Self-leveling Drink Holder

Final Conceptual Design Report

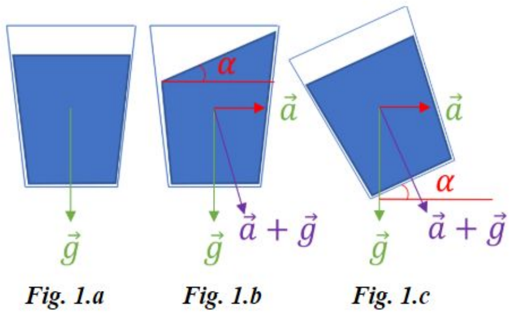
MIE438: Microcontrollers and Embedded Microprocessors

Group 5

Abdul Derh	1001301110
Rakshit Sharma	1001587641
Tongwha Kim	1002327305

1. Background

A common problem for drivers is liquid spillage during their commute. In order to prevent drink spillage from disruptive inertial forces, the self-levelling cup holder should counteract the inertial forces by adjusting the container's angular position. The degree of displacement is proportional to the amount of inertial force. As such, it is desirable to rotate the container, as shown in Figure 1 (a-c), on the right. To account for the problem's dynamicity, the self-levelling cup holder should be implemented with an embedded system to provide continuous control.



2. Original Design

This section details the original proposed design. Relative to the project scope, the design team was primarily considering the desired steady-state behavior, such that transient effects are not considered.

2.1. Mechanical

The original design was constructed for roll and pitch motion using the design shown in Figure 2 below. It was meant to be constructed using 3D printed parts. Table 1 summarizes the original selected components.

Table 1: *Mechanical Components for Driving Mechanism*

Component	Function/Operation	Selection
Position Control Motor	To provide driving forces for the pitch & roll axes respectively.	2 x Servo Motor MG996R [1]
Rotary Shaft	To support the radial load.	3D printed parts (PLA)
Structural (Brackets/ Mounts)	To house the motors, and interface the motors & platform.	

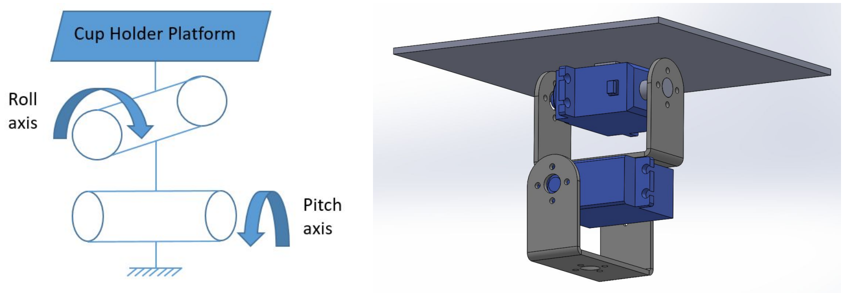


Figure 2. *High level diagram of cup stabilizer configuration.*

2.2. Electrical & Software

The original design operated under closed loop, PID operations. High-level software functionality is shown in Figure 3 below. The required electrical components are detailed in Table 2 below.

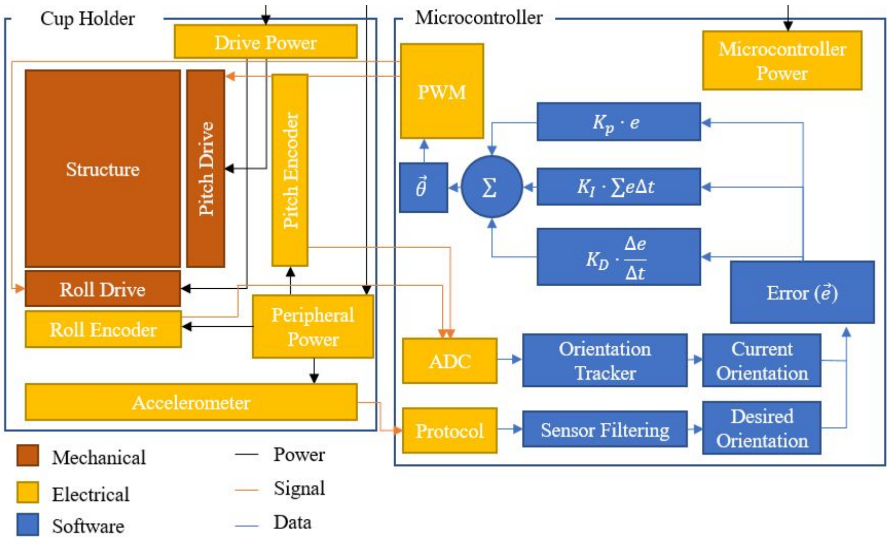


Figure 3: *Proposed Design Operation at System-level*

The self-levelling cup holder system will receive measured inertial changes of the vehicle, which will subsequently be transmitted to the orientation tracker for dynamic adjustment of the angular position. Roll and pitch drives are needed to actuate the positional adjustments for the cup holder system.

Table 2: *Electrical Components*

Component	**Function/Operation**	**Selection**
Power	Separately provides the required power for the drive, peripheral, & microcontroller units.	3 x Power Sources
Absolute Rotary Encoder	Provides an absolute measurement for the pitch & roll angular position. An absolute measurement minimizes the CPU load as offsets do not need to be calculated.	2 x Rotary Encoders Bourns Absolute Encoder [2]
Accelerometer	Provides information on the inertial forces experienced by the vehicle & its contained items.	1 x Accelerometer MPU6050 [3]
Analog-to-Digital Converter (ADC)	Converts transmitted signals from roll & pitch encoders for the orientation tracker.	1 x ADC ADS1115 [4]
Microcontroller	Adjusts the angular position relative to the received sensory information. The MCU should include ports for relevant communication protocols (I^2C).	1 x MCU Arduino Mega2560 [5]
Pulse-Width Modulation (PWM) Generator	From the closed control feedback loop, the calculated angular adjustment needs to be transmitted to the pitch & roll drives.	1 x PWM Part of MCU

3. Proof of Concept & Testing

On-hand components and simple manufacturing methods were used to construct a proof of concept prototype. This section discusses the design methodology, testing process and results analysis to refine the final, intended design.

The proof of concept was built from cardboard due to limited 3D printer access, hence, it uses a stiffer, box-like structure instead of the proposed design. There is also only one control axis (instead of two), and a more powerful HS-645 MG servo is used. However, in software, the system still outputs to two servos to get a better idea of performance. There are no encoders, thus, there is no control loop either. The design team aims to quantify this proof of concept to move towards a more complete prototype and final design.

