April 2011

Design, Analysis, and Optimization of a FSAE Racecar

Alexander Foster Scott
Worcester Polytechnic Institute

James Thomas Loiselle
Worcester Polytechnic Institute

John Paul McCann
Worcester Polytechnic Institute

Project Number: ECC-A101

Design and Analysis of a FSAE Racecar

A Major Qualifying Project Report

Submitted to the Faculty

of the

Worcester Polytechnic Institute

in Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science

Submitted By

_____

James T. Loiselle

_____

John Paul McCann

_____

Alexander F. Scott

Date:

Approved by:

_____

Professor Eben C. Cobb

Abstract

Each year the Society of Automotive Engineers hosts the Formula SAE Colligate Design Competition for engineering students from around the world. The competition is judged based on the engineering, performance, and cost of a Formula style car designed to be produced in a small 3000 unit production run. This project used an uncompleted FSAE racecar that was analyzed by the group. Once the car was analyzed, the group designed and redesigned the different systems of the car, including the intake, exhaust, and pedal box. Another aspect of this project was studying the different systems of the car and determining how they can best be optimized to produce an increase in performance by either increasing the power of the car or reducing the weight, both of which would lead to lower lap times and a faster car. A data acquisition system was designed to record data from different systems to be used to help optimize the car. The data system was designed to be easily transferred between cars for use by future FSAE teams at WPI.

Acknowledgments

The group would like to thank the following people:
Barbara Furhman, for all her help with ordering parts and guidance
Adam Sears, for all his guidance with machining
Neil Whitehouse, for all the random questions he answered
2012 FSAE MQP, for their help throughout the project
Sabertooth Robotics, for keeping us sane throughout this project
Campus Police, for helping setup testing

This project couldn't have been completed without all the help and support from those people.

Table of Contents

List of Figures

List of Tables

List of Equations

Introduction

In 2009, the WPI FSAE team designed and began manufacturing a racecar for the 2009 Formula SAE competition. At the end of the year the team was unable to complete the fabrication of the 2009 FSAE racecar. Using the design and completed parts from the 2009 FSAE racecar, the MQP group from this year completed the FSAE racecar. In order to complete the car the group started by studying the car and determining what parts were never manufactured, what parts were never designed, and what parts needed to be redesigned. Once the car was evaluated, the MQP group began to manufacture, design, and redesign the appropriate parts.

Analysis of the 2009 car

The 2009 car was first analyzed by the group to determine which components needed to be designed or redesigned and which components needed to be manufactured. A visual inspection of each system of the car was carried out to determine what components were missing. A list of missing components was compiled from this visual inspection. The 2009 MQP report and CAD files were reviewed by the group to determine what parts were designed during the 2009 MQP, but never manufactured. . Another important part of analyzing the car was determining which of the parts that had been designed and manufactured needed to be redesign or remanufactured.

Parts Never Designed

The components that were never designed was one of the first things the MQP group worked on, since some of them were needed to get the car running with the engine that was on hand using the stock unrestricted tune. The first system the team found was missing was the exhaust system. Another part that was missing from the car was a throttle body to use with the new intake. In 2009 an intake with an integrated restrictor was purchased for the MQP so one was never designed. Over the past year the throttle body that was ordered had been misplaced so the team decided to fabricate a throttle body instead of ordering a new one.

Parts Never Manufactured

There were several parts from the previous MQP that were designed but never manufactured. As the group was going over the car one design of the car kept puzzling the group; why was the chain adjuster designed to also adjust the CVT belt in the opposite direction. The part was designed such that tightening the chain would loosen the belt. It wasn't until looking at the CAD model of the car and reading the design report submitted to SAE for competition that the group discovered the swing arm cam adjuster.

Parts Redesigned

Three major parts needed to be redesigned on the 2009 FSAE racecar. The most pressing system that needed to be redesigned was the intake from the previous MQP. The two biggest problems with the initial intake design was the fuel injector location and the three welded bends in the intake runners right before the

intake port of the cylinder head. Another part of the car that was in need of a redesign was the pedal location. The original pedal location had the accelerator and brake pedal overhung in the foot well, positioning the master cylinder reservoirs outside of the roll envelope of the car, in violation of the FSAE rules. The final part that was redesigned was the front hubs for the car. The 2009 hub design was modified to promote ease of manufacture.

Data Acquisition System

Data Acquisition is the process of sampling data from real world systems so that it can be numerically analyzed. This is necessary in racing applications as there is a large degree of adjustability in racecars in order to adapt them to track conditions as well as the driver. By quantifying the parameters of the car, doubt is removed; no longer must one adjust the suspension based on driver feedback alone. Commercial Data Acquisition System (DAQ) are available from a variety of manufacturers, however they do not offer the flexibility that is needed to integrate as many inputs as desired on the Formula SAE racecars while remaining small and light enough to be packaged in the car without being overly obtrusive.

When discussing what vehicle parameters to collect, the team decided on the following list:

- Physical Parameters
    - Position
    - Acceleration
    - Rate of Rotation
- Suspension Parameters

    - Suspension Position
    - Steering Position
    - Tire surface temperature Gradient
- Engine Parameters
    - Engine Speed (RPM)
    - Manifold Pressure (MAP)
    - Throttle Position
    - Air/Fuel Ratio

These parameters would allow the team to fully analyze the dynamics of the car, as well as the performance of the driver.

Parts Manufactured

Swing Arm Cam Adjuster

The swing arm cam adjusters were one of the few parts that were designed, but never manufactured. To adjust the swing axle the mounting bolts that hold the swing axle to the chassis mount to a slot allowing the position of the swing axle to be adjusted. This part was also designed to adjust the chain tension independent of the CVT belt. The discovery of this part answered one of the biggest mysteries of the 2009 car, how the chain tension is adjusted without adjusting the CVT Belt tension.

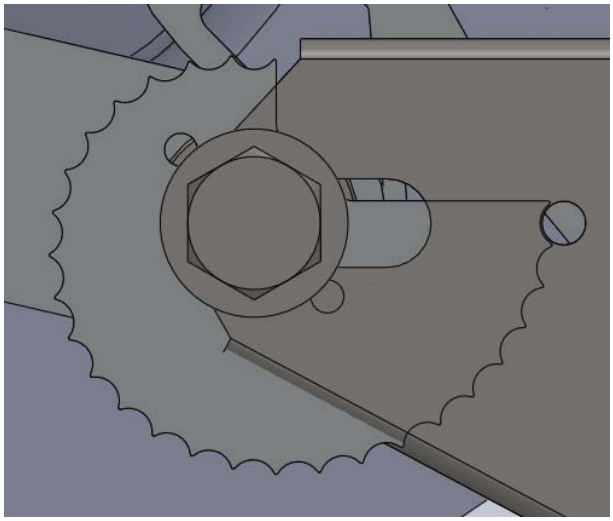


Figure 1: Initial Design or Swing Arm Cam Adjuster

The swing arm cam adjuster is a very simple way of adjusting the chain tension. One of the major advantages of this is the ease of manufacturing. To manufacture the cam adjuster two holes were drilled into the stock material. Once the two holes were drilled the stock was mounted to a fixture plate. A minor design change made to the swing arm cam adjuster was how the swing arm adjuster is used. Initially the adjuster was mounted permanently using the same bolt that the swing arm mounts with and is adjusted using a spanner wrench, which can be seen in Figure 1. After making a prototype from acrylic the team noticed that spanner wrench used to adjust the swing arm cam adjuster did not fit in the allotted space and could not be adjusted. To remedy this problem the team redesigned the swing arm cam adjuster to be used as an alignment tool used when tightening the swing arm and no longer a permanent part of the assembly. Once the swing arm is bolted in place the adjuster is removed from the car. To do this the mounting hole was machined the same size as the spacer already used in the swing arm mounting bolts.

Figure 2: Final Swing Arm Adjust Installed

Figure 3: Final Design of Swing Arm Cam Adjusters

Parts Designed

Throttle Body

To complete the intake system it was necessary that a throttle body be designed.

Before deciding on which style of throttle body to design, different styles of throttle

bodies were researched. The group focused their research on three different types of throttle body designs; barrel (Figure 7), butterfly (**Error! Reference source not found.**), and spike (Figure 6). A fourth type of throttle body that was considered, but not researched, the iris (Figure 5). The iris throttle body was used in a previous year at WPI, but the group decided not to pursue the idea based on the complexity of the motion needed to open the throttle body, and the complex geometry of the throttle plates.

Figure 4: Butterfly Throttle Body

Figure 5: Iris Throttle Body

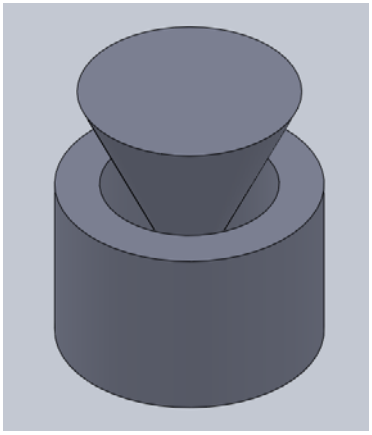


Figure 6: Simple CAD Model of a Spike Throttle Body

One of the major design advantages of the spike and barrel is that at wide open throttle there is no restriction in the flow path of the throttle body, unlike the

butterfly throttle body. An advantage of the barrel and the butterfly throttle body is the simplicity of the throttle plate motion. Both the butterfly and the barrel throttle bodies use simple rotational motion to open the throttle, where the spike requires the use of a lever arm to open the throttle. The barrel throttle body also had an ease of manufacturability and assembly. The barrel throttle body uses a simple housing and rotating barrel to open and close the throttle and can be assembled easily. The butterfly throttle body is also simple to manufacture, but is more complex to assemble since the throttle plates need to be attached to the shaft one the shaft is installed into the throttle body. The spike throttle body is easy to assemble, but uses much more complex shapes for the throttle body which decrease manufacturability.

After researching all three options for throttle bodies the group decided that the barrel was the best option. The barrel throttle body had many advantages and few disadvantages. The barrel was a good compromise between ease of manufacture and the reduced restriction seen in other throttle body assemblies. The 2009 MQP had planned to use a butterfly throttle body on this car and had purchased one. When driving this car the throttle will be wide open, giving the advantage to the lack of restriction when at wide open throttle.

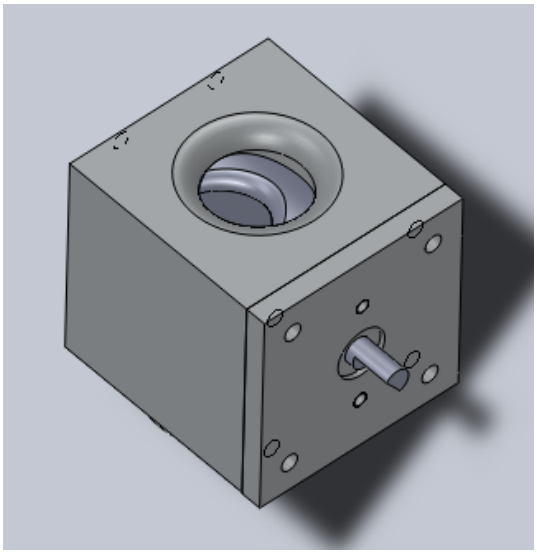


Figure 7: CAD Model of Barrel Throttle Body

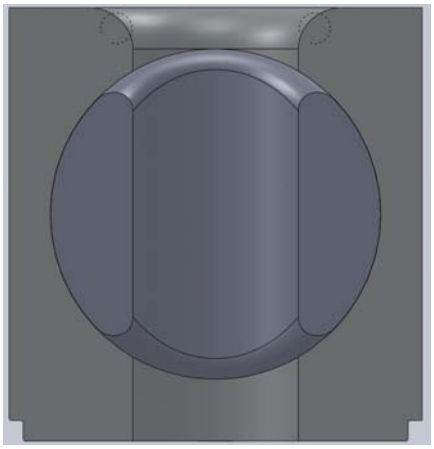


Figure 8: Section View of Throttle Body

Exhaust

The next major system that was never designed was the exhaust system for the engine. The engine used for the Formula car this year has only two cylinders making the fabrication of the exhaust much simpler. With only two cylinders the exhaust system needed only two primary tubes on the exhaust, which saved both fabrication time and money spent on the raw materials. Another advantage of having a two cylinder engine is the reduction of space needed to run the exhaust.

Fewer cylinders and exhaust primaries reduced the amount of space needed to run the exhaust. Since there were fewer primaries there was also a reduction in weight of the exhaust system, which increases the power to weight ratio, which is a major advantage in the FSAE program.

Figure 9: Exhaust Primaries

Parts Needing Redesign

Intake

The intake was one of the parts of the intake that was in need of a major redesign. The intake designed for the car in 2009 had many design features that were in need of a redesign. The original intake design used sections of aluminum tubing welded together to create the needed bends. Another design flaw of the intake was the location of the fuel injectors.

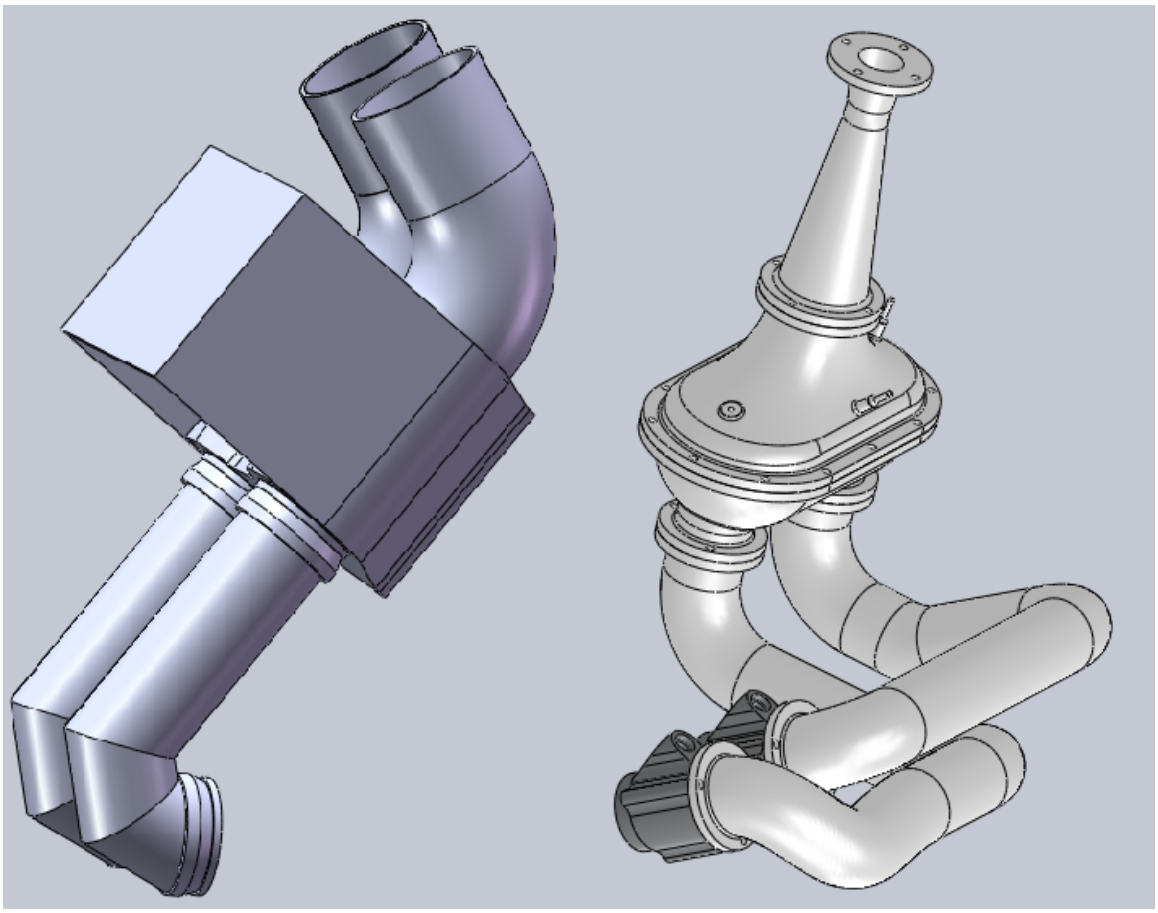


Figure 10: 2009 (left) and 2011 (right) Intake Designs

Injector Location

The location of the fuel injectors was the first item that needed to be redesigned on the intake. When the engine comes from the factory the injectors are located approximately 1-2 inches from the intake valves. This allows the injectors to spray fuel right at the intake valves, allowing the fuel to atomize while entering the engine cylinder. In the stock intake design the fuel injectors are mounted in the same assembly as the throttle body. The intake design from 2009 also makes use of the throttle body assembly to position the fuel injectors. While the 2009 intake design uses the same throttle body assembly to hold the injectors, the location of the assembly was very different than the stock location. The 2009 design took the

original throttle body assembly, removed the throttle plates, and mounted them approximately 10 inches away from the intake valves. With the fuel injectors being located so far from the intake valves the fuel would not atomize as well when entering the cylinder, causing a decrease in performance. Since the FSAE rules require that a 20mm restrictor be used for naturally aspirated engines, the intake itself needs to be optimized to reduce any decrease in performance. To keep the injectors in the stock location the group decided to design and manufacture an aluminum block similar to the original throttle body assembly. Since the throttle body location was no longer going to be as close to the cylinder, the new part was designed to only hold the injectors in place, making it easier to design and manufacture.

Figure 11: Unfinished New Injector Manifold

Runners

The next part of the intake that was redesigned was the intake runner plumbing. In the design from 2009 the intake runners had two welded bends right before the air would enter the engine. With this design the airflow would be turbulent when entering the cylinder, and decrease the power created by the engine. Since the entire intake was not fabricated, the team did not know if the intake had been tuned for any specific RPM range, or if the intake runner lengths were just an arbitrary length. When redesigning the intake runners, the length was designed for a specific RPM range. Since the 2011 car uses a CVT instead of a traditional 5 or 6 speed transmission, the intake runners were designed for the CVT lock up speed. Since CVTs use two pulleys to create an infinite number of gears the engine speed remains constant once the CVT locks. Since the engine will stay at a constant RPM the intake runners were designed around the CVT lock up speed of 7500 RPM. Using the equation for intake runner length the length of the runners was calculated to be approximately 29 inches long when using the first reflective value. Since the intake is located on the backside of the engine the runners were able to use the first reflective value for the runner length. In past years, due to space restraints, the intake runner length needs to be designed using the second or third reflective value. By using the first reflective value the intake length can be optimized for the CVT lock up engine speed to get the most useful power from the engine.

