A Data Acquisition and Monitoring Platform for Diesel Engines based on the CAN-bus Protocol

Konstantinos Dimitriadis

School of Naval Architecture and Marine Engineering
Laboratory of Marine Engineering
National Technical University of Athens
Greece

Supervisor: Associate Professor George Papalambrou

Committee Member : Prof. N. Kyrtatos
Committee Member : Prof. C. Papadopoulos

September 2020

Abstract

The goal of this project is the design of a complete communications system with the capability of simultaneous remote monitoring control of the experimental engine testbed installed in the Laboratory of Marine Engineering. The system has been tested and implemented on the HIPPO-2 configuration, and gives the user capability of data acquisition and logging, live monitoring through real-time plotting and remote control. Data acquisition and engine control are achieved through a hardware setup which includes an Arduino UNO micro-controller, an MCP 2515 CAN transceiver and a PCF 8523 real-time clock running on a Raspberry Pi single board computer. This diploma project will describe in detail the function, communication protocol, connectivity and programming of the system. The main objectives were the following:

- Understanding the CAN bus architecture as well as the CAN-OPEN and J1939 protocols.
- Selecting all the parameters that would be monitored and/or controlled, including but not limited to the Engine Speed, Fuel Temperature, Power Output etc.
- Study and integration of the system components. Specifically, these included the microcontroller, CAN transceiver, real-time clock, single board computer and their respective programming languages and hardware connectivity.
- Integration of the system components to achieve communication between them and the HIPPO-2 installation.
- System optimization to run as efficiently as possible on the single board computer taking into consideration its limited processing power and graphic capabilities.
- Experimentation on the test bed to display live results, store data for further analysis and control specific parameters of the engine or request additional data that would otherwise not be transmitted.

Contents

List of Figures **4**

List of Tables **6**

1 Introduction **8**
1.1 Experimental Engine Testbed 8
1.2 Scope of Study 8
1.3 Remote Monitoring Systems 10

2 Communication Protocols **13**
2.1 Introduction to CAN 13
2.1.1 Physical Layer 14
2.1.2 CANopen Protocol 16
2.1.3 CAN J1939 Protocol 17
2.2 CAN Message Architecture 20
2.2.1 Error Checking 21
2.3 CAN Bus Compared to Other Communication Protocols 23
2.3.1 CAN & RS 232 23
2.3.2 CAN & RS 485 24

3 Data Acquisition **26**
3.1 Hardware 26
3.1.1 Microcontroller 26
3.1.2 CAN Transceiver 29
3.1.3 Real Time Clock 31
3.1.4 Single Board Computer 32
3.1.5 Hardware Connections 34
3.2 Programming 37
3.2.1 Programming Language and Data Analytics 38
3.2.2 Decoding and Processing Measurements 41
3.3 Data Representation 49
3.4 Transmitting Messages 50

4 Experiment **54**
4.1 Introduction 54
4.2 dSpace Monitoring System 54
4.3 Comparisson of Measurements 54

5 Conclusion and Future Works **62**
5.1 Summary 62
5.2 Future Developments 62

Bibliography 64

List of Figures

1.1 3-D layout of the hybrid test bed HIPPO-2. 9
1.2 Caterpillar C 9.3 Electronic Diesel Engine 9
1.3 Vessel Remote Monitoring System Data Collection Points 11
1.4 Vessel Remote Monitoring Architecture 12

2.1 Cheminax CAN Cable 15
2.2 CAN Physical Layer 16
2.3 CAN Dominant and Recessive Bus States 17
2.4 CAN Open Protocol 18
2.5 Difference Between the Standard and Extended Identifier 19
2.6 CAN Message Breakdown 20
2.7 J1939 CAN 2.0 Identifier 21
2.8 CAN Message Data-Frame Format 22
2.9 RS 485 and CAN Bus Output Differential 24

3.1 Arduino UNO Rev. 3 27
3.2 Arduino Serial Monitor displaying CAN data 27
3.3 Arduino UNO Schematic Representation 29
3.4 MCP2515 CAN Transceiver 30
3.5 MCP2515 Block Diagram 31
3.6 PCF 8523 Real Time Clock 32
3.7 Raspberry Pi 3 34
3.8 Location of connectors and main ICs on Raspberry Pi 3 34
3.9 Thermal performance of similar circuit to Raspberry Pi 35
3.10 CAN Transceiver connection to CAN Bus 35
3.11 CAN Transceiver wiring with twisted pair cables 36
3.12 Connection Pins of the MCP2515 CAN Transceiver 36
3.13 Arduino UNO and MCP 2515 Connection diagram 37
3.14 System Componects Layout 38
3.15 Data Flow Diagram 39
3.16 Increase of Python's use as a programming language 40
3.17 Monitoring of Unprocessed CAN Data in a terminal window 42
3.18 Message Identifier Breakdown 44
3.19 Message ID F004 Breakdown 45
3.20 Difference in Endian Byte Architecture 47
3.21 Data Representation in Real-Time 51
3.22 Data Representation in Real-Time 51
3.23 Data representation of all parameters monitored 52

4.1 HIPPO-2 Control Interface 55
4.2 Arduino, MCP and RTC connections during the experiment 55

4.3 Comparison of measurements, Experiment No.1 - Engine's Fuel Consumption Rate . . . 56
4.4 Comparison of measurements, Experiment No.2 - Engine's Fuel Consumption Rate . . . 57
4.5 Comparison of measurements, Experiment No.1 - Engine's Operating Speed 57
4.6 Comparison of measurements, Experiment No.2 - Engine's Operating Speed 58
4.7 Comparison of measurements, Experiment No.1 - Engine's Actual Percent Torque . . . 58
4.8 Comparison of measurements, Experiment No.2 - Engine's Actual Percent Torque . . . 59
4.9 Comparison of measurements, Experiment No.1 - Engine's Intake Manifold Temperature . . . 60
4.10 Comparison of measurements, Experiment No.2 - Engine's Intake Manifold Temperature . . . 60
4.11 Comparison of measurements, Experiment No.1 - Engine's Intake Manifold Pressure . . . 61
4.12 Comparison of measurements, Experiment No.2 - Engine's Intake Manifold Pressure . . . 61

List of Tables

2.1 CAN Bus length in relation to transmission rate 16

3.1 CAN J1939 Data Link Parameters Reference List 43
3.2 Engine Fuel Economy Parameters 48
3.3 Hexadecimal to Binary Conversion Table 49
3.4 J1939 Parameter Reference Table 49
3.5 Turbo Boost Pressure Transmission Rate 50
3.6 Engine Information Broadcasted upon user request 53

Abbreviations

Abbreviation	Description
NTUA	National Technical University of Athens
HIPPO	Hybrid Integrated Propulsion POwertrain
CAT	Caterpillar trade mark
EM	Electric Motor
EB	Electric Brake
CAN	Controller Area Network
RPM	Rotations Per Minute
SBC	Single Board Computer
ECU	Engine Control Unit
PGN	Parameter Group Number
PGL	Parameter Group Label
DLC	Data Length Code
PDU	Protocol Data Unit
RTC	Real Time Clock
OSI	Open System Interconnection
AMS	Alarm Monitoring System
ECR	Engine Control Room
SPI	Serial Peripheral Interface
SRR	Substitute Remote Request
RTR	Remote Transmission Request
IDE	Identifier Extension
SOF	Start Of Frame
EOF	End Of Frame

Chapter 1

Introduction

1.1 Experimental Engine Testbed

The latest installation at the Laboratory of Marine Engineering is the HIPPO-2 testbed. The acronym HIPPO stands for Hybrid Integrated Propulsion Powertrain. The system was installed with the goal of giving the laboratory more in-depth insight into hybrid power plant operation. HIPPO-2 is comprised of the following components:

- Caterpillar 9.3 Internal Combustion Diesel Engine with a power output of 261 kW and Tier 4 emission capability. The engine is equipped with an Integrated Particulate filter, Selective Catalytic Reduction Unit Oxidation Catalyst and Amonia Trap.
- ABB Dynamometer (315kW) with an AC induction motor.
- ABB Electric motor/generator (90kW) with an AC induction motor.

The primary propulsion is provided by the 261 kW CATERPILLAR 6-cylinder, 9.3-liter diesel engine, whose maximum power output is achieved at 1800-2200 rpm as shown in Figure 1.2. The diesel engine is fitted with advanced emission reduction technology, Exhaust Gas Re-circulation (EGR) and a Selective Catalytic Reactor (SCR) NOx abatement system. The electric motor/generator is a AC asynchronous-induction 3-phase motor, with a rated power of 90 kW. A frequency inverter unit enables the torque output regulation of the electric motor under closed loop control. Mechanical load is applied to the system by ways of an electric dynamometer, which is a 315 kW AC asynchronous-induction 3-phase motor. In this setup, the thermal and electric engine provide mechanical power simultaneously, with identical rotational speeds. All components that comprise the HIPPO-2 system can be operated either in torque- or speed-control mode. A layout of the testbed is shown in Figure 1.1

For experimentation and testing purposes we are able to apply various loading profiles with this setup. The mechanical load, applied by the dynamometer has the purpose of simulating the vessel's propeller demand, by using variable speed and torque, or a generator's loading profile with changing load at constant speed, in the same way that a governor would adjust the engine's demand.

1.2 Scope of Study

This project studies the signal transmittance and control of the HIPPO-2 experimental engine testbed. The connection between our communications system and the testbed is based on the CAN protocol. The sensor information transmitted from the engine testbed

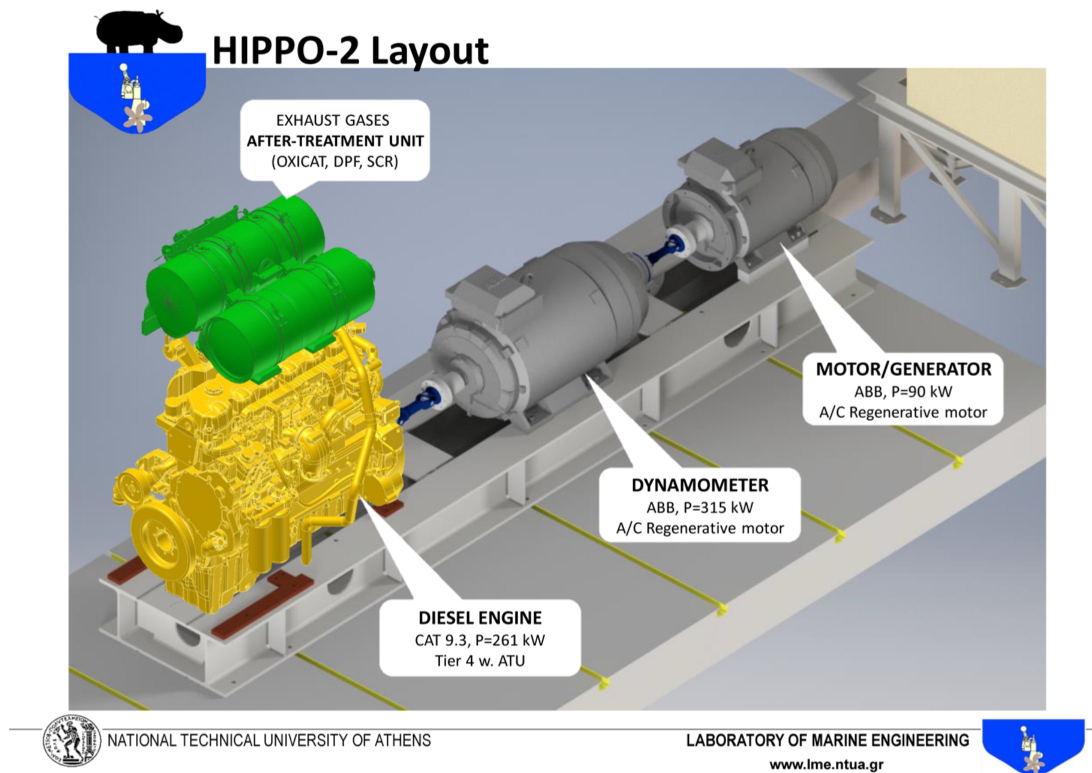


Figure 1.1: 3-D layout of the hybrid test bed HIPPO-2.

Figure 1.2: Caterpillar C 9.3 Electronic Diesel Engine

includes, among others, Engine Speed, Torque, Fuel Consumption and many other parameters. To achieve communication, data logging and display of the data we are using a system comprised of a micro-controller, CAN transceiver, digital date and time clock and

single board computer.

In this project we are describing the procedure of designing, programming, testing and implementing this communications and control system. The main objective of this diploma project, i.e. the communication and control of the HIPPO-2 installation, was achieved through the following steps:

- Understanding the CAN-OPEN and J1939 communications protocols.
- Connecting to the testbed and acquiring of the analog signals transmitted by the engine
- Segregating and decoding the signal based on the desired parameters that we wanted to monitor and control
- Storing the decoded data that was acquired for further analysis
- Plotting the desired parameters

1.3 Remote Monitoring Systems

Smart tracking and monitoring devices can now be found in all kinds of industrial assets, heavy equipment, industrial machines, reefers, trailers, containers, and more. Some estimates gather that 8.4 billion devices of all kinds connected are in use today, with the number expected to climb to 20 billion devices by 2020.

In recent years we have seen the introduction and rapidly increasing adaptation of ship remote monitoring systems on the market. Many companies offer configurations that monitor the vessel's operational data and transmit it to the technical superintendents and owners who can assess the ships performance and overall condition over time or compare the performance of sister vessels.

Until now, performance data was monitored through Noon Reports, performance evaluation reports or when specifically requested from the vessel. Usually that meant having information for one point in the day, or a summary of that days figures but not a continuous string of data to accurately represent the operation throughout the voyage.

Utilizing this new technology, users can access diagnostic and efficiency reports and configure the system to send alarms when irregularities are detected, thus minimizing the need for constant monitoring. Alerts include engine performance, fuel consumption, fuel burn rate, RPM, fuel temperature, exhaust temperature, oil pressure, boost, throttle and more. In advanced applications, the system automatically collects vessel data in near-real-time and sends it to a portal where it is remotely accessible. The operation of such monitoring systems is greatly aided by advances in telecommunications that make transmitting large amounts of data using satellite communications cost effective. On average two gigabytes of data may be downloaded from a fully equipped vessel that includes sensors for all its major components

Remote Monitoring Systems can maximize on-board resources, since processes that require human intervention, such as walkthroughs, inspections and data collection and entry, are automatically executed by the software to reduce human error and ensure access to accurate data for efficient planning, troubleshooting and problem-solving. The system helps anticipate and resolve issues before they occur. It also minimizes manual processes, allowing fleet owners to use crews more efficiently, potentially saving several hours per day of their time and reduce the number of technical personnel on-board. It can provide historical data to identify patterns and backtrack any malfunctions so that they can be closely investigated and prevent their re-occurrence.

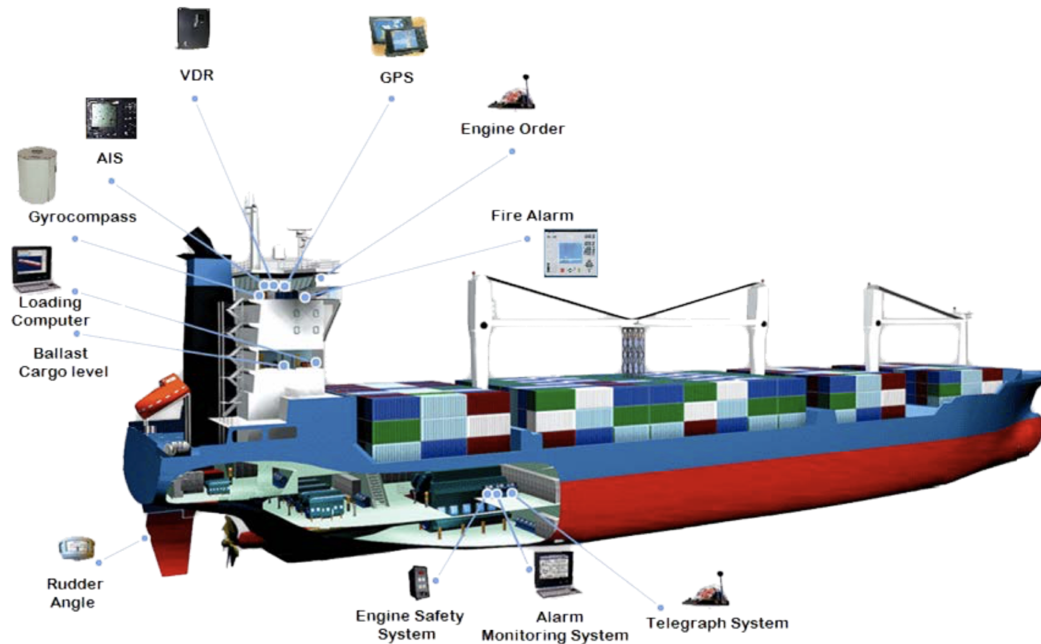


Figure 1.3: Vessel Remote Monitoring System Data Collection Points

Such systems can also help to improve efficiencies and reduce operational costs by monitoring engine status and hours of operation, and by notifying operators of upcoming service. Specific alerts can be set, so that operators are automatically informed when a parameter exceeds a certain value, reducing the need to constantly monitor the system. This is especially helpful in cases where a designated performance monitoring department in not available to constantly operate the monitoring software. A remote vessel monitoring system can also help reduce downtime caused by failure or unplanned engine maintenance, which can have a great cost on shipowners. In addition, it can feature powerful fuel reporting and analytics to help companies optimize fuel consumption. The system can enable managers assess how a vessel is performing compared to the rest of the fleet or the manufacturer's data . This can help to compare the performance of sister vessels as well as their deviation from sea trials and equipment shop tests. Finally, by having a complete image of the vessel's performance and specifically fuel consumption, owners can better describe the vessel to charterers, avoiding in the process speed and consumption claims that may lead to profit loss.

Most often Remote Monitoring Systems are installed in the vessel's bridge, cargo holds and engine control room as shown in Figure 1.3 & 1.4. From the bridge they may gather data regarding the vessel's speed, position, course as well as weather measurements, by connecting to the center console. Data is also collected from the water ingress system and sensors in the cargo holds measure humidity and temperature to ensure that the cargo is being stored in the correct conditions. From the ECR they can collect all data that the AMS (Alarm Monitoring System) receives, from various sensors installed or they may be connected directly to the propulsion system and the Diesel Generators.

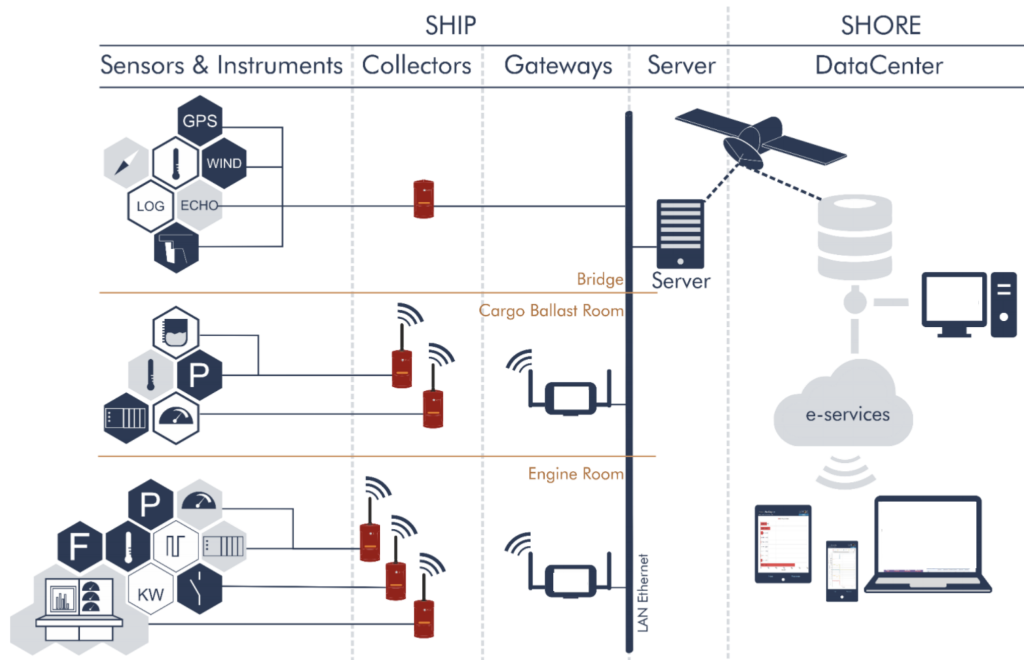


Figure 1.4: Vessel Remote Monitoring Architecture

Chapter 2

Communication Protocols

2.1 Introduction to CAN

In order to achieve communication between its various components and sensors as well as external devices, the experimental engine testbed in the LME uses the CAN protocol. CAN, also known as Controller Area Network is a serial communication bus designed for industrial and automotive applications, originally developed for the automotive industry to replace the complex wiring harness with a two-wire bus. It is a message-based protocol used for communication between multiple devices. When multiple CAN devices are connected together, the connection forms a network acting like our central nervous system allowing any device to speak with any other component in the node, also known as ECUs or Electronic Control Units. Unlike a traditional network such as USB or Ethernet, CAN does not send large blocks of data point-to-point from node A to node B under the supervision of a central bus master. In a CAN network, many short messages like temperature or RPM are broadcast to the entire network, which provides for data consistency in every node of the system. Modern CAN bus systems, such as cars or other automotive vehicles may have as many as 70 ECUs, including but not limited to engine control, airbags, lights and locks.

