Lightweight Embedded Thrust Vector Control on the RP2040

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by:

Maxwell Klugherz

MEng Field Advisor: Van Hunter Adams

Degree Date: December 2024

Table of Contents

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title:
Lightweight Embedded Thrust Vector Control on the RP2040

Author:
Maxwell Klugherz

Abstract:
Gimbaled thrust vector control (TVC), first introduced in the 1940s for missile guidance, enables precise control of a propelled vehicle's trajectory. After eight decades, gimbaled TVC has evolved to become an integral part of high-cadence commercial launches by allowing reusability. This project serves as an exploration and application of TVC technologies, with the ultimate goal of creating a small-scale launch vehicle capable of maintaining a stable flight trajectory within the thrust duration of a commercially available solid-propellant rocket motor (approximately 2.5 seconds). Key objectives include the prototyping of a 2-axis gimbal mechanism for pitch and yaw control, the design, fabrication and testing of a custom microcontroller-equipped printed circuit board, the integration of bare-metal firmware, and the simulation of control and estimation systems alongside vehicle dynamics in MATLAB Simulink. As such, this project serves as a comprehensive demonstration of how industry-standard design techniques can be applied to the development a hobbyist-scale launcher.

Executive Summary

The Saturn V Launch Vehicle Digital Computer (1960s) weighed 70 lbs, contained up to 106 kB of memory, and could execute 12190 instructions per second (IPS). By comparison, Raspberry Pi's RP2040 microcontroller (2021) can execute 3.3 MIPS and contains 264 kB of SRAM, all within a QFN package. These evolutions in available memory and compute – along with 21st-century rapid prototyping and simulation tools – have significantly improved how hobbyists design, manufacture and test physical systems.

This project serves as a challenge in applying all these tools in the design of a small model rocket. First, a model for the vehicle's rotation in an inertial reference frame is derived. Then, a 3D-printed 2-axis gimbal is designed to deflect motor torque in both pitch and yaw. These two deflections are controlled with a pair of PWM-driven 9g servos.

Vehicle requirements motivate the integration of multiple small electrical components within a single structure, so a custom printed circuit board is designed, sent for manufacture, and tested. This board includes the RP2040 microcontroller, supporting clock and power hardware, the BNO055 IMU for attitude data collection, a handful of user-interfacing components, and multiple mechanisms for on-board data logging.

Finally, the vehicle dynamics, controls and estimation architecture are simulated in MATLAB Simulink. A proportional-derivative (PD) controller is developed to return the vehicle to a desired attitude and preliminary steps are taken towards the creation of an Extended Kalman Filter for estimation of the vehicle's attitude.

Due to the wide range of tools and techniques used to design the system, this project maintains a level of generalism. No one aspect of this project carries particular weight; rather, each feature is attended to with similar specificity.

Ultimately, each and every aspect within this project is extensible. As such, this project reflects the foundational practice in developing a system in the most simple form while still enabling the introduction of further complexity down the road. In consequence, the work presented here will not be the end!

1. Introduction

Thrust vector control (TVC) is a control technique that allows propelled vehicles to change their attitude by adjusting the direction of their motor's or engine's thrust (i.e., vectoring). This control method is common in launch vehicles, guided missiles, and advanced aircraft.

This technique appears through different means. In launch vehicle applications, engine assemblies are mounted to flexible structures that contain an actuation mechanism; a pivot of the structure directs the thrust, which changes the direction of torque applied to the vehicle. Alternatively, vehicle designers might employ jet vanes, which are aerodynamic flight control surfaces that direct the exhaust from an engine. It is important to note that any control involving the manipulation of thrust can only be performed while the thrust is occurring; once the thrust ends, the vehicle loses all control.

For this project, a gimbaled mechanism is developed. This mechanism is 3D-printed, which facilitates rapid iterative changes to its mounting and assembly.

Thrust vectoring enables control of all 6 of a vehicle's degrees of freedom: 3 rotational degrees for the orientation of the body with respect to the inertial reference frame (roll, pitch and yaw), and 3 translational degrees for motion in an inertial reference frame. This design ignores the translational degrees of freedom and focuses on the vehicle's attitude. While the Euler angle representation of attitude is non-singular (i.e., $\cos(\theta) \to 0$ for $\theta \to \frac{\pi}{2}$, which might produce some values that approach ∞ in the flight software), these singularities are ignored, since a nominal flight trajectory should not contain any horizontal nor near-horizontal attitude.

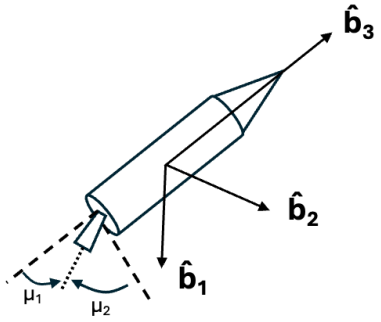


Figure 1: Body Reference Frame

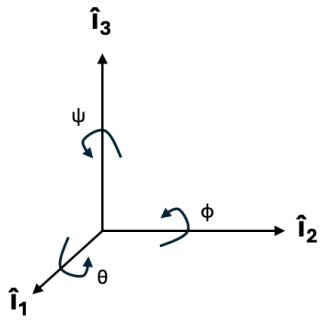


Figure 2: Inertial Reference Frame

In the image above, μ_1 and μ_2 correspond to the control angle for the 2-axis gimbal and $[\psi, \theta, \varphi]^T$ correspond to the roll, pitch and yaw of the vehicle respectively. With knowledge of the torques applied to the vehicle, the vehicle's angular rate can be determined with Euler's equation for rigid body dynamics:

$$ {}^{B}I \; \dot{\omega}^{B/N} = -\omega^{x} \; {}^{B}I \; \omega^{\mathrm{B/N}} + {}^{B}\tau $$

In the above equation, $\omega^{\mathrm{B/N}}$ corresponds to the angular velocity of the body with respect to an inertial reference frame and ω^{x} is the skew-symmetric matrix for $\omega^{\mathrm{B/N}}$. To further simplify the model, the vehicle is assumed to be a disk. This reduces the assumed inertial properties of the vehicle to only moments of inertia and no products of inertia.

$$ {}^{B}I = \begin{pmatrix} I_t & 0 & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_s \end{pmatrix} \qquad \begin{pmatrix} \ddot{\theta} \\ \ddot{\varphi} \end{pmatrix} = \begin{pmatrix} \frac{Tl}{I_s}\sin(\mu_1) \\ \frac{Tl}{I_s}\sin(\mu_2) \end{pmatrix} $$

With the assumption that angular rate ω is small, the gyroscopic term $\omega^{x}\ {}^{B}I\ \omega^{\mathrm{B/N}}$ is ignored. Thus, the dynamics of the vehicle are decoupled and control for pitch can be acquired independently of control for yaw. Roll is ignored, as this cannot be controlled with a 2-axis gimbal [1]. In the above equation, T corresponds to the magnitude of thrust produced by the motor, l corresponds to the distance between the gimbaling point and the vehicle's center of mass, and I_s is the vehicle's lateral moment of inertia. With these dynamics in mind, a physical design can be pursued.

2. Design

A thrust vector controller must perform two simultaneous tasks: determine the attitude of the vehicle and derive a control. This is executed on the RP2040 processor with bare-metal firmware – each of which deserve their own design sections. The mechanics of the vehicle dynamics follow from the mechanical implementation. Finally, the complete system can be modeled in Simulink.

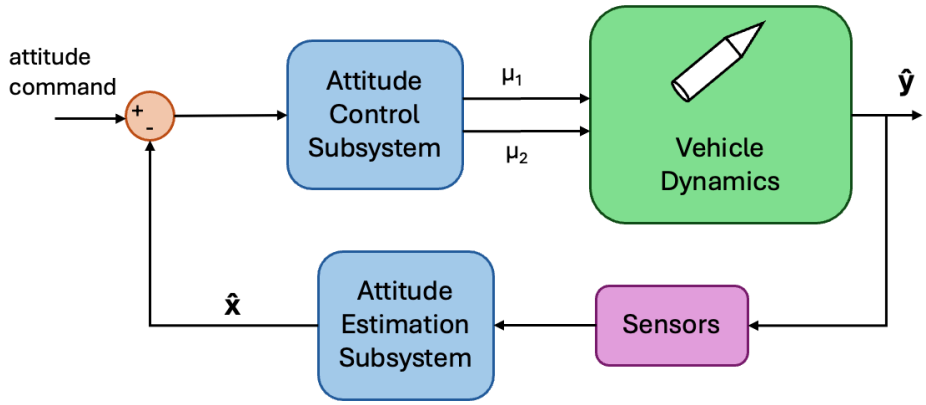


Figure 3: Top Level System Model

2.1. Mechanical Implementation

The physical design of this vehicle includes a 2-axis gimbal, a mounting structure for the avionics PCB, and the integration of these components with commercial off-the-shelf model rocket components. The body of the vehicle is an 18″ long cardboard tube with a 3″ outer

diameter. This tube fits a 15″ polystyrene nosecone, though significant weight can be removed with the design of a custom nosecone.

