Contract Course Report

Microcontroller Based Engine Governor Simulator – Fall 2022

Griffin White – Prof. Nary Subramanian

Introduction

Many internal combustion engines are designed to run continually at a specified RPM. This is the case in various industrial applications, lawn mowers, generators, etc. Oftentimes, this is essential for proper behavior of the device which the engine is powering. For example, a traditional AC generator's voltage / output frequency depends on the rate at which it is rotating. As such, engine RPM needs to be precisely controlled.

In these applications, an engine's speed is regulated by a governor. This is a mechanical or electronic device which adjusts the engine's throttle in order to maintain a specified RPM.

One of the simplest implementations of a governor is a centrifugal governor, pictured to the right. As the assembly rotates more quickly, the weights move outwards, actuating a linkage and closing the throttle valve. This design has been in use for centuries. It has proven itself to be robust, inexpensive, and relatively simple.

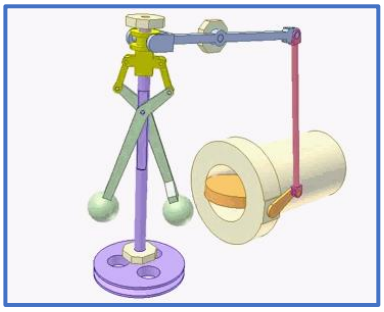


Figure 1: Centrifugal Governor Animation

However, as internal combustion engines become increasingly reliant on computers and electronics, mechanical governors like this have largely fallen out of favor for electronic designs. Computerized engine governors have several potential advantages:

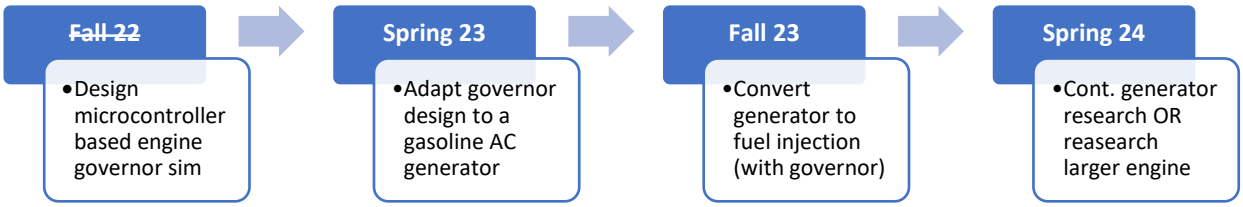
- **Increased Accuracy / Awareness:** Simple mechanical governors operate independently of any RPM signal. As such, these devices do not know the actual speed of the engine. They simply tell the engine to speed up or slow down. Computerized governors have access to an accurate RPM signal and know exactly how their adjustments are affecting the engine.
- **Increased Precision:** Depending on how they are implemented, computerized governors can perform very minute adjustments. For example, through the use of stepper motors, computerized governors can adjust the throttle valve by fractions of a degree.
- **Increased Consistency / Reliability:** Over time, mechanical governors may become worn out and need adjustment. However, computerized governors are far less likely to need adjustment as they age. Barring any outright failure, computerized governors should behave the same throughout their lifespan.

Project Summary & Further Research

This semester, I have designed and constructed an engine simulator which is controlled by a computerized governor. This apparatus simulates an internal combustion engine and ensures that it is operating at a specified RPM. In the following sections, I will provide further detail about this apparatus.

This contract course is the first of several related research projects which I hope to pursue over the coming semesters. I am pleased with this semester's results, and I am ready to apply my governor design to an actual engine (Spring 2023).

Figure 2: Tentative Research Timeline



Apparatus Overview

The engine simulator / governor can be broken down into several major components, each of which are described in this section. Additionally, at the end of the document, I have provided logical and physical diagrams which illustrate how these components interact with one another.

Major Components

Arduino Mega 2560

This microcontroller acts as the computerized governor. It calculates the RPM and adjusts the system as necessary.