To tune the intake runners wave theory equations were used to determine the ideal runner length for our engine. The equation used to calculate the runner length was:

Equation 1: Intake Runner Length

$$\text{Runner Length}=((\text{Effective Cam Duration}*.25*2550)/(\text{RPM}*\text{Reflective Value}))-(.5*\text{Diameter})$$

When using the equation for calculated runner length the variables are defined as:
Effective Cam Duration (ECD) equals 720-Cam Duration-30 in degrees
2550 is the calculated wave velocity use from past MQP research in inches per second
RPM is the targeted speed of the engine in revolutions per minute
Reflective Value is the reflected wave (1, 2, 3,… n)
Diameter is the diameter of the runner in inches

Using that equation the runner length could be calculated with the ideal RPM that we wanted to tune the intake runners to. Since the engine speed will be 7500 RPM most of the time the group decided to tune the intake runners for ideal power at 7500 RPM. We also calculated the intake runner length for multiple reflective values and then determined what length runner could be fit into the car. For our application the first reflective value was used to wrap the intake runners around the engine due to the space constraints of the seat and chassis. For the diameter of the intake runners the stock intake diameter was used. The intake runner length calculations can be seen in Table 1.

Effective Cam Duration (Degrees)	RPM	Diameter (Inches)	RV 1 Length (Inches)	RV2 Length (Inches)	RV 3 length (Inches)
402	8500	1.62	29.34	14.265	9.24

Table 1: Intake Runner Length Calculations

Figure 12: Intake Runners

Plenum

Another part of the intake system that was designed was the intake plenum. In the research that the team had done about intake plenums did not really specify a specific volume for the plenum. Since there is no "perfect" plenum volume the team decided to design a plenum with an adjustable volume. All of the research the team had done recommended a plenum volume in the range of one to four times the displacement of the engine. In order to create an adjustable volume plenum the intake plenum was designed to be two pieces. The plenum was separated into the upper and lower plenum. The minimum size of the intake plenum is approximately two times the displacement of the engine. In order to adjust the plenum volume center sections can be added to the plenum.

Figure 13: Inside Lower Plenum

Restrictor

For ease of manufacturing the intake restrictor was designed in SolidWorks and then printed using the rapid prototyping machine. The restrictor is a converging diverging nozzle. If this was to be machined out a piece of aluminum it would be a multi-fixturing operation, which would add complexity to the part and possibly lead to a machining error during the re-fixturing of the part. By rapid prototyping the restrictor it could be manufactured in one operation and eliminate the chance of having the two sides not being concentric. When manufacturing parts on the rapid prototyping machine there is little to no setup and the rapid prototype doesn't need someone to watch over the operation to prevent a crash of the machine tool. Another advantage of using the rapid prototyping machine is the weight of the material. The rapid prototyping machine prints the restrictor in ABS plastic which is lighter than aluminum. When designing a racecar any reduction in weight will be

advantageous. Since the power on the FSAE cars are restricted one of the best ways to gain an advantage is by reducing the weight of the car to increase the power to weight ratio.

Figure 14: Restrictor and Upper Plenum

Pedal Box

The pedal box was also redesigned as part of this year's MQP. The initial design for pedals used overhung accelerator and brake pedal. The reason behind the original design was to reduce the space needed for the pedal and brake assembly, which would allow a taller drive to sit more comfortably in the car. While the overhung pedals did allow for more driver room, the top of the brake pedal and master cylinders were both located outside of the roll envelope of the car. With the top of the pedal and the master cylinders located outside of the roll envelope it

would allow both the master cylinders and the pedal to be damaged in the event of a roll over or an impact on the top of the chassis. Since the team saw this as a safety issue the pedal box was redesigned to keep all of the braking system enclosed within the roll envelope of the car.

The new pedal box design used floor mounted purchased pedals for easy of assembly and to eliminate the complexity of machining the pedals or the master cylinder mounts. By using floor mounted pedals all of the pedal could be mounted within the roll envelope of the car and be fully protected in the event of a crash or rollover. Another advantage of moving the pedals to the floor was the easy of running the brake line, while also keeping them contained within the chassis of the car. Running the brake lines inside of the frame of the car eliminates any potential chance of a brake line getting damaged in the event of a collision or impact.

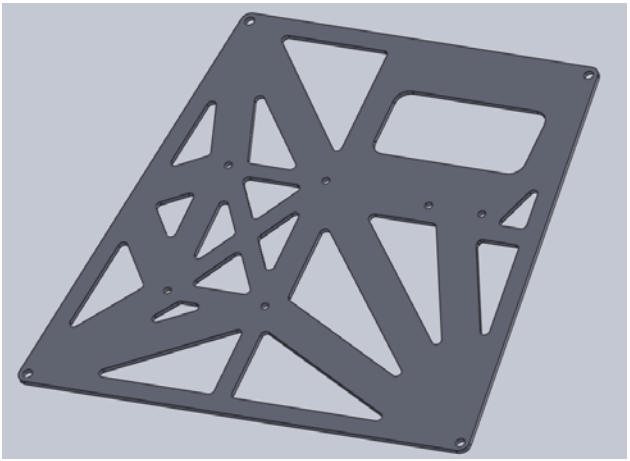


Figure 15: Redesigned Pedal Plate

While redesigning the pedal box the brake system was also studied and improved. When the car was initially tested in early b term the master cylinders sized for the car in 2009 could not lock the rear tires when testing the brakes. After rechecking the brake calculations from 2009 the group had determined that the

brakes were sized correctly using the assumption that the wheels are just a freely rotating mass. After more testing the team determined that the assumption of the rear wheels freely rotating was incorrect. Since the car uses a CVT and has no clutch there is no way to disengage the engine from the axle. Since the engine and axle remain connected, the engine will continue trying to drive the wheels while the brakes are trying to slow the wheels down. The engine driving the wheels applies a moment on the axle in the opposite direction of the brake caliper, which reduces the effective brake torque applied to the axle. In order to increase the effective brake torque the master cylinder in the rear was increased to provide enough effective brake torque to lock the rear wheels of the car. In order to keep the correct front to rear brake bias the master cylinders used for the front brakes was recalculated and determined that a small master cylinder should be used.

Hub

The front hubs originally machined back in 2009 were machined 0.060 inches out of round. This was due to movement in the fixture when the hub was flipped for the final operation in the mill. To avoid this error again and to increase in the manufacturability, the inner collar on which the bearing race sits was replaced with two grooves for snap rings. By using snap rings and removing the inner collar we are able to machine the inner section in one operation on the lathe ensuring the trueness of the hub. The two hub designs can be seen below in Figure 16. The new hubs were machined out of 7075 aluminum.

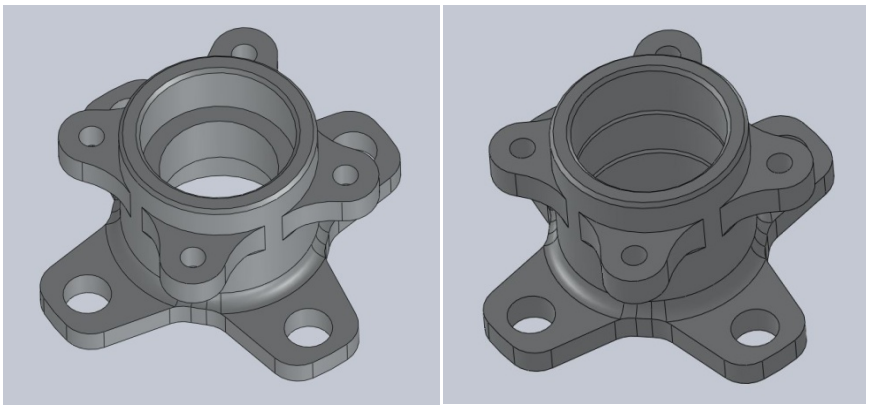


Figure 16: 2009 (Left) and 2011 (Right) Hub Designs

Even with the changes to the design of the hub, three setups, one lathe, two mill, are still required to machine the hubs, one in the Haas SL 20 and two in the Haas Minimill. The CAM software Esprit was used to generate the tool paths and NC code for both the turning operations and the milling operations. The first setup in the lathe contours the outer shape and inner portion of the hub. Figure 17 shows the hub in the lathe after it has been through the first setup.

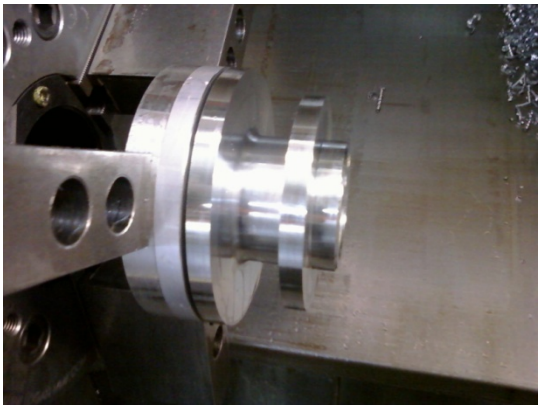


Figure 17: Hub Stock After Turning Operation

The final two setups contour the tabs for the studs to be pressed into and the brake rotor to be bolted to. The tool paths can be seen in Figure 18. Figure 19 shows the completed hub.

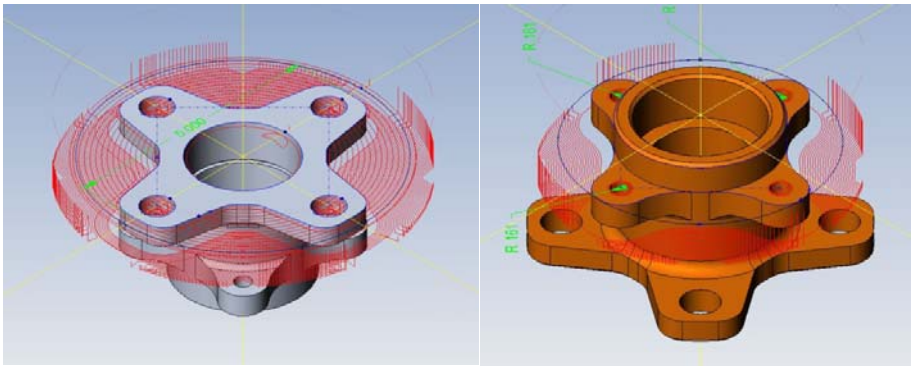


Figure 18: Milling Operation Tool Path for Hubs

Figure 19: Completed Hub

Data Acquisition System Hardware Design:

File Considerations

When deciding what format to use, the software packages used to interpret and analyze the data was an important factor on the data storage method. Most commonly used software, such as MATLAB and Microsoft Excel, is capable of reading a comma-separated-value (.csv) file. This common file type is also easily

generated in low powered hardware such as a microcontroller. Also, the simple nature of these files keeps file size and write-time to a minimum.

Another usability concern was the retrieval of the collected data from the DAQ for analysis. In many solutions which utilize internal storage, a tether is required in order to transfer data from the DAQ to the computer for analysis. This can cause complications since the DAQ which is fixed to the car is cdirectly connected to the computer, and requires the DAQ to be powered in order to transfer data, which could cause issues should the racecar lose power mid-transfer. This prompted the use of removable media, in the form of either a USB mass-storage device, such as a thumb drive, or other removable media such as a compact flash or SD-card.

Microcontroller Selection

With these basic hardware constraints in mind, the search for a suitable microcontroller was significantly narrowed. Among the contenders were embedded ARM based systems, such as KIT-ARM by Technologic Systems, as well as an relatively obscure chip called the USBizi by GHI Electronics. Both of these options offered plenty of processor speed while fitting in the tight spaces of the racecar. GHI Electronics, however, also offers their USBizi systems in varying packages of different features and sized, as well as offering software libraries for a variety of expansions. In particular, the FEZ Rhino offered a small package, making use of the 72 MHz USBizi, along with onboard peripherals such as Dual CAN controllers, two SPI Channels, I2C, four UARTs, a MicroSD slot and USB Host and Client Controllers. All of these features came supported by a Standard Development Kit distributed and

often updated by the manufacturer. Additionally, GHI offered a GPS extension for the FEZ Rhino.

Figure 20: FEZ Rhino microcontroller

The FEZ Rhino also was the only microcontroller in consideration that was able to be programmed in C-Sharp (C#), particularly in the Microsoft .Net Micro framework. C# allows quicker application development and a more user-friendly programing experience through a combination of C++ performance with Java-esque syntax. C#, like Java and C++ is an object oriented language which allows for the abstraction of data beyond that of purely procedural languages like C or assembly. This allowed for much easier integration of sensors which operate on different interfaces. This also allowed the use of Microsoft Visual Studio, a much more powerful development environment when compared with other options, such as Eclipse.

Physical Parameter Sensors

Position

GHI Electronics, the manufacturer of the FEZ Rhino microcontroller, offered a GPS extension which would integrate easily into the microcontroller. This was an obvious choice as it was designed to integrate easily into the microcontroller and was offered with a premade software library.

Figure 21: GHI Electronics GPS Extension

Acceleration

In order to measure the acceleration of the vehicle, an Accelerometer was required. In particular, both lateral and linear acceleration measurements were desired. This led to the search for a dual-axis accelerometer; however, not many dual axis accelerometers exist. Most applications use a three-axis unit since the cost difference is negligible and allows for additional mounting options. Additionally, most all accelerometer chips are some variety of surface mount packages. This is a

minor annoyance since soldering surface mount components is much more difficult than traditional through-hole components.

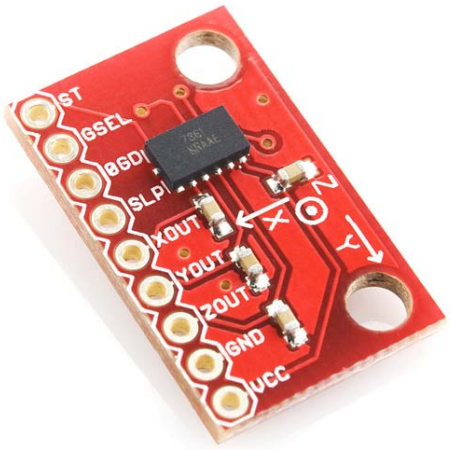


Figure 22: Sparkfun MMA7361 breakout board

Thankfully, many companies, such as SparkFun Electronics, Parallax, and Pololu Robotics & Electronics, offer premade break-out boards that allow use of surface mount components in traditional 0.1 inch pitch through hole applications. In particular, SparkFun Electronics offers the Freescale Semiconductor MMA7361L 3-axis accelerometer in a breakout board with an overall size of less than one square inch and providing easy access to all of the necessary pins on the accelerometer. The MMA7361 offers a unique feature of selectable sensitivities. By driving a pin high or low, one can choose between either ±1.5 g or ±6 g ranges. In theory, the racecar should not be able to reach far beyond ±1.0 g lateral acceleration, however there have been FSAE cars in the past that reach well over 2.0g with advanced aerodynamic packages that have since been banned from competition. Thus, for the needs of the Formula team, the lower setting should be sufficient, however, if it is

necessary to move to the higher setting, simply changing a jumper, or switching a wire will allow 4 times greater range.

Rate of Rotation

In order to measure the rate of rotation of the chassis, a Gyroscope was necessary. The rate of rotation around the vertical axis was the main topic to watch, as well as body roll, the rotation around the axis passing through the car along its length to help analyze transitions from turning left and right, such as when driving a slalom. 2 axis MEMS (Micro Electro-Mechanical System) Gyros are commonly available from a variety of manufacturers, such as Analog Devices, InvenSense, and ST Microelectronics. As with the Acclerometers, the majority of MEMS based Gyros are made in surface mount packages. This again prompted the use of a breakout board, again from SparkFun Electronics. In this case, the breakout board uses the IXZ-500 dual axis gyro from InvenSense. This component has a high and low precision output for both axes. This allows the monitoring of small variations in rotation, such as during driving, and then measure rapid rotation such as during a spin.

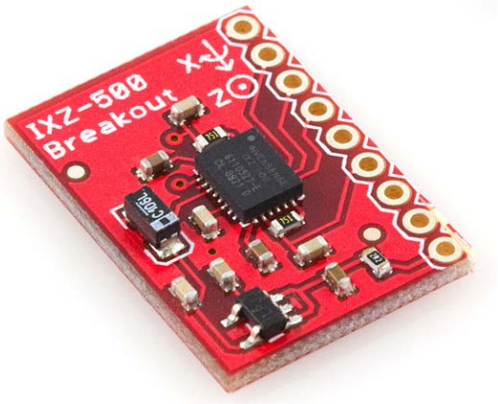


Figure 23: SparkFun IXZ-500 breakout board

Velocity

The Velocity of the vehicle is not necessarily an easy value to measure. The speedometer on a road vehicle is based off of the calibrated input from a vehicle speed sensor on the output shaft of the transmission. Consequently, changing the size of the tires, or the gearing after the transmission can dramatically throw off the reading. Additionally, if the driven wheels are not in a constant state of traction, these readings can not necessarily be trusted.

Given that velocity is the rate of change of the position of an object, and the integral of acceleration, calculating the velocity based off of the data collected from the GPS and accelerometer is a much safer way to calculate velocity without being affected by changes in the drivetrain.

Suspension Parameter Sensors

Suspension Position

Physical sensor

In order to find a solution to measure the position of the suspension, common racing data acquisition systems were looked at for inspiration. The standard way to measure the suspension is a linear motion potentiometer, connected in parallel with the spring/damper assembly. However, the unique design of the front suspension on the current racecar presented a challenge since the lower pushrod connection, the rocker, and the damper/spring assembly are non-planar with one another. At this point, a few ideas were discussed, such as a rotational potentiometer placed on one of the suspension mounting points, or the rocker. However, this would provide

very limited sensing ability since the suspension arms do not rotate through a very large range of motion. At this point, it was determined that measuring the suspension position based off of the compression of the spring would be the best way to approach the problem. By using a shorter linear motion potentiometer (not to be confused with a linear rotational potentiometer, where linear refers to the relationship to resistance and potentiometer position, as opposed to a logarithmic potentiometer used in audio applications), and measuring directly at the spring, compliance in the control arms, pushrod, and rocker is ignored by the system and the damper and spring are measured directly, as they are what is adjustable on the car.

