INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Engine Control Unit

Unidade de Controlo de Motor

João Filipe Ferreira Vicente

Dissertation submited to obtain the degree of Master in

Electronic Engineering

Jury

President:	Prof. João Costa Freire
Supervisor:	Prof. Moisés Piedade
Co-Supervisor:	Prof. Francisco Alegria
Member:	Prof. António Serralheiro

December 2009

Acknowledgements

I would like to thank my parents, Aníbal and Maria Eugénia Vicente, for all the help and wisdom they gave me. They will always be the most important things to me.

I would also like to thank professor Moisés Piedade for helping me whenever I needed it.

A special thanks to all the people in scout group 19 for always cheering me on through the difficult times.

And a big thanks to my colleagues for all the good moments and inspiration provided during these five years together.

Abstract

Electronic Fuel Injection systems are very important components in today's automotive industry. Its use on modern engines allows manufactures to develop new engine designs while increasing engine efficiency and lowering fuel consumption and exhaust gas emissions.

EFI systems also increased engine reliability by providing a smooth start and run under most weather conditions.

The goal is to replace the original Engine Control Unit present in the Honda F4i engine with a fully programmable, low cost ECU based on a standard electronic circuit based on a dsPIC30f6012A that can be mass produced and used in many of the onboard systems present in the car. The ECU must make use of all the temperature, pressure, position and speed sensors as well as the original injectors and ignition coils that are already available on the F4i engine.

The ECU must provide the user access to all the maps and allow their full customization simply by connecting it to a PC. This will provide the user with the capability to adjust the engine's performance to its needs quickly and easily.

Keywords:

Electronic Fuel Injection, Engine Control Unit, Formula Student

Resumo

Os sistemas de Injecção Electrónica de combustível são, nos dias de hoje, um dos componentes mais importantes na indústria automóvel. O seu uso nos motores modernos permitem aos fabricantes desenvolver novas geometrias para os motores ao mesmo tempo que reduzem os consumos de combustível e a emissão de gases poluentes.

Os sistemas de Injecção Electrónica também permitiram um aumento da fiabilidade ao permitir um funcionamento e um arranque suaves sob quase todas as condições atmosféricas.

O objectivo é substituir a Unidade de Controlo do Motor já existente no motor F4i da Honda por uma Unidade que possa ser totalmente reprogramada e de baixo custo baseada num circuito electrónico standard com um dsPIC30f6012A que possa ser produzido em massa e aplicado aos mais variados sistemas presentes no carro. A Unidade de Controlo do Motor deve fazer uso dos sensores de temperatura, pressão, posição e velocidade assim como dos injectores e velas já disponíveis no motor F4i.

A Unidade de Controlo do Motor deve permitir o total acesso e customização dos seus mapas simplesmente ligando-a a um PC. Isto permite ao utilizador ajustar a performance do motor as suas necessidades de forma rápida e simples.

Palavras Chave:

Injecção Electrónica , Unidade de Controlo do Motor, Formula Student

Index

Figure index

Acronyms

EFI Electronic Fuel Injection

ECU Engine Control Unit

MAP Manifold Absolute Pressure

IAT Intake Air Temperature

ECT Engine Coolant Temperature

TDC Top Dead Center

BDC Bottom Dead Center

WOT Wide Open Throttle

RPM Revolutions Per Minute

NTC Negative Temperature Coefficient

VVT Variable Valve Timing

GPS Global Positioning System

Can-Bus Controller Area Network Bus

1. Introduction

This project is part of the Formula Student project being developed at Instituto Superior Técnico that for the European series of the Formula Student competition.

Created by the Institution of Mechanical Engineers (IMechE), the Formula Student competition follows the model implemented in the American series and strives to promote careers and excellency in engineering by challenging students to design, build and develop a prototype based on a autocross or sprint racing car with a cost below €21000 and for a production of 1000 units a year while being reliable and easily maintained. During the competition the cars are submitted to a series of tests to evaluate their performance and design, while being judged by engineers from renowned companies in the automotive and aeronautical industries.

The car used (Figure 1.1) has a fibreglass body and uses a Honda F4i engine taken from the Honda CBR 600.

Figure 1.1 – FST03.

The F4i engine is an in-line, 4-cylinder engine with 599 cubic centimetres and electronically controlled injection.

1.1 Purpose and motivation

While being able to change how an engine behaves under different loads is a very important asset, most current Engine Control Units (ECU) on the market are very expensive and require the use specialized tools and parts due to their complexity. Remapping stock ECUs can also be very hard or impossible due to the manufacturers warranty limitations and their ECU's internal design.

The construction of a simple ECU based on a standard electronic circuit that can be mass produced and used in different systems of the vehicle could help bring down the overall cost of the car while providing an efficient and reliable way of commanding the engine and ensuring a smooth operation.

The use of an ECU made specifically for a certain engine would also ensure a perfect compatibility with all the existing sensors and the possibility to add additional sensors and actuators as required or desired.

1.2 Objectives

This paper's objective consists in designing and building an Engine Control Unit (ECU) for a Honda CBR 600R f4i engine by using a standard electronic circuit that contains a dsPic30f6012A 16-bit microprocessor (Figure 1.2).

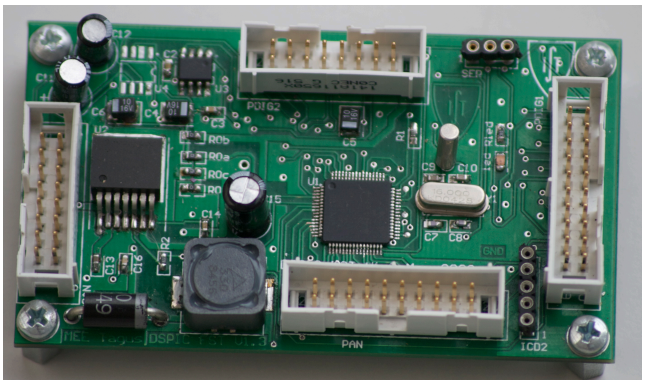


Figure 1.2 – Standard electronic circuit for the ECU.

The ECU must allow the quick reprogramming of the injection maps while using all the sensors and actuators already available in the engine. The ECU must also exchange information via CAN-Bus (Controller Area Network Bus) to a PC enabling the remapping of the ECU on the fly.

2. State of the art

Since its creation, the internal combustion engine has undergone constant changes on its design. The introduction of electronic systems inside of the engine allowed manufacturers a have bigger control over the engine and achieve lower emissions of polluting agents wile increase the performance and efficiency.

Today's electronic fuel injection systems provide smooth engine start and operation even in extreme cold and extreme heat conditions. In motorsport, electronic fuel injection systems allow engineers to adapt the cars to the demands of the circuit whether it demands a greater response at low engine speed or greater top speed.

Advanced electronic engine management systems allow users to fine tune nearly every aspect of the engine's operation. And later high-end products can be used in Variable Valve Timing (VVT) engine, work with multispark systems (multiple sparks are created in rapid succession to improve fuel combustion at low engine speed and cold engine temperatures) and adapt to new types of engine and manifold designs.

New types of sensors can be connected to the ECU. Knock sensors correct ignition timing and prevent the fuel's premature ignition, sensors placed on turbocharged engines help control turbo boost. Some ECU's can also store the data collected from their sensors in rates from 20 to 200 times per second and even add GPS data for later analysis.

There is also an open-sourced ECU that can be fully modified to suit the users needs and specifications.

On the academic level there have been several takes on electronic engine management systems including the construction of an Electronic Control Unit (ECU) for the Shell Eco-marathon project that pursued the lowest possible fuel consumption possible and the ECU for a Volkswagen beetle engine.

3. Electronic Fuel Injection

Due to the growing concern of fuel economy and lower emissions, Electronic Fuel Injection (EFI) systems can be seen on most of the cars being sold today.

EFI systems provide comfort and reliability to the driver by ensuring a perfect engine start under most conditions while lessening the impact on the environment by lowering exhaust gas emissions and providing a perfect combustion of the air-fuel mixture.

In order to ensure the efficient operation of the engine, the EFI system must collect information from a range of sensors located inside the engine, process the information in the ECU and make the necessary adjustments to the quantity of fuel injected and the ignition timing.

With this information the ECU can determine the load set upon the engine and make the necessary adjustments in the fuel injection to ensure a smooth and quick response to the load applied.

In order to better understand how the EFI system works, it can be separated in three parts: the sensors, the ECU and the actuators. Figure 3.1 represents how these parts are connected.

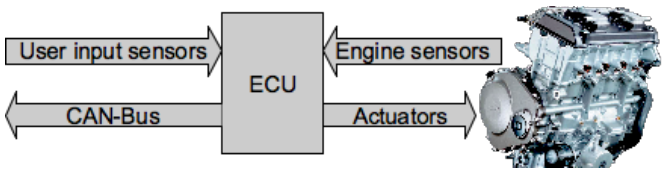


Figure 3.1 – EFI system diagram.

The sensors are used to monitor the status of the engine are Engine Speed, Camshaft Position, Throttle Position, Manifold Absolute Pressure (MAP), Engine Coolant Temperature and Intake Air Temperature. The actuators are comprised of the injectors and ignition coils.

3.1 Sensors

For the ECU to respond correctly to the engine's condition it must rely on an array of sensors. These sensors provide information on temperatures, speed and position. Figure 3.2 shows the flow of information from the sensors to the ECU.

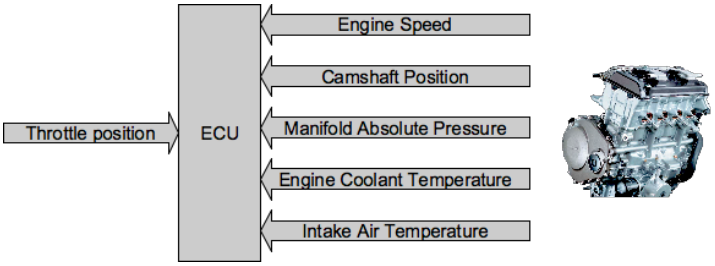


Figure 3.2 – Sensor flow of data.

3.1.1. Engine Speed Sensor

The Engine Speed Sensor relays the engine's current angular speed, in Revolutions Per Minute (RPM), to the ECU. It uses a magnetized cog with 12 teeth attached to the engine's crankshaft (Figure 3.3) and a Hall effect sensor.

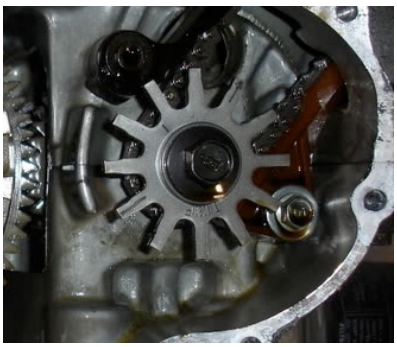


Figure 3.3 – Engine Speed sensor.

When one of the cog's teeth passes near the near the sensor it creates a magnetic field (B) perpendicular to the sensor's surface (Figure 3.4).

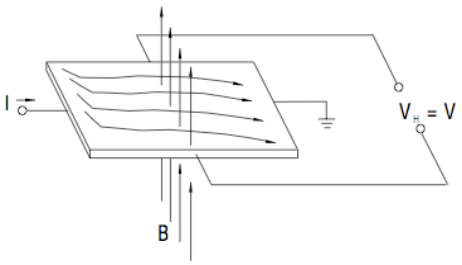


Fig 3.4 – Hall effect sensor's surface [1].

This magnetic field causes a voltage difference across the sensors surface known as the Hall voltage (V_H). This voltage is proportional to the current passing through the sensors surface (I) times the magnetic field (B) (Equation 1) [1].

$$V_H \propto I \times B \qquad \text{(Equation 1)}$$

The Hall voltage is then amplified and sent through a comparator (Figure 3.5) so that we obtain a pulse at the sensor's output pin every time one of the cog's teeth passes near the sensor's surface (Figure 3.6).

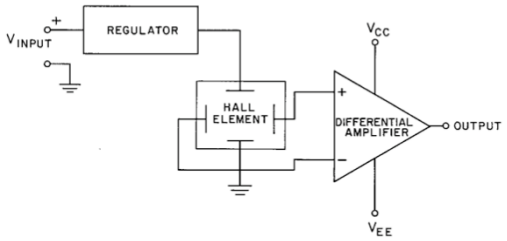


Figure 3.5 – Engine speed sensor internal conditioning circuit.

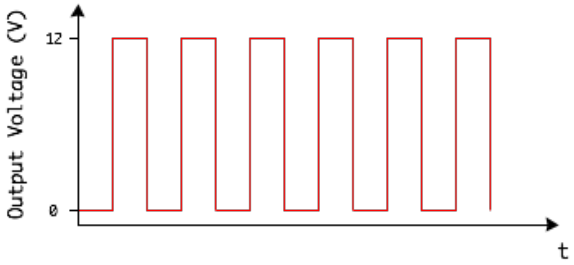


Figure 3.6 – Engine speed sensor output.

When reaching the ECU, the voltage output of the sensor is too high to be placed directly on the dsPIC's pins and therefore must be lowered to a range between 0 and 5 Volts. In order to do this the signal is connected to a zener diode (Figure 3.7) and lowered from 12 Volts to 4,7 Volts.

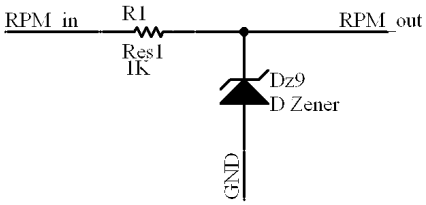


Figure 3.7 – Engine speed signal conditioning.

The signal is then monitored by the ECU that determines the signal's frequency (f_s) by counting the time interval between pulses. The engine's speed in RPM is obtained using Equation 2.

$$RPM = 5 \times f_s \qquad \text{(Equation 2)}$$

3.1.2 Camshaft Position Sensor

The Camshaft Position Sensor works the same way as the Engine Speed Sensor as it also uses a cog and a Hall effect sensor. The main difference between the sensors is the cog used.

The cog used by the Camshaft Position Sensor is attached to the engines camshaft (Figure 3.8) and only has 3 teeth designated by Top Dead Center (TDC), Bottom Dead Center (BDC) and reference (Figure 3.9).

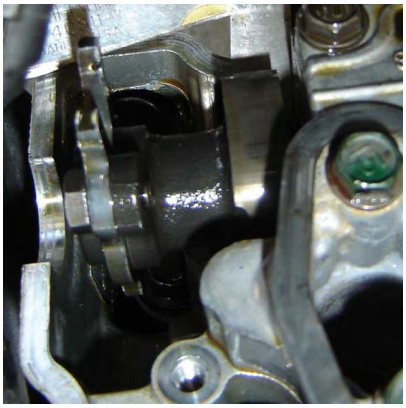


Figure 3.8 – Camshaft Position sensor cog.