A brief history of the CAN bus is outlined below:

- Pre CAN: Car ECUs relied on complex point-to-point wiring
- 1986: Bosch developed the CAN protocol as a solution
- 1991: Bosch published CAN 2.0 (CAN 2.0A: 11 bit, 2.0B: 29 bit)
- 1993: CAN is adopted as international standard (ISO 11898)
- 2003: ISO 11898 becomes a standard series (11898-1, 11898-2, etc.)
- 2012: Bosch released the CAN FD 1.0 (flexible data rate)
- 2015: The CAN FD protocol is standardized (ISO 11898-1)
- 2016: The physical CAN layer for data-rates up to 5 Mbit/s standardized in ISO 11898-2

Today, the CAN protocol is standard in practically all vehicles (cars, trucks, buses, tractors, etc.) as well as ships, planes, EV batteries, industrial machinery and more. Further, more exotic cases include drones, radar systems, submarines or even prosthetic limbs.

The CAN Bus communications system provides several advantages over other protocols, the most significant of which are listed below:

1. **Low Cost**

   To understand the efficiency of the system we simply need to take a look at one of its first applications, a BMW 850. Implementation of CAN bus architecture reduced the length of wiring in the BMW 850 by 1.25 miles, which in turn reduced its weight by well over 100 pounds. Based on the current cost of copper wiring, the total cost savings from the saved materials would amount to nearly 600 USD. Not only that, but the speed of communication was increased, with signal rates ranging from 125 kbps to 1 Mbps. Low cost of implementation is one of the main reasons that we are seeing widespread adoption of the CAN bus protocol. Less wiring means less labor and lower material costs for embedded engineers.

2. **Built-in Error Detection.**

   While each node is capable of sending and receiving messages, not all nodes can be communicating at once. The CAN bus protocol uses a technique called lossless bitwise arbitration to resolve these situations and determine which node should be given "priority" to communicate its message first. Error handling is built into the CAN protocol, with each node checking for errors in transmission and maintaining its own error counter. Nodes transmit a special Error Flag message when errors are detected and will destroy the offending bus traffic to prevent it from spreading through the system. Even the node that is generating the fault will detect its own error in transmission, raising its error counter and eventually leading the device to "bus off" and cease participating in network traffic. In this way, CAN nodes can both detect errors and prevent faulty devices from creating useless bus traffic.

3. **Robustness**

   Durability and reliability are key areas of concern when choosing a communication protocol for deployment in any engineering project. CAN high-speed bus lines are highly resistant to electrical disturbances, and the CAN controllers and transceivers that communicate with electronic devices are available in industrial or extended temperature ranges, thus withstanding the demanding conditions that exist around internal combustion engines.

4. **Speed**

   High Speed CAN offers signal transfer rates of between 40 kbps and 1 Mbps, depending on the length of the cable, far surpassing the speed delivered by systems that came before it.

5. **Flexibility**

   The CAN bus protocol is known as a message-based communication protocol. In this type of protocol, nodes on the bus have no identifying information associated with them. As a result, nodes can easily be added or removed, through a process called hot-plugging, without performing any software or hardware updates on the system. This feature makes it easy for engineers to integrate new electronic devices into the CAN bus network without significant programming overhead and supports a modular system that is easily modified to suit any specs or requirements.

2.1.1 Physical Layer

The Physical Layer is the basic hardware required for a CAN network, i.e. the ISO 11898 electrical specifications. It converts 1's and 0's into electrical pulses leaving a node, then back again for a CAN message entering a node. Although the other layers may be

implemented in software or in hardware as a chip function, the Physical Layer is always implemented in hardware.

A CAN Network will consist of only two wires CAN High and CAN Low for bi-directional data transmission, in a form of a twisted pair, where each node terminates on each end with 120 Ohm resistors, as shown in Figure 2.1. The twin axial cables used for this application need to meet rigorous electrical and environmental performance requirements. Namely, they need to have below features:

- Light weight and small size

  Especially for the automotive industry, weight reduction in a key objective and a very important parameter in the vehicles performance and fuel efficiency.

- Broad temperature range of -65 to 200 degrees Celsius

  The area on and around an internal combustion engine can reach very high temperatures and any additional hardware has to be able to withstand this demanding environment for long periods of operation.

- Low capacitance

  Capacitance affects the signal level and is frequency-dependent. The higher the frequency, the greater the reactance caused by the capacitance and the greater the signal loss. High-frequency loss from the cable gradually becomes noticeable as it results in noise generated in the system . Raising the source impedance or increasing the length of the cable increases the loss. With an increase in length of the Bus, there is a loss in Bit rate. The effect of capacitance on the Bus length and the transmission rate can be seen in Table 2.1.

- High data rates

  Data transmittance can reach very high values for applications using CAN Bus. As mentioned above data may be transmitted at a rate of up to 1 Mbps.

- Excellent shop handling

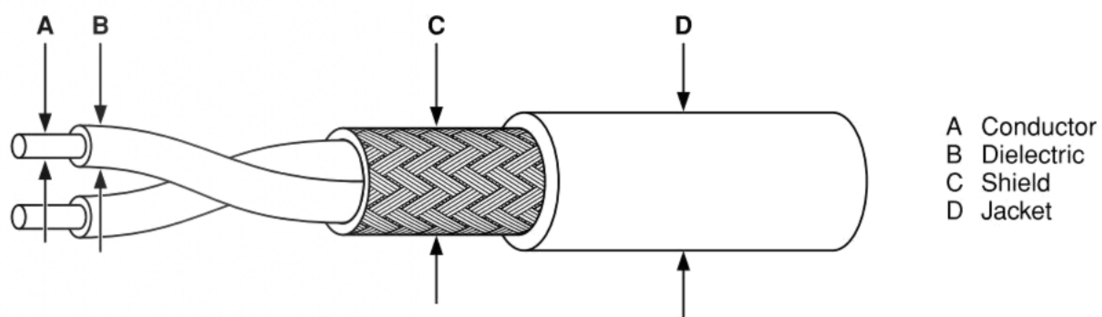


Figure 2.1: Cheminax CAN Cable

The pair of twisted wires are copper cables with common ground connection. They are differential, meaning that the transmission occurs by the potential difference between them, which is where CAN derives its robust noise immunity and fault tolerance. Balanced differential signaling reduces noise coupling and allows for high signaling rates over twisted-pair cable. Balanced means that the current flowing in each signal line is equal but opposite in direction, resulting in a field-canceling effect that is a key to low noise emissions. The

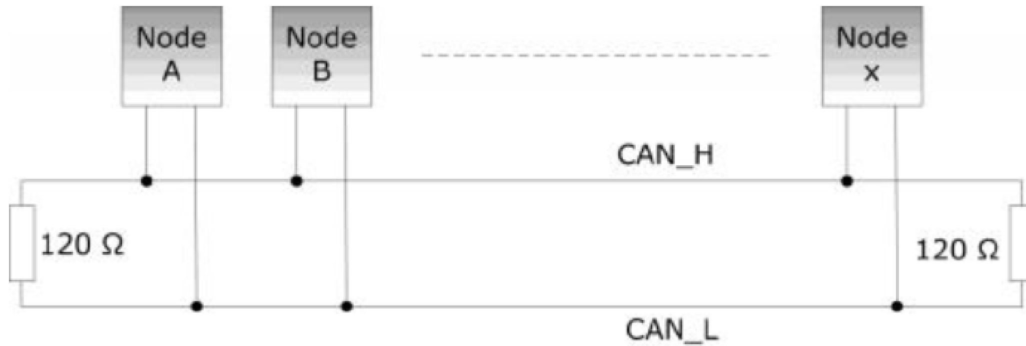


Figure 2.2: CAN Physical Layer

Table 2.1: CAN Bus length in relation to transmission rate

Bus length	Bit rate tradeoff
1M bit/sec	40 meters
500k bit/sec	100 meters
250k bit/sec	200 meters
125k bit/sec	500 meters

use of balanced differential receivers and twisted-pair cabling enhance the common-mode rejection and high noise immunity of a CAN bus.

The voltage between the wires depends on the signal transmitted and is 3.5 V for the CAN High wire and 1.5 V for the CAN Low when "0" is transmitted, meaning when the bus is on the dominant state, while the voltage when "1" is transmitted, meaning when the bus is in the quiescent recessive state is 2.5 V for both wires as shown in Figure 2.3. Typically the communication speed for CAN ranges from 50 Kbps to 1Mbps and the distance can range from 40 meters at 1Mbps to 1000 meters at 50 kpbs. Thus is it clear that depending on our application and the length of wiring needed there is a restriction on our networks speed. Vice versa, if our application requires a minimum speed for data transmittance, there is a maximum length restriction that applies. The CAN bus length and data transmission speed correlation is shown in Table 2.1. Whereas there is not a specified maximum number of nodes that may be connected on a CAN bus, there is a limitation imposed by the driving capability of a CAN transceiver. Typically we may see a connection of 64 nodes. It is worth noting that all nodes should support the same data-rate and bit-timing settings. Figure 2.2 shows how nodes a connected on the CAN Bus.

2.1.2 CANopen Protocol

CANopen is a CAN-based communication system. It comprises of higher-layer protocols and profile specifications. CANopen has been developed as a standardized embedded network with highly flexible configuration capabilities. It was designed originally for motion-oriented machine control systems, such as handling systems. Today it is used in various application fields, such as medical equipment, off-road vehicles, maritime electronics, railway applications, or building automation. CANopen unburdens the developer

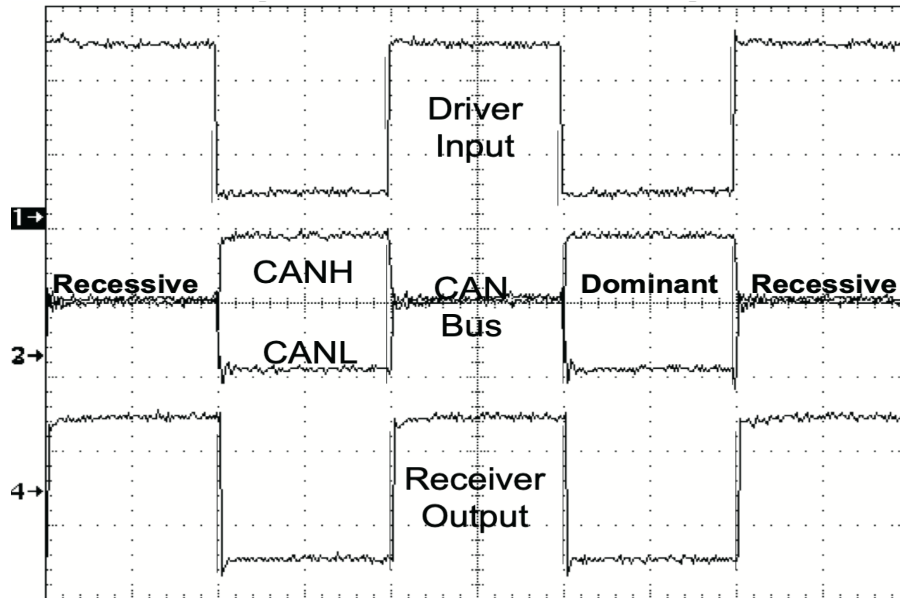


Figure 2.3: CAN Dominant and Recessive Bus States

from dealing with CAN hardware-specific details such as bit timing and acceptance filtering. It provides standardized communication objects (COB) for time-critical processes, configuration as well as network management data.

In terms of the Open Systems Interconnection or OSI communication systems model, CAN covers the first two levels: the physical layer and the data link layer. The physical layer defines the lines used, voltages, high-speed nature, etc. The data link layer includes the fact that CAN is a frame-based messages protocol. CANopen covers the top five layers: network (addressing, routing), transport (end-to-end reliability), session (synchronization), presentation (data encoded in standard way, data representation) and application. The application layer describes how to configure, transfer and synchronize CANopen devices.

CANopen provides several communication objects, which enable us to implement desired network behavior into a device. With these communication objects, we can offer devices that can communicate process data, indicate device-internal error conditions or influence and control the network behavior, similar to the system we have designed. As CANopen defines the internal device structure, we have the ability to know exactly how to access a CANopen device and how to adjust the intended device behavior.

2.1.3 CAN J1939 Protocol

CAN J1939 standard is a protocol maintained by the Society of Automotive Engineers (SAE) and is very widely used in recent years, having a vast majority of applications especially in the automotive field. It is now considered the industry standard. J1939 offers a standardized method of communication across ECUs. The protocol has multiple layers. For the purposes of this project we will mainly deal with the following:

- **J1939-11 :** Physical layer
- **J1939-15 :** Reduced Physical Layer, utilizing a bit-rate of 250Kbits/s and an Unshielded Twisted pair of wires
- **J1939-21 :** Data Link Layer

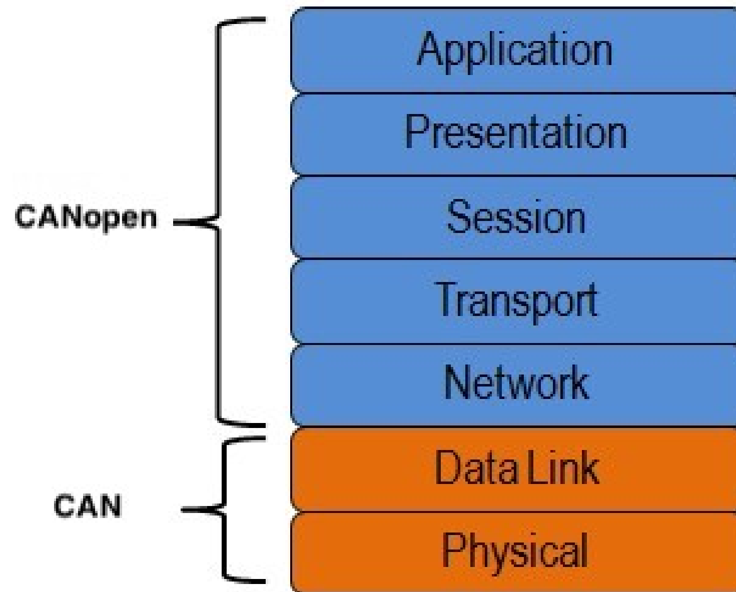


Figure 2.4: CAN Open Protocol

- **J1939-31 :** Network Layer
- **J1939-71 :** Vehicle Application Layer
- **J1939-75 :** Application Layer Generator Sets and Industrial
- **J1939-73 :** Application Layer Diagnostics
- **J1939-81 :** Network Management

For our application we shall be focusing on the Data Link and Vehicle Application Layers, J1939-21 and J1939-71 respectively. More specifically, the Physical Layer is characterized by a transmittance rate of 250 kbps, the use of a twisted pair of wires which are shielded and end on 120 Ohm resistors. Furthermore the bus is linear and has a maximum length of 40 meters. In addition the maximum amount of nodes that may be connected to the network are 30 for the Physical Layer and 10 for the Reduced Physical Layer. The nodes are connected to the bus via stubs that may have a maximum length of 1 meter or 3 meters depending on whether we are using the protocol for the physical Layer or the Reduced Physical layer respectively.

Difference Between CAN J1939 and Standard CAN Bus

While the standard CAN message frame uses an 11-bit message identifier (CAN 2.0A), which is sufficient for the use in regular automobiles and any industrial application, J1939 is a higher layer protocol which implements an enhanced 29-bit message identifier. The ISO 11898 amendment for an extended frame format (CAN 2.0B) was introduced in 1995.

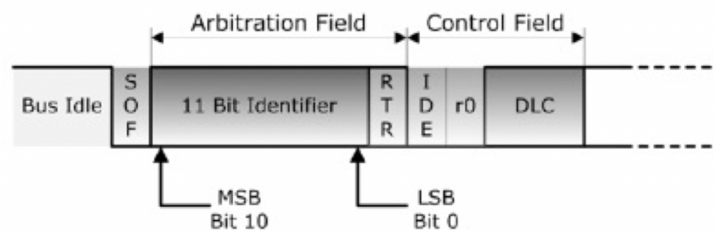
The 29-bit message identifier found in CAN J1939 consists of the regular 11-bit base identifier and an 18-bit identifier extension. The distinction between CAN base frame format and CAN extended frame format is accomplished by using the IDE bit inside the Control Field as shown in Figure 2.5. A low (dominant) IDE bit indicates an 11-bit message identifier, while a high (recessive) IDE bit indicates a 29-bit id. In terms of

flexibility and expansion of the CAN network an 11-bit identifier (standard format) allows a total of 2^{11} (= 2048) different messages. At the same time, a 29-bit identifier (extended format) allows a total of 2^{29} (= 536+ million) messages.

In is important to note that both formats, Standard (11-bit message ID) and Extended (29-bit message ID) may co-exist on the same CAN bus. During bus arbitration, the standard 11-bit message ID frame will always have a higher priority than the extended 29-bit message ID frame with an identical 11-bit base identifier and thus gain bus access.

The Extended Format has some trade-offs for the increased capability that it provides. Bus latency time is longer (minimum 20 bit-times), messages in extended format require more bandwidth (about 20 %), and the error detection performance is reduced (because the chosen polynomial for the 15-bit CRC is optimized for frame length up to 112 bits).

Standard Format 11 Bit message identifier:



Extended Format: 29 Bit message identifier:

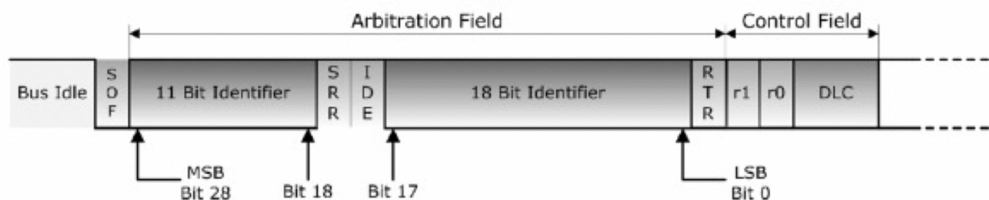


Figure 2.5: Difference Between the Standard and Extended Identifier

The 29-bit message identifier consists of the regular 11-bit base identifier and an 18-bit identifier extension. Between the SOF (Start of Frame) bit and the end of the 11-bit (base) message identifier, both frame formats, Standard and Extended, are identical.

Following the 11-bit base identifier, the Extended Format uses an (always recessive) SRR (Substitute Remote Request) bit, which, as its name implies, replaces the regular RTR (Remote Transmission Request). The following IDE (Identifier Extension) bit is also kept at a recessive level. With the use of a recessive SRR plus a recessive IDE bit it is guaranteed that standard message frames (11-bit identifier) will always have higher priority than extended message frames (29-bit identifier) with identical 11-bit base identifier.

2.2 CAN Message Architecture

The engine's ECU transmits data using the J1939 protocol, which, as mentioned above, uses CAN version 2.0B, meaning it has an extended 29 bit identifier. All CAN Open messages are structured as show in the layout in Figure 2.6.

Figure 2.6: CAN Message Breakdown

Breaking down the diagram in Figure 2.6 we will go through each part of a CAN message:

- **Start Of Frame:**

  The single dominant start of frame (SOF) bit marks the start of a message, and is used to synchronize the nodes on a bus after being idle.

- **Message ID:**

  Establishes the priority of the message. Lower binary values have a higher priority. On the protocol used for our application the length is the extended 29 bits. Further analysis of the CAN identifier can be found in a following chapter where we break down the filtering of incoming data.

- **Remote Transmission Request (RTR):**

  The single remote transmission request bit is dominant when information is required from another node. All nodes receive the request, but the identifier determines the specified node. The responding data is also received by all nodes and used by any node interested. In this way, all data being used in a system is uniform.

- **Control Field:**

  Specifies the number of bytes that the data following it will have.

- **Cyclic Redundancy Check (CRC):**

  CRC contains a 16 bit redundancy code. CAN data frames and remote frames contain a safeguard based on a CRC polynomial: The transmitter calculates a check sum from the transmitted bits and provides the result within the frame in the CRC field. The receivers use the same polynomial to calculate the check sum from the bits as seen on the bus-lines. Afterwards, the self-calculated check sum is compared with the received on. If it matches, the frame is regarded as correctly received and the receiving node transmits a dominant state in the ACK slot bit, overwriting the recessive state of the transmitter. In case of a mismatch, the receiving node sends an Error Frame after the ACK delimiter.

- **Data:**

  This section comprises of the value contained within this specific ID. Up to 64 bits of application data may be transmitted. Given the large size, the data frame contains information for multiple parameters. The measurements which we are monitoring are 8 bits in length and contained within this frame.

- **Acknowledge Field (ACK):**

  Every node receiving an accurate message overwrites this recessive bit in the original message with a dominate bit, indicating an error-free message has been sent. Should a receiving node detect an error and leave this bit recessive, it discards the message and the transmitting node repeats the message after re-arbitration occurs. In this way, each node acknowledges (ACK) the integrity of its data. ACK is 2 bits in length, with one being the acknowledgment bit and the second being a delimiter.

- **End Of Frame (EOF):**

  EOF is a 7-bit field that marks the end of a CAN frame (message) and disables bit-stuffing, indicating a stuffing error when dominant. When 5 bits of the same logic level occur in succession during normal operation, a bit of the opposite logic level is stuffed into the data.