DC Motor

This electric motor fills the role of a gasoline engine. It spins at anywhere from ~150 to ~900 RPM. Its speed is monitored and controlled by the Arduino.

Flywheel

This flywheel acts similarly to a flywheel on a gasoline engine. It is driven by the DC motor and rotates at the same rate. There are eight magnets affixed along its circumference. These magnets allow the governor to calculate the RPM.

Hall-Effect Sensor

The hall effect sensor helps calculate the RPM at which the flywheel is spinning. Each time a magnet passes in front of the sensor, it provides an electrical signal to the Arduino.

PWM Motor Controller

This device acts like the throttle on a gasoline engine and controls the speed of the DC motor. It features a small knob (potentiometer). Turning this knob clockwise increases the motor's speed; turning it counterclockwise decreases the motor's speed. This is the same movement as actuating a throttle valve, which allows this system to be adapted to an actual engine.

Stepper Motor

This stepper motor is attached to the knob on the PWM motor controller. The Arduino controls the stepper motor, telling it which direction to rotate and how far to rotate, thereby adjusting the speed of the DC motor.

Variable Brake

This brake simulates load on a gasoline engine (Ex. Plugging in an electrical appliance to a gasoline AC generator). This brake applies friction to the flywheel, changing the amount of force required to spin the flywheel. As this force changes, the governor must act according by providing more or less power to the DC motor.

Other Components

LCD Display

This displays the current RPM and the RPM which the governor is trying to achieve. This display is controlled by the Arduino.

Keypad

This keypad allows a user to interact with the device. The user can modify the desired RPM, and the governor will change the motor's speed accordingly.

Ammeter

This sensor monitors how much current is being pulled by the DC motor. This illustrates how much load is applied to the motor.

The ammeter I am using is rather imprecise, and it doesn't provide any useful data. This device is present on the apparatus; however, it is not being utilized.

Code Explanation

For this project, I had to write a specialized Arduino program which performs all of the governor control logic. In this section, I will provide a high-level overview of how my program functions. For more detailed information, please refer to my attached program; it is commented throughout for clarity.

Main Loop

Programs written for the Arduino microcontroller are typically based on a main loop; I employed this approach in my program. The code in this main loop is executed over and over, until the device is powered off. Each time the loop is run, my program will do the following:

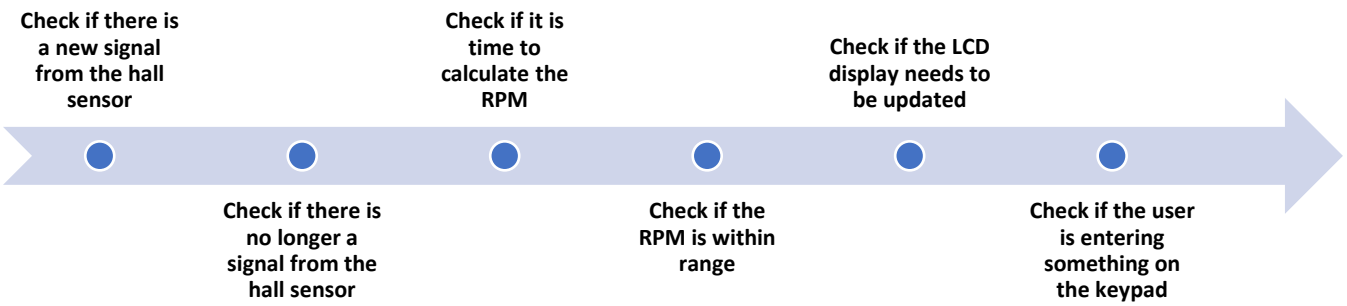


Figure 3: Program Main Loop

Check if there is a new signal from the hall sensor

If so, the Arduino will increment the number of pulses and mark that the hall sensor is currently active.

It is important to only count discrete pulses from the hall sensor (The sensor turning from off to on). For example, if the flywheel is stationary with a magnet stuck in front of the sensor, this must not be interpreted as an endless number of pulses.