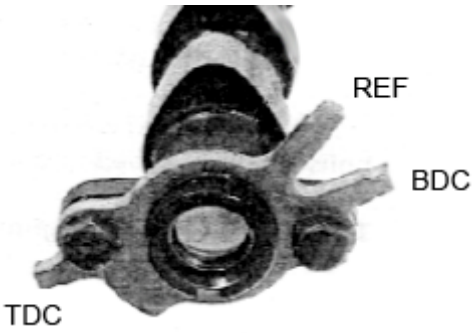


Figure 3.9 – Camshaft Position sensor's cog detail.

TDC occurs when cylinders 1 and 4 reach the highest point inside the combustion chamber (Figure 3.10a) and cylinders 2 and 3 are at their lowest point (Figure 3.10b).

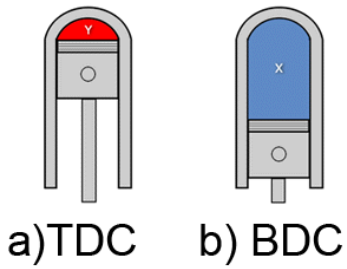


Figure 3.10 – Cylinder position: a) TDC b) BDC.

BDC occurs 180° after TDC and represents the opposite point of TDC. In BDC cylinders 2 and 3 are at the highest point and 1 and 4 are at the lowest.

Finally, the reference pulse occurs slightly after BDC to allow the ECU to distinguish between TDC and BDC pulses. The reference pulse also provides the ECU with a way to determine the camshaft's position at engine start.

The Camshaft Position Sensor output is shown in Figure 3.11.

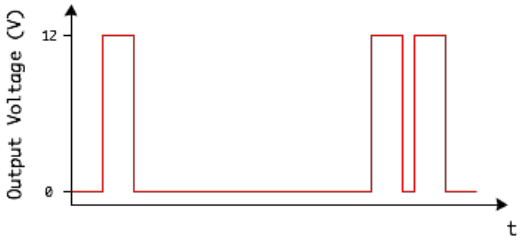


Figure 3.11 – Camshaft Position sensor output.

The Camshaft Position Sensor uses the same signal conditioning circuit as the Engine Speed Sensor (Figure 3.5) and the ECU requires a full camshaft revolution to determine it's position during engine start.

Since the camshaft and the crankshaft are mechanically connected and 1 revolution of the crankshaft equals 1 revolution of the camshaft, it is possible to determinate the camshaft's position by comparing the time between impulses of both sensors (Figure 3.12).

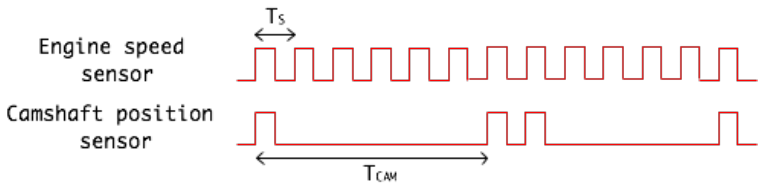


Figure 3.12 – Engine Speed and Camshaft Position signals.

If we ignore, for now, the presence of the reference pulse it is possible to see that TDC and BDC pulses occur every 6 pulses from the Engine Speed Sensor, in other words, the camshaft's sensor period is 6 times the period of the crankshaft's sensor (Equation 3).

$$T_{CAM} = 6 \times T_S \qquad \text{(Equation 3)}$$

Taking the reference pulse back into consideration, the pulse sequence outputted during the first camshaft revolution can have 3 distinct orders (Figure 3.13).

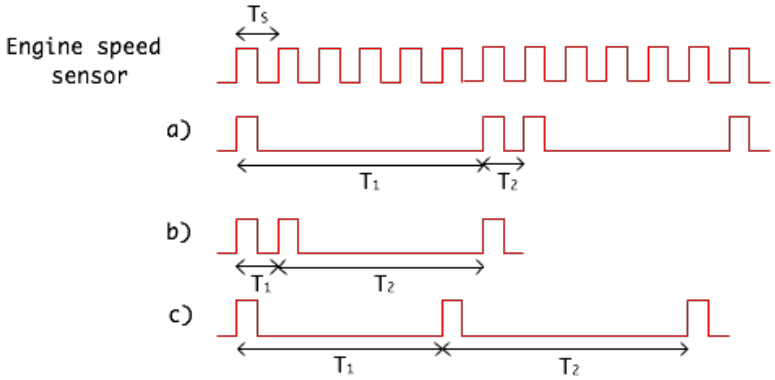


Figure 3.13 – possible sequences during start with T1 and T2 and comparisons

If T1 is equal to the camshaft period (without reference pulse) determined from the Engine Speed sensor output, then the last pulse registered will be the reference and will be followed by the TDC pulse (Figure 3.13a).

In the event that T1 does not match the camshaft's period, the ECU compares T2. If T2 matches, the last pulse of the revolution will be BDC and will be followed by the reference pulse (Figure 3.13b).

If none of the time intervals match the camshaft's period, the last pulse in the revolution will be TDC (Figure 3.13c).

After determining the camshafts position the ECU can start the correct injection and ignition timings for each of the cylinders.

3.1.3 Throttle Position Sensor

The Throttle Position Sensor (TPS) is attached to the throttle valve (Figure 3.14) and relays the current throttle pedal position to the ECU.

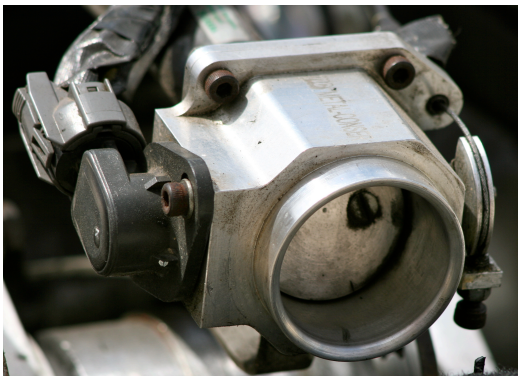


Figure 3.14 – TPS sensor and valve in FST03.

The TPS sensor is composed by a potentiometer connected to the valve's axis. When the driver accelerates, the valve is opened and the potentiometer's wiper arm moves along the resistor changing the output voltage of the sensor (Figure 3.15). The throttle position is determined by measuring the voltage difference between the wiper arm and ground.

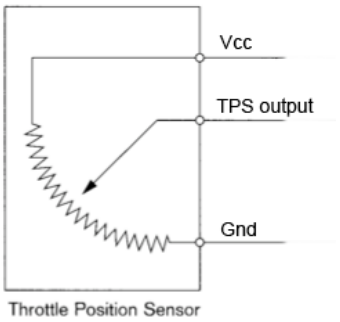


Figure 3.15 – TPS sensor's internal view[2].

The TPS sensor output voltage varies linearly, between Ground and V_{CC}, with the throttle pedal's position (Figure 3.16).

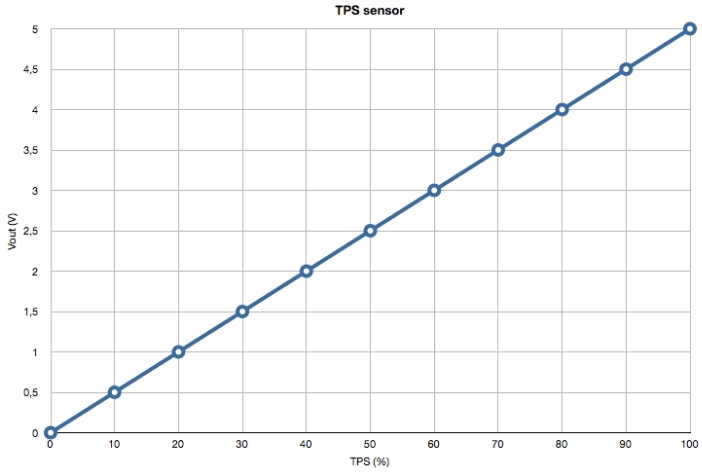


Figure 3.16 – TPS output.

Before converting the TPS voltage to a digital value, there is a low-pass filter to remove any noise that could prevent the Analog-to-Digital Converter (ADC) from making an accurate reading (Figure 3.17).

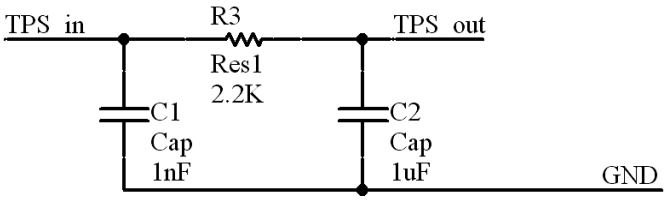


Figure 3.17 – TPS sensor conditioning circuit.

The throttle pedal's position is then transformed into a percentage representing how much throttle is being applied.

3.1.4 Manifold Absolute Pressure

The Manifold Absolute Pressure (MAP) sensor (Figure 3.18) is responsible for measuring the air pressure inside the intake manifold and providing the ECU with the load currently being applied on the engine. With the increase of the engine's load, the volume of air entering the engine through the intake manifold causes a rise of air pressure that is registered by the MAP sensor.

Figure 3.18 – MAP sensor.

Inside the MAP sensor is a small silicon chip (Figure 3.19) placed between a vacuum chamber and a line leading to the intake manifold.

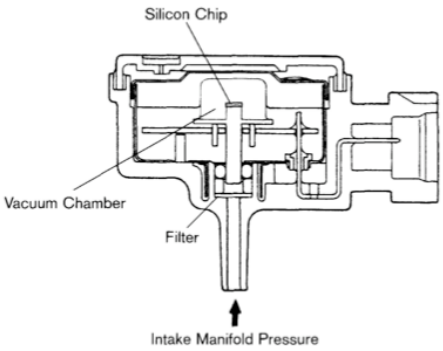


Figure 3.19 – internal views of the sensor [3]

As the pressure increases inside the manifold, the silicon chip flexes (Figure 3.20), acting like a strain gauge and changing its resistance.

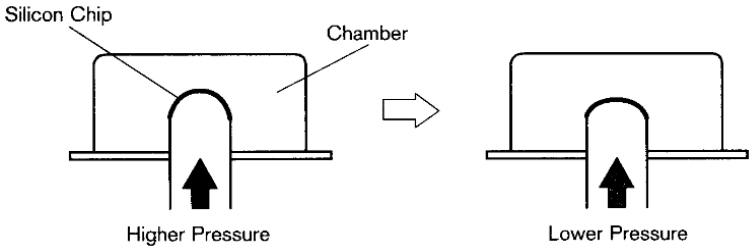


Figure 3.20 – chip flexing under pressure [3].

Inserting the chip in a Wheatstone bridge creates a voltage difference proportional to the strain applied on the silicon chip (figure 3.21) [4].

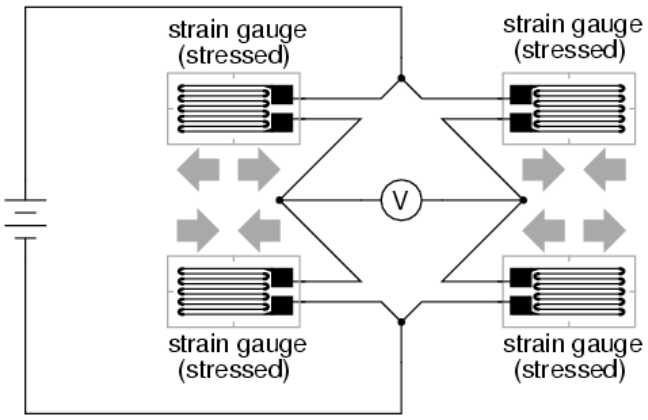


Figure 3.21 – Wheatstone bridge [4].

Before engine start, the MAP sensor registers the Barometric Atmosphere Pressure (BAP) to provide a reference point for calculating engine load.

Figure 3.22 shows the sensor's voltage output for different altitudes.

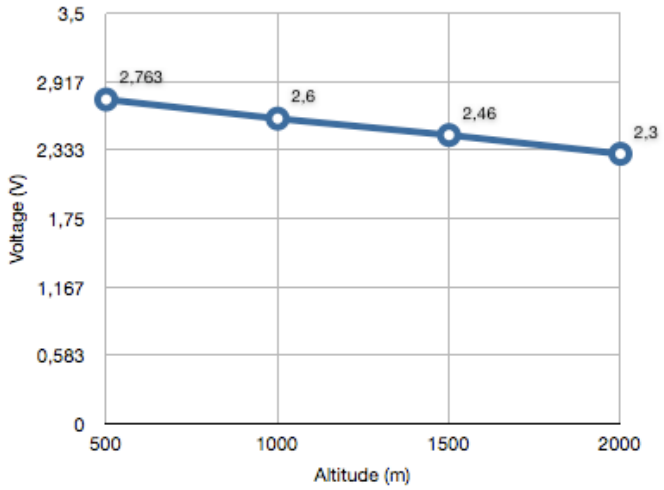


Figure 3.22 – **MAP sensor voltage output for different altitudes [11].**

Engine load is determined by comparing the MAP sensor's value before engine start and the values obtained during engine operation. The value registered for the atmospheric pressure is the same that is registered during Wide Open Throttle (WOT) periods, in other works, 100% engine load. Knowing this, and the fact the output voltage is 0 when the intake manifold is in perfect vacuum, it is possible to calculate the engine's load (Equation 3).

$$load = \frac{MAP \times 100}{BAP} \qquad \text{(Equation 3)}$$

The MAP sensor's voltage output undergoes the same conditioning as the TPS sensor and is filtered by a low-pass filter prior to conversion.

3.1.5 Engine Coolant Temperature

Knowing the Engine Coolant Temperature (ECT) is a good way for the ECU to assess the engine's overall temperature. Elevated engine temperatures can lead to premature damage to the engine's internal components.

The ECT sensor is located near the engine where the coolant exits the engine to cool down in the radiator and is composed by a Negative Temperature Coefficient (NTC) thermistor (Figure 3.23) immersed in the coolant. Due to the Negative temperature coefficient, the resistor's value will decrease with the increase of the coolant's temperature (Figure 3.24).

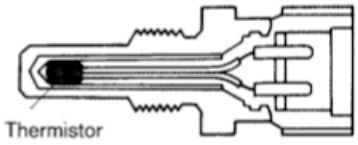


Figure 3.23 – sensor internal view [5].

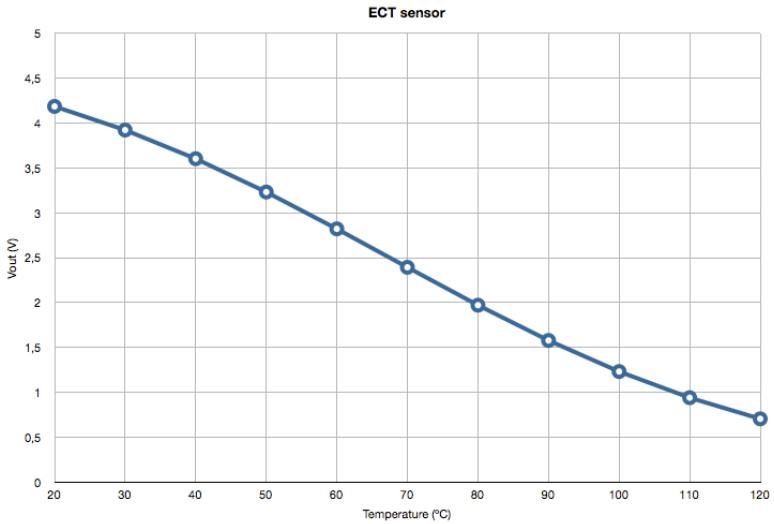


Figure 3.24 – ECT sensor's voltage output.