The springs on the car are 150 mm in length, which is significantly less than most applications. Thus, a smaller linear motion potentiometer than those commonly used in racing was necessary. A common application of small sliding potentiometers is audio mixing boards. Conveniently, audio components are usually high quality which leads to a robust and precise potentiometer.

Figure 24: PTS60 and PTS100 sliding potentiometers

Bourn's, a manufacturer of many different shapes and sizes of potentiometers, has many selections, including the PTS series of slide potentiometers. Available in many different configurations, including 3 different lengths, and varying degrees of linearity, the PTS series would serve our purpose quite well. Particularly, the PTS60 – 02L-103B2 offered 60 mm of travel, a linear resistance taper between 0 and 10,000 Ohms, and solder lug connections.

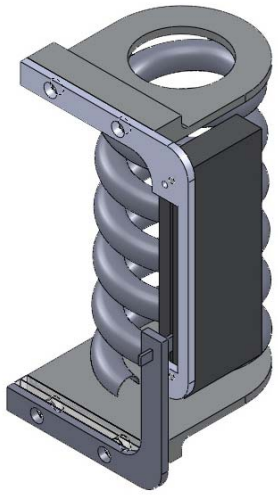


Figure 25: Spring based mount for PTS slide Potentiometer.

Sensor Interface

In order to convert the physical data from the Position of the potentiometer into a signal understood by the microcontroller, an Analog to Digital Converter (ADC) is needed. The Fez Rhino has an onboard 10-bit ADC. Thus, there are 1024 possible values that the onboard ADC can read. Given the travel of the PTS60 (60 mm) and the precision of the ADC, the accuracy of the sensor can be calculated:

Equation 2: 10 bit ADC precision calculation for 60mm potentiometer

$$minimum\ value = \frac{sensor\ range}{2^{ADC\ precision}}$$

$$\frac{60\ mm}{2^{10}} = \frac{60\ mm}{1024\ ticks} = 0.05859375\ mm\ per\ tick$$

While this is certainly precise enough to locate the position of the suspension, the wiring must also be considered. Running a wire from the front or rear suspension all the way to wherever the microcontroller ended up getting mounted would not be the safest idea in terms of signal purity; racecars are very noisy electrically, especially around the ignition system. In order to preserve the integrity and

accuracy of the data, a serial signal is preferred over an analog on for long cable runs.

With this in mind, a serially connected ADC was sought such that it could be placed close to the sensors to minimized electrical noise influencing the sensor reading. The Microchip MCP3204 and MCP3208 offered 12-bit precision, along with a Serial Peripheral Interface (SPI), which the FEZ Rhino already had onboard hardware to communicate through. The 12-bit value allows for an extra bit of precision as well:

Equation 3: 12 bit ADC precision calculation for 60mm potentiometer

$$\frac{60\ mm}{2^{12}} = \frac{60\ mm}{4096\ ticks} = 0.0146484375\ mm\ per\ tick$$

±15 μm is more than adequate for the position of the suspension as this number is smaller than a thousandth of an inch. This sensor setup is also applicable to longer spring/damper setups as Bourn's also makes the PTS100 series, a 100 mm equivalent to the PTS60 used currently. For example, if a future WPI FSAE car required the extra 40mm of travel, the system would still remain extremely precise:

Equation 4: 12 bit ADC precision calculation for 100mm potentiometer

$$\frac{100\ mm}{4096\ tick} = 0.0244140625\ mm\ per\ tick$$

Steering Position

Typically, steering position would be measured using a rotational potentiometer, or an encoder on the shaft that the steering wheel is connected to. However, when the car was originally created, no provisions for mounting such solutions were provided. Additionally, the area around the steering wheel and steering shaft is also

the same area occupied by the driver's legs. This required that the mounting of the sensor dictate the sensor selection, similarly to the suspension position. Since there was no open area to mount a rotational sensor that would not interfere with the driver's entrance or exit of the vehicle (an important safety aspect and measured performance at competition as drivers must be able to exit the vehicle in a set amount of time in case of a fire), a linear solution was considered. It turns out that the steering rack has less than 60 mm of travel from lock to lock, which is convenient since the PTS60 sliding potentiometer had already been researched for use in the suspension position. By using the same sensors for both suspension and steering position, less development is necessary since the precision of the position is the same as the suspension, at just under 15 micrometers.

Tire Temperature Gradient

When suspension parameters were initially discussed, instantaneous measurements of camber was discussed. There was no way to package a system of sensors to measure camber in relation to the track without having a major impact on the dynamics of the suspension due to increased unsprung weight or hanging off the sides of the car which could potentially clip cones at an auto-cross course.
Instead, a method to validate if a current camber setting was desired. As a car turns, the outside wheel is loaded, and the tire sidewalls flex, effectively pulling the contact patch of the tire under the car, causing more work to be done on the outside edge of the tire if the car was not set up with enough negative camber. With the increase in friction, the surface temperature of that area of the tire increases proportionally. Ideally, the contact patch of the tire would be flat at the most extreme lateral

acceleration to get the most grip possible out of the tire. This would cause an even temperature across the entire width of the tire.

Figure 26: MLX90614 Infrared Temperature Sensor

In order to measure the temperature of the surface of the tire, without affecting the tire, a non-contact temperature sensor was needed. Many tire temperature sensor arrays are available commercially, which are designed for use with commercial Data Acquisition systems, costing over 200 dollars per sensor. A more cost effective solution was found from Melexis. The MLX90614 infrared temperature sensor has a built in 16-bit ADC, in a package that is less than 10 mm in diameter. The standard model measures surface temperatures in a 90 degree cone emanating from its center. By placing the sensors approximately 1.5 to 2 inches away from the surface of the tire, three MLX90614 sensors can be aimed at the surface of the tire with no overlap in order to measure the temperature in three radial bands around the inside, middle, and outside of the tire surface. Consideration must also be made in the case of an impact from debris or other objects such as a cone on an autocross course to protect the sensors from colliding with the tire. In Figure 27, the circular

guards prevent the sensors from contacting the tire surface, as well as providing a wire management solution to reduce the chance of damage to the circuitry. This mounting bracket was designed to be manufactured using a laser cutter, making parts easy to produce and modify to suit the specific needs of current and future FSAE racecars.

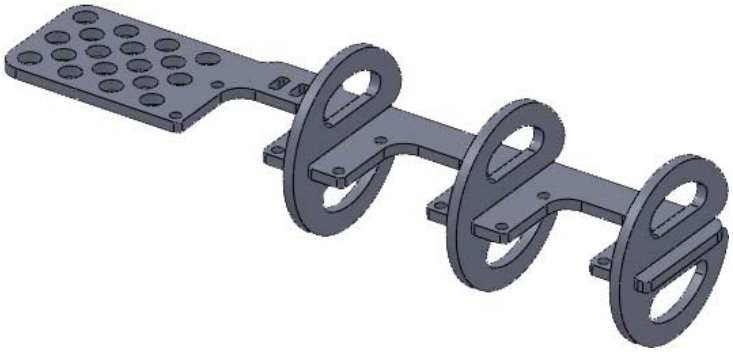


Figure 27: Tire Temperature Sensor mounting bracket

The MLX90614 uses an I2C interface to communicate with the microcontroller. This method of interface requires only two wires in addition to the power and ground wires for the sensor. Additionally, the two communication wires are common to all sensors on the I2C bus, which allows the sensors to be daisy chained in line, whereas the SPI interface on the MCP320X ADC chips requires a chip select signal unique to each chip in addition to its 3 common communication lines. The MLX90614 is precise to ±0.02 Celsius, which is more than necessary for noting the difference in temperature on the tire.

Other possible uses for the MLX90614 could be for monitoring brake temperatures, or any other surface temperature desired.

Engine Parameter Sensors

The parameters that were desired to be sampled from the engine, such as engine speed and throttle position, are already sampled by the engine itself in order to determine the fuel ignition events. It would be quite inconvenient to add redundant sensors in order to sample these parameters; thus, a more elegant solution was desired. The Haltech Sport 2000 ECU has a CANBUS output designed to work with their proprietary dashboard. This streams all of the engine parameters to the dashboard, including the parameters that were desired to for logging. However, due to the nature of CANBUS, there is no way to determine which values belong to which sensors. Upon contacting Haltech, they graciously provided a page of references for which values correspond to which sensors.

Figure 28: Haltech Sport 2000 ECU

CANBUS systems communicate on a bus that uses a pair of data lines, one operating at +12 volts and the other mirroring it at -12 Volts. This poses a problem to integrate with a microcontroller that runs on +3.3V. This is remedied by the use of a CAN Transceiver, such as the Microchip MCP2551, which is a meets all standards for the 8-pin CAN Transceiver footprints.

Data Acquisition System Software Design

The Focus of the software design was to integrate all of the different sensors in an easy to use format while maintaining as little computational overhead as possible in order to keep the time required to sample each value as low as possible. Because the FEZ Rhino uses the Microsoft .Net Micro Framework (NetMF), which is programmed in C#, the advantages of the high level object oriented language could be realized. One major advantage of an object oriented language, such as C# or Java, is the level of abstraction which can be applied to the interaction between the main execution and hardware interaction. This abstraction is accomplished through the use of an interface class which sets a list of required methods that all objects using that interface must contain.

Interface Class Design

In order to represent data in a .csv file, each point of data that is being recorded needs two things: a title to be recorded in the first line of the .csv file, and reading that gets printed in each subsequent line of the .csv file. In order to do this efficiently, a program would use a loop in order to iterate through a list of sensor objects, all operating in the same manner. This can easily be done if all of the sensors are read from the same source, such as the onboard 10-bit ADC. However, things become more complex when a variety of sensor communication methods are used. Knowing this, an interface class was developed to govern what all the

different sensors must be able to report when required by the main loop of the program.

In this application, the *Sensor* interface was developed with two methods. The *Name()* method was designed to return the unique name of the input represented by this iteration of the object, and the *Read()* method was developed to return the most recent reading of the input. Both of these methods would return their values in the form of a *String* object, which consists of the value already in ASCII character form for immediate printing to the .csv file. By requiring a ASCII output, it is possible to represent complex data in one field on the .csv file, such as the GPS data which is given in Latitude and Longitude components, instead of requiring four fields for the minutes and seconds of each component of the position.

Since all that the interface describes is a basic blueprint, the objects representing each different type of input are free to add other additional methods, as long as they also include the methods required by the interface.

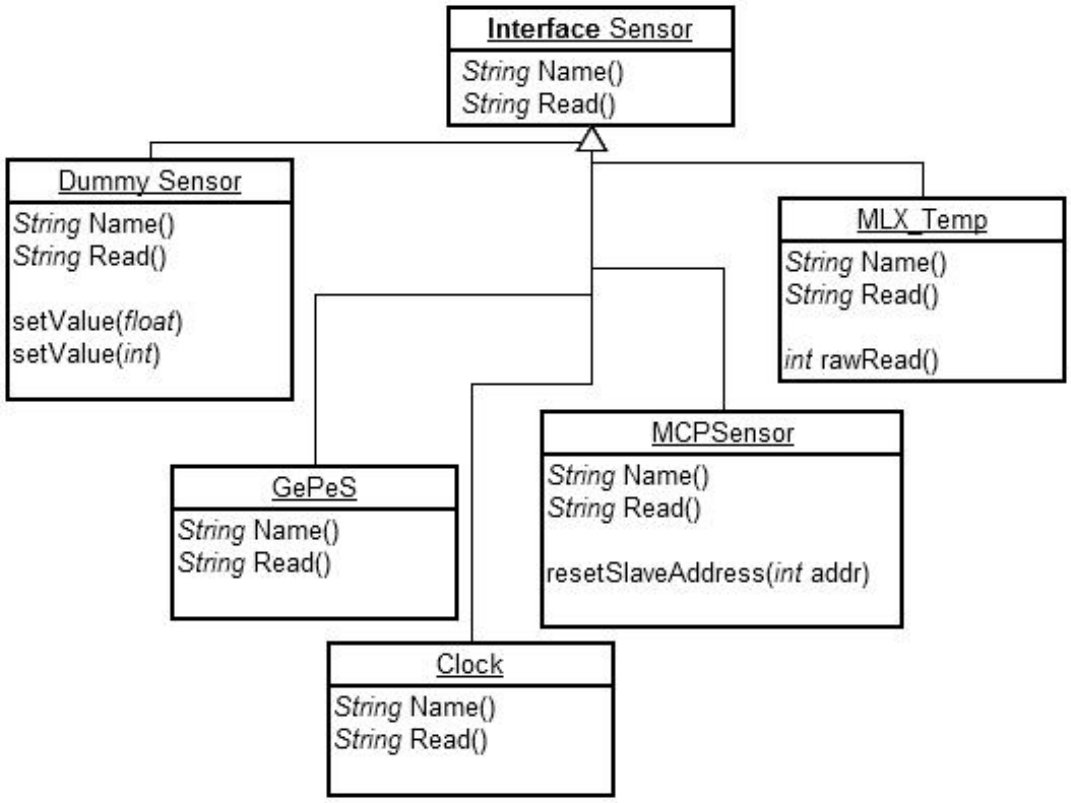


Figure 29: Class Diagram of *Sensor* Interface

External Analog to Digital Converter

Many of the sensors used require an ADC to interpret their outputs in order to be useful to the microcontroller. For this task the Microchip MCP3204 and MCP3208 12 bit ADC's were chosen. Offering either 4 or 8 channels of conversion controlled through an SPI connection, the MCP320X is able to interface with all of the analog output sensors, including the accelerometer, gyro, suspension position, and the steering position sensors. Due to the variety of the outputs of these sensors, a way to adjust the values accordingly was necessary.

Data Scaling and Offset

When the ADC returns a value, it is scaling a voltage between its ground a voltage reference values from 0 to its maximum value, which in the case of the 12-bit MCP320X is 4095. This number has no meaning until it is conditioned to give meaning to the value. For example, if the input was a simple measure of analog DC voltage, between 0 and 12 Volts, a scale could be applied, relating a maximum voltage of 12 to the maximum value of 4095:

Equation 5: Analog to Digital Conversion Scalar calculation

$$\frac{4095\ tick}{12\ Volts} = 341.25\ ticks\ per\ Volts$$

Thus, if the input is divided by the resulting value of 341.25, the analog voltage can be found, such as a reading of 2047, half the value of the maximum reading:

Equation 6: Analog to Digital Conversion Precision loss example

$$\frac{2047\ ticks}{341.25\ Volts/tick} = 5.9985348\ Volts$$

This also illustrates the loss of precision by converting to a digital value. In this case, 6 volts is read as 5.998 volts. These values are quite close however and the miniscule difference would likely not be missed.

In addition to merely scaling the sensor outputs, sometimes they must be offset in order to read values less than 0. For example, if the voltage meter mentioned in the previous example was required to read a system that varied from -6 volts to +6 volts, the scale would be the same since the potential difference between the ground and voltage reference is still 12 volts, however a 0 volt reading would report as 2047 from the ADC. For this reason, in addition to the scalar, an offset parameter

was added to each object representing the external ADC connections. This allows the programmer to use the MCP320X chip to interface with a variety of sensors without having to write specific drivers for each sensor. For example, the accelerometer output varies between 0 and 3.3 volts which represents -1.5g to +1.5g accelerations, while the suspension position sensors have a range of 0 to 60mm, but have a static position that could potentially vary each time a different alignment or driver is recorded.

SPI Communication

In addition to the data conditioning requirements, a few other parameters must be added for communication with the MCP320X chips. Since the chip uses a Serial Peripheral Interface (SPI) connection, it must have a pin on the micro controller assigned as the chip select for each individual chip. SPI has 3 common pins that connect to all devices on the bus, the Master Out Slave In (MOSI) communication line, the Master In Slave Out (MISO) line, and the a synchronizing clock pulse, as well as a line unique to each chip, the Chip Select, which controls which chips is actively communicating with the microcontroller at a given time. Since the chip select varies from chip to chip, it is a settable value in the constructor of the *MCPSensor* object. Also, the clock frequency is an additional adjustment that can be made, although there is little reason to not run at maximum speed, unless there is a communication issue that must be debugged and the available instruments (such as an Oscilloscope) are unable to operate as fast as the chip's maximum speed.

The only additional input needed is which channel of the ADC is desired for reading. The MCP3204 offers 4 channels, while the MCP3208 offers 8 channels. These

channels are addressed starting with 0 up to one less than the number of channels. This is represented by a 3 bit space in the communications, which is the perfect size to send a number between 0 and 7 (000 and 111 in binary). If an MCP3204 receives a number above 3 (011 in binary) it ignores the leading 1, effectively subtracting 4 from the number, as such it is only necessary to write one set of methods that will work for both the MCP3204 and MCP3208.

MLX90614 Infrared Temperature Sensor

The MLX90614 infrared temperature sensors report their values in a 16-bit unsigned value. Each tick here is the equivalent of 0.02 Kelvin. In order to scale the value from the raw data form into Celsius degrees the following operation is performed:

Equation 7: Temperature Sensor Scaling calculation

$$[(unsigned\ reading) * 0.02] - 272.0 = Celsius\ Degrees$$

This allows the temperature gradient across the tire to be easily seen as differences as slight at 0.02 degrees Celsius are recorded.

The major trouble with working with multiples of the MLX90614 is the communication. This unit uses an I2C communication protocol. I2C is a very common and very useful protocol as it uses only 2 wires in addition to the power and ground for the chip it is connecting to, as opposed to 4 wires for SPI. However, since I2C does not have a chip select function, every I2C device has a Slave Address set in its EEPROM (Electronically Erasable Programmable Read Only Memory), and in the case of the MLX90614 and most I2C devices, it is set to a default value from

the factory. In order to change the slave address of the sensor, it must be first reset to 0x000 (0 in hexadecimal, representing 12 zeros in binary) and then set to the desired new address. The MLX90614 must then be power cycled for the new address to take effect. This process may seem simple, however, in order to write to the EEPROM in the MLX90614, a Cyclic Redundancy Check (CRC) is required. The CRC is an additional byte of data that is calculated based on the bytes preceding it in the packet. This is used to check for any sort of transmission error. The main issue however, is that the FEZ Rhino has software libraries to calculate 16 and 32 bit CRC values, and the MLX90614 uses an 8 bit CRC. Since only 6 sensors needed to have their addresses changed, hand calculations using an online guide were performed in order to spend less time writing a method to calculate the 8 bit CRC values. After the process of changing each address, it was written in marker on the side of the MLX90614's body. This would make it much easier to mount the sensors, knowing the address of each sensor mounted in each location, instead of having to go around and try to guess which sensors had address 0x010 versus 0x015.

