Higher Technological Institute

Faculty of Engineering

Mechatronics Department

Practical Implementation of an automobile Engine Control Module [ECM]

Graduation Project Submitted by:

Omar Abdelghany Ahmed	20160603
Mohamed Yehia Shaaban	20160927
Donia Khaled Amer	20160368
Ahmed Ashraf Fathy Abdel-Aziz	20160036
Ali Tarek Abdel-hai El Hadidy	20160563
Omar Adel Ahmed Mohamed	20160601
Mohamed Mostafa Abdel-Mawgood	20160908

Under the supervision of

Asso. Prof. Dr. Amal Ibrahim

2021

ACKNOWLEDGEMENT

Praise to ALLAH, All Mighty, by whose grace this work has been done.

Our gratitude goes to our project supervisor Dr. Amal Ibrahim.

We would like to express our very great appreciation to Technician Reda and the Whole crew for Supporting our project and providing us with the technical information.

We would like to offer our special thanks to each and every person who assisted us by putting their work and guidance on the internet in an open-source format including: ResearchGate.net, Lucas Sangoi Mendonça.

We wish to acknowledge every entity that made the components for the project available for purchase or provided us with guidance from where to buy them: Kitchen Maker Space, RAM Electronics, Free Electronics.

We are extremely grateful to our parents and families for their continuous love, prayers, sacrifice, care and support. This whole project would not have been possible without them.

ABSTRACT

Modern automobiles are electrically controlled by means of electronic control units "ECUs" which consist of microprocessors or microcontrollers. Generally, these control units are programmed according to standardized software architecture called the AUTOSAR Standard. Thus, these ECUs have a lot of restrictions when it comes to modifications.

For achieving an ECU model with fewer restrictions on modifications, this project was designed and implemented as to replace the standard automobile ECU with a simple student-scale easily modifiable ECU.

The current work presented works on controlling the IC engine operation (is Fuel Injectors, Ignition Coils) with the feedback received from the crank position sensor and an external potentiometer for the purpose of operating the engine at the idle run condition. To implement this concept, a Volkswagen engine 1400cc was used as a testing engine for the ECU being developed along the period of work. Initially, preparations were done in order to bring the engine and its sensors and actuators to work efficiencies on a reference ECU that was in company of the testing engine. After that, the actual design procedures were carried out in Both the hardware and software domains in order to acquire the required data from the reference ECU, calibrate the actuators and sensors used in the system and finally produce an operational ECU with the required hardware protection to maintain an acceptable level of durability.

The designed ECU at hand was successfully tested and proved the capability of controlling the operation of the engine at idle start condition. The potentiometer was used in the test for modifying the firing angle manually to observe the engine operation versus each angle for the purpose of finding the optimum firing angle relative to the idle speed of the crankshaft speed.

As this designed ECU is in the initial phase, future developments are expected to complement or modify this design to work on different operating modes and to introduce other sensors and actuators to the control system as to improve the system's performance and reduce the fuel consumption.

Table of Contents

List of Figures

Chapter 1

Introduction

Electronic devices in automotive area have been used on large scale and a control unit is mandatory for greater engine efficiency. By performing the spark advance and injection timing correctly in internal-combustion engines, an engine control unit improves drivability and helps to reduce fuel consumption.

Air pollution was recognized as a major problem in most cities around the world. as the requirements for lower and lower emissions continue, together with the need for better performance, other areas of engine control are constantly being investigated. This control is becoming even more important as the possibility of carbon dioxide emissions being included in future regulations increases. Some of the current and potential areas for further control of engine operation are included

Furthermore, with the embedded systems growth, powerful electronic control systems were applied for internal-combustion engines, considering that modern cars incorporate approximately one third of their parts in electronic components.

The ECU of an internal-combustion engine is a microprocessor that receives input signals from the engine and executes various tasks. the main output signals from ECU are the injection and ignition command. The signal for injection is a pulse-width signal with duty cycle related to injection timing according to injection map. The ignition system generates a very high voltage and sends it to each sparkplug which introduces energy into the chamber and produces a spark between the electrodes, initializing combustion.

The ECU receives the signals from the sensors, processes data and realizes calculations to generate output signals for the actuators. A software program is stored in ECU´s memory and is executed by a microcontroller that realizes tasks in order to improve efficiency.

This book presents an automotive engine control unit development considering the architecture of tasks for spark advance and injection timing using embedded C programming language for an ATmega128 - 8-bit AVR Microcontroller.

Scoop:

1- **Chapter 2:** a review of automobile electronic control units is presented. Its different types are discussed with details on each type and the communication protocols used.
2- **Chapter 3:** the hardware or any component used in this project is presented. The different types of hardware are discussed with details on each type.
3- **Chapter 4:** In this chapter the overall system layout will be presented. Then, the design procedures of the ECU will be discussed in deep details
4- **Chapter 5:** Used Software and Designed Code will be presented
5- **Chapter 6:** Subjects for future developments and improvements are presented, some recommendations for further working and the conclusion for the project at hand is presented.

Chapter 2

Review of automobile Electronic Control Units “ECUs”

Chapter description: in this chapter, a review of automobile electronic control units is presented. Its different types are discussed with details on each type and the communication protocols used. Then, a dive into the means, schemes and procedures that the manufactures follow in order to design an ECU according to both the standard software architecture and the necessary hardware circuits the offers the required level of safety and signal processing to/from the I/O ports of the ECU will be presented with a comparison between the technique followed in the designation of the current project and the technique followed by the ECU manufactures, showing the differences in the resulting designs.

2.1 Introduction

Before emissions laws were enacted, it was possible to build a car engine without microprocessors. With the enactment of increasingly stricter emissions laws, sophisticated control schemes were required in order to regulate the air/fuel mixture so that the catalytic converter could remove a lot of the pollution from the exhaust. In achieving that, a control unit was introduced as to be part of the solution. This control unit works on applying control on IC engines & regulating the air/fuel ratio mostly electrically. Afterwards, developments to the control of automobiles kept to be conducted which led to the modern form of control units, known as the Electronic Control Unit. An Electronic Control Unit (or ECU) in a modern automobile is an embedded electronic device or basically a digital computer that works on controlling and observing various crucial & important operations in an automobile. The control is based on information received by the ECU from different sensors placed in various places. These sensors produce signals that represent the automobile components' condition; which when received by the ECU, enable for precise measurements of the required parameters hence the right actions to be taken. The ECU also keeps a check on the performance of the car components and facilitates fault detection in any automated operation by connecting the ECU to an external OBD (On-Board Diagnosis) device.

The ECU uses closed-loop control, which is a scheme that monitors the output of a system in order to control the inputs sent to that system. It works on gathering massive number of data, looking up for values in tables and solving numerous equations each second in order to determine the spark timing and fuel injectors

operating duration. A typical ECU can consist of a 32-bit and 40 MHz processor. This is suitable for the code used in ECU software as it is estimated to take 1 MB of memory (in average).

Fig.1 shows the electronic control unit for a Mercedes-Benz C43 car.

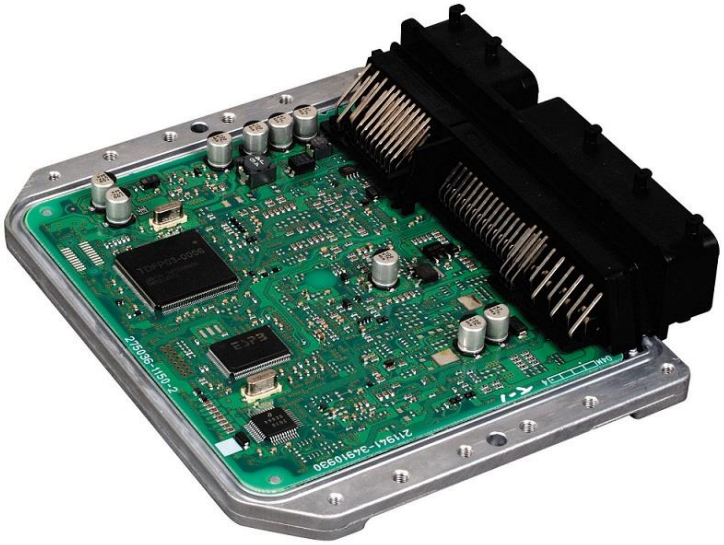

Figure 1: ECU for a Mercedes-Benz C43 car

An ECU can adjust the operation of some components of the automobile as the case with the throttle pedal. On slower and twister tracks, the engine control system Will help the driver gain more control over the throttle input by making the throttle's first half very sensitive trough circuit set-up. On the other hand, at high speeds, the circuit preparation for the accelerator will be made as to apply full engine acceleration at small movement.

2.2 ECU types

- Types of ECUs (according to their use) are listed below:
    1. ECM: Engine Control Module.
    2. EBCM: Electronic Brake Control Module.
    3. PCM: Powertrain Control Module.
    4. VCM: Vehicle Control Module.
    5. BCM: Body Control Module.
- A briefing of each type is listed in the following section:

1. Engine Control Module (ECM):

The ECM is an ECU in an internal combustion engine that control various engine functions like fuel injection, ignition timing and idle speed control system. The control is done based on data (like engine coolant temperature, air flow, crank etc.) received from various sensors.

2. Electronic Brake Control Module (EBCM):

This ECU is used in the ABS (anti-lock braking system) module of a Car. It was introduced to improve vehicle braking irrespective of the road and whether conditions. The EBCM regulates the braking systems on the basis of five inputs that it receives:

- The Brake: This input gives the status of the brake pedal i.e., deflection or assertion. This information is acquired in a digital or analog format.
- The 4 WD: This input gives the status in digital format whether the vehicle is in the wheel-drive mode.

- The Ignition: This input registers if the ignition key is in place, and if the engine is running or not.
- Vehicle Speed: This input gives the information about the speed of the vehicle.
- Wheel speed: In a typical application this will represent a set of 4 input signals that convey the information concerning the speed of each wheel. This information is used to derive all necessary information for the control algorithm.

3. Powertrain Control Module (PCM):

The PCM monitors and controls speed control, A/C, charging and Automatic

Transmission. The inputs that are fed to the PCM are from:

- Throttle position sensor,
- Output shaft speed sensor.
- Vehicle speed sensor.
- Erwine (CO).
- Brake switch.
- Cruise control switches.
- Ignition.
- Overdrive on/off switch.
- Governor pressure sensor.

Using these inputs, it does transmission control, valve control through PWM outputs, torque Converter clutch and transmission protection relay control and provides the feedback to the driver through the lamp.

4. **Vehicle Control Module (VCM):**

VCM is an ECU that takes care of systems like:

- Electronic Power steering (EPS) systems
- Adaptive Cruise control (ACC) systems
- Airbag control system (ACS) systems.
- Electronic Stability Control (ESC) systems.

The VCM is connected to various kinds of sensors to control various systems in the car. It takes inputs from crash sensors (micro-machined accelerometers) and sensors that detect occupant seater position, seat belt use and seat position to determine the force with which the frontal air bags should deploy. Similarly, it takes inputs from Steering wheel angle sensors, Wheel speed sensors, Yaw rate sensors, Lateral acceleration sensors to provide an output to the ESC for the safest driving experience.

5. **Body Control Module (BCM):**

It is an ECU that takes care of seating control unit, wiper control, power windows and power hoods in convertible cars (e.g., Benz SL Roadster).

2.3 Common automobile sensors & the associated I/O signals

Engine sensors that are almost used in every automobile engine work on monitoring the crucial parameters required for generating the signal for the fuel injectors and spark plugs.

These sensors are mentioned as follows:

1. Oxygen sensor.
2. Throttle position sensor.
3. Crankshaft position sensor.
4. MAP sensor.
5. Engine coolant temperature sensor.
6. MAF Sensor.

In the following. a brief explanation for the concept of work for each sensor is listed:

1. **Oxygen sensor:**

This sensor is installed on the exhaust manifold and works on measuring the ratio of the air-fuel mixture. It determines whether the burning is lean or rich. The signal that the oxygen sensor returns is an analog signal. The oxygen sensor is shown in Fig.2.

Figure 2: Oxygen sensor

2. **Throttle position sensor (or conventionally known as TPS):**

This sensor is a potentiometer that is located at the butterfly spindle which measures the throttle position of the butterfly valve to determine how much air is flowing in. Typically, this sensor sends input data for:

- Traction control.
- Fuel injectors.
- Cruise control.

The TPS sends its signal in the form of varying voltage as it is a variable resistance. Fig.3 shows the TPS.

Figure 3: Throttle position sensor

3. Crankshaft position sensor:

This sensor is an encoder that works on monitoring the crank shaft position in order to control ignition timing and other important engine parameters. It gives feedback in the form of frequency. The crankshaft position sensor is shown in Fig.4

Figure 4: Crankshaft position sensor

4. MAP sensor:

Manifold Absolute pressure sensor, is a pressure sensor that measures the pressure of air inside the intake manifold. This data is crucial for the ECU to determine how much fuel is needed to be injected by the fuel injectors. This sensor can do the application of Mass Air Flow sensor (MAFI by calculating the air density; this implies the need for data received from air temperature sensor. The MAP sensor sends its data in analog form. Fig.5 shows the MAP sensor.

Figure 5: MAP sensor

5. **Engine coolant temperature sensor (or conventionally known as ECT sensor):**

This sensor works on measuring the engine coolant temperature. This measured

Temperature is sent to the ECU in order to adjust the fuel injection or the ignition timing. ECT sensor sends data in the form of varying voltage as it consists of a thermistor which is the main sensing component of the sensor. Engine coolant temperature sensor is shown in Fig.6.

Figure 6: Engine coolant temperature sensor

6. **Mass Air Flow sensor (or conventionally known as MAF sensor):**

Mass Air Flow sensor is a sensor that measures the mass flow of air entering the intake manifold. In a typical system, either MAF or MAP is used for this purpose; rarely does it happen to include both in a single system. This sensor's signal is initially a varying current is converted to digital signal by means of a circuit. Fig.7 shows the MAF sensor.

Figure 7: Mass Air Flow sensor

2.4 Types of protocols

Automobile protocols can be divided into the following three major categories:

- Diagnostics
- Body and powertrain
- Multimedia and drive by wire

A brief explanation for each category is shown in the following section:

1. **Diagnostics:**

On-board diagnostics are into existence from the early 1980's. But in the recent years they have become highly sophisticated. Thus, there are highly reliable protocols just used for on-board diagnostics.

Some of the most frequently used ones are:

- **OBDII protocol:** This is a one of the most popularly used standards introduced in the mid 90's and takes care of the complete engine control and monitoring of the chassis and the accessories. It is used by almost all the automakers.
- **CAN ISO 11898:** Another very popular protocol used by almost all the automaker for on- board diagnostics. The pin details are as below.

- Pin 2 - J1850 Bus+
- Pin 4 - Chassis Ground
- Pin 5 - Signal Ground
- Pin 6 -CAN High 0-2284)
- Pin 7 - ISO 9141-2 K Line

- Pin 10 -11850 Bus
- Pin 14 - CAN Low (1-2284)
- Pin 15 - ISO 9141-2 L Line
- Pin 16 - Battery Power

– **Keyword 2000 and J1850:** These protocols are basically used by GM, Chrysler for on- board diagnostics. J1850 is a very old protocol and is being phased out.

2. **Body and Powertrain:**

Body and Powertrain networks may consist of CAN, LIN, or J1850 protocols. CAN is versatile protocol and is majorly used in various categories of in-vehicle networks. High speed CAN is often used for Powertrain applications such as engine timing to ensure that the car runs efficiently.

LIN: Local Interconnect Network (LIN) is a IJART based network that was developed strictly for body applications. For example, a LIN network connects all of the electric devices in a car door. LIN and CAN may coexist. It is majorly used by Chrysler, BMW and Volkswagen.

3. **Multimedia and drive by wire:**

– **MOST**: It is a fiber optic network that has been optimized for use in the automobile. It is designed for use with simple devices such as microphones and speakers along with more complex devices such as security devices such as those used to locate stolen automobiles. MOST technology is being developed and promoted by a Cooperation, which includes BMW, Daimler-Chrysler, and Audi.

– **IDB 1394:** It is the latest addition to the IDB family of in-vehicle networks,

designed for high-speed multimedia applications that require large amounts of information to be moved quickly on a vehicle. previously known as IDB-M, IDB-1394 is built on IEEE-1394 technology that has already gained wide acceptance in the consumer electronics community. The IDB-1394 specification defines the automotive grade physical layers (e.g., cables, connectors), power modes, and the higher layer protocols needed to assure interoperability of all I DB-1394 devices.

– **FlexRay:** It is a scalable, flexible high-speed communication system, which meets the increasing technical demands in the automobile industry. With its data rate of up to 10 MBits/s, it is ideal for X- by wire applications.

2.5 Conventional ECU design procedures & testing

Earlier, the traditional way to develop automotive embedded systems was to build hardware boards that represent all or part of each ECU type and part of its surroundings, often called plant models, and use them for bench testing. Unfortunately, the bench approach has many limitations, which are:

- Creating all the needed hardware boards is costly.
- The performance requirements of the most powerful ECU types (i.e., PCM) are so demanding that it is no longer possible to build boards that allow adequate measurements to be taken.
- The bench testing approach is based on a sequential design process where hardware is developed, plant model prototypes are built then software development begins, so it is time consuming.

In order to overcome these limitations, control design engineers have adopted a highly efficient design process often referred to as the "V" diagram. Though originally developed to encapsulate the design process of software applications, many different versions of this diagram can be found to describe different product design cycles. Fig.8 shows the design cycle followed by ECU manufacturers.

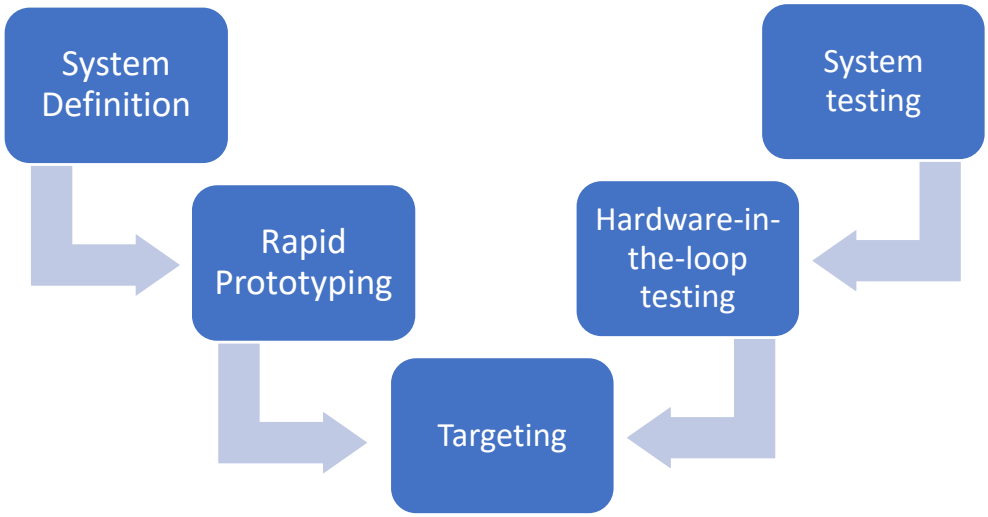

Figure 8: The design cycle followed by ECU manufacturers

Following the V-diagram, the general progression of time in the development stages starts from left to right. However, this is often an iterative process, and the actual development will not proceed linearly through these steps. Instead, you will spend time in each step and even have to backtrack from time to time. The goal is to make this cycle as efficient as possible by minimizing the amount of backtracking between steps as well as the time spent in each step.

The y-axis of this diagram Can be thought of as the level at which the system components are considered. Early on in the development, the requirements of the overall system must be considered. As the system is divided into sub-systems and

components, the process becomes very low-level down to the point of loading code onto individual processors.

Afterwards, components are integrated and tested together until such time that the entire system can enter final production testing. Therefore, the top of the diagram represents the high-level system view and the bottom of the diagram represents a very low-level view.

A brief description for the steps of the V-diagram is shown in the following section:

1. **System Definition:**

In this step, the design engineers document the needs and requirements of the project using spreadsheet or word processing applications. The documentation also takes care of the various specifications of the engine and the various norms it needs to comply with. It also marks the limit levels of the parameters involved in controlling the engine.

Once the specifications are documented the actual design process begins in which first a software model of the ECU and the engine is built. Then, a software-in-the-loop simulation is performed. This is done by connecting both the models generated earlier together in a closed feedback loop and then simulating to analyze the dynamic characteristics of the entire system.

During the simulation, the ECU model monitors the output from the Engine model and adjusts the inputs to the Engine model in order to improve the of various engine functions like fuel injection, ignition etc.

2. **Targeting:**

In this step, the core ECU model is modified to interface with the I/O available in the actual ECU and then is converted into a C code using a C — code generator. And then this code is downloaded as the control algorithm to the 32-bit microcontroller inside the ECU.

3. **Hardware-in-the-loop simulation (HIL):**

Once the code containing the control algorithm is downloaded to the ECU, testing of the performance of the ECU under extreme conditions can be performed. These conditions cannot be achieved in the real world; the only way to achieve them is by the HIL simulation.

In performing the HIL simulation, the actual ECU is tested by simulating an engine using the Engine model that was already created in the system definition step. In vice-versa to the RCP step, in HIL the software model of the Engine is downloaded to a Real-time hardware and the appropriate I/O interfaces are provided. These I/O are then connected to the actual ECU. then various engine conditions can be simulated and the ECU can be tested to its limits which wouldn't be possible if it was tested using an actual Engine.

4. **System testing:**

System testing is performed continuously in order to set checkpoints and determine the need to perform backtracks and adjustments in the software. This is a critical task as the ECU software consists of millions of lines of code in order to comply with standards and restrictions set for controlling automotive emissions mainly.

This step is partially carried out in. every previous step, and can be simply put in the following:

- Fault finding, defect reporting and verification.
- Defect reporting and verification.
- Validation of automotive software at various levels of testing starting from unit level until system testing.
- Test suite automation using scripts and modeling tools like LabVIEW Validation on vehicle simulators / lab cars.
- Vehicle Diagnostics Services /Automotive Diagnostics Services / OBD Diagnostics services.
- Hardware-in-the-loop simulations & Engine-in-the-loop simulations.

2.6 Present work concept

Automotive manufacturing is highly restricted in terms of their control units due to the need for both following the standards and complying with the governing rules that limit and retard the production speed of this industry. These restrictions thus render the automobile ECUs standardized and highly limited in parameters' modification. This ensures safety ease of operation for the normal use but forms a great ordeal for researchers looking for investigating a certain phenomenon or attempting to improve the performance, reduce fuel consumption…etc.

In order to achieve a platform for researchers and amateurs to carry out their investigations and introduce their suggested modifications with little restrictions that only ensure the basic level of safety, a different way for the development of the ECU software was to be followed. This software development way is basically

based on interfacing the ECUs microcontroller to the different sensors and actuators of the automobile, then calibrating the controlling parameters by trial and error with the help of data obtained from an operational reference ECU. The reference ECU was observed while operating a real engine. The observation was in terms of its I/O signals to different sensors and actuators connected to it for every/ corresponding unique throttle valve opening.

This student-scale simplified ECU designation has also paved the way for the educational field to deeply immerse in automotive control as well as other IC engines phenomena. It is an open source for anyone willing to research, investigate or in the operation and control of automobiles.

The ECU software at hand is a simple version that requires a lot of improvements for it to be fully functional. At the meantime, it only controls the operation of the engine (Electronic control with limitations for both acceleration and deceleration not to be of high values. It also has a flexible architecture as it can be modified to comply with any type of engines by changing certain control parameters in the C code and redesigning the PCB interfacing circuit.

Chapter 3

Hardware

Chapter description: In this chapter, the hardware or any component used in this project is presented. The different types of hardware (Mechanical, Electrical, Electronic) are discussed with details on each type. Some of working principles will be discussed also.

3.1 Mechanical

Internal Combustion Engine

The piston completes four separate strokes while turning the crankshaft. A stroke refers to the full travel of the piston along the cylinder, in either direction. The four separate strokes are termed:

1. Intake: This stroke of the piston begins at top dead center (T.D.C.) and ends at bottom dead center (B.D.C.). In this stroke the intake valve must be in the open position while the piston pulls an air-fuel mixture into the cylinder by producing vacuum pressure into the cylinder through its downward motion.

2. Compression: This stroke begins at B.D.C, or just at the end of the suction stroke, and ends at T.D.C. In this stroke the piston compresses the air-fuel mixture in preparation for ignition during the power stroke. Both the intake and exhaust valves are closed during this stage.

3. Power (Combustion): This is the start of the second revolution of the four-stroke cycle. At this point the crankshaft has completed a full 360-degree revolution. While the piston is at T.D.C. (the end of the compression stroke) the compressed air-fuel mixture is ignited by a spark plug (in a gasoline engine) or by heat generated by high compression (diesel engines), forcefully returning the piston to B.D.C. This stroke produces mechanical work from the engine to turn the crankshaft.

4. Exhaust: During the exhaust stroke, the piston once again returns from B.D.C. to T.D.C. while the exhaust valve is open. This action expels the spent air-fuel mixture through the exhaust valve.

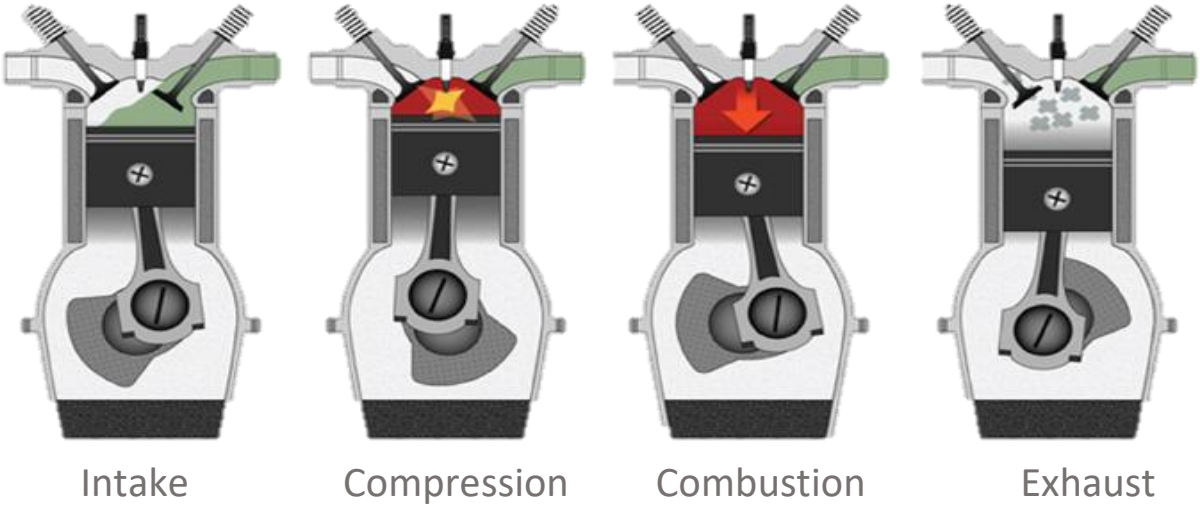

Figure 9: 4-Stroke (Otto Cycle)

Used Engine

- Model: Volkswagen polo 2000
- configuration: inline 4 cylinder
- Displacement: 1.4 Liter (1400 CC)
- Fuel injection: multi-point
- Valvetrain: DOHC (16 valve)
- Compression ratio: 10.5:1

3.2 Electrical

3.2.1 Ignition Coil

Now most automobiles run on gasoline, which they burn in an internal combustion engine to convert into motion. For combustion to take place, a spark is needed to ignite the fuel mixture in the engine. The vehicle's ignition system is designed so that a 12-volt battery can generate the very high voltage required to create such a discharge. The heart of this system is a device called an ignition coil. Which the following figure, Fig.10 shows the internal components of it.

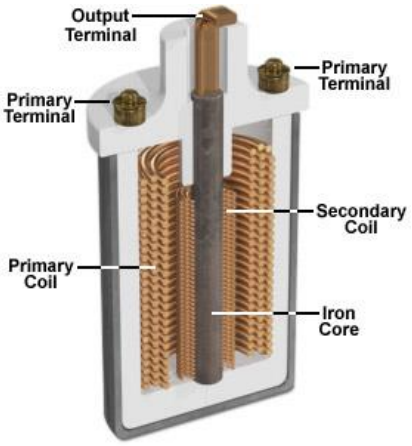

Figure 10: Internal of ignition coil

This coil is a kind of transformer. Transformers transfer voltage from one circuit to another, either as a higher voltage (as in a step-up transformer, of which the ignition coil is an example), or a lower voltage (a step-down transformer).

The key principle that makes transformers work is electromagnetic induction: a moving magnetic field, or a change in a stationary magnetic field (the case in our ignition coil), can induce a current in a wire exposed to that field.

This ignition coil is a pulse-type transformer. Like other transformers, it consists, in part, of two coils of wire. These are both wrapped around the same iron core. Because this is a step-up transformer, the secondary coil has far more turns of wire than the primary coil, which is wrapped around the secondary. In fact, the secondary coil has several thousand turns of thin wire, whereas the primary coil has just a few hundred. In your car, this allows some 40,000 volts of electricity to be generated by a modest battery.

- **The basic principle of an ignition coil**

To produce the required high voltages, ignition coils make use of the relationships that exist between electricity and magnetism. when an electric current flow through an electrical conductor such as a coil of wire, it creates a magnetic field around the coil. The magnetic field (or, more precisely, magnetic flux) is effectively a store of energy, which can then be converted back into electricity, as shown in Fig.11.

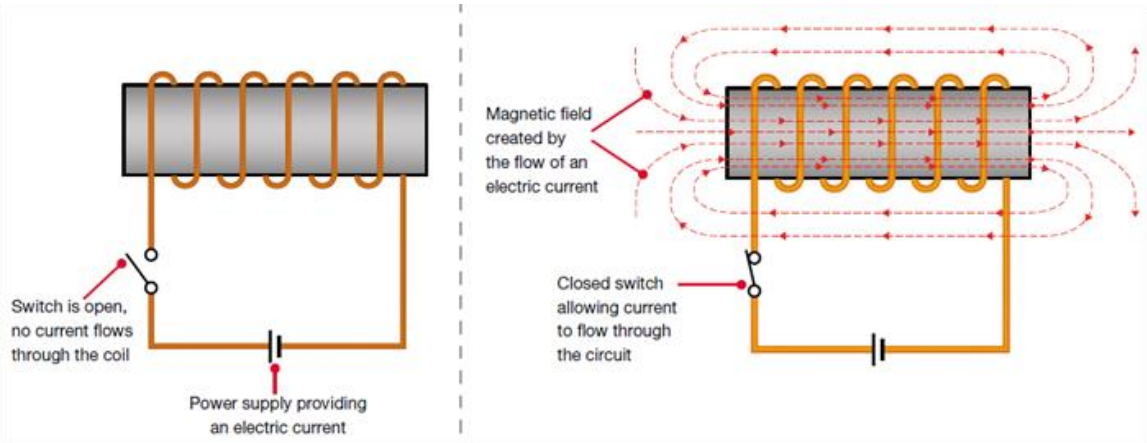

Figure 11: Creating a magnetic field by flowing electric current through a coil

When the electric current is initially switched on, the current flow rapidly increases to its maximum value. Simultaneously, the magnetic field or flux will progressively grow to its maximum strength, and will become stable when the electric current is stable. When the electric current is then switched off, the magnetic field will collapse back in towards the coil of wire.

- **Used Ignition systems**

Wasted spark ignition systems are a type of DIS, which uses one coil for every two cylinders. The coil provides the spark for one of the paired cylinders on the compression stroke and to the other on the exhaust stroke. Because the coil fires the spark plug on the exhaust stroke as well, it is appropriately named 'wasted spark ignition'. In effect, the spark plugs fire simultaneously and twice as often, which is shown in the following figures, Fig.12 and Fig.13.

Figure 12: WASTED SPARK IGNITION COIL

Ignition Coil one of the two paired spark plugs is always negative polarity while the other spark plug is always positive polarity. Negative polarity means the spark plug's center electrode is negatively charged and its ground electrode is positively charged. Positive polarity is the opposite. Each time the plug fires, a rapid exchange of the protons and electrons occurs, called ionization.

In a four-cylinder engine, for example, cylinders 1 and 4 are connected to one ignition coil, and cylinders 2 and 3 to another. The ignition coils are triggered by the ignition output stages in the electronic control unit. This receives the TDC signal from the crankshaft sensor in order to begin triggering the correct ignition coil.

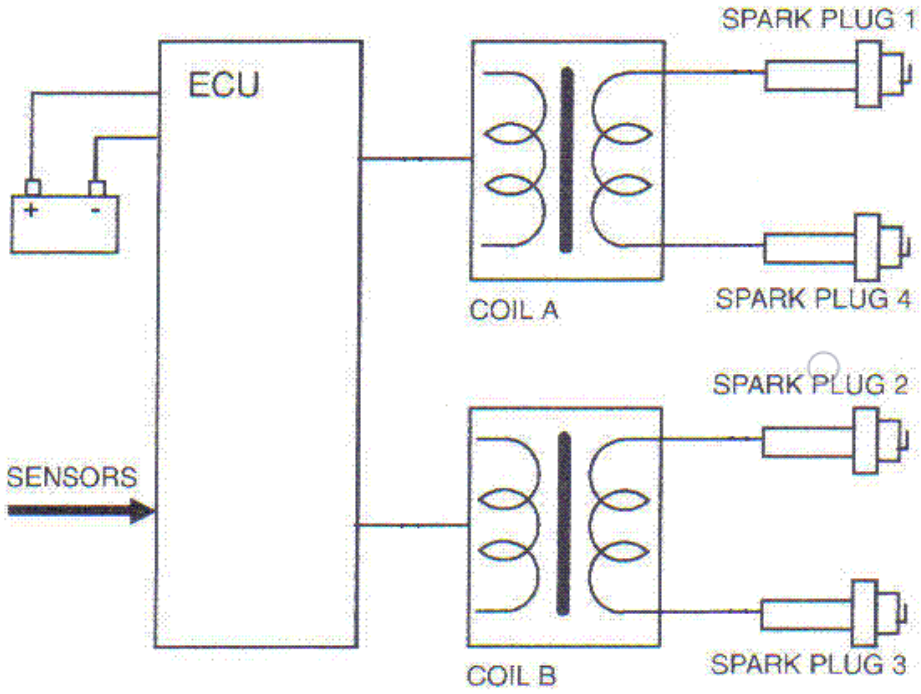

Figure 13: WASTED SPARK IGNITION SYSTEMS

3.2.2 Spark Plug

A spark plug is an electrical device that fits into the cylinder head of some internal combustion engines and ignites compressed aerosol gasoline by means of an electric spark, shown in the following figure Fig.14. Spark plugs have an insulated center electrode which is connected by a heavily insulated wire to an ignition coil or magneto circuit on the outside, forming, with a grounded terminal on the base of the plug, a spark gap inside the cylinder. Internal combustion engines can be divided into spark-ignition engines, which require spark plugs to begin combustion.

Figure 14: Spark Plug

- **The spark plug has two primary functions**

1. To ignite the air/fuel mixture. Electrical energy is transmitted through the spark plug, jumping the gap in the plugs firing end if the voltage supplied to the plug is high enough. This electrical spark ignites the gasoline/air mixture in the combustion chamber. To remove heat from the combustion chamber. Spark plugs cannot create heat, they can only remove heat. The temperature of the end of the plug\'s firing end must be kept low enough to prevent pre-ignition, but high enough to prevent fouling.

2. The spark plug works as a heat exchanger by pulling unwanted thermal energy from the combustion chamber and transferring heat to the engines cooling system. The heat range of a spark plug is defined as its ability dissipate heat from the tip.

- **Operation of Spark plug**

The plug is connected to the high voltage generated by an ignition coil or magnet. As the electrons flow from the coil, a voltage difference develops between the center electrode and side electrode. No current can flow because the fuel and air in the gap is an insulator, but as the voltage rises further, it begins to change the structure of the gases between the electrodes. Once the voltage exceeds the dielectric strength of the gases, the gases become ionized. The ionized gas becomes a conductor and allow electrons to flow across the gap. Spark plugs usually require voltage in excess of 20,000 volts to 'fire' properly.

The heat and pressure force the gases to react with each other, and at the end of the spark event there should be a small ball of fire in the spark gap as the gases burn on their own. The size of this fireball or kernel depends on the exact composition of the mixture between the electrodes and the level of combustion chamber turbulence at the time of the spark. A small kernel will make the engine run as though the ignition timing was retarded, and a large one as though the timing was advanced.

3.2.3 Fuel pump

A fuel pump is an essential component of the fuel system on a car. Fuel pumps use in carbureted and injected engines. By the depending to where fuel pump is using, they divided on the different types of fuel pumps, which is shown in Fig.16

- **Purpose of Fuel Pump:**

The Fuel pump`s purpose is to supply fuel from fuel tank to the engine cylinder. Fuel pump of modern injection systems must create the high pressure, so are often used Electrical fuel pumps.

The modern fuel injected engines often use electric fuel pumps, which are mounted inside the fuel tank..

- **Electric Fuel Pump:** In our time by the widespread adoption of electronic fuel injection are used electric fuel pumps, shown in Fg.15.

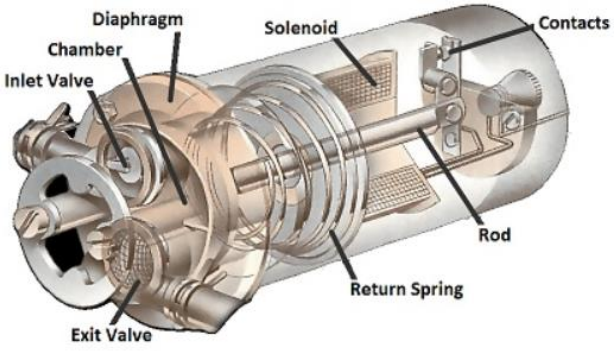

Figure 15: Electrical Fuel Pump

- **How Electric Fuel Pump Works:**

Electric fuel pumps are fairly simple but rely on many other systems to function properly. The Electric fuel pump is responsible for delivering pressurized fuel from the fuel tank to the engine.

When the ignition is activated, the (PCM) energizes a relay that supplies voltage to all electric fuel pumps.

3.2.4 Injectors

- **Fuel Injection System**

Fuel injection is a system for mixing fuel with air in an internal combustion engine.

The functional objectives for fuel injection systems vary but all of them share the central task of supplying fuel to the combustion process, Fig.16 describes the fuel injection system.

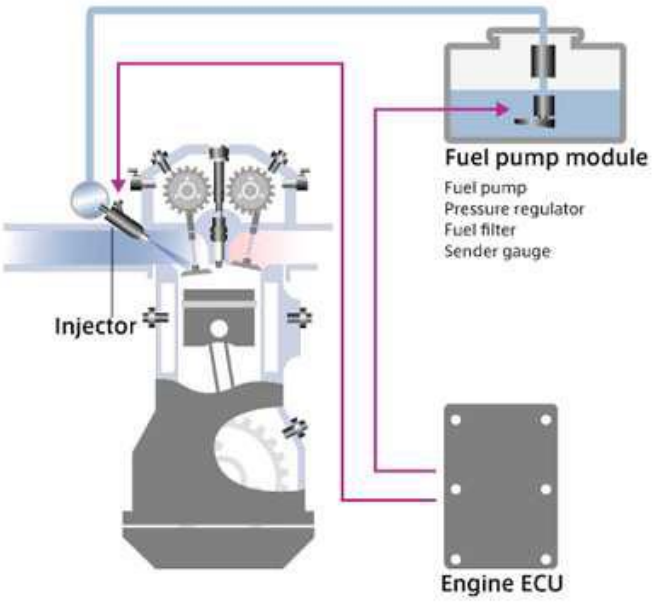

Figure 16: Fuel Injection System

- **Fuel Injectors**

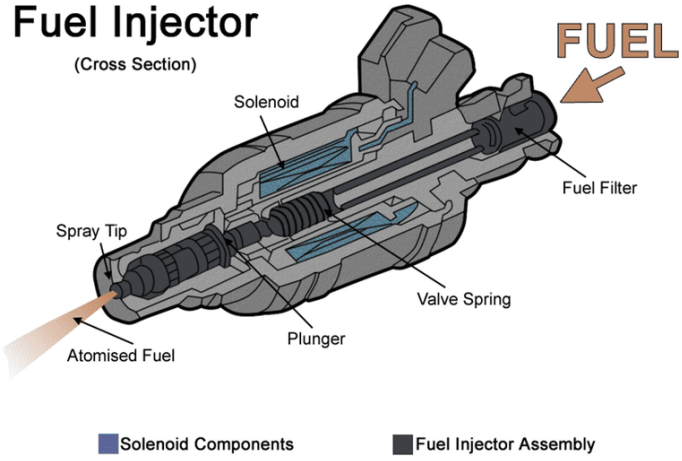

Figure 17: Fuel Injector

The fuel injector is a small nozzle into which liquid fuel is injected at high pressure. It works like a spray nozzle of a pressure washer. The placement of the injector can be in a different part of the engine depending upon the type of fuel injection system is being used. High pressure from the fuel pump feeds the fuel to the fuel injector, as shown in Fig.17.

The fuel injector pumps are individually cam-operated, of the spring-return plunger type and are driven off a common camshaft. They have means of ensuring equal distribution and metering of the fuel to all cylinders under all conditions of loading. Each pump is capable of being manually primed.

The fuel system is self-venting and any fuel spill from the injectors is returned directly to the fuel oil bulk storage tank.

Fuel pipes from the pumps to the injectors are sheathed such that any leakage is piped to a tank fitted with a level alarm.

Dual full-flow fine filters are provided upstream of the fuel injector pumps.

The amount of fuel supplied to the engine is determined by the amount of time the fuel injector stays open. This is called the “pulse width”, and it is controlled by the ECU.

The injectors are mounted in the intake manifold so that they spray fuel directly at the intake valves. A pipe called the “fuel rail”, which is shown in Fig.18. Supplies pressurized fuel to all of the injectors. In order to provide the correct amount of fuel for every operating condition, the engine control unit (ECU) has to monitor a huge number of input sensors.

Figure 18: Fuel Rail

- **Direct injection:**

Direct injection is a method in which fuel is introduced into internal combustion engines. It is common in gasoline (petrol) engines, as Fig.19 shows the direct and port injections. Direct injection systems are used in a gasoline engine to increase the efficiency and specific power output and also to reduce exhaust emissions.

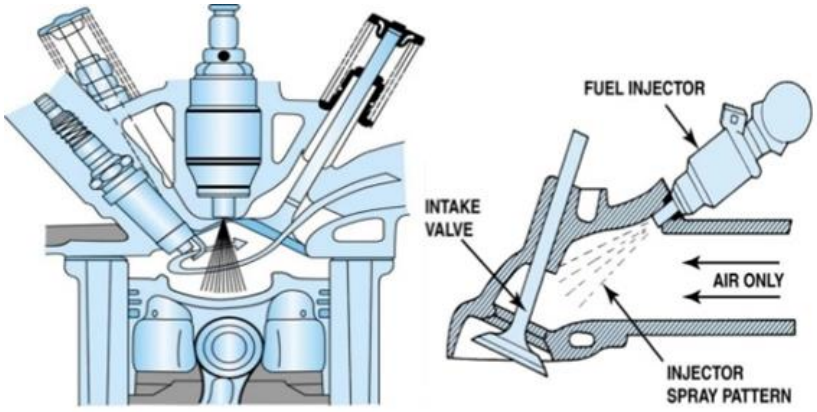

Figure 19: Direct injection and Port injection

- **What is a direct injection system?**

A direct injection system is a fuel injection process that allows fuel to be injected directly to the top of the piston in the combustion chamber. A gasoline direct

injection (GDI) which is also known as petrol direct injection (PDI) is a mixture formation system for internal combustion engines that work with gasoline (petrol). Fuel is injected into the combustion chamber directly.

it became more popular when introduced in an electronic GDI system in 1996 by Mitsubishi. However, the system has been widely adopted by the automotive industry in recent years.

3.2.5 Cars Electrical System

The electrical system of a car has two main functions. First, it must supply electrical energy to start and operate the engine. Secondly, it must provide the power to operate the lights, instruments and other electrical accessories, shown in Fig.20.

The modern car has about 60 meters of wire joining its electrical components. A color code has been standardized in most cars to allow quick recognition of the different circuits when any repairs are necessary.

The electrical system is usually divided into five circuits. They are the ignition circuit, starter circuit, charging circuit, lighting circuit and the accessories circuits, which are sometimes controlled by the ignition switch and in most cases protected by a fuse.

The electrical components in a car are connected through switches to one side of the battery, the other side of the battery is connected to the car body or chassis. In this way, the circuit to any component is completed through the car body which becomes the earth return. This method of wiring saves the cost of about 30 m of wire and also reduces the possibility of disconnection and simplifies fault-finding.

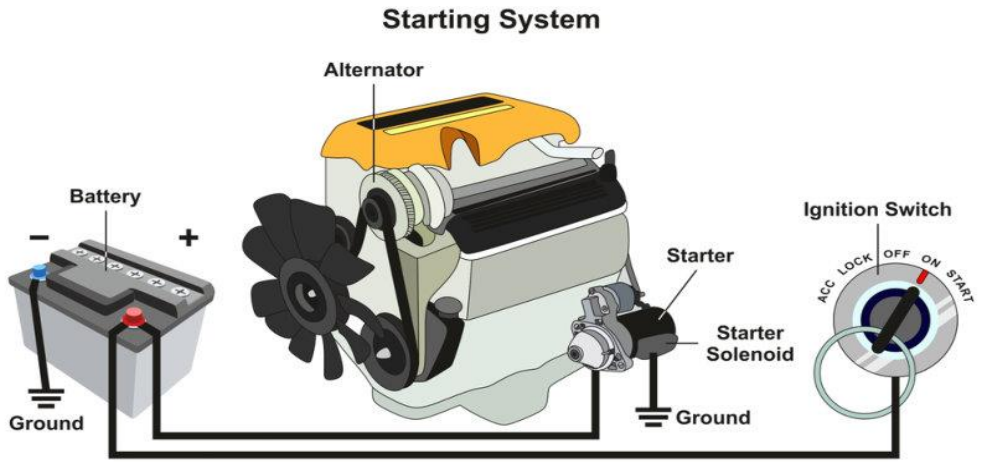

Figure 20: Car Electrical system

- **Main Components of The Electrical System**

Current for a car's electrical system is supplied by the battery when the engine is not running and by the generator, which is often a dynamo on older models. In most modern cars now, an alternator is fitted as it more easily supplies the current needed for the various accessories.

All the current is at the voltage of the battery usually 12 V or the generator (approx. 15 1/2 volts) except the current to the spark plugs which is boosted by the ignition system to as much as 30,000 volts.

- **Battery**

Figure 21: Battery

The battery is a rechargeable battery that is used to start a motor vehicle. Its main purpose is to provide an electric current to the electric-powered starting motor, which in turn starts the chemically-powered internal combustion engine that actually propels the vehicle, shown in Fig.21. Once the engine is running, power for the car's

electrical systems is still supplied by the battery, with the alternator charging the battery as demands increase or decrease.

- **The Alternator**

Alternators are typically found near the front of the engine and are driven by the crankshaft, which converts the pistons up-and-down movement into circular movement which is used to charge the battery and power all the electrical systems and accessories, shown in Fig.22.

Figure 22: The Alternator

Most alternators are mounted using brackets that bolt to a specific point on the engine. One of the brackets is usually a fixed point, while the other is adjustable to tighten the drive belt. A regulator used with the alternator protects the battery and other electrical equipment from receiving too much electrical power.

- **The Starter**

The starter motor converts electrical energy from the battery into mechanical energy, which is shown in Fig.23. However, when you turn the key in your car's ignition, the engine turns over and then cranks. However, getting it to crank is actually much more involved than you might think.

Figure 23: The Starter

It requires a flow of air into the engine, which can only be achieved by creating suction. If your engine isn't turning over, there's no air. No air means that fuel can't combust.

The starter motor is responsible for turning the engine over during ignition and allowing everything else to happen. When you turn the ignition on, the starter motor engages and turns the engine over allowing it to suck in air.

On the engine, a flywheel, with a ring gear attached around the edge, is fitted to the end of the crankshaft.

On the starter, the pinion is designed to fit into the grooves of the ring gear. When you turn the ignition switch, the electromagnet inside the body engages & pushes out a rod to which the pinion is attached. The pinion meets the flywheel and the starter motor turns. This spins the engine over, sucking in air with fuel.

As the engine turns over, the starter motor disengages, and the electromagnet stops. The rod retracts into the starter motor once more, taking the pinion out of contact with the flywheel and preventing potential damage.

3.2.6 Crankshaft Position Sensor

The crankshaft position sensor measures the rotation speed (RPMs) and the precise position of the engine crankshaft. Without a crankshaft position sensor, the engine wouldn't start.

The sensor is installed close to the main pulley (harmonic balancer). the sensor could be installed at the transmission bell housing, or in the engine cylinder block. In the technical literature, the crankshaft position sensor is abbreviated to CKP.

The sensor detects signals used by the engine ECU to calculate the crankshaft position, and the engine rotational speed.

- **Working principle of crankshaft position sensor**

The crankshaft position sensor is positioned so that teeth on the reductor ring attached to the crankshaft pass close to the sensor tip. The reductor ring has one or more teeth missing to provide the engine computer (ECU) with the reference point to the crankshaft position.

As the crankshaft rotates, the sensor produces a pulsed voltage signal, where each pulse corresponds to the tooth on the reductor ring.

The ECU uses the signal from the crankshaft position sensor to determine at what time to produce the spark and in which cylinder. The signal from the crankshaft position is also used to monitor if any of the cylinder's misfires. Crankshaft position sensor signal on the oscilloscope screen. If the signal from the sensor is missing, there will be no spark and fuel injectors won't operate.

- **Types of crankshaft position sensor**

- Inductive
- Hall sensor effect

- **Oscilloscope Measurements**

- **Inductive**

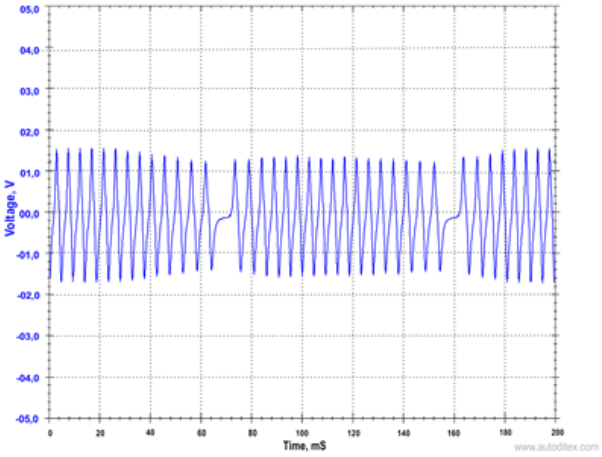

Figure 24: Inductive

- **Hall Sensor effect**

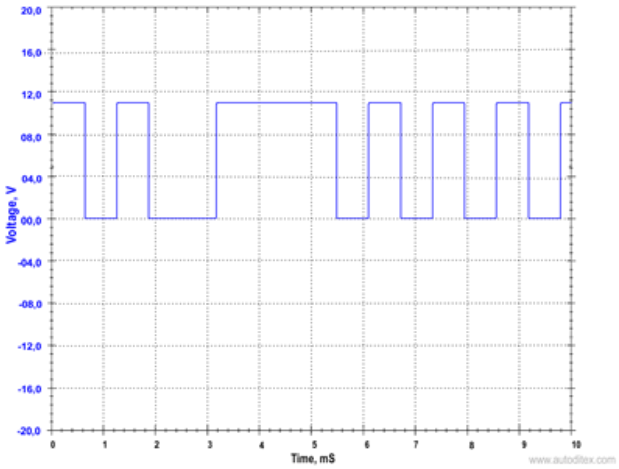

Figure 25: Hall sensor effect

We use **Hall Effect Sensor** that's generate square wave, signal it creates a digital signal, as shown in the following figure, Fig.26. Anyways, magnetic sensors convert magnetic or magnetically encoded information into electrical signals for processing by electronic circuits.

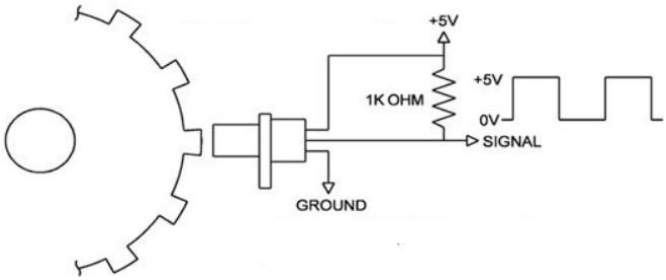

Figure 26: hall Effect sensor

3.3 Electronics

3.3.1 AVR AtMega128

- **Overview**

The Atmel® AVR® ATmega128 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture, shown in the following figure, Fig.27. The Atmel® AVR® core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

- **Pin Configurations**

VCC: Digital supply voltage.

GND: Ground.

Port A (PA7.PA0): Port A is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit).

Port B (PB7.PB0): Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit).

Port C (PC7.PC0): Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit).

Port D (PD7.PD0): Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit).

Port E (PE7.PE0): Port E is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit).

Port F (PF7.PF0): Port F serves as the analog inputs to the A/D Converter.

Port F also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. Port pins can provide internal pull-up resistors (selected for each bit).

Port F also serves the functions of the JTAG interface.

Port G (PG4.PG0): Port G is a 5-bit bi-directional I/O port with internal pull-up resistors (selected for each bit).

RESET: Reset input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running. Shorter pulses are not guaranteed to generate a reset.

XTAL1: Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

XTAL2: Output from the inverting Oscillator amplifier.

AVCC: AVCC is the supply voltage pin for Port F and the A/D Converter. It should be externally connected to VCC, even if the ADC is not used. If the ADC is used, it should be connected to VCC through a low-pass filter.

AREF: AREF is the analog reference pin for the A/D Converter.

PEN: PEN is a programming enable pin for the SPI Serial Programming mode, and is internally pulled high. By holding this pin low during a Power-on Reset.

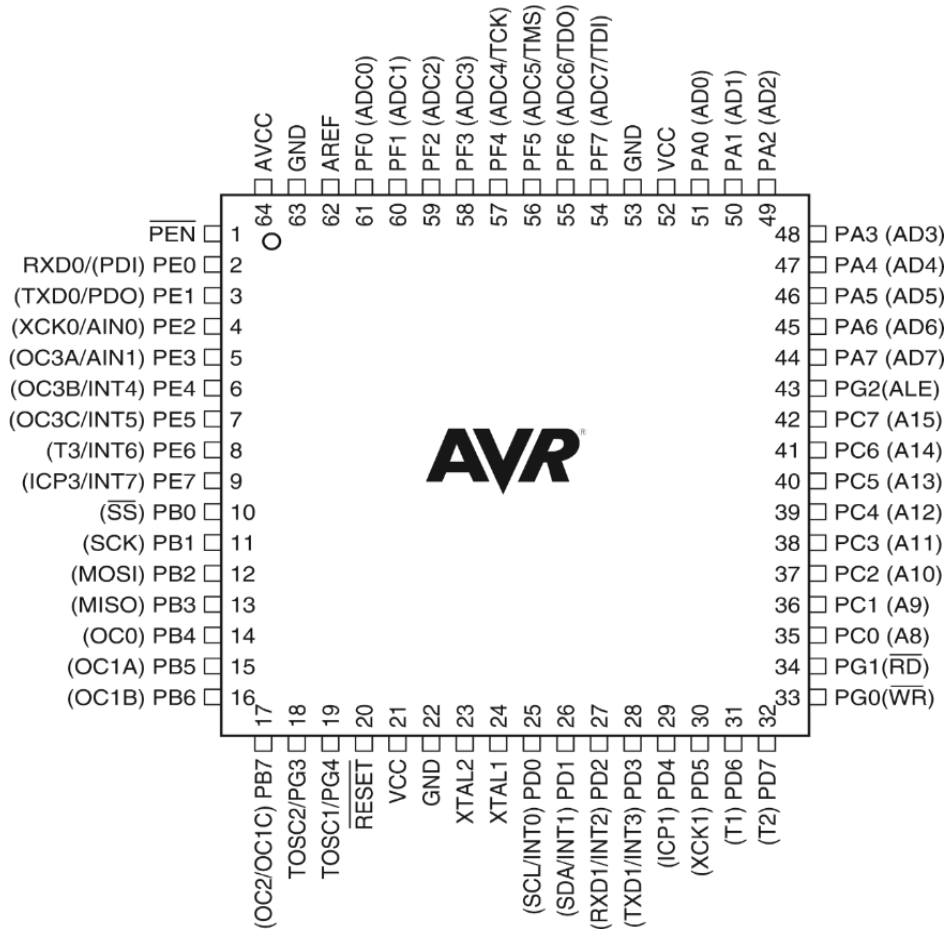

Figure 27: ATMEL AVR128

Why choosing this microcontroller?

Features:

- High-performance, Low-power Atmel®AVR®8-bit Microcontroller
- Advanced RISC Architecture

  – 32 x 8 General Purpose Working Registers + Peripheral Control Registers

  – Fully Static Operation

  – Up to 16MIPS Throughput at 16MHz

- High Endurance Non-volatile Memory segments
  - – 128Kbytes of In-System Self-programmable Flash program memory
  - – 4Kbytes EEPROM
  - – 4Kbytes Internal SRAM
  - – Write/Erase cycles: 10,000 Flash/100,000 EEPROM
  - – Data retention: 20 years at 85°C/100 years at 25°C (1)
  - – Optional Boot Code Section with Independent Lock Bits
  - – Up to 64Kbytes Optional External Memory Space
  - – Programming Lock for Software Security
  - – SPI Interface for In-System Programming
- JTAG (IEEE std. 1149.1 Compliant) Interface
  - – Boundary-scan Capabilities According to the JTAG Standard
  - – Extensive On-chip Debug Support
  - – Programming of Flash, EEPROM, Fuses and Lock Bits through the JTAG Interface

- Peripheral Features

  – Two 8-bit Timer/Counters with Separate Pre-scalers and Compare Modes

  – Two Expanded 16-bit Timer/Counters with Separate Pre-scaler, Compare Mode and Capture Mode

  – Real Time Counter with Separate Oscillator

  – Two 8-bit PWM Channels

  – 6 PWM Channels with Programmable Resolution from 2 to 16 Bits

  – Output Compare Modulator

  – 8-channel, 10-bit ADC

    - 8 Single-ended Channels
    - 7 Differential Channels
    - 2 Differential Channels with Programmable Gain at 1x, 10x, or 200x

  – Byte-oriented Two-wire Serial Interface

  – Dual Programmable Serial USARTs

  – Master/Slave SPI Serial Interface

  – Programmable Watchdog Timer with On-chip Oscillator

  – On-chip Analog Comparator

- Special Microcontroller Features

  – Power-on Reset and Programmable Brown-out Detection

  – Internal Calibrated RC Oscillator

  – External and Internal Interrupt Sources

  – Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby, and Extended Standby

  – Software Selectable Clock Frequency

  – ATmega103 Compatibility Mode Selected by a Fuse

  – Global Pull-up Disable

- I/O and Packages

  – 53 Programmable I/O Lines

  – 64-lead TQFP and 64-pad QFN/MLF

- Operating Voltages

  – 4.5 - 5.5V ATmega128

- Speed Grades

  – 0 - 16MHz ATmega128

3.3.2 USBasp

- Overview

  USBasp is a USB in-circuit programmer for Atmel AVR controllers. It simply consists of an ATMega8 and a few passive components. The programmer uses a firmware-only USB driver, no special USB controller is needed.

- Some of the key features include
  - Works under multiple platforms. Linux, Mac OS X and Windows are tested.
  - Programming speed is up to 5kBytes/sec.

    Slow SCK option to support targets with low clock speed (< 1.5MHz).

- **Layout**

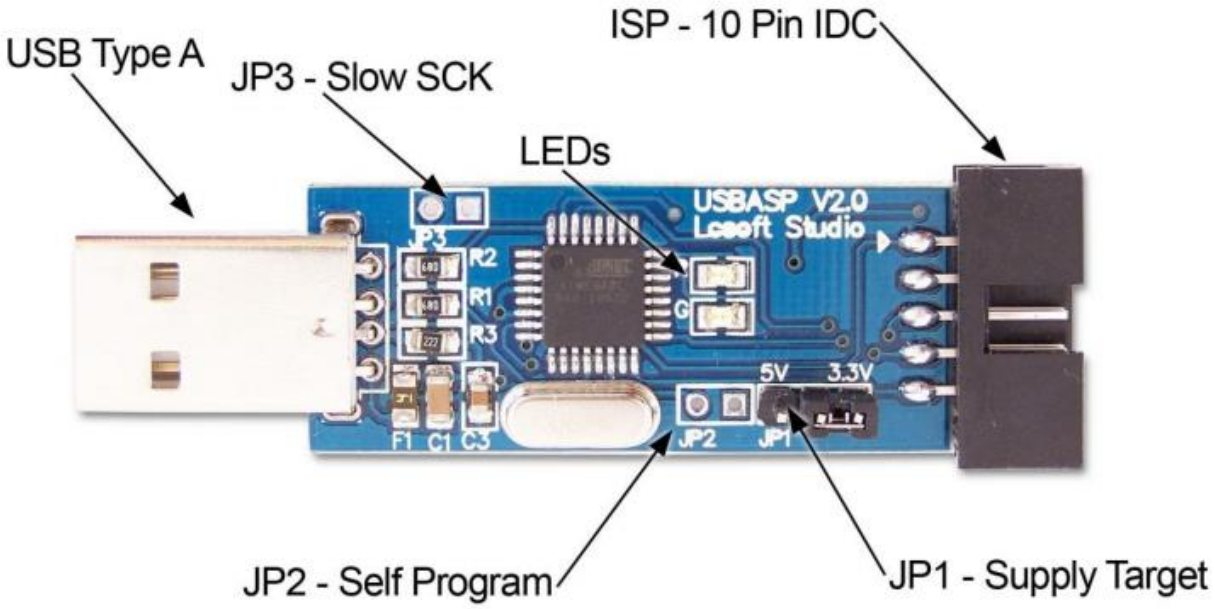

Figure 28: USBasp Layout

USB Type A

The USB end of the programmer connects directly into your computers USB port.

- **ISP – 10 pin IDC**
  - The 10 pin ISP connection provides an interface to the microcontroller. This interface uses a

    10 pin IDC connector and the pinout

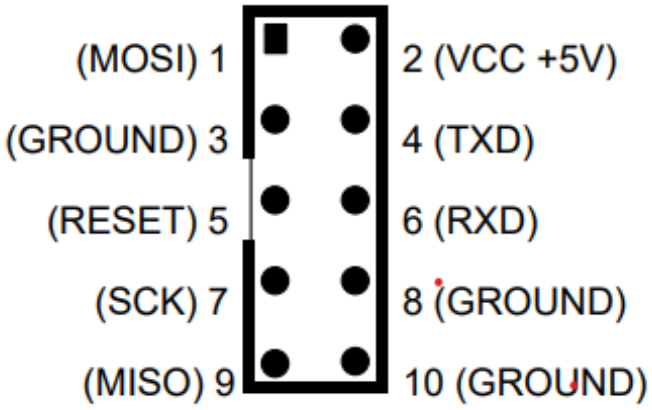

Figure 29: USBasp's pin configuration

- **JP1 – Supply Target**

  This jumper controls the voltage on the ISP VCC connector. It can be set to +3.3V, +5V or disable this jumper if the target device has its own power source.

- **JP2 – Self Program**

  This jumper is used to update the firmware of the USBasp programmer. In order to update the firmware, you will need 2 programmers. One to be programmed and the other to do the programming.

- **JP3 – Slow SCK**

  When this jumper is selected, the slow clock mode is enabled. If the target clock is lower than 1.5 MHz, you need to set this jumper. Then SCK is scaled down from 375 kHz to about 8 kHz.

- **LEDs**

The USBASP programmer has 2 LEDs near the ISP connection. These have the following functions:

LED R – Programmer communicating with target device

LED G – Power

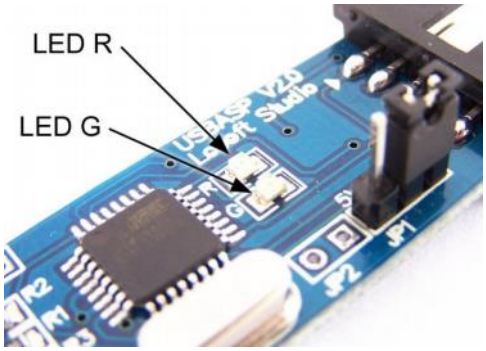

Figure 30: USBasp's Leds

Whilst you might assume that LED R is red and LED G is green, they do vary depending on the batch. The one I use for instance has 2 red LEDs.

- **Circuit Diagram**

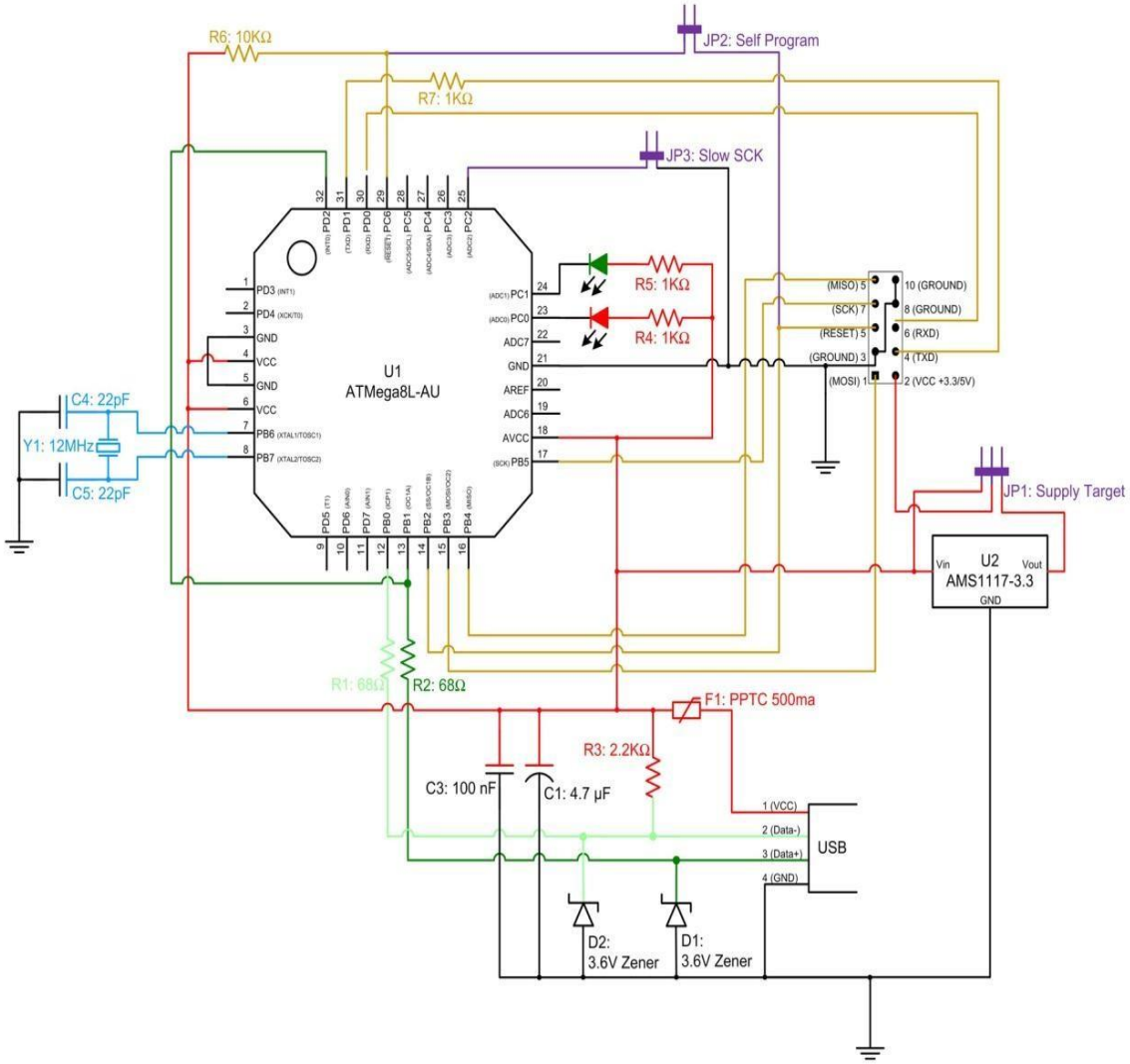

Figure 31: Circuit Diagram

3.3.3 TTL (Serial to USB communication)

CP2102 chip from SiLabs is a single chip USB to UART Bridge IC. It requires minimal external components. CP2102 can be used to migrate legacy serial port-based devices to USB. Hobbyists can use it as a powerful tool to make all kinds of PC interfaced projects.

- **Working**

This is an USB2.0 to TTL UART Converter module which is based on CP2102 Bridge by SiLabs. This module can be used with Laptop's which don't have standard serial port. This module creates a virtual COM port using USB on your computer which can support various standard Baud Rates for serial communication. You just need to install the driver using a setup file which automatically installs correct driver files for Windows XP/Vista/ 7. After driver installation, plug the module into any USB port of your PC. Finally, a new COM port is made available to the PC. The feature which makes it more convenient is the TTL level data I/o. So, you don't need to make a RS232 to TTL converter using chips like MAX232. The Rx and Tx pin can be connected directly to the MCUs pins (assuming 5v I/o).

- **Pinouts**

This module has 6 pin breakout which includes

- TXD = Transmit Output - Connect to Receive Pin (RXD) of Micro controller. This pin is TX pin of CP2102 on board.
- RXD = Receive Input - Connect to Transmit Pin (TXD) of Micro controller. This pin is RX pin of CP2102 on board.
- GND = Should be common to microcontroller ground.
- 3V3 = Optional output to power external circuit up to 50mA.
- 5V = Optional output to power external circuit up to 500mA

- DTR/RST = Optional output pin to reset external microcontrollers like Arduino.

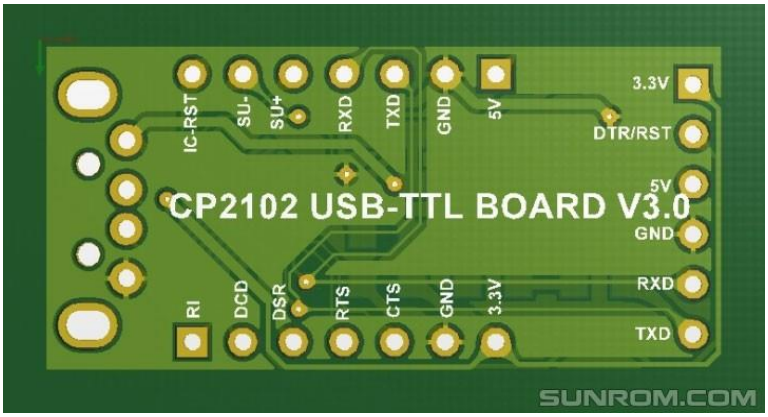

Figure 32: Pinouts

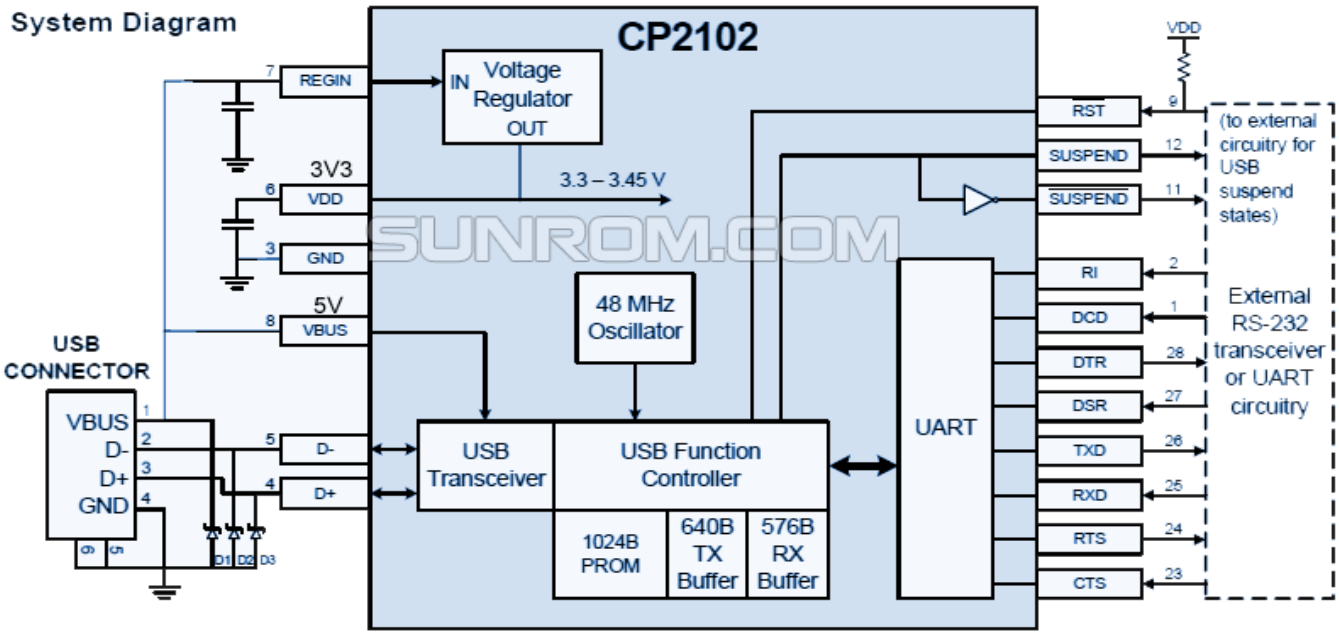

Figure 33: System Diagram

High Quality USB to TTL converter, comes with a 4-pin extension cable.

Features:

- Stable and reliable chipset CP2102.
- USB specification 2.0 compliant with full-speed 12Mbps.
- Standard USB type A male and TTL 6pin connector.
- 6pins for 3.3V, RST, TXD, RXD, GND & 5V.
- All handshaking and modem interface signals.
- Baud rates: 300 bps to 1.5 Mbps.
- Byte receive buffer; 640 bytes transmit buffer.
- Hardware or X-On/X-Off handshaking supported.
- Event character support Line break transmission.
- USB suspend states supported via SUSPEND pins.
- Temperature Range: -40 to +85.
- Size: 42mm X 15mm.

- **Baud Rate**.

The frequency of the bits of binary code as they are communicated is known as the computer's baud rate. Baud rate is expressed in bits per second (b/S), kilobits per second (KKB/S), or megabits per second (Mb/S).

Chapter 4

System layout, Design procedures & Implementation practices

Chapter description: In this chapter the overall system layout will be presented. Then, the design procedures of the ECU will be discussed in deep details. The implementation practices carried out in both hardware and software domains are also discussed. Documentations for both the hardware circuits' schematic and the software flowcharts companied by detailed explanations for each component are also included

4.1. Introduction

The design strategy for the present automobile ECU at hand varies from that followed in the industry domain. Thin strategy is initially based on direct interfacing of the different sensors of the automobile with the microcontroller that will form the main functional component in the ECU which performs data acquisition, analysis, calculations and decision making with the help of hardware circuitry.

The code development technique is basically based on the trial-and-error method and the reference performance behavior for the ECU was based on an actual operational ECU connected to a real Volkswagen 1400cc engine. This operational ECU had its I/O signals observed in run time by means of an oscillator, a multimeter and serial port monitor.

The C code developed for the ECU's microcontroller; The microcontroller used is atmega128 -8-bits with a speed limit of 16MHZ. Fig.34 shows the MCU used for this project

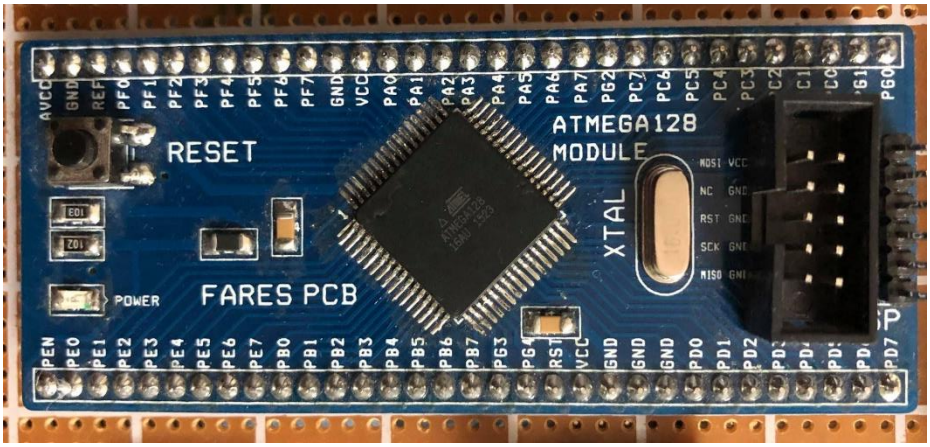

Figure 34: MCU used in project

4.2 system layout

The designed ECU at hand with its hardware and software components forms an initial designation phase for an open-source ECU. This initial phase aims for setting the first building blocks for the full designation to be implemented in the future; by controlling the operation of the very fundamental and crucial component of an automobile which is the engine.

In achieving though, many steps were followed as will be seen later in this chapter; to finally produce the required hardware circuit for conditioning the input/output signals from/to the various sensors/actuators of concern for this phase and provide the required initial interfacing with the main processing unit "the microcontroller". These steps also directly contributed in the development of the microcontroller that forms the main processing unit for the ECU and carries out required calculations and decisions for ensuring precise control of the automobile actuators on bases of data acquired from various sensors.

This phase concluded at the successful run of the engine in the idle start condition without including the throttle position sensor feedback for control such that, changes in the throttle position sensor led to the terminator of the engine run Fig.35 shows the system layout in the form of a block diagram

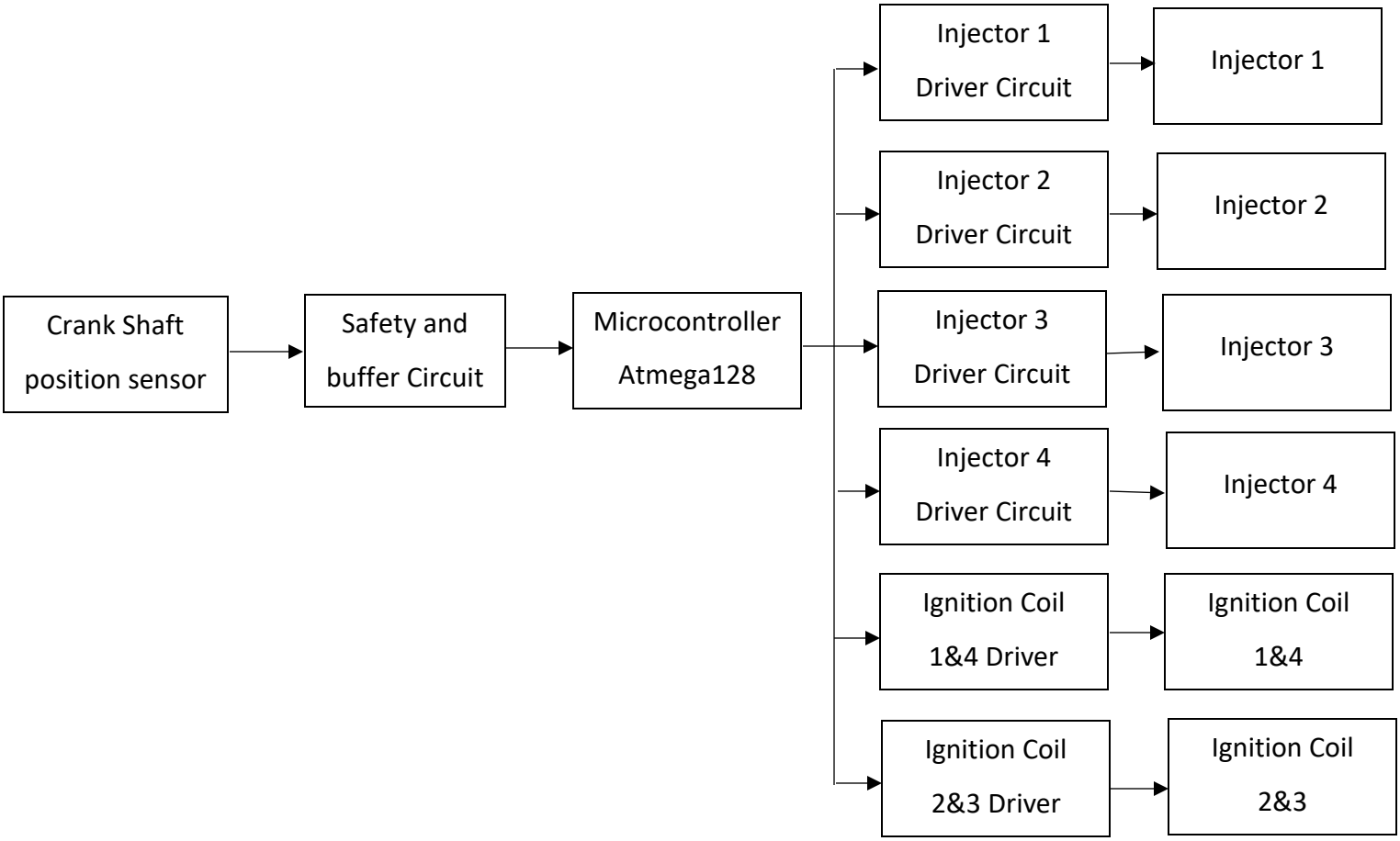

Figure 35: The overall system layout for idle start condition of the engine

4.3. Work order and design procedures

The work for this project was initiated in Sept/2020 but initial researches were done before the actual work was carried out. It took 8 months in total to perform a successful run of the designed ECU presented. The work sequence is listed and discussed in the following:

1. **A customized car was borrowed:** to carry a 1400cc Volkswagen engine
2. **A full checkup of engine parts was performed:** The checkup of the engine parts resulted in the detection of missing components that required purchase.