The drop in the sensor's resistance is conditioned by the ECU's conditioning circuit (Figure 3.25) before being converted to a temperature value by the dsPIC.

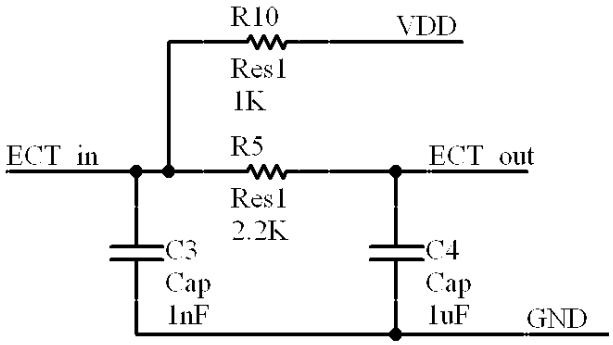


Figure 3.25 – ECT sensor's conditioning circuit.

Engine temperature readings are used by the ECU to make small corrections to the amount of fuel injected and ignition advance resulting in faster engine warm-up during cold starts and helping to lower engine temperature when needed [6].

3.1.6 Intake Air Temperature

The Intake Air Temperature (IAT) sensor (Figure 3.26) is identical to the one used for ECT but instead of being immersed in coolant, the sensor is exposed to the outside air. The IAT sensor is located inside the air intake manifold (figure 3.27) measuring the temperature of the air being channelled into the cylinders.

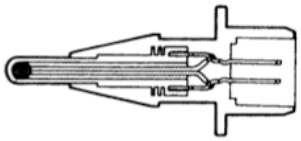


Figure 3.26 – IAT sensor [5].

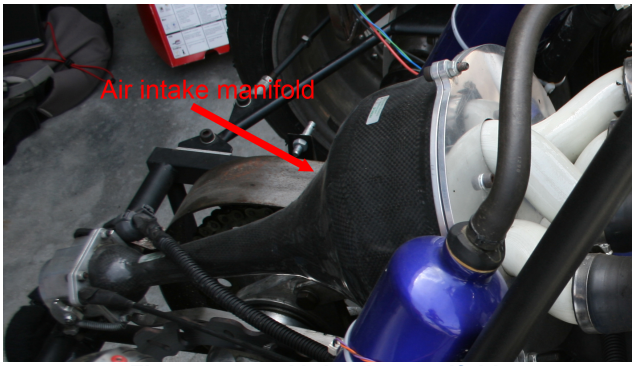


Figure 3.27 – Air intake manifold.

The voltage output is also identical to the output of the ECT sensor but is used in a smaller range of temperature values (Figure 3.28).

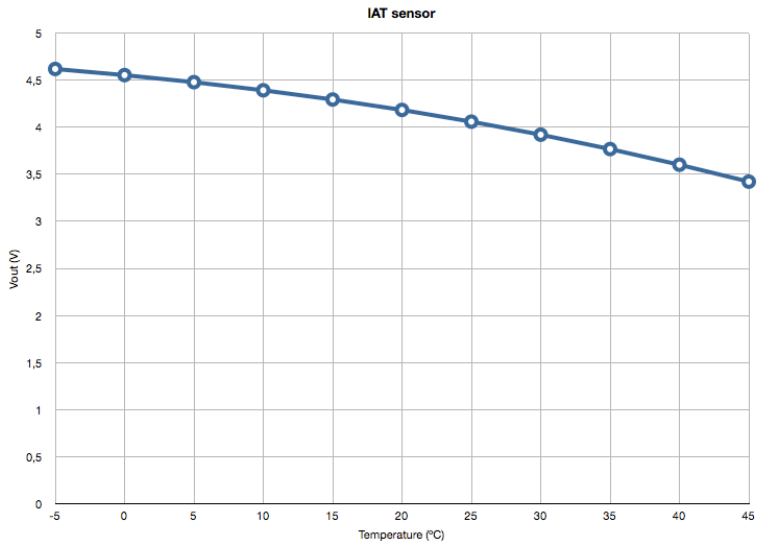


Figure 3.28 – IAT sensor output.

The temperature of the air entering the engine is used to make small adjustments to the fuel injection and ignition advance to provide a more efficient combustion [6].

3.2 Electronic Control Unit

Once the ECU has collected all of the information from the sensors in the engine, it has to determine when and how much fuel to inject in the combustion chambers.

Ideally the ECU tries to achieve the optimum air to fuel ratio of 14.7:1(stoichiometric ratio) but the changing operating conditions require small changes to the ratio. At lighter loads the ECU can use leaner air to fuel mixtures (higher air to fuel ratio) to save fuel and user richer air to fuel mixtures (lower air to fuel ratio) to help reduce engine temperature or warm up the engine. An air to fuel ratio of 12.6:1 (rich) provides maximum power while a ratio of 15.4:1 provides the best fuel economy. Figure 3.12 shows the power and fuel consumption curves versus the air to fuel ratio [7].

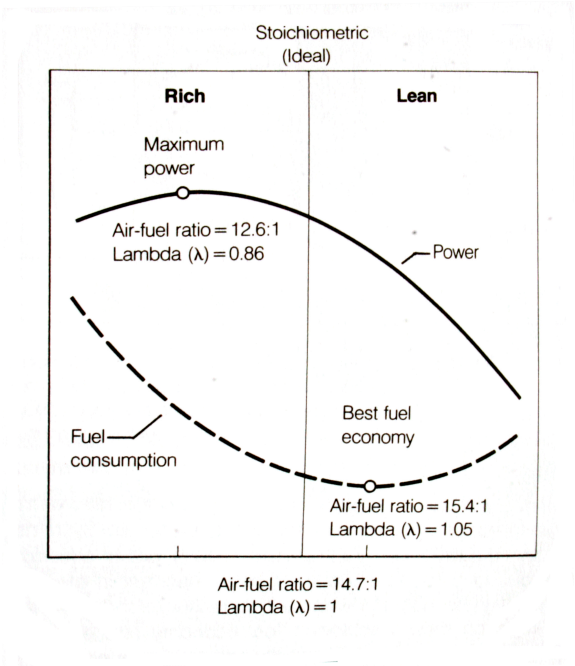


Figure 3.29 – Power and fuel consumption curves versus air to fuel ratio [7].

The stoichiometric ratio is a compromise between rich and lean mixtures with very little sacrifice of power or fuel economy and helps reduce emission of pollutant gases in the combustion process.

The quantity of fuel injected and ignition advance applied for the current engine speed and load is based on a series of tables, or maps, that are fully user configurable.

The ECU can be divided into 2 separate parts: the processing unit and the signal input and output.

The processing unit is the circuit that contains the dsPIC and all sub-circuits necessary for it's operation (Figure 3.30) and the circuit for signal input (sensors) and output (actuators)

contains all driving circuits for the injector and ignition coils as well as the sensor conditioning circuits (Figure 3.31).

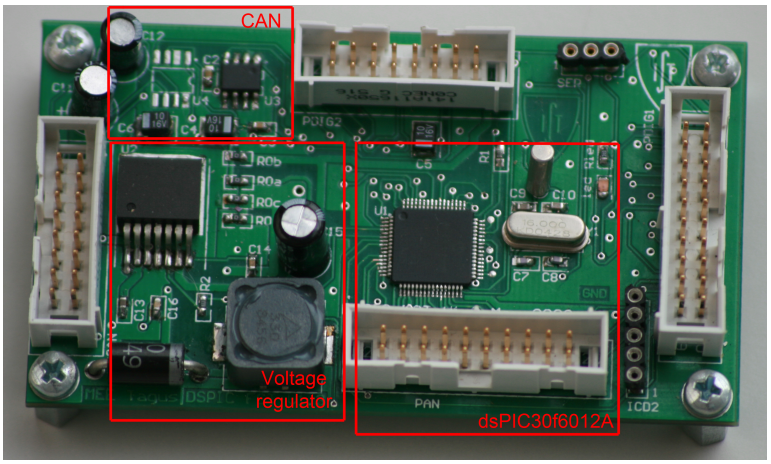


Figure 3.30 – dsPIC30f6012A circuit.

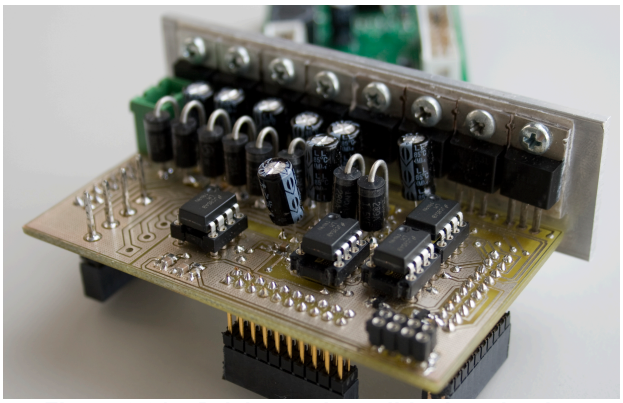


Figure 3.31 – Driving and conditioning circuits.

The full circuit schematic can be found in appendix B.

During operation the ECU first starts by reading all the sensors values and determining the engine's speed. With the current engine speed and load, the ECU indexes the Base Fuel and the Base Ignition Maps (Figure 3.32).

	500 RPM	1000 RPM	1500 RPM	2000 RPM	2500 RPM	3000 RPM	3500 RPM	4000 RPM	4500 RPM	500
10%	100	100	56	50	50	50	50	50	50	50
20%	100	58	57	51	51	50	50	50	50	50
30%	56	57	57	51	51	50	50	50	50	50
40%	56	57	57	51	51	50	50	50	50	50
50%	56	57	57	51	51	50	50	50	50	50
60%	56	57	57	51	51	50	50	50	50	50
70%	50	51	51	51	51	50	50	50	50	50
80%	50	51	51	51	51	50	50	50	50	50
90%	50	50	50	50	50	50	50	50	50	50
100%	50	50	50	50	50	50	50	50	50	50

Figure 3.32 – map example

The Base Fuel Map determines how long the injectors stay open during a camshaft revolution. The values stored in this map are in the form of percentages. Figure 3.33 shows

how the variation of the Base Fuel Map's value affects the amount of fuel being injected into the combustion chamber.

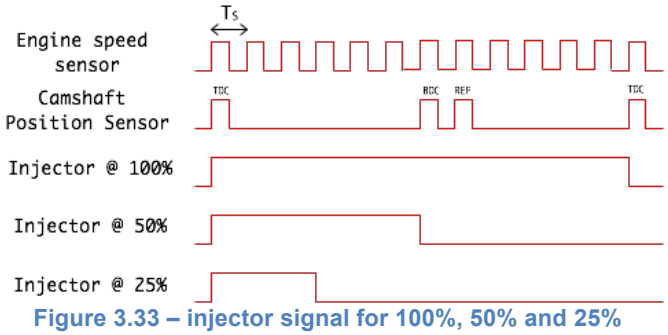


Figure 3.33 – injector signal for 100%, 50% and 25%

The Base Ignition Map sets how many degrees of ignition advance (before TDC) must the ignition have so that the spark occurs in the desired point of ignition.

Figure 3.34 shows the injection and ignition signals for one of the cylinders of the engine.

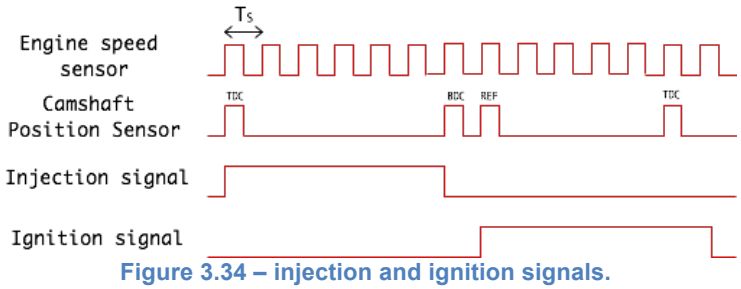


Figure 3.34 – injection and ignition signals.

Since the ECU is time sensitive, all operations regarding ADC conversions, engine speed, camshaft position and the generation of all output signals are made using interruptions. Interruptions allow the ECU to perform other functions without the need of polling the inputs.

When an Engine Speed Sensor's pulse reaches the ECU an interrupt is generated and the time passed from the last pulse is stored so that the engine's speed can be updated.

During camshaft related interruptions the ECU sets the output pulses for the appropriate injectors and ignition coils. These pulses are generated using the dsPIC's internal hardware and only require the start and the end of the pulse to be set. The injectors and ignition coils are fired according to the engine's firing sequence: 1-2-4-3 [11]. This means the first cylinder ignites first, followed by the second, the forth and finally the third cylinder.

Figure 3.35 shows the detailed operation of the camshaft's interruption.

