Contract Course Report

Microcontroller Based Small Engine Governor – Spring 2023

Griffin White – Prof. Prabha Sundaravadivel

Abstract

Many internal combustion engines are designed to run continuously at a given RPM. This is seen in industrial applications as well consumer-grade products, such as lawn mowers, generators, etc. In these applications, having the engine run at a fixed speed is often integral to the proper function of the device which they are powering. For example, in a traditional gasoline AC generator, the engine's speed determines the output voltage.

In these types of applications, it is essential that the engine's speed is properly regulated. This can be achieved by using a governor. A governor is a mechanical or electronic device which monitors the speed of an engine and adjusts the throttle accordingly. If the engine is running too slowly, the governor will increase the throttle; if the engine is running too quickly, the governor will decrease the throttle.

One of the simplest and most prevalent governor mechanisms is the centrifugal governor. This is a purely mechanical device which regulates an engine's throttle through proportional control. These systems are commonly found on small engines , due to their low cost and lack of complexity.

However, many internal combustion engines have become dependent on computerized control; this allows for the use of electronic governor systems. An electronic system has many potential advantages over a mechanical governor, such as improved control algorithms, improved RPM consistency, and improved overall engine behavior.

Goal of this Project

This project seeks to test the potential advantages of an electronic governor and detail the process of implementing an electronic governor system.

Experiments will be conducted on a 5000-watt gasoline AC generator, initially controlled by its factory-installed mechanical governor. The engine will then be modified with an electronic governor; this process will be documented in detail. After the electronic governor is installed, another round of experiments will be conducted. Once all the experiments are complete, the results of the mechanical and electronic governors will be compared.

Background Information

This section will provide specific information about the equipment used in this project. It will also provide general information about engine control theory and the operating principle of governor systems.

Equipment Used

Generator

The engine testbed for this project is a Coleman Powermate PM0435001 generator, manufactured in 2007. It outputs 120 and 240 VAC single phase, supplying a maximum of 5000 watts continuous power.

The generator features a 9 HP Subaru Robin EX30 engine. This is an air-cooled, single-cylinder, overhead camshaft, four-stroke gasoline engine, displacing 287cc. It utilizes a side draft carburetor and a centrifugal governor; it is designed to run at a constant 3600 RPM.

Electronic Governor Components

The electronic governor system is controlled by an Arduino Mega 2560 microcontroller. Engine RPM is measured using an NJK-5002C normally open Hall Effect sensor. The throttle is actuated using a 28BYJ-48 4-pole stepper motor and a ULN2003 stepper motor driver.

Testing Equipment

During the experiments, an electrical load was placed on the generator. This was done with a 1500-watt Comfort Zone DQ2016 space heater and an 1800-watt Ridgid CM14500 Abrasive Cut-Off machine. Current draw was measured with a P3 P4400 Kill-a-Watt electricity usage monitor. Engine temperature was monitored with an Ames 63985 infrared thermometer.

Operation of a Throttle

A throttle regulates the speed of a gasoline engine by metering the amount of fuel and air which is supplied to the engine. The amount of fuel and air supplied to an engine determines the amount of energy which can be released though combustion, thereby determining how much power the engine outputs.

In most gasoline engines, the throttle takes the form of a butterfly valve placed within the intake. When fully closed, the valve is perpendicular to the direction of airflow and blocks off the air supplied to the engine. When fully open (turned 90 degrees from fully closed), the valve is parallel to the direction airflow and allows the air to pass through with little obstruction. Adjusting the position of this butterfly valve allows for precise control of the amount of air and fuel allowed into the engine.

Butterfly valves have a 90-degree range of motion, rotating from fully closed to wide open. The valve is typically actuated by a

The Need for a Governor

A governor serves to keep an engine running at a constant speed, even when the load on the engine is variable. If the load on an engine is constant, then it can maintain a reasonably constant speed by simply locking the throttle in place, without the use of a governor.

For example, consider a car travelling down a straight and level road. The car is able to maintain a speed of 55 mph on this road with the throttle opened to 30%. If the throttle was fixed in place at 30%, the car would maintain its speed, without any need for interference. When the load placed on an engine is constant, a governor is not necessary; the throttle position simply needs to remain constant as well.

However, consider another example. Say that the level road comes to an end, and the car begins travelling up a steep hill. The throttle remains locked at 30%, and the car starts to slow down. Now, travelling up this hill, 30% throttle only produces a speed of 40 mph. In order to maintain the original speed of 55 mph, some action must be taken; the throttle must be opened beyond 30%. In situations like this, where there is a variable load on the engine, a governor can be used to adjust the throttle; the throttle must vary with the engine load.

In practice, few environments reflect that first example. An engine is seldom placed under a perfectly constant load. As such, governors will almost always be present on engines which are designed to run at a fixed RPM.

Operation of a Centrifugal Governor

Centrifugal governors use the inertial forces affecting a rotating apparatus to actuate an engine's throttle. This apparatus is driven by the engine, and its rotational speed changes along with the engine's speed. As the apparatus rotates, a pair of flyweights change their position and manipulate a throttle linkage. As the apparatus increases in speed, the flyweights move outward, acting on a linkage, and decreasing the throttle. As the apparatus decreases in speed, the flyweights move inward, acting on a linkage, and decreasing the throttle. There are many different governor designs which utilize these basic principles.

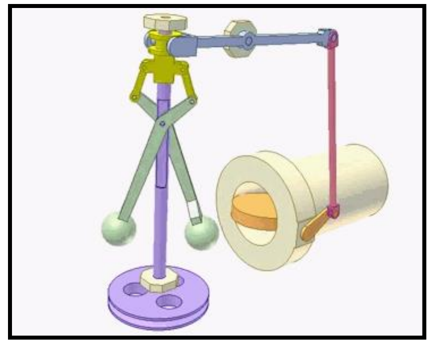


Figure 1: Centrifugal Governor Animation

The Subaru EX30 engine used in this project features centrifugal governor. It is a streamlined design compared to the model pictured above; however, it functions using the same principles. The only additional feature of this engine's governor is the use of a throttle return spring. This spring provides resistance to the governor whenever it attempts to close the throttle. By adjusting the spring pre-load, the governor can be fine-tuned to the proper RPM.

Advantages

- **Low Cost:** Centrifugal governors are often very simple and cheap to manufacture. For example, the governor used on the EX30 is largely comprised of plastic and stamped steel; the complete mechanism is only around 15 parts. This low manufacturing cost is particularly important on small engines, which have a much lower price point than industrial engines.
- **No Need for Electronics:** Centrifugal governors are commonly used on engines with very basic electronics. For example, the only electronic system on the EX30 engine is its ignition system. The EX30 has no battery, no DC charging system, no engine computer, and no sensors for monitoring engine speed. A purely mechanical governor system is well-suited for these applications. To use an electronic governor system, an engine requires a rudimentary electrical system.

Disadvantages

- **Only Capable of Proportional Control:** Centrifugal governors utilize proportional control. They adjust the throttle according to the discrepancy between the current engine speed and the desired engine speed. The governor does not consider factors such as the rate at which the engine speed is accelerating; it is purely concerned with whether the current speed is too fast or too slow. In practice, this can lead to poor engine behavior.

  For example, consider an engine during startup. Initially the engine is stationary and thus below the desired speed. As such, the governor will adjust the engine to full throttle. Once the engine starts, it quickly begins accelerating to the desired speed. If engine speed continues to accelerate at this rate, it will overshoot the desired speed. However, the governor, only concerned with the proportional difference between the current and desired speed, continues to run the engine at full throttle. Once the engine reaches the desired speed, the governor cannot react quickly enough, and the speed continues to rise, going beyond the desired speed. The governor then adjusts the throttle down, decreasing the engine speed to the desired value.

This phenomenon can be minimized, but it remains an inherent limitation of all governors which rely on proportional control. Proportional control does not consider acceleration, cumulative error, and other similar factors. Electronic governors do not have this limitation. They can consider multiple factors in their throttle adjustments.

- **Lacks Direct Speed Awareness:** For lack of a better term, mechanical governors do not “know” the actual engine speed. For example, a centrifugal governor knows that at a certain speed, a pair of flyweights move outwards a certain distance, thus actuating the throttle a certain amount. Here, the flyweights are being used as an analog for the engine speed.

  Compare this to an electronic governor. Using a sensor, the electronic governor can count exactly how many revolutions the engine has made and then divide that value by the time elapsed. The electronic governor knows the actual engine speed; it does not rely on an analog to represent the engine speed.

  Since mechanical governors do not “know” the true engine speed, this can lead to performance issues. Mechanical governors are susceptible to miscalibration and may perform differently depending on engine load, environmental conditions, etc.

Operation of a Microcontroller Electronic Governor

A microcontroller electronic governor features a few key components, namely a speed sensor, a throttle actuator, and the microcontroller itself. The speed sensor serves as the governor’s input; it may take the form of a Hall Effect, variable reluctance, or optical sensor, among others. The sensor provides a signal(s) to the microcontroller each time that the engine rotates, allowing the microcontroller to accurately calculate the engine speed. The throttle actuator serves as the governor’s output; commonly this is either a servo or stepper motor. This actuator manipulates the throttle valve, allowing the microcontroller to change the engine speed.

Advantages

- **Capable of Complex Control Schemes:** Electronic governors are not limited to proportional control. The microcontroller can be programmed with any sort of control algorithm. For example, this electronic governor used in the project utilizes a proportional derivative (PD) control algorithm. The governor makes adjustments given the difference between the current and desired speed (proportional) and the rate at which the engine’s speed is accelerating (derivative). Electronic governors can employ more complex and potentially more effective throttle control schemes than a mechanical governor.
- **Direct Speed Awareness:** Electronic governors “know” the actual engine speed at any given time. The governor knows exactly how its adjustments are affecting the engine speed. As such, they are less prone to miscalibration and should perform the same, regardless of engine load, environmental conditions, etc.