Check if there is no longer a signal from the hall sensor

If so, the Arduino will mark that the hall sensor is no longer sending a signal.

This helps to count discrete pulses, by keeping track of when the hall sensor stops giving a signal.

Check if it is time to calculate the RPM

If so, the Arduino will calculate the current RPM and then reset the number of pulses & timer.

Calculating the RPM based off each individual pulse from the hall sensor would create rather inaccurate results. Instead, the program finds the average RPM over 20 pulses (2.5 revolutions).

Check if the RPM is within range

If so, no action is taken. If it is out of range, the Arduino will adjust the throttle.

The amount of adjustment is proportional to the difference between the current RPM and the desired RPM. If there is a slight discrepancy, only a slight adjustment will be made. If there is a large discrepancy, a large adjustment will be made.

Check if the LCD display needs to be updated

If so, the Arduino will print the new info to the LCD display.

Printing to the LCD display is a rather slow process, so there is a risk that a hall sensor pulse could be missed while printing to the LCD. When printing a large amount of text, the Arduino only prints one character for each cycle of the main loop. This ensures that it has a chance to check for pulses.

Check if the user is entering something on the keypad

If so, the Arduino will take in the user's input.

The Arduino will only accept RPM values in the range of 150 to 900. This prevents the user from damaging the device by entering an unachievable RPM.

Conclusion

At the start of this project, I set out to achieve the following artifacts and outcomes: 1. Design and construct a microcontroller-based engine RPM governor. 2. Design and construct an engine simulator. I believe that I have achieved these goals. Furthermore, I feel prepared to go forward with my research and adapt my findings to an actual engine.

Issues and Potential Improvements

While I consider this project to have been a success, I did encounter some issues, and there are some areas which could have been improved.

Ammeter

As noted earlier in the report, the ammeter I utilized did not meet my expectations; I was unable to implement this feature. To implement this, I would need an ammeter with finer precision and less noise. I need to do some additional research to find a suitable device.

If I can get this working, I plan on implementing preemptive throttle control. In theory, a change in amperage (engine load), can be observed more quickly than a change in RPM. Additionally, amperage provides insight on how drastic of a throttle adjustment is necessary. As such, this would allow the governor to begin adjusting the throttle more quickly and do so more accurately.

LCD Display Control

Writing directly to the LCD display is by far the most time-consuming process for the Arduino, taking 10-15ms to write a full line of text. This could be overcome by adding a separate LCD controller. Effectively, the Arduino would communicate with the controller, which in turn communicates with the LCD. This lets the LCD controller handle the slow communication while the Arduino continues running its program.

Multiprogramming / Multitasking

This program would benefit from a multiprogramming / multitasking design, jumping between multiple tasks rather than completing tasks in a linear order. This would require a complete rewrite of my program.

RPM Calculation Improvement

I need to conduct further research on alternative means of calculating RPM, particularly if I plan on implementing my governor on a real engine. My current program constantly checks for a signal from the hall sensor and calculates the RPM every 20 pulses. This approach is satisfactory, but I believe that actual engine computers use a different approach. I need to do further research.

Conclusion

This entire process was quite fulfilling, and I expanded my knowledge in a variety of areas. For example, this was my introduction into the Arduino platform. I found it to be versatile and straightforward. I will definitely make use of it in future projects, likely in future contract courses.

Additionally, this project introduced me to a whole host of unique programming challenges which I had never encountered. In particular, I was presented with time complexity issues. On several occasions, I had to rewrite portions of my code to prevent missing any signals from the hall sensor. It was also interesting to see how these issues manifested themselves. In one case, the governor would miss signals from the hall sensor, causing it to calculate erroneously low RPM figures and then mistakenly increase the speed higher and higher. The programming portion of this assignment was quite rewarding.

Constructing this apparatus was also enjoyable. It was very cathartic to slowly piece this device together, watching as my ideas materialized. Overall, I feel far more confident in my ability to design and construct these sorts of electronics-based projects.

I thoroughly enjoyed this project, and I am excited to see where this research takes me in future semesters. I am looking forward to picking things up next spring.

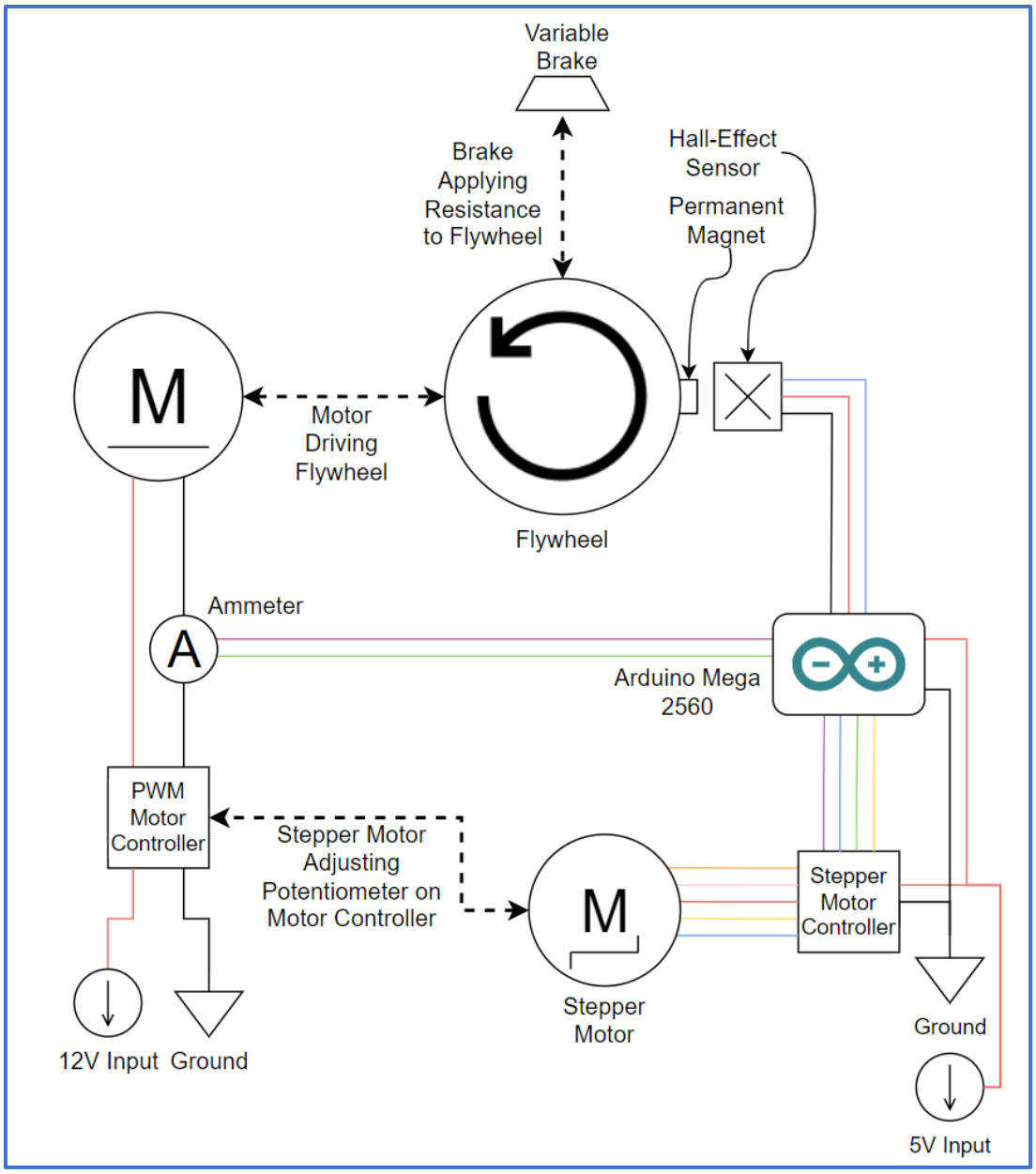


Figure 4: Logical Diagram of Governor / Engine Simulator

