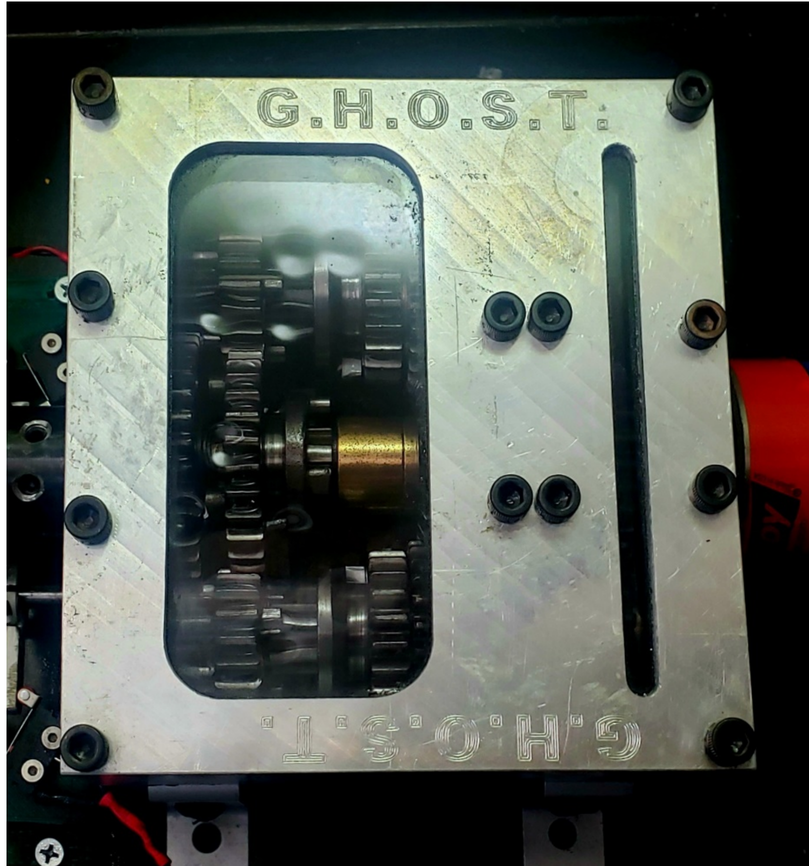
Project G.H.O.S.T

Final Design Report



Team Members:

Austin Conrad – auconrad@calpoly.edu

Zack Gordon – zagordon@calpoly.edu

Conan Estes – cwestes@calpoly.edu

Matt Morell – morell@calpoly.edu

Mechanical Engineering

California Polytechnic State University, San Luis Obispo

March 12, 2021

©2021 Austin Conrad, Zack Gordon, Conan Estes, Matt Morell

Abstract

Project G.H.O.S.T aimed to produce a model transmission with a control system for high performance/racing electric vehicles to utilize the full capability of the electric motor at higher speed applications. Current electric vehicles have a large amount of torque and use a fixed gear ratio that compromises between both high and low speeds with performance peaking around 65 miles per hour. A typical electric motor used in this application has approximately 1800 ft-lbs of torque up to 5,000 rpm. Past approximately 5,000 rpm, the torque rapidly drops off and so does the overall vehicle performance. Though transmissions exist that could have sufficient gear ratios, typical transmissions are not built to withstand torque figures of an electric motor. Even once implemented, transmissions have a second problem of high RPM gaps between gears due to the large gear ratios which results in a difficulty shifting quickly.

The current model design used the form factor of the Liberty's Gears 5-speed Equalizer Transmission (dual counter shaft) but with smaller, 125cc Honda dirt bike gears. The housing was designed to contain the three shafts, bearings, shifter forks and recombining gear for the model. The model motor was a DIY electric skateboard motor with over 4,000 RPM max and 1.9 N-m of torque which brought the entire model transmission and output flywheel up to max speed before shifting in under the desired 6 seconds. The transmission was controlled using the STM Nucleo L467RG microcontroller, programmed using C++ to better utilize the microcontroller's high processor speed of 80 MHz. Finally, the system was tested based upon the ability to hit the engineering requirements and tuned for shifting speed using model gains.

Table of Contents

Chapter 1 : Introduction 1
Chapter 2 : Background 2
Chapter 3 : Objectives 6
3.a Problem Statement 6
3.b Boundary Diagram 6
3.c Customer Wants and Needs Summary 7
3.d Engineering Specification Table 7
3.f Objectives Summary 8
Chapter 4 : Concept Design Development 9
4.a System Overview: 9
4.b Input Motor 9
4.c Key Transmission Components 10
4.d Flywheel 10
4.e User Interface 10
4.f Control System 10
4.h Controls Method 11
Chapter 5 : Final Design 12
5.a System Overview 12
5.b Input Motor & ESC Subassembly 12
5.c Transmission Subassembly 15
5.d Flywheel Subassembly 18
5.e Servo & Pushrod Subassembly 20
5.f Controller 22
5.g Daughter Board 23
5.h User Interface 24
5.i Software Design 25
5.j FMEA 27
Chapter 6 : Manufacturing 29
6.a Mechanical Manufacturing : Component Selection 29
6.b Mechanical Manufacturing : 3D Printing Prototypes 29
6.c Mechanical Manufacturing : Final Product Manufacturing 33

6.d Mechanical Manufacturing: Final Product Assembly 41
6.f Electrical Manufacturing 47
Chapter 7 : Design Verification 54
7.a Preliminary Mechanical Testing 54
7.b Preliminary Electrical Testing 56
7.c Preliminary Controls Testing 58
7.d Preliminary Full Shift Testing 61
7.e Final Full Speed Shift Testing 64
7.f Final Testing Results 65
Chapter 8 : Project Management 67
Chapter 9 : Conclusion 69
Chapter 10 Works Cited 71
Chapter 11 Appendices 72
Appendix A: Engineering Requirements Flow Down Chart 72
Appendix B: Preliminary Design Report Material (Initial Concept Design) 73
Appendix C: PDR Design Housing 74
Shifting System 75
Recombining Section 76
Appendix D: Engineering Design Exploded Views and BOM 78
Appendix E: Equivalent Inertia seen by the Transmission Derivation 88
Appendix F: Project Budget/Purchasing 92
Appendix G: Shift and Time to Speed Testing 94
Appendix H: Project Code 95

List of Figures

Figure 2.1 Dual-carrying sequential transmission from Liberty's Gears. 2
Figure 2.2 Synchronizers vs Dogs 3
Figure 2.3 Sample Hand Calculations.... 4
Figure 2.4 STM32 L476RG Nucleo Microcontroller 5
Figure 3.1 Boundary diagram for this project. Encompasses transmission and the shifting system 6
Figure 4.1 Concept Design Diagram for Project GHOST 9
Figure 5.1 Full CAD Assembly of Final Design.... 12
Figure 5.2 Input motor used for this system 13
Figure 5.3 Electronic Speed Controller (ESC) used to for input motor.... 13
Figure 5.4 (Left) Hall Effect Sensor, (Right) RPM Rotor & Steel Pins.... 14
Figure 5.5 CAD model of input motor with motor mount.... 14
Figure 5.6 Plastic 3D printed housing featuring clam-shell design. 15
Figure 5.7 Transmission housing lower half.... 16
Figure 5.8 CAD assembly of transmission internal components.... 17
Figure 5.9 Recombining Gear and Output Shaft.... 17
Figure 5.10 CAD models of Single and Dual Shifter Fork.... 18
Figure 5.11 Flywheel CAD Assembly.... 19
Figure 5.12 Output to Flywheel Shear Coupler 20
Figure 5.13 Shift servos selected for shifting mechanism 21
Figure 5.14 Shift servo mount and limit switches.... 21
Figure 5.15 Full dual countershaft shifting mechanism assembly.... 22
Figure 5.16 Nucleo L476Rg Microcontroller 23
Figure 5.17 Daughter Board.... 24
Figure 5.18 Example of State Transition Diagram for Button Task 25
Figure 5.19 Input Motor Speed Task State Transition Diagram.... 26
Figure 6.1 Completion of 3D Printed Models.... 30
Figure 6.2 3D Printed Mock-Up with Shift Forks 31
Figure 6.3 Layout of the 3D Printed System.... 32
Figure 6.4 Updated Drive System.... 33
Figure 6.5 Recombining Shaft Drawing and Machined Part 34
Figure 6.6 Completed Recombining Shaft Assembly.... 34
Figure 6.7 Aluminum Raw Stock for Housings and After Manual Machining Work 35
Figure 6.8 CNC Milling Machine Used for Manufacturing.... 35
Figure 6.9 CAM Programming for CNC Tool Paths 36
Figure 6.10 Upper Case Drawing and Final Machining of Housings on CNC Mill.... 36
Figure 6.11 Machining of the Lower Housing.... 37
Figure 6.12 Completion of Upper and Lower Housings.... 37
Figure 6.13 Boring the Bearing Surfaces into the Housing 38
Figure 6.14 Machining the Pushrods into the Lower Housing 39
Figure 6.15 Finished Pushrod Assembly and Shaft Seals.... 39
Figure 6.16 Initial Fit Up of Gear Train in Housing.... 40
Figure 6.17 Testing of Viewing Window Plastic.... 40

Figure 6.18 Final Product Assembly Exploded View 41
Figure 6.19 Final Assembly of the Transmission 42
Figure 6.20 Revised Flywheel for the Simulated System Inertia 43
Figure 6.21 Building the Sub-Assemblies Per Drawings 43
Figure 6.22 Assembly of the Three Main Sub-Assemblies 44
Figure 6.23 Final Layout of Transmission Assembly 44
Figure 6.24 Transmission with Shift Servos and Gear Sensors 45
Figure 6.25 Top-Down View of Transmission Assembly 46
Figure 6.26 Input Motor and Flywheel Assemblies 46
Figure 6.27 Custom Aluminum Shifter for Transmission 47
Figure 6.28 Daughter Board for STM 32 Nucleo Microcontroller 48
Figure 6.29 Figure of the soldered and wired-up Switches/buttons 49
Figure 6.30 Final wiring on testing assembly 49
Figure 6.31 Construction of Emergency Power Cuttoff Switch 50
Figure 6.32 Main Battery Packs and Battery Monitoring 51
Figure 6.33 Final Wiring of Inputs and Outputs to Main Control Board 51
Figure 6.34 Shift selector mounted on the UI 52
Figure 6.35 LCD and Button Housing for User Interface 53
Figure 7.1 Shear Test Coupon 54
Figure 7.2 Shear Couplers Breaking as Intended and Revised Strengthened Coupler 56
Figure 7.3 Throttle Pedal Input Setup 56
Figure 7.4 Pedal Signal to PWM Signal Code 57
Figure 7.5 RPM Sensor Speed Value and Optical Tachometer Verification 57
Figure 7.6 Servo 1 PWM values for moving to neutral and First Gear positions 58
Figure 7.7 Mastermind FSM diagram. 59
Figure 7.8 Initial Test of Transmission RPM Sensing During Motor Deceleration 60
Figure 7.9 Plot and Function of Motor RPM Vs. Input PWM 61
Figure 7.10 Finite State Machine diagram for the shift task 62
Figure 7.11 Rev Match RPM Algorithm Code 64
Figure 7.12 Final Full Speed Testing 65

List of Tables

Table 3-1 System Parameters and Values (model).. 7
Table 7-1 Shift time/Time to speed testing results.. 65
Table 7-2 Shift time/Time to speed test results. Includes Z and P values.. 66
Table 7-3 Summary Engineering Requirements Table 66

Chapter 1 : Introduction

Project GHOST (Garage-made High-Torque Optimizable Sequential Transmission) had but one simple goal: to build a model transmission suitable for the high torque capacity and design a control system that allowed for quick shifting speeds between large RPM bands. This project was self-sponsored, and the first consumer of it will likely be the head of the team and primary financial benefactor, Austin Conrad. Based on the wants and needs of Mr. Conrad, the project parameters and engineering requirements were developed for a scale model of the dual countershaft transmission design for an electric vehicle. The first sections of the project discussed the detailed preliminary research and conceptual design leading up to a functional transmission model. Inspiration was taken from similar transmission designs and modified to utilize specific gear train components to aid in the manufacturing process. The transmission shift controller and electronics utilized both off the shelf components in addition to custom hardware to function with code written to accomplish the specified shift tasks. The manufacturing of the transmission was prototyped from 3D printed components and later completed with aluminum parts. Lastly, the transmission system was verified for its ability to perform the specified tasks and meet the set sponsor requirements. Due to safety concerns, the torque carrying capacity and life were not tested and instead was left as an analytical exercise.

Chapter 2 : Background

For the mechanical design, the idea of having a dual countershaft transmission is not an original one. This concept has been employed by professionals in the racing industry. One example is the Equalizer Clutchless Transmission from Liberty's Gears (Figure 2-1). With this type of transmission, the torque carrying capacity is increased over two-fold from typical single countershaft models. Torque values for this transmission can reach up to 3000 ft-lbs, while most other transmissions can only reach up to around 600 ft-lbs. Additionally, the dual countershaft design allows the overall transmission weight and size to decrease with better load distribution.

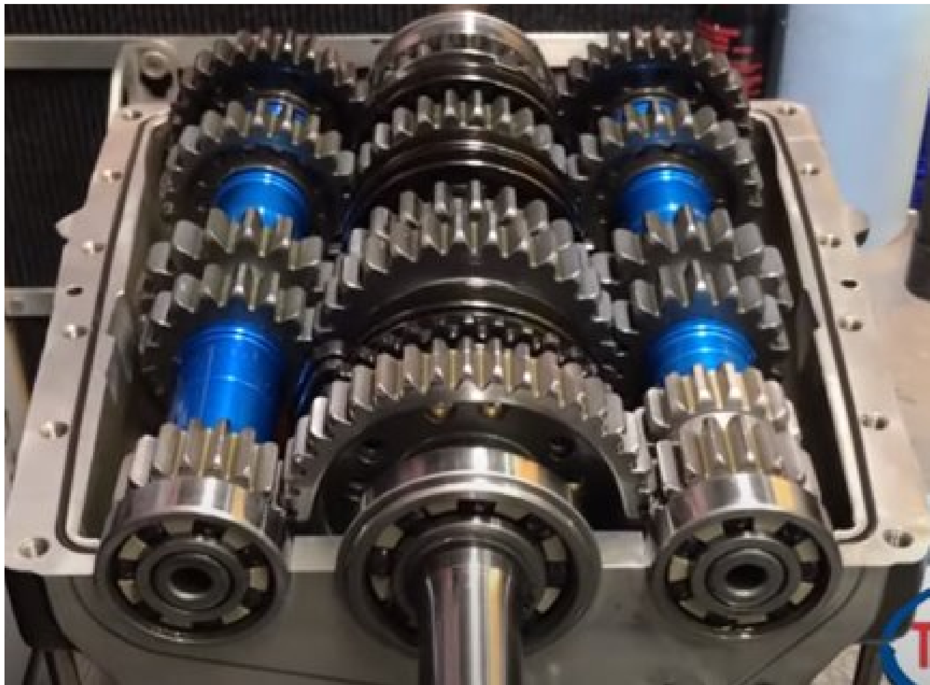
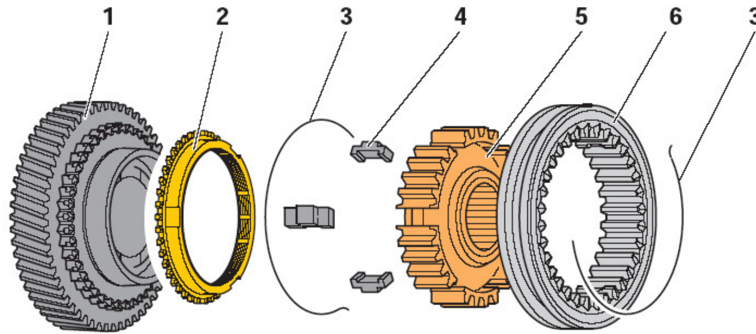


Figure 2.1 Dual-carrying sequential transmission from Liberty's Gears.

Some of the critical components of a transmission, which dictate torque carrying capacity and ease/speed of shifting, include synchronizers (synchros) and dogs as seen in the images below.



Synchronizers

Dogs

Figure 2.2 Synchronizers vs Dogs

The use of synchronizers in high performance vehicles inherently produces a weak link as synchronizers grind down, like a clutch or brake pad, to match different gear rpms between shifting. This means that the use of dogs in higher torque situations provide a better option as these are not engineered wear components, and only suffer from general wear or fatigue as a mode of failure. The Equalizer Transmission from Liberty's Gears also uses dogs to allow for the different rpms between gears. However, the dogs used in the Equalizer are designed in such a way that they will not allow for engine braking, where the vehicle slows down with the motor and not the brakes. This is because unlike the dogs seen in the images, they are cut to have an angle on one side which means that if engine braking were to occur, the slide collars would kick into neutral. This is a safety feature for the drag racing vehicle but makes it unsuitable in a high-performance streetcar or track racing situation. The dogs do not require as closely matched rpm gap as the synchronizers to complete the shift, which results in faster shift times. Aside from the increased speed, dogs also offer another benefit in that they are easier to model for the controls as there is no frictional component (synchro).

The mechanics of the dual-carrying model are relatively simple. In essence, this design changes one parameter, the distance between the force-couple required to make a torque. It does this via two facts: The system carries torque symmetrically between the two countershafts, and the forces on either side of the input- and output-shaft are oppositely directed.

For the symmetric property, this is derived from the very design of the transmission. Either countershaft is identical in construction, uses identical mounting methods, identical bearings, and uses identical gear ratios. The input-, output-, and countershaft are all coplanar to avoid strange loading cases. Therefore, there is no reason to suggest that the model is asymmetric.

As for the oppositely directed forces, this comes from simple statics. If the input gear spins clockwise, either countershaft would spin counterclockwise. The left side of the input gear spins upwards, the right side spins upwards. The respective reactions from the countershafts are opposite this. Thus, the forces are oppositely directed.

In summary, the additional countershaft simply doubles the distance between the force-couple that makes up the torque. If input torque is held constant, then the result is that the transmitted loads are, consequently, halved. This is illustrated in Figure 2-4.

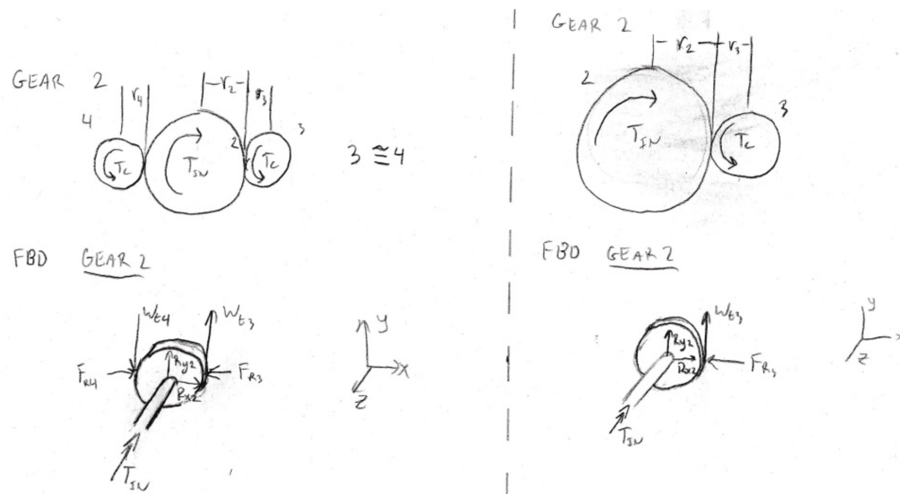


Figure 2.3 Sample Hand Calculations

To increase the vehicle performance, it was desired to have an automated manual transmission by having rev matching between shifts, and having shifts be actuated electronically rather than manually. What this means is that when a shift request occurs, the controller will send signals to two small servo motors connected to the shifting mechanism. By utilizing the servo motor arms, the shift collars will be moved into the neutral position, disconnecting the input shaft from the output. With the collars in neutral, the controller will then increase/decrease the motor speed depending on the direction of the shift (up-gear or down-gear) and then signal the servos to move the collars back into place. This process will only involve a single user input (pushing a button for either a up- or down-shift); the rest will be handled entirely by the controller. In order to do this, a controller with sufficient processing power and I/O capabilities had to be selected. The STM32 L476RG Nucleo (shown below) was chosen for this reason.

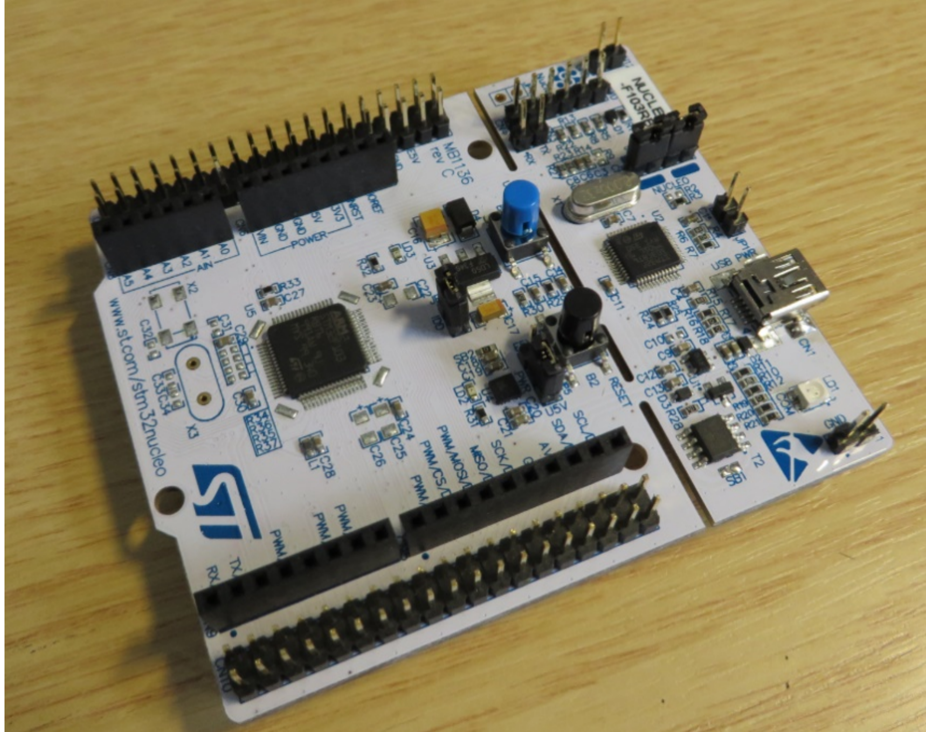


Figure 2.4 STM32 L476RG Nucleo Microcontroller

Initially, the proposed solution for the first tests was to use a hobby electronics board called the Teensy 3.5 which ran an ARM architecture at 120 MHz. Hobby boards such as this had sufficient power and speed to accomplish this task, however it was not selected due to a couple key reasons: 1) The Nucleo was already owned by two of the team members and 2) The Nucleo was the same microcontroller being used in ME 507, Controls Systems Design. Two of the team members took ME 507 concurrent to this project and utilized the class' term project for this transmission. After gaining approval from the instructor of ME 507 to have the term project be designing the board for Project G.H.O.S.T, the Nucleo was confirmed to be the microcontroller used for this project. The Nucleo also runs an ARM architecture, but only at 80 MHz. However, because C++ is being used with the Nucleo as opposed to Arduino language with the Teensy 3.5, the overall program was running at a couple orders of magnitude faster (microseconds as opposed to milliseconds).

Chapter 3 : Objectives

3.a Problem Statement

High performance electric vehicles such as the Tesla Model S P100D utilized the advantages of high torque and long power band provided by electric motors to produce an excellent driving experience for normal automotive users. However, these advantages produced problems when the vehicle needed to go faster than average road speeds due to the torque and power drop off at higher speeds. This was especially problematic when this type of vehicle was used in a racing context. The use of a transmission in high-performance applications where higher speeds were desired has seen difficulties with how to manage the high torque produced by electric vehicles. To solve this, a dual countershaft transmission design has been proposed as a plausible solution. A scale model of this type of transmission was built and subsequently tested and verified using a custom designed control system.

3.b Boundary Diagram

Figure 3.1 shows the boundary diagram for the project. The boundary diagram is a quick visual representation of the scope of the project. In this case, the boundary encloses the transmission itself (including the motor input) and the control system used to shift gears. The output torque of the transmission that crosses this boundary is not technically a part of this project, but the model being built was used to drive a mock inertia (A.K.A. a flywheel). Similarly, the inputs to the transmission were the rotational speed and torque of the electric motor, which was not shown on this diagram. Also not shown were the user's inputs. The user interface utilized a custom- built shifting mechanism, an electronic pedal to control the input motor speed, and an LCD screen for reading system data.

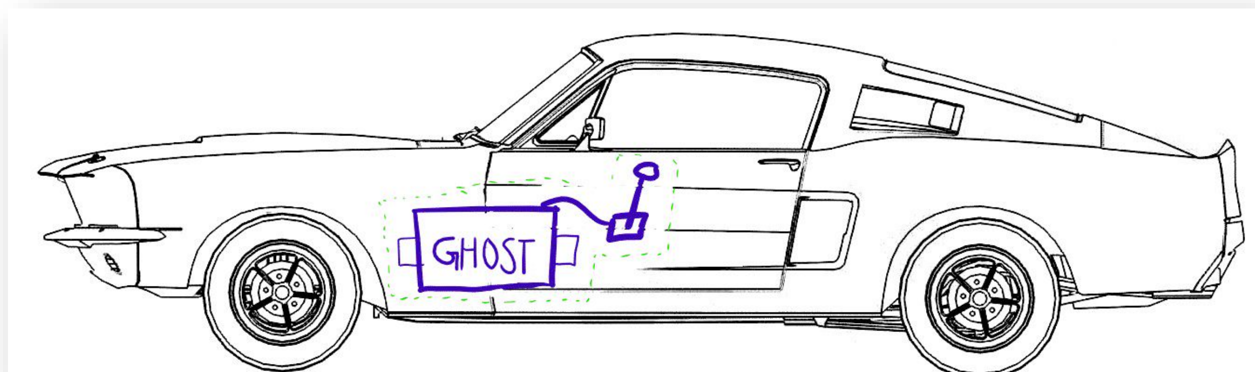


Figure 3.1 Boundary diagram for this project. Encompasses transmission and the shifting system

3.c Customer Wants and Needs Summary

As per the customer wants and needs, the modeled transmission was a high torque, rev-matching, sequential transmission that could be used as a design basis for transmissions in high-performance electric vehicles. The dual-carrying gear transmission was the design for this project because it provided increased performance by increasing torque carrying capacity while still being easily serviceable. Table 3-1 depicts the required system parameters and values, including the given tolerance range. Parameters including an increase in torque capacity, measuring input and output rpm, and measuring shifter fork position are labeled as "low risk" items. These items were essential to the function of the transmission and could not be left out. The additional requirements listed were "medium risk" as the requirements were semi-flexible.

3.d Engineering Specification Table

Table 3-1 below lists the target system parameters for this project. These parameters were determined based on the qualitative and quantitative requirements given by the sponsor. Note that all these values were determined with the scale model in mind, not a full-size transmission.

System Parameter:	Value	Tolerance	Risk	Compliance
Time to Speed	6s	+ 1s	M	A,T
Motor rpm	MIN 4000	MIN	M	A,T
Input Torque Capacity Increase	80%	- 5%	L	A
Shifting Speed	1.0 s	+ 0.1 s	M	T
Meas. Input and Output RPM	MIN 8000	MIN	L	T
Shifter Dog Position	12.7 mm	± 2.0 mm	M	A, T
Driveline Inertia (Flywheel)	0.01 kg-m^2	± 5%	L	A, T
Number of speeds	2	MIN	M	I

Table 3-1 System Parameters and Values (model)

Because of the self-sponsored nature of this project, a different tactic had to be taken when it came to generating specific engineering requirements. While the system parameters given to the group by the sponsor and research are high level requirements, they did not immediately indicate what specific engineering values were needed for the different sub-systems of the projects. In order to process the system parameters, the flowchart in Appendix B was generated. This flowchart began with the sponsor requirements/system parameters that set the foundation for the whole project. From these system parameters, engineering requirements were generated. These engineering requirements fall under two main categories: Mechanical and Controls. These engineering requirements then affected all the subsystems of the project, with some requirements affecting multiple sub-systems. A full discussion of the specfic parameters for the different components of the subsystems can be seen in Chapter 5.

One example of how these system parameters and values were used to better define subsystem parameters is the servo motor selection. To shift gears for this transmission, servo motors must have been able to move the shifter fork of a given mass, a set distance, within a certain time, and

this dictates an acceleration. From this, a force/torque, speed and throw tolerance was developed and used to justify the purchase of a given servo. Similarly, the input motor selected must have been able to meet the time to speed requirement of 6 seconds as well as the input speed requirement of 4000 RPM. This was done by looking at the motor speed/torque parameters like K_v and K_t.

3.f Objectives Summary

The proposed transmission took advantage of the positive attributes of the NHRA racing transmission while including daily driving aspects such as smooth shifting and dynamic braking. Large drive dog teeth as well as dual countershafts were used to achieve an 80% torque capacity increase when compared to single countershaft designs. The current model was built with two gear ratios: direct drive and overdrive. A stretch goal for the project was to add to two additional gear ratios.

Shifting was accomplished by using arms attached to the servo motor rotors to linearly actuate the shifting forks. The user would be able to request a shift using a custom-made shifter stick which has buttons attached to it. Once the user requested a shift, the control system would interpret the signal and subsequently send a PWM signal to the servos. Limit switches and other sensors were used to check the shift fork movement and ensure that full gear engagement has been made after a shift is requested by the user.

The control system for this project was designed to ensure that the user has full control of input speed and when to shift. Because of this, the output and input rotational speeds were measured to within $\pm$ 10 RPM as described in the system parameters/requirements table. This was accomplished using quality Hall Effect sensors mounted near the output and input shafts. Assuming a desired shift speed of up to 1 second as described in the specification table, the microcontroller used for the control system needed to have a high enough processing speed to be able to send and receive signals within a short time period. The overall control system design at the software level also play into this. Each task of the control system software was designed with an appropriate amount of latency to ensure proper function of the system.

Chapter 4 : Concept Design Development

4.a System Overview:

The system is composed of 4 major subsystems outlined in Figure 4.1. These 4 subsystems include: the input (drive) motor (1), transmission assembly (2), flywheel or simulated inertia (3), and user interface and controller (4). The goal of the design was to integrate each of these subassemblies into one control system, which took in a shift request from the user and completed a rev-matched shift to better utilize the torque of the electric motor at higher RPM values.

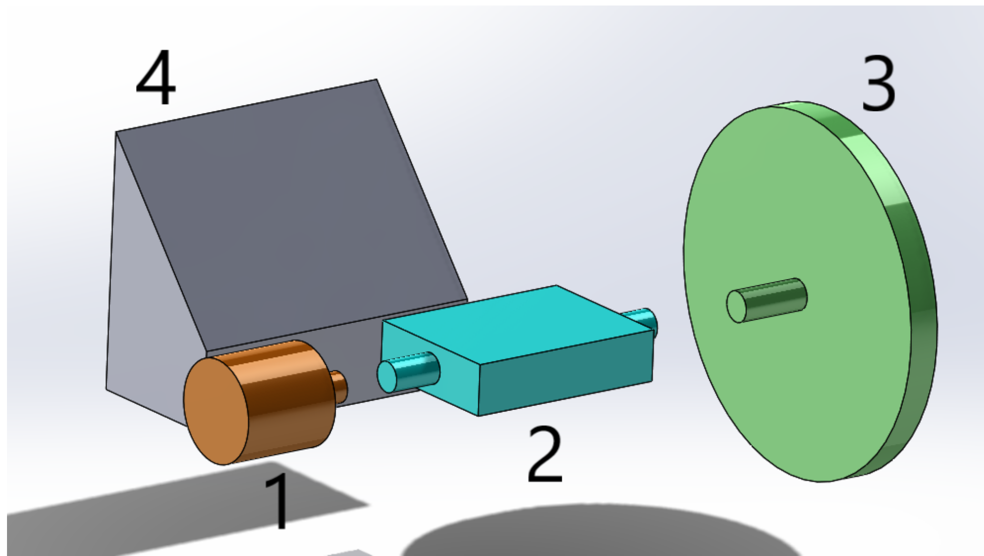


Figure 4.1 Concept Design Diagram for Project GHOST

4.b Input Motor

The key components within the input (drive) subassembly include: main motor, shear coupler, rpm sensor and motor mount. The motor provides the input torque to drive the system and was controlled via the programmed microcontroller. To ensure proper control feedback, the input RPM was measured using an RPM sensing assembly consisting of a hall effect sensor and a wheel set with evenly spaced metal pins. When the pins pass by the sensor, the sensor provides feedback for the microcontroller. The shear coupler was a component designed to fail under a given torque load. The shear coupler was used as a safety feature to ensure that the systems could be isolated in the event of a mis-shift/error, protecting more expensive components and preventing the motor from drawing too high of a current for extended periods of time.

4.c Key Transmission Components

The key components within the transmission assembly include: the split (or clam shelled) housing, dual countershaft/gears, dogs, pushrods and shifter forks, and shifting servos. The clam shelled housing would allow for ease of manufacturing, maintenance, and the troubleshooting. The other advantage of the clam shelled housing was ease of access to internal structures/support for critically dimensioned parts, such as the dual counter shafts. The purpose of the dual counter shafts was to provide an increased torque carrying capacity through symmetrical loading. This increased torque carrying capacity better utilizes the advantages of the electric motor by allowing for more gears than a traditional single carrying gear transmission. The dual carrying gears transfer their torque through a recombining gear at the output of the transmission. The gears would be engaged/disengaged via dogs as described in Chapter 2, to transfer torque from the input shaft through the desired gear. To position the dogs forwards/backwards (into the desired gear ratio), they were moved via a servo and pushrod, which was controlled via the control system.

4.d Flywheel

The simulated inertial system was primarily a flywheel of proportional rotational inertia for the model transmission. The key components within the simulated inertia sub assembly (aka flywheel) consisted of: the flywheel, flywheel mount, and RPM sensing assembly. The flywheel allowed the motor and transmission to experience a comparable rotational resistance to that experienced by an actual vehicle. The RPM sensor allowed for the controller to monitor the output RPM, which is then feed into the control loop. The output RPM could then be used to determine the point at which a shift can occur successfully. The shear coupler on the output was to prevent harm to the transmission internals in the event of a miss-shift when there is sufficient rotational inertia in the flywheel.

4.e User Interface

The user interface allowed an operator to drive the simulated transmission with a series of actuators and feedback including: an acceleration pedal, hand brake, shift selector, start, stop and mode button, as well as an LCD screen. The LCD displayed the current input RPM, mode, simulated output speed and provide indicators for when to shift given RPM limits. The shift selector allowed the operator to select a desired gear by pulling or pushing on the selector to up or down shift the transmission respectively. The pedal worked as expected by allowing variable throttle/brake request to the motor or flywheel and having the rpm change accordingly.

4.f Control System

The control system consisted of a microcontroller and daughter board. The microcontroller was a small, pre-programmed computer which once supplied with power, performed the desired tasks indefinitely. These are common to all modern control systems including automotive. Once programmed, the microcontroller took in the signals from the given sensors (RPM, pedal, etc.) and continuously performed operations and calculations based upon the predetermined programming.

To ensure proper connection and mitigate hardware failure, a daughter board was attached which consists of all the required connectors for the sensors to effectively communicate with the microcontroller.

4.h Controls Method

Microcontrollers operate at speeds far exceeding the speeds mechanical systems require, with many capable of performing loops in the tens of microseconds compared to a typical mechanical system which operates in the milliseconds. The orders of magnitude difference allowed many different operations to be performed before mechanical response times notice the difference. To achieve this fast operation, it was important to employ cooperative multitasking and finite state machines. The cooperative multitasking took advantage of the quick cycles to perform a series of small operations quickly and repeatedly, without taking so long as to delay another task. The finite state machines ensured that the given tasks are performing the correct operations given the appropriate inputs. Together, these tools allowed for high level design of the control system and make for much simpler coding implementation.

Chapter 5 : Final Design

5.a System Overview

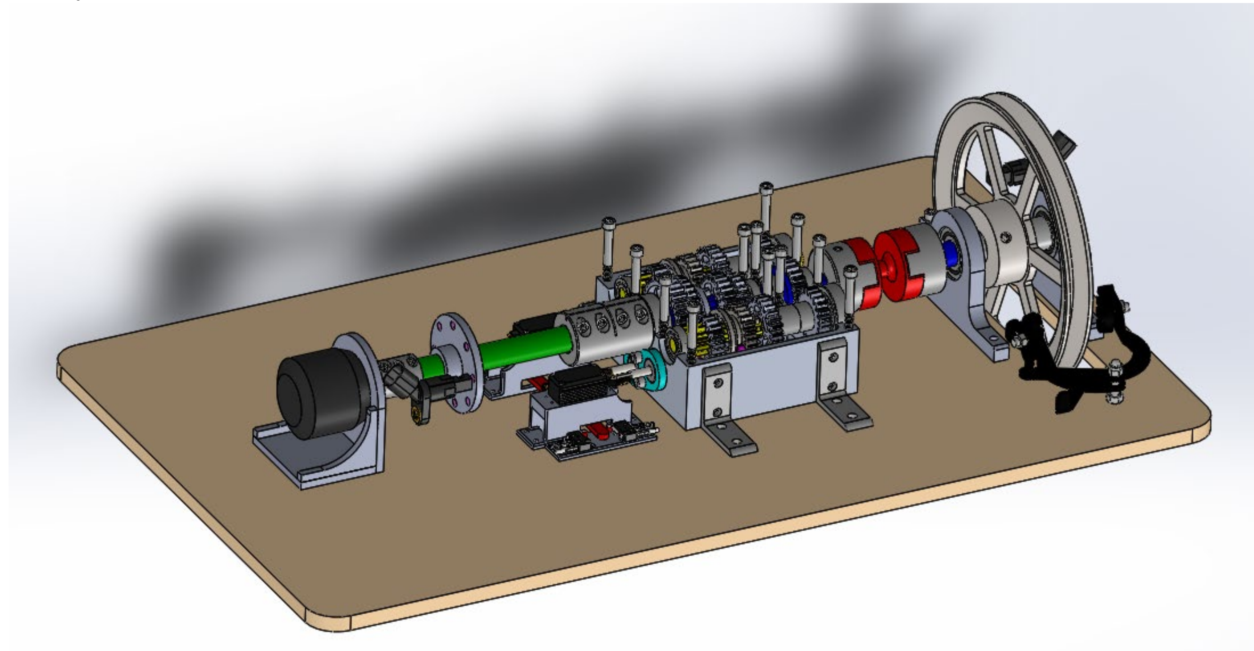


Figure 5.1 Full CAD Assembly of Final Design

The final design of the model transmission test assembly is shown above in Figure 5.1. From left to right there are the three major mechanical subassemblies including the input/drive motor, transmission and inertia simulation or flywheel. The systems were designed to provide the appropriate inputs and outputs to control the transmission using a microcontroller and user interface (not pictured). Once programmed, a user would be able to spool up the transmission and flywheel as desired and upon selecting a shift requested through the shift selector, the control system would take control of the system, performing a rev matched shift into the selected gear, before returning controls back to the operator. Some of the remaining hardware not depicted in the main assembly included the flywheel shield and RPM sensor mounts.

5.b Input Motor & ESC Subassembly

To select the input motor for the system, several design considerations were considered. First, the total drivetrain inertia was calculated, which included the transmission in addition to the flywheel system. With the time to speed of 6 seconds from the engineering specifications and a resulting angular acceleration of .309 rad/s^2, the required input torque was found to be 1.19 N-m. A motor would need to be selected that had a minimum torque of 1.19N-m and a minimum of 4,000 RPM per the requirements. The brushless 3 phase DC outrunner was chosen for its low RPM and torque capability. Adversely, due to the nature of three phase open loop motor, motor cogging from zero to initial start-up RPM under load is possible. This would affect the time to speed if the motor first cogs on startup. To solve this problem, a motor utilizing hall effect sensors and closed loop position feed back to the speed controller was chosen. The motor that met the RPM, torque, and closed

loop requirements was a DIY electric skateboard motor seen in Figure 5.2. This motor is capable of 1.9 N*m of torque and a max 8000 RPM at 45 amps.

Figure 5.2 Input motor used for this system

The selected electronic speed controller or ESC (see Figure 5.3) was selected based on current requirements, closed loop feedback and control method. This speed control allowed for 12 cell – 50v operation while delivering 50 Amps capability. It was noted that the speed control receives a PWM signal from the microcontroller to drive the motor. Unfortunately, upon testing this combination, it was discovered that this speed control varies motor power by modulating motor current, not RPM. This resulted in a max motor RPM at any throttle input, with the throttle signal regulating output torque. This combination would not work as an adjustable motor RPM is needed to allow for proper RPM matching in the gear train. The sacrifice that needed to be made to achieve RPM modulation was to source another speed control. The final decision of speed control was a Castle Creations Phoenix Edge 60 Amps High Voltage controller shown in Figure 5.3. This speed control was known to modulate motor RPM but had the drawback of no closed loop motor position sensing. The sacrifice of possible motor cogging at startup was necessary to achieve the required control type. It will be explained later that the input motor was changed due to oversights in the transmission modeling and system drag. Subsequently, the ESC was resized to account for the different motor.

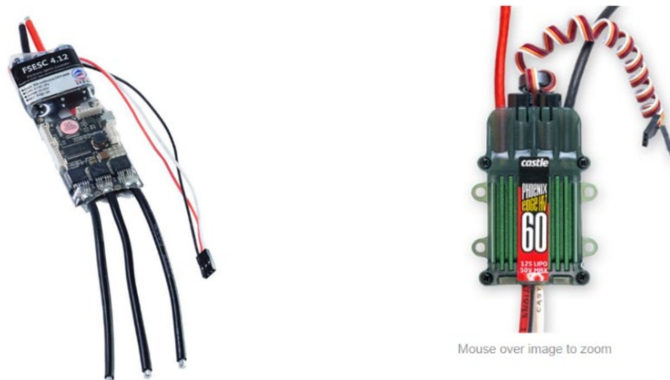


Figure 5.3 Electronic Speed Controller (ESC) used to for input motor

To accurately shift the model transmission using rev matching, a measurement of RPM was required for the microcontrollers control loop. To acquire the input and output RPM of the shafts, hall effect sensors were used as seen in Figure 5.4. The sensor options considered were optical, ESC or hall effect. The optical sensors could have worked for input and output but would have been more difficult to implement. The ESC RPM sensing would have been somewhat difficult to pick up, however it would have only provided input RPM. Hall effect sensors work on both the input and output of the transmission making a simple solution. The spacing requirements from the hall effect sensor eliminated the use of the splines on the input shafting for picking up the rotary signal. To get the input RPM signal an input RPM rotor seen in Figure 5-4 was designed. The eight steel pins were placed such that pick up is sufficient for the hall effect sensors while being in a location which was easier to mount. For the motor, the max input speed of 4000 RPM and the eight pegs on the rpm wheel corresponded to a max frequency of 533 Hz. The max operating frequency of the sensor is 15 kHz, which resulted in a safety factor of 28 for the input side of the transmission.