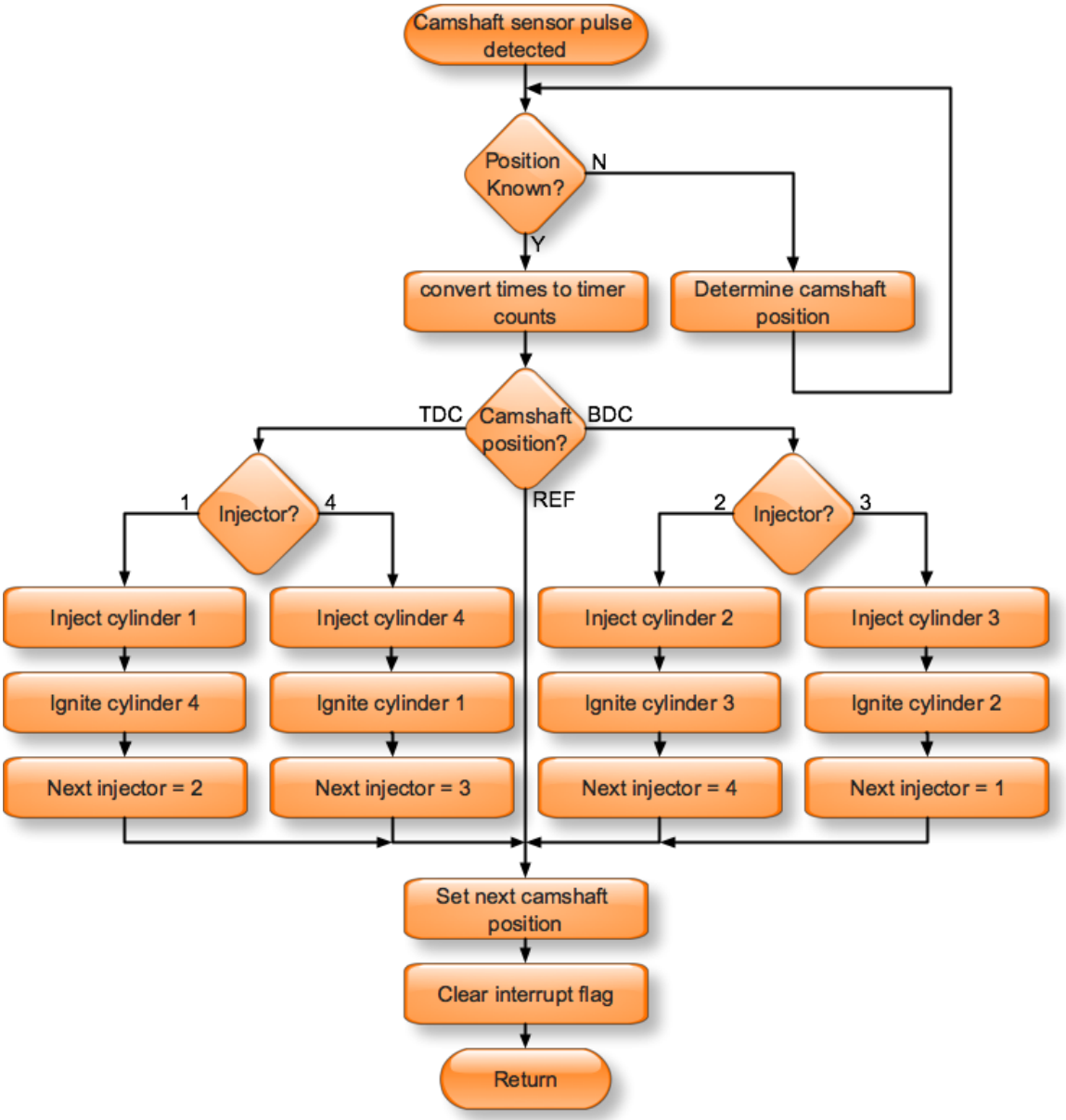


Figure 3.35 – Camshaft Position interrupt diagram.

Between interruptions the ECU uses the new sensor data to update the current injection and ignition advance timings. These are calculated using the ECU's maps values but due to the fact that the values contained in the maps do not include all possible values for engine speed and load, a bilinear interpolation must be used.

A bilinear interpolation is an extension of the linear interpolation to functions with 2 variables. For example, determining the value of a given point P (shown as a green dot in Figure 3.36).

Since P is inside the range of values of the map, we can define 4 known points near P (4 red dots in Figure 3.36)

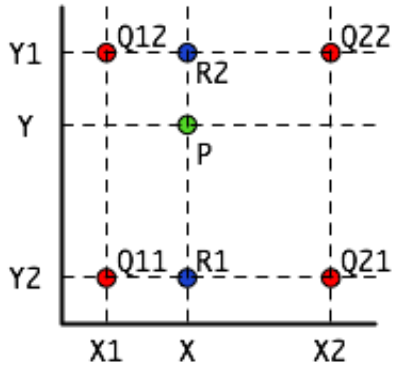


Figure 3.36 – bilinear interpolation.

By using linear interpolation along one of the axis, in this case the x, we obtain R1 and R2 (blue dots in Figure 3.36) by using equations 4 and 5 respectively.

$$f(R1) \approx \frac{X_2 - X}{X_2 - X_1} f(Q_{11}) + \frac{X - X_1}{X_2 - X_1} f(Q_{21}) \qquad \text{(Equation 4)}$$

$$f(R2) \approx \frac{X_2 - X}{X_2 - X_1} f(Q_{12}) + \frac{X - X_1}{X_2 - X_1} f(Q_{22}) \qquad \text{(Equation 5)}$$

Having determined R1 and R2, we can interpolate along the remaining axis to obtain the maps value on P.

$$f(P) \approx \frac{Y_2 - Y}{Y_2 - Y_1} f(R1) + \frac{Y - Y_1}{Y_2 - Y_1} f(R2) \qquad \text{(Equation 6)}$$

After the injection and ignition advance timings are determined, they will be used on the next camshaft interruption.

Figure 3.37 details the main ECU operations.

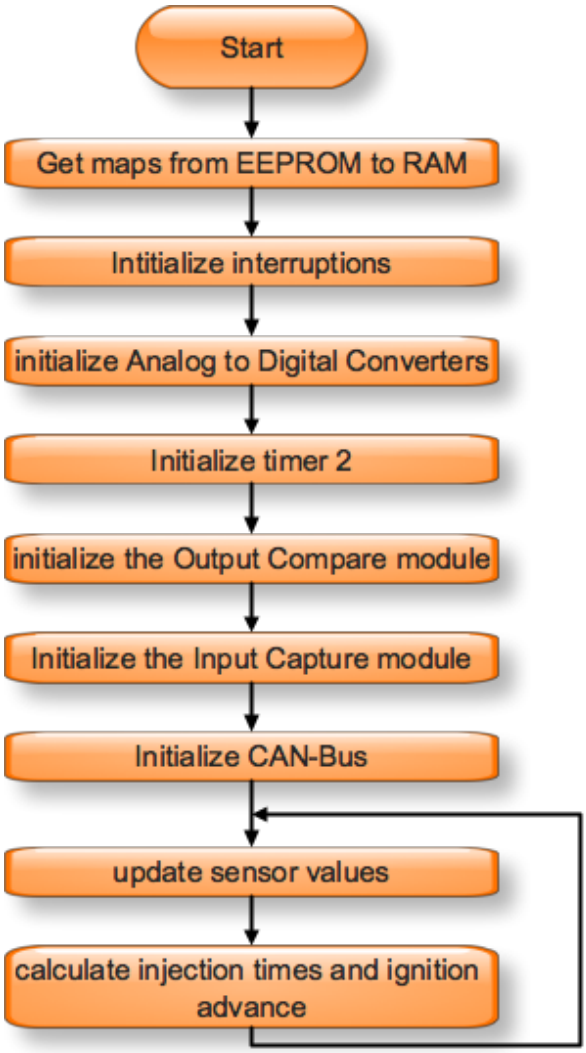


Figure 3.37 – flowchart main.

3.3 Actuators

After performing all the calculations the ECU must command the injectors and the ignition coils in order to inject and ignite the fuel inside the combustion chamber of each cylinder. Precise injection and ignition timing can provide the engine a greater power output and smooth operation while an incorrect timing setup can result in a significant loss of power and efficiency and subsequently cause severe damage to the engine.

3.3.1 Injectors

The delivery of fuel to the engine is made by a set of injectors (one per cylinder). The injectors (Figure 3.38) are small electronically controlled nozzles located in the air intake,

upstream of each combustion chamber (Figure 3.39) that spray fuel into the chamber several times per second.

Figure 3.38 – Injector.

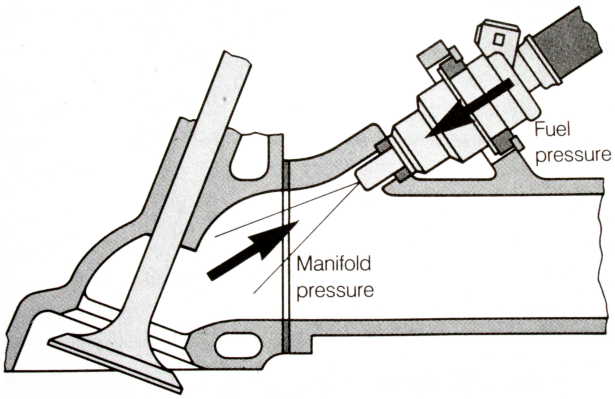


Figure 3.39 – injector location [7].

The injectors are comprised of an electric coil (Figure 3.40) that opens the valve allowing the flow of fuel into the combustion chamber.

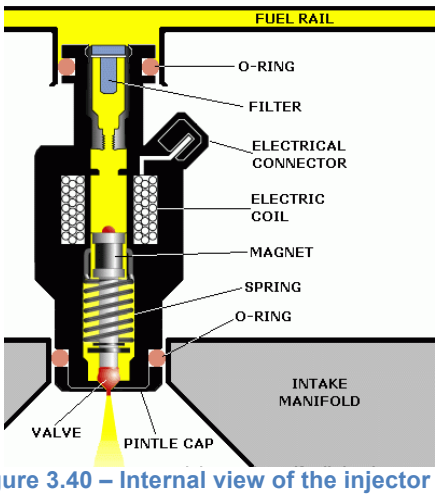


Figure 3.40 – Internal view of the injector [8].

The high pressure of the fuel inside the fuel lines along with the small valve opening atomize the fuel (Figure 3.41) mixing it with the air inside the chamber providing a more efficient fuel combustion [7].

Figure 3.41 – injector spray.

The injectors are controlled using the injector signals created by the ECU (Figure 3.42).

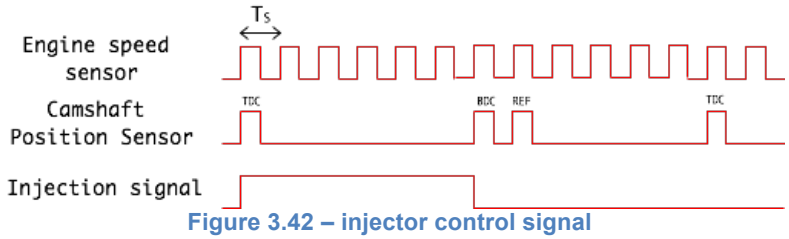


Figure 3.42 – injector control signal

This signal is sent to the injection driver that controls the opening and closing of the injector. Figure 3.43 shows the schematic diagram of the injectors driving circuit.

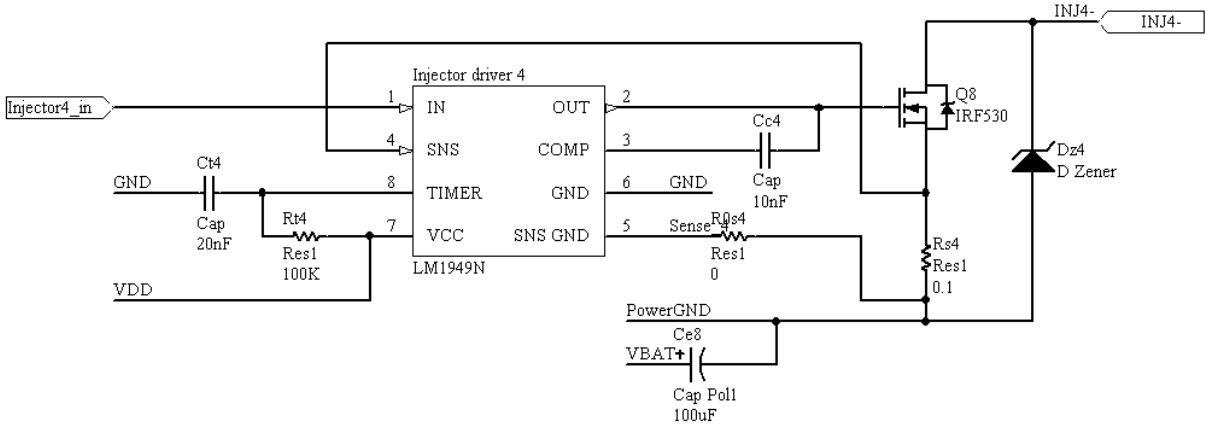


Figure 3.43 – Schematic of the injector drivers.

The LM1949 integrated circuit drives the IGBT transistor while monitoring the current going through the injector's solenoid. If the current reaches 4 A or takes more than 2 ms to do so, the driver reduces the current to less than half. This preserves the injectors since the current needed to maintain the injector open is significantly less than the current needed to open it (Figure 3.44).

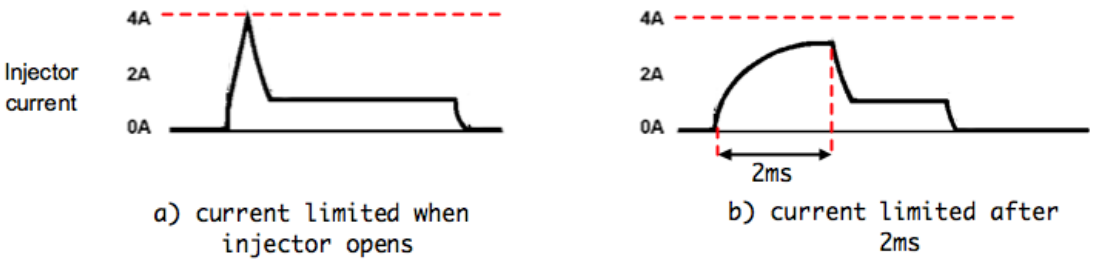


Figure 3.44 – Injector current [14].

The diode Dz4 helps the quick discharge of the solenoid when the injector is closed.

3.3.2 Ignition coils

After the fuel has been injected into the combustion chamber, the ECU must ignite the fuel efficiently. The optimum point of ignition occurs when the combustion chamber is at its maximum compression point (approximately 10° after TDC) [9]. The point of ignition can vary slightly from the optimum combustion point to help reduce the engine's temperature.

Since the spark cannot be instantly created, the ignition coils (Figure 3.45) store an electromagnetic charge that is later discharged in the form of a spark at the tip of the sparkplug. The time taken for the spark to occur from the moment current starts passing through the coil is called dwell [10].

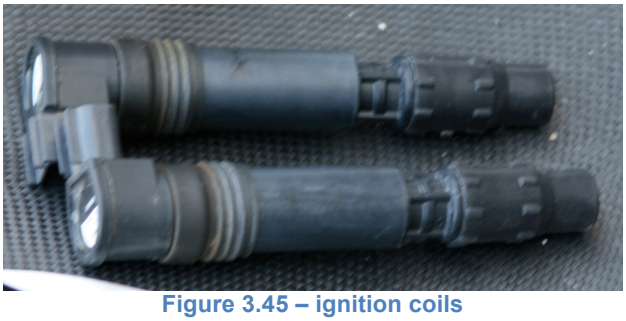


Figure 3.45 – ignition coils

Dwell time (Figure 3.46) depends of the ignition coils used and, in this case, the dwell is equal to 3 milliseconds [11]. Increasing the dwell (over-dwell) will not provide additional energy to the combustion and overheats the coil shortening its life span.

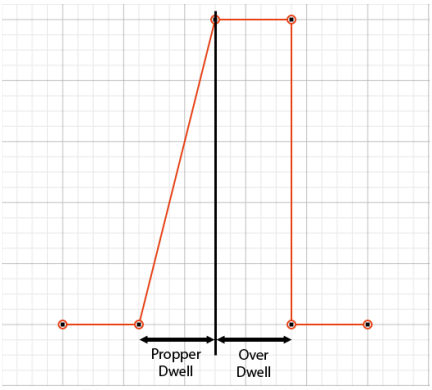


Figure 3.46 – Dwell [10].

The ignition is controlled using a signal created by the ECU for each cylinder (figure 3.47) that is sent to the corresponding ignition driver. Figure 3.48 shows the ignition driver's circuit schematic.