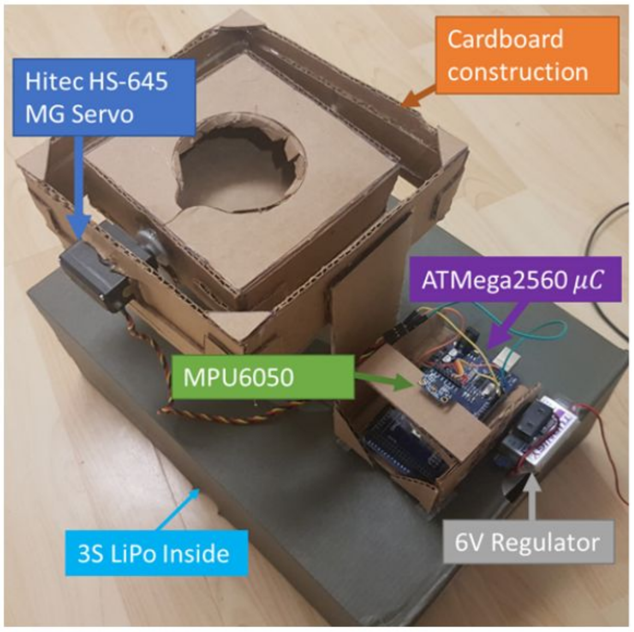


Figure 4. *Proof of concept overview.*

A video demonstration is available:
https://youtu.be/ZMVGCV2L7J8

3.1. Consideration of Real-World Factors

Typical cars under normal operation accelerate and decelerate anywhere from 0.2 to 0.5 $\overline{g}$, with some electric vehicles approaching 1.2 $\overline{g}$ [14]. Using the initial linear model, this implies an angle of:

$$\theta = \arctan(\frac{a_x}{a_z}) = \arctan(\frac{1.2}{1}) = 50.194^\circ$$

The feedback system, primarily the drive and encoder-based feedback system, should be designed for $\theta \in [-50, 50]^\circ$ or a reasonable subset, such as $\theta \in [-45, 45]^\circ$. In the case of the Inertial Measurement Unit (IMU), this implies a linear acceleration of $\pm 1.2\,\overline{g}$ reading is sufficient - the lowest actual selectable for MPU6050 is $\pm 2g$. Configuring the MPU6050 in this region provides the most accuracy, however, this information could also be used to refine the IMU selection further (for mass production).

3.2. Inertial Measurement Unit (IMU) Characterization and Refining

The proposed method involved using only the accelerometer to determine the angle that the cup should be held. However, the accelerometer is subject to noise due to small motions. To determine the applicability and relevance of a filter, the design team ran a test by looking at raw data from the IMU. The accelerometer provides 16-bit output in the full-scale range (FSR) $[-32767,\ 32767]$ with an adjustable reading (e.g., $\pm 2\overline{g}, \pm 4\overline{g}, \pm 8\overline{g}$). For this application, $\pm 2\overline{g}$ is sufficient based on the above. The following plot shows an experiment where the system is static and then lifted by hand and shaken. The data was then post-processed to include filters.

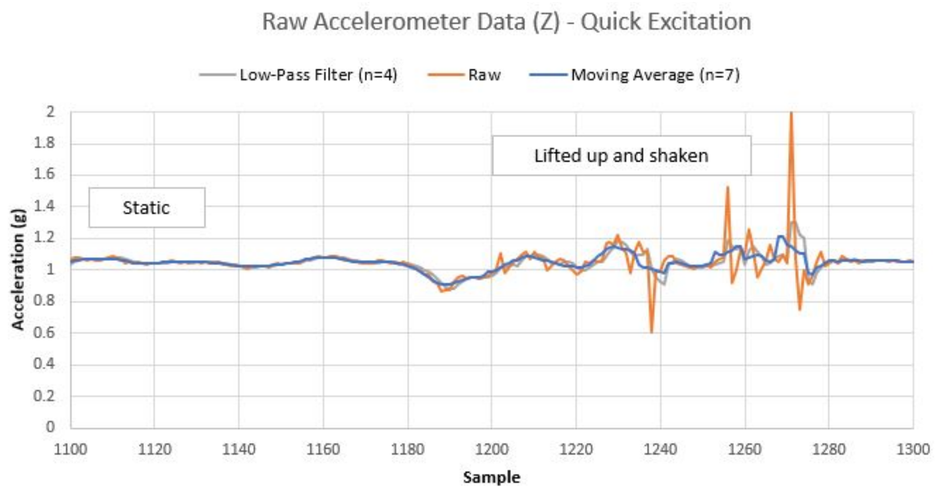


Figure 5. *Raw, low-pass filter (n=4) and moving average (n=7 - +/-3 center) filter. On average, the sampling rate for this test is 78.74 Hz.*

To determine if the data is noisy as-is, consider the output angle function:

$$\theta = \arctan(1/a_z \cdot a_{x,y})$$

For evaluating uncertainties, the following holds:

$$\sigma_\theta = \pm\sqrt{\left(\frac{\partial\theta}{\partial\sigma_{a_x}}\sigma_{a_x}\right)^2 + \left(\frac{\partial\theta}{\partial a_z}\sigma_{a_z}\right)^2}$$

$$\frac{\partial\theta}{\partial\sigma_{a_x}} = \frac{1/a_z}{1+(a_x/a_z)^2};\ \frac{\partial\theta}{\partial\sigma_{a_z}} = \frac{-a_x/a_z^2}{1+(a_x/a_z)^2}$$

Note: the error contribution from the Z-axis is nil when the X-axis (or Y-axis) nominal acceleration is 0 (such as in static conditions and Z-axis oriented with gravity).

Consider the expanded plots of accelerometer raw data for static conditions, as shown below:

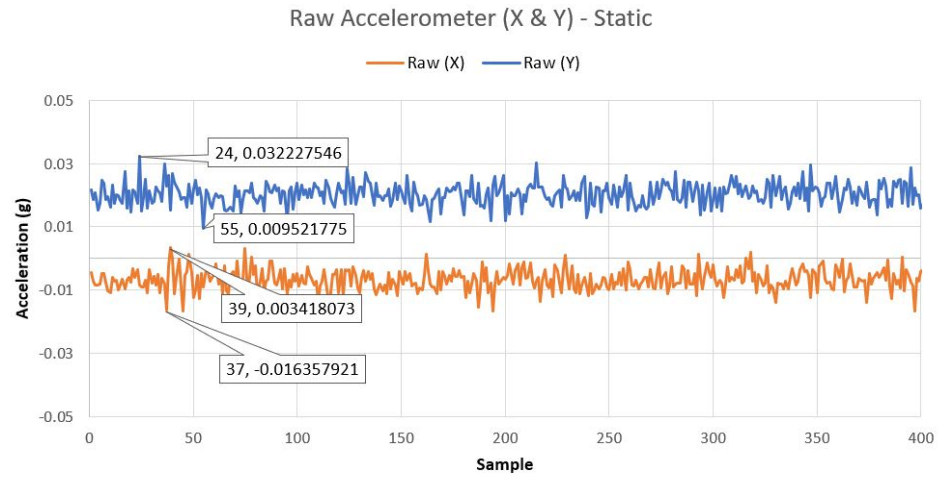


Figure 6(a). *MPU6050 IMU output when held steady on a level table.*

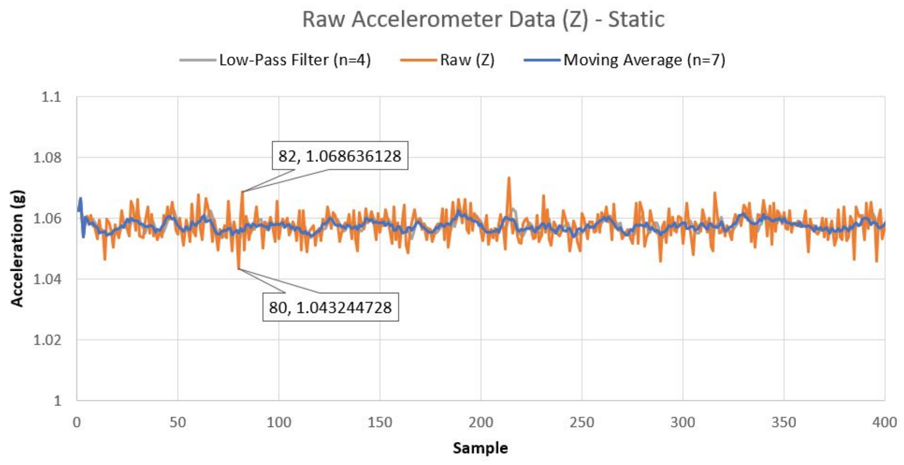


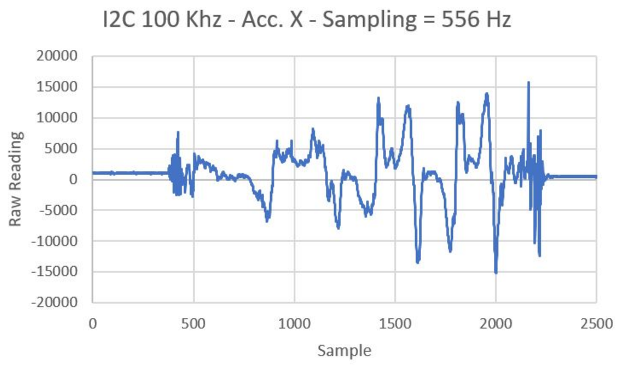
Figure 6(b). *MPU6050 IMU output when held steady on a level table.*