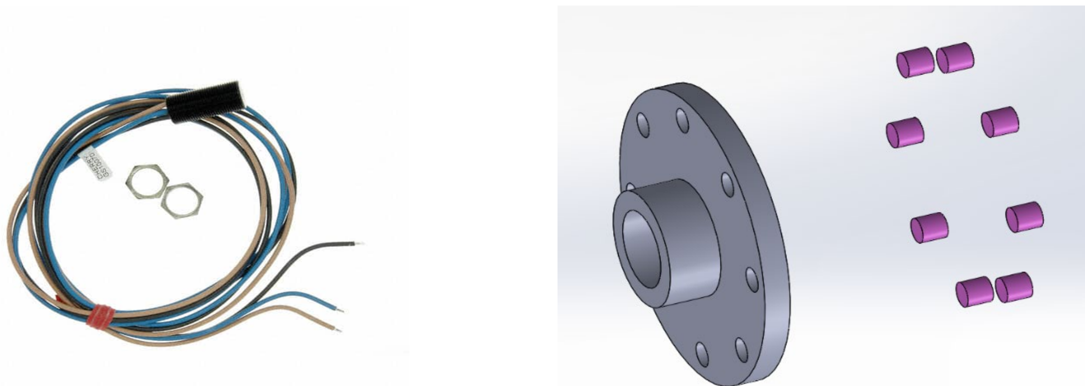


Figure 5.4 (Left) Hall Effect Sensor, (Right) RPM Rotor & Steel Pins

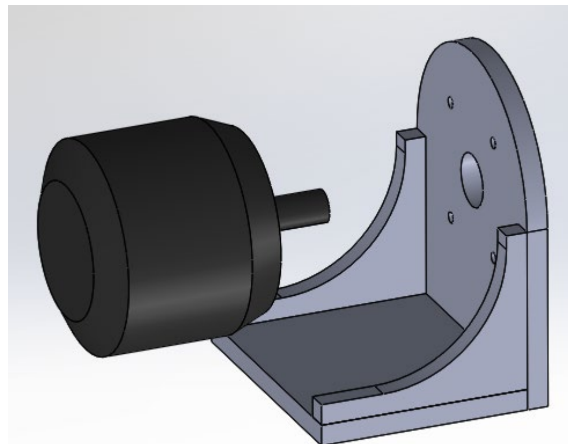


Figure 5.5 CAD model of input motor with motor mount

The motor mount shown in Figure 5.5 was set at the correct height off the mounting plate to align the motor with the input to the transmission. The mount was 3D printed for the first testing iteration and in the final form was be machined steel plate and then welded together. There were not major

model considerations for the bracket other than ensuring it holds the motor to the correct height to prevent excessive wear or bending.

The last consideration for the motor subsystem was a shear coupler. The shear coupler for the input side of the transmission helps ensure that the motor is also protected in the event of a transmission lock up (where more than one gear is selected at a time). The addition of this input shear coupler was not likely as needed as the output shear coupler due to the low inertia of the motor and input shaft. The motor also had an electrical fuse to prevent stall and over current from cooking the motor in the event of a lock up. This was not a difficult part to implement/remove, and preliminary testing proved its necessity. For more details on the form and function of a shear coupler, see section 5.d.

5.c Transmission Subassembly

The transmission subassembly consisted of three major components, the housing, gear train and shifter-forks. The design was modeled after the Liberty 5 speed transmission (depicted in Figure 2.1) which uses the gear symmetry to improve the load distribution resulting in a transmission capable of higher torque loads. The model housing used the same form factor as the Liberty 5 Speed, which is a horizontally parted "clam-shelled" design to allow for ease of access and servicing and can be seen in Figure 5.6.

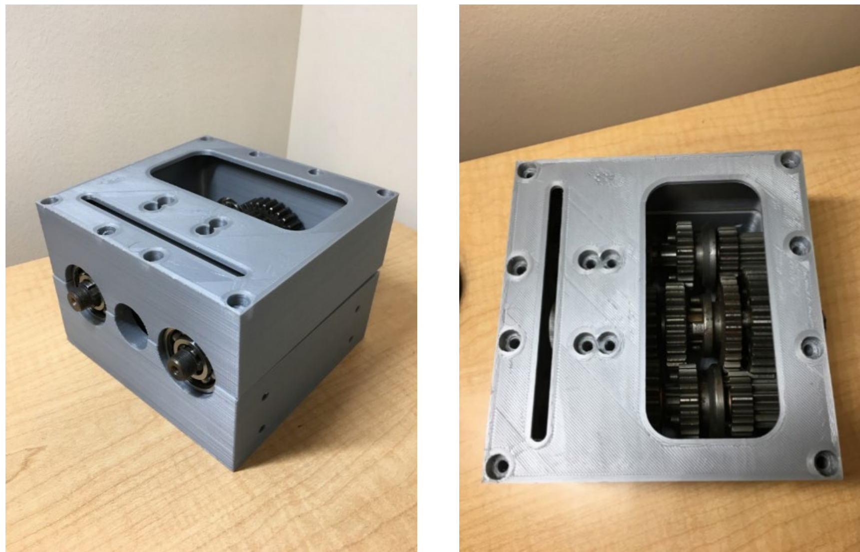


Figure 5.6 Plastic 3D printed housing featuring clam-shell design.

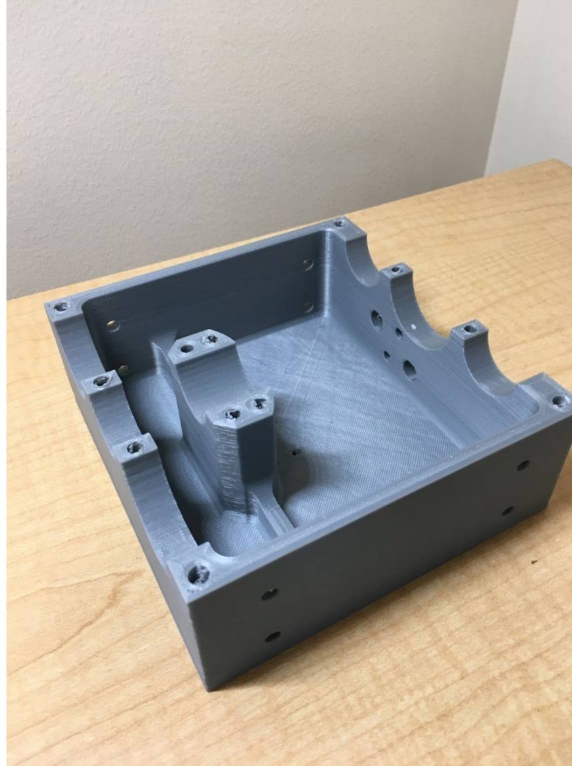


Figure 5.7 Transmission housing lower half.

The housing used for the model transmission also includes a set of viewing ports on the top cover to allow for visual confirmation of the shift as well as easy of shift debugging during initial testing. The visual ports were made from ½" thick Polycarbonate plate which was tested using comparable ballistics (AKA a pellet gun) to confirm safety for testing. Testing was performed later in the report. The first test housings were printed in PLA to confirm shaft spacing, clearances and overall part function. The housing was designed with DFM in mind with a final iteration being CNC machined from aluminum billet. The housing contained hydraulic fluid for lubrication with the pushrod bushings as well as mating surfaces all having the proper seals to contain the fluid. Given the small size and the fact that the transmission durability was not a testing parameter, splash lubrication was used.

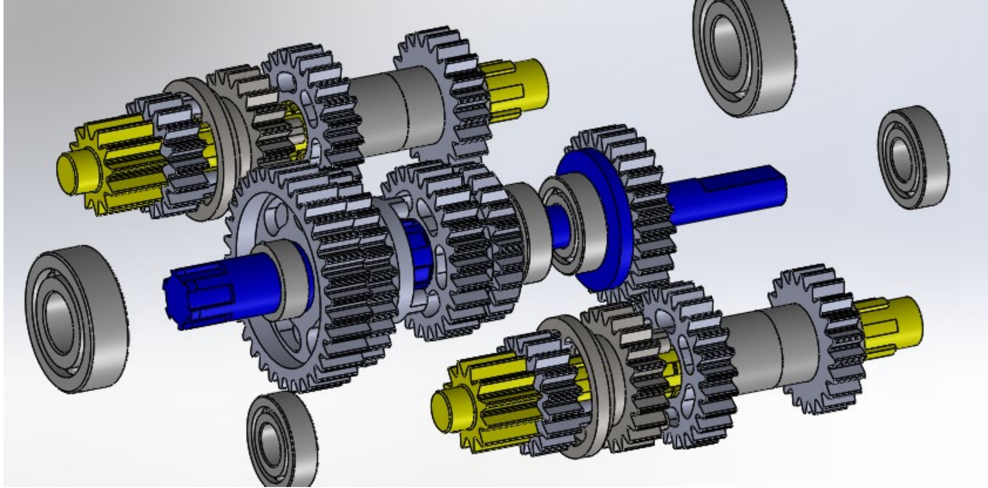


Figure 5.8 CAD assembly of transmission internal components

The gear assembly used in the model were out of a 125-cc pit bike transmission (Figure 5.8) and were selected for their form factor, availability and cost. Given the poor manufacturing reputation for the parts country of origin, these parts were not physically tested for cycle life or strength. The gear geometry was used for inertia calculations, modeling and theoretical strength/cycle life metrics to satisfy the engineering specification. Given the gear geometries and assumed material properties, the transmission torque carrying capacity increased 100% over that of the single carrying gear model, which exceeds the 80% engineering specification. Without the ability to alter gear geometries, the gear ratios of the transmission as provided would have been too low for the desired model. To fix this, the gear assembly for the model was run in reverse and will only use first and fourth gear, or as seen by the model approximately 1:0.8 and 1:2.35 for input to output ratio of the model's first and fourth respectively. The large gear ratio means that a shift request at max input speed of 4000 RPM results in a required change of 2667 RPM to achieve rev matching for the subsequent gear. The change in RPM is nearly 700 RPM greater than the specifications requirement of 2000 RPM.

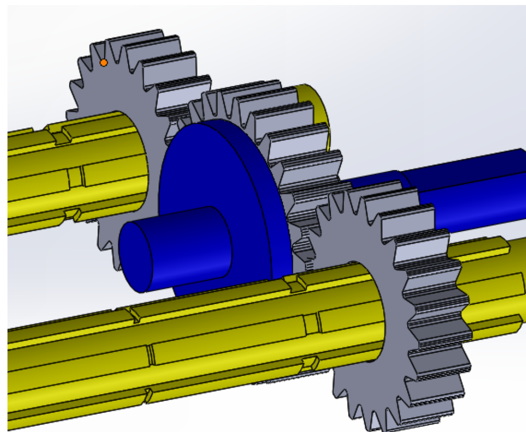


Figure 5.9 Recombining Gear and Output Shaft

The use of premade gears allowed for several savings including time and money designing and manufacturing the gear geometry and shafting. However, due to the OEM gears being a single carrying gear design, the final output of the transmission left two rotating shafts for the output instead of one. The solution was a recombining gear and output shaft. Several options were considered including multiple 3D printed gears and a few machined aluminum/brass gears. Eventual the decision was made to use the gear out of the original gear set and adding a custom recombining gear output shaft. This decision provided a backup set of gears in case of any gear failures. The recombining shaft shown in Figure 5.9 was made from a single piece of aluminum turned on a lathe and used dowel pins to transfer torque from the dog pockets of the gear through the output shaft. The pins were positioned such that there is no play to mitigate impact loads during rapid rpm changes. Given the model transmission being intended for controls, the aluminum output shaft was deemed acceptable given that the point of failure in the shaft is the plastic shear coupler on the output.

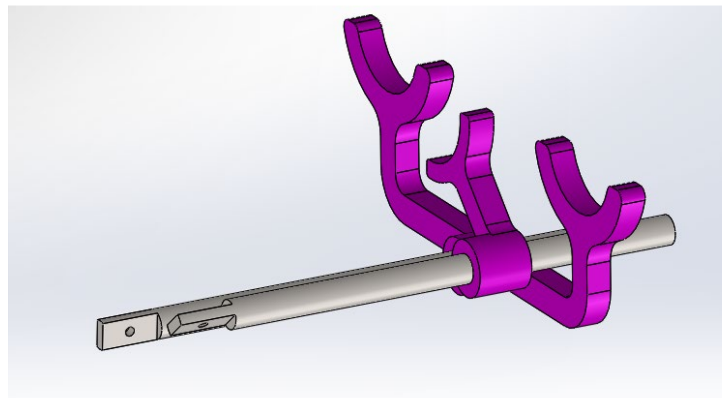


Figure 5.10 CAD models of Single and Dual Shifter Fork

The final component of significance within the transmission design are the shifter forks. These arms seen in purple in Figure 5.10 are actuated via the pushrods and servo assembly. When a gear selection is required, the forks move along the slide collars of the given shaft, positioning the dogs into the correct gear. In a typical transmission, there would only be single forks as the system would be machined for the dual carrying application. However, due to the repurposed transmission internals, it is necessary to utilize a dual shifter fork design to engage the gears needed. These were first tested using 3D printed parts to guarantee function and form factor. Once completed, they were be machined out of aluminum with the possible addition of brass inserts near the mating surface to prevent galling which might interfere with the shift. Both forks and their respective shift rods were designed to travel the full 12.7 mm without interference with other portions of the transmission satisfying the engineering requirements of section 3.d.

5.d Flywheel Subassembly

To accurately model the function of the transmission, a simulated output rotational inertia (AKA flywheel) was implemented. The flywheel (seen in Figure 5.11) provides the equivalent resistance to an electric vehicle given the size of the selected input motor. The flywheel is comprised of two

brake discs mated together with a rotational moment of inertial of 0.1 kg*m^2. Using the linear graph-normal tree method seen in Appendix F, a symbolic equation for the equivalent rotational inertia seen by the transmission was derived. The specific value for the model was then selected by comparing the mass of the sponsor's car to the scale model and unit analysis methods such as Buckingham Pi. The flywheel would keep the output of the transmission spinning during the time of a shift which would provide a more accurate model. The flywheel was supported by a shaft riding on two bearings which allowed for the minimal friction during normal use. The flywheel was made of cast iron which was beneficial for the RPM sensor. The RPM sensor on the flywheel was the same one used on the external input RPM sensor. The hall effect sensor required a ferromagnetic material to operate. With a max output speed of 11,320 RPM and the flywheel having a total of six spokes, the maximum sensor frequency used was 1.132 kHz. The maximum sensor frequency has a SF of 13 given it had a maximum operating frequency of 15 kHz. It is later discussed that the flywheel for the system was changed to a higher rotational inertia due to inaccurate assumptions about the transmission. Some of the issues pertaining flywheel sizing were not discovered until the results section of the report and are discussed in section 7.f.

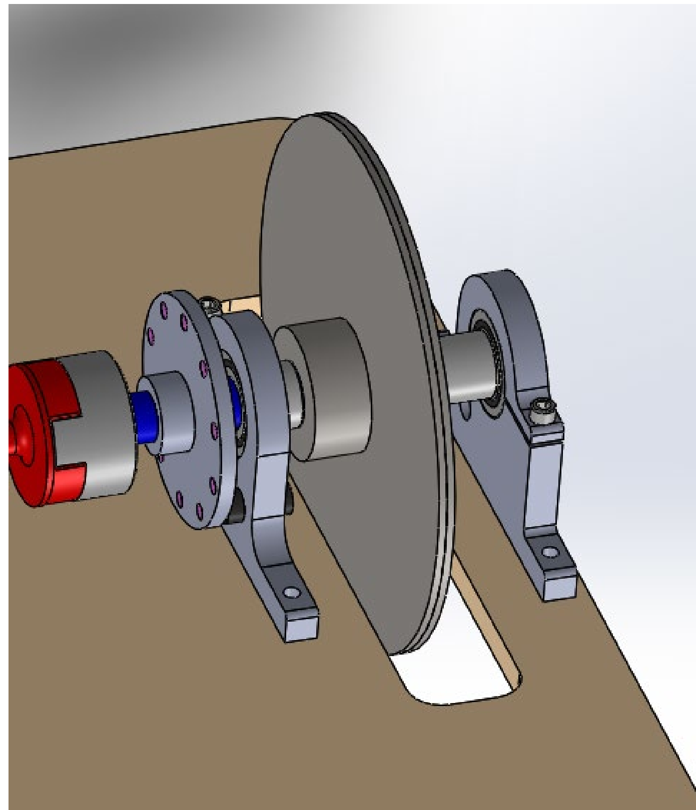


Figure 5.11 Flywheel CAD Assembly

Aside from the flywheel, there are a few other safety features that have been implemented to ensure model and operator safety. One of these is the shear coupler seen in red in Figure 5.12. The shear coupler is connected between the output shaft of the transmission and the flywheel. The shear coupler is a part designed to fail under a given load to protect the other, more expensive, hard to replace components. This piece will be 3D printed and physically tested to verify breaking at a set

torque value of ~2-3 times the expected output torque of the transmission. The major reason for this part is that the transmission used is not typical to how a dual carrying gear transmission would be designed. Due to the limits of the physical hardware, there is a possibility of two gears being selected at the same time. In the event two are selected, the transmission would instantly lock up (not rotate) and if this were to occur while at higher speeds, there is sufficient rotational kinetic energy within the flywheel to likely damage the gear train components. The anticipated use of this part is primarily a mechanical fuse to eliminate this issue during preliminary testing while proofing out the code. However, it remained through the final iteration for safety.

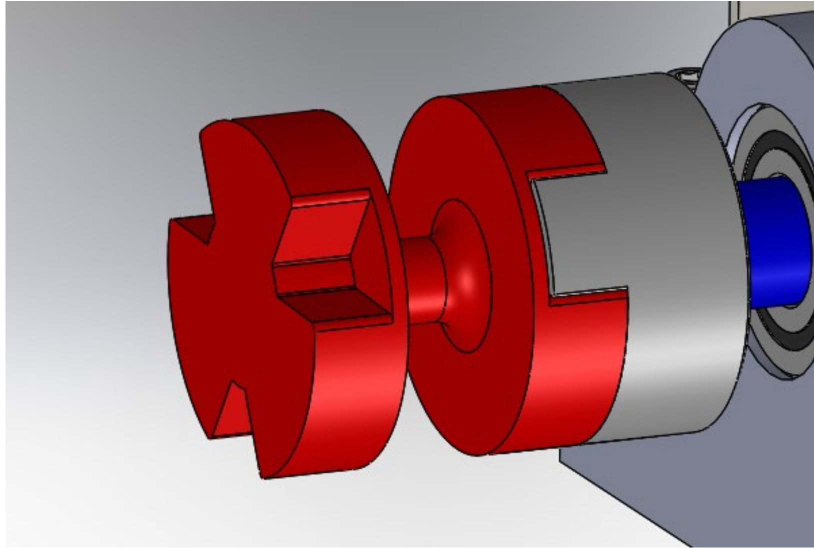


Figure 5.12 Output to Flywheel Shear Coupler

5.e Servo & Pushrod Subassembly

The shift actuation assembly consists of the shift servo, servo housing, pushrod, pushrod supports and seals. From the CAD model, the inertia of the shifting assembly, which includes the rod, fork, and gear, were found. From the inertia calculations, the required servo torque requirements were found to be 26 oz-in for the appropriate length lever arm for a ¼" shift fork throw during a 0.3 second duration. Using previous experience, a Hitec 7955 servo was chosen (Figure 5.13). This servo offers 333 oz-in of torque at 0.08 seconds / 60 degree rotation speed that far exceeds the requirements set out for the shift. It was noted that this servo runs off of 6V and accepts PWM input to control its position. This requirement was observed for use with the control system.

Figure 5.13 Shift servos selected for shifting mechanism

The servo mount shown in Figure 5.14 was designed to allow for secure mounting of the servo while retaining the servo arm's full range of motion. Additionally, the mount incorporates limit switches that trigger when the servo arm, and thus the drive dogs, have fully engaged. Failure to see a limit switch input after a requested shift will notify the control system that the shift was incomplete. The limit switches for each end point are adjustable to allow for precise location of the drive dog engagement.

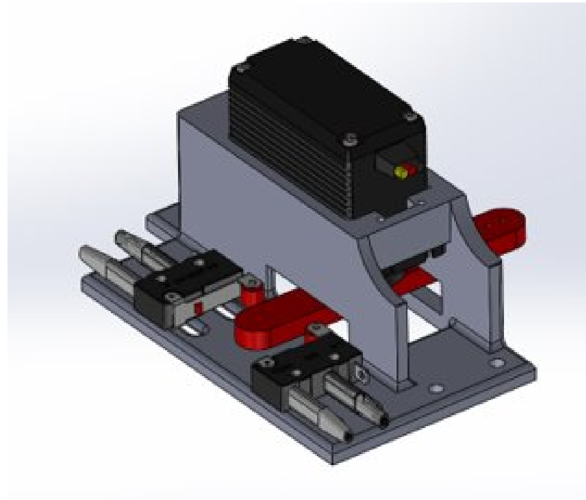


Figure 5.14 Shift servo mount and limit switches

Transferring the linear motion to the shift rods is done using ball joints. These were selected as they offer three degrees of freedom about the rotation axis and no movement in the translational planes. This freedom will compensate for any component misalignments between the servo arm and shift rods. The ball joint assembly can be seen below in Figure 5.15.

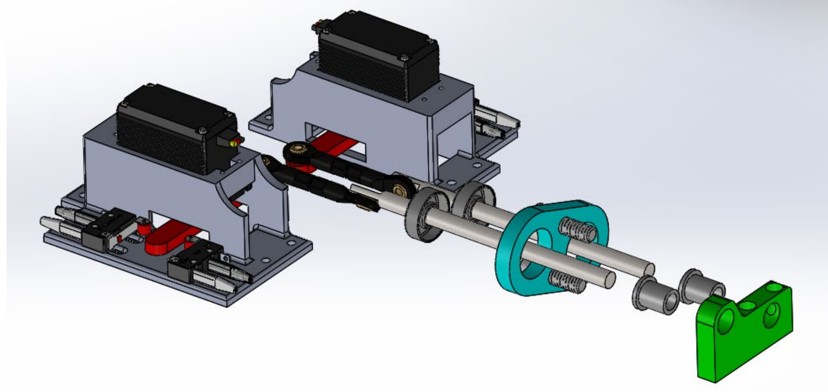


Figure 5.15 Full dual countershaft shifting mechanism assembly

Supporting the pushrod utilizes Oilite bronze bushings at both ends. Oilite bronze was chosen for its wear characteristics as this component will see a large amount of motion. These bushings were designed to be replaceable in the event the wear becomes too great. Additionally, the splash lubrication for the gear train will also aid in the wear resistance and heat dissipation of the bushings. As previously mentioned, the transmission sump will be filled with lubricating oil for the gear train. Due to the location of the pushrods, this places them below the oil level of the transmission. This required the use of oil seals on the pushrod exit holes leaving the housing. A spring-loaded rotary shaft seal was utilized for this application. This seal in conjunction with the external seal plate allowed for the case to be sealed from oil leakage.

5.f Controller

To control the model transmission, a microcontroller is required as it can take in various signals and data while still performing calculations to control the subsystems at high speeds. The microcontroller first considered was a teensy 3.5 with a 120 MHz processor. The speed of the board would have been more than sufficient for the operations, however upon closer inspection, it did not provide sufficient IO capability for the model. The Nucleo L476RG (see Figure 5-16) was selected for its IO capability and still exceptionally fast 80 MHz processor speed. The Nucleo microcontroller also has the advantages in the board layout as it is Arduino compatible, meaning a large amount of documentation and free resources exist.

Figure 5.16 Nucleo L476Rg Microcontroller

The microcontroller was programmed in C++ which runs full cooperative multitask loops with cycle times in the tens of microseconds compared to the teensy board running Arduino which would have been in the milliseconds. The faster C++ programming language is implemented through a software called VSCode which allows the user the visual functionality of Arduino but the programming speeds and operation of C++. The other feature of using a microcontroller is that this system behaves similar to how a real transmission control unit (or TCU) would. However, the TCU of the real vehicle would likely be communicating with an motor control unit (MCU) and a few other microcontrollers. After discussion with Mechatronics professors, the decision was made that the second controller does not provide any sufficient difference in model, only added chaos in the constrained project time frame.

5.g Daughter Board

The microcontroller used for this project uses prototyping and testing hardware called headers. Though these headers are great for quick testing code, they are not a good solution for long term/robust hardware interfacing. To get around the use of wires and breadboards which can be easily faulted and lead to misfunctioning code, a daughter board was designed and built and is shown in Figure 5.17.

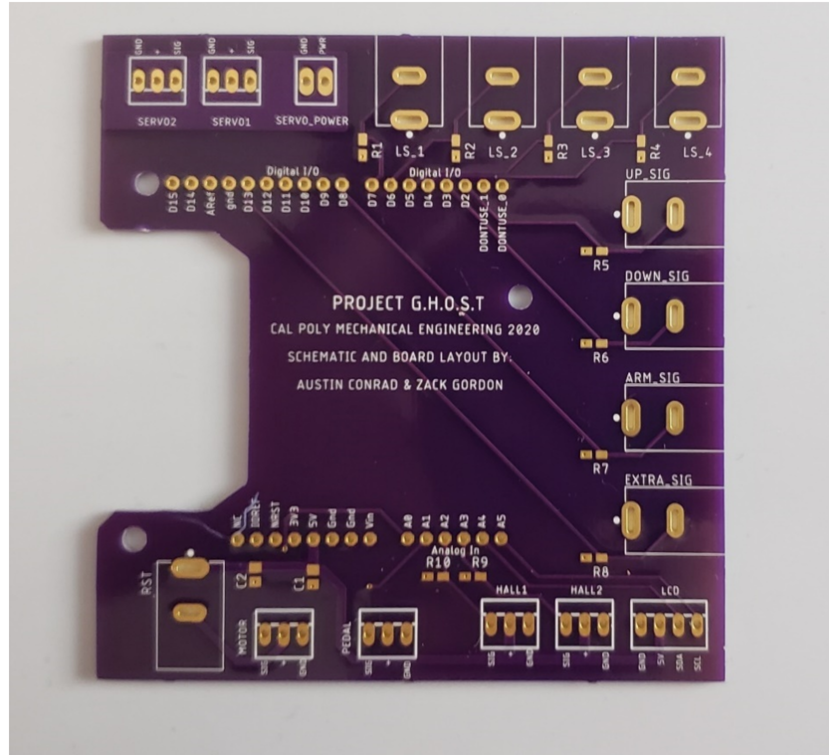


Figure 5.17 Daughter Board

The daughter board designed in Eagle PCB has all the pads, holes and slots for the hardware required by the microcontroller. Once assembled, the board makes for a convenient and robust interface between the microcontroller and all of its respective hardware. The board is labeled to ensure easy of assembly, and major damageable components such as the servos, microcontroller and motor power have symmetric wiring such that power cannot be mis-plugged in, which would result in board failure.

5.h User Interface

The major features for the user interface include buttons, an LCD screen, shift selector and pedal. The major buttons include: Start, Soft Stop, E-Stop, and Mode. Upon power up of the Nucleo, the system will be in a wait to start state until the operator presses the start button. Once the start button is pressed, the system is ready to run (assuming no faults have been found based on the systems check). The soft-stop button simply cuts power to the microcontroller, which would halt any functions currently being run and is a simple and safe way of ensuring the system stops. The E-stop button disconnects power to the entire system (board, motors, servos, etc) resulting in an immediate stop to all current through the model. The mode button allows the operator to run through a preset number of modes within the microcontroller and drive it accordingly.

The operator is able to get feedback from the model and microcontroller via the LCD screen. The screen is able to display, the current input RPM as well as when to shift, output “speed”, current gear, mode, most recent shift speed, and any possible electrical failure/faults that may arise. The

screen allows the operator to better control the system and provides a near real time display for the state of the model.

To drive the model, the operator has an acceleration pedal which takes in the operators desired throttle response, converts it to an analog signal for the microcontroller, allowing it to then be used in the motor output. The pedal is an OEM automotive pedal, using with an internal potentiometer and spring return to alleviate the manufacturing.

The entirety of the UI hardware (buttons, LDC screen) is in a 3D printed enclosure to allow for easy packaging and use during testing. The UI housing also contains the microcontroller and daughterboard to ensure safety during testing.

5.i Software Design

To properly design the control system, the code used will first need to be designed itself. Code for a control system is typically designed using two key tools: Finite State Machine (FSM) State Transition Diagrams and Task Flow Charts. After this, the actual code can be written using the framework created by these two tools. For this project, the code will be written using C++ and Microsoft's Visual Studio Code integrated development environment (IDE).

A Finite State Machine is a method of building software tasks, where a certain task contains a certain number of states. Each of the tasks in this project was first designed individually. Typically, this is done using State Transition Diagrams. First, each state for the specific task was designed described and stated. In the example shown in Figure 5.18, the button task used to acknowledge button presses is shown.

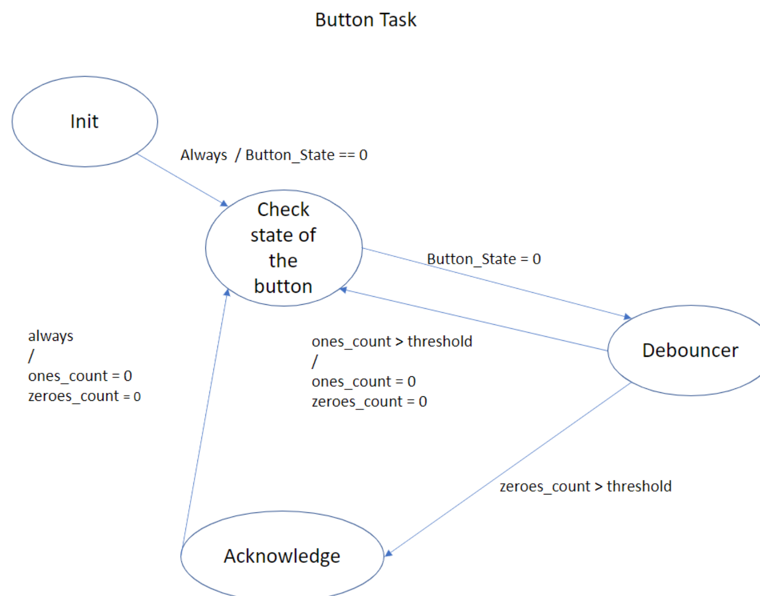


Figure 5.18 Example of State Transition Diagram for Button Task

In this example diagram, the task will transition to the next task so long as the correct conditions are met. For example, the task will move from State 1 to State 2 if the flag Button_State is set to 0, which in this case corresponds to a potential button press. Figure 5.19 shows another example, this time of a task that controls the input motor speed.

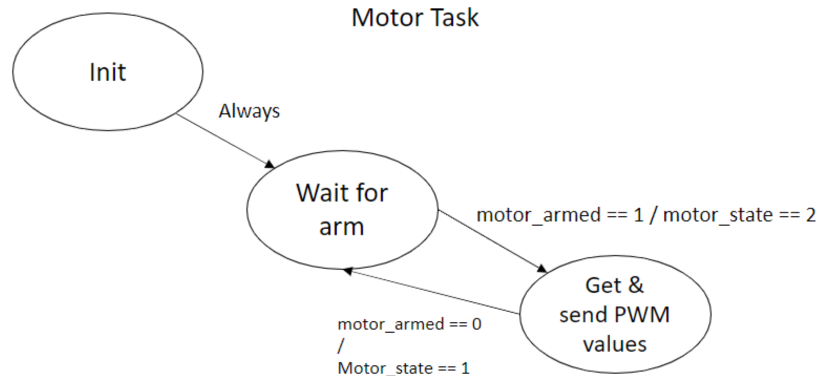


Figure 5.19 Input Motor Speed Task State Transition Diagram

This task has only 3 states: an initialize state, a waiting for arm state, and a change speed state. Essentially, after the first initialization (which declares all the necessary variables and sets them to their proper startup values), the task simply waits for the flag "motor_armed" to be set to 1. This flag indicates whether or not the motor's electronic speed controller (ESC) has been powered on and armed. Once done, the motor task simply looks into what is known as a "Share" to find the current PWM value that should be sent to the motor. These PWM values can come from either the user's inputs on the pedal or from the shifting gears task. If the "motor_armed" flag is ever lowered (most likely in the case of a power cycle or emergency off), the motor will go back to the waiting for arm state. State Transition Diagrams like this have been created for each task of the control system.

Two team members, Austin Conrad and Zack Gordon, used the opportunity of being enrolled in ME 507 Control System Design to develop the control system for this project. Part of this class involves making the above diagrams for the class term project. By making the term project for ME 507 the control system for this project, team members had access to all the resources and advice from the instructor, Dr. John Ridgley.

The software design also includes algorithms for things like motor control, button press, and mis-shift. The mis-shift algorithm checks to see that the shift request does in fact complete in the time allotted, if the dogs have not completed their shift, then the corresponding limit switch will not have been engaged. In this case, the microcontroller instructs the servo to move the pushrod back to center and try to re-engage the dogs again. The tuning of all of these algorithms was completed through testing and implementation of the controls model gains obtained through iteration. Further testing information will be discussed in section 7.

The full code for the project has been included in this report and attached as Appendix H.

5.j FMEA

This system, from the ground up, was designed to be a scale model for the purpose of controlling a transmission automatically. The flywheel was picked specifically to represent a scaled equivalent of a real car's inertial resistance to angular accelerations. The input motor, similarly, was picked to operate in the 8000 RPM range, but with a torque scaled down for the model size. The servos were picked based on the design requirements and the mass of the systems they were to move, thus guaranteeing their capability to move within the required specifications. Lastly, the control board and language were picked specifically to minimize control logic cycle time, with a processor that is capable of reading within a microsecond, so control times should be negligible, allowing for the controller to make decisions and issue commands more than fast enough to make the specifications.

Each individual component has an individual mode of failure; however, all failures fall under 4 categories: motor damage, gearbox damage, damage to the system outside the gearbox, and control system failures.

The first category is motor damage. These are failures that could burn up or otherwise destroy the input motor. This could be caused by an unexpected change in output speed, or through power spikes in the control circuit. To combat unexpected changes in motor speed, a shear coupler is being added to the motor's output shaft. This shear coupler will be rated to break at two- to three times the maximum motor output torque. Thus, any event that would cause an unexpected change in speed (thus, applying a torque) that the motor is not equipped to handle, the shear coupler will break and save the motor. For overcurrent protection, there is a fuse in the electrical system rated slightly higher (50A) than the maximum current limit of the motor (45A).

The second category is gearbox damage. Any sort of mechanical damage to gears, the box that houses them, and the shafts they spin with is considered "gearbox damage". This is considered unlikely, due to the fact that the gears were taken from a motorcycle's transmission. However, in the event that the gears bind, fracture, or otherwise misbehave, the system will have a shear coupler on either side of the gearbox. Should the gearbox bind while the motor is attempting to deliver power, the input-side shear coupler will fail, whereas if the gears bind while at high speed, the inertia of the flywheel will break the output coupler. Should any component undergo any sort of rapid kinetic disassembly, the gearbox will be within a box that should dissipate most of a projectile's energy and the entire system will be surrounded by a polycarbonate box.

The third failure category is physical damage to the system outside of the gearbox. This would include any sort of mounting failures, flywheel destruction, or damage to electronics. These failures are considered unlikely, since these components consist of mounts, circuit boards, and sensors. However, all of these are encased in the polycarbonate box, so that if these components do fail, they will not harm bystanders. Most of these would only be damaged by debris from a gearbox failure, and therefore, safety concerns with these components include a second-hand concern for the gearboxes.

Lastly, control system failures are a very serious concern. Any sort of controls logic failure could cause any of the above failures. For this, an e-stop is present. Should the controller ever make a

bad decision and cause potential damage to the system, the e-stop will simply cut power to the entire system. An example of a control system failure would be if the system tries to actuate both servos at once, which would bind out gear train immediately, causing damage to the gearbox and motor. Alternatively, if the servos get the command to shift before the rev matching is complete, this could also cause gearbox damage. Secondarily, there is a software-based logic to identify shift failures early. These include limit switches and RPM sensors that tell the controller if the system is ready to or has completed a shift.

Chapter 6 : Manufacturing

6.a Mechanical Manufacturing : Component Selection

With most of the mechanical assembly produced in CAD, the use of standard components was advantageous to produce the system. Utilizing the given CAD part files and drawings for each component on McMaster-Carr allowed for the proper fitment and function of each of the required components. The resulting model was dimensionally accurate and functional. This proper modeling of the system allowed for ease of production and assembly later in the build. Additionally, the maximization of standard part usage reduced production time and cost significantly. The vast variety of raw materials and documentation available from McMaster-Carr made it a primary source for component acquisition. The purchased and sourced components used is included as Appendix G.

6.b Mechanical Manufacturing : 3D Printing Prototypes

The transmission assembly required the production of several non-standard components using two main methods. The initial prototyping phase utilized 3D plastic FDM printing to prototype all specialty components in the system. This quick and cost-effective method of production allowed for fit up of the transmission components and hands on analysis of changes the design. This method also allowed for repeated changes to the design without significant cost or time consequences. The transmission was mocked up with these 3D printed parts and used to run functional tests. These tests were conducted at low speeds to ensure the safety of the system as the structural integrity of the printed parts was significantly reduced from the final production material. Additionally, the prototype was used to make modifications alter the fit and motion of the internal components if needed to ensure proper function. The 3D print was used to test and dial in the completely functional controls system before time was spent on production of non-standard parts in different materials.

Beginning the manufacturing process, the transmission components that were 3D printed included the upper and lower housings, shift forks, bearing mounts, and pushrod support. Seen in Figure 6.1, the components were printed using PLA material with 30% infill to allow for a structural part while conserving on material. The first test performed with the printed parts involved verifying the shaft spacing and gear mesh. It was found that the gear mesh was too tight and caused minor binding in the rotation of the system. This information was used to change the shaft spacing on the subsequent housing iteration.

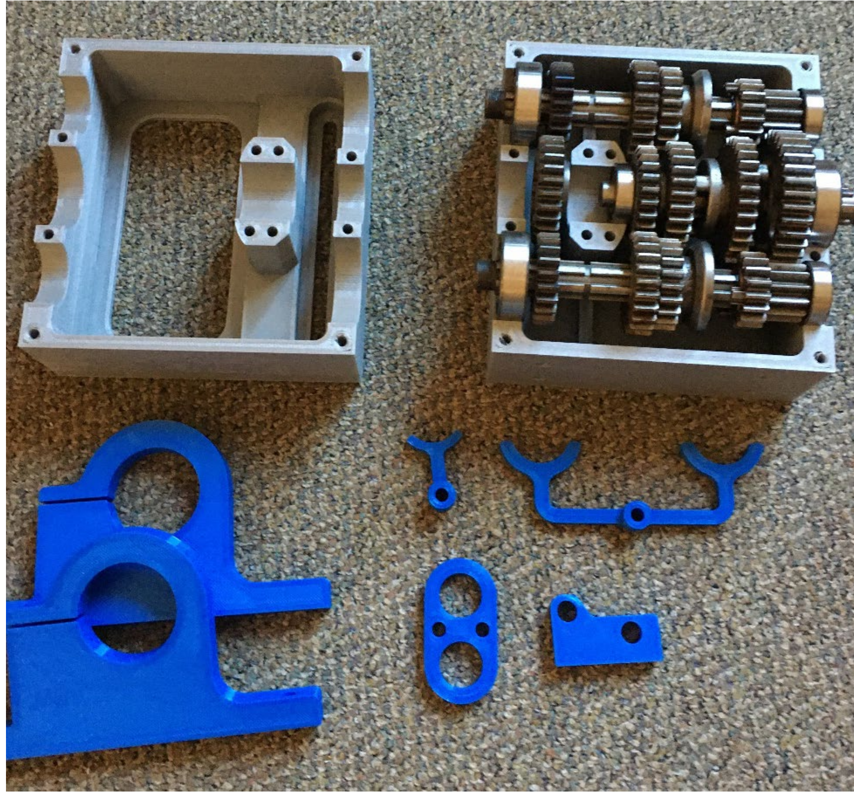


Figure 6.1 Completion of 3D Printed Models

Following the fitment of the main shafts, the pushrods, mounts, forks, and seals were installed. Seen in Figure 6.2, the interface between the shift collar and shift fork were closely examined to ensure proper clearance to reduce wear while maintaining a tight assembly. Additionally, the range of motion of the pushrods, forks, and shift collars was checked to ensure full range of motion was achieved for shifting. Lastly, the pushrod seals and interface were examined to confirm no transmission oil would leak during operation as the assembly is located below the oil level line.

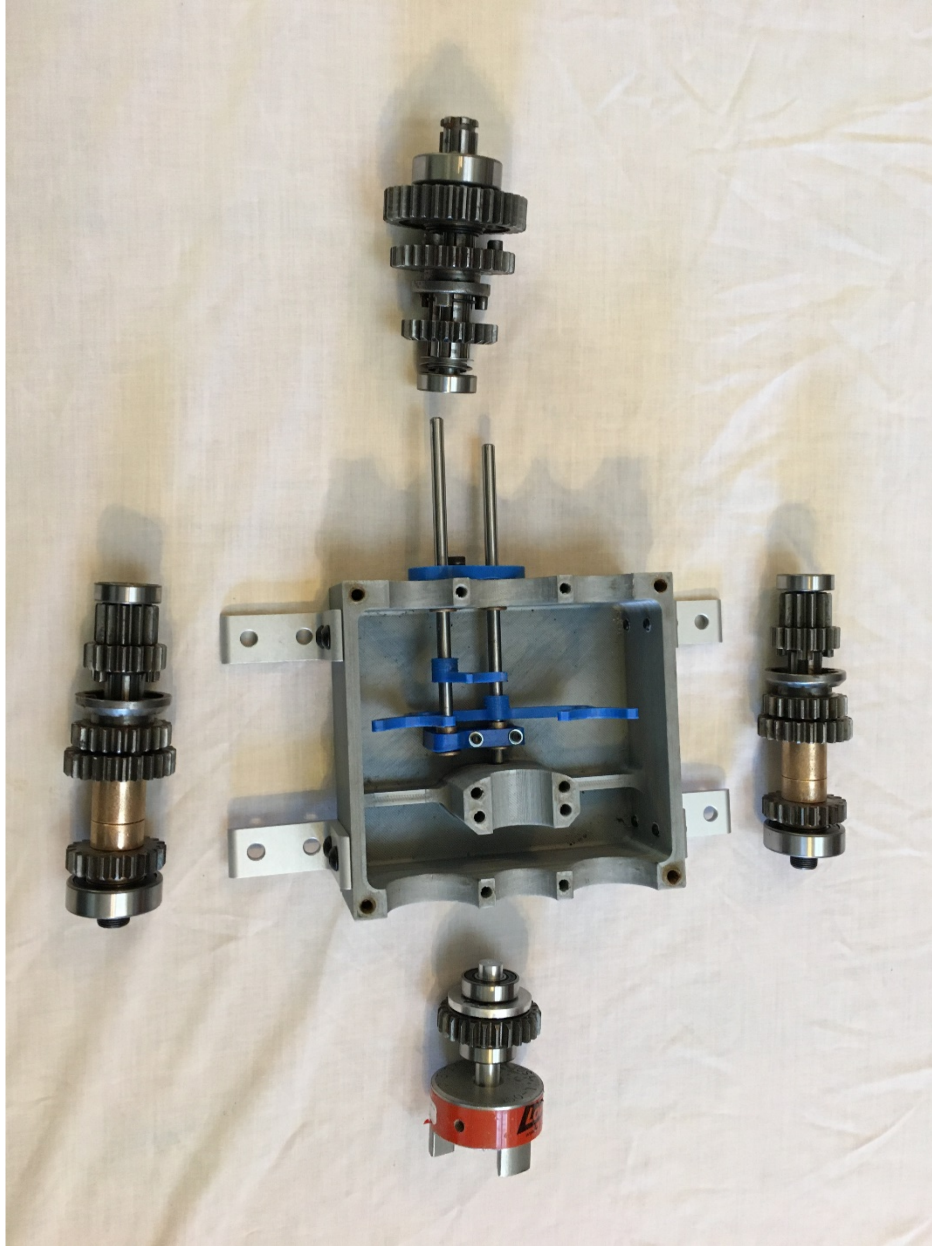


Figure 6.2 3D Printed Mock-Up with Shift Forks

The final step of the prototype was to lay out the entire system. Figure 6.3 shows the three main subassemblies, which include the drive motor, transmission, and flywheel. At this point, all shafts and subassembly joining components were manufactured and tested. A preliminary run of the system was performed and included rotating the transmission while shifting using the drive motor. No external sensors or feedback was used for this test. The test run confirmed the ability to shift the transmission under load and all components were working in harmony. A small roadblock was hit with this 3D printed system. Due to the relative inaccuracy of the 3D print, shaft spacing and alignment for the main shafts in addition to the shift rods was compromised and resulted in binding. Efforts were made to resolve this problem, but the malleable and inconsistent nature of the PLA

plastic made it exceedingly difficult. After progressing as far as possible with the 3D printed components, it was decided to begin machining the components out of aluminum.