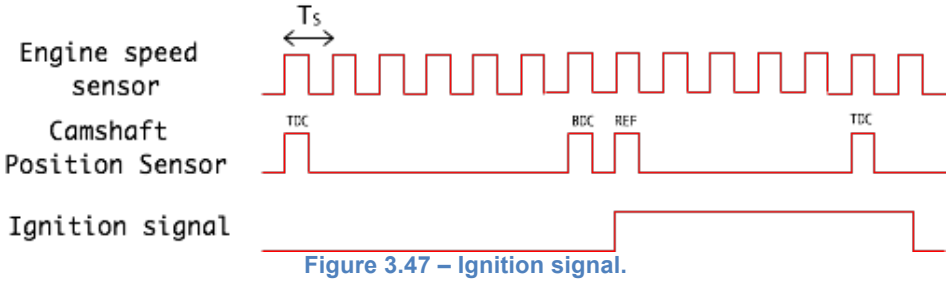


Figure 3.47 – Ignition signal.

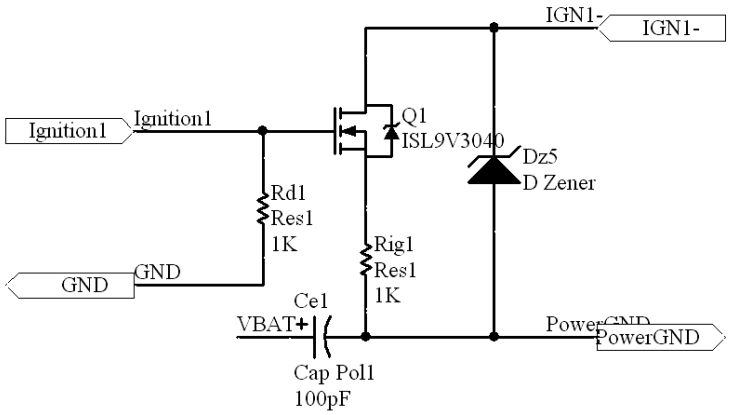


Figure 3.48 – Ignition driver circuit.

The ISLV3040P3 is an ignition driver IGBT that is able to monitor and limit the current passing through the ignition coil without the use of additional external components.

3.4 Controller Area Network Bus

The Controller Area Network Bus (CAN-Bus) is a broadcast protocol bus mainly used in the automotive industry. Being a broadcast bus, all nodes connected to the bus receive the messages sent through it. It is up to the nodes to decide whether to keep or discard the message received.

There are 4 different types of CAN messages: Data frame, remote frame, error frame or an overload frame.

The most common message used, and the one used by the ECU, is the data frame. The data frame can be split into 4 major fields: Arbitration field, data field, Cyclic Redundancy Check (CRC) field and the acknowledge slot (Figure 3.49) [12].

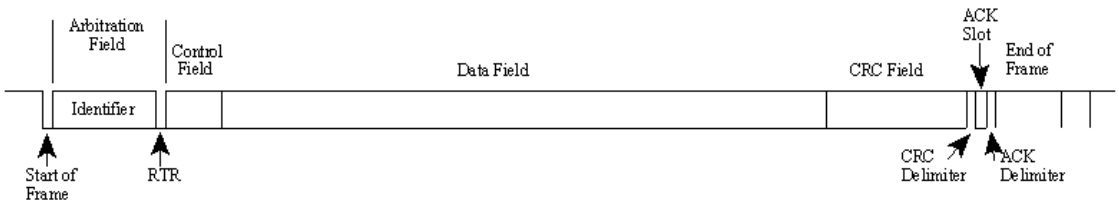


Figure 3.49 – Dataframe message [12].

The arbitration field determines the priority of the message and its used by the nodes as a way to decide which messages to keep. The Arbitration field is 11 or 29 bits long depending on whether the message uses standard or extended CAN identifiers.

Highest priority is given to the lowest message identifier.

The data field contains the message itself. It can take up to 8 bytes of data.

The CRC field is part of an error detection mechanism used by the CAN-Bus protocol. It contains a 15 bit long checksum used to verify the integrity of the message.

Finally, the acknowledgement (ACK) slot is a short interval were the listening nodes send confirmation of the correct arrival of the message. Since all nodes in the bus receive the message, it is impossible to use this slot to make sure that the intended receiving node has received the message sent. Other methods must be used to ensure the arrival of the message to the correct node.

During operation, the ECU uses 3 different message identifiers (Table 3.1).

Table 3.1 - CAN message identifiers.

Identifier	Function
1	Base Fuel Map value
2	Base Ignition Value
3	Acknowledge

Of these identifiers, only 2 are accepted by the ECU in incoming messages. These identifiers correspond to the messages carrying Base Fuel Map or Base Ignition Map values.

The third identifier is used by the ECU as an ACK message signalling the correct arrival of the message.

Figure 3.50 shows how the messages are processed by the ECU. Note that the message discarding is done by specific hardware inside the dsPIC and therefore is not represented.

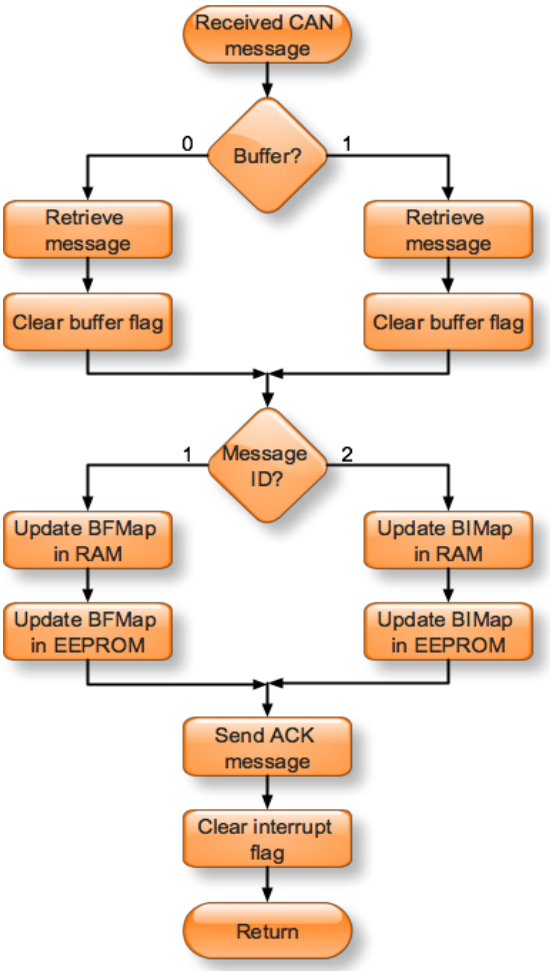


Figure 3.50 – CAN routine.

Upon the reception of the message, the corresponding value is updated in the dsPIC's RAM and EEPROM versions of the maps.

4. Map editor software

Adjustments to the fuel and ignition maps of the ECU are made using a specifically built windows application. This application communicates with the ECU via a USB to CAN converter.

Within the application users can modify the ECU maps in real-time just by changing the values of the maps presented on the application's user interface. Figure 4.1 shows the application's user interface.

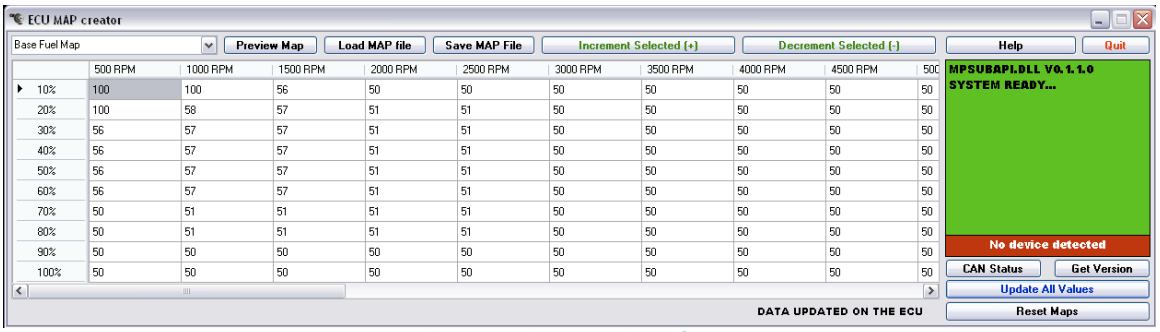


Figure 4.1 – User interface.

All values changed in the user interface are automatically updated in the ECU (if connected) as soon as the user is finished editing them. The application also allows the user to increment or decrement a series of selected values by using the appropriate buttons and save or load the ECU maps with the option of updating the complete maps to the ECU on loading a file.

The currently selected map can also be previewed in graphic form. The user can preview the map as a Surface, contour, XY graphs. Figure 4.2 shows some of the available representations.

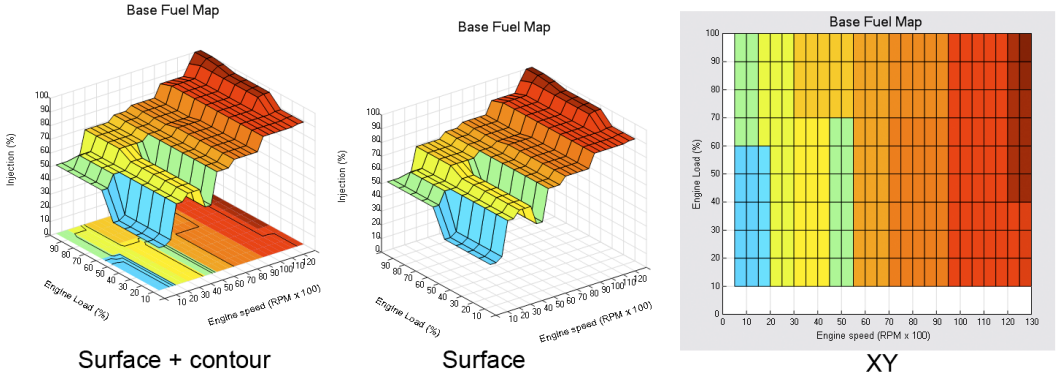


Figure 4.2 – Preview charts.

The changes to the maps are sent to the ECU via CAN-Bus and to ensure the correct transmission of the new map values a small protocol had to be implemented.

All messages sent between the PC and the ECU follow the configuration shown in Figure 4.3.

Figure 4.3 – Message configuration.

SID is the message identifier referred in the previous chapter and, in this case, determines the map that must be updated.

The length field contains the number of bytes of data being transmitted in the message. In this application the length is always 3 bytes of data.

Bytes 1 to 8 carry the information being sent. In this case, the 3 bytes sent correspond to the row, column and value respectively. In the ACK message the all data received is sent back.

Along with the use of an ACK message the application also implements a communication timeout and checks the CAN buffer status for unsent messages that might result in a communication failure. Upon a failing to send a message, the application provides the user with a list of possible causes for the failure.

Figure 4.4 details the communication process.

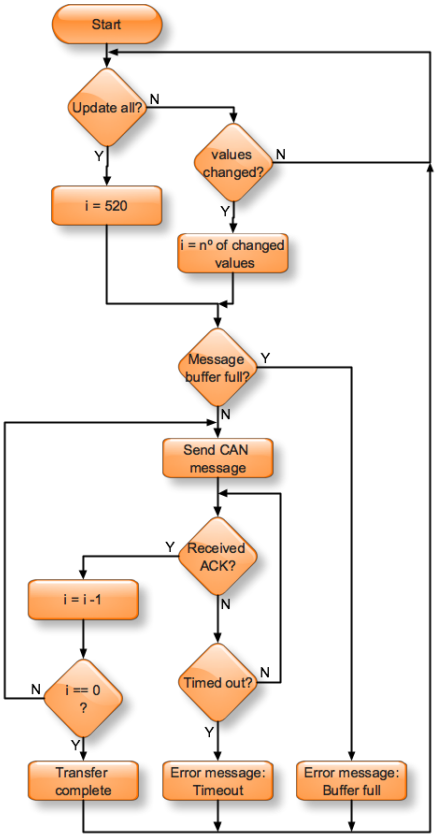


Figure 4.4 – C# flowchart

To enable the communication between the PC and the ECU a USB to CAN-Bus conversion circuit (Figure 4.5) must be used. The converter uses a PIC18f4550 to convey the messages to and from the CAN-bus.

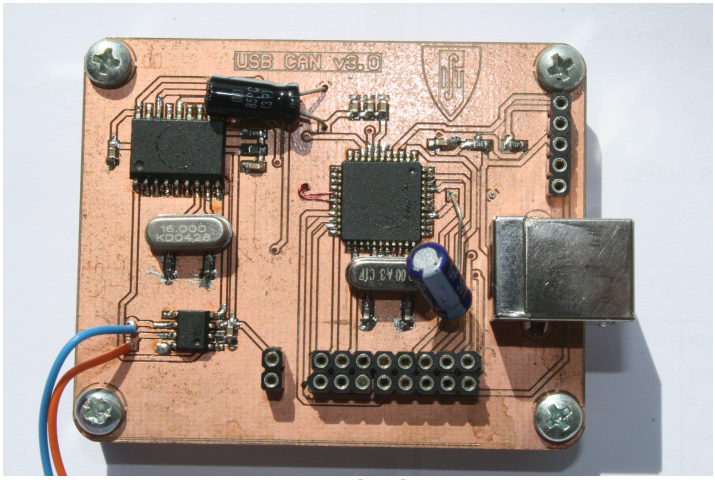


Figure 4.5 – USB-CAN board

5. Results

All results obtained during the course of the ECU's build were done in a laboratory using voltage sources and function generators to simulate all of the necessary sensors.

The injection and ignition control signals were captured using a BitScope PC Oscilloscope (Figure 5.1) and the BitScope DSO software.

Figure 5.1 – Bitscope.

Figure 5.2 shows the layout used during testing.

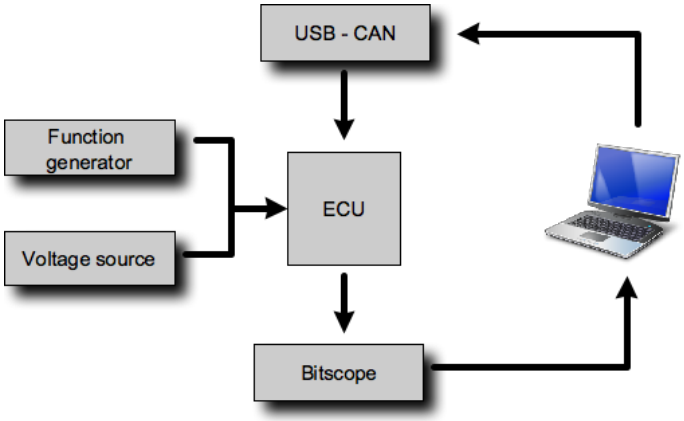


Figure 5.2 – layout

The injection pulses of all 4 injectors for an engine speed of 3000 RPM is shown in Figure 5.3. Note the pattern of the injection sequence (1-2-4-3).