On the z-axis, the raw data has a fluctuation of about $\pm 0.0127g$ (i.e., raw ± 208 which encompasses 1.27% of FSR). On the x and y axes, it is about $\pm 0.01135g$ (raw ± 185) and $0.00989g$ (raw 162), respectively. This may have an impact in the form of jitter. Substituting the raw uncertainties: $\sigma_{a_x} = 185$, $\sigma_{a_z} = 208$ at nominal values of $a_x = 0$ and $a_z = 1g = 16384$:

$$\sigma_\theta = \pm \sqrt{\left(\frac{1}{16384} \cdot 185\right)^2 + (0)^2} = \pm 0.0113\ rad = \pm 0.647^\circ$$

Consider that the (original) intended operation was using an 8-bit PWM output ($180^\circ/256 = 0.703^\circ/step$), the error mentioned above is thus relevant. Taking multiple measurements reduces error (assuming the system is relatively static - i.e., quasi-static). This is similar to what the low-pass filter (LPF) is doing. When realized, the system will experience less jitter with a filter, however, it may have a slight time lag. The significance of time lag depends on the number of samples, sampling rate, overall code performance and how data is buffered. With a burst sampling rate of about $78.74\ Hz$, 4 samples (such as the basic LPF shown below) take approximately $50.8\ ms$. The moving average filter takes 88.9 *ms*. It is possible to sample the IMU's accelerometer at 1 KHz and the gyroscope at 8Khz. For this test, however, sampling time included the time to output Serial data to the console.

The I^{2}C protocol on the MPU6050 can operate in "Fast" mode at a frequency of 400 KHz. The previous test used an I^{2}C speed of 100 Khz. The following plots are generated for a side-to-side motion test at I^{2}C communication frequencies of 100 KHz (default) and 400 Khz (fast), respectively.



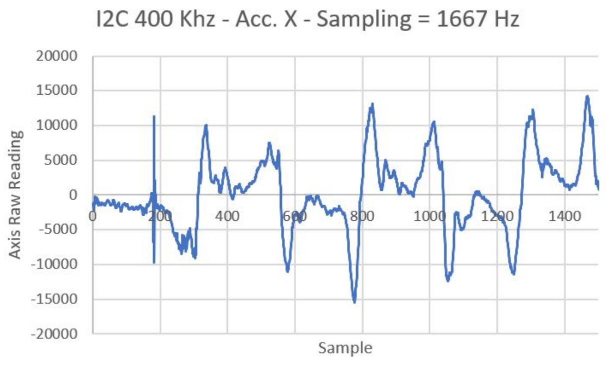


Figure 7. *IMU raw output data to isolate I2C communication speed improvements.*

Based on the above data and isolating the timer at the I^2C calls, using fast I^2C allows sampling the IMU at up to 1.67 Khz. However, the accelerometer itself updates at 1 Khz. Therefore, reading accelerometer registers above 1 Khz will give duplicate readings. The gyroscope can be read at up to 8 Khz. Once the system is fully integrated, the actual sampling time will depend on algorithmic performance. The important thing is to note that, without filtering, the time between a read request and data received over a 400 Khz I^2C link is about 300 us or 4800 CPU cycles (ATmega @ 16 Mhz).

With sampling rate improved, the design team revisited adding a low pass filter. The MPU 6050 includes a digital low pass filter (DLPF) that is configured to 200 Hz by default. First, a target variation is set. The design team opted for 25% of the original error, as it is clear of the PWM/step region. This requires a raw reading variation of:

$$\sigma_{a_x} = \frac{\pm 0.0113}{4} \cdot 16384 = \pm 54.48$$

The design team experimented with this filter, as covered in Appendix A and settled on a 44 Hz filter, which adds a time delay of 4.9 ms. A rolling buffer based, low-pass filter (via arithmetic average) implementation on the microcontroller could achieve better results, however, this proved sufficient. Also, one could use the interrupt pin on the IMU to get results instead. This is discussed in the microprocessor performance section. Finally, reacting to vibrational frequencies above $N_f = f/2 = 1000/2 = 500\ Hz$ would be erroneous, as per Nyquist's sampling theorem. The low DLPF cutoff frequency helps avoid this.

3.3. Drive Performance

Based on tests performed, the bottleneck in the proof of concept is the servo drive system. It appears to work well enough for simple motions, however, it does overshoot at times. In the next prototype, one could wire into the control unit for a more complete characterization, as shown in the figure below. One could also explore a custom servo solution, with a different motor and motor controller selection.

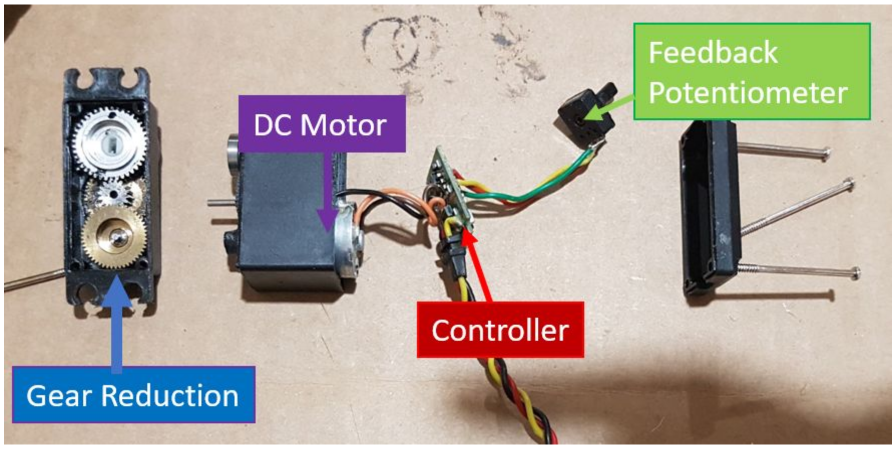


Figure 8. *Servo teardown for this project. One could connect into the controller and potentiometer to sample reaction performance. The part number on the controller was not readable, thus, we could not explore further.*

3.4. Sensor Communication

Communication between the sensors and the microcontroller is carried out by the I^2C protocol. The I^2C protocol was selected as all components are compatible, and serial data transfer can be safely and consistently executed. The expected inputs and outputs for the sensory network is shown in Fig. 9 below.

