Weber State University

Senior Design Documentation

ORC - Off-Road Roll Control

Corbin Roennebeck
April 16, 2025

Date	Revision	Release Notes
	Rev 01	Initial Release

Figure 1: Completed ORC test vehicle

Contents

List of Figures

List of Tables

List of Abbreviations and Definitions

ADC	Analog to Digital Converter
AHRS	Automatic Heading and Reference System
DAC	Digital to Analog Converter
ESC	Electronic Speed Controller
ESP	Espressif
FreeRTOS	Free Real Time Operating System
HAL	Hardware Abstraction Layer
IDF	Integrated Development Framework
IMU	Inertial Measurement Unit
IOT	Internet Of Things
ISR	Interrupt Service Routine
LiPo	Lithium Polymer
MCU	Micro-Controller Unit
ORC	Off-road Roll Control
RC	Radio Control
SOC	System On Chip
VSCode	Visual Studio Code

1 Introduction

Off-road Roll Control (ORC) is inspired by the designer's passion for rock crawling and modifying his 1996 Jeep Cherokee to be better at this task. In this endeavor, many decisions had to be made that traded comfort or on-road stability for off-road performance. This revealed a gap in the market where the active suspension systems present in modern luxury vehicles were not reaching in to the aftermarket. Bringing active suspension to off-road vehicles can improve comfort, stability, and performance simultaneously.

To make this ambition fit into a short project, a 1/10th scale prototype was constructed. The smaller size helped bring down costs as well as reduce the need for extensive and complex mechanical engineering.

To have a more quantitative measurement, the roll, pitch, and jerk of the vehicle frame is analyzed. Reducing these dynamics should correlate with an improvement in comfort, stability, and performance.

2 Scope

The control algorithm, it's implementation and performance will be discussed. The hardware design will be extensively covered, analyzing both the strengths and shortcomings of the design. The assembly of the Radio Control (RC) vehicle will be excluded, as the exact vehicle is hardly important to the design, but the modifications made will be noted. The open source libraries and operating system utilized will only be covered at a high level, with the inner working only described when necessary. Data analysis will be detailed, again only at a high level, as the exact implementation of library functions is not necessary to understand their results. Statistical test will be discussed and the reader is expected to be familiar with the test and their calculation. PCB construction is excluded from this discussion (the reader is expected to be knowledgeable in this matter), except for the necessary repairs after fabrication. Potential improvements to the design, barring their low level implementation, will also be discussed.

3 Design Overview

3.1 Requirements

ORC's main goal is to significantly reduce the experienced roll and vertical jerk compared to a vehicle with no dampers or anti-roll bars. This will be measured from the main chassis where a passenger would be located. The roll and jerk are analyzed over rough and uneven

terrain. The system will provide a way to log vehicle dynamics and share them to a PC for analysis. For the best results, the logged data collection rate should be maximized as much as possible.

A system contained to only the equipped vehicle is desirable. This guarantees the system and vehicle can operate without preparing a route and without prior knowledge of the particular terrain. The system will only complement the suspension, meaning all major components (springs, linkages) should not be modified or replaced as changes to these components could impact vehicle dynamics significantly. Replacing large parts of the suspension would make it hard to examine the specific effects of ORC.

3.2 Constraints

Consistent with WSU's ECE department requirements, ORC will be complete and demonstrated to the faculty before graduation. This is set to be April of 2025.

ORC doesn't have a set maximum budget, but cost are not expected to exceed $3000 USD.

3.3 Applicable Standards

ORC primarily uses SPI interfaces to communicate with peripherals, but the SD bus protocol, USB, and UART are also present.

3.3.1 SPI

Multiple SPI busses are utilized to communicate with the Digital to Analog Converter (DAC), Inertial Measurement Unit (IMU), and module memory. The module's memory bus is managed by the MCU's operating system - Free Real Time Operating System (FreeRTOS) - and the others as peripherals with varying clock speeds, polarities, and sampling types. The implementation of these is handled by peripheral drivers included in the Espressif (ESP) Integrated Development Framework (IDF).

3.3.2 SD Bus

Again, peripheral drivers in the ESP IDF handle the implementation of the the SD bus protocol. Unsurprisingly, this is used to interface with an on-board SD card. This SD card stores the selected vehicle dynamics described in section 1 and 4.7.

3.3.3 USB and UART

The use of USB and UART protocols is limited to the programming of the MCU and some debug messages. The programming circuitry includes a USB to UART bridge and facilitate communication between a PC and the MCU. This is described in more detail in section 4.1.5.

3.4 Dependencies

The obvious requirement of ORC is to have a vehicle to attach it to. This implementation uses RC4WD's Trail Finder II with leaf spring suspension on all four corners. This kit requires a motor, electronic speed controller, radio control transmitter, radio control receiver, and a battery to be drivable. A body should also be supplied to protect the electronics from debris and unplanned inversions. More details on the entire mechanical system, including the vehicle, can be found in section 4.9.

The ORC PCB is powered by two batteries with xt60 connectors, and while any battery chemistry may work, Lithium Polymer (LiPo) batteries are recommended for their high burst currents and reasonable energy density. For the best performance, 3 cell LiPo batteries exceeding 2000 mAH are recommended, but 2 cell 1000 mAH batteries are sufficient for the board power supplies. Battery pack voltage should not exceed $24V$ when fully charged. The board also offers no battery charging or protection beyond over-current draw, these must be provided externally. The charging circuit does not need to be contained on the vehicle. LIPO protection is necessary while the batteries are connected to the ORC system. A simple low voltage alarm is sufficient protection so long as the batteries are left unplugged when not in use.

Data analysis is done with a Python 3.12 script. Recompilation, which is currently required to rename the plots and change the input file location, requires the following libraries: numpy, scipy, matplotlib, csv, collections.

The roll control program is compiled and loaded to the MCU with the ESP IDF v5.4.0. This links all the necessary peripheral and FreeRTOS libraries (not included in this document) to the roll control code.

3.5 Theory of Operation

The theory of operation is summarized in figure 2, with a more detailed explanation to follow:

At it's core, ORC is 3 PID controllers in parallel. The pitch, roll, and vertical acceleration of the vehicle are the dynamics that the PID controllers seek to control. This control is

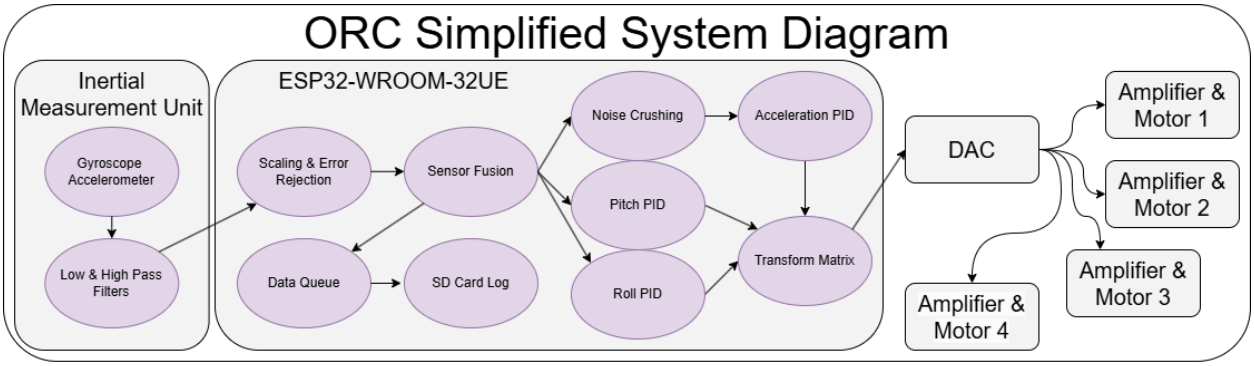


Figure 2: System Overview

accomplished by combining the 3 PID controller outputs into 4 actuator movements. The combination is simplified by use of the small angle approximation. The exact formula for this transformation is found in section 4.4.3. The actuator movements are sent to the drivers via a $\pm 2.5V$ analog signal, generated by the included DAC. This signal is amplified by the bridge drive circuit, turning the $\approx \pm 12V$ supply into a $-16V$ to $+16V$ output. More details in section 4.6.

The DC gear-motors that compromise the actuators are driven directly off this bridge drive. To turn the rotational force into an approximately linear force, a lever arm and linkage is used. The end of this lever arm is positioned directly above the axle of the vehicle, and the linkage that goes between the two is mostly horizontal. More details on this mechanical system are in section 4.9.

Doing this control system in the real world necessitates the need for sensors, in this case an IMU. The selected IMU outputs 7 measurements, linear acceleration in x,y, and z; angular rate over x, y and z; and temperature. For ORC the temperature measurement can be ignored. To get a pitch and roll reading from these measurements, the angular rate must be integrated. To keep this integrand from drifting too far from it's true value, it is checked against the direction of gravity. Both of these are accomplished by the sensor fusion algorithm described in section 4.3.

The data collection part of ORC is accomplished by taking the rectified sensor data and saving it to an SD card. Due to the limits of random access on an SD card, a large data queue is implemented and the data is written in batches, described in detail in section 4.7.

3.6 Design Alternatives

While designing ORC, several alternatives could have been implemented. The choice of actuators is perhaps the most influential. The ideal choice would have been voice coil motors. These would have been driven by variable current control much the same as a the selected gear motors, but are better suited for the constant force application found here. The major drawback, and ultimate reason for the voice coils not be selected was cost. At the force and stroke length needed to cover the entire suspension travel, the actuators were prohibitively expensive.

A vastly cheaper alternative would have been to use servo motors. The drive could be removed and the motor could instead be driven by a driver similar to what was implemented. Servos were not chosen as the force required for the suspension to rotate the motor as it flexed was too high. This created a strong damping effect that was not active. This defeated the purpose of removing the dampers in the first place and could lead to better performance simply by connecting the suspension to the servo. There are servos that have minimal back-spin-torque, but the cost of these exceeded the voice coil motors.

The variable current control was selected as the motor drive over a more standard PWM control as it allowed the force applied by the motor to be constant instead of pulsed. The PWM control would reduce power consumption and waste heat, but could create vibration in the system at low duty cycles. Perhaps very high frequencies could mitigate this, but then custom drives would still be necessary and the extra complexity would increase part count and development time significantly.

The IMU selected was the highest performance available on a cheap development board. Having the sensor on it's own board also allowed it to be placed more optimally. This also brought overall cost down as a 3rd board did not need to be developed. To decrease development time, an Adafruit board was selected as they had extensive application examples. The highest sampling rate IMU was selected from Adafruit's library. This topped out at 6.7kHz. Higher performance IMUs exist, and with more time and budget their implementation could improve performance. There also exist fully embedded Automatic Heading and Reference System (AHRS) systems that could provide vehicle dynamics information while also reducing MCU load. These have a very large price range, but all start quite far above the Adafruit IMU board. These sensor fusion algorithms could be higher performance than this implementation, but that would need to be more thoroughly investigated. The main benefit that could be realized here, besides increased accuracy, is even higher sampling rates and freeing up MCU clock cycles.

4 Design Details

4.1 ESP32

The ESP32 is a low cost System On Chip (SOC) aimed at Internet Of Things (IOT) applications. Available as complete modules or lone IC's, the ESP32 series features up to two processors running a custom version of FreeRTOS and peripheral hardware such as Analog to Digital Converter (ADC), DAC, SPI, UART, I2C, Pulse counters, Bluetooth, Wi-Fi, and more. The wide array of peripherals on a board alongside a price of just a few dollars for a module makes the ESP32 an attractive choice for many hobbyist and professionals developing embedded systems. This has fostered a large knowledge base and a very feature rich IDF.

The dual core processors and integrated communication protocols is of great interest to this project. This allowed both the control logic and data logging to run truly in parallel while share peripherals and freed up development time that would have been spent debugging and optimizing a ground up solution for SPI bit banging.

4.1.1 ESP IDF

SOCs wouldn't be much use without a way to program them. For this purpose, Espressif has devised the ESP IDF. This tool chain allows c and c++ programs to be compiled for and written to ESP32 targets. Also included are a vast collection of libraries essential to make use of SOC's many features. While the tool chain can be used on files made in vim, nano, notepad, or the likes, a much better experience can be had with full featured IDEs such as Visual Studio Code (VSCode) and Clion. Of these IDEs, VSCode is the easiest to set up. An official extension exist that streamlines the setup of the IDF and includes debugging features. The details on setting this up are best described in official ESP documentation.[1]

With a working IDE and IDF, it is rather trivial to create a new project and upload it to the board. Again, official ESP documentation guides this process well. The only common complication here is the need for USB drivers. This is dependent on the OS used and what drivers are pre-installed or can be automatically installed. Using Windows 10 and a Silicon Labs CP2102 USB-UART bridge yielded very good results with only a few minor steps necessary to facilitate communication (see section 4.1.5 for details).

4.1.2 FreeRTOS

FreeRTOS allows powerful enough SOCs to effectively multi-thread. FreeRTOS also helps manage memory on these systems and provides watchdogs to monitor threads (tasks as they are known) and their memory usage. The ability to start multiple processes with multiple cores allows truly parallel operation.

The OS also gives the ability to create scheduling paradigms with built in semaphores, mutexes, queues, notifications, and other complex data structures. Many of these provide specific functions to modify their contents from an Interrupt Service Routine (ISR), particularly useful is the notification paradigm that allows one task to unblock another with a status code passed between. In ORC, the use of notifications allows a data ready interrupt from the IMU to trigger a polling transaction that collects the IMU data. The collected data is also eventually passed between cores using a queue. This queue allows the receiving task (the data logger) to process data in batches without missing any data points when a longer memory writing operation comes along.

The FreeRTOS distribution for the ESP32 offers extensive configuration options. Because this SOC collection is aimed at IOT applications, the performance of the device can be tuned to meet strict (battery) power requirements. In the case of ORC, the processing speed needed to be maximized to achieve the highest sampling rate possible. Configuration is accomplished through the automatically generated "sdkconfig" file. By modifying the file as summarized in table 1, the ESP32 achieves significantly higher execution speeds.

4.1.3 SPI Peripheral

The builtin peripherals of the ESP32 allow the SOC to schedule multiple communication transactions without blocking the processor for significant amount of times. The SPI peripheral is no exception. There are 4 SPI controllers on the ESP32, 2 of which are meant for accessing on chip/module memory, leaving 2 for general usage.

ORC uses only one device on each of the 2 general purpose SPI busses. This configuration isn't strictly necessary, as the SPI peripheral supports different modes (clock edge and idle level) for different devices on the same bus and different message lengths at different speeds. To further speed up ORC execution, simultaneous reading of the IMU and writing to the DAC could be implemented, requiring separate busses and thus controllers.

The IMU SPI bus is configured with 1 command bit for reading and writing and 7 address bits. SPI mode 3 (read on rising edge, clock high in idle) is used at a speed of 9 MHz. The DAC SPI bus is configured with 4 command bits for the possible commands and 4 address bits to specify a channel. SPI mode 0 (read on rising edge, clock low in idle) is used at a speed of 50 MHz.

4.1.4 SD Bus Peripheral

The SD bus communication protocol is similar to SPI with the addition of several more data lines. For the microSD card used in ORC, a 4 bit data bus at 45 MHz facilitates logging 3 data points 3332 times per second. The SD bus peripheral also provides functions

Option	Setting
CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION_SIZE	not set
CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION_PERF	=y
CONFIG_ESPTOOLPY_FLASHMODE_DIO	not set
CONFIG_ESPTOOLPY_FLASHMODE_QIO	=y
CONFIG_ESPTOOLPY_FLASHFREQ_40M	not set
CONFIG_ESPTOOLPY_FLASHFREQ_80M	=y
CONFIG_COMPILER_OPTIMIZATION_DEBUG	not set
CONFIG_COMPILER_OPTIMIZATION_PERF	=y
CONFIG_SPI_MASTER_IN_IRAM	=y
CONFIG_ESP_SPI_BUS_LOCK_FUNCS_IN_IRAM	=y
CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_240	=y
CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ	=240
CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER	not set
CONFIG_FLASHMODE_QIO	=y
CONFIG_FLASHMODE_DIO	not set
CONFIG_OPTIMIZATION_LEVEL_DEBUG	not set
CONFIG_COMPILER_OPTIMIZATION_LEVEL_DEBUG	not set
CONFIG_COMPILER_OPTIMIZATION_DEFAULT	not set
CONFIG_ESP32_DEFAULT_CPU_FREQ_160	not set
CONFIG_ESP32_DEFAULT_CPU_FREQ_240	=y
CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ	=240

Table 1: sdkconfig modifications for higher performance

for mounting a file system to the card. This allows the data to be formatted nicely in a CSV file.

4.1.5 Programming Considerations

The ESP32 requires bootstrapping to choose between normal operation or download mode. There are also bootstrapping options to select flash memory voltages and timing. Most of these pins need to be pulled high at boot for normal operation and internal pullups exist for this purpose. This is compatible with the devices already attached to these pins with the exception of GPIO 12 and GPIO 2. These pins must be held low for the internal voltage to be set correctly and the chip to enter download mode, respectively. Both these pins are used on the SD bus data lines, which must be pulled up for proper operation. The ESP32-WROOM-32UE Datasheet (version 1.7) describes these boot configuration in great

detail.

To work around the issues with GPIO 2, an NPN transistor is connected to the USB-UART bridge's RTS and DTR lines (see Appendix A). This fix is an extension of the transistors used on GPIO 0 and the EN input in the schematic for the ESP32 development kit, where the transistors can selectively pull down the pins they are attached to. When not pulled down by the transistor, pullup resistors (either internal or part of the SD bus) bring these pins high. These work together to correctly choose what mode to boot into.

The conflict on GPIO 12 can be resolved by burning e-fuses in the ESP32. Setting both the *EFUSE_SIO_TIEH* and *EFUSE_SDIO_FORCE* fields to 1 correctly sets the flash voltage to $3.3V$. A python script (espefuse.py) provided in the ESP IDF can handle this process automatically. By burning these e-fuses, GPIO 12 is no longer used to determine the flash voltage, and thus a pullup resistor can be connected without issue.

A boot button and reset button are provided on the ORC main board. These also come from the development kit, allowing the module to be reset or put in download mode manually. This proves useful in debugging and stopping a boot looping program from blocking new programs from being downloaded.

The USB-UART bridge is also a development kit implementation. This provides a communication interface between a PC and the ESP32 for programming the device and receiving status messages. A significant error exist in this implementation. The TX and RX lines should cross over such that TX of the bridge goes to RX of the ESP32 and vice versa. The failure to do this crossover prevents the ESP32 from receiving any data from the PC.

4.1.6 Status LEDs

The ORC main board has provisions for up to 5 status LEDs. One is tied to constant power to signal the board has power, another is tied to the SD card slot's card detect pin to indicate if a card is inserted. Another is tied to GPIO 0 and signals that the program has booted and is ready for operation. The final two share a pin with the interrupts from the IMU. It was originally uncertain if the interrupt signals would be used, but interrupt 1 was implemented as a data ready signal. Interrupt 2 is not used and there is no strong need for another status to be displayed anyways, but removing the connection to interrupt 2 would allow this LED to be utilized as a data recording indicator if desired.

4.2 IMU

As previously mentioned, the IMU provides acceleration and angular rate data to the Micro-Controller Unit (MCU). A wide variety of these sensors are available, ranging greatly in precision, measurement ranges, sampling rates, and cost. The ISM330DHCX met a good

middle ground in all these respects. A large deciding factor was the availability of a Hardware Abstraction Layer (HAL) library that could do complex configuration in a single command. No part of the initial selection decisions, but proving useful late in the project, is the on-chip filtering for the output data. This helped significantly improve the quality of the returned data without requiring the MCU to do any more work.

4.2.1 Daughter-board

To make the positioning of the IMU easier and to reduce the complexity of the ORC main board, the IMU was placed on a daughter board with wires soldered to it and a connector on the main board. This allowed the IMU to be placed closer to the vehicles center of mass. The IMU could also just be bought pre-soldered on a board with all the necessary passive components already attached. An Adafruit board - part number 4569 - was selected. In retrospect, Adafruit part number 4502 would have been a better choice as the magnetometer was not used and would have greatly reduced board size. This development board being low cost was also a large factor in selecting this IMU.

4.2.2 HAL

The manufacturer of the ISM330DHCX provides a library to interface with the device. This library is written in C and is processor agnostic, as the actual communication interface, SPI in this case, must be implemented by the end user. In the case of the ESP32 this only requires the SPI bus be configured and a transaction be created and sent to the driver.

This library provided functions to read and/or modify every register on the device, bit-wise in some cases. It also un-abstracted the settings for these registers, giving the possible settings human readable names in several enumerations.

4.2.3 Configuration

For ORC, the IMU is configured to output data at a rate of 3332 Hz and Both the accelerometer and gyroscope are put in high performance mode. The accelerometer is configured to pass data through 2 low pass filters with a cutoff of 833 Hz and 400 Hz. The gyroscope data is put through a 65 mHz high pass filter and a 4.3 Hz low pass filter. No further filtering is needed before the data is processed in the sensor fusion algorithm (section 4.3). Lastly, the IMU's interrupts are set to trigger when new data is ready for the accelerometer, which is in sync with the gyroscope. The interrupt is push pull type, which seemed most reliable for edge-triggered interrupts in testing. The interrupt is also latched on till data is read, which isn't necessary using edge-triggered interrupts, but does allow logic

analyzers to measure how long it takes for the data to be read after an interrupt is triggered. This was very useful while debugging.