3. **The engine oil was changed.**
4. **First engine run attempt was performed (succeeded):** This engine run was performed after essential missing and deficient sensors and actuators for initial run; were bought, maintained and installed. The run of the engine was controlled by the default ECU of the 1400cc Volkswagen engine.
5. **An engine Start/Stop interfacing switches was designed:** This board was designed for the purpose of facilitating the run attempts of the engine, reducing the risk of flame propagation due to the electric sparks created in manual latching of the battery's terminals with the circuit and separately control the operation of both the ECU and the fuel pump. More details on this will be discussed in the "Implementation in the Hardware domain" section.
6. **Fuel tank was maintained and rearranged:** The inlet and outlet rearrangement were performed to ensure ease of run for the fuel pump and to avoid applying huge loads on it.
7. **Reverse engineering of the engine cable was performed:** The engine cable connects all sensors and actuators with their corresponding pins in the ECU for controlling the automobile operation and acquiring data from its sensors. This reverse engineering process resulted in obtaining the wiring diagram of the engine cable which was the first crucial step in the ECU design. More details on this will be discussed in the "Implementation in the Hardware domain" section.
8. **Bridge configured wires were prepared and installed:** These wires work on connecting the engine cable's terminals with their corresponding pins of the Ecu while providing a testing point for each.

9. **An initial Breadboard for data acquisition was designed (failed):** This board was designed to receive the signals from the automobile sensors of concern and perform the required processing and Conditioning in order for the data acquiring microcontroller to read them. More details on this will be discussed in the "Implementation in the Hardware domain" section.

10. **Coding of the data acquisition MCU:** The code was developed on AtMega128 module. The coding approach was the standard AVR coding method. succeeded in reading the data from the automobile sensors with a non-satisfying time response. More details on this will be discussed in the "Implementation in the Software domain" section.

11. **Analyzing the crank position sensor output signal for index detection:** The index signal of the crank sensor indicates the reference position of the crank shaft which ensures that control operations of the engine are being carried out in their correct timings. For that, it was very crucial to precisely detect this signal. In achieving so, a code was used to represent the signal visually and facilitate mapping it out. More details on this will be discussed in the "Implementation in the Hardware domain" section.

12. **A crank sensor signal conditioning circuit was designed (succeeded):** This circuit was designed as to receive the frequency signal generated by the crank position sensor and filter it. more details on this will be discussed in the "Implementation in the Hardware domain" section.

13. **Developing a code and designing a circuit for reading the ignition coils' and injectors' timing and pulse width (succeeded):** Knowing the timing and the pulse width of the injectors and the ignition coils is very important for designing a driver

for them. A wrong timing for any of them may result in the breakdown of the engine and a wrong pulse width may lead to the burn of the coils or injecting an undesired amount of fuel for the current condition. More details on this will be discussed in the "Implementation in the Hardware domain" section.

14. A calibration code, circuit and set-up for injectors control were developed, designed and prepared (succeeded): The circuit alongside with the code developed enabled the calibration through trial and observation via syringe tubes to be performed. This resulted in the precise control of the injectors' operation. More details on this will be discussed in both the "Implementation in the Hardware domain" & “implementation in the Software domain” sections.

15. An initial code for driving the ignition coils was developed (failed): This code was meant to precisely control the operation of the ignition coils in terms of timing while taking the pulse width requirement in consideration. As will be seen later, this caused the failure (burn) of the ignition coils due to over charging the coils. More details on this will be discussed in both the "Implementation in the Hardware domain" & "Implementation in the Software domain" sections.

16. A driver circuit for both the injectors and the ignition coils was designed (succeeded): The circuit required for driving both of the injectors and the ignition coils were implemented successfully as the error that will arise in the upcoming first test was found to be a software error. More details on this will be discussed in the "Implementation in the Hardware domain”.

17. **First real testing of the designed ECU (failed):** In this test, the injectors were operating normally and the ECU was reading the expected data from the crank sensor (operated by the starter) but the ignition never happened thus, the engine didn't start.

18. **Inspection process was carried out to highlight the problem:** The inspection revealed that the ignition coils were burned as a result of the test. Further inspection revealed the reason which was the overcharging of the coils, which lead to the increase in the coils temperature thus burning them.

19. **The main ECU code was modified to properly control the ignition coils (succeeded):** The modification introduced was to ensure passing the current through the ignition coils for 3ms only and giving the low signal at the exact angle of ignition. More details on this will be discussed in the "Implementation in the Software domain" section.

4.4. Initial preparations

In order to design a generic ECU for automobiles, it was required to gather information about the automobile sensors and actuators initially. It was also a necessity to view the method followed by the ECU manufacturers to design ECUs based on the standard architecture "AUTOSAR", which ensures high levels of safety, reliability and optimum performance versus fuel consumption.

After researching the topic frequently, it was decided to design an initial ECU version that is based on data acquired from a reference operational ECU and a calibration method that is based on trial and error. This led to a design that is highly flexible, forming a platform for amateurs and researchers to implement their

experiments and observe certain phenomena of their interest freely without the restrictions that otherwise would form a great barrier if a normal ECU was used.

Before actual work was carried out, initial preparations in both the theoretical and software domains were done. These preparations are listed in the following:

1. **Researches in the internal combustion engines' operation field were conducted**: These researches were conducted in order to specify the values of the crucial control parameters and variables versus the different conditions that an automobile exhibit.
2. **Researches in the automobile sensors were also conducted:** These researches helped in the determination of the most important sensors that are effective in the operation of the engine and that without their feedback; it would lead to a faulty run of the engine. These researches also gave a hint out of how each sensor's output signal should be at certain running conditions for the sake of the detection for a fault in the sensor so that it can be replaced.
3. **preparations in the software field were done extensively**: As the ECU's main processing unit is an ATmega128 module; which is an AVR microcontroller, it was required to develop the main code using the AVR coding approach that will be burned on the chip. These initial preparations were mainly about creating the required libraries and drivers for the processor’s peripherals of concern. These peripherals provide the MCU pins with the ability to receive and send different signal types and provide the processor with the means to analyze, process the data received and make the right decisions (based on the main code created).

4.5. Implementations in the Hardware domain

4.5.1. In data acquisition field

The implementations in this field aim for facilitating data fetching from the different automobile sensors of concern and providing the necessary signal conditioning for the sensors' signals received. These implementations are listed in the following:

1. **The reverse engineering of the engine cable:** This cable connects each senor/actuator terminal with its corresponding pin in the ECU; for data fetching or control. It was performed by tracking each sensor/actuator pin using a digital multimeter to find its corresponding pin connected to the ECU. Fig.36 shows the wiring diagram of the engine cable.

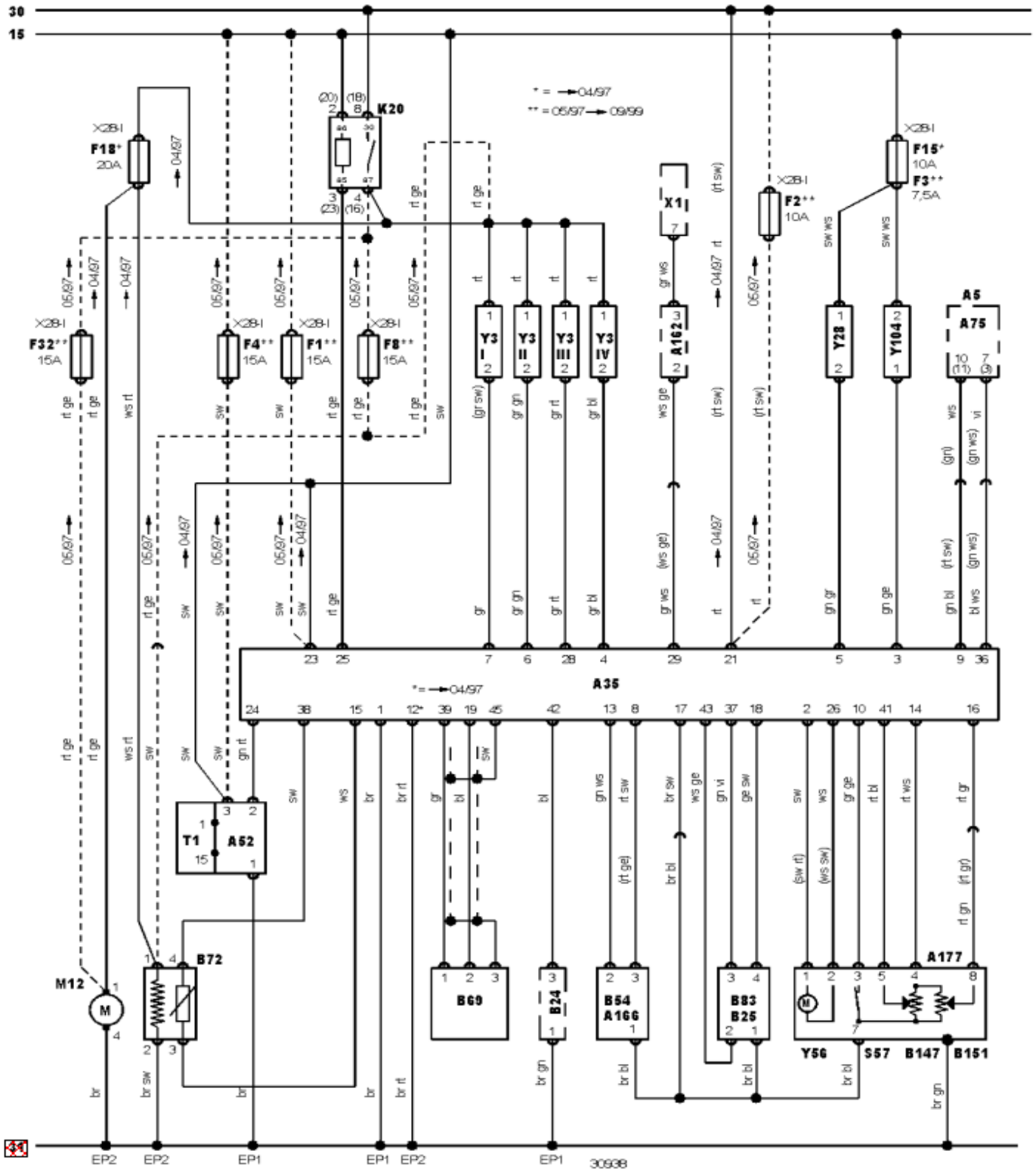

Figure 36: Wiring Diagram of the Engine and ECU connection

2. Initial data acquisition circuit schematic & PCB and configured wires: This PCB was supposed to provide the required signal processing to the automobile sensors' signals of concern at run time of the real engine (controlled by the reference ECU). The data was to be acquired versus different conditions that the automobile exhibits. The input signals to the PCB were Obtained through customized configured wires that provide an additional terminal for each input pin of the reference ECU.

The data acquisition circuit was first designed as to deliver the processed signals to the data acquisition microcontroller for analysis.

3. Crank shaft position sensor's signal conditioning circuit schematic and design: This circuit was designed as to buffer the frequency output signal of the crank position sensor to get a noise less signal, taking into consideration the index pulse generated at the crank sprocket index detection. More on this will be discussed in the calibration field. Fig.37 show the crank shaft position sensor's signal conditioning circuit schematic.

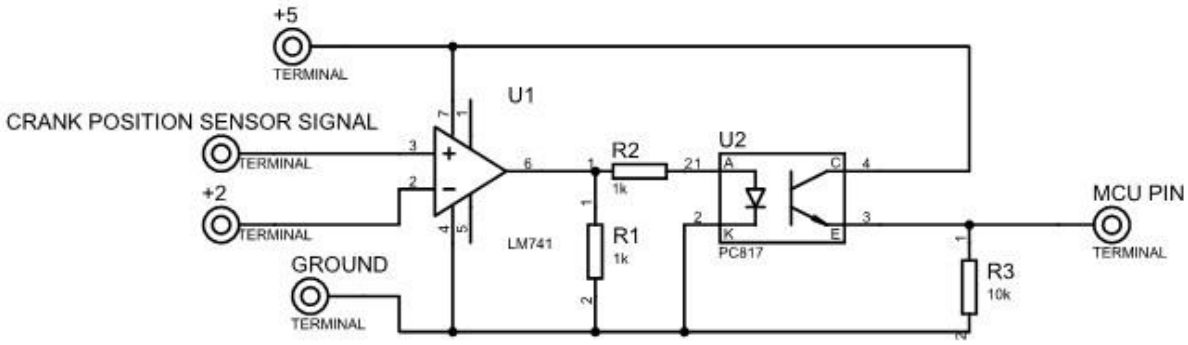

Figure 37: crank shaft position sensor's signal conditioning circuit schematic

4. Ignition coils' and injectors' data acquisition circuit schematic and design:

This circuit was used for acquiring the timing and pulse width of both the ignition coils and the injectors separately. This was done by reversing the configuration of certain terminals of the circuit in One case relative to the Other, as will be shown. Fig.38 show the Ignition coils' and injectors' data acquisition circuit schematic.

- Injectors' mode: The configuration was to connect the IN/OUT 1 terminal to a

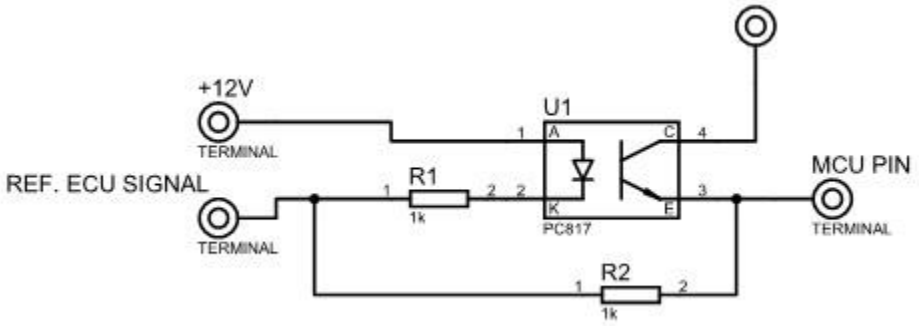

Figure 38: Ignition coils' and injectors' data acquisition circuit schematic

  12V source and the IN/OUT 2 terminal to the reference ECU pins specialized for controlling injectors operation. As a result, when the reference sends out a negative signal; current flows through the optocoupler's LED, lighting it and sending a buffered signal to data acquisition microcontroller. This data represents the start time of the injectors as well as the pulse width.

- Ignition coils' mode: The configuration for the ignition coils' mode was connecting the IN/OUT 1 terminal to a negative 12V source and the reference ECU to the IN/OUT 2 terminal. Such that, when the ECU pin sends out a high signal, current flows through the Optocoupler's LED, lighting it and sending out a buffered signal to the data acquisition microcontroller. This data represents the start time of the coils charging process and duration and when the signal goes low, the firing occurs.

4.5.2. In calibration field

The implementations in this field aim for finding the controlling parameters and variables that would lead to the precise control of the automobile actuators and reading the correct data from the sensors of concern. In achieving so, different techniques and methodologies were followed and many calibration circuits were designed as will be seen in the following:

1. Analyzing the crank shaft position sensor's output signal for index detection: As the output signal of the crank shaft position sensor is in the form of frequency that vary against the crank shaft speed. It was extremely crucial to precisely detect the index pulse just as it passes through the sensor's detection range. For that, an oscilloscope was used to observe the output signal of the sensor OSCILLOSCOPE V-212. Fig.39: Crank position sensor's normal pulses represented on the oscilloscope.

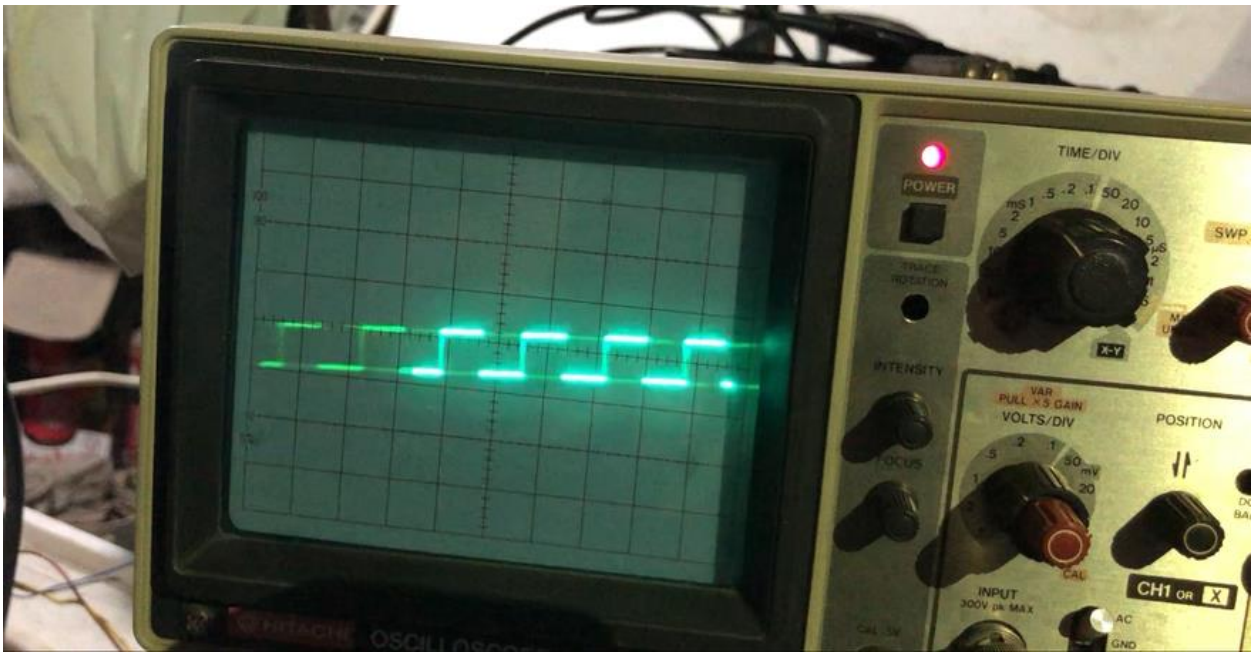

Figure 39: Crank position sensor's normal pulses represented on the oscilloscope

2. Injector's calibration circuit schematic, design & set-up: Based on the data obtained from data acquisition circuit, the calibration circuit was designed as to drive the fuel injectors by operating pulse width values sent by the driving microcontroller. The calibration process was to observe how much fuel is injected into scaled syringe tubes in certain amount of time for each pulse width value. The operating pulse width values were: 5ms, 10ms & 20ms. Fig.40 show the circuit schematic for the injectors' calibration.

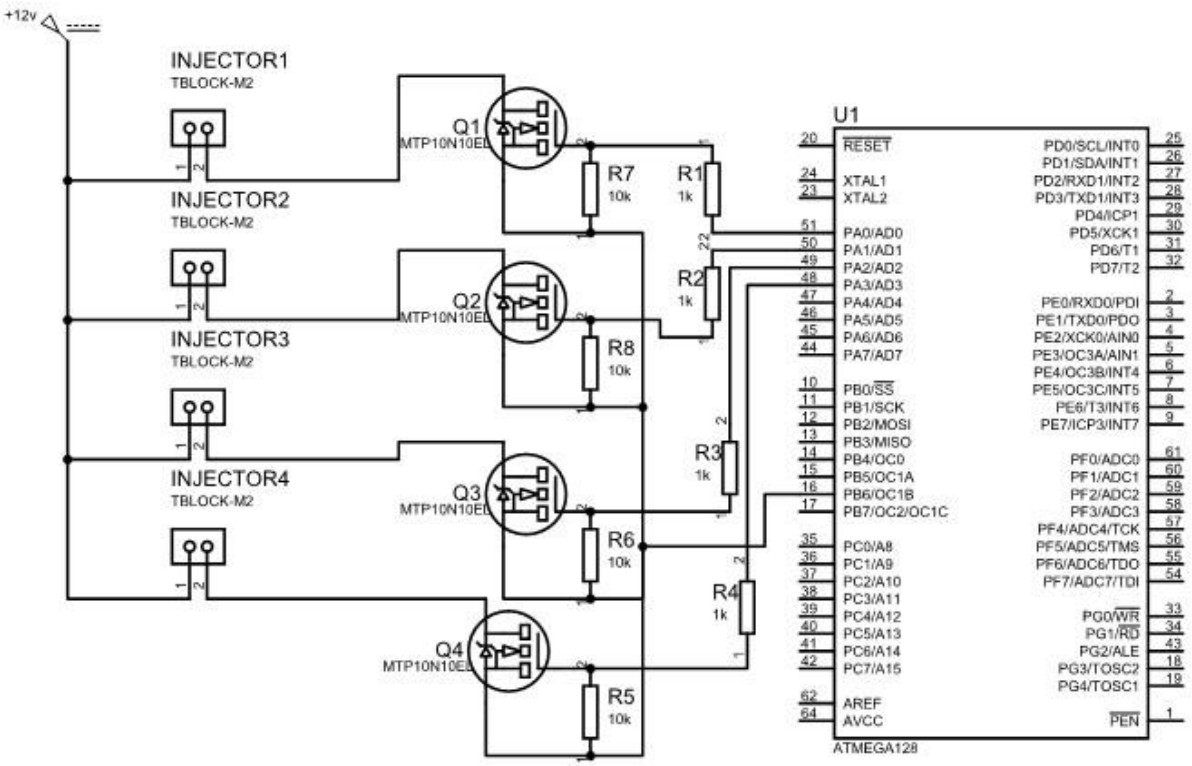

Figure 40: circuit schematic for the injectors' calibration

This circuit with the provided set-up, showed an almost linear behavior. The mis-linearity might be due to a bad sealing method of the syringes with the injectors.

3. Ignition coils' and Injectors' calibration circuit schematic & design: This circuit integrates both the driving circuits of the ignition coils an and the injectors' Fig.41 show the ignition coils and injectors calibration circuit schematic.

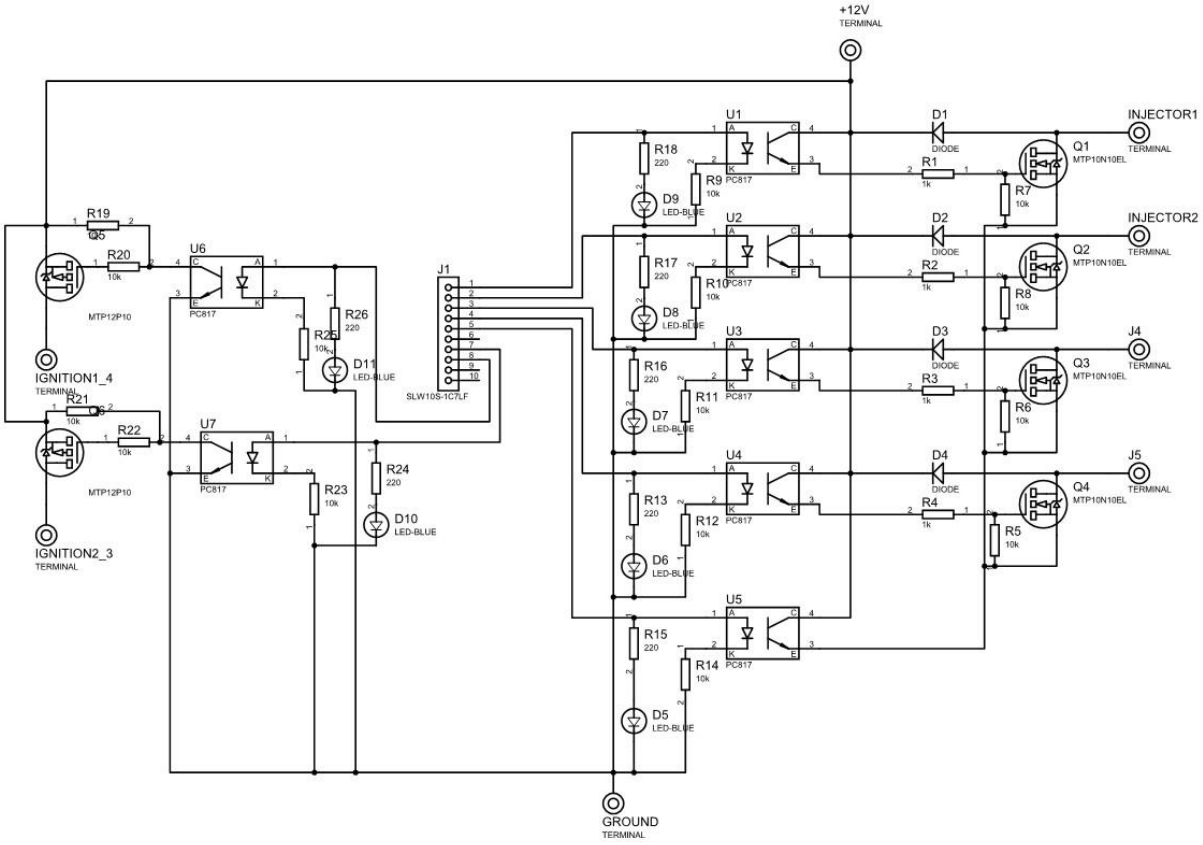

Figure 41: Ignition coils and injectors calibration circuit schematic

As can be seen in the previous schematic, injectors 1-4 have the same start timing, operating duration and operating pulse width. This is generalized for each of the following: injectors 2-3, Ignition coils 1-4 and ignition coils 2-3. This is similar to the technique followed by the reference ECU; which leads to simplifying the control of the ignition coils and injectors operation.

4. Our results

	Signal Angle	*Operating Time*	*Revolution*
INJECTION 1	4°	15-20ms	2
INJECTION 2	34°	15-20ms	1
INJECTION 3	34°	15-20ms	2
INJECTION 4	4°	15-20ms	1
IGNITION 1	2°	3-5ms	1
IGNITION 2	32°	3-5ms	2
IGNITION 3	32°	3-5ms	1
IGNITION 4	2°	3-5ms	2

4.5.3 In the designed ECU field

Implementations in this field are the operational hardware components for the designed ECU which works on conditioning the input/output signals to/from the ECU pins. These implementations also include a minor interfacing board used to facilitate conducting tests, as will be seen in the following:

1. **Engine Start/Stop interfacing control board:** This simple ON/OFF control board was designed and implemented to facilitate conducting tests on the real engine and to reduce the risk of the development of flames due to the electric spark that is generated when the test is conducted manually (by latching the battery's terminals with the system) and the presence of fuel in the place where the test is conducted.
2. **Main ECU circuit schematic & layout:** The designed ECU circuit integrates all the actuators' driving circuits and the sensors' signal conditioning circuits into one major circuit that provides all the necessary features for the microcontroller to carry out its calculations and decision making precisely. Fig.42 shows the ECU circuit schematic. As the circuit's schematic is quite big and contain a lot of components as well as connecting tracks.

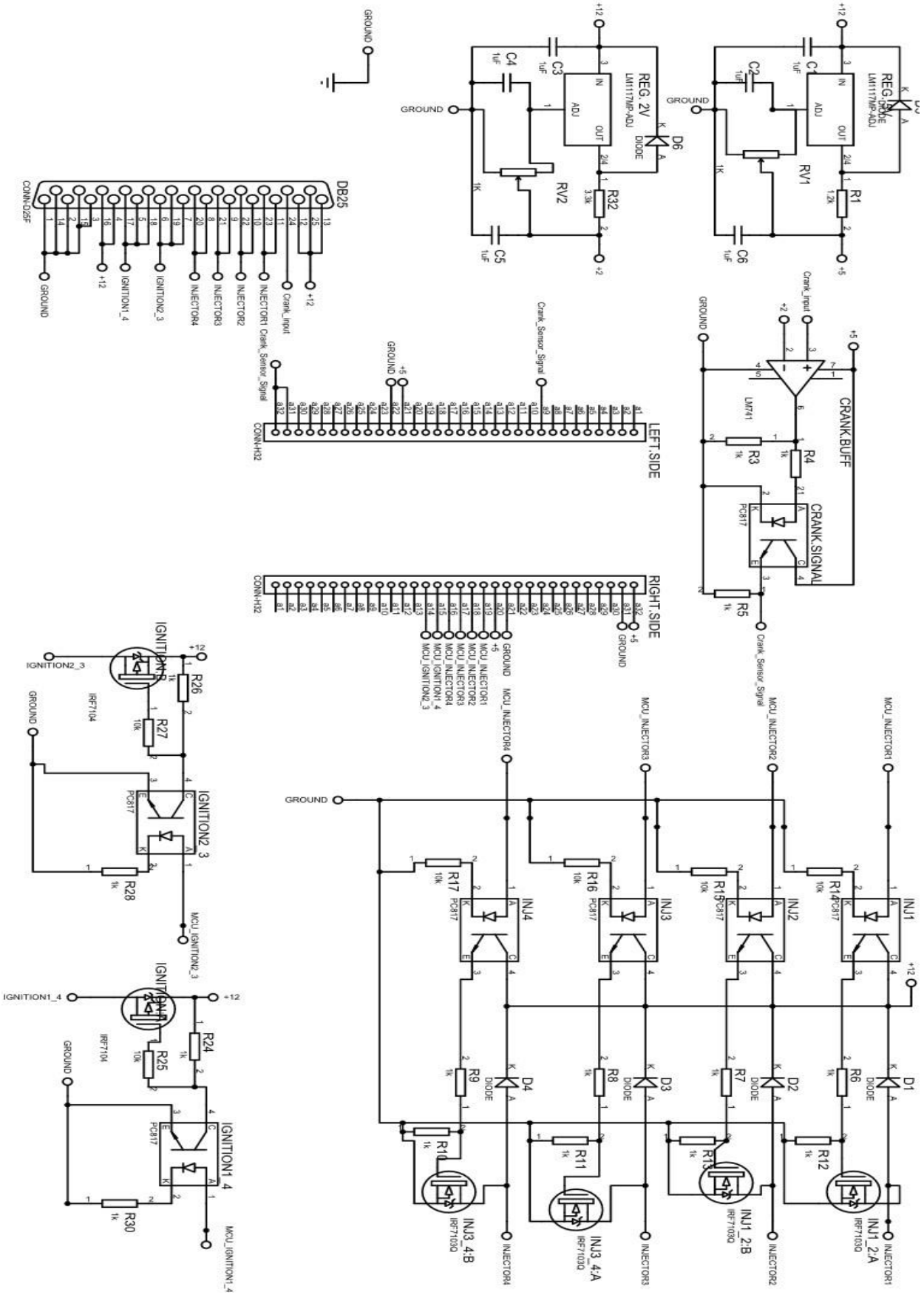

Figure 42: The ECM Circuit Schematic

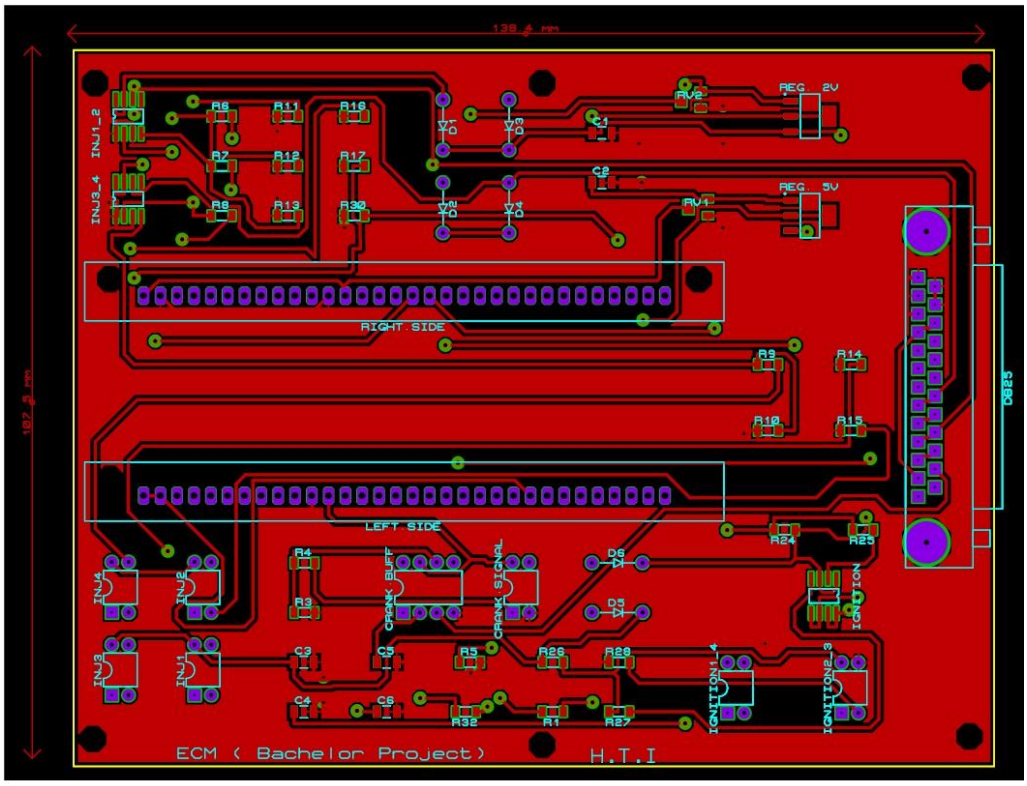

Figure 44: PCB Top-Layer

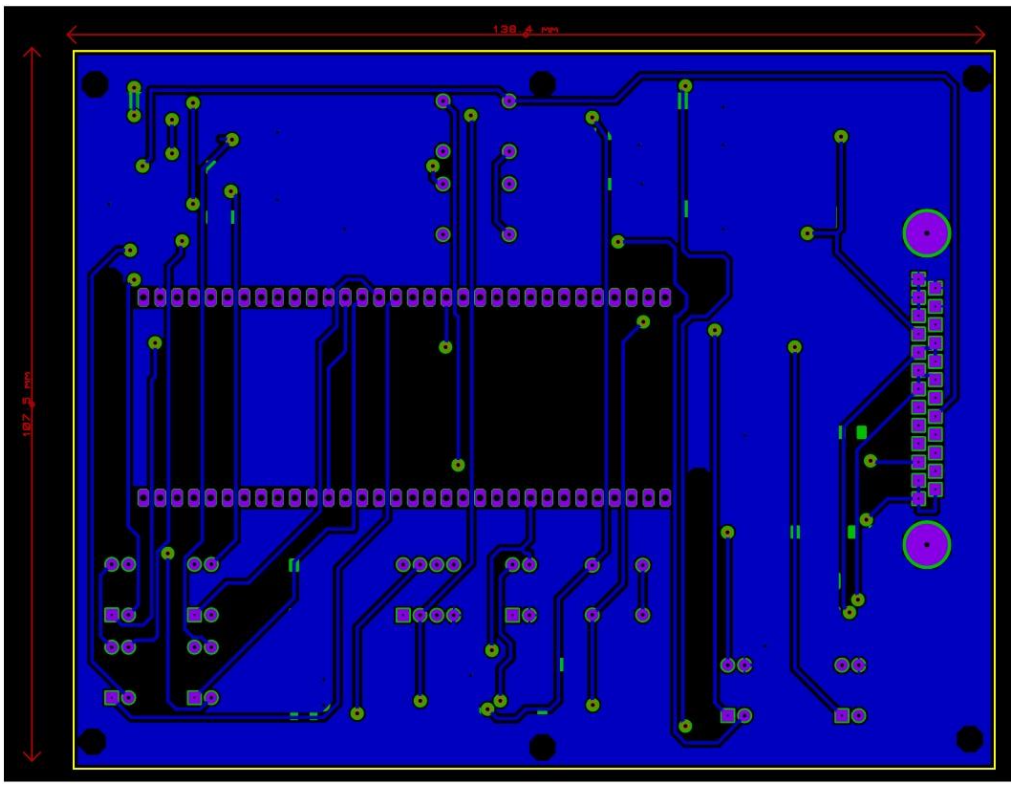

Figure 43:PCB Bottom-Layer

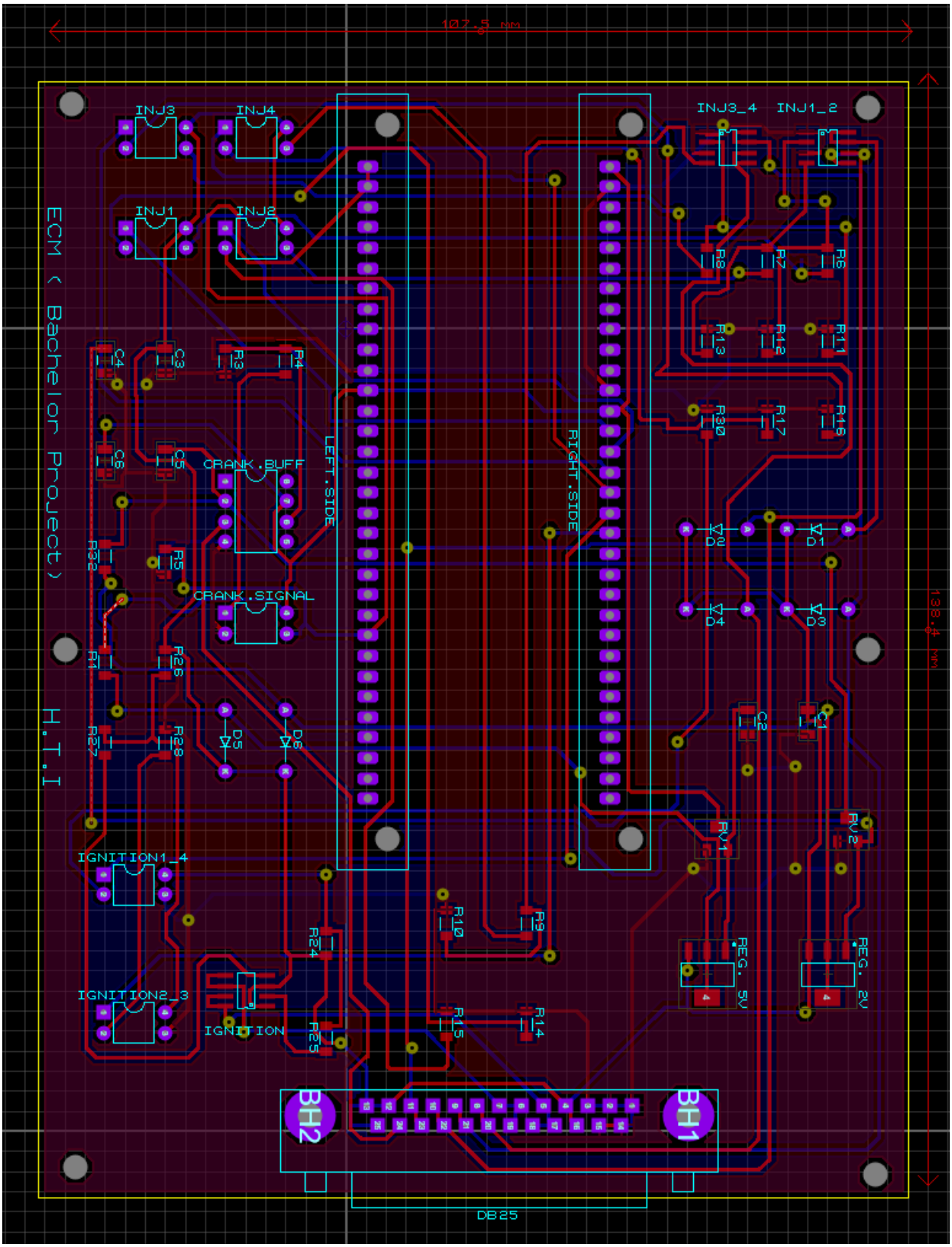

Figure 45: Final PCB Layout

3. Detailed Cost

ITEM	COST
ENGINE	6000
ELECTRONIC COMPONENTS	1155
FUEL TANK	255
CONTROL BOARD	155
FINAL PCB	170
BATTERY	650
WIRING	100
TRANSPORTATION	200
TOTAL	**8685**

Chapter 5

Used Software & Designed Code

Chapter description: In this chapter the overall software used in the project will be presented. Then, the designed code and developed drivers for the ECU signal acquisition and final result of microcontroller operating code will be discussed in deep details.

5.1 Software

5.1.1 Autodata v3.45

An application that allows to understand the whole working process of vehicles. As it gives the ability to view all the important information about the vehicles effortlessly.

A tool that helps to understand the complicated sections of cars or other vehicles and check the condition of every part (i.e., airbags, AC, fuel system, and combustion…etc.), the interface of this application performs all functions easily as shown in the following figures. As it provides an aerial view of every part of the car with descriptions and diagrams.

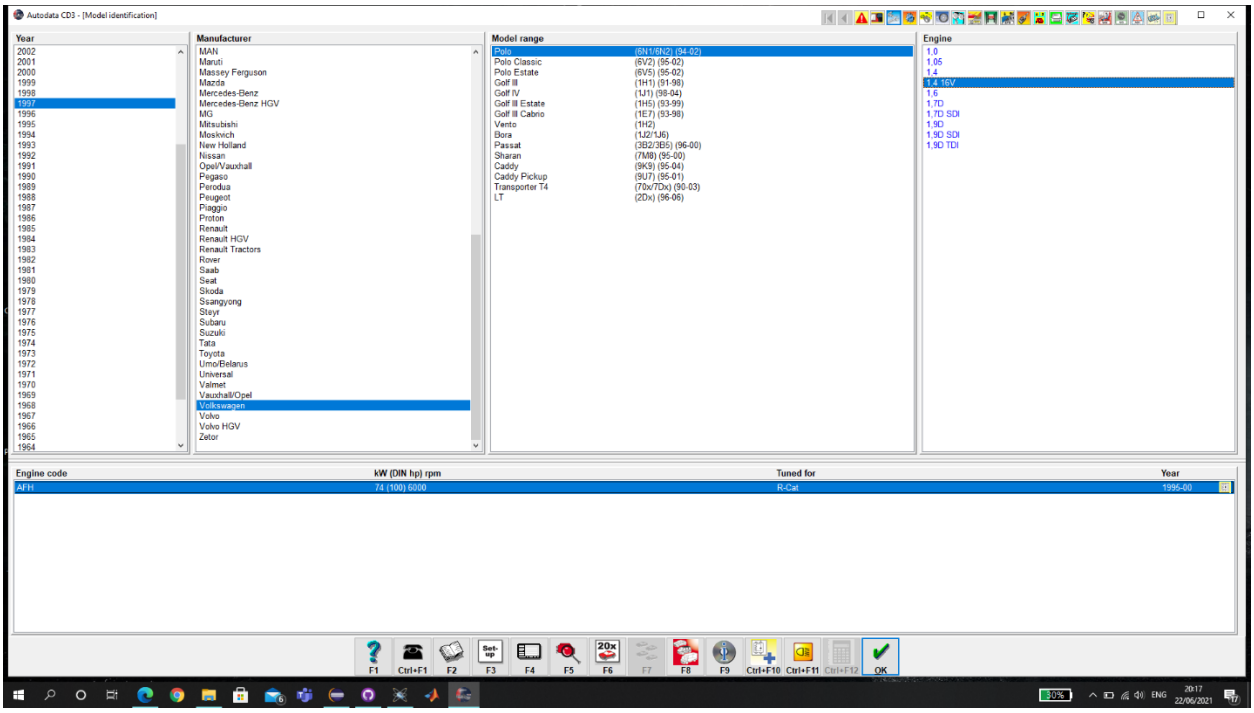

Figure 46: AutoData Interface A

Figure 47: AutoData Interface B

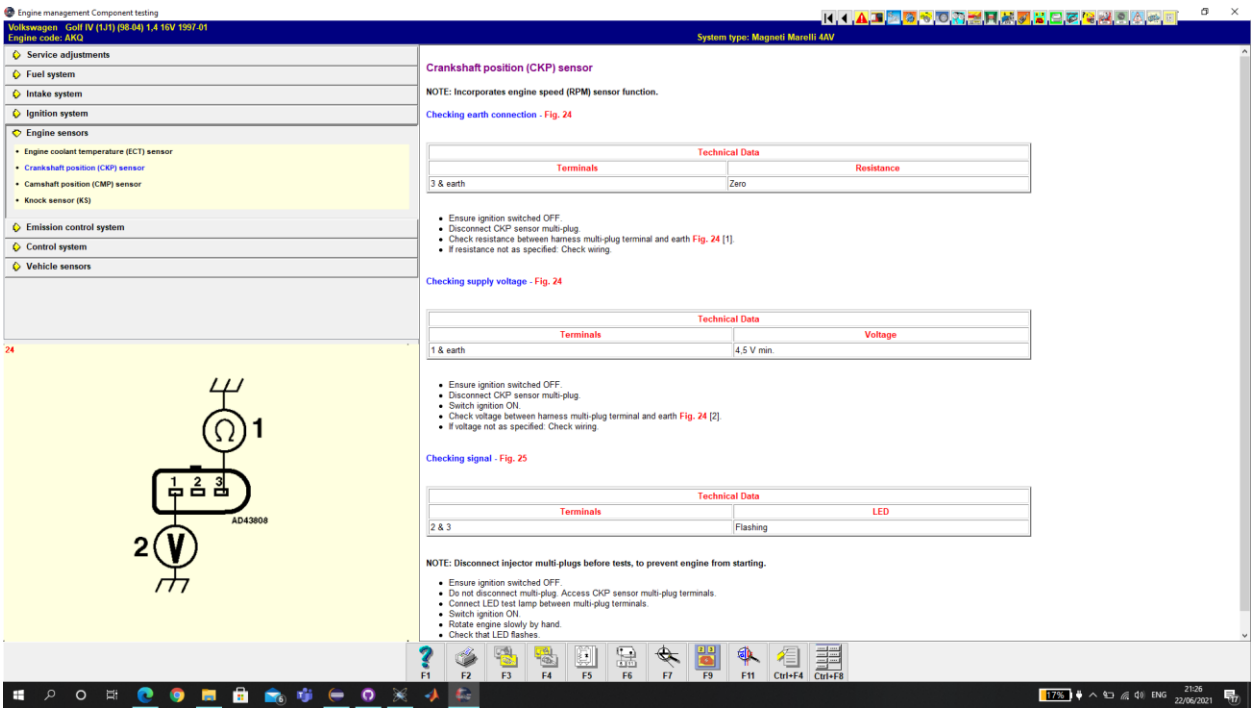

Figure 48: AutoData Interface C

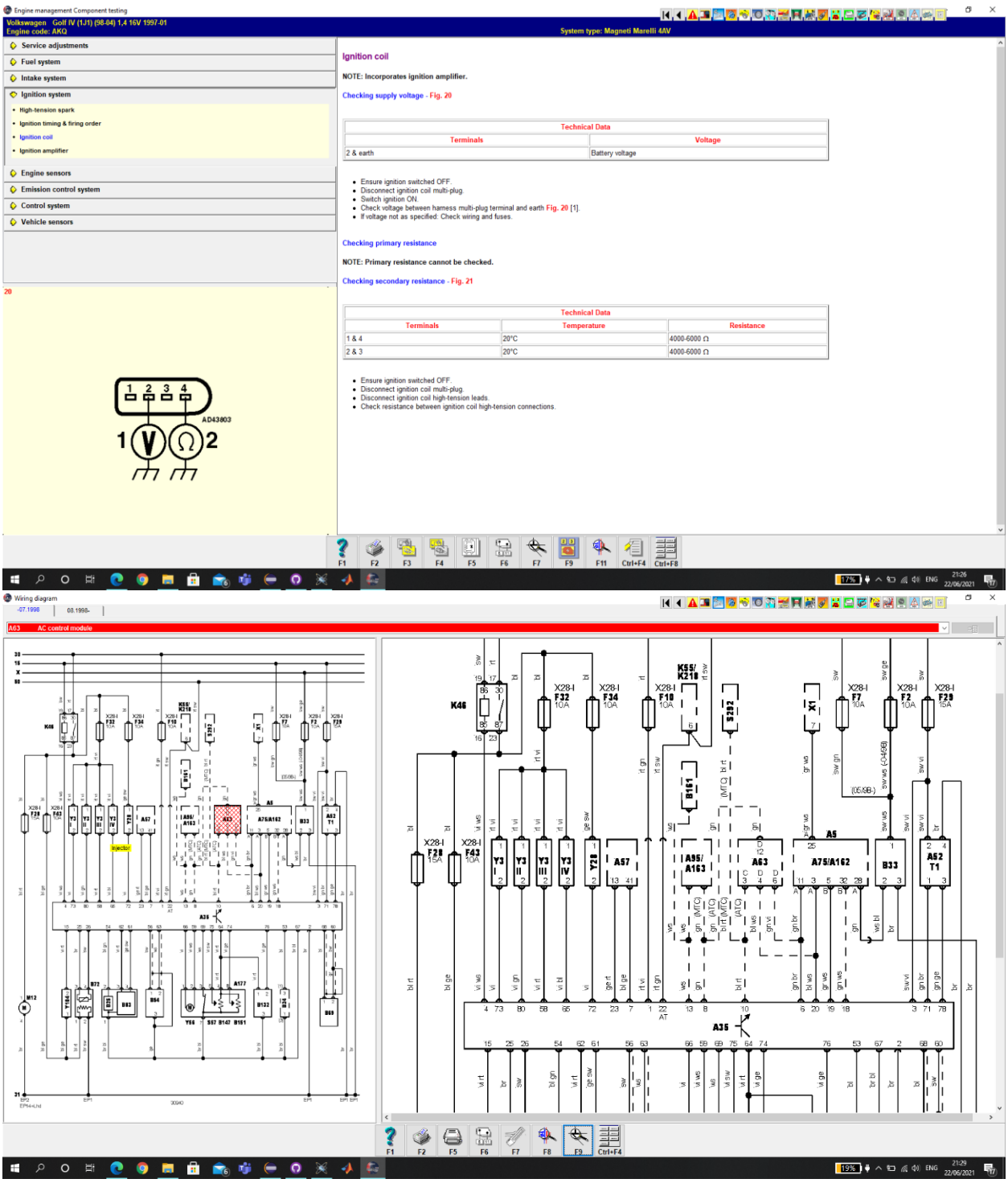

Figure 49: AutoData Interface D

5.1.2 Eclipse v 03-2021

Eclipse is an integrated development environment (IDE) for developing applications using the Java programming language and other programming languages such as C/C++, Python, PERL, Ruby etc.

The Eclipse platform which provides the foundation for the Eclipse IDE is composed of plug-ins and is designed to be extensible using additional plug-ins (i.e., AVR plug-in). C/C++ Development Tools (CDT) is a plug-in that allows Eclipse to be used for developing application using C/C++, as it as well allows debugging and fetching the codes used.

5.1.3 Proteus v8.9

A simulation tool, that design and configure the electronic devices. Which are based on the various microcontroller of different families. The software allows to introduce the circuit in the graphic editor, model its operation and develop the printed circuit board, including three-dimensional visualization. Proteus provides the support of SPICE-models, which are often given by the manufacturers of electronic components. The software is also compatible with a huge number of digital and analog device models. Proteus allows to perform the testing for possible mistakes at the end of work over the board.

5.1.4 AVRdude

It is a program for downloading and uploading the on-chip memories of Atmel's AVR microcontrollers. It can program the Flash and EEPROM, which is supported by the serial programming protocol, it can program fuse and lock bits.

AVRDUDE also supplies a direct instruction mode allowing one to issue any programming instruction to the AVR chip regardless of whether AVRDUDE implements that specific feature of a particular chip.

5.1.5 Github

GitHub is a web-based version-control and collaboration platform for software developers. Which is used to store the source code for a project and track the complete history of all changes to that code. It allows developers to collaborate on a project more effectively by providing tools for managing possibly conflicting changes from multiple developers. GitHub allows developers to change, adapt and improve software from its repositories.

5.1.6 Serial Port Monitor

An application which monitors and records any data going through serial ports of a computer. This application comes in handy for development, testing and debugging of COM-based programs and devices. However, it uses Serial Protocol Analyzer that makes it simple to read, filter and save data generated by applications and hardware connected to any system's COM interfaces.