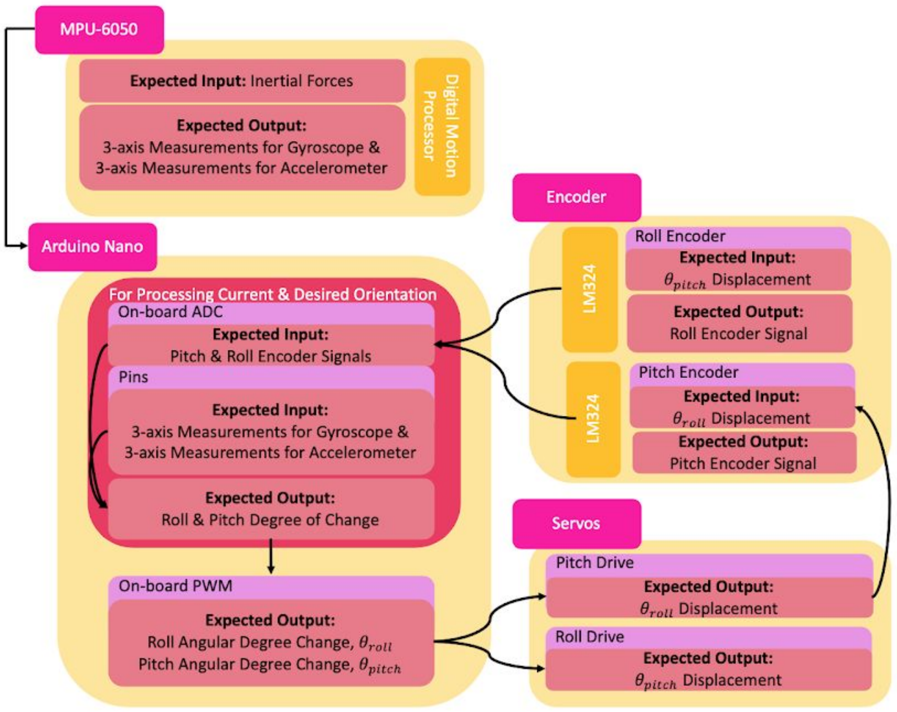

MPU-6050
Expected Input: Inertial Forces
Expected Output: 3-axis Measurements for Gyroscope & 3-axis Measurements for Accelerometer
Digital Motion Processor
Arduino Nano
For Processing Current & Desired Orientation
On-board ADC
Expected Input: Pitch & Roll Encoder Signals
Pins
Expected Input: 3-axis Measurements for Gyroscope & 3-axis Measurements for Accelerometer
Expected Output: Roll & Pitch Degree of Change
On-board PWM
Expected Output: Roll Angular Degree Change, θ_{roll} Pitch Angular Degree Change, θ_{pitch}
Encoder
LM324
Roll Encoder
Expected Input: θ_{pitch} Displacement
Expected Output: Roll Encoder Signal
LM324
Pitch Encoder
Expected Input: θ_{roll} Displacement
Expected Output: Pitch Encoder Signal
Servos
Pitch Drive
Expected Output: θ_{roll} Displacement
Roll Drive
Expected Output: θ_{pitch} Displacement

Figure 9: *Expected Inputs & Outputs for Sensors/Drives*

Data transfer for I^{2}C, which uses one data line, and one clock line; is serial as data transmission is transmitted and received separately. Data transmission is synchronous and reliable due to the start and stop conditions, acknowledgement bits for successful data receival, and address bits to prepare the specified slave for receival. It is important to note that I^{2}C devices can influence the clock rate via clock stretching for synchronization purposes. The data transmission is shown in Fig. 10 below. In addition, as I^{2}C only requires two communication lines, relative to other comparable protocols, like SPI, I^{2}C is less susceptible to noise, and allows for multi-master and multi-slave configurations. In multi-master configurations, multi-masters can operate concurrently.

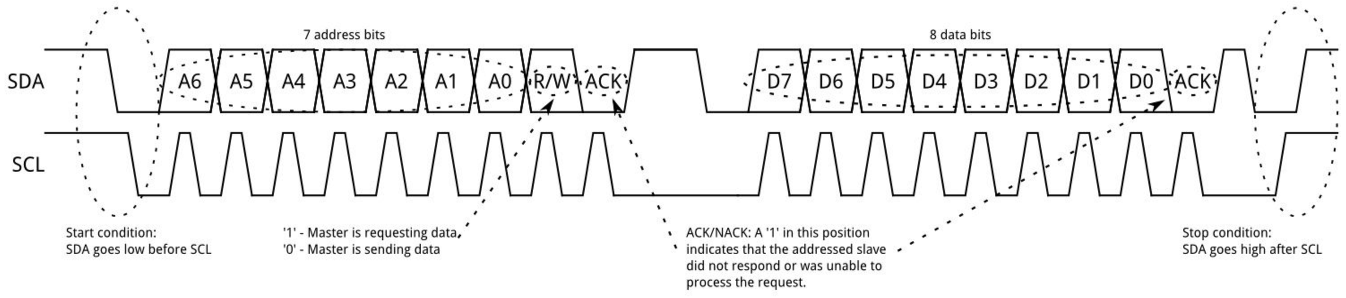

7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition: SDA goes low before SCL
'1' - Master is requesting data, '0' - Master is sending data
ACK/NACK: A '1' in this position indicates that the addressed slave did not respond or was unable to process the request.
Stop condition: SDA goes high after SCL

Figure 10: *I2C protocol. [13]*

For embedded applications, impeded data flow is a concern, which is typically attributed by insufficient parameters for speed, and undersized hardware. However, as per Section 3.3, the communication protocol is not the bottleneck.

3.5. Microprocessor and Code Performance

The program was written in a high-level C language due to readability, rapid development and exploring the AVR toolchains capabilities in optimization. A key high-level code optimization by the design team is a reduction of function calls via the *inline* keyword, without compromising readability. The program also utilizes Arduino's

Servo libraries (for setting up PWM timers for typical RC-hobby servo format), and the code is based on Jeff Rowberg's MPU6050 implementation [19]. Looking into the assembly output (discussed in Appendix D) shows that the program could be optimized by reducing function calls from external libraries (e.g., *atan2(y,x)* in *math.h* or *Servo.write(x)* in *Servo.h*). The average execution time of the code is about 750 microseconds per loop or a sampling frequency of 1.33 Khz - a breakdown is provided in Appendix C.

Since the accelerometer is only capable of sampling at 1 Khz, there are two possible refinements: the processor speed can be reduced (refine selection) and one could use an interrupt to reduce overall runtime (and improve system power efficiency). The design team will opt for the latter, as reducing overall clock speed has an impact on communications as well. One could put the MCU in a low-power mode while it waits for an interrupt.

Finally, the current proof of concept shows a system of components with self-contained filters and control loops. Depending on off-the-shelf availability, this may be an ideal solution. That is, select a low-pin, low register, 20 Mhz clock alternative MCU (e.g. within the AVR family) that performs only angular calculations and relays it to the servo drive, while the self-contained drive system ensures positioning via it's own control loop. The AVR family is acceptable for this application, however, the designer should consider purchasing proper debugging tools (e.g., JTAG or ATMEL's I^2C debugger) or selecting a different family of microcontrollers.

4. Design Modifications and Improvements

This section details the design modifications from the proposed design, which considers course content and performance limitations of the originally selected components. The design modifications are summarized in Table 3 below.

Table 3: *Summary of Design Modifications*

Originally Proposed Component	Newly Selected Component
Arduino Mega2560	Arduino Nano [9]
-	LM324 [10]
ADS1115	On-board Arduino Nano ADC [9]

The microcontroller was needed to adjust the angular position of the cup holder relative to the received sensory information from the accelerometer. The Arduino Mega2560 was originally selected due to the technical ability and memory capacity of the integrated microcontroller for prototyping purposes. The ATmega2560 includes an on-board PWM generator, pins for a Serial (UART), I^2C and SPI protocols, 8 KB SRAM and 256 KB Flash (program) memory [5][11].

The MPU6050 has six degrees of freedom as the inertial force measurements are provided by a 3-axis gyroscope, and 3-axis accelerometer [2]. From the MPU6050, the input is processed by the on-board Digital Motion Processor (DMP), and integrated Motion Interface for tracking and processing technology [2]. The DMP functions to process the accelerometer and gyroscope readings to generate accurate 3D values using the Motion Interface for motion tracking algorithms [2]. As a result, the MPU6050 outputs six values, respective to the gyroscope and accelerometer axes.