Disadvantages

- **Cost and Complexity:** An electronic will likely be more expensive to manufacture than a comparable mechanical governor. Furthermore, while small engine manufacturers can produce all components of a mechanical governor in-house, these companies would likely have to outsource the production of the semiconductors, sensors, etc. used in an electronic governor.

- **Requires an Electrical System:** Many small engines feature a very basic electrical system, incapable of running an electronic governor system and any required accessories. Adding the required electrical capabilities to such an engine would further increase the cost and complexity of an electronic governor system.
- **Reaction Time:** Electronic governors are limited by the speed of the throttle actuator. The servo or stepper motor being used must be able to rapidly change its position and direction. This is particularly important when large changes in throttle position are required. If the actuator is slow to react, throttle response will be lethargic and engine performance will suffer.

Hardware Design Process

This section will detail the process of designing, constructing, and revising the electronic governor system for this project.

Stepper Motor Carburetor Assembly

The EX30 engine uses a side-draft carburetor with a butterfly valve to control its throttle. The governor will actuate this butterfly valve to regulate the engine's speed. A pair of identical, new carburetors were purchased for this project; one was modified for use with the electronic governor, the other was left unmodified and used with the mechanical governor.

The electronic governor uses a 4-pole stepper motor with a 64:1 gear reduction. This allows for precise throttle adjustment; in practice, the stepper motor has roughly 1200 positions between fully closed and wide-open throttle. This precision comes at the cost of speed, but the reaction time of the stepper motor remains acceptable.

An image of the modified carburetor assembly is shown to the right. The carburetor features a pair of unused M6 x screw holes on either side of the fuel inlet. These screw hole allowed a custom bracket to be mounted on the side of the carburetor. The stepper motor is mounted to this bracket such that its output shaft rotates on the same axis as the throttle butterfly valve. A ball-joint linkage then connects the stepper motor's output shaft to the butterfly valve. Additionally, there is a normally open limit switch mounted to the bracket. This switch is activated when the stepper reaches the end of its travel. Once this switch is activated, the governor will stop attempting to move the throttle any further, preventing the system from causing damage to itself.

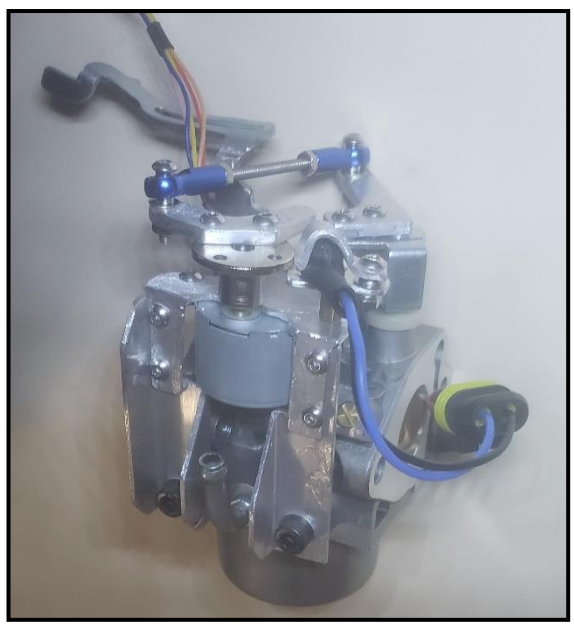


Figure 3: Stepper Motor Carburetor Assembly, Rev. 2

Revisions

The stepper motor carburetor assembly went through several revisions as the project progressed, each revision improving the functionality of the governor system. Additional photos can be found at the end of this report which illustrate each revision.

Initial Mock-Up

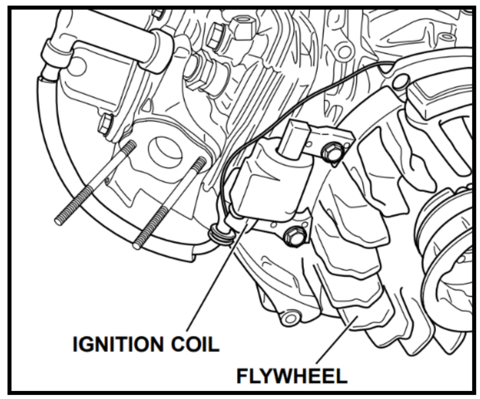
The initial version of the device was purely proof-of-concept; it was never used on a running engine. This version lacked a limit switch and utilized a bent metal rod to serve as a linkage between the

Revision 1

Revision 2

Revision 3

Hall Effect Sensor

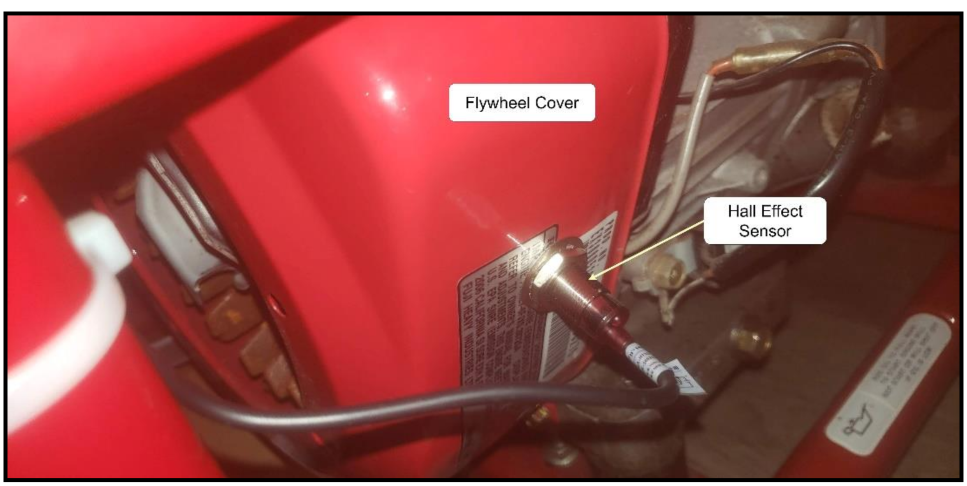


The governor uses a normally open Hall Effect sensor as the engine speed sensor; this sensor monitors the speed of the engine's flywheel.

The EX30 engine uses a simple magneto-based ignition system; it features a single permanent magnet located on the engine's flywheel. For each revolution of the engine, this magnet passes by the ignition coil and generates an electric current, providing spark to the engine.

A Hall-Effect sensor is used in conjunction with this magnet on the flywheel. Each time that the magnet passes in front of the sensor, its normally open contact closes. This provides a consistent, discrete signal to the Arduino. The Arduino records the time elapsed between each pulse and calculates the engine RPM. For example, the sensor will generate a pulse every 16.67 milliseconds when the engine is running at 3600 RPM.

Shown below is an image of the Hall-Effect sensor installed on the engine. A hole was drilled in the flywheel cover. The sensor is mounted in this hole, using a nut and a lock washer on both sides of the flywheel cover. The end of the sensor rests a few millimeters from the edge of the flywheel. The sensor features a 5 VDC input, ground, and an output. These wires are routed along the top of the generator and plugged into the electronics enclosure using a weatherproof connector.



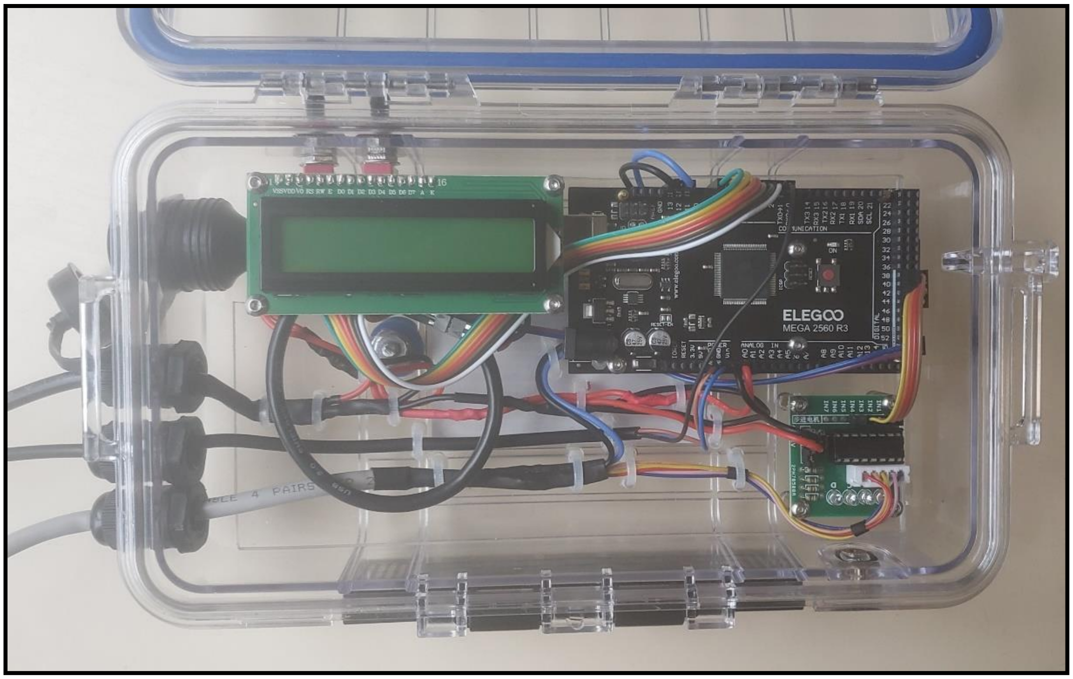
Electronics Enclosure

The governor relies on sensitive electronic components, all of which are housed in a weatherproof enclosure, attached to the side of the generator.