4.2.4 Issues

The IMU setup is not without issues. The SPI interface isn't perfect and occasionally flips the first bit of the accelerometer data. These bad data points had to be filtered out, but were few enough not to significantly impact performance. The max reliable speed of the SPI interface at 9 MHz is also pretty abysmally slow. With more optimized programming and data processing, this interface speed and the max data rate of 6.6 kHz would be a hard limit in how smooth the system could be operating.

4.3 Sensor Fusion

In the continuous domain, gyroscope data could be integrated over time to perfectly track the pitch, roll, and bearing of the system. In the discrete domain, sampling errors compound quickly to make this method unreliable. Thankfully, algorithms have been developed that use acceleration due to gravity to reference the integrated gyroscope data. Even better, open source libraries exist that do these calculations in an efficient matter.

Fusion[7] is library developed by xioTechnologies for use in their data logging products. They have kindly released this software under the MIT license on GitHub.

4.3.1 Implementation

Fusion requires just a few function calls to initialize it, then just needs to be updated every time new IMU data is available. The sampling period reported by the IMU is used in Fusion's calculations. The standard settings of this library appeared to be appropriate for this application.

Because ORC does not require that the bearing of the vehicle be tracked, magnetometer measurements are not provided to Fusion. Per their documentation, this acceptable.

4.3.2 Issues

The library itself achieves very high computational performance and decent accuracy, but the reliance on a gravity presented some issues described in detail in section 5.3.4.

4.4 PID Controllers

Three distinct PID controllers are the heart of the control system. These controllers work to independently control the pitch, roll, and vertical acceleration.

4.4.1 PID Implementation

The PID control library is largely based on Phil Salmony's "PID Controller in Software" video from his YouTube channel - Phil's Lab.

This PID controller varies from the traditional model in two distinct ways. The integrator implements some anti-wind-up functionality and the differentiator implements a low pass filter.

The integrator anti-wind-up functions by limiting the maximum value the integral term can be. This limit is equal to the output limit less the proportional term. The output limit is determined while initializing the PID control. If the proportional term is greater than the output limit, the integral term is set to zero. This prevents the integral term from saturating the input when the setpoint can't be achieved (such as a extended period spent off camber).

The differentiator has been programmed to act only on measurement. In this application, where the setpoint isn't changing, this isn't strictly necessary. The single-pole low pass filter then seeks to lower the impact of noise.

The output of this PID controller is clamped to ensure it doesn't produce too large of values for the following transformation (section 4.4.3).

4.4.2 Additional Filters

The only other filter to note is the noise rejection applied to the acceleration input. This filter's output is zero when the input is below the noise threshold (both positive and negative input/outputs) and equal to the input minus the noise threshold otherwise. This helps the acceleration PID act only on large values.

4.4.3 Transform Matrix

The transform matrix is perhaps the most vital part of the control system. This takes our PID outputs, corresponding to desired pitch, roll, and vertical displacement movements, and translates them to the motor movements on each corner using some small angle approximations. This equation (1) is based on the one written by DI Tan, Chao Lu, and Zueyi Zhang.[6]

$$\begin{bmatrix} f_{fd} \\ f_{fp} \\ f_{rd} \\ f_{rp} \end{bmatrix} = \begin{bmatrix} \frac{l_r}{2(l_f+l_r)} & -\frac{1}{2(l_f+l_r)} & \frac{1}{2(l_d+l_p)} \\ \frac{l_r}{2(l_f+l_r)} & -\frac{1}{2(l_f+l_r)} & -\frac{1}{2(l_d+l_p)} \\ \frac{l_f}{2(l_f+l_r)} & \frac{1}{2(l_f+l_r)} & \frac{1}{2(l_d+l_p)} \\ \frac{l_f}{2(l_f+l_r)} & \frac{1}{2(l_f+l_r)} & -\frac{1}{2(l_d+l_p)} \end{bmatrix} \begin{bmatrix} f_Z \\ f_\theta \\ f_\phi \end{bmatrix} \quad (1)$$

l_f	Distance from sensor to front axle
l_r	Distance from sensor to rear axle
l_d	Distance from sensor to driver side tires
l_p	Distance from sensor to passenger side tires
f_{fd}	Actuator force on front driver corner
f_{fp}	Actuator force on front passenger corner
f_{rd}	Actuator force on rear driver corner
f_{rp}	Actuator force on read passenger corner
f_Z	Control force for vertical displacement
f_θ	Control force for pitch
f_ϕ	Control force for roll

Table 2: Symbol definitions for equation 1

The solution to this equation is fairly trivial. A few implementation notes: The constants for this equation can be precomputed and by seeing the negative terms as subtraction, the transform can be solved for a given control force with four multiplications, four subtractions, and four additions.

In ORC, the transform function also safely converts float inputs to unsigned integer outputs. These outputs are 16 bits long and are centered around 32768. As explained in section 4.5, this is the format the DAC expects to see it's inputs in.

4.5 DAC

In order to command the motor outputs, a multichannel dual supply DAC was required. The LTC2664 provides 4 output channels at 16 bit resolution. In retrospect, the 12 bit resolution would have been acceptable. Importantly the DAC could be configured to output $\pm 2.5V$ without an external reference. This signal range was small enough to be amplified by the minimum amplifier gain and was large enough to be fairly invulnerable to noise causing large output swings.

The LTC2664 having a high speed SPI interface was also a deciding factor. SPI allowed fairly low latency updates with little data framing and formatting needed.

4.5.1 HAL

Borrowing concepts from the IMU's HAL library (section 4.2.2), a library was written for the DAC. This library only contains the necessary functions for ORC.

The main function takes the desired DAC output, adds an necessary offset, ensuring the input won't wrap extend to invalid or nonsense data. Pre-computing the input limits saves clock cycles while trying to do an update.

The partner function to this takes an integer valued offset (positive or negative) and saves this information as two unsigned integers. These unsigned integers can easily be added to or subtracted from the desired input.

4.5.2 Implementation

The DAC was placed as close as possible to the amplifier inputs to reduce the analog signal's trace length. This extended the SPI interface traces significantly, but there have been no issues with reliable communication. The datasheet's recommended passive components were implemented on the ORC board to ensure proper operation. The data out from the DAC could be left disconnected as reading back the input or daisy chaining the device was not desired.

4.5.3 Issues

The only problem ORC had with the DAC was a mix up of the SPI lines. The data out from the MCU should have been connected to the data in on the DAC. This was instead connected to the data out of the DAC. This is the exact same issue noted in section 4.1.5.

4.6 Motor Amplifiers

To manipulate the suspension, fairly large gear motors were ultimately picked. With some testing, these $24V$ motors wouldn't instantly burn up when stalled at $16V$ continuously. At this voltage, the motor draws about $2A$.

In designing a circuit for this, minimizing the required overhead voltage and ensuring the driver could continuously provide 2 A were priorities. Browsing amplifiers and their datasheets revealed great amplifiers that would have performed better, but their cost was absurdly high. The LM675 chosen provided good balance for cost to performance.

A custom circuit had to be constructed to drive the motors rather than an off the shelf motor drive. Varying the current allows the motor torque to vary, and thus the force exerted on the suspension. Commercially available motor drives that drive varying current don't

really exist at this time. Most drives use PWM to drive the full voltage periodically. This maximizes torque even at low speeds.

4.6.1 Circuit

To minimize the overhead voltage required, a modified bridge drive circuit is used. The bridge drive is compromised of both a inverting and non-inverting amplifier. This allows the two outputs to be connected to the two motor terminals, letting a 16 V to -16 V output range be realized with just a $\pm 11.1V$ supply.

4.6.2 Heat-sink Considerations

According to both my calculations and the LM675's data sheet, operating at full tilt requires a heat sink capable of better than $10° \frac{C}{W}$ per amplifier. The cheapest and smallest heat sink that could almost accomplish this was a dual package design capable of $6.4° \frac{C}{W}$ passively. This doesn't meet the requirement for operating at full tilt; however, the amplifier isn't expected to be operating at max for extended periods. Higher performance could be realized at great expense to board space on both the main board and amplifiers.

4.6.3 Stability

The input to the circuit is reduced with a simple resistor divider to give the amplifier a $\pm 0.8V$ input to achieve the 8 to $-8V$ range required per amplifier. This provides the minimum $10V/V$ gain need for the amplifiers to be stable.

The stability of the constructed amplifiers was checked. A sample of this test, in the form of a bode plot is found in figure 3.

4.6.4 Calibration

Because of the analog nature of the amplifiers, there is a need to deal with varied DC offsets of each amplifier. The chosen design utilizes a comparator to iteratively move the DC offset up or down till a small enough error is realized. Channels are selected through an analog mux.

This solution would have worked, and did work at low voltages. The problem with the current design is the mux has no current limiting resistors. At higher voltage, this led to excess current heating up and destroying the mux. The simple fix would be to add current limiting to the mux circuit.

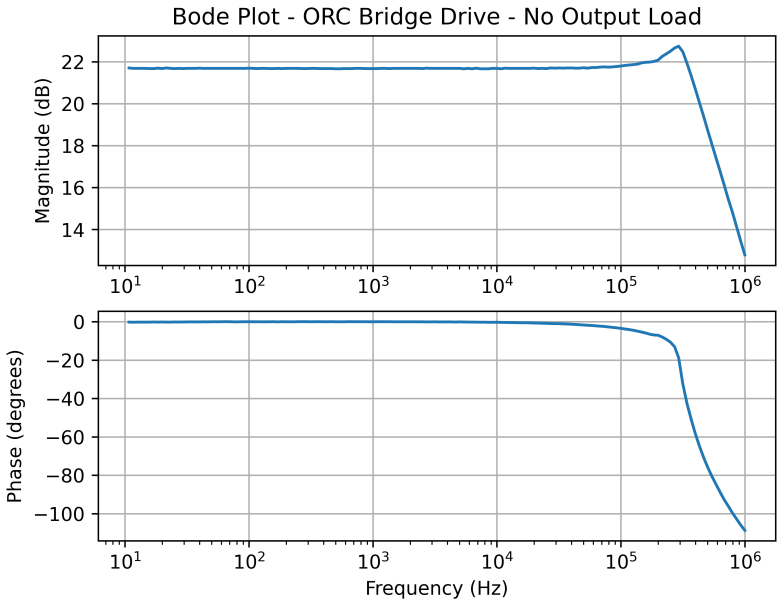


Figure 3: ORC motor amplifier stability analysis

4.7 Data Logging

In order to analyze system performance, the vehicle dynamics under PID control are logged to an SD card for later processing.

4.7.1 SD Bus and SD Cards

As described previously, the SD bus operates similar to SPI busses. The extra data lines allow significantly more information to be transmitted.

The clock speed the card can handle is directly related to the class of card. ORC uses a class 10 SD card capable of consistent operation at 45 MHz.

4.7.2 Limitations

Because the SD card performs wear-leveling operations whenever data is written to it, the log file can't be appended to at very high frequency. This frequency limit sits only slightly above a few hundred data points a second. To combat this, data is written in batches to the card. A buffer, or cache as it's called in the code, is used to temporarily hold 64kb of data points, working out to around 2 seconds of data at the fastest achieved sampling rate.

4.7.3 Data Processing

The collected data is processed with a single python script. Each log file can contain multiple runs of different types. In some cases these runs must be reordered by hand for correct processing. Each data set is passed through a median filter to remove any remaining noise or erroneous logs. Statistics for the pitch, roll and vertical acceleration data sets are calculated individually, statistics for the set of data set statistics are calculated and finally a select sample is plotted. More details can be found in Appendix B.

The input file and saved plot names are hard coded and must be changed in situ. The saved plot was randomly chosen to be the third one. most of the processing is accomplished with external libraries.

4.8 Power Supplies

ORC's upper and lower bounds were determined from the necessary overhead in the amplifier to drive $+16V$ to $-16V$ across the motor terminals. To make the system portable, batteries were the power source of choice. Several different supply voltages were necessary to drive the various parts of the board.

4.8.1 Battery Supply Rail

The required $\pm 16V$ was rounded up to the nearest LiPo battery size of six $3.7V$ cells in series (6s1p). These six cells were split into 2 groups, one for the positive rail, one for the negative rail, giving a center tap for the system ground. This gave nominal voltage of $+11.1V$, $0V$, and $-11.1V$. With the batteries fully charged, the ± 11.1V became $\pm 12.6V$. While it is possible to run the system with 2 cells per group, full motor performance may not be realized.

The battery input is over-current protected with $12A$ fuses on both the negative and positive rail. No other protection is provided.

Large traces feed this battery power directly to the motor amplifier slots and to the DC-DC converters for the other power rails.

4.8.2 $3.3V$ Rail

ORC has two $3.3V$ rails. One provides power to the high switching frequency devices: USB interface, SD card, and MCU. The other rail provides power to all the analog interfaces: DAC, comparator, and mux. This separation is intended to reduce noise influencing analog outputs.

Both these rails are powered from linear regulators that take $5V$ from either the USB power supply or a switching regulator that drops the battery voltage down. Both possible $5V$ sources are diode protected to allow them to coexist. This is the only rail to implement a switching regulator, as the current draw is rather significant when the MCU is running at full tilt. The AP63205 reference design is selected for this purpose.

4.8.3 $\pm 9V$ Rail

The comparator and mux, which would handle calibration, need a stable supply near the voltages they take in and put out. For this purpose, a linear regulator supplies both the $+9V$ and $-9V$ rails.

4.8.4 $\pm 5V$ Rail

Similar to the comparator, the DAC needs a supply near the voltages it can output. The DAC is configured to output $\pm 2.5V$ and the minimum allowable supply for this output range is $\pm 4.5V$. To reduce BOM count, pre-calibrated $\pm 5V$ linear regulators (oversized to ensure sufficient heat dissipation) are used.

4.9 Mechanical System

A major component of ORC is the ability to exert force on the suspension. Examining the multitude of ways to accomplish this task, gear motors were selected. These offered high torque at modest power requirements. The alternatives, voice coils, servos, hydraulics, had major disadvantages in one or more ways. The cost of voice coils made them out of reach for this iteration. The force required to back-drive the servo was so high as to restrict the suspension movement significantly. Hydraulics and pneumatics have such a long response time at this scale that they couldn't properly respond to stimulus. At larger scale, a cost effective solution would be hydraulics. The highest performance would be voice coils, but their power requirements are quite significant.

4.9.1 Motor Selection

Having measured the force required to compress the suspension, a target motor torque could be realized. To reduce back-drive torque, a larger motor at a smaller gear reduction was required.

Not many motors had actual test data for their torque vs. voltage. Pololu validates all their products with extensive test data. From this test data, the choices of motors to test

could be reduced to two. A 10:1 and 6.3:1 reduction were tested. Both had very low back-drive torque. because there was very little difference, the stronger 10:1 ratio was selected for ORC.

4.9.2 Lever Arms and Linkages

Having selected a motor, finding a balance between lever arm length and torque requirements was necessary. At the extremes of operation the motor torque is applied to the suspension at an angle, effectively reducing the transmitted force. This can be mitigated with a longer lever, at the cost maximum torque. The lever length that minimized the torque requirement was found to be 1.41 inches.

Knowing the required length, the lever could be designed. The lever attaches to the motor with a D shaped shaft and for extra security, a screw clamps the lever to the shaft. The linkages that extend down to the axle are are attached with an M3 screw and have pivots on both ends to allow the suspension to twist freely. Adjustable linkages have been selected so the lever can be positioned horizontally when the vehicle is at rest.

4.9.3 Custom Brackets

The motors, of course, needed to be mounted to the frame. Because the ultimate force these motors could apply was relatively low, a 3D printed bracket was appropriate. These brackets located the motors above the axle they were controlling, allowing the linkage to be fairly vertical, reducing the force lost to off-axis application.

4.9.4 Vehicle Details

For completeness, the vehicle used to test ORC is detailed here. While the control system ORC implements is platform agnostic, recreating these results verbatim would require similar kit.

A RC4WD 1/10th scale trail finder 2 kit was selected. This kit offered near realism with it's scale proportions and leaf spring suspension. The ladder frame design also allowed for the various parts of ORC to be mounted to the vehicle with ease.

To complete this kit, a Holmes Hobbies "Revolver" BLDC motor was driven from a Castle Creations "Mamba Micro" ESC. A 7.4V 2 cell lipo battery powers the drivetrain and steering. The steering is controlled with a cheap hobby servo.

While not absolutely necessary, a JConcepts Ford Aerostar body custom painted in a WSU livery (see figure 1) was implemented. This boxy body helps highlight the vehicle dynamics when demonstrating the system.

5 Testing

5.1 Tests

This section details the four test tracks and the drive characteristics they are examining. The results of these test are also discussed. Each test was performed 20 times with ORC enabled and 20 times without unless otherwise noted.

5.1.1 Camber Test

The camber test is a simple test that slowly increases the angle of the terrain perpendicular to the vehicle's direction of travel. This test uses the wedge-shaped terrain found in figure 4. The vehicle is driven along the length of the terrain, keeping the front tires as high on the terrain as possible. This test was performed with about 25% throttle in low gear. This throttle value was programmed into the RC transmitter and all the operator had to do was hold the throttle trigger at it's max position. This was a common technique used across the test to maintain consistent driving speeds between tests.

This test's only focus was to examine the ORC platform's ability to reduce the maximum roll angle experienced by the body of the vehicle. The pitch of the vehicle can be observed as well, but is not the main focus of the test.

5.1.2 Terrain Test

The terrain test is a controlled off-road terrain featuring multiple obstacles of various sizes with different angles of approach. As revealed in figure 5, these obstacles are a series of wooden dowels placed at sporadic angles to the direction of travel. The raised base was just to give the screws somewhere to go. This terrain isn't the most representative of the rocky and dusty off road trails found here in Utah, but the small and portable nature of this terrain let it be used indoors. This meant the terrain would stay consistent and not be as influenced by operator skill.

This test was performed at 35% throttle, set as described in section 4.

This test focuses on examining the dynamic pitch and roll control of ORC as well as the ability to act as shock absorbers. This test is the most comprehensive, examining all 3 vehicle dynamics collected on-board as well as the derived vertical jerk.

5.1.3 Vibration Test

The vibration test examines the ability of ORC to act as a shock absorber. The test course, consisting of a series of washboard-like boards set both in and out of phase between

Figure 4: Camber Test Track

the driver and passenger side (figure 6), was performed at two speeds, once at 25% throttle and again at 75% throttle. At the higher speed only 5 test were performed. The narrow nature of the track made it difficult to keep the vehicle on course and engaged with the terrain across the entire run.

Only testing shock absorption, the vertical acceleration and jerk are the vehicle dynamics under review in this test.

5.1.4 Corner Test

The corner test varied wildly from the other three. This test examined if ORC improved the vehicles ability to take corners at high speed. The plan for this test was to hold a consistent steering input and slowly increase the speed of the vehicle. Done in the transmission's high gear, a relatively high speed could be realized. This test ends when the vehicle either maxes out it's speed, or experiences an unplanned inversion (aka: rolls over).

The vehicle speed is controlled by a ESP32_Devkitc_V4 and linearly ramps the PWM

Figure 5: Terrain Test Track

signal sent to the vehicle Electronic Speed Controller (ESC). The Devkit is powered on and started from the same PWM switch described in section 5.2. This allows the test recording to be synchronized with the vehicle starting and stopping.

This test examines both the pitch and roll of the vehicle, as the vehicle will be experiencing a strong centrifugal force as well as a constant acceleration forward (tangent to the circle).

5.2 Vehicle Modifications

Some modifications to the test platform were made to facilitate testing. This includes a remote start/stop switch and plastic wheels. Also of note was the addition of relays to connect and disconnect the battery.

The remote start/stop switch was a PWM controlled $5V$ switch. This was an off the shelf part meant for controlling lights on an RC vehicle. To work with ORC, the output of this switch was put through a resistor divider to drop it down to approximately $3.3V$ before taking place of the switch that originally started and stopped the test. This allowed the test

Figure 6: Vibration Test Track

operator to start a test and stop a test without touching the vehicle. This prevented the touch from influencing the data collected. This also facilitated the test to be done without walking along with the vehicle, which improved the test operator's ability to drive straight and be consistent from run to run.

The plastic wheels were used on half the wheels during the corner test. These were positioned on the side of the vehicle that the turn was point in (i.e.: turning passenger = plastic wheels on passenger side). The test vehicle was four wheel drive with locked differentials, which led to a lot of binding and jumping when trying to do corners on grippy surfaces. It was desirable to do the corner test on concrete to minimize surface defects that could destabilize the vehicle. This surface had a lot more grip than soft dirt, but could be swept clean to remove gravel and other debris easily.

The added relays in the battery supply path allowed both power supplies to be connected at the same time. This helped the test operator not have to do a specific power up sequence every time the batteries needed to be connected. This relay was triggered off the test vehicles electrical system, meaning when the test vehicle was shut down, the batteries for the ORC

main board were disconnected.

5.3 Results

For each test (described in section 5.1), the collected data was hand sorted into two groups, actuators enabled or actuators disabled. For each test run, the RMS, minimum, and maximum value were recorded. All the test runs' data was compiled, and a two sample one sided t-test was used to show statistically significant improvement within a 5% confidence interval (CI). This test does assume the data follows a normal distribution. It is important to note that the t-test can only show if there is improvement in the parameter under test, but will not determine if performance was worse. For clarity, only a single run's data is plotted.

5.3.1 Camber Test

The camber test showed significant improvement in the essentially static performance of ORC. The maximum pitch and roll was not reduced significantly, but that is to be expected with the test only forcing the vehicle in the minimum direction. Figure 7 and 8 show the results.

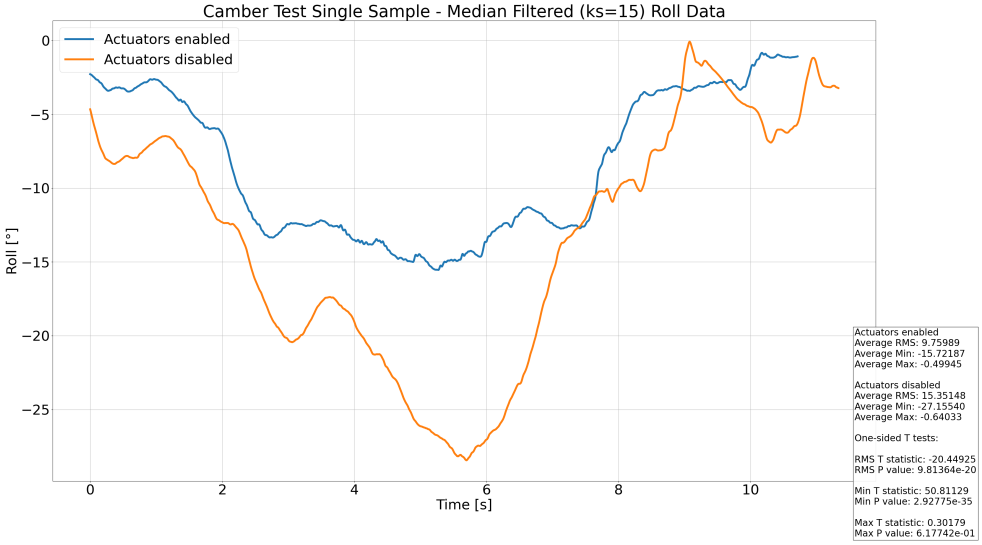


Figure 7: Roll plot with statistics - camber test

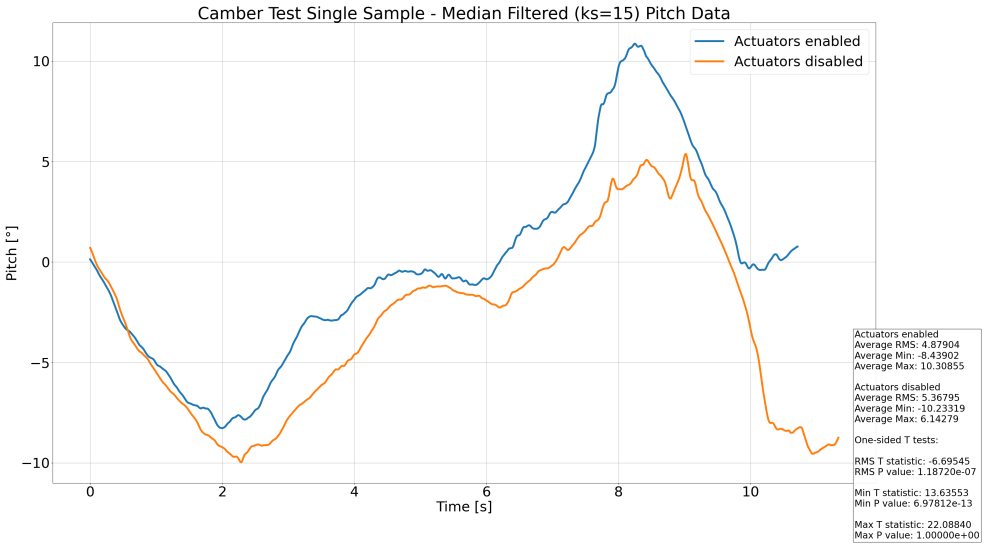


Figure 8: Pitch plot with statistics - camber test

5.3.2 Terrain Test

The terrain test was more of a mixed bag. The pitch and roll results showed good performance across all metrics. Acceleration and Jerk had more mixed results. The plots help tell this story. Figure 9 and 10 visibly capture the reduction in pitch and roll. Figure 11 and 12 reveal no distinct difference in the results, other than the acceleration having less total oscillations with a bit lower magnitude, which is in line with the significant reduction in RMS.

5.3.3 Vibration Test

At both speeds, the vibration test showed no improvement in reducing vertical jerk. That's not to say it performed worse, but there is no significant improvement(see figures 13 and 14). The acceleration (figures 15 and 16) was better almost across the board, except the minimum parameter in the low speed test. This may be an anomaly, or perhaps the PID over-reacted at this oscillation frequency.

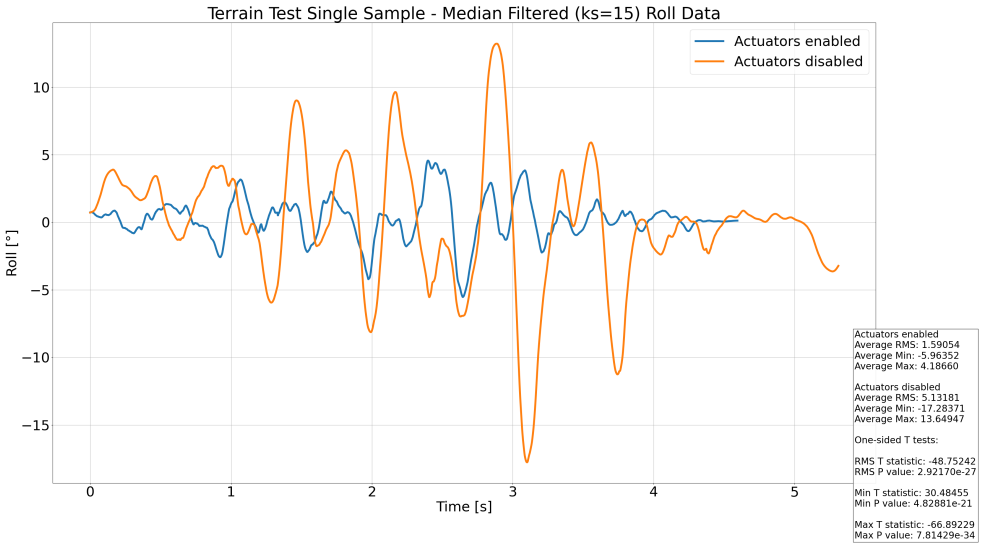


Figure 9: Roll plot with statistics - terrain test

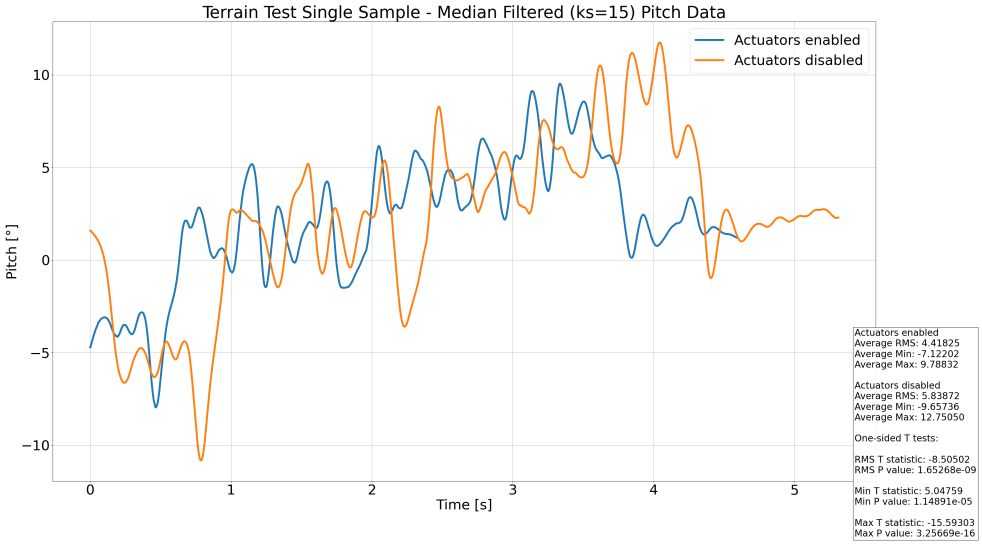


Figure 10: Pitch plot with statistics - terrain test

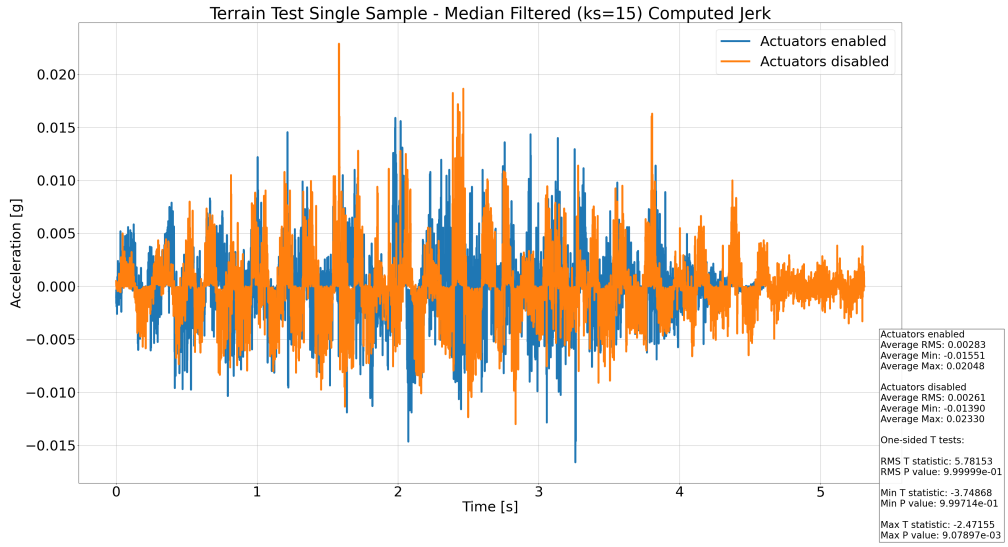


Figure 11: Jerk plot with statistics - terrain test

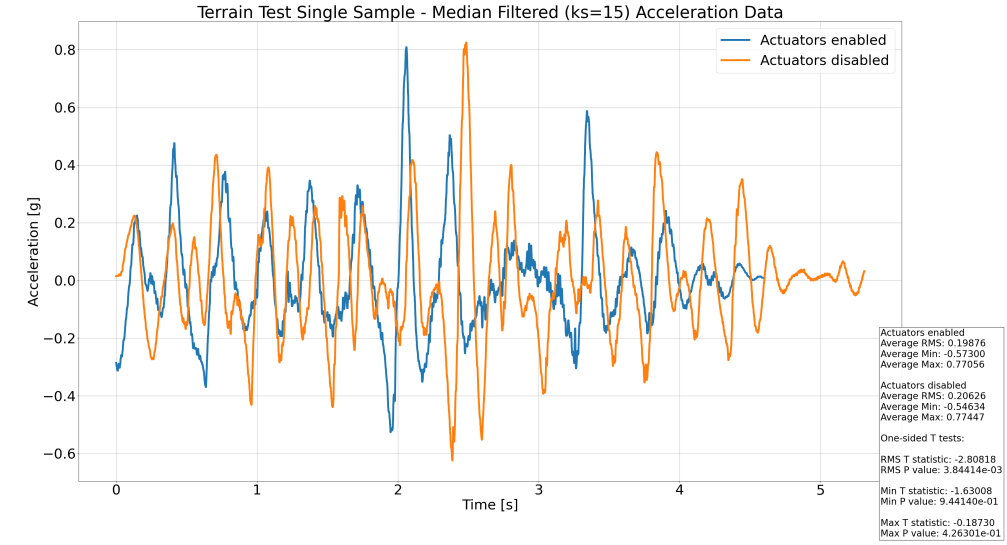


Figure 12: Vertical Acceleration plot with statistics - terrain test

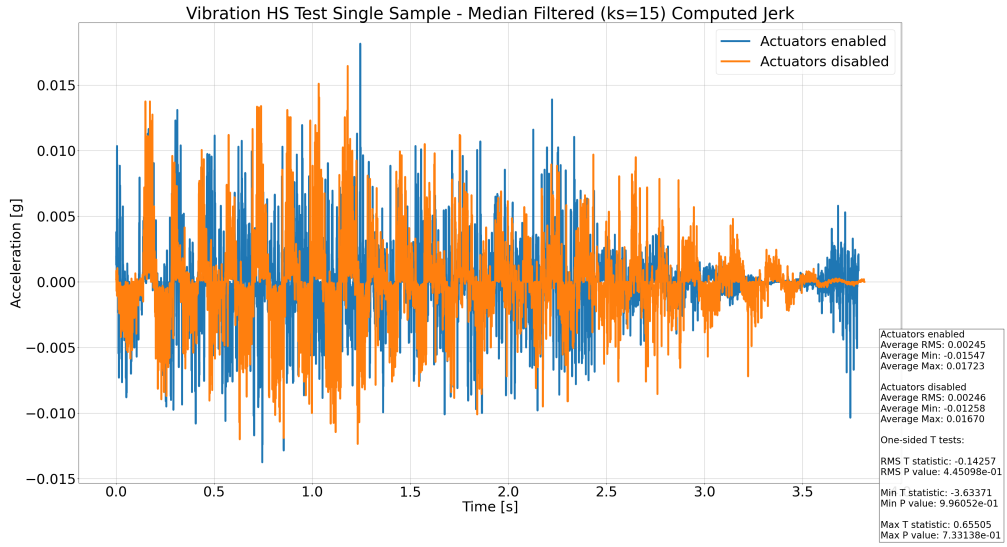


Figure 13: Jerk plot with statistics - high speed vibration test

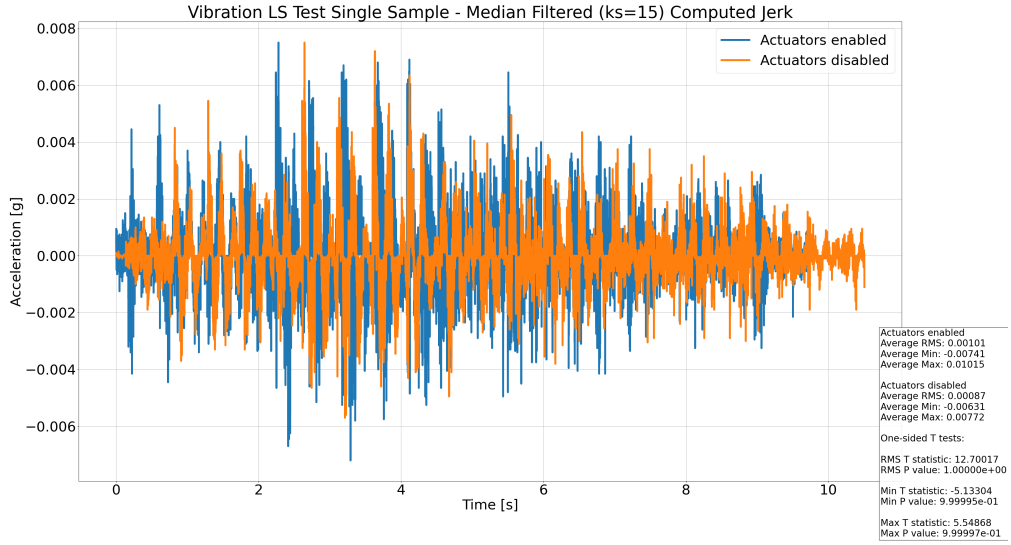


Figure 14: Jerk plot with statistics - low speed vibration test

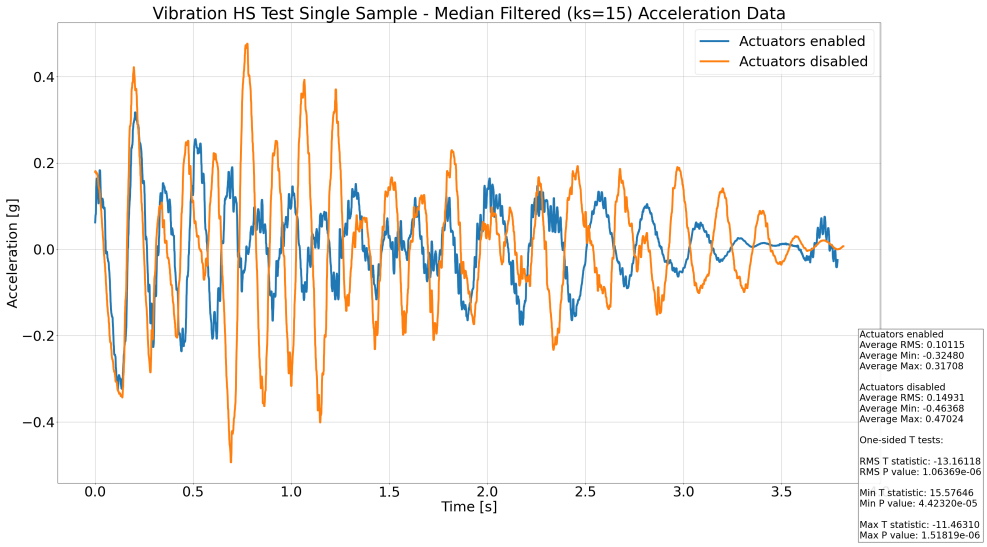


Figure 15: Acceleration plot with statistics - high speed vibration test

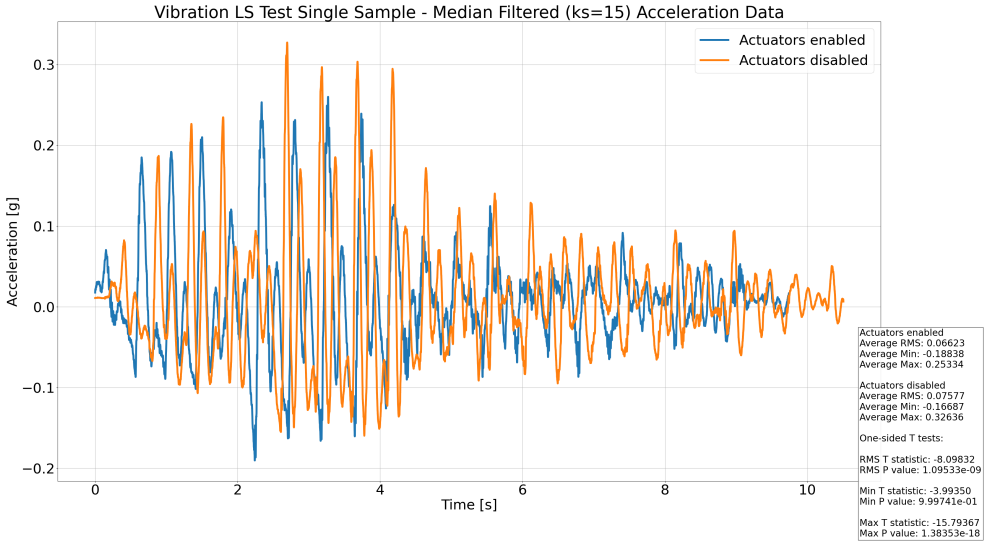


Figure 16: Acceleration plot with statistics - low speed vibration test

5.3.4 Corner Test

The corner test are largely unreliable. The plotted response does not resemble what the vehicle was actually doing. In both figure 17 and 18, the actuator enabled test appeared to experience huge plateaus in the direction that would cause the vehicle to flip. In reality the vehicle was leaning in to the turn, with a positive roll angle. It's likely the sensor fusion algorithm broke down under the presence of a large centrifugal force. This force caused the algorithm to misinterpret where gravity was and thus what angle the vehicle was at relative to earth. This caused the wildly incorrect measured data as well as the leaning behavior.

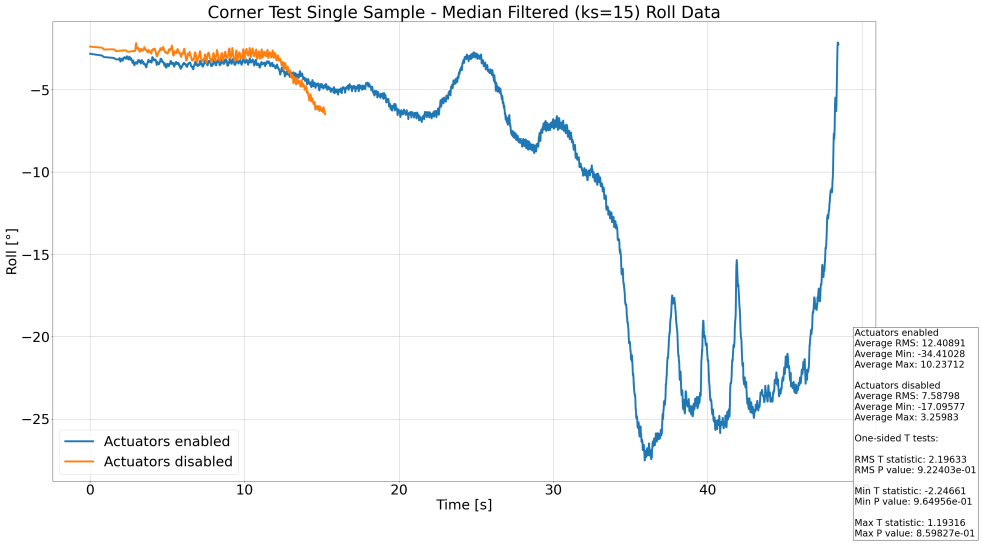


Figure 17: Roll plot with statistics - corner test

The fortunate reality of this misinterpretation is the vehicle was more stable, completing the test on many occasions. Seeing this behavior in the first test, it was decided that the number of revolutions should be counted to a quarter turn accuracy. This revealed a significant improvement in sustained corner performance, but it is not certain if this behavior would translate to a sudden cornering event.

5.3.5 Summary

Table 3 summarizes the t-test results across all the tests and all vehicle dynamics. Table 4 summarizes the means of each test parameter for both the actuators enabled and disabled. This table also displays the results of counting the completed revolutions in the corner test.