CANBUS Engine Parameters

The engine parameters that were desired for logging were engine speed (RPM), Throttle Position (TPS), Air/Fuel Ratio (Lambda or AFR),and Manifold Absolute Pressure (MAP). Since these are all available over the CANBUS interface from the Haltech Sport 2000 ECU, that was the best way to capture this data as it would require the least amount of additional hardware.

The CANBUS system works similar to I2C where everything has an address, however the packets are limited to 8 bytes of data in order to keep a single device

from overwhelming others. Thus, this limits the amount of data that can come from a single address. Since the Haltech has more than 8 bytes of data that it is capable of sending, it sends packets from multiple addresses. The Haltech was designed to work with an aftermarket dashboard over the CANBUS connection, such as long as the feature is enabled in the Haltech software, it should send data continuously over the CANBUS connection. When a CANBUS message is received by the FEZ Rhino, it creates a *CANMessageReceivedEvent* which is handled by a static method in the main execution program. In order to store the data from these event handlers, a sort of placeholder object was needed. This led to the creation of the *DummySensor* object. The *DummySensor* is basically storage location encapsulated within an object that complies with the *Sensor* interface. When the even handler parses the data from the *CANMessage* object, it stores the appropriate data in the *DummySensor* objects so that they can be read by the next iteration of the main data recording loop.

Global Positioning System (GPS)

The GPS Extension available from GHI Electronics for use with the FEZ microcontrollers comes with driver to act as a layer of communication between the main program and the GPS hardware. When this was tested, it functioned as designed, however it left something to be desired due to a simple mistake. Whoever GHI Electronics had write their driver used integer numbers for both the Hour and Minute components of the GPS coordinates. This limited the precision of the device severely, as one minute of latitude in Worcester (the length of each degree varies depending on latitude) is over one mile, and a minute of longitude is .85 miles. This

was not nearly detailed enough to provide the necessary data to trace the motion of the car over the course of a small autocross course.

This prompted further research into the origin and operation of the GHI Electronics GPS extension. The GPS module, manufactured by Sirf Technology, uses a standard UART interface and transmits the Degrees in integer form, and the minutes of Latitude and Longitude to four decimal places. There is no logical reason why the GHI provided driver did not include the extra precision. Thankfully the source code for the driver was provided, so it was then modified to include the added precision which greatly improved the accuracy of the unit.

This better version of the driver was then abstracted within an object that meets the requirements set by the *Sensor* interface. The output format is a simple Latitude and Longitude: Degrees & Minutes Latitude Degrees & Minutes Longitude. This takes only one slot in the .csv file, and is very compact, and easy to read.

Timing

The Fez Rhino includes a Real Time Clock (RTC) which can be set manually, or by requesting the current date and time from the GPS system. The RTC keeps time down to the millisecond, which is adequate for the needs of the DAQ as 1000 or more samples per second are highly unlikely. The RTC is abstracted in a class, aptly named *clock* which returns the time in a standard clock form:

Hours : Minutes : Seconds . Milliseconds

This makes the output easy to read, as well as being easy to parse when loaded into analysis software.

Main Execution Program

The main execution program starts by loading all of the necessary objects into memory. This includes constant things like the file name and which pins control lights or are connected to switches, as well as the objects pertaining to the various sensors connected to the system. All of the *Sensor* objects are added to an *ArrayList* object, which is dynamically sized array. Any *Sensor* objects which require zeroing before the start of logging are zeroed before being added to the *ArrayList*. Next, the USB storage device is detected, and the filesystem is loaded. After loading the file system, the DAQ scans the files listed in the root directory to make sure not to overwrite any previous logs, incrementing the number on the end of the file until it determines that the file is not present on the media. Next, the file is created and the first line is written based off of the *Sensor* objects in the *ArrayList*. Next, the system goes into a semi-infinite loop. The loop will only exit when the dashboard switch is moved to the off position. During the loop, the DAQ will continuously read from its sensors sequentially, and write the readings to the .csv file. When the loop is exited, the file is saved and closed. At this point the system goes into another loop waiting for the switch to be flipped back into the on position or for the system to turn off. If the switch is reactivated, the program jumps back to the file creation process and begins collecting data once again.

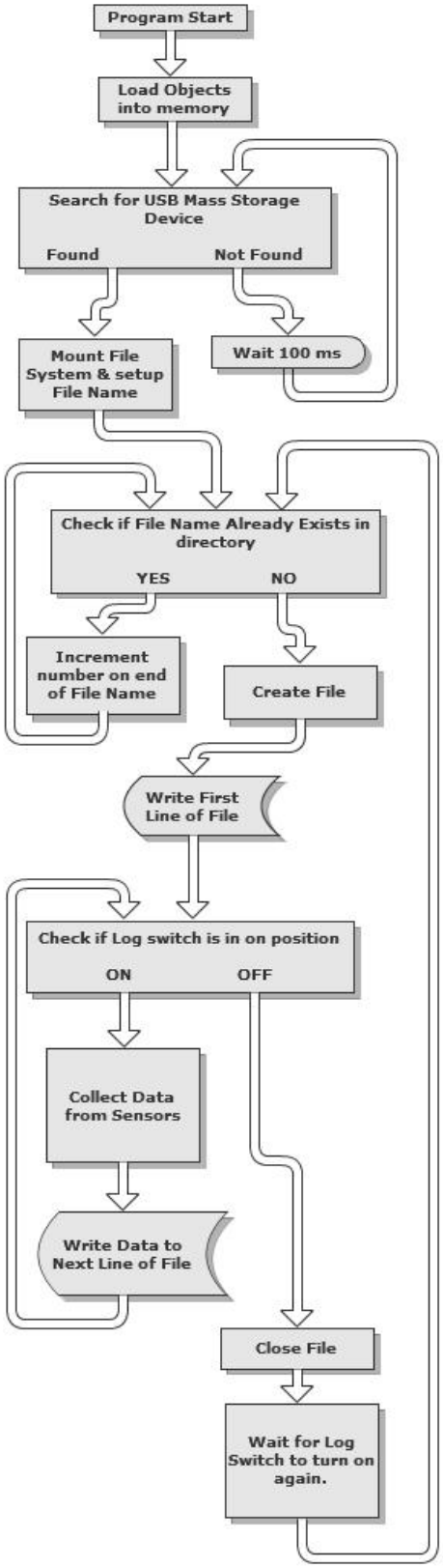


Figure 30: Program Flowchart

Conclusions and Recommendations

Using the engineering knowledge gained from the past four years spent at Worcester Polytechnic Institute the group was able to redesign the intake, pedal plate, and hubs for the 2009 FSAE racecar, while also designing a throttle body and exhaust. Much like years past, testing time was limited and did not allow the team time to test the car and determine if any other systems would fail and need to be remanufactured or redesigned to prevent future failures.

Intake

The intake was redesigned to eliminate the potential of poor fuel atomization and turbulent flow of the air before entering the engine. The new location of the fuel injectors will now spray the fuel right at the intake valves, which is ideal for a fuel injected engine. Removing the welded bends from the intake runners and replacing them with smooth round bends promotes laminar flow of air into the engine, which will help more effectively fill the cylinder**.** All of these design changes will greatly improve the performance of the racecar.

In order to optimize the intake runners and plenums the next step would be to dyno the car to measure the power, and then adjust the runner length and plenum volume and see how the changes affect the power and torque of the engine. Testing the car on a dynometer would give the team data regarding the optimum runner length and plenum volume. While doing the calculation to determine the intake runner length and plenum volume give a base line to start the design at testing different runner lengths and plenum volumes and monitoring the change in

horsepower and torque is the only way to validate the calculations. Testing different sized runners and plenums also allows the team to reduce any errors derived from estimations in the length of intake port in the head and any portion of the plenum not accounted for in the initial volume calculations. Dyno testing would have also proved beneficial if the car had been dynoed with the original intake setup to determine the gains from the redesigned intake runners and plenum. If the group was able to measure improvements from testing the different changes made to the intake, it could be very useful to future WPI FSAE teams when they need to design an intake system. Whenever data is available it can help a team determine how to approach the different systems of the car. If there were only small gains had from different runner lengths and plenum volumes then future teams could spend less time on an intake design and put more effort into another system of the car.

Throttle Body

The throttle body is another area of the car that could benefit greatly from dyno testing while changing the type of throttle body being used. If the team had time to machine a butterfly throttle body and spike throttle body and then dyno test each setup that data could provide valuable insight into the optimal throttle body design for a Formula SAE racecar. Since the power on the car is so limited any testing done to determine where more power can be made will make a large difference when a car is taken to competition. Also with more testing data design changes could be made to the throttle body to try and get a little more power from a throttle body design, by making minor changes to it.

Data Acquisition System

The Data Acquisition system was tested in a laboratory environment, however due to the inability to test the car in dynamic events, there was never a chance to test the DAQ on the car in race conditions. The Bourn's PTS potentiometers work well when combined with the Microchip MCP320X analog to digital converters under indoor testing, however they have not been able to be subjected to the conditions that will be met during the course of a dynamic event. All the computer simulation in the world cannot fully replace real world testing. Testing is the main recommendation for the DAQ, as it will expose any issues that may arise.

Going forward, The DAQ could easily evolve into a larger system with more than just data collection functions. Any functions on the car that cannot be directly controlled by the ECU, but require more than simple on/off control from the driver could be automated through the microcontroller in the DAQ. Additionally, more sensors could be added as there is plenty of room on the FEZ Rhino for expansion. Also, the topic of adding telemetry to the car was discussed, and then put on the back burner as it was not necessary for the collection of viable data. Telemetry would be a useful tool in the, as would some sort of wireless link to the ECU to allow engine tuning without having to stop the car in order to reduce downtime while making modifications to the engine calibration. As more functionality is added, the tasks may outgrow the computing hardware as the FEZ Rhino is only a 72 MHz processor. Upgrading to something along the lines of a PowerPC chip may be

necessary in order to maintain the frequency of data collection as well a controlling other functions.

Recommendations for Future Teams

Over the year the group learned many things that were not directly related to the sections of the car that were manufactured or redesigned. While these were not major breakthroughs, they were a lot of little things here and there that should not be over looked by future teams. One of the biggest issues the group repeatedly ran into was trying to mount little things here and there, which proved to be difficult. In future years the team should design mounts and locations for commonly overlooked parts and components such as catch cans, engine electronics, safety guards and the battery. Testing proved that the initial battery location was close to exhaust, which caused the battery casing to melt, resulting in battery charging issues. The engine orientation was another minor detail that can be easily overlooked, but can make a big difference when the car is getting assembled at the end of the year. Although the engine orientation on the current car is inconvenient, there are design constraints which require it to be in this orientation. However, other the orientation of other components and systems could have been altered to alleviate space constraints. The cockpit of the car is another section that can be easily overlooked when designing the car. To facilitate the removal of seat inserts, the harness should be able to be easily passed through the insert.

One final recommendation for future teams would be to consider all aspects of the car when in the design phase of the project. If all the systems are designed with consideration of one another it can help to make a better overall car. Designing

systems and components independently of each other may result in parts interfering with each other, causing last minute design changes, which can lead to bad engineering practices. This was a common occurrence in the completion of the racecar.

Appendix A: Data Acquisition System Code

Main Execution Program

```
using System;
using System.Threading;
using System.IO;
using System.Text;

using Microsoft.SPOT;
using Microsoft.SPOT.IO;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.USBHost;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.System;



namespace FSAE_DAQ_2
{
    public class Program
    {
        static CAN.Message[] msgList;
        static DummySensor rpm ;
        static DummySensor tps;
        static DummySensor map;
        static DummySensor AFR;
        static DummySensor ign;

        public static void Main()
        {

            string fileName = "FSAEDaq_log";
            int fileNumber = 001;

            //DAQ on off switch, connected to board pin#1, switching with ground.
            InputPort DAQ_ON = new InputPort(((Cpu.Pin)FEZ_Pin.Digital.IO62),
false, Port.ResistorMode.PullUp);
            OutputPort usbLight = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.IO60,
false);
            OutputPort DAQGood = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.IO64,
false);
            OutputPort DAQBAD = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.IO66,
false);

            //set error light & USB light during setup
            DAQBAD.Write(true);
            usbLight.Write(true);

            StreamWriter log;
            PersistentStorage usb;
```

```
            System.Collections.ArrayList sensors = new
System.Collections.ArrayList();

            //CANBUS sentup on CAN channel 1 for 1 Mbit/s
            int T1 = 15;
            int T2 = 8;
            int BRP = 3;//change this number to change clock rate  3 = 1Mbit/s; 6
= 500 Kbit/s
            CAN can = new CAN(CAN.Channel.Channel_1, (uint)(((T2 - 1) << 20) |
((T1 - 1) << 16) | ((BRP - 1) << 0)));

            //initialize CAN message array
            msgList = new CAN.Message[100];
            for (int i = 0; i < msgList.Length; i++)
            {
                msgList[i] = new CAN.Message();
            }

            can.DataReceivedEvent += new
CANDataReceivedEventHandler(can_DataReceivedEvent);
            can.ErrorReceivedEvent += new
CANErrorReceivedEventHandler(can_ErrorReceivedEvent);




            //setup sensors
            sensors.Add(new clock());
            //either use GPS or the bruteforce clock settting... if the GPS is
working you get accurate time from the satellites to set the RTC


            //sensors.Add(new GePeS());
            RealTimeClock.SetTime(new DateTime(2011, 4, 10, 12, 0, 0));

            //MCP3204, CS pin on IO51, board pin#9
            sensors.Add(new MCPSensor("X-axis Accel",
(Cpu.Pin)FEZ_Pin.Digital.IO51, 0x00, 1, 0));
            sensors.Add(new MCPSensor("Y-axis Accel",
(Cpu.Pin)FEZ_Pin.Digital.IO51, 0x01, 1, 0));
            sensors.Add(new MCPSensor("Z-axis Accel",
(Cpu.Pin)FEZ_Pin.Digital.IO51, 0x02, 1, 0));
            sensors.Add(new MCPSensor("Gyro VREF", (Cpu.Pin)FEZ_Pin.Digital.IO51,
0x03, 1, 0));

            //MCP3204, CS pin on IO52, board pin#11
            sensors.Add(new MCPSensor("X-axis Gyro",
(Cpu.Pin)FEZ_Pin.Digital.IO52, 0x00, 1, 0));
            sensors.Add(new MCPSensor("X-axis Gyro, precision",
(Cpu.Pin)FEZ_Pin.Digital.IO52, 0x01, 1, 0));
            sensors.Add(new MCPSensor("Z-axis Gyro",
(Cpu.Pin)FEZ_Pin.Digital.IO52, 0x02, 1, 0));
            sensors.Add(new MCPSensor("Z-axis Gyro, precision",
(Cpu.Pin)FEZ_Pin.Digital.IO52, 0x03, 1, 0));

            //MCP3208, CS pin on IO53, board pin#13
```

```
            MCPSensor LF = new MCPSensor("Left Front Suspension Position",
(Cpu.Pin)FEZ_Pin.Digital.IO53,0x00, 1, 0);
            MCPSensor RF = new MCPSensor("Right Front Suspension Position",
(Cpu.Pin)FEZ_Pin.Digital.IO53, 0x01, 1, 0);
            MCPSensor LR = new MCPSensor("Left Rear Suspension Position",
(Cpu.Pin)FEZ_Pin.Digital.IO53, 0x02, 1, 0);
            MCPSensor RR = new MCPSensor("Right Rear Suspension Position",
(Cpu.Pin)FEZ_Pin.Digital.IO53, 0x03, 1, 0);
            MCPSensor steering = new MCPSensor("Steering Position",
(Cpu.Pin)FEZ_Pin.Digital.IO53, 0x04, 1, (float)-2047.0);

            //set position offsets to current values, accounting for different
static positions due to alignment & driver/car weight
            LF.offset = LF.rawRead();
            LR.offset = LR.rawRead();
            RF.offset = RF.rawRead();
            RR.offset = RR.rawRead();

            //Add suspension sensors to sensors ArrayList
            sensors.Add(LF);
            sensors.Add(LR);
            sensors.Add(RF);
            sensors.Add(RR);
            sensors.Add(steering);


            //tire temperature sensors.
            sensors.Add(new MLX_Temp("LF outer temp", 0x010, 100, 1, 0));
            sensors.Add(new MLX_Temp("LF middle temp", 0x011, 100, 1, 0));
            sensors.Add(new MLX_Temp("LF inner temp", 0x012, 100, 1, 0));
            sensors.Add(new MLX_Temp("RF inner temp", 0x013, 100, 1, 0));
            sensors.Add(new MLX_Temp("RF middle temp", 0x014, 100, 1, 0));
            sensors.Add(new MLX_Temp("RF outer temp", 0x015, 100, 1, 0));



            //CAN-Data sensors, updated automatically by CAN received event.
            rpm = new DummySensor("RPM", 1, 0);
            tps = new DummySensor("TPS %", (float)0.1, 0);
            map = new DummySensor("MAP mBar", 1, 0);
            AFR = new DummySensor("AFR, Lambda", (float)0.001, 0);
            ign = new DummySensor("ignition advance", (float)0.1, 0);


            //detect storage devices
            Debug.Print("power on \ndetecting USB storage devices");
            Thread.Sleep(10);
            int c = 0;
            while(USBHostController.GetDevices().Length == 0)
            {
                //**set error light, wait for USB insertion.
                Debug.Print("Working..." + c);
                c++;
                Thread.Sleep(100);
            }
            //**turn off error light if set
            usbLight.Write(false);
```

```
            //mount USB file system
            Debug.Print("USB Storage Device found");
            usb = new PersistentStorage(USBHostController.GetDevices()[0]);
            Thread.Sleep(25);
            usb.MountFileSystem();
            Debug.Print("File System mounted");
            Thread.Sleep(10);


            //setup file(s)
            string root = VolumeInfo.GetVolumes()[0].RootDirectory;
            Thread.Sleep(10);

            Debug.Print("Scanning Files");

            String nextLine = "";



        fileStart:

            nextLine = "";
            for (int i = 0; i < sensors.Count; i++)
            {
                nextLine += ((Sensor)sensors[i]).Name() + ", ";
            }
            nextLine += rpm.Name() + ", ";
            nextLine += tps.Name() + ", ";
            nextLine += map.Name() + ", ";
            nextLine += AFR.Name() + ", ";
            nextLine += ign.Name() + ", ";

            while (File.Exists(root + @"\" + fileName + fileNumber +
".csv"))//check to see if current file number exists
            {
                Debug.Print("File " + fileName + fileNumber + ".csv found.");
                fileNumber++; //if it exists, increment the number
            }
            Debug.Print("Creating File " + fileName + fileNumber + ".csv");
            log = new StreamWriter(root + @"\" + fileName + fileNumber + ".csv",
false);


            log.WriteLine(nextLine);
            Debug.Print(nextLine);


            //check to see if time is valid, if no, set from GPS
            while(RealTimeClock.GetTime().Year < 2010)
            {
                RealTimeClock.SetTime(FEZ_Extensions.GPS.GetUTCDateTime());
            }
            DAQBAD.Write(false);


            //sample sensors 10 times, saving data to logfile in .csv format
```

```
            while(DAQ_ON.Read()){
                DAQGood.Write(true);
                nextLine = "";
                for (int j = 0; j < sensors.Count; j++)
                {
                    nextLine += ((Sensor)sensors[j]).Read() + ", ";

                }

                //CANBUS dummy sensors
                nextLine += rpm.Read() + ", ";
                nextLine += tps.Read() + ", ";
                nextLine += map.Read() + ", ";
                nextLine += AFR.Read() + ", ";
                nextLine += ign.Read() + ", ";

                Debug.Print(nextLine);
                log.WriteLine(nextLine);
            }
            DAQGood.Write(false);
            log.Close();
            log.Dispose();

            //if switch is flipped to stop log...
            while (!DAQ_ON.Read()){
                DAQBAD.Write(true);

                //sit here doing nothing, waiting for switch to go live again.

            }
            //when the switch is turned on again, go back to the file start and
get ready to log again.
            goto fileStart;



            }

        static void can_DataReceivedEvent(CAN sender, CANDataReceivedEventArgs
args)
        {
            Debug.Print(">>> can_DataReceivedEvent <<<");

            // read as many messages as possible
            int count = sender.GetMessages(msgList, 0, msgList.Length);
            for (int i = 0; i < count; i++)
            {

                //IF the can message matches those expected from the Haltech ECU,
parse the data to thge necessary DummySensors.
                Debug.Print("MSG: ID = " + msgList[i].ArbID.ToString("x") + " at
time = " + msgList[i].TimeStamp);
                if (msgList[i].ArbID == 0x0010)
                {
                    rpm.setValue((float)((msgList[i].Data[0] << 8) |
(msgList[i].Data[1])));
                    Debug.Print("RPM: " + rpm.Read());
```

```
                }
                else if (msgList[i].ArbID == 0x0011)
                {
                    tps.setValue((float)((msgList[i].Data[6] << 8) |
(msgList[i].Data[7])));
                    Debug.Print("TPS: " + tps.Read());
                }
                else if (msgList[i].ArbID == 0x0012)
                {
                    map.setValue((float)((msgList[i].Data[0] << 8) |
(msgList[i].Data[1])));
                    AFR.setValue((float)((msgList[i].Data[6] << 8) |
(msgList[i].Data[7])));
                    Debug.Print("MAP: " + map.Read());
                    Debug.Print("AFR: " + AFR.Read());
                }
                else if (msgList[i].ArbID == 0x0013)
                {
                    ign.setValue((float)((msgList[i].Data[0] << 8) |
(msgList[i].Data[1])));
                    Debug.Print("IGN Advance: " + ign.Read());
                }
                else
                {

                }


            }
        }

        static void can_ErrorReceivedEvent(CAN sender, CANErrorReceivedEventArgs
args)
        {
            Debug.Print(">>> can_ErrorReceivedEvent <<<");

            switch (args.Error)
            {
                case CAN.Error.Overrun:
                    Debug.Print("Overrun error. Message lost");
                    break;

                case CAN.Error.RXOver:
                    Debug.Print("RXOver error. Internal buffer is full. Message
lost");
                    break;

                case CAN.Error.BusOff:
                    Debug.Print("BusOff error. Reset CAN controller.");
                    sender.Reset();
                    break;
            }
        }



        }
```

```
    }
```