Upon component-level analysis, the processing ability of the MPU6050 was determined to be efficient in regards to runtime, and precision, which subsequently minimized the computational processing to be executed by the Arduino Mega2560. As the overall computation executed by the Arduino Mega2560 would be minor, the Arduino Mega2560 would be oversized in respect to the project application and program requirements. For example, the final program only uses 8.3 KB (3% of 256 KB) program memory and 582 bytes (7% of 8 KB) SRAM. Thus, the Arduino Nano will be used instead as the integrated microcontroller. The ATmega328, has a more suitable memory capacity for the application, while encompassing high CPU and pipelining performance (i.e., similar clock cycle and instruction performance as the ATmega 2560), and on-board PWM generator, as shown in Table 4 below. The final design will only use two ADC pins (encoders), 1 interrupt pin, 2 PWM output pins, an I2C line, two GPIO pins (e.g., for indicating the system is on via an LED and for toggling the system on/off). In general, the ATMega328's reduced pin count satisfies both quantity and functional requirements (as discussed in Appendix E). The three memory types compared are the flash memory, SRAM, and EEPROM. The flash memory is where the program code is stored, while long term data is stored in the EEPROM, and the SRAM is used to create and manipulate variables. The pipelining detail describes the number of millions of instructions per second at high throughput clock frequency of $16\,MHz$. The CPU speed describes the clock frequency in which the microcontroller operates at.

Table 4: *Summary of Technical Specifications for Arduino Boards*

	Flash Memory	**SRAM**	**EEPROM**	**Pipelining**	**CPU Speed**
Arduino Mega2560 [11]	256 KB	8 KB	4 KB	16 MIPS	$16\,MHz$
Arduino Nano [12]	32 KB	2 KB	1 KB	16 MIPS	$16\,MHz$

Another design modification was the addition of the LM324 operational amplifier. Two rotary encoders are needed to measure the pitch and roll angular position of the system relative to the vehicular motion. The selected rotary encoders have a resolution of 128 absolute positions [2]. Relative to the number of absolute encoded positions, the sensitivity is calculated in Appendix A as 1.42 positions per degree. As this is a low level of sensitivity, the LM324 is implemented to apply an amplifying gain to the input signals, such that the output signal range is increased, as shown in Fig. 11 (right).

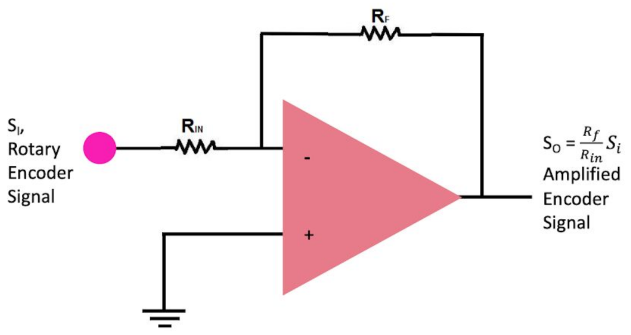

R_F
R_{IN}
S_I, Rotary Encoder Signal
-
+
$S_O = \frac{R_f}{R_{in}} S_i$ Amplified Encoder Signal

Fig. 11: *Op-amp Schematic*

It is important to note that the selected design components are 8-bit, including the Arduino Nano, and rotary encoders. As such, the Arduino Nano's on-board 10-bit ADC [12] will be used instead of the previously selected 16-bit resolution ADC. The proof of concept shows that this performs well enough.

4.1 Control Loop Improvements

For prototype purposes, the internal servo controller was used for position control as it was sufficient for proof of concept. The servo internal controller acts as a proportional controller.

For an industrial prototype, an external controller would be used to control the motor position. A hardware time stamping method to generate a rolling buffer of data would be ideal for the interrupt-based method. From the proof of concept, it is evident that the capability of the ATmega 2560 exceeds the requirement as the resolution allows data to be tracked down to microseconds. Having a PID control loop would allow for the elimination of steady state error and can be tuned to minimize rise and settling time. Trapezoidal and first order divided differences are sufficient for this application and due to the short calculation overhead, the chance of skipping readings due to time lag is lower. The control loop is represented in Fig. 12 below.

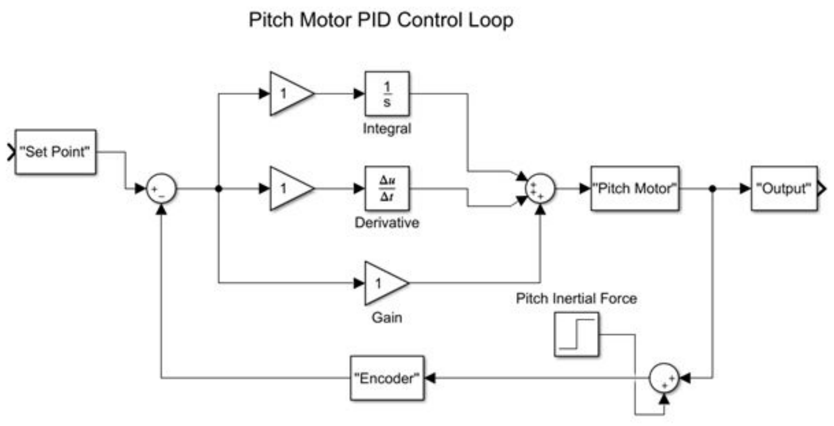


Figure 12. *High level control loop representation of the PID control loop for industrial application.*

5. Conclusion & Summary of Lessons Learned

The design team applied several MIE438 course concepts to determine the feasibility of this project. A proof of concept was created as a substitute of the original prototype. Unlike the proposed prototype, this system had no feedback encoders (thus no control loop). The following list summarizes the applied concepts:

- Control Loops and Peripheral Interaction
  - Analyzing timing and performance requirements on a system and component level
  - Utilizing on-board processing of peripherals for faster operation and/or cost reduction
- Communication Protocols
  - Understanding performance of different protocols and characterizing them for performance validation
- Code Performance
  - Relating assembly instruction set with high-level code
  - Understanding performance limitations of the MCU and optimization benefits in high-level code
  - Optimizing register usage in function calls, communication and data manipulation for faster speed
  - Using interrupts instead of continuous polling for optimized performance
  - Avoiding floating point computations when a dedicated floating point system is not present
  - Minimizing use of subroutine (JSR) to improve performance
- Microcontroller Selection
  - Relating system to microcontroller pin count and functionality (e.g., GPIO, communications, interrupt, PWM)
  - Microcontroller memory size (e.g., memory and code size match use-cases)
  - Characterizing the performance of built-in components (e.g., sampling rate of on-board 10-bit ADC)

6. References

[1] *Electronicoscaldas.com*, 2020. [Online]. Available: https://www.electronicoscaldas.com/datasheet/MG996R_Tower-Pro.pdf. [Accessed: 07- Apr- 2020]

[2] *EAW - Absolute Contacting Encoder (ACE)*. Bourns, 2014, pp. 1-5 [Online]. Available: https://www.digikey.ca/products/en?keywords=EAW0J-B24-BE0128L-ND. [Accessed: 29- Jan- 2020]

[3] *MPU-60X0-Datasheet*, 3rd ed. Sunnyvale: InvenSense, 2013, p. 31 [Online]. Available: https://43zrtwysvxb2gf29r5o0athu-wpengine.netdna-ssl.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf. [Accessed: 29- Jan- 2020]