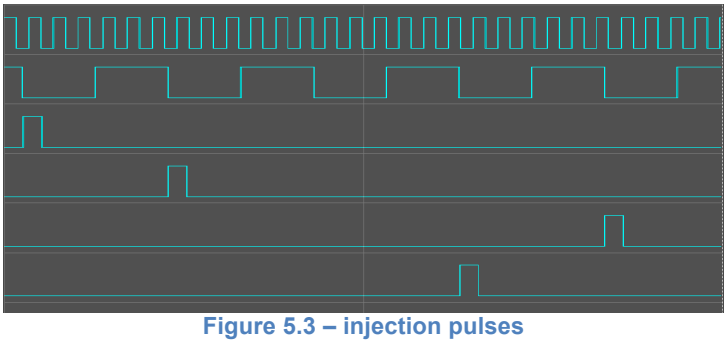


Figure 5.3 – injection pulses

From Figure 5.3 is also noticeable that every 3 pulses from the camshaft one of them is discarded. This pulse is the camshaft reference pulse.

Figure 5.4 shows the ignition pulses for all cylinders with the same engine speed.

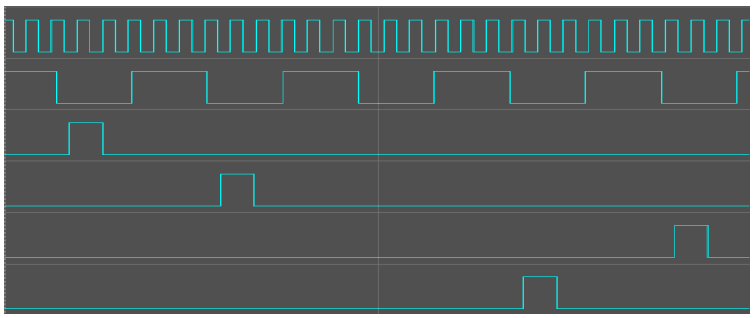


Figure 5.4 – ignition pulses

All Base Fuel and Base Ignition map values used for these simulations are fictional and require prior calibration on a dynamometer.

6. Conclusions

The work presented, the ECU is capable of creating all the necessary signals to operate the engine according to the fuel and ignition maps stored in its memory. It also enables the user to change both maps while the engine is running. This is useful when tuning the engine on a dynamometer and empowers the user to change the ECU's settings according to its preferences.

7. Future work

Future work on the ECU can focused on improving its overall performance.

The ECU's performance can be improved by adding additional maps to make small corrections in the fuel injection and ignition advance based on throttle variations, engine temperature and intake air temperature.

These maps can help the engine run more smoothly and respond better.

The ECU has to be configured while the car is placed on a dynamometer. This will set the correct values in the ECU's maps allowing the engine to run smoothly and correctly.

8. References

[1] HoneyWell, *Hall effect sensing and application*, available from http://content.honeywell.com/sensing/proinfo/solidstate/technical/chapter2.pdf

[2] Toyota university, *Position sensors*, Toyota Motor Sales, U.S.A., available from http://autoshop101.com/forms/h33.pdf

[3] Toyota university, *Pressure sensors*, Toyota Motor Sales, U.S.A., available from http://autoshop101.com/forms/h35.pdf

[4] Kuphaldt, Tony R., 2006, *Lessons in Electronic Circuits – Volume I*, fifth edition, available from http://www.allaboutcircuits.com/pdf/DC.pdf

[5] Toyota university, *Position sensors*, Toyota Motor Sales, U.S.A., available from http://autoshop101.com/forms/h32.pdf

[6] Strader, Ben, *Building & Tuning High-Performance Electronic Fuel Injection*, Car Tech, ISBN 978-1-884089-79-4

[7] Probst, Charles, October 1989, *Bosch Fuel Injection and Engine Management*, Bentley Publishers

[8] Photo taken from http://www.waynesgarage.com/articles/fuelinjectors.htm

[9] Toyota university, *Position sensors*, Toyota Motor Sales, U.S.A., available from http://autoshop101.com/forms/h39.pdf

[10] Motec, 2004, *Engine Management & Data Acquisition Systems*, Formula SAE seminar, Detroit

[11] Honda Motor Company, 2003, *CBR600 F4i service manual*

[12] Kvaser, *The CAN Bus*, available from http://www.kvaser.com/can/intro/index.htm

[13] Photo from Bay Riders Area Forum, http://www.bayarearidersforum.com

[14] National semiconductor, 2008, *LM1949 injection controller datasheet*

Appendix A. Electronic Control Unit program

A1. Main

```
#include <p30f6012A.h>
#include "timers.h"
#include "Out_compare.h"
#include "adc.h"
#include "interrupts.h"
#include "Input_Capture.h"
#include "Maps.h"
#include "can1.h"

#define true 1
#define false 0

#define clock 16000000
#define TimerMax 0xFFFF
#define RPM_increment 500
#define Load_increment 10

#define IGNpulse 0x0465                 // duration of the ignition pulse in timer
counts
#define alpha 0x0004                    // small delay

unsigned int INJsig=1;                                  // counter to select injector
unsigned int IGNsig=2;                                  // counter to select ignition
coil

unsigned int T=0;                                               // Input
capture 1 variable 1
unsigned int T2=0;                                      // Input capture 1
variable 2
unsigned int T3=0;                                      // Input capture 1
variable 3

unsigned int T4=0;                                      // Input capture 2
variable 1
unsigned int T5=0;                                      // Input capture 2
variable 2
unsigned int T6=0;                                      // Input capture 2
variable 3

int go=0;                                               // timing end flag
int Tcam=0;                                     // Cam rotation period calculated
from engine RPM
int CAMstate=0;                 // identifies Cam pulse as TDC, BDC or REF

int timing[3] = { 0 , 0 ,0 };   // array to store the time between cam
pulses
int timing_count=0;             // timing pulse counter;

int TPS=0;                                              // ADC conversion result -
Throttle position
int ECT=0;                                              // ADC conversion result -
Coolant temperature
```

```
int TP=0;                                                   // ADC conversion result -
Temperature
int MAP=0;                                                  // ADC conversion result -
Manifold pressure
int IAT=0;                                                  // ADC conversion result -
Air temperature

int IGNdelay;                                               // ignition pulse delay
int INJpulse=0;                                             // duration of the injector pulse in
timer counts

int config=0;
int BAP=0;

//CAN variables
unsigned int bufferRX_CAN1[4] = {0, 0, 0, 0};
unsigned int sid,lenght,data[4];

void __attribute__((interrupt, no_auto_psv)) _IC1Interrupt(void){

    T=T2;
    T2=IC2BUF;                                              // save timer count


    if(T2>T)
      T3=T2-T;
    if(T>T2)
      T3=(TimerMax-T)+T2;


    IFS0bits.IC1IF=0;                           // clear interrupt flag
    return;


}

//Input Capture 2 interruption code
    void __attribute__((interrupt, no_auto_psv)) _IC2Interrupt(void){

      unsigned int ONpulse;
      unsigned int OFFpulse;

      unsigned int ON_Ignpulse;
      unsigned int OFF_Ignpulse;


      if(timing_count!=3 && go!=true){
            T4=T5;
            T6=IC2BUF;                                                  // save timer count


            if(T5>T4)
                  T6=T5-T4;
            if(T4>T5)
                  T6=(TimerMax-T4)+T5;

            timing[timing_count]=T6;

            if (timing[1] == Tcam){
                  CAMstate = 3;
```

```
                        INJsig=1;                               //set next injector
                        go = true;
                } else {
                        if(timing[2] == Tcam){
                                CAMstate = 2;
                                INJsig=1;                       //set next injector
                                go = true;
                        } else {
                                if(timing[1]!=0 || timing[2] != 0){
                                        CAMstate = 1;
                                        INJsig=2;               //set next injector
                                        go = true;
                                }else{
                                        go=false;
                                }
                        }
                }
        }

        if (timing_count < 3){
                timing_count++;
        }else{

        // determine start and stop time for ignition pulse
        ON_Ignpulse=TMR2 + alpha + IGNdelay;
        OFF_Ignpulse=TMR2 + alpha + IGNdelay + IGNpulse;

        if (ON_Ignpulse>0xFFFF)
                ON_Ignpulse=ONpulse-0xFFFF;

        if (OFF_Ignpulse>0xFFFF)
                OFF_Ignpulse=OFFpulse-0xFFFF;

        // determine start and stop time for injection pulse
        ONpulse=TMR2 + alpha;
        OFFpulse=TMR2 + alpha + 0x01F2;//INJpulse;

        if (ONpulse>0xFFFF)
                ONpulse=ONpulse-0xFFFF;

        if (OFFpulse>0xFFFF)
                OFFpulse=OFFpulse-0xFFFF;

        switch(CAMstate){

                case 1:
                        switch (INJsig){
                                case 1:
                                        //injector 1
                                        IEC0bits.OC1IE=1;               //enable OC1
interrupt
                                        OC1R=ONpulse;                           //set
impulse start
                                        OC1RS=OFFpulse;                         //set
impulse stop
                                        //ignition 4
                                        IEC2bits.OC8IE=1;               //enable OC8
interrupt
                                        OC8R=ON_Ignpulse;               //set impulse start
                                        OC8RS=OFF_Ignpulse;             //set impulse stop
```

```
                                        INJsig=2;                       //set next
injector
                                        break;

                                case 4:
                                        //injector 4
                                        IEC1bits.OC4IE=1;       //enable OC4
interrupt
                                        OC4R=ONpulse;                   //set
impulse start
                                        OC4RS=OFFpulse;                 //set
impulse stop
                                        //ignition 1
                                        IEC2bits.OC5IE=1;       //enable OC5
interrupt
                                        OC5R=ON_Ignpulse;       //set impulse start
                                        OC5RS=OFF_Ignpulse;     //set impulse stop

                                        INJsig=3;                       //set next
injector
                                        break;
                        }


                        CAMstate = 2;
                        break;

                case 2:
                        switch (INJsig){
                                case 2:
                                        //injector 2
                                        IEC0bits.OC2IE=1;       //enable OC2
interrupt
                                        OC2R=ONpulse;                   //set
impulse start
                                        OC2RS=OFFpulse;                 //set
impulse stop
                                        //ignition 3
                                        IEC2bits.OC7IE=1;       //enable OC7
interrupt
                                        OC7R=ON_Ignpulse;       //set impulse start
                                        OC7RS=OFF_Ignpulse;     //set impulse stop

                                        INJsig=4;                       //set next
injector
                                        break;

                                case 3:
                                        //injector 3
                                        IEC1bits.OC3IE=1;       //enable OC3
interrupt
                                        OC3R=ONpulse;                   //set
impulse start
                                        OC3RS=OFFpulse;                 //set
impulse stop
                                        //ignition 2
                                        IEC2bits.OC6IE=1;       //enable OC6
interrupt
```

```
                                        OC6R=ON_Ignpulse;           //set impulse start
                                        OC6RS=OFF_Ignpulse;         //set impulse stop

                                        INJsig=1;                                   //set next
injector
                                        break;
                                }
                                CAMstate = 3;
                                break;

                        case 3:
                                CAMstate=1;
                                break;

                }
        }
                IFS0bits.IC2IF=0;                       // clear interrupt flag
                return;
        }

        void __attribute__((interrupt, auto_psv)) _C1Interrupt(void){



        if (C1INTFbits.RX0IF) {
            sid = C1RX0SIDbits.SID;
            lenght = C1RX0DLCbits.DLC;
            data[0] = C1RX0B1 & 0x00ff;
            data[1] = (C1RX0B1 & 0xff00)>>8;
            data[2] = C1RX0B2 & 0x00ff;
            data[3] = (C1RX0B2 & 0xff00)>>8;
            C1RX0CONbits.RXFUL = 0;
            C1INTFbits.RX0IF = 0;
        }
        if (C1INTFbits.RX1IF) {
            sid = C1RX1SIDbits.SID;
            lenght = C1RX1DLCbits.DLC;
            data[0] = C1RX0B1 & 0x00ff;
            data[1] = (C1RX0B1 & 0xff00)>>8;
            data[2] = C1RX0B2 & 0x00ff;
            data[3] = (C1RX0B2 & 0xff00)>>8;
            C1RX1CONbits.RXFUL = 0;
            C1INTFbits.RX1IF = 0;
        }

        switch (sid){

            case 1:
                        _RD11=1;
                        BFMap[ data[1] ] [ data[0] ] = data[2];
                        break;

            case 2:
                        BIMap[ data[1] ] [ data[0] ] = data[2];
                        break;

            default:
                        break;
        }
```

```
        config_envia_can1( 3, 3 , data[0] , data[1] , data[2] , data[3]  );
        IFS1bits.C1IF=0;
}

        // ADC interruption
        void __attribute__((interrupt, no_auto_psv)) _ADCInterrupt (void){

            int TPS0;
            int ECT0;
            int MAP0;
            int IAT0;


            TPS0 = ADCBUF0;                                 // save ADC conversion - Throttle
position
            ECT0 = ADCBUF1;                                 // save ADC conversion - Coolant
Temperature
            MAP0 = ADCBUF3;                                 // save ADC conversion - Manifold
pressure
            IAT0 = ADCBUF4;                                 // save ADC conversion - Air
Temperature

            //TPS processing
            TPS = (TPS0 * 100)/5;

            //MAP processing
            if (config == 0){
                    BAP = MAP0;
                    config01;
            }else{
                    MAP = (MAP0 * 100)/BAP;
            }

            //


            IFS0bits.ADIF = 0;

            return;
}




//=========================OUTPUT COMPARE
INTERRUPTIONS============================

//                  ===INJECTOR INTERRUPTIONS===

// Output Compare 1 interruption - INJECTOR 1
void __attribute__((interrupt, no_auto_psv)) _OC1Interrupt (void){


        IEC0bits.OC1IE=0;
        OC1CONbits.OCM=0;
        OC1CONbits.OCM=4;       // single pulse mode - must be reset in order to generate
another pulse
        IFS0bits.OC1IF = 0;
```

```
	return;
}

// Output Compare 2 interruption - INJECTOR 2
void __attribute__((interrupt, no_auto_psv)) _OC2Interrupt (void){


	IEC0bits.OC2IE=0;
	OC2CONbits.OCM=0;
	OC2CONbits.OCM=4;			// single pulse mode - must be reset in order
to generate another pulse
	IFS0bits.OC2IF = 0;

	return;
}

// Output Compare 3 interruption - INJECTOR 3
void __attribute__((interrupt, no_auto_psv)) _OC3Interrupt (void){


	IEC1bits.OC3IE=0;
	OC3CONbits.OCM=0;
	OC3CONbits.OCM=4;			// single pulse mode - must be reset in order
to generate another pulse
	IFS1bits.OC3IF = 0;

	return;
}

// Output Compare 4 interruption - INJECTOR 4
void __attribute__((interrupt, no_auto_psv)) _OC4Interrupt (void){


	IEC1bits.OC4IE=0;
	OC4CONbits.OCM=0;
	OC4CONbits.OCM=4;			// single pulse mode - must be reset in order
to generate another pulse
	IFS1bits.OC4IF = 0;

	return;
}

//			===IGNITION INTERRUPTIONS===

// Output Compare 5 interruption - IGNITION COIL 1
void __attribute__((interrupt, no_auto_psv)) _OC5Interrupt (void){


	IEC2bits.OC5IE=0;
	OC5CONbits.OCM=0;
	OC5CONbits.OCM=4;			// single pulse mode - must be reset in order
to generate another pulse
	IFS2bits.OC5IF = 0;

	return;
}

// Output Compare 6 interruption - IGNITION COIL 2
void __attribute__((interrupt, no_auto_psv)) _OC6Interrupt (void){
```

```
	IEC2bits.OC6IE=0;
	OC6CONbits.OCM=0;
	OC6CONbits.OCM=4;			// single pulse mode - must be reset in order
to generate another pulse
	IFS2bits.OC6IF = 0;

	return;
}

// Output Compare 7 interruption - IGNITION COIL 3
void __attribute__((interrupt, no_auto_psv)) _OC7Interrupt (void){


	IEC2bits.OC7IE=0;
	OC7CONbits.OCM=0;
	OC7CONbits.OCM=4;			// single pulse mode - must be reset in order
to generate another pulse
	IFS2bits.OC7IF = 0;

	return;
}

// Output Compare 8 interruption - IGNITION COIL 4
void __attribute__((interrupt, no_auto_psv)) _OC8Interrupt (void){


	IEC2bits.OC8IE=0;
	OC8CONbits.OCM=0;
	OC8CONbits.OCM=4;			// single pulse mode - must be reset in order
to generate another pulse
	IFS2bits.OC8IF = 0;

	return;
}

//=====================================================================
=========

int main( void ){
	int RPM=0;
	int Load=0;
	int index1=0;
	int index2=0;
	unsigned int q11,q12,q21,q22;
	unsigned int r1,r2;
	unsigned int x1,x2,y1,y2;

	unsigned int Ttotal = 0;				// time between events on a cilinder
	unsigned int BF = 0;					// value retrived from the
Base Fuel Map
	unsigned int BI = 0;					// value retrived from the
Base Ignition Map
	int aux=0;

	TRISDbits.TRISD8   = 1;		// port RD8 as input
	TRISDbits.TRISD9   = 1;		// port RD9 as input
	TRISDbits.TRISD11 = 0;		// port RD11 as output
	TRISDbits.TRISD0   = 0;		// port RD0 as output
```

```
        TRISDbits.TRISD1   = 0;              // port RD1 as output
        TRISDbits.TRISD2   = 0;              // port RD2 as output
        TRISDbits.TRISD3   = 0;              // port RD3 as output
        TRISDbits.TRISD4   = 0;              // port RD4 as output
        TRISDbits.TRISD5   = 0;              // port RD5 as output
        TRISDbits.TRISD6   = 0;              // port RD6 as output
        TRISDbits.TRISD7   = 0;              // port RD7 as output

        init_timer1( );

        init_can1( );

        INTinit( );

        initTIMER2 ( );

        InputCapINIT ( );

        OUTCOMPinit( );

        ADCinit ( );


        config_recebe_can1( );
        config_envia_can1( 1 , 3 , 2 , 1 , 58 , 0 );

        _RD11=1;


        while(1){

        RPM = ( 5 * clock ) / ( 32 * T3 );              //calculo da velocidade do motor


        Tcam = 6 * T3;

        Ttotal = (2 * Tcam) - ( alpha);

//      Load = MAP;
        Load = 20;

        // ********************************************* MAP calculations
*********************************************************

        index1=0;
        index2=0;
        while ( ( 500 + index1 * 500 ) <= RPM )
           index1++;
        while ( ( 10 + index2 * 10 ) <= Load )
           index2++;


        q11 = BFMap [ index2 - 1 ] [ index1 - 1 ];
        q21 = BFMap [ index2 - 1 ] [ index1 ];
        q12 = BFMap [ index2 ] [ index1 - 1 ];
        q22 = BFMap [ index2 ] [ index1 ];


        x1 = RPM_increment + (( index1 - 1 ) * RPM_increment);
        x2 = RPM_increment + (index1 * RPM_increment);
```

```
        y1 = Load_increment + (( index2 - 1 ) * Load_increment);
        y2 = Load_increment + (index2 * Load_increment);

        r1 = ( ( ( x2 - RPM ) * q11 ) / RPM_increment ) + ( ( ( RPM - x1 ) * q21 ) /
RPM_increment );
        r2 = ( ( ( x2 - RPM ) * q12 ) / RPM_increment ) + ( ( ( RPM - x1 ) * q22 ) /
RPM_increment );


        BF = ( ( ( y2 - Load ) * r1 ) / Load_increment ) + ( ( ( Load - y1 ) * r2 ) / Load_increment
);

        if(BF !=100)
           INJpulse = (Ttotal * BF) / 100;
        else
           INJpulse = Ttotal;

        q11 = BIMap [ index2 - 1 ] [ index1 - 1 ];
        q21 = BIMap [ index2 - 1 ] [ index1 ];
        q12 = BIMap [ index2 ] [ index1 - 1 ];
        q22 = BIMap [ index2 ] [ index1 ];

        r1 = ( ( ( x2 - RPM ) * q11 ) / RPM_increment ) + ( ( ( RPM - x1 ) * q21 ) /
RPM_increment );
        r2 = ( ( ( x2 - RPM ) * q12 ) / RPM_increment ) + ( ( ( RPM - x1 ) * q22 ) /
RPM_increment );

        BI = ( ( ( y2 - Load ) * r1 ) / Load_increment ) + ( ( ( Load - y1 ) * r2 ) / Load_increment );

        IGNdelay = 0x01B2;


        }
        return 0;
}
```