Sensor Interface and Classes:

```
using System;
using System.Threading;

using Microsoft.SPOT;
using Microsoft.SPOT.IO;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.USBHost;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.Hardware;
using System.IO;

namespace FSAE_DAQ_2
{
    interface Sensor
    {
        string Name();
        string Read();
    }

    class MCPSensor : Sensor
    {
        string name;
        public float scalar = 1;
        public float offset = 0;
        public byte channel = 0;
        public Cpu.Pin chipSelect = (Cpu.Pin) FEZ_Pin.Digital.IO10;

        const bool csActive = false;
        const bool clockEdge = true;
        const bool clockIdle = false;
        const UInt16 clockRate = 2000;
        const UInt16 csSetup = 0;
        const UInt16 csHold = 0;

        public MCPSensor(Cpu.Pin cs)
        {
            name = "";
            chipSelect = cs;
        }

        public MCPSensor(string n, Cpu.Pin cs)
        {
            name = n;
            chipSelect = cs;
        }

        public MCPSensor(string n,Cpu.Pin cs, float s, float o)
```

```
        {
            name = n;
            scalar = s;
            offset = o;
            chipSelect = cs;
        }

        public MCPSensor(string n, Cpu.Pin cs, byte chan, float s, float o)
        {
            name = n;
            scalar = s;
            offset = o;
            channel = chan;
            chipSelect = cs;
        }

        public string Name()
        {
            return name;
        }

        public string Read()
        {
            return (((float) rawRead()) * scalar + offset).ToString() ;
        }

        public int rawRead()
        {
            byte one = (byte) ((channel >> 2) | 0x06);
            byte two = (byte) ((channel << 6) & 0xB0);
            int output = 0x0000;

            byte[] write = {one, two, 0x00};
            byte[] read = {0x00 , 0x00, 0x00};

            SPI.Configuration conf = new SPI.Configuration(chipSelect, csActive,
csSetup, csHold, clockIdle, clockEdge, clockRate, SPI.SPI_module.SPI1);
            SPI spi = new SPI(conf);

            spi.WriteRead(write, read);


            //debugging print stuff, delete once it is confirmed working.
            for (int i = 0; i < read.Length; i++)
            {
                Debug.Print("byte " + i + " = " + read[i] + " \n");
            }

            output = output | (((UInt16)read[1]) << 8);
            output = output | (((UInt16)read[2]));

            spi.Dispose();

            return output;
        }
```

```
    }

    class MLX_Temp : Sensor
    {
        string name;
        public float scalar = 1;
        public float offset = 0;
        byte slaveAddress = 0x5A; //slave adress, default for MLX IR temp sensor
is 0x5A
        int clock = 50; //clock rate in KHz
        I2CDevice.Configuration config;

        public MLX_Temp()
        {
            name = "MLX_IR Temp";
            config = new I2CDevice.Configuration(slaveAddress, clock);
        }

        public MLX_Temp(byte sa, int clk)
        {
            slaveAddress = sa;
            clock = clk;
            config = new I2CDevice.Configuration(slaveAddress, clock);
        }

        public MLX_Temp(String n, byte sa, int clk, float scale, float o)
        {
            name = n;
            slaveAddress = sa;
            clock = clk;
            scalar = scale;
            offset = 0;
            config = new I2CDevice.Configuration(slaveAddress, clock);
        }

        public String Name()
        {
            return name;
        }

        /**
         * reads the temperature in Celsius.
         */
        public String Read()
        {
            I2CDevice bus = new I2CDevice(config);
            byte[] wr = new byte[1];
            byte[] rd = new byte[3];

            wr[0] = 0x07;//RAM address 0x07.

            I2CDevice.I2CTransaction[] send = new I2CDevice.I2CTransaction[2];
            send[0] = I2CDevice.CreateWriteTransaction(wr);
            send[1] = I2CDevice.CreateReadTransaction(rd);
```

```
            bus.Execute(send,10);//turn down the numer if this works, turn it up
if it doesn't work...

            uint reading = (uint)((((byte)rd[1]) << 8) | (((byte) rd[0])));
            bus.Dispose();

            return ((float)((double)reading * 0.02) -272.0).ToString();

        }

        /**
         * WARNING!  DO NOT USE THE FOLLOWING METHOD EVER! IT WILL SERIOUSLY BREAK
THINGS IF YOU AREN'T CAREFUL
         * IF ANY MLX90614 SENSORS ARE CONNECTED, THEY WILL ALL HAVE THEIR SLAVE
ADDRESSES RESET TO THE HARD
         * CODED VALUE.  ALSO, A DIFFERENT 4TH BIT NEEDS TO BE CALCULATED FOR EACH
NEW SLAVE ADDRESS
         *
         * IF YOU HAVE MORE THAN ONE MLX90614 CONNECTED, THEY WILL ALL BE SET TO
THE NEW SLAVE ADDRESS
         *
         * THUS, THINGS WON'T WORK RIGHT SINCE YOU'LL BE TALKING TO ALL THE
MLX90614 SENSORS AT THE SAME TIME,
         * ON THE SAME I2C BUS.  THIS WILL RUIN YOUR LIFE.  DON'T DO IT. EVER.
         *
         * YOU HAVE BEEN WARNED.  ALSO, DON'T LET LEAFY ANYWHERE NEAR THIS CODE.
EVER.
         */
        public void resetSlaveAddress(byte s)
        {
            I2CDevice.Configuration test = new I2CDevice.Configuration(0x0000,
50); //ALL MLX temp sensors should answer to 0x00 according to data sheet.
            I2CDevice bus = new I2CDevice(test);
            byte[] wr = new byte[4];
            byte[] rd = new byte[4];
            I2CDevice.I2CTransaction[] send;

            Debug.Print("YOU MAY HAVE BROKEN A BUNCH OF STUFF JUST NOW....");
            Debug.Print("SERIOUSLY, THIS METHOD MAY BREAK THINGS!");


            wr[0] = (byte) 0x2e;//eeprom slave address write access
            wr[1] = 0x00;//clear slave address to zero
            wr[2] = 0x00;
            wr[3] = 0x6F;//crc-8 for 0x2e0000
            send = new I2CDevice.I2CTransaction[1];
            send[0] = I2CDevice.CreateWriteTransaction(wr);
            bus.Execute(send, 10);

            Thread.Sleep(50);

            wr[0] = (byte)0x2e;//eeprom slave address write access
            wr[1] = 0x16;//set slave address to 0x16
            wr[2] = 0x00;
            wr[3] = 0x46;//crc-8 for 0x2e1600
            send = new I2CDevice.I2CTransaction[1];
            send[0] = I2CDevice.CreateWriteTransaction(wr);
            bus.Execute(send,10);
```

```
            Thread.Sleep(50);

            bus.Dispose();


        }

        /**
         * This method won't break anything, but it will return garbage if more
than one MLX90614 is connected.
         */
        public void getSlaveAddress()
        {
            I2CDevice.Configuration test = new I2CDevice.Configuration(0x0000,
50); //ALL MLX temp sensors should answer to 0x00 according to data sheet.
            I2CDevice bus = new I2CDevice(test);
            byte[] wr = new byte[3];
            byte[] rd = new byte[3];
            I2CDevice.I2CTransaction[] send;
            wr = new byte[1];
            wr[0] = (byte)0x2e;
            send = new I2CDevice.I2CTransaction[2];
            send[0] = I2CDevice.CreateWriteTransaction(wr);
            send[1] = I2CDevice.CreateReadTransaction(rd);
            bus.Execute(send, 10);
            Debug.Print("new Slave Adress is: " + rd[0]);

            bus.Dispose();
        }




    }

    class DummySensor : Sensor
    {
        String name = "dummy sensor";
        float value = 0;
        float scalar = 1;
        float offset = 0;

        public DummySensor()
        {
        }

        public DummySensor(String n, float s, float o)
        {
            name = n;
            scalar = s;
            offset = o;
        }

        public String Name()
        {
            return name;
        }
```

```
        public String Read()
        {
            return ((float)value * scalar + offset).ToString();
        }

        public void setValue(float v)
        {
            value = v;
        }

        public void setValue(UInt16 v)
        {
            value = (float)v;
        }




    }

    class clock : Sensor
    {
        public String name = "Time";
        DateTime t;

        public clock()
        {
        }

        public String Read()
        {
            t = RealTimeClock.GetTime();
            return (t.Hour + ":" + t.Minute + ":" + t.Second + "." +
t.Millisecond);
        }
        public String Name()
        {
            return name;
        }

    }

    class GePeS : Sensor
    {
            int LatH, LonH;
            double LatM, LonM;
            bool North, East;
            String pos;

            public GePeS()
            {
                FEZ_Extensions.GPS.Initialize();
            }

            public String Name()
            {
                return "GPS Location";
```

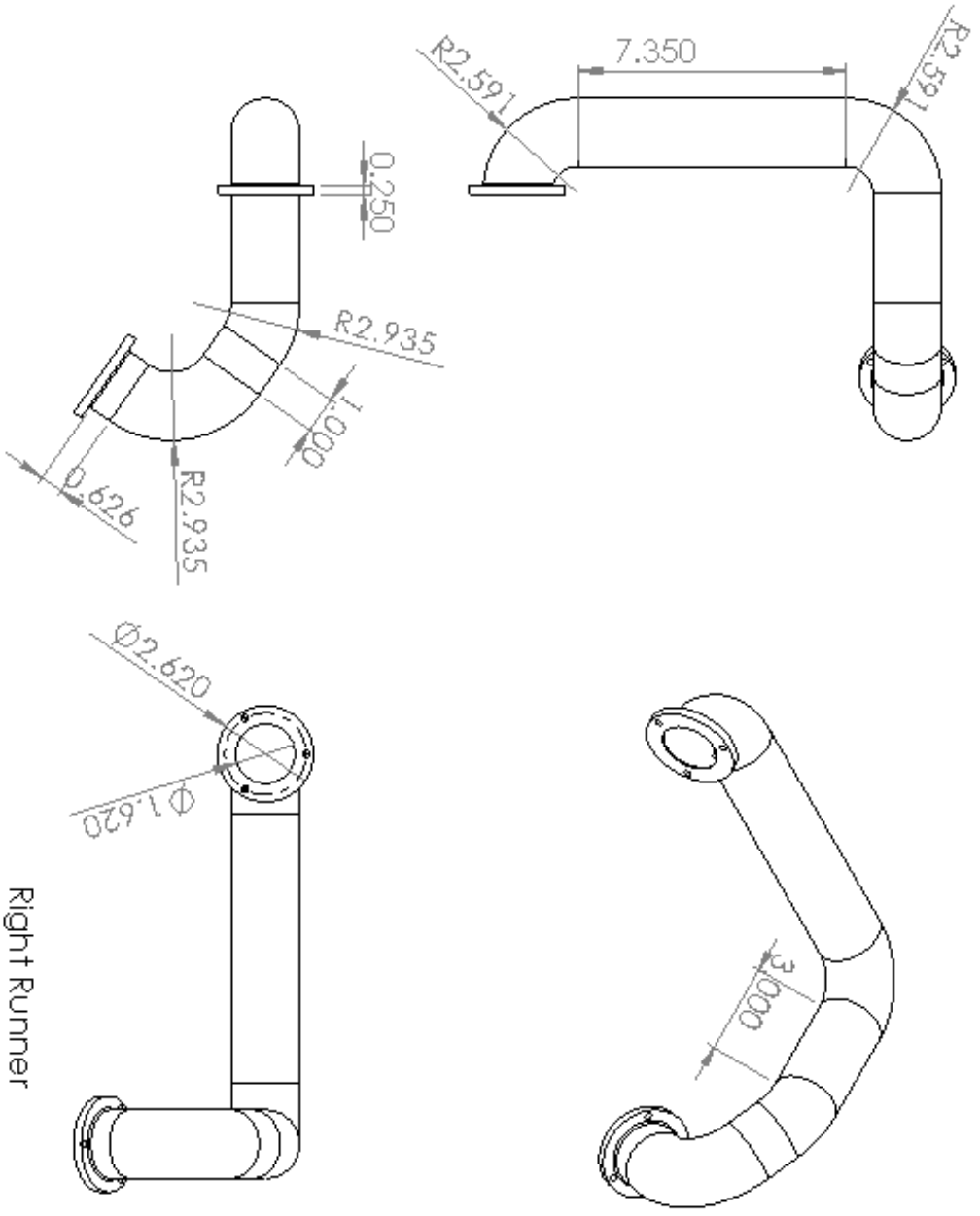
```
            }

            public String Read()
            {

                FEZ_Extensions.GPS.GetPositionF(out LatH, out LatM, out North, out
LonH, out LonM, out East);
                pos = LatH + " " + LatM + " N " + LonH + " " + LonM + " W";
                return pos;

            }

    }


}
```

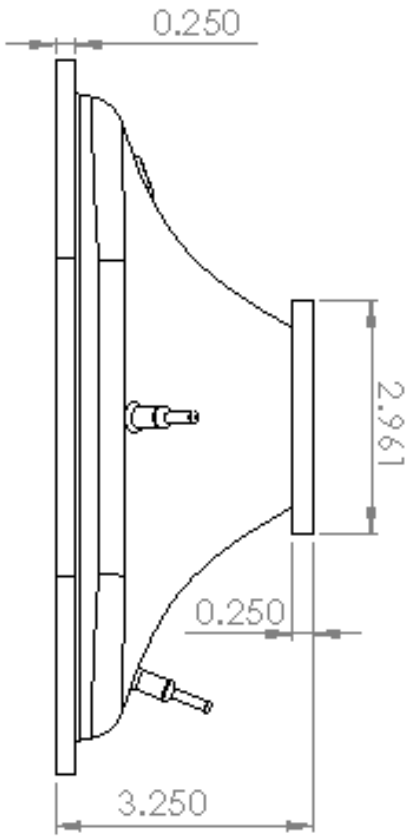
Modified Methods for GPS Driver Class:

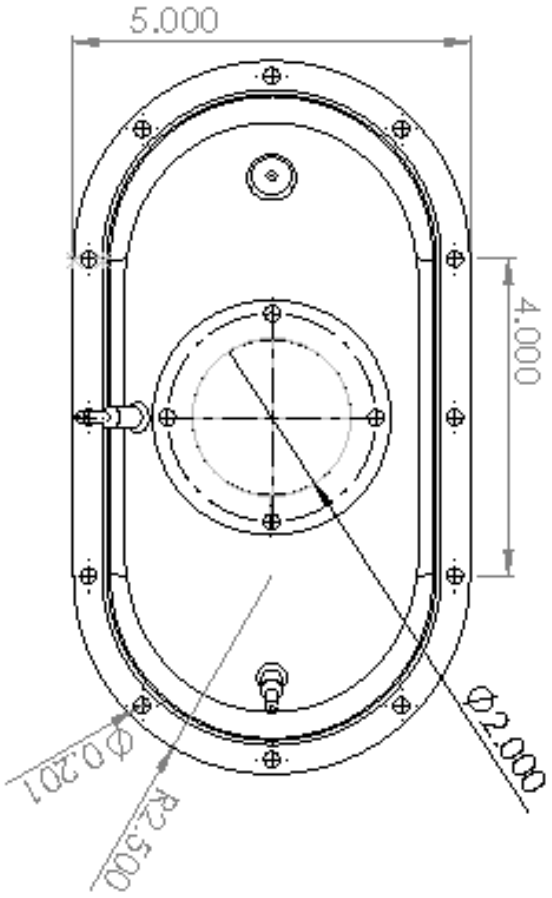
```
            public static bool GetPositionF(out int LatHour, out double LatMinute,
out bool North, out int LongHour, out double LongMinute, out bool East)
            {
                lock (GPRMC_sentence_array)
                {
                    if (_data_is_valid == false)
                    {
                        // nothign to return
                        LatHour = LongHour = 0;
                        LatMinute  = LongMinute = 0;
                        North = East = false;
                        return false;
                    }

                    string sentence_string = new string(GPRMC_sentence_array, 0,
GPRMC_sentence_array_length);
                    sentence_fields = sentence_string.Split(split_char);
                    if (sentence_fields.Length < 9)
                    {
                        // nothign to return
                        LatHour = LongHour = 0;
                        LatMinute =  LongMinute = 0;
                        North = East = false;
                        return false;
                    }

                    // compute values from ASCII
                    LatHour = GetValue2(sentence_fields[3], 0);
                    LatMinute = GetValue2F(sentence_fields[3], 2);
                    if (sentence_fields[4][0] == 'S')
                        North = false;
                    else
                        North = true;
                    LongHour =  GetValue3(sentence_fields[5], 0);
                    LongMinute = GetValue2F(sentence_fields[5], 3);
                    if (sentence_fields[6][0] == 'W')
                        East = false;
```

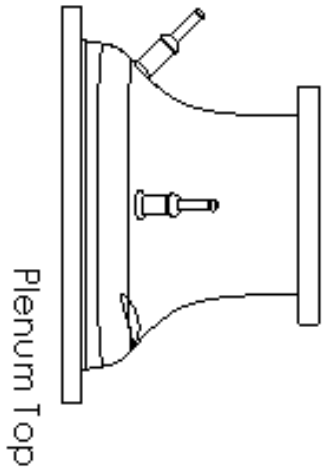
```
                    else
                        East = true;

                    _data_is_valid = false;
                    return true;
                }
            }

private static double GetValue2F(string str, int index)
            {
                double ret;
                ret = str[index] - '0';
                ret *= 10;
                ret += str[index + 1] - '0';
                ret += ((str[index + 3] - '0') * 0.1);
                ret += ((str[index + 4] - '0') * 0.01);
                ret += ((str[index + 5] - '0') * 0.001);
                ret += ((str[index + 6] - '0') * 0.0001);
                return ret;
            }
```

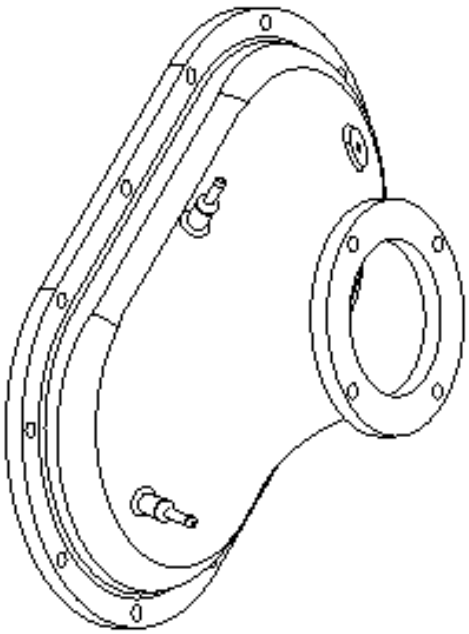
R2.935
4.570
R2.935
Φ2.620
Φ1.620
Φ0.201
2.500
4.570
Left Runner

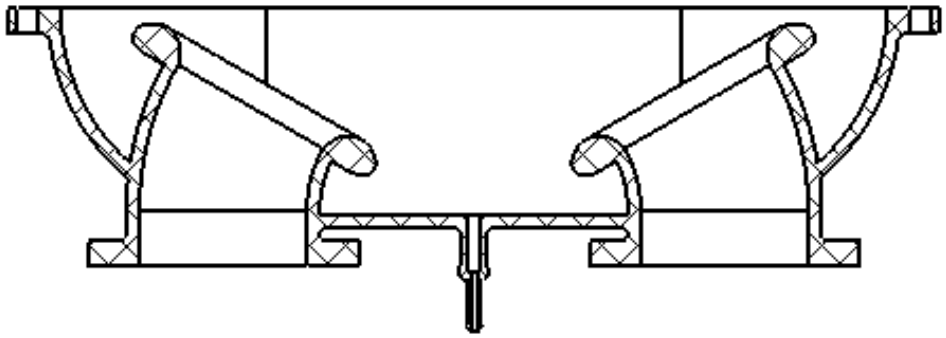

R2.591
7.350
R2.591
0.250
R2.935
1.000
0.626
R2.935
Ø2.620
Ø1.620
3.000
Right Runner