The gimbal is designed as a nested structure. An outer (master) mount is fixed to the vehicle body. An inner (child) mount is constrained to the outer mount by a set of two M3 bolts, which allow it to rotate in pitch. A motor mount is fixed to the inner mount by another set of of M3 bolts, allowing the inner mount to rotate in yaw. The combination of these two degrees of freedom are what provide 2-axis gimbaling. A 9g SG90 servo is placed on each of the outer and inner mounts. Only the static case for servo strength is considered. Because the motor thrust is always orthogonal to its axis of rotation, no torque is delivered to the servos. As such, servo torque is not a constraint in their selection.

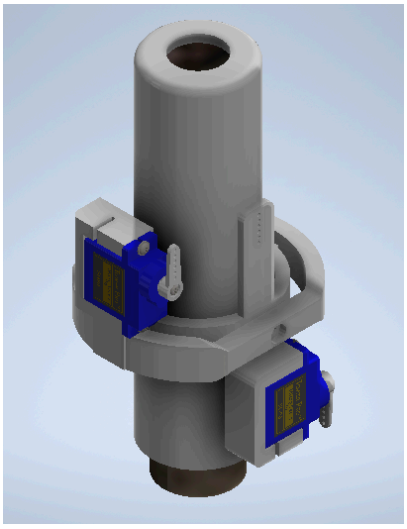


Figure 4: Gimbal CAD

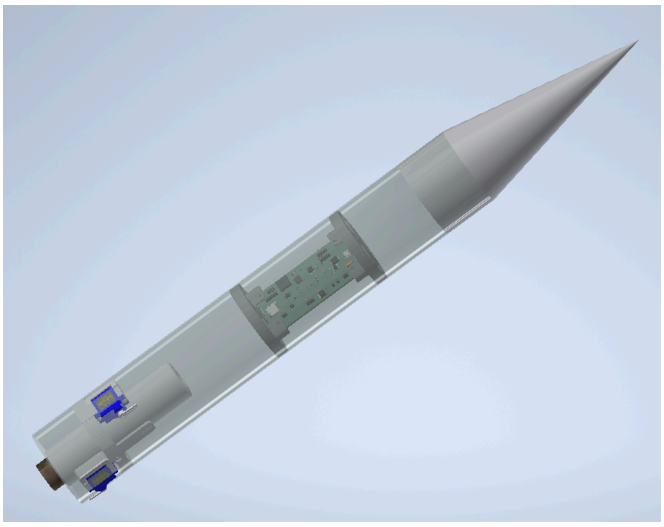


Figure 5: Vehicle CAD

Small metal linkages (not shown in the vehicle CAD) connect the servo arms to the gimbal hardware. Notice how the servo arm has multiple mounting holes. Also notice that the axis of rotation of the servo is not level with the axis of rotation of the gimbal. Thus, the placement of these linkages introduces a unique 4-bar linkage problem with an interesting set of tradeoffs and constraints. First, the length of the linkage is predetermined such that a 0° servo position corresponds to a 0° gimbal deflection. Additionally, the linkage mounting must be selected to allow for full range (albeit not perfectly 1:1) of the gimbal – in this case, ±12°. Finally, minimizing the distance x from the servo rotation point maximizes servo strength in the static case. For both the outer and inner mounts, the 3rd hole from the servo axis is selected.

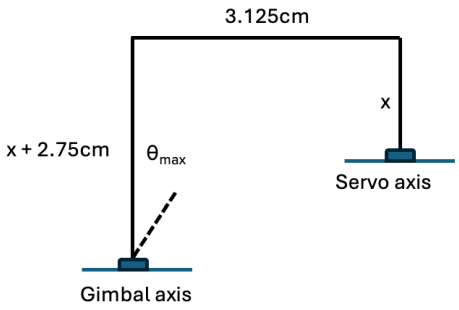


Figure 6: Four Bar Linkage

The PCB mounting hardware is designed to rigidly mount the PCB to the body of the vehicle, while maintaining access to the USB port on the top and the SD card slot on the bottom. The PCB is held between two tabs at all four corners with M2 bolds and a washer stackup. Both the gimbal and PCB mounting hardware can be held to the body with external screws. An access panel can be cut out from the body and held down with its own set of screws.

2.2. Electrical Implementation

The vehicle's electrical system consists of both an avionics package a power distribution system. The avionics must derive an estimate of the vehicle's attitude and command angles to the gimbal servo, and the power system must deliver power to the gimbal servos and avionics.

The avionics includes the following:

- A switching regulator to convert energy from a 11.1V LiPo source to 5V for the servo.
- A linear regulator to supply V_{DD} for the microcontroller.
- An inertial measurement unit (IMU) for attitude estimation
- Data logging functionality

The RP2040 microcontroller is chosen because of its wealth of documentation and ease of use. The RP2040 has a dual-core architecture, has sufficient RAM, is powerful enough in an unconstrained power budget, and has sufficient I/O.

The above requirements motivate the design of a custom PCB. This PCB is designed with the RP2040 as its focal feature. Peripherals include the BNO055 9-DoF IMU (which provides accelerometer, gyroscope and magnetometer readings), a BMP388 barometer, a USB jack, a microSD card slot, a buzzer and RGB LED for debugging and state identification on the test bench and launch pad, an auxiliary relay, a 5V DC/DC and LDO power regulation setup, and optional 32 MB data flash. The optional flash allows for an intermediate collection of data, until the vehicle is recovered and data can be written to the SD card. The RP2040 does not have internal flash, so an external chip is required to store the program image. Servo connections, relay connections and a battery input are all mapped to 0.1″ pitch male headers, along with test points for 5V, 3V3, I2C, SPI and the RP2040 SWD port.

If not powered over USB, the board can accept DC voltages within a 10.8 V to 19.8 V range. The TPS5430 buck regulator produces a 5V output voltage with a 30mV ripple, and can be loaded up to 3A. This output rail is OR'ed with the USB 5V rail, allowing either source to power the rest of the board independently. The NCP1117 LDO further regulates power to 3V3 for the microcontroller. This particular component is recommended in the Raspberry Pi Hardware Design Manual [2].

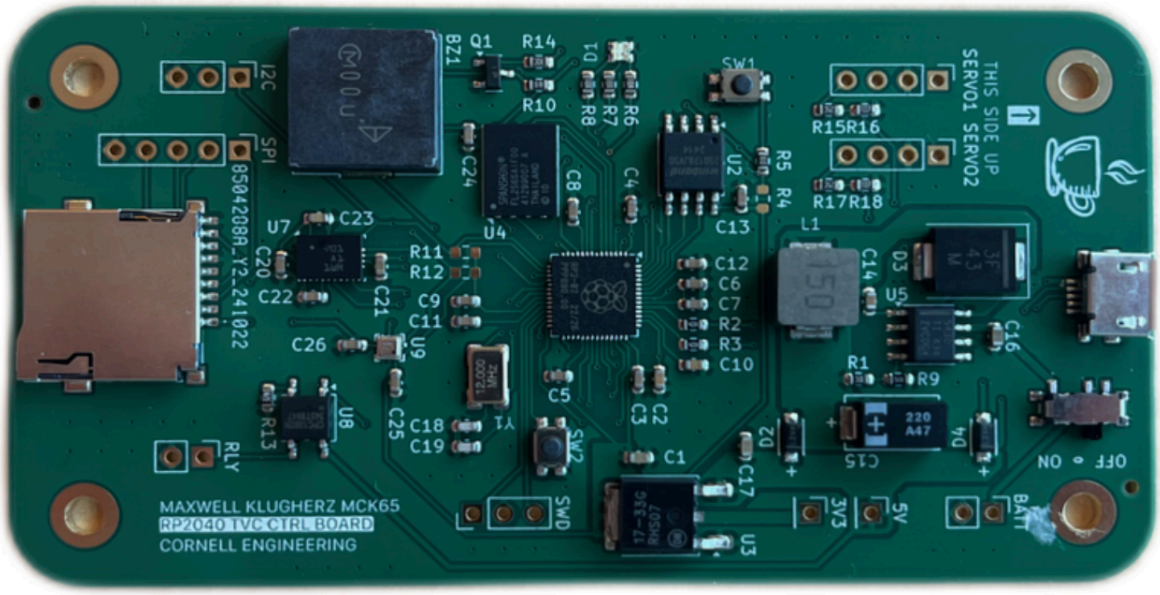


Figure 7: Printed Circuit Board

This board was designed in KiCAD. High current paths and ground pours are made with polygons. Almost all other traces are 0.2mm wide. While it is only a 2 layer board and is not overly complex, it would benefit from having 4 layers; high speed traces cannot currently meet characteristic impedance requirements because of the lack of a consistent ground plane.

This board was designed to the requirements of JLCPCB, the manufacturer of choice. They also populated the board, so components were selected from JLC's inventory. Only three components were left unpopulated: a set of pull-up resistors for I2C and a pull-up resistor for the program flash.