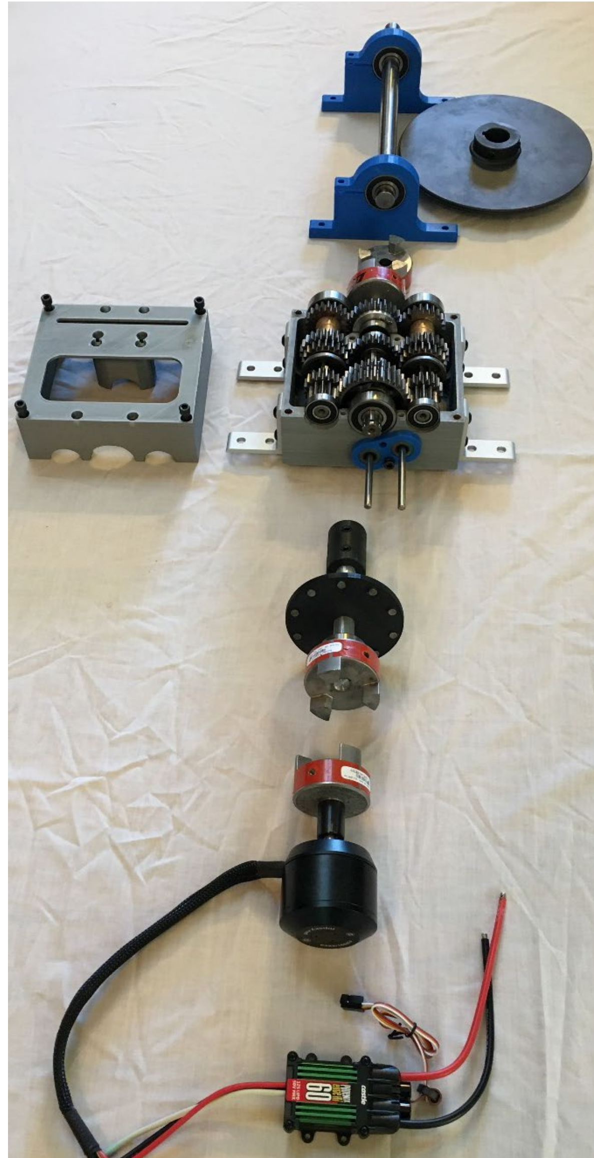


Figure 6.3 Layout of the 3D Printed System

During the initial testing of the 3D printed transmission prototype, the friction in the system was significantly higher than previously calculated. This substantial increase in friction resulted in an under sizing of the drive motor and ultimately ruined the drive motor from excess current draw. Taking the new drag into consideration, a new power system was chosen. The new motor was rated at 4000 watts at 48V. Additionally, the new ESC used was rated for 48V at 130 amps. Testing the system with this new power combination gave a better result while testing. The input power to drive train resistance was considerably closer to the ratio that was expected during theoretical calculations.

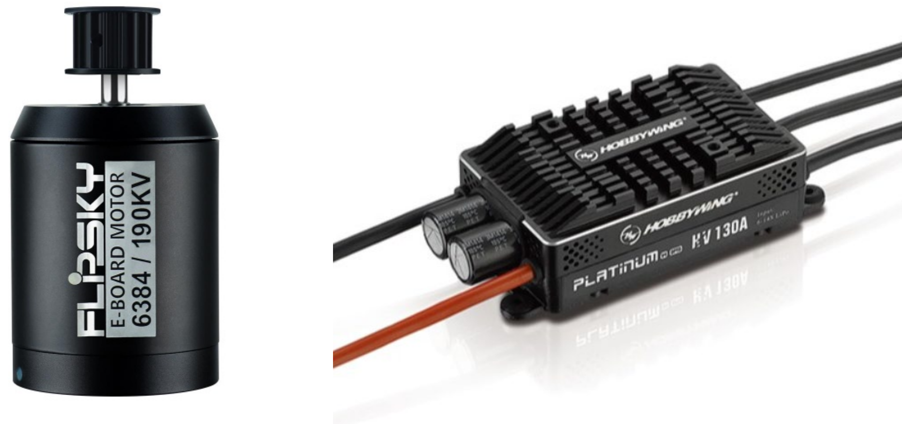


Figure 6.4 Updated Drive System

6.c Mechanical Manufacturing : Final Product Manufacturing

Upon completion of the functional 3D printed prototypes, the next step was to machine major load bearing components of the model from aluminum. These components consisted of the transmission upper and lower housings, shift forks, motor mount, and flywheel bearing supports. The primary option to acquire these parts was to have them machined in the IME machining lab. Access to the Haas VF2 and TL1 with accompanying tooling was going to be used to construct the parts as it keeps the manufacturing in house and costs down. This plan was contingent on the school and shop access dictated by the current pandemic.

Ultimately, access to the machine shop and school resources was not possible, so the decision was made to manufacture the parts using home machine tools. The intermediate shafts in addition to the output recombining shaft were manufactured on a lathe. Figure 6.5 shows the engineering drawing used to manufacture the part, while Figure 6.6 show the completed shaft assembly.

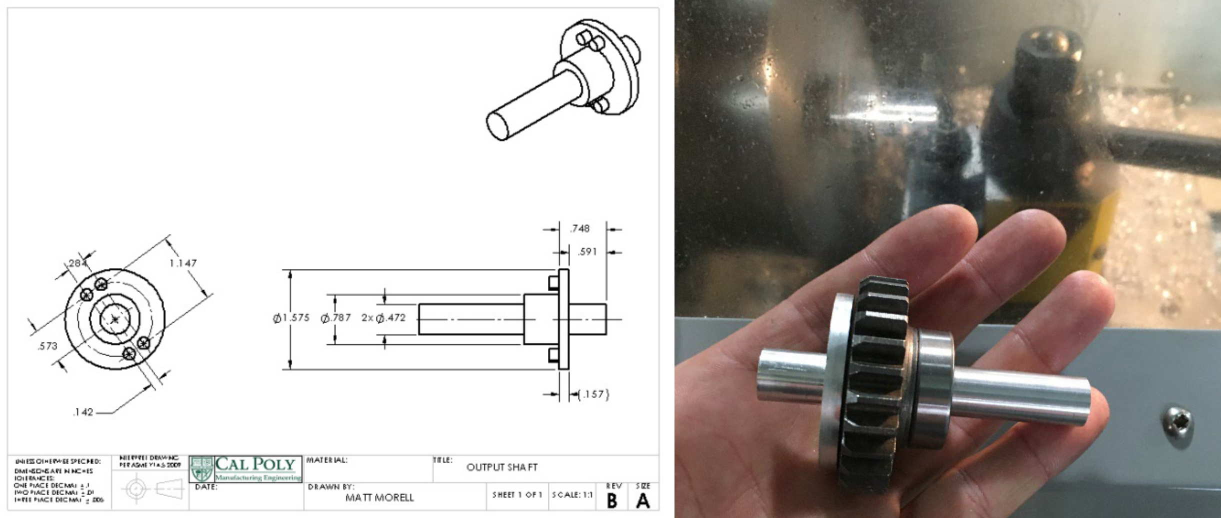


Figure 6.5 Recombining Shaft Drawing and Machined Part

Figure 6.6 Completed Recombining Shaft Assembly

A significant portion of the manufacturing was attributed to the upper and lower housings. This was a large undertaking to machine due to the reduced machine access. The housings started out as 6061-T6 aluminum blocks that were machined to outer dimension size as seen in Figure 6.7. Full sized drawings were attached to each billet block and used as a pattern to rough out the material by hand on a manual milling machine. This was done to reduce the amount programming and CNC machine time needed in subsequent operations. The roughed out upper and lower

aluminum housings can be seen in Figure 6.7. Two sets of housings were manufactured as a redundancy in case there was a manufacturing flaw with one set.

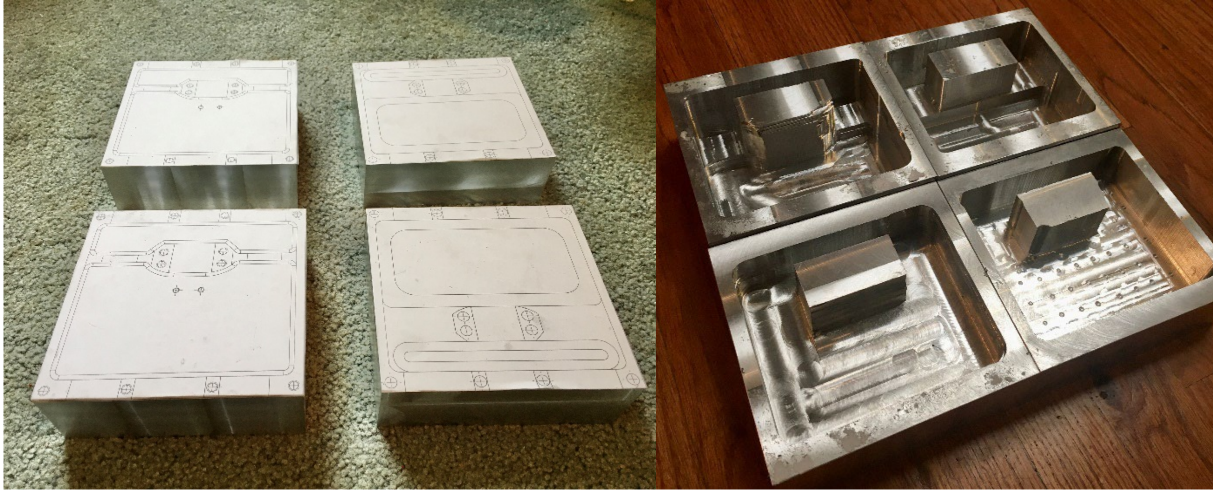


Figure 6.7 Aluminum Raw Stock for Housings and After Manual Machining Work

Machining the upper and lower housings to final dimension was accomplished with the use of a benchtop milling machine converted to CNC. All machined parts were first programmed with a Computer Aided Machining program (CAM) to create the necessary cut patterns and steps necessary to create the finished parts. Figure 6.8 shows the CNC milling machine setup used to manufacture all the aluminum parts for the project, while Figure 6.9 shows the result of the created tool paths for the upper case and shift forks.

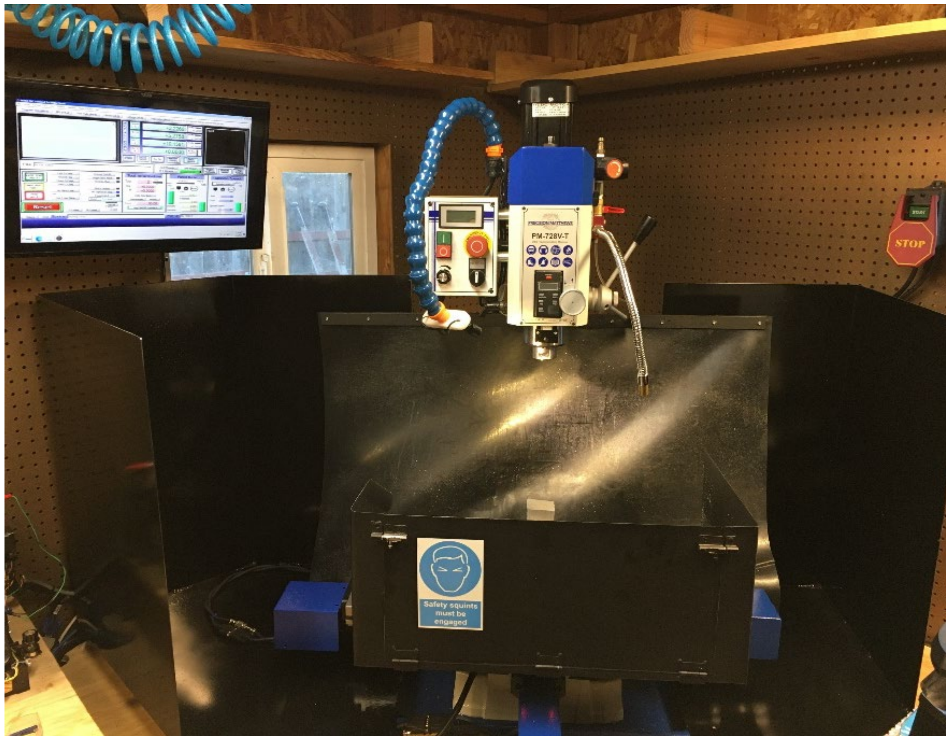


Figure 6.8 CNC Milling Machine Used for Manufacturing

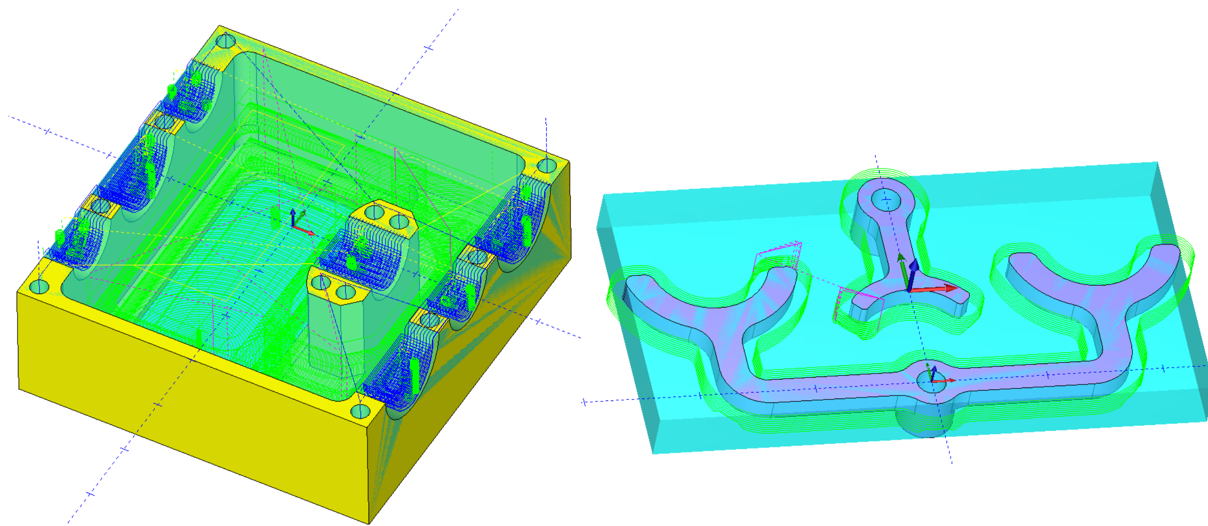


Figure 6.9 CAM Programming for CNC Tool Paths

Seen in Figure 6.10, engineering drawings were created for each part and used to machine the components in addition to providing dimensions for part inspection. Critical features were noted and GD+T callouts were used to ensure proper function of the part after manufacturing. The most critical features required were the shaft spacing and bearing surfaces. Proper shaft spacing allowed for the required gear mesh in the transmission in addition to keeping the shafts parallel to each other and perpendicular to the housing. Furthermore, the surfaces to which the shaft bearings seated needed to be a critical diameter to allow for the bearing to be easily installed while retaining it firmly when the transmission halves were clamped together.

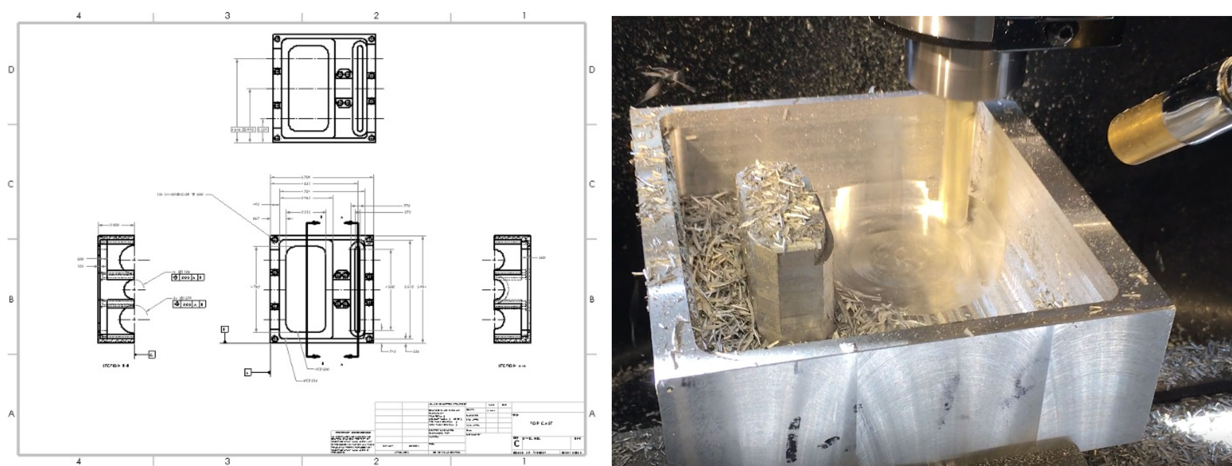
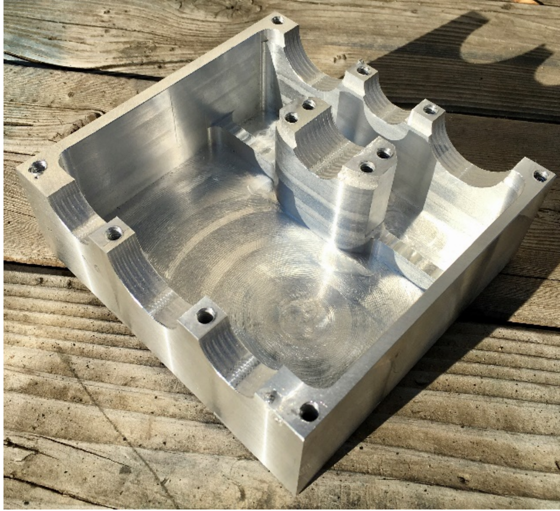


Figure 6.10 Upper Case Drawing and Final Machining of Housings on CNC Mill



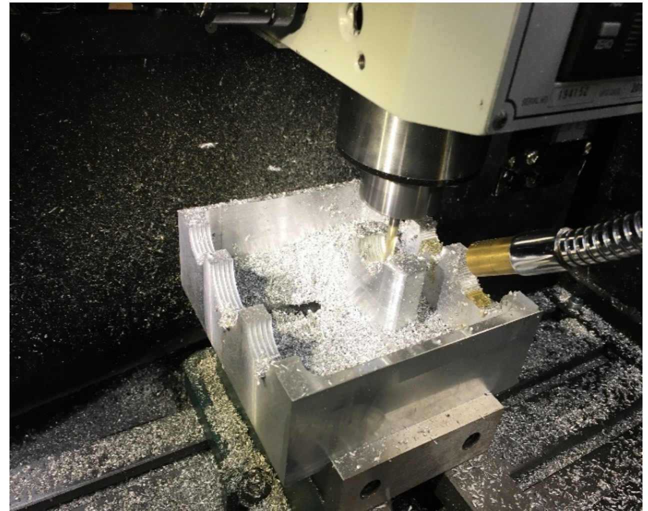
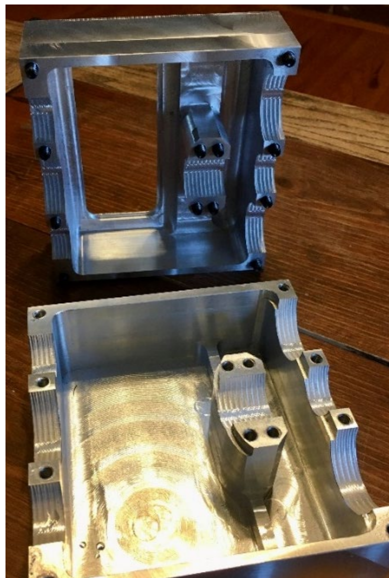


Figure 6.11 Machining of the Lower Housing



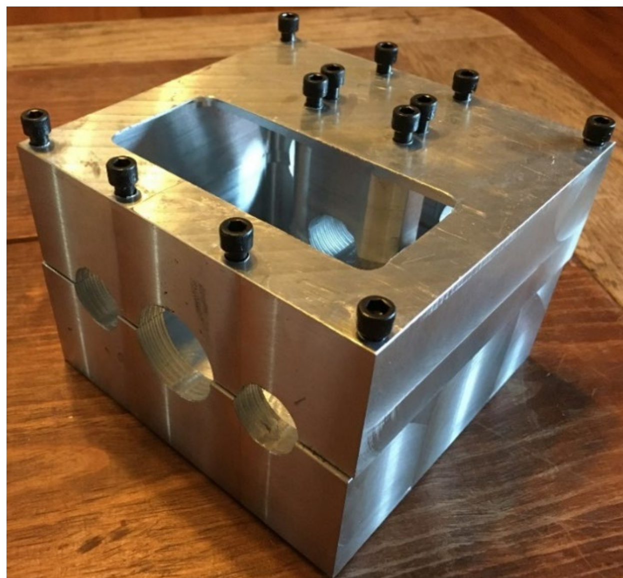


Figure 6.12 Completion of Upper and Lower Housings

As stated previously, the essential bearing bore diameter was achieved with a boring head on the milling machine. The bores were created after the two housing halves were finished machined. The halves were then bolted together and machined as one to create a perfectly concentric bore. Additionally, .007" shim stock, which can be seen in Figure 6.13 was inserted between the halves

to allow for a gap between the halves while boring the bearing surfaces. This allocates space for the transmission housing to clamp onto the shaft bearings and retain them in place.

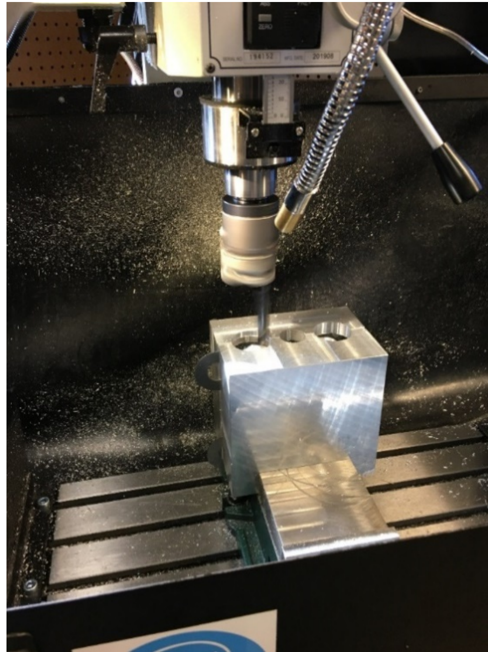


Figure 6.13 Boring the Bearing Surfaces into the Housing

The next major step in the manufacturing process was the installation of the shift rods and forks. The placement and precision of this element was critical to the proper function of the transmission. The interface between the shift collars and the shift forks needed to have a clearance fit while reducing as much free play as possible. Additionally, the shift rods needed to actuate parallel to the rotating shafts while moving smoothly. The placement of the shift rod holes in the housing and rear support were positioned theoretically in CAD. When this was tested on the 3D printed model, there was significant binding of the rods that inhibited their operation. With the final aluminum model, a modified approach was taken to eliminate this problem. First, the busing holes were accurately drilled in the housing and rear support. Once the bushings were pressed into each, the rear support was then mounted into its permanent location. The housing assembly was then mounted in the mill and the shift rod bore was milled in in one operation, ensuring perfect hole alignment. This operation can be seen in Figure 6.14. Additionally, the finished pushrod assembly can also be seen mounted in the lower housing.

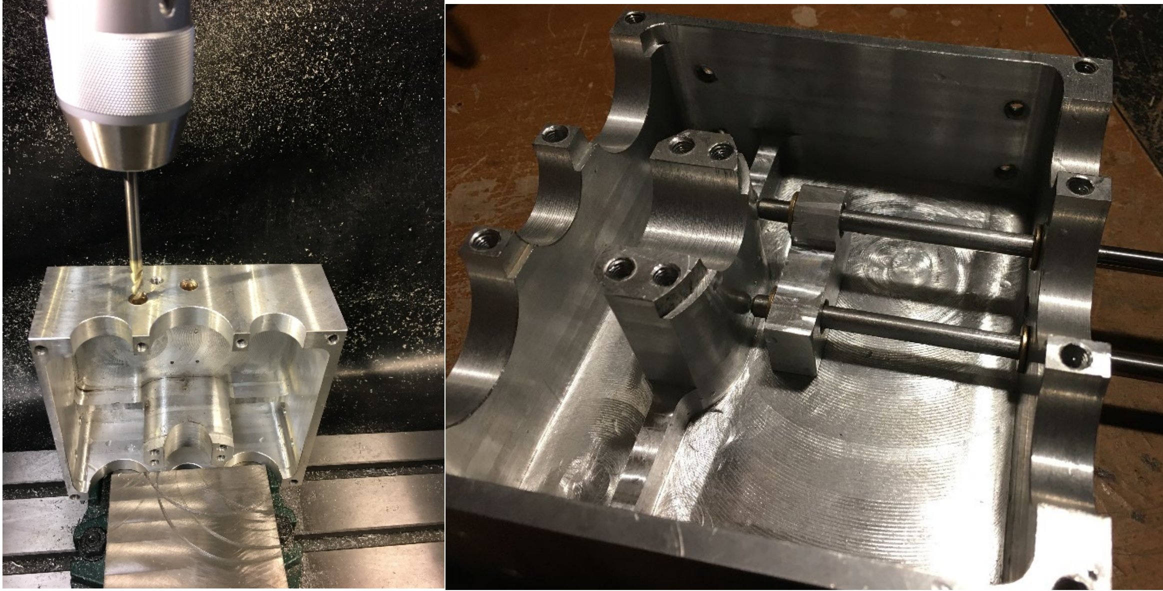


Figure 6.14 Machining the Pushrods into the Lower Housing

The completed shift assembly can be seen in Figure 6.15. With the improvements mentioned previously, the shifting motion was smooth with very little friction. Additionally, the shaft seals can be seen mounted to the case to prevent transmission oil from exiting the housing.

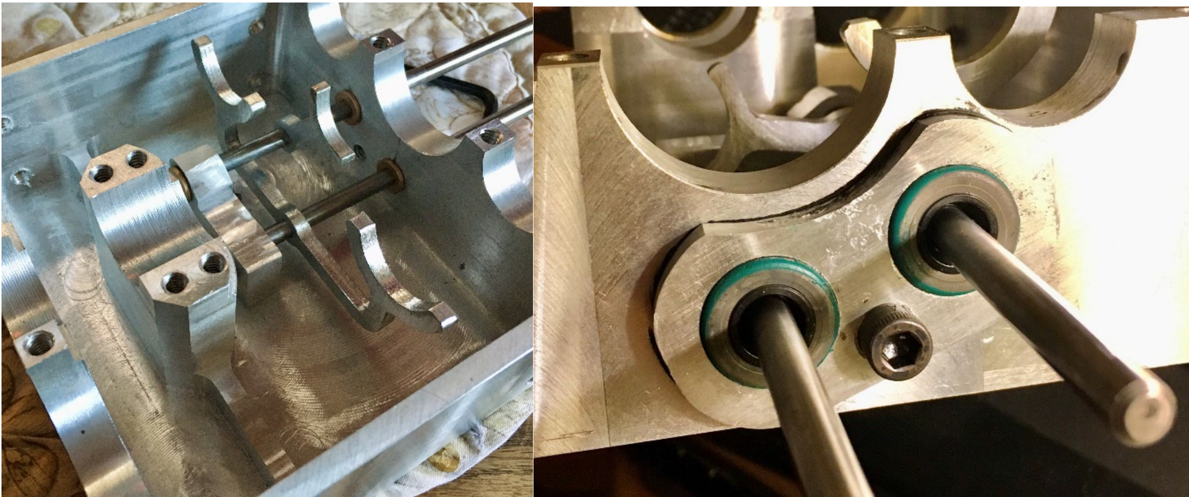


Figure 6.15 Finished Pushrod Assembly and Shaft Seals

The final manufactured part internal to the transmission were the bronze gear spacers. Seen in Figure 6.16, the oil-lite bronze bushings were used properly space the gears apart on the shaft while providing an adequate wear surface. The final step was to install the gear train into the housing and bolt the halves together. Initial testing was done to ensure that the gear train rotated freely and the shifting assembly functioned properly.

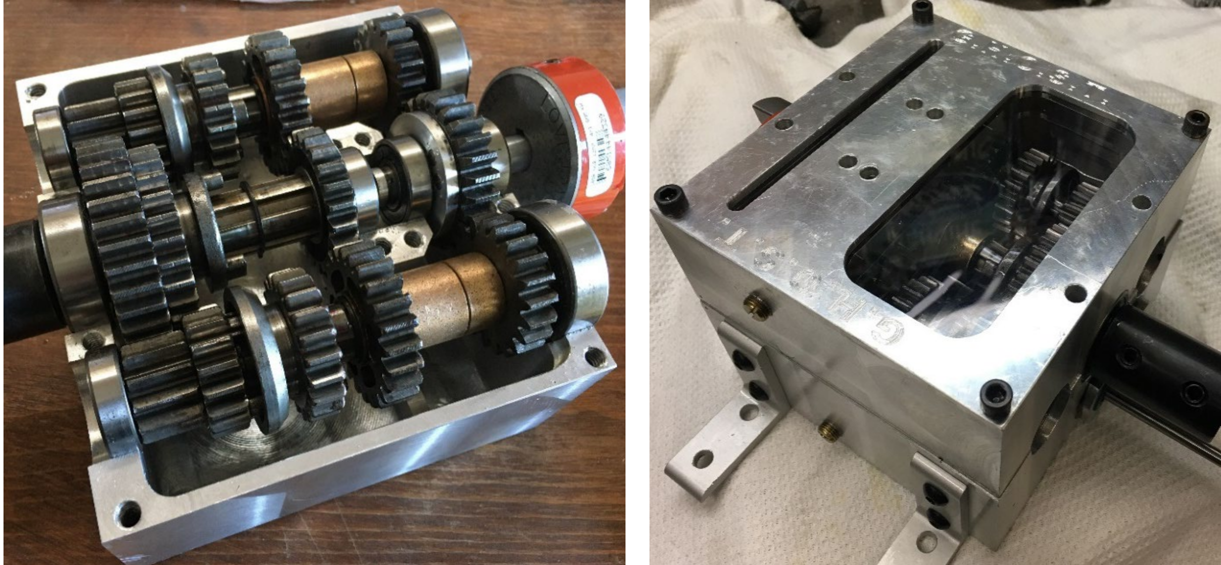


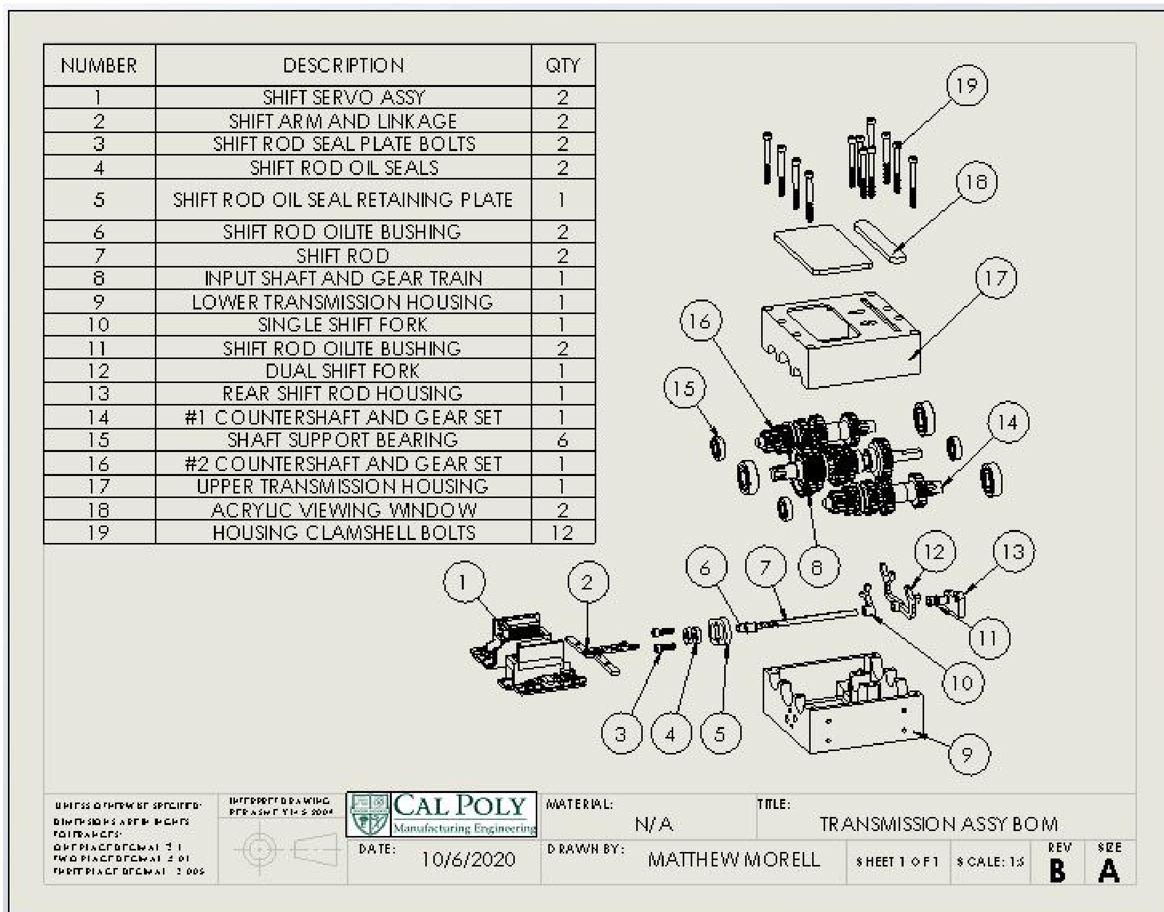
Figure 6.16 Initial Fit Up of Gear Train in Housing

Before the installation of the plastic viewing window, tests were performed to ensure the structural ability of the material chosen. With the possibility of a gear tooth or other small components dislodging from the gear train at high speed, acrylic plastic was chosen as it is used in the similar application of ballistic shields. To ensure the safety of the operator, a very generous .5" thick acrylic was chosen. To test this plastic, a sample was shot with a pellet gun. The lead pellet of weight .547g that was shot around 1200 FPS closely resembled a worst-case scenario of a gear tooth dislodging at double the max transmission rated RPM. The resulting test shown in Figure 6.17 revealed that the plastic was more than sufficient to withstand the impact. The impact left a divot of approximately 1/32" in the ½" thick acrylic.

Figure 6.17 Testing of Viewing Window Plastic

6.d Mechanical Manufacturing: Final Product Assembly

The majority of the components in the system utilize bolted connections. This method of joint was used for its simplicity in addition to its ease of disassembly. The transmission halves are held together using 12 x ¼-20 bolts as seen at item #19 in Figure 6.18. Additionally, other components such as shift forks and pushrod seals are retained with bolted connections. Furthermore, the three main sub-assemblies of the system are secured to the wood mounting board using wood screws. The motor mount, constructed of aluminum, is the only non-bolted connection on the system as it is a welded assembly.



NUMBER	DESCRIPTION	QTY
1	SHIFT SERVO ASSY	2
2	SHIFT ARM AND LINKAGE	2
3	SHIFT ROD SEAL PLATE BOLTS	2
4	SHIFT ROD OIL SEALS	2
5	SHIFT ROD OIL SEAL RETAINING PLATE	1
6	SHIFT ROD OILITE BUSHING	2
7	SHIFT ROD	2
8	INPUT SHAFT AND GEAR TRAIN	1
9	LOWER TRANSMISSION HOUSING	1
10	SINGLE SHIFT FORK	1
11	SHIFT ROD OILITE BUSHING	2
12	DUAL SHIFT FORK	1
13	REAR SHIFT ROD HOUSING	1
14	#1 COUNTERSHAFT AND GEAR SET	1
15	SHAFT SUPPORT BEARING	6
16	#2 COUNTERSHAFT AND GEAR SET	1
17	UPPER TRANSMISSION HOUSING	1
18	ACRYLIC VIEWING WINDOW	2
19	HOUSING CLAMSHELL BOLTS	12

Figure 6.18 Final Product Assembly Exploded View

Following the assembly exploded view seen in Figure 6.18, the transmission was built with all of the manufactured components previously produced. Assembly started by installing the shift rods, bushings, rear support, and shaft seals into the lower housing. After ensuring the shift rods had a smooth range of motion, the shift forks were installed. Next, the input shaft, two countershafts, and recombining shaft were placed into the lower housing with their corresponding bearings. Before this step was complete, in important detail was required. As seen in Figure 6.18, the single second gear shift collar was located on the input shaft. Because there was only one collar to shift, no clocking of the drive dogs and corresponding gears was required. In contrast, first gear requires

moving two shift collars, one on each countershaft. This required the clocking of both sets of drive dogs in addition to clocking both mating gears on the shaft. This allowed for both shift collars to engage simultaneously when actuated. If this step was not completed and the dogs were not clocked properly, the transmission would never shift into first gear as the shift collars were mechanically linked together.

The last component that was installed on the transmission before closing the halves was the acrylic viewing windows. Seen in Figure 6.19, the ½" acrylic windows were seated into two pockets machined into the bottom side of the upper housing. The windows were manufactured to have a press fit into those pockets. The press fit in combination with a small amount of sealant ensured that the joint would not allow transmission fluid to escape during operation.

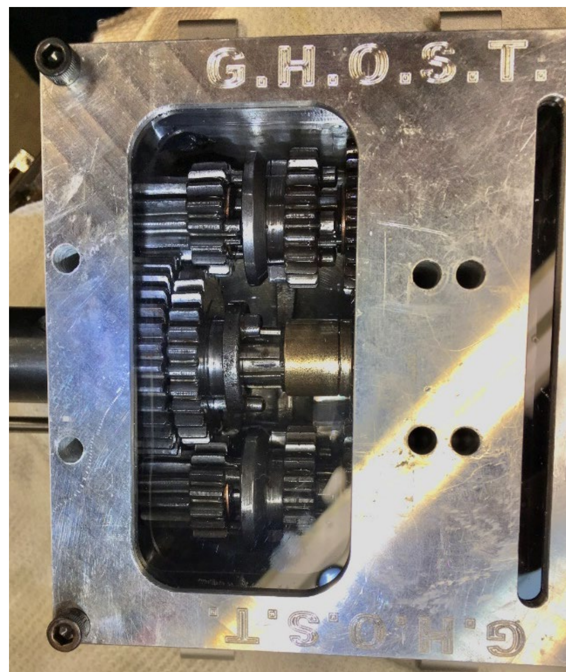


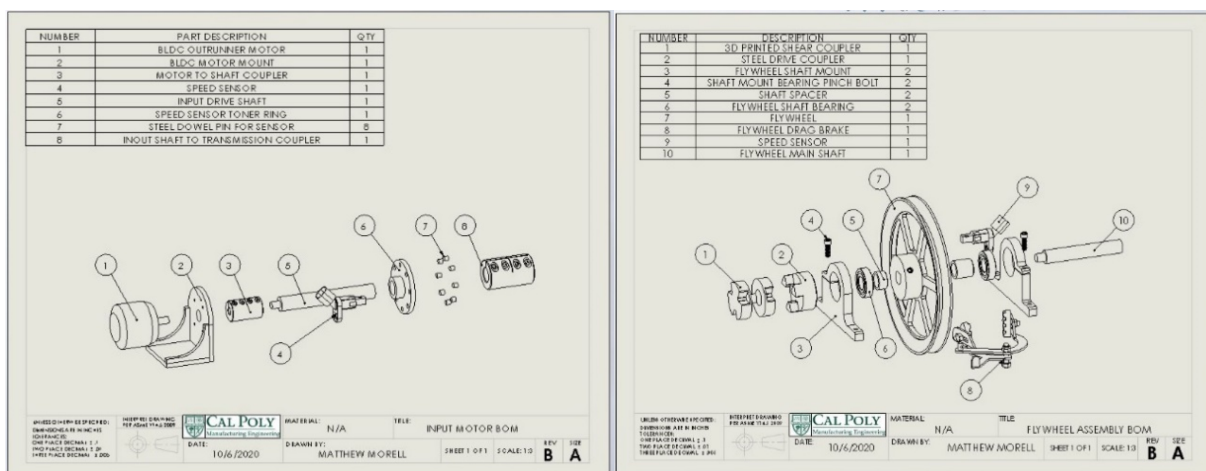
Figure 6.19 Final Assembly of the Transmission

From preliminary tests with the prototype and final transmission, it was determined that the previously selected flywheel was not sufficient for the system. The previously selected flywheel did not account for frictional losses and as such was too small to adequately hold output rpm during rev-matching. From the limited manufacturing resources available, aspects of the transmission were not aligned as perfectly as they could have been. One such example was the transmission housings. Such extremely tight tolerances of shaft alignment on critical bearing features was most likely not held. This caused a very slight shaft rotational resistance, and in combination with the other three shafts, resulted in excess friction. Additionally, the bearings that should have been used if in a production unit were hydrostatic bearings. These offer very little rotational friction to the shaft. Again, from the limited manufacturing capability and budget, traditional grease sealed ball bearings were used. The use of grease in the bearings did not allow for very high rotational speed of the bearings. The upper speed of 8000 RPM was just shy of the maximum rated speed. This

resulted in significant drag on the system from the viscous grease in the bearing. With this knowledge, a new flywheel was selected. The new flywheel, seen in Figure 6.20, resulted in a moment of inertia of approximately 0.1 kg-m^2 and was accomplished with the use of two Go Kart brake rotors.

Figure 6.20 Revised Flywheel for the Simulated System Inertia

The next steps that followed the completion of the transmission sub-assembly included building both the power system and flywheel sub-systems. The power assembly utilized the motor mount, speed wheel, and input shaft that were manufactured. These in combination with the purchased components and the new motor system completed the assembly. Similarly, the flywheel sub-assembly was composed of several manufactured parts, including the main shaft and bearing blocks.