0.250
2.961
0.250
3.250

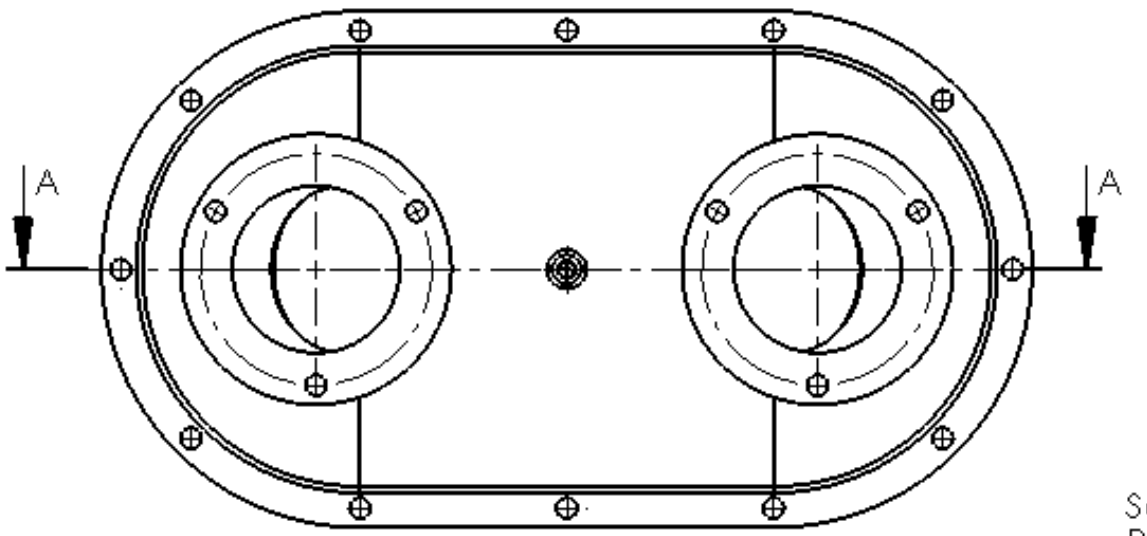

5.000
4.000
Ø2.000
Ø0.201
R2.500


Plenum Top

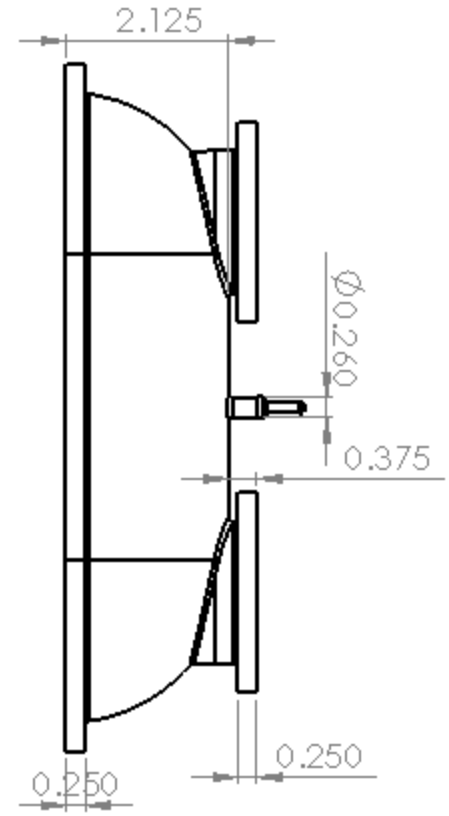


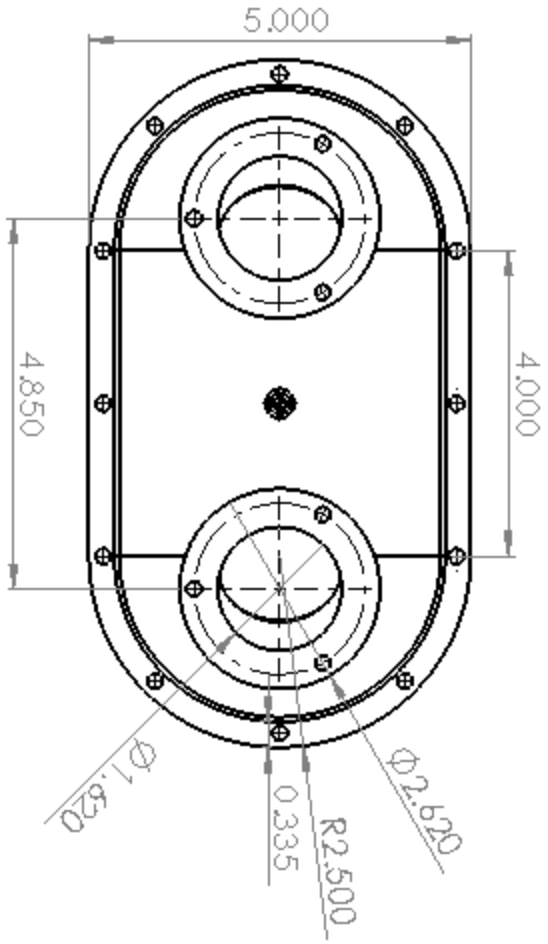


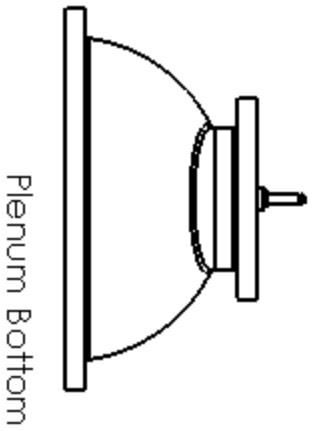
SECTION A-A

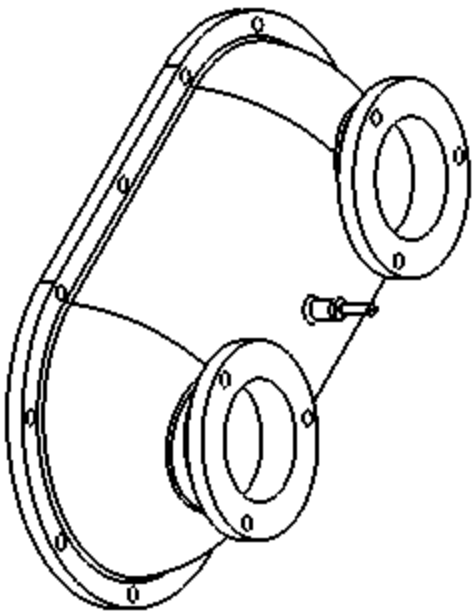


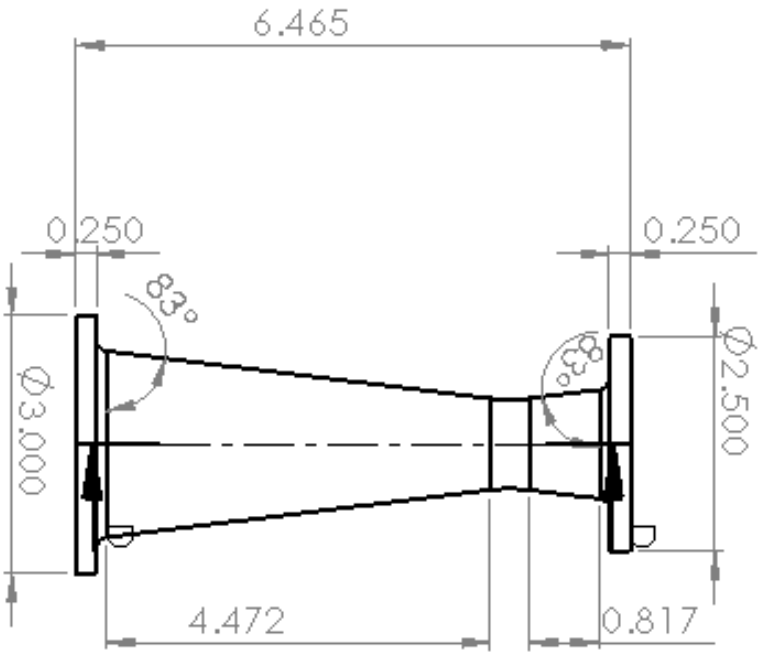
Section View of
Plenum Bottom


2.125
Ø0.260
0.375
0.250
0.250

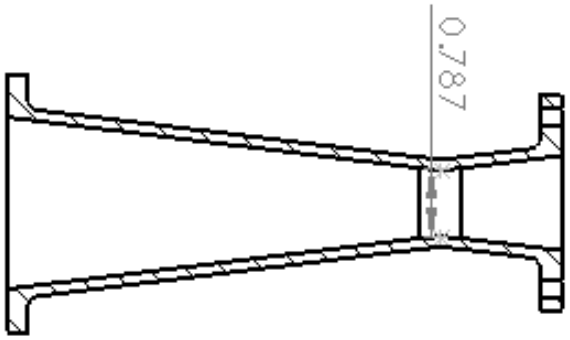

5.000
4.850
4.000
Ø1.620
0.335
R2.500
Ø2.620


Plenum Bottom

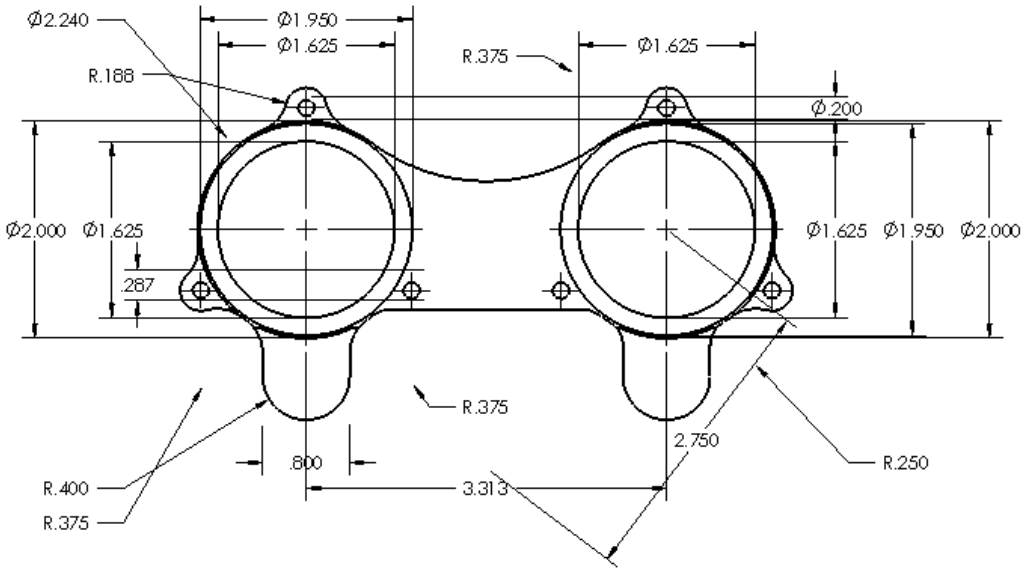


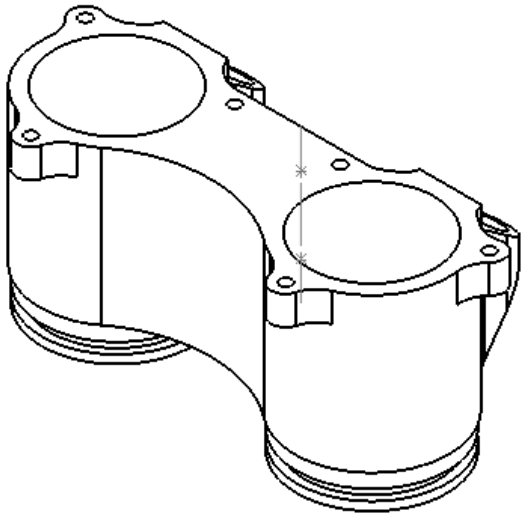


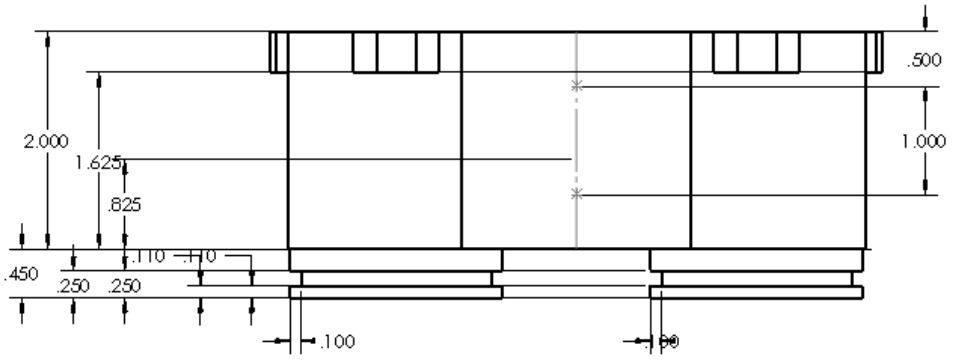
Section View of Restrictor

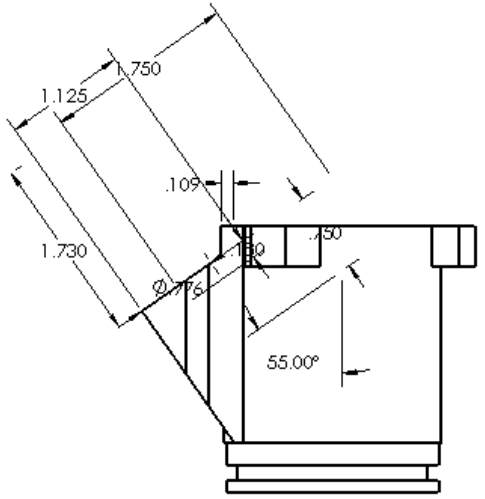


SECTION D-D

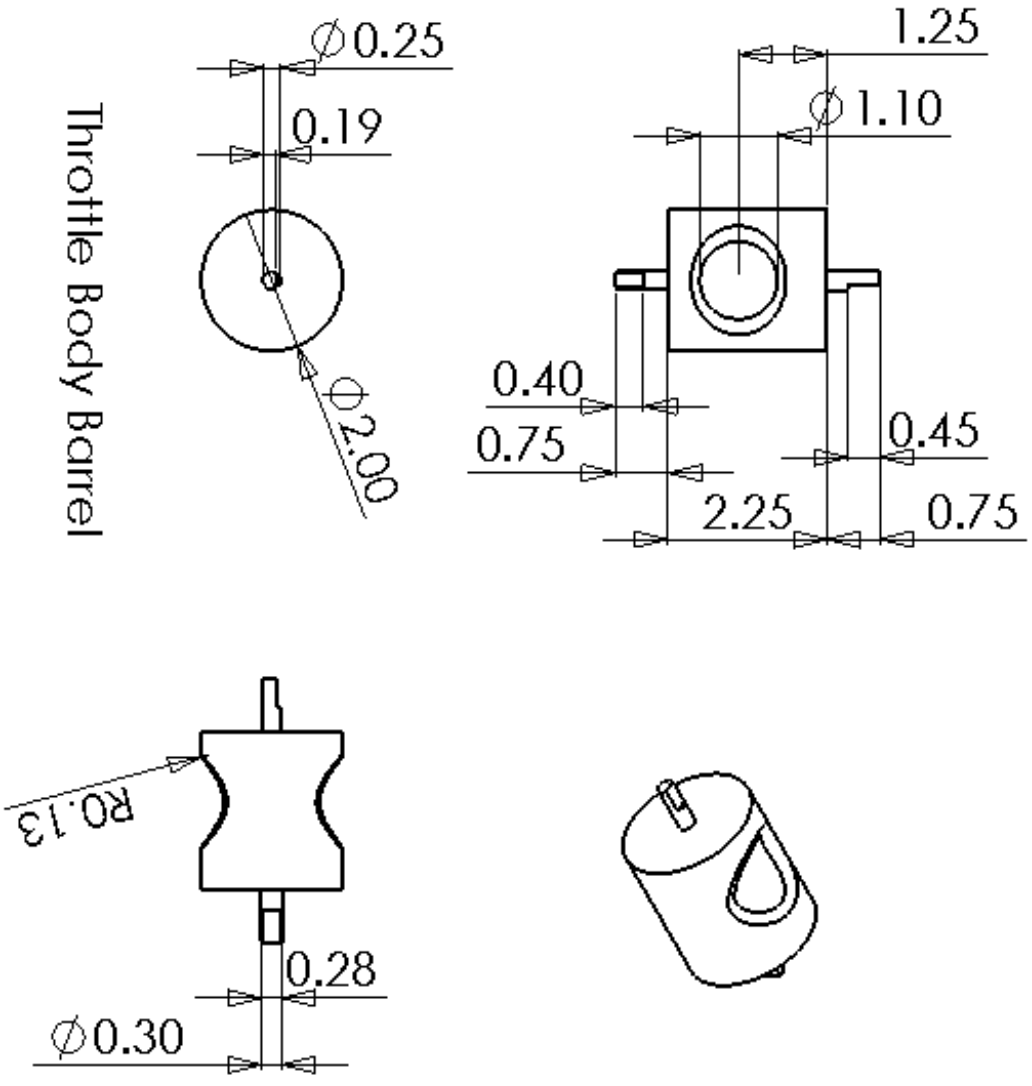


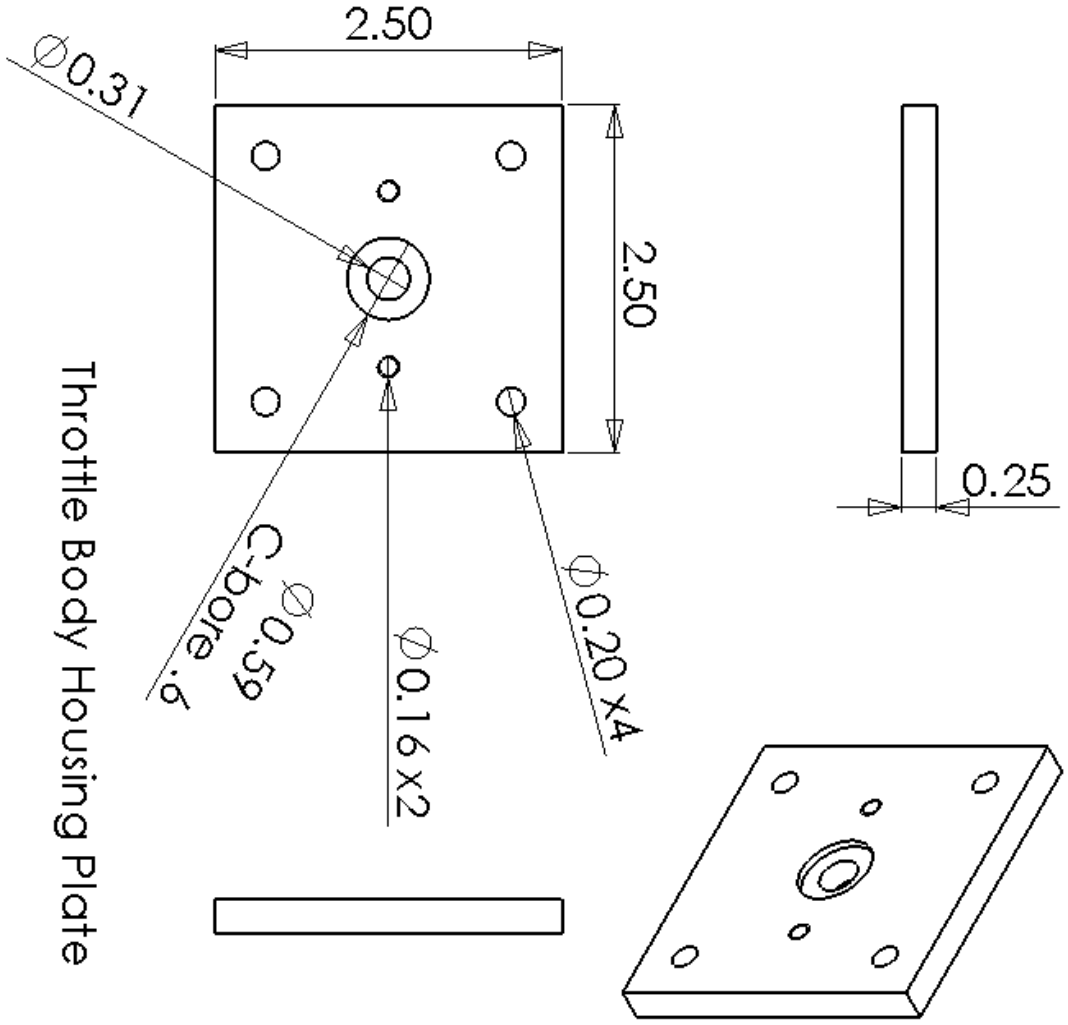


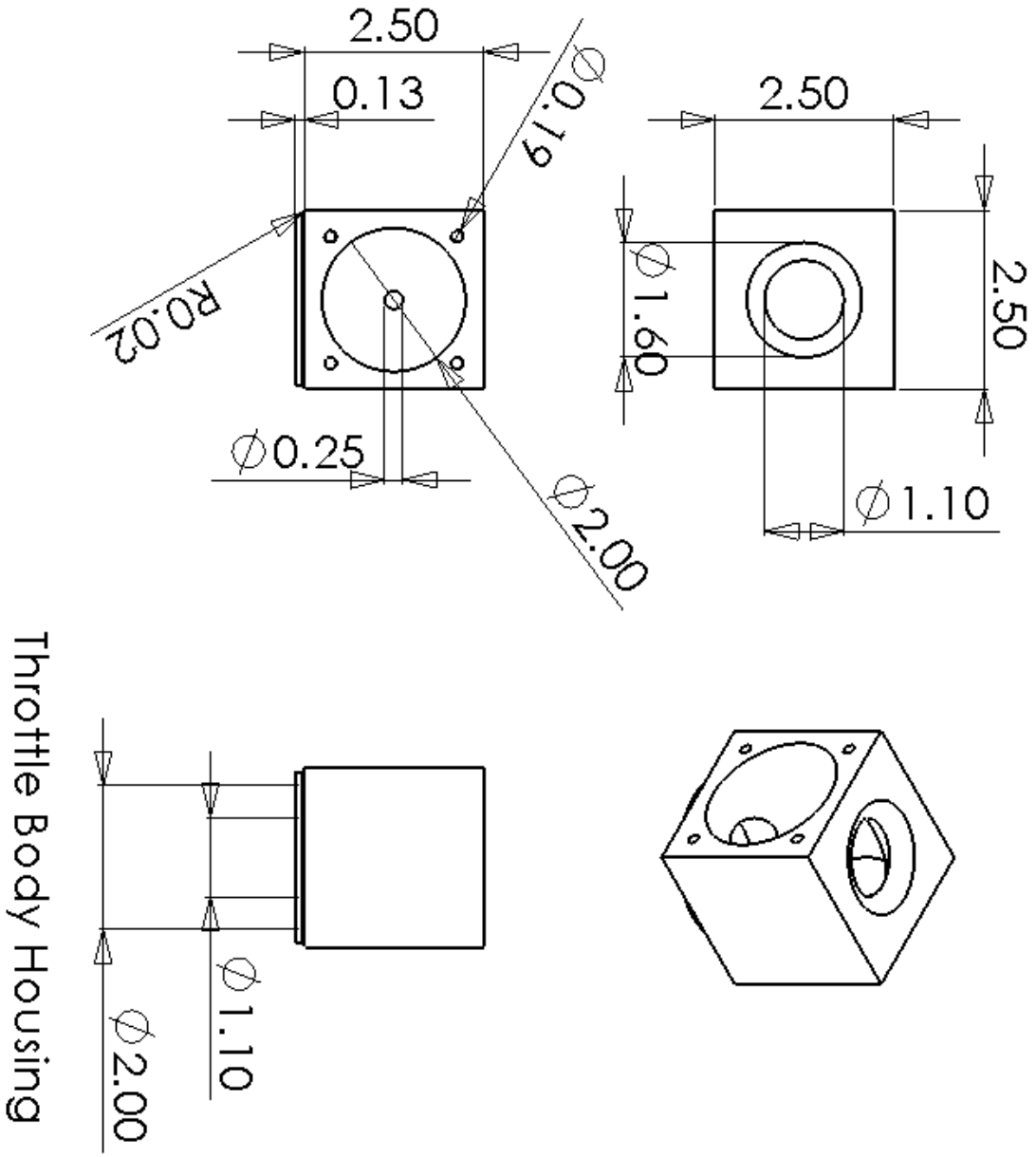




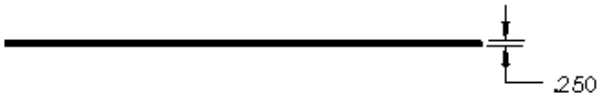
Injector Manifold

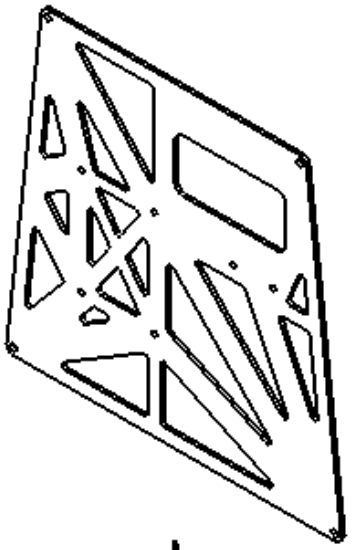

Throttle Body Barrel
∅0.25
0.19
∅2.00
1.25
∅1.10
0.40
0.75
0.45
2.25
0.75
R0.13
0.28
∅0.30

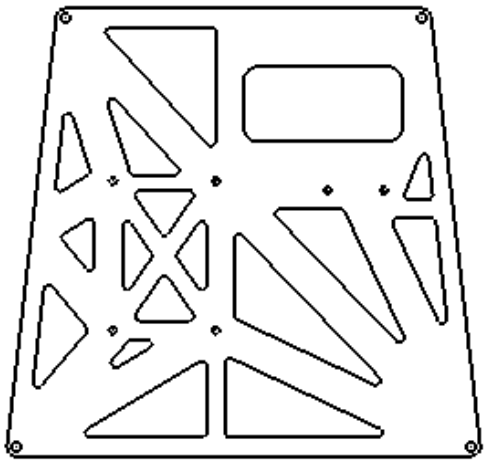

2.50
Ø0.31
2.50
0.25
C-bore Ø0.56 x.5
Ø0.16 x2
Ø0.20 x4
Throttle Body Housing Plate