2.3. Firmware Implementation

The C-SDK was leveraged for all RP2040 programs. This allowed for easy interfacing with standard C libraries, APIs for microcontroller-specific hardware, and communication peripherals. Pictured below is a map of the RP2040 interfaces:

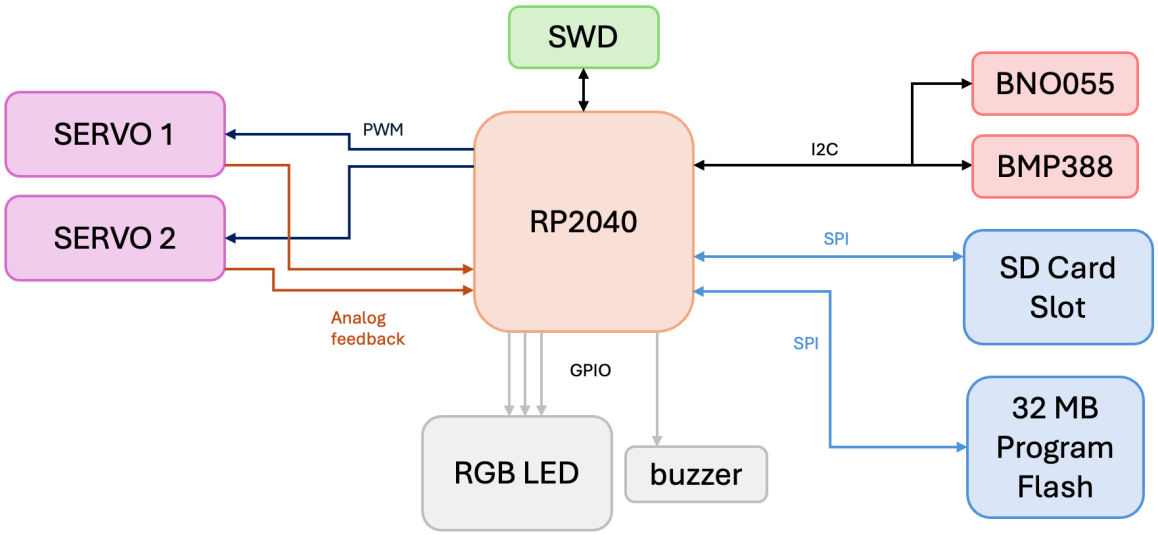


Figure 8: RP2040 Interface Map

To interface with an SD card over SPI, the *no-OS-FatFS-SD-SDIO-SPI-RPi-Pico* library by carlk3 [3] is used. This library abstracts all FAT-32 file system-level memory interaction away, leaving just `f_open`, `f_printf`, and `f_close` function calls (among others). This makes for data logging as simple as a stdio `printf` call.

The BNO055 IMU is interfaced with over I2C. Its x, y, and z axes can be remapped to the desired coordinate orientation in the vehicle. It is configured to report lateral acceleration in m/s^2, angular rates in rad/s, and magnetic field strength in μT. Each measurement for each axis is stored in 2 adjacent registers. Because all of these registers are stored in a contiguous address space, a single read request of 18 bytes can be made at once. From there, accel, gyro and magnetometer data can be directed from the 18 byte buffer into their respective data registers.

A complementary filter was implemented to validate the axis mappings and unit configurations of the IMU. This filter follows the form for each of pitch and yaw:

$$\theta = \alpha \cdot \theta_{gyro} + (1 - \alpha) \cdot \theta_{accel}$$

where θ_{accel} is the ratio of a transverse accelerometer reading to the the vertical accelerometer reading and θ_{gyro} is the time integral of the angular rate over the sampling interval. Note that this method of deriving an attitude has not been validated for use in TVC and is likely insufficient. Also note that ratios of acceleration only work for the unloaded case when the only force acting upon the vehicle is gravity [4].

Due to necessary precision in the gimbal actuation, care was taken with respect to the servo pulse width modulation (PWM) signals. The requirements for the servo were set as 0.1° granularity between +90° and −90° arm positions. While documentation for 9g servo PWM inputs varies, most datasheets indicate that the servos require a 50 Hz pulse train with pulses of

1-2 ms (where a pulse length of 1 ms corresponds to a −90° arm angle and a pulse length of 2 ms corresponds to a +90° arm angle):

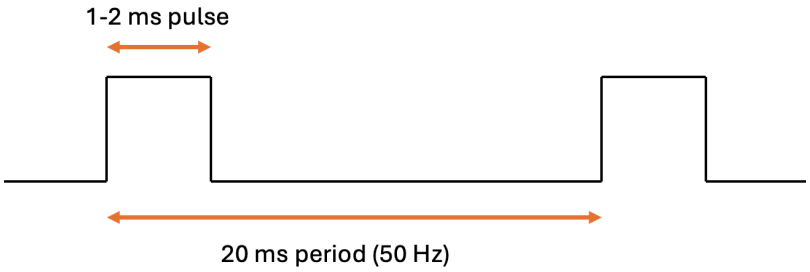


Figure 9: PWM Pulse Specification

The RP2040 PWM hardware operates by continuously incrementing a “counter compare” register at a divided frequency. In free running mode, this register wraps to 0 when equal to the value stored in a wrap register. When counter compare is less then a desired level, the output is high. Otherwise, the output is low [5]. With these requirements, the PWM hardware configurations can be derived as follows:

$$\frac{180° \text{ angle range}}{0.1° \text{ precision}} = 1800 \text{ "counter compares" per ms}$$

$$1800 \frac{\text{cc}}{\text{ms}} \cdot 20 \text{ ms} = 36000 \text{ wrap value}$$

$$\frac{f_{\text{sys}}}{f_{\text{PWM}}} = \frac{125 \text{ MHz}}{50 \text{ Hz}} = 36000 \cdot \left(\text{DIV\_INT} + \frac{\text{DIV\_FRAC}}{16}\right)$$

These characteristics are configured by instantiating a `pwm_config` struct and setting `DIV_INT`, `DIV_FRAC` and `TOP` to the specified values above. With these in place, the servo angle need only be adjusted by changing the value of the `LEVEL` register.

2.4. Simulation Implementation

Given that this vehicle may move at high speed and contains a propellant that is ignited at launch, it is in everyone’s best interest if this system is first simulated, as to ensure safety and mission success. MATLAB Simulink is an effective means for simulating control algorithm design, vehicle dyanmics, and attitude estimation. A rendering of the top-level model is pictured below:

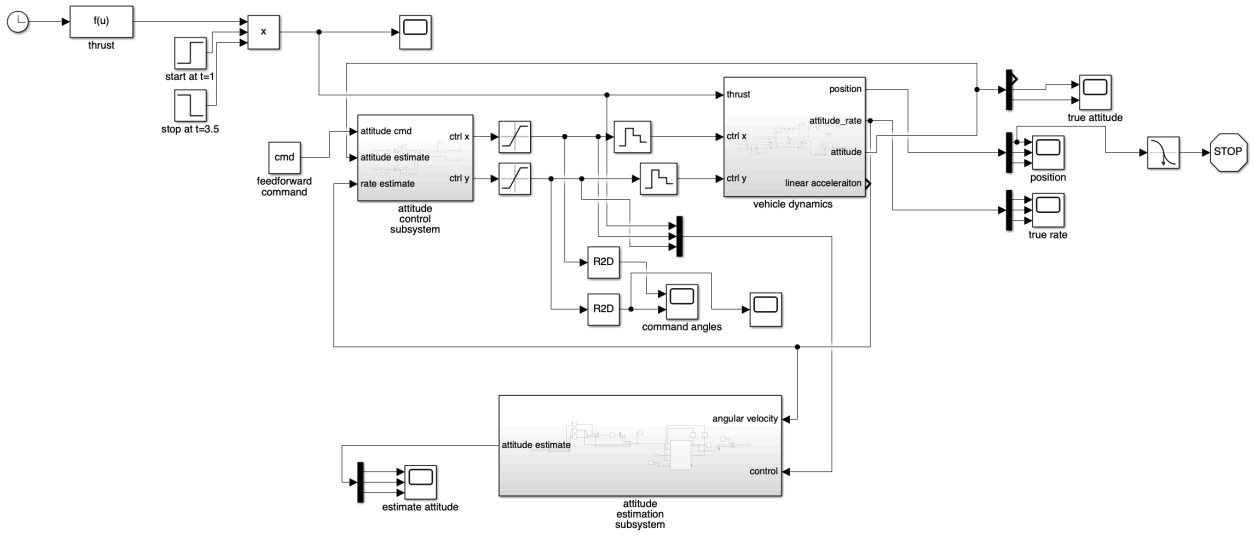


Figure 10: Simulink Top-Level

The top left block contains the attitude control subsystem. The top right block contains vehicle dynamics. The bottom block contains an attitude estimation subsystem. Note that the estimation block is incomplete, and as such, does not complete the loop. Instead, angular rates and attitudes are fed directly from the output of the dynamics block into the control block.

In the top-level, a thrust profile is enabled between t = 1 second and t = 3.5 seconds (which is taken as a function of exponentials to best model the thrust characteristics of an Estes F15 motor [6]). The simulations only last 5 seconds, but an additional simulation stop block is included for when the vehicle hits the ground. Between the attitude control subsystem and the vehicle dynamics, command angles μ_1 and μ_2 are saturated to model $|\mu_{\max}|$ and are held under zero-order holds to model the 100 Hz interrupt-based control frequency.