NUMBER	PART DESCRIPTION	QTY
1	BLDC OUTRUNNER MOTOR	1
2	BLDC MOTOR MOUNT	1
3	MOTOR TO SHAFT COUPLER	1
4	SPEED SENSOR	1
5	INPUT DRIVE SHAFT	1
6	SPEED SENSOR TONER RING	1
7	STEEL DOWEL PIN FOR SENSOR	8
8	INOUT SHAFT TO TRANSMISSION COUPLER	1

NUMBER	DESCRIPTION	QTY
1	3D PRINTED SHEAR COUPLER	1
2	STEEL DRIVE COUPLER	1
3	FLYWHEEL SHAFT MOUNT	2
4	SHAFT MOUNT BEARING PINCH BOLT	2
5	SHAFT SPACER	2
6	FLYWHEEL SHAFT BEARING	2
7	FLYWHEEL	1
8	FLYWHEEL DRAG BRAKE	1
9	SPEED SENSOR	1
10	FLYWHEEL MAIN SHAFT	1

Figure 6.21 Building the Sub-Assemblies Per Drawings

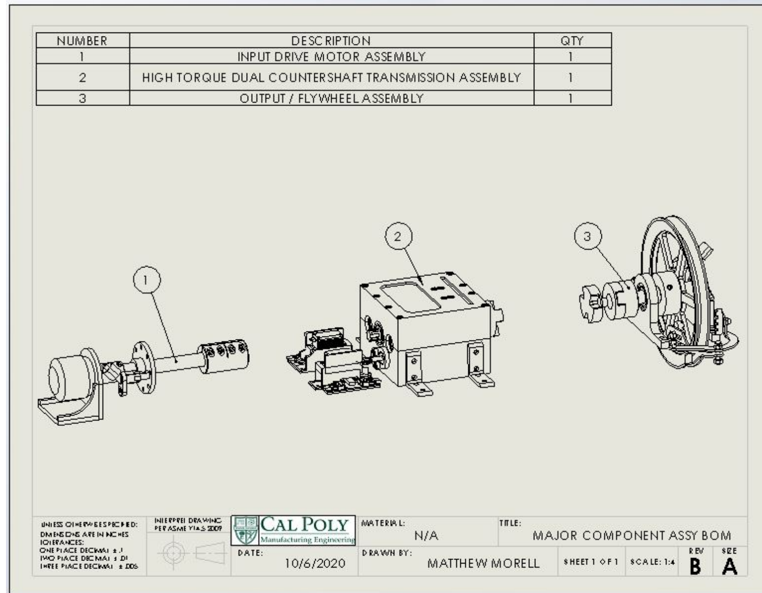


Figure 6.22 Assembly of the Three Main Sub-Assemblies

Figure 6.23 shows the final assembly of the transmission system. Here, the three major sub-assemblies were combined onto a central mounting board. Great care was taken to ensure that the shafts that connected the sub-assemblies remained parallel to each other to reduce binding and excess wear. Additionally, it can be seen in the Figure that two plugs were added to the side of the transmission housing. This was to allow for easy transmission oil fill and drain without having to split the housings apart and disassemble the transmission. A break in cycle was run to allow the transmission and all rotary parts to seat in. The rotating friction of the assembly was greatly reduced after this break in process. Subsequently, the precision and overall function of the transmission was greatly improved due to the structural rigidity and precision of the aluminum parts over the 3D printed plastic.

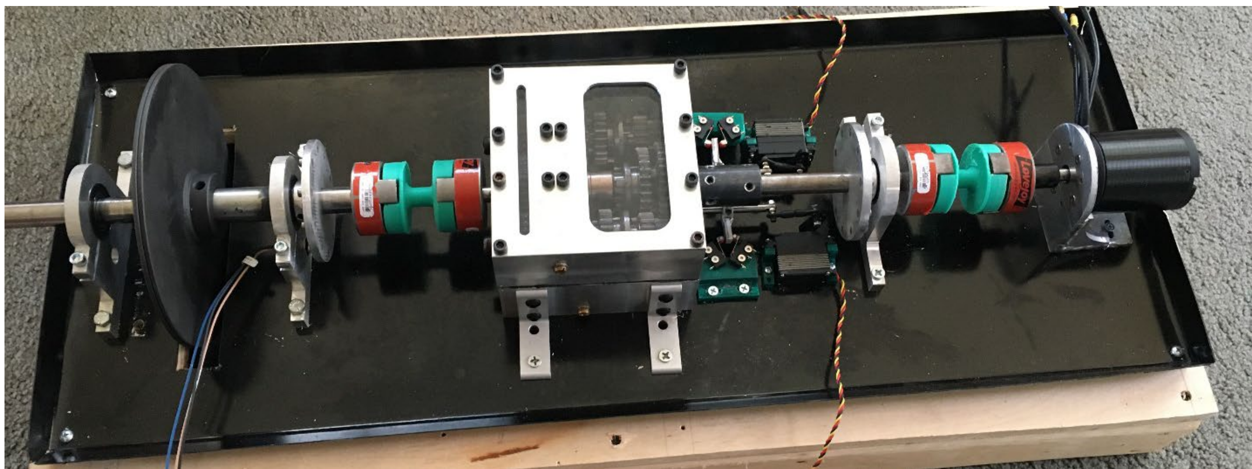


Figure 6.23 Final Layout of Transmission Assembly

With the three major sub-assemblies installed on the mounting board and the system running manually, the next task that was completed was the installation of the shift servo assembly. The initial design allotted for the shift completion switches to be integrated into the servo mounts. Upon completion of the overall layout, it was decided that the previous design would not work and a different approach was taken. Seen in Figure 6.24, the shift servos were mounted alone on a 3D printed mound and attached to the main board. From there, the shift limit switches were secured to a separate 3D printed mount. The mount itself had slotted mounting holes to allow the whole assembly to slide for fine tuning of the switch placement. Furthermore, each limit switch was mounted such that they could pivot which allowed for each limit switch trigger location to be adjusted. For actuation of the switches, a split collar with a welded tab was secured onto the shift rod. The resulting assembly allowed for the shifting of both first and second gear, in addition to the indication of whether the dogs were engaged or in neutral.

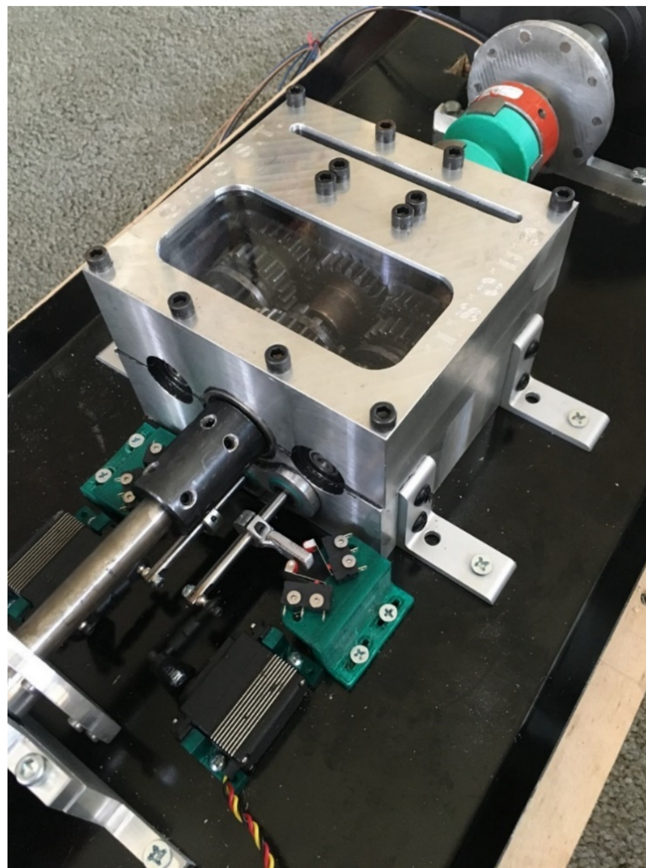


Figure 6.24 Transmission with Shift Servos and Gear Sensors

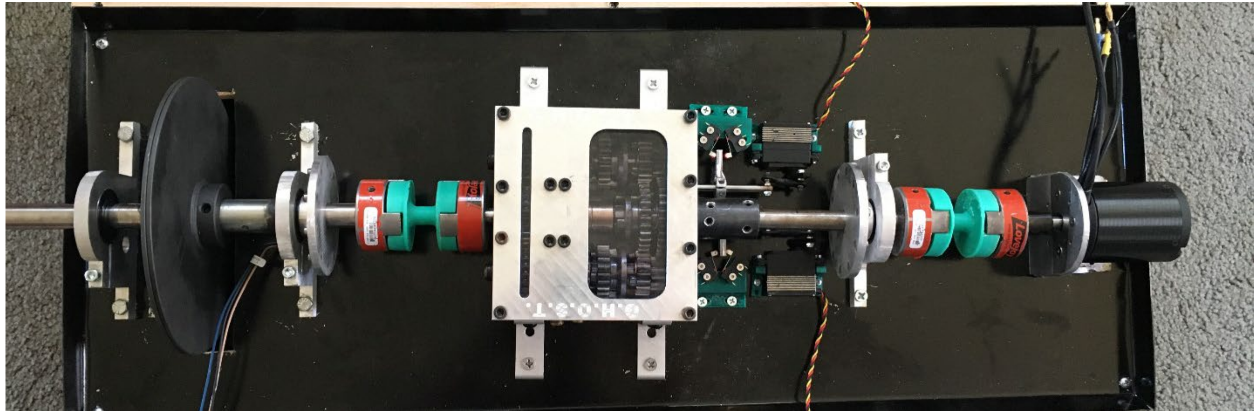


Figure 6.25 Top-Down View of Transmission Assembly

Before moving forward with the automated shifting of the transmission, the system was first checked for functional operation. The system was hooked up to a remote-controlled system, which allowed the operator to manually control the motor speed as well as the position of the two shift servos. Tests were conducted at various speeds which consisted of shifting the transmission into first gear, increasing the motor RPM, and then shifting into neutral. From there, the motor RPM was decreased to match the RPM required for second gear. The transmission was then shifted into second gear and motor RPM increased to the max operating speed. Additionally, the same procedure was conducted in reverse to check for the functionality of rev matching while downshifting.

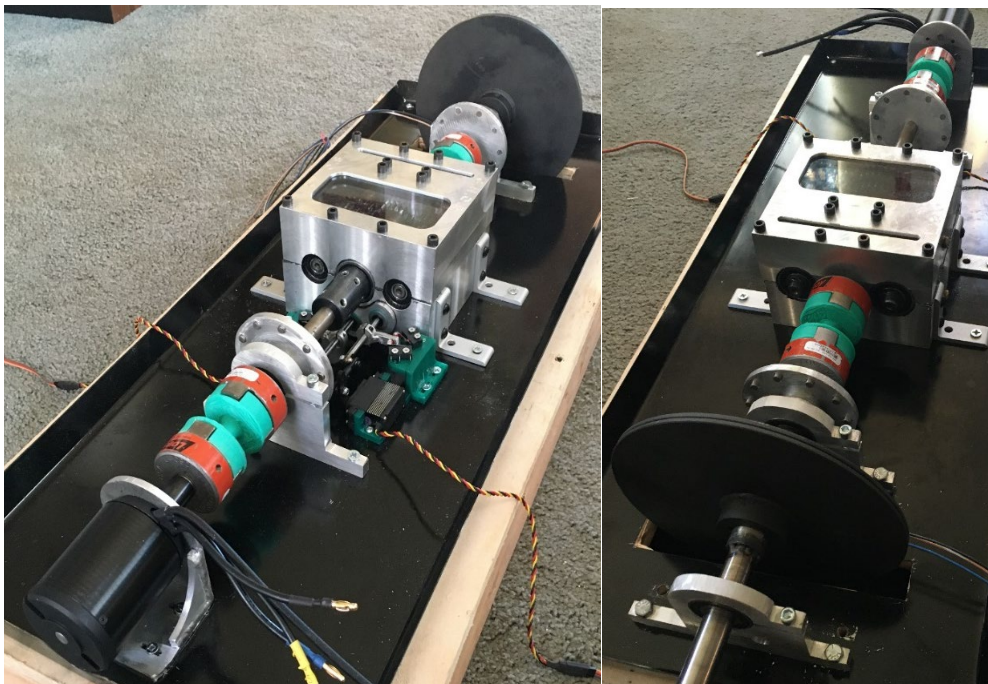


Figure 6.26 Input Motor and Flywheel Assemblies

The last manufactured component needed for the operation of the transmission was a means to request a shift. To accomplish this, a rally/race style “bump” shifter was constructed out of aluminum. This bump shifter was spring loaded with a ball detent in the center to allow for a bump forward for a down shift and a bump backwards for an upshift. A protruding tab at the base of the shifter allowed for the installation of two limit switches which converted the shifting motion to a useful electrical signal.

Figure 6.27 Custom Aluminum Shifter for Transmission

6.f Electrical Manufacturing

The electrical sub-system of the overall transmission first started with the requirements from the user and mechanical aspects. The speed sensing aspect of the design, for example, required a sensor that meet and exceeded the number of input pulses of the transmission to detect speed. Failing to meet the requirement would result in loss of pulses and a miscalculation of shaft speed. Sensor size and cost was also considered in the selection process. Other off the shelf input components, such as switches and buttons, were also selected based on the mechanical requirements of the system where they were located. From there, the main processing board was selected from the requirements provided by the manufacture of each input and output component previously selected. This included electronic interface type, input and output voltages, number of I/O ports, and operating speed and language. From the specified requirements, the STM32 L476RG Nucleo Microcontroller was chosen as the main control board.

A central hub for all incoming and outgoing data and power has been designed as a daughter board for the Nucleo. Eagle PCB design software was used to create circuit schematic diagrams. With the circuit verified, the PCB daughter board layout was created using the component size and pin requirements. The resulting daughter board layout was sent out to be manufactured by Oshpark, a 3rd party manufacturer. The physical board's connectors were subsequently assembled and soldered by hand.

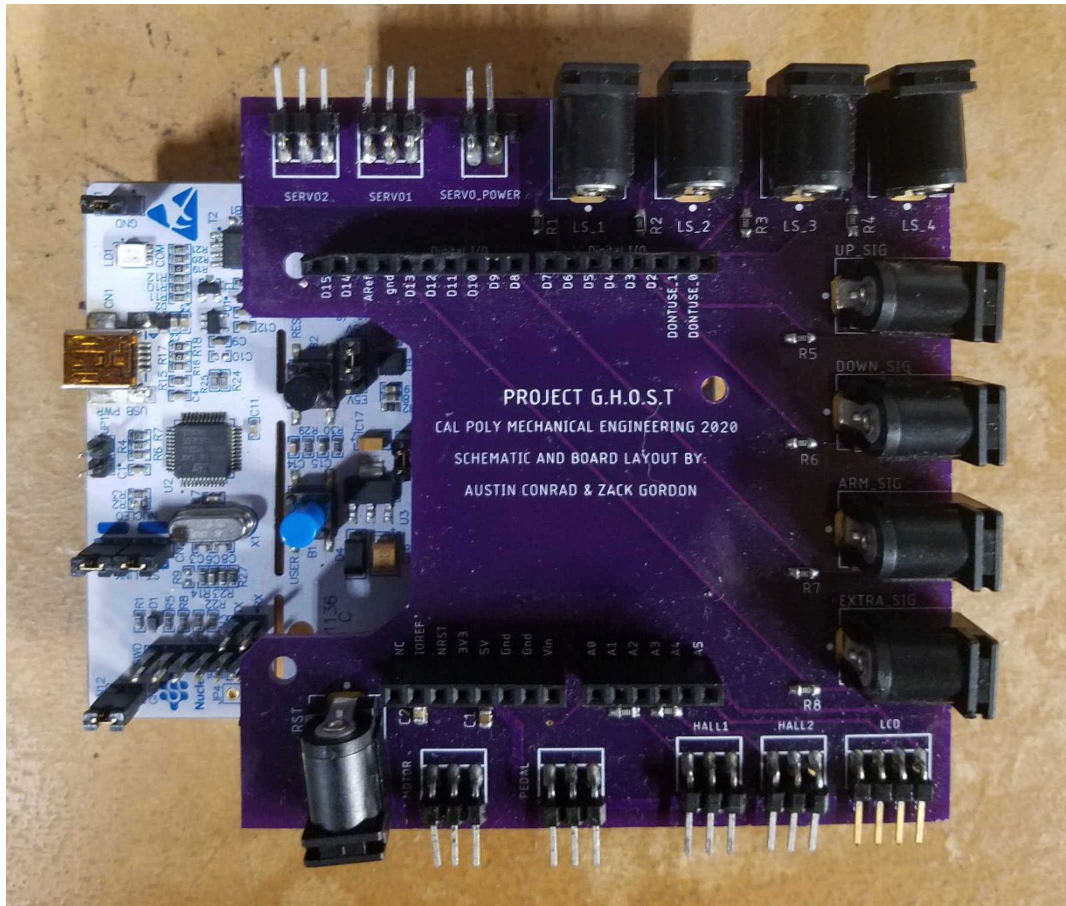


Figure 6.28 Daughter Board for STM 32 Nucleo Microcontroller

The board makes use of many barrel connectors for signal inputs from the various buttons and limit switches featured in the system. Although this may seem like an unnecessary use of space, the barrel connectors make plugging and unplugging wires much easier than with the typical servo connector pins. The 3-pin servo connectors seen on the board are used for Hall Effect sensors, the analog input pedal, the BLDC motor ESC, and the shift servo motors. A 4-pin header is used for the LCD screen.

The daughter board also feature connectors that are directly attached to the Nucleo's general IO pins. These connectors are the ones shown closer to the center of the board and have various uses such as acting as power sources and/or logic pins.

Upon completion of the daughter board, tests were performed to ensure that the Nucleo and daughter board were to properly drive the servos and motor as well as acknowledge the input

switches and limit switches. This was done using simple soldered wires and servo 3-pin connectors. After ensuring the mocked-up system worked correctly, proper wiring harnesses were made to length and outfitted with connectors to allow for serviceability.

For all buttons and switches, the required pins were soldered and heat shrink was used to help ensure no shorting. Each component was checked for continuity prior to final installation and after installation to also verify there were not hardware level faults prior to testing. Some of these components can be seen in Figure 6.29 below.

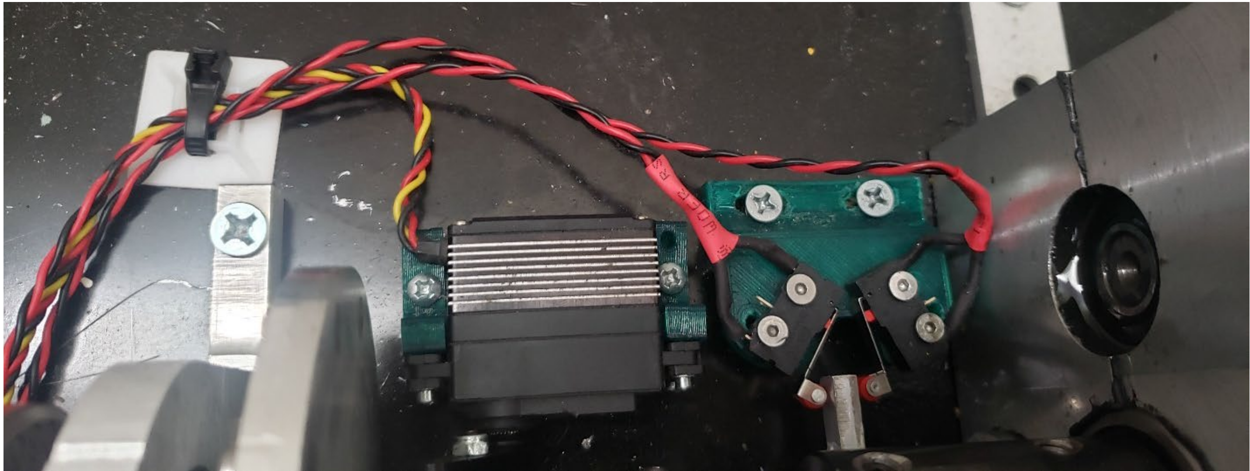


Figure 6.29 Figure of the soldered and wired-up Switches/buttons

To ensure electrical parts were not damaged or stressed during testing, adequate slack and cable ties were implemented to ensure ease of servicing while remaining out of the way during testing.

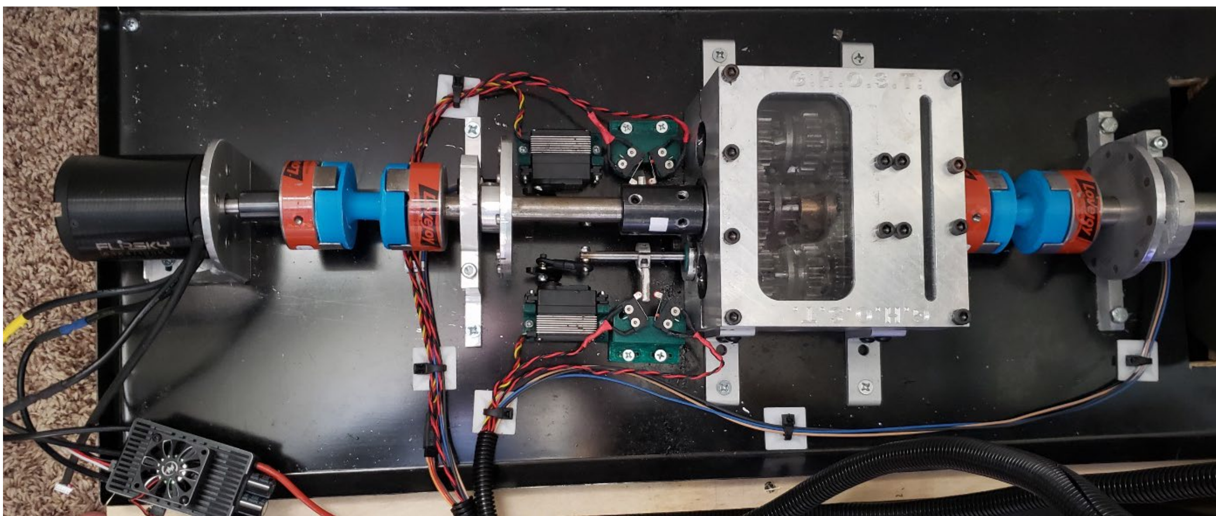


Figure 6.30 Final wiring on testing assembly

The batteries presented a point of safety concern due to the high current capacity. To ensure operator safety, the battery disconnect was positioned at the front of the UI such that it can be quickly switched off as needed. In conjunction with the disconnect, a battery health meter was used to check the state of charge throughout testing to ensure battery voltage never dropped below a safe state of charge. Both of these features can be seen in the figures below.

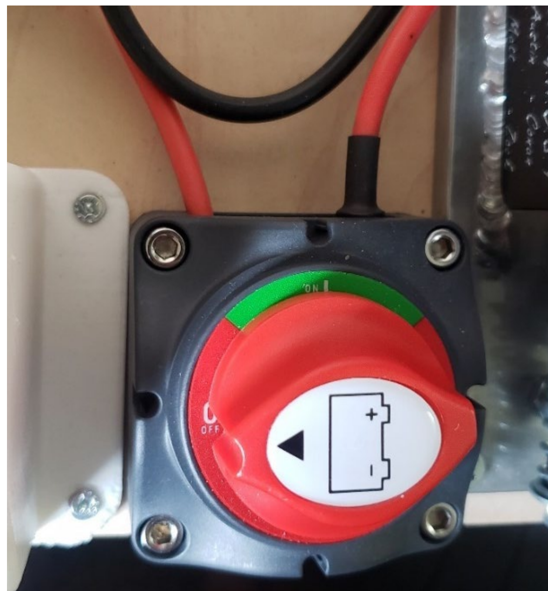


Figure 6.31 Construction of Emergency Power Cuttoff Switch

Figure 6.32 Main Battery Packs and Battery Monitoring

The connectors were all labeled prior to installation and wire loom as this helped to mitigate any testing errors due to simple mix match of the cables to ports.

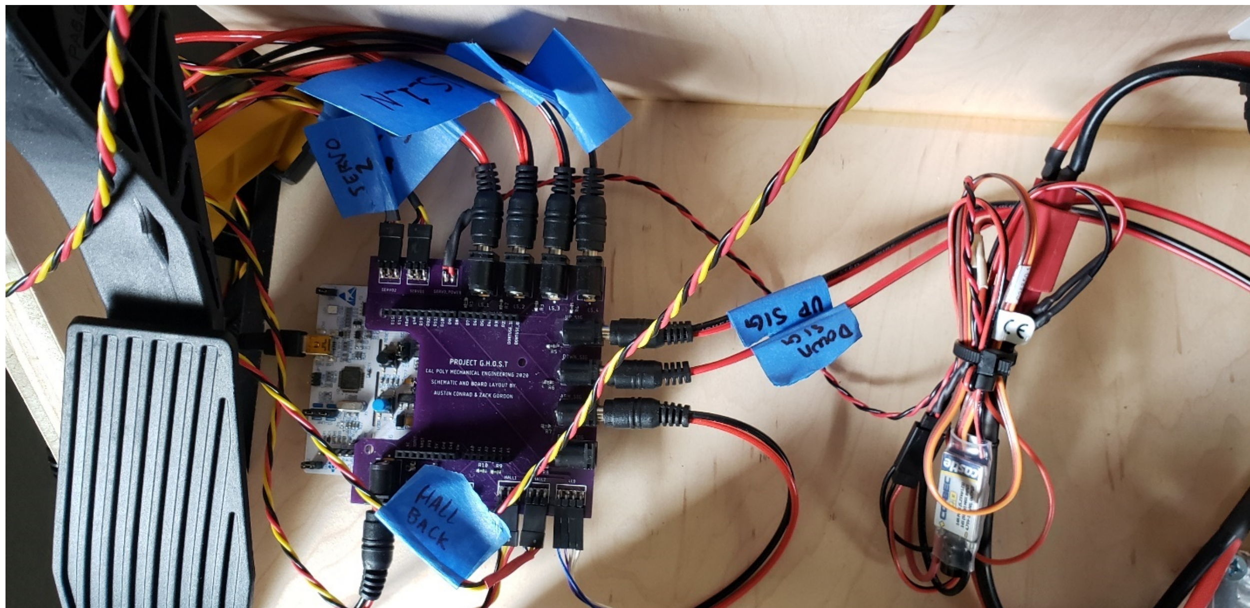


Figure 6.33 Final Wiring of Inputs and Outputs to Main Control Board

The billet shift selector was wired up with both buttons and mounted the UI on the right-hand side as the main operator was right-handed. The switches were checked for continuity and plugged in to match typical racing shift selection, pull for up shift, push for down shift. The mounted shifter can be seen in the Figure below.

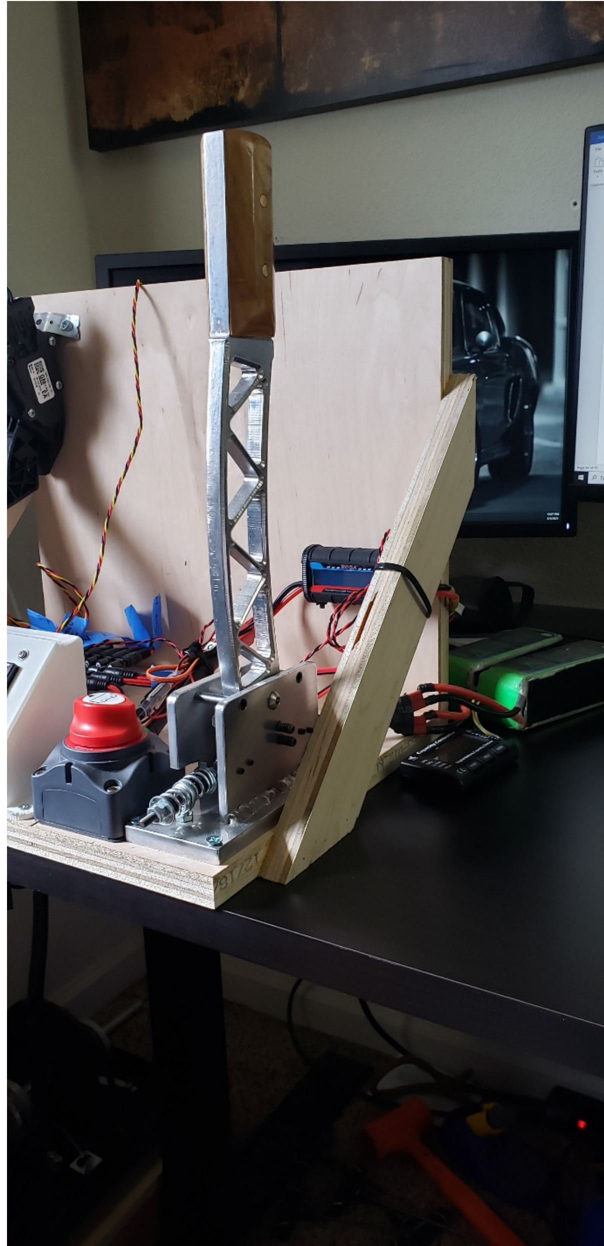


Figure 6.34 Shift selector mounted on the UI

To aid the user while operating the model and contain the major user buttons, a 3D printed UI was used to mount the LCD screen as well as the commonly used user interface buttons. Though the original design implemented 3 buttons (software soft restart, Arm button and Extra button) the extra button was not used as it turned out that the specific pin was required for reference voltage in RPM sensing and would cause even more errors than already experienced. The final

implementation of the LCD and Button UI can be seen in the figure below. The Mode to the upper left of the display is shows arm on start up and after hitting the arm button and finding first, switches into high speed testing mode. The "Speed" is the output rpm, the "Gear" indicates the current gear of the transmission, the "ST" indicates the most recent shift time in seconds. The "MSG:" indicates any messages for the user such as a shift request being too early or max RPM reached. The "RPM:" at the bottom is the input rpm from no pixels to 16 being 0 to 4000 rpm.

Figure 6.35 LCD and Button Housing for User Interface

Chapter 7 : Design Verification

Testing the final design consisted of three main categories, mechanical, electrical and controls system. The mechanical system consisted of components that needed to meet requirements which included the safety polycarbonate window, input and output shear couplers, and overall function of the transmission. The electrical consisted of the wiring, daughter board installation and auxiliary equipment installation. These requirements needed to be tested and completed before the control systems was be tested.

7.a Preliminary Mechanical Testing

The first component that was tested was the polycarbonate viewing window in the top of the housing shell. The purpose of this test was to see if the plastic could withstand an impact of a foreign object and remain structurally stable to retain the safety of the model. In the event of a failure, gear teeth or other material could dislodge at a high rate of speed and hit the viewing window. As discussed previously, the tests were performed before the transmission was finished as the acrylic window needed to be installed during the manufacturing process. The tests were extremely successful and provided a safe viewing point to the transmission. The results discussed in section 6c fully satisfied the design criteria and allowed for moving forward with the testing procedure.

Subsequently, input and output shear couplers were tested to ensure proper torsional breaking strength. The torque requirement for the input coupler was designed to be 2 times the motor torque.

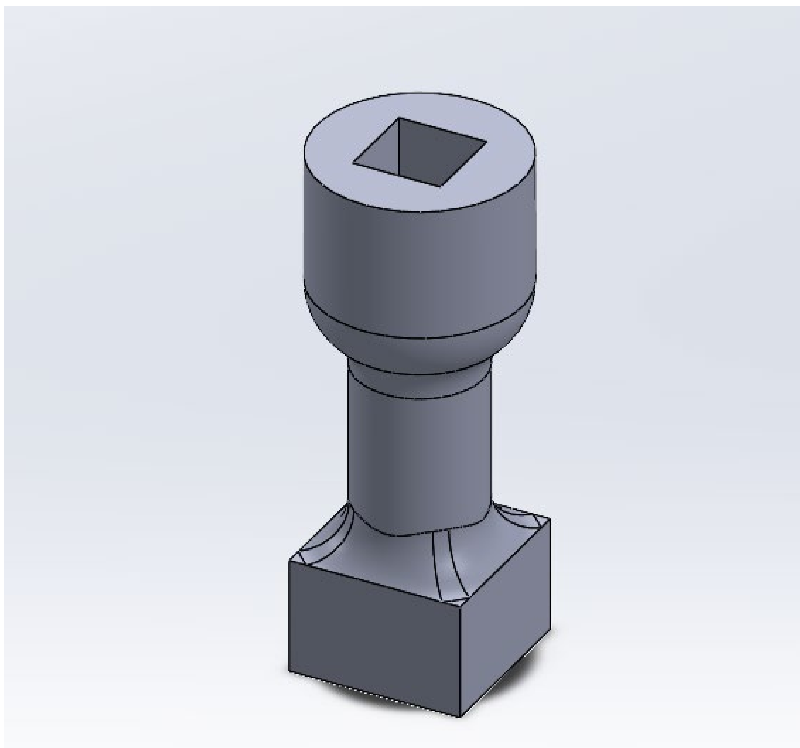


Figure 7.1 Shear Test Coupon

This accounted for variation in motor torque and any minor impulses in the system. This overload factor also applied to the output coupler as the highest torque it will see is a 1-1 ratio with the motor. Initially, testing the couplers to find the proper 3D printed diameter was done by creating a modified shear coupler. This coupler was fixed at one end and attached to a torque wrench on the other. Testing the coupler resulted in stepping up the torque on the coupler in increments until failure. From the results, the plastic shear diameter was modified until the desired 2 times torque input requirement was met. Once the final design was found, three sets on input and output shear couplers were manufactured.

These were then used in the initial test runs of the transmission. As it will be discussed later in the testing process, initial testing resulted in the shearing of all six couplers. This was a great result as most of the failures were due to improper shifts of the transmission. In that aspect, the couplers worked flawlessly. Later in the testing process, when the input power was turned up, two of the couplers sheared for no apparent reason during a completed shift. From the sheared cross section, it was determined that the coupler's mechanical integrity could have been slightly compromised by moisture absorption in the PLA plastic. Additionally, fatigue was also a concern as there may have been slight shaft misalignment between the two drive couplers. The decision was made to increase the shear diameter by .050" to account for any moisture absorption and/or cyclic fatigue in the coupler. Upon testing of the new couplers, the results were proven successful as the transmission functioned reliably with no unintentional coupler failures while maintaining the safety fuse aspect of the coupler.

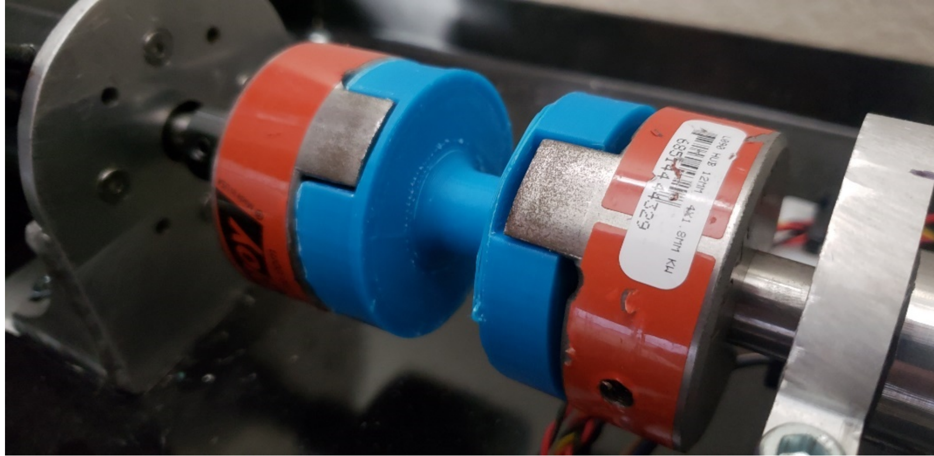


Figure 7.2 Shear Couplers Breaking as Intended and Revised Strengthened Coupler

7.b Preliminary Electrical Testing

Due to initial electrical design and progress, all components had already been tested for their ability to interface with the Nucleo. Though this had been done, several components from early-stage testing had changed to the final versions such as the Pedal going from a simple Arduino controller POT to a Subaru electronic foot pedal.

Figure 7.3 Throttle Pedal Input Setup

Before the transmission could be tested, the input motor was configured to ensure that it would meet the requirements set by the transmission and electrical systems. The first test conducted was to ensure the motor would run using the throttle pedal as an input. The throttle pedal used supplied a variable voltage based on throttle position to the Nucleo. The Nucleo then mapped the pedal position to a pulse width value that was used to drive the motor speed controller. Once the sub-system was functioning, checks were done to ensure that the motor responded accurately and proportionally to the throttle pedal command.

```
pedal_input = analogRead(A1);       // Read the analog signal from the pin the pedal is connected to.
//Filter for PWM Signal
pedal_input = abs(pedal_input - Max_Pedal);
//Serial << "Pedal sig after abs = "<< pedal_input << endl;
pedal_input = pedal_input/Pedal_Range;
//Serial << "Pedal sig percentage = "<< pedal_input << endl;
pedal_input = pedal_input*(PWM_Max - PWM_Min) + PWM_Min;
//Serial << "Filtered Signal =" << pedal_input << endl;
PWM_VAL.put((int)pedal_input);                     // Putting the analog input from the footpedal into the share
```

Figure 7.4 Pedal Signal to PWM Signal Code

The next system that was tested following the motor and shear couplers was the RPM sensing/calculating system. The first test done was to validate the mechanical and electrical systems functionality. This was done by rotating the input and output shafts to allow the RPM wheels to move past the hall effect sensors. The manual data output of the Nucleo was monitored and used to ensure the sensor picked up every metal timing pin at both low and high speeds. With the tests successful, both the mechanical hall effect pickup system in addition to the wiring and circuit board system were confirmed to be working properly. The next step was to confirm the RPM values being read by the Nucleo were accurate to within ±100 RPM. To confirm the speed value was correct, an optical tachometer was used with a reflective piece of tape placed in the flywheel.

Figure 7.5 RPM Sensor Speed Value and Optical Tachometer Verification

The calculated RPM from the Nucleo was confirmed at predetermined motor speeds with the tachometer over the entire RPM range of the transmission. The motor was given a set PWM value so that a theoretically constant speed would be achieved. The software calculated values were compared to the speed read by the tachometer, and subsequently calibrated according to the tachometer readings.

7.c Preliminary Controls Testing

The first stage of testing the control system was to check the shift servos to ensure they moved to the correct position when specific PWM values were sent. For example, to shift to first gear, one servo had to be moved into the engaged position while the other servo had to move into the neutral position. Issues arose at the start of testing because the servos appeared to not move to the correct position despite being sent a PWM value. After reading over the documentation for these servos, it was found that the servos were shipped with preset PWM limits that restricted the allowable PWM to a specific range. Once this was discovered, adjustments were made to the servo tasks in the code in order to ensure that the servos moved to the correct positions and the PWM values were the limits. Figure 7.6 shows the values that were needed for servo 1. Appropriate values were set for servo 2 as well.

```
void task_servo1 (void* p_params)
{
    (void)p_params;                                // Shuts up a compiler warning

    // CHANGE PWM VALUES ACCORDINGLY
    uint8_t  servo1state = 0;                       // State variable for servo 1 task
    uint16_t neutral_pwm = 1000;                    // PWM value for moving servo into neutral position
    uint16_t first_pwm   = 2000;                    // PWM value for moving servo into first gear position
```

Figure 7.6 Servo 1 PWM values for moving to neutral and First Gear positions

Because this system needs to interact with the user in order to run, measures were taken to ensure a properly functioning user interface. The interface mainly consists of several buttons and the shifting stick as seen earlier. One well-known issue that can arise when using physical buttons is a phenomenon called “bouncing”. This phenomenon arises when the user presses the button, but the physical contacts in the mechanism bounce up and down, essentially sending rapid “on-off” signals for the first few milliseconds of the depression. This phenomenon can be filtered out with a “de-bouncing” task, such as the one described in section 5.i. In order to properly parse out the shift button signals, a Mastermind task was introduced. This task (the FSM diagram of which is shown below) checks the button flag that is set every time a specific button is pushed. This task will then interpret the button flag which corresponds to the button pressed and tells the rest of the system how to act accordingly. In this way, the user can request shifts without having to worry about the system not being able to interpret the requests.

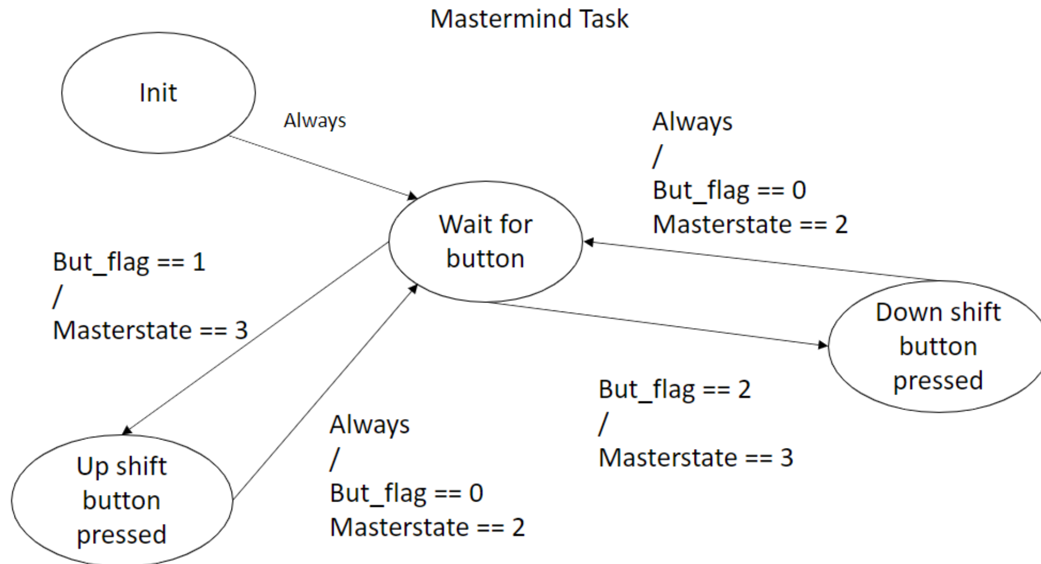


Figure 7.7 Mastermind FSM diagram.

An issue that arose in the testing came up when plotting the RPM values during a motor speed sweep. It can be seen in Figure 7.8 during a motor deceleration test, the RPM values were stepped. This was an issue as the problem resulted in a coarse resolution of the read RPM values and resulted in improper transmission shifts. The solution to this issue was fairly simple, primarily involving decreasing the amount of time the RPM calculation task took before running another calculation. More importantly, the hardware timer prescaler value also had to be decreased. The prescaler value determines two things for the timer: 1. It determines at what value the timer will reach overflow (when it rolls over from its maximum value back to 0) and 2. It determines how fast the timer increases its count. The second point relates more closely to the issue of RPM value resolution, as the timer may run too slow to account for higher speeds. Essentially, although the RPM sensor may be picking up signals at an extremely fast rate, the timer will move so slow that it will appear as though there was no change in time between the previous sensor pulse and the current one. Taking into account the operating speeds of the transmission, the prescaler was decreased from 12000 to 2440, allowing for much higher resolution RPM calculations.