[4] *ADS111x Ultra-Small, Low-Power, I2C-Compatible, 860-SPS, 16-Bit ADCs with Internal Reference, Oscillator, and Programmable Comparator,* 2nd ed. Texas Instruments, 2018 [Online]. Available: http://www.ti.com/lit/ds/symlink/ads1113.pdf. [Accessed: 29-Jan- 2020]

[5] "Arduino Mega 2560 Rev3 | Arduino Official Store", Store.arduino.cc, 2020. [Online]. Available: https://store.arduino.cc/usa/mega-2560-r3. [Accessed: 29- Jan- 2020]

[6] "Safe to read volatile variables?", *Forum.arduino.cc*, 2020. [Online]. Available: https://forum.arduino.cc/index.php?topic=331307.0. [Accessed: 29- Jan- 2020]

[7] "When is an Absolute Encoder Right for Your Design? | CUI Devices", *CUI Devices*, 2020. [Online]. Available: https://www.cuidevices.com/blog/when-is-an-absolute-encoder-right-for-your-design. [Accessed: 29- Jan- 2020]

[8] C. Mummadi, F. Leo, K. Verma, S. Kasireddy, P. Scholl, J. Kempfle and K. Laerhoven, "Real-Time and Embedded Detection of Hand Gestures with an IMU-Based Glove", *Informatics*, vol. 5, no. 2, p. 28, 2018.

[9] "Arduino Nano | Arduino Official Store", Store.arduino.cc, 2020. [Online]. Available: https://store.arduino.cc/usa/arduino-nano. [Accessed: 7- Apr- 2020]

[10] *Texas Instrument,* 2020. [Online]. Available: http://www.ti.com/lit/ds/symlink/lm324.pdf. [Accessed: 08- Apr- 2020]

[11] *Ww1.microchip.com*, 2020. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_Summary.pdf. [Accessed: 07- Apr- 2020]

[12] *Ww1.microchip.com*, 2020. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. [Accessed: 07- Apr- 2020]

[13] "I2C - learn.sparkfun.com", *Learn.sparkfun.com*, 2020. [Online]. Available: https://learn.sparkfun.com/tutorials/i2c/all. [Accessed: 08- Apr- 2020]

[14] Testing, Testing - The Motor Trend Way - Motor Trend - MotorTrend", MotorTrend, 2020. [Online]. Available: https://www.motortrend.com/news/motor-trend-testing/. [Accessed: 09- Apr- 2020]

[15] “ATMega 256X - Detailed Specifications Sheet”, *www.microchip.com*, 2019. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf

[16] “AVR Instruction Set Manual”, www.microchip.com, 2020. [Online.] Available: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

[17] “DigiKey: ATMega 2560 Cost”, www.digikey.com, 2020. [Online]. Available: https://www.digikey.ca/product-detail/en/microchip-technology/ATMEGA2560-16AUR/ATMEGA2560-16AURCT-ND/3789436

[18] “DigiKey: ATMega 328 Cost”, www.digikey.com, 2020. [Online]. Available: https://www.digikey.ca/product-detail/en/microchip-technology/ATMEGA328-MUR/ATMEGA328-MURTR-ND/2271027

[19] “I2C Device Library Collection”, www.github.com, 2020. [Online.] Available: https://github.com/jrowberg/i2cdevlib

Appendix A: Rotary Encoder Sensitivity

This appendix details the sensitivity calculation for the Bourns Absolute Encoder. To calculate the encoder sensitivity, the following equation was used:

$$Sensitivity = Resolution \,/\, Range\ of\ Degrees$$

As per the encoder datasheet, the resolution is 128 absolute encoded positions [2]. The range of degrees is relative to the vehicle's range of motion. As vehicles have nonholonomic motion, the range of motion is limited to 90 degrees, such that:

$$Sensitivity = 128\ positions \,/\, 90°$$
$$Sensitivity = 1.42\ positions\ per\ degree$$

Appendix B: Digital Low-Pass Filter (DLPF) in MPU6050

This appendix provides additional graphs retrieved from testing the DLPF module on the MPU6050. The baseline is the DLPFCFG register set to 0. The following table is provided as reference for impacts on delays and sampling time (gyro impacted only). Only X-axis accelerometer readings are shown, however, the effects are similar for other axis.

Table A.1. *Bandwidth and delay from using the digital low pass filter built-in the MPU6050.*

DLPF_CFG	Accelerometer (F_s = 1kHz)		Gyroscope		
	Bandwidth (Hz)	Delay (ms)	Bandwidth (Hz)	Delay (ms)	Fs (kHz)
0	260	0	256	0.98	8
1	184	2.0	188	1.9	1
2	94	3.0	98	2.8	1
3	44	4.9	42	4.8	1
4	21	8.5	20	8.3	1
5	10	13.8	10	13.4	1
6	5	19.0	5	18.6	1
7	RESERVED		RESERVED		8

The first test was on a DLPF_CFG of 1.

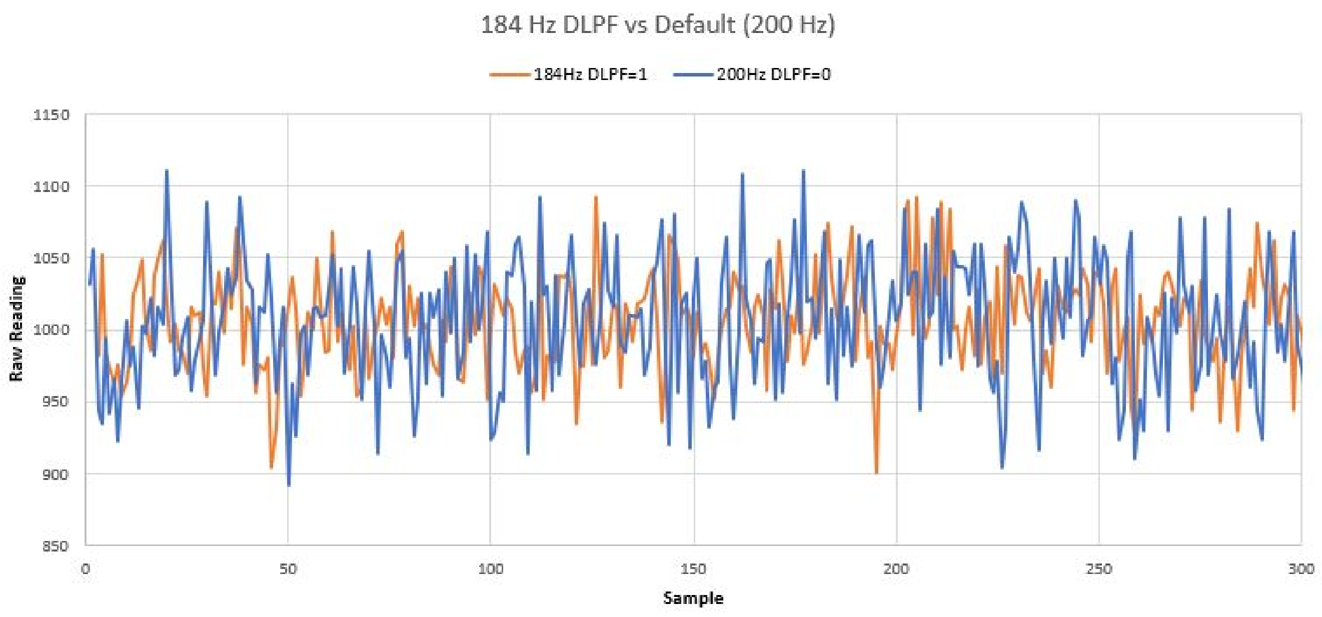


Figure A.1. *Effects of a 184 Hz DLPF compared to baseline performance.*

This marginally reduced noise, however, the required variation of ± 92 isn't always cleared. The next test was with the maximum setting (DLPF_CFG = 6 or 5 Hz).