The Arduino Mega 2560 microcontroller performs all the calculations required for the governor system to function. It calculates the current RPM and commands the stepper motor to adjust the throttle as needed. Additionally, the Arduino is configured to output useful debugging info via serial.

The LCD display is controlled by the Arduino. It displays the desired RPM and the current RPM, updating the current RPM every 400 milliseconds. The stepper motor driver takes in a signal from the Arduino and then provides current to the necessary poles in the stepper motor.

Shown below is a photo of the enclosure. It is made of transparent plastic, allowing the LCD display to remain visible from the outside. All power / data cables exit through cable glands; these cables then connect to the throttle assembly and Hall-Effect sensor using weatherproof connectors. There is also a USB passthrough which allows the Arduino to be reprogrammed without removing it from the enclosure. On top of the enclosure there are a pair of switches; one switch controls power to the system, the other enables / disables the stepper motor.



Software Design

This section will detail the Arduino code which was written for this project. It will provide a high-level overview of how the program functions and describe how the control logic functions. This section will also cover some of the programming challenges this project presented and explain how they were addressed.

High-Level Overview

The program is built around a loop which continuously cycles while the Arduino is powered on. During each cycle of this loop, the program performs the following checks.

Throughout this process, the program is also waiting to receive an interrupt signal from the Hall-Effect sensor. Whenever the Arduino receives this signal, it will pause whatever code is currently being executed and record that it has received a pulse from the sensor.

Check if it is time to calculate the RPM

The program calculates the RPM whenever a set number of pulses from the Hall Effect sensor has been received. In other words, it calculates the RPM whenever **`sensorActivations`** has reached the **`rpmCalcInterval`** value.

```
if (sensorActivations >= rpmCalcInterval)
  calcRpm();
```

The **`rpmCalcInterval`** can be adjusted as desired. Decreasing the interval causes the RPM value to be updated at a higher frequency; however, this may lead to increased deviation between the values. Increasing the interval slows the frequency at which RPM values are updated, but it can lead to smoother results with less deviation. In the current design, the **`rpmCalcInterval`** has a value of 1; the program should calculate the RPM each time it receives a pulse from the Hall Effect sensor.

The RPM is calculated in the **`calcRpm()`** method using the following formula:

$$rpm = \frac{\frac{sensorActivations}{timeElapsed} * (1{,}000{,}000\ \mu s) * (60\ \text{seconds})}{numMagnets}$$

The current design of the governor utilizes one magnet, so **`numMagnets`** has a constant value of 1. As such, the formula can be simplified.

$$rpm = \frac{sensorActivations}{timeElapsed} * (1{,}000{,}000\ \mu s) * (60\ \text{seconds})$$

During a typical calculation, **`sensorActivations`** is equal to 1 (since the **`rpmCalcInterval`** is also set to 1), and **`timeElapse`** is equal to roughly 16,667 microseconds.

$$\frac{1}{16{,}667\ \mu s} * (1{,}000{,}000\ \mu s) * (60\ \text{seconds}) = 3600\ rpm$$

Check if the RPM is Within Range

Once the program has an RPM value, it needs to determine if that value is acceptable. If that value is acceptable, the program does nothing. If the value is not acceptable and the engine is running, then the program will adjust the throttle and attempt to correct the error.

```
if (abs(rpmDiff) > rpmPrecision && rpm > minRpm)
  adjustThrottle();
else
  stepperMotor.stop();
```

The program determines if the RPM is acceptable by comparing the absolute value of **`rpmDiff`** to the **`rpmPrecision`**. It then checks to see if the RPM is greater than the **`minRpm`**.

The **`rpmDiff`** is the difference between the desired RPM and the actual RPM. In this project, the desired RPM has a constant value of 3600. So, if the RPM was 3560, then the **`rpmDiff`** would be 40. If the RPM was 3700, the **`rpmDiff`** would be -100.

The **`rpmPrecision`** defines a range above and below the desired RPM which the program considers to be acceptable. In the current program, **`rpmPrecision`** has a constant value of 20. So, if the RPM was 3560, the program would adjust the throttle to correct the RPM. If the RPM was 3615, the program would consider this acceptable, make no adjustments to the throttle, and stop the throttle stepper motor.

Adjusting the Throttle

If the program determines that the current RPM is not acceptable, it will call the **`adjustThrottle()`** method. Each time this method is called, it rotates the throttle stepper motor one step in a specified direction, until the desired number of steps has been reached or reset.

```
void adjustThrottle(){
  // If the stepper switch is turned on.
  if (digitalRead(stepperSwitchPin) == LOW){
    // If the RPM has been updated since the last method call, recalculate the number of stepper motor steps.
    if (throttleRpmUpdated){
      stepsRemaining = calculatePid();  // Set stepsRemaining equal to the output of the PID loop.
      throttleRpmUpdated = false;       // Flagging that the new RPM has been acknowldged.
    }

    // If there are stepper motor steps remaining, and it isn't attempting to open beyond wide-open.
    if ((stepsRemaining > 0) && (directionFlag || digitalRead(limitSwitch) == HIGH)){
      stepperMotor.step(directionFlag); // Stepping in the desired direction.
      stepsRemaining--;                 // Decrementing the remaining steps.
      // Flagging that the stepper motor has been moved from its initial position.
      stepperInitialized = false;
    }
  }
}
```

This section of code will only execute if the **`stepperSwitchPin`** is pulled to ground (LOW). This is achieved by switching on the stepper motor switch, located on the electronics enclosure. Turning the switch off allows the throttle to be locked in place for testing purposes.

If the number of **stepsRemaining** is greater than zero and the program is not attempting to open the throttle beyond full throttle, the program will rotate the stepper motor one step in the desired direction. Afterwards, the number of **stepsRemaining** is decremented by one.

The direction of the stepper motor is controlled by the **directionFlag**. If the **diectionFlag** is true, the stepper will rotate counterclockwise and decrease the throttle. If the **directionFlag** is false, the stepper will rotate clockwise and increase the throttle.

Determining how Much to Adjust the Throttle with PD Control

Whenever the program calculates a new RPM value, if that value is not acceptable, the program will call the **calculatePid()** method. This method uses proportional derivative (PD) calculations to determine how much the throttle should be adjusted.

This method will set the **directionFlag** value, indicating whether the throttle needs to be increased or decreased. It will then return a positive integer value, representing how many steps the throttle needs to be adjusted. The returned positive integer value is the absolute value of the sum of the proportional and derivative calculations.

The proportional calculation is dependent on the current RPM. If the RPM is too low, the value will be positive (increase throttle); if the RPM is too high, the value will be negative (decrease throttle).

The derivative calculation is dependent on the rate of change of the RPM. If the RPM is decreasing, the value will be positive (increase throttle); if the RPM is increasing, the value will be negative (decrease throttle).

	RPM Too High	RPM Too Low
RPM Increasing	**P:** Negative **D:** Negative	**P:** Positive **D:** Negative
RPM Decreasing	**P:** Negative **D:** Positive	**P:** Positive **D:** Positive

Proportional Value

The proportional value, **pidP**, is calculated using the following code:

```
// Calculating Proportional Value: (P-Gain * RPM Difference)
pidP = Kp * rpmDiff;
```

The proportional value is equal to the **rpmDiff** multiplied by a preset gain, **Kp**. The **rpmDiff** is equal to the difference between the desired RPM and the actual RPM. In the current program, the **Kp** gain has a value of 0.013. Consider the following examples:

$$pidP = (0.013) * (240\ rpm) = 3.12\ steps$$

In this example, the **rpmDiff** is 240, meaning that the engine is running at 3360 RPM instead of 3600 RPM. Multiplying 240 by 0.013 results in a value of 3.12 steps. The stepper motor would be commanded to increase the throttle by 3 steps.

$$pidP = (0.013) * (-160\ rpm) = -2.08\ steps$$

In this example, the **rpmDiff** is -160, meaning that the engine is running at 3760 RPM instead of 3600 RPM. Multiplying -160 by 0.013 results in a value of -2.08 steps. The stepper motor would be instructed to decrease the throttle by 2 steps.

Derivative Value

The derivative value, **pidD**, is calculated using the following code:

```
pidD = Kd * ((rpmDiff - rpmDiffPrev) / pidTimeElapsed.read());
```

The proportional value is determined by the rate at which the engine RPM is changing. It is equal to the change in RPM over time (**rpmDiff** - **rpmDiffPrev**) multiplied by a preset gain, **Kd**. In the current program, the **Kp** gain has a value of 0.1, and the **pidTimeElapsed** is typically around 16 milliseconds. Consider the following examples:

$$pidD = (0.1) * \frac{(300\ rpm - 700\ rpm\ )}{17\ ms} = -2.35\ steps$$

This example illustrates a rapid RPM increase. The **rpmDiff** is -300, meaning that the engine is currently running at 3300 RPM, and **rpmDiffPrev** is 700, meaning that the engine was running at 2900 RPM during the prior calculation. This results in a -400 RPM difference.

Between these two RPM calculations, 17 milliseconds have elapsed. Dividing -400 RPM by 17 milliseconds results in a rate of -23.53 RPM/ms. Multiplying this by the 0.1 Kd gain results in a final value of -2.35 steps. The stepper motor would be instructed to decrease the throttle by 2 steps.

