Troubleshooting Scania Vehicles, Marine and Industrial Engines with External Sensors

Oliver Kiffer
Abstract

Troubleshooting Scania Vehicles, Marine and Industrial Engines with External Sensors

Oliver Kiffer

One factor controlling a vehicle’s total cost is the time it is available to generate revenue for the customer, time spent in a repair shop decreases that amount. Scania vehicles are equipped with onboard diagnostics which produces fault-codes if the internal signals deviate from their intended values. These fault codes are not always enough to isolate the problem which means that the workshop staff are required to use external measuring systems to diagnose the vehicle. Combining data from internal sensors with an external measuring system can be problematic and the results inconclusive.

This thesis presents a solution where signals from external and internal sensors can be logged and analyzed together. A study on various hardware typologies are presented, key problems are identified and discussed.

A prototype was created based on the research done during this project. This prototype was successfully used to troubleshoot and find a problem on a test rig where an error had been purposely introduced, reducing the whole troubleshooting process to a fraction of the time it normally takes.

The hardware and software necessary to create this prototype is described in this thesis.
Abstract

One factor controlling a vehicle’s total cost is the time it is available to generate revenue for the customer, time spent in a repair shop decreases that amount. Scania vehicles are equipped with onboard diagnostics which produces fault-codes if the internal signals deviate from their intended values. These fault codes are not always enough to isolate the problem which means that the workshop staff are required to use external measuring systems to diagnose the vehicle. Combining data from internal sensors with an external measuring system can be problematic and the results inconclusive.

This thesis presents a solution where signals from external and internal sensors can be logged and analyzed together. A study on various hardware typologies are presented, key problems are identified and discussed. A prototype was created based on the research done during this project. This prototype was successfully used to troubleshoot and find a problem on a test rig where an error had been purposely introduced, reducing the whole troubleshooting process to a fraction of the time it normally takes. The hardware and software necessary to create this prototype is described in this thesis.

Populärvetenskaplig sammanfattning

## Contents

1 Introduction ................................................. 6  
  1.1 Statement of Originality ................................. 6  
  1.2 Background ............................................... 6  
  1.3 Goal ..................................................... 6  
  1.4 Scope ................................................... 6  
  1.5 Delimitation ............................................. 7  

2 Theory .................................................. 7  
  2.1 OSI model ................................................ 7  
  2.2 CAN ...................................................... 8  
    2.2.1 Architecture ......................................... 8  
    2.2.2 Physical layer ....................................... 9  
    2.2.3 Data link layer ..................................... 10  
  2.3 OBD-II and EOBD ......................................... 11  
    2.3.1 Diagnostic connector ................................. 11  
    2.3.2 Signal protocols ................................... 12  
  2.4 Modular Performance Analysis with Real-Time Calculus 12  
    2.4.1 Curves .............................................. 13  
    2.4.2 Components ......................................... 13  
    2.4.3 Analysis ............................................ 14  
  2.5 SPI ...................................................... 14  

3 Scania .................................................. 15  
  3.1 SPD3 .................................................... 15  
    3.1.1 BasAPP .............................................. 15  
    3.1.2 Scomm ............................................... 15  
    3.1.3 Vehicle computer interface .......................... 15  
  3.2 CAN ...................................................... 15  
  3.3 Common sensors ......................................... 16  
    3.3.1 Temperature sensors ................................. 16  
    3.3.2 Pressure sensor ..................................... 16  
    3.3.3 NOx sensors ......................................... 17  
    3.3.4 Other smart sensors ................................. 17  

4 Background interviews .................................... 17  
  4.1 Case I - Marine and Industrial Engine ................... 17  
  4.2 Case II - Exhaust after-treatment system ............... 18  
    4.2.1 Exhaust emission control system problems ......... 18  
    4.2.2 Exhaust measuring system .......................... 18  
  4.3 Case III - Fuel system .................................. 18  
  4.4 Requirements ........................................... 19  