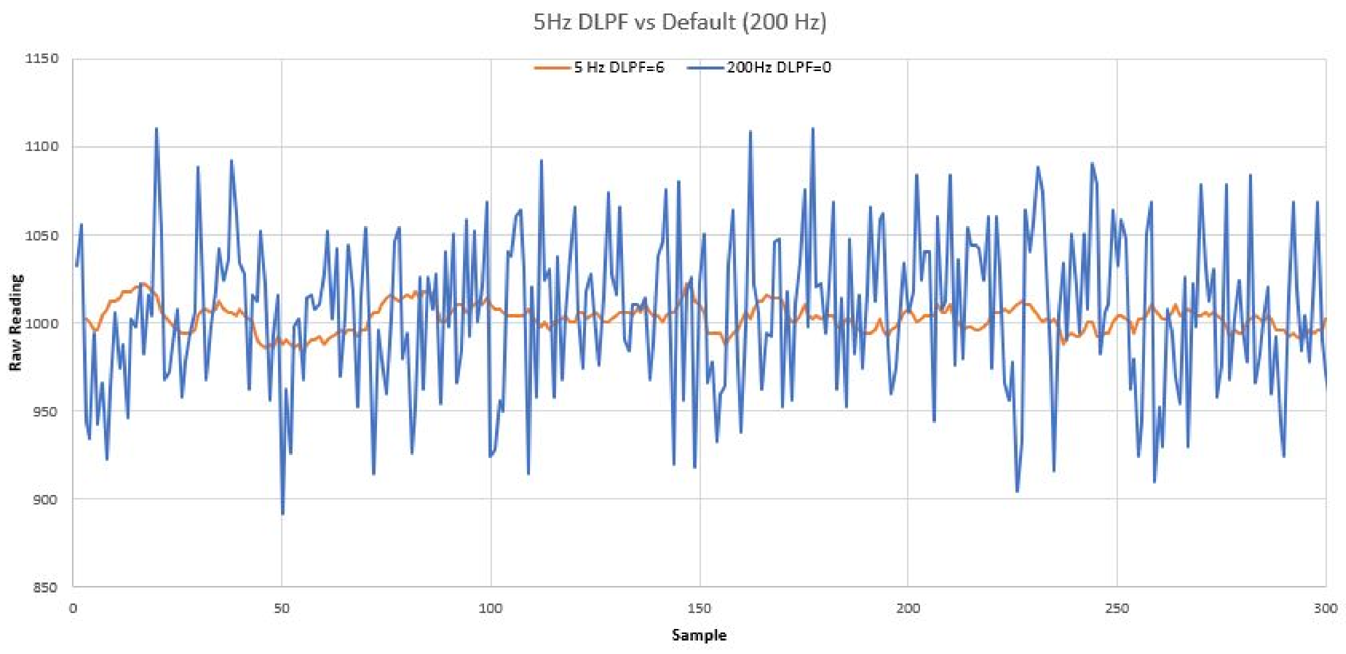


Figure A.2. *Effects of a 5 Hz DLPF compared to baseline performance.*

As expected, this produced the best results, with an average variation of ± 32 and time delay of 19 ms.

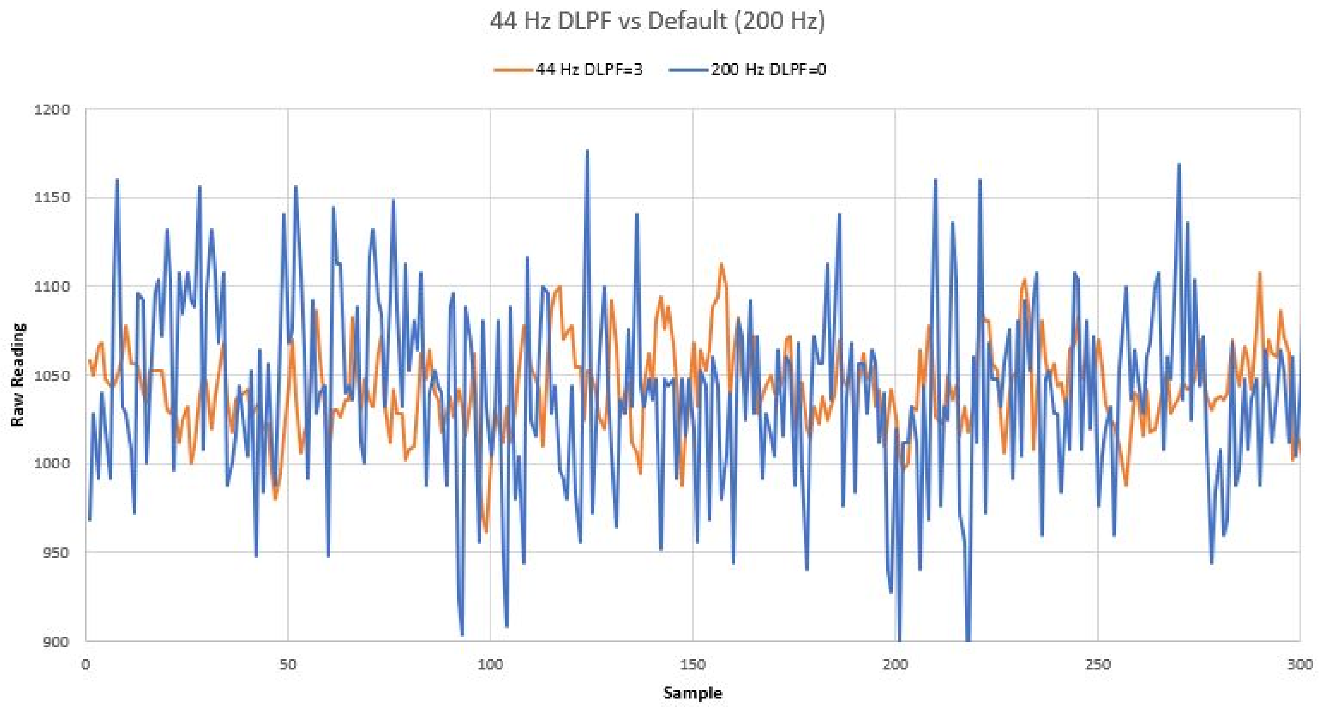


Figure A.3. *Effects of a 44 Hz DLPF compared to baseline performance.*

This has a ± 50 raw reading to reduce the error below the jitter range of the servo. This was a safe pick.

Appendix C: Microprocessor Performance Breakdown

The following table outlines benchmarking results on the code.

Function	Code	Time (*us*)

Retrieving accelerometer data over I2C (Fast Mode - 400 KHz).	Wire.beginTransmission(MPU_ADDR);
	Wire.write(0x3B);
	Wire.endTransmission(false);
	Wire.requestFrom(MPU_ADDR, 6, true);
	ax = (Wire.read() << 8 | Wire.read());
	ay = (Wire.read() << 8 | Wire.read());
	az = (Wire.read() << 8 | Wire.read());
		296

Calculating Pitch and Roll Angles	pitchAngle = floor(-atand(ay, az));
	rollAngle = floor(-atand(ax, az));
		384

Changing PWM output settings for Servos.	pServo.write(P_SERVO_ZERO + pitchAngle);
	rServo.write(R_SERVO_ZERO + rollAngle);
		68

Appendix D: High-Level Code Disassembly

Using the AVR toolchain's *objdump* executable, the design team explored the compiler's behaviour. The following are our observations from looking into the *loop()* function. In general, the compiler did not inline functions when they were not specified as such. Also, stack usage was limited to function calls (SRAM was prioritized for intermediary values). As such, performance improvement may be had by approaching this problem from assembly. Alternatively, instead of using the external libraries as-is, one could rewrite them for the use-case in hand (e.g., specify inline).