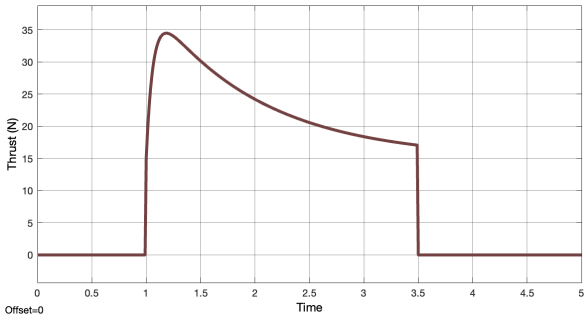


Figure 11: Thrust Profile

The attitude control subsystem receives an attitude reference and both an estimated attitude and angular rate. Errors in both attitude and rate are calculated in both pitch and yaw, and these errors are used in a simple PD controller for command angles, due to its global stability. Note that because angular rate bias is excluded, the integral term from a PID controller can be discarded, i.e., there is no steady-state error to account for in this model. This controller is designed with Kp = 0.5 and Kd = 0.2.

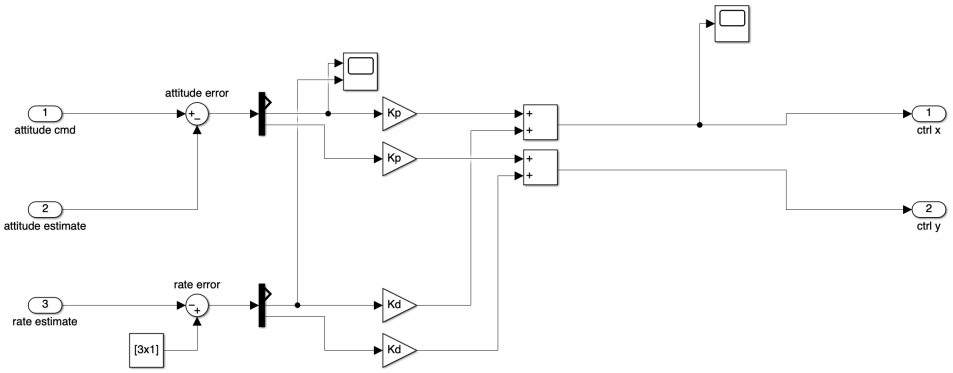


Figure 12: Attitude Control Subsystem

The vehicle dynamics block computes angular acceleration in both pitch and yaw by dividing torque along each axis $Tl\sin(\mu_i)$ by the lateral moments of inertia I_s. Once again, toll is ignored. Angular accelerations are integrated to derive angular velocities, which are integrated once more to derive an attitude. An attempt is made to derive the vehicle's position in an internal reference frame, but the use of the 3x3 rotation matrix block undesirably maps rotations around the body's x axis to rotations around the body's y axis. Also note that system noise is modeled with the addition of a gaussian noise generator for angular velocity around each axis.

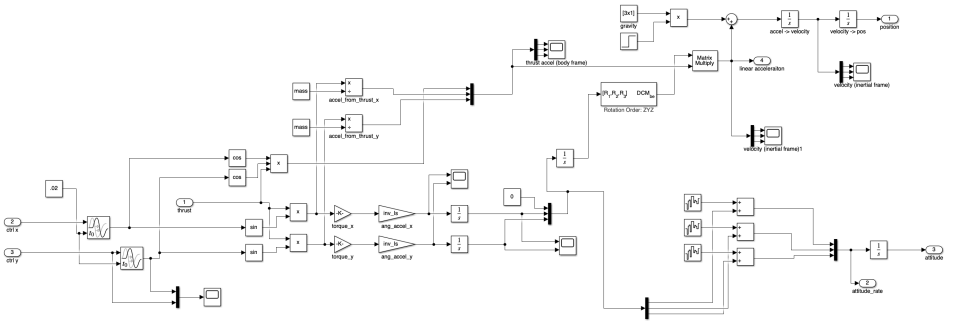


Figure 13: Vehicle Dynamics

For attitude estimation, an Extended Kalman Filter (EKF) is modeled. The EKF is designed to couple noisy and biased sensor readings with the linearized version of the system's dynamics. The EKF is best explained by Lefferts, Markley and Shuster [7]. This particular receives a noisy measurement y for attitude and angular rate, and and propagates an estimation $\hat{x}$ for the control inputs.

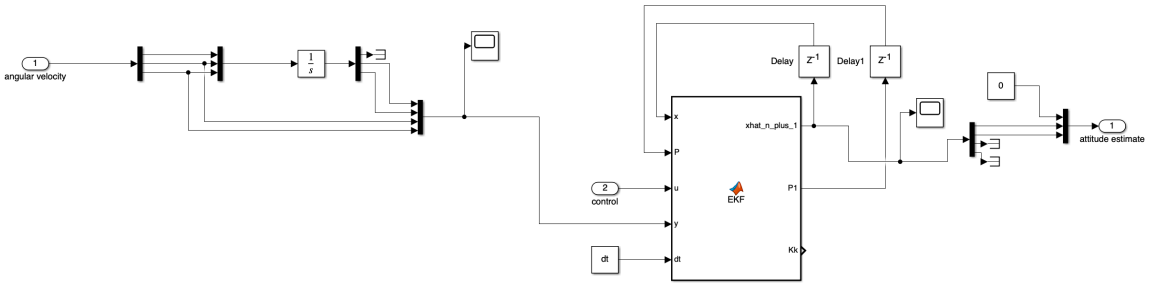


Figure 14: Attitude Estimation Subsystem

This filter relies on a linear state-space model $\dot{x} = Ax + Bu$ with the small-angle approximation for μ_1 and μ_2:

$$\frac{d}{dx}\begin{pmatrix}\theta\\ \varphi\\ \dot{\theta}\\ \dot{\varphi}\end{pmatrix} = \begin{pmatrix}0&0&1&0\\0&0&0&1\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}\theta\\ \varphi\\ \dot{\theta}\\ \dot{\varphi}\end{pmatrix} + \begin{pmatrix}0&0\\0&0\\\frac{-Tl}{I_s}&0\\0&\frac{-Tl}{I_s}\end{pmatrix}\begin{pmatrix}\mu_1\\ \mu_2\end{pmatrix}$$

This state-space representation is associated with an observation $y = Cx$ (assuming no direct feedthrough). Although a true estimator would receive data directly from an on-board IMU, which contains an accelerometer, gyroscope and magnetometer which may only report angular rates, for model simplicity, we assume an identity observation matrix:

$$y = \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}x$$

Unlike the linear state-space model, the EKF operates at discrete-time intervals. For each interval, the filter begins by making an approximation for an estimated next step with the original nonlinear model dynamics. Simultaneously, process covariance P is propagated with the Jacobian of the nonlinear state transition matrix and some process noise Q. This is called the propagation step. In this step, the process covariance should increase:

$$x_{n+1} = f(x_n, u_n) = \begin{pmatrix}\theta + \triangle t\dot{\theta}\\ \varphi + \triangle t\dot{\varphi}\\ \dot{\theta} - \triangle t\frac{Tl}{I_s}\sin(\mu_1)\\ \dot{\varphi} - \triangle t\frac{Tl}{I_s}\sin(\mu_2)\end{pmatrix} \qquad F = \frac{df(x_n,u_n)}{dx} = \begin{pmatrix}1&0&\triangle t&0\\0&1&0&\triangle t\\0&0&1&0\\0&0&0&1\end{pmatrix}$$

$$\hat{x}_{n+1}^{-} = f(x_n, u_n)$$

$$P_{n+1}^{-} = F \cdot P_n \cdot F^T + Q$$

The prediction step is followed by the update step, by which the Kalman gain is calculated, x is propagated with sensor readings, and process covariance is updated once again (where is the sensor noise matrix):

$$K = P_{n+1}^{-} \cdot C^T \cdot \left[C \cdot P_{n+1}^{-} \cdot C^T + R\right]^{-1}$$

$$\hat{x}_{n+1}^{+} = \hat{x}_{n+1}^{-} + K \cdot \left(y - C\hat{x}_{n+1}^{-}\right)$$

$$P_{n+1}^{+} = (I - K \cdot C) \cdot P_{n+1}^{-}$$

In the model, the outputs P_{n+1}^{+} and $\hat{x}_{n+1}^{+}$ are held through time delay blocks of length $\triangle t$ to follow the discrete-time nature of the filter.

3. Testing

This section will serve as a recollection of mechanical, hardware, firmware and simulation testing. Because the past year of work is scattered, it is best to divide testing into each category. Some testing was verified on a protoboard that integrated development boards for each of the Raspberry Pi Pico, the BNO055 and an SD card slot.

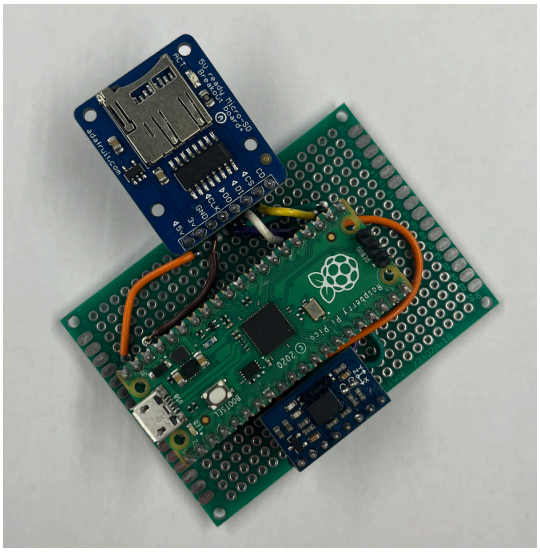


Figure 15: Raspberry Pi Pico Protoboard

3.1. Mechanical Testing

Beyond rapid prototyping, fitment and sizing of printed gimbal parts, minimal work was done to verify the strength and performance of the servo armature. Ultimately, the strength and stiffness of the gimbal can best be verified with a hot-fire, in which the gimbal assembly is rigidly mounted to a test fixture. In this test, gimbal angles are swept as a motor is ignited. Special 9g servos that break-out an analog potentiometer feedback can be used to characterize how vibe is distributed onto the gimbal mechanism.

A particular point of concern is the delay between a commanded angle and the angular motion of the servo. Thus, a test was performed to better characterized this delay. For each test, the gimbal command angle was changed by adjusting the PWM level, an ADC sampled the servo feedback pin for 1000 subsequent 1ms intervals, and the data was logged over UART.

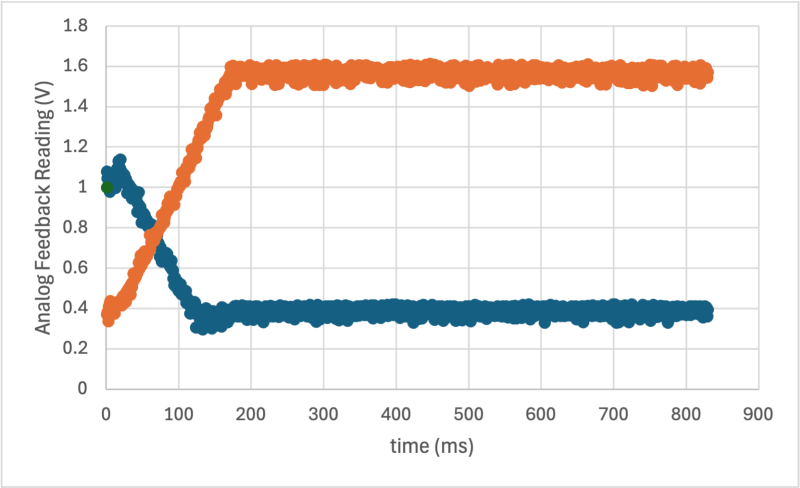


Figure 16: Servo Feedback Delay

In the above figure, the orange plot corresponds to a change in command angle from −90° to +90°, whereas the blue plot corresponds in a change in angle from 0° to −90°. For the 180° sweep, it took approximately 180ms for the servo to settle. For the 90° sweep, it took the servo approximately 120ms. In reality, changes in servo angle request would be much smaller in magnitude, but some linear delay should be factored into the model.

3.2. Hardware Testing

Hardware testing began once the PCB arrived from JLCPCB. Quick continuity tests for traces and discontinuity tests between power and ground traces were followed by unsuccessful attempts to flash the RP2040 over USB. This inability to leverage the on-chip RP2040 USB driver introduced new challenges. One explanation could be that the differential impedance for the USB_D+ and USB_D- traces does not meet the USB specification. These traces were still only 0.2mm wide and had an impedance of 220Ω – far from the 90Ω requirement. Alternatively, the external 12 MHz crystal oscillator is not the exact component that Raspberry Pi recommends in their hardware design guide [2] (the original oscillator was not available for placement by JLCPCB).

An alternative method was used to flash the RP2040. The microcontroller could be flashed over SWD with a debugger and openOCD. The Raspberry Pi Debug Probe allowed was a useful tool because it integrated a SWD port with a UART serial probe. This enables flashing code to the microcontroller, but it requires that the board receive some alternative power source, like a battery or a DC power supply.

3.3. Firmware Testing

Various features of the firmware were validated on-bench. After the ability to flash to the microcontroller was confirmed, each peripheral was validated in isolation.

Some trouble was encounterd with discovering the BNO055 on the I2C bus. After failed attempts with NACKs on the I2C line, a more methodical approach was used: the bus_scan program. The BNO055 I2C address can be configured one of two values by pulling one of its pins high or low. To verify that this address followed the design, the Raspberry Pi C-SDK bus_scan example program was leveraged. This allowed for a complete understanding of devices on the bus. In the figure below, the 0x28 address corresponds to the BNO055 IMU and the 0x77 address corresponds to the BMP388 barometric pressure sensor.

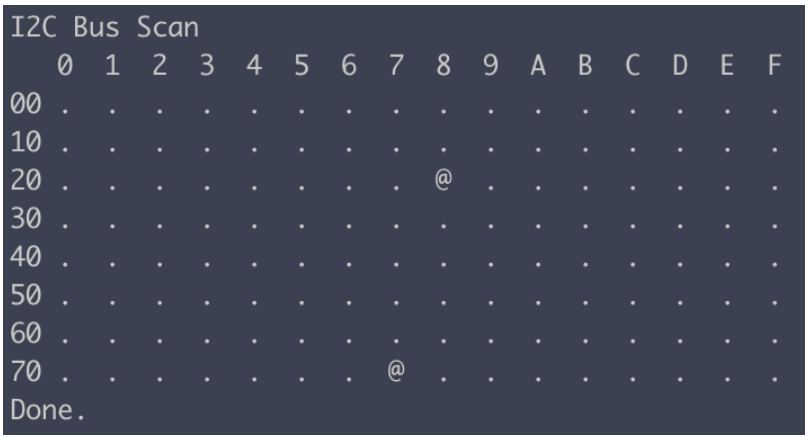


Figure 17: I2C Bus Scan

More troubleshooting was encountered with the SD card firmware. Notably, some SD card slot breakout boards are designed for 5V logic levels, and either contain simple voltage dividers or level shifters. These boards in particular cannot work with the 3V3-level RP2040.

When using openocd to flash the RP2040, a 5-10ms sleep is required to prevent code on core 1 to halt after a couple seconds. The cause for this is unknown. This was discovered after a core 1 “blink” program would stop blinking.

4. Results

After managing to troubleshoot some of the interfaces on the PCB, the RP2040 successfully performed all of its critical functions: read measurements from the BNO055 IMU over I2C, derive control angles for each servo, command those angles over PWM, and log data to the SD card slot.

The PD controller is effective in stabilizing the vehicle’s attitude for any given initial condition. Pictured below is a plot of pitch angle for a set of random initial conditions. All trajectories settle within the allotted thrust time (before t = 3.5 seconds). Also pictured is the gimbal command angle. Notice the oscillation in the command angle – this is an effect of the zero-order holds. The controller “lags” behind the dynamics, because the dynamics are continuous and the controls occur at 100 Hz. This “lag” is what produces the oscillation.

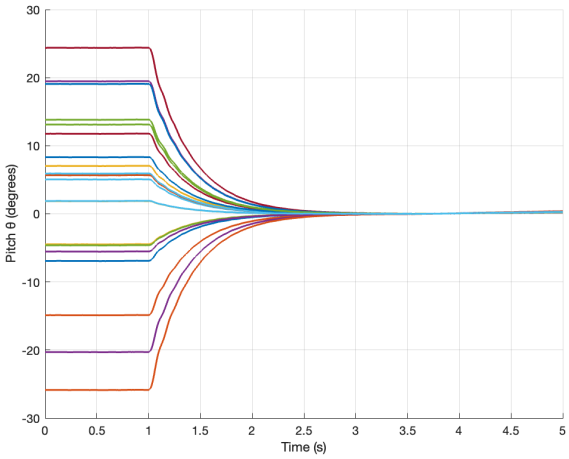


Figure 18: Pitch Settling for Random Initial Conditions

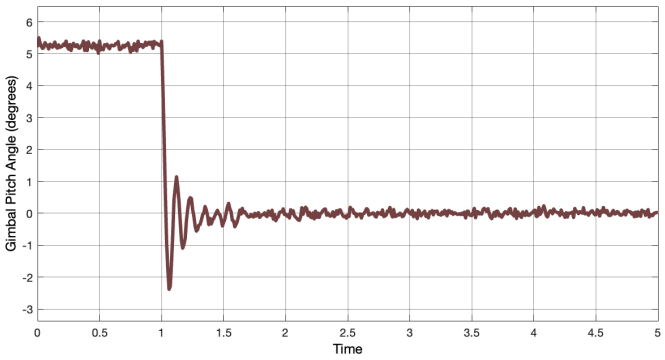


Figure 19: Gimbal Command Angle

The Simulink model makes many assumptions that likely do not hold in reality. The most notable are as follows:

- The mass properties of the vehicle do not lend to a perfectly diagonal inertia matrix. As such, the vehicle can not be modeled as a disk.
- The MEMS gyroscope contains some amount of bias that when integrated, produces significant (and increasing) errors in a propagated attitude [8]. The current estimator design does not account for these biases.
- The magnetometer electric field readings might be mildly corrupted by magnetic features on the PCB, like the switching regulator circuitry.
- More torques are be acting on the vehicle than just thrust. Most notably, aerodynamic drag can produce torques that destabilize the vehicle's flight trajectory.
- The mechanical assembly of the vehicle and configuration of servos can result in a misalignment between desired control angles and actual control angles.

Nevertheless, it is also likely that these realities can be ignored in the case of a 2.5 second thrust profile. For a longer duration flight, like that of the AeroTech G8ST, which has a thrust duration of approximately 20 seconds [9], model accuracy bears much more significance.