Here, the current RPM is below the 3600 RPM setpoint, and proportional control would instruct the throttle to be increased. However, since the RPM is rapidly increasing, the derivative calculation instructs the throttle to decrease. This helps the system minimize RPM overshoot, by preemptively closing the throttle, before the engine has achieved the desired RPM.

$$pidD = (0.1) * \frac{(400\ rpm - (-50\ rpm\ ))}{19\ ms} = 2.37\ steps$$

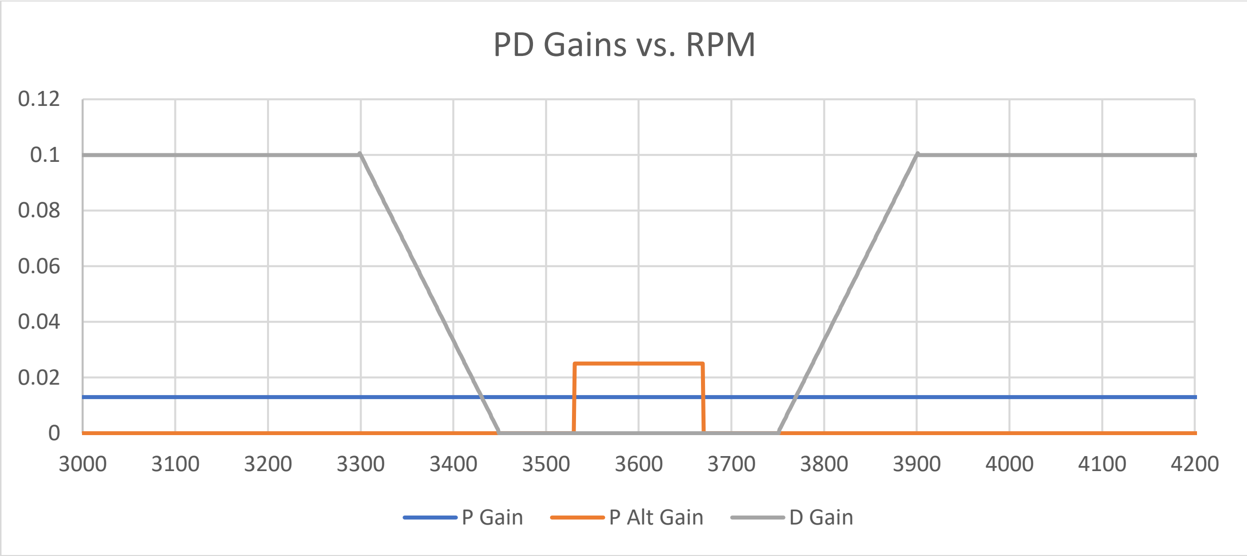
This example illustrates a rapid RPM decrease. The **rpmDiff** is 400, meaning that the engine is currently running at 3200 RPM, and **rpmDiffPrev** is -50, meaning that the engine was running at 3650 RPM during the prior calculation. This results in a 450 RPM difference.

Between these two RPM calculations, 19 milliseconds have elapsed. Dividing 450 RPM by 19 milliseconds results in a rate of 23.68 RPM/ms. Multiplying this by the 0.1 Kd gain results in a final value of 2.37 steps. The stepper motor would be instructed to increase the throttle by 2 steps.

Here, the current RPM is below the 3600 RPM setpoint, and proportional control would instruct the throttle to be increased. Since the RPM is rapidly increasing, the derivative calculation also instructs the throttle to increase.

Dynamic PD Gain Values

Depending on the current engine RPM, the program will modify the PD gains. This is designed to improve the governor performance when the engine is near the 3600 RPM setpoint. The following graph illustrates how the PD gains change depending on the current engine RPM.



Proportional (Kp) Gain

The proportional gain, **`Kp`**, maintains a value of 0.013, regardless of the current engine RPM.

Alternate Proportional (Kp) Gain

When the engine is near the 3600 RPM setpoint, the program will begin supplementing throttle control with alternate proportional calculations. This is accomplished through the following code.

```
// If the RPM is near the target.
if (abs(rpmDiff) < KpAltRange)
  // Calculating alternate PID P value.
  pidPAlt = (KpAlt * rpmDiff);
// Else, the alternate P value is 0;
else
  pidPAlt = 0;
```

These alternate calculations are enabled whenever the **`rpmDiff`** is within the **`KpAltRange`**. In the current version of the program, the **`KpAltRange`** has a value of 70 RPM. So, if the engine was running at 3650 RPM, the **`rpmDiff`** would be -50. This value is within the 70 RPM range, and the alternate calculations would be enabled. If the was running at 3500 RPM, the **`rpmDiff`** would be 100. This value is outside of the 70 RPM range, and the alternate calculations would not be enabled.

The alternate proportional value is equal to the **`rpmDiff`** multiplied by a preset gain, **`KpAlt`**. The **`rpmDiff`** is equal to the difference between the desired RPM and the actual RPM. In the current program, the **`KpAlt`** gain has a value of 0.025. Consider the following example:

$$pidPAlt = (0.025) * (50\ rpm) = 1.25\ steps$$

$$pidP = (0.013) * (50\ rpm) = 0.65\ steps$$

In this example, the **`rpmDiff`** is 50, meaning that the engine is running at 3550 RPM instead of 3600 RPM. Multiplying 50 by 0.025 results in a value of 1.25 steps. The stepper motor would be commanded to increase the throttle by 1 step.

Note that the original proportional calculation, **pidP**, only results in a value 0.65 steps, which would be truncated to zero. With a gain of 0.013, the original proportional calculations are ineffective when the engine RPM is this close to the 3600 RPM setpoint.

Dynamic Derivative (Kd) Gain Decrease

Once the engine RPM is near the setpoint, the derivative calculations tend to overcompensate for small errors and cause the RPM to oscillate above and below the desired RPM. When the engine is near the 3600 RPM setpoint, the program will begin decreasing the influence of the derivative calculations, until disabling them altogether. This is accomplished through the following code.

```
// Calculating Derivative Value: (D-Gain * (Change in RPM / Time Elapsed))
if (abs(rpmDiff) <= 300){
  if (abs(rpmDiff) <= 150)
    pidD = 0;
  else
    pidD = ((Kd * ((abs(rpmDiff) - 150))/150)) * ((rpmDiff - rpmDiffPrev) / pidTimeElapsed.read());
```

This section of code is only activated when the engine is within 300 RPM of the 3600 RPM setpoint. The code decreases the influence of the derivative calculations as the engine grows closer to the 3600 RPM setpoint. This decrease will continue until the RPM is within 150 RPM of the setpoint, at which point **pidD** is set to zero, disabling the derivative calculations entirely. The graph at the start of this section illustrates this behavior.

The decreased derivative gains are calculated using the following formula.

$$decreased\ gain = Kd\frac{(abs(rpmDiff) - 150\ rpm)}{150\ rpm}$$

This formula will decrease the **Kd** gain at a linear rate, over a 150 RPM range. When the absolute value of **rpmDiff** equals 300 RPM, **Kd** will retain its original value. As the **rpmDiff** decreases, the **Kd** gain will also decrease, until the absolute value of **rpmDiff** falls below 150 RPM and the **Kd** gain is set to zero. Consider the following example.

$$decreased\ gain = (0.1)\frac{(250\ rpm - 150\ rpm)}{150\ rpm} = 0.0667$$

$$pidD = (0.0667) * \frac{(250\ rpm - 600\ rpm\ )}{17\ ms} = \ -1.373\ steps$$

Here, the **rpmDiff** is 250, meaning that the current RPM is 3350. This falls within the range of 300 and 150 RPM, so the **Kd** gain will be decreased. 250 RPM minus 150 RPM results in a value of 100 RPM. 100 RPM divided by 150 RPM results in a value of 0.667. The **Kd** value of 0.1 is then multiplied by 0.667, giving a final **Kd** gain of 0.0667.

This new **Kd** gain can now be used with the original derivative equation. As mentioned, the **rpmDiff** is 250. The **rpmDiffPrev** is 600, meaning that the engine was running at 3000 RPM during the prior calculation. This results in a -350 RPM difference.

Between these two RPM calculations, 17 milliseconds have elapsed. Dividing -350 RPM by 17 milliseconds results in a rate of -20.59 RPM/ms. Multiplying this by the 0.0667 **Kd** gain results in a final value of -1.37 steps. The stepper motor would be instructed to increase the throttle by 1 step.

Check if it is Time to Update the LCD

The governor system features a 16 character, 2 row LCD display which outputs the current RPM and the desired RPM. Each time that the program cycles through the main loop, it checks to see if it needs to update the LCD.

```
// If it is time to update the values on the LCD display.
if (displayUpdateTimer.read() >= lcdUpdateInterval){
  updateDisplay();
}
```

The program checks to see if the time elapsed on the **displayUpdateTimer** has reached the **lcdUpdateInterval** value. In this version of the program, the **lcdUpdateInterval** is set to 400, so the values on the LCD will be updated approximately every 400 ms. As long as the time elapsed is over 400 ms, the program will call the **updateDisplay()** method.

```
// Method for updating the LCD display.
void updateDisplay(){
  if (stringIndex == 7){
    stringIndex = 0;
    lcd.setCursor(8,1);
    displayUpdateTimer.start();
  }
  else{
    if (stringIndex == 0){
      stringRpm = String(rpm) + "        ";
    }

    lcd.print(stringRpm.charAt(stringIndex));
    stringIndex++;
  }
}
```

The **updateDisplay()** method updates a single character on the LCD each time that it is called. This is done to speed up the program. Printing an entire line of text to the LCD can take ~15 ms; this is far too slow. This could cause missed Hall Effect sensor signals and other unintended effects in the program. As such, it is preferable to print one character for each cycle of the main loop.

If **stringIndex** is 0, this means that the method has just been called for the first time. It will fetch the current RPM and convert it to a String value, **stringRpm**. It will then print the first character of **stringRpm** and increment the **stringIndex**.