Table D.1. Observations from disassembly. Please refer to the *assembly.txt* file for reference.

C-code	Assembler	Notes

void loop() {
#ifdef DEBUG_LOOP_TIME
startTime = micros();
#endif
..
	[Lines: 4270 - 4277]
	
	call 0x5bc; 0x5bc <micros>
	sts 0x0282, r22; 0x800282 <startTime>
	sts 0x0283, r23; 0x800283 <startTime+0x1>
	sts 0x0284, r24; 0x800284 <startTime+0x2>
	sts 0x0285, r25; 0x800285 <startTime+0x3>
		Function call for using micros(). In addition, there is performance overhead from using a "long" data type, as seen with multiple 8-bit registry data transfers to SRAM.

Requesting Data from MPU		
Wire.beginTransmission(MPU_ADDR); // 0x68	[Lines 4280 - 4303]	This function and it's contained functions were made inline, but only because the developers

		specified them as such.
Wire.write(0x3B);	[Lines 4310 - 4325] ldi r25, 0x03; 3 **rcall** .-3086; 0xaf2 ldi r24, 0x00; 0 **rcall** .-1448; 0x115c ldi r20, 0x01 ; 1 ldi r22, 0x06 ; 6 ...	Few subroutines/calls used.

Wire.endTransmission(false);
Wire.requestFrom(MPU_ADDR, 6, true);
ax = (Wire.read() << 8 | Wire.read());
ay = (Wire.read() << 8 | Wire.read());
az = (Wire.read() << 8 | Wire.read());
	Lines [4343 - 4367]	The compiler does split up register usage fairly well. It also reuses them, and it tries to use registers for function calls. However, there are a few SRAM access/storage which could probably be done via the stack instead.

Creating Output Angles		

pitchAngle = floor(-atand(ay,az));
 	
rollAngle = floor(-atand(ax,az));
	rcall .+1180 ; 0x1c14 <__floatsisf>	Floating point emulation via software. The microcontroller could benefit from a floating point unit on the MCU.

	[Lines 4372-4403] [Lines 4409-4445]	The *atand* (arc tangent in degrees) function was made inline. This makes sense, given that the function was specified as inline. However, there are a few other function calls present (mostly floating point arithmetic emulation).
Output angles to servo after factoring in “zero” angle		

pServo.write(P_SERVO_ZERO + pitchAngle); // 34us
 	
rServo.write(R_SERVO_ZERO + rollAngle);
	[Lines 4441 - 4449]
	[Lines 4450 - 4455]	

End of loop()		

Appendix E: ATmega328 Implementation and Justification

Figure E.1 below show the ATMega 328 pinout. In terms of connections, the :

- Unique for this application:
    - Pin 28, 27 (SCL, SDA) for I2C communications
    - Pin 26, 25 for encoder inputs
    - Pins 11, 12 for PWM outputs for pitch and roll control via external H-bridge
    - Pin 4 for external interrupt (data ready) from IMU (MPU-6050)
- General/Setup:
    - Pins 9, 10 would be occupied for the oscillator
    - Pins 7,8, 22, 21, 20 are a part of general power setup and distribution
    - Pins 1,2,3 could be used to provide an interface for programming or for communication via UART to other components of the vehicle (e.g., fault check, programmatic on/off)

When compared to the ATmega2560, which has 54 digital input/output pins, this is a better selection. Also, the program memory size (32KB) and EEPROM memory (8 KB) is better suited for this application. The ATmega2560 costs $19.47 per chip [17], while the ATmega328 costs $2.57 per chip [18]. Since another prototype is likely, the AVR chipset is suggested again, however, other chip families may help reduce cost further.

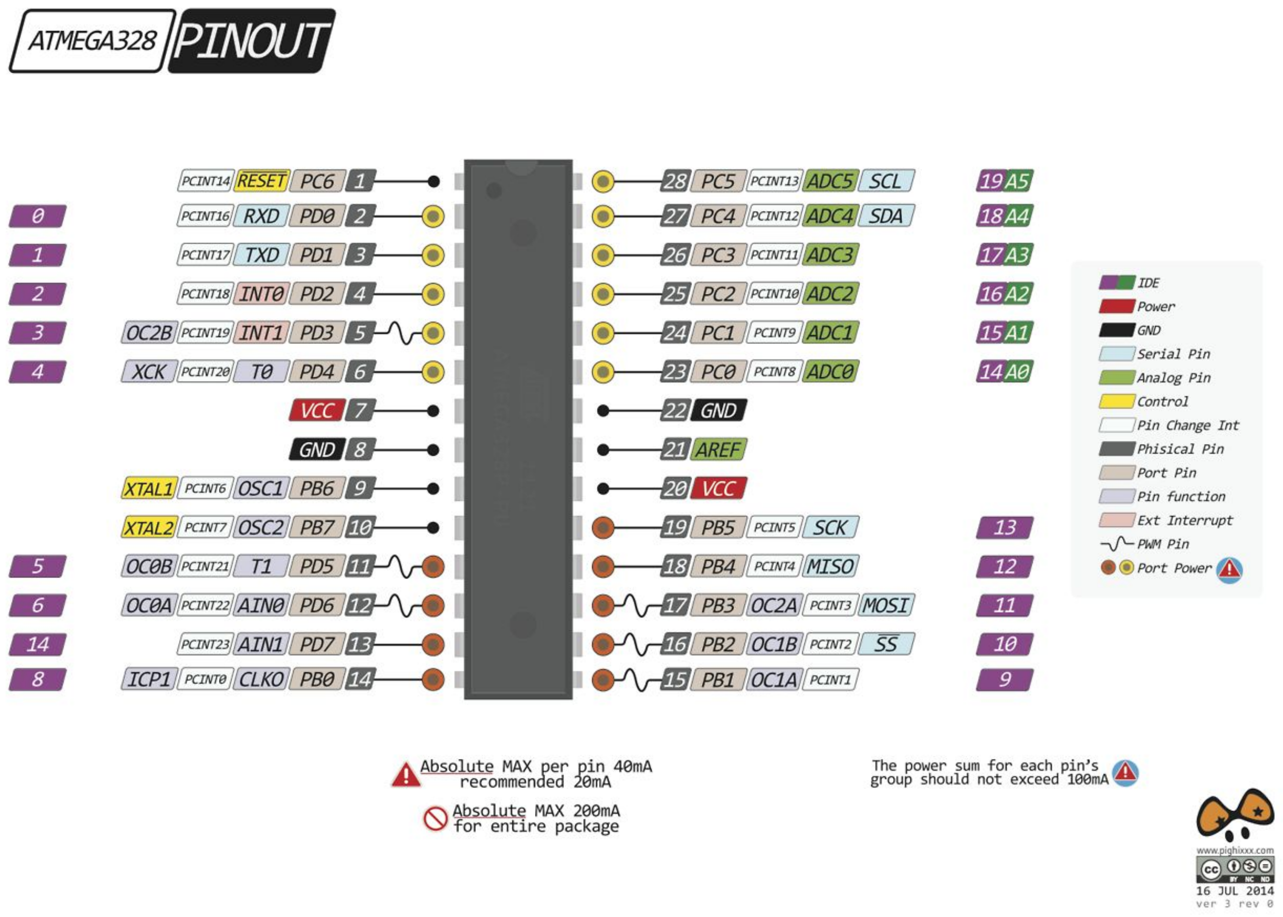


Figure E.1. *ATMega328 pinout schematic (Source: https://www.circuito.io/blog/arduino-uno-pinout/)*