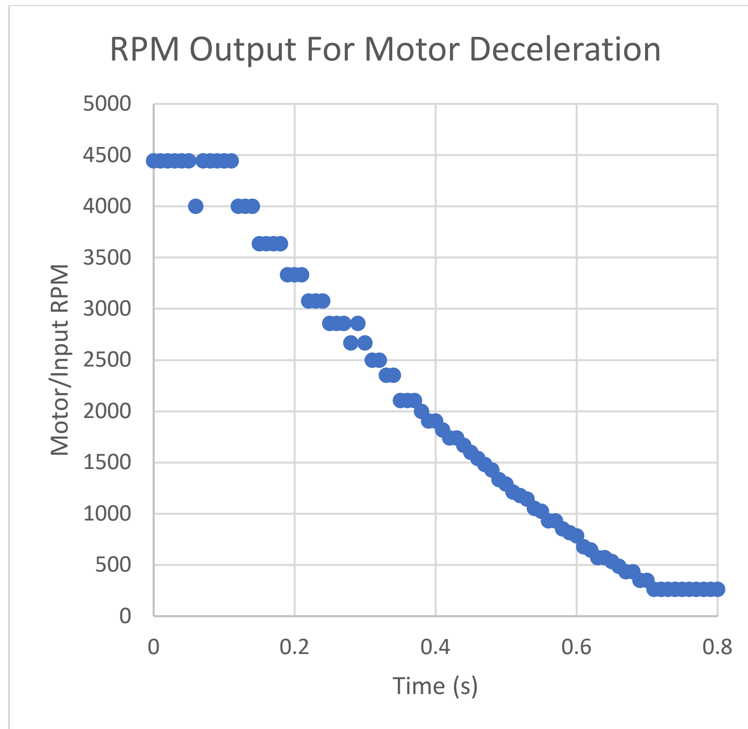


Figure 7.8 Initial Test of Transmission RPM Sensing During Motor Deceleration

With the power system functioning, the attention was moved towards the transmission. The first task that was accomplished was a startup procedure. During the startup of the transmission system, the servos move to a default neutral shift position. To begin driving, the transmission needed to be put into first gear. On first power up, the microcontroller waits for the operator to select the "Arm" button which isn't active until after sufficient time has passed for the ESC to arm. After this is selected, the Nucleo finds first gear and hands the controls to the user. This was accomplished by commanding the motor to rotate at a very slow RPM while the first gear servo is actuated. The motor continued to rotate until the first gear shift collar engaged and the limit switch was actuated, confirming a complete shift. Just as there can be full speed mis-shifts, the low-speed equivalent can occur as well. To reduce this issue, a pulsing of input rpm and servo position was implemented to ensure the first gear did in fact contact the appropriate gear. This method of initializing was highly effective and reliable.

With all the preliminary sub-systems functioning, attention was turned to upshifting and downshifting the transmission. Initial rpm matching for the shifting function was to be performed based on the transmission inertial values and a controls modeling. However, this ended up not being viable as a result of the using an ESC instead of a motor driver. The ESC is effectively a black box with it's own control algorithms for which the microcontroller is not privy to. Thus, after conferring with the 507 controls professor Ridgley, it was determined that the best method for RPM matching would actually be a combined PWM to RPM map and a K_p value determined through testing.

This PWM to RPM function was found by mapping the output RPM value for a given input PWM signal to the motor speed controller (see Figure 7.4). This relationship allowed for a quick transition to a “ball-park” RPM value where the motor needs to be based on the output rpm and desired new gear ratio. From there, a simple proportional-only controller was used in conjunction with RPM sensors to properly match the input motor speed to the output motor, and thus allow the dogs to be moved into position and into gear. The controller would calculate the error between the current input motor speed and the desired speed, and then augment the estimated PWM value calculated from the PWM map.

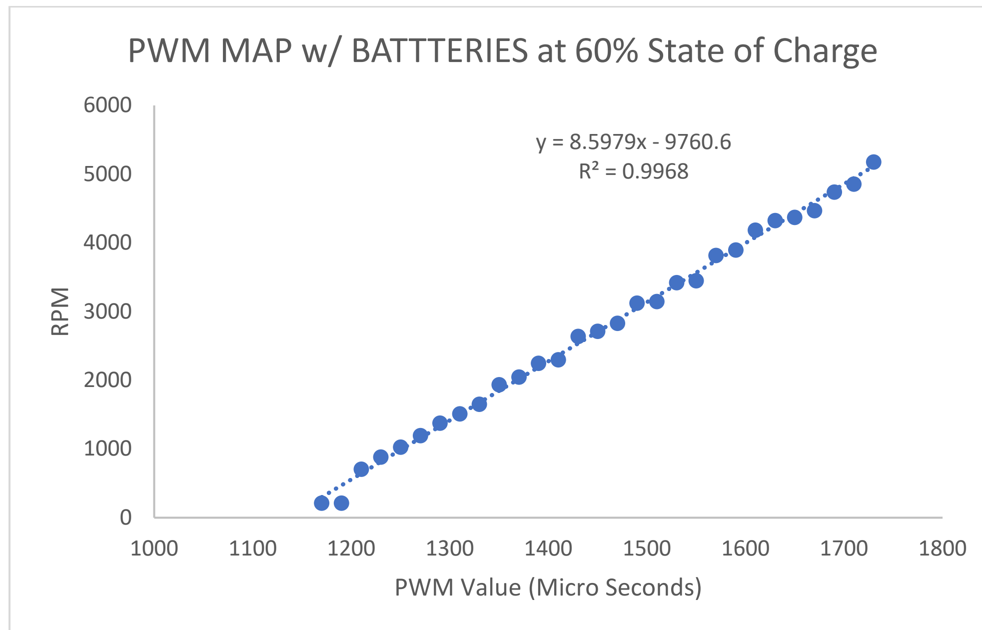


Figure 7.9 Plot and Function of Motor RPM Vs. Input PWM

7.d Preliminary Full Shift Testing

Shift testing began with low-speed motor runs with a maximum input speed of around 1500 – 2000 RPM. This was done in an attempt to minimize the risk of breaking internal transmission components from a mis-shift at higher speeds. Initial upshift tests were promising as the second gear shift was successful about half of the time. Adjusting and fine-tuning parameters such as attempted servo shift time and the delta RPM rev match values resulted in a more reliable and faster upshift. One issue with shift testing which caused inconsistent shifts was the reading of the RPM sensors.

When testing the upshift, the transmission would unpredictably miss the shift. Upon further investigation, the RPM sensor would intermittently output a rogue. After extensive testing and troubleshooting, the problem was found to be with a faulty resistor in the daughter board. It was suggested by the Mechatronics faculty advisor, John Ridgley, that the pin settings for the sensor be changed from INPUT to INPUT_PULLUP. By changing this setting, the Nucleo will actually use an internal resistor native to the microcontroller as a pullup resistor for that pin. A pullup resistor essentially increases the voltage drop from the signal side of the pin to the controller side

("pulling up" the voltage). This helps to filter out noise and dirty signals from the pin, as the large amount of resistance prevents sensitivity to fluctuations. As soon as this change was made in the software, the issues regarding bad RPM signals were gone, and testing could continue.

An additional challenge that was brough up during the preliminary shift testing was the unreliable nature of the down shift. As described before, the transmission functionality was dependent on the matching of the gears. A result of purchasing the "same" gear sets from different vendors and/or differences in manufacturing processes resulted in slight variations in gear geometry. This was very apparent in the first gear assembly due to the double shift collars present. The shift collar dogs and mating slots on the gear were manufactured such that the clocking of each between the right and left sides was slightly off. As a result, when both gear sets were installed and mechanically coupled together, half of the dog engagement positions on either countershaft engaged smoothly. The other half did not engage smoothly and/or engage at all. This significantly affected the downshifting of the transmission as it resulted in some inconsistent and failed shifts. This problem was unfortunately unresolvable as it would require reworking or replacement of the entire gear train to obtain matching gears. The decision moving forward was to still program the transmission downshift as normal but not hold the testing parameter time for the down shift to the 1 s required for the upshift.

The FSM diagram below shows the underlying logic of the shift task software. Once the user requests a shift, the microcontroller runs through the loop checking that the appropriate conditions are met at each state. Two counters are implemented in the software: one in the "Move into neutral position" state and one in the "Move into engaged position" state. Simply put, if after a certain amount of time the system has not been able to perform the desired task, the microcontroller "times out" and tells the system to go back to a previous state in order to attempt the task again. This does not interfere with the performance of other tasks in the whole code, given the nature of Free RTOS.

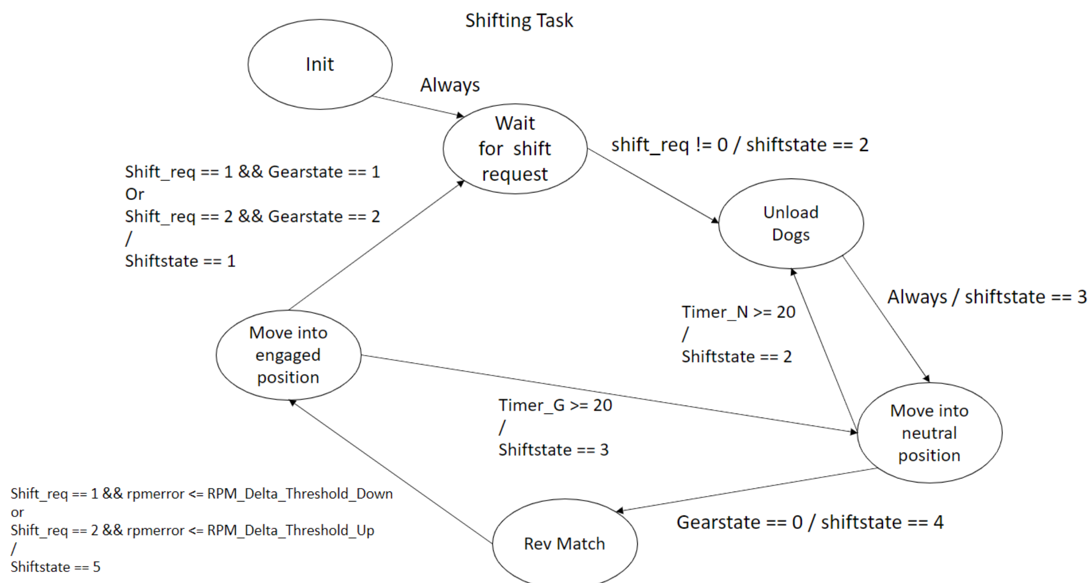


Figure 7.10 Finite State Machine diagram for the shift task

The shift verification process required a large amount of testing to achieve the set requirements. Small errors in the shift code that intermittently output a false RPM values plagued the shift sequence and resulted in several sheared output couplers. The logical error was in looking for an rpm error less than the required value, negatives are always less than the set value. Thus, a few shifts with a rev-match rpm error of ~ -1500 rpm were attempted and the resulting shifts broke the shear couplers. The fix was simply altering the code to look for an absolute value of rpm error to be less than the desired set point.

Additionally, the shift loops had a bug that resulted in incorrect timer values, which caused inaccurate servo firing and shift times. The solution to these problems was found by making adjustments to both the software calculation algorithm and the timer prescaler values. Professor Ridgley of Mechatronics noted that there was a conflict between the variable types ("types" meaning either 16-bit integers, floating decimal values, etc) and the hardware timer selected for the calculations. By adjusting the variables to be 16-bit signed integers, overflow problems were avoided, and the inaccurate timer values remedied. To verify the timer consistency, a test function was created which output timer values at a regular interval accurate to within .1 ms which used a vTaskDelayUntil(). The timer then prints to the screen the calculated timer delta and this was used to verify not only the calibration but also that the overflow issue was resolved. With this test performed, the timers were now accurately recording the time between hall effect triggers which resulted in an accurate rpm sensing.

After much troubleshooting, the problems were resolved and attention was turned to optimizing the shift times. The three main variables that were changed in the shift were the motor feedback gain, the servo shift attempt time, and the rev match delta RPM value. The motor feedback gain was implemented during the rev matching process after the initial "ball-park" motor RPM was achieved as described earlier. The gain was changed based in the response of the motor during the shift as a result of watching the printed RPM parameters on the display. The final motor gain value used was $K_p = .005$.

```
revpmin_in.get(rpm_in);                 //Current output rpm into local variable
revpmin_out.get(rpm_out);               //Current input rpm into local variable
rpm_target = rpm_out/First_Gear_Ratio;  //Calculate rpm target based on gear ratio
//Serial << "Target = " << rpm_target << endl;
rpm_error = rpm_target - rpm_in;        //Calculate error in RPM based on target and current rpm in, - is below, + is above
//Serial << "RPMIN = " << rpm_in << " RPM ERR = " << rpm_error << endl;
//Serial << "RPM ERROR = " << rpm_target << endl;
if(abs(rpm_error) <= RPM_Delta_Threshold_Down)      //Check to see if error is small enough, if so set state to complete shift
{
    Serial << "Reached RPM Threshold" << endl;
    Serial << "RPMin = " << rpm_in << " RPMout = " << rpm_out << endl;
    shift_state = 5;                    //Set the next state to shift state = 5, complete the shift
}
else
{
    PWM_total = 1600;                 //Testing value & shouldn't be here in the end
    PWM_map = .1159*rpm_target + 1136.3;    //Calculate map pwm based on target rpm value, see excel chart for graph
    PWM_kp = rpm_error*kp;              //Calcuate proprotional PWM based on error and kp value
    PWM_total = PWM_map + PWM_kp;       //Sum error into output PWM
    //Saturation block for max and min PWM values
    if(PWM_total > PWM_max)             //Set saturation to ensure output PWM is within limits
    {
        PWM_total = PWM_max;            //Setting the motors PWM value to the maximum allowable
    }
    else if(PWM_total < PWM_min)
    {
        PWM_total = PWM_min;            //Setting the motor's PWM value to the minimum allowable
    }
    PWM_VAL.put(PWM_total);             //Setting the share variable for the PWM
    Serial << "PWM_Total = " << PWM_total << endl;
}
```

Figure 7.11 Rev Match RPM Algorithm Code

To allow for a mis-shift algorithm, the servo shift duration specified the time the servo would try to complete the shift while motor was at the rev matched state. Past this time, the servo would return to neutral and the system would retry the rev matching process. The shift duration needed to be long enough for the dogs to mate and engage while being short enough to allow for several more attempts to retry the shift if failed. The servo shift time chosen was .5 seconds. This time proved to be sufficient for most well rev-matched shifts on an upshift.

Lastly, the rev-match delta RPM was set. With a low input to output speed differential, the dogs would have more time to engage in the mating gear, however the number of possible engagement points per given time is decreased. Subsequently, if the value is say 0 RPM, there is the strong possibility that the dogs will hit the mating dogs and never engage during the shift. On the other hand, a large delta RPM value would increase the number of dog engagement opportunities per given time but the time the dog has to engage is reduced. A happy medium was achieved through many trial and error test runs of the transmission. The resulting delta RPM used was 100 RPM.

7.e Final Full Speed Shift Testing

For the full speed testing, the shift testing and time to speed were measured at the same time for each run. The time to speed was measured via a stopwatch, using the RPM output to the LCD and serial port. The up-shift shift time was recorded via the microcontroller with an accuracy of .0001 s. For reasons previously stated, the down shift times were not recorded. For each run, the battery health was observed to ensure the pack did not drop below 40%. Allowing several seconds between runs to ensure battery temperatures remained manageable. The testing was done over two sessions as the packs only had enough energy for ~15 full speed runs. The 25 tests performed are outlined

in Appendix G and show the times for each shift and corresponding time to speed. The basic statistics of each quantity were computed into Table 7-1 below.

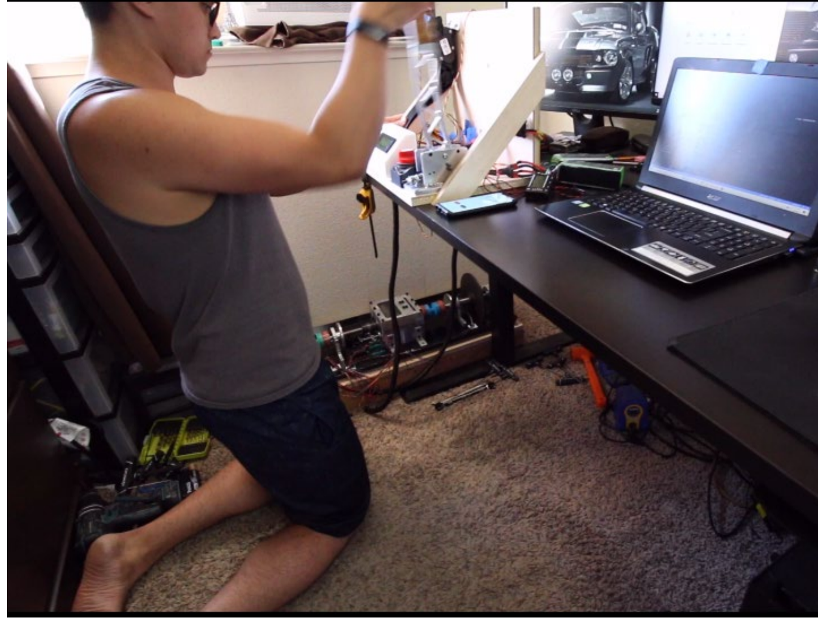


Figure 7.12 Final Full Speed Testing

Shift Time Statistics		Time to Speed Statistics	
Mean (s)	0.937	Mean (s)	5.226
SD	0.070	SD	0.561
n	25	n	25

Table 7-1 Shift time/Time to speed testing results.

7.f Final Testing Results

To determine the true shift time and time to speed for the transmission, a statistical hypothesis test was performed for each against the engineering requirement. With these values used as μ and a level of confidence, α = .0005, the following was performed:

The null hypothesis is given by:

$$H_0: \mu = 1$$

The alternative hypothesis is given by:

$$H_a: \mu < 1$$

The level of confidence is given by:

$$\alpha = .0005$$

To compute the corresponding z value, the following equation is used:

$$Z = \frac{\bar{X} - \mu}{\frac{SD}{\sqrt{n}}}$$

With the Z value, a corresponding P-value can be obtained from a normal distribution table. Given a Z value below -4, there is little area under the curve so to three decimal place accuracy, the resultant Z value is 0.000, which is below 0.0005 by way of rounding.

Shift Time Statistics		Time to Speed Statistics	
Mean	0.937	Mean	5.226
SD	0.070	SD	0.561
n	25	n	25
Z-value	-4.527	Z-value	-6.896
P-value	< 0.0005	P-value	< 0.0005

Table 7-2 Shift time/Time to speed test results. Includes Z and P values.

Based on the P-value obtained from the hypothesis test, there is sufficient evidence to reject the Null hypothesis and confirm that the transmission does in fact have a shift time below 1s with a time to speed of less than 6 seconds.

System Parameter:	Required		Actual	Requirement Met
	Value	Tolerance	Value	
Time to Speed	6s	+ 1s	5.23s	Yes
Motor rpm	MIN 4,000	MIN	5,000	Yes
Input Torque Capacity Increase	80%	- 5%	80% Theoretical	Yes
Shifting Speed	1.0 s	+ 0.1 s	.94 s	Yes
Meas. Input and Output RPM	MIN 8000	MIN	10,500	Yes
Shifter Dog Position	12.7 mm	±2.0 mm	12.7mm	Yes
Driveline Inertia (Flywheel)	0.01 kg-m^2	±5%	0.1 kg-m^2	No
Number of speeds	2	MIN	2	Yes

Table 7-3 Summary Engineering Requirements Table

In summary, the transmission does in fact meet all engineering requirements as specified in Table 7-3 above. The only requirement not met was not recognized until final completion of the report. It turned out, that one of the primary causes for the mis-sized original motor was not just the neglect of friction, but in fact a missing 0 in the driveline inertia resulted in a flywheel 10 times larger than expected. Thus, the motor originally sized had no real chance of moving the system. The oversized flywheel also explains why the final system worked well with the new motor, which was roughly 10 times more powerful than the originally sized one.

Chapter 8 : Project Management

Project GHOST preliminarily began one quarter prior to Spring of 2020 with Matt and Austin putting together a private project and group. The intention was to get a team of more experienced students working on a project of interest and significance, independent of the academic governing bodies of Cal Poly. The project was originally planning to utilize more of the manufacturing resources of the advanced machine shops in the IME lab as Matt is a shop tech and both he and Austin have CNC certification there. However, by the start of spring quarter, the COVID-19 pandemic had begun and pushed the project entirely remote. The solution was to move all design through what manufacturing resources remained.

Spring quarter of 2020 was a quarter of moving around timelines, ordering parts to fit the project needs based upon the circumstances, and designing the transmission according to the manufacturing methods still available to the team. The overall timeline of the project remained relatively unchanged. In the Fall quarter of the same year Zack and Austin took ME 507, Control System Design, using the senior project as the project for the quarter. This allowed the team to leverage the resources available to the classroom for the project such as the Software and Board design which dramatically expedited the process. As the Fall quarter continued, Matt was able to utilize the minimal access to the IME lab to produce some of the preliminary parts required for the project before these resources became unavailable. Moving into the final quarter of the project, Spring of 2021, the final details of final manufacturing and coding were all that was left. Using MIM services the entirety of the prototype construction was completed on time, the end of the fourth week of the quarter. This provided the remaining 6 weeks of the quarter for Austin to complete the wiring of the project as well as finalize all programing and complete testing.

The final testing was plagued with consistent output RPM sensor errors. These resulted in a difficult to locate faulty pullup resistor on the RPM sensor wire which continued to plague the project for weeks of various testing. Once this issue was resolved and final UI testing had been completed, the final testing was relatively quick as the major points of concern included the shift times and time to speed which were both conducted simultaneously on each testing run.

The first of the major lessons learned for this project is “you get what you pay for”. To ensure completion of the project once the COVID pandemic had begun, the design shifted to nearly 100% store bought gear train components for the purpose of time and resources. However, being that typical gear manufacturing is quite expensive if done properly, the only gear sets in the size and price range the team could accommodate were of cheap overseas import quality. The result was slight discrepancies between the sets which resulted in clocking issues that added to the manufacturing difficulties.

The next lesson learned for the project was “plan to be done early, so when you are late, you are still early”. The planning for the project was continuously updated and modified according to the changing circumstances, but to ensure project completion, aggressive deadlines were set for MIM services which upon minimal delays, produces a finished mechanical prototype by the end of week

4. Thus when the RPM errors arose in the code delaying progress by nearly two weeks, the final project was still able to cross the finish line on time.

The last major lesson of the project was a reinforcement of an old habit of experience in timed projects, “when in doubt, buy two”. Having a team of experienced builders and engineers meant and understanding that things frequently go wrong. To mitigate timeline alterations, which are ultimately not extendable, duplicate items were machined, purchased and designed as fail safes to the inevitable murphy’s law. These proved invaluable in multiple occasions from the duplicate microcontroller boards used to fix problems when one fried, to the duplicate daughter boards at two separate locations and team members to help troubleshoot code vs hardware issues. These numerous over expenditures are vital to completion of aggressive projects with aggressive timelines and fortunately, it worked out.

For any future team looking to tackle a similar project, the largest thing to take away from this is a recognition of the team which completed this project. All four team members are JC transfers with over 4 years of education prior to time at Cal Poly. All four students have a strong desire to learn not for the desire of a piece of paper that says engineer, but for a desire to develop their skills at solving problems and using the fundamentals of engineering and experience of those who came before to create new solutions to the world’s problems. This project was shot down by several advisors and faculty for the perceived feasibility of typical students. If it were not for the outstanding performance of the members of this team, there would have been no chance for completion of this project.

Chapter 9 : Conclusion

This project aimed to design and verify a dual countershaft model transmission design by using a custom-made control system. The project was initiated by conducting background research into existing dual countershaft designs, as well as the reason for them. The design that was used in the project was chosen for its advantage of the high torque capacity and ability to use the offered motor torque without concern for damaging internal components in the transmission. The mechanical design was split into three subsystems, each with their own relationship to the control system. These systems were: 1. Input motor assembly, 2. Transmission gearbox, and 3. Output flywheel assembly. The input motor and output flywheel assemblies were mainly comprised of common-off-the-shelf (COTS) components with some mounting hardware and fixtures being custom designed. The transmission gearbox was almost completely custom designed and made, with the only COTS parts being hardware like bearings and bolts. The gears themselves were not COTS, but rather salvaged gears from a 125-cc pit bike. The control system utilized the STM Nucleo microcontroller due to its I/O capabilities as well as compatibility with Arduino hardware and C++ programming language. With C++, the control algorithms were able to run in the order of microseconds, which was necessary given the high rotational speeds of the system. The custom PCB used in the system interfaced with the Nucleo and allowed for the connection between inputs and control outputs. This integrated the input motor, RPM sensors, shifter, and user interface together into one. Extensive code was written and altered to take the inputs to the transmission and output the correct signals to control the transmission and optimize its shift patterns. With the integration of both the mechanical and electrical systems, the transmission was functional and operated as intended. The system successfully upshifted and downshifted at all desired ranges of speed and input torque with high reliability. The critical parameters that were met included the increase in torque carrying capacity of the transmission, a shift time of less than one second, and a time to speed of less than six seconds.

The resources used to design and build the code for the control system using the resources were offered in ME 507 Controls System Design, namely the experience of the professor John Ridgely who provided invaluable input on the direction the control system took as well as the proper methods and techniques for complete integration and control of the system. Additionally, Matt was able to utilize the many tools at his disposal to manufacture the transmission assembly. A large portion of the manufactured parts were constructed on the CNC mill, which was first built before manufacturing began. Lastly, Conan provided the many 3D printed components used for prototyping and final production models.

There are several aspects that would be improved upon in future iterations of this system. First, the transmission would utilize a matched set of double helical cut gears. Double helical gears would be used to decrease transmission gear noise while increasing the load carrying capacity. The gears and matching dogs would also be machined to eliminate the gear clocking that plagued the current design. Additionally, without the constrain of using pre-manufactured gear sets, the double shift collar present on the first gear assembly would be substituted for a single shift collar. This collar would be present on the input shaft in the same manner as the second gear setup is in the

current transmission. This would eliminate the dog clocking issues previously described while reducing the complexity and size of the shift forks. Subsequently, actuating the transmission shift would be accomplished in a different manner. A shift barrel, similar to those used in motorcycle transmissions, would be used in place of the two shift servos. This barrel with precisely located slots would rotate and consequently move both the shift forks the appropriate amount at the appropriate time. This would also eliminate the possibility of a mis-shift as the two shift forks are mechanically bound together. Using a shift barrel would also reduce the input actuation needed to one component rather than the two shift servos needed currently. On the controls subsystem, a feature that was desired to be implemented was an auto tune function for the transmission. This programmed routine would run as a "calibration" for the transmission to account for different vehicles the system was used in. The routine would attempt to shift the transmission several hundred times with different shift parameters and monitor the resulting actions. Based in whether the shift was completed and how fast it was accomplished, the values would be stored based on the best shift. After the auto tune was complete, the transmission would utilize the optimized parameters for the system it was installed in. Considering this project sponsor is also the project lead, there is strong hope that all of the future plans will be implemented in a final iteration in the future.

Chapter 10 Works Cited

Chandan, et al. “Different Microcontrollers Used in Automobiles.” *ElProCus*, 20 Apr. 2020,

“Clutchless Transmission.” *Liberty's Gears*, libertysgears.com/our-products/clutchless-transmission/.

“Hamstra, George. “HyPer 9 IS ™ - SRIPM Integrated System.” *Motor Information*, https://www.thunderstruck-ev.com/hyper-9-is-integrated-system.html.

“MPC5566: 32-Bit MCU for Automotive Powertrain Applications.” *NXP*, https://www.nxp.com/products/processors-and-microcontrollers/legacy-mpu-mcus/mpc55xx-mcus/32-bit-mcu-for-automotive-powertrain-applications:MPC5566.

“MPC564xA: Ultra-Reliable MPC564xA MCU for Automotive & Industrial Engine Management.” *NXP*, https://www.nxp.com/products/processors-and-microcontrollers/power-architecture/mpc5xxx-microcontrollers/ultra-reliable-mpc56xx-mcus/ultra-reliable-mpc564xa-mcu-for-automotive-industrial-engine-management:MPC564xA.

“NHRA 101: Inside a Pro Stock Car Transmission.” *NHRA*, https://www.nhra.com/videos/2017/nhra-101-inside-pro-stock-car-transmission.

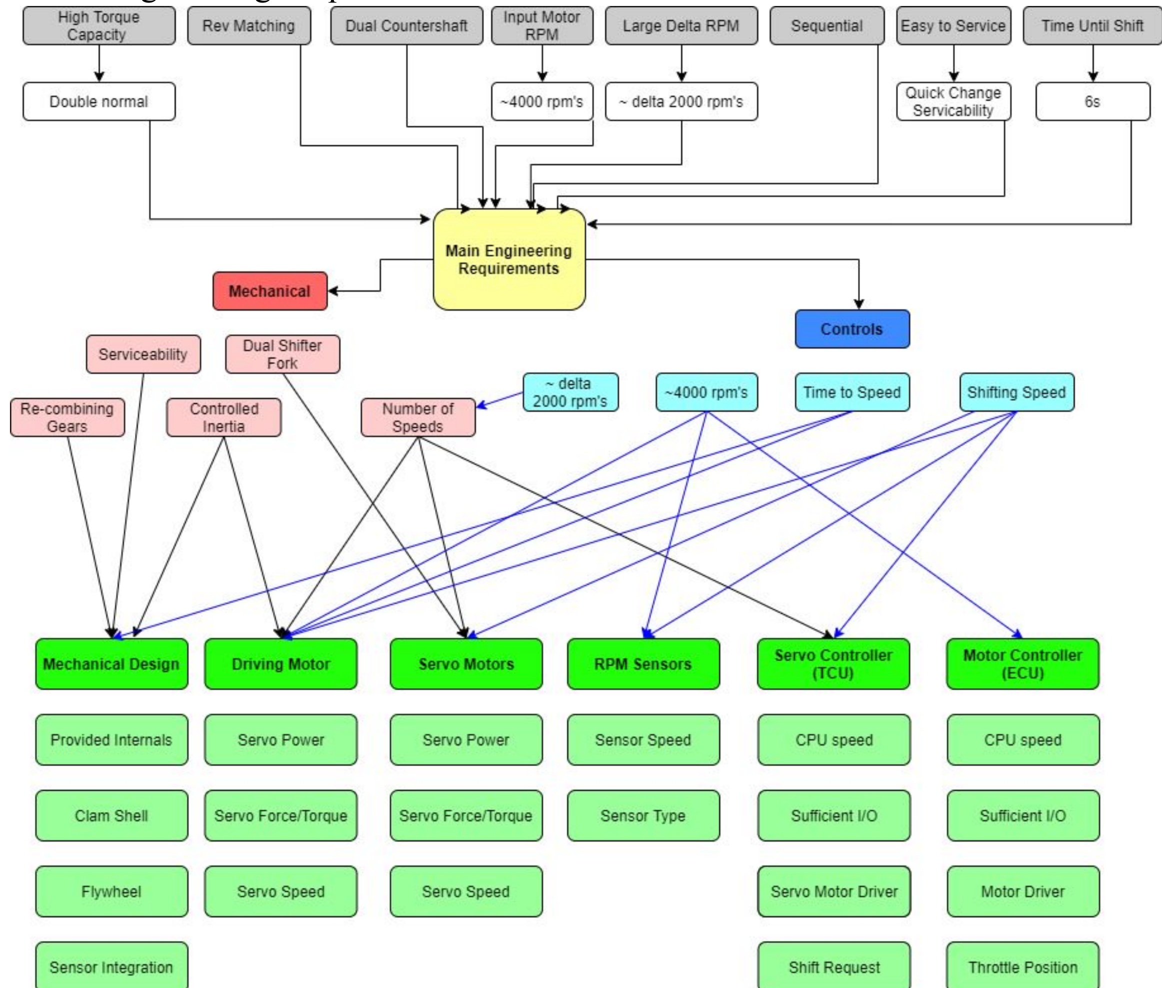
“SPC5 32-Bit Automotive MCUs.” *STMicroelectronics*, www.st.com/en/automotive-microcontrollers/spc5-32-bit-automotive-mcus.html#overview.

BUDYNAS, RICHARD G. *SHIGLEY'S MECHANICAL ENGINEERING DESIGN*. MCGRAW-HILL EDUCATION, 2019.

Devore, Jay L., et al. *Applied Statistics for Engineers and Scientists*. Duxbury, 2014.

Chapter 11 Appendices

Appendix A: Engineering Requirements Flow Down Chart



This chart was used as the initial generator of engineering requirements for the project. The top level shows the requirements desired by the sponsor. These are then interpreted and distilled by the team into the mechanical requirements and the controls requirements. From these, a third level of requirements is formed (shown in pink for the mechanical side and light blue for the controls side). Both mechanical requirements and controls requirements are then used to determine the final level of design requirements such as servo power, CPU speed, internal construction, etc.

Appendix B: Preliminary Design Report Material (Initial Concept Design)

Fork Design Decision Matrix			
Criteria	Options		
	Triple Fork	Stacked (Single & Double)	Staggerd (Single & Double)
Size	Compact Depth	Largest Depth	Medium Depth
Machining Complexity	Single Program (3x), 1 op, Housing Machining High	2 Programs (1x), 1op each, Housing Machining Med	2 Programs (1x), 1op each, Housing Machining Med
Cost	$$$, Extra Actuation, Bushing, Bearing and Seal	$$, Larger Dual Billet Size	$, Medium Billet, No Extra Hardware
Controls Complexity	Triple Actuation	Dual Actuation	Dual Actuation

Table B1: Decision Matrix for the Fork Design Concepts

Recombining Methods Decision Matrix			
Criteria	Options		
	Gears	Belt Driven	Chain & Sprockets
Manufacturing	High Cost/Time if self-manufactured, Low if purchased	Custom pulleys, tensioner required	Custom sprocket, tensioner required
Maintenance/ Servicebility	Low	High	High
Load Carrying Capacity	High	Low	High
Durability/Resistance to Wear	High	Low	Medium

Table B2: Decision Matrix for the Recombining Methods

Recombining Gears Decision Matrix				
Criteria	Options			
	Machine Custom Set	3D Print Custom Set	Purchase OEM Set	Purchase Aftermarket Set
Manufacturing Time/Purchasing Time	2 op 4-axis, custom broaching, CAD/CAM	CAD/Slicer, print time, post-machining	Shipping time? Availability. Little post-machining (direct fit)	No guaranteed fit. Lead time for manufacture and shipping. Little post-machining.
Cost	$$$	$	$$	$$$
Durability	High	FAIL	High	High
Replace-ability	Tied to cost/time	Easiest	Easy	Tied to cost/time

Table B3: Decision Matrix for the Recombining Gears

Appendix C: PDR Design Housing

While the overall concept of the dual-carrying transmission is based on the Liberty's Gears Equalizer 5-speed, there are several sub-systems that had to be designed specifically for this project. First, the housing for the transmission had to be designed. While the housing is not necessarily a critical sub-system, it is an obvious necessity.

The transmission housing (Figure 1) is a clam shell design which provides easy of manufacturing as well as serviceability. The housing dimensions were determined based on the gearset chosen for this model, as well as the shifter fork and shifting mechanism design. The housing was optimized for manufacturability based on CNC tooling availability and 3D printing platform dimensions. The gears were taken from a Honda 125cc Pit Bike. These gears were bought because of their small size and low cost, which fit well with the given scope for the physical model.

In the final design, the upper housing will have a viewing port for testing and shift verification. The final design will also have sealing and mounting to ensure proper transmission function throughout the controls testing.

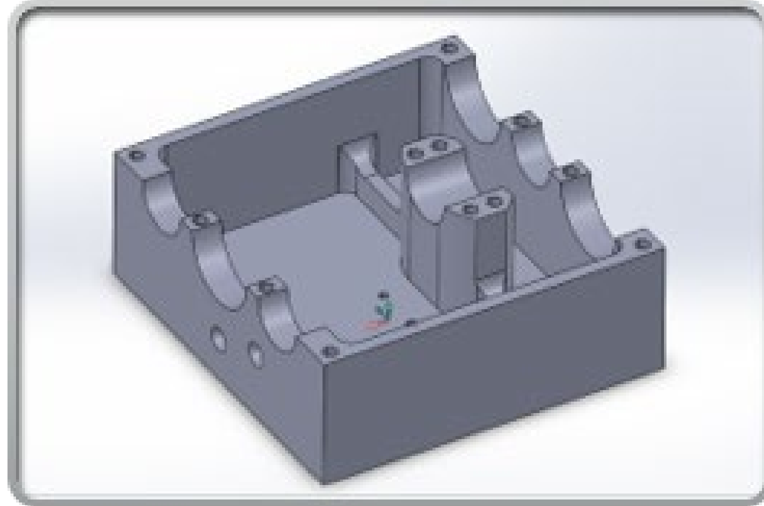
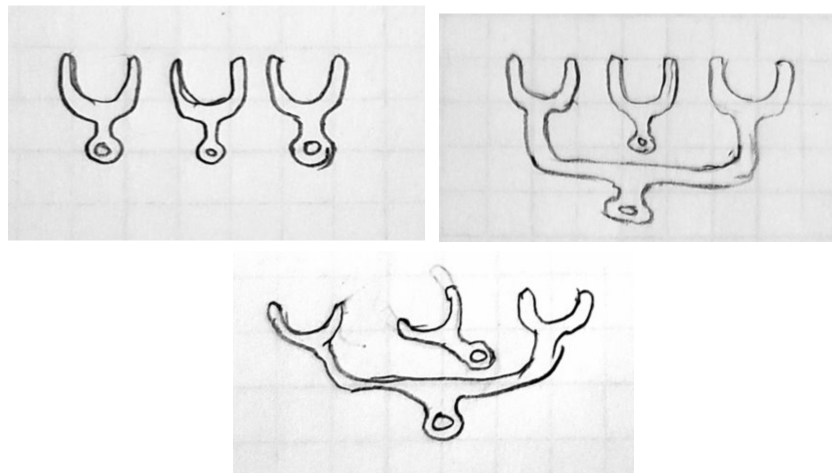


Figure C1: The conceptual case is horizontally parted, creating a clam shell design. This allows for easy access to the transmission when servicing is needed.

Shifting System

For the shifter fork design, three designs were considered (Figure 2) to actuate the shift collars. The first design used three individual forks, one for each slide collar as shown in the left picture. The second design used a single central fork and double outer fork stacked on top of each other as shown in the middle picture. The double fork could be used as the two outer countershafts are identical and the slide collars needed to be actuated at the same instant. Lastly, the third design used a single central fork and double outer fork but staggered to save space in the transmission case as shown on the right.



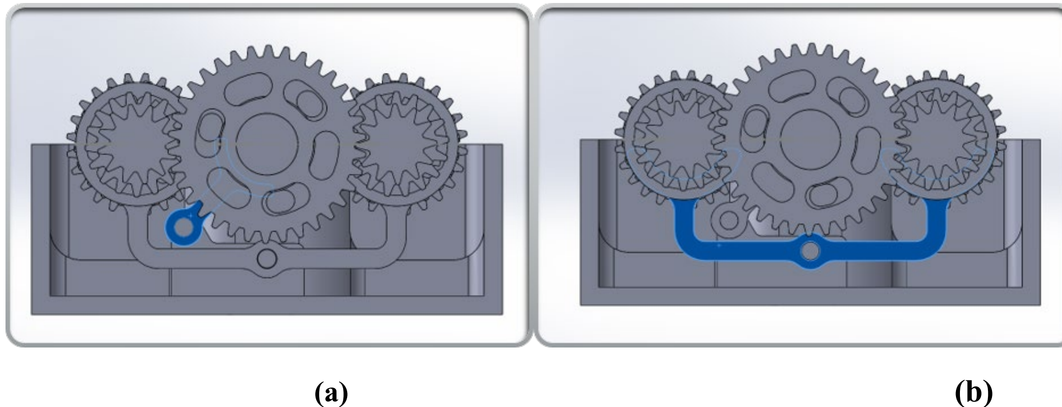
(a) **(b)** **(c)**

Figure C2: The three concepts for shifter fork design and layout.

(a) Triple Fork design. (b) Stacked Fork design. (c) Staggered Fork Design

Using a decision matrix (Table 1) the best design was found to be the staggered fork. The criteria considered were size, machining complexity, cost, and controls complexity. One criterion was the cost to produce the parts. The stacked and staggered designs had the same manufacturing cost, while the three single fork design had a higher cost due to the increased number of shafts, seals, and actuation devices.

In Figure 3, the winning dual fork can be seen moving the two countershaft collars in the left picture while the single fork can be seen actuating the single input shaft collar on the right. This design retained a low case profile while keeping manufacturing costs down.



(a) **(b)**

Figure C3: Concept CAD for the shifting forks.
(a) Dual shift fork for dual countershafts. (b) Shift fork for main shaft

Once the fork design was finalized, methods needed to actuate the shift forks were considered. A similar matrix was constructed for the shifting methods of the transmission. After considering many options, the viable shift options were found to be a barrel and servo design, a pushrod and servo design, a pushrod and solenoid design, and a linear actuator and pushrod design. The servo, solenoid, and linear actuator method types all relied on linear motion to move the shift rods and thus actuate the shift forks.

The shifter barrel design is the same used in a motorcycle transmission, moving channels cut in a rotating barrel convert rotary motion to linear motion, which can be used to shift multiple forks at once. The barrel setup would require a lot of machining and mounting to the case and was a considerable drawback. The solenoid appears to meet all the requirements but fails in the fact that it cannot shift into neutral, without having a three-position solenoid. From the matrix, the winning design was the servo and pushrod design for its simplicity and tunability.

Recombining Section

An additional problem needed to be solved was the combination of the two countershafts into one output shaft. Methods of recombining that were considered included gears, belts, and chains (see Table 3). Based on the criteria of manufacturing, maintenance, load carrying capacity, and durability, it was chosen to use a gear system. The gear system was easily serviced, had a high load carrying capacity, and had a high wear resistance. It also does not require tensioners like belts and chains would, reducing the complexity of the system.

Because gears were selected as the best option for the recombining section, another matrix was constructed to find the best option for acquiring them. The suggested options are shown in Table 4. Self-machining the gears would allow for a highly specific and custom gearset, but it would be time-consuming and costly. 3D printing, while cheap and relatively fast, also creates the weakest gearset as they are not wear or temperature resistant. The remaining options became purchasing aftermarket gears or buying the original engineering manufacturer's set (OEM). These gears (whether OEM or aftermarket) would have to match with the same Honda 125cc Pit Bike set used for the rest of the transmission. The aftermarket sets would have no guaranteed fit and would also be more expensive. Because of this, buying the OEM sets was the obvious best choice. The extra manufacturing needed with this option is almost non-existent, and the gears can be easily swapped if needed to produce different final drive ratios.