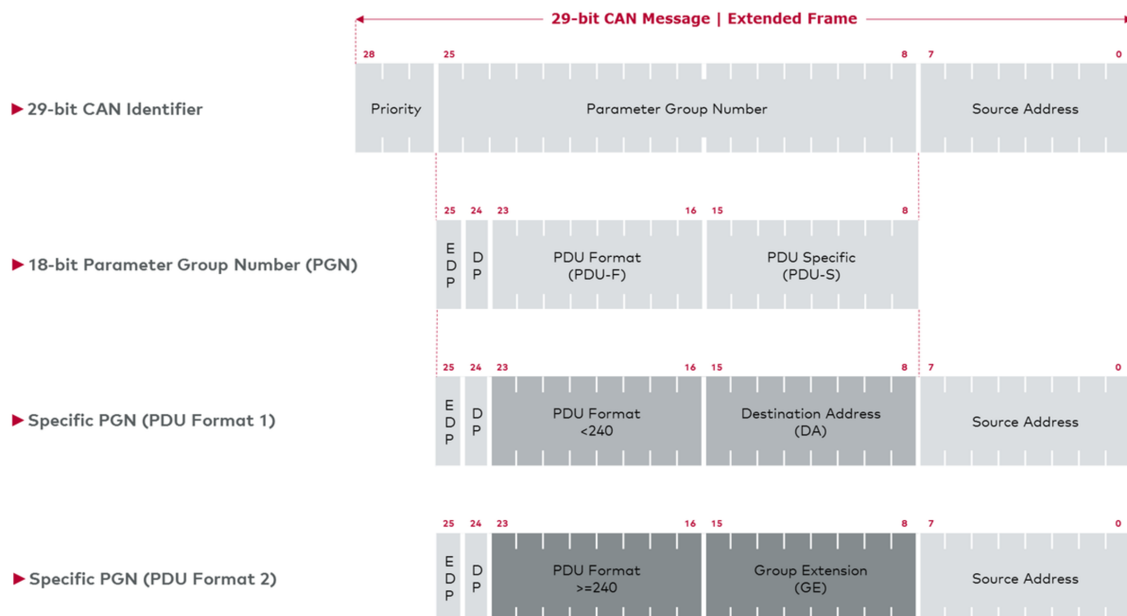


Figure 2.7: J1939 CAN 2.0 Identifier

In the J1939 documentation, messages are identified through their Parameter Group Number or PGN as we will further analyze below. PGNs comprise 18 of the 29 total bits of the identifier, as is shown in Figure 2.7. Certain PGNs are reserved for proprietary use by each respective manufacturer, ranging from value 00FF00 to 00FFF. Byte 0xFF reflects non available data, while byte 0xFE reflects an error.

2.2.1 Error Checking

In addition to arbitration, the data link layer, layer 2 of the CAN OSI model, shown in Figure 2.4 also contributes to the robustness of the overall CAN system, in part to its

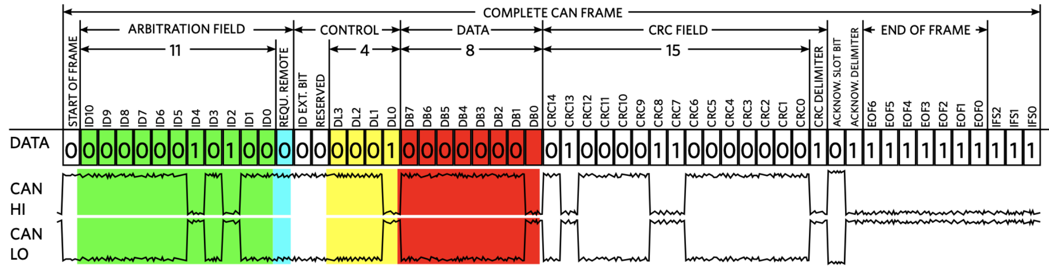


Figure 2.8: CAN Message Data-Frame Format

abundant error-checking procedures. In this layer, the frame message is repeatedly checked for accuracy and errors, incorporating five methods of checking: two at the message level and three at the bit level.

If a message is received with errors, an error frame is sent out. This forces the transmitting node to resend the message until it is received correctly. However, if a faulty node hangs up a bus by continuously repeating an error, its transmit capability is removed by its controller after an error limit is reached.

The error frame consists of two different fields: the error flag and the error delimiter. From a message- level perspective we have the cyclic redundancy check and the frame check.

- Cyclic Redundancy Check (CRC) safeguards the information in the frame by adding redundant check bits at the end of transmission, which are then checked on the receiving side. If they do not match, then a CRC error has occurred.

- Frame or Form check looks for fields in the message which must always be recessive bits and verifies the structure by checking the bit fields against the fixed format and frame size of SOF, EOF, ACK, and CRC delimiter bits. If a dominant bit is detected, an error is generated.

From a bit-level perspective, there are three checks for errors: acknowledgment, bit monitoring, and bit stuffing.

- Acknowledgment errors are detected when the transmitter does not read a dominant ACK bit (0). This indicates a transmission error detected by the recipients, which means either the ACK was corrupted or there were no receivers.

- Bit monitoring checks the bus level for each node for sent and received bits. If a data bit (not arbitration bit) is written onto the bus and its opposite is read, an error is generated. The only exceptions to this are with the message identifier field which is used for arbitration, and the acknowledge slot which requires a recessive bit to be overwritten by a dominant bit.

- Bit stuffing is a method that "stuffs" or inserts an extra opposite bit when five of the same bits occur in succession. The opposite bit helps to differentiate error frames and EOF bits. On the receiving side, the extra bit is removed. If the sixth bit is the same as the previous five, then an error is detected by all CAN nodes and error frames are sent out. The original message will need to be re-transmitted and pass through arbitration if there is contention on the line. Stuffing ensures that rising edges are available for on-going synchronization of the network. Stuffing also ensures that a stream of bits are not mistaken for an error frame, or the seven-bit inter-frame

space that signifies the end of a message. Stuffed bits are removed by a receiving node's controller before the data is forwarded to the application. With this logic, an active error frame consists of six dominant bits—violating the bit stuffing rule. This is interpreted as an error by all of the CAN nodes which then generate their own error frame. This means that an error frame can be from the original six bits to twelve bits long with all the replies. This error frame is then followed by a delimiter field of eight recessive bits and a bus idle period before the corrupted message is retransmitted. It is important to note that the retransmitted message still has to contend for arbitration on the bus.

2.3 CAN Bus Compared to Other Communication Protocols

For the purpose of achieving the serial communication between an engine's ECU and a microcontroller there are many protocols that could be used. In similar applications, especially older ones, we can see the use of both RS 232 and RS 485 serial protocols which are popular standards in fieldbus systems. However, the CAN bus provides several advantages over the other protocols.

2.3.1 CAN & RS 232

Although both the CAN Bus and RS 232 are serial protocols, they are very different from each other mainly on the physical and transfer layers.

Physical Layer

The basic RS232 can be accomplished by the use of 2 wires, but can also be extended to the complex multi wire protocol. On the physical layer, the standard defines a logic 1 with a voltage between –3 and –25 V and a logic 0 as a voltage level between +3 and + 25 V . Signal levels are commonly referred to as a mark for logic 1 and a space for logic 0. Voltages between $\pm 3V$ are invalid, providing a huge noise margin for the interface. Noise voltages in this range are rejected. In common practice, logic 0 and 1 levels are typically as low as $\pm 5V$ and as high as ± 12 or $\pm 15V$. The transmitter and receiver configurations are single-ended (not differential) with a ground reference. The fact that the configuration is not differential makes the RS232 susceptible to noise and cannot be used at high baud rates and large distances. It also is a single master protocol.

On the other hand, the physical layer for CAN consists of 3 wires CAN-H, CAN-L and ground. The physical value of the signal is resolved by the differential voltage between CAN-H and CAN-L. When the difference is maximum(dominant) its resolved as a logic level 0 and when the difference is minimum(recessive) its resolved as a logic level 1. With this simple technique we can ensure high susceptibility to noise as it will affect both wires at the same time and hence the difference would be the same. This enables high baud rate up to 1Mbps with higher redundancy.

Transfer Layer

RS 232 is a master slave protocol. The master initiates a communication and the slave in turn responds. It can be either full duplex or half depending on how we are using it, make it a very simple protocol.

CAN, however is really complex. It is a multi master protocol. This means that we have the ability to connect multiple nodes on a CAN bus and expect each of them to

be the master at any point. But at a given instant of time, only one node can be a master. This is achieved by the ID arbitration, meaning that each node is responsible for sending a certain number of messages with a particular ID. When multiple masters start transmitting at the same time, the master who is transmitting a higher priority ID wins the bus and becomes the master, this is where the priority byte plays a major role. After the transmission of this message, the arbitration process again kicks in to select the next master.

Summarizing, CAN is a really powerful multi-drop communications protocol which helps connect multiple communication nodes at a higher baud rate with more noise efficiency, which is the main difference between RS232 and CAN.

2.3.2 CAN & RS 485

RS 485 and CAN bus have very similar characteristics, with the latter however being better suited for engine data transfer applications. To begin with, both protocols feature differential outputs. The RS-485 output is a classical differential signal where one signal is the inverted, or mirror, version of the other. (Output A is the non-inverting line and output B is the inverting line.) The differential range from +1.5V to +5V is a '1' or mark and -1.5V to -5V is a '0' or space. The area between -1.5V and +1.5V is undefined.

For CAN, the output differential is slightly different where the two outputs, CANH and CANL data lines, are a reflection of each other as depicted and represent opposite logic. In the dominant state (a zero bit, used to determine message priority), CANH-CANL are defined to be logic '0' when the voltage across them is between +1.5V and +3V. In the recessive state (a 1-bit and the state of the idle bus), the driver is defined to be logic '1' when differential voltage is between -120mV and +12mV, or when it is near zero. For the receiver side, the RS-485 standard defines the input differential to be in between ±200 mV to +5V. For CAN, the input differential signal is between +900mV and +3V, while the recessive mode is in between -120mV and +500mV. When the bus is idle or when it's not loaded, the transceiver is in a recessive state where CANH and CANL must be between 2V and 3V. Both RS-485 and CAN have room for margin in applications where the signal can be attenuated by the quality (shielded or unshielded) or length of the cables, which may affect the capacitance of the overall system.

Figure 2.9: RS 485 and CAN Bus Output Differential

Additionally, both standards have termination resistors of the same 120 Ohm value at the ends of the network, to match the characteristic impedance of the transmission line and avoid reflection. Another feature common in both CAN and RS-485 transceivers is fault protection. Fault-protected devices have an internal overvoltage circuit on the driver output and receiver inputs to protect the devices from accidental shorts between a local

power supply and the data lines of the transceivers.

One of the major reasons for industrial applications to design in CAN versus RS-485 transceivers is how messages are handled on the bus. In a RS-485 system with many nodes communicating to the microprocessor, there may be instances where there are several messages sent out from multiple nodes onto a bus simultaneously that may result in a collision of messages, otherwise known as contention. When this happens, the bus state could possibly be invalid or indeterminate, causing data errors. Furthermore, contention could damage or degrade the signal performance when multiple RS-485 transceivers on the bus are in one state and one single transceiver is in the opposite state. In such a condition, the lone RS-485 would cause significant current draw that would likely cause thermal shutdown of the IC or permanent damage to the system This is where CANbus has a big advantage over the RS-485 protocol.

On the other hand, with CANbus, there is a way to resolve multiple messages on the line by way of ranking each message. Prior to bringing the system up, different faults are assigned different priorities by the system engineer. Earlier, it was mentioned that CAN had a dominant and recessive state. During contention, the message with the most consecutive dominant state is given priority and will continue to transmit, while other nodes with lower priority will see the dominant bit and stop transmission. This method is called arbitration, where the messages are prioritized and received in an order of status. A node that loses arbitration will resend its message. This continues for all nodes until there is one node left transmitting.

With CAN features such as arbitration, error-message checking, improved bandwidth, and a larger data field, it is easy to understand the widespread use of CAN bus in the industrial market. CAN is suitable for applications that require robust communications and reliability in harsh environments. CAN systems are able to prioritize the importance of frame messages and treat critical ones appropriately. Many different systems can be exposed to either electrically noisy sources or a local service personnel that may accidentally short to local supply rails.

Chapter 3

Data Acquisition

3.1 Hardware

In this chapter we will examine all the components that comprise our system and the process of integrating them all, including the connectivity between them, in order to collect and analyze data from the CAN-BUS. These include a Microcontroller, CAN Transceiver, Real Time Clock and Single Board Computer, along with wiring and peripherals.

3.1.1 Microcontroller

The Arduino UNO is a microcontroller based on the ATmega328P high performance microchip. The board is equipped with sets of digital and analog input/output (I/O) pins that may be interfaced to various expansion boards (shields) and other circuits. More specifically, it includes 14 digital I/O pins, 6 of which may be used as PWN (Pulse Width Modulation) outputs. It also includes 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, power jack and ICSP header. For this application the board is powered via the USB cable from the single board computer, though it may also be powered by a battery or directly through a power socket.

Arduino can be used to communicate with a computer, another Arduino board or other microcontrollers. The ATmega328P microcontroller provides UART TTL (5V) serial communication which can be done using digital pin 0 (Rx) and digital pin 1 (Tx), shown in the schematic diagram in Figure 3.3. An ATmega16U2 on the board channels this serial communication over USB and appears as a virtual com port to software on the computer, allowing us to collect all transmitted data and also send data of our own to the microcontroller. The ATmega16U2 firmware uses the standard USB COM drivers, and no external driver is needed. This further contributes to the ease of integration into many different kinds of set-ups. The Arduino software includes a serial monitor which allows simple textual data to be sent to and from the Arduino board. The serial monitor may provide a quick representation of the information received, however it may not be directly exported to a separate file for further analysis, and thus we have to write a separate software in order to incorporate such a function.

To set-up the serial monitor we need to define the baudrate with which the Arduino is configured in our software and the port to which it is connected. In this case, the baudrate has been set to 115200 in the Arduino IDE code and the serial port is 'cu.usbmodem14301' but may differ depending on the device.

There are two RX and TX LEDs on the Arduino board which will flash when data is being transmitted via the USB-to-serial chip and USB connection to the computer (not for serial communication on pins 0 and 1). A SoftwareSerial library allows for serial

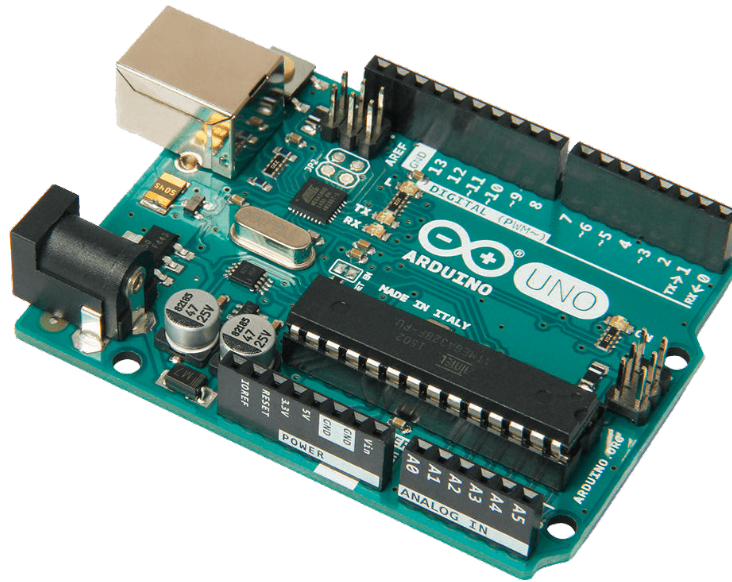


Figure 3.1: Arduino UNO Rev. 3

```
/dev/cu.usbmodem14301                                                    Send
5.3200001716,98FEE900,62 2 0 0 59 2 0 0
5.3439998626,8C000001,1 E0 15 89 0 0 0 0
5.3670001029,8C000001,1 E0 15 89 0 0 0 0
5.3909997940,98EF9117,F0 2E 80 E 19 D 1 2
5.4140000343,98F00E91,FF FF FF FF FF FF FF FF
5.4380002021,98A51781,55 55 FF FF FF FF FF FF
5.4629998207,98FEF000,FF FF FF FF FF FF FF FF
5.4860000610,8C000001,1 E0 15 89 0 0 0 0
5.5100002288,8CEC9117,11 3 1 FF FF 0 EF 0
5.5329999923,8CF02391,FF FF F1 FF FF FF FF FF
5.5580000877,98FEF200,0 0 FF FF FF FF 0 FF
5.5809998512,98EA0017,8C FE 0
5.6040000915,8C000001,1 E0 15 89 0 0 0 0
5.6279997825,8C000001,1 E0 15 89 0 0 0 0
5.6519999504,8CF00300,FF FF 0 FF FF FF FF FF
5.6760001182,8C000001,1 E0 15 89 0 0 0 0
5.6989998817,98FEDF91,FF E0 15 FF FF 66 0 FF
5.7230000495,8C000001,1 E0 15 89 0 0 0 0
5.7459998130,8CEC9117,11 3 1 FF FF 0 EF 0
5.7699999809,98FEF000,FF FF FF FF FF FF FF FF
5.7940001487,8C000001,1 E0 15 89 0 0 0 0
5.8179998397,8C000001,1 E0 15 89 0 0 0 0
5.8410000801,8CF00400,FF 83 83 0 0 FF FF FF
5.8649997711,8C000001,1 E0 15 89 0 0 0 0
5.8889999389,98FCEF00,D9 3 FF FF FF FF FF FF
5.9120001792,8C000001,1 E0 15 89 0 0 0 0
5.9369997978,8C000001,1 E0 15 89 0 0 0 0
5.9600000381,98F00F91,FF FF FF FF FF FF FF FF
5.9840002059,8CEC1791,10 1D 0 5 FF 0 EF 0
6.0069999694,8CF00300,FF FF 0 FF FF FF FF FF
6.0310001373,8C000001,1 E0 15 89 0 0 0 0
6.0539999008,8C000001,1 E0 15 89 0 0 0 0
6.0780000686,8CEA0000,AA FE 0
6.1020002365,8C000001,1 E0 15 89 0 0 0 0
Autoscroll   Show timestamp          Newline   115200 baud   Clear output
```

Figure 3.2: Arduino Serial Monitor displaying CAN data

communication on any of the Uno's digital pins. The ATmega328P also supports I2C (TWI) and SPI communication, which is the communication protocol that will be used with the MCP2515. The Arduino software includes, along with numerous others, a Wire

library to simplify use of the I2C bus.

In addition, the Arduino has 14 digital input/output pins which can be used as input or output pins by using pinMode(), digitalRead() and digitalWrite() functions in the Arduino programming software. Each pin operates at 5V and can provide or receive a maximum of 40mA current, and has an internal pull-up resistor of 20-50 KOhms which are disconnected by default. Out of these 14 pins, some pins have specific functions as listed below:

- Serial Pins 0 (Rx) and 1 (Tx): Rx and Tx pins are used to receive and transmit TTL serial data. They are connected with the corresponding ATmega328P USB to TTL serial chip.
- External Interrupt Pins 2 and 3: These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value.
- PWM Pins 3, 5, 6, 9 and 11: These pins provide an 8-bit PWM output by using analogWrite() function.
- SPI Pins 10 (SS), 11 (MOSI), 12 (MISO) and 13 (SCK): These pins are used for SPI communication.
- In-built LED Pin 13: This pin is connected with an built-in LED, when pin 13 is HIGH – LED is on and when pin 13 is LOW, its off.

Along with 14 Digital pins, there are 6 analog input pins, each of which provide 10 bits of resolution, i.e. 1024 different values. They measure from 0 to 5 volts but this limit can be increased by using AREF pin with analog Reference() function.

Analog pin 4 (SDA) and pin 5 (SCA) also used for TWI communication using Wire library. Arduino Uno has a couple of other pins as explained below:

- AREF: Used to provide reference voltage for analog inputs with analogReference() function.
- Reset Pin: Making this pin LOW, resets the microcontroller.

The Arduino provides several benefits that greatly enable us in designing and implementing our system:

- Small size. The dimensions of the board make it ideal for portable applications, that require very little space and may call for installation in different locations.
- Low power consumption. It only requires power between 7 and 20 volts. The minimal energy requirements the board has allows us to power it through a USB cable, usually connected to a portable or single board computer or via a battery, adding to its portability and convenience.
- Multiple I/O pins provide the capability to connect more than one CAN bus device, in case we need to receive signals from more than one source.
- Ease of use. The Arduino is paired with its own IDE open-source software with many pre programmed applications and a large support network.
- Large library database which is available to implement and edit very easily, suitable for most engineering applications.

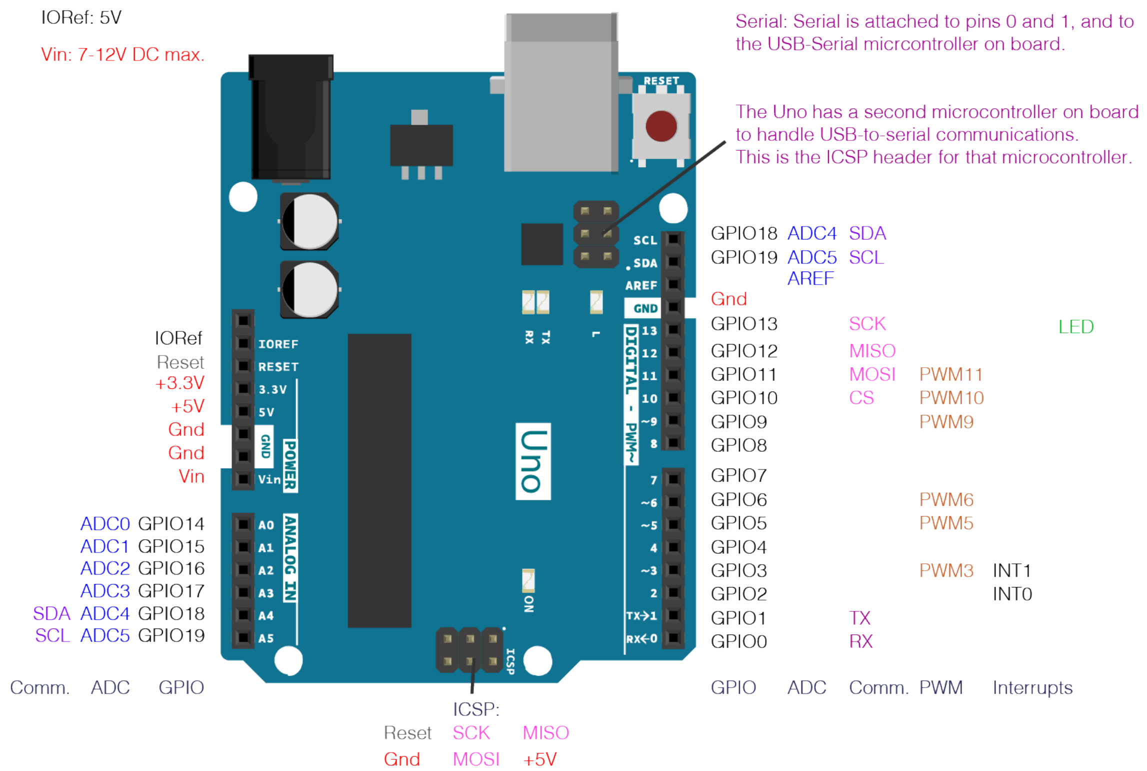


Figure 3.3: Arduino UNO Schematic Representation

3.1.2 CAN Transceiver

Given that the Arduino does not have built-in CAN communication capabilities, we have connected to it the an MCP 2515 CAN transceiver. MCP2515 is a stand-alone Controller Area Network (CAN) controller that implements the CAN specification, Version 2.0B as is used by the HIPPO-2 testbed. It is capable of transmitting and receiving both standard as well as extended data and remote frames. The MCP2515 has two acceptance masks and six acceptance filters (both for extended 29-bit messages), meaning that it is able to filter out unwanted messages, thereby reducing the host MCU's overhead, i.e. the processing load required from the microcontroller. The MCP2515 interfaces with microcontrollers (MCUs) via an industry standard Serial Peripheral Interface (SPI), which is compatible with our microcontroller. The main features of the MCP2515 are outlined below:

- Includes a high speed TJA1050 CAN transceiver
- Dimensions are 40 x 28 mm making it ideal for portable applications
- High speed 10 MHz SPI control to communicate between CAN devices
- 8MHz crystal oscillator, acting as microprocessor's frequency control
- 120 Ohm thermal resistance, as needed at the end of the CAN bus twisted pair of cables
- Data transmittance up to 1 Mb/s, capable of handling all of the traffic in the system
- Low-power CMOS technology allowing operation from 2.7V to 5.5V. 5 mA active current (typical) and 1μA standby current on Sleep Mode

- -40°C to +85°C temperature operating range makes the transceiver capable of withstanding the testbed and engine room environments

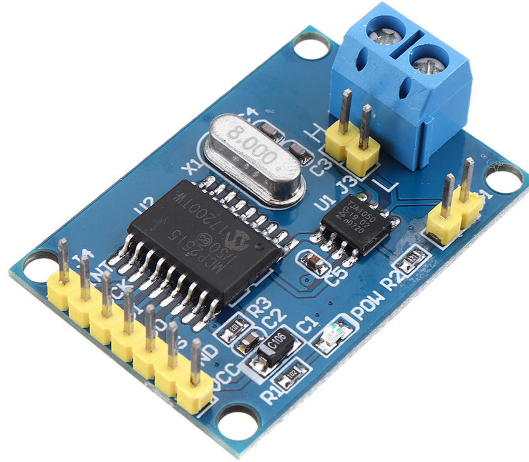


Figure 3.4: MCP2515 CAN Transceiver

A simple block diagram of the MCP2515 is shown in Figure 3.5. The device consists of three main blocks:

1. The CAN module, which includes the CAN protocol engine, masks, filters, transmit and receive buffers.
2. The control logic and registers that are used to configure the device and its operation.
3. The SPI protocol block.

CAN Module

The CAN module handles all functions for receiving and transmitting messages on the CAN bus. Messages are transmitted by first loading the appropriate message buffer and control registers. Transmission is initiated by using control register bits via the SPI interface or by using the transmit enable pins. Status and errors can be checked by reading the appropriate registers. Any message detected on the CAN bus is checked for errors and then matched against the user-defined filters to see if it should be moved into one of the two receive buffers.

Control Logic

The control logic block controls the setup and operation of the MCP2515 by interfacing to the other blocks in order to pass information and control. Interrupt pins are provided to allow greater system flexibility. There is one multipurpose interrupt pin (as well as specific interrupt pins) for each of the receive registers that can be used to indicate a valid message has been received and loaded into one of the receive buffers. Use of the specific interrupt pins is optional. The general purpose interrupt pin, as well as status registers (accessed via

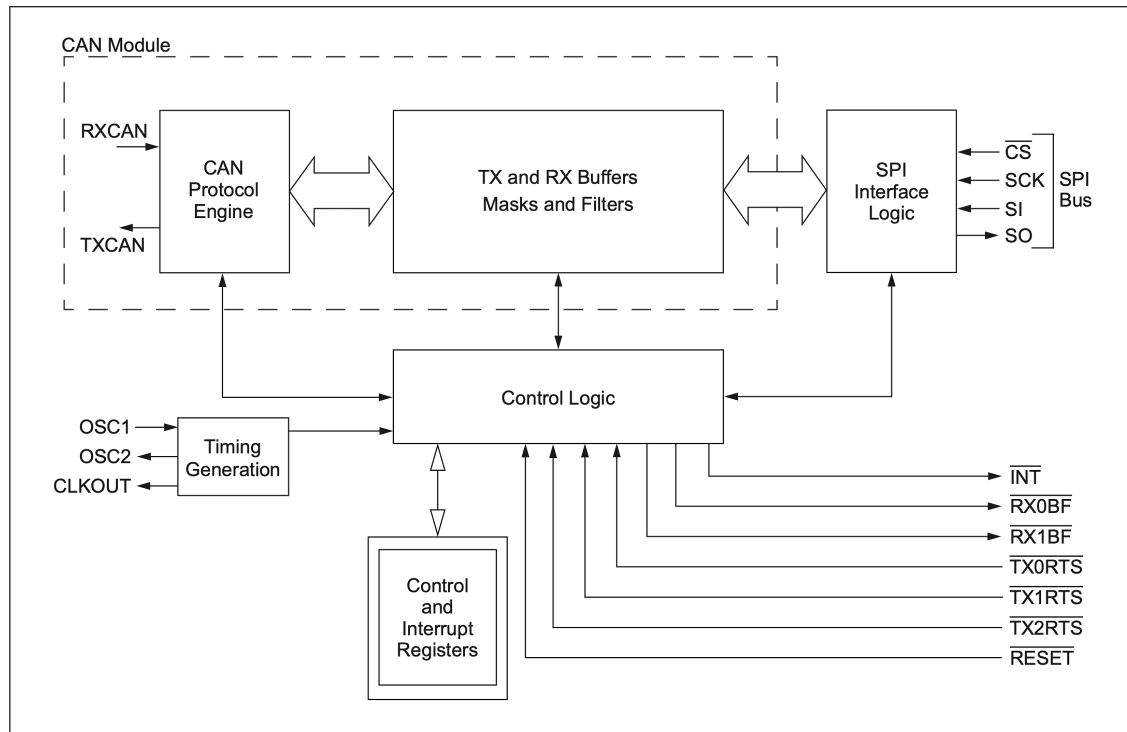


Figure 3.5: MCP2515 Block Diagram

the SPI interface), can also be used to determine when a valid message has been received. Additionally, there are three pins available to initiate immediate transmission of a message that has been loaded into one of the three transmit registers. Use of these pins is optional, as initiating message transmissions can also be accomplished by utilizing control registers accessed via the SPI interface.

SPI Protocol Block

The MCU interfaces to the device via the SPI interface. Writing to, and reading from, all registers is accomplished using standard SPI read and write commands, in addition to specialized SPI commands.

3.1.3 Real Time Clock

In order to give our system the capability of having the actual date and time as well as assigning them to the data received we have installed a PCF 8523 RTC (Real Time Clock). The PCF8523 is a CMOS1 Real-Time Clock (RTC) and calendar optimized for low power consumption. Data is transferred serially via the I2C-bus with a maximum data rate of 1000 kbit/s. Alarm and timer functions are available with the possibility to generate a wake-up signal on an interrupt pin. An offset register allows fine-tuning of the clock. The PCF8523 has a backup battery switch-over circuit, which detects power failures and automatically switches to the battery supply when a power failure occurs. The real time and date stamp capabilities this RTC offers, enables us to known exactly at what point each measurement occurred, since we are storing the data received. This means that in case an abnormal measurement is collected or a malfunction occurs we may know the exact point in time that this happened. Although this real time clock is only able to provide an accuracy of one second, we are able to pair it with the Arduino's timer

function which can accurately measure up to one millisecond. A millisecond accuracy is essential to our application, as data is received hundreds of times every second, sometimes with the same identifier. The main features and benefits of the PCF 8523 are outlined below:

- Provides year, month, day, weekday, hours, minutes, and seconds based on a 32.768 kHz quartz crystal
- Resolution: seconds to years
- Clock operating voltage: 1.0 V to 5.5 V
- Low backup current: typical 150 nA at V_{DD} = 3.0 V and T_{amb} = 25 C
- 2 line bidirectional 1 MHz Fast-mode Plus (Fm+) I2C interface, read D1h, write D0h2
- Battery backup input pin and switch-over circuit
- Freely programmable timer and alarm with interrupt capability
- Selectable integrated oscillator load capacitors for CL = 7 pF or CL = 12.5 pF
- Oscillator stop detection function
- Internal Power-On Reset (POR)
- Open-drain interrupt or clock output pins
- Programmable offset register for frequency adjustment

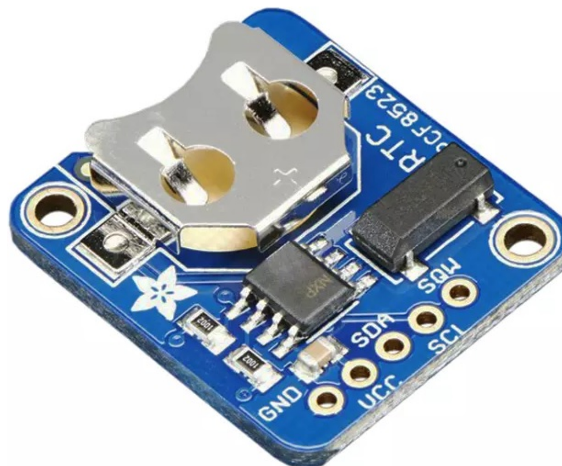


Figure 3.6: PCF 8523 Real Time Clock

3.1.4 Single Board Computer

For the purpose of displaying the data transmitted by the Arduino we need a portable computer running either a Windows, Mac OS or Linux operating system. In this application we have opted for the Raspberry Pi which is an ARM based credit card sized SBC (Single Board Computer) created by Raspberry Pi Foundation. The board runs Debian based GNU/Linux operating system Raspbian and ports of many other OSes exist for this

device. It is equipped with a 1.2 GHz processor, 1 GB of RAM and a 400 MHz GPU making capable to run our application on Python 3 and support the Live graphics. The Raspberry also has multiple I/O ports including, but not limited to, 4 USB 2.0 ports, an HDMI port and micro USB port for the power supply. The main benefits of the Raspberry Pi are outlined below:

- **Portable Size**

  Since the size of the device is no bigger than a smartphone it enables us to use it in remote applications or change locations from which the CAN signal is received.

- **Flash Memory**

  The Raspberry Pi has expandable memory through its SD card slot, meaning we can always replace it if its full and also remove it if we want to examine the data in another computer

- **Low Cost**

  The average cost of the Raspberry Pi 3 is around 40 euros, making it one of the most affordable portable computer systems

- **Large processing power for a compact board**

- **Multiple Interfaces**

  The SBC includes HDMI, multiple USB, Ethernet, onboard Wi-Fi and Bluetooth, many GPIOs, USB powered, etc. allowing us to make all the necessary connections to acquire data from the micro

- **Supports Linux & Python**

  A Unix operating system combined with Python help us take full advantage of the computers processing capabilities. Python compatibility is especially crucial, since our application mainly deals with data handling, for which this programming language is ideal.

- **Readily available examples with community support**

  The community and forum support offered is invaluable in starting to build applications and providing feedback while resolving any problems that may occur during development.

- **Overclocking capability**

  The Raspberry Pi can be overclocked if there are performance problems with the application used, but it is at the user's risk to do this as we will mention below.

While the Raspberry Pi offers numerous advantages for our applications there are also challenges that we have to face. Firstly, the operating system runs on an SD, which makes it easy to expand the memory, however, ruggedized applications will pose a problem as this SD card connection may have issues with vibrations in the field i.e. the engine room. There is no provision to ensure connections are intact while in operation and a possible consequence would be to end up using unreliable workarounds (double-sided tapes, glue, etc.). Another problem with this SD card situation is that it can become corrupted if it is written heavily or if the board is not powered down properly while the file-write operation was still going on. In order to solve this, we may end up providing an external battery or

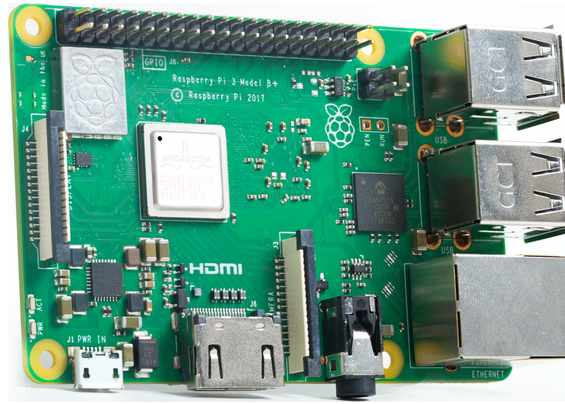


Figure 3.7: Raspberry Pi 3

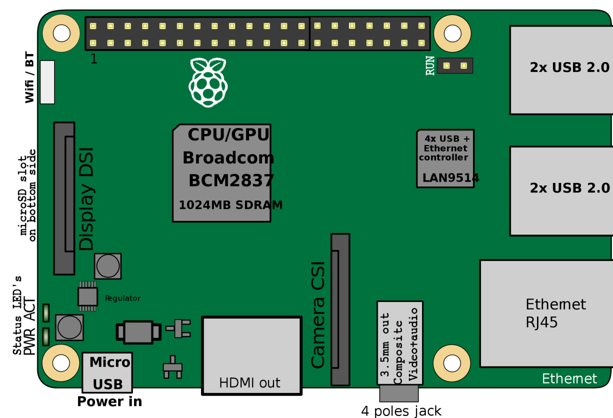


Figure 3.8: Location of connectors and main ICs on Raspberry Pi 3

supercapacitor-based power supply to provide a main power-off interrupt signal so that the software can handle proper shutdown before the board shuts off.

Furthermore, the microprocessor on the Raspberry Pi generates heat which must be managed or else it may impact the board's reliability. As of now, only a small heatsink with glue can be used. There are no mounting holes provided near the processor so that heatsink can be fastened properly. Ultimately, if we were to use the Raspberry Pi's computation abilities up to 70-90% we would need a more appropriate heatsink, capable of handling the board's rising temperatures which can come up to 75 degrees Celsius as shown in the testing performed on a similar circuit, photographed by thermal cameras in Figure 3.9. In this case we would end up using mounting holes on the board and a bigger heatsink, which may not be readily available. Although there is the possibility to custom design a cooling device and get it manufactured, this comes at extra cost and effort.

3.1.5 Hardware Connections

Now that we have analyzed each one fo our system's components separately we need to summarize how they are physically connected to achieve data transmittance.

The connection between the MCP module and the CAN bus is achieved by plugging the twisted pair of wires into their corresponding CAN High and CAN Low ports as demonstrated in Figure 3.10 and Figure 3.11.

All CAN Bus data traffic is captured through the MCP 2515 CAN transceiver which

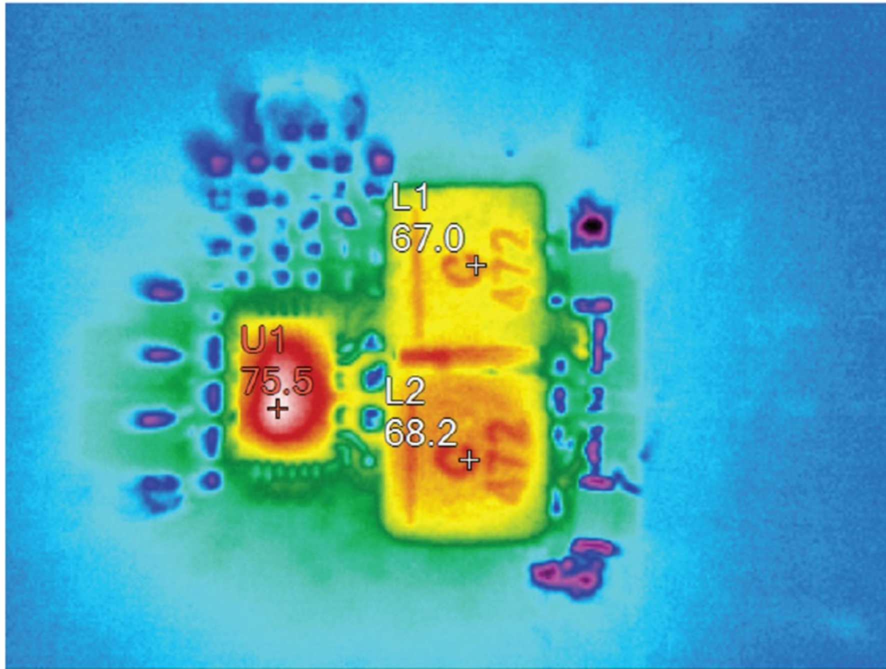


Figure 3.9: Thermal performance of similar circuit to Raspberry Pi

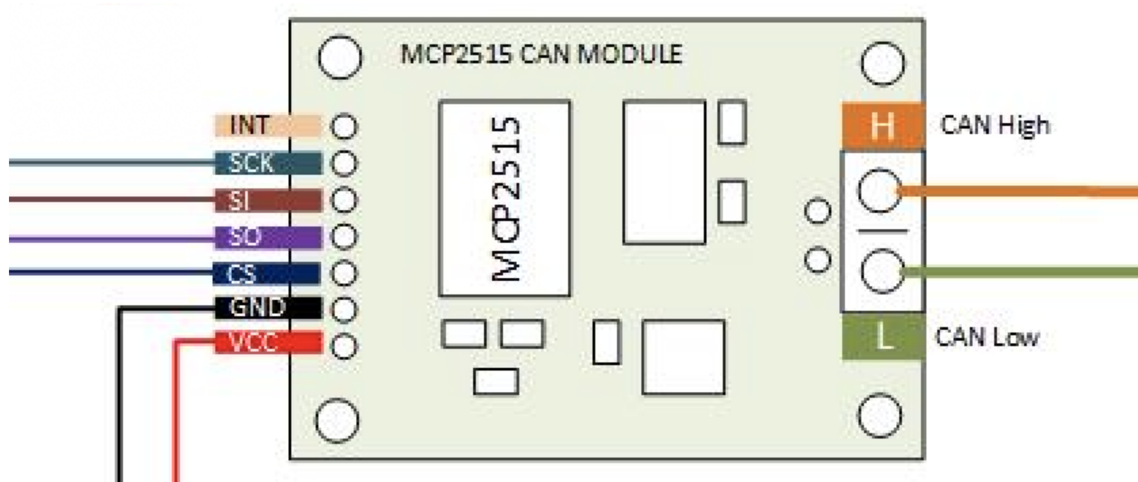


Figure 3.10: CAN Transceiver connection to CAN Bus

is in turn connected to the Arduino UNO microcontroller as shown in Figure 3.13.

Firstly, we will briefly analyze what the function of the pins on the MCP2515 is and then we will dive into the connectivity between the CAN transceiver and the Micro Controller. The MCP2515 pins, as shown in Figure 3.12, are the following:

- VCC: Ground connection for logic and I/O pins
- GND: Ground connection
- CS: Chip Select input for SPI (Serial Peripheral Interface) interface
- SO: Data output pin for SPI interface
- SI: Data input pin for SPI interface

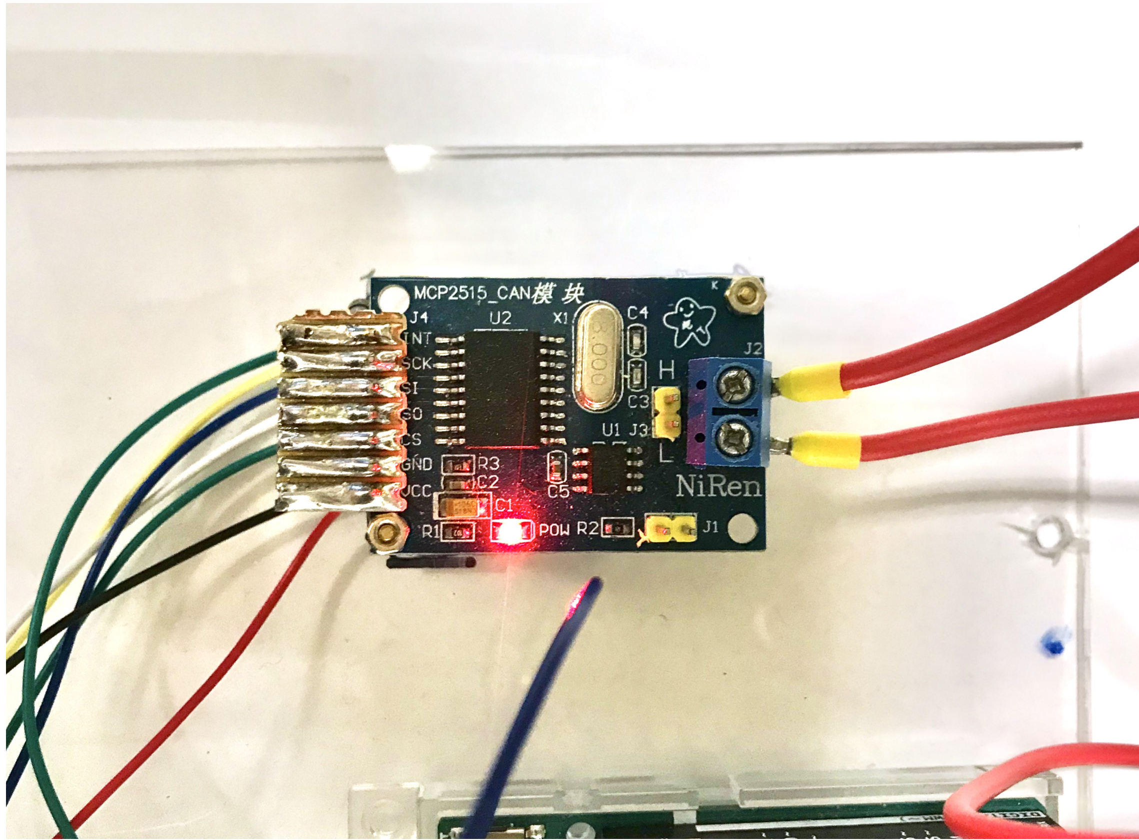


Figure 3.11: CAN Transceiver wiring with twisted pair cables

- SCK: Clock Input for SPI interface
- INT: Interrupt output pin

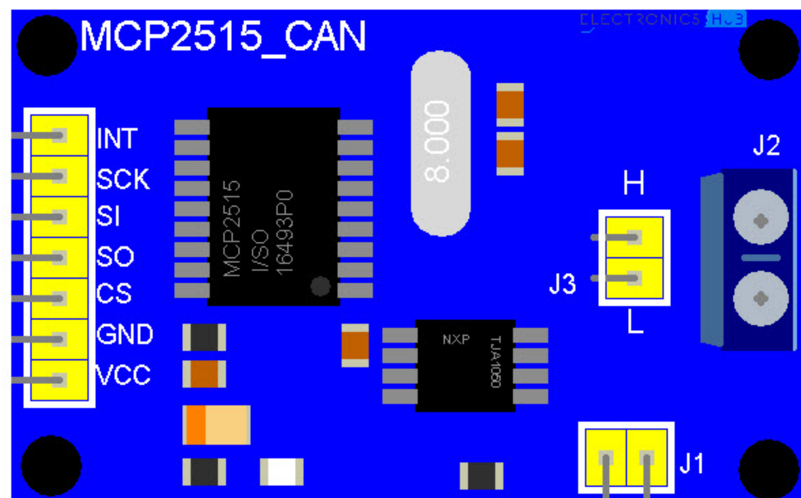


Figure 3.12: Connection Pins of the MCP2515 CAN Transceiver

The connections between the MCP transceiver and the Arduino are outlined in the following diagram and are as per Figure 3.13 (from MCP2515 to Arduino):

- VCC and GND to ground
- CS to Arduino digital pin 10
- SO to Arduino digital pin 12
- SI to Arduino digital pin 11
- SCK to Arduino digital pin 13
- INT to Arduino digital pin 2

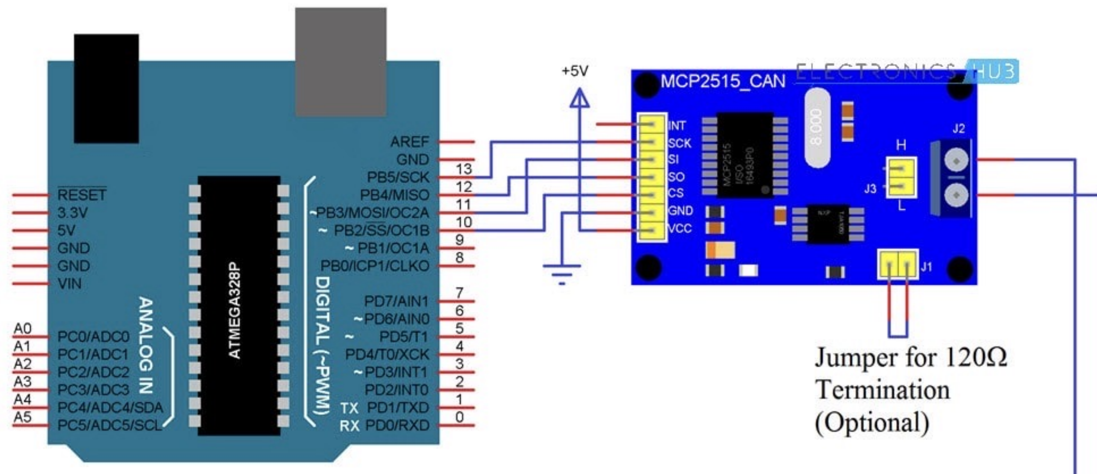


Figure 3.13: Arduino UNO and MCP 2515 Connection diagram

The MCP 2515 CAN transceiver is connected to the engine CAN bus module via a pair of twisted wires and to the Arduino as outlined above. In turn the Arduino is also connected to the PCF 8523 real-time clock as per the above schematic and to the Raspberry Pi via a USB cable. The Raspberry Pi is then connected to a mouse and keyboard via USB cables and to a monitor using its HDMI output. A layout of our system's components is presented in Figure 3.14.

Data Flow

Information from the engine's sensors is transmitted by the ECU to the CAN transceiver and Microcontroller, which in turn transmit the raw data to the single board computer for further processing. Through the SBC we are able to decode the data and plot our measurements. This chain of data transmission is presented in the data flow diagram in Figure 3.15.

3.2 Programming

Having already connected the microcontroller and transceiver setup to the engine via the pair of twisted wires, we will now describe the process behind acquiring, decoding and analyzing the CAN signals transmitted by the engine.

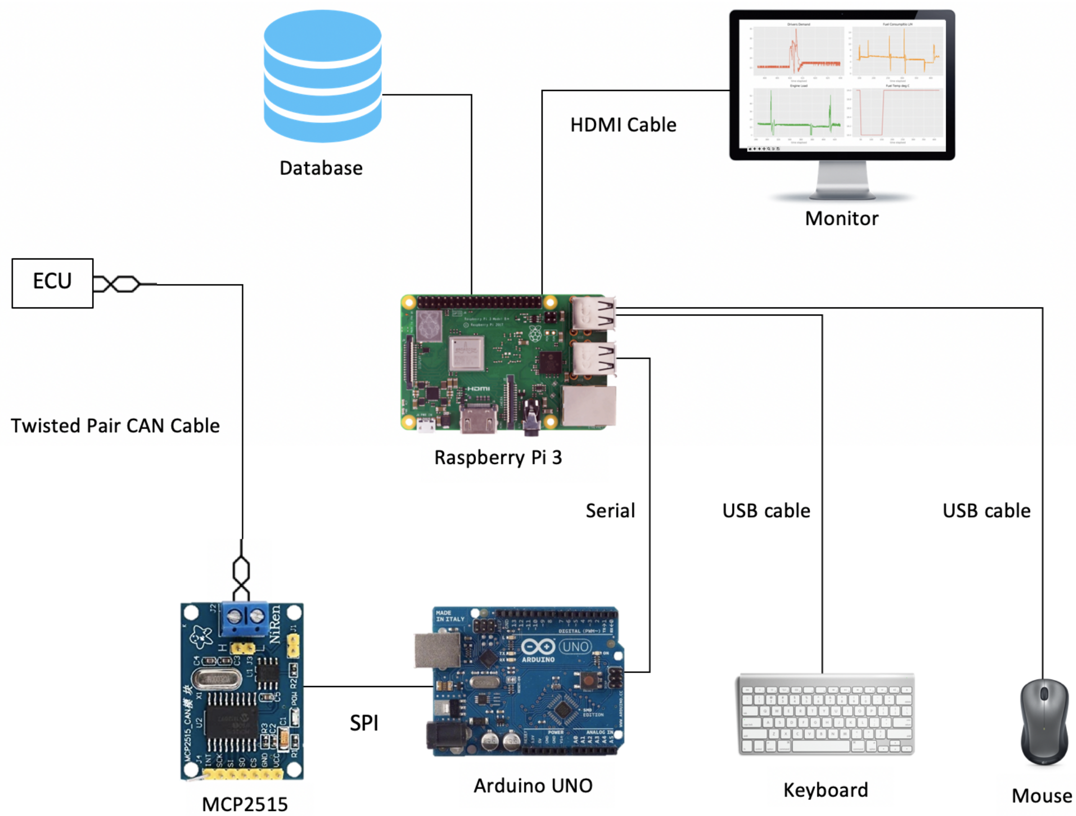


Figure 3.14: System Componects Layout

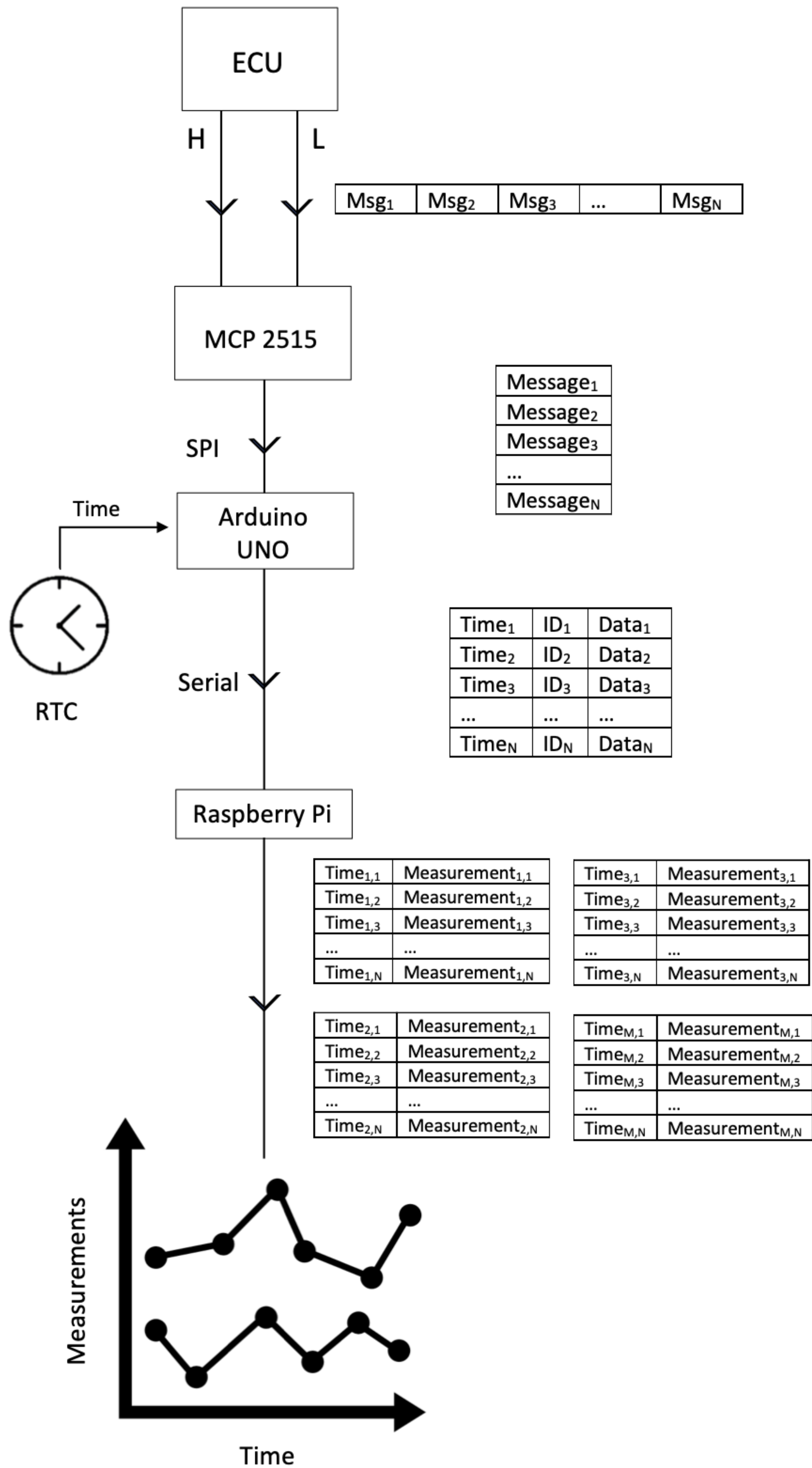
3.2.1 Programming Language and Data Analytics

For the purpose of reading the incoming data from the Arduino, decoding, sorting, storing and plotting it we have used the Python programming language, running on the IDLE compiler, both on the UNIX machine, where the initial development and testing was performed, as well as on the Raspberry Pi for our final application.

Python was used due to its simplicity and capability of easily handling large amounts of data. As a programming language it has seen a rapid rise in the programming and engineering community in the past few years since it offers numerous advantages, which as herewith outlined.

- **Simple Coding**

  Python programming involves fewer lines of code as compared to other languages available for programming. It is able to execute programs in the least lines of code. Moreover, Python automatically offers assistance to identify and associate data types. Python programming follows an indentation based nesting structure. The language can process lengthy tasks within a short span of time. As there is no limitation to data processing. As a result, we are able to compute data in commodity machines, laptop, cloud, and desktop with limited processing power, which is especially helpful when dealing with applications that involve dynamic graphic representation such as ours.


ECU
H
L
Msg1
Msg2
Msg3
...
MsgN
MCP 2515
Message1
Message2
Message3
...
MessageN
SPI
Time
Arduino UNO
RTC
Serial
Time1 ID1 Data1
Time2 ID2 Data2
Time3 ID3 Data3
TimeN IDN DataN
Raspberry Pi
Time1,1 Measurement1,1
Time1,2 Measurement1,2
Time1,3 Measurement1,3
Time1,N Measurement1,N
Time3,1 Measurement3,1
Time3,2 Measurement3,2
Time3,3 Measurement3,3
Time3,N Measurement3,N
Time2,1 Measurement2,1
Time2,2 Measurement2,2
Time2,3 Measurement2,3
Time2,N Measurement2,N
TimeM,1 MeasurementM,1
TimeM,2 MeasurementM,2
TimeM,3 MeasurementM,3
TimeM,N MeasurementM,N
Measurements
Time

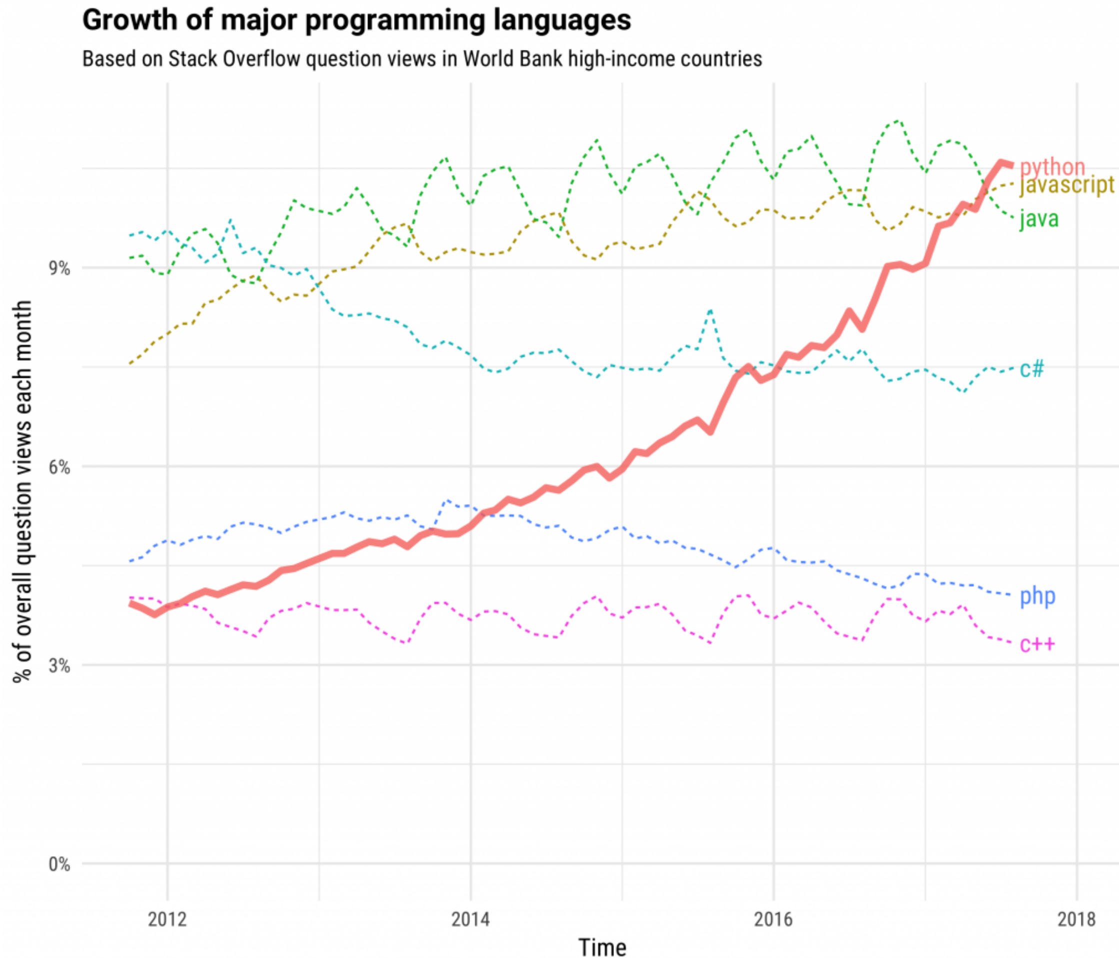


Figure 3.16: Increase of Python's use as a programming language

- **Open Source**

  Developed with the help of a community-based model, Python is an open-source programming language. Being an open-source language, it supports multiple platforms. In addition, it may be run in various environments such as Windows, Mac and Linux, making it easily adaptable to different hardware configurations.

- **Library Support**

  Python programming offers the use of multiple libraries. This makes it a famous programming language in fields like scientific computing and data analytics. The main library that enables us to deal with the large amounts of data involved in our application is Pandas, on which will we expand further below.

- **Scope** Python allows users in simplifying data operations. As Python is an object-oriented language, it supports advanced data structures. Some of the data structures that Python manages include lists, sets, tuples, dictionaries and many more. Besides this, Python helps in supporting scientific computing operations such as matrix operations and data frames, with the latter being the data structures around which this application has been developed. These features of Python help to enhance the scope of the language thus enabling it to speed up data operations.

Data Processing Library

For this application, the data processing library used was Pandas. The name is a combination of the words panel and data which has roots in econometric and Python data analysis. Pandas deals with the data processing and analysis in five steps: load, prepare, manipulate, model and analyze. It is a widely used tool, particularly in data wrangling and munging. Moreover, it is available for everyone as an open source project and free to use.

The pandas forms a core component of the Python data analysis corpus. The distinguishing feature of pandas is the suite of data structures that it provides, which is naturally suited to data analysis, primarily the DataFrame and to a lesser extent Series (1-D vectors) and Panel (3D tables). In addition, Pandas has very powerful tools for reading and writing data between computer memory and inbuilt data structures. Tools for supporting different formats include plain text, Comma Separated Values (CSV), Relational Databases and HDF5 for fast access

Some of the numerous advantages this library offers to data science applications are mentioned below:

- **Data structure:** Pandas has a fast and efficient Data Structure i.e. DataFrame for data manipulation. A DataFrame is a 2-dimensional data structure with rows and columns. It's a table like structure in SQL or similar to a spread sheet. Pandas object replicated like a dictionary from a Python perspective.

- **Data representation:** It can easily represent data in a form naturally suited for data analysis via its DataFrame and Series data structures in a concise manner. Doing the equivalent in Java/C/C++ would require many lines of custom code, as these languages were not built for data analysis but rather networking and kernel development.

- **Data sub-setting and filtering:** It provides for easy subsetting and filtering of data, procedures that are a staple of doing data analysis. It also performs intelligent label based slicing, performance quick indexing and fast sub setting of large data sets in addition to handling missing values from data, and data alignment

- **Concise and clear code:** Its concise and clear API allows the user to focus more on the core goal at hand, rather than have to write a lot of scaffolding code in order to perform routine tasks. For example, reading a CSV file into a DataFrame data structure in memory takes two lines of code, while doing the same task in Java/C/C++ would require many more lines of code or calls to non-standard libraries

3.2.2 Decoding and Processing Measurements

The raw, unprocessed data is first transmitted to the Raspberry Pi or any other computer we choose to use via the serial port from the Arduino. Using the 'Serial Read' script we have written, we are able to display all the data on our monitor, through a terminal window and save it to a text document. The information we receive, before decoding it, includes the DLC, Identifier and value in Hexadecimal form along with the time stamp for each message. We can either use a timestamp which will be the time elapsed since the connection or the actual time when the message was received using the PCF real time clock we have installed, or both if needed. The information must then be sorted, decoded and plotted in real time, which is achieved through the second python script.

```
353.8469848632,98FD9800,8 0 0 0 FF FF FF FF
353.8699951171,98F00F91,FF FF FF FF FF FF FF FF
353.8940124511,8CF00400,FF 81 9B 61 21 FF FF FF
353.9179992675,8C000001,2 90 1A 9B 0 0 0 0
353.9419860839,98FD7C00,F8 FF F3 FF FF FF E3 FF
353.9660034179,8CF00400,FF 81 9B 3E 21 FF FF FF
353.9899902343,98F00F91,FF FF FF FF FF FF FF FF
354.0140075683,98FEF000,FF FF FF FF FF FF FF FF
354.0379943847,8C000001,2 90 1A 9B 0 0 0 0
354.0620117187,8C000001,2 90 1A 9B 0 0 0 0
354.0849914550,98FDD500,FF FF FF FF 0 0 FF FF
354.1090087890,98F00F91,FF FF FF FF FF FF FF FF
354.1340026855,8CF00400,FF 81 9B 26 21 FF FF FF
354.1570129394,8C000001,2 90 1A 9B 0 0 0 0
354.1820068359,8C000001,2 90 1A 9B 0 0 0 0
354.2049865722,8CF00400,FF 81 9B FE 20 FF FF FF
354.2290039062,98FEDF91,FF C0 12 FF FF 7 7 FF
354.2529907226,8CF00400,FF 81 9B 19 21 FF FF FF
354.2770080566,8CEC1791,FF FF FF FF FF 0 EF 0
354.2999877929,8C000001,2 90 1A 9B 0 0 0 0
354.3250122070,98FEF000,FF FF FF FF FF FF FF FF
```

Figure 3.17: Monitoring of Unprocessed CAN Data in a terminal window

To decode the incoming messages we used the documentation for "J1939 Parameters and Parameter Group Information" that provides a detailed analysis of all messages transmitted based on the J1939-11 and J1939-71 protocols. Firstly, we needed to identify the messages that we want to decode from the vast amount of raw data received. On average the engine transmits 230 messages every second, so filtering out the unwanted data, greatly reduces the processing load of our system. Each signal/message is characterized by a PGN or Parameter Group Number for ease of sorting and searching in this large database. As

mentioned beforehand, the PGN is the J1939 identifier. In our case, it is transmitted within the message ID, meaning that we cannot exactly match the ID with the PGN, but we have to separate the 18 bit PGN from the identifier whose length is 29 bits, as shown in Figure 3.18. We must then convert the hexadecimal value to a decimal one as the list of the Parameter group Numbers is given in decimal form in the documentation. The list of parameters we have chosen to monitor for the purposes of this project are summarized in Table 3.1.

Table 3.1: CAN J1939 Data Link Parameters Reference List

Broadcast								
Parameter Group	Identifier	Rate(msec)	PGN		Default Priority	Parameters Supported	SPN	Coming from ECU*
			DEC	HEX				
Electronic Engine Control 2 (EEC2)	8CF00300	50	61443	F003	3	Accelerator Pedal Position 1	91	00
						Engine Percent Load at Current Speed	92	00
						Accelerator Pedal Position 2	29	00
Electronic Engine Control 1 (EEC1)	8CF00400	15	61444	F004	3	Actual Engine – Percent Torque	513	00
						Engine Speed	190	00
						Driver Demand Eng Percent Torque	512	00
Electronic Engine Control 3 (EEC3)	98FEDF00	250	65247	FEDF	6	Engine's Desired Operating Speed	515	00,91
						Nominal Friction - Percent Torque	514	00
						Aftertreatment 1 Gas Mass Flow	3236	91
Engine Temperature (ET1)	98FEEE00	1000	65262	FEEE	6	Engine Coolant Temperature	110	00
						Engine Fuel Temperature 1	174	00
Engine Fluid Level/Pressure (EFLP1)	98FEEF00	500	65263	FEEF	6	Engine Fuel Delivery Pressure	94	00
						Engine Oil Pressure	100	00
						Engine Coolant Level 1	111	00
						Engine Crankcase Pressure 1	101	00
FueL Economy (Liquid) (LFE1)	98FEF200	100	65266	FEF2	6	Engine Fuel Rate	183	00
						Engine Throttle Valve 1 Posit	51	00
Inlet/Exhaust Conditions (IE1)	98FEF600	500	65270	FEF6	6	Engine Intake Manifold # 1 Pı	102	00
						Engine Intake Manifold 1 Ten	105	00
						Engine Air Intake Pressure	106	00
Intake Maniforld Information 1	98FEA600	500	65190	FEA6	6	Engine Turbocharger 1 Boost	1127	00

*00 = Engine ECU, 91 = Aftertreatment ECU

J1939 messages are built on top of CAN 2.0b and make specific use of extended frames. The first three bits are reserved for the priority field. This field sets the message's priority on the network and helps ensure messages with higher importance are sent/received before

lower priority messages. Zero represents the highest priority. The next bit is reserved for future use. This field is set to zero. The following bit is the Data Page field. This is used to expand the maximum number of possible messages.

The next eight bits make up the Protocol Data Unit Format (PDU F) field. This is used to determine if the message is intended for a specific device on the network or if the message is intended for the entire network. If the value of PDU F is less than 240 (0xF0 in hexadecimal form), the message is meant for a specific device. If the value is 240 or greater, the message is intended for all devices.

The next eight bits form the Protocol Data Unit Specific (PDU S) field. The definition of this field is based on value of the PDU F field. Meaning that if PDU F is intended for a specific device (less than 239), PDU S is interpreted as the address of that specific device. In this case, the PDU S field is referred to as the Destination Address field. This format is defined as PDU 1. If PDU F is intended for all devices (greater than or equal to 240), PDU S is interpreted as a Group Extension field. This group extension is used to increase the number of possible broadcast messages. This format is referred to as a PDU 2 and these type of messages are also known as global destination messages. The last eight bits identify the address of the device that transmitted the current message. This is known as the Source Address Field. A breakdown of the CAN extended frame can be seen in Figure 2.7.

On standard CAN networks, identifiers are used to uniquely define each message. This concept exists within J1939 as well. Because the priority and source address fields can change, they are not used for this purpose. This leaves the reserve, data page, PDU F and PDU S fields. This new combination of fields is referred to as the Parameter Group Number (PGN). Each message in J1939 must have its own unique PGN. The J1939-71 standard is responsible for assigning these unique PGNs to standard messages.

As an example we will use the identifier 8CF00400, which corresponds to PGN 61444 and provides the *"Electronic Engine Controller 1"* Information. This message contains the Actual Engine - Percent Torque. In this example the Identifier is given in hexadecimal form with the first two bytes representing the Priority, the next four are the PGN and the last two the source address. In this case 8C is the priority, F004 is the PGN, which when converted from its hexadecimal format to decimal equals 61444 and 00 is the source address, meaning the signal is coming from the ECU.

A breakdown of the above is shown in Figure 3.18. If the source address where equal to 91 that would mean that the signal was coming from the Aftertreatment of the ECU.

Identifier Breakdown:

8C	F004	00
Priority	**Parameter Group Number**	**Source Adress**

Figure 3.18: Message Identifier Breakdown

For each ID of a CAN message we may have multiple Parameters contained within it, since it contains a large number of bytes. For instance, for the message with ID F004 we can acquire data for the following parameters:

- **Measurement 1:** Engine Torque Mode (Bits 1-4)
- **Measurement 2:** Actual Engine - Percent of maximum Torque (high resolution) (Bits 5-8)
- **Measurement 3:** Driver's Demand, percent of maximum Engine Torque (Bits 9-16)
- **Measurement 4**: Actual Engine - Percent of maximum Torque (Bits 17-24)
- **Measurement 5:** Engine Speed (Bits 25-40)
- **Measurement 6:** Source Address (Bits 41-48)
- **Measurement 7:** Engine Starter Mode (Bits 49-56)
- **Measurement 8:** Engine Demand - Percent of maximum Torque (Bits 57-64)

To decode each message we need the length of the PGN, as well as the start and end position of each parameter contain within it, as outlined in Figure 3.19.

Example of data received from Identifier 8CF00400:

Original Value (HEXADECIMAL):

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
FF	7E	7E	00	00	FF	FF	FF

Conversion from Hexadecimal to Binary using predefined function in Python

Converted Value (BINARY):

Bits 1-4	Bits 5-8	Bits 9-16	Bits 17-24	Bits 25-32	Bits 33-40	Bits 41-48	Bits 49-56	Bits 57-64
1111	1111	01111110	01111110	00000000	00000000	11111111	11111111	11111111
Engine Torque Mode	Actual Engine - Percent of maximum Torque (high resolution)	Driver's Demand – Percent of maximum Torque	Actual Engine - Percent of maximum Torque	Engine Speed		Source Address	Engine Starter Mode	Engine Demand – Percent of maximum Torque

Figure 3.19: Message ID F004 Breakdown

The length is the amount of bytes dedicated to give the value of each parameter inside the entire 64-bit message. It is worth noting that in some cases the IDs contain data with a length of 56 bits or less. For the purpose of this project however, we were not examining these values, as they do not include parameters that we wish to monitor or control.

Now we move on to decode and process the data, which was described in theory above. We must first split the hexadecimal values received into bytes, 8 in total, as the messages

we have selected are 64 bits in length. Each byte will be entered into a new column of the dataframe. At the same time, we are running a test on the values that equal zero to convert them to '00' and have a length of 64 bits in all numbers as described previously. By having each byte into a separate column we can easily segregate each part of the message and choose the byte we need by selecting its corresponding column as well as being able to switch the order of bytes in case a parameter is defined by two of more bytes. Specifically, if the parameter is defined by bytes 'n' and 'n+1' then the byte numbered 'n+1' will be placed first and the byte 'n' will be placed after it. This process is further analyzed in the below subsection.

Big Endian & Little Endian

To gain a better understanding of the order in which the bytes need to be arrange for us to read the data correctly we must first understand the concept of endianness.

In computing, endianness is the order or sequence of bytes of digital data in computer storage. Endianness is primarily expressed as big-endian (BE) or little-endian (LE). A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest, this way we can see the information as it would be written, meaning from left to right. A little-endian system, in contrast, stores the least-significant byte at the smallest address. In some analogy, the order or sequence in which the bits are transmitted over a communication channel is sometimes termed endianness.

Specifically, in big endian when dealing with decimal numbers the bytes representing the largest values come first. Regular integers are printed this way. For instance, the number "1025" shows the numeral one first which represents "1000". This is a representation most comfortable to humans. This most significant value first is represented in bytes for computer memory representation. The number 1025 is represented in hex as 0x0401 where 0x0400 represents 1024 and 0x0001 represents the numeral 1. Their sum is 1025. The most significant (larger value) byte is listed first in this big endian representation. On the other hand when dealing with little endian the number 1025 would be represented in hex as 0x0104. Endian byte ordering affects integer and floating point data but does not affect character strings as they maintain the string order as viewed and intended by the programmer. In other words, the bytes representing the entire number are swapped. It is crucial to note that only the bytes are reversed and the bits within the byte are not reversed. This is why in the above example the number 1025 is written in little endian as 0x0104 and not as 0x1040.

Computer processors store data in either large (big) or small (little) endian format depending on the CPU processor architecture. The Operating System (OS) does not factor into the endianess of the system. Big endian byte ordering is considered the standard or neutral "Network Byte Order". Big endian byte ordering is in a form for human interpretation and is also the order most often presented by hex calculators. In our application we are using an ARM type processor which is bi-Endian, meaning it can interpret information in both ways.

The difference between big endian and little endian byte architecture is demonstrated in Figure 3.20 using a 32-bit integer in hexadecimal form as an example.

In our application the data is transmitted by the engine's ecu in both big endian and little endian format. Firstly, the entirety of the message is transmitted in big endian format, thus it should be read with the bytes in order from left to write. However, if a measurement is represented by more than one byte of data, then it is given in little endian format, meaning that to convert its value correctly we must swap the order of the bytes

To demonstrate the above procedure we will use as an example the Engine Fuel Rate, which is a parameter received within the data of identifier '98FEF200'. The value is

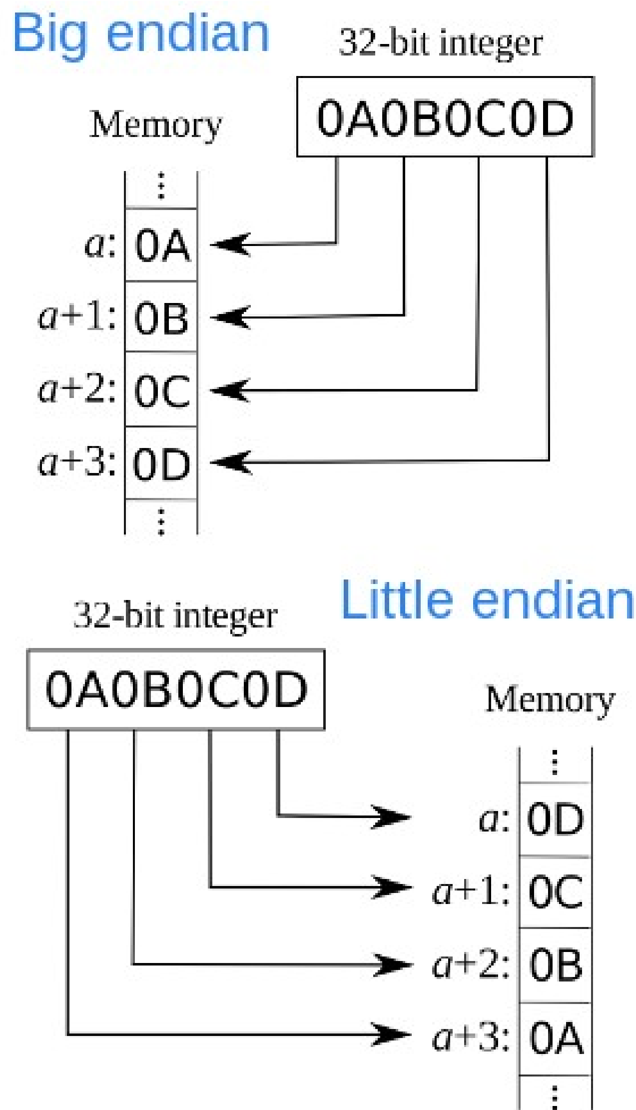


Figure 3.20: Difference in Endian Byte Architecture

defined by bits 1 to 16 meaning the first two bytes of the hexadecimal value. In this case we will have to rearrange the order of the HEX numbers and place firstly the second byte, named *byte 1*, and secondly the first byte, named *byte 0.*

At this stage, it is important to note that the length and position refer to the BINARY form of the CAN message. The messages are originally displayed in hexadecimal form and their conversion to binary and finally to decimal format will be covered later on. Furthermore, for every PGN there is a specific transmission rate. This defines the time intervals at which the engine transmits the information contained in each specific ID. The transmission rate enables us to display data for a certain period of time elapsed, a capability which will be expanded upon below. Each Parameter transmitted is also characterized by a certain Data Range. This is the predefined range a value or a status parameter may have, based on its offset and resolution. The actual range of values that may be represented by a parameter are defined by the "Operational Range". In addition, if applicable, the value of each Parameter is modified by a Resolution and Offset. The former represents the bit scaling of the value, meaning its multiplication by a certain value, and the latter is a negative value which is added to the decimal format of the value.

Finally, each parameter is defined by an SI unit to determine the physical meaning.

Example of Signal Decoding

We will now apply the above process to one of the parameters we are plotting using data gathered from a recent experiment. More specifically we shall be examining the Engine Fuel Rate, the data for which is contained within the 98FEF200 identifier.

Table 3.2: Engine Fuel Economy Parameters

Broadcast								
Parameter Group	**Identifier**	**Rate(msec)**	**PGN**		**Default Priority**	**Parameters Supported**	**SPN**	**Coming from ECU***
			DEC	**HEX**				
Fuel Economy (Liquid) (LFE1)	98FEF200	100	65266	FEF2	6	Engine Fuel Rate	183	00
						Engine Throttle Valve 1 Position 1	51	00

*00 = Engine ECU, 91 = Aftertreatment ECU

As with the 8CF00400 identifier examined above, there are multiple PGLs transmitted in the data corresponding to this ID. The parameter we will be focusing on, Engine Fuel Rate, is the information contained from bits 1 to 16. Since the data is 16 bits (or 2 bytes) in length we will have to reverse the order of the bytes in order to decode it correctly as mentioned before.

During the test performed on the engine testbed, we received the following data in hexadecimal form for this identifier:

23 **0** FF FF FF FF **0** FF

The first step to decoding this value is to fix our single digit values where applicable. This means that numbers equaling zero should be converted from '0' to '00' in order for the full 64 bits of the message to be displayed. If we skip this step, we will be selecting the wrong part of the message and our measurement will be inaccurate. Thus, the data will now be:

23 **00** FF FF FF FF **00** FF

Then we are isolating the first and second bytes that contain the wanted data, named "byte 0" and "byte 1" respectively. We are in turn reversing the order of the bytes, since they are being transmitted in the little-endian format, placing the second byte before the first. Now our data is:

00 23

At this stage we will convert the data, which is displayed in hexadecimal form into binary, using the corresponding function in our code. The value will be:

$(0023)_{16} = (100011)_2$

The above conversion is performed by breaking down the hexadecimal number and converting each part to binary. In this case we disregard the value zero in the beginning and break down the rest as follows:

$(0023)_{16} = (20 + 3)_{16} = (100000 + 11)_2 = (100011)_2$

Table 3.3 shows the conversion from hexadecimal to binary and helps us acquire a basic understanding of how the function converts numbers.

The data is now converted from binary to decimal using, once again, a function defined in our code:

$(100011)_2 = (35)_{10}$

Table 3.3: Hexadecimal to Binary Conversion Table

Hexadecimal	Binary
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
A	00001010
B	00001011
C	00001100
D	00001101
E	00001110
F	00001111

The conversion from binary to decimal is performed as per below:
$(100011)_2 = ((1 * 2^5) + (0 * 2^4) + (0 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0))_{10} = (35)_{10}$
Finally, we are applying the resolution and offset to the above value as outlined in the J1939 documentation as show in Table ??. In this case the resolution is 0.05 L/h per bit and there is no offset. Thus, our final value is:
$Fuel\ Consumption = 1.75\ L/h$

Table 3.4: J1939 Parameter Reference Table

Parameter Group Label	Position	Transmittion Rate(msec)	SPN Length	SPN	Name	Description	Data Range	Resolution	Offset	Unit
Fuel Economy (Liquid) (LFE1)	1-2	100	16	183	Engine Fuel Rate	Amount of fuel consumed by engine per unit of time.	0 to 3,212.75 L/h	0.05 L/h per bit	0	L/H

3.3 Data Representation

Our objective is to plot the measurements received from the engine during it's operation to a display in real time, by updating the plots constantly. For this purpose we have chosen to use Matplotlib, a comprehensive library for creating static, animated, and interactive visualizations in Python that gives us such a capability. Specifically, we have used a function which updates the data on our graphs with every iteration of the Python code, every five seconds. This refresh rate has been chosen in order to not surpass the graphic capabilities of the Single Board Computer, which are slightly limited.

The amount of data collected during the engine's operation becomes too great to plot in one graph after a certain amount of time as we have too many sets of values, a result of the very high trasmittion rate that many of the measurements have. As a result we have chosen to plot the data which corresponds to the last 60 seconds of the engine's operation. This method results in a graph which is easier to read as it contains much fewer values and whose variations in the Y axis are more clear to us.

To achieve the above we use the transmission rate of each parameter which we can find using its corresponding PGN as it is given in the J1939 documentation. Specifically, by knowing the rate at which the data is received we can calculate the amount of values we need to save in order to plot its progression for a certain period of time.

For instance, the Turbo Boost Pressure, whose identifier is 98FEA600, gets transmitted every 0.5 seconds by the ECU, meaning we only need the last 120 rows of the dataframe to plot its progression for 60 seconds.