A2. CAN

```
#include <p30f6012a.h>


#define FCY             16000000                                        // (16 MHz *
PLL4)/4
#define BITRATE         1000000                                                 // 1
Mbps
#define NTQ             8
        // Number of Tq cycles which will make the CAN Bit Timing
#define BRP_VAL                 ((FCY/(2*NTQ*BITRATE))-1)     // Formulae used for
C1CFG1bits.BRP



void init_can1( void ){

//Config relacionado com o Baud Rate
        C1CTRLbits.REQOP = 4;  //Configuration mode
        while(C1CTRLbits.OPMODE != 4);

        C1CTRLbits.CANCKS = 0;
        C1CFG1bits.SJW = 0;                             //Sync = 1 Tq
        C1CFG1bits.BRP = BRP_VAL;
        C1CFG2bits.SEG2PH = 2;                  //Phase2 = 3Tq
        C1CFG2bits.SEG2PHTS = 1;
        C1CFG2bits.SAM = 0;                             //1=sample 3 vezes
        C1CFG2bits.SEG1PH = 1;                  //Phase1 = 2Tq
        C1CFG2bits.PRSEG = 1;                   //Prog Seg = 2Tq

}


void config_recebe_can1( void ){

//Prepara para receber
        C1RX0CON = C1RX1CON = 0x0000;       // Receive Buffer1 and 0 Status

        //Mascara Fica preparado para standard e extended
        C1RXM0SID  = C1RXM1SID  = 0x1FFC;
        C1RXM0EIDH = C1RXM1EIDH = 0x0FFF;
        C1RXM0EIDL = C1RXM1EIDL = 0xFC00;

        //Filtros
        //F0   Velocidade da roda frente esquerda??????????
        C1RXF0SIDbits.SID = 1;              //SID 11 bits
        C1RXF0SIDbits.EXIDE =  0;           //Enable do filtro para standard
        //F0

        //F1   Velocidade da roda frente direita??????????
        C1RXF1SIDbits.SID = 2;              //SID 11 bits
        C1RXF1SIDbits.EXIDE =  0;           //Enable do filtro para standard


}


void config_envia_can1( int sid, char nbytes, int byte_byte1, int byte_byte2, int byte_byte3, int
```

```
byte_byte4 ){


//Prepara para envio
	C1TX0CON = 0x0003;		// High priority

	C1TX0SIDbits.SID5_0 = sid & 0x03f;
	C1TX0SIDbits.SID10_6 = ( sid >> 6 ) & 0x1f;
	C1TX0SIDbits.SRR = 0;			//Normal message, nao È request a remote
transmission
	C1TX0SIDbits.TXIDE = 0;			//Transmite em standard

	C1TX0EID = 0x0000;				// EID = 0000000000000000000
(0x00000)

	C1TX0DLCbits.TXRTR = 0;				//Normal message, nao È request a
remote transmission
	C1TX0DLCbits.TXRB1 = 0;				//Zero como definicao do protocolo
CAN
	C1TX0DLCbits.TXRB0 = 0;				//Zero como definicao do protocolo
CAN
	C1TX0DLCbits.DLC = nbytes;		//Numero de bytes que vao ser enviados

	C1TX0B1 = (byte_byte2<<8)| byte_byte1 ; //Buffer carregado
	C1TX0B2 = (byte_byte4<<8)| byte_byte3 ; //Buffer carregado


 //Normal Operation Mode
	C1CTRLbits.REQOP = 0;
	while(C1CTRLbits.OPMODE != 0);//Espera que passe a normal operation mode


 //Inicio da transmissao
	C1TX0CONbits.TXREQ = 1;

//Prepara para envio



}
```

A3. Interrupts

```
#include <p30f6012A.h>

void INTinit(void){                                  //external interrupt initialization

        // interrupÁ„o do timer 1
        IFS0bits.T1IF = 0;                  // Clear timer 1 flag
//      IEC0bits.T1IE = 1;                  // Enable Timer 1 Interrupt
        IPC0bits.T1IP = 5;                  // Set Timer 1 interrupt priority

        //Interrupcao CAN1
        C1INTF = 0;                                     //Reset all The CAN Interrupts
        IFS1bits.C1IF = 0;                          //Reset the Interrupt Flag status register
        IPC6bits.C1IP = 3;                  //Interrup priority CAN1
        C1INTEbits.RX0IE = 1;               //Enable do receive Buffer 0 interrupt
        C1INTEbits.RX1IE = 1;               //Enable do receive Buffer 1 interrupt
        C1INTEbits.ERRIE = 1;               //Enable do interrupt de erro
        IEC1bits.C1IE = 1;                  //Enable the CAN1 Interruption

        return;

}
```

A4. Input Capture

```
#include <p30f6012A.h>

void InputCapINIT (void){

        IC1CON=0;                       // Disables Input Capture 1
        IC1CON=0x82;                    // Configures Input Capture 1
        IPC0bits.IC1IP=4;               // Set interrupt priority to 4
        IFS0bits.IC1IF=0;               // Clear interrupt flag

        IC2CON=0;                       // Disables Input Capture 2
        IC2CON=0x82;                    // Configures Input Capture 2
        IPC1bits.IC2IP=4;               // Set interrupt priority to 4
        IFS0bits.IC2IF=0;               // Clear interrupt flag

        IEC0bits.IC1IE=1;               // Enables the interruption for IC1
        IEC0bits.IC2IE=1;               // Enables the interruption for IC2

        return;
}
```

A5. Output Compare

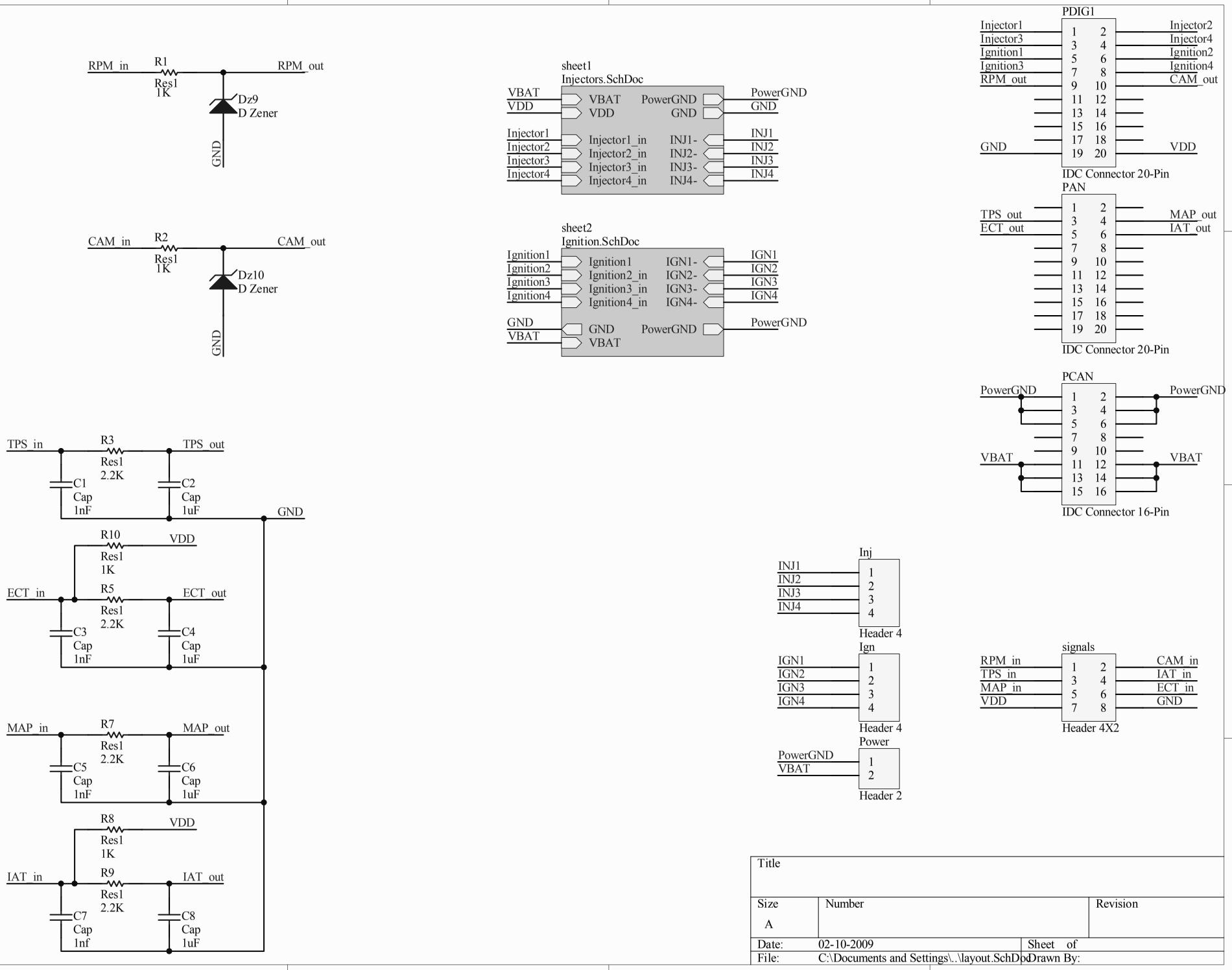
```
#include <p30f6012A.h>

// Output compare 1 initialization
void OUTCOMPinit(void){

// Output compare 1 configuration
        OC1CON=0x0;                             // clear register
        OC1CONbits.OCSIDL=0;        // continue working in idle mode
        OC1CONbits.OCTSEL=0;        // select timer 2
        OC1CONbits.OCM=4;                       // single pulse mode

        IFS0bits.OC1IF=0;           // clear interrupt flag
        IEC0bits.OC1IE=1;                       // enable interruption
        IPC0bits.OC1IP=4;                       // set interrupt priority

// Output compare 2 intialization
        OC2CON=0x0;                             // clear register
        OC2CONbits.OCSIDL=0;        // continue working in idle mode
        OC2CONbits.OCTSEL=0;        // select timer 2
        OC2CONbits.OCM=3;                       // single pulse mode

        IFS0bits.OC2IF=0;           // clear interrupt flag
        IEC0bits.OC2IE=1;                       // enable interruption
        IPC1bits.OC2IP=3;                       // set interrupt priority

// Output compare 3 intialization
        OC3CON=0x0;                             // clear register
        OC3CONbits.OCSIDL=0;        // continue working in idle mode
        OC3CONbits.OCTSEL=0;        // select timer 2
        OC3CONbits.OCM=3;                       // single pulse mode

        IFS1bits.OC3IF=0;           // clear interrupt flag
        IEC1bits.OC3IE=1;                       // enable interruption
        IPC4bits.OC3IP=3;                       // set interrupt priority

// Output compare 4 intialization
        OC4CON=0x0;                             // clear register
        OC4CONbits.OCSIDL=0;        // continue working in idle mode
        OC4CONbits.OCTSEL=0;        // select timer 2
        OC4CONbits.OCM=3;                       // single pulse mode

        IFS1bits.OC4IF=0;           // clear interrupt flag
        IEC1bits.OC4IE=1;                       // enable interruption
        IPC5bits.OC4IP=3;                       // set interrupt priority

// Output compare 5 intialization
        OC5CON=0x0;                             // clear register
        OC5CONbits.OCSIDL=0;        // continue working in idle mode
        OC5CONbits.OCTSEL=0;        // select timer 2
        OC5CONbits.OCM=3;                       // single pulse mode

        IFS2bits.OC5IF=0;           // clear interrupt flag
        IEC2bits.OC5IE=1;                       // enable interruption
        IPC8bits.OC5IP=3;                       // set interrupt priority

// Output compare 6 intialization
        OC6CON=0x0;                             // clear register
```