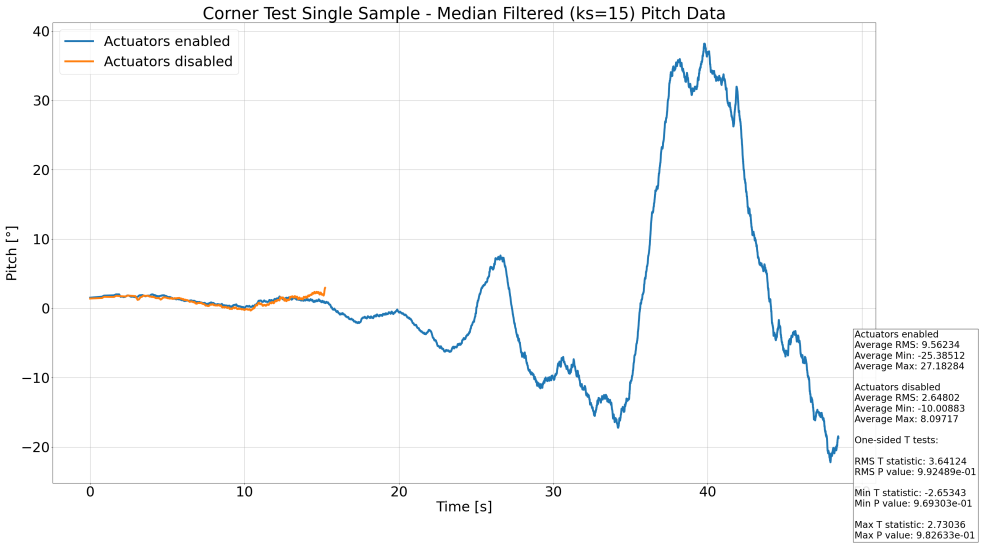


Figure 18: Pitch plot with statistics - corner test

6 Conclusion

ORC is successful in it's main goal of controlling the roll of the equipped vehicle. The tests performed saw a statistically significant reduction in the maximum, minimum, and average roll experienced by the vehicle. Much the same can be said for the pitch control. The damping ability, and thus the comfort aspect of the design was somewhat lacking. In most test, no significant improvement in reducing jerk was realized. The acceleration results were mixed as well, but over all showed some significant improvement.

It should be noted the acceleration PID controller was the most difficult to tune. There is likely some improvement that could be realized just by re-tuning the PID controller. It is probable the other two PID controllers could also be better tuned for faster responses with less or no overshoot. If more performance is needed, other control methods as described in "Modern Semi-Active Control Schemes for a Suspension with MR Actuator for Vibration Attenuation"[2] or "Quantized Feedback Control of Active Suspension Systems Based on Event Trigger"[5] could be implemented to examine their efficacy.

Speaking to performance, many hardware and software improvements could be realized. A cheaper, faster DAC could be implemented. Sacrificing some resolution for an increase in speed could allow the max sampling rate of the IMU being realized. The calibration system could be reworked as previously described. The SPI communications could have been better

One Sided T Test Statistics For Improved (Absolutely Less) RMS, Minimum, and Maximum Values					
Parameter Test	Camber (min direction)	Terrain	Vibration High Speed*	Vibration Low Speed	Corner*
Acceleration RMS	6.27E-20	3.84E-03	1.06E-06	1.10E-09	1.00E+00
Pitch RMS	1.19E-07	1.65E-09	1.65E-01	3.10E-33	9.92E-01
Roll RMS	9.81E-20	2.92E-27	1.54E-02	1.02E-24	9.22E-01
Jerk RMS	1.00E+00	1.00E+00	4.45E-01	1.00E+00	9.89E-01
Acceleration MIN	7.92E-14	9.44E-01	4.42E-05	1.00E+00	7.28E-01
Pitch MIN	6.98E-13	1.15E-05	1.78E-01	4.59E-01	9.69E-01
Roll MIN	2.93E-35	4.83E-21	3.16E-04	5.79E-26	9.65E-01
Jerk MIN	1.00E+00	1.00E+00	9.96E-01	1.00E+00	8.33E-01
Acceleration MAX	9.97E-01	4.26E-01	1.52E-06	1.38E-18	9.89E-01
Pitch MAX	1.00E+00	3.26E-16	9.15E-03	2.53E-27	9.83E-01
Roll MAX	6.18E-01	7.81E-34	3.91E-04	2.40E-12	8.60E-01
Jerk MAX	1.00E+00	9.08E-03	7.33E-01	1.00E+00	8.30E-01
*Only tested 5 samples instead of full 20+					
(Gray fill cells not relevant to particular test, included for completeness)				>5% CI	<5% CI

Table 3: Summary of t-test results

Average RMS, Minimum, and Maximum values. Actuators Enabled | Actuators Disabled												
Parameter Test	Camber (min direction)		Terrain		Vibration High Speed*		Vibration Low Speed		Corner*		Corner Completed Revolutions*	
Acceleration RMS	0.018	0.053	0.199	0.206	0.101	0.149	0.066	0.076	0.139	0.030	Average	
Pitch RMS	4.879	5.368	4.418	5.839	2.501	2.658	2.104	2.498	9.562	2.648	30.850	5.500
Roll RMS	9.760	15.351	1.591	5.132	1.082	1.312	0.813	1.995	12.409	7.588		
Jerk RMS	0.00034	0.00028	0.00283	0.00261	0.00245	0.00246	0.00101	0.00870	0.00302	0.00128		
Acceleration MIN	-0.079	-0.125	-0.573	-0.546	-0.325	-0.464	-0.188	-0.167	-0.249	-0.136	Minimum	
Pitch MIN	-8.439	-10.233	-7.122	-9.657	-1.945	-2.676	-3.946	-3.958	-25.385	-10.009	15.250	4.750
Roll MIN	-15.722	-27.155	-5.964	-17.284	-2.314	-4.105	-2.853	-5.830	-34.410	-17.096		
Jerk MIN	-0.00367	-0.00260	-0.01551	-0.01390	-0.01547	-0.01258	-0.00741	-0.00631	-0.17662	-0.01615		
Acceleration MAX	0.090	0.073	0.771	0.774	0.317	0.470	0.253	0.326	0.523	0.139	Maximum	
Pitch MAX	10.309	6.143	9.788	12.751	5.592	6.246	3.754	6.030	27.183	8.097	55.000	6.000
Roll MAX	-0.499	-0.640	4.187	13.649	0.339	1.927	1.891	4.248	10.237	3.260		
Jerk MAX	0.00523	0.00290	0.02048	0.02330	0.01723	0.01670	0.01015	0.00772	1.77660	0.01360		
*Only tested 5 samples instead of full 20+										CC t-test statistic:		0.01659

Table 4: Averaged test run parameters, raw comparison

managed. Having these be done in the background would free up processor time to computed the fusion algorithm and PID controller updates. These could also allow the maximum IMU sampling rate to be realized.

ORC is a huge success, even with so much performance left on the table. Future work will hopefully see this performance utilized.

References

[1] Espressif. Esp-idf programming guide. https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/index.html.

[2] Kevin Herubiel Floreán-Aquino, Manuel Arias-Montiel, Jesús Linares-Flores, José Gabriel Mendoza-Larios, and Álvaro Cabrera-Amado. Modern semi-active control schemes for a suspension with mr actuator for vibration attenuation. *Actuators*, 10(2), 2021.

[3] Corbin Roennebeck. Orc. https://github.com/CJJeepster/ORC, 2025.

[4] STMicroelectronics. ism330dhcx-pid. https://github.com/STMicroelectronics/ism330dhcx-pid, 2021.

[5] Jinwei Sun, Jingyu Cong, Weihua Zhao, and Yonghui Zhang. Quantized feedback control of active suspension systems based on event trigger. *Shock and Vibration*, 2021(1):8886069, 2021.

[6] Di Tan, Chao Lu, and Xueyi Zhang. Dual-loop PID control with PSO algorithm for the active suspension of the electric vehicle driven by in-wheel motor. *Journal of Vibroengineering*, 18(6):3915–3929, sep 2016.

[7] xioTechnologies. Fusion. https://github.com/xioTechnologies/Fusion/tree/maind, 2024.

Appendix A

On the following pages, the ORC main board and motor amplifier schematics, as well as their associated PCB layouts, are included.

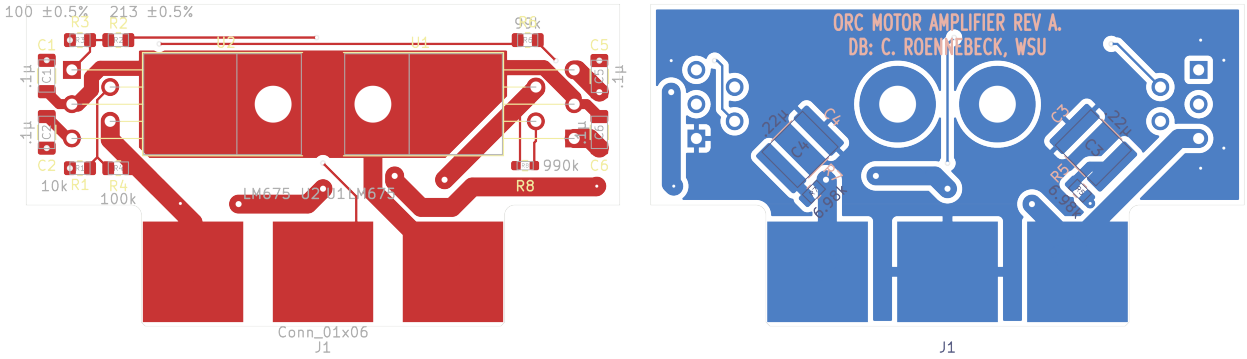


Figure 19: Motor Amplifier PCB Front and Back View

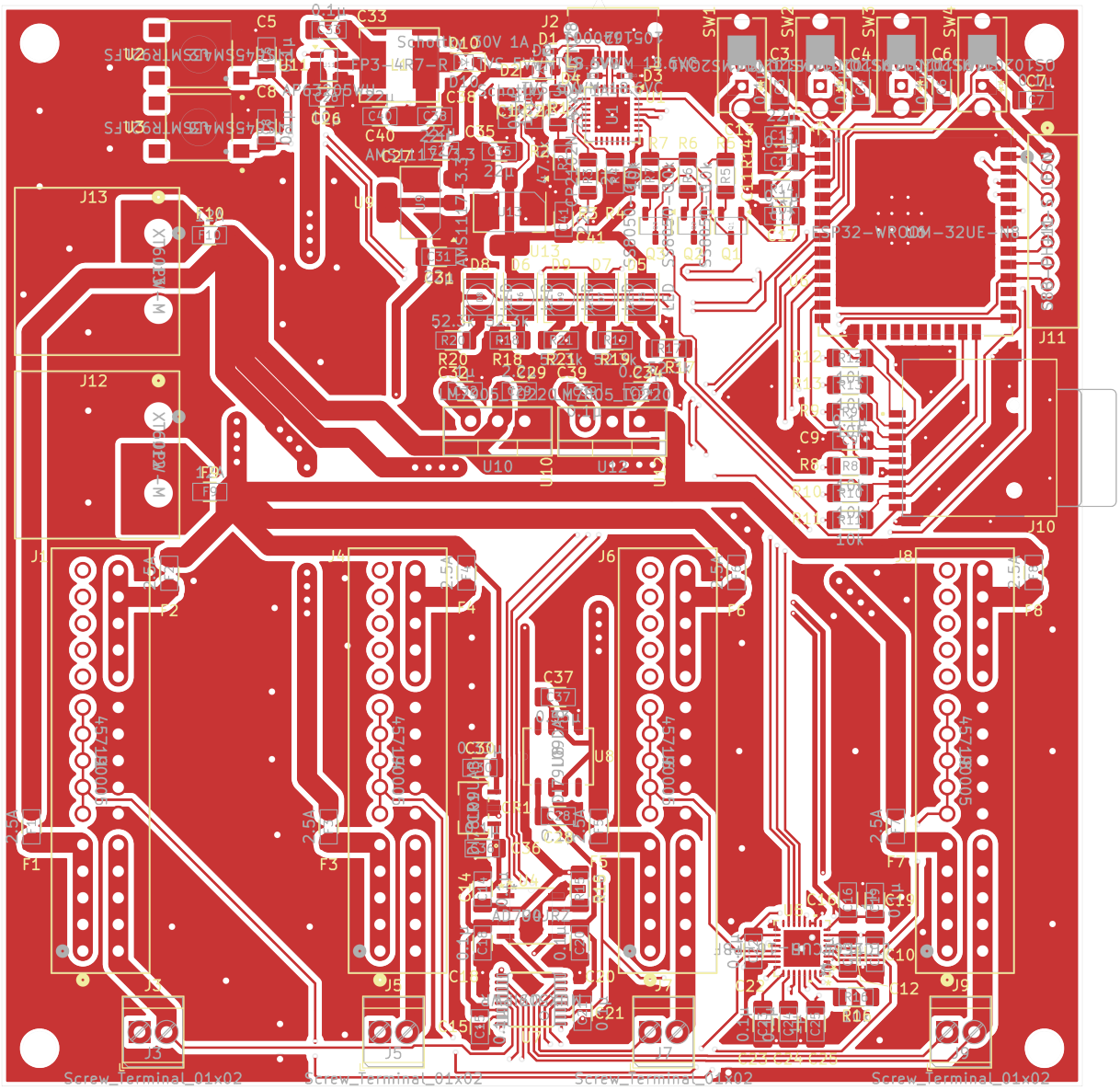


Figure 20: Main Board PCB Front View

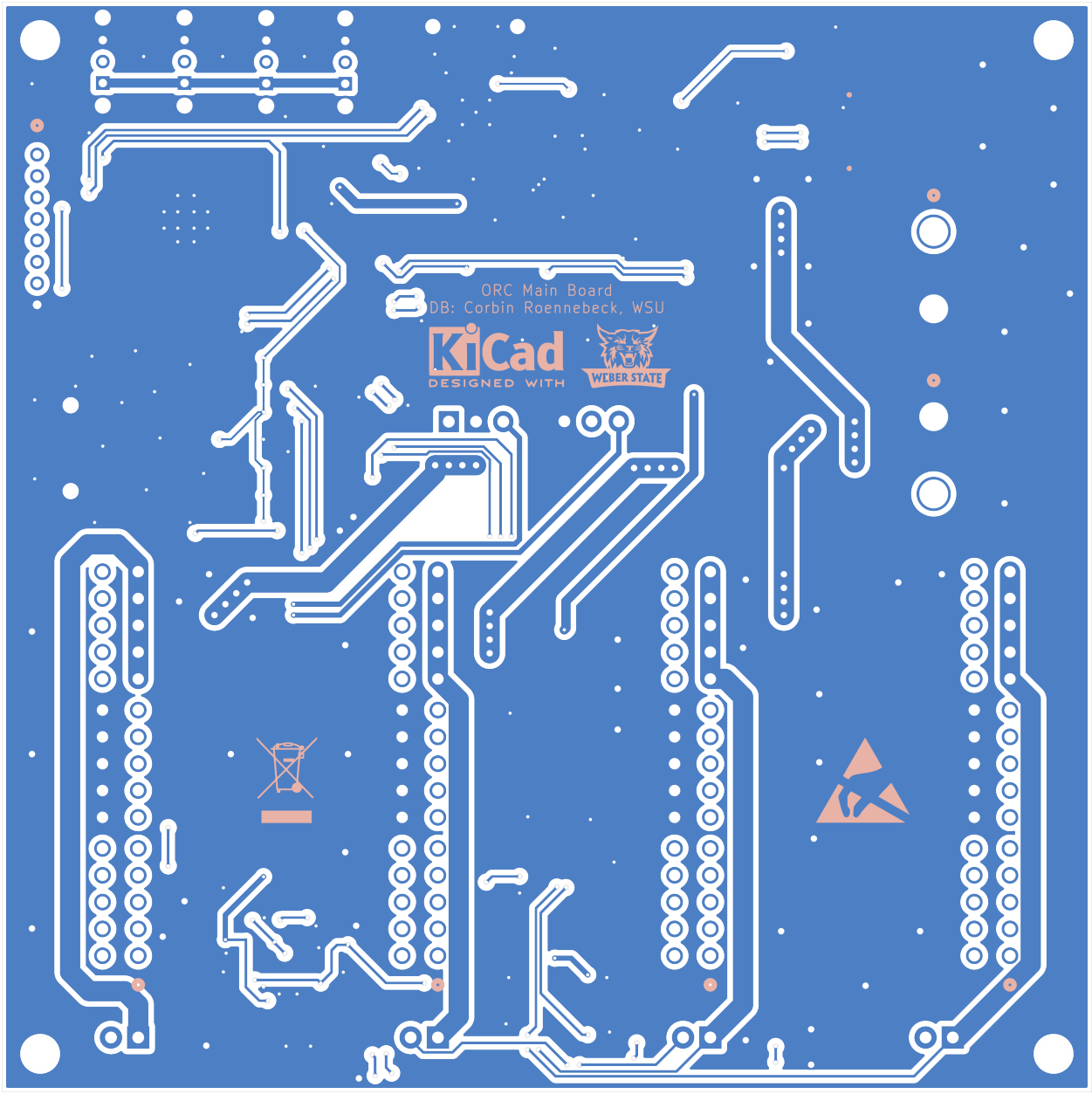
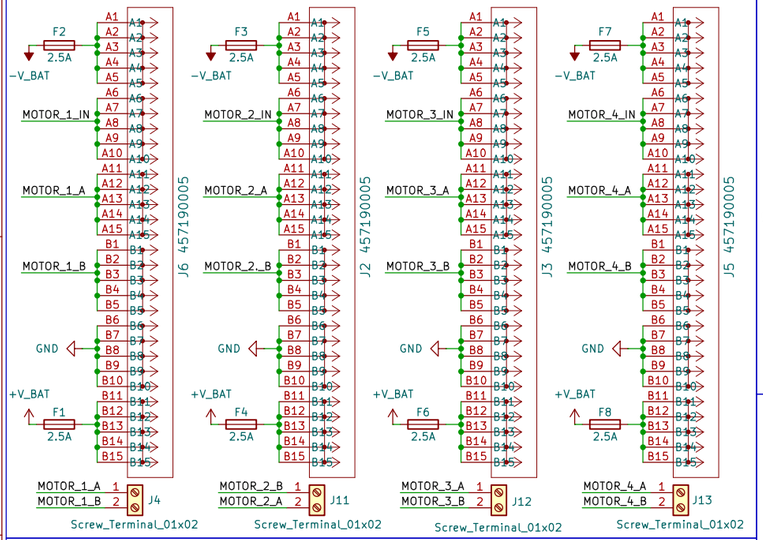
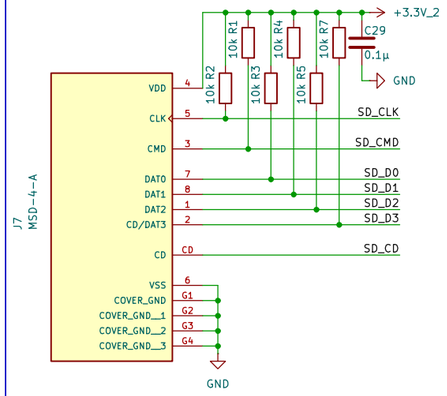


Figure 21: Main Board PCB Back View

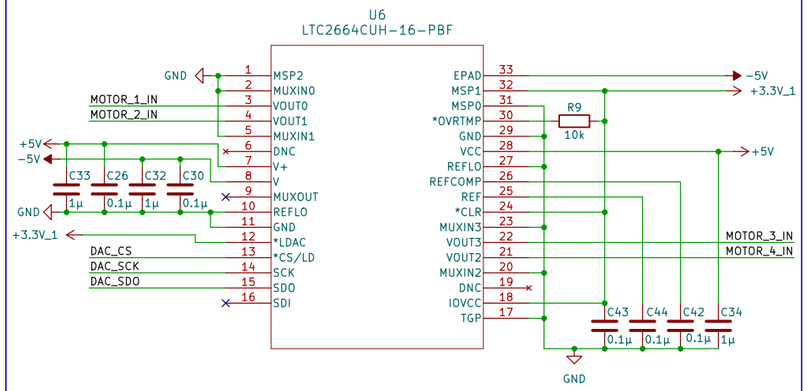
Motor Driver Card Slots And Motor terminals



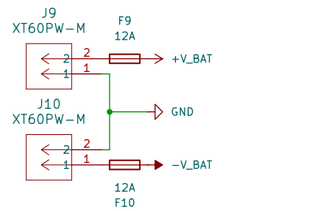
SD Card Slot



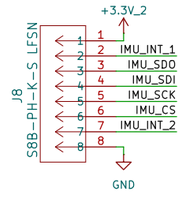
DAC – Motor Command



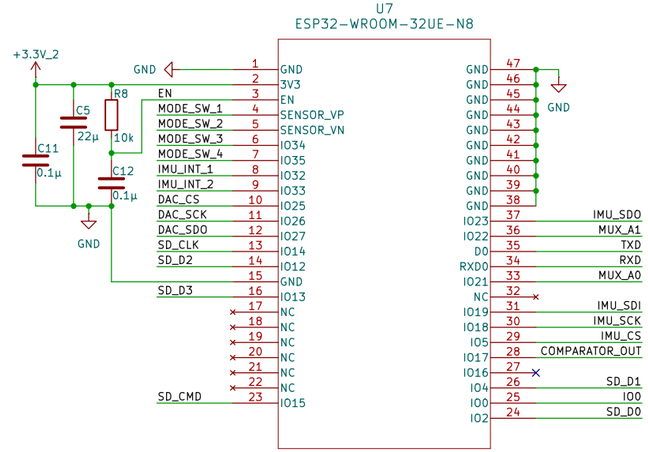
Battery In



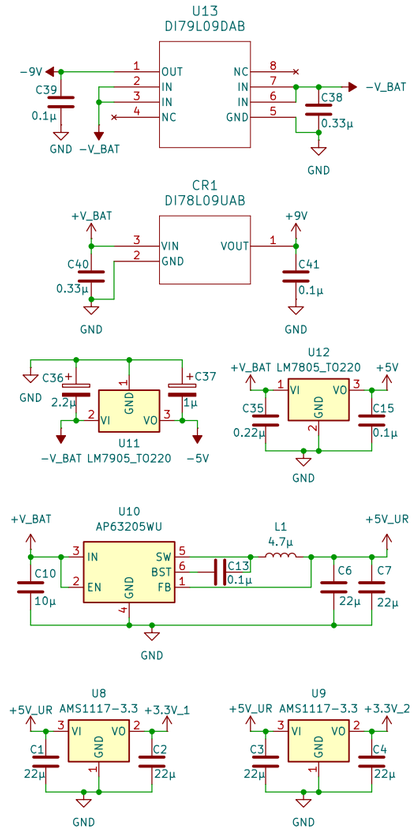
IMU Remote Daughterboard Header



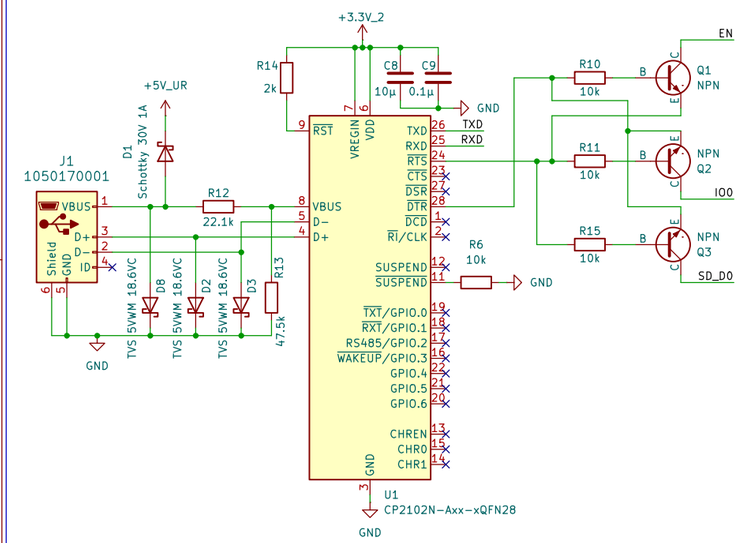
MCU



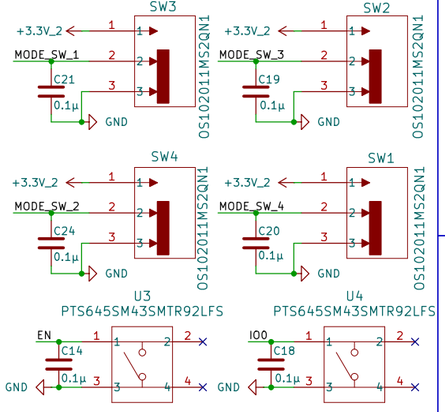
Voltage Step-Down/Regulation



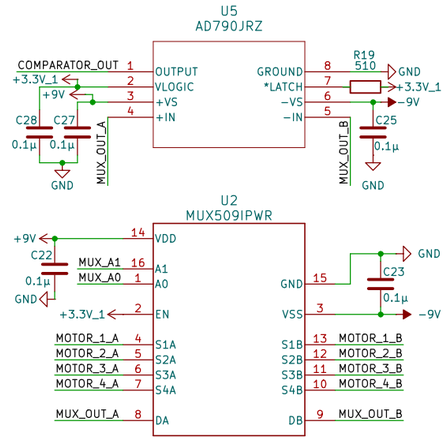
USB Interface And Bootstraping



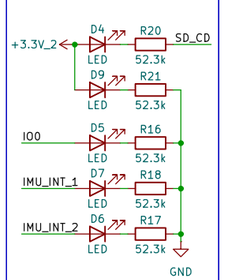
UI: Mode/Reset/Bootstrap



Amplifier Calibration Measuring



Indicator LEDs

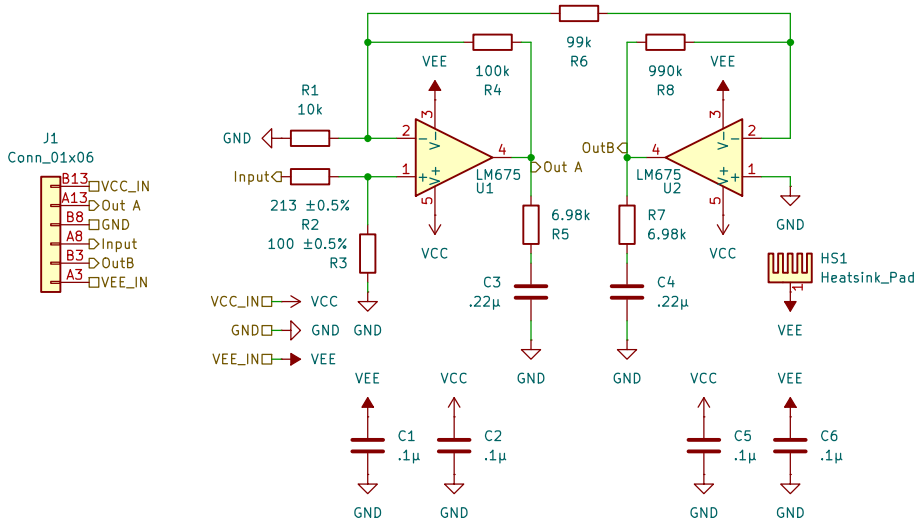


NOTES:
1. All resistors measured in Ohms, tolerance of 5% or better unless noted otherwise.
2. All capacitors measured in Farads, tolerance of 10% or better unless noted otherwise.
3. All inductors measured in Henrys, tolerance of 15% or better unless noted otherwise.
4. Amplifier slots must be spaced minimum 1" apart.
5. Batteries attached to board must have external low voltage protection.
6. Fuses to be rated for current indicated and be fast blow type unless noted otherwise.
7. IMU mounted remotely for better positioning. PH08 pigtail required.

Project: ORC
Designer: Corbin Roennebeck
Weber State University
Sheet: /
File: Main Board.kicad_sch

Title: ORC Main Board

Size: B	Date: 2025-01-22	**Rev: IR**
KiCad E.D.A. 8.0.5		Id: 1/1



NOTES:
1. All resistors measured in Ohms, tolerance of 5% or better unless noted otherwise.
2. All capacitors measured in Farads, tolerance of 10% or better unless noted otherwise.
3. Heatsink to be attached to both amplifiers. 6-32 hardware required.

Project: ORC
Designer: Corbin Roennebeck
Weber State University
Sheet: /
File: Servo Amplifier Rev A.kicad_sch

Title: Motor Amplifier

Size: A	Date: 2025-01-21	**Rev: A**
KiCad E.D.A. 8.0.5		Id: 1/1

Appendix B

This appendix contains all the code custom written for ORC. A GitHub repository[3] contains downloadable versions of these files. The libraries for sensor fusion and the ISM330DHCX can be found at their respective GitHub repositories[7].[4]

ORC_CONFIG.H

```
/**
  ******************************************************************************
  * @file    ORC_CONFIG.h
  * @author  Corbin Roennebeck, WSU
  * @brief   Constants and Configurations for ORC_Main
  ******************************************************************************
  */

#ifndef ORC_CONFIG_H
#define ORC_CONFIG_H

/* Calibration System IO / pin masks */
#define MUX_PIN_A0       33
#define MUX_PIN_A1       32
#define COMPARATOR_PIN   21
#define GPIO_MUX_PINS    ((1ULL<<MUX_PIN_A0) | (1ULL<<MUX_PIN_A1))
#define GPIO_COMP_PINS   (1ULL<<COMPARATOR_PIN)
/* SD Card IO / constants */
#define CACHE_SIZE       8192*8    //bytes
#define MOUNT_POINT      "/sdcard"
#define SD_PIN_D3        13
#define SD_PIN_D2        12
#define SD_PIN_CLK       14
#define SD_PIN_CMD       15
#define SD_PIN_D0        2
#define SD_PIN_D1        4
/* IMU IO / pin masks / spi controller */
#define IMU_HOST              VSPI_HOST
#define IMU_BOOT_TIME         10 //ms
#define IMU_PIN_NUM_MISO      23
#define IMU_PIN_NUM_MOSI      19
#define IMU_PIN_NUM_CLK       18
#define IMU_PIN_NUM_CS        5
#define IMU_PIN_INT1          22
#define GPIO_INT_PIN_SEL      (1ULL<<IMU_PIN_INT1)
#define GPIO_MISO_PIN         (1ULL<<IMU_PIN_NUM_MISO)
/* DAC IO / spi controller */
#define DAC_HOST              HSPI_HOST
#define DAC_PIN_NUM_MISO      -1
#define DAC_PIN_NUM_MOSI      27
#define DAC_PIN_NUM_CLK       26
#define DAC_PIN_NUM_CS        25
/* UI IO / pin masks */
#define STATUS_LED1           17
#define STATUS_LED2           0
#define GPIO_STATUS_PINS      ((1ULL<<STATUS_LED2) | (1ULL<<STATUS_LED1))
#define MODE_SW_EN_LOGGING    34  //enable logging
#define MODE_SW_EN_ACTUATOR   35  //enable actuators
#define MODE_SW_START_LOG     36  //start and stop log via this switch
#define MODE_SW_APPEND_LOG    39  //choose to start new log (same file) or append to previous after pause
#define GPIO_MODE_SWS_HIGH    ((1ULL<<MODE_SW_EN_ACTUATOR) | (1ULL<<MODE_SW_EN_LOGGING) | (1ULL<<
    MODE_SW_APPEND_LOG))
#define GPIO_MODE_SWS_LOW     (1ULL<<MODE_SW_START_LOG)
/* nonlinear filter constants */
#define IMU_XL_PEAK_REJ       30738 //eq to 3.75g in int16 output data format, reject xl values above this
    limit (below for negative)
#define IMU_GY_PEAK_REJ       28572 //eq to 500dps in int16 output data format, reject gy values above this
    limit (below for negative)
#define IMU_NOISE_FLOOR       0.003f //crush all xl data +- this value to zero
/* distances */
#define FA_COG       0.011     //distance to front axle linkages (in meters) from COG
```

```
59 #define RA_COG       0.018     //distance to rear axle linkages (in meters) from COG
60 #define DA_COG       0.008     //distance to drive side linkages (in meters) from COG
61 #define PA_COG       0.008     //distance to passenger side linkages (in meters) from COG
62 /* PID constants */
63 #define Z_XL_KP      -30000.0f
64 #define Z_XL_KI      -0.0f     //integral can be unstable, without providing much to response, possible
       value: -8000
65 #define Z_XL_KD      -1800.0f
66 #define PITCH_KP     -70.0f
67 #define PITCH_KI     -1.50f
68 #define PITCH_KD     -1.0f
69 #define ROLL_KP      -50.0f
70 #define ROLL_KI      -1.50f
71 #define ROLL_KD      -1.0f
72
73
74 #endif //ORC_CONFIG_H
75
76 //eof
```