Table 3.5: Turbo Boost Pressure Transmission Rate

Parameter Group Number	Parameter Group Label	Position	Transmittion Rate(msec)	SPN Length	SPN	Name	Unit
65190	Intake Manifold Information 1	1-2	500	16	1127	Engine Turbocharger Boost Pressure	kPa

As mentioned beforehand we are plotting 8 parameters in total, using 2 figures which contain 4 plots each in a two by two layout shown in the below figure. In case we wish to monitor a different set of parameters we can modify our script accordingly so that they may be plotted instead. Plotting more than 4 lines per figure will not only decrease the performance of our code, but will result in plots that are too small and thus difficult to read.

3.4 Transmitting Messages

In addition to receiving and decoding engine data, our setup can be used to also transmit or request information to the engine. This capability allows us to control certain parameters without using the standard ECU but also to acquire information that is only transmitted upon request from the operator.

To transmit a message to the engine we are utilizing the same setup, only making modifications in the code running on the Arduino. The process includes setting up the MCP2515 in the script as with the receiving code. Afterwards, for each message we want to transmit, we need to define its ID, DLC and the value that will be included in every byte of the data, the amount of which is dependent on the DLC. Data is transmitted in the same format as it is being received. This means that the entirety of the message is in big-endian format, but if a parameter is defined by two of more bytes then they are organized in little-endian order as described above. More than one CAN message may be transmitted per iteration of the code, allowing us to control more than one parameter of the engine operation at a time. Our code has the ability to run either at the smallest possible interval for the Arduino or at an interval specified by the user. The transmission capability not only allows us to control the engine via the set-up that we have created but also to request from the ECU information that it would otherwise not transmit. The

Figure 3.21: Data Representation in Real-Time

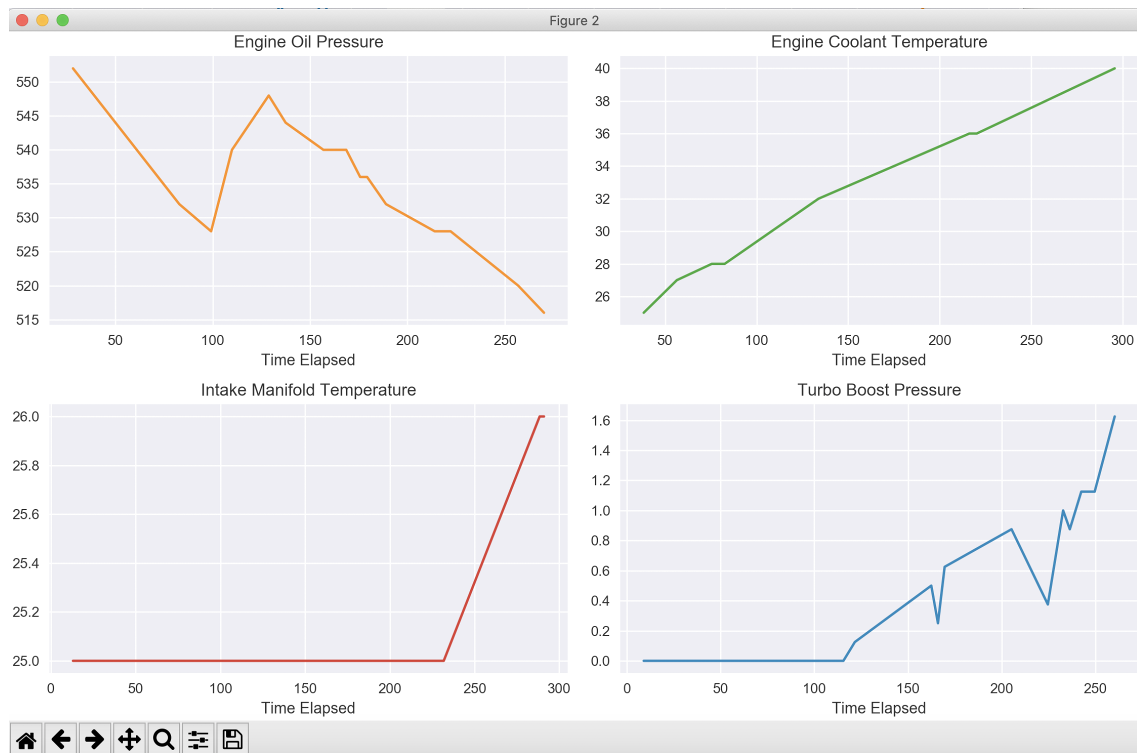


Figure 3.22: Data Representation in Real-Time

list of messages that the engine only transmits upon request is outlined in the J-1939 documentation. These include:

- **ECU Identification Information** (ECU part numbers & serial numbers). To

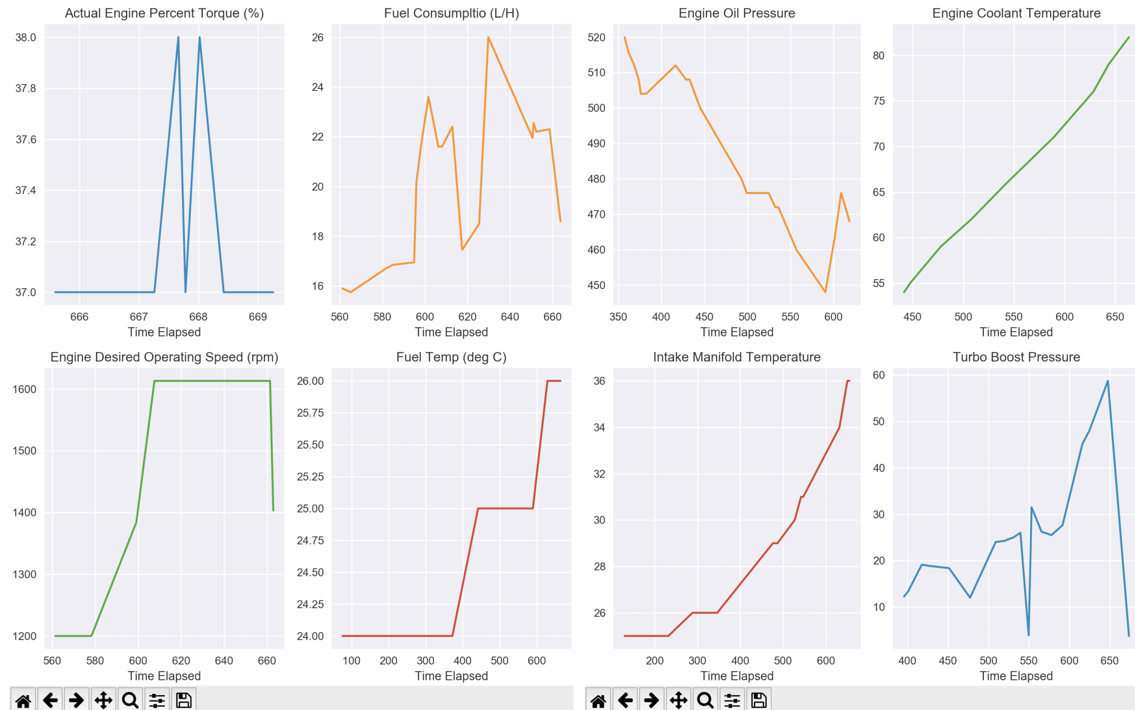


Figure 3.23: Data representation of all parameters monitored

assist in case we need to replace parts, making sure that we are using the correct components.

- **Service Information** (Components identification)
- **Auxiliary Analog Information** (Temperature and pressure measured by auxiliary sensors installed). In case we have installed additional monitoring sensors their output can only be received upon request.
- **Engine Hours, Revolutions**. This information can help us monitor service and overhauling intervals to ensure proper operation of the engine.
- **Fuel Consumption** (Total engine fuel used). Since only specific fuel consumption information is given by the engine, the total consumption can help us monitor the amount of fuel remaining and plan accordingly.

Table 3.6: Engine Information Broadcasted upon user request

On Request								
Parameter Group	**Identifier**	**Rate(msec)**	**PGN**		**Default Priority**	**Parameters Supported**	**SPN**	**Coming from ECU***
			DEC	**HEX**				
Auxiliary Analog Information	1CFE8C00	On Request	65164	FE8C	7	Auxiliary Temperature 1	441	00
						Auxiliary Pressure # 1	1387	00
						Auxiliary Temperature 2	442	00
Engine Hours, Revolutions	8CF00400	On Request	65253	FEE5	6	Engine Total Hours of Operat	247	00
						Engine Total Revolutions	249	00
Fuel Consumption (Liquid)	98FEDF00	On Request	65257	FEE9	6	Engine Total Fuel Used	250	00
						Engine Trip Fuel	182	00
ECU Identification Information	98FEEE00	On Request	64965	FDC5	6	ECU Part Number	2901	00
						ECU Serial Number	2902	00
Service Information	98FEEF00	On Request	65216	FEC0	6	Service Component Identification	911	00

*00 = Engine ECU, 91 = Aftertreatment ECU

Chapter 4

Experiment

4.1 Introduction

On June of 2020 we conducted a series of two experiments on the HIPPO-2 testbed to examine the accuracy of our system compared to the already established engine control setup, which runs on Matlab Simulink and which also captures the CAN Bus data transmitted by the engine.

For this experiment the engine was loaded based on a ship and propeller model which required power from the propulsion system, as a way of simulating actual operating conditions at sea. In order to achieve the application of variable loads corresponding to the conditions established by the model to the engine, reverse torque was applied by the Electric Brake and Electric Motor. The data to establish the dynamic model was taken from a propeller loading profile on Marine Traffic.

4.2 dSpace Monitoring System

The experimental facility is equipped with a monitoring and control system which is controlled by dSpace Microautobox II equipment. ControlDesk is the dSPACE experiment software for used to achieve seamless ECU integration. The user is able to operate, troubleshoot and collect data from any testbed operating under dSMA II by using a customizable user interface. Some of the possibilities that ControlDesk provides are the development process stages of rapid control prototyping, hardware-in- the-loop simulation, ECU calibration and diagnostics, analysis of vehicle bus systems and many more. ControlDesk Next Generation makes it easy to access data and to control simulation platforms, measurement devices, bus interfaces and ECUs. The system is integrated through Matlab and runs on Matlab Simulink. The typical user control interface is show in Figure 4.1.

4.3 Comparisson of Measurements

The system that was built for the purposes of this project was operated simultaneously with the dSpace Micro Auto Box 2 monitoring and control system of the ECR (Engine Control Room) in the LME (Laboratory of Marine Engineering). Both systems monitored and captured the data received during the two experiments that were conducted, testing the engine's performance.

Data from the dSpace system was captured on Matlab Simulink, while data from our system was captured from Python as we established in previous chapters. Both sets of

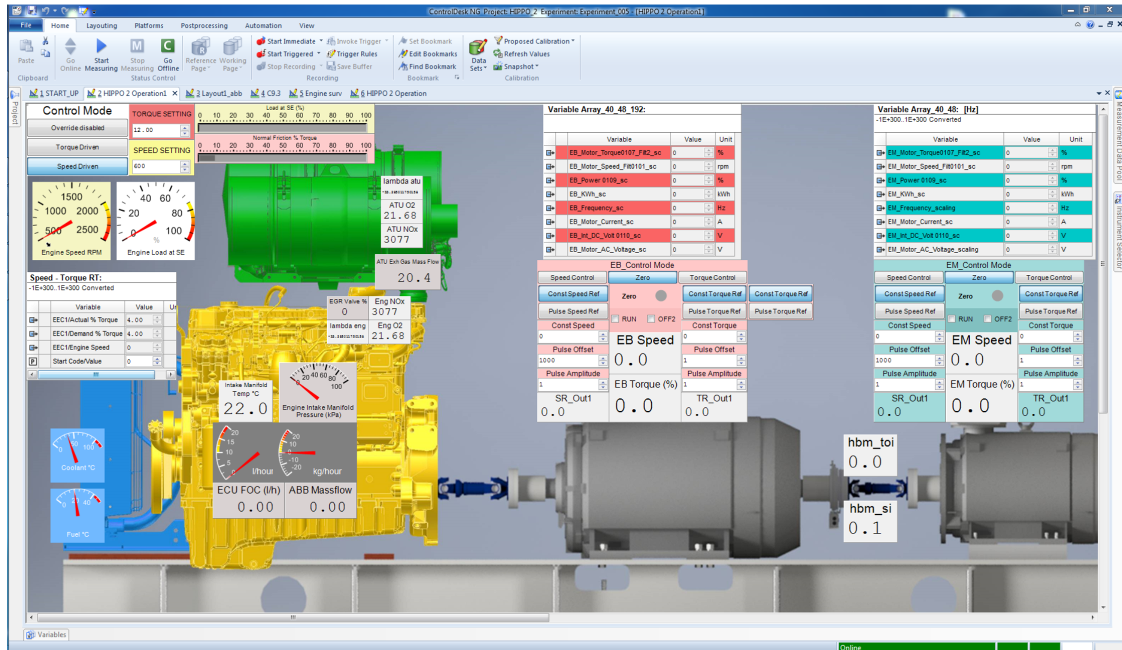


Figure 4.1: HIPPO-2 Control Interface

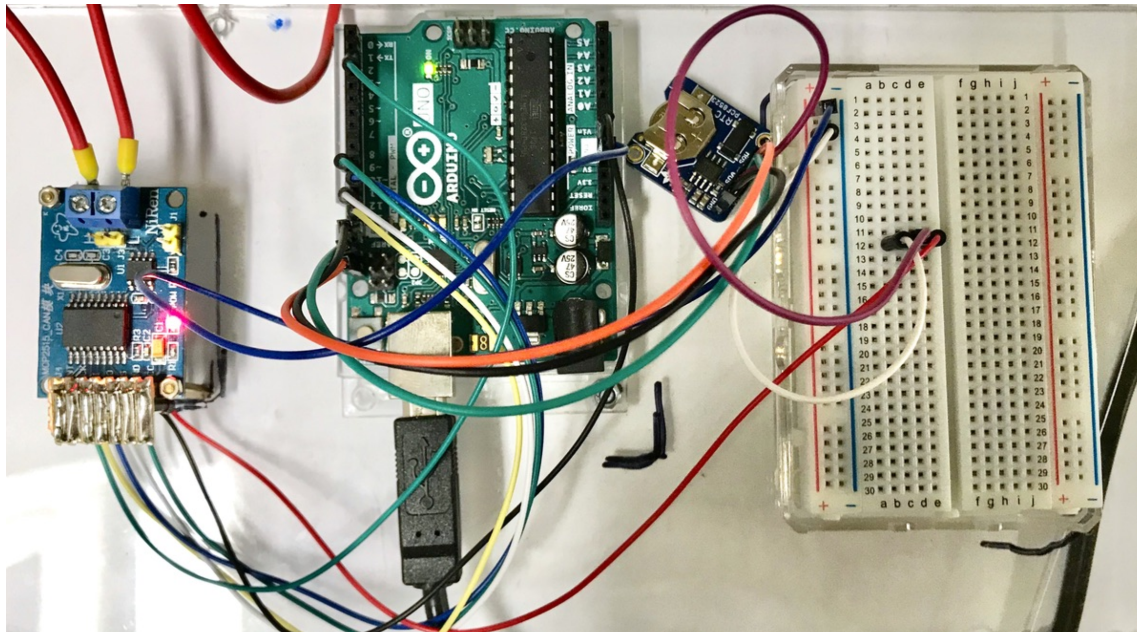


Figure 4.2: Arduino, MCP and RTC connections during the experiment

measurement were exported to text files which were then processed and plotted using Python.

To compare measurements received from both systems more easily, we have plotted them in the same canvas. For this comparison, we have chosen to showcase five parameters, Engine's Actual Percent Torque, Engine's Operating Speed, Fuel Consumption Rate, the Engine's Intake Manifold Temperature and the Engine's Intake Manifold Pressure. The purpose of this comparison was to test the accuracy of the Arduino and Raspberry setup compared to an already established and reliable system. This series of experiments was

ideal for such a task, as they provided us with rapid changes in the values of most of the parameters. The fact that we were examining variable loads helped us determine how accurately our system can record variations in the engines operating parameters.

Firstly we shall examine the measurements taken for the engine's fuel consumption rate. For the graph regarding the first experiment, Figure 4.3, we can see that the measurements taken by the Arduino and Raspberry set-up closely follow these of the dSpace Micro Autobox system. Since our system receives data at a slightly slower rate than the dSpace module, the measurements recorded in the graph are fewer for our system. This leads to the line interpolation deviating at certain points from the one plotted for the dSpace measurements. However, we can see that the points recorded by both systems coincide, meaning that our measurements where accurate. Moving on to the graph plotted during the course of the second experiment and shown in Figure 4.4, we can once again see that the measurements taken from both systems coincide with each other. This time, due to less rapid changes in the values of this parameter the lines plotted match more closely.

Figure 4.3: Comparison of measurements, Experiment No.1 - Engine's Fuel Consumption Rate

Now we examine the Engine's Operating Speed as captured during the experiments. This is a parameter that belongs to identifier F004, which has a very high trasmittion rate. Thus, our system has captures enough measurements to follow the dSpace data. This can be seen in Figure 4.5 where we have plotted the data for the first experiment. Both curves' data points coincide giving us an almost identical representation. The same is true for the graphs plotted for the second experiment as shown in Figure 4.6

Continuing, we will look into the data from the Engine's Actual Percent of Torque. In Figure 4.7 we have plotted the graphs for the first experiment. Measurements captured by our system and the dSpace configuration match each other very closely. This time, no major deviations can be seen in the graph. The data for this parameter is received at a very high rate, ensuring better interpolation of the curve than measurements which are transmitted at longer intervals. Regarding the second experiment, the plots for which are

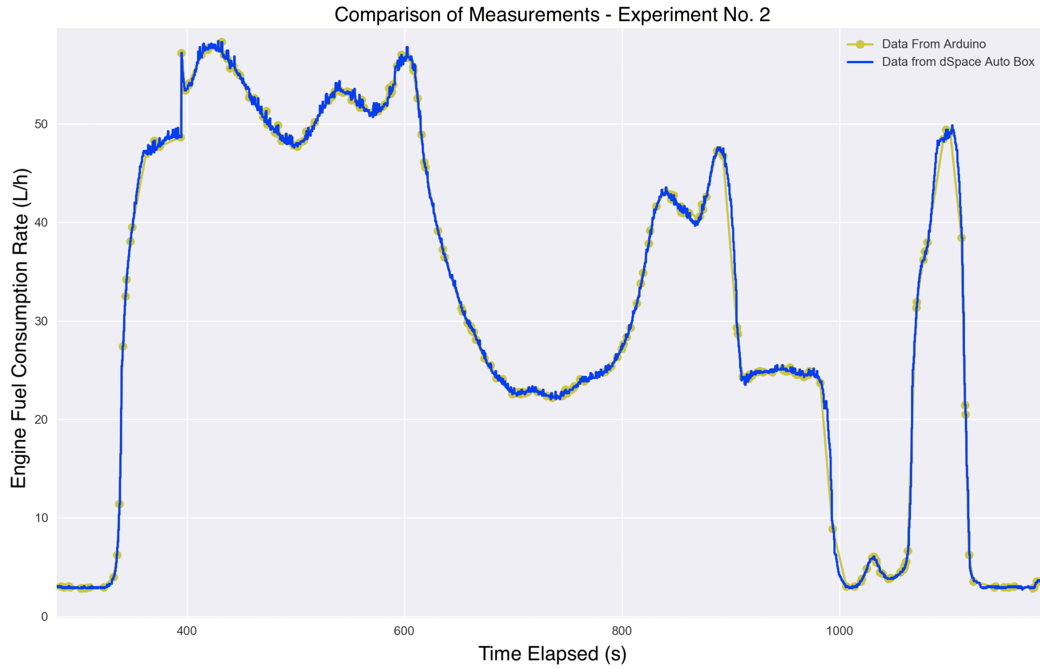


Figure 4.4: Comparison of measurements, Experiment No.2 - Engine's Fuel Consumption Rate

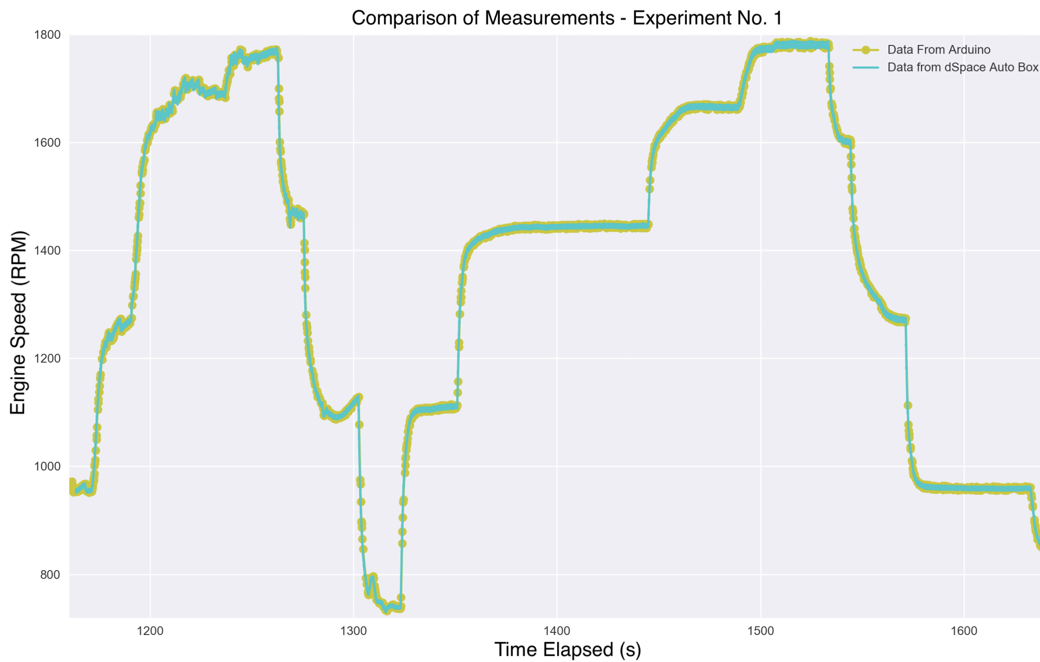


Figure 4.5: Comparison of measurements, Experiment No.1 - Engine's Operating Speed

shown in Figure 4.8, once again the measurement plotted by both systems mimic each very closely and no deviations in the data recorded is observed.