For each successive method call, a new character will be updated on the LCD, until the **stringIndex** equals 7. This means that the **stringRpm** has been completely printed. The method then sets the

stringIndex back to zero, sets the LCD cursor to the first character of the RPM section, and resets the **displayUpdateTimer**.

Check if it is Time to Output to Serial

The program prints debugging info to the serial output. Each time that the program cycles through the main loop, it checks to see if it needs to update the LCD.

```
// If it is time to print the serial output.
if (serialOutputTimer.read() >= serialUpdateInterval){
  serialOutput();
}
```

The program checks to see if the time elapsed on the **serialOutputTimer** has reached the **serialUpdateInterval** value. In this version of the program, the **serialUpdateInterval** is set to 50, so the values on the LCD will be updated approximately every 50 ms.

```
// Method for printing data logging info to the serial output.
void serialOutput(){
  serialOutputTimer.start();
  // Print time elapsed.
  Serial.print(millis());
  Serial.print(',');
  // Print current RPM.
  Serial.print(rpm);
  // Print the number of stepper steps remaining.
  Serial.print(',');
  ...
```

The **serialOutput()** method outputs debugging data in the form of a comma-separated list. Each time the method is called, it prints out a single line containing the total time elapsed in milliseconds, the current RPM, the number of stepper motor steps remaining, the current PD calculations, the change in RPM since the last measurement, and the total number of steps commanded by the PD calculations.

The serial communication is configured for 115200 baud, so there is no need to print this info one character at a time. This line of output can be printed all at once without any noticeable program slowdown.

Check if the Engine is Still Running

The program keeps track of whether the engine is running. Each time that the program cycles through the main loop, it checks to see if the engine has just started or if has just stopped.

```
// If the rpm has risen above the minRpm. (The engine has started.)
if (rpm > minRpm)
  // Flagging that the engine is running.
  engineRunning = true;
```

If the RPM is above the **minRpm**, the program will mark that the engine is running. In the current version of the program, the **minRpm** is set to 300. Whenever the RPM is below 300 and the **engineRunning** variable is set to false, the governor system will not attempt to adjust the throttle.

```
// If the engine has stopped.
if (timeElapsed.read() / 1000 > stallTimeout)
  // Setting the rpm to 0.
  rpm = 0;
```

If the governor has waited for longer than the **stallTimeout** without receiving a signal from the Hall Effect sensor, it will set the current RPM to zero. In the current version of the program, the **stallTimout** is set to 200, which means the engine is considered stalled if there has not been a Hall-Effect sensor signal in 200 ms. Without this code, the program would continue to display the last calculated RPM once the engine comes to a stop.

```
// If the engine has not been started or it was running and has stopped.
if ((timeElapsed.read() / 1000 > stallTimeout && engineRunning) || (!stepperInitialized && !engineRunning)){
  // Initializig the stepper motor, preparing for the engine to be restarted.
  initializeStepper();
}
```

This section of code checks to see if the engine has just stopped or if the governor system has just been powered on. In either case, it will call the **initializeStepper()** method, which places the throttle in its default potion. In the current version of the program, the throttle will be placed 850 steps away from full throttle.

Experiment Process

These experiments were designed to test the performance characteristics of the mechanical governor and the microcontroller governor. These tests measure how each governor performs at a constant engine load, as well as a changing engine load, measuring RPM deviation, RPM range, etc. The testing environment is pictured below.

Five different experiments were conducted, each testing the engine under different conditions: 1. No Engine Load, 2. Constant Engine Load, 3. Abrupt Load Decrease, 4. Abrupt Load Increase (Initially No Load), 5. Abrupt Load Increase (Initially Constant Load). Ten trials were completed for each test with each governor. These tests should provide insight into each governor's performance over a wide range of engine operating conditions.

During each experiment, results were gathered via a laptop receiving serial data from the Arduino. This data was received every 50 milliseconds. This data is in the form of comma separated values and was later imported into Microsoft Excel for analysis. The data includes the total time elapsed in milliseconds, the current RPM, the number of stepper motor steps remaining, the current PD calculations, the change in RPM since the last measurement, and the total number of steps commanded by the PD calculations.

During the experiments, an electrical load was placed on the generator. This was done with a 1500-watt space heater and an 1800 abrasive cut-off machine. The space heater was used as a constant load; in practice, it typically pulled ~1460 watts. The abrasive cutoff machine was used as an abrupt load. It was difficult measure its initial startup draw; it likely pulled somewhere between 2000 and 3000 watts at startup. Once it was running, it typically pulled ~750 watts.

These experiments were conducted between March 31 and April 5, 2023. The ambient temperature varied between 76 and 98 degrees Fahrenheit. At the start of each day of testing, the generator was run for several minutes before the experiments began, ensuring that it had reached the proper operating temperature. During each experiment trial, the ambient temperature and cylinder head temperature head temperature were recorded.

Before these experiments were conducted, general maintenance was performed on the generator. A new air filter was installed, a new NGK BR6HS was gapped and installed. Two oil changes were performed, ensuring that any old oil had been flushed from the engine; during the experiments, the engine was using Shell Rotella T4, conventional 15W40 motor oil. The engine was run on regular 87 octane gasoline, with up to 10% ethanol.

Experiment Data Explanation

The following sections will present data collected from the experiments. Each section presents a set of tables, summarizing the data from that experiment. This data is taken from 10 trials. Some of the data represents an average from those 10 trials, some of the data represents the maximum and minimum values taken from those trials. When data includes “TRIMMEAN: 20%” in the title, this data is an average which excludes the highest and lowest values from the 10 trials.

Following is an explanation of some of the specialized values included in the data.

Dev from Setpoint

Similar to standard deviation, this value represents the RPM deviation from the 3600 RPM setpoint. This is the average of how far each RPM value deviates from the 3600 RPM setpoint.

Startup Time

This value represents how long, in milliseconds, it takes the engine to start up and reach the desired RPM. This is the time elapsed from when the engine first sends a non-zero RPM signal to when the engine first achieves an RPM that is within 100 RPM of the 3600 RPM setpoint.

Overshoot Time

This value represents how long, in milliseconds, the engine RPM is more than 100 RPM over the 3600 RPM setpoint. This typically occurs when the engine first starts up or when the load suddenly decreases. This is the time elapsed while the engine remains running at least 100 RPM over the 3600 RPM setpoint.

Undershoot Time

This value represents how long, in milliseconds, the engine RPM is more than 100 RPM below the 3600 RPM setpoint. This typically occurs when the load suddenly increases. This is the time elapsed while the engine remains running at least 100 RPM below the 3600 RPM setpoint.

Experiment 1: No Load

This experiment tests the performance of the engine at idle, with no load applied. Neither the space heater nor abrasive cut-off machine were used in this experiment. At is started 10 seconds into the experiment, the engine is started. The engine is then left to run for 60 seconds. At 70 seconds, the engine is shut off. At 80 seconds, the experiment ends.

Control: No Throttle Adjustment

For this experiment, no governor was used. The engine was initially run with the microcontroller governor. Once the engine stabilized near 3600 RPM, the governor was disabled, and the throttle was locked in place. After locking the throttle in place, the engine was shut off.

Ten trials of the experiment were then run with the throttle locked in place.

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3613.906226	32.67058817	37.85657553	
Min RPM	**Max RPM**	**Range**	
3506.961	3732.109	225.148	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3666.905	2785	#DIV/0!	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3614.758541	32.51143077	36.82308208	
Min RPM	**Max RPM**	**Range**	
3507.25	3727.155	227.3425	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3664.55375	2737.5	#NUM!	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3564.857553	28.82616373	27.63867665	
Min RPM	**Max RPM**	**Range**	
3423.09	3690.94	158.67	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3624.06	2450	0	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3656.13638	37.78827184	56.34242197	
Min RPM	**Max RPM**	**Range**	
3588.52	3812.91	274.07	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3728.56	3500	0	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	71.7	179.9	31.41%
60 RPM	47	122.5	21.16%
80 RPM	15.1	45.4	7.55%
100 RPM	3.6	13.6	2.15%
120 RPM	0.5	3.1	0.45%
140 RPM	0.3	0.5	0.10%
160 RPM	0.1	0.2	0.04%
200 RPM	0	0.1	0.01%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	54.5	165.5	27.47%
60 RPM	33.875	108.625	17.79%
80 RPM	10.25	37.625	5.98%
100 RPM	1.875	10.25	1.51%
120 RPM	0.25	2.5	0.34%
140 RPM	0.125	0.375	0.06%
160 RPM	0	0	0.00%
200 RPM	0	0	0.00%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Mechanical Governor

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3623.258367	33.38791005	32.65228215	
Min RPM	**Max RPM**	**Range**	
3423.649	3823.963	400.314	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3866.228	700	85	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3623.146019	33.20737197	32.62889513	
Min RPM	**Max RPM**	**Range**	
3417.9525	3822.97125	396.28	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3863.89375	756.25	81.25	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3618.964732	30.77506683	29.93099875	
Min RPM	**Max RPM**	**Range**	
3360.97	3768.84	308.35	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3766.01	100	50	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3628.450787	37.44505793	35.56066167	
Min RPM	**Max RPM**	**Range**	
3531.9	3887.02	524.55	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3985.12	850	150	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	54.4	108.3	20.31%
60 RPM	1.8	39.2	5.12%
80 RPM	1.4	14.1	1.94%
100 RPM	1.4	11.8	1.65%
120 RPM	1.2	8.9	1.26%
140 RPM	1.2	6	0.90%
160 RPM	1.2	4.1	0.66%
200 RPM	0.4	1.9	0.29%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	55.25	109	20.51%
60 RPM	1.625	39	5.07%
80 RPM	1.25	13.75	1.87%
100 RPM	1.25	11.625	1.61%
120 RPM	1	8.875	1.23%
140 RPM	1	5.75	0.84%
160 RPM	1	3.625	0.58%
200 RPM	0.25	1.625	0.23%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Microcontroller Governor