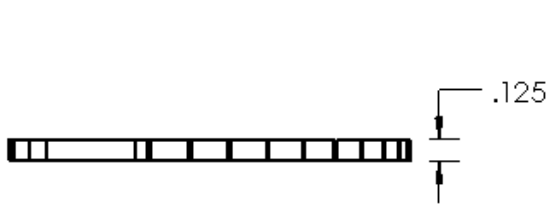
Throttle Body Housing

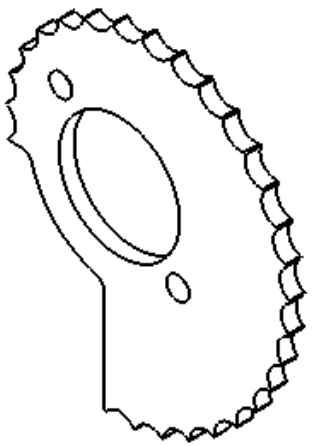


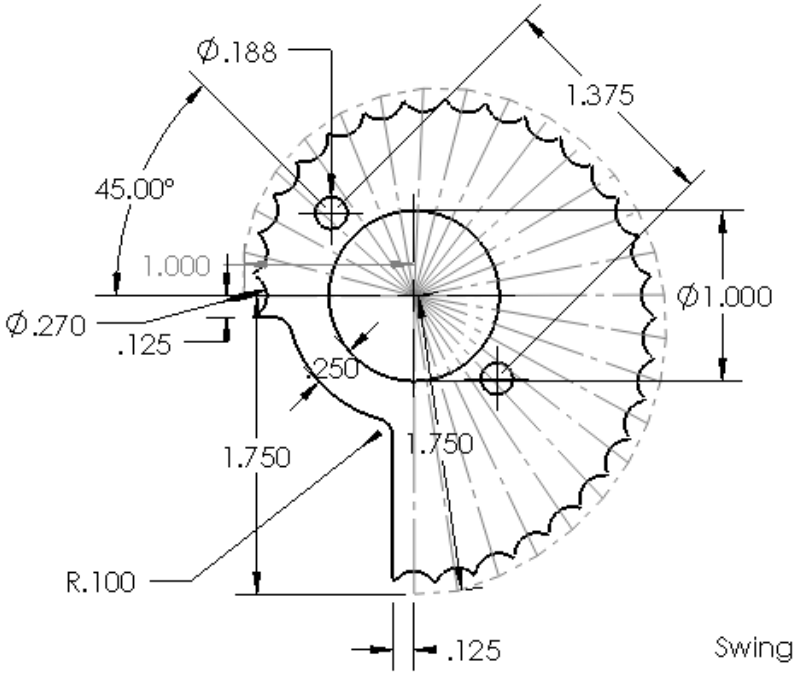




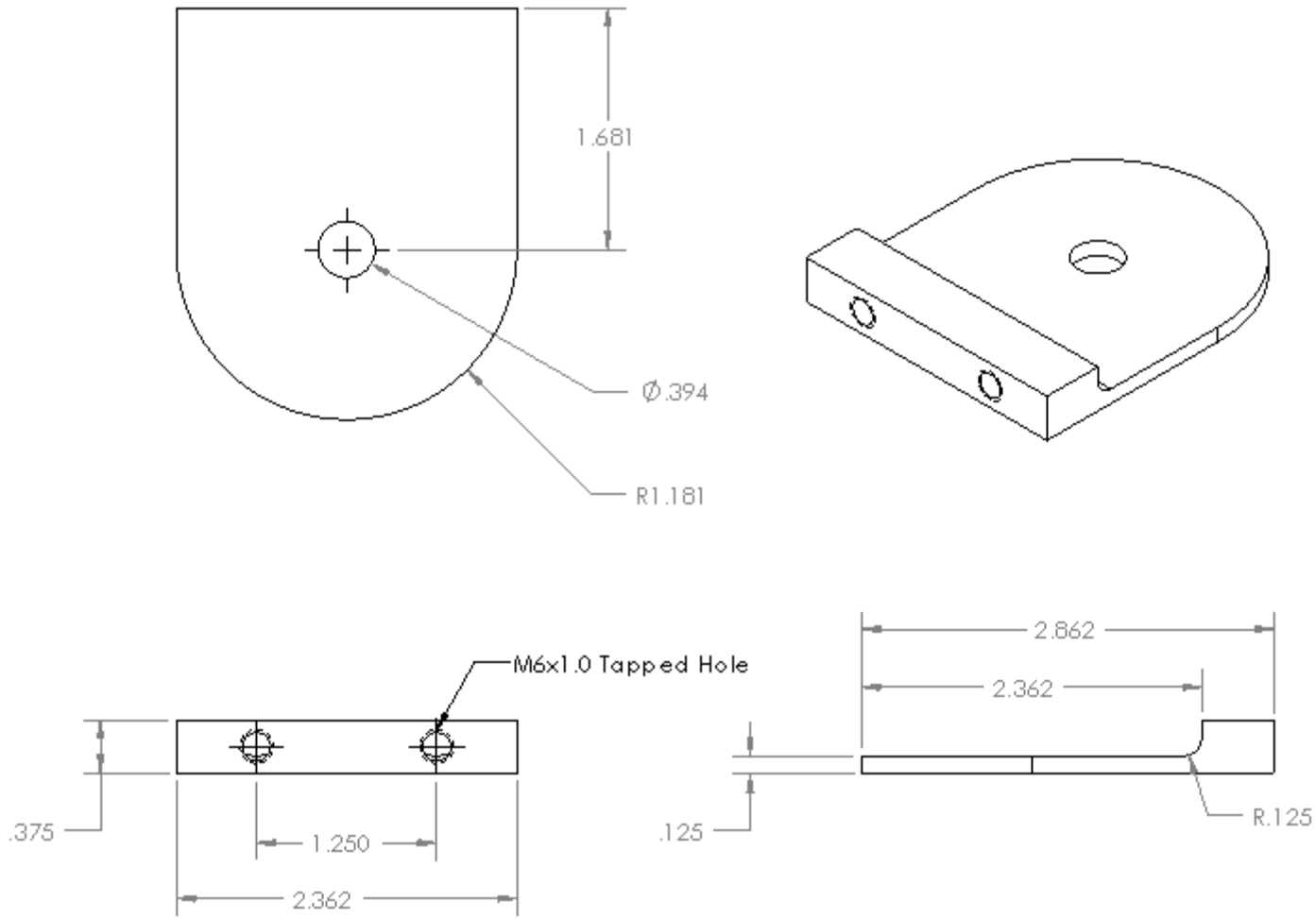
Pedal Plate



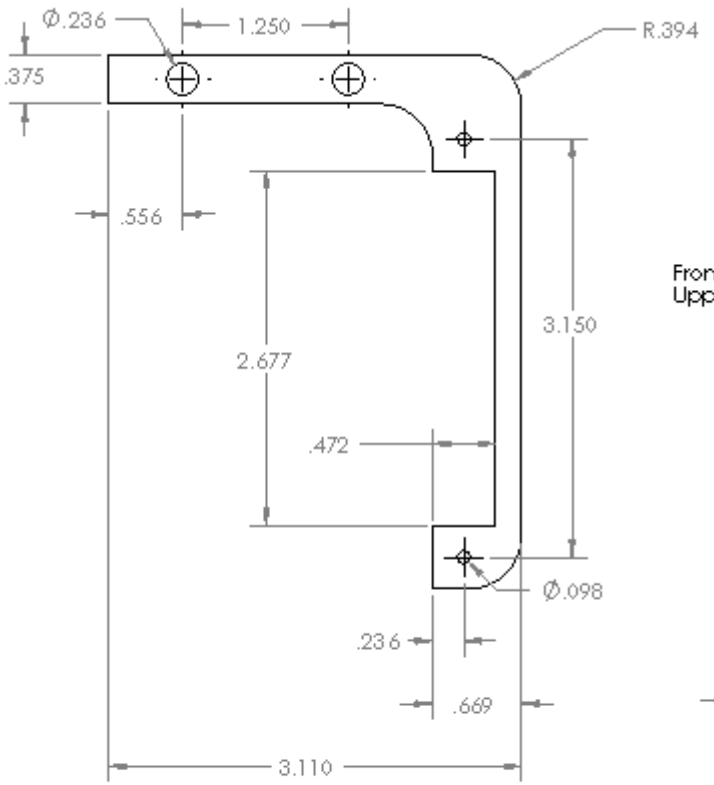




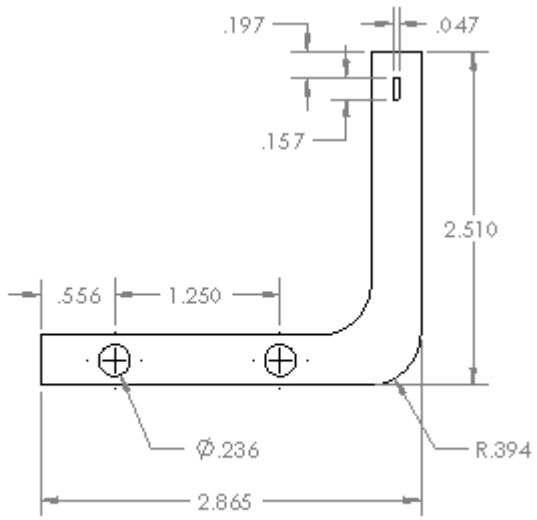
Swing Arm Adjuster Cam

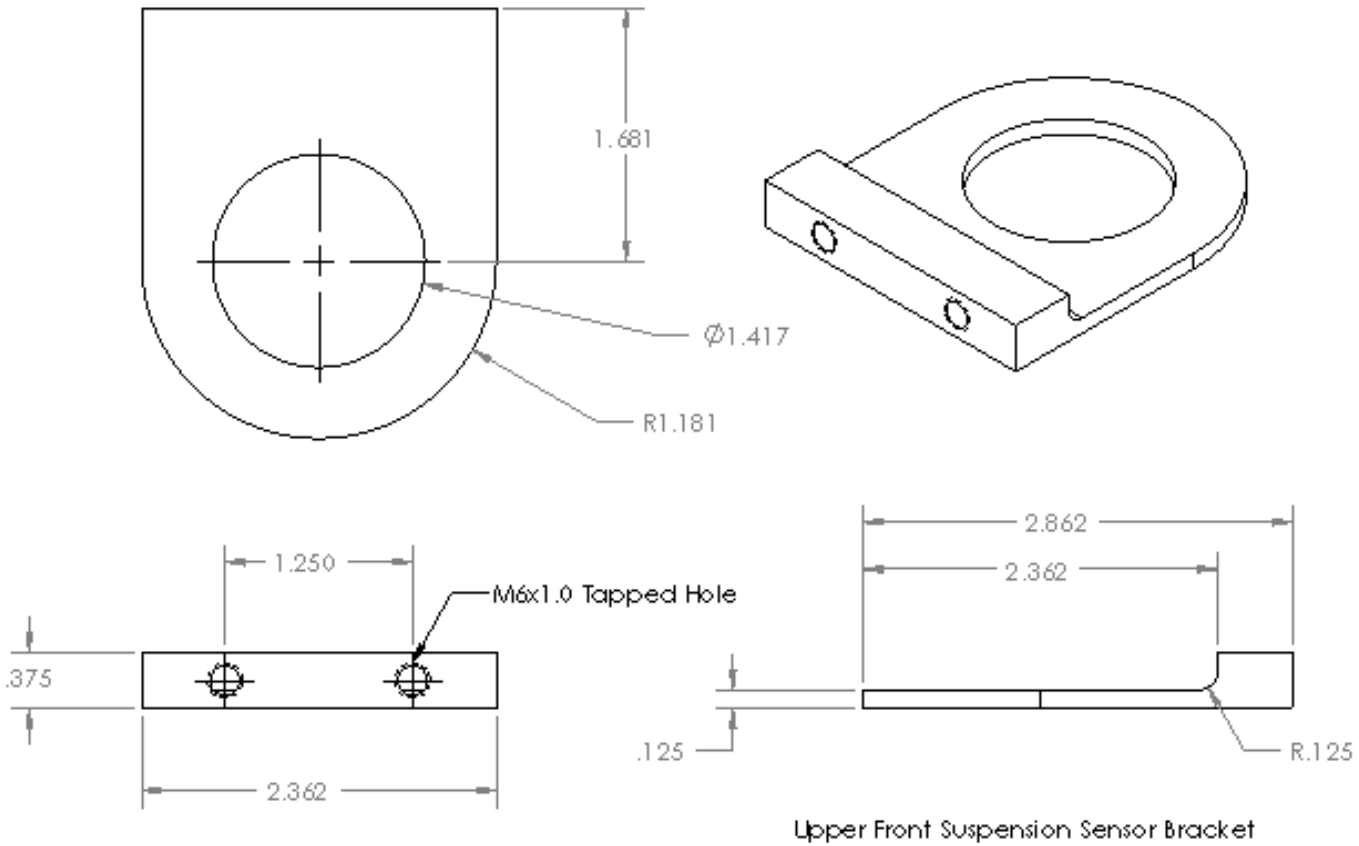


Lower Suspension Sensor Bracket

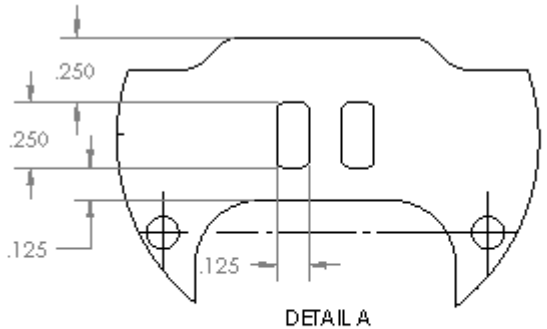


Front Susepension Position Sensor
Upper and Lower Potentiometer Brackets

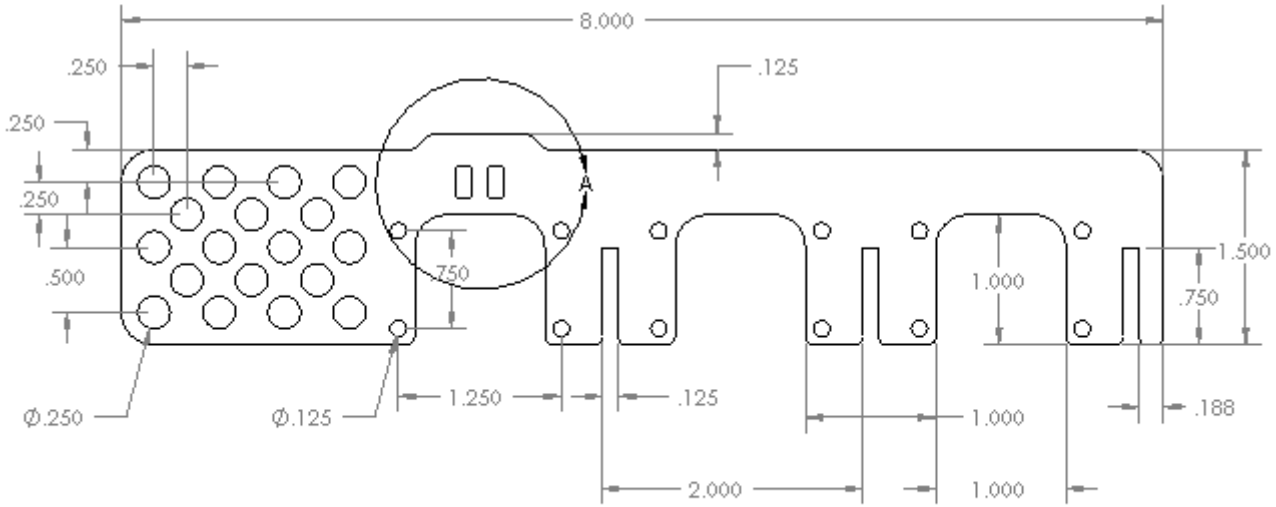




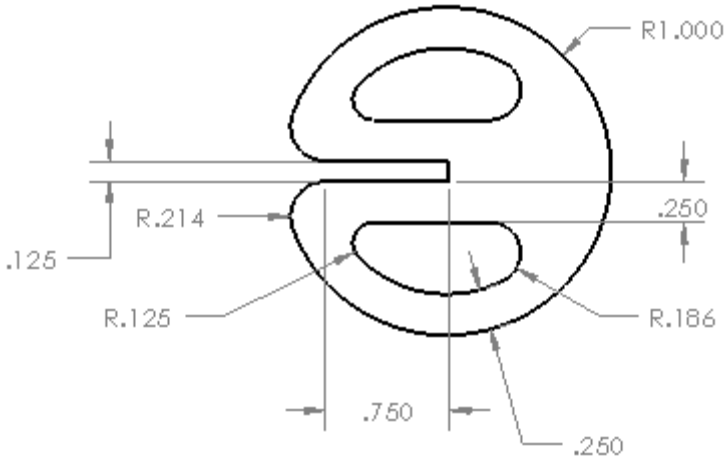
Upper Front Suspension Sensor Bracket



DETAIL A
SCALE 2 : 1



Tire Temp Sensor Bracket



Tire Temp Bracket Guard

Appendix C: Brake Calculation

Initial Brake Calculation

Brake Calculations

$F_{pedal\_in} := 70lbf$ $Pedal\_Ratio := 6$ $Diameter\_tire := 20.5in$ $Bal\_Bar := .55$

$Decel := 1.5$ $cg := 12in$ $Wheel\_base := 60in$ $Area_{piston} := 2.4in^2$

$Fric_{Pad} := .4$ $Radius_{effect\_rotor\_f} := 3.5in$ $Radius_{effect\_rotor\_r} := 3.7in$

$Weight := 650lbf$

$Weight_f := .4 \cdot Weight = 260lbf$ $F_{pedal\_out} := F_{pedal\_in} \cdot Pedal\_Ratio = 420lbf$

$Weight_r := .60 \cdot Weight = 390lbf$

$$Weight := \frac{Weight \cdot Decel \cdot cg}{Wheel\_base} = 195lbf$$

$Weight_{Dynamic\_r} := Weight_r - Weight = 195lbf$

$Weight_{Dynamic\_f} := Weight_f + Weight = 455lbf$

$$Required\_Brake\_Torque_f := \frac{Weight_{Dynamic\_f}}{2} \cdot \frac{Diameter\_tire}{2} \cdot Decel = 291.484lbf \cdot ft$$

$$Required\_Brake\_Torque_r := \left(\frac{Weight_{Dynamic\_r}}{2} \cdot \frac{Diameter\_tire}{2} \cdot Decel \right) = 124.922lbf \cdot ft$$

$$P_f := \frac{Required\_Brake\_Torque_f}{Area_{piston} \cdot Radius_{effect\_rotor\_f} \cdot Fric_{Pad} \cdot 2} = 520.508psi$$

$$P_r := \frac{Required\_Brake\_Torque_r}{Area_{piston} \cdot Radius_{effect\_rotor\_r} \cdot Fric_{Pad}} = 422.033psi$$

$F_{mc\_pushrod\_f} := F_{pedal\_out} \cdot Bal\_Bar = 231lbf$

$F_{mc\_pushrod\_r} := F_{pedal\_out} \cdot (1 - Bal\_Bar) = 189lbf$

$$Size_{mc\_f} := 2\sqrt{\frac{F_{mc\_pushrod\_f}}{P_f}} = 0.752in$$

$$Size_{mc\_r} := 2\sqrt{\frac{F_{mc\_pushrod\_r}}{P_r}} = 0.755in$$

Final Brake Calculations

Brake Calculations

$F_{pedal\_in} := 70lbf$ $Pedal\_Ratio := 6$ $Diameter\_tire := 20.5in$ $Bal\_Bar := .55$

$Decel := 1.5$ $cg := 12in$ $Wheel\_base := 60in$ $Area_{piston} := 2.4in^2$

$Fric_{Pad} := .4$ $Radius_{effect\_rotor\_f} := 3.5in$ $Radius_{effect\_rotor\_r} := 3.7in$

$Weight := 650lbf$

$Weight_f := .4 \cdot Weight = 260\,lbf$ $F_{pedal\_out} := F_{pedal\_in} \cdot Pedal\_Ratio = 420\,lbf$

$Weight_r := .60 \cdot Weight = 390\,lbf$ $Torque_{engine} := 50lbf \cdot ft$

$$Weight := \frac{Weight \cdot Decel \cdot cg}{Wheel\_base} = 195\,lbf$$

$Weight_{Dynamic\_r} := Weight_r - Weight = 195\,lbf$

$Weight_{Dynamic\_f} := Weight_f + Weight = 455\,lbf$

$$Required\_Brake\_Torque_f := \frac{Weight_{Dynamic\_f}}{2} \cdot \frac{Diameter\_tire}{2} \cdot Decel = 291.484 \cdot lbf \cdot ft$$

$$Required\_Brake\_Torque_r := \left(\frac{Weight_{Dynamic\_r}}{2} \cdot \frac{Diameter\_tire}{2} \cdot Decel \right) + Torque_{engine} = 174.922 \cdot lbf \cdot ft$$

$$P_f := \frac{Required\_Brake\_Torque_f}{Area_{piston} \cdot Radius_{effect\_rotor\_f} \cdot Fric_{Pad} \cdot 2} = 520.508\,psi$$

$$P_r := \frac{Required\_Brake\_Torque_r}{Area_{piston} \cdot Radius_{effect\_rotor\_r} \cdot Fric_{Pad}} = 590.952\,psi$$

$F_{mc\_pushrod\_f} := F_{pedal\_out} \cdot Bal\_Bar = 231\,lbf$

$F_{mc\_pushrod\_r} := F_{pedal\_out} \cdot (1 - Bal\_Bar) = 189\,lbf$

$$Size_{mc\_f} := 2\sqrt{\frac{F_{mc\_pushrod\_f}}{P_f \cdot}} = 0.752 \cdot in$$

$$Size_{mc\_r} := 2\sqrt{\frac{F_{mc\_pushrod\_r}}{P_r \cdot}} = 0.638 \cdot in$$