Moving on, we will be examining the third parameter plotted for the purposes of our comparison, the Engine's Intake Manifold Temperature. Measurements for the first

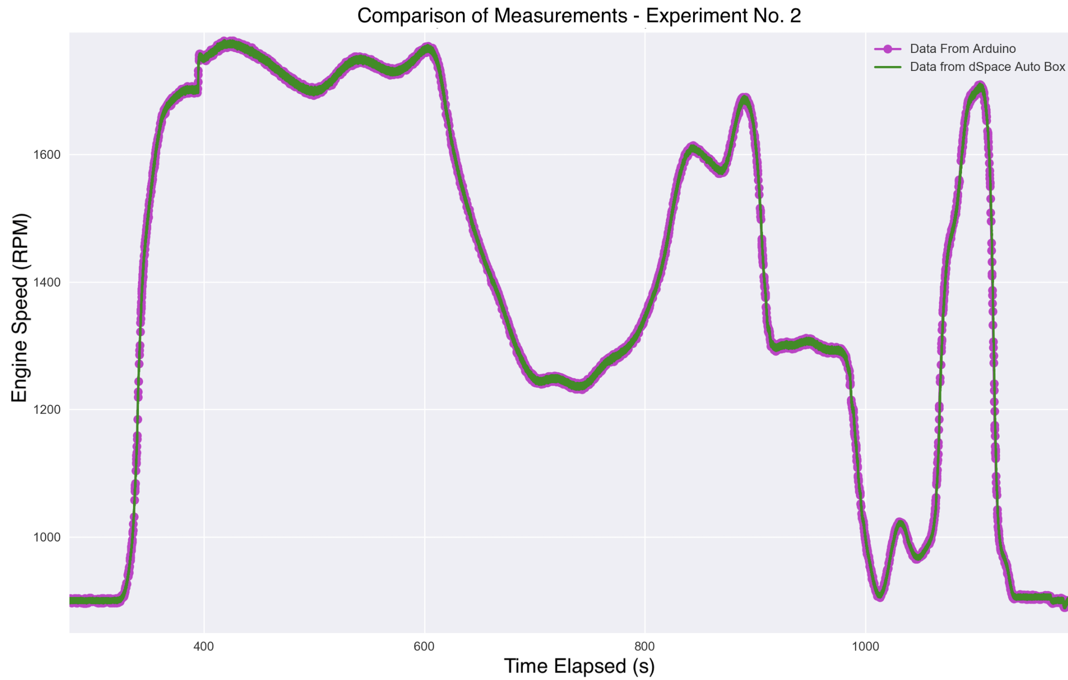


Figure 4.6: Comparison of measurements, Experiment No.2 - Engine's Operating Speed

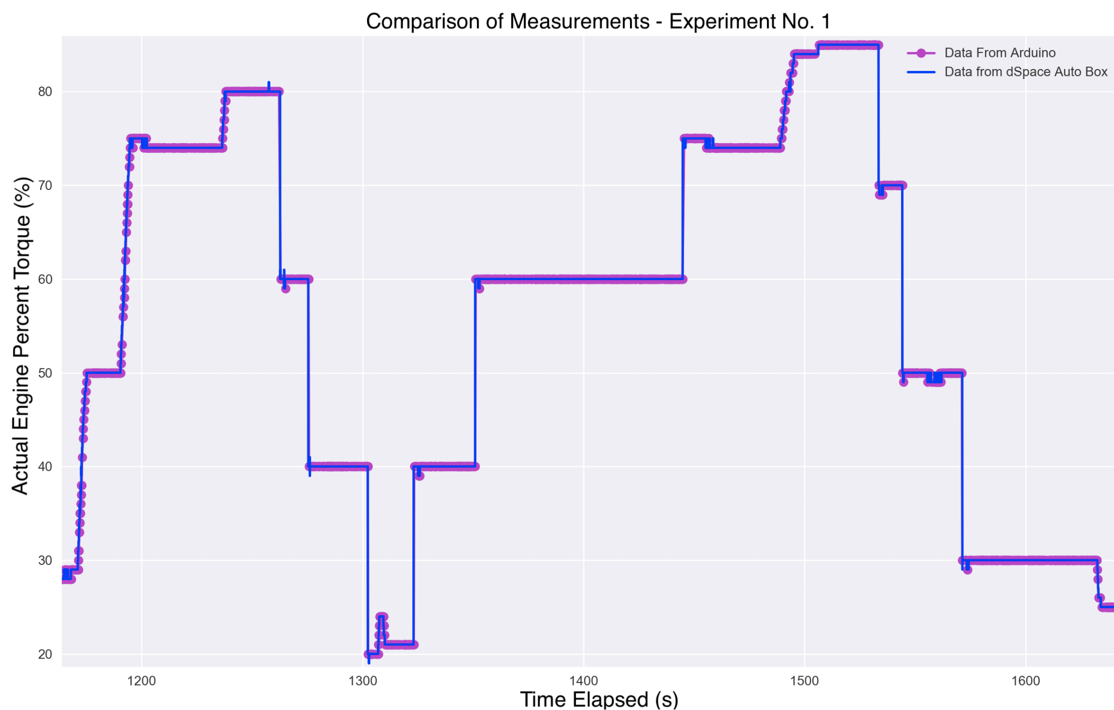


Figure 4.7: Comparison of measurements, Experiment No.1 - Engine's Actual Percent Torque

experiment have been plotted in Figure 4.9, where we can see that the data points recorded by our system match those of the dSpace module. The deviation we can see is due to the curve interpolation. Since this parameter is received at a slower rate by our system there are fewer points plotted on the graph, and thus the interpolation is not as accurate. The

Figure 4.8: Comparison of measurements, Experiment No.2 - Engine's Actual Percent Torque

same occurrence is apparent in Figure 4.10. Here we have plotted the measurements for the second experiment and we can see that both sets of data points match. Once again, due to the fact that we have recorded fewer measurements compared to the dSpace module, a small difference due to interpolation may be observed in the curves.

Finally, we shall compare the data captured for the Engine's Intake Manifold Pressure. For the graph showing data for our first experiment we can see that although the data points of our system and the dSpace module match, the former has capture fewer data points due to lower rate of data reception. This is especially apparent by the curve interpolation. As a result fluctuations in the value of this parameter have not been represented by our system. This phenomenon is less apparent in the graph for the second experiment due to the fact that the fluctuations of the value were less acute and therefore, even a smaller set of data was able to capture the progression of this parameter.

Conclusions

In summary, we can see that when monitoring parameters whose trasmittion rate is high, the resulting data presentation is very accurate and in-line with what is captured by the dSpace monitoring system. Even small fluctuations in the value of a parameter can be seen when plotting the data, giving us a very accurate image of the engine's actual operating condition. In the case of parameters that are transmitted at a lower rate, such as the Engine's Intake Manifold Temperature and Pressure data, there is a possibility of missing fluctuations in measurements during the operation, especially when their values change rapidly during a short period of time. For parameters those values progress more gradually however, this is not an issue, since fewer measurements represent their progression accurately.

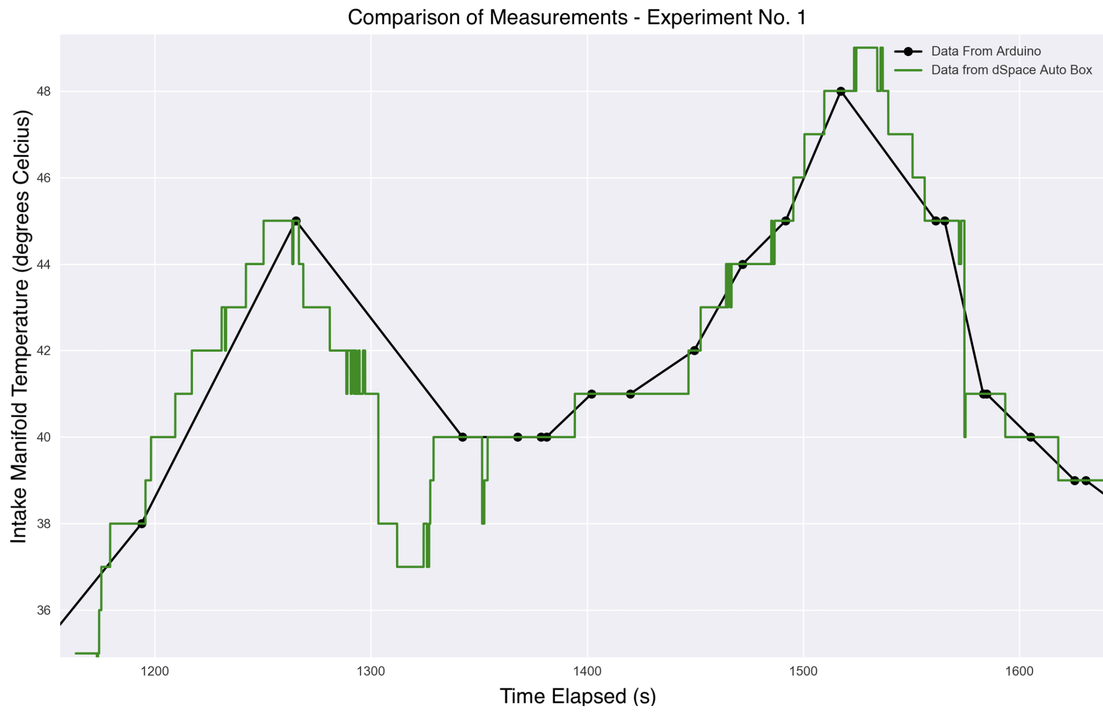


Figure 4.9: Comparison of measurements, Experiment No.1 - Engine's Intake Manifold Temperature

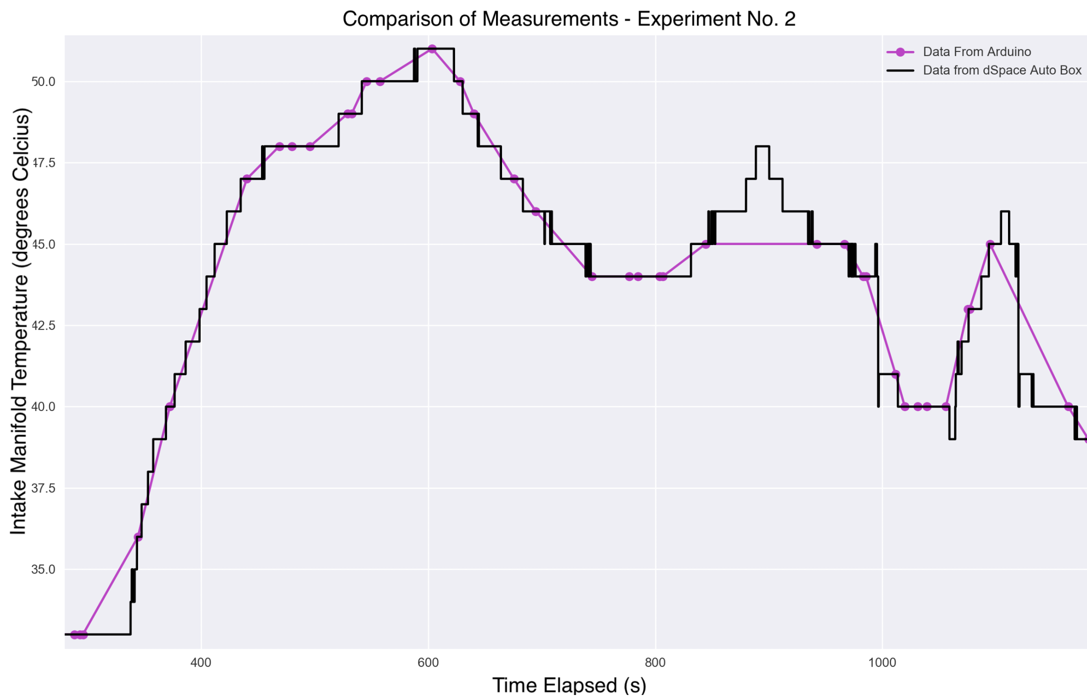


Figure 4.10: Comparison of measurements, Experiment No.2 - Engine's Intake Manifold Temperature

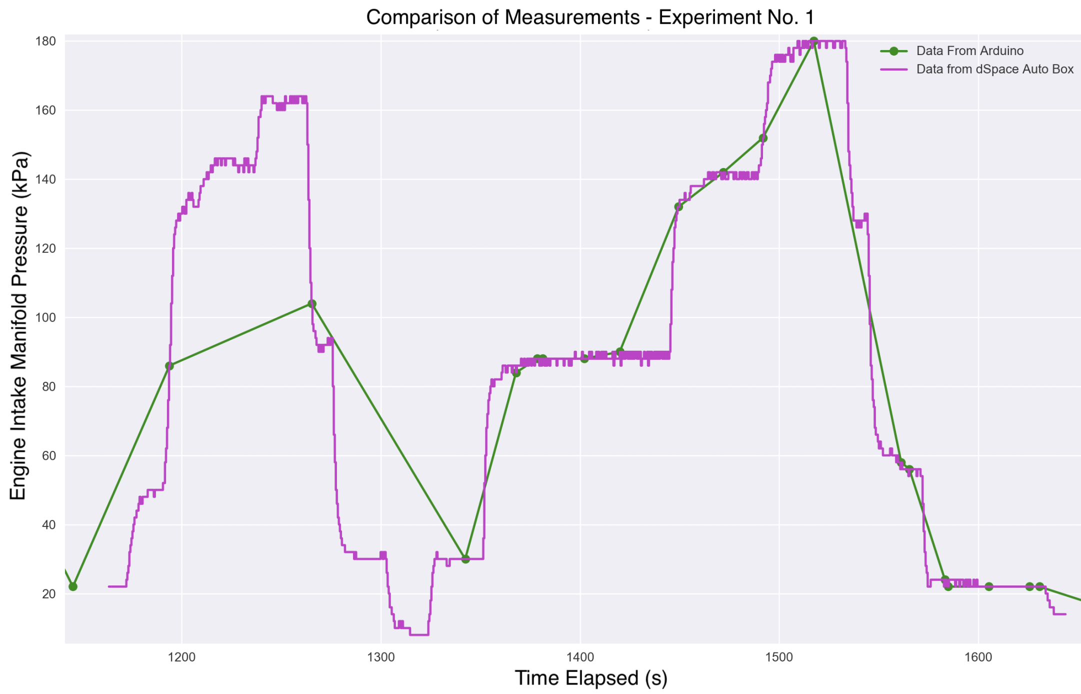


Figure 4.11: Comparison of measurements, Experiment No.1 - Engine's Intake Manifold Pressure

Figure 4.12: Comparison of measurements, Experiment No.2 - Engine's Intake Manifold Pressure

Chapter 5

Conclusion and Future Works

This project has managed to offer the following contributions in communicating with the engine testbed. Firstly, the establishment of a cheap, open source and portable system comprising of easy to use and customizable components that provided communication and control of the diesel engine. Secondly, the ability to filter and visual represent the desired operating parameters of the engine using live graphs.

5.1 Summary

During the development and implementation of this project we encountered several challenges. Regarding the choice of hardware set-up, we initially set off to develop the project around a combined single board computer and microcontroller, the BeagleBone Black. At first this equipment seemed more suitable for our application as it combined the capabilities of the Raspberry computer and the Arduino microcontroller in one device, thus requiring fewer connections and familiarization with one piece of hardware instead of two. However, in order to connect the BeagleBone to the engine, a CAN bus shield or cape was required. Shields or capes as they are often referred to are boards that can be mounted on top of the microcontroller board. The shield pins are inserted into the sockets located on the microcontroller board to expand its connections. Unfortunately, we were unable to receive a signal with this hardware configuration after connecting it to the ECU and so we moved on to develop our current setup.

In addition, a big challenge in the development of our application was the optimization of the software, so that it could handle large amounts of incoming data from the engine's ECU with the minimum use of processing power. Since hundreds of parameters are transmitted by the engine every second it was important to build an application that could sort the data while running on a computer with limited capabilities. Furthermore, the way the data was read by our system proved challenging in correctly converting it to the final value of each parameter. Since incoming data often seemingly had less bytes than the actual message we had to adjust our computer code to identify the points where data bytes had to be modified in order to get accurate results. This process involved identifying when the engine used the big and little endian orders as well as the exclusion of the numeral zero at the beginning of each byte.

5.2 Future Developments

Taking into consideration the unique set-up of the HIPPO-2 installation in conjunction with the open-source nature of the Can-bus and the hardware used for our application this project could expand in the following areas:

- Establishment of an online database with the purpose of accessing remotely the measurements recorded. Simultaneous real-time plotting of the data using a web based application to monitor the engine's operation at any location.
- Upgrade of the hardware components to include a more powerful single board computer, such as the Raspberry 4, in order to improve the graphics performance. This upgrade would also leave room for a more complex application to be developed, if desired.
- Use of more than one micro controllers and Can bus modules to monitor simultaneously data from multiple sources, either a second engine, the Aftertreatment system or any devices that transmits information using the Can bus protocol. In the case of data being monitored from multiple diesel engines of the same make and type, such as a ships auxiliary engines we would be able to compare and assess their performance at the same time.
- Testing of the possibility to communicate with the sensors and controllers of the diesel engine and the emission control system via Can-bus directly.

Bibliography

[1] Robert Bosch GmbH 1991 CAN Specification Version 2.0 $bosch - semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can2spec.pdf$

[2] Steve Corrigan SLOA101B–August 2002 Introduction to the Controller Area Network (CAN) *ti.com/lit/an/sloa101b/sloa101b.pdf*

[3] Sunil Kumar Reddy Gurram 2011 Thesis Implementation of Controller Area Network (CAN) bus in an autonomous All-Terrain vehicle

[4] https://www.can-cia.org/

[5] Pfeiffer, O.; Ayre, A. & Keydel, C. 2008. Embedded Networking with CAN and CANopen. *Copperhill Technologies Corporation.*

[6] Olaf Pfeiffer, Christian Keydel, Andrew Ayre 2005. CANopen on general serial networks. *CAN in Automation, iCC.*

[7] dSPACE. Technical Manual of dSPACE RapidPro and MicroAutoBox, 2013.

[8] CSS Electronics, www.csselectronics.com. *css electronics.*

[9] http://www.byteme.org.uk/canopenparent/canopen/

[10] User's Manual CANopen Adapter Module RCAN-01 - 3AFE64504231 REV B EN / EFFECTIVE: 16.12.2008

[11] Firmware manual - System Control Program of ACS800 ABB - 3AFE64670646 REV H EN / EFFECTIVE: 2014-06-02

[12] HBM Torque Flange T10F mounting instructions - A0608-13.1 en

[13] https://www.quora.com/What-is-the-difference-between-RS232-and-CAN

[14] https://www.maximintegrated.com/content/dam/files/design/technical-documents/white-papers/can-wp.pdf

[15] CAN Transceiver http://ww1.microchip.com

[16] CAN vs. RS-485: Why CAN Is on the Move, https://www.maximintegrated.com/content/dam/files/design/technical-documents/white-papers/can-wp.pdf

[17] Ship & Bunker: Effective Use of Big Data and Remote Monitoring https://shipandbunker.com/news/features/industry-insight/702741-effective-use-of-big-data-remote-monitoring

[18] ORBCOMM: The Connected Ship: Remote Vessel Monitoring Improves Performance While Reducing Costs https://blog.orbcomm.com/the-connected-ship-remote-vessel-monitoring-improves-performance-while-reducing-costs/

[19] SteveCorrigan. Texas Instruments: Introduction to the Controller Area Network (CAN)

[20] Hua (Walker) Bai, Dual-Channel, 42 V, 4 A Monolithic Synchronous Step-Down Silent Switcher 2 with 6.2 micro A Quiescent Current https://www.analog.com/en/design-notes/dual-channel-42v-4a-monolithic-synchronous-step-down-silent-switcher-2-with-6-2-a-quiescent-current.html

[21] Chris Clay Clay, Raspberry Pi: 11 reasons why it's the perfect small server https://www.zdnet.com/article/raspberry-pi-11-reasons-why-its-the-perfect-small-server/

[22] 5 Advantages of CAN Bus Protocol, https://www.totalphase.com/blog/2019/08/5-advantages-of-can-bus-protocol/