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3595.865453	47.1880583	36.33313858	
Min RPM	**Max RPM**	**Range**	
3406.712	3765.55	358.838	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
4123.878	1005	1060	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3595.783235	47.17078248	36.40963951	
Min RPM	**Max RPM**	**Range**	
3410.8375	3755.6775	344.93125	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
4049.32875	1093.75	1087.5	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3593.187378	40.89195619	31.84474407	
Min RPM	**Max RPM**	**Range**	
3309.8	3711.95	241.33	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3927.73	100	700	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3599.201273	53.62236698	40.20952559	
Min RPM	**Max RPM**	**Range**	
3470.62	3898.13	587.6	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
4916.42	1200	1200	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	119.4	91	26.27%
60 RPM	84.3	65.8	18.74%
80 RPM	51.3	22.6	9.23%
100 RPM	17.4	6.9	3.03%
120 RPM	8.4	2.8	1.40%
140 RPM	5.2	1.4	0.82%
160 RPM	3.1	0.6	0.46%
200 RPM	1.3	0.2	0.19%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	118.625	90.25	26.08%
60 RPM	84.125	66	18.74%
80 RPM	50.75	22.625	9.16%
100 RPM	17	6.25	2.90%
120 RPM	7.875	2.375	1.28%
140 RPM	4.5	1.125	0.70%
160 RPM	2.25	0.375	0.33%
200 RPM	0.625	0.125	0.09%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Experiment 2: Constant Load

This experiment tests the performance of the engine at a constant load. The space heater was used to provide a load of 1500 watts. Prior to the experiment, the space heater is plugged in and switched on. At 10 seconds into the experiment, the engine is started. The engine is then left to run for 60 seconds. At 70 seconds, the engine is shut off. At 80 seconds, the experiment ends.

Control: No Throttle Adjustment

For this experiment, no governor was used. The engine was initially run with the microcontroller governor installed and the heater running. Once the engine stabilized near 3600 RPM, the governor was disabled, and the throttle was locked in place. After locking the throttle in place, the engine was shut off.

Ten trials of the experiment were then run with the throttle locked in place.

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3600.099973	28.69888217	27.58760549	
Min RPM	**Max RPM**	**Range**	
3509.778	3703.749	193.971	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3663.019	1110	#DIV/0!	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3601.537903	28.49370707	26.74118914	
Min RPM	**Max RPM**	**Range**	
3512.66875	3704.8675	193.02125	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3661.09875	1106.25	#NUM!	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3564.453795	26.13321288	21.96123596	
Min RPM	**Max RPM**	**Range**	
3457.02	3652.3	150.48	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3626.69	1000	0	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3624.242709	32.90595223	39.98530587	
Min RPM	**Max RPM**	**Range**	
3539.41	3746.25	245.06	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3714.71	1250	0	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	68.6	57.5	15.74%
60 RPM	42.6	23.1	8.20%
80 RPM	13.9	3.2	2.13%
100 RPM	3.1	0.9	0.50%
120 RPM	0.5	0.5	0.12%
140 RPM	0.1	0.1	0.02%
160 RPM	0	0	0.00%
200 RPM	0	0	0.00%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	48.25	50.625	12.34%
60 RPM	28.375	18.375	5.84%
80 RPM	6.625	2.875	1.19%
100 RPM	1.25	0.75	0.25%
120 RPM	0.25	0.375	0.08%
140 RPM	0	0	0.00%
160 RPM	0	0	0.00%
200 RPM	0	0	0.00%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Mechanical Governor

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3591.036511	28.59153512	22.62482272	
Min RPM	**Max RPM**	**Range**	
3421.415	3828.879	407.464	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3808.065	810	85	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3591.018532	28.38395039	22.6023221	
Min RPM	**Max RPM**	**Range**	
3419.13625	3826.50625	405.90125	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3804.89125	787.5	81.25	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3588.930175	24.1934204	21.2413608	
Min RPM	**Max RPM**	**Range**	
3357.21	3663.9	240.81	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3713.79	700	50	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3593.286679	34.65032771	24.18828964	
Min RPM	**Max RPM**	**Range**	
3503.85	4012.84	586.62	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3927.73	1100	150	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	60.9	6.1	8.36%
60 RPM	3.3	5.5	1.10%
80 RPM	2.2	4.2	0.80%
100 RPM	1.7	3.5	0.65%
120 RPM	1.2	3.3	0.56%
140 RPM	1.2	3	0.52%
160 RPM	1.1	2.8	0.49%
200 RPM	0.5	1.8	0.29%
300 RPM	0	0.2	0.02%
400 RPM	0	0.1	0.01%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	60.125	6	8.26%
60 RPM	3.25	5.375	1.08%
80 RPM	1.875	4.125	0.75%
100 RPM	1.375	3.375	0.59%
120 RPM	1	3.125	0.51%
140 RPM	1	2.75	0.47%
160 RPM	0.875	2.5	0.42%
200 RPM	0.375	1.625	0.25%
300 RPM	0	0.125	0.02%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Microcontroller Governor

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3601.420891	267.3100685	42.77790762	
Min RPM	**Max RPM**	**Range**	
3429.035	8606.601	5177.566	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3889.187	1015	650	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3591.168421	39.04763626	31.22240793	
Min RPM	**Max RPM**	**Range**	
3437.52375	3765.92875	343.395	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3887.7525	1112.5	631.25	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3577.908577	33.27409705	26.28214732	
Min RPM	**Max RPM**	**Range**	
3309.07	3673.77	226.29	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3810.01	100	150	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3706.952971	2327.445498	151.7176654	
Min RPM	**Max RPM**	**Range**	
3481.09	52264.81	48802.21	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3979.84	1150	1300	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	116.4	36.8	19.13%
60 RPM	82.7	18.2	12.60%
80 RPM	40.6	5.5	5.76%
100 RPM	11.1	3.1	1.77%
120 RPM	2.2	2.3	0.56%
140 RPM	1.3	1.7	0.37%
160 RPM	0.8	0.7	0.19%
200 RPM	0.5	0.6	0.14%
300 RPM	0	0.3	0.04%
400 RPM	0	0.2	0.02%
500 RPM	0	0.2	0.02%
600 RPM	0	0.2	0.02%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	111.5	34.375	18.21%
60 RPM	77.125	15.875	11.61%
80 RPM	37.25	3.125	5.04%
100 RPM	9.5	1.75	1.40%
120 RPM	1.75	1.5	0.41%
140 RPM	1	1.5	0.31%
160 RPM	0.375	0.5	0.11%
200 RPM	0.25	0.375	0.08%
300 RPM	0	0.125	0.02%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Experiment 3: Abrupt Load Decrease

This experiment tests the performance of the engine during an abrupt load decrease. The space heater was used to provide an initial load of 1500 watts. Prior to the experiment, the space heater is plugged in and switched on. At 10 seconds into the experiment, the engine is started. At 40 seconds, the space heater is switched off, and the engine continues running. At 70 seconds, the engine is shut off. At 80 seconds, the experiment ends.

Mechanical Governor

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3604.124026	34.96268966	26.6819563	
Min RPM	**Max RPM**	**Range**	
3449.891	3829.066	379.175	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3816.028	1010	72.22222222	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3604.33096	34.91150942	26.58625156	
Min RPM	**Max RPM**	**Range**	
3453.03	3820.85875	365.32875	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3810.87625	812.5	72.22222222	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3598.196392	32.3806697	24.37926342	
Min RPM	**Max RPM**	**Range**	
3360.97	3766.01	276.83	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3724.86	650	50	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3608.396192	37.95415156	29.75028714	
Min RPM	**Max RPM**	**Range**	
3513.7	3957.78	592.29	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3948.41	2950	150	

Average Before Load Change			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 35 sec*
3588.102648	28.72197952	23.17124585	
Min RPM	**Max RPM**	**Range**	
3487.308	3788.098	300.79	
Average During Load Change			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*35 sec to 45 sec*
3610.141572	37.69479889	29.61589055	
Min RPM	**Max RPM**	**Range**	
3513.823	3803.597	289.774	
Undershoot Time (ms)	**Overshoot Time (ms)**		
#DIV/0!	50		
Average After Load Change			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*45 sec to 60 sec*
3616.155382	32.11806248	28.24546179	
Min RPM	**Max RPM**	**Range**	
3478.748	3775.515	296.767	

Average Before Load Change (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 35 sec*
3588.305718	28.86933965	23.23134136	
Min RPM	**Max RPM**	**Range**	
3495.96125	3800.2025	301.545	
Average During Load Change (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*35 sec to 45 sec*
3610.305914	37.68163874	29.63157338	
Min RPM	**Max RPM**	**Range**	
3521.0525	3792.775	285.975	
Undershoot Time (ms)	**Overshoot Time (ms)**		
#NUM!	50		
Average After Load Change (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*45 sec to 60 sec*
3616.236773	32.2543114	28.01489618	
Min RPM	**Max RPM**	**Range**	
3484.8	3773.8375	291.07375	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	58	50.1	13.50%
60 RPM	3	24.5	3.43%
80 RPM	2	10.7	1.59%
100 RPM	1.4	8.2	1.20%
120 RPM	0.8	7.3	1.01%
140 RPM	0.7	5.4	0.76%
160 RPM	0.7	4.2	0.61%
200 RPM	0.3	1.2	0.19%
300 RPM	0	0.1	0.01%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	57.75	57.75	14.42%
60 RPM	2.875	2.875	0.72%
80 RPM	2	2	0.50%
100 RPM	1.375	1.375	0.34%
120 RPM	0.75	0.75	0.19%
140 RPM	0.625	0.625	0.16%
160 RPM	0.625	0.625	0.16%
200 RPM	0.125	0.125	0.03%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Microcontroller Governor