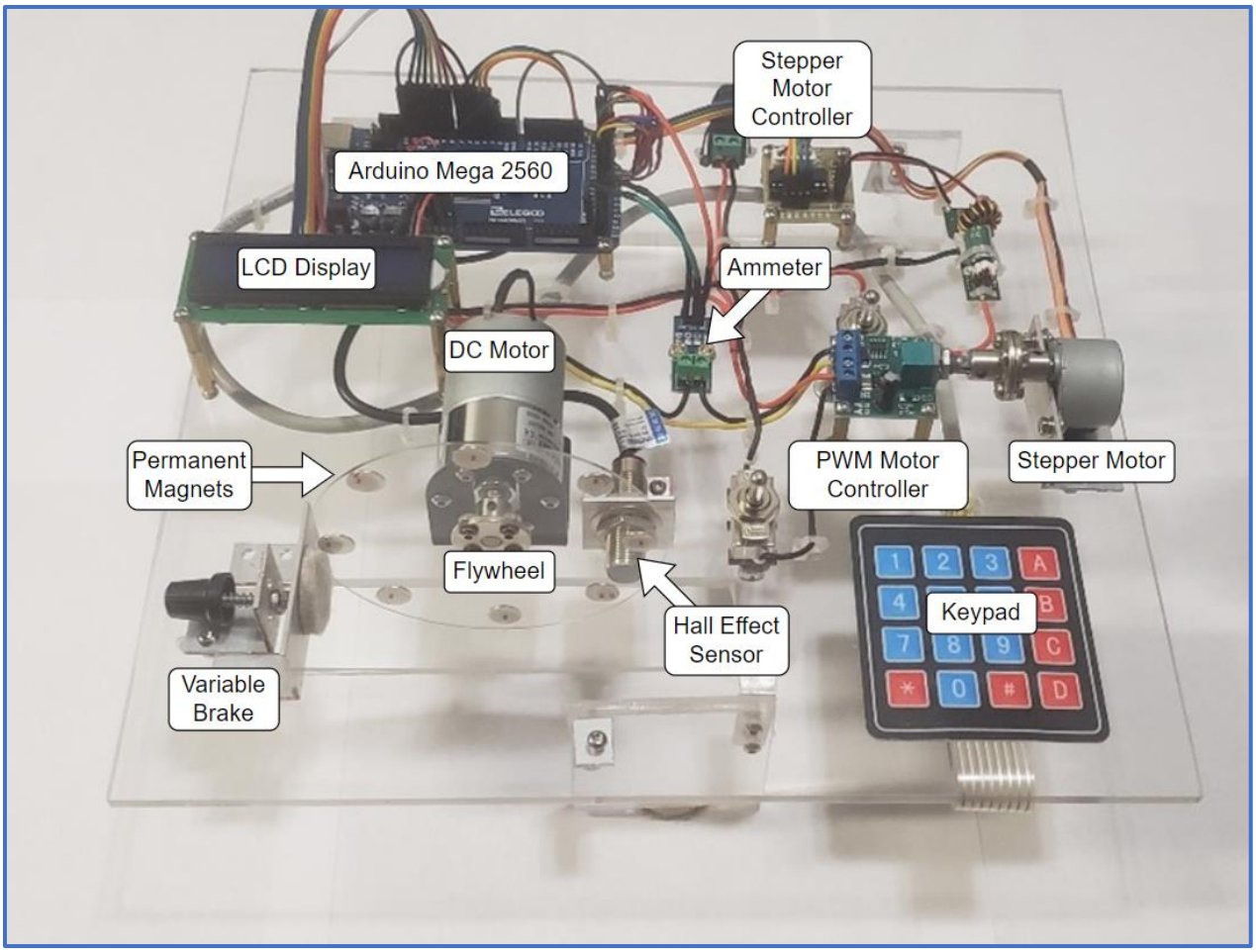


Figure 5: Physical Diagram of Governor / Engine Simulator