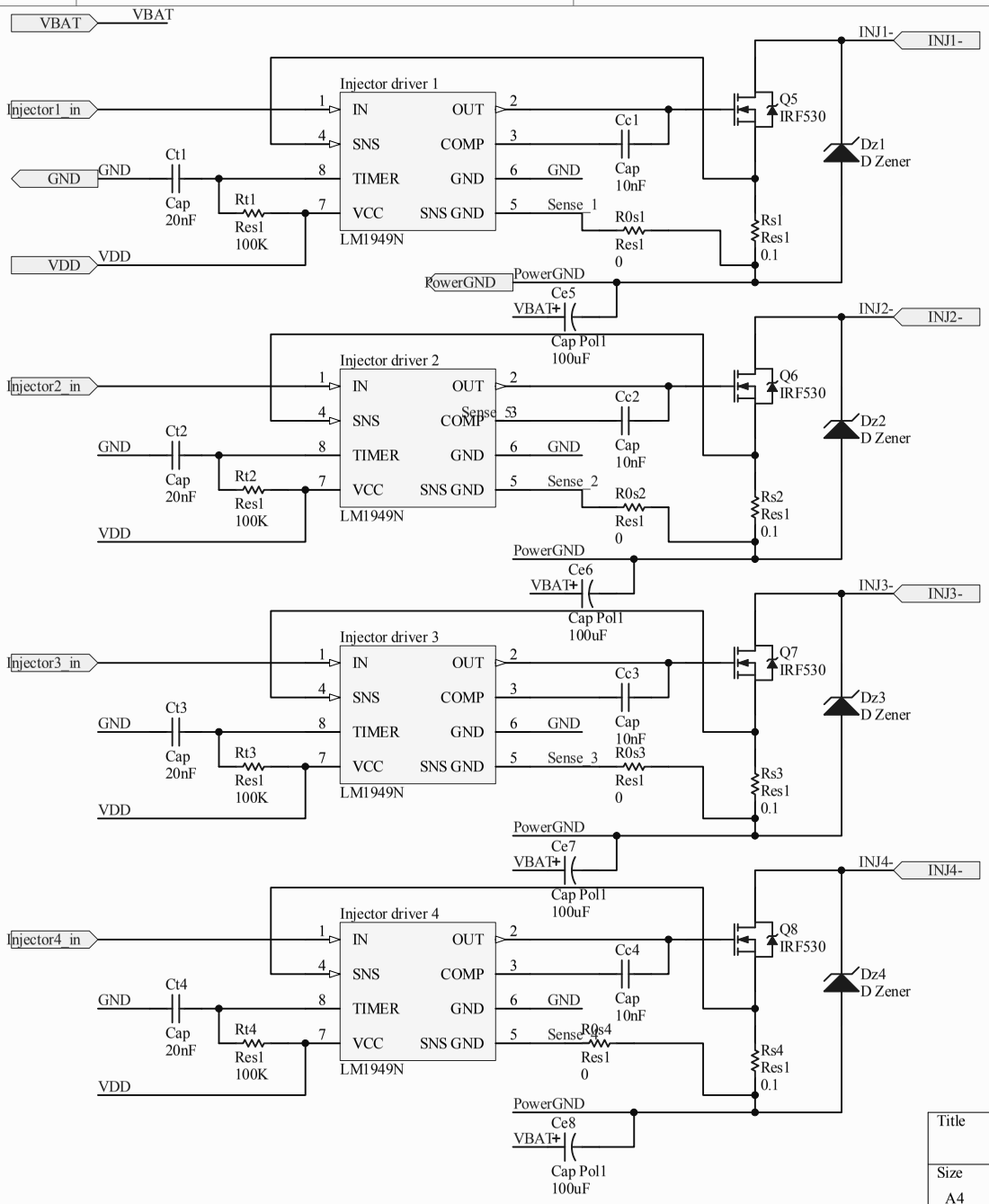
```
        OC6CONbits.OCSIDL=0;            // continue working in idle mode
        OC6CONbits.OCTSEL=0;            // select timer 2
        OC6CONbits.OCM=3;                       // single pulse mode

        IFS2bits.OC6IF=0;               // clear interrupt flag
        IEC2bits.OC6IE=1;                       // enable interruption
        IPC8bits.OC6IP=3;                       // set interrupt priority

// Output compare 7 intialization
        OC7CON=0x0;                                     // clear register
        OC7CONbits.OCSIDL=0;            // continue working in idle mode
        OC7CONbits.OCTSEL=0;            // select timer 2
        OC7CONbits.OCM=3;                       // single pulse mode

        IFS2bits.OC7IF=0;               // clear interrupt flag
        IEC2bits.OC7IE=1;                       // enable interruption
        IPC8bits.OC7IP=3;                       // set interrupt priority

// Output compare 8 intialization
        OC8CON=0x0;                                     // clear register
        OC8CONbits.OCSIDL=0;            // continue working in idle mode
        OC8CONbits.OCTSEL=0;            // select timer 2
        OC8CONbits.OCM=3;                       // single pulse mode

        IFS2bits.OC8IF=0;               // clear interrupt flag
        IEC2bits.OC8IE=1;                       // enable interruption
        IPC8bits.OC8IP=3;                       // set interrupt priority

}
```

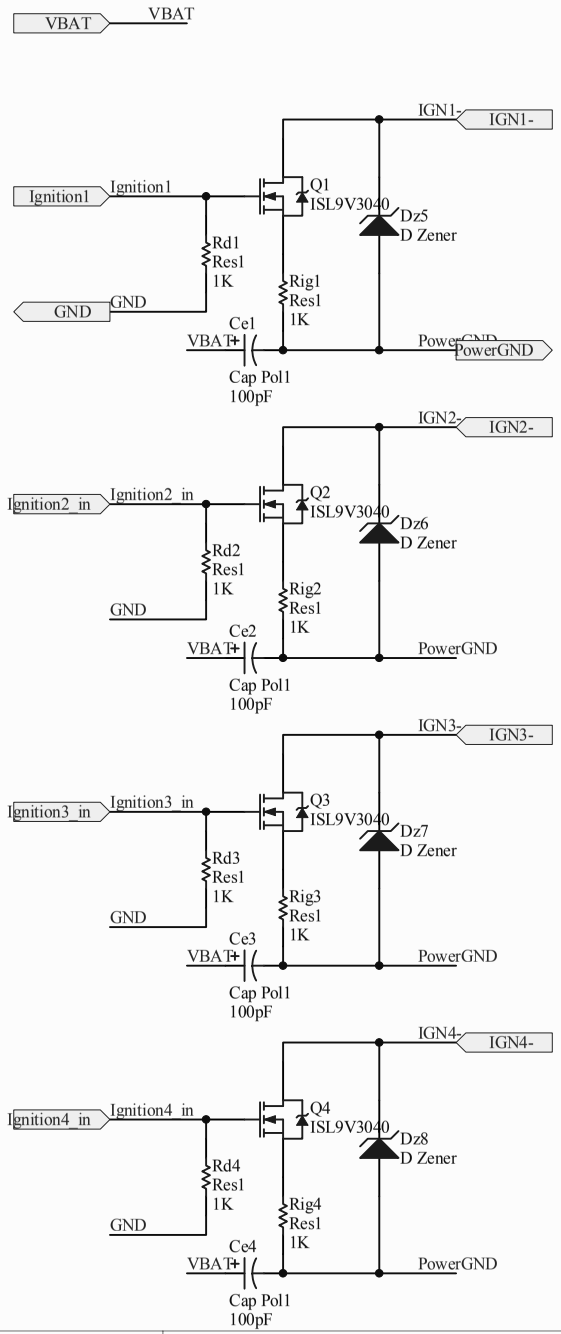
A6. Analog-to-Digital converter

```
#include <p30f6012A.h>


//ADC initialization
void ADCinit (void){

        IEC0bits.ADIE = 1;              // Enable ADC interrupts
        IPC2bits.ADIP = 3;              // Interrupt priority level 3
        IFS0bits.ADIF = 0;              // clear ADC flag

        ADCON1bits.ADON = 0;            // Stop ADC

        ADCON1=0x0;                     // clear ADCON1 register
        ADCON2=0x0;                     // clear ADCON2 register
        ADCON3=0x0;                     // clear ADCON3 register

        ADCON1bits.SSRC = 7;            // Conversion trigger source  - auto-convert
        ADCON1bits.FORM = 0;            // return integrer
        ADCON1bits.ASAM = 1;            // auto-start next conversion

        ADCON2bits.CSCNA= 1;            // scan inputs
        ADCON2bits.VCFG = 0;            // select AVDD e AVSS
        ADCON2bits.SMPI = 4;            // interrupt after 5 conversions

        ADCON3bits.SAMC = 31;
        ADCON3bits.ADRC = 0;            // Conversion clock derived from system clock
        ADCON3bits.ADCS = 1;            // X TCy

        ADCSSL = 0x013C;                // select pins to be scanned
        ADPCFG = 0xFEC3;                // configure ADC bits
        //ADCSSL = 0;
        //ADCHS = 0x0002;               // configure ADC channels

        ADCON1bits.ADON = 1;            // Start ADC conversions

        return;
}
```

Appendix B. Drivers circuit schematic


RPM_in
R1
Res1
1K
RPM_out
Dz9
D Zener
GND
CAM_in
R2
Res1
1K
CAM_out
Dz10
D Zener
GND
sheet1
Injectors.SchDoc
VBAT
VDD
Injector1
Injector2
Injector3
Injector4
VBAT
VDD
Injector1_in
Injector2_in
Injector3_in
Injector4_in
PowerGND
GND
INJ1-
INJ2-
INJ3-
INJ4-
PowerGND
GND
INJ1
INJ2
INJ3
INJ4
sheet2
Ignition.SchDoc
Ignition1
Ignition2
Ignition3
Ignition4
GND
VBAT
Ignition1
Ignition2_in
Ignition3_in
Ignition4_in
GND
VBAT
PowerGND
IGN1-
IGN2-
IGN3-
IGN4-
PowerGND
IGN1
IGN2
IGN3
IGN4
PDIG1
Injector1
Injector3
Ignition1
Ignition3
RPM_out
GND
Injector2
Injector4
Ignition2
Ignition4
CAM_out
VDD
IDC Connector 20-Pin
PAN
TPS_out
ECT_out
MAP_out
IAT_out
IDC Connector 20-Pin
PCAN
PowerGND
VBAT
PowerGND
VBAT
IDC Connector 16-Pin
TPS_in
R3
Res1
2.2K
TPS_out
C1
Cap
1nF
C2
Cap
1uF
GND
R10
Res1
1K
VDD
ECT_in
R5
Res1
2.2K
ECT_out
C3
Cap
1nF
C4
Cap
1uF
MAP_in
R7
Res1
2.2K
MAP_out
C5
Cap
1nF
C6
Cap
1uF
R8
Res1
1K
VDD
IAT_in
R9
Res1
2.2K
IAT_out
C7
Cap
1nf
C8
Cap
1uF
Inj
INJ1
INJ2
INJ3
INJ4
Header 4
Ign
IGN1
IGN2
IGN3
IGN4
Header 4
Power
PowerGND
VBAT
Header 2
signals
RPM_in
TPS_in
MAP_in
VDD
CAM_in
IAT_in
ECT_in
GND
Header 4X2
Title
Size
A
Number
Revision
Date:
02-10-2009
Sheet of
File:
C:\Documents and Settings\..\layout.SchDoc
Drawn By:


VBAT
VBAT
INJ1-
INJ1-
Injector driver 1
Injector1_in
IN
OUT
SNS
COMP
TIMER
GND
VCC
SNS GND
LM1949N
Cc1
Cap
10nF
Q5
IRF530
Dz1
D Zener
Ct1
Cap
20nF
Rt1
Res1
100K
R0s1
Res1
0
Rs1
Res1
0.1
Sense_1
GND
VDD
PowerGND
Ce5
VBAT+
Cap Pol1
100uF
INJ2-
INJ2-
Injector driver 2
Injector2_in
Cc2
Q6
IRF530
Dz2
D Zener
Ct2
Rt2
R0s2
Rs2
Sense_2
Ce6
INJ3-
INJ3-
Injector driver 3
Injector3_in
Cc3
Q7
IRF530
Dz3
D Zener
Ct3
Rt3
R0s3
Rs3
Sense_3
Ce7
INJ4-
INJ4-
Injector driver 4
Injector4_in
Cc4
Q8
IRF530
Dz4
D Zener
Ct4
Rt4
R0s4
Rs4
Sense_4
Ce8
Title
Size
A4
Number
Revision
Date:
02-10-2009
Sheet of
File:
C:\Documents and Settings\..\Injectors.SchDoc
Drawn By:



Title		
Size A4	Number	Revision
Date: 02-10-2009		Sheet of
File: C:\Documents and Settings\..\Ignition.SchDoc		Drawn By: