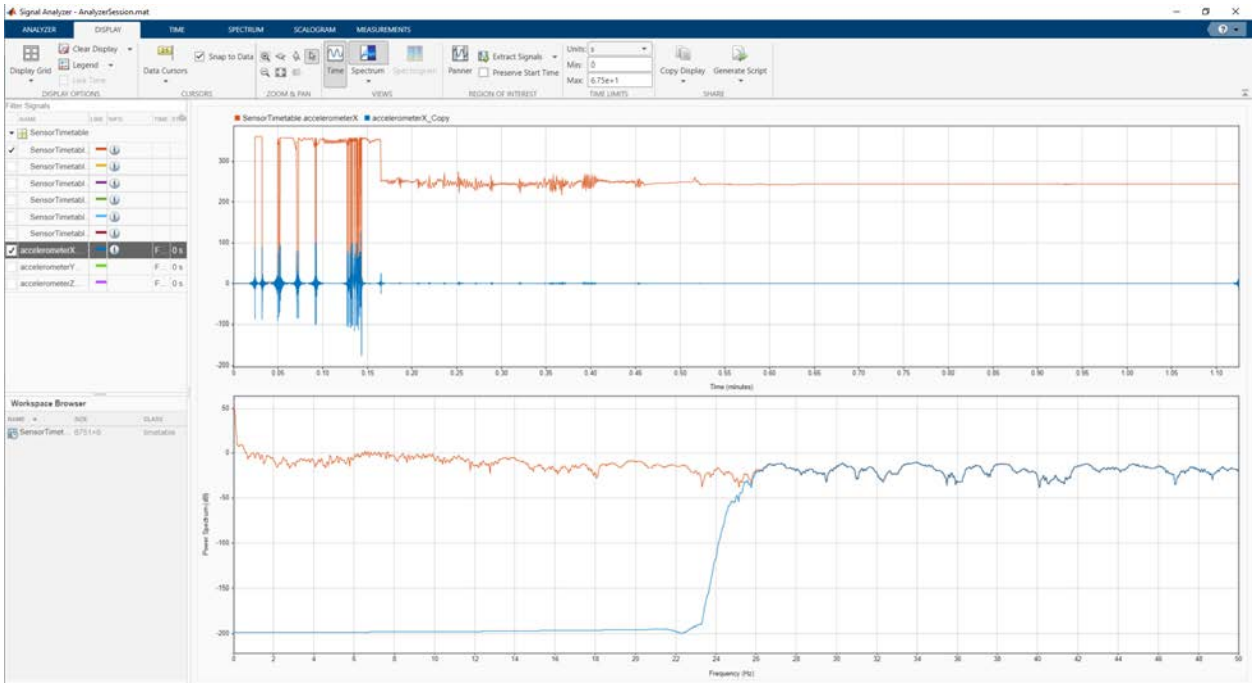
Thus, given $H_d(\omega)$, we can determine the unit sample response $h_d(n)$. In general, the unit sample response $h_d(n)$ is infinite in duration and must be truncated at some point, sat at $n = M - 1$, to yield an FIR filter of length M. Truncation of $h_d(n)$ to a length $M - 1$ is equivalent to multiplying $h_d(n)$ by a “rectangular window”. It is instructive to consider the effect of the window function on the desired frequency response $H_d(\omega)$. Recall that multiplication of the window function $w(n)$ with $h_d(n)$ is equivalent to convolution of $H_d(\omega)$ with $W(\omega)$, where $W(\omega)$ is the frequency-domain representation (Fourier Transform) of the window function. Thus, the convolution of $H_d(\omega)$ with $W(\omega)$ yields the frequency response of the (truncated) FIR filter (Proakis).

The characteristics of the rectangular window play a significant role in determining the resulting frequency response of the FIR filter obtained by truncating $h_d(n)$ to length M. Specifically, the convolution of $H_d(\omega)$ with $W(\omega)$ has the effect of smoothing $H_d(\omega)$. As M is increased, $W(\omega)$ becomes narrower, and the smoothing provided by $W(\omega)$ is reduced. On the other hand, the large sidelobes of $W(\omega)$ result in some undesirable ringing effects in the FIR

filter frequency response $H(\omega)$, and also in relatively large sidelobes in $H(\omega)$. These undesirable effects are best alleviated by the use of windows that do not contain abrupt discontinuities in their time-domain characteristics, and have correspondingly low sidelobes in their frequency-domain characteristics (Proakis).

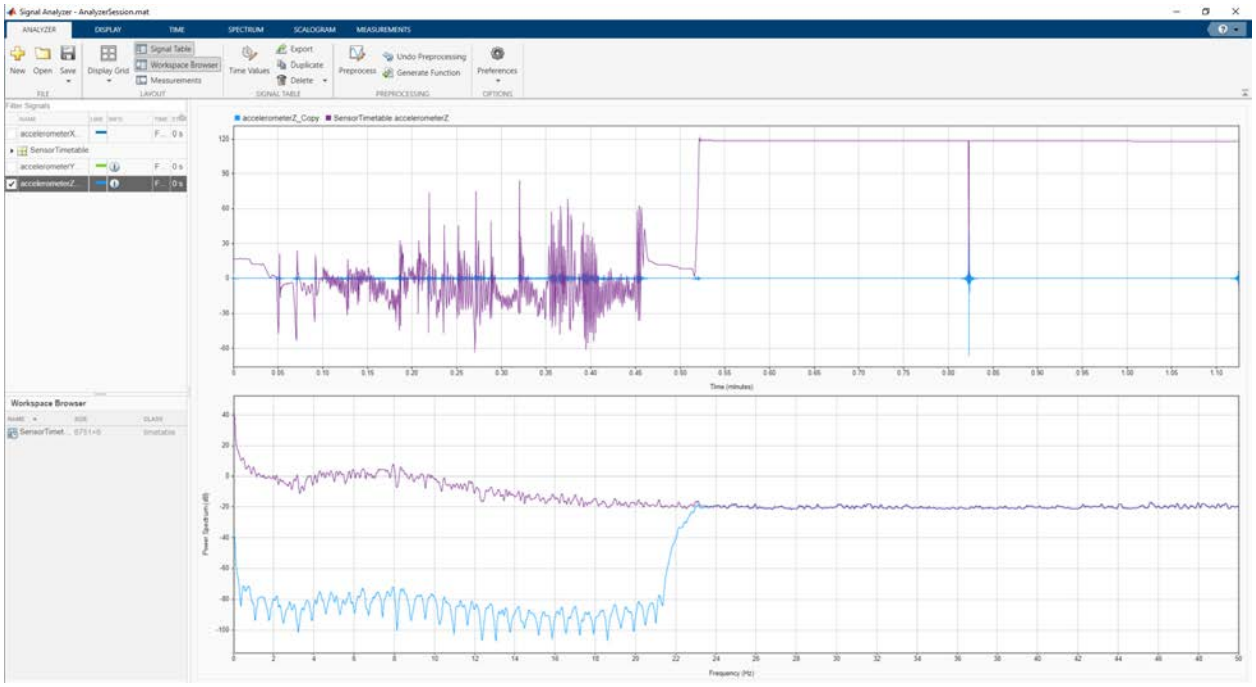
To start our modeling and simulation we extracted information from the Inertial Measurement Unit. We added the data to a Timetable and then use the app called Signal Analyzer from where we cloned the signals and used the copy to preprocess with a high-pass filter and then use the spectrum diagram to compare the effectiveness of the filter, also if it did indeed removed the noise while preserving the spikes. The selection of bands was done completely experimentally. The results for each axis are shown below.

Acceleration in x



Acceleration in y

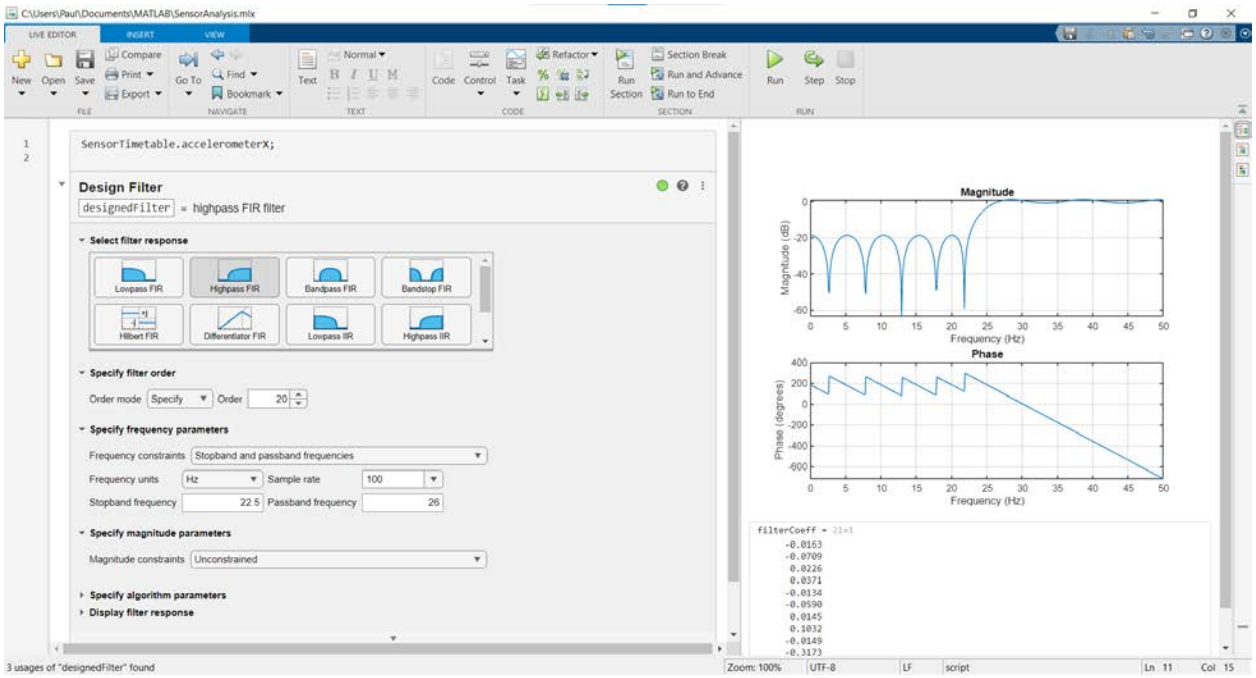
Acceleration in z



Once we plotted and verified the filter properties we went to the Design Filter task in a live script where we would basically replicate the processes and extract the coefficients from a variable.

We only did x because of timing issues. We went for a FIR because according to the lecturer it was the simplest yet functional solution to our application.

Filter coefficients for x



In summary, the methodology was as follows: we had to determine filter specifications which was to identify the specifications of our FIR filter, including the desired filter type like low-pass or high-pass, cutoff frequency, filter order, and any other relevant parameters. Similarly, we had to choose a design method. Fortunately, MATLAB provides various functions for designing FIR filters. We tried the fir1 function which is commonly used for designing filters based on windowing methods. After selecting the function and parameters. We had to specify the filter order which was 20 as a default and any other required passbands and stopbands were retrieved from the signal analyzer. This step yielded the filter coefficients. To test the filter we used directly the data from the Inertial Measurement Unit to create a sample input signal that we used to test the FIR filter. As mentioned, we then plot input and output signals and visualize the input and output signals using MATLAB's plotting functions. This step allowed us to observe the effects of the FIR filtering on the input signal. Naturally, we had to adjust parameters of our

filter because in order to achieve desired performance it is necessary to iterate on the design process by adjusting filter specifications, order, or other parameters. Again, by replotting, we assessed the performance of the FIR filter based on the application requirements. This involved analyzing the frequency response, checking for distortion, and ensuring that the filter meets its design objectives. We tried to explore additional features provided by MATLAB's signal processing toolbox.

As described in previous sections the filter will remove the noise generated by gentle movements and will preserve aggressive motions retrieved by the accelerometer. The filter is embedded in the microcontroller through a snippet of C code. It has already been mentioned how this is achieved but we will still review it briefly. Firstly, we needed to configure the STM32 microcontroller to communicate with the IMU using appropriate communication protocol I2C. Then, we initialized the IMU by setting up necessary registers and parameters for data acquisition. We went for default settings. An snipped to read IMU raw data from the IMU using the configured communication protocol. Furthermore, we processed the raw IMU data through the FIR filter. In other words, we applied the FIR filter algorithm to filter relevant signals obtained from the IMU. This step enhances data accuracy by reducing noise. This high-level methodology guides the overall process of integrating an FIR filter with an STM32 microcontroller and an IMU.

The following is a description of the utilized functions to implement the filter in STM32.

CMSIS DSP Software Library

Finite Impulse Response (FIR) Filters

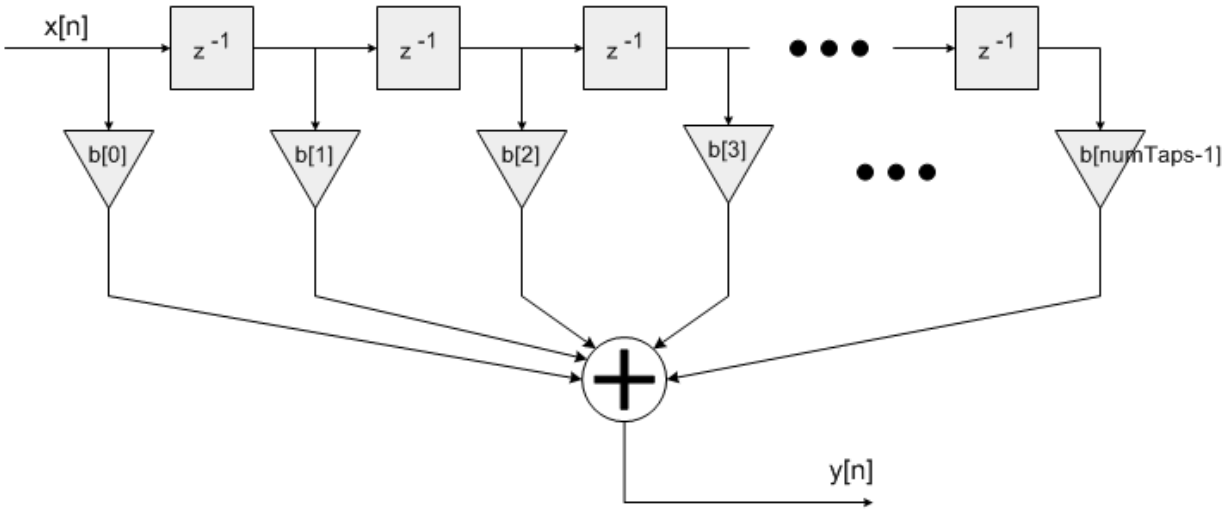
Description

This set of functions implements Finite Impulse Response (FIR) filters for Q7, Q15, Q31, and floating-point data types. Fast versions of Q15 and Q31 are also provided. The functions operate on blocks of input and output data and each call to the function processes `blockSize`

samples through the filter. `pSrc` and `pDst` points to input and output arrays containing `blockSize` values.

Algorithm

The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient `b[n]` is multiplied by a state variable which equals a previous input sample `x[n]`.



`pCoeffs` points to a coefficient array of size `numTaps`. Coefficients are stored in time reversed order.

`{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}`

`pState` points to a state array of size `numTaps + blockSize - 1`

Note that the length of the state buffer exceeds the length of the coefficient array by `blockSize-1`. The increased state buffer length allows circular addressing, which is traditionally used in the FIR filters, to be avoided and yields a significant speed improvement. The state variables are updated after each block of data is processed; the coefficients are untouched.

```
void arm_fir_f32(
```

```
        const arm_fir_instance *S,
        const float32_t *pSrc,
        float32_t *pDst,
        uint32_t blockSize
    )
```

Processing function for the floating-point FIR filter.

Parameters

`[in]` **S** points to an instance of the floating-point FIR filter structure

`[in]` **pSrc** points to the block of input data

`[out]` **pDst** points to the block of output data

`[in]` **blockSize** number of samples to process

Returns

none

```
void arm_fir_init_f32(
           arm_fir_instance_f32 *S,
           uint16_t numTaps,
           const float32_t *pCoeffs
           float32_t *pState,
           uint32_t blockSize
      )
```

Parameters

`[in, out]` **S** points to an instance of the floating-point FIR filter structure

`[in]` **numTaps** number of filter coefficients

`[in]` **pCoeffs** points to the filter coefficients of the buffer

`[in]` **pState** points to the state buffer

`[in]` **blockSize** number of samples processed per call

Returns

none

Complex FFT Functions

Description

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT). The FFT can be orders of magnitude faster than the DFT, especially for long lengths. The algorithms described in this section operate on complex data. A separate set of functions is devoted to handling real sequences.

The FFT functions operate in-place. That is, the array holding the input data will also be used to hold the corresponding result. The input data is complex and contains `2*fftLen` interleaved values as shown below.

`{real[0], imag[0], real[1], imag[1], ...}`

The FFT result will be contained in the same array and the frequency domain values will have the same interleaving.

Floating-point

The floating-point complex FFT uses a mixed-radix algorithm. Multiple radix-8 stages are performed along with a single radix-2 or radix-4 stage, as needed. The algorithm supports lengths of [16, 32, 64, ..., 4096] and each length uses a different twiddle factor table.

The function uses the standard FFT definition and output values may grow by a factor of `fftLen` when computing the forward transform. The inverse transform includes a scale of `1/fftLen` as part of the calculation and this matches the textbook definition of the inverse FFT.

```
void arm_cfft_f32(
           const arm_cfft_instance_f32 *S,
           float32_t *p1,
           uint8_t ifftFlag,
           uint8_t bitReverseFlag
     )
```

Parameters

`[in]` **S** points to an instance of the floating-point CFFT structure

`[in, out]` **p1** points to the complex data buffer of size `2*fftLen`.

`[in]` **ifftFlag** flag that selects transform direction

value = 0: forward transform

value = 1: inverse transform

`[in]` **bitReverseFlag** flag that enables / disables bit reversal of output

value = 0: diables bit reversal of output

value = 1: enables bit reversal of output

Returns

None

```
arm_status arm_cfft_init_f32(
            arm_cfft_instance_f32 *S,
            uint16_t fftLen
      )
```

Parameters

`[in, out]` **S** points to an instance of the floating-point CFFT structure

`[in]` **fftLen** fft length (number of complex samples)

Returns

execution status

ARM_MATH_SUCCESS: Operation successful

ARM_MATH_ARGUMENT_ERROR: an error is detected

The necessary code to implement the filter is

```
#define SAMPLE_TIME 0.01
#define NUM_TAPS 21
#define BLOCK_SIZE 32

const float32_t firCoeffs[NUM_TAPS] = {
    -0.0162988415008265,
    -0.0709271864972703,
    0.0225851291371347,
    0.0371278894358235,
    -0.0134217682863348,
    -0.0589846533854996,
    0.0145099553645701,
    0.103203330310147,
    -0.0148779327102912,
    -0.317339878494921,
    0.515006915991497,
    -0.317339878494921,
    -0.0148779327102912,
    0.103203330310147,
```

```
    0.0145099553645701,
    -0.0589846533854996,
    -0.0134217682863348,
    0.0371278894358235,
    0.0225851291371347,
    -0.0709271864972703,
    -0.0162988415008265
};

float32_t firState[NUM_TAPS + BLOCK_SIZE - 1];  // Filter state buffer
arm_fir_instance_f32 FIR_Filter;
float32_t inputBuffer[BLOCK_SIZE];
float32_t outputBuffer[BLOCK_SIZE];

arm_fir_init_f32(&FIR_Filter, NUM_TAPS, (float32_t*)firCoeffs,
     firState, BLOCK_SIZE);

void IMU_dsp(void) {
     arm_fir_f32(&FIR_Filter, inputBuffer, outputBuffer, BLOCK_SIZE);
}
```