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3593.468886	55.79376167	40.35171286	
Min RPM	**Max RPM**	**Range**	
3348.066	3904.763	556.697	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3865.345	1090	815	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3594.030179	55.9153824	40.44712859	
Min RPM	**Max RPM**	**Range**	
3342.8775	3875.16875	536.48875	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3861.8825	1087.5	812.5	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3584.627441	45.56922697	33.52397004	
Min RPM	**Max RPM**	**Range**	
3309.07	3799.39	405.79	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3797.47	1050	450	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3597.819988	65.04533054	46.41612984	
Min RPM	**Max RPM**	**Range**	
3428.57	4246.89	869.27	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3960.92	1150	1200	

Average Before Load Change			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 35 sec*
3589.27606	39.93517362	31.72183389	
Min RPM	**Max RPM**	**Range**	
3447.195	3717.48	270.285	
Average During Load Change			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*35 sec to 45 sec*
3601.599886	75.23545737	53.26130846	
Min RPM	**Max RPM**	**Range**	
3410.428	3904.763	494.335	
Undershoot Time (ms)	**Overshoot Time (ms)**		
377.7777778	630		
Average After Load Change			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*45 sec to 60 sec*
3592.327312	51.90299745	40.3322691	
Min RPM	**Max RPM**	**Range**	
3407.329	3722.083	314.754	

Average Before Load Change (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 35 sec*
3589.607259	39.88554554	31.71556894	
Min RPM	**Max RPM**	**Range**	
3457.41125	3708.88	266.8025	
Average During Load Change (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*35 sec to 45 sec*
3601.604447	74.24662056	52.38727612	
Min RPM	**Max RPM**	**Range**	
3412.745	3875.16875	470.78875	
Undershoot Time (ms)	**Overshoot Time (ms)**		
377.7777778	656.25		
Average After Load Change (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*45 sec to 60 sec*
3593.049007	51.63284352	40.1847093	
Min RPM	**Max RPM**	**Range**	
3410.39125	3719.255	308.955	

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	138.5	88.8	28.38%
60 RPM	102.7	65.7	21.02%
80 RPM	61	32.3	11.65%
100 RPM	30.7	20.7	6.42%
120 RPM	17.2	16.9	4.26%
140 RPM	11.6	11.9	2.93%
160 RPM	8	9.2	2.15%
200 RPM	3.3	4	0.91%
300 RPM	0	0.3	0.04%
400 RPM	0	0.2	0.02%
500 RPM	0	0.1	0.01%
600 RPM	0	0.1	0.01%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	135.25	135.25	33.77%
60 RPM	99.25	99.25	24.78%
80 RPM	58.125	58.125	14.51%
100 RPM	28.75	28.75	7.18%
120 RPM	16.75	16.75	4.18%
140 RPM	11.75	11.75	2.93%
160 RPM	8	8	2.00%
200 RPM	3	3	0.75%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Experiment 4: Abrupt Load Increase (Initially No Load)

This experiment tests the performance of the engine during an abrupt load increase. This is accomplished by using the abrasive cut-off machine. At 10 seconds into the experiment, the engine is started. At 40 seconds, the cut-off machine is powered on. At 50 seconds the cut-off machine is powered off. At 70 seconds, the engine is shut off. At 80 seconds, the experiment ends.

Mechanical Governor

Average of Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3621.392909	36.40855016	32.64970537	
Min RPM	**Max RPM**	**Range**	
3355.882	3848.323	492.441	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3806.079	735	70	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3621.261028	35.93307733	32.47378433	
Min RPM	**Max RPM**	**Range**	
3353.0775	3845.3625	493.80375	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3806.28125	737.5	62.5	

Min Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3618.867303	33.94419031	30.05661673	
Min RPM	**Max RPM**	**Range**	
3313.45	3793.63	449.15	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3758.46	650	50	

Max Results			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 60 sec*
3624.973558	42.67669272	36.6501623	
Min RPM	**Max RPM**	**Range**	
3420.75	3926.7	524.83	
Max Startup RPM	**Startup Time (ms)**	**Overshoot Time (ms)**	*0 sec to 20 sec*
3852.08	800	150	

Average Before Load Change			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 35 sec*
3626.987807	33.95464033	35.38513621	
Min RPM	**Max RPM**	**Range**	
3495.642	3813.442	317.8	
Average - Load Start			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*35 sec to 45 sec*
3614.558323	43.47048406	32.2049005	
Min RPM	**Max RPM**	**Range**	
3355.882	3768.222	412.34	
Undershoot Time (ms)	**Overshoot Time (ms)**		
190	50		
Average - Load Stop			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*45 sec to 55 sec*
3619.635915	32.67329987	29.63389552	
Min RPM	**Max RPM**	**Range**	
3513.294	3777.776	264.482	
Undershoot Time (ms)	**Overshoot Time (ms)**		
#DIV/0!	62.5		

Average Before Load Change (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*20 sec to 35 sec*
3626.978796	34.03053195	35.202201	
Min RPM	**Max RPM**	**Range**	
3498.71125	3807.3325	320.91625	
Average - Load Start (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*35 sec to 45 sec*
3614.775224	42.74785594	31.57356343	
Min RPM	**Max RPM**	**Range**	
3353.0775	3767.4325	417.85625	
Undershoot Time (ms)	**Overshoot Time (ms)**		
187.5	50		
Average - Load Stop (TRIMMEAN: 20%)			
Avg RPM	**Std Dev**	**Dev from Setpoint**	*45 sec to 55 sec*
3619.85829	32.80600059	29.71018035	
Min RPM	**Max RPM**	**Range**	
3520.87375	3776.38375	255.51	
Undershoot Time (ms)	**Overshoot Time (ms)**		
#NUM!	62.5		

Average Number of Readings Out-of-Range			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	53.5	110.5	20.47%
60 RPM	6.3	43.8	6.25%
80 RPM	5.7	13.1	2.35%
100 RPM	5	11.6	2.07%
120 RPM	4.1	8.7	1.60%
140 RPM	3.4	5.7	1.14%
160 RPM	3.3	3.6	0.86%
200 RPM	1.7	1.7	0.42%
300 RPM	0	0.1	0.01%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	**Below**	**Above**	**Percent of Total**
50 RPM	52.75	110.625	20.40%
60 RPM	6.125	44.625	6.34%
80 RPM	5.75	12.875	2.33%
100 RPM	5.125	11.25	2.04%
120 RPM	4	8	1.50%
140 RPM	3.375	5.25	1.08%
160 RPM	3.25	3.125	0.80%
200 RPM	1.75	1.625	0.42%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Microcontroller Governor

Average of Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3591.251318	78.85041627	51.01983021	
Min RPM	Max RPM	Range	
3116.703	3880.832	764.129	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
4106.042	980	1150	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3591.374443	78.62774414	51.09879682	
Min RPM	Max RPM	Range	
3121.31875	3878.8975	764.6025	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
4018.07125	1068.75	1193.75	

Min Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3586.755506	72.51614917	46.11093633	
Min RPM	Max RPM	Range	
3034.59	3810.98	687.22	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3939.08	100	650	

Max Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3594.762135	86.96606045	55.29699126	
Min RPM	Max RPM	Range	
3161.89	3966.16	837.25	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
4976.77	1150	1300	

Average Before Load Change			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 35 sec*
3593.938266	49.68488371	38.51293023	
Min RPM	Max RPM	Range	
3419.114	3744.438	325.324	
Average - Load Start			
Avg RPM	Std Dev	Dev from Setpoint	*35 sec to 45 sec*
3579.196169	125.7263391	80.79483582	
Min RPM	Max RPM	Range	
3116.703	3880.139	763.436	
Undershoot Time (ms)	Overshoot Time (ms)		
810	870		
Average - Load Stop			
Avg RPM	Std Dev	Dev from Setpoint	*45 sec to 55 sec*
3597.55206	53.82580681	42.69618905	
Min RPM	Max RPM	Range	
3429.941	3725.672	295.731	
Undershoot Time (ms)	Overshoot Time (ms)		
200	187.5		

Average Before Load Change (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 35 sec*
3593.804751	49.56909729	38.42489203	
Min RPM	Max RPM	Range	
3423.30875	3737.72	325.25	
Average - Load Start (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*35 sec to 45 sec*
3578.286306	124.8781846	80.48283582	
Min RPM	Max RPM	Range	
3121.31875	3878.03125	763.73625	
Undershoot Time (ms)	Overshoot Time (ms)		
806.25	875		
Average - Load Stop (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*45 sec to 55 sec*
3597.267711	53.18833246	42.0657898	
Min RPM	Max RPM	Range	
3436.1175	3722.26625	291.3425	
Undershoot Time (ms)	Overshoot Time (ms)		
200	187.5		

Average Number of Readings Out-of-Range			
Discrepancy	Below	Above	Percent of Total
50 RPM	157.4	123.8	35.11%
60 RPM	123.4	98.6	27.72%
80 RPM	83.9	53.4	17.14%
100 RPM	50.3	31.8	10.25%
120 RPM	35.6	22.9	7.30%
140 RPM	29	16.1	5.63%
160 RPM	23.8	13.4	4.64%
200 RPM	16.7	6.9	2.95%
300 RPM	10.1	0.3	1.30%
400 RPM	5.8	0	0.72%
500 RPM	0.3	0	0.04%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	Below	Above	Percent of Total
50 RPM	156.5	125.25	35.17%
60 RPM	123	99.375	27.76%
80 RPM	83.625	53.75	17.15%
100 RPM	49.5	30.875	10.03%
120 RPM	35.375	22	7.16%
140 RPM	28.625	16	5.57%
160 RPM	23.5	13.375	4.60%
200 RPM	16.375	7	2.92%
300 RPM	10.125	0.25	1.30%
400 RPM	5.75	0	0.72%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Experiment 5: Abrupt Load Increase (Initially Constant Load)

This experiment tests the performance of the engine during an abrupt load increase. This is accomplished by using the abrasive cut-off machine and the space heater. Prior to the experiment, the space heater is plugged in and switched on. At 10 seconds into the experiment, the engine is started. At 40 seconds, the cut-off machine is powered on. At 50 seconds the cut-off machine is powered off. At 70 seconds, the engine is shut off. At 80 seconds, the experiment ends.