This project is necessarily unfinished; further work could expand on nearly every aspect. The gimbal could be redesigned for more stability or even be designed to become a machined component. Another revision of the PCB can be ordered with fixed USB_D+ and USB_D- impedances. The board can also be integrated with a GPS receiver for more accurate position sensing, or a wireless transceiver for real-time telemetry. Firmware can be written to leverage the on-board data flash chip. A quick program can be written to play the Cornell Alma Mater with the on-board buzzer. The PD controller can be replaced with a more robust controller, like a linear quadratic regulator (LQR), or even model predictive control (MPC) – which would be particularly unique as real-time quadratic programming on a microcontroller. The EKF can be

further evaluated and verified. The Simulink model can be expanded to incorporate fixes to the aforementioned assumptions.

All of these ideas should not distract from the most fundamental next step: the vehicle *must* fly.

5. Bibliography

[1] P. D. Santos and P. Oliveira, “Thrust vector control and state estimation architecture for low-cost small-scale launchers.” [Online]. Available: https://doi.org/10.48550/arXiv.2303.16983

[2] “Hardware design with RP2040.” [Online]. Available: https://datasheets.raspberrypi.com/rp2040/hardware-design-with-rp2040.pdf

[3] C. J. K. III, “no-OS-FatFS-SD-SDIO-SPI-RPi-Pico.” [Online]. Available: https://github.com/carlk3/no-OS-FatFS-SD-SDIO-SPI-RPi-Pico

[4] V. H. Adams, “Complementary Filters.” [Online]. Available: https://vanhunteradams.com/Pico/ReactionWheel/Complementary_Filters.html

[5] “RP2040 Datasheet.” [Online]. Available: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

[6] “Estes F15 ThrustCurve.” [Online]. Available: https://www.thrustcurve.org/motors/Estes/F15/

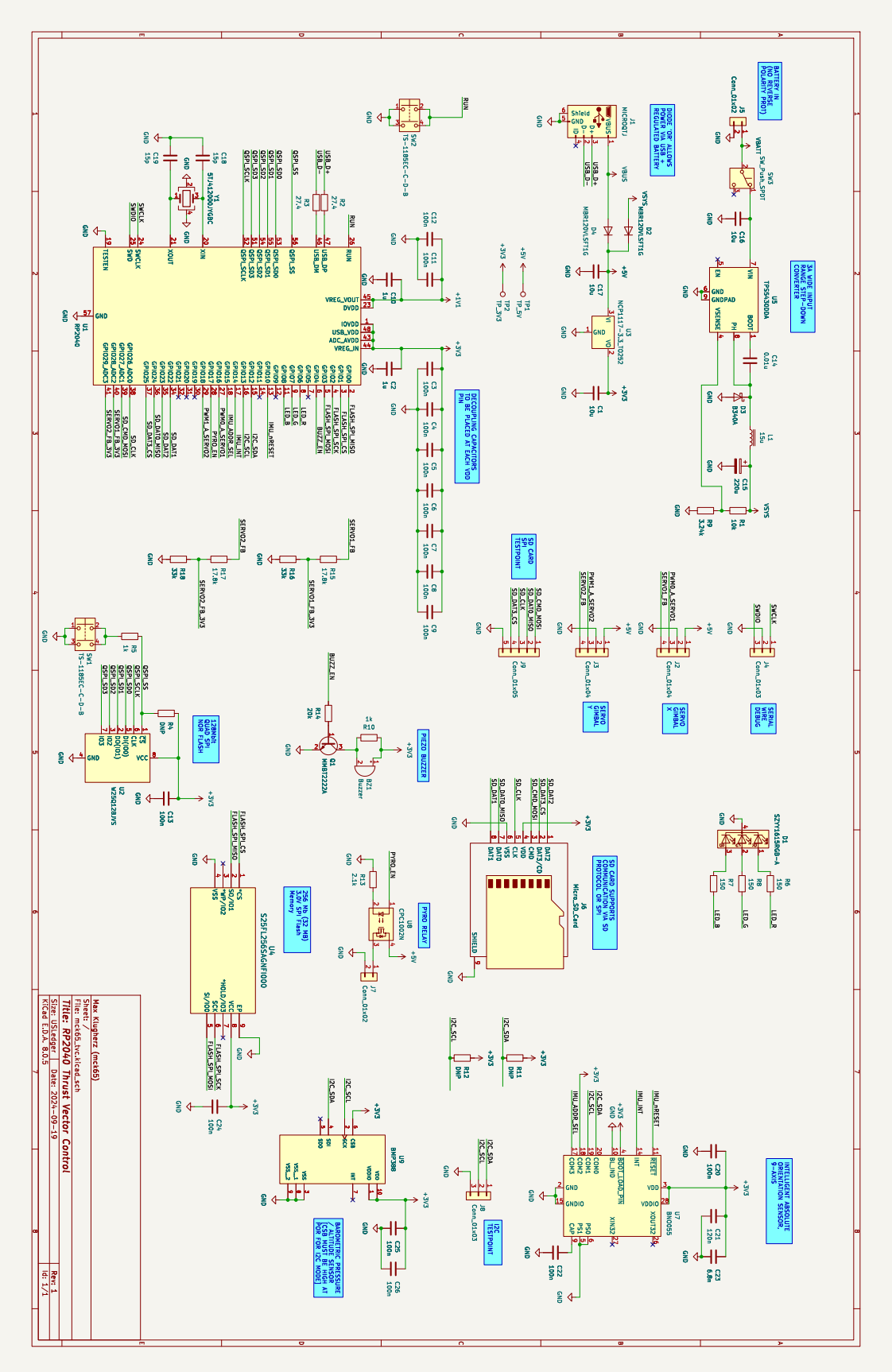
[7] E. Lefferts, L. Markley, and M. Shuster, “Kalman filtering for spacecraft attitude estimation,” *Journal of Guidance, Control, and Dynamics*, vol. 5, 1982, doi: 10.2514/3.56190.

[8] S. B. Farahan, Machado, F. Gomes, and M., “9-DOF IMU-Based attitude and heading estimation using an extended kalman filter with bias consideration,” *Sensors*, vol. 22, no. 9, 2022, doi: 10.3390/s22093416.

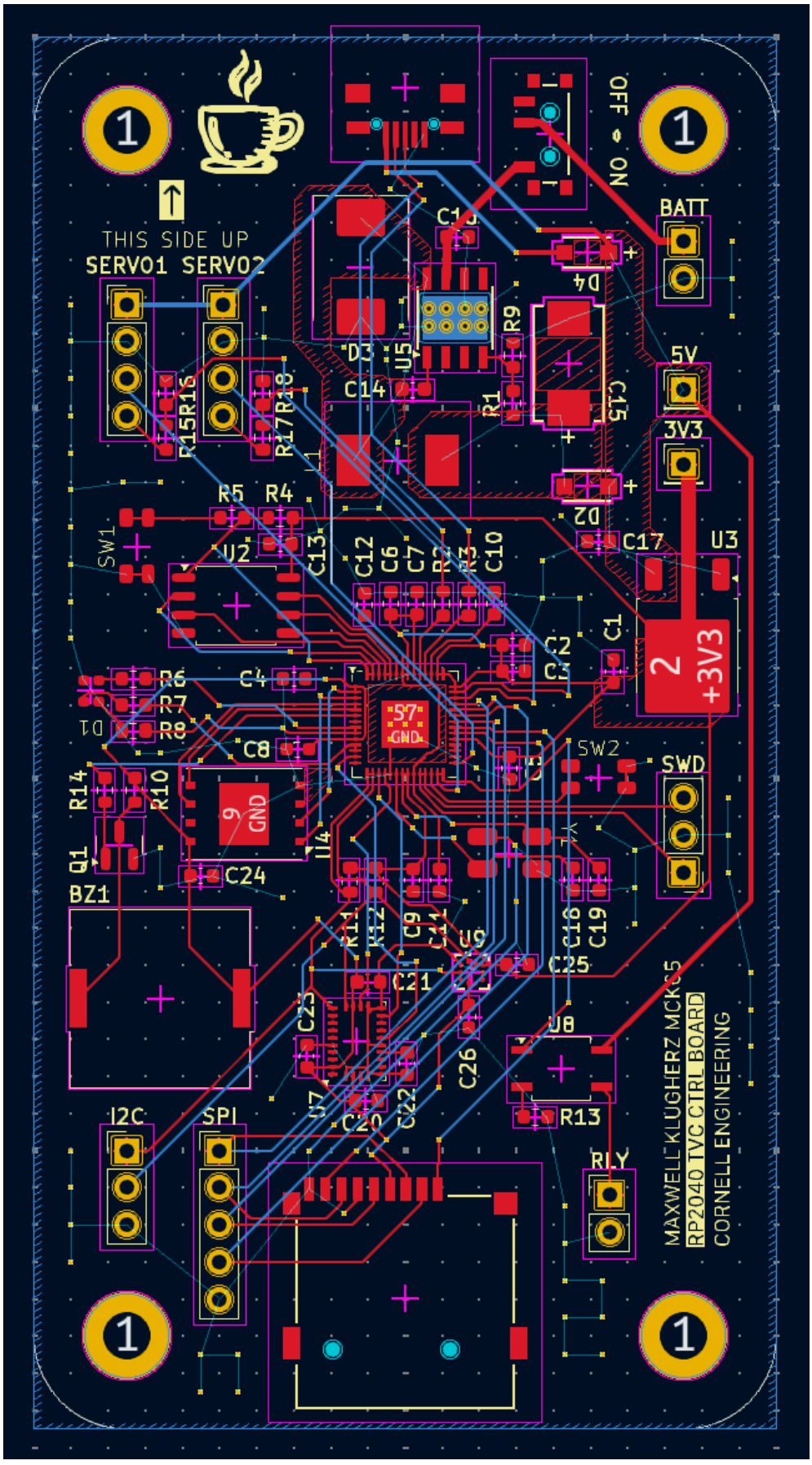
[9] “AeroTech G8ST ThrustCurve.” [Online]. Available: https://www.thrustcurve.org/motors/AeroTech/G8ST/