ORC_Main.c

```
1  /**
2    ******************************************************************************
3    * @file     ORC_Main.c
4    * @author   Corbin Roennebeck, WSU
5    * @brief    Controls actuators and logs data for ORC project as defined by
6    *               "Theory of Operation", below.
7    ******************************************************************************
8    */
9
10   /**
11    *     Theory of Operation:
12    *         The motions of 4 actuators between the vehicle frame and axle,
13    *         located both front and rear on both the driver and passenger side,
14    *         is controlled by 3 PID controllers. These controlers use measured
15    *         vehicle dynamics - vertical acceleration (less gravity), pitch, and
16    *         roll - to define actuator motion that cancels out the aforementioned
17    *         vehicle dynamics. The measured vehicle dynamics are also capable of
18    *         being logged to the onboard SD card. These vehicle dynamics are
19    *         measured by the attached IMU, filtered onboard the IMU, and processed
20    *         by the AHRS Fusion library to provide accurate, realtime information.
21    *
22    *     More detailed information can be found in the "ORC Documentation.pdf" file
23    */
24
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <string.h>
28 #include <sys/unistd.h>
29 #include <sys/stat.h>
30 #include "esp_vfs_fat.h"
31 #include "sdmmc_cmd.h"
32 #include "driver/sdmmc_host.h"
33 #include "freertos/FreeRTOS.h"
34 #include "freertos/task.h"
35 #include "driver/spi_master.h"
36 #include "driver/spi_common.h"
37 #include "driver/gpio.h"
38 #include "sdkconfig.h"
39 #include "esp_log.h"
40 #include "esp_timer.h"
41 #include "esp_intr_types.h"
42 #include "esp_random.h"
43
44 #include "Fusion.h"
45 #include "LTC2664_reg.h"
46 #include "ism330dhcx_reg.h"
47 #include "Transform.h"
48 #include "PID.h"
49
50 #include "ORC_CONFIG.h"
```

```
51
52 //spi functions for devices, platform delay for advanced IMU functions
53 static int32_t DAC_write(void *handle, ltc2664_DACS_t dac, uint8_t command, const uint16_t *data);
54 static int32_t IMU_write(void *handle, uint8_t reg, const uint8_t *bufp, uint16_t len);
55 static int32_t IMU_read(void *handle, uint8_t reg, uint8_t *bufp, uint16_t len);
56 static void platform_delay(uint32_t ms);
57
58 //initialization and configuration functions
59 static esp_err_t init_spi_IMU();
60 static esp_err_t init_spi_DAC();
61 static void init_pid_controllers(void);
62 static esp_err_t config_IMU(void);
63 static esp_err_t config_DAC(void);
64
65 //data structure to pass data from IMU Sampling to SD writing
66 typedef struct
67 {
68     float_t acceleration_z;
69     float_t pitch;
70     float_t roll;
71
72 }log_data_t;
73 static QueueHandle_t log_data_queue = NULL;
74
75 //spi handle and device reference for IMU
76 static stmdev_ctx_t ISM330DHCX_dev_ctx;
77 static spi_device_handle_t IMU_spi;
78
79 //spi handle and device reference for DAC
80 static ltcdev_ctx_t LTC2664_dev_ctx;
81 static spi_device_handle_t DAC_spi;
82
83 //reference for IMU_sample_task
84 static TaskHandle_t IMU_TASK;
85
86 //time between IMU samples, reported by IMU
87 static float_t float_time = 0;
88
89 //PID Contoller Handles
90 static PIDController_t accel_z_controller={};
91 static PIDController_t pitch_controller={};
92 static PIDController_t roll_controller={};
93
94 //Boot Config booleans
95 static bool en_logging = false;
96 static bool en_actuators = false;
97
98
99 /**
100  * @brief Capture interupt from IMU and tell the IMU_sample_task there's data avalible
101  */
102 static void IRAM_ATTR imu_isr_handler(void* arg){
103     BaseType_t higherPriorityTask = pdTRUE;
104     vTaskNotifyGiveFromISR(IMU_TASK,&higherPriorityTask);
105 }
106
107 /**
108  * @brief Initialize the IMU, it's interupts, and the fusion library, then perform continuous sampling.
109  */
110 static void IMU_sample_task(void* arg)
111 {
112     esp_err_t err;
113     //buffers for IMU data to be placed in
114     int16_t data_raw_acceleration[3];
115     int16_t data_raw_angular_rate[3];
116     //data frame for log data to be queued
117     log_data_t df;
118     //PID outputs
119     float_t acceleration_z_float = 0;
120     float_t pitch_float;
121     float_t roll_float;
122     //Fusion handle, outputs, and inputs
123     FusionAhrs ahrs;
124     FusionEuler euler;
125     FusionVector linear;
126     FusionVector gyroscope = {{0,0,0}};
127     FusionVector accelerometer = {{0,0,0}};
```

```
128     //Transformed PID outputs to be sent to DAC
129     uint16_t actuator1, actuator2, actuator3, actuator4;
130 
131     const char TAG[] = "IMU Sampling Task";
132 
133     /* Initialize and configure IMU*/
134     err = init_spi_IMU();
135     ESP_ERROR_CHECK(err);
136     err = config_IMU();
137     ESP_ERROR_CHECK(err);
138 
139     //set up fusion algorithm, use standard values as defined in Fusion Library
140     FusionAhrsInitialise(&ahrs);
141     FusionAhrsSettings * const ahrs_settings = malloc(sizeof(FusionAhrsSettings));
142     ahrs_settings->accelerationRejection =  10; //degrees
143     ahrs_settings->gyroscopeRange =         500;//dps
144     ahrs_settings->gain =                   0.5f; //influence of gyroscope
145     ahrs_settings->convention =             FusionConventionEnu; //earth axes convention (consistent with
        ISM330DHCX)
146     ahrs_settings->recoveryTriggerPeriod =  5.0/float_time; //approx 5 second recovery trigger period
147     FusionAhrsSetSettings(&ahrs, ahrs_settings);
148 
149     ESP_LOGI(TAG, "AHRS Initialized");
150 
151     /* Initialize PID Controllers */
152     init_pid_controllers();
153 
154     /* Initialize  and configure DAC */
155     err = init_spi_DAC();
156     ESP_ERROR_CHECK(err);
157     err = config_DAC();
158     ESP_ERROR_CHECK(err);
159 
160     /* Initialize Transform Matrix Distances */
161     err = set_distances(FA_COG,RA_COG,DA_COG,PA_COG);
162     ESP_ERROR_CHECK(err);
163     ESP_LOGI(TAG,"Distaces set\nF-%.3f\nR-%.3f\nD-%.3f\nP-%.3f", FA_COG, RA_COG, DA_COG, PA_COG);
164 
165     /* Calibrate the rest angle and acceleration for the vehicle */
166     /*-----------------------------------------------------------*/
167     /* Read a register from the IMU to kick off interupt sampling */
168     ism330dhcx_angular_rate_raw_get(&ISM330DHCX_dev_ctx, data_raw_angular_rate);
169     ism330dhcx_acceleration_raw_get(&ISM330DHCX_dev_ctx, data_raw_acceleration);
170 
171     /* While fusion isn't stable, keep reading the IMU.*/
172     FusionAhrsFlags fusionFlags = FusionAhrsGetFlags(&ahrs);
173     while(fusionFlags.initialising || fusionFlags.angularRateRecovery){
174         ulTaskNotifyTake(pdFALSE,portMAX_DELAY);
175         //when IMU_PIN_INT1 rising edge
176 
177         //read accel/gyro
178         memset(data_raw_acceleration, 0x00, 3 * sizeof(int16_t));
179         ism330dhcx_acceleration_raw_get(&ISM330DHCX_dev_ctx, data_raw_acceleration);
180         ism330dhcx_angular_rate_raw_get(&ISM330DHCX_dev_ctx, data_raw_angular_rate);
181 
182         //convert to dps for use in Fusion
183         //uses full scale of 500dps for int to float conversion
184         //reuse old data if peak detected
185         for(int i=0; i<3; i++){
186             if(data_raw_angular_rate[i] < IMU_GY_PEAK_REJ && data_raw_angular_rate[i] > -IMU_GY_PEAK_REJ){
187                 gyroscope.array[i] = (float_t)data_raw_angular_rate[i] * 0.0175f;
188             }
189         }
190 
191         //convert to g for use in Fusion
192         //uses full scale of 4 g for int to float conversion
193         //reuse old data if peak detected
194         for(int i=0; i<3; i++){
195             if(data_raw_angular_rate[i] < IMU_XL_PEAK_REJ && data_raw_angular_rate[i] > -IMU_XL_PEAK_REJ){
196                 accelerometer.array[i] = (float_t)data_raw_acceleration[i] * 0.000122f;
197             }
198         }
199         //update Fusion
200         FusionAhrsUpdateNoMagnetometer(&ahrs, gyroscope, accelerometer, float_time);
201         //refresh flags
202         fusionFlags = FusionAhrsGetFlags(&ahrs);
203     }
```

```
204     //once data is stable, set zeros to use for PID setpoint
205     euler = FusionQuaternionToEuler(FusionAhrsGetQuaternion(&ahrs));
206     float_t acceleration_zero = 0;  //set to zero, as noise crushing performed on xl data later would
        remove this offset
207     float_t pitch_zero = euler.angle.pitch;
208     float_t roll_zero = euler.angle.roll;
209 
210     ESP_LOGI(TAG, "Acceleration zero point: %.3f", acceleration_zero);
211     ESP_LOGI(TAG, "Pitch zero point: %.3f", pitch_zero);
212     ESP_LOGI(TAG, "Roll zero point: %.3f", roll_zero);
213 
214     //delay to ensure SD card is rady
215     vTaskDelay(100);
216 
217     /* Turn on Status LEDs */+
218     gpio_set_level(STATUS_LED2, 1);
219     gpio_set_level(STATUS_LED1, 1);
220 
221     /* Read a register from the IMU to kick off interupt sampling again, log collected data, control
        actuators as needed */
222     /*
        ------------------------------------------------------------------------------------------------------------------------
        */
223     ism330dhcx_angular_rate_raw_get(&ISM330DHCX_dev_ctx, data_raw_angular_rate);
224     ism330dhcx_acceleration_raw_get(&ISM330DHCX_dev_ctx, data_raw_acceleration);
225     ESP_LOGI(TAG,"Sampling started");
226 
227     while(1){
228         ulTaskNotifyTake(pdFALSE,portMAX_DELAY);
229         //when IMU_PIN_INT1 rising edge
230 
231         //read accel/gyro
232         memset(data_raw_acceleration, 0x00, 3 * sizeof(int16_t));
233         ism330dhcx_acceleration_raw_get(&ISM330DHCX_dev_ctx, data_raw_acceleration);
234         ism330dhcx_angular_rate_raw_get(&ISM330DHCX_dev_ctx, data_raw_angular_rate);
235 
236         //convert to dps for use in Fusion
237         //uses full scale of 500dps for int to float conversion
238         //reuse old data if peak detected
239         for(int i=0; i<3; i++){
240             if(data_raw_angular_rate[i] < IMU_GY_PEAK_REJ && data_raw_angular_rate[i] > -IMU_GY_PEAK_REJ){
241                 gyroscope.array[i] = (float_t)data_raw_angular_rate[i] * 0.0175f;
242             }
243         }
244 
245         //convert to g for use in Fusion
246         //uses full scale of 4 g for int to float conversion
247         //reuse old data if peak detected
248         for(int i=0; i<3; i++){
249             if(data_raw_angular_rate[i] < IMU_XL_PEAK_REJ && data_raw_angular_rate[i] > -IMU_XL_PEAK_REJ){
250                 accelerometer.array[i] = (float_t)data_raw_acceleration[i] * 0.000122f;
251             }
252         }
253         //update Fusion, and get the resulting values
254         FusionAhrsUpdateNoMagnetometer(&ahrs, gyroscope, accelerometer, float_time);
255         euler = FusionQuaternionToEuler(FusionAhrsGetQuaternion(&ahrs)); //angles
256         linear = FusionAhrsGetLinearAcceleration(&ahrs); //acceleration less gravity
257 
258         //send data to be logged
259         df.acceleration_z = linear.axis.z;
260         df.pitch = euler.angle.pitch;
261         df.roll = euler.angle.roll;
262         xQueueSendToBack(log_data_queue,&df,0);
263 
264         //crush acceleration data to 0 if below noise floor, crushing is based on absolute value of input
        data
265         float_t conditioned_acceleration_z = 0;
266         if (linear.axis.z > IMU_NOISE_FLOOR){
267             conditioned_acceleration_z = linear.axis.z - IMU_NOISE_FLOOR;
268         }
269         else if (linear.axis.z < -IMU_NOISE_FLOOR){
270             conditioned_acceleration_z = linear.axis.z + IMU_NOISE_FLOOR;
271         }
272 
273         if(en_actuators == true){// below sections not needed if actuators disabled
274             //pid updates
275             acceleration_z_float = PIDController_Update(&accel_z_controller, &acceleration_zero, &
```

```
    conditioned_acceleration_z);
276             pitch_float = PIDController_Update(&pitch_controller, &pitch_zero, &euler.angle.pitch);
277             roll_float = PIDController_Update(&roll_controller, &roll_zero, &euler.angle.roll);
278 
279             //transform
280             transform(&actuator1,&actuator2,&actuator3,&actuator4,&acceleration_z_float,&pitch_float,&
    roll_float);
281 
282             //write to dac refer to page 73 of Notebook or design document for mapping, dependent on
    physical wiring
283             ltc2664_write_and_update_1_dac(&LTC2664_dev_ctx, LTC2664_DAC_1, &actuator1);
284             ltc2664_write_and_update_1_dac(&LTC2664_dev_ctx, LTC2664_DAC_2, &actuator2);
285             ltc2664_write_and_update_1_dac(&LTC2664_dev_ctx, LTC2664_DAC_0, &actuator3);
286             ltc2664_write_and_update_1_dac(&LTC2664_dev_ctx, LTC2664_DAC_3, &actuator4);
287 
288         }
289     }
290 }
291 
292 /**
293     @brief Initialize the ISM330DHCX_device's SPI bus
294 */
295 esp_err_t init_spi_IMU(void){
296     esp_err_t ret;
297 
298     static const char init_spi_TAG[] = "IMU";
299     ESP_LOGI(init_spi_TAG, "Initializing bus SPI%d...", IMU_HOST + 1);
300 
301     spi_bus_config_t *buscfg = malloc(sizeof(spi_bus_config_t)); //save this config in the heap
302     memset(buscfg,0xFF,sizeof(spi_bus_config_t)); //disable extra config pins
303     buscfg->miso_io_num = IMU_PIN_NUM_MISO;
304     buscfg->mosi_io_num = IMU_PIN_NUM_MOSI;
305     buscfg->sclk_io_num = IMU_PIN_NUM_CLK;
306     buscfg->quadwp_io_num = -1,
307     buscfg->quadhd_io_num = -1,
308     buscfg->max_transfer_sz = 8; //limit transfer sz to 8 bytes
309     buscfg->flags = 0;
310     buscfg->data_io_default_level = 0;
311     buscfg->intr_flags = 0;
312     buscfg->isr_cpu_id = ESP_INTR_CPU_AFFINITY_AUTO;
313 
314     ret = spi_bus_initialize(IMU_HOST, buscfg, SPI_DMA_DISABLED); //initialize the bus using the
    controller defined in the preamble
315 
316     if(ret != ESP_OK){ // if we have an error
317         return (ret); //cancel any further operation and return
318     }
319     //else continue setup
320 
321     ESP_LOGI(init_spi_TAG, "Init Success. Adding device...");
322 
323     spi_device_interface_config_t *devcfg = malloc(sizeof(spi_device_interface_config_t));
324     memset(devcfg,0x00,sizeof(spi_device_interface_config_t));//zero out memory space
325     devcfg->command_bits =       1;//one command bit (R/W)
326     devcfg->address_bits =       7;//seven address bit (register address)
327     devcfg->mode=                3;//Clock high in idle, read on rising edge
328     devcfg->clock_source =       SPI_CLK_SRC_DEFAULT;
329     devcfg->duty_cycle_pos =     128;//50%/50% duty cycle
330     devcfg->cs_ena_pretrans =    0;//CS timed with clock
331     devcfg->cs_ena_posttrans =   0;
332     devcfg->clock_speed_hz=      9*1000*1000; //10Mhz
333     devcfg->input_delay_ns =     0;
334     devcfg->spics_io_num=        IMU_PIN_NUM_CS;
335     devcfg->flags =              SPI_DEVICE_NO_DUMMY; //disable high freq(>100MHZ) workaround
336     devcfg->queue_size=          7;  //queue 7 transactions at most
337     devcfg->pre_cb =             NULL; //No callbacks
338     devcfg->post_cb =            NULL;
339 
340     ret = spi_bus_add_device(IMU_HOST, devcfg, &IMU_spi);
341 
342     if(ret != ESP_OK){ // if we have an error
343         return (ret); //cancel any further operation and return
344     }
345 
346     //now the IMU device can have it's associated functions and handle defined
347     ISM330DHCX_dev_ctx.write_reg = IMU_write;
348     ISM330DHCX_dev_ctx.read_reg = IMU_read;
```

```
349     ISM330DHCX_dev_ctx.mdelay = platform_delay;
350     ISM330DHCX_dev_ctx.handle = IMU_spi;
351 
352     ESP_LOGI(init_spi_TAG, "Device driver ready for use");
353 
354     return(ret);
355 
356 }
357 
358 /**
359     @brief Initialize the LTC2664 device's SPI bus
360 */
361 esp_err_t init_spi_DAC(void){
362     esp_err_t ret;
363 
364     static const char init_spi_TAG[] = "DAC";
365     ESP_LOGI(init_spi_TAG, "Initializing bus SPI%d...", DAC_HOST + 1);
366 
367     spi_bus_config_t *buscfg = malloc(sizeof(spi_bus_config_t)); //save this config in the heap
368     memset(buscfg,0xFF,sizeof(spi_bus_config_t)); //disable extra config pins
369     buscfg->miso_io_num = DAC_PIN_NUM_MISO;
370     buscfg->mosi_io_num = DAC_PIN_NUM_MOSI;
371     buscfg->sclk_io_num = DAC_PIN_NUM_CLK;
372     buscfg->quadwp_io_num = -1,
373     buscfg->quadhd_io_num = -1,
374     buscfg->max_transfer_sz = 3; //limit transfer sz to 3 bytes
375     buscfg->flags = 0;
376     buscfg->data_io_default_level = 0;
377     buscfg->intr_flags = 0;
378     buscfg->isr_cpu_id = ESP_INTR_CPU_AFFINITY_AUTO;
379 
380     ret = spi_bus_initialize(DAC_HOST, buscfg, SPI_DMA_DISABLED); //initialize the bus using the
     controller defined in the preamble
381 
382     if(ret != ESP_OK){ // if we have an error
383         return (ret); //cancel any further operation and return
384     }
385     //else continue setup
386 
387     ESP_LOGI(init_spi_TAG, "Init Success. Adding device...");
388 
389     spi_device_interface_config_t *devcfg = malloc(sizeof(spi_device_interface_config_t));
390     memset(devcfg,0x00,sizeof(spi_device_interface_config_t));//zero out memory space
391     devcfg->command_bits =       4;//four command bit (R/W)
392     devcfg->address_bits =       4;//4 address bit (dac number)
393     devcfg->mode=                0;//Clock low in idle, read on rising edge
394     devcfg->clock_source =       SPI_CLK_SRC_DEFAULT;
395     devcfg->duty_cycle_pos =     128; //50%/50% duty cycle
396     devcfg->cs_ena_pretrans =    0;//CS timed with clock
397     devcfg->cs_ena_posttrans =   0;
398     devcfg->clock_speed_hz=      50*1000*1000;//50Mhz
399     devcfg->input_delay_ns =     0;
400     devcfg->spics_io_num=        DAC_PIN_NUM_CS;
401     devcfg->flags =              SPI_DEVICE_NO_DUMMY; //disable high freq(>100MHZ) workaround
402     devcfg->queue_size=          7;  //queue 7 transactions at most
403     devcfg->pre_cb =             NULL; //No callbacks
404     devcfg->post_cb =            NULL;
405 
406     ret = spi_bus_add_device(DAC_HOST, devcfg, &DAC_spi);
407 
408     if(ret != ESP_OK){ // if we have an error
409         return (ret); //cancel any further operation and return
410     }
411 
412     LTC2664_dev_ctx.write_reg = DAC_write;
413     LTC2664_dev_ctx.handle = DAC_spi;
414 
415     ESP_LOGI(init_spi_TAG, "Device driver ready for use");
416 
417     return(ret);
418 
419 }
420 
421 /**
422  * @brief Set PID constatnts and initialize the 3 controllers.
423  * @attention WILL BLOCK until FLOAT_TIME has been initialized
424  */
```

```
425 void init_pid_controllers(void){
426 
427     static const char TAG[] = "PID init";
428 
429     float_t derivative_cutoff_hz = 100.0f;
430     float_t tau = 1.0f / (M_PI * derivative_cutoff_hz);
431     /*  Limits defined for Integers as z,pitch,roll values
432         will be both positve and negative, easy to transform
433         into uint16 as our DAC expects. Constrain PID
434         outputs to not oversaturate the DAC/PID output */
435     float_t max_small_angle_coeff_accel = 1.1f * MAX(get_small_angle_a_2ab(),get_small_angle_b_2ab());
436     float_t max_small_angle_coeff_angle = 1.1f * MAX(get_small_angle_1_2ab(),get_small_angle_1_2cd());
437     float_t accelUpperLimit = INT16_MAX/max_small_angle_coeff_accel;
438     float_t accelLowerLimit = INT16_MIN/max_small_angle_coeff_accel;
439     float_t angleUpperLimit = INT16_MAX/max_small_angle_coeff_angle;
440     float_t angleLowerLimit = INT16_MIN/max_small_angle_coeff_angle;
441 
442     /* PIDs require float_time variable to be initialized, stall till ready */
443     while (float_time == 0){
444         vTaskDelay(10);
445     }
446 
447     //controller constants assigned
448     accel_z_controller.Kp = Z_XL_KP;
449     accel_z_controller.Ki = Z_XL_KI;
450     accel_z_controller.Kd = Z_XL_KD;
451     accel_z_controller.tau = tau;
452     accel_z_controller.limMin = accelLowerLimit;
453     accel_z_controller.limMax = accelUpperLimit;
454     accel_z_controller.T = float_time;
455 
456     pitch_controller.Kp = ROLL_KP;
457     pitch_controller.Ki = ROLL_KI;
458     pitch_controller.Kd = ROLL_KD;
459     pitch_controller.tau = tau;
460     pitch_controller.limMin = angleLowerLimit;
461     pitch_controller.limMax = angleUpperLimit;
462     pitch_controller.T = float_time;
463 
464     roll_controller.Kp = PITCH_KP;
465     roll_controller.Ki = PITCH_KI;
466     roll_controller.Kd = PITCH_KD;
467     roll_controller.tau = tau;
468     roll_controller.limMin = angleLowerLimit;
469     roll_controller.limMax = angleUpperLimit;
470     roll_controller.T = float_time;
471 
472 
473     ESP_LOGI(TAG, "Acceleration Controller Kp: %.3f", accel_z_controller.Kp );
474     ESP_LOGI(TAG, "Ki: %.3f", accel_z_controller.Ki );
475     ESP_LOGI(TAG, "Kd: %.3f", accel_z_controller.Kd);
476     ESP_LOGI(TAG, "Pitch Controller Kp: %.3f", pitch_controller.Kp );
477     ESP_LOGI(TAG, "Ki: %.3f", pitch_controller.Ki );
478     ESP_LOGI(TAG, "Kd: %.3f", pitch_controller.Kd);
479     ESP_LOGI(TAG, "Roll Controller Kp: %.3f", roll_controller.Kp );
480     ESP_LOGI(TAG, "Ki: %.3f", roll_controller.Ki );
481     ESP_LOGI(TAG, "Kd: %.3f", roll_controller.Kd);
482 
483     PIDController_Init(&accel_z_controller);
484     PIDController_Init(&roll_controller);
485     PIDController_Init(&pitch_controller);
486 
487 }
488 
489 /**
490     @brief configures the ISM330DHCX, checks that device is connected and working. Implements delay at
         start to allow device boot time. Must be after bus initialized.
491 */
492 esp_err_t config_IMU(void){
493     static const char Config_IMU_TAG[] = "IMU Config";
494     //give the IMU enough time to properly boot
495     vTaskDelay(IMU_BOOT_TIME / portTICK_PERIOD_MS);
496     esp_err_t ret;
497     //attempt to check device ID counter
498     uint8_t count = 0;
499     //
500     uint8_t whoAmI, rst;
```

```
501
502     //check that device is connected properly, try max 10 times
503     do{
504         ret = ism330dhcx_device_id_get(&ISM330DHCX_dev_ctx, &whoAmI);
505         ESP_ERROR_CHECK(ret); //read error checked here
506         platform_delay(10);
507         count ++;
508     }while(whoAmI != ISM330DHCX_ID && count < 10);
509     ESP_LOGI(Config_IMU_TAG, "Device found...");
510     if (whoAmI != ISM330DHCX_ID){
511         return ESP_ERR_NOT_FOUND;
512     }
513     //end device check
514
515     //reset the device to factory settings
516     ret = ism330dhcx_reset_set(&ISM330DHCX_dev_ctx, PROPERTY_ENABLE);
517     if (ret != 0){
518         return(ret);
519     }
520     //check that reset worked
521     do{
522         ESP_LOGI(Config_IMU_TAG, "In reset loop");
523         ret = ism330dhcx_reset_get(&ISM330DHCX_dev_ctx,&rst);
524         ESP_ERROR_CHECK(ret);
525     } while (rst);
526     //end device reset
527
528     /* Start device configuration. */
529     ret &= ism330dhcx_device_conf_set(&ISM330DHCX_dev_ctx, PROPERTY_ENABLE);
530     /* Enable Block Data Update */
531     ret &= ism330dhcx_block_data_update_set(&ISM330DHCX_dev_ctx, PROPERTY_DISABLE);
532     /* Set Output Data Rate */
533     ret &= ism330dhcx_xl_data_rate_set(&ISM330DHCX_dev_ctx, ISM330DHCX_XL_ODR_3332Hz);
534     ret &= ism330dhcx_gy_data_rate_set(&ISM330DHCX_dev_ctx, ISM330DHCX_GY_ODR_3332Hz);
535     /* Set full scale */
536     ret &= ism330dhcx_xl_full_scale_set(&ISM330DHCX_dev_ctx, ISM330DHCX_4g);
537     ret &= ism330dhcx_gy_full_scale_set(&ISM330DHCX_dev_ctx, ISM330DHCX_500dps);
538     /* Set Power Mode to high performance */
539     ret &= ism330dhcx_xl_power_mode_set(&ISM330DHCX_dev_ctx,ISM330DHCX_HIGH_PERFORMANCE_MD);
540     ret &= ism330dhcx_gy_power_mode_set(&ISM330DHCX_dev_ctx,ISM330DHCX_GY_HIGH_PERFORMANCE);
541     /* Internal filtering setup */
542     ret &= ism330dhcx_xl_hp_path_on_out_set(&ISM330DHCX_dev_ctx,ISM330DHCX_LP_ODR_DIV_800);
543     ret &= ism330dhcx_xl_filter_lp2_set(&ISM330DHCX_dev_ctx, PROPERTY_ENABLE);
544     ret &= ism330dhcx_gy_hp_path_internal_set(&ISM330DHCX_dev_ctx,ISM330DHCX_HP_FILTER_65mHz);
545     ret &= ism330dhcx_gy_lp1_bandwidth_set(&ISM330DHCX_dev_ctx, ISM330DHCX_ULTRA_LIGHT);
546     ret &= ism330dhcx_gy_filter_lp1_set(&ISM330DHCX_dev_ctx, PROPERTY_ENABLE);
547     /* Set Interupt 1 Output mode */
548     ret &= ism330dhcx_pin_mode_set(&ISM330DHCX_dev_ctx, ISM330DHCX_PUSH_PULL);
549     ret &= ism330dhcx_int_notification_set(&ISM330DHCX_dev_ctx, ISM330DHCX_BASE_LATCHED_EMB_PULSED);
550     /* Set interupt type (interupt on drdy for accelerometer. in sync w/ gyroscope) */
551     ism330dhcx_pin_int1_route_t pin_int1_route;
552     ret &= ism330dhcx_pin_int1_route_get(&ISM330DHCX_dev_ctx, &pin_int1_route);
553     pin_int1_route.int1_ctrl.int1_drdy_xl   = PROPERTY_ENABLE;
554     pin_int1_route.int1_ctrl.int1_boot      = PROPERTY_DISABLE;
555     pin_int1_route.int1_ctrl.int1_cnt_bdr   = PROPERTY_DISABLE;
556     pin_int1_route.int1_ctrl.den_drdy_flag  = PROPERTY_DISABLE;
557     pin_int1_route.int1_ctrl.int1_drdy_g    = PROPERTY_DISABLE;
558     pin_int1_route.int1_ctrl.int1_fifo_full = PROPERTY_DISABLE;
559     pin_int1_route.int1_ctrl.int1_fifo_ovr  = PROPERTY_DISABLE;
560     pin_int1_route.int1_ctrl.int1_fifo_th   = PROPERTY_DISABLE;
561     ret &= ism330dhcx_pin_int1_route_set(&ISM330DHCX_dev_ctx, &pin_int1_route);
562
563     /* Gather frequency data from IMU */
564     uint8_t freq;
565     ret &= ism330dhcx_read_reg(&ISM330DHCX_dev_ctx,ISM330DHCX_INTERNAL_FREQ_FINE,&freq,1);
566     /* Convert this raw data to seconds */
567     float_time = 2.0/(6666.6666666666666666+(10.0*freq)); //1.0 coresponds to ODR_Coeff page 60 of ST app.
        note AN5398
568     ESP_LOGI(Config_IMU_TAG, "Sample Period: %0.9f", float_time);
569
570     ESP_LOGI(Config_IMU_TAG,"IMU Configured");
571     return ret;
572
573 }
574
575 /**
576  * @brief Find the offset where the Motor amplifier is outputting approximetly equal voltages on the A and
```

```
     B channel, stay at that offset.
 * @param dac number corresponding to dac to calibrate.
 */
esp_err_t DAC_find_offset(ltc2664_DACS_t dac){
    esp_err_t ret;

    static const char TAG[] = "DAC";

    //zero out the offsets used in writing to the dac
    int32_t signed_offset = 0;
    ltc2664_save_offset(dac, &signed_offset);

    uint16_t input = 32625;//UINT16_MAX/2;
    uint16_t adjuster = input/64;
    //initialize the dac to midscale
    ret = ltc2664_write_and_update_1_dac(&LTC2664_dev_ctx, dac, &input);
    if(ret != ESP_OK) return ret;
    // //assign mux to match selected dac, mapping described in "ORC Documentation.pdf"
    // uint8_t mux[4] = {0,3,2,1};

    // gpio_set_level(MUX_PIN_A0, mux[dac] & 0x0001);
    // gpio_set_level(MUX_PIN_A1, (mux[dac] & 0x0002) >> 1);
    // //narrow the adjuster while moving the input value up or down according to the comparator input
    // while(adjuster > 2){
    //     vTaskDelay(15); //allow the DAC, Comparator, and amplifier to settle
    //     if(gpio_get_level(COMPARATOR_PIN)){ //comparator determines if output A is greater than B for the given channel
    //         input -= adjuster;  //and adjust the offset value appropriety
    //     }
    //     else{
    //         input += adjuster;
    //     }
    //     ret = ltc2664_write_and_update_1_dac(&LTC2664_dev_ctx, dac, &input); //see if the new offset is correct
    //     if(ret != ESP_OK) return ret;
    //     adjuster = adjuster/2;
    // }
    signed_offset = input-(UINT16_MAX/2);
    //save that offset to be used in DAC writing function;
    ltc2664_save_offset(dac, &signed_offset);

    ESP_LOGI(TAG, "DAC %i zero code: %i", dac, input);

    return ret;
}

/**
 * @brief All setup for DAC goes in this function, includes zero-offset finding
 */
esp_err_t config_DAC(void){
    static const char TAG[] = "DAC";
    ESP_LOGI(TAG, "Calibrating Motor Amplifiers");
    esp_err_t ret;
    for(int dac = 0; dac < 4; dac++){ //brute force the possible dacs found in ltc2664_DACS_t
        ret = DAC_find_offset(dac); //find offsets for all of them.
        if(ret != ESP_OK) return ret;
    }
    ESP_LOGI(TAG, "Calibration complete, DAC ready");
    return ret;
}

/**
 * @brief Main function. Initializes all GPIO, starts IMU task, and does SD logging
 */
void app_main(void){
    static const char TAG[] = "Main";
    esp_err_t err;

/* GPIO CONFIGURATION SECTION */
    /* Configure Interupt Pin For IMU */
    gpio_config_t io_conf = {};
    io_conf.intr_type = GPIO_INTR_POSEDGE;
    io_conf.pin_bit_mask = GPIO_INT_PIN_SEL;
    io_conf.mode = GPIO_MODE_INPUT;
    io_conf.pull_up_en = 0;
    io_conf.pull_down_en = 0;
    gpio_config(&io_conf);
```

```
651     /* Configure IMU_MISO Pin */
652     io_conf.intr_type = GPIO_INTR_DISABLE;
653     io_conf.pin_bit_mask = GPIO_MISO_PIN;
654     io_conf.mode = GPIO_MODE_INPUT;
655     io_conf.pull_up_en = 1;
656     io_conf.pull_down_en = 0;
657     gpio_config(&io_conf);
658     /* Configure Mux Address Pins */
659     io_conf.intr_type = GPIO_INTR_DISABLE;
660     io_conf.pin_bit_mask = GPIO_MUX_PINS;
661     io_conf.mode = GPIO_MODE_OUTPUT;
662     io_conf.pull_down_en = 0;
663     io_conf.pull_up_en = 0;
664     gpio_config(&io_conf);
665     /* Configure Comparator Signal Pin */
666     io_conf.intr_type = GPIO_INTR_DISABLE;
667     io_conf.pin_bit_mask = GPIO_COMP_PINS;
668     io_conf.mode = GPIO_MODE_INPUT;
669     io_conf.pull_down_en = 0;
670     io_conf.pull_up_en = 0;
671     gpio_config(&io_conf);
672     /* Configure Status LED Pin */
673     io_conf.intr_type = GPIO_INTR_DISABLE;
674     io_conf.pin_bit_mask = GPIO_STATUS_PINS;
675     io_conf.mode = GPIO_MODE_OUTPUT;
676     io_conf.pull_down_en = 0;
677     io_conf.pull_up_en = 0;
678     gpio_config(&io_conf);
679     /* Configure Mode Switches Pin */
680     io_conf.intr_type = GPIO_INTR_DISABLE;
681     io_conf.pin_bit_mask = GPIO_MODE_SWS_HIGH;
682     io_conf.mode = GPIO_MODE_INPUT;
683     io_conf.pull_down_en = 0;
684     io_conf.pull_up_en = 0;
685     gpio_config(&io_conf);
686     io_conf.intr_type = GPIO_INTR_DISABLE;
687     io_conf.pin_bit_mask = GPIO_MODE_SWS_LOW;
688     io_conf.mode = GPIO_MODE_INPUT;
689     io_conf.pull_down_en = 1;
690     io_conf.pull_up_en = 0;
691     gpio_config(&io_conf);
692     /* Read boot config switches, write global bools */
693     if(gpio_get_level(MODE_SW_EN_LOGGING) == 1){
694         ESP_LOGI(TAG,"Logging Enabled");
695         en_logging = true;
696     }
697     if(gpio_get_level(MODE_SW_EN_ACTUATOR) == 1){
698         ESP_LOGI(TAG,"Actuators Enabled");
699         en_actuators = true;
700     }
701     /* Turn off Status LED, not ready yet */
702     gpio_set_level(STATUS_LED2, 0);
703     gpio_set_level(STATUS_LED1, 0);
704 
705     /* Create Queue for transfering data to SD card function */
706     log_data_queue = xQueueCreate(7000, sizeof(log_data_t));
707 
708     /* Start tasks for IMU sampling*/
709     ESP_LOGI(TAG, "Starting IMU Task");
710     xTaskCreatePinnedToCore(IMU_sample_task, "IMU_sample_task", 4096, NULL, 9, &IMU_TASK, APP_CPU_NUM);
711 
712     /* Set up ISR */
713     ESP_LOGI(TAG, "Installing ISR");
714     err = gpio_install_isr_service(0);
715     ESP_ERROR_CHECK(err);
716     err = gpio_isr_handler_add(IMU_PIN_INT1, imu_isr_handler, (void*) IMU_PIN_INT1);
717     ESP_ERROR_CHECK(err);
718 
719     if(en_logging == true){//do not continue if logging is not enabled
720         /* Initialize SD Card*/
721         esp_vfs_fat_sdmmc_mount_config_t mount_config = {
722             .max_files = 5,
723             .allocation_unit_size = 16 * 1024
724         };
725         sdmmc_card_t *card;
726         const char mount_point[] = MOUNT_POINT;
727         ESP_LOGI(TAG, "Initializing SD card");
```

```
728
729         ESP_LOGI(TAG, "Using SDMMC peripheral");
730
731         //Increase the speed here as much as possible
732         sdmmc_host_t host = SDMMC_HOST_DEFAULT();
733         host.max_freq_khz = 45000;
734
735         sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
736         slot_config.width = 4;
737         slot_config.clk = SD_PIN_CLK;
738         slot_config.cmd = SD_PIN_CMD;
739         slot_config.d0 = SD_PIN_D0;
740         slot_config.d1 = SD_PIN_D1;
741         slot_config.d2 = SD_PIN_D2;
742         slot_config.d3 = SD_PIN_D3;
743         slot_config.flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP;
744
745         ESP_LOGI(TAG, "Mounting filesystem");
746         err = esp_vfs_fat_sdmmc_mount(mount_point, &host, &slot_config, &mount_config, &card);
747
748         if (err != ESP_OK) {
749             if (err == ESP_FAIL) {
750                 ESP_LOGE(TAG, "Failed to mount filesystem.");
751             } else {
752                 ESP_LOGE(TAG, "Failed to initialize the card (%s).", esp_err_to_name(err));
753             }
754             return;
755         }
756         ESP_LOGI(TAG, "Filesystem mounted");
757
758         sdmmc_card_print_info(stdout, card);
759
760         char temp_string[33];//char array to have each line of log temporarily printed to. allows
    formating to be applied
761         char *cache = malloc(CACHE_SIZE);    //cache allocation to store blocks of data to write to SD card
    , helps maintain write speed as less wear leveling takes place
762
763         char *file_name = MOUNT_POINT"/ORClog.csv";//specify the file to be written to
764
765         //attempt to open file
766         ESP_LOGI(TAG, "Opening file %s", file_name);
767         FILE *f = fopen(file_name, "a");
768         if (f == NULL) {
769             ESP_LOGE(TAG, "Failed to open file for writing");
770             return;
771         }
772
773         /* Log title block*/
774         int written_length = snprintf(temp_string, 32, "Log #: %li\n", esp_random() % 10000);//random 4
    digit log number, differentiates multiple test in one file
775         memcpy(cache, temp_string, written_length);
776         uint32_t lines = written_length;
777         if(en_actuators){
778             written_length = snprintf(temp_string, 32, "Actuators enabled\n");
779         }
780         else{
781             written_length = snprintf(temp_string, 32, "Actuators disabled\n");
782         }
783         memcpy(cache + (lines), temp_string, written_length);
784         lines += written_length;
785         while(float_time == 0){
786             vTaskDelay(100);
787         }
788         written_length = snprintf(temp_string, 32, "Interval:%f\n", float_time);
789         memcpy(cache + (lines), temp_string, written_length);
790         lines += written_length;
791         written_length = snprintf(temp_string, 32, "Acceleration, Pitch, Roll\n");
792         memcpy(cache + (lines), temp_string, written_length);
793         lines += written_length;
794
795         ESP_LOGI(TAG, "Begin Normal Operation");
796         log_data_t log_data;
797         while(gpio_get_level(MODE_SW_START_LOG) == 0){
798             vTaskDelay(100);
799         }
800         //clear the queue right before we start saving data.
801         xQueueReset(log_data_queue);
```

```
802         while(1){
803             if(xQueueReceive(log_data_queue,&log_data, 0)){
804                 //when a new sample comes in, write it to the cache
805                 written_length = snprintf(temp_string, 32, "%.4f, %.4f, %.4f\n", log_data.acceleration_z,
    log_data.pitch, log_data.roll);
806                 memcpy(cache + (lines), temp_string, written_length);
807                 lines += written_length;
808                 //if the cahce is nearly full, write it to the sd card
809                 if(lines >= CACHE_SIZE-64){ //lose as much as 1 second of data by saving approx. every
    second.
810                     cache[lines] = '\0';
811                     lines = 0;
812                     fprintf(f, cache);
813                     //close file, saves data written
814                     fclose(f);
815                     if(gpio_get_level(MODE_SW_START_LOG) == 0){//if log not paused
816                         //wait till next log should begin
817                         while(gpio_get_level(MODE_SW_START_LOG) == 0){
818                             vTaskDelay(100);
819                         }
820 
821                         if(gpio_get_level(MODE_SW_APPEND_LOG)==0){ //if not appending to log, start new
    entry
822                             //see if actuator status has changed
823                             if(gpio_get_level(MODE_SW_EN_ACTUATOR) == 1){
824                                 ESP_LOGI(TAG,"Actuators Enabled");
825                                 en_actuators = true;
826                             }
827                             else{
828                                 ESP_LOGI(TAG,"Actuators Disabled");
829                                 en_actuators = false;
830                             }
831                             /* Log title block*/
832                             written_length = snprintf(temp_string, 32, "Log #: %li\n", esp_random() %
    10000);
833                             memcpy(cache, temp_string, written_length);
834                             lines += written_length;
835                             if(en_actuators){
836                                 written_length = snprintf(temp_string, 32, "Actuators enabled\n");
837                             }
838                             else{
839                                 written_length = snprintf(temp_string, 32, "Actuators disabled\n");
840                             }
841                             memcpy(cache + (lines), temp_string, written_length);
842                             lines += written_length;
843                             written_length = snprintf(temp_string, 32, "Interval:%f\n", float_time);
844                             memcpy(cache + (lines), temp_string, written_length);
845                             lines += written_length;
846                             written_length = snprintf(temp_string, 32, "Acceleration, Pitch, Roll\n");
847                             memcpy(cache + (lines), temp_string, written_length);
848                             lines += written_length;
849                         }
850                         else{ //append to log, mark as such
851                             written_length = snprintf(temp_string, 32, "Log Paused, now resuming:\n");
852                             memcpy(cache + (lines), temp_string, written_length);
853                             lines += written_length;
854                         }//end if append log
855                         xQueueReset(log_data_queue); //clear out data that is out of date
856                     }//end if log paused
857                     //open the file again
858                     f = fopen(file_name, "a");
859                 }
860             }//end if check for new sample
861             else
862             {
863                 vTaskDelay(1); //allow idle state to feed watchdog
864             }
865         }
866     }//logging enabled if statement
867     return;
868 }
869 
870 
871 static void platform_delay(uint32_t ms)
872 {
873     vTaskDelay(ms / portTICK_PERIOD_MS);
874 }
```

```
875
876 /**
877  * @brief  Read generic device register (platform dependent)
878  *
879  * @param  handle    customizable argument. In this examples is used in
880  *                   order to select the correct sensor bus handler.
881  * @param  reg       register to read
882  * @param  bufp      pointer to buffer that store the data read
883  * @param  len       number of consecutive register to read
884  *
885  */
886 static int32_t IMU_read(void *handle, uint8_t reg, uint8_t *bufp,
887                         uint16_t len)
888 {
889     esp_err_t ret;
890     spi_transaction_t t;
891     memset( &t, 0x00, sizeof(t) );
892     t.cmd = 1;
893     t.addr = reg;
894     t.length = len*8;
895     t.rxlength = len*8;
896     t.rx_buffer = bufp;
897     t.flags = 0;
898
899     ret = spi_device_transmit(handle,&t);
900
901     return (int)ret;
902 }
903
904 /**
905  * @brief  Write generic device register (platform dependent)
906  *
907  * @param  handle    customizable argument. In this examples is used in
908  *                   order to select the correct sensor bus handler.
909  * @param  reg       register to write
910  * @param  bufp      pointer to data to write in register reg
911  * @param  len       number of consecutive register to write
912  *
913  */
914 static int32_t IMU_write(void *handle, uint8_t reg, const uint8_t *bufp,
915                          uint16_t len)
916 {
917     esp_err_t ret;
918     spi_transaction_t t;
919     memset( &t, 0x00, sizeof(t) );
920     t.cmd = 0;
921     t.addr = reg;
922     t.length = len*8;
923     t.tx_buffer = bufp;
924     t.flags = 0;
925
926     ret = spi_device_transmit(handle,&t);
927
928     return (int)ret;
929 }
930
931 /**
932  * @brief Write DAC register
933  *
934  * @param   handle  pointer to spi device handle
935  * @param   dac     DAC number to write to
936  * @param   command Command to to write
937  * @param   data    pointer to data to write
938  *
939  * @return  ret     0 = no error
940  */
941 static int32_t DAC_write(void *handle, ltc2664_DACS_t dac, uint8_t command, const uint16_t *data)
942 {
943     uint16_t temp_data = ((*data & 0xFF00) >> 8) |((*data & 0x00FF) << 8);
944     esp_err_t ret;
945     spi_transaction_t t;
946     memset( &t, 0x00, sizeof(t) );
947     t.cmd = command;
948     t.addr = dac;
949     t.length = 16;
950     t.tx_buffer = &temp_data;
951     t.flags = 0;
```

```
952
953     ret = spi_device_transmit(handle, &t);
954
955     return (int)ret;
956 }
957 //eof
```