5.2 Drivers & Designed Code

5.2.1 Our Drivers

5.2.1.1 Timers

```
/*-------------------------------------------------------------------------
 - File Name: TIMERS.c
 -
 - Module: Timer0,Timer2 (8-bit) ------ Timer1,Timer3 (16-bit)
 -
 - Description: Source file for the Timers driver for AtMega128
 -
 -------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------
-
                            NOTES About Timer Driver
 -------------------------------------------------------------------------
-*/
/*
 - Notes:       - Timer0/Timer2 Maximum time in 1 overflow is 256ms
 -                    as 1 clock = 1ms
 -                    ( Maximum => Using small clock = 1Mhz , and large
prescaler = 1024 )
 -
 -                  - TIMER1/Timer3 Maximum time in 1 overflow is 65535ms (
65.5 Sec )
 -                    as 1 clock = 1ms
 -                    ( Maximum => Using small clock = 1Mhz , and large
prescaler = 1024 )
 -
 - Pins:        - PB3/OC0  -> Square Wave and PWM Mode in Timer0
 -                  - PD7/OC2  -> Square Wave and PWM Mode in Timer2
 -                  - PD5/OC1A -> Square Wave and PWM Mode in TIMER1A
 -                  - PD4/OC1B -> Square Wave  Mode in TIMER1B
 -
 - Clock And Prescaler :
 -                  Time = Prescaler / F_CPU * Ticks
 -                  For clock=1Mhz and prescale F_CPU/1024 every count will
take 1ms
 -                  Timer Clock => 1,000,000 / 1024 = 1Khz
 -                  clock timer period = 1 / 1Khz = 1ms
 -                  so put initial timer counter = 0  0 --> 255 (256ms per
overflow)
```

```
 -                   so we need timer to overflow 4 times to get a 1 second
 -
 - Example for Config Struct:
 -                   // F_CPU = 8Mhz       TIMER1 CTC Mode  1 Sec.
 -                   TIMER_ConfigType TIMER1_Config =
 -                        {.clock = F_CPU_1024, .mode = CTC, .OCRValue = 8000
};
 -                   TIMER1_Init(&TIMER1_Config);
 ------------------------------------------------------------------------------
-*/

#include "TIMERS.h"

/*-------------------------------------------------------------------------
                            Global Variables
 -------------------------------------------------------------------------*/

static void (*g_TIMER0_callBackPtr)(void) = NULL_PTR;
static void (*g_TIMER2_callBackPtr)(void) = NULL_PTR;
static void (*g_TIMER1_callBackPtr)(void) = NULL_PTR;
static void (*g_TIMER3_callBackPtr)(void) = NULL_PTR;
static uint8 g_T0clock, g_T1clock, g_T2clock , g_T3clock;

/*******************************************************************************
*
 *                       Interrupt Service Routines
*
*******************************************************************************
/

/*
 *  Description : Interrupt Service Routine for TIMER0 CTC Mode
 */
ISR(TIMER0_COMP_vect)
{
    if(g_TIMER0_callBackPtr != NULL_PTR)
    {
         /* Call The Call Back function in the application after the timer
value = OCR0 Value*/
         (*g_TIMER0_callBackPtr)() ;
    }
}

/*
```

```
 *  Description : Interrupt Service Routine for TIMER0 OverFlow(Normal) Mode
 */
ISR(TIMER0_OVF_vect)
{
    if(g_TIMER0_callBackPtr != NULL_PTR)
    {
        /* Call The Call Back function in the application after the timer
value = 1023 */
        (*g_TIMER0_callBackPtr)() ;
    }
}

/*
 *  Description : Interrupt Service Routine for TIMER2 CTC Mode
 */
ISR(TIMER2_COMP_vect)
{
    if(g_TIMER2_callBackPtr != NULL_PTR)
    {
        /* Call The Call Back function in the application after the timer
value = OCR0 Value*/
        (*g_TIMER2_callBackPtr)() ;
    }
}

/*
 *  Description : Interrupt Service Routine for TIMER2 OverFlow(Normal) Mode
 */
ISR(TIMER2_OVF_vect)
{
    if(g_TIMER2_callBackPtr != NULL_PTR)
    {
        /* Call The Call Back function in the application after the timer
value = 1023 */
        (*g_TIMER2_callBackPtr)() ;
    }
}

/*
 *  Description : Interrupt Service Routine for TIMER1 CTC Mode
 */
ISR(TIMER1_COMPA_vect)
{
    if(g_TIMER1_callBackPtr != NULL_PTR)
    {
```

```
          /* Call The Call Back function in the application after the timer
value = OCR1A Value*/
          (*g_TIMER1_callBackPtr)() ;
     }
}
/*
 *  Description : Interrupt Service Routine for TIMER1 CTC Mode
 */
ISR(TIMER1_COMPB_vect)
{
     if(g_TIMER1_callBackPtr != NULL_PTR)
     {
          /* Call The Call Back function in the application after the timer
value = OCR1B Value*/
          (*g_TIMER1_callBackPtr)() ;
     }
}
/*
 *  Description : Interrupt Service Routine for TIMER1 CTC Mode
 */
ISR(TIMER1_COMPC_vect)
{
     if(g_TIMER1_callBackPtr != NULL_PTR)
     {
          /* Call The Call Back function in the application after the timer
value = OCR1C Value*/
          (*g_TIMER1_callBackPtr)() ;
     }
}


/*
 *  Description : Interrupt Service Routine for TIMER1 OverFlow(Normal) Mode
 */
ISR(TIMER1_OVF_vect)
{
     if(g_TIMER1_callBackPtr != NULL_PTR)
     {
          /* Call The Call Back function in the application after the timer
value = 65,535 */
          (*g_TIMER1_callBackPtr)() ;
     }
}

ISR(TIMER3_COMPA_vect)
```

```
{
    if(g_TIMER3_callBackPtr != NULL_PTR)
    {
          /* Call The Call Back function in the application after the timer
value = OCR3A Value*/
          (*g_TIMER3_callBackPtr)() ;
    }
}
ISR(TIMER3_COMPB_vect)
{
    if(g_TIMER3_callBackPtr != NULL_PTR)
    {
          /* Call The Call Back function in the application after the timer
value = OCR3B Value*/
          (*g_TIMER3_callBackPtr)() ;
    }
}
ISR(TIMER3_COMPC_vect)
{
    if(g_TIMER3_callBackPtr != NULL_PTR)
    {
          /* Call The Call Back function in the application after the timer
value = OCR3C Value*/
          (*g_TIMER3_callBackPtr)() ;
    }
}

/*
 *  Description : Interrupt Service Routine for TIMER3 OverFlow(Normal) Mode
 */
ISR(TIMER3_OVF_vect)
{
    if(g_TIMER3_callBackPtr != NULL_PTR)
    {
          /* Call The Call Back function in the application after the timer
value = 65,535 */
          (*g_TIMER3_callBackPtr)() ;
    }
}

/*-------------------------------------------------------------------------
                            TIMER0 Functions
 -------------------------------------------------------------------------*/
        /*------------------- Description --------------------
         -  Function to initialize TIMER0
```

```
          -  1. Set Timer initial value to 0
          -  2. Configure Control Register TCCR0 depending on
                desired configurations
          -  3. Enable Interrupts depending on selected mode
          ----------------------------------------------------*/
void TIMER0_init (TIMER_ConfigType *config_ptr)
{
    g_T0clock = (config_ptr ->clock);

    /* Set Timer initial value to 0 */
    TCNT0 = 0;
/*--------------------- TCCR0 ----------------------
-  1. FOC0 = 1 in non-PWM modes so mask it with selected mode
-  2. WGM01:0(bit3,6) Waveform generation mode
-  3. COM01:0 Compare match output mode (OC)
-  3. CS02:0 Clock Select
----------------------------------------------------*/
    TCCR0 = (((config_ptr ->mode)<<FOC0)& 0x80) | (((config_ptr -
>mode)<<WGM00)& 0x40)
              | (((config_ptr ->mode)<<(WGM01-1)) & 0x08)
              | ((config_ptr ->OC)<<COM00) | ((config_ptr ->clock)<<CS00);

    /* Enable Timer overflow Interrupt if NORMAL mode selected */
    if (config_ptr ->mode == NORMAL)
    {
         SET_BIT (TIMSK,TOIE0);                  /* Timer overflow Interrupt
Enable */
    }

    /* Enable Compare match Interrupt if Compare mode selected */
    else if (config_ptr ->mode == COMP)
    {
         SET_BIT (TIMSK,OCIE0);                  /* Compare match Interrupt
Enable */
         OCR0 = (uint8)(config_ptr ->OCRValue); /* Compare Value */
    }

    /* Square wave generator (Toggle) */
    else if ((config_ptr ->mode == COMP) & (config_ptr ->OC == TOGGLE))
    {
         SET_BIT (TIMSK,OCIE0);                  /* Compare match Interrupt
Enable */
         SET_BIT (DDRB,PB4);                     /* Configure PB4/OC0 Pin as
output pin */
         OCR0 = (uint8)(config_ptr ->OCRValue); /* Compare Value */
```

```
    }

    /* PWM mode specials */
    else if (config_ptr ->mode == PWM)
    {
         SET_BIT (DDRB,PB4);                    /* Configure PB4/OC0 Pin as
output pin */
         OCR0 = (uint8)(config_ptr ->OCRValue); /* Duty Cycle Value */
    }
}

        /*------------------- Description --------------------
                 Function to reset timer to 0 again
         ----------------------------------------------------*/
void TIMER0_resetTimer(void)
{
    TCNT0 = 0;
}

        /*------------------- Description --------------------
             Function to stop timer (Set Clock to NO_CLOCK)
         ----------------------------------------------------*/
void TIMER0_stopTimer(void)
{
    /* Clear Clock Bits */
    TCCR0 &= 0xF8;
}

        /*------------------- Description --------------------
         Function to start timer (Set Clock to g_TIMER0_Clock)
         ----------------------------------------------------*/
void TIMER0_restartTimer(void)
{
    /* Clear Clock Bits */
    TCCR0 &= 0xF8 ;

    /* Set Clock Bits */
    TCCR0 |= (g_T0clock << CS00 ) ;
}

        /*------------------- Description --------------------
             Function to change compare value/duty cycle
         ----------------------------------------------------*/
void TIMER0_ticks(const uint8 Ticks)
{
```

```
    OCR0 = Ticks;
}

        /*------------------- Description --------------------
            Function to call back function to be use in ISR
         ----------------------------------------------------*/
void TIMER0_callBack (void (*ptr)(void))
{
    /* Save the address of the function to be Called back  in a global variable
*/
    g_TIMER0_callBackPtr = ptr;
}

        /*------------------- Description --------------------
            Function to call back function to be use in ISR
         ----------------------------------------------------*/
void TIMER0_deinit (void)
{
    TCCR0 = 0x00;
}
/*------------------------------------------------------------------------
                            TIMER2 Functions
 ------------------------------------------------------------------------*/
        /*------------------- Description --------------------
         -  Function to initialize TIMER2
         -  1. Set Timer initial value to 0
         -  2. Configure Control Register TCCR2 depending on
               desired configurations
         -  3. Enable Interrupts depending on selected mode
         ----------------------------------------------------*/
void TIMER2_init (TIMER_ConfigType *config_ptr)
{
    g_T2clock = (config_ptr ->clock);
    /* Set Timer initial value to 0 */
    TCNT2 = 0;
/*--------------------- TCCR2 ----------------------
-  1. FOC2 = 1 in non-PWM modes so mask it with selected mode
-  2. WGM21:0(bit3,6) Waveform generation mode
-  3. COM21:0 Compare match output mode (OC)
-  3. CS22:0 Clock Select
----------------------------------------------------*/
    TCCR2 = (((config_ptr ->mode)<<FOC2)& 0x80) | (((config_ptr -
>mode)<<WGM20)& 0x40)
              | (((config_ptr ->mode)<<(WGM21-1)) & 0x08)
              | ((config_ptr ->OC)<<COM20) | ((config_ptr ->clock)<<CS20);
```

```
    /* Enable Timer overflow Interrupt if NORMAL mode selected */
    if (config_ptr ->mode == NORMAL)
    {
         SET_BIT (TIMSK,TOIE2);                  /* Timer overflow Interrupt
Enable */
    }

    /* Enable Compare match Interrupt if Compare mode selected */
    else if (config_ptr ->mode == COMP)
    {
         SET_BIT (TIMSK,OCIE2);
         SET_BIT (DDRB,PB7);                    /* Compare match Interrupt
Enable */
         OCR2 = (uint8)(config_ptr ->OCRValue); /* Compare Value */
    }

    /* PWM mode specials */
    else if (config_ptr ->mode == PWM)
    {
         SET_BIT (DDRB,PB7);                     /* Configure PB7/OC2 Pin as
output pin */
         OCR2 = (uint8)(config_ptr ->OCRValue); /* Duty Cycle Value */
    }
}

        /*------------------- Description --------------------
                Function to reset timer to 0 again
         ----------------------------------------------------*/
void TIMER2_resetTimer(void)
{
    TCNT2 = 0;
}

        /*------------------- Description --------------------
             Function to stop timer (Set Clock to NO_CLOCK)
         ----------------------------------------------------*/
void TIMER2_stopTimer(void)
{
    /* Clear Clock Bits */
    TCCR2 &= 0xF8;
}

        /*------------------- Description --------------------
         Function to start timer (Set Clock to g_TIMER2_Clock)
```

```
          ------------------------------------------------------*/
void TIMER2_restartTimer(void)
{
    /* Clear Clock Bits */
    TCCR2 &= 0xF8 ;

    /* Set Clock Bits */
    TCCR2 |= (g_T2clock << CS00 ) ;
}

        /*------------------- Description --------------------
             Function to change compare value/duty cycle
         ------------------------------------------------------*/
void TIMER2_ticks(const uint8 Ticks)
{
    OCR2 = Ticks;
}

        /*------------------- Description --------------------
            Function to call back function to be use in ISR
         ------------------------------------------------------*/
void TIMER2_callBack (void (*ptr)(void))
{
    /* Save the address of the function to be Called back  in a global variable
*/
    g_TIMER2_callBackPtr = ptr;
}

        /*------------------- Description --------------------
            Function to call back function to be use in ISR
         ------------------------------------------------------*/
void TIMER2_deinit (void)
{
    TCCR2 = 0x00;
}

/*-------------------------------------------------------------------------
                            TIMER1 Functions
 -------------------------------------------------------------------------*/
/*
 * Description : Function to initialize the Timer driver
 *   1. Select Timer Mode ( Normal - Compare - Square - PWM )
 *   2. Set the required clock.
 *   3. Set Compare Value if The Timer in Compare Mode
 *   4. Set OC1A/OC1B Pin Mode in Square Mode
```

```
 *   5. Enable the Timer Normal-Compare Interrupt.
 *   6. Initialize TIMER1 Registers
 */
void TIMER1_init(TIMER_ConfigType *Config_ptr)
{
    /* set a global variable for clock to use it in
     * restart timer function */
    g_T1clock = Config_ptr->clock ;

    /* Set TIMER1 In Normal Mode */
    if (Config_ptr->mode == NORMAL)
    {
          /* Set Timer initial value to 0 */
          TCNT1 = 0;

          /*  Enable TIMER1 Overflow Interrupt */
          TIMSK |= (1<<TOIE1);

          /* Configure the timer control register
           * 1. Non PWM mode     => FOC1A=1 & FOC1B=1
           * 2. Normal Mode      => WGM11=0 & WGM10=0 & WGM13=0 & WGM12=0
           * 3. OC1A/B Mode      => COM1A1 & COM1A0 & COM1B1 & COM1B0
(Disconnected)
           * 4. clock            => CS12 & CS11 & CS10
           * TCCR1A => COM1A1 COM1A0 COM1B1 COM1B0 FOC1A FOC1B WGM11 WGM10
           * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
           */

          TCCR1C = (1<<FOC1A) | (1<<FOC1B) | (1<<FOC1C) ;
          TCCR1B = ((Config_ptr->clock) << CS10) ;
    }

    /* Set TIMER1 In Compare Mode with OCR1A Value */
    else if (Config_ptr->mode == COMP)
    {

          /* Channels interrupt Enable */
          TIMSK |= (1 << OCIE1A) | (1 << OCIE1B) | (1 << OCIE1C);

          /* Initial Value of The Timer TCNT*/
          TCNT1 = 0;

          /* Compare Value for Channel A */
          OCR1A = Config_ptr->OCRValue;
          /* Compare Value for Channel B */
```

```
        OCR1B = Config_ptr->OCR1BValue;
        /* Compare Value for Channel C */
        OCR1C = Config_ptr->OCR1CValue;
        /* using ICR1 as Top Value for Duty Cycle for all Channels
         * Duty Cycle for PWM Mode = OCR1x/ICR1 */
        ICR1 = Config_ptr->ICR1Value;

        /* Configure the timer control register
         * 1. Non PWM mode    => FOC1A=1 & FOC1B=1
         * 2. CTC Mode        => WGM11=0 & WGM10=0 & WGM13=0 & WGM12=1
         * 3. OC1A/B Mode     => COM1A1 & COM1A0 & COM1B1 & COM1B0
         * 4. clock           => CS12 & CS11 & CS10
         * TCCR1A => COM1A1 COM1A0 COM1B1 COM1B0 COM1C1 COM1C0 WGM11 WGM10
         * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
         * TCCR1C => FOC1A FOC1B FOC1C
         */

        TCCR1A = ((Config_ptr->OC) << COM1A0) | ((Config_ptr->OC1B) <<
COM1B0)
                  | ((Config_ptr->OC1C) << COM1C0);
        TCCR1B = (1 << WGM12) | ((Config_ptr->clock) << CS10);
        TCCR1C = (1 << FOC1A) | (1 << FOC1B) | (1 << FOC1C);
    }

    /* Set TIMER1 In CTC Square Wave Mode */
    else if (Config_ptr->mode == CTC)
    {
        /* Set pin PB5/OC1A as output pin */
        SET_BIT (DDRB,PB5);
        /* Set pin PB6/OC1B as output pin */
        SET_BIT (DDRB,PB6);
        /* Set pin PB7/OC1A/OC2 as output pin */
        SET_BIT (DDRB,PB7);

        /* Channels interrupt Enable */
        TIMSK |= (1<<OCIE1A) | (1<<OCIE1B) | (1<<OCIE1C);

        /* Initial Value of The Timer TCNT*/
        TCNT1 = 0 ;

        /* Compare Value for Channel A */
        OCR1A = Config_ptr->OCRValue ;
        /* Compare Value for Channel B */
        OCR1B = Config_ptr->OCR1BValue;
        /* Compare Value for Channel C */
```

```
        OCR1C = Config_ptr->OCR1CValue;

        /* Configure the timer control register
         * 1. Non PWM mode    => FOC1A=1 & FOC1B=1
         * 2. CTC Mode        => WGM11=0 & WGM10=0 & WGM13=0 & WGM12=1
         * 3. OC1A/B Mode     => COM1A1 & COM1A0 & COM1B1 & COM1B0
         * 4. clock           => CS12 & CS11 & CS10
         * TCCR1A => COM1A1 COM1A0 COM1B1 COM1B0 COM1C1 COM1C0 WGM11 WGM10
         * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
         * TCCR1C => FOC1A FOC1B FOC1C
         */

        TCCR1A = ((Config_ptr->OC) << COM1A0) | ((Config_ptr->OC1B) <<
COM1B0) | ((Config_ptr->OC1C) << COM1C0);
        TCCR1B = (1<<WGM12) | ((Config_ptr->clock) << CS10) ;
        TCCR1C = (1<<FOC1A) | (1<<FOC1B) | (1<<FOC1C);
    }

    /* Set TIMER1 In PWM Mode with Duty Cycle */
    else if (Config_ptr->mode == PWM)
    {
        /* Set pin PB5/OC1A as output pin */
        SET_BIT (DDRB,PB5);
        /* Set pin PB6/OC1B as output pin */
        SET_BIT (DDRB,PB6);
        /* Set pin PB7/OC1A/OC2 as output pin */
        SET_BIT (DDRB,PB7);

        /* Initial Value of The Timer TCNT*/
        TCNT1 = 0 ;

        TIMSK |= (1<<OCIE1A) | (1<<OCIE1B) | (1<<OCIE1C);

        /* Duty Cycle Value for Channel A */
        OCR1A = Config_ptr->OCRValue;
        /* Duty Cycle Value for Channel B */
        OCR1B = Config_ptr->OCR1BValue;
        /* Duty Cycle Value for Channel C */
        OCR1C = Config_ptr->OCR1CValue;

        /* using ICR1 as Top Value for Duty Cycle for all Channels
         * Duty Cycle for PWM Mode = OCR1x/ICR1 */
        ICR1 = Config_ptr->ICR1Value;

        /* Configure the timer control register
```

```
             * 1. PWM mode         => FOC1A=0 & FOC1B=0
             * 2. PWM Mode         => WGM11=1 & WGM10=0 & WGM13=1 & WGM12=1
             * 3. OC1A/B/C Mode    => COM1A1 & COM1A0 & COM1B1 & COM1B0 & COM1C1 &
COM1C0
             * 4. clock            => CS12 & CS11 & CS10
             * TCCR1A => COM1A1 COM1A0 COM1B1 COM1B0 COM1C1 COM1C0 WGM11 WGM10
             * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
             *
             */
            TCCR1A = ((Config_ptr->OC) << COM1A0) | ((Config_ptr->OC1B) <<
COM1B0)
                      |((Config_ptr->OC1C) << COM1C0) | (1<<WGM11);

            TCCR1B = (1<<WGM13) | (1<<WGM12) | ((Config_ptr->clock) << CS10) ;
    }
}

/*
 * Description: Function to clear the TIMER1 Value to start count from ZERO
 */
void TIMER1_resetTimer(void)
{
    TCNT1 = 0;
}

/*
 * Description: Function to Stop the TIMER1
 * CLEAR CS12 CS11 CS10
 * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
 */
void TIMER1_stopTimer(void)
{
    /* Clear Clock Bits */
    TCCR1B &= 0xF8 ;
}

/*
 * Description: Function to Restart the TIMER1
 * CLEAR CS12 CS11 CS10
 * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
 */
void TIMER1_restartTimer(void)
{
    /* Clear Clock Bits */
    TCCR1B &= 0xF8 ;
```

```
    /* Set Clock Bits */
    TCCR1B |= ( g_T1clock << CS10) ;
}

/*
 * Description: Function to Change Ticks (Compare Value) of Timer
 *                   using also to Change Duty Cycle in PWM Mode
 */
void TIMER1_Ticks(const uint16 Ticks1A, const uint16 Ticks1B)
{
    OCR1A = Ticks1A;
    OCR1B = Ticks1B;
}

/*
 * Description: Function to set the Call Back function address for TIMER2 .
 */
void TIMER1_setCallBack(void(*a_ptr)(void))
{
    /* Save the address of the Call back function in a global variable */
    g_TIMER1_callBackPtr = a_ptr;
}

        /*------------------- Description --------------------
          Function to cancel the initializations of the timer
         ----------------------------------------------------*/
void TIMER1_deinit (void)
{
    TCCR1A = 0x00;
    TCCR1B = 0x00;
    TCCR1C = 0x00;
}

/*-------------------------------------------------------------------------
                            TIMER3 Functions
 -------------------------------------------------------------------------*/
/*
 * Description : Function to initialize the Timer driver
 *   1. Select Timer Mode ( Normal - Compare - Square - PWM )
 *   2. Set the required clock.
 *   3. Set Compare Value if The Timer in Compare Mode
 *   4. Set OC1A/OC1B Pin Mode in Square Mode
 *   5. Enable the Timer Normal-Compare Interrupt.
 *   6. Initialize TIMER1 Registers
```

```
 */
void TIMER3_init(TIMER_ConfigType *Config_ptr)
{
    /* set a global variable for clock to use it in
     * restart timer function */
    g_T3clock = Config_ptr->clock ;

    /* Set TIMER1 In Normal Mode */
    if (Config_ptr->mode == NORMAL)
    {
        /* Set Timer initial value to 0 */
        TCNT3 = 0;

        /*  Enable TIMER3 Overflow Interrupt */
        TIMSK |= (1<<TOIE3);

        /* Configure the timer control register
         * 1. Non PWM mode    => FOC3A=1 & FOC3B=1
         * 2. Normal Mode     => WGM31=0 & WGM30=0 & WGM33=0 & WGM32=0
         * 3. OC3A/B Mode     => COM3A1 & COM3A0 & COM3B1 & COM3B0
(Disconnected)
         * 4. clock           => CS32 & CS31 & CS30
         * TCCR3A => COM3A1 COM3A0 COM3B1 COM3B0 FOC3A FOC3B WGM31 WGM30
         * TCCR3B => ICNC3 ICES3 – WGM33 WGM32 CS32 CS31 CS30
         */

        TCCR3C = (1<<FOC3A) | (1<<FOC3B) | (1<<FOC3C) ;
        TCCR3B = ((Config_ptr->clock) << CS30) ;
    }

    /* Set TIMER3 In Compare Mode with OCR1A Value */
    else if (Config_ptr->mode == COMP)
    {
        /* Set pin PE3/OC3A as output pin */
        SET_BIT (DDRE,PE3);
        /* Set pin PE4/OC3B/INT4 as output pin */
        SET_BIT (DDRE,PE4);
        /* Set pin PE5/OC3C/INT5 as output pin */
        SET_BIT (DDRE,PE5);

        /* Channels interrupt Enable */
        TIMSK |= (1<<OCIE3A) | (1<<OCIE3B) | (1<<OCIE3C);

        /* Initial Value of The Timer TCNT*/
        TCNT3 = 0 ;
```

```
        /* Compare Value for Channel A */
        OCR3A = Config_ptr->OCRValue ;
        /* Compare Value for Channel B */
        OCR3B = Config_ptr->OCR1BValue;
        /* Compare Value for Channel C */
        OCR3C = Config_ptr->OCR1CValue;

        /* Configure the timer control register
         * 1. Non PWM mode    => FOC3A=1 & FOC3B=1
         * 2. CTC Mode        => WGM31=0 & WGM30=0 & WGM33=0 & WGM32=1
         * 3. OC3A/B/C Mode   => COM3A1 & COM3A0 & COM3B1 & COM3B0 & COM3C1 &
COM3C0
         * 4. clock           => CS32 & CS31 & CS30
         * TCCR3A => COM3A1 COM3A0 COM3B1 COM3B0 COM3C1 COM3C0 WGM31 WGM30
         * TCCR3B => ICNC3 ICES3 – WGM33 WGM32 CS32 CS31 CS30
         * TCCR3C => FOC3A FOC3B FOC3C
         */

        TCCR3A = ((Config_ptr->OC) << COM3A0) | ((Config_ptr->OC1B) <<
COM3B0) | ((Config_ptr->OC1C) << COM3C0);
        TCCR3B = (1<<WGM33) | (1<<WGM32) | ((Config_ptr->clock) << CS30) ;
        TCCR3C = (1<<FOC3A) | (1<<FOC3B) | (1<<FOC3C);
    }

    /* Set TIMER3 In CTC Square Wave Mode */
    else if (Config_ptr->mode == CTC)
    {
        /* Set pin PE3/OC3A as output pin */
        SET_BIT (DDRE,PE3);
        /* Set pin PE4/OC3B/INT4 as output pin */
        SET_BIT (DDRE,PE4);
        /* Set pin PE5/OC3C/INT5 as output pin */
        SET_BIT (DDRE,PE5);

        /* Channels interrupt Enable */
        TIMSK |= (1<<OCIE3A) | (1<<OCIE3B) | (1<<OCIE3C);

        /* Initial Value of The Timer TCNT*/
        TCNT3 = 0 ;

        /* Compare Value for Channel A */
        OCR3A = Config_ptr->OCRValue ;
        /* Compare Value for Channel B */
        OCR3B = Config_ptr->OCR1BValue;
```

```
        /* Compare Value for Channel C */
        OCR3C = Config_ptr->OCR1CValue;

        /* Configure the timer control register
         * 1. Non PWM mode    => FOC3A=1 & FOC3B=1
         * 2. CTC Mode        => WGM31=0 & WGM30=0 & WGM33=0 & WGM32=1
         * 3. OC1A/B Mode     => COM3A1 & COM3A0 & COM3B1 & COM3B0 & COM3C1 &
COM3C0
         * 4. clock           => CS32 & CS31 & CS30
         * TCCR3A => COM3A1 COM3A0 COM3B1 COM3B0 COM3C1 COM3C0 WGM31 WGM30
         * TCCR3B => ICNC3 ICES3 – WGM33 WGM32 CS32 CS31 CS30
         * TCCR3C => FOC3A FOC3B FOC3C
         */

        TCCR3A = ((Config_ptr->OC) << COM3A0) | ((Config_ptr->OC1B) <<
COM3B0) | ((Config_ptr->OC1C) << COM3C0);
        TCCR3B = (1<<WGM32) | ((Config_ptr->clock) << CS30) ;
        TCCR3C = (1<<FOC3A) | (1<<FOC3B) | (1<<FOC3C);
    }

    /* Set TIMER1 In PWM Mode with Duty Cycle */
    else if (Config_ptr->mode == PWM)
    {
        /* Set pin PE3/OC3A as output pin */
        SET_BIT (DDRE,PE3);
        /* Set pin PE4/OC3B/INT4 as output pin */
        SET_BIT (DDRE,PE4);
        /* Set pin PE5/OC3C/INT5 as output pin */
        SET_BIT (DDRE,PE5);

        /* Channels interrupt Enable */
        TIMSK |= (1<<OCIE3A) | (1<<OCIE3B) | (1<<OCIE3C);

        /* Initial Value of The Timer TCNT*/
        TCNT3 = 0 ;

        /* Duty Cycle Value for Channel A */
        OCR3A = Config_ptr->OCRValue;
        /* Duty Cycle Value for Channel B */
        OCR3B = Config_ptr->OCR1BValue;
        /* Duty Cycle Value for Channel C */
        OCR3C = Config_ptr->OCR1CValue;

        /* using ICR1 as Top Value for Duty Cycle for all Channels
         * Duty Cycle for PWM Mode = OCR1x/ICR1 */
```

```
        ICR3 = Config_ptr->ICR1Value;

        /* Configure the timer control register
         * 1. PWM mode          => FOC3A=0 & FOC3B=0
         * 2. PWM Mode          => WGM31=1 & WGM30=0 & WGM33=1 & WGM32=1
         * 3. OC3A/B/C Mode     => COM3A1 & COM3A0 & COM3B1 & COM3B0 & COM3C1 &
COM3C0
         * 4. clock             => CS32 & CS31 & CS30
         * TCCR3A => COM3A1 COM3A0 COM3B1 COM3B0 COM3C1 COM3C0 WGM31 WGM30
         * TCCR3B => ICNC3 ICES3 – WGM33 WGM32 CS32 CS31 CS30
         *
         */
        TCCR3A = ((Config_ptr->OC) << COM3A0) | ((Config_ptr->OC1B) <<
COM3B0)
                  |((Config_ptr->OC1C) << COM3C0) | (1<<WGM31);

        TCCR3B = (1<<WGM33) | (1<<WGM32) | ((Config_ptr->clock) << CS30) ;
    }
}
/*
 * Description: Function to clear the TIMER1 Value to start count from ZERO
 */
void TIMER3_resetTimer(void)
{
    TCNT3 = 0;
}

/*
 * Description: Function to Stop the TIMER1
 * CLEAR CS12 CS11 CS10
 * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
 */
void TIMER3_stopTimer(void)
{
    /* Clear Clock Bits */
    TCCR3B &= 0xF8 ;
}

/*
 * Description: Function to Restart the TIMER1
 * CLEAR CS12 CS11 CS10
 * TCCR1B => ICNC1 ICES1 – WGM13 WGM12 CS12 CS11 CS10
 */
void TIMER3_restartTimer(void)
{
```

```
    /* Clear Clock Bits */
    TCCR3B &= 0xF8 ;

    /* Set Clock Bits */
    TCCR3B |= ( g_T3clock << CS30) ;
}

/*
 * Description: Function to Change Ticks (Compare Value) of Timer
 *                   using also to Change Duty Cycle in PWM Mode
 */
void TIMER3_Ticks(const uint16 Ticks3A, const uint16 Ticks3B)
{
    OCR3A = Ticks3A;
    OCR3B = Ticks3B;
}

/*
 * Description: Function to set the Call Back function address for TIMER2 .
 */
void TIMER3_setCallBack(void(*a_ptr)(void))
{
    /* Save the address of the Call back function in a global variable */
    g_TIMER3_callBackPtr = a_ptr;
}

        /*------------------- Description --------------------
          Function to cancel the initializations of the timer
         ----------------------------------------------------*/
void TIMER3_deinit (void)
{
    TCCR1A = 0x00;
    TCCR1B = 0x00;
    TCCR1C = 0x00;
}
```

```
/*------------------------------------------------------------------------
 - File Name: TIMERS.h
 -
 - Module: Timer0,Timer2 (8-bit) ------ Timer1,Timer3 (16-bit)
 -
 - Description: Header file for the Timers driver for AtMega128
 -
 ------------------------------------------------------------------------*/

/*------------------------------------------------------------------------
--
                            NOTES About Timer Driver
 ------------------------------------------------------------------------
--*/
/*
 - Notes:           - Timer0/Timer2 Maximum time in 1 overflow is 256ms
 -                    as 1 clock = 1ms
 -                    ( Maximum => Using small clock = 1Mhz , and large
prescaler = 1024 )
 -
 -                  - TIMER1/Timer3 Maximum time in 1 overflow is 65535ms (
65.5 Sec )
 -                    as 1 clock = 1ms
 -                    ( Maximum => Using small clock = 1Mhz , and large
prescaler = 1024 )
 -
 - Pins:       - PB3/OC0  -> Square Wave and PWM Mode in Timer0
 -                  - PD7/OC2  -> Square Wave and PWM Mode in Timer2
 -                  - PD5/OC1A -> Square Wave and PWM Mode in TIMER1A
 -                  - PD4/OC1B -> Square Wave  Mode in TIMER1B
 -
 - Clock And Prescaler :
 -                  Time = Prescaler / F_CPU * Ticks
 -                  For clock=1Mhz and prescale F_CPU/1024 every count will
take 1ms
 -                  Timer Clock => 1,000,000 / 1024 = 1Khz
 -                  clock timer period = 1 / 1Khz = 1ms
 -                  so put initial timer counter = 0  0 --> 255 (256ms per
overflow)
 -                  so we need timer to overflow 4 times to get a 1 second
 -
 - Example for Config Struct:
 -                  // F_CPU = 8Mhz       TIMER1 CTC Mode  1 Sec.
 -                  TIMER_ConfigType TIMER1_Config =
```

```
 -                      {.clock = F_CPU_1024, .mode = CTC, .OCRValue = 8000
};
 -                  TIMER1_Init(&TIMER1_Config);
 ------------------------------------------------------------------------
--*/

#ifndef TIMERS_H_
#define TIMERS_H_

#include "typedef.h"
#include "macros.h"
#include "micro_config.h"

/*-------------------------------------------------------------------------
                            Types Declaration
 -------------------------------------------------------------------------*/

typedef enum{
   NORMAL, CTC, COMP, PWM
}TIMER_Mode;

typedef enum{
   DISCONNECTED = 0, TOGGLE = 1, CLEAR = 2, SET = 3 , NON_INVERTING = 2 ,
INVERTING = 3
}OC_Pin_Mode;

typedef enum{
   NO_CLOCK, F_CPU_CLOCK, F_CPU_8, F_CPU_T2_64,F_CPU_T2_256,
F_CPU_T2_1024,EXTERNAL_FALLING_EDGE,EXTERNAL_RISING_EDGE

   /* additional Clock For TIMER2 */
   ,F_CPU_T0_32=3, F_CPU_T0_64,F_CPU_T0_128 ,F_CPU_T0_256, F_CPU_T0_1024
}TIMER_Clock;


typedef struct{
   /* For TIMER0:
    - NO_CLOCK, F_CPU_CLOCK, F_CPU_8, F_CPU_32 , F_CPU_64, F_CPU_128 ,
F_CPU_256, F_CPU_1024
    - For TIMER2:
    - F_CPU_T2_64, F_CPU_T2_256, F_CPU_T2_1024,
EXTERNAL_FALLING_EDGE,EXTERNAL_RISING_EDGE
    */
   TIMER_Clock clock ;
```

```
    /* NORMAL, COMP, CTC, PWM */
    TIMER_Mode mode ;

    /* values 0:255 ***** But in OCR1A 0:65535
     * using also for set Duty Cycle in PWM Mode
     * using as OCR0 , OCR1A , OCR2 Value */
    uint16 OCRValue ;

    /* CTC Mode => TOGGLE, CLEAR, SET
     * PWM Mode => NON_INVERTING, INVERTING
     * using as OC0 , OC1A , OC2 */
    OC_Pin_Mode OC ;

    /* values 0:65535
     * using for OCR1B value in TIMER1
     * and for ICR1 value as TOP value in TIMER1 PWM Duty Cycle */
    uint16 OCR1BValue ;

    /* CTC Mode => TOGGLE, CLEAR, SET
     * PWM Mode => NON_INVERTING, INVERTING */
    OC_Pin_Mode OC1B ;

    /* values 0:65535
     * using for OCR1C value in TIMER1 */
    uint16 OCR1CValue ;

    /* CTC Mode => TOGGLE, CLEAR, SET
     * PWM Mode => NON_INVERTING, INVERTING */
    OC_Pin_Mode OC1C ;

    /* Top Value for Duty Cycle in TIMER1/TIMER3 */
    uint16 ICR1Value;

}TIMER_ConfigType;


/*-------------------------------------------------------------------------
                        Timer0 Functions Prototypes
 -------------------------------------------------------------------------*/

void TIMER0_init(TIMER_ConfigType *Config_ptr);

void TIMER0_resetTimer(void);

void TIMER0_stopTimer(void);
```

```c
void TIMER0_restartTimer(void);

void TIMER0_ticks(const uint8 Ticks);

void TIMER0_callBack(void(*a_ptr)(void));

void TIMER0_deinit (void);

/*-------------------------------------------------------------------------
                        Timer2 Functions Prototypes
 -------------------------------------------------------------------------*/

/*
 * Description : Function to initialize the Timer driver
 * 1. Select Timer Mode ( Normal - Compare - Square - PWM )
 * 2. Set the required clock.
 * 3. Set Compare Value if The Timer in Compare Mode
 * 4. Set OC2 Pin Mode in Square Mode
 * 5. Enable the Timer Normal-Compare Interrupt.
 * 6. Initialize Timer2 Registers
 */
void TIMER2_init(TIMER_ConfigType *Config_ptr);

void TIMER2_resetTimer(void);

void TIMER2_stopTimer(void);

void TIMER2_restartTimer(void);

void TIMER_ticks(const uint8 Ticks);

void TIMER2_callBack(void(*a_ptr)(void));

void TIMER2_deinit (void);

/*-------------------------------------------------------------------------
                        Timer1 Functions Prototypes
 -------------------------------------------------------------------------*/
/*
 * Description : Function to initialize the Timer driver
 * 1. Select Timer Mode ( Normal - Compare - Square - PWM )
 * 2. Set the required clock.
 * 3. Set Compare Value if The Timer in Compare Mode
 * 4. Set OC1A/B Pin Mode in Square Mode
```

```
 * 5. Enable the Timer Normal-Compare Interrupt.
 * 6. Initialize TIMER1 Registers
 */
void TIMER1_init(TIMER_ConfigType *Config_ptr);

void TIMER1_resetTimer(void);

void TIMER1_stopTimer(void);

void TIMER1_restartTimer(void);

void TIMER1_Ticks(const uint16 Ticks1A, const uint16 Ticks1B);

void TIMER1_setCallBack(void(*a_ptr)(void));

void TIMER1_deinit (void);

/*-------------------------------------------------------------------------
                      Timer3 Functions Prototypes
 -------------------------------------------------------------------------*/

/*
 * Description : Function to initialize the Timer driver
 * 1. Select Timer Mode ( Normal - Compare - Square - PWM )
 * 2. Set the required clock.
 * 3. Set Compare Value if The Timer in Compare Mode
 * 4. Set OC1A/B Pin Mode in Square Mode
 * 5. Enable the Timer Normal-Compare Interrupt.
 * 6. Initialize TIMER1 Registers
 */
void TIMER3_init(TIMER_ConfigType *Config_ptr);

void TIMER3_resetTimer(void);

void TIMER3_stopTimer(void);

void TIMER3_restartTimer(void);

void TIMER3_Ticks(const uint16 Ticks1A, const uint16 Ticks1B);

void TIMER3_setCallBack(void(*a_ptr)(void));

void TIMER3_deinit (void);

#endif /* TIMERS_H_ */
```

5.2.1.2 UART

```
/*-------------------------------------------------------------------------
 - File Name: UART.c
 -
 - Module: UART0,UART1
 -
 - Description: Source file for the UART driver for AtMega128
 -
 -------------------------------------------------------------------------*/

#include "UART.h"

/*-------------------------------------------------------------------------
                          Functions of UART1
 -------------------------------------------------------------------------*/
        /*------------------- Description --------------------
         -  Function to initialize the UART0 driver
         -  1. Select parity mode.
         -  2. Select stop bit.
         -  3. Select data size.
         -  4. Set baud rate.
         ----------------------------------------------------*/
void UART0_init (const UART_ConfigType *config_ptr)
{
    /* U2X0 = 1 for double transmission speed for Asynchronous */
    UCSR0A = (1<<U2X0);
    /*------------------- UCSRB Description --------------------
     - RXCIE0 = 0 Disable RX Complete Interrupt Enable
     - TXCIE0 = 0 Disable TX Complete Interrupt Enable
     - UDRIE0 = 0 Disable Data Register Empty Interrupt Enable
     - RXEN0  = 1 Receiver Enable
     - TXEN0  = 1 Transmitter Enable
     - UCSZ02 & RXB80 & TXB80 Used for 9-bit data mode
     ----------------------------------------------------------*/
    UCSR0B = (1<<RXEN0) | (1<<TXEN0);

    if ((config_ptr -> data_size) == 7)
    {
    UCSR0B |= (1<<UCSZ02) | (1<<RXB80) | (1<<TXB80);
    }

    /*------------------- UCSRC Description --------------------
     - UMSEL   = 0 Asynchronous Operation
     - UPM01:0    Parity Mode Selected
```

```
     - USB0S      Stop Bit Selected
     - UCPOL0   = 0 (Used with the Synchronous operation only)
     - UCSZ01:0   Data Size Selected
     ------------------------------------------------------------*/
    UCSR0C = ((config_ptr -> parity_mode)  <<UPM00) |
             ((config_ptr -> stop_bit)     <<USBS0) |
             (((config_ptr -> data_size)&3) <<UCSZ00);

    /*Equation for calculating UBRR depending on baud rate
     - UBRR0L for the least 8-bits
     - UBBR0H for the most 4-bits
     */
    UBRR0L = (((F_CPU/((config_ptr -> baud_rate) * 8))-1));
    UBRR0H = ((((F_CPU/((config_ptr -> baud_rate) * 8))-1)) >> 8);
}

        /*------------------- Description --------------------
         -  Function to send a byte
         -  1. Receive desired byte to be sent
         -  2. Use Polling method
         ----------------------------------------------------*/
void UART0_sendByte (const uint8 data)
{
    /* UDRE0 is 1 when the buffer is empty and ready to receive
     * new data, So wait until the flag is 1*/
    while (CHECK_CLEAR(UCSR0A,UDRE0)){}
    /* Write data into UDR register */
    UDR0 = data;
}

        /*------------------- Description --------------------
         -  Function to receive a byte
         -  1. Return data into UDR
         -  2. Use Polling method
         ----------------------------------------------------*/
uint8 UART0_receiveByte (void)
{
    /* RXC0 is 1 when there are unread in a receive buffer
     * (Receive Completed), So wait until the flag is 1*/
    while (CHECK_CLEAR(UCSR0A,RXC0)){}
    /* return data from UDR register */
    return UDR0;
}

        /*------------------- Description --------------------
```

```
          -  Function to send a String
          -  1. Receive desired string to be sent into pointer
          ------------------------------------------------------*/
void UART0_sendString (const uint8 *str)
{
    while (*str != '\0')
    {
         UART0_sendByte(*str);
         str++;
    }
}

        /*------------------- Description --------------------
        -  Function to receive a String
        -  1. Receive desired string to be sent into pointer
        -  2. Use Polling method
        ------------------------------------------------------*/
void UART0_receiveString (uint8 *str)
{
    str = UART0_receiveByte;
    while (*str != '#')
    {
         str++;
         str = UART0_receiveByte;
    }
    str = '\0';
}

/*-------------------------------------------------------------------------
                          Functions of UART1
 -------------------------------------------------------------------------*/
        /*------------------- Description --------------------
         -  Function to initialize the UART1 driver
         -  1. Select parity mode.
         -  2. Select stop bit.
         -  3. Select data size.
         -  4. Set baud rate.
         ------------------------------------------------------*/
void UART1_init (const UART_ConfigType *config_ptr)
{
    /* U2X1 = 1 for double transmission speed for Asynchronous */
    UCSR1A = (1<<U2X1);
    /*------------------- UCSRB Description --------------------
     - RXCIE1 = 0 Disable RX Complete Interrupt Enable
     - TXCIE1 = 0 Disable TX Complete Interrupt Enable
```

```
     - UDRIE1 = 0 Disable Data Register Empty Interrupt Enable
     - RXEN1  = 1 Receiver Enable
     - TXEN1  = 1 Transmitter Enable
     - UCSZ12 & RXB81 & TXB81 Used for 9-bit data mode
     ------------------------------------------------------------*/
    UCSR1B = (1<<RXEN1) | (1<<TXEN1);

    if ((config_ptr -> data_size) == 7)
    {
    UCSR1B |= (1<<UCSZ12) | (1<<RXB81) | (1<<TXB81);
    }

    /*------------------- UCSRC Description --------------------
     - UMSEL1   = 0 Asynchronous Operation
     - UPM11:0    Parity Mode Selected
     - USB1S      Stop Bit Selected
     - UCPOL1   = 0 (Used with the Synchronous operation only)
     - UCSZ11:0   Data Size Selected
     ------------------------------------------------------------*/
    UCSR0C = ((config_ptr -> parity_mode)  <<UPM10) |
             ((config_ptr -> stop_bit)     <<USBS1) |
             (((config_ptr -> data_size)&3) <<UCSZ10);

    /*Equation for calculating UBRR depending on baud rate
     - UBRR1L for the least 8-bits
     - UBBR1H for the most 4-bits
     */
    UBRR1L = (((F_CPU/((config_ptr -> baud_rate) * 8))-1));
    UBRR1H = ((((F_CPU/((config_ptr -> baud_rate) * 8))-1)) >> 8);
}

        /*------------------- Description --------------------
         -  Function to send a byte
         -  1. Receive desired byte to be sent
         -  2. Use Polling method
         ------------------------------------------------------*/
void UART1_sendByte (const uint8 data)
{
    /* UDRE1 is 1 when the buffer is empty and ready to receive
     * new data, So wait until the flag is 1*/
    while (CHECK_CLEAR(UCSR1A,UDRE1)){}
    /* Write data into UDR register */
    UDR1 = data;
}
```

```
        /*------------------- Description --------------------
         -  Function to receive a byte
         -  1. Return data into UDR
         -  2. Use Polling method
         ----------------------------------------------------*/
uint8 UART1_receiveByte (void)
{
    /* RXC1 is 1 when there are unread in a receive buffer
     * (Receive Completed), So wait until the flag is 1*/
    while (CHECK_CLEAR(UCSR1A,RXC1)){}
    /* return data from UDR register */
    return UDR1;
}

        /*------------------- Description --------------------
          -  Function to send a String
          -  1. Receive desired string to be sent into pointer
          ----------------------------------------------------*/
void UART1_sendString (const uint8 *str)
{
    while (*str != '\0')
    {
         UART1_sendByte(*str);
         str++;
    }
}

        /*------------------- Description --------------------
        -  Function to receive a String
        -  1. Receive desired string to be sent into pointer
        -  2. Use Polling method
        ----------------------------------------------------*/
void UART1_receiveString (uint8 *str)
{
    str = UART1_receiveByte;
    while (*str != '#')
    {
         str++;
         str = UART1_receiveByte;
    }
    str = '\0';
}
```

```c
/*-------------------------------------------------------------------------
 - File Name: UART.h
 -
 - Module: UART0,UART1
 -
 - Description: Header file for the UART driver for AtMega128
 -
 -------------------------------------------------------------------------*/

#ifndef UART_H_
#define UART_H_


#include "macros.h"
#include "typedef.h"
#include "micro_config.h"


/*-------------------------------------------------------------------------
                              Types Declaration
 -------------------------------------------------------------------------*/

typedef enum{
    NO_PARITY,EVEN_PARITY = 2, ODD_PARITY
}UART_ParityMode;

typedef enum{
    _1_bit,_2_bit
}UART_StopBitSelect;

typedef enum{
    _5_bit,_6_bit,_7_bit,_8_bit,_9_bit=7
}UART_DataSize;

typedef enum{
    BR2400 = 2400,BR4800 = 4800,BR9600 = 9600, BR14400=14400,
    BR19200 = 19200,BR28800=28800,BR38400 = 38400,BR57600 = 57600,
    BR76800=76800,BR115200 = 115200,BR230400=230400, BR250000=250000,
    BR500000=500000, BR1M=1000000
}UART_BaudRate;

typedef struct{
    UART_ParityMode    parity_mode;
    UART_StopBitSelect stop_bit;
    UART_DataSize      data_size;
```

```
    UART_BaudRate      baud_rate;
}UART_ConfigType;
/*-------------------------------------------------------------------------
                      Functions UART0 Prototypes
 -------------------------------------------------------------------------*/

void UART0_init (const UART_ConfigType *);

void UART0_sendByte (const uint8);

uint8 UART0_receiveByte (void);

void UART0_sendString (const uint8 *);

void UART0_receiveString (uint8 *);

void UART0_intgerToString(uint8 variable,uint8 *arr);

/*-------------------------------------------------------------------------
                     Functions UART1 Prototypes
 -------------------------------------------------------------------------*/

void UART1_init (const UART_ConfigType *);

void UART1_sendByte (const uint8);

uint8 UART1_receiveByte (void);

void UART1_sendString (const uint8 *);

void UART1_receiveString (uint8 *);

#endif /* UART_H_ */
```

5.2.1.3 Input Capture Unit [ICU]

```
/*-------------------------------------------------------------------------
-
 -
 - Module: ICU
 -
 - File Name: icu.c
 -
 - Description: Source file for the AVR ICU driver
 -
 -
 -------------------------------------------------------------------------
-*/

#include "ICU.h"

/*-------------------------------------------------------------------------
--
                              Global Variables
 -------------------------------------------------------------------------
--*/

/* Global variables to hold the address of the call back function in the
application */
static void (*g_ICU1_callBackPtr)(void) = NULL_PTR;

static void (*g_ICU3_callBackPtr)(void) = NULL_PTR;

/*-------------------------------------------------------------------------
--
                         Interrupt Service Routines
 -------------------------------------------------------------------------
--*/

ISR(TIMER1_CAPT_vect)
{
    if(g_ICU1_callBackPtr != NULL_PTR)
    {
          /* Call the Call Back function in the application after the edge is
detected */
          (*g_ICU1_callBackPtr)(); /* another method to call the function using
pointer to function g_callBackPtr(); */
    }
}
```

```c
ISR(TIMER3_CAPT_vect)
{
    if(g_ICU3_callBackPtr != NULL_PTR)
    {
         /* Call the Call Back function in the application after the edge is
detected */
         (*g_ICU3_callBackPtr)(); /* another method to call the function using
pointer to function g_callBackPtr(); */
    }
}

/*-------------------------------------------------------------------------
--
                        Functions Definitions
 -------------------------------------------------------------------------
--*/

/*-------------------------------------------------------------------------
--
                                ICU TIMER 1
-------------------------------------------------------------------------
-*/
/*
 * Description : Function to initialize the ICU driver
 *   1. Set the required clock.
 *   2. Set the required edge detection.
 *   3. Enable the Input Capture Interrupt.
 *   4. Initialize Timer1 Registers
 */
void ICU1_init(const ICU_ConfigType * Config_Ptr)
{
    /* Configure ICP3/PE3 as i/p pin */
    CLEAR_BIT(DDRE,PD4);
    //SET_BIT(PORTE,PD4);

    /* Timer1 always operates in Normal Mode */
    TCCR1C = (1<<FOC1A) | (1<<FOC1B) | (1<<FOC1C);

    /*
     * insert the required clock value in the first three bits (CS30, CS31 and
CS32)
     * of TCCR1B Register
     */
    TCCR1B = (TCCR1B & 0xF8) | (Config_Ptr->clock);
```

```
    /*
     * insert the required edge type in ICES3 bit in TCCR3B Register
     */
    TCCR1B = (TCCR1B & 0xBF) | ((Config_Ptr->edge)<<6);

    /* Initial Value for Timer1 */
    TCNT1 = 0;

    /* Initial Value for the input capture register */
    ICR1 = 0;

    /* Enable the Input Capture interrupt to generate an interrupt when edge is
detected on ICP1/PD6 pin */
    TIMSK |= (1<<TICIE1);
}

/*
 * Description: Function to set the Call Back function address.
 */
void ICU1_setCallBack(void(*a_ptr)(void))
{
    /* Save the address of the Call back function in a global variable */
    g_ICU1_callBackPtr = a_ptr;
}

/*
 * Description: Function to set the required edge detection.
 */
void ICU1_setEdgeDetectionType( Icu_EdgeType a_edgeType)
{
    /*
     * insert the required edge type in ICES1 bit in TCCR1B Register
     */
    TCCR1B = (TCCR1B & 0xBF) | (a_edgeType<<6);
}

/*
 * Description: Function to get the Timer1 Value when the input is captured
 *              The value stored at Input Capture Register ICR1
 */
uint16 ICU1_getInputCaptureValue(void)
{
    return ICR1;
}
```

```c
/*
 * Description: Function to clear the Timer1 Value to start count from ZERO
 */
void ICU1_clearTimerValue(void)
{
    TCNT1 = 0;
}

/*
 * Description: Function to disable the Timer1 to stop the ICU Driver
 */
void ICU1_DeInit(void)
{
    /* Clear All Timer1 Registers */
    TCCR1A = 0;
    TCCR1B = 0;
    TCCR1C = 0;
    TCNT1 = 0;
    ICR1 = 0;

    /* Disable the Input Capture interrupt */
    ETIMSK &= ~(1<<TICIE1);
}


/*------------------------------------------------------------------------------
--
                                ICU TIMER 3
------------------------------------------------------------------------------
-*/
/*
* Description : Function to initialize the ICU driver
*    1. Set the required clock.
*    2. Set the required edge detection.
*    3. Enable the Input Capture Interrupt.
*    4. Initialize Timer1 Registers
*/
void ICU3_init(const ICU_ConfigType * Config_Ptr)
{
    /* Configure ICP3/PE3 as i/p pin */
    CLEAR_BIT(DDRE,PE7);
    //SET_BIT(PORTE,PE7);

    /* Timer1 always operates in Normal Mode */
    TCCR3C = (1<<FOC3A) | (1<<FOC3B) | (1<<FOC3C);
```

```
    /*
     * insert the required clock value in the first three bits (CS30, CS31 and
CS32)
     * of TCCR1B Register
     */
    TCCR3B = (TCCR3B & 0xF8) | (Config_Ptr->clock);
    /*
    * insert the required edge type in ICES3 bit in TCCR3B Register
     */
    TCCR3B = (TCCR3B & 0xBF) | ((Config_Ptr->edge)<<6);

    /* Initial Value for Timer1 */
    TCNT3 = 0;

    /* Initial Value for the input capture register */
    ICR3 = 0;

    /* Enable the Input Capture interrupt to generate an interrupt when edge is
detected on ICP1/PD6 pin */
    ETIMSK |= (1<<TICIE3);
}

/*
* Description: Function to set the Call Back function address.
*/
void ICU3_setCallBack(void(*a_ptr)(void))
{
    /* Save the address of the Call back function in a global variable */
    g_ICU3_callBackPtr = a_ptr;
}

/*
* Description: Function to set the required edge detection.
*/
void ICU3_setEdgeDetectionType(Icu_EdgeType a_edgeType)
{
    /*
     * insert the required edge type in ICES1 bit in TCCR1B Register
     */
    TCCR3B = (TCCR3B & 0xBF) | (a_edgeType<<6);
}

/*
* Description: Function to get the Timer1 Value when the input is captured
```

```
*              The value stored at Input Capture Register ICR1
*/
uint16 ICU3_getInputCaptureValue(void)
{
    return ICR3;
}

/*
* Description: Function to clear the Timer1 Value to start count from ZERO
*/
void ICU3_clearTimerValue(void)
{
    TCNT3 = 0;
}

/*
* Description: Function to disable the Timer1 to stop the ICU Driver
*/
void ICU3_DeInit(void)
{
    /* Clear All Timer1 Registers */
    TCCR3A = 0;
    TCCR3B = 0;
    TCCR3C = 0;
    TCNT3 = 0;
    ICR3 = 0;

    /* Disable the Input Capture interrupt */
    ETIMSK &= ~(1<<TICIE3);
}
```

```c
/*-------------------------------------------------------------------------------
 -
 - Module: ICU
 -
 - File Name: icu.h
 -
 - Description: Header file for the AVR ICU driver
 -
 -------------------------------------------------------------------------------*/

#ifndef ICU_H_
#define ICU_H_

#include "macros.h"
#include "typedef.h"
#include "micro_config.h"

/*-------------------------------------------------------------------------------
                            Types Declaration
 -------------------------------------------------------------------------------*/
typedef enum
{
    ICU_NO_CLOCK, ICU_F_CPU_CLOCK, ICU_F_CPU_8, ICU_F_CPU_T2_64,
ICU_F_CPU_T2_256, ICU_F_CPU_T2_1024
    ,ICU_EXTERNAL_FALLING_EDGE,ICU_EXTERNAL_RISING_EDGE
}Icu_Clock;

typedef enum
{
    FALLING,RISING
}Icu_EdgeType;

typedef struct
{
    Icu_Clock clock;
    Icu_EdgeType edge;
}ICU_ConfigType;

/*-------------------------------------------------------------------------------
-
                          Functions Prototypes
 -------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------
                              ICU Timer 1
 -------------------------------------------------------------------------------*/
```

```c
/*
 * Description : Function to initialize the ICU driver
 *   1. Set the required clock.
 *   2. Set the required edge detection.
 *   3. Enable the Input Capture Interrupt.
 *   4. Initialize Timer1 Registers
 */
void ICU1_init(const ICU_ConfigType * Config_Ptr);

/*
 * Description: Function to set the Call Back function address.
 */
void ICU1_setCallBack(void(*a_ptr)(void));

/*
 * Description: Function to set the required edge detection.
 */
void ICU1_setEdgeDetectionType(const Icu_EdgeType edgeType);

/*
 * Description: Function to get the Timer1 Value when the input is captured
 *              The value stored at Input Capture Register ICR1
 */
uint16 ICU1_getInputCaptureValue(void);

/*
 * Description: Function to clear the Timer1 Value to start count from ZERO
 */
void ICU1_clearTimerValue(void);

/*
 * Description: Function to disable the Timer1 to stop the ICU Driver
 */
void ICU1_DeInit(void);

/*-------------------------------------------------------------------------
                            ICU Timer 3
 ------------------------------------------------------------------------- */
/*
 * Description : Function to initialize the ICU driver
 *   1. Set the required clock.
 *   2. Set the required edge detection.
 *   3. Enable the Input Capture Interrupt.
 *   4. Initialize Timer1 Registers
 */
```

```c
void ICU3_init(const ICU_ConfigType * Config_Ptr);

/*
 * Description: Function to set the Call Back function address.
 */
void ICU3_setCallBack(void(*a_ptr)(void));

/*
 * Description: Function to set the required edge detection.
 */
void ICU3_setEdgeDetectionType(const Icu_EdgeType edgeType);

/*
 * Description: Function to get the Timer1 Value when the input is captured
 *              The value stored at Input Capture Register ICR1
 */
uint16 ICU3_getInputCaptureValue(void);

/*
 * Description: Function to clear the Timer1 Value to start count from ZERO
 */
void ICU3_clearTimerValue(void);

/*
 * Description: Function to disable the Timer1 to stop the ICU Driver
 */
void ICU3_DeInit(void);
#endif /* ICU_H_ */
```

5.2.1.4 Types Definitions

```
/*-------------------------------------------------------------------------
 - File Name: typedef.h
 -
 - Description: Some types decelerations to make the code more Generic
 -
-------------------------------------------------------------------------*/

#ifndef TYPEDEF_H_
#define TYPEDEF_H_

/* Boolean Data Type */
typedef unsigned char bool;

/* Boolean Values */
#ifndef FALSE
#define FALSE       (0u)
#endif
#ifndef TRUE
#define TRUE        (1u)
#endif

#define HIGH        (1u)
#define LOW         (0u)

typedef unsigned char         uint8;
typedef signed char           sint8;
typedef unsigned short        uint16;
typedef signed short          sint16;
typedef unsigned long         uint32;
typedef signed long           sint32;
typedef unsigned long long    uint64;
typedef signed long long      sint64;
typedef float                 float32;
typedef double                float64;

#endif /* TYPEDEF_H_ */
```

5.2.1.5 Micro Configurations

```
/*------------------------------------------------------------------------
 - File Name: micor_config.h
 -
 - Description: Main Libraries used in every project using AVR MC
 -
------------------------------------------------------------------------*/
#ifndef MICRO_CONFIG_H_
#define MICRO_CONFIG_H_

#ifndef F_CPU
#define F_CPU 8000000UL /*8MHz Clock frequency*/
#endif

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>


#endif /* MICRO_CONFIG_H_ */
```

5.2.1.6 Macros

```
/*------------------------------------------------------------------------
 - File Name: macros.c
 -
 - Description: Most used common macros
 -
------------------------------------------------------------------------*/
#ifndef MACROS_H_
#define MACROS_H_

/* NULL define */
#ifndef NULL_PTR
#define NULL_PTR ((void*)(0))
#endif

#define SET_BIT(REG,BIT_NUM)       (REG |= (1 << BIT_NUM))

#define CLEAR_BIT(REG,BIT_NUM)     (REG &= (~(1 << BIT_NUM)))

#define TOGGLE_BIT(REG,BIT_NUM)    (REG ^= (1 << BIT_NUM))

#define ROTATE_BIT_R(REG,BIT_NUM)  (REG = (REG >> BIT_NUM)|(REG << (8 -
BIT_NUM)))

#define ROTATE_BIT_L(REG,BIT_NUM)  (REG = (REG << BIT_NUM)|(REG >> (8 -
BIT_NUM)))

#define CHECK_SET(REG,BIT_NUM)     (REG & (1 << BIT_NUM))

#define CHECK_CLEAR(REG,BIT_NUM)   (!(REG & (1 << BIT_NUM)))

#endif /* MACROS_H_ */
```

5.2.2 Designed Code

5.2.2.1 ECU Signal Calibration

```
/*-----------------------------------------------------------------------
 - File Name: Angle_Detection.h
 -
 - Description: Header file for ECU signal calibration
 -
 -----------------------------------------------------------------------*/

#ifndef ANGLE_PWM_DETECTION_H_
#define ANGLE_PWM_DETECTION_H_

#include "TIMERS.h"
#include "UART.h"
#include "INT.h"
#include "ICU.h"

void PulseMeasure(void);
void angleDetection_INT0(void);
void dutyCycle_Calc(void);
void revCounter_TIMER2 (void);


#endif /*ANGLE_PWM_DETECTION_H_ */
```

```c
/*-------------------------------------------------------------------------
 - File Name: Angle_Detection.c
 -
 - Description: Source file for ECU signal calibration
 -
 -------------------------------------------------------------------------*/
#include "Angle_PWM_Detection.h"

volatile uint8    rev=0;
volatile uint8    ticks=0 ;
volatile uint16   T_ON = 0;
volatile uint16   T_OFF = 0;
volatile uint16   T_TOTAL = 0;
volatile uint8    PulseState = 0;
volatile float32  DutyCycle = 0 ;
volatile uint8    g_edgeCount = 0;
volatile uint16   g_timeHigh = 0;
volatile uint16   g_timePeriodPlusHigh = 0;
volatile uint16   g_timePeriod = 0;
volatile float32  g_period = 0;

void PulseMeasure(void)
{

    PulseState++;

    switch (PulseState)
    {
    case 1:
         ICU3_setEdgeDetectionType(FALLING);
         ICU3_clearTimerValue();

         break;

    case 2:
         T_ON = ICU3_getInputCaptureValue();
         ICU3_setEdgeDetectionType(RISING);
         ICU3_clearTimerValue();

         break;

    case 3:
         T_OFF = ICU3_getInputCaptureValue();
         ICU3_setEdgeDetectionType(FALLING);
         ICU3_clearTimerValue();
```

```
		break;

	case 4:
		T_TOTAL = ICU3_getInputCaptureValue();
		ICU3_setEdgeDetectionType(RISING);
		ICU3_clearTimerValue();
		PulseState = 0;

		break;

	default:
		/* Do Nothing */
		break;
	}
}

void periodMeasure (void)
{
	g_edgeCount++;
	switch (g_edgeCount)
	{
	case 1:
		/*
		 * Clear the timer counter register to start measurements from the
		 * first detected rising edge */
		ICU1_clearTimerValue();
		/* Detect falling edge */
		ICU1_setEdgeDetectionType(FALLING);
		break;

	case 2:
		/* Store the High time value */
		g_timeHigh = ICU1_getInputCaptureValue();
		/* Detect rising edge */
		ICU1_setEdgeDetectionType(RISING);
		ICU1_clearTimerValue();

		break;

	case 3:
		/* Store the Period time value */
		g_timePeriod = ICU1_getInputCaptureValue();
		/* Detect falling edge */
```

```
            ICU1_setEdgeDetectionType(FALLING);
            ICU1_clearTimerValue();
          break;

    case 4:
            /* Store the Period time value + High time value */
            g_timePeriodPlusHigh = ICU1_getInputCaptureValue();
            /* Clear the timer counter register to start measurements again */
            /* Detect rising edge */
            ICU1_setEdgeDetectionType(RISING);
            ICU1_clearTimerValue();
            g_edgeCount = 0;
            /* calculate the period */
            break;

    default:
            break;
    }
}

void periodMeasure_Calc (void)
{
    float32 period;
    ticks = TCNT2;
    uint8 arr[20];
    uint8 arr1[20];
    g_period = (((float32) (g_timeHigh)) /
((float32)(g_timePeriod)+(float32)(g_timeHigh))) * 100.00;

    sprintf(arr, "%d", (uint16)g_period);
    UART0_sendString("Period of each crank tooth : ");
    UART0_sendString(arr);
    UART0_sendString(" %");
    UART0_sendByte('\r');
    sprintf(arr1, "%d", (uint8)ticks);
    UART0_sendString("Tick No. : ");
    UART0_sendString(arr1);
    UART0_sendByte('\r');
}
void angleDetection_INT0(void)
{
    uint8 arr1[3];
    uint8 arr2[3];
```

```
    UART0_sendByte('\r');
    dutyCycle_Calc();
    periodMeasure_Calc ();



    UART0_sendByte('\r');

    UART0_sendString("Revolution number = ");
    sprintf(arr2, "%d", rev);
    UART0_sendString(arr2);
    UART0_sendByte('\r');


}

void dutyCycle_Calc(void)
{
    uint8 arr[4];

    DutyCycle = (((float32) (T_ON)) / ((float32)(T_OFF)+(float32)(T_ON))) *
100.00;
    sprintf(arr, "%d", (uint16)DutyCycle);
    UART0_sendString("Duty Cycle is : ");
    UART0_sendString(arr);
    UART0_sendString(" %");
    UART0_sendByte('\r');
}

void revCounter_TIMER2 (void)
{
    rev++;
    if (rev == 3)
         rev = 1;
}



int main(void)
{
    sei();


    ICU_ConfigType ICU1_Config = { .clock = ICU_F_CPU_T2_256, .edge = RISING };
    ICU1_init(&ICU1_Config);
```

```
    ICU1_clearTimerValue();
    ICU1_setCallBack(periodMeasure);

    ICU_ConfigType ICU3_Config = { .clock = ICU_F_CPU_T2_256, .edge = RISING };
    ICU3_init(&ICU3_Config);
    ICU3_clearTimerValue();
    ICU3_setCallBack(PulseMeasure);

    TIMER_ConfigType TIMER2_Config ={.clock=EXTERNAL_RISING_EDGE, .mode=COMP ,
.OCRValue=60 };
    TIMER2_init(&TIMER2_Config);
    TIMER2_callBack(revCounter_TIMER2);


    UART_ConfigType UART_Config
={.parity_mode=EVEN_PARITY,.stop_bit=_1_bit,.baud_rate=BR115200,
                .data_size=_8_bit};
    UART0_init(&UART_Config);
    UART0_sendString(" Graduation Project --> Signal from ECU ");


    INT0_init();
    INT0_setCallBack(angleDetection_INT0);


    while(1)
    {

    }
}
```

5.2.2.2 Operating Code

```
/*-----------------------------------------------------------------------
 - File Name: ECU.h
 -
 - Description: Header file for the ECU project
 -
 -----------------------------------------------------------------------*/

#ifndef ECU_H_
#define ECU_H_

#include "TIMERS.h"
#include "UART.h"
#include "ICU.h"

/*-----------------------------------------------------------------------
                            Functions Prototypes
 -----------------------------------------------------------------------*/

void startEngine (void);

void APP_edgeProcessing(void);

void revCounter_TIMER2 (void);


#endif /* ECU_H_ */
```

```c
/*-------------------------------------------------------------------------
 - File Name: ECU.c
 -
 - Description: Source file for the ECU project
 -
 -------------------------------------------------------------------------*/

#include "ECU.h"

/*-------------------------------------------------------------------------
                           Global Variables
 -------------------------------------------------------------------------*/

/* Global Variable to indicate the crank revolution   */
uint8 rev = 0;
uint8 injection_duty_cycle = 12 ; // 20ms
float32 ignition_duty_cycle = 1.8 ; // 3ms
uint8 injection_cylinder1_4_angle_on = 4;
uint8 injection_cylinder1_4_angle_off = 10;
uint8 injection_cylinder2_3_angle_on = 34;
uint8 injection_cylinder2_3_angle_off = 40;
uint8 ignition_cylinder1_4_angle_on = 2;
uint8 ignition_cylinder1_4_angle_off = 4;
uint8 ignition_cylinder2_3_angle_on = 32;
uint8 ignition_cylinder2_3_angle_off = 34;

uint8  g_edgeCount = 0;
uint16 g_timeHigh = 0;
uint16 g_timePeriod = 0;
uint16 g_timePeriodPlusHigh = 0;
float32 g_period =0;


/*-------------------------------------------------------------------------
                            Main Function
 -------------------------------------------------------------------------*/

uint8 main (void)
{
    sei();

    ICU_ConfigType icu3_config = {.clock=ICU_F_CPU_8, .edge=FALLING};
    ICU3_init(&icu3_config);
    ICU3_setCallBack(APP_edgeProcessing);
```

```
    TIMER_ConfigType timer2_config = { .clock=EXTERNAL_RISING_EDGE, .mode=COMP,
.OCRValue = 60};
    TIMER2_init(&timer2_config);
    TIMER2_callBack(revCounter_TIMER2);

    /*--------------------- Description ----------------------
     - Channel A -> Injection
     - Channel B -> Ignition
     ------------------------------------------------------------*/
    TIMER_ConfigType timer1_config = { .clock=EXTERNAL_RISING_EDGE, .mode=PWM,
.ICR1Value=60 ,.OCRValue =ignition_duty_cycle, .OC=NON_INVERTING,
                                       .OCR1BValue=injection_duty_cycle ,
.OC1B=NON_INVERTING};
    TIMER1_init(&timer1_config);
    TIMER1_stopTimer();

    DDRA = 0xFF;

    startEngine();


    while(1)
    {

        /*------------------PORTA Pins Description-----------------
                            Firing Order (1,3,4,2)
         - PA0 -> Injection 1
         - PA1 -> Injection 3
         - PA2 -> Injection 4
         - PA3 -> Injection 2
         - PA4 -> Ignition Cylinder 1 & 4
         - PA5 -> Ignition Cylinder 2 & 3
         ------------------------------------------------------------ */
        if (rev==0)
        {
            /**/
            if(TCNT2 == ignition_cylinder1_4_angle_on)
            {
                TIMER1_restartTimer();
                SET_BIT(PORTA,PA4);
            }
            else if(TCNT2 == ignition_cylinder1_4_angle_off)
            {
                CLEAR_BIT(PORTA,PA4);
            }
```

```
	/**/
	else if(TCNT2 == injection_cylinder1_4_angle_on)
	{
		TIMER1_restartTimer();
		SET_BIT(PORTA,PA2);
	}
	else if(TCNT2 == injection_cylinder1_4_angle_off)
	{
		CLEAR_BIT(PORTA,PA2);
	}
	/**/
	else if(TCNT2 == ignition_cylinder2_3_angle_on)
	{
		TIMER1_restartTimer();
		SET_BIT(PORTA,PA5);
	}
	else if(TCNT2 == ignition_cylinder2_3_angle_off)
	{
		CLEAR_BIT(PORTA,PA5);
	}
	/**/
	else if(TCNT2 == injection_cylinder2_3_angle_on)
	{
		TIMER1_restartTimer();
		SET_BIT(PORTA,PA3);
	}
	else if(TCNT2 == injection_cylinder2_3_angle_off)
	{
		CLEAR_BIT(PORTA,PA3);
	}
}

/**/
else if (rev==1)
{
	/**/
	if(TCNT2 == ignition_cylinder1_4_angle_on)
	{
		TIMER1_restartTimer();
		SET_BIT(PORTA,PA4);
	}
	else if(TCNT2 == ignition_cylinder1_4_angle_off)
	{
		CLEAR_BIT(PORTA,PA4);
	}
```

```
                /**/
                else if(TCNT2 == injection_cylinder1_4_angle_on)
                {
                      TIMER1_restartTimer();
                      SET_BIT(PORTA,PA0);
                }
                else if(TCNT2 == injection_cylinder1_4_angle_off)
                {
                      CLEAR_BIT(PORTA,PA0);
                }
                /**/
                else if(TCNT2 == ignition_cylinder2_3_angle_on)
                {
                      TIMER1_restartTimer();
                      SET_BIT(PORTA,PA5);
                }
                else if(TCNT2 == ignition_cylinder2_3_angle_off)
                {
                      CLEAR_BIT(PORTA,PA5);
                }
                /**/
                else if(TCNT2 == injection_cylinder2_3_angle_on)
                {
                      TIMER1_restartTimer();
                      SET_BIT(PORTA,PA1);
                }
                else if(TCNT2 == injection_cylinder2_3_angle_off)
                {
                      CLEAR_BIT(PORTA,PA1);
                }
          }
     }
}

/*-------------------------------------------------------------------------
                              Functions
 -------------------------------------------------------------------------*/

void startEngine (void)
{
          /* Set the Call back function pointer in the ICU driver */


          SET_BIT(PORTA,PA0);
```

```
        _delay_ms(500);
        CLEAR_BIT(PORTA,PA0);

        while(  g_period < 50.00 )
        {

                    SET_BIT(PORTA,PA1);
                    g_edgeCount = 0;
                    /* calculate the period */
                    g_period = (((float32) (g_timeHigh)) /
((float32)(g_timePeriod)+(float32)(g_timeHigh))) * 100.00;

        }
}

void APP_edgeProcessing(void)
{
    g_edgeCount++;
    switch (g_edgeCount)
    {
    case 1:
        /*
         * Clear the timer counter register to start measurements from the
         * first detected rising edge */
        ICU3_clearTimerValue();
        /* Detect falling edge */
        ICU3_setEdgeDetectionType(FALLING);
        break;

    case 2:
        /* Store the High time value */
        g_timeHigh = ICU3_getInputCaptureValue();
        /* Detect rising edge */
        ICU3_setEdgeDetectionType(RISING);
        ICU3_clearTimerValue();

        break;

    case 3:
        /* Store the Period time value */
        g_timePeriod = ICU3_getInputCaptureValue();
        /* Detect falling edge */
        ICU3_setEdgeDetectionType(FALLING);
        ICU3_clearTimerValue();
      break;
```

```
    case 4:
          /* Store the Period time value + High time value */
          g_timePeriodPlusHigh = ICU3_getInputCaptureValue();
          /* Clear the timer counter register to start measurements again */
          /* Detect rising edge */
          ICU3_setEdgeDetectionType(RISING);
          ICU3_clearTimerValue();
          g_edgeCount = 0;
          /* calculate the period */
          break;

    default:
          break;
    }
}

void revCounter_TIMER2 (void)
{
    rev++;
    if (rev == 2)
          rev = 0;
}
```

Chapter 6

Recommendations on future developments and Conclusion

Chapter description: In this chapter, subjects for future developments and improvements are presented. Recommendations on the safety approach to be followed in implementation & testing are given. And lastly, the conclusion for the project at hand is presented.

6.1 subjects for future developments

As this phase of design for the ECU at hand concluded at the successful run of the engine at the idle start condition using this as the operational control unit, future subjects can be complementary to what this initial and fundamental design phase reached. It could also greatly modify the operating mode of the engine under control (I.e. allowing for accelerations and decelerations), improve performance, fuel economy..etc.

Other subjects could be out of the engine control scope and focus other complementary sub-systems in the automobile, like: controlling the braking system, cruise control or even introducing artificially intelligent functions (i.e. autonomous drive).

Further researches could also be conducted to modify the operation of the engine from the fuel consumption aspect (i.e. turning off the fuel injectors for some pistons when high power output is not required).

6.2 Recommendations for safety

Maintaining high safety measures is extremely necessary when it comes to controlling the operation of internal combustion engines. Some faults in the operation of internal combustion engines could lead to devastating consequences, so to avoid that, utmost care must be taken if the researcher is to modify the operation from the firing timing for example. A faulty early firing angle of the fuel raises the possibility for the development of knocks at the piston head, while retarding the firing angle reduces the power output of the engine.

So, when working on the introduction of new functions or the modification of the initially created ECU, pre-work researches must be conducted in the field of concern in order to obtain a strong background of nature of the work to be done.

6.3 Conclusion

The designed ECU presented in its initial design phase that is capable of operating the engine under its control in the idle start run condition, forms the fundamental base for the designation of a fully operational, student-scale ECU. This ECU is an open-source, easily modifiable platform offering the chance for amateurs and researchers to implement their ideas and search for phenomena of their concern (after introducing their desired modifications) easily Without the restrictions that they would otherwise face if they were to use a normal operational automobile ECU. The normal vehicle ECU follows strict standards (AUTOSAR standard) which highly restrict the control parameters of the ECU's software so that it can ensure safety, durability, fuel efficiency and high performance.

This project also paves the path for the educational scope to dive into and include automobile control not only from the theoretical aspect but from the practical aspect as well.

References

BOOKS

[1] Bolton, W. (2015). *Mechatronics (Electronic control systems in mechanical and electrical engineering)(Sixth Edition).* London: Pearson Education Limited.

[2] Hatch, S. V. (2012). *Computerized Engine Control (9th Edition).* New York: Cengage Learning.

[3] Isermann, R. (2014). *Engine Modeling and Control (Modeling and Electronic Management of Internal Combustion Engines).* Berlin : Springer Heidelberg.

[4] Jain, P. K., & Asthana, R. B. (2002). *Automobile Engineering.* New Delhi: Tata McGraw-Hill Publishing Company Limited.

[5] Parmar, J. (2016, April 05). *Jignesh Parmar*. Retrieved from Slide Share: https://www.slideshare.net/jigneshparmar33/automotive-electronics-in-automobile-electronic-control-unit

[6] Pulkrabek, W. W. (2003). *Engineering Fundamentals of the Internal Combustion Engine.* New Jersey: Prentice Hall .

[7] Reif, K. (2015). *Bosch Professional Automotive inforation (Gasoline Engine Management (Systems and Components)).* Wiesbaden: Springer Vieweg.

Websites

[1] https://www.slideshare.net/jigneshparmar33/automotive-electronics-in-automobile-electronic-control-unit

Appendix

Git-Hub link of our repository:

https://github.com/M-Yehia/EngineControlUnit-ECU-/

- **Note:** the git hub link provided contains the ECU's MCU code, the schematic and files for PCB design and a softcopy for this book.