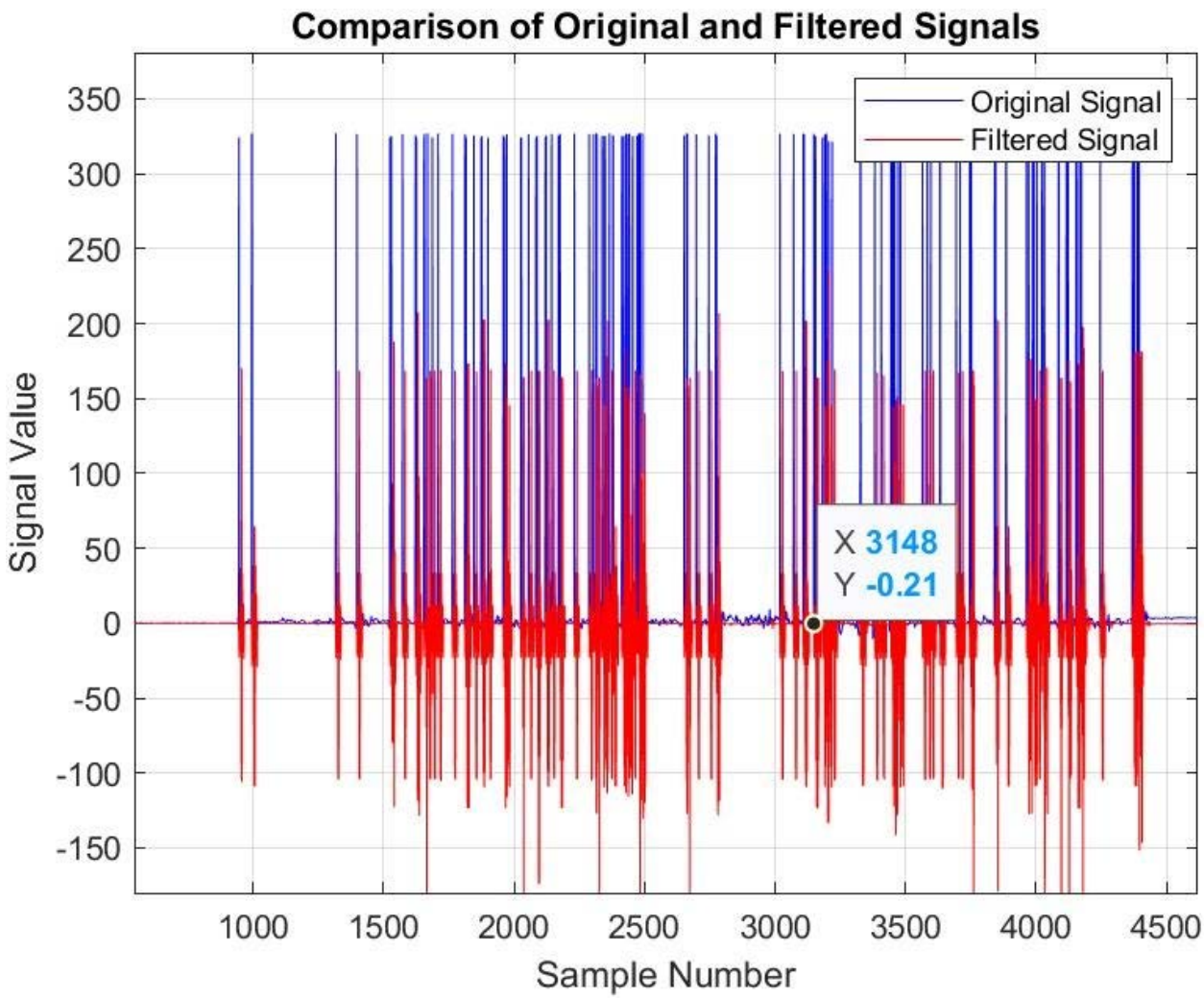
Those are the generalities. Now the task itself is as follows.

```
void StartIMUprocess(void const * argument)
{
  /* USER CODE BEGIN StartIMUprocess */
     static size_t bufferIndex = 0; // Keeps track of the current
position in the buffer
  /* Infinite loop */
  for(;;)
  {

       IMU_acceleration();
       inputBuffer[bufferIndex] = acc_z; // Store the current sample in
the buffer
       bufferIndex++; // Move to the next position in the buffer

       if (bufferIndex >= BLOCK_SIZE) {
            bufferIndex = 0; // Reset buffer index
```

```
            IMU_dsp();
            acceleration_data_conversion(acc_x, acc_y,
outputBuffer[BLOCK_SIZE], message); //Sends the data
      }

      osDelay(2);
  }
```

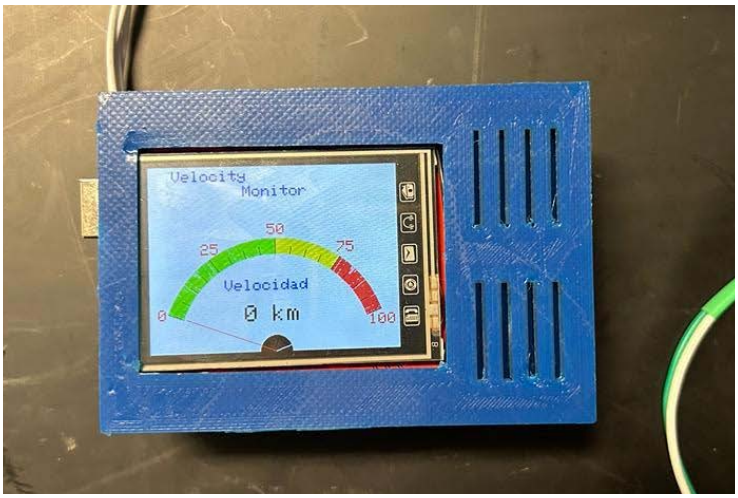
It will be running to retrieve data from the accelerometer in blocks specified by the user.