PID.h

```
1  /**
2   ****************************************************************
3   * @file     PID.h
4   * @author   Corbin Roennebeck, WSU
5   * @brief    This file contains all the functions prototypes for the
6   *           PID.c file
7   ****************************************************************
8   */
9
10 /* Define to prevent recursive inclusion -------------------------*/
11 #ifndef PID_H
12 #define PID_H
13
14 #ifdef __cplusplus
15 extern "C" {
16 #endif
17
18 /* Includes ------------------------------------------------------*/
19 #include <stdint.h>
20 #include <stddef.h>
21 #include <esp_err.h>
22 #include "esp_log.h"
23 #include <math.h>
24
25 typedef struct PIDController_t {
26     /* Controller gains */
27     float_t Kp;
28     float_t Ki;
29     float_t Kd;
30
31     /* Derivative low-pass filter time constant */
32     float_t tau;
33
34     /* Output limits */
35     float_t limMin;
36     float_t limMax;
37
38     /* Sample time (in seconds) */
39     float_t T;
40
41     /* Controller "memory" */
42     float_t integrator;
43     float_t prevError;
44     float_t differentiator;
45     float_t prevMeasurement;
46
47     /* Controller output */
48     float_t out;
49
50 }PIDController_t;
51
52 void PIDController_Init(PIDController_t *pid);
53 float_t PIDController_Update(PIDController_t *pid, float_t *setpoint, float_t *measurement);
54 float_t PIDController_Update_zero_setpoint(PIDController_t *pid, float_t *measurement);
55
56 #ifdef __cplusplus
57 }
58 #endif
59
60 #endif
61 //eof
```

PID.c

```c
/**
  ******************************************************************************
  * @file    PID.c
  * @author  Corbin Roennebeck, WSU
  * @brief   PID controller implementation file
  * @note    Largely Based on a video from Phil's Lab:
  *          https://www.youtube.com/watch?v=zOByx3Izf5U&
  ******************************************************************************
  */

#include "PID.h"

void PIDController_Init(PIDController_t *pid){
    /* Clear controller variables */
    pid->integrator = 0.0f;
    pid->prevError = 0.0f;

    pid->differentiator = 0.0f;
    pid->prevMeasurement = 0.0f;

    pid->out = 0.0f;
}

float_t PIDController_Update(PIDController_t *pid, float_t *setpoint, float_t *measurement){

    /*
    * Error Signal
    */
    float_t error = *setpoint - *measurement;

    /*
    * Proportional
    */
    float_t proportional = pid->Kp * error;

    /*
    * Integral
    */
    pid->integrator = pid->integrator + 0.5f * pid->Ki * pid->T * (error + pid->prevError);

    /*
    * Anti-wind-up via dynamic integrator clamping
    */
    float_t limMinInt = 0.0f, limMaxInt = 0.0f;

    /* Compute integrator limits */
    if(pid->limMax > proportional){
        limMaxInt = pid->limMax - proportional;
    }
    if(pid->limMin < proportional){
        limMinInt = pid->limMin - proportional;
    }

    /* Clamp integrator */
    if(pid->integrator > limMaxInt){
        pid->integrator = limMaxInt;
    }
    else if (pid->integrator < limMinInt){
        pid->integrator = limMinInt;
    }

    /*
    * Derivative (band-limited differentiator)
    */

   pid->differentiator = (-2.0f * pid->Kd * (*measurement - pid->prevMeasurement)
                        + (2.0f * pid->tau - pid->T) * pid->differentiator)
                        / (2.0f * pid->tau + pid->T);

    /*
    * Comput output and apply limits
    */

    pid->out = proportional + pid->integrator + pid->differentiator;
```

```

    if(pid->out > pid->limMax){
        pid->out = pid->limMax;
    } else if (pid->out < pid->limMin){
        pid->out = pid->limMin;
    }

    /*
    * Store error and measurement for later use
    */

    pid->prevError = error;
    pid->prevMeasurement = *measurement;

    return (int16_t)pid->out;

}

float_t PIDController_Update_zero_setpoint(PIDController_t *pid, float_t *measurement){
    float_t zero = 0.0f;
    return PIDController_Update(pid,&zero,measurement);
}

//eof
```

LTC2664_reg.h

```
/**
 ******************************************************************
 * @file    LTC2664_reg.h
 * @author  Corbin Roennebeck, WSU
 * @brief   This file contains all the functions prototypes for the
 *          LTC2664_reg.c driver.
 ******************************************************************
 */

/* Define to prevent recursive inclusion -------------------------------*/
#ifndef LTC2664_REGS_H
#define LTC2664_REGS_H

#ifdef __cplusplus
extern "C" {
#endif

/* Includes ------------------------------------------------------------------*/
#include <stdint.h>
#include <stddef.h>
#include "esp_log.h"


 typedef enum
 {
    LTC2664_DAC_0 = 0,
    LTC2664_DAC_1 = 1,
    LTC2664_DAC_2 = 2,
    LTC2664_DAC_3 = 3,
 } ltc2664_DACS_t;


/** @brief       This section provide a set of functions used to read and
  *              write a generic register of the device.
  *              MANDATORY: return 0 -> no Error.
  *
  */
typedef int32_t (*ltcdev_write_ptr)(void *, ltc2664_DACS_t, uint8_t, const uint16_t *);

typedef struct
{
  /** Component mandatory fields **/
  ltcdev_write_ptr write_reg;
  void *handle;
} ltcdev_ctx_t;
```

```
46
47
48 void ltc2664_save_offset(ltc2664_DACS_t dac, int32_t *offset);
49
50 int32_t ltc2664_write_reg(const ltcdev_ctx_t *ctx, ltc2664_DACS_t dac, uint8_t command,
51                           const uint16_t *data);
52 /**
53   * @}
54   *
55   */
56
57
58
59
60
61
62 // /* Write to DAC register functions*/
63 // int32_t ltc2664_write_to_1_dac(const ltcdev_ctx_t *ctx, uint8_t dac, uint16_t data);
64 // int32_t ltc2664_write_to_all(const ltcdev_ctx_t *ctx, uint16_t data);
65
66 // /* Set output span functions*/
67 // int32_t ltc2664_set_span_1_dac(const ltcdev_ctx_t *ctx, uint8_t dac, uint16_t data);
68 // int32_t ltc2664_set_span_all(const ltcdev_ctx_t *ctx, uint16_t data);
69
70 // /* Update DAC Ouput functions*/
71 // int32_t ltc2664_update_1_dac(const ltcdev_ctx_t *ctx, uint8_t dac);
72 // int32_t ltc2664_update_all(const ltcdev_ctx_t *ctx);
73
74 /* Write and update functions*/
75 int32_t ltc2664_write_and_update_1_dac(const ltcdev_ctx_t *ctx, ltc2664_DACS_t dac, uint16_t* data);
76 // int32_t ltc2664_write_and_update_all(const ltcdev_ctx_t *ctx, uint16_t data);
77 // int32_t ltc2664_write_1_update_all(const ltcdev_ctx_t *ctx, uint8_t dac, uint16_t data);
78
79 // /* Power down (disable) functions*/
80 // int32_t ltc2664_power_down_dac(const ltcdev_ctx_t *ctx, uint8_t dac);
81 // int32_t ltc2664_power_down_chip(const ltcdev_ctx_t *ctx);
82
83 // /* Register toggle funcitons*/
84 // int32_t ltc2664_toggle_select(const ltcdev_ctx_t *ctx, uint8_t dac, uint16_t data);
85 // int32_t ltc2664_toggle_select_all(const ltcdev_ctx_t *ctx, uint16_t data);
86
87 #ifdef _cplusplus
88 }
89 #endif
90
91 #endif /* LTC2664_REGS_H*/
92
93 //eof
```