Appendix D: Engineering Design Exploded Views and BOM

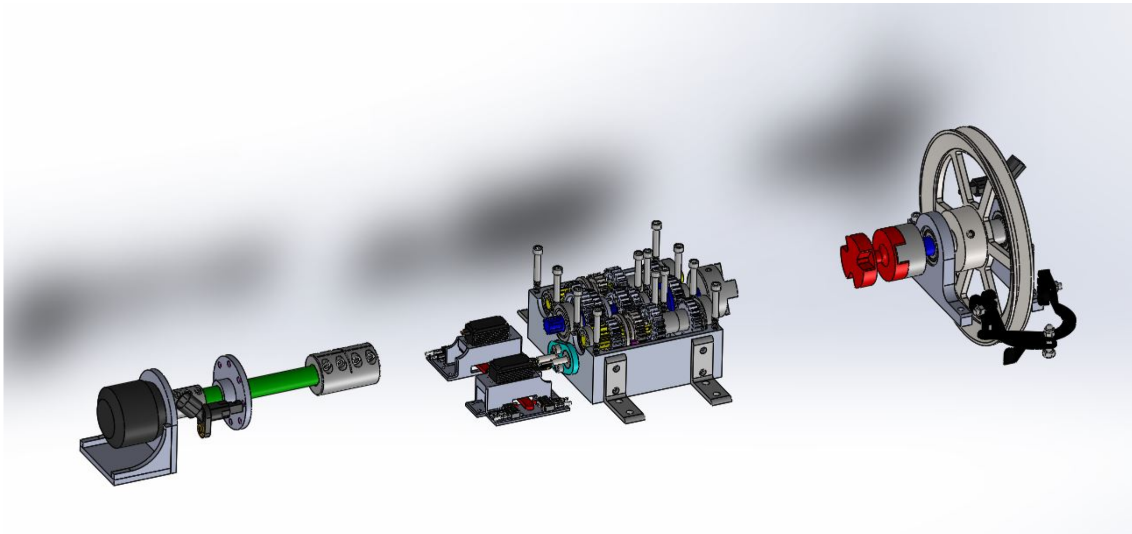


Figure D1: Three Major Sub-Assemblies of the System

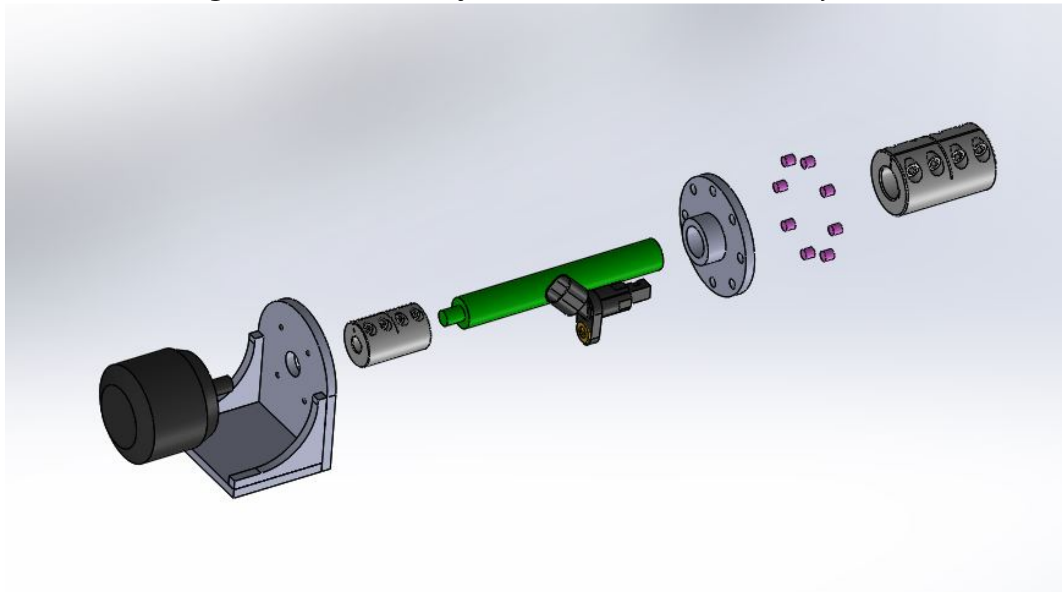


Figure D2: Drive Motor Sub-Assembly Exploded View

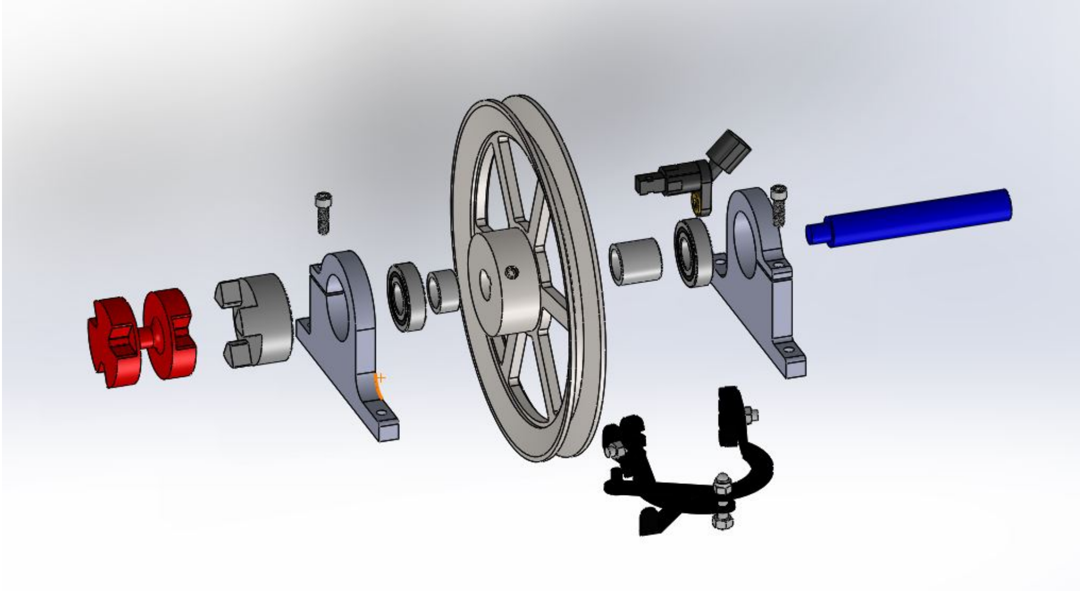


Figure D3: Flywheel Sub-Assembly Exploded View

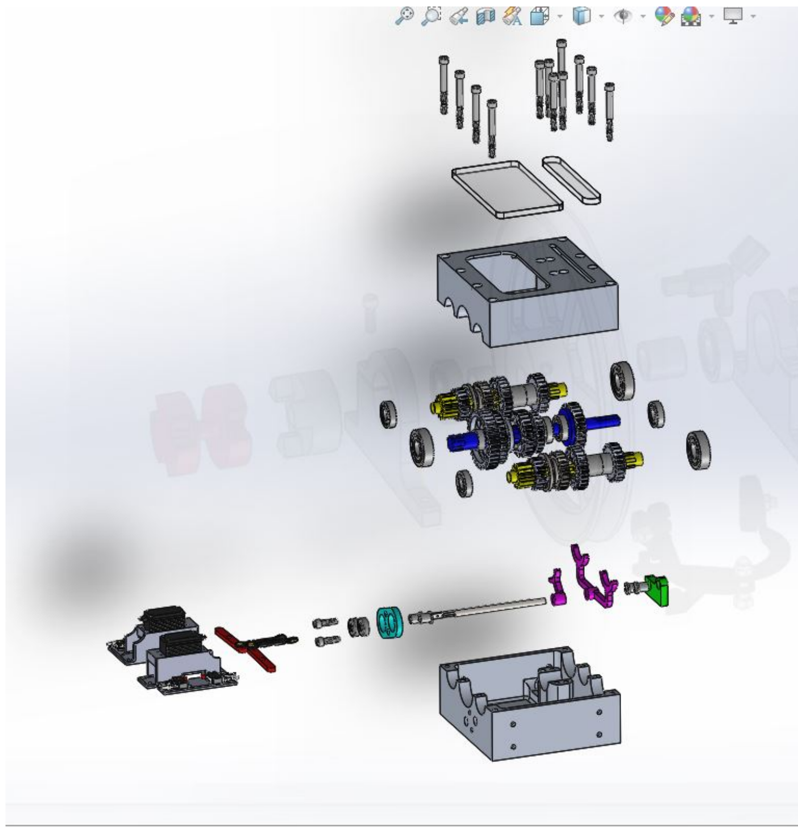
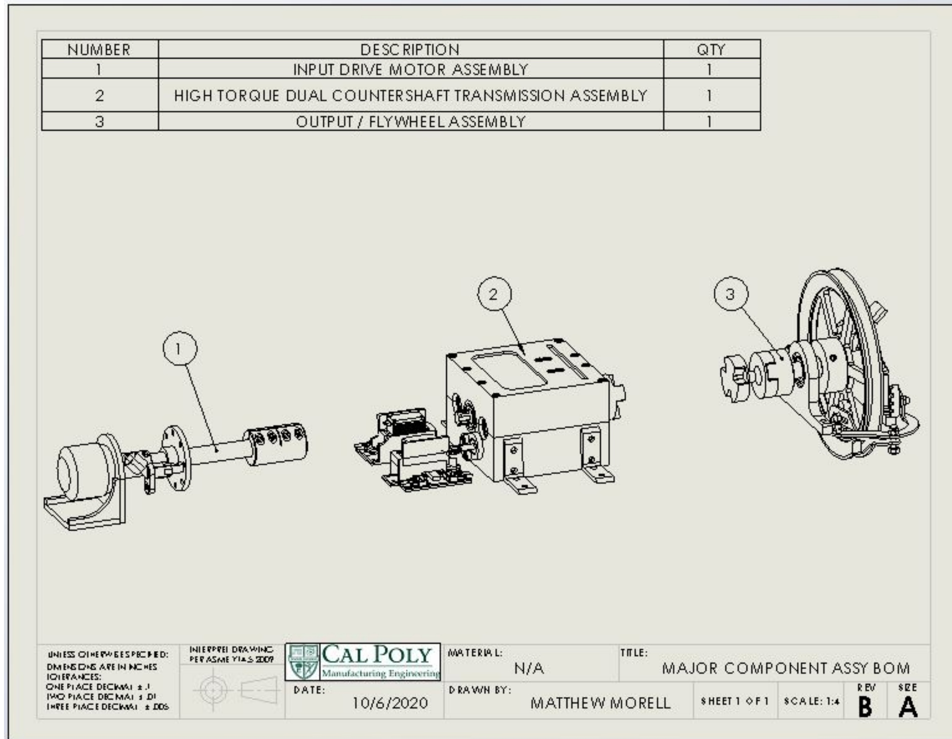
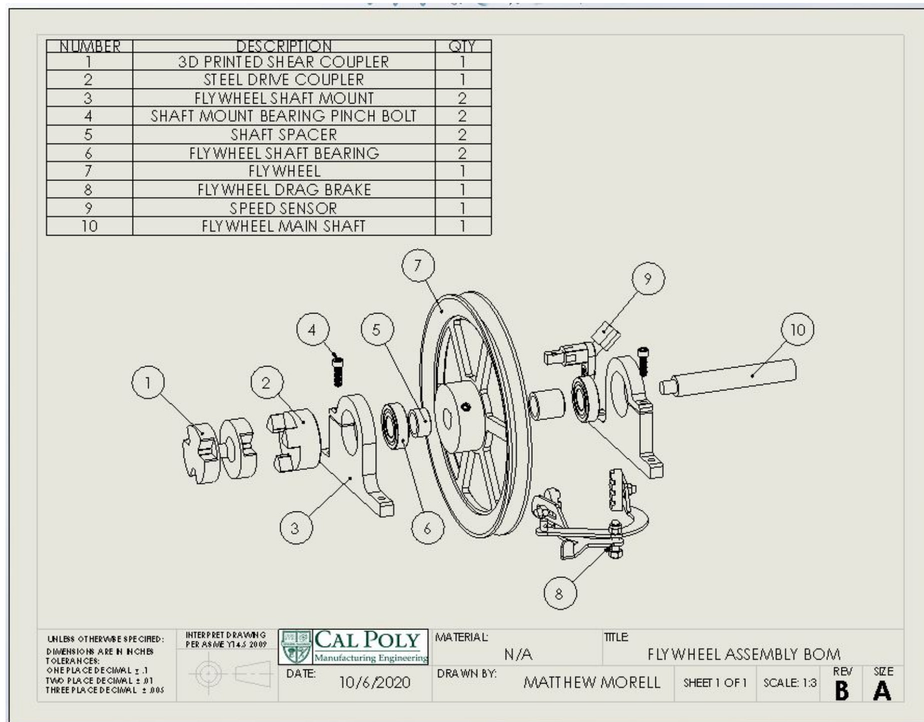


Figure D4: Transmission Sub-Assembly Exploded View



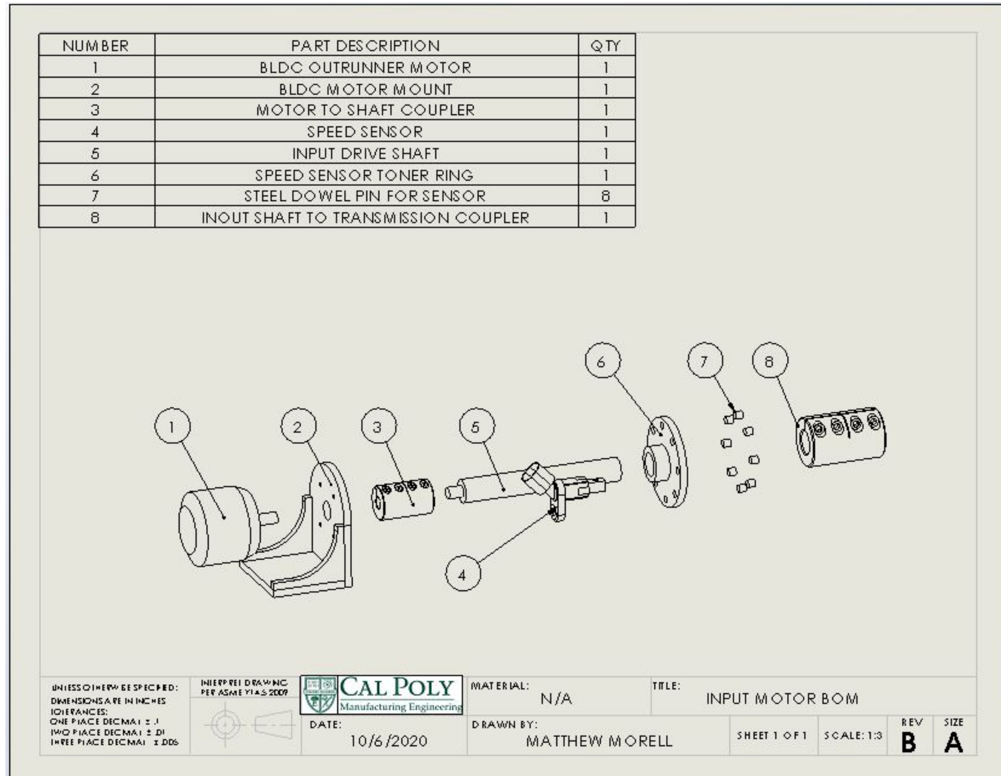
NUMBER	DESCRIPTION	QTY
1	INPUT DRIVE MOTOR ASSEMBLY	1
2	HIGH TORQUE DUAL COUNTERSHAFT TRANSMISSION ASSEMBLY	1
3	OUTPUT / FLYWHEEL ASSEMBLY	1

Figure D5 Subassemblies of the project



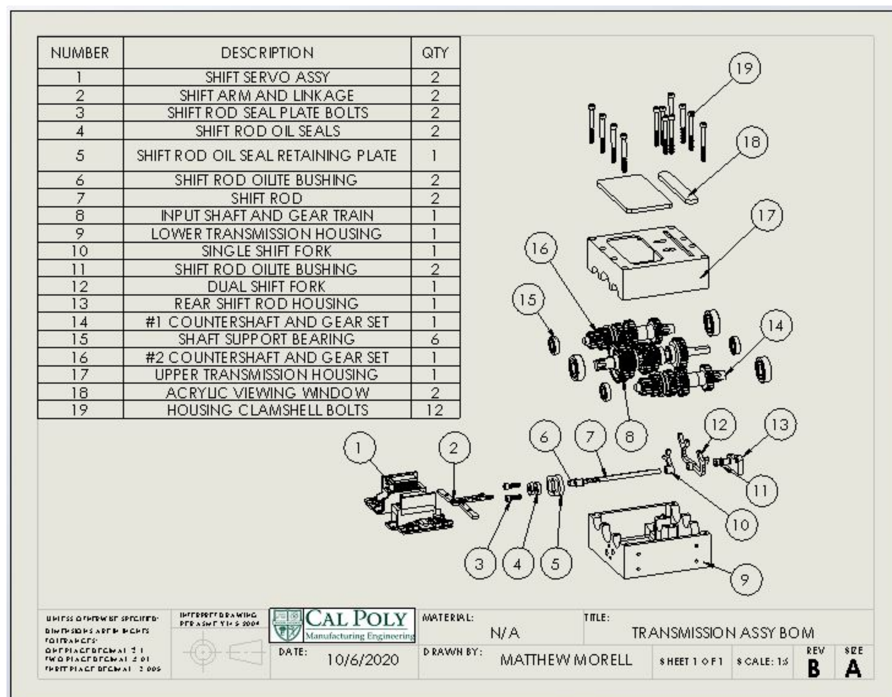
NUMBER	DESCRIPTION	QTY
1	3D PRINTED SHEAR COUPLER	1
2	STEEL DRIVE COUPLER	1
3	FLYWHEEL SHAFT MOUNT	2
4	SHAFT MOUNT BEARING PINCH BOLT	2
5	SHAFT SPACER	2
6	FLYWHEEL SHAFT BEARING	2
7	FLYWHEEL	1
8	FLYWHEEL DRAG BRAKE	1
9	SPEED SENSOR	1
10	FLYWHEEL MAIN SHAFT	1

Figure D6 Flywheel subassembly BOM



NUMBER	PART DESCRIPTION	QTY
1	BLDC OUTRUNNER MOTOR	1
2	BLDC MOTOR MOUNT	1
3	MOTOR TO SHAFT COUPLER	1
4	SPEED SENSOR	1
5	INPUT DRIVE SHAFT	1
6	SPEED SENSOR TONER RING	1
7	STEEL DOWEL PIN FOR SENSOR	8
8	INOUT SHAFT TO TRANSMISSION COUPLER	1

Figure D7 Input motor subassembly BOM



NUMBER	DESCRIPTION	QTY
1	SHIFT SERVO ASSY	2
2	SHIFT ARM AND LINKAGE	2
3	SHIFT ROD SEAL PLATE BOLTS	2
4	SHIFT ROD OIL SEALS	2
5	SHIFT ROD OIL SEAL RETAINING PLATE	1
6	SHIFT ROD OILITE BUSHING	2
7	SHIFT ROD	2
8	INPUT SHAFT AND GEAR TRAIN	1
9	LOWER TRANSMISSION HOUSING	1
10	SINGLE SHIFT FORK	1
11	SHIFT ROD OILITE BUSHING	2
12	DUAL SHIFT FORK	1
13	REAR SHIFT ROD HOUSING	1
14	#1 COUNTERSHAFT AND GEAR SET	1
15	SHAFT SUPPORT BEARING	6
16	#2 COUNTERSHAFT AND GEAR SET	1
17	UPPER TRANSMISSION HOUSING	1
18	ACRYLIC VIEWING WINDOW	2
19	HOUSING CLAMSHELL BOLTS	12

Figure D8 Transmission Gearbox subassembly BOM

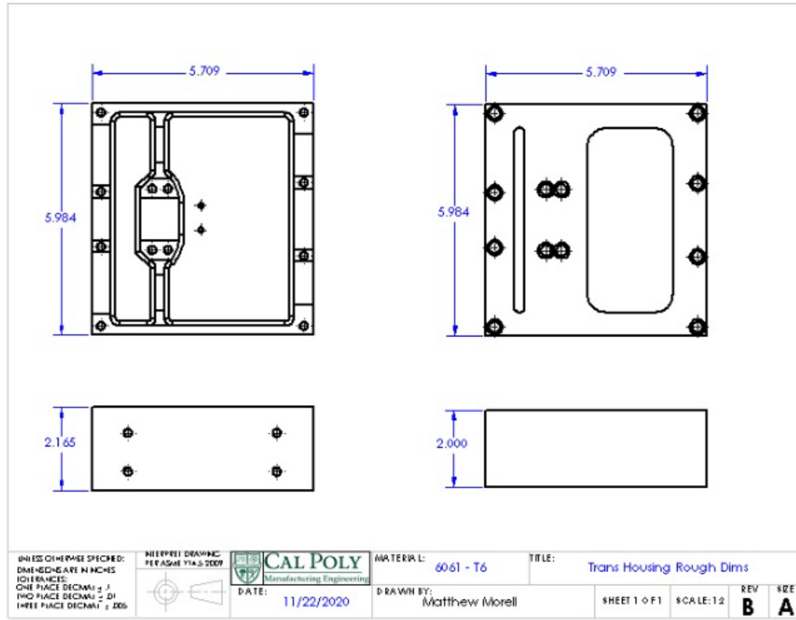


Figure D8 Transmission Gearbox subassembly BOM

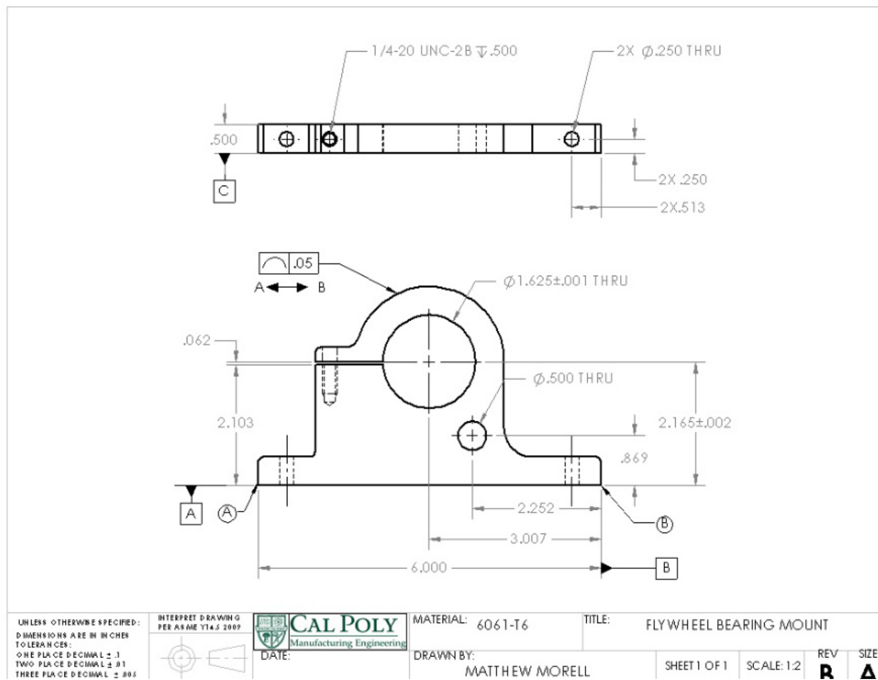


Figure D9 Flywheel Bearing Mount

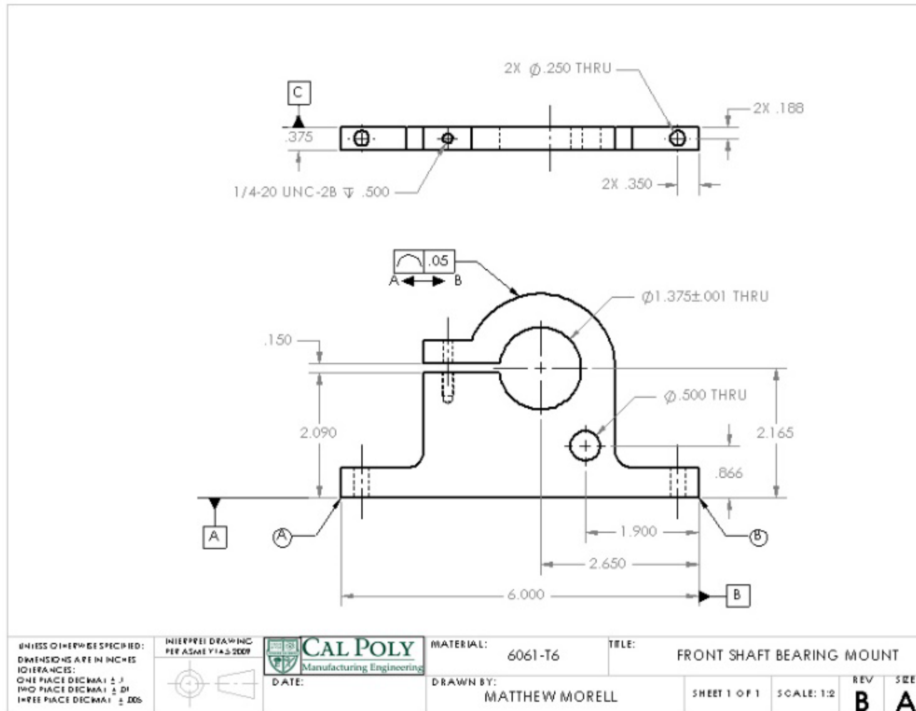


Figure D10 Front Shaft Bearing Mount

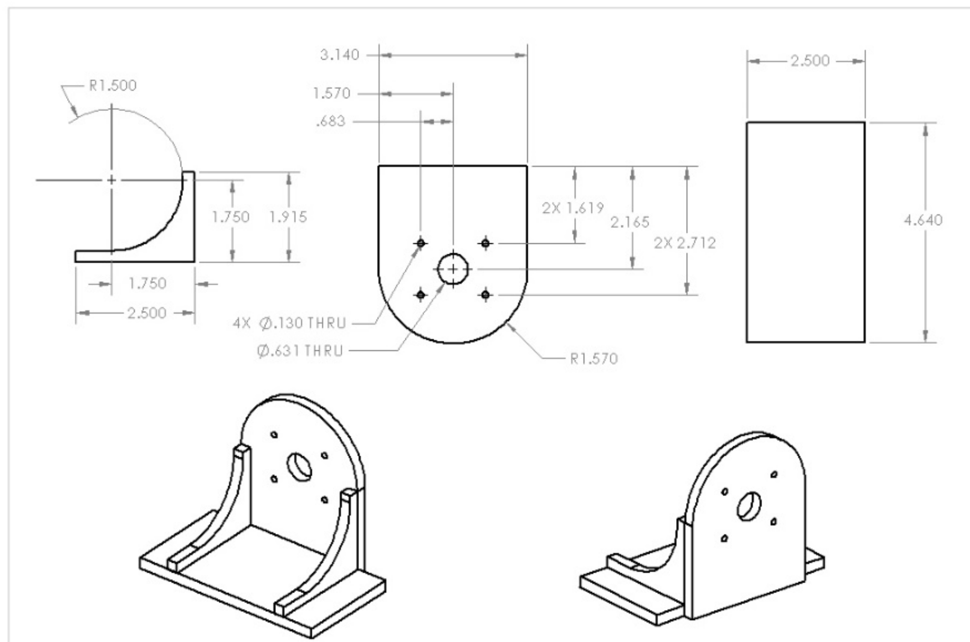


Figure D11 Motor Mount Assembly

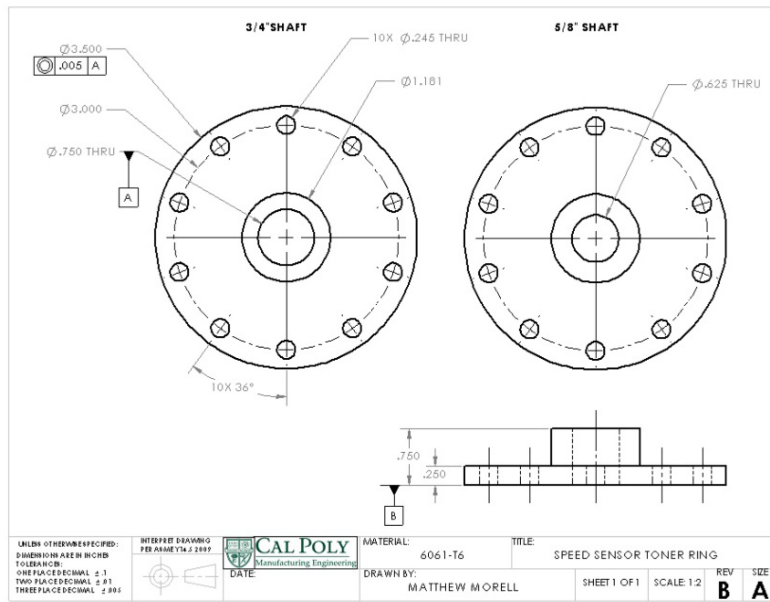


Figure D12 Input and Output RPM Sensor Wheel

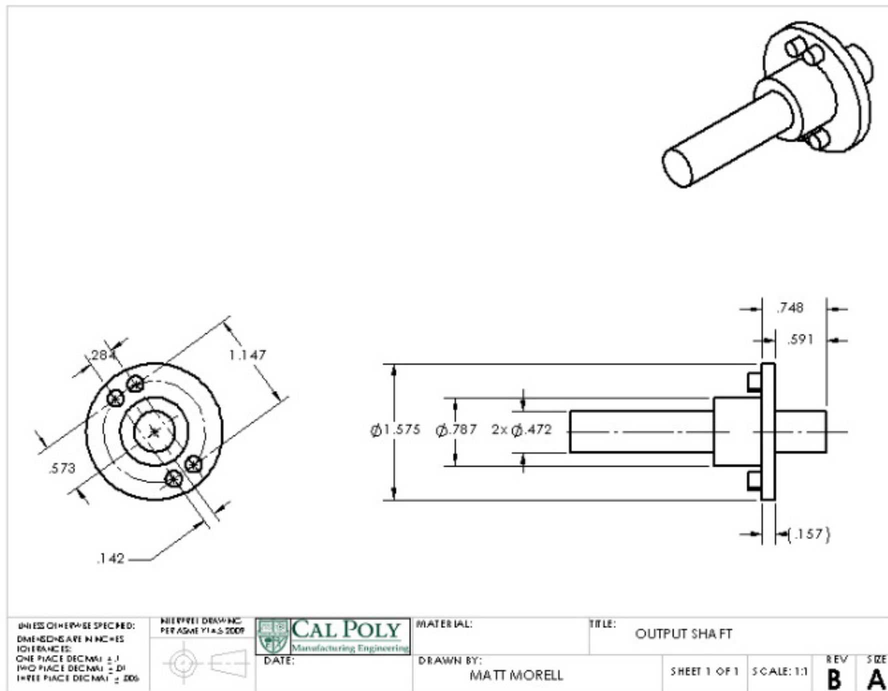


Figure D13 Recombining Shaft Drawing

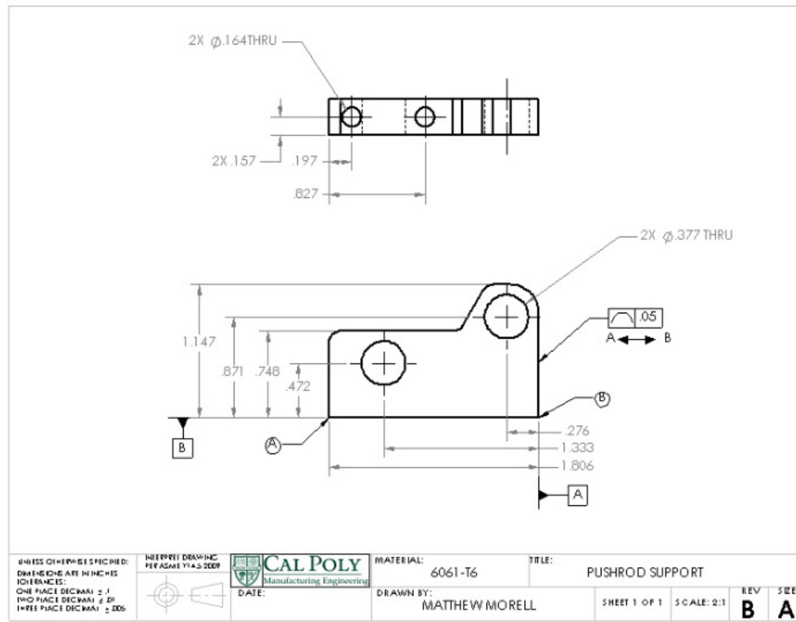


Figure D14 Rear Pushrod Support

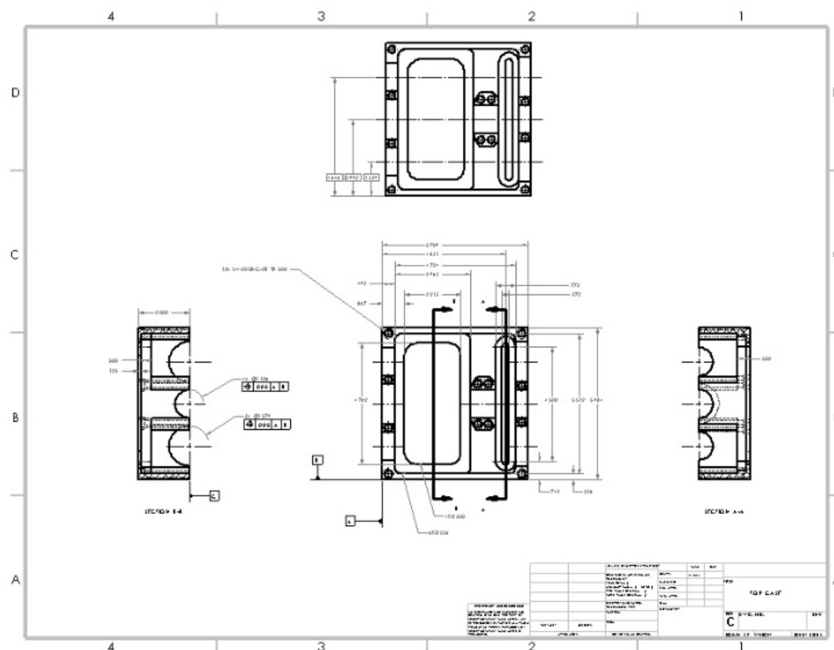


Figure D15 Upper Transmission Housing

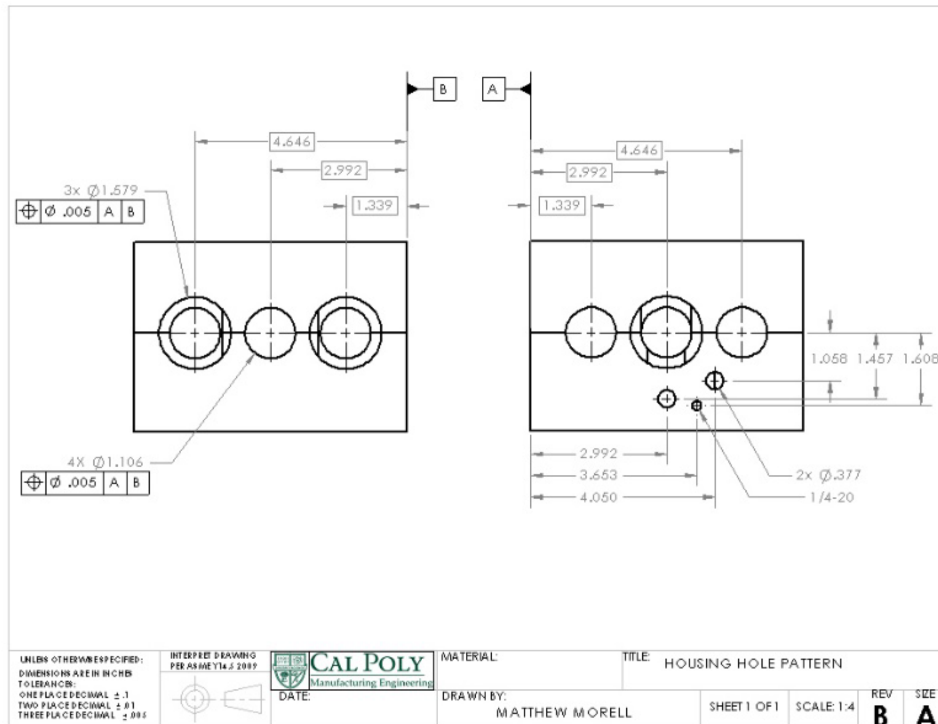


Figure D16 Upper Transmission Housing

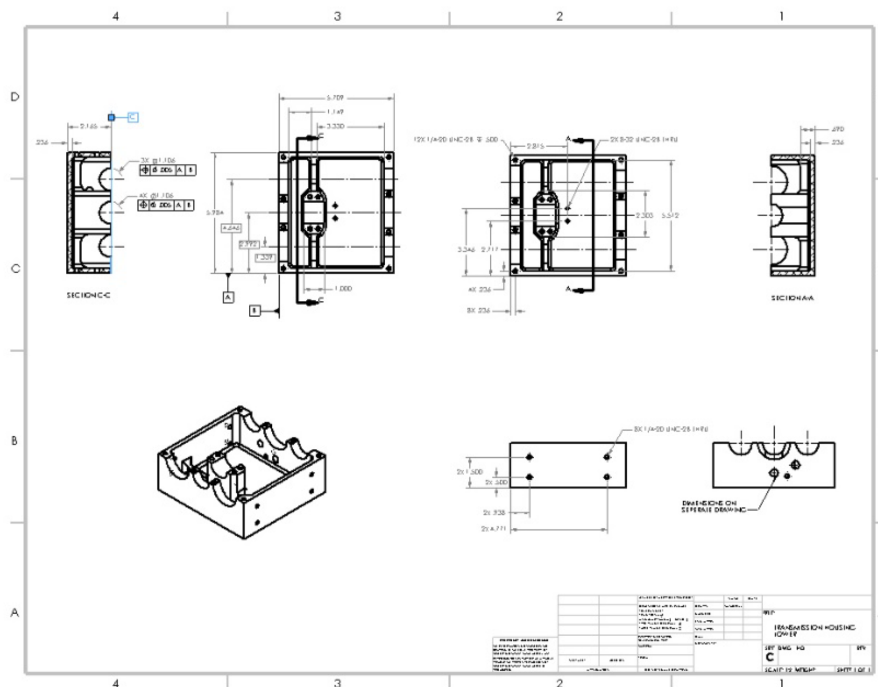


Figure D17 Lower Transmission Housing

Figure D18 Billet Aluminum Shifter

Table D1: Input motor specification table

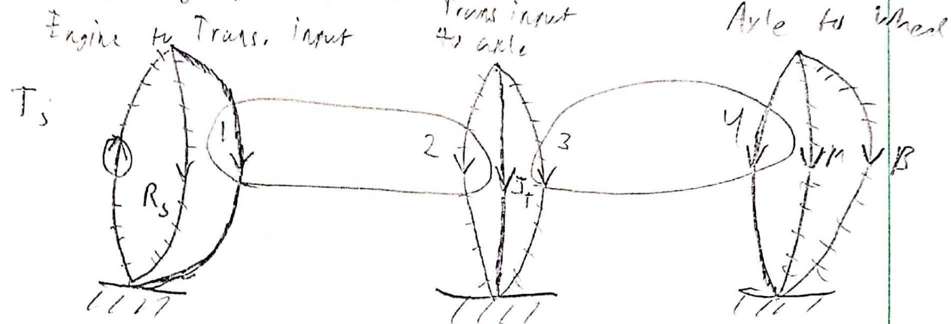
Input Motor Req		
Kv	160.000	rmp/volt
Torque	0.060	N*m/amp
Torque (in spec units)	0.528	in*lb/amp
RPM max	8000.000	RPM
Current Capacity	20.000	amps
Total Torque	10.565	in*lb
Total Torque (metric)	1.194	N*m
Angular Acceleration	0.309	rad/s^2
Angular Acceleration (metric)	119.366	rad/s^2
Time to Speed	1354.859	s
Time to speed (metric)	3.509	s
Power Draw	1000.000	Watts

Appendix E: Equivalent Inertia seen by the Transmission Derivation

Zack Gordon | Project GHOST | 10/23/2020 | 1/4

Equivalent inertia seen by the transmission:

Use linear graph – normal tree method:



Where T_s is the engine input torque, R_s is a general internal engine resistance, J_T is the total rotational inertia of the transmission, m is the mass of the vehicle, and B is a general drag resistance.

Assume the differential from the transmission output to the axle is negligible.