6. Appendix

6.1. PCB Schematic



6.2. PCB Layout



6.3. RP2040 Firmware

6.3.1. main.c

```
/**
 * Max Klugherz (mck65@cornell.edu)
 * MEng Design Project: TVC
 * Advisor: Hunter Adams (vha3@cornell.edu)
 */


#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include "pico/stdlib.h"
#include "bno055.h"
#include "attitude.h"
#include "tvc_pwm.h"
#include "pico/multicore.h"
#include "hardware/i2c.h"
#include "hardware/irq.h"

// FATFS
#include "hw_config.h"
#include "f_util.h"
#include "ff.h"

// LED Blink Time Delay (ms)
#define BLINK_DELAY 500

// The LED is connected to pins 678
#define R_LED_PIN 6
#define G_LED_PIN 7
#define B_LED_PIN 8

bool led_vals[3] = {0, 0, 0};
int led_pins[3] = {6, 7, 8};

// constants
#define PI 3.14159
#define GIMBAL_MAX 25.0

// accel lowpass
#define alpha 0.3

// PROGRAM GLOBALS
static float accel_fl[3] = {0,0,0};
static float gyro_fl[3] = {0,0,0};
```

```
static float mag_fl[3] = {0,0,0};
static lambda complementary_angles = {.phi = 0, .theta = 0, .psi = 0};

static float Kp = 0.5;
static float Kd = 0.2;

// BNO055 return fields
int16_t accel[3];
int16_t gyro[3];
int16_t mag[3];

// PROJECT GLOBALS - FAT FILESYSTEM
FIL fil;
int16_t accel[3];

int j = 0;
bool estimation_callback(__unused struct repeating_timer *t) {

    // bno055_read_status();
    bno055_read_raw(accel, gyro, mag);

    // accelerometer lowpass happens here
    // https://vanhunteradams.com/Pico/ReactionWheel/Digital_Lowpass_Filters.html
    // equivalent to RC with time constant of 16 samples

    // accel_fl[0] = accel_fl[0] + (accel_fl[0] - (float)(accel[0]))/16 / 100.0;
    // accel_fl[1] = accel_fl[1] + (accel_fl[1] - (float)(accel[1]))/16 / 100.0;
    // accel_fl[2] = accel_fl[2] + (accel_fl[2] - (float)(accel[2]))/16 / 100.0;
    accel_fl[0] = alpha * ((float)accel[0]) / 100.0 + (1 - alpha) * accel_fl[0];
    accel_fl[1] = alpha * ((float)accel[1]) / 100.0 + (1 - alpha) * accel_fl[1];
    accel_fl[2] = alpha * ((float)accel[2]) / 100.0 + (1 - alpha) * accel_fl[2];

    gyro_fl[0] = ((float)gyro[0]) / 900.0;
    gyro_fl[1] = ((float)gyro[1]) / 900.0;
    gyro_fl[2] = ((float)gyro[2]) / 900.0;

    // complementary filter --> get roll, pitch and yaw angles
    comp_step(&complementary_angles, accel_fl, gyro_fl);

    return true;
}

// angles and rates for control
float last_phi, this_phi, phi_rate;
float last_theta, this_theta, theta_rate;

float x_pos, y_pos;
```

```
bool control_callback(__unused struct repeating_timer *t) {

    // derive angles and rates in deg
    last_phi = this_phi;
    this_phi = complementary_angles.phi;
    phi_rate = (this_phi - last_phi) / 0.01; // 10ms estimation interval

    last_theta = this_theta;
    this_theta = complementary_angles.theta;
    theta_rate = (this_theta - last_theta) / 0.01;

    // PD Controller
    x_pos = (-1 * Kd * this_phi - Kp * phi_rate) * 180 / PI;
    y_pos = (-1 * Kd * this_theta - Kp * theta_rate) * 180 / PI;

    // boundary checks (no more than +/- 15deg)
    if(x_pos > 15.0) x_pos = GIMBAL_MAX;
    else if(x_pos < -15.0) x_pos = -1*GIMBAL_MAX;

    if(y_pos > 15.0) y_pos = GIMBAL_MAX;
    else if(y_pos < -15.0) y_pos = -1*GIMBAL_MAX;

    set_pwm_pos(0, x_pos);
    set_pwm_pos(1, y_pos);

    return true;
}

// CORE 1: LED BLINK
void core1_entry() {

    // initialize tvc
    tvc_pwm_init();

    // Initialize the pin(s)
    gpio_init(R_LED_PIN);
    gpio_init(G_LED_PIN);
    gpio_init(B_LED_PIN);

    // Configure the LED pin(s) as an output
    gpio_set_dir(R_LED_PIN, GPIO_OUT);
    gpio_set_dir(G_LED_PIN, GPIO_OUT);
    gpio_set_dir(B_LED_PIN, GPIO_OUT);

    // instantiate timer
    struct repeating_timer timer2;
```

```
    add_repeating_timer_ms(-100, control_callback, NULL, &timer2);

    int cnt = 0;
    // Loop
    while (1) {
        cnt += 1;
        // get random int
        int sel = (int)rand() % 3;

        led_vals[sel] = !led_vals[sel];
        gpio_put(led_pins[sel], led_vals[sel]);

        // printf("toggling pin %d\n", led_pins[sel]);
        // Sleep
        sleep_ms(BLINK_DELAY);
    }
}

// Main (runs on core 0)
int main() {

    sleep_ms(10);
    // initialize I/O
    stdio_init_all();

    // core 1 blink_entry
    multicore_launch_core1(&core1_entry);


    // See FatFs - Generic FAT Filesystem Module, "Application Interface",
    // http://elm-chan.org/fsw/ff/00index_e.html
    FATFS fs;
    FRESULT fr = f_mount(&fs, "", 1);
    while (FR_OK != fr) {
        panic("f_mount error: %s (%d)\n", FRESULT_str(fr), fr);
        sleep_ms(5000);
         fr = f_mount(&fs, "", 1);
    }

    // Open a file and write to it

    const char* const filename = "log_file.txt";
    fr = f_open(&fil, filename, FA_OPEN_APPEND | FA_WRITE);
    if (FR_OK != fr && FR_EXIST != fr) {
       while(1){
          panic("f_open(%s) error: %s (%d)\n", filename, FRESULT_str(fr), fr);
      }
```

```
    }


    // initialize i2c
    i2c_init(I2C_CHAN, I2C_BAUD_RATE) ;
    gpio_set_function(SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(SCL_PIN, GPIO_FUNC_I2C);
    gpio_set_pulls(SDA_PIN, 1, 0);
    gpio_set_pulls(SCL_PIN, 1, 0);


    // bno055 init
    bno055_reset();

    // instantiate timer
    struct repeating_timer timer;
    add_repeating_timer_ms(-10, estimation_callback, NULL, &timer);

    // stay here forever logging to  SD
    while(1) {
        sleep_ms(10);
        char buffer[100];
        sprintf(buffer, "Theta: %.5f, Phi: %.5f, x_pos: %.5f, y_pos: %.5f\n",
complementary_angles.theta, complementary_angles.phi, x_pos, y_pos);

        if (f_printf(&fil, buffer) < 0) {
            while(1){
                printf("f_printf failed\n");
            }
        }
    }

    bool cancelled = cancel_repeating_timer(&timer);
    printf("cancelled...%d\n", cancelled);
    sleep_ms(2000);

    // Close the file
    fr = f_close(&fil);
    if (FR_OK != fr) {
        printf("f_close error: %s (%d)\n", FRESULT_str(fr), fr);
    }

    // Unmount the SD card
    f_unmount("");

    return 0;
}
```