LTC2664_reg.c

```
1  /**
2    ******************************************************************************
3    * @file    LTC2664_reg.c
4    * @author  Corbin Roennebeck, WSU
5    * @brief   LTC2664 driver file
6    ******************************************************************************
7    */
8
9
10 #include "LTC2664_reg.h"
11
12 //positive and negative offsets are saved to allow for easy addition and less math in each write call.
13 static uint16_t dac_offset_positive[4] = {0,0,0,0};
14 static uint16_t dac_offset_negative[4] = {0,0,0,0};
15 static uint16_t dac_upper_limit[4] = {UINT16_MAX,UINT16_MAX,UINT16_MAX,UINT16_MAX}; //any input larger
       than this will cause problems (wraparound)
16 static uint16_t dac_lower_limit[4] = {0,0,0,0}; //any input less than this will cause problems (wraparound
       )
17
18 /**
19  * @brief Generic write to register
```

```
20  *
21  * @param ctx        read / write interface definitions(ptr)
22  * @param dac       DAC address
23  * @param command   8 bit
24  * @param data      pointer to data to write
25  */
26 int32_t ltc2664_write_reg(const ltcdev_ctx_t *ctx, ltc2664_DACS_t dac, uint8_t command, const uint16_t *
        data){
27     int32_t ret =0;
28 
29     if (ctx == NULL){
30         return -1;
31     }
32     if (dac > 3){
33         return -1;
34     }
35 
36     ret = ctx->write_reg(ctx->handle, (uint8_t)dac, command, data);
37 
38     return ret;
39 }
40 
41 
42 /**
43  * @brief save library offset values as appropriate, takes int_32 offset and corrects to uint_32
44  *
45  * @param dac     DAC to edit offset values for
46  * @param offset  Pointer to offset data
47  */
48 void ltc2664_save_offset(ltc2664_DACS_t dac, int32_t *offset){
49   if(*offset < 0){
50     dac_offset_positive[(int)dac] = 0;
51     dac_offset_negative[(int)dac] = (uint16_t)(*offset * -1);
52     dac_lower_limit[(int)dac] = dac_offset_negative[dac];
53     dac_upper_limit[(int)dac] = UINT16_MAX;
54   }
55   else{
56     dac_offset_positive[(int)dac] = (uint16_t)(*offset);
57     dac_offset_negative[(int)dac] = 0;
58     dac_lower_limit[(int)dac] = 0;
59     dac_upper_limit[(int)dac] = UINT16_MAX - *offset;
60   }
61 }
62 
63 
64 /**
65  * @brief write data to 1 dac and update dac output. prevents wraparound when applying offset values.
66  *
67  * @param ctx   read / write interface definitions(ptr)
68  * @param dac   DAC address
69  * @param data  pointer to data to write
70  */
71 
72    //static const char DACTAG[] = "1Dac";
73 int32_t ltc2664_write_and_update_1_dac(const ltcdev_ctx_t *ctx, ltc2664_DACS_t dac, uint16_t *data){
74     int32_t ret  =0;
75     uint16_t temp_data;
76     if(*data > dac_upper_limit[dac]){ //properly clamp high inputs
77       temp_data = UINT16_MAX;
78     }
79     else if(*data < dac_lower_limit[dac]){ //properly clamp low inputs
80       temp_data = 0;
81     }
82     else{ //apply offsets only if they won't cause wraparound
83       temp_data = *data + dac_offset_positive[dac] - dac_offset_negative[dac];
84     }
85     //write to dac, and update, command = 3
86     //ESP_LOGI(DACTAG, "ctx %u, dac %i, data %i", ctx,dac,&temp_data);
87     ret = ltc2664_write_reg(ctx, dac, 3, &temp_data);
88 
89     return ret;
90 }
91 //eof
```

Transform.h

```c
/**
 ****************************************************************
 * @file     Transform.h
 * @author   Corbin Roennebeck, WSU
 * @brief    This file contains all the functions prototypes for the
 *           Transform.c file
 ****************************************************************
 */

/* Define to prevent recursive inclusion -------------------------*/
#ifndef Transform_H
#define Transform_H

#ifdef __cplusplus
extern "C" {
#endif

/* Includes ------------------------------------------------------*/
#include <stdint.h>
#include <stddef.h>
#include <esp_err.h>
#include "esp_log.h"
#include <math.h>

float_t get_small_angle_a_2ab(void);
float_t get_small_angle_b_2ab(void);
float_t get_small_angle_1_2ab(void);
float_t get_small_angle_1_2cd(void);
esp_err_t set_distances(float_t a, float_t b, float_t c, float_t d);
void transform(uint16_t *act1, uint16_t *act2, uint16_t *act3, uint16_t *act4, float_t *fZ, float_t *
    fTheta, float_t *fPhi);



#ifdef __cplusplus
}
#endif

#endif
//eof
```

Transform.c

```c
/**
  ******************************************************************************
  * @file     Transform.c
  * @author   Corbin Roennebeck, WSU
  * @brief    Transform Algorithm file
  ******************************************************************************
  */

#include "Transform.h"

//distances between actuator linkage and IMU small angle aprox.
static float_t _a_2ab = 0;// a/(2(a+b))
static float_t _b_2ab = 0;// b/(2(a+b))
static float_t _1_2ab = 0;// 1/(2(a+b))
static float_t _1_2cd = 0;// 1/(2(c+d))

//member access
float_t get_small_angle_a_2ab(void){ return _a_2ab; }
float_t get_small_angle_b_2ab(void){ return _b_2ab; }
float_t get_small_angle_1_2ab(void){ return _1_2ab; }
float_t get_small_angle_1_2cd(void){ return _1_2cd; }

static const char TAG[] = "Transform";

/**
 * @brief set vehicle actuator to IMU dimensions and associated transform multiplier
 * @bug NEED TO DEFINE UNITS FOR DISTANCE
```

```
 *
 * @param   a   distance to front axle linkages (in meters)
 * @param   b   distance to rear axle linkages (in meters)
 * @param   c   distance to drive side linkages (in meters)
 * @param   d   distance to passenger side linkages (in meters)
 */
esp_err_t set_distances(float_t a, float_t b, float_t c, float_t d){
    if(a < 0 || b < 0 || c < 0 || d <0 ){
        ESP_LOGE(TAG, "All distances must be positive!");
        return ESP_ERR_NOT_SUPPORTED;
    }

    _a_2ab = a/(2.0*(a+b));
    _b_2ab = b/(2.0*(a+b));
    _1_2ab = 1.0/(2.0*(a+b));
    _1_2cd = 1.0/(2.0*(c+d));
    return ESP_OK;
}

/**
 * @brief Transform vertical acceleration, pitch, and roll forces into 4 corner forces using small angle approximations
 * @attention float data must be constrained to int16 min/max: -32768 to 32767
 *              this allows easy conversion to uint16 data: 0->65535, which the DAC will enjoy
 *
 * @param   act1    pointer to actuator 1 (front driver) force output
 * @param   act2    pointer to actuator 2 (front passenger) force output
 * @param   act3    pointer to actuator 3 (rear driver) force output
 * @param   act4    pointer to actuator 4 (rear passenger) force output
 * @param   fZ      pointer to overall Z force needed
 * @param   fTheta  pointer to overall pitch force needed
 * @param   fPhi    pointer to overall roll force needed
 *
 */
void transform(uint16_t *act1, uint16_t *act2, uint16_t *act3, uint16_t *act4, float_t *fZ, float_t *fTheta, float_t *fPhi){
    //generate all multiplicands (all library constants are garunteed between -1 and 1, typically -1/2 and 1/2)
    float_t _b_fZ = _b_2ab * (*fZ);
    float_t _a_fZ = _a_2ab * (*fZ);
    float_t _p_fT = _1_2ab * (*fTheta);
    float_t _p_fP = _1_2cd * (*fPhi);
    //perform addition to find each actuator force and zero reference the result.
    float_t inter_act[4];
    inter_act[0] = _b_fZ - _p_fT + _p_fP + 32768.0f;
    inter_act[1] = _b_fZ - _p_fT - _p_fP + 32768.0f;
    inter_act[2] = _a_fZ + _p_fT + _p_fP + 32768.0f;
    inter_act[3] = _a_fZ + _p_fT - _p_fP + 32768.0f;
    for (int i = 0; i < 4; i++){
        if(inter_act[i]>(float_t)UINT16_MAX) inter_act[i] = UINT16_MAX;
        else if (inter_act[i]<0.0f) inter_act[i] = 0;
    }

    *act1 = (uint16_t)inter_act[0];
    *act2 = (uint16_t)inter_act[1];
    *act3 = (uint16_t)inter_act[2];
    *act4 = (uint16_t)inter_act[3];
}
//eof
```

Python Plotting main

```
#CSV data plotting and stats analysis program for ORC

import matplotlib.pyplot as plt
import csv
import numpy as np
from scipy.signal import medfilt
from scipy.stats import ttest_ind
from collections import defaultdict
#default plot text size
plt.rcParams.update({'font.size': 50})  # Sets default font size to 14
```

```
# File path
filename = r"D:\ORCLOG.csv"  # Update as needed

# Data storage
logs = defaultdict(lambda: {"Acceleration [g]": [], "Pitch [°]": [], "Roll [°]": [], "Time [s]": []})
current_log = None
actuator_status = None
interval = 1.0  # Default interval in case it's missing
sample_number = 0

# Read the file
with open(filename, "r") as file:
    reader = csv.reader(file)
    #iterate through each row in file
    for row in reader:
        #only read rows with content
        if row:
            #row starts with log number, save that
            if row[0].startswith("Log #"):
                current_log = row[0]
                sample_number = 0
            #the following row should have the actuator status (enabled/disabled)
            elif "Actuators" in row[0]:
                actuator_status = row[0]
            #the row after that should have the interval between samples (for conversion to actual seconds)
            elif row[0].startswith("Interval:"):
                try:
                    interval = float(row[0].split(":")[1].strip())
                    key = f"{actuator_status}"
                    logs[key]["Acceleration [g]"].append([])
                    logs[key]["Pitch [°]"].append([])
                    logs[key]["Roll [°]"].append([])
                    logs[key]["Time [s]"].append([])
                except ValueError:
                    interval = 1.0  # Fallback if parsing fails
            #rows of data have the acceleration, pitch, and roll as floats in that order.
            elif len(row) == 3 and current_log and actuator_status:
                try:
                    acc, p, r = map(float, row)
                    key = f"{actuator_status}"
                    logs[key]["Acceleration [g]"][sample_number].append(acc)
                    logs[key]["Pitch [°]"][sample_number].append(p)
                    logs[key]["Roll [°]"][sample_number].append(r)
                    logs[key]["Time [s]"][sample_number].append(len(logs[key]["Time [s]"][sample_number]) * interval)
                except ValueError:
                    continue  # Skip invalid rows
            #each time a row starts with "Log Paused" that means a new run has started. These are broken apart to allow us to preform a t test on the RMS, max, and min
            elif row[0].startswith("Log Paused"):
                sample_number += 1
                key = f"{actuator_status}"
                logs[key]["Acceleration [g]"].append([])
                logs[key]["Pitch [°]"].append([])
                logs[key]["Roll [°]"].append([])
                logs[key]["Time [s]"].append([])
            #if anything else, the line can be ignored

# Function to apply median filter
def filter_data(data):
    return medfilt(data, kernel_size=15)

# Function to compute stats
def data_stats(data):
    rms = np.sqrt(np.mean(np.square(data)))  # Compute RMS value
    min_val, max_val = np.min(data), np.max(data)  # Find min/max
    return rms, min_val, max_val

# Function to plot data and display statistics
def plot_data(logs, data_key, title, filename,stats=False,derivative=False):
    # default plot text size
    if stats:
        plt.rcParams.update({'font.size': 36})  # Sets default font size to 14
    else:
        plt.rcParams.update({'font.size': 50})
```

```
    fig, ax = plt.subplots(figsize=(38.4, 21.6), dpi=100)  # 4K resolution
    stats_texts = []
    rms_populations = []
    min_populations = []
    max_populations = []
    for key, data in logs.items():
        rms_vals = []
        min_vals = []
        max_vals = []
        first_3_count = 0
        time_index = 0
        for i in data[data_key]:
            filtered_data = filter_data(i)
            if derivative:
                filtered_data = np.gradient(filtered_data)
            rms, min_val, max_val = data_stats(filtered_data)
            rms_vals.append(rms)
            min_vals.append(min_val)
            max_vals.append(max_val)
            if(first_3_count ==2): #plot only the 11th sample
                ax.plot(data["Time [s]"][time_index], filtered_data, label=key, linestyle = "-", linewidth
 = 5.0)
            first_3_count += 1
            time_index += 1
        rms_populations.append(rms_vals)
        min_populations.append(min_vals)
        max_populations.append(max_vals)
        stats_texts.append(f"{key}\nAverage RMS: {np.mean(rms_vals):.5f}\nAverage Min: {np.mean(min_vals)
:.5f}\nAverage Max: {np.mean(max_vals):.5f}")
    rms_t_stat, rms_p_value = ttest_ind(rms_populations[0], rms_populations[1], alternative="less",
 equal_var=False)
    min_t_stat, min_p_value = ttest_ind(min_populations[0], min_populations[1], alternative="greater",
 equal_var=False)
    max_t_stat, max_p_value = ttest_ind(max_populations[0], max_populations[1], alternative="less",
 equal_var=False)
    stats_texts.append(f"One-sided T tests:")
    stats_texts.append(f"RMS T statistic: {rms_t_stat:.5f}\nRMS P value: {rms_p_value:.5e}")
    stats_texts.append(f"Min T statistic: {min_t_stat:.5f}\nMin P value: {min_p_value:.5e}")
    stats_texts.append(f"Max T statistic: {max_t_stat:.5f}\nMax P value: {max_p_value:.5e}")

    # Set plot labels and grid
    ax.set_xlabel("Time [s]")
    ax.set_ylabel(data_key)
    ax.set_title(title)
    ax.legend()
    ax.grid(True)

    # Add statistics text to the right of the plot
    stats_text = "\n\n".join(stats_texts)
    if stats:
        plt.gcf().text(.88, .19, stats_text, fontsize=24, verticalalignment='center',
                bbox=dict(facecolor='white', alpha=0.9))

    # Save and close the figure
    plt.savefig(filename, dpi=100)  # Save as high-res image
    plt.close()

# Generate separate plots
plot_data(logs, "Acceleration [g]", "Corner Test Single Sample - Median Filtered (ks=15) Acceleration Data
", "acceleration_Corner.png")
plot_data(logs, "Pitch [°]", "Corner Test Single Sample - Median Filtered (ks=15) Pitch Data", "
pitch_Corner.png")
plot_data(logs, "Roll [°]", "Corner Test Single Sample - Median Filtered (ks=15) Roll Data", "roll_Corner.
png")
plot_data(logs, "Acceleration [g]", "Corner Test Single Sample - Median Filtered (ks=15) Computed Jerk", "
jerk_Corner.png", derivative=True)
plot_data(logs, "Acceleration [g]", "Corner Test Single Sample - Median Filtered (ks=15) Acceleration Data
", "acceleration_Corner_stats.png",stats=True)
plot_data(logs, "Pitch [°]", "Corner Test Single Sample - Median Filtered (ks=15) Pitch Data", "
pitch_Corner_stats.png",stats=True)
plot_data(logs, "Roll [°]", "Corner Test Single Sample - Median Filtered (ks=15) Roll Data", "
roll_Corner_stats.png",stats=True)
plot_data(logs, "Acceleration [g]", "Corner Test Single Sample - Median Filtered (ks=15) Computed Jerk", "
jerk_Corner_stats.png", derivative=True, stats=True)
print("Plots saved.")

#EOF
```

Appendix C

The following BOMs include all the necessary components to construct the PCBs for ORC. These BOMs don't include flux, solder, solder paste, or wire. A decent lab should have these in bulk.

The actual vehicle and mechanical parts (except gearmotor) have been excluded as ORC should be platform agnostic. These parts were also largely sourced from the designers personal collection and may no longer be obtainable at reasonable prices. An estimate of these parts is $750, assuming original retail pricing.

Motor Amplifier BOM

Note, 4 Motor Amplifiers are needed per Main Board, BOM priced per board.

Manufacturer Part Number	Source	Quantity	Unit Price	Unit of Measure	Extended Price
504222B00000G	DigiKey	1	2.39	EACH	$2.39
LM675T/LF02	DigiKey	2	5.7	EACH	$17.10
RN73R2ATTD2130D25	DigiKey	1	0.11	EACH	$0.22
RK73H2ATTD1000D	DigiKey	1	0.23	EACH	$0.46
RT0603BRD07988KL	DigiKey	1	0.1	EACH	$0.20
RNCS0805BKE100K	DigiKey	1	0.17	EACH	$0.34
RNCS0805BKE10K0	DigiKey	1	0.17	EACH	$0.34
RT0805BRD0790K9L	DigiKey	1	0.16	EACH	$0.32
RMCF0805FT6K98	DigiKey	2	0.1	EACH	$0.30
GRM55DR72J224KW01L	DigiKey	2	1.03	EACH	$3.09
GCJ31CR72E104KXJ3L	DigiKey	4	0.32	EACH	$1.60
NT-H2	Noctua	1	14.95	SYRINGE	$14.95
92095A182	McMaster-Carr	2	0.07	EACH	$0.14
90592A009	McMaster-Carr	2	0.0223	EACH	$0.04
PCB: Motor Amplifier	OSH Park	1	4.85	EACH	$4.85
4689	Pololu	1	32.95	EACH	$32.95
Total					**$79.29**

Table 5: ORC Motor Amplifier BOM, excluding shipping, tax, tariff, MOQ

Main Board BOM

Manufacturer Part Number	Source	Quantity	Unit Price	Unit of Measure	Extended Price
ESP32-WROOM-32UE-N8	DigiKey	1	5.28	EACH	$5.28
AD790JRZ	DigiKey	1	13.27	EACH	$13.27
MUX509IPWR	DigiKey	1	2.58	EACH	$2.58
LTC2664CUH-16	DigiKey	1	36.98	EACH	$36.98
MSD-4-A	DigiKey	1	0.36	EACH	$0.72
LO T67K-K1L2-24-0-2-R18-Z	DigiKey	5	0.31	EACH	$1.86
BAT760-7	DigiKey	1	0.56	EACH	$1.12
GSEZ5B159	DigiKey	3	0.1	EACH	$0.40
SS8050-G	DigiKey	3	0.24	EACH	$0.96
AMS1117-3.3	DigiKey	2	0.63	EACH	$1.89
RC1206FR-0710KL	DigiKey	12	0.018	EACH	$0.23
RNCP1206FTD22K1	DigiKey	1	0.1	EACH	$0.20
RC1206FR-0747K5L	DigiKey	1	0.1	EACH	$0.20
RC1206FR-072KL	DigiKey	1	0.1	EACH	$0.20
GRM31CR71A226KE15K	DigiKey	7	0.47	EACH	$3.76
GRM31CR71E106KA12L	DigiKey	2	0.32	EACH	$0.96
GCJ31CR72E104KXJ3L	DigiKey	24	0.19	EACH	$4.94
CP2102N-A02-GQFN28	DigiKey	1	5.79	EACH	$11.58
GRM31CR71E105KA01L	DigiKey	4	0.54	EACH	$2.70
RC1206JR-07510RL	DigiKey	1	0.1	EACH	$0.20
OS102011MS2QN1	DigiKey	4	0.68	EACH	$3.40
PTS645SM43SMTR92 LFS	DigiKey	2	0.3	EACH	$0.90
AP63205WU-7	DigiKey	1	1.38	EACH	$2.76
FP3-4R7-R	DigiKey	1	3.69	EACH	$7.38
LM7905CT/NOPB	DigiKey	1	1.53	EACH	$3.06
LM7805CT/NOPB	DigiKey	1	1.74	EACH	$3.48
GRM31C5C1E224JE02L	DigiKey	1	0.47	EACH	$0.94
282834-2	DigiKey	4	2.74	EACH	$13.70
DI79L09DAB	DigiKey	1	0.51	EACH	$1.02
DI78L09UAB	DigiKey	1	0.47	EACH	$0.94
GRM31C5C1E334JE01L	DigiKey	2	0.49	EACH	$1.47
C1F 2.5	DigiKey	8	0.258	EACH	$4.13
CABLE-PH08	DigiKey	1	2.82	EACH	$2.82
S8B-PH-K-S	DigiKey	1	0.34	EACH	$0.34
0685H9120-01	DigiKey	2	0.36	EACH	$1.44
GRM31MR71E225KA93L	DigiKey	1	0.35	EACH	$0.70
RC1206FR-07750RL	DigiKey	5	0.1	EACH	$0.60
1051640001	DigiKey	1	0.93	EACH	$1.86
457190008	DigiKey	4	12.01	EACH	$60.05
XT60PW	Amazon	1	9.99	PACK	$9.99
PCB: Main Board	OSH Park	1	26.67	EACH	$26.67
4569	adafruit	1	29.50	EACH	$29.50
2280-3S1P	SMC Racing	2	13.95	EACH	$27.90
Total:					**$295.08**

Table 6: ORC Main Board BOM, excluding shipping, tax, tariff, MOQ