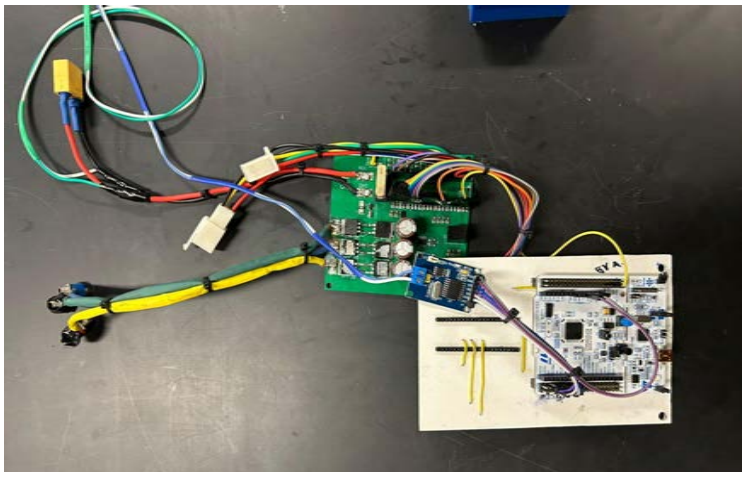
Validating a filter during an integration process involves assessing its performance in the context of your specific application and requirements. These were some of the things we researched on to perform validation. We generated some test signals that cover the expected range of input signals in our application.We included signals with varying frequencies, amplitudes, and dynamics. The signal represented real-world scenarios. Basically, we plugged the IMU and checked the data output by agitating it to introduce noise, disturbances, or other environmental factors to help evaluate how well our filter handles these conditions. We also used analysis tools to examine our filter's behavior across different frequency components. We plotted the frequency response in time-domain and ensured that the filter is attenuating unwanted frequencies while passing desired ones.