Other parameters:

$N \equiv$ the general gear reduction from engine input to transmission output

$r_w \equiv$ wheel radius

Cont →

Scanned with CamScanner

Transformer matrix for (1 2):

$$\begin{bmatrix} \Omega_1 \\ T_1 \end{bmatrix} = \begin{bmatrix} N & 0 \\ 0 & -\frac{1}{N} \end{bmatrix} \begin{bmatrix} \Omega_2 \\ T_2 \end{bmatrix}$$

$$\Omega_1 = N \Omega_2$$

$$T_1 = -\frac{1}{N} T_2$$

Transformer matrix for (3 4):

$$\begin{bmatrix} \Omega_3 \\ T_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{r_w} & 0 \\ 0 & -r_w \end{bmatrix} \begin{bmatrix} V_4 \\ F_4 \end{bmatrix}$$

$$\Omega_3 = \frac{1}{r_w} V_4$$

$$T_3 = -r_w F_4$$

Write out elemental equations:

$$F_m = m \frac{dV_m}{dt} \quad (1)$$

$$F_B = B V_B \quad (2)$$

$$V_4 = r_w \Omega_3 \quad (3)$$

$$T_3 = -r_w F_4 \quad (4)$$

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T} T_{J_T} \quad (5)$$

$$T_2 = -N T_1 \quad (6)$$

$$\Omega_1 = N \Omega_2 \quad (7)$$

cont →

Scanned with CamScanner

Write out constraint equations:

$V_B = V_4$ (8)

$V_m = V_4$ (9)

$\Omega_3 = \Omega_{J_T}$ (10)

$\Omega_2 = \Omega_T$ (11)

$F_4 = -F_m - F_B$ (12)

$T_{S_{J_T}} = -T_3 - T_2$ (13)

$T_1 = T_s - R_s$ (14)

(13), (4), (6) → (5):

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T}\left(r_w F_4 + N T_1\right) \quad (15)$$

(12), (14) → (15):

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T}\left[r_w(-F_m - F_B) + N(T_s - R_s)\right] \quad (16)$$

(1), (2) → (16):

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T}\left[r_w\left(-m\frac{dV_m}{dt} - BV_B\right) + N(T_s - R_s)\right] \quad (17)$$

(8), (9) → (17):

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T}\left[r_w\left(-m\frac{dV_4}{dt} - BV_4\right) + N(T_s - R_s)\right] \quad (18)$$

(3), (10) → (18):

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T}\left[r_w\left(-m\frac{d}{dt}(r_w\Omega_{J_T}) - Br_w\Omega_{J_T}\right) + N(T_s - R_s)\right] \quad (19)$$

cont →

CS Scanned with CamScanner

Continue derivation:

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T}\left[-r_w^2 m \frac{d\Omega_{J_T}}{dt} - B r_w^2 \Omega_{J_T} + N(T_s - R_s)\right]$$

$$\frac{d\Omega_{J_T}}{dt} + \frac{r_w^2}{J_T} m \frac{d\Omega_{J_T}}{dt} = \frac{N}{J_T}(T_s - R_s) - \frac{B r_w^2}{J_T}\Omega_{J_T}$$

$$\therefore \quad \frac{d\Omega_{J_T}}{dt} = \frac{1}{1 + \frac{r_w^2}{J_T} m}\left[\frac{N}{J_T}(T_s - R_s) - \frac{B r_w^2}{J_T}\Omega_{J_T}\right]$$

$$\frac{d\Omega_{J_T}}{dt} = \frac{1}{J_T + r_w^2 m}\left[N(T_s - R_s) - B r_w^2 \Omega_{J_T}\right]$$

By convention, this denominator is the equivalent rotational inertia seen by the transmission.

$$\therefore \quad J_{eq} = J_T + r_w^2 m$$

If it is assumed that $J_T \ll r_w^2 m$,

$$\therefore \quad \boxed{J_{eq} \approx r_w^2 m}$$

CS Scanned with CamScanner

Appendix F: Project Budget/Purchasing

PROJECT BUDGET

Sponsor Budget	$ 2,000.00
MESFAC-CPCONNECT	$ -
Total Budget	$ 2,000.00
Spent	$ 1,925.52
Remaining Budget	$ 74.48

Spent / Remaining Budget

ITEM	Subteam Team	Quantity	Cost Per Item	Cost Total	Order Date	ETA	Status	LINK	Purchaser	Requestee	Reimbursement Status
Drivetrain Components	Manufacturing	2	$ 68.00	$ 136.00			Delivered	https://www.ebay.com/itm/	Matt	Austin	YES
Teensy 3.5 Board	Controls	1	$ 34.86	$ 34.86	4/28/2020	5/6/2020	Delivered	https://www.amazon.com/g	Austin	Austin	YES
Test Servo's	Controls	1	$ 19.00	$ 19.00	5/11/2020	5/14/2020	Delivered	https://www.amazon.com/g	Austin	Austin	YES
3rd Drive Train Component	Manufacturing	1	$ 53.20	$ 53.20	5/14/2020	5/21/2020	Delivered		Austin	Austin	YES
4th Drive Train Component	Manufacturing	1	$ 75.10	$ 75.10	6/4/2020	9/14/2020	Delivered	https://www.amazon.com/g	Austin	Austin	YES
Input Motor	Controls	1	$ 75.00	$ 75.00	9/17/2020	9/24/2020	Delivered	https://flipsky.net/collectio	Matt	Matt	
ESC	Controls	1	$ 70.00	$ 70.00	9/17/2020	2/24/2020	Delivered	https://flipsky.net/collectio	Matt	Matt	
Servo	Controls	2		$ -			Delivered	http://hitecrcd.com/product	Matt	Matt	
6v BEC Regulator	Controls	1	$ 39.95	$ 39.95			Delivered	https://www.castlecreations	Matt	Matt	
XT60 Female-Male Splitter	Controls	1	$ 5.84	$ 5.84			Delivered	https://www.amazon.com/E	Matt	Zack	
2.1x5.5 Male Header Cable	Controls	2	$ 9.89	$ 19.78			Delivered	https://www.amazon.com/4	Austin	Zack	
2.1x5.5 Female Barrel Header 90 Degree	Controls	35	$ 0.54	$ 18.90			Delivered	https://www.digikey.com/p	Austin	Zack	
Foot Pedal Potentiometer	Controls	1	$ 40.15	$ 40.15			Delivered	https://www.amazon.com/g	Matt	Austin	
RPM Sensor	Controls	4	$ 32.08	$ 128.32			Delivered	https://www.digikey.com/p	Austin	Zack	
LCD Screen	Controls	2	$ 7.99	$ 15.98			Delivered	https://www.amazon.com/H	Austin	Zack	
10' 22 awg 2 wire	Controls	5	$ 3.99	$ 19.95	9/29/2020		Delivered	https://www.servocity.com/	Matt	Matt	
50' 22 awg 3 wire	Controls	1	$ 15.99	$ 15.99	9/29/2020		Delivered	https://www.servocity.com/	Matt	Matt	
Servo Connectors - Crimp on	Controls	2	$ 10.99	$ 21.98			Delivered	https://www.amazon.com/F	Austin	Matt	
EMO		1	$ 42.57	$ 42.57			Delivered	https://www.mcmaster.com	Austin	Zack	
Heat Shrink		2	$ 6.99	$ 13.98			Delivered	https://www.amazon.com/S	Austin	Zack	
Solder		1	$ 8.59	$ 8.59			Delivered	https://www.amazon.com/N	Austin	Zack	
Screw Terminal Blocks		2	$ 1.48	$ 2.96			Delivered	https://www.digikey.com/e	Austin	Zack	
Potentiometer 10kom		3	$ 1.08	$ 3.24			Delivered	https://www.digikey.com/e	Austin	Austin	
10mm to 10mm coupler		1	$ 12.00	$ 12.00			Delivered	https://www.mcmaster.com	Austin		
20mm x 12" alu input shaft		1	$ 24.00	$ 24.00			Delivered	https://www.mcmaster.com	Austin		
25pk .25" dowl pin for input sensor wheel		1	$ 4.10	$ 4.10			Delivered	https://www.mcmaster.com	Austin		
5/8 shaft coupler		1	$ 16.00	$ 16.00			Delivered	https://www.mcmaster.com/6408K73/			
.5" 6x6 polycarb sheet		2	$ 10.56	$ 21.12			Delivered	https://www.mcmaster.com	Austin		
pushrod shaft seal		2	$ 8.00	$ 16.00			Delivered	https://www.mcmaster.com	Austin		
pushrod seal plate bolt 50pk		1	$ 7.86	$ 7.86			Delivered	https://www.mcmaster.com	Austin		
pushrod oillite bushings		4	$ 1.11	$ 4.44			Delivered	https://www.mcmaster.com	Austin		
pushrods		2	$ 5.00	$ 10.00			Delivered	https://www.mcmaster.com	Austin		
smaller 12mm ID bearing		5	$ 8.48	$ 42.40			Delivered	https://www.mcmaster.com	Austin		
larger 17mm ID bearing		3	$ 6.72	$ 20.16			Delivered	https://www.mcmaster.com	Austin		
gear shaft spacers		8	$ 6.23	$ 49.84			Delivered	https://www.mcmaster.com	Austin		
Main housing split bolts 25pk		1	$ 8.84	$ 8.84			Delivered	https://www.mcmaster.com	Austin		
pushrod support mount bolts 25pk		1	$ 6.37	$ 6.37			Delivered	https://www.mcmaster.com	Austin		
upper housing aluminum		1	$ 45.00	$ 45.00			Delivered	https://www.midweststeels	Matt		
lower housing aluminum		1	$ 45.00	$ 45.00			Delivered	https://www.midweststeels	Matt		
trans housing mounts		4	$ 5.25	$ 21.00			Delivered	https://www.mcmaster.com	Austin		
shear coupler drive dogs		4	$ 11.00	$ 44.00			Delivered	mcmaster 6413K144	Austin		
flywheel bearings		2	$ 11.12	$ 22.24			Delivered	https://www.mcmaster.com	Austin		
flywheel shaft		1	$ 11.50	$ 11.50			Delivered	https://www.mcmaster.com	Austin		
flywheel pinch bolt 50pk		1	$ 7.86	$ 7.86			Delivered	https://www.mcmaster.com	Austin		
flywheel shaft spacer		1	$ 9.00	$ 9.00			Delivered	https://www.mcmaster.com	Austin		

flywheel spacer		1	$ 10.00	$ 10.00			Delivered	https://www.mcmaster.com	Austin		
0805 10k Resistor 1/4W		100	$ 0.03	$ 2.83			Delivered	https://www.digikey.com/e	Austin	Austin	
0805 1K Resistor 1/4W		10	$ 0.07	$ 0.70			Delivered	https://www.digikey.com/e	Austin	Austin	
Capacitor 47uf 6.3v		10	$ 0.33	$ 3.25			Delivered	https://www.digikey.com/e	Austin	Austin	
Capacitor .1uf 16v		10	$ 0.17	$ 1.73			Delivered	https://www.digikey.com/e	Austin	Austin	
Arduino Header Pack		6	$ 1.50	$ 9.00			Delivered	https://www.digikey.com/e	Austin	Austin	
Servo Header 3 pin		30	$ 0.29	$ 8.82			Delivered	https://www.digikey.com/e	Austin	Austin	
Servo Header 4 pin		6	$ 0.33	$ 1.98			Delivered	https://www.digikey.com/e	Austin	Austin	
Digikey shipping		1	$ 10.99	$ 10.99			Delivered		Austin		
1/4" alu plate		1	$ 25.50	$ 25.50			Delivered		Matt		
1/2" alu plate		1	$ 32.00	$ 32.00			Delivered		Matt		
flywheel #2 6"		1	$ 31.71	$ 31.71			Delivered		Matt		
flywheel #3 8"		1	$ 57.20	$ 57.20			Delivered	https://www.bmikarts.com/	Matt		
flywheel shaft reducer			$ 6.03	$ -			Delivered	https://www.mcmaster.com	Matt		
1/8" alu plate		1	$ 18.00	$ 18.00			Delivered		Matt		
1/4-20 x 2.5"		1	$ 9.80	$ 9.80			Delivered		Matt		
1/4" shaft collar		6	$ 2.50	$ 15.00			Delivered		Matt		
Motor #2		1	$ 100.00	$ 100.00			Delivered		Matt		
ESC#3 (Hobbywing)		1	$ 250.00	$ 250.00			Delivered		Matt		
1/4" alu plate		1	$ 19.95	$ 19.95			Delivered		Matt		
Buttons		1	$ 6.99	$ 6.99			Delivered	https://www.amazon.com/g	Austin		

Appendix G: Shift and Time to Speed Testing

Table G.1 Shift Time and Time to Speed Final Testing Results

Test #	Shift Time (s)	Time to Speed (s)
1	1.00	6.82
2	0.95	5.80
3	0.96	4.85
4	0.95	5.50
5	0.96	5.26
6	0.90	5.44
7	0.91	4.95
8	0.85	5.64
9	0.90	5.71
10	0.95	5.20
11	1.05	5.60
12	0.90	5.20
13	1.05	4.79
14	1.00	4.66
15	0.96	5.83
16	0.90	5.60
17	1.10	5.40
18	0.80	4.82
19	0.85	4.77
20	0.95	5.50
21	0.90	4.20
22	0.83	4.52
23	0.95	4.45
24	0.90	5.30
25	0.95	4.85