6.3.2. tvc_pwm.c

```c
/*
 * Max Klugherz (mck65@cornell.edu)
 * tvc_pwm.h
 */

// SET UP PIN ALLOCATIONS
// #define LED_PIN 25
#define SERVO1_PWM_PIN 16
#define SERVO2_PWM_PIN 18

// initialize PWM primitive
void tvc_pwm_init(void);

// set PWM position primitive
// takes slice and requested degree (prerequisite: between -90 and +90 deg)
// assumes channel A for now
void set_pwm_pos(int slice, float degree);
```

```c
/*
 * Max Klugherz (mck65@cornell.edu)
 * tvc_pwm.c
 */

#include "tvc_pwm.h"

#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/pwm.h"

void tvc_pwm_init(void) {
    // Initialize PWM pins
    gpio_set_function(SERVO1_PWM_PIN, GPIO_FUNC_PWM);
    gpio_set_function(SERVO2_PWM_PIN, GPIO_FUNC_PWM);

    // initialize config for both servo 1 and servo 2 PWMs
    pwm_config servo_config =  pwm_get_default_config();
    // set PWM clock divider: 69.4375 to arrive at 2.49975 MHz PWM freq
    pwm_config_set_clkdiv_int_frac(&servo_config, 69, 7);
    // set PWM wrap (TOP) to 36000 such that period is 50Hz
    pwm_config_set_wrap(&servo_config, 36000);
    // assign to slice 0 and slice 1 (without enabling)
    pwm_init(0, &servo_config, false);
    pwm_init(1, &servo_config, false);

    // servo 1 (slice 0, channel A): set position to -90deg by setting counter
```

```
compare to 2700
    pwm_set_chan_level(0, PWM_CHAN_A, 2700);
    // servo 1 (slice 1, channel A): set position to -90deg by setting counter
compare to 2700
    pwm_set_chan_level(1, PWM_CHAN_A, 2700);

    // enable both slices
    pwm_set_enabled(0, true);
    pwm_set_enabled(1, true);
}

void set_pwm_pos(int slice, float pos) {
   pwm_set_chan_level(slice, PWM_CHAN_A, (int)((pos + 90.0) * 20.0 + 900));
}
```

6.3.3. bno055.c

```
/**
 * Max Klugherz (mck65@cornell.edu)
 * bno055.h
 */
 #include <stdint.h>

#define BNO_ADDRESS 0x28

#define BNO_GPIO_ADDR_SEL_PIN 15
#define BNO_GPIO_nRESET_PIN 10

#define I2C_CHAN i2c0
#define SDA_PIN  12
#define SCL_PIN  13
#define I2C_BAUD_RATE 100000

void bno055_reset(void);
void bno055_calibrate(void);
void bno055_read_raw(int16_t accel[3], int16_t gyro[3], int16_t mag[3]);
void bno055_read_status(void);

/**
 * Max Klugherz (mck65@cornell.edu)
 * bno055.c
 */

 #include "bno055.h"
 #include "hardware/i2c.h"
 #include "pico/stdlib.h"
 #include <stdint.h>
 #include <stdio.h>
```

```c
void bno055_reset(void) {

  // set up i2c and stuff
  // upon startup, the bno055 enters config mode

  // ACC_Config
  // G range = 8 G
  // Bandwidth = 500 Hz
  // Operation mode = Normal
  uint8_t acc_cfg[] = {0x08, 0b00011010};
  i2c_write_blocking(I2C_CHAN, BNO_ADDRESS, acc_cfg, 2, false);

  // UNIT_SEL
  // accept acceleration units in m/s^2
  // accept angular rate units in rad/s
  uint8_t unit_cfg[] = {0x3B, 0x02};
  i2c_write_blocking(I2C_CHAN, BNO_ADDRESS, unit_cfg, 2, false);


  // AXIS REMAP
  // Z --> y
  // Y --> X
  // X --> Z
  uint8_t axis_cfg[] = {0x41, 0b00010010};
  i2c_write_blocking(I2C_CHAN, BNO_ADDRESS, axis_cfg, 2, false);

  uint8_t axis_sign_cfg[] = {0x42, 0b00000110};
  i2c_write_blocking(I2C_CHAN, BNO_ADDRESS, axis_sign_cfg, 2, false);

  // exit config mode
  // for now, only in non-fusion accel, magnetometer, gyro mode
  uint8_t mode[] = {0x3D, 0x07};
  i2c_write_blocking(I2C_CHAN, BNO_ADDRESS, mode, 2, false);

  printf("leaving BNO055 setup/reset ...\n");
}

void bno055_read_raw(int16_t accel[3], int16_t gyro[3], int16_t mag[3]) {
  // request uncompensated accel data
   uint8_t buffer[18];

  // the register address is automatically incremented
  // so more than one byte can be sequentially read out
  // { ACCEL_X_LSB, ... , ACCEL_Z_MSB, GYRO_X_LSB, ... , GYRO_Z_MSB,
MAG_X_LSB, ... , MAG_Z_MSB }
```

```
  uint8_t ACC_DATA_X_LSB = 0x08;
  int ret = i2c_write_blocking(I2C_CHAN, BNO_ADDRESS, &ACC_DATA_X_LSB, 1,
true); // keep control of master bus
  int ret2 = i2c_read_blocking(I2C_CHAN, BNO_ADDRESS, buffer, 18, false);  //
False - finished with bus

  // fill accel[]
  for(int i = 0; i < 3; i++){
    accel[i] = (buffer[2 * i + 1] << 8 | buffer[2 * i]);
    // accel[i] = accel[i] / 100;
  }

  // fill gyro[]
  const int gyro_buffer_offset = 12;
  for(int i = 0; i < 3; i++){
    gyro[i] = (buffer[2 * i + 1 + gyro_buffer_offset] << 8 | buffer[2 * i +
gyro_buffer_offset]);
  }

  // fill mag[]
  const int mag_buffer_offset = 6;
  for(int i = 0; i < 3; i++){
    mag[i] = (buffer[2 * i + 1 + mag_buffer_offset] << 8 | buffer[2 * i +
mag_buffer_offset]);
  }

  // printf("ax %d, ay %d, az %d\n", accel[0], accel[1], accel[2]);
  if(ret == PICO_ERROR_GENERIC) {
    printf("WRITE NO ACK \n");
  }
  else if(ret2 == PICO_ERROR_GENERIC){
    printf("READ NO ACK \n");
  }
  else {
    // printf("ax %d, ay %d, az %d\n", accel[0], accel[1], accel[2]);
    // printf("hex: %x, %x, %x, %x %x, %x\n", buffer[0], buffer[1], buffer[2],
buffer[3], buffer[4], buffer[5]);
  }

}

void bno055_read_status(void) {

  uint8_t status[1];

  uint8_t SYS_STATUS = 0x01;
  int ret = i2c_write_blocking(I2C_CHAN, BNO_ADDRESS, &SYS_STATUS, 1, true); //
```

```
keep control of master bus
  int ret2 = i2c_read_blocking(I2C_CHAN, BNO_ADDRESS, status, 1, false);  //
False - finished with bus


  if(ret == PICO_ERROR_GENERIC) {
    printf("WRITE NO ACK \n");
  }
  else if(ret2 == PICO_ERROR_GENERIC){
    printf("READ NO ACK \n");
  }
  else {
    printf("status: %x\n", status[0]);
  }
}
```

6.3.4. attitude.c

```
/**
 * Max Klugherz (mck65@cornell.edu)
 */

#include <math.h>

// constants
#define g (float) 9.81
#define oneeightyoverpi (float) 57.2957795131

// time interval in seconds (and milliseconds)
#define delta_t (float) 0.01
#define delta_t_ms 10

// complementary filter weights
#define comp_accel_wt (float) 0.01
#define comp_gyro_wt (float) 0.99


typedef struct {
    float phi;      // z_axis, roll, phi
    float theta;    // x_axis, pitch, theta
    float psi;      // y_axis, yaw, psi

} lambda;

// ======================================================================
// COMPLEMENTARY FILTER
// ======================================================================
```

```
// void comp_init(lambda *comp_angle);
void comp_step(lambda *comp_angle, float imu_accel[3], float imu_gyro[3]);

/**
 * Max Klugherz (mck65@cornell.edu)
 */

#include "attitude.h"
#include <stdio.h>

// returns complementary angle after one time step
// roll(phi), pitch(theta), yaw(psi) in radians
void comp_step(lambda *comp_angle, float imu_accel[3], float imu_gyro[3]) {
    // accel and gyro data accepted in [x, y, z] format
    // AKA [pitch, yaw, roll]

    // take small angle approximation
    // arctan(a) ~= a
    float accel_roll = imu_accel[1] / imu_accel[0]; // y/x
    float accel_pitch = imu_accel[1] / imu_accel[2]; // y/z
    float accel_yaw = imu_accel[0] / imu_accel[2]; // y/z

    float gyro_roll_delta = -1 * imu_gyro[2] * delta_t;
    float gyro_pitch_delta = imu_gyro[0] * delta_t;
    float gyro_yaw_delta = -1 * imu_gyro[1] * delta_t;

    // why is gyro int subtracted and accel int added?
    comp_angle->phi = (comp_angle->phi + gyro_roll_delta) * comp_gyro_wt; //
accel_roll * comp_accel_wt;
    comp_angle->theta = (comp_angle->theta + gyro_pitch_delta) * comp_gyro_wt +
accel_pitch * comp_accel_wt;
    comp_angle->psi = (comp_angle->psi + gyro_yaw_delta) * comp_gyro_wt +
accel_yaw * comp_accel_wt;
}
```

6.4. MATLAB Source Code

```
% TVC Model Simulation
% Max Klugherz mck65

cmd = [0 0 0].';
% vehicle params
gimbal_arm_len = 0.2; % meters

Is = 0.02838604638; % kg*m^2
It = 0.00054050543;
```

```
inv_Is = 1/Is;

I = [Is 0 0
     0 Is 0
     0 0 It];

mass = 0.85; % kg

att_init = [0 .174 .174].';

% gimbal params
mu_max = 12; % degrees
mu_min = -12;

% gains
Kp = .5;
Kd = .2;

dt = 0.010;
```