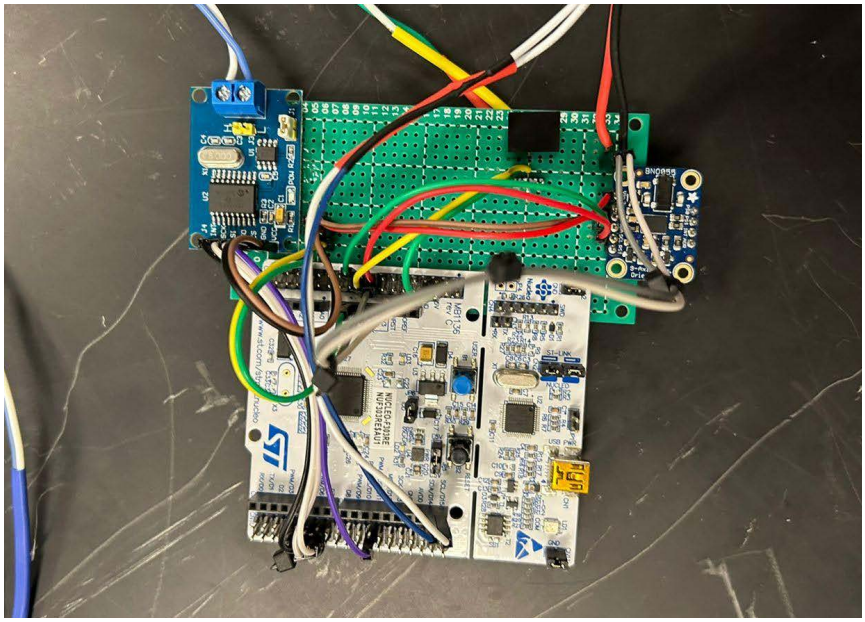


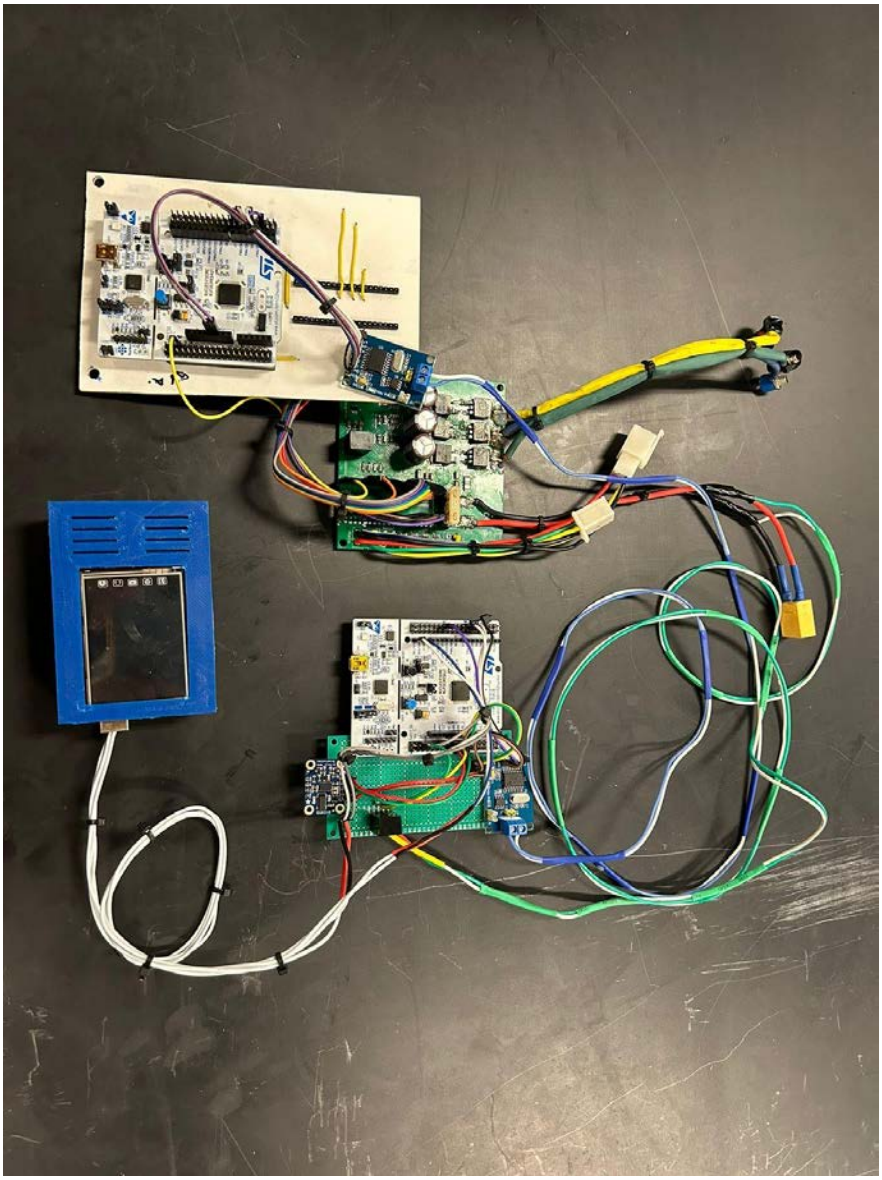
We have yet to test if the filter is fast enough for the real time application. We plan to do this by measuring the processing time meets the real-time constraints of our system. By the displayed results we believe it is necessary to perform some iterative optimization, namely, that based on the validation results, we must consider making adjustments to the filter design or parameters. By tweaking filter coefficients, adjusting the filter order, or fine-tuning other parameters to enhance performance. Last but not least an end-to-end system integration. Basically, to integrate the filtered signals into the larger system and evaluate the overall system performance. We will ensure at some point in the future that the filter seamlessly integrates with other components and contributes positively.

Final Integration










MIGHTY ROCK
BENOTTO
BENOTTO

References

Agarwal, T. (2014, 23 agosto). *Under and overvoltage protection circuits and workings*. ElProCus - Electronic Projects for Engineering Students. https://www.elprocus.com/under-and-overvoltage-protection-circuit/#:~:text=1.,mains%20supply%20take%20place%20frequently.

Alvarez, G. P. (2020). Real-Time fault detection and diagnosis using intelligent monitoring and supervision systems. En *IntechOpen eBooks*. https://doi.org/10.5772/intechopen.90158

Flinn, C. (s. f.). *Overload and overcurrent protection*. Pressbooks. https://pressbooks.bccampus.ca/basicmotorcontrol/chapter/overload-and-over-current-protection/

Michigan Scientific Corp. "*Estudio de caso de transductor de pulso de rueda | Comparación de datos GPS y WPT.*" Michigan Scientific Corp, 5 August 2020, https://es.michsci.com/comparing-gps-and-wpt-data/. Accessed 4 December 2023.

Welsh, Jonathan. "*Which Is More Accurate: Speedometer or GPS Device?*" The Wall Street Journal, 6 January 2009, https://www.wsj.com/articles/SB123119286106955181. Accessed 4 December 2023.

CMSIS-DSP: Finite impulse response (FIR) filters. (n.d.). https://arm-software.github.io/CMSIS-DSP/latest/group__FIR.html

CMSIS-DSP: Complex FFT Functions. (n.d.). Arm-Software.github.io. Retrieved December 6, 2023, from https://arm-software.github.io/CMSIS-DSP/latest/group__ComplexFFT.html

Proakis, J. G. (2014). *Digital signal processing.* Pearson.

Oppenheim, A. V., Schafer, R. W., & Buck, J. R. (1999). *Discrete-time Signal Processing*.