Appendix H: Project Code

```
/** @file main.cpp
 *    This file contains the main code for running a model transmission.
 *    Includes the functions necessary to operate the transmission during normal running
 *    and when a gear shift is requested by the user.
 *
 *  @author  Austin Conrad & Zack Gordon
 *
 */

#include <Arduino.h>
#include <PrintStream.h>
#if (defined STM32L4xx || defined STM32F4xx)
    #include <STM32FreeRTOS.h>
#endif
#include "taskshare.h"
#include "LiquidCrystal_I2C.h"
#include "Servo.h"
#include "HardwareTimer.h"
#include "debouncer.h"


// Create hardware timer object for calculating deltas between pulses
static HardwareTimer *Timer = new HardwareTimer(TIM2);

// Create hardware timer object for calculating shift time
static HardwareTimer *Timer2 = new HardwareTimer(TIM5);

uint16_t current_ticks_out = 0;     // Current timer tick count (was uint32_t) for output
uint16_t old_ticks_out = 0;         // Old timer tick count (was uint32_t) for output
int16_t delta_ticks_out = 0;        // Change in ticks (was float when the overflow bug was going on) for output

uint8_t ISRout_state = 0;           // ISR state variable for output

uint16_t current_ticks_in = 0;      // Current timer tick count (was uint32_t) for input
uint16_t old_ticks_in = 0;          // Old timer tick count (was uint32_t) for input
int16_t delta_ticks_in = 0;         // Change in ticks (was float when the overflow bug was going on) for input

uint8_t ISRin_state = 0;        // ISR state variable for input

const int16_t TMR1_Prescaler = 2440;        //Timer overflow prescaler value
const int16_t TMR2_Prescaler = 8000;        //Timer overflow prescaler value (8k should make the shift timer counter resolution be
~.0001s/tick)

// Shares and queues should go here

    // ISR related
    Share<int16_t> ticks_delta_out ("Output Change in Ticks");    // Share for output interrupt
    Share<int16_t> ticks_delta_in ("Input Change in Ticks");      // Share for input interrupt

    // Shifting/revmatching related
    Share<float>    accel_in       ("Acceleration");        // Share for the acceleration on the input
    Share<float>    accel_out       ("Acceleration");       // Share for the acceleration on the output
    Share<int8_t>   shift_req ("Shift Request");            // Flag for shift requests, 0 = no request, 1= shift into 1st, 2 = shift into 2nd
    Share<uint16_t> radpsec_in ("Radians per second");      // Angular speed in radians per second
    Share<uint16_t> revpmin_in ("Revolutions per minute");  // Angular speed in revolutions per minute
    Share<uint16_t> radpsec_out ("Radians per second");     // Angular speed in radians per second
    Share<uint16_t> revpmin_out ("Revolutions per minute"); // Angular speed in revolutions per minute
    Share<uint8_t>  servo1pos ("Servo 1 position");         // Super boolean for dictating where servo 1 should shift
    Share<uint8_t>  servo2pos ("Servo 2 position");         // Super boolean for dictating where servo 2 should shift
    Share<uint8_t>  ESC_Armed_FLag ("ESC is ready");        // Boolean for knowing if the esc has had enough time to arm 0 = not armed, 1 =
armed
    Share<uint8_t>  Shift_Busy_Flag ("Task is Shifting");   // Shifting task is in the middle of a shift, don't try and shift again

    // Display task related
    Share<int8_t>   RPM_DISP ("RPM Display");               // Share for RPM Display (Value between 0-16)
    Share<int8_t>   MODE ("Mode");                          // Share for the total MCU Mode (Value )
    Share<int8_t>   GEAR_STATE ("Gear State");              // Share for the Gear State (0=N, 1 = 1st, 2 = 2nd, 3 = 3rd, 4 = 4th)
    Share<float>    SHIFT_TIME ("Last Shift TIme");         // Share Variable for the most recent shift time
    Share<int8_t>   MESSAGE ("MESSAGE");                    // Share variable for the message state
    Share<int16_t>  PWM_VAL ("Motor Output");               // Share variable for Motor PWM Value Request (1000-2000) for (0-100% Duty Cycle)
    Share<int16_t>  Servo1PWM ("Servo 1 Output");           // Share variable
    Share<int8_t>   PEDAL_OVERRIDE ("Pedal Overide");       // Share Variable for the Pedal Overide During Shift Request

    Share<bool>   RPM_UPD_FLAG ("RPM Disp Update");         //Share Variable for the Display to update the new RPM value
    Share<bool>   MODE_UPD_FLAG ("MODE Disp Update");       //Share Variable for the Display to update the new MODE value
    Share<bool>   GEAR_UPD_FLAG ("GEAR Disp Update");       //Share Variable for the Display to update the new GEAR value
    Share<bool>   MESSAGE_UPD_FLAG ("MESSAGE Disp Update"); //Share Variable for the Display to update the new MESSAGE value
    Share<bool>   SHIFT_UPD_FLAG ("SHIFT Disp Update");     //Share Variable for the Display to update the new SHIFT value

    Share<uint8_t> Button_State_Upshift ("Upshift Button State");           //Share for Upshift Button State (0-not pressed, 1 - pressed)
    Share<uint8_t> Button_State_Downshift ("Downshift Button State");       //SHare for Downshift Button State (0 - not pressed, 1 - pressed)
    Share<uint8_t> Button_State_ExtraSig ("Extra Signal Button State");     //Share for Extra Sig Button State (0- not pressed, 1- pressed)
    Share<uint8_t> Button_State_ArmSig ("Arm Signla Button State");         //Share for the Arm Sig Button State (0-not pressed, 1 - pressed)
```

```
    Share<uint8_t> Button_State_LS1 ("1st Gear Switch State");              //Share for servo 1, 1st gear switch state (0-not pressed, 1 - 
pressed)
    Share<uint8_t> Button_State_LS1N ("1st Gear Neutral Button State");     //Share for servo 1, 1st gear neutral state (0-not pressed, 1 - 
pressed)
    Share<uint8_t> Button_State_LS2 ("2nd Gear Switch State");              //Share for servo 2, 2nd gear switch state (0-not pressed, 1 - 
pressed)
    Share<uint8_t> Button_State_LS2N ("2nd Gear Neutral Button State");     //Share for servo 2, 2nd gear neutral state (0-not pressed, 1 - 
pressed)

    //***TESTING: Might change which values are which Share for the Button Flag (0-no change, 1-Upshift, 2 - downshift)
    Share<uint8_t> Button_Flag  ("Button Flag");


/** @brief   The interrupt service routine used to find the speed of the output shaft.
 *  @details The method for calculating the output shaft speed is interrupt driven.
 *           Using a hall effect sensor and a plate with evenly spaced metal pins, the speed can be calculated
 *           by finding the time delta between sensor pulses in microcontroller clock ticks.
 *  @param   current_ticks_out   The current value of the hardware timer clock ticks.
 *  @param   old_ticks_out       The previous value of the hardware timer clock ticks.
 *  @param   delta_ticks_out     The difference between the current and previous values of clock ticks.
 *  @param   ISRout_state        Variable used to determine the state of the interrupt service routine.
 */
void SPEEDSENSOROUT_ISR()
{
       //Serial << "Interupt" << endl;
        if(ISRout_state == 1)
       {
         current_ticks_out = Timer->getCount();                             // Get the current ticks count
         if(current_ticks_out == old_ticks_out)
         {
            old_ticks_out     = current_ticks_out;                          // Store the current tick value
         }
         else
         {
            delta_ticks_out   = current_ticks_out - old_ticks_out;          // Calculate the delta
            old_ticks_out     = current_ticks_out;                          // Store the current tick value
            //Serial << "ISROUT Delta =" << delta_ticks_out<<endl;          //
            ticks_delta_out.ISR_put(delta_ticks_out);                       // put delta into share
         }

       }
       else
       {
         old_ticks_out = Timer->getCount();                     // Store the current tick value for the first time
         ticks_delta_out.ISR_put(0);                            // put delta into share
         ISRout_state = 1;                                      // Move into the main ISR state
       }
}

/** @brief   The interrupt service routine used to find the speed of the input shaft.
 *  @details The method for calculating the input shaft speed is interrupt driven.
 *           Using a hall effect sensor and a plate with evenly spaced metal pins, the speed can be calculated
 *           by finding the time delta between sensor pulses in microcontroller clock ticks.
 *  @param   current_ticks_in   The current value of the hardware timer clock ticks.
 *  @param   old_ticks_in       The previous value of the hardware timer clock ticks.
 *  @param   delta_ticks_in     The difference between the current and previous values of clock ticks.
 *  @param   ISRin_state        Variable used to determine the state of the interrupt service routine.
 */
void SPEEDSENSORIN_ISR()
{
       if(ISRin_state == 1)
       {
         current_ticks_in = Timer->getCount();                          // Get the current ticks count
         if(current_ticks_in == old_ticks_in)
         {
            old_ticks_in     = current_ticks_in;                       // Store the current tick value
         }
         else
         {
            delta_ticks_in   = current_ticks_in - old_ticks_in;        // Calculate the delta
            old_ticks_in     = current_ticks_in;                       // Store the current tick value
            ticks_delta_in.ISR_put(delta_ticks_in);                     // put delta into share
         }

       }
       else
       {
         old_ticks_in = Timer->getCount();                      // Store the current tick value for the first time
         ticks_delta_in.ISR_put(0);                             // put delta into share
         ISRin_state = 1;                                       // Move into the main ISR state
       }
}

/** @brief   The mastermind task.
 *  @details This task is the mastermind that interprets all of the user inputs
```

```
 *          and gives commands to the rest of the program given the context.
 *  @param   p_params     A pointer to function parameters which we don't use.
 *  @param   masterstate  Variable used to determine the state of the mastermind task.
 */
void task_mm (void* p_params)
{
    (void)p_params;                            // Shuts up a compiler warning

    uint8_t masterstate = 1;                   // Defines the state of the mastermind task ***TESTING should be 0 normally
    uint8_t But_flag = 0;                      // local button flag variable
    uint16_t UpShift_RPM_Threshold = 3000;     // Variable to check if we have hit sufficient RPM to make a shift to 2nd Gear
    uint16_t DownShift_RPM_Threshold = 1500;   // Variable to check if we have hit sufficient RPM to make a shift to 2nd Gear
    int8_t Current_gear = 0;                   // local variable for checking the gear
    uint16_t Current_RPM_in = 0;               //local variable for the current RPM in
    int8_t Shiftreq = 0;                       // local variable for checking to see if we are shifting

    // Init state
    // Set all shares and flags to appropriate starting values
    PEDAL_OVERRIDE.put(0);

    for (;;)
    {
        if (masterstate == 1)
        {
            Button_Flag.put(0); //***TESTING this shouldn't be here, I'm testing the buttons
            // Startup state (state 1)
            // Possible state where everything is told to move into a pre-armed state
            masterstate = 2;    //Setting mastermind to the hub state
        }
        else if (masterstate == 2)
        {
            // Hub state (state 2)
            //Check for errors
                // If there is an error somewhere, find out what it is and then act accordingly.
            Button_Flag.get(But_flag);          //getthe the local variable for the button press
            if(But_flag != 0)       //There was a button pressed
            {
                masterstate = 3;    //Setting the mastermind state to the button handler
            }
            else
            {
                //Code?
            }

        }
        else if (masterstate == 3)
        {
            // Button Handler (state 3)
                Button_Flag.get(But_flag);      // Get the button that was pressed
                // Based on what button was pressed, act accordingly
                if(But_flag == 1)           // Upshift Button has been pressed
                {
                    GEAR_STATE.get(Current_gear);   //getting the gear state and putting it into the current gear variable
                    revpmin_in.get(Current_RPM_in); //getting the current rpm on the input
                    shift_req.get(Shiftreq);  //getting the current shifting flag value

                    //Check to see if we are in 1st gear, past min shift rpm and if we are in the middle of a shift or not
                    if((Current_gear == 1) && (Current_RPM_in >= UpShift_RPM_Threshold) && (Shiftreq == 0))
                    {
                        shift_req.put(2);       // Setting the request for shift to second gear
                        Button_Flag.put(0);     // else clear flag
                        masterstate = 2;        // Set the mm task to the hub state
                    }
                    else
                    {
                        Button_Flag.put(0);     //else clear flag and return to hub state
                        masterstate = 2;        // Set the mm task to the hub state
                    }

                }
                else if(But_flag == 2)      // Downshift Button has been pressed
                {
                    GEAR_STATE.get(Current_gear);   //getting the gear state and putting it into the current gear variable
                    revpmin_in.get(Current_RPM_in); //getting the current rpm on the input
                    shift_req.get(Shiftreq);        //getting the current shifting flag value

                    //Check to see if we are in 1st gear, past min shift rpm and if we are in the middle of a shift or not
                    if((Current_gear == 2) && (Current_RPM_in <= DownShift_RPM_Threshold) && (Shiftreq == 0))
                    {
                        shift_req.put(1);       // Setting the request for shift to first gear
                        Button_Flag.put(0);     // else clear flag
                        masterstate = 2;        // Set the mm task to the hub state
                    }
                    else
                    {
```

```
                Button_Flag.put(0);     //else clear flag and return to hub state
                masterstate = 2;        // Set the mm task to the hub state
            }
        }
        else if(But_flag == 3)      // Extra Button has been pressed ***TESTING this button might fuck with timer values
        {
            //stuff Once we get through some other testing, check and see if this button is even usable
            Button_Flag.put(0);     //else clear flag and return to hub state
            masterstate = 2;        // Set the mm task to the hub state
        }
        /* else if(But_flag == 4)      // Arm Button has been pressed
        {
            //Check to see if this is the initial arm press
            //Set the appropriate armed values if so
            //Otherwise, clear flag
        } */
        else
        {
            //Nothing?
        }

        Button_Flag.put(0);             // Clear the button flag
        masterstate = 2;                // Go back to hub
    }
    else if (masterstate == 4)
    {
       // Error handler (state 4)
            // Depending on the error, act accordingly

            // Clear the error flag

            masterstate = 2;         // Go back to hub
    }

    vTaskDelay(50);
  }
}

/** @brief   Task which sends a PWM signal to the motor.
 *  @details This task sends PWM signals to the motor. The values for the PWM signals are received
 *           from the analog pedal task.
 *  @param   p_params    A pointer to function parameters which we don't use.
 *  @param   PWM_Val_Var Variable that contains the PWM value to be sent to the motor.
 */
void task_motor (void* p_params)
{
    (void)p_params;                             // Shuts up a compiler warning
    ESC_Armed_FLag.put(0);
    int16_t PWM_Val_Var = 1000;
    Servo Motor;                                    // Setting up the Motor Object
    Motor.attach(A0);                               // Setting the motor signal pin to output
    Motor.writeMicroseconds(PWM_Val_Var);           // Setting the motor to zero upon start up
    uint8_t Motor_State = 0;                        // Motor State Variable & set to 0 for init
    uint16_t Init_timer = 0;                        // Timer for the motor initialization
    uint8_t Arm_State = 0;                          // Local variable for the arm state

    for (;;)
    {
        if(Motor_State == 0)            // State = 0, waiting for ESC to initialize using timer: move to state 1 if arm is pressed
        {
            Init_timer ++;
            if(Init_timer >= 100)
            {
                Init_timer = 100;
                Button_State_ArmSig.get(Arm_State);
                if(Arm_State == 1)
                {   Serial << "Motor read that arm button was pressed" << endl;
                    ESC_Armed_FLag.put(1);                                  // Setting the esc armed flag to a 1
                    Motor_State = 1;
                }
            }

        }
        else if(Motor_State == 1)        // State = 1, Normal motor task
        {
            PWM_VAL.get(PWM_Val_Var);                                   // Getting the contents of share duty_cycle and
            //Serial << "PWM_VAL = "<< PWM_Val_Var << endl;             // putting it into duty_cycle_var
            Motor.writeMicroseconds(PWM_Val_Var);                       //Output the correct duty cycle
        }
        else
        {
            //Code
        }

        vTaskDelay(50);
```

```
    }
}

/** @brief   Task which sends a PWM signal to servo 1.
 *  @details This task sends PWM signals to servo 1. The values for the PWM signals are determined
 *           based on where the servo needs to be in order to make the correct shift.
 *  @param   p_params        A pointer to function parameters which we don't use.
 *  @param   servo1state     State variable that determines what state servo 1 task should be in.
 *  @param   neutral_pwm     PWM value for moving servo into neutral position
 *  @param   first_pwm       PWM value for moving servo into first gear position
 *  @param   servo1position  Super boolean for position. 1 = neutral, 2 = first gear
 */
void task_servo1 (void* p_params)
{
    (void)p_params;                                // Shuts up a compiler warning

    // CHANGE PWM VALUES ACCORDINGLY
    uint8_t  servo1state = 0;                       // State variable for servo 1 task
    uint16_t neutral_pwm = 1000;                    // PWM value for moving servo into neutral position
    uint16_t first_pwm   = 2000;                    // PWM value for moving servo into first gear position

    uint8_t  servo1position = 0;                     // Super boolean for position. 1 = neutral, 2 = first gear
    // Putting this zero here is only for startup ***Testing this should be here in the final itteration
    //servo1pos.put(servo1position);

    uint8_t ls1n;                                      // Flag for the neutral limit switch (0 = not in neutral, 1 = in neutral)

    Servo servo1;                                   // Setting up the Servo Object
    servo1.attach(D10);                             // Setting the motor signal pin to output

    //servo1.writeMicroseconds(neutral_pwm);  //Should be neutral_pwm

    for (;;)
    {
        if(servo1state == 0)
        {
            // On startup, move the servo into the neutral position
            servo1.writeMicroseconds(neutral_pwm);  //Should be neutral_pwm
            //Serial << "Servo 1 State = 0" << endl;
            // Check if the servo has actually moved into position
            Button_State_LS1N.get(ls1n);
            if(ls1n == 1)
            {
                servo1state = 1;
            }

        }
        else if(servo1state == 1) //Servo 1 hub state checking to see where it needs to move to
        {
            servo1pos.get(servo1position);

            // Hub state
            if(servo1position == 1)
            {
                // Move into neutral next time through
                servo1state = 2;
            }
            else if (servo1position == 2)
            {
                // Move into first gear next time through
                servo1state = 3;
            }
            else
            {
                //nothing
            }

        }
        else if (servo1state == 2)
        {
            // move into neutral
            servo1.writeMicroseconds(neutral_pwm);
            //Serial << "Servo 1 State = 2" << endl;
            servo1state = 1;
        }
        else if (servo1state == 3)
        {
            // move into second gear
            servo1.writeMicroseconds(first_pwm);
            //Serial << "Servo 1 State = 3" << endl;
            servo1state = 1;
        }
        else
        {
            //nothing
```

```
        }

        vTaskDelay(50);
    }
}

/** @brief   Task which sends a PWM signal to servo 2.
 *  @details This task sends PWM signals to servo 2. The values for the PWM signals are determined
 *           based on where the servo needs to be in order to make the correct shift.
 *  @param   p_params        A pointer to function parameters which we don't use.
 *  @param   servo2state     State variable that determines what state servo 1 task should be in.
 *  @param   neutral_pwm     PWM value for moving servo into neutral position
 *  @param   second_pwm      PWM value for moving servo into first gear position
 *  @param   servo2position  Super boolean for position. 1 = neutral, 2 = first gear
 */
void task_servo2 (void* p_params)
{
    (void)p_params;                                 // Shuts up a compiler warning

    // Tested PWM Values for landing in 1st and Neutral without crushing switches
    uint8_t  servo2state = 0;                        // State variable for servo 2 task
    uint16_t neutral_pwm = 1000;                     // PWM value for moving servo into neutral position
    uint16_t second_pwm  = 2000;                     // PWM value for moving servo into second gear position

    uint8_t  servo2position = 0;                      // Super boolean for position. 1 = neutral, 2 = second gear
    // Putting this zero here is only for startup ***Testing this should probably be here in the final itteration
    //servo2pos.put(servo2position);

    uint8_t ls2n;                                       // Flag for the neutral limit switch (0 = not in neutral, 1 = in neutral)

    Servo servo2;                                    // Setting up the Servo Object
    servo2.attach(D11);                              // Setting the motor signal pin to output


    for (;;)
    {
        if(servo2state == 0)
        {
            // On startup, move the servo into the neutral position
            servo2.writeMicroseconds(neutral_pwm);

            // Check if the servo has actually moved into position
            Button_State_LS2N.get(ls2n);
            if(ls2n == 1)
            {
                servo2state = 1;
            }

        }
        else if(servo2state == 1)
        {
            servo2pos.get(servo2position);

            // Hub state
            if(servo2position == 1)
            {
                // Move into neutral next time through
                servo2state = 2;
            }
            else if (servo2position == 2)
            {
                // Move into Second gear next time through
                servo2state = 3;
            }
            else
            {
                //nothing
            }

        }
        else if (servo2state == 2)
        {
            // move into neutral
            servo2.writeMicroseconds(neutral_pwm);

            servo2state = 1;
        }
        else if (servo2state == 3)
        {
            // move into second gear
            servo2.writeMicroseconds(second_pwm);

            servo2state = 1;
        }
        else
```

```
        {
            //nothing
        }

        vTaskDelay(50);
    }
}

/** @brief   Task which reads analog signals from a footpedal.
 *  @details This task runs quickly and reads the analog input from a footpedal.
 *           The value read from the footpedal is put into a share and then used by
 *           the motor task to set the PWM value for the motor.
 *  @param   p_params                 A pointer to function parameters which we don't use.
 *  @param   pedal_input              Variable used to hold the analog input signal from the pedal.
 *  @param   Max_Pedal                The maximum input value that can be read from the pedal.
 *  @param   PWM_Max                  Maximum PWM value that is used to calculate the input from the pedal as a percentage.
 *  @param   PWM_Min                  Minimum PWM value that is used to calculate the input from the pedal as a percentage.
 *  @param   Pedal_Override_Check     Flag which checks if the pedal input should be temporarily ignored.
 */
void task_pedal (void* p_params)
{
    (void)p_params;                            // Shuts up a compiler warning
    uint16_t Max_Pedal = 805;                 // Max Analog Read value for test is 1022 ***TESTING should change for real pedal
    uint16_t Pedal_Range = 669;              //delta between the analog readings of 0 & 100 percent throttle
    float pedal_input  = 1150;                  // Pedal Value TESTING this should be 0 in final itteration
    uint16_t PWM_Max   = 1940;                 //***TESTING this variable might be one that can be changed based on mode in final version
    uint16_t PWM_Min   = 1100;                // ***Testing this should be 0 for the real pedal (or something close to it)
    //uint16_t PWM_Offset = 510;              // ***TESTING This might not be something that gets used at all
    int8_t Pedal_Override_Check = 0;
    //PEDAL_OVERRIDE.put(0);                    // moved to mastermind
    PEDAL_OVERRIDE.put(1);                   // moved to mastermind or shift task

    pinMode(A1,INPUT);                         // Setting the pedal pin to an input for ADC

    for (;;)
    {
        PEDAL_OVERRIDE.get(Pedal_Override_Check);
        if(Pedal_Override_Check == 0)             //If overide is cleared (meaning use the pedal as intended)
        {
            pedal_input = analogRead(A1);       // Read the analog signal from the pin the pedal is connected to.
            //Filter for PWM Signal
            pedal_input = abs(pedal_input - Max_Pedal);
            //Serial << "Pedal sig after abs = "<< pedal_input << endl;
            pedal_input = pedal_input/Pedal_Range;
            //Serial << "Pedal sig percentage = "<< pedal_input << endl;
            pedal_input = pedal_input*(PWM_Max - PWM_Min) + PWM_Min;
            //Serial << "Filtered Signal =" << pedal_input << endl;
            PWM_VAL.put((int)pedal_input);                            // Putting the analog input from the footpedal into the share


            //Testing uncomment the stuff above once done with the rpm signal
            //pedal_input = pedal_input + 20;
            //PWM_VAL.put(pedal_input);              //***Testing this is only to brute force our way through the start up sequence for rpm
verification

            /* pedal_input = analogRead(A1);       // Read the analog signal from the pin the pedal is connected to.
            Serial << "Pedal Analog Read Value = "<< pedal_input << endl;
            //Filter for PWM Signal
            pedal_input = pedal_input/Max_Pedal;                      //Converting to a percent value
            pedal_input = pedal_input*(PWM_Max - PWM_Min) + PWM_Min;
            Serial << "Filtered Signal =" << pedal_input << endl;
            PWM_VAL.put((int)pedal_input);                            // Putting the analog input from the footpedal into the share */
        }
        else
        {
            /* Pedal is overriden, don't do anything and exit */
        }

        vTaskDelay(500);        //TESTING this used to be ~250 for user values & may need to be even faster in final itteration
    }
}

/** @brief   Task which displays information.
 *  @details This task displays all the important information to the user on the LCD screen.
 *  @param   p_params       A pointer to function parameters which we don't use.
 *  @param   LCD_State      Variable used to determine the state of the LCD_PRINT task.
 *  @param   RPM_P          Variable used to print the RPM gauge to the screen.
 *  @param   MODE_P         Variable that is set depending on the mode of the system (armed or unarmed)
 *  @param   GEAR_P         Variable that is set depending on what gear the system is in.
 *  @param   SHIFT_TIME_P   Variable for displaying the shift time from the previous shift.
 *  @param   MESSAGE_P      Variable for displaying messages.
 */
void LCD_PRINT (void* p_params)
{
    (void)p_params;            // Does nothing but shut up a compiler warning
```

```
// State Variables
uint8_t LCD_State = 0;              // LCD State Variable for FSM
int8_t RPM_P = 0;                   // Variable for printing the rpm gauge
int8_t MODE_P = 0;                  // Initialized to armed mode = 0, 1 = low, 2 = high
int8_t GEAR_P = 0;                  // Initialized to Neautral?
float SHIFT_TIME_P = 0;             // Initialized to 0
int8_t MESSAGE_P = 0;               // Initialized Message Value
int16_t RPM_TEMP = 0;                // ***TESTING ONLY
float TEMP_2 =0;                    // ***TESTING ONLY?

// Local Variables:
uint16_t RPM_in = 0;                // local variable for the input rpm value
uint16_t RPM_max = 4500;            // local variable for the max input rpm
uint16_t RPM_min = 0;               // local variable for the minimum input rpm


// I2C SDA & SCL pin Reconfiguration
Wire.begin(A4,A5);                  // LCD Screen SDA = A4, SCL = A5

// LCD Screen Specifications:
uint16_t lcd_addr = 0x27;           // LCD I2C Address (Some are 0x3F)
uint8_t col = 20;                   // LCD # of Collumns
uint8_t row = 4;                    // LCD # of Rows

// LCD Object Declaration
LiquidCrystal_I2C lcd(lcd_addr,col,row);   // set the LCD address to 0x27 for a 16 chars and 2 line display
lcd.init();  //initialize the lcd
lcd.setBacklight(1);  //open the backlight


//Initialization Screen: ***TESTING Not needed in final
lcd.setCursor ( 0, 0 );            // go to the top left corner
lcd.print("    Initializing    "); // write this string on the top row
lcd.setCursor ( 0, 1 );            // go to the 2nd row
lcd.print("     Be Patient     "); // pad string with spaces for centering
lcd.setCursor ( 0, 2 );            // go to the third row
lcd.print("                    "); // pad with spaces for centering
lcd.setCursor ( 0, 3 );            // go to the fourth row
lcd.print("                    ");

//Testing LCD prints everything to screen correctly:
//MODE.put(2);                        //System is in High speed mode
//GEAR_STATE.put(2);                  //Transmission is in 2nd gear
//SHIFT_TIME.put(.90);                //Most recent shift time

// The task's infinite loop goes here
for (;;)
{
    if(LCD_State == 0) // Home Screen
    {
        //For Testing w/out actual initializing delay ***TESTING
        delay(1000);

        // Home Screen Set:
        lcd.setCursor ( 0, 0 );            // go to the top left corner
        lcd.print("MODE:    |SPEED|GEAR"); // Writing the Top Row Lables: Mode, Speed, Gear
        lcd.setCursor ( 0, 1 );            // go to the 2nd row
        lcd.print("ST:     s|    s|    "); // Writing the Second Row: Shift time
        lcd.setCursor ( 0, 2 );            // go to the third row
        lcd.print("MSG:                "); // Error Space
        lcd.setCursor ( 0, 3 );            // go to the fourth row
        lcd.print("RPM:                "); // RPM Gauge w/ 16 pixels for rpm



        LCD_State = 2;                     // Should be a 1 ***TESTING

    }
    else if(LCD_State == 1) //Check Flags
    {
        //stuff more
        //Check flags for update requests & branch accordingly


    }
    else if(LCD_State == 2) //RPM Update
    {
        lcd.setCursor ( 0, 3 );                             // go to the fourth row
        lcd.print("RPM:                ");                  // RPM Gauge w/ 16 pixels for rpm

        revpmin_in.get(RPM_in);                             //Getting the current RPM variable & putting it into the local variable
        TEMP_2 = RPM_in;                                  //Type casting RPM in to a float
        //Serial << "RPM_in =" << RPM_in << endl;
        TEMP_2 = (TEMP_2-RPM_min)/(RPM_max-RPM_min);        // convertin rpm in to a percentage
```

```
            //Serial << "Temp_2 = " << TEMP_2 << endl;
            /* //***TESTING ONLY
            PWM_VAL.get(RPM_TEMP);
            TEMP_2 = RPM_TEMP;
            TEMP_2 = (TEMP_2 - 1000)/1000; */

/*              RPM_TEMP = (float)RPM_TEMP;
            RPM_TEMP = (RPM_TEMP-1000);
            TEMP_2 = RPM_TEMP/1000;   // HERE's the problem for some reason?? */
            //Serial.println(TEMP_2);
            RPM_P = TEMP_2*16;
            //RPM_P = int(RPM_P); //This should be here in normal code
            RPM_P = 14;             //Corresponds to ~8000rpm out in 2nd gear
            //Serial << "RPM_P =" << RPM_P << endl;

            //RPM_P = int(RPM_in);
            //Serial.println(RPM_P);
            lcd.setCursor ( 4, 3 );

/*              //RPM Print Loop:
            lcd.setCursor ( 4, 3 );             // Positioning the cursor for RPM Printing
            RPM_DISP.get(RPM_P);                // Getting the RPM Share for print loop */

            for(int i=0 ; i<RPM_P ; i++)
            {
                lcd.print(char(255));           // LCD Printing Block
            }

            //LCD_State = 1;                    //***TESTING this should be uncommented in normal use
        }
        else if(LCD_State == 3) //RPM Update Blink?
        {
            //
        }
        else if(LCD_State == 4) //Mode Update
        {
            MODE.get(MODE_P);                   // Get's the Current Mode & Puts it into the Display Mode_P Variable
            lcd.setCursor(5,0);                 // Set's the cursor to the correct position
            if(MODE_P == 0)                     // Armed State (Waiting for )
            {
                lcd.print("ARMD");               // Armed state
            }
            else if(MODE_P == 1)
            {
                lcd.print("LOW ");
            }
            else if(MODE_P == 2)
            {
                lcd.print("HIGH");
            }
            else
            {
                //stuff
            }
            LCD_State = 1;                      // Waiting State
        }
        else if(LCD_State == 5) // GEAR
        {
            //Setting up the Gear Variable:
            GEAR_STATE.get(GEAR_P);             //Getting the Gear State value and putting it into the Gear print variable
            lcd.setCursor(17,1);                //Setting the cursor at the correct position (for single line print)
            if(GEAR_P == 0)
            {
                lcd.print("N");                 //print neutral
            }
            else if(GEAR_P == 1)                //Printing First Gear Shifter Position
            {
                lcd.print(GEAR_P);              //Print 1st
            }
            else if(GEAR_P == 2)                //Printing Second Gear Shifter Postion
            {
                lcd.print(GEAR_P);              //Print 2nd
            }
            else if(GEAR_P == 3)                //Printing Third Gear Shifter Postion
            {
                lcd.print(GEAR_P);              //Print 3rd
            }
            else if(GEAR_P == 4)                //Printing Fourth Gear Shifter Postion
            {
                lcd.print(GEAR_P);              //Print 4th
            }
            else
            {
                lcd.print("WTF");               // printing Invalid Gear State
            }
```

```
            LCD_State = 1;                      // Setting the LCD_State back to waiting state
        }
        else if (LCD_State == 6) //Shift time update
        {
            //Print the update shift time
            SHIFT_TIME.get(SHIFT_TIME_P);       // Getting the Share value of Shift time and putting it into the print variable
            lcd.setCursor(3,1);                 // Setting hte LCD position
            lcd.print(SHIFT_TIME_P);            // Automatically truncates & Rounds to two sigs after the decimal (Probably good enough)
            LCD_State = 1;                      // Setting the LCD state back to the waiting state
        }
        else if (LCD_State == 7)
        {   //lcd.print("MSG:                       "); // Stock Space for reset
            //Printing error messages
            MESSAGE.get(MESSAGE_P);
            lcd.setCursor(0,2);                         // Setting the cursor at row three to print message

            if(MESSAGE_P == 0)
            {
                lcd.print("DriveLikeYouStoleIt!");      // Default Mesasge
            }
            else if (MESSAGE_P == 1)                    // Clear Message
            {
                lcd.print("MSG:                ");      // Error Message
            }
            else if (MESSAGE_P == 2)                    // Duplicate Error states as needed
            {
                lcd.print("MSG: WTF Happened? ");       // Error Essage
            }
            LCD_State = 1;
        }
        else
        {
            /* code */
        }

        vTaskDelay (200);                                           // RTOS delay so the task runs properly

    }
}

/** @brief   Task which calculates RPM for use by other tasks.
 *  @details This task takes the Hall Effect sensor data read in the ISR and converts it into usable
             RPM values. The RPM data is then stored into a separate queue to be used by other tasks such as the
             the shift task or the display.
 *  @param   p_params       A pointer to function parameters which we don't use.
 *  @param   pins           Variable that holds the number of pins on the sensor wheel
 *  @param   RPM            Variable that holds the current value of the output/input speed in RPM
 *  @param   time           Variable that holds the value of the time difference between pulses in seconds.
 *  @param   radpersec_new  Variable that holds the current value of the speed in radians per second.
 *  @param   radpersec_old  Variable that holds the previous value of the speed in radians per second.
 *  @param   acceleration   Variable that holds the current value of the acceleration in radians per second per second.
 *  @param   delta          Variable that contains the value of the time delta between sensor pulses in clock ticks.
 */
void task_rpmupdate (void* p_params)
{
    (void)p_params;                     // Shuts up a compiler warning

    uint8_t pins = 10;                  // Number of pins on the wheel
    float Prescaler = TMR1_Prescaler;            // Number used to modify the timer for the rpm sensing
    float ClockSpeed = 80000000;        // Clock speed of the microcontroller

    float RPM_in = 0;                      // Speed in revolutions per minute
    float RPM_in_old = 0;                   // old rpm input value for zero saturation block
    float time_in = 0;                     // Time between pulses in seconds
    uint8_t Input_counter = 0;              // Counter for incrementing 0 rpm threshold

    float radpersec_new_in = 0;            // Current speed in radians per second
    float radpersec_old_in = 0;            // Previous speed in radians per second

    float acceleration_in = 0;

    int16_t delta_in = 0;                      // Variable containing the delta between sensor pulses in clock ticks

    float RPM_out = 0;                      // Speed in revolutions per minute

    float time_out = 0;                     // Time between pulses in seconds

    float radpersec_new_out = 0;            // Current speed in radians per second
    float radpersec_old_out = 0;            // Previous speed in radians per second

    float acceleration_out = 0;

    int16_t delta_out = 0;                      // Variable containing the delta between sensor pulses in clock ticks

    int16_t PWM = 0;                        //TESTING dummy variable for getting the PWM value
```

```
    int8_t print_timer = 0;                 //TESTING print timer for the rpm to serial

    ticks_delta_in.put(10000);              //Testing??
    ticks_delta_out.put(10000);

    for (;;)
    {


        //INPUT RPM Calculations --------------------------------------------
        // Convert from the pulses to RPM
        ticks_delta_in.get(delta_in);
        delta_in = (float)delta_in;
        time_in = delta_in*(Prescaler/ClockSpeed);                  // In final version, use variables for prescaler / clock speed //TESTING
was .00015

        // Multiply time between pulses by the number of pins to get the time for one full revolution
        // Divide 60 by this number to get RPM
        RPM_in = 60/(pins*time_in);

        //Adding a zero rpm saturation block:
        if(RPM_in < 250)        //checking to see if the rpm is below the minimum driving condition
        {
            RPM_in = 0;         //Saturate to 0

        }
        else if(RPM_in > 4500)
        {
            RPM_in = 4500;      //Setting the input rpm to max out at 4500
        }
        else
        {
            //Code
        }


        //Serial << "Input Signal is" <<
        radpersec_new_in = RPM_in*2*PI*60;                        // Convert to radians per second
        acceleration_in = (radpersec_new_in - radpersec_old_in)/time_in; // Calculate the acceleration
        radpersec_old_in = radpersec_new_in;                      // Store the new radpersec value as the old one

        // Put the radpersec, RPM, and acceleration values into shares for use by other things
        radpsec_in.put(radpersec_new_in);
        revpmin_in.put(RPM_in);

        //Serial << "Acceleration is " << acceleration_in << endl;
        accel_in.put(acceleration_in);

        //OUTPUT RPM Calculations --------------------------------------------

        // Convert from the pulses to RPM
        ticks_delta_out.get(delta_out);
        delta_out = (float)delta_out;
        time_out = delta_out*(Prescaler/ClockSpeed);                  // In final version, use variables for prescaler / clock speed
//TESTING was .00015

        //Serial << "delta_out =" << delta_out << " s" << endl;

        // Multiply time between pulses by the number of pins to get the time for one full revolution
        // Divide 60 by this number to get RPM
        RPM_out = 60/(pins*time_out);

        if(RPM_out < 250)        //checking to see if the rpm is below the minimum driving condition
        {
            RPM_out = 0;         //Saturate to 0
        }
        else if(RPM_out > 13000)
        {
            RPM_out = 13000;     //Saturating to 15000
        }
        else
        {
            //code
        }

        //Serial << "Output RPM is " << RPM_out << endl;
        radpersec_new_out = RPM_out*2*PI*60;                        // Convert to radians per second
        acceleration_out = (radpersec_new_out - radpersec_old_out)/time_out; // Calculate the acceleration
        radpersec_old_out = radpersec_new_out;                      // Store the new radpersec value as the old one

        // Put the radpersec, RPM, and acceleration values into shares for use by other things
        radpsec_out.put(radpersec_new_out);
        revpmin_out.put(RPM_out);

        //Serial << "Acceleration is " << acceleration << endl;
```

```
        accel_out.put(acceleration_out);

        if(print_timer >= 25)
        {
            //Serial << "Input RPM = " << RPM_in << endl;
            //Serial << "Time In =" << time_in << endl;
            //Serial << "Output RPM = " << RPM_out << endl;
            //Serial << "delta_out =" << delta_out << endl;
            //Serial << "Working?" << endl;
            print_timer = 0;
        }
        else
        {
            print_timer++;
        }

        vTaskDelay(10);
    }
}

/** @brief   Task which runs through the shifting process.
 *  @details This task will sit in a waiting state until a shift request is made by the user.
 *           A shift request will cause the task to run through the shifting process and then return to the
 *           waiting state. This task functions differently depending on whether an up-shift or down-shift is requested.
 *  @param   p_params        A pointer to function parameters which we don't use.
 *  @param   shiftreq        Flag that is raised when a shift is requested.
 *  @param   acceleration    Variable that holds the current value of the acceleration in radians per second per second.
 *  @param   shift_state     Variable that determines the current state of the shift task.
 */
void task_shift (void* p_params)
{
    (void)p_params;                         // Shuts up a compiler warning

    int8_t shiftreq;                          // Shift request flag

    uint8_t shift_state = 0;                // Move the task into the waiting state
    uint8_t Button_state = 0;
    uint8_t ESC_Armed = 0;                  //esc armed flag variable
    uint8_t First = 0;                      //Dummy variable for first time entering the loop
    uint8_t Cycle = 0;                      //Dummy variable for timing the cycles on start up to find first gear

    //Variables for calculating shift time
    float Prescaler = TMR2_Prescaler;             // Number used to modify the timer for the rpm sensing TESTING used to be 12000 to improve
res.
    float ClockSpeed = 80000000;        // Clock speed of the microcontroller
    float shift_time = 0;               // The time in seconds of the shift time
    int16_t delta_shift_ticks = 0;      // The time in clock ticks of the shift time
    uint16_t shift_ticks_cur = 0;       // Variable for holding the current timer value
    uint16_t shift_ticks_old = 0;       //  Variable for holding the initial timer value


    //Load & unload dogs variables for the motor control:
    float acceleration;                     // Acceleration variable
    int16_t current_PWM;                    // local variable for current pwm value

    //Shifting Variables for the Servos
    uint8_t Servo_Neutral = 1;              // local variable for neutral position for servo
    uint8_t Servo_First = 2;                // local variable for First gear for servo position
    uint8_t Servo_Second = 2;               // local variable for Second gear for servo position
    int8_t  Gearstate;                      // local variable for the current gear state
    uint8_t timer_N = 0;                        // timer variable for checking to see how long it's taking to hit neutral
    uint8_t timer_G = 0;                        // timer variable for checking how long it's taking to get into gear
    uint8_t dummy =0;                       //TESTING this also should be here in final

    //Rev matching variables:
    //TESTING: These values need to be confirmed through the CAD/prior documentation!!!
    float First_Gear_Ratio = .75;         //First gear ratio which is output/input Should be :.9583??
    //Testing first gear ratio from rpm ratios is ~.75-.76, might change at some point and see what happens
    float Second_Gear_Ratio = 2.3;        //Second Gear ratio which is output/input (Testing: Theoretical value should be 2.833?)
    uint16_t rpm_in;                        //Local variable for input rpm
    uint16_t rpm_out;                       //Local variable for the output rpm
    float rpm_error;                        //Local variable for the error between desired and current rpm
    float rpm_target;                       //Local variable for the target rpm value
    float kp = .005;                        //Local variable for the kp value to get a good rpm match
    uint16_t PWM_map;                       //Local variable for the pwm proportion from the map value
    uint16_t PWM_kp;                        //Local variable for the PWM from the kp constant based on error
    uint16_t PWM_total;                     //Local variable for the total PWM to send to the motor for rev match control
    uint16_t RPM_Delta_Threshold_Up = 100;  //Shift variable for how close the rev match needs to be to try and complete the shift
    uint16_t RPM_Delta_Threshold_Down = 100;//Shift variable for how close the rev match needs to be to try and complete the shift
    uint16_t PWM_max = 1700;                //Max pwm value for the esc
    uint16_t PWM_min = 1100;                //Min PWM value for the esc

    //Shift time calculation variables:

    for (;;)
```

```
{
    if(shift_state == 0)
    {
        //Initializing and trying to put the transmission into 1st gear

        ESC_Armed_FLag.get(ESC_Armed);                      //Wait for the motor ESC to arm
        if(ESC_Armed == 1 && First == 0)              //Wait for the arm button to be pressed
        {
            PWM_VAL.put(1250);              //set the input motor to rotate slowly
            servo1pos.put(2);               //tell servo 1 to move into 1st gear
            First = 1;                      //So you're not constantly resetting the pwm value and servo1 position flag
        }
        else if(First == 1)
        {

            if(Cycle == 20)          //check if enough cycles have passed to update pwm
            {
                PWM_VAL.put(1150);              //set the input motor to rotate slowly
                servo1pos.put(2);               //tell servo 1 to move into 1st gear
            }
            else if(Cycle == 40)
            {
                PWM_VAL.put(1250);              //Set the input motor to rotate faster
                servo1pos.put(1);               //tell servo 1 to move into neutral
                Cycle = 0;                      //resetting the cycle counter
            }
            Cycle ++;                           //increment cycles

            Button_State_LS1.get(Button_state);
            if(Button_state == 1)       //look for the first gear button to be pressed
            {
                servo1pos.put(2);               //tell servo 1 to move into 1st gear
                PWM_VAL.put(1100);              //turn off the input motor
                PEDAL_OVERRIDE.put(0);          //turning the pedal back on for the user
                shift_state = 1;                //move shift state to 1
            }

        }

    }
    // State 1 waiting state
    else if(shift_state == 1)
    {

        shift_req.get(shiftreq);      // Get the contents of the shift_req share and put into shiftreq variable

        //revpmin_in.get(rpm_in); //use this in the if statement for a testing auto mode (rpm_in > 3000) && (dummy ==0)

        // If the user has requested a shift
        if(shiftreq != 0) //used to be this: shiftreq != 0 for manual mode
        {
            //dummy = 1;                      //TESTNG!! THis shouldn't be here in manual mode
            shift_req.put(2);               //TESTING!!! This shouldn't be here in the manual mode
            shift_state = 2;                // Set the task to the next state
            PEDAL_OVERRIDE.put(1);          // Making sure the user can't screw up the shift task, clear at end of shift
            //Shift_Busy_Flag.put(1);       // Set the Shifting task to busy so we don't take on a second request
            Serial << "Shift Request Acknowledged by Shift Task"<<endl;

            // To track shift time, begin by looking a the current timer value
            shift_ticks_cur = Timer2->getCount();
        }
        else
        {
            // do nothing
        }
    }
    // State 2 unload dogs
    else if(shift_state == 2)
    {
        // Check the status of the output

        accel_in.get(acceleration);             //Getting the current acceleration for the input
        PWM_VAL.get(current_PWM);               //Getting the current pwm value for the motor
        // Serial << "Acceleration is " << acceleration << endl;

        if(acceleration >= 0)
        {   Serial << "Acceleration is positive or 0 when unloading" << endl;
            current_PWM = current_PWM*.95;              // lower the speed by a percentage of original pwm value
            if(current_PWM > PWM_max)                   //Set saturation to ensure output PWM is within limits
            {
                current_PWM = PWM_max;                  //Setting the motors PWM value to the maximum allowable
            }
            else if(current_PWM < PWM_min)
            {
                current_PWM = PWM_min;                  //Setting the motor's PWM value to the minimum allowable
```

```
            }
            else
            {
                PWM_VAL.put(current_PWM);               // Set the PWM value to the motor
            }
            shift_state = 3;                            // Branch to next state for moving servo
        }

        if(acceleration < 0)
        {   Serial << "Acceleration is Negative when unloading" << endl;
            current_PWM = current_PWM*.95;              // Increase the speed by a percentage of original pwm value
            if(current_PWM > PWM_max)                   //Set saturation to ensure output PWM is within limits
            {
                current_PWM = PWM_max;                  //Setting the motors PWM value to the maximum allowable
            }
            else if(current_PWM < PWM_min)
            {
                current_PWM = PWM_min;                  //Setting the motor's PWM value to the minimum allowable
            }
            else
            {
                PWM_VAL.put(current_PWM);               // Set the PWM value to the motor
            }
            shift_state = 3;                            // Branch to next state for moving servo
        }

    }
    // State 3 Move to Neutral
    else if(shift_state == 3)
    {
        servo1pos.put(Servo_Neutral);           //Set Servo 1 into neutral position
        servo2pos.put(Servo_Neutral);           //Set Servo 2 into neutral position
        GEAR_STATE.get(Gearstate);              //Getting the local variable for the gear state

        if(Gearstate == 0)                       //Check the share variable for gear state to be in neutral
        {
            Serial << "Reached Neutral"<<endl;
            shift_state = 4;                    // to move to next state
            timer_N = 0;                        //Reseting the timer for the next time through
        }
        else if(timer_N >= 20)                   //This value may need to be changed over time but it shouldn't be needed if unload works
well
        {
            Serial << "Timed out trying to get to neutral" <<endl;
            shift_state = 2;                    //return to the unload of the dogs
        }
        else
        {
            timer_N++;                          //increment the timer
        }

    }
    // State 4 RPM Match
    else if(shift_state == 4)
    {
        //Serial << "hi" << endl;

        shift_req.get(shiftreq);
        if(shiftreq == 1) // Request to shift into 1st gear
        {
            //Serial << "Did we break it" << endl;

            revpmin_in.get(rpm_in);                     //Current output rpm into local variable
            revpmin_out.get(rpm_out);                   //Current input rpm into local variable
            rpm_target = rpm_out/First_Gear_Ratio;      //Calculate rpm target based on gear ratio
            //Serial << "Target = " << rpm_target << endl;
            rpm_error = rpm_target - rpm_in;            //Calculate error in RPM based on target and current rpm in, - is below, + is
above
            //Serial << "RPMIN = " << rpm_in << " RPM ERR = " << rpm_error << endl;
            //Serial << "RPM ERROR = " << rpm_target << endl;
            if(abs(rpm_error) <= RPM_Delta_Threshold_Down)        //Check to see if error is small enough, if so set state to complete
shift
            {
                Serial << "Reached RPM Threshold" << endl;
                Serial << "RPMin = " << rpm_in << " RPMout = " << rpm_out << endl;
                shift_state = 5;                        //Set the next state to shift state = 5, complete the shift
            }
            else
            {
                PWM_total = 1600;                   //Testing value & shouldn't be here in the end
                PWM_map = .1159*rpm_target + 1136.3;    //Calculate map pwm based on target rpm value, see excel chart for graph
                PWM_kp = rpm_error*kp;                  //Calcuate proprotional PWM based on error and kp value
                PWM_total = PWM_map + PWM_kp;           //Sum error into output PWM
                //Saturation block for max and min PWM values
                if(PWM_total > PWM_max)                 //Set saturation to ensure output PWM is within limits
```

```
                {
                    PWM_total = PWM_max;                //Setting the motors PWM value to the maximum allowable
                }
                else if(PWM_total < PWM_min)
                {
                    PWM_total = PWM_min;                //Setting the motor's PWM value to the minimum allowable
                }
                PWM_VAL.put(PWM_total);                 //Setting the share variable for the PWM
                Serial << "PWM_Total = " << PWM_total << endl;
            }



        }

        else if(shiftreq == 2) //Request to shift into 2nd gear
        {
            revpmin_in.get(rpm_in);                     //Current output rpm into local variable
            revpmin_out.get(rpm_out);                   //Current input rpm into local variable
            rpm_target = rpm_out/Second_Gear_Ratio;     //Calculate rpm target based on gear ratio
            rpm_error = rpm_target - rpm_in;              //Calculate error in RPM based on target and current rpm in, - is below, + is 
above
            Serial << "Target = " << rpm_target << endl;
            Serial << "RPMIN = " << rpm_in << " RPM ERR = " << rpm_error << endl;
            if(abs(rpm_error) <= RPM_Delta_Threshold_Up)        //Check to see if error is small enough, if so set state to complete 
shift
            {
                Serial << "Reached RPM Threshold" << endl;
                Serial << "RPMin = " << rpm_in << " RPMout = " << rpm_out << endl;
                shift_state = 5;                        //Set the next state to shift state = 5, complete the shift
            }
            else
            {
                PWM_map = .1159*rpm_target + 1136.3;    //Calculate map pwm based on target rpm value, see excel chart for graph
                PWM_kp = rpm_error*kp;                  //Calcuate proprotional PWM based on error and kp value
                PWM_total = PWM_map + PWM_kp;           //Sum error into output PWM
                //Saturation block for max and min PWM values
                if(PWM_total > PWM_max)                 //Set saturation to ensure output PWM is within limits
                {
                    PWM_total = PWM_max;                //Setting the motors PWM value to the maximum allowable
                }
                else if(PWM_total < PWM_min)
                {
                    PWM_total = PWM_min;                //Setting the motor's PWM value to the minimum allowable
                }
                PWM_VAL.put(PWM_total);                 //Setting the share variable for the PWM
                Serial << "PWM_Total = " << PWM_total << endl;
            }
        }
        else
        {
            //code?
        }

        // Knowing the gear ratio desired and current output speed,
        // change input motor speed to desired value (have a tolerance band to allow easy mating for the dogs)

        //Serial << "Change the motor speed" << endl;
        //shift_state = 5; // to move to next state
    }
    // State 5 Complete the Shift
    else if(shift_state == 5)
    {
        shift_req.get(shiftreq);                //Check the requested shift direction
        GEAR_STATE.get(Gearstate);              //Getting the gear state to put into the local variable
        //Branch into the corresponding Servo state
        if(shiftreq == 1)                       //Request to move into first gear
        {
            if(Gearstate == 1)                  // The shift is complete and we are in first gear
            {
                Serial <<"Completed shift into 1st" <<endl;
                shift_state = 1;                // Sets the shift state back to a 1
                shift_req.put(0);               // Lower the shift request flag
                PEDAL_OVERRIDE.put(0);          // Gives back the user control over the pedal
                timer_G = 0;                    // Reset the timer for finding the next gear
                servo1pos.put(Servo_First);     //Set the first servo to move into position 1

                // Calculate the time it took to complete this shift
                shift_ticks_old = shift_ticks_cur;
                shift_ticks_cur =Timer2->getCount();

                // Convert from ticks to seconds
                delta_shift_ticks = shift_ticks_cur - shift_ticks_old;
                shift_time = delta_shift_ticks*(Prescaler/ClockSpeed);
                Serial << "Shift time into first was = " << shift_time << " s" << endl;
```

```
                }
                else if(timer_G >= 30)
                {
                    Serial << "Timed out trying to find first gear" <<endl;
                    timer_G = 0;                    // Reset the timer for finding the next gear
                    shift_state = 3;                // Reset the shift state back to finding neutral
                }
                else
                {
                    servo1pos.put(Servo_First);     //Set the first servo to move into position 1
                    timer_G++;                      //Increment the timer value
                }
            }
            else if(shiftreq == 2)                  //Request to move into Second gear
            {
                if(Gearstate == 2)
                {   Serial << "Completed the shift into second gear"<<endl;
                    shift_state = 1;                // Sets the shift state back to a 1
                    shift_req.put(0);               // Lower the shift request flag
                    PEDAL_OVERRIDE.put(0);          // Gives back the user control over the pedal
                    timer_G = 0;                    // Reset the timer for finding the next gear
                    servo2pos.put(Servo_Second);     //Set the first servo to move into position 1

                    // Calculate the time it took to complete this shift
                    shift_ticks_old = shift_ticks_cur;
                    shift_ticks_cur = Timer2->getCount();

                    // Convert from ticks to seconds
                    delta_shift_ticks = shift_ticks_cur - shift_ticks_old;
                    shift_time = delta_shift_ticks*(Prescaler/ClockSpeed);
                    Serial << "Shift time into second was = " << shift_time << " s" << endl;
                }
                else if(timer_G >= 20)              // The shift is complete and we are in first gear
                {
                    Serial << "Timed out trying to find second gear"<<endl;
                    timer_G = 0;                    // Reset the timer for finding the next gear
                    shift_state = 3;                // Reset the shift state back to finding neutral
                }
                else
                {
                    servo2pos.put(Servo_Second);     //Set the first servo to move into position 1
                    timer_G++;                      //Increment the timer value
                }
            }
        }
        else
        {
            //Code?
        }
        vTaskDelay(25); //might need to be faster?
    }
}

/** @brief   Task which reads digital signals from the buttons & limitswitches
 *  @details This task runs quickly and reads the digital signal from the buttons
 *           and limitswtiches to provide the appropriate flags for controlling
 *           when to shift as well as when the shift collars have reached the
 *           full throw
 *  @param   p_params            A pointer to function parameters which we don't use.
 *  @param   NC_Button           Constant for a normally closed button (i.e. triggers on a 1).
 *  @param   NO_Button           Constant for a normally open button (i.e. triggers on a 1).
 *  @param   Button_Task_State   Variable that determines the state of the button task.
 *  @param   Flag_temp           Flag that is raised when a button has been pressed. Lowered when the button is processed.
 *
 */
void task_button (void* p_params)
{
    (void)p_params;                                 // Shuts up a compiler warning
    //Universal Button Variables:
    uint8_t NC_Button = 1;                          // Normally closed button, triggers on 1
    uint8_t NO_Button = 0;                          // Normally open button, triggers on 0
    uint8_t Button_Task_State = 0;                  // Button Task State for intialization vs other states
    uint8_t Flag_temp = 0;                          // Flag variable to get and put the real share flag into for checking state

    //Creating the upshift button object:
    uint8_t Debouncer_Threshold_Upshift = 2;        // debouncer threshold for the upshift button
    uint8_t Pin_Upshift = D2;                       // Upshift pin: 0 = Not Pressed, 1 = Pressed
    uint8_t State_Upshift_OLD = 0;                  // Intitialized to 0, will be set in the init phase
    uint8_t State_Upshift_CUR = 0;                  // Intitialized to 0, will be set in the init phase
    Debouncer Upshift (Debouncer_Threshold_Upshift,Pin_Upshift,NC_Button); // Button Class Object

    //Creating the downshift button object:
    uint8_t Debouncer_Threshold_Downshift = 2;      // debouncer threshold for the Arm Signal button
    uint8_t Pin_Downshift = D3;                     // Down shift pin: 0 = Not Pressed, 1 = Pressed
    uint8_t State_Downshift_OLD = 0;                // Intitialized to 0, will be set in the init phase
```

```
uint8_t State_Downshift_CUR = 0;                // Intitialized to 0, will be set in the init phase
Debouncer Downshift (Debouncer_Threshold_Downshift,Pin_Downshift,NC_Button); // Button Class Object

//Creating the Arm Signal button object:
uint8_t Debouncer_Threshold_ArmSig = 2;         // debouncer threshold for the Arm Signal button
uint8_t Pin_ArmSig = D8;                        // Arm Signal pin: 0 = Not Pressed, 1 = Pressed
uint8_t State_ArmSig_OLD = 0;                   // Intitialized to 0, will be set in the init phase
uint8_t State_ArmSig_CUR = 0;                   // Intitialized to 0, will be set in the init phase
Debouncer ArmSig (Debouncer_Threshold_ArmSig,Pin_ArmSig,NC_Button); // Button Class Object

//Creating the Extra Signal button object:
uint8_t Debouncer_Threshold_Extra = 2;          // debouncer threshold for the Arm Signal button
uint8_t Pin_Extra = D13;                        // Extra Signal pin: 0 = Not Pressed, 1 = Pressed
uint8_t State_Extra_OLD = 0;                    // Intitialized to 0, will be set in the init phase
uint8_t State_Extra_CUR = 0;                    // Intitialized to 0, will be set in the init phase
Debouncer ExtraSig (Debouncer_Threshold_Extra,Pin_Extra,NC_Button); // Button Class Object

//Creating the 1st Gear Limit Switch Button Object:
uint8_t Debouncer_Threshold_LS1 = 2;            // debouncer threshold for the first limit switch
uint8_t Pin_LS1 = D7;                           // First Gear Limit Switch Pin
uint8_t State_LS1_OLD = 0;                      // Intitialized to 0, will be set in the init phase
uint8_t State_LS1_CUR = 0;                      // Intitialized to 0, will be set in the init phase
Debouncer LS1 (Debouncer_Threshold_LS1,Pin_LS1,NC_Button); // Button Class Object

//Creating the 1st Neutral Gear Limit Switch Button Object:
uint8_t Debouncer_Threshold_LS1N = 2;           // debouncer threshold for the first limit switch
uint8_t Pin_LS1N = D6;                          // First Gear Neutral limit Switch Pin
uint8_t State_LS1N_OLD = 0;                     // Intitialized to 0, will be set in the init phase
uint8_t State_LS1N_CUR = 0;                     // Intitialized to 0, will be set in the init phase
Debouncer LS1N (Debouncer_Threshold_LS1N,Pin_LS1N,NC_Button); // Button Class Object

//Creating the 2nd Gear Limit Switch Button Object:
uint8_t Debouncer_Threshold_LS2 = 2;            // debouncer threshold for the first limit switch
uint8_t Pin_LS2 = D5;                           // Second Gear Limit Switch Pin
uint8_t State_LS2_OLD = 0;                      // Intitialized to 0, will be set in the init phase
uint8_t State_LS2_CUR = 0;                      // Intitialized to 0, will be set in the init phase
Debouncer LS2 (Debouncer_Threshold_LS2,Pin_LS2,NC_Button); // Button Class Object

//Creating the 2nd Neutral Gear Limit Switch Button Object:
uint8_t Debouncer_Threshold_LS2N = 2;           // debouncer threshold for the first limit switch
uint8_t Pin_LS2N = D4;                          // Second Gear Neutral  Limit Switch Pin
uint8_t State_LS2N_OLD = 0;                     // Intitialized to 0, will be set in the init phase
uint8_t State_LS2N_CUR = 0;                     // Intitialized to 0, will be set in the init phase
Debouncer LS2N (Debouncer_Threshold_LS2N,Pin_LS2N,NC_Button); // Button Class Object

//Testing pinmode setup:
//pinMode(Pin_Upshift,INPUT);                 //Test setting up the upshift pin for an input

for (;;) //***TESTING: The global share button flag variable values should also be reordered based on priority
{
    if(Button_Task_State == 0)              // Initialization state, runs through & updates the share variable for each button
    {
        Upshift.debounce();                             // Running the debounce on the upshift button to check current state
        State_Upshift_CUR = Upshift.update();           // Getting the current upshift state variable
        Button_State_Upshift.put(State_Upshift_CUR);    // Putting the current upshift state into the share variable

        Downshift.debounce();
        State_Downshift_CUR = Downshift.update();
        Button_State_Downshift.put(State_Downshift_CUR);    //Setting the State to the current value of the button after debounce

        ExtraSig.debounce();
        State_Extra_CUR = ExtraSig.update();
        Button_State_ExtraSig.put(State_Extra_CUR);    //Setting the State to the current value of the button after debounce

        ArmSig.debounce();
        State_ArmSig_CUR = ArmSig.update();
        Button_State_ArmSig.put(State_ArmSig_CUR);    //Setting the State to the current value of the button after debounce

        LS1.debounce();
        State_LS1_CUR = LS1.update();
        Button_State_LS1.put(State_LS1_CUR);    //Setting the State to the current value of the button after debounce

        LS1N.debounce();
        State_LS1N_CUR = LS1N.update();
        Button_State_LS1N.put(State_LS1N_CUR);    //Setting the State to the current value of the button after debounce

        LS2.debounce();
        State_LS2_CUR = LS2.update();
        Button_State_LS2.put(State_LS2_CUR);    //Setting the State to the current value of the button after debounce

        LS2N.debounce();
        State_LS2N_CUR = LS2N.update();
        Button_State_LS2N.put(State_LS2N_CUR);    //Setting the State to the current value of the button after debounce
```

```
            Button_Task_State = 1;              //Setting the button task state to 1 so it will continuously update the button states
            //Serial << "Button Init State = " << State_Upshift_CUR << endl;
        }
        else if(Button_Task_State == 1)         // Continuous Button Mode, Might be worth adding a condition for start up here?
        {
            //Put a Key flag variable 0 = no buttons pressed, 1-Number corresponds to the specific button
            //Add a Key flag busy state
            // Up Shift Button update Code:
            State_Upshift_OLD = Upshift.update();
            Upshift.debounce();
            State_Upshift_CUR = Upshift.update();
            Button_State_Upshift.put(State_Upshift_CUR);    //Setting the State to the current value of the button after debounce
            Button_Flag.get(Flag_temp);
            if((State_Upshift_CUR > State_Upshift_OLD) && (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS, 
possibly impliment Que if not working
            {

                Button_Flag.put(1);                             //Setting the flag to show that the button has gone through a change
                Serial << "Upshift Button State =" << State_Upshift_CUR << endl;
                //Set the Global Button Flag Share to the Upshift button value
            }
            else
            {
                //Nothing?
            }
            // Down Shift Button update Code:
            State_Downshift_OLD = Downshift.update();
            Downshift.debounce();
            State_Downshift_CUR = Downshift.update();
            Button_State_Downshift.put(State_Downshift_CUR);    //Setting the State to the current value of the button after debounce
            Button_Flag.get(Flag_temp);
            if((State_Downshift_CUR > State_Downshift_OLD) && (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS, 
possibly impliment Que if not working
            {

                Button_Flag.put(2);                     //Setting the flag to show that the button has gone through a change
                Serial << "DownShift Button State =" << State_Downshift_CUR << endl; //Testing** This can be removed in final
                //Set the Global Button Flag Share to the Upshift button value
            }
            else
            {
                //Nothing?
            }
            // Extra Shift Button update Code:
            State_Extra_OLD = ExtraSig.update();
            ExtraSig.debounce();
            State_Extra_CUR = ExtraSig.update();
            Button_State_ExtraSig.put(State_Extra_CUR);    //Setting the State to the current value of the button after debounce
            Button_Flag.get(Flag_temp);
            if((State_Extra_CUR > State_Extra_OLD) && (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS, possibly 
impliment Que if not working
            {

                Button_Flag.put(3);                     //Setting the flag to show that the button has gone through a change
                Serial << "Extra Button State =" << State_Extra_CUR << endl; //Testing** This can be removed in final
                //Set the Global Button Flag Share to the Upshift button value
            }
            else
            {
                //Nothing?
            }
            // Arm Shift Button update Code:
            State_ArmSig_OLD = ArmSig.update();
            ArmSig.debounce();
            State_ArmSig_CUR = ArmSig.update();
            Button_State_ArmSig.put(State_ArmSig_CUR);    //Setting the State to the current value of the button after debounce
            Button_Flag.get(Flag_temp);
            if((State_ArmSig_CUR > State_ArmSig_OLD) && (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS, 
possibly impliment Que if not working
            {

                Button_Flag.put(4);                     //Setting the flag to show that the button has gone through a change
                Serial << "Arm Button State =" << State_ArmSig_CUR << endl; //Testing** This can be removed in final
                //Set the Global Button Flag Share to the Upshift button value
            }
            else
            {
                //Nothing?
            }
            // 1st Gear Limit Switch Button update Code:
            State_LS1_OLD = LS1.update();
            LS1.debounce();
            State_LS1_CUR = LS1.update();
            Button_State_LS1.put(State_LS1_CUR);    //Setting the State to the current value of the button after debounce
            //Button_Flag.get(Flag_temp);
```

```
            /* if((State_LS1_CUR != State_LS1_OLD) & (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS, possibly
impliment Que if not working
            {

                Button_Flag.put(5);                     //Setting the flag to show that the button has gone through a change
                Serial << "LS1 Button State =" << State_LS1_CUR << endl; //Testing** This can be removed in final
                //Set the Global Button Flag Share to the Upshift button value
            } */

            // 1st Gear Neautral Limit Switch Button update Code:
            State_LS1N_OLD = LS1N.update();
            LS1N.debounce();
            State_LS1N_CUR = LS1N.update();
            Button_State_LS1N.put(State_LS1N_CUR);    //Setting the State to the current value of the button after debounce
            //Button_Flag.get(Flag_temp);
            /* if((State_LS1N_CUR != State_LS1N_OLD) & (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS,
possibly impliment Que if not working
            {
                Button_Flag.put(5);                     //Setting the flag to show that the button has gone through a change
                Serial << "LS1N Button State =" << State_LS1N_CUR << endl; //Testing** This can be removed in final
                //Set the Global Button Flag Share to the Upshift button value
            } */

            // 2nd Gear Limit Switch Button update Code:
            State_LS2_OLD = LS2.update();
            LS2.debounce();
            State_LS2_CUR = LS2.update();
            Button_State_LS2.put(State_LS2_CUR);    //Setting the State to the current value of the button after debounce
            /* Button_Flag.get(Flag_temp);
            if((State_LS2_CUR != State_LS2_OLD) & (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS, possibly
impliment Que if not working
            {

                Button_Flag.put(6);                     //Setting the flag to show that the button has gone through a change
                Serial << "LS2 Button State =" << State_LS2_CUR << endl; //Testing** This can be removed in final
                //Set the Global Button Flag Share to the Upshift button value
            } */

            // 2nd Gear Neautral Limit Switch Button update Code:
            State_LS2N_OLD = LS2N.update();
            LS2N.debounce();
            State_LS2N_CUR = LS2N.update();
            Button_State_LS2N.put(State_LS2N_CUR);    //Setting the State to the current value of the button after debounce
            /* Button_Flag.get(Flag_temp);
            if((State_LS2N_CUR != State_LS2N_OLD) & (Flag_temp == 0)) // ***Testing: Make this rising edge, make it double for LS, possibly
impliment Que if not working
            {

                Button_Flag.put(7);                     //Setting the flag to show that the button has gone through a change
                Serial << "LS2N Button State =" << State_LS2N_CUR << endl; //Testing** This can be removed in final
                //Set the Global Button Flag Share to the Upshift button value
            } */
            if((State_LS1N_CUR == 1) && (State_LS2N_CUR == 1))
            {
                GEAR_STATE.put(0);          // Set the current gear to 0
            }
            if((State_LS1_CUR == 1) && (State_LS2N_CUR == 1))
            {
                GEAR_STATE.put(1);          // Set the current gear to 1
            }
            if((State_LS2_CUR == 1) && (State_LS1N_CUR == 1))
            {
                GEAR_STATE.put(2);          // Set the current gear to 2
            }
            else
            {
                //Nothing?
            }
            //***Testing: Repeat the above code for the remaining buttons?
        }
        else
        {
            //Nothing
        }

        vTaskDelay(50);

    }
}

/** @brief   Task which sends a PWM signal to servo 1.
 *  @details This task sends PWM signals to servo 1. The values for the PWM signals are determined
 *           based on where the servo needs to be in order to make the correct shift.
 *  @param   p_params        A pointer to function parameters which we don't use.
 *  @param   servo1state     State variable that determines what state servo 1 task should be in.
```

```
 *  @param   neutral_pwm     PWM value for moving servo into neutral position
 *  @param   first_pwm       PWM value for moving servo into first gear position
 *  @param   servo1position  Super boolean for position. 1 = neutral, 2 = first gear
 */
void task_timer (void* p_params)
{
    (void)p_params;                                        // Shuts up a compiler warning

    //First Timer Delta Calculation Variables:
    uint16_t current_ticks = 0;                            // Local variables for current tick value
    uint16_t old_ticks = 0;                                // Local variables for old tick value
    int16_t delta_ticks = 0;                              // Local variables for delta ticks value
    float Prescaler1 = TMR1_Prescaler;                                // Number used to modify the timer for the rpm sensing
    float ClockSpeed1 = 80000000;                           // Clock speed of the microcontroller
    float Timer1_time = 0;                                 // Local variables for time

    const TickType_t sim_period = 250;                     // RTOS ticks (ms) between runs
    TickType_t xLastWakeTime = xTaskGetTickCount();        // Gets the first tick count

    for (;;) //***TESTING: The global share button flag variable values should also be reordered based on priority
    {
        //First Timer delta calculations:
        current_ticks = Timer->getCount();                     // Get the current ticks count
        //current_ticks = TIM2->CNT;

        delta_ticks  = current_ticks - old_ticks;              // Calculate the delta
        old_ticks    = current_ticks;                          // Store the current tick value
        Timer1_time = delta_ticks*(Prescaler1/ClockSpeed1);    // Timer 1 time between delta values of the timer

        //Sample Code from 405 for handling the overflow issue
        /* #Overflow/Underflow logic
        if self.delta < -32767:
            self.delta = 65535 - self.pos_old + self.pos_cur
        elif self.delta >= 32767:
            self.delta = 65535 + self.pos_old - self.pos_cur */


        //Serial << "Delta_ticks = " << delta_ticks << endl;

        vTaskDelayUntil(&xLastWakeTime,sim_period);
    }
}

/** @brief   Arduino setup function which runs once at program startup.
 *  @details This function sets up a serial port for communication and creates
 *           the tasks which will be run.
 */
void setup ()
{
    // Start the serial port, wait a short time, then say hello. Use the
    // non-RTOS delay() function because the RTOS hasn't been started yet
    Serial.begin (115200);
    delay (2000);
    Serial << endl << endl << "Initializing G.H.O.S.T Software" << endl;
    digitalWrite(A10,HIGH);         //Enable Motor A
    digitalWrite(D4,LOW);         //Set Other Motor Lead Low
    //RPM_DISP.put(6);                //Setting the Share RPM_DISP ***TESTING use this for the RPM Computing Task to Update & set Flags
    //MODE.put(0);                    //Setting the Share Mode *** TEsting use this in the UI button setup & Read in most other tasks
    //GEAR_STATE.put(1);              //Setting the Share Gear State to Nuetral *** TESTING Used by servo limit switches
    //SHIFT_TIME.put(1.35651);        //Setting the Share Shift time Variable to something too Larger for truncation check *** TESTING should
be from shift time calculator Task
    //MESSAGE.put(2);                 //Setting the Share Message variable to test print ***TESTING These would be used by any tasks looking
to communicate with user
    //Serial << "Is the VS Code being a dumb pain in the butt?"<<endl;
    //servo1pos.put(2);               //for testing to put the servo into a given state
    //delay (2000);
    //servo1pos.put(1);

    // For all xTaskCreate, higher priority numbers run first, IE: 10 is going to run before 9, 9 before 8 & so on

    // Create a task which checks for shift requests (TESTING ONLY)
    xTaskCreate (task_button,
                 "button",
                 1024,                            // Stack size
                 NULL,
                 10,                               // Priority
                 NULL);

    // Create a mastermind task
    xTaskCreate (task_mm,
                 "mastermind",
                 1024,                            // Stack size
                 NULL,
                 9,                               // Priority
                 NULL);
```

```
// Create a task which prints a slightly disagreeable message
xTaskCreate (LCD_PRINT,
             "LCD Print",                     // Name for printouts
             1536,                            // Stack size
             NULL,                            // Parameters for task fn.
             1,                               // Priority
             NULL);                           // Task handle

// Create a task which reads the analog signal from the foot pedal
xTaskCreate (task_pedal,
             "pedal",
             1024,                            // Stack size
             NULL,
             3,                               // Priority
             NULL);

// Create a task which sends PWM signals to the motor
xTaskCreate (task_motor,
             "motor",
             1024,                            // Stack size
             NULL,
             8,                               // Priority
             NULL);

// Create a task which sends PWM signals to the motor
xTaskCreate (task_servo1,
             "servo 1",
             1024,                            // Stack size
             NULL,
             7,                               // Priority
             NULL);

    // Create a task which sends PWM signals to the motor
xTaskCreate (task_servo2,
             "servo 2",
             1024,                            // Stack size
             NULL,
             7,                               // Priority
             NULL);

// Create a task which reads the signals from the RPM sensors
xTaskCreate (task_rpmupdate,
             "rpmupdate",
             1024,                            // Stack size
             NULL,
             6,                               // Priority
             NULL);

    // Create a task which runs through the shifting process
xTaskCreate (task_shift,
             "shift",
             1024,                            // Stack size
             NULL,
             5,                               // Priority
             NULL);

xTaskCreate (task_timer,
             "timer",
             1024,                            // Stack size
             NULL,
             5,                               // Priority
             NULL);



// Hall Effect Sensor Interrupt Service Routine setup
uint8_t Hall1Pin = A2;             // Output hall sensor
uint8_t Hall2Pin = A3;             // Input hall sensor  pin should be:A3
pinMode(Hall1Pin, INPUT);
pinMode(Hall2Pin, INPUT_PULLUP);    // Changed from regular to pullup

attachInterrupt(digitalPinToInterrupt(Hall1Pin), SPEEDSENSORIN_ISR, RISING);
attachInterrupt(digitalPinToInterrupt(Hall2Pin), SPEEDSENSOROUT_ISR, RISING);

//Hardware Timer setup
Timer->setMode(1, TIMER_OUTPUT_COMPARE, D13);  // Set the mode of the timer
Timer->setPrescaleFactor(TMR1_Prescaler);                // See prescaler calc
Timer2->setMode(1, TIMER_OUTPUT_COMPARE);      // Set the mode of the timer
Timer2->setPrescaleFactor(TMR2_Prescaler);              // See prescaler calc

//Prescaller documentation for the interupts:
// http://docs.leaflabs.com/static.leaflabs.com/pub/leaflabs/maple-docs/0.0.9/lang/api/hardwaretimer.html#lang-hardwaretimer-modes
```

```
    Timer->resume();                                 // Start the delta pulse calc timer for input
    Timer2->resume();                                // Start the timer for the shift time calculations

    // If using an STM32, we need to call the scheduler startup function now;
    // if using an ESP32, it has already been called for us
    #if (defined STM32L4xx || defined STM32F4xx)
        vTaskStartScheduler ();
    #endif
}

/** @brief   Arduino's low-priority loop function, which we don't use.
 *  @details A non-RTOS Arduino program runs all of its continuously running
 *           code in this function after @c setup() has finished. When using
 *           FreeRTOS, @c loop() implements a low priority task on most
 *           microcontrollers, and crashes on some others, so we'll not use it.
 */
void loop ()
{
}
```

"Zack-janic"