Mechanical Governor

Average of Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3584.925905	32.75827472	26.46512609	
Min RPM	Max RPM	Range	
3354.257	3775.106	420.849	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3783.134	700	83.33333333	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3584.824622	32.73308754	26.40999532	
Min RPM	Max RPM	Range	
3354.59625	3781.12	429.31	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3778.70125	725	83.33333333	

Min Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3582.592871	29.12169641	25.19189763	
Min RPM	Max RPM	Range	
3327.42	3675.57	297.19	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3714.71	100	50	

Max Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3588.069201	36.5963505	28.17940075	
Min RPM	Max RPM	Range	
3378.38	3826.53	476.82	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3887.02	1100	100	

Average Before Load Change			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 35 sec*
3590.907033	26.10521268	22.15285382	
Min RPM	Max RPM	Range	
3485.319	3691.095	205.776	
Average - Load Start			
Avg RPM	Std Dev	Dev from Setpoint	*35 sec to 45 sec*
3572.347035	39.19747976	33.88760199	
Min RPM	Max RPM	Range	
3358.289	3693.23	334.941	
Undershoot Time (ms)	Overshoot Time (ms)		
195	#DIV/0!		
Average - Load Stop			
Avg RPM	Std Dev	Dev from Setpoint	*45 sec to 55 sec*
3584.749294	30.75023939	26.54731343	
Min RPM	Max RPM	Range	
3490.177	3720.293	230.116	
Undershoot Time (ms)	Overshoot Time (ms)		
#DIV/0!	75		

Average Before Load Change (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 35 sec*
3590.785544	26.00428066	22.30250831	
Min RPM	Max RPM	Range	
3493.67375	3683.71125	199.135	
Average - Load Start (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*35 sec to 45 sec*
3572.356499	39.34319314	33.71128731	
Min RPM	Max RPM	Range	
3357.14875	3693.5525	334.55	
Undershoot Time (ms)	Overshoot Time (ms)		
187.5	#NUM!		
Average - Load Stop (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*45 sec to 55 sec*
3585.002612	30.65382941	26.44612562	
Min RPM	Max RPM	Range	
3502.29125	3717.945	225.58375	
Undershoot Time (ms)	Overshoot Time (ms)		
#NUM!	75		

Average Number of Readings Out-of-Range			
Discrepancy	Below	Above	Percent of Total
50 RPM	125	6.2	16.38%
60 RPM	26.6	5.7	4.03%
80 RPM	10.7	4.2	1.86%
100 RPM	7.1	3.9	1.37%
120 RPM	5.1	3.4	1.06%
140 RPM	4.8	2	0.85%
160 RPM	4.2	1.9	0.76%
200 RPM	2.5	0.6	0.39%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	Below	Above	Percent of Total
50 RPM	126	6.375	16.53%
60 RPM	25.625	5.75	3.92%
80 RPM	10.5	4.25	1.84%
100 RPM	7.125	3.875	1.37%
120 RPM	5.125	3.25	1.05%
140 RPM	4.875	1.625	0.81%
160 RPM	4.25	1.625	0.73%
200 RPM	2.5	0.5	0.37%
300 RPM	0	0	0.00%
400 RPM	0	0	0.00%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Microcontroller Governor

Average of Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3590.146853	69.39110553	41.97250062	
Min RPM	Max RPM	Range	
3125.638	3865.506	739.868	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3851.939	1060	510	

Average of Results (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3590.238026	69.22807848	41.82801186	
Min RPM	Max RPM	Range	
3123.9275	3862.77125	737.30625	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3855.6975	1062.5	493.75	

Min Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3585.042297	66.07216598	39.15411985	
Min RPM	Max RPM	Range	
3101.74	3760.34	609.42	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3772.64	1000	250	

Max Results			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 60 sec*
3594.522022	74.01426142	45.94679151	
Min RPM	Max RPM	Range	
3163.22	3992.55	890.81	
Max Startup RPM	Startup Time (ms)	Overshoot Time (ms)	*0 sec to 20 sec*
3901.17	1100	900	

Average Before Load Change			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 35 sec*
3591.349146	37.0248312	29.89272757	
Min RPM	Max RPM	Range	
3477.175	3696.502	219.327	
Average - Load Start			
Avg RPM	Std Dev	Dev from Setpoint	*35 sec to 45 sec*
3579.8351	116.4943261	70.99444279	
Min RPM	Max RPM	Range	
3125.638	3837.336	711.698	
Undershoot Time (ms)	Overshoot Time (ms)		
885	625		
Average - Load Stop			
Avg RPM	Std Dev	Dev from Setpoint	*45 sec to 55 sec*
3597.818507	50.07379632	37.23135323	
Min RPM	Max RPM	Range	
3490.037	3804.437	314.4	
Undershoot Time (ms)	Overshoot Time (ms)		
50	300		

Average Before Load Change (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*20 sec to 35 sec*
3591.141204	36.62670552	30.04412375	
Min RPM	Max RPM	Range	
3493.3275	3694.9325	208.84875	
Average - Load Start (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*35 sec to 45 sec*
3579.832096	116.4856428	70.85608831	
Min RPM	Max RPM	Range	
3123.9275	3835.135	712.78625	
Undershoot Time (ms)	Overshoot Time (ms)		
887.5	650		
Average - Load Stop (TRIMMEAN: 20%)			
Avg RPM	Std Dev	Dev from Setpoint	*45 sec to 55 sec*
3597.912662	50.95211471	37.40224502	
Min RPM	Max RPM	Range	
3491.74625	3789.71125	299.87375	
Undershoot Time (ms)	Overshoot Time (ms)		
50	300		

Average Number of Readings Out-of-Range			
Discrepancy	Below	Above	Percent of Total
50 RPM	119.7	72.2	23.96%
60 RPM	90.6	55.3	18.21%
80 RPM	53.5	36.8	11.27%
100 RPM	26.7	26.5	6.64%
120 RPM	20.5	21.7	5.27%
140 RPM	17.9	13.6	3.93%
160 RPM	16.7	8.4	3.13%
200 RPM	14.8	3.2	2.25%
300 RPM	10.1	0.4	1.31%
400 RPM	5.1	0	0.64%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

Average Number of Readings Out-of-Range (TRIMMEAN: 20%)			
Discrepancy	Below	Above	Percent of Total
50 RPM	120	72.5	24.03%
60 RPM	91	55.125	18.24%
80 RPM	54	37.375	11.41%
100 RPM	26	27.25	6.65%
120 RPM	20	22.75	5.34%
140 RPM	17.5	14	3.93%
160 RPM	16.625	8.5	3.14%
200 RPM	14.75	3	2.22%
300 RPM	10.125	0.375	1.31%
400 RPM	5.25	0	0.66%
500 RPM	0	0	0.00%
600 RPM	0	0	0.00%

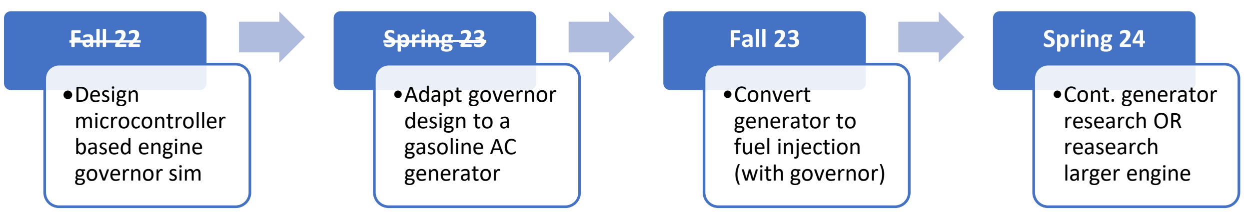
Conclusion

This project has been a mixed success. This project succeeded in its main goal, which was designing and implementing a functional electronic governor system. This is the functional version of the simulated governor system which was created in Fall of 2022. It improved upon that system by using interrupts instead of polling within the program and by implementing PD control, rather than just proportional control. Beyond that, this project has provided a great deal of experimental data, which will be a great resource as this research continues.

However, the performance of the electronic governor proved to be somewhat lackluster compared to the original mechanical centrifugal governor. Hopefully, the performance of the electronic governor can be improved in the future.

Future Research

This research began last semester with a simulated electronic governor system. This semester saw that governor system implemented on an actual engine. In the future, this research project will continue to evolve.



Currently, the next stage in this project is to implement an electronic fuel injection system on the generator, alongside the electronic governor system. In a similar vain to this semester's research, this fuel injection project would involve replacing a mechanical system with an electronic system. The mechanical carburetor would be replaced with a standalone computer-controlled fuel injection system. Additionally, the ignition system may also be replaced with a computer-controlled alternative. This project would likely use either the MegaSquirt or Speeduino aftermarket engine computer platform.

Next semester's project will either involve the fuel injection conversion, or it will involve improving the design of the electronic governor on the current engine. Either way, this research project will continue, come next Fall.