5 Ad hoc or Existing solution? ............................... 19  
  5.1 Key factors ............................................. 19  
  5.2 Already existing solution ............................... 20  
    5.2.1 Ipetronik ............................................ 20  
    5.2.2 Dewe ............................................... 21  
    5.2.3 Cheaper already existing solutions ............... 21
10 Results
10.1 Testing and Verification ................................................................. 63
10.2 Problem testing ........................................................................... 66
10.3 Response time analysis ............................................................... 69

11 Discussion
11.1 Test and verification ..................................................................... 70
11.2 Problem testing ........................................................................... 70
  11.2.1 Test 1, open valve and no dosing ............................................ 70
  11.2.2 Test 2, choked valve and no dosage apply ............................... 70
  11.2.3 Test 3, choked valve and dosage apply .................................... 70
  11.2.4 Test 4, closed valve and no dosage apply ............................... 70
  11.2.5 What to make from tests and how does the external sensor help ................................................................. 71
11.3 Response time testing ............................................................... 71

12 Future work
12.1 Hardware ................................................................................. 73
12.2 Software .................................................................................... 73
12.3 Recommended future setup ....................................................... 73

13 Conclusion ................................................................................. 74
Glossary

Ad hoc  Latin phrase -"for this". Ad hoc solution signifies solutions for a specific task. 20

CAN  Controller Area Network. 8

COO  Coordinator System, Coordinates traffic between internal CAN buses. 56

DIP  Dual in line package, common component casing with two parallel rows of pins. 31

ECU  Electronic Control Unit. 15, 56

EMC  Electromagnetic compatibility. 33

EMI  Electromagnetic interference. 33

FIFO  First In First Out, a type of buffer. 56

Full duplex  allows communicating devices to send and receive data at the same time in both directions. 14

GUI  Graphical User Interface. 43

IDE  Integrated Development Environment. 27

MPA  Modular Performance Analysis. 12, 56

NOx  Nitrogen oxide. 16

NTC  Negative temperature Coefficient. 16

RTC  Real-Time Calculus, Matlab toolbox. 56

SBC  Single board computer. 28

SDP3  Scania Diagnose and Programmer 3. 56

VCI  Vehicle computer interface. 15

VCI  Virtual CAN Interface, used to connect computer with OBD-port. 22

XML  Extensible Markup Language. 53
1 Introduction

1.1 Statement of Originality

This thesis was written by two people from different programs in Uppsala University. The project was divided into thirds where each author contributed with one third each and the last part is a joint effort. Oliver Kiffer has been focusing on problems relating to hardware while Paul Norström have worked mostly with software problems.

- Oliver: section 6.2, 7
- Paul: section 8, 10.3, 11.3
- The rest have been combined work.

1.2 Background

One of the factors controlling a vehicles total cost is availability, which means the time a vehicle is available to generate revenue for the customer. Time spent in a shop decreases the amount of available time for that vehicle. Scania strives to reduce the amount of time spent in a shop and one way of doing that is to reduce the time it takes to troubleshoot a vehicle that has been affected by an error.

Scania’s vehicles are equipped with onboard diagnostics, which simplified is a series of tests run by the control unit which evaluates the signals from the sensors that are used to regulate the vehicles system. If the signals deviate from the modeled set point values a fault code is generated. At best this fault code will pinpoint the error. However the onboard diagnostics information is not always enough to isolate the problem into a replaceable component or to discover an error at all.

To add more sensors to the system is not an option because it increases the initial cost of the vehicle as well as adds the possibility of another source of error. The workshop staff is thereby required to use external measuring systems to check the vehicle when the vehicles own diagnosis is not sufficient.

1.3 Goal

To find and evaluate the best platform and solution to connect external sensors to the Scania vehicles in order to get extra parameters during analysis and troubleshooting of the vehicle. The external sensors should be placed so they can narrow down the troubleshooting process and at the same time they should also work as an analyzer for the internal sensors.

1.4 Scope

The scope of this thesis is to research the following:

- Investigate possible hardware solutions and their impact on the problems mentioned below.
- Investigate possible system architectures and their impact on the following problems.
• Investigate possibilities to compensate for delays that may occur in the internal diagnostic communication.

• Investigate possibilities to compensate for delays that may occur in the external communication.

• Investigate the possibilities of synchronizing external and internal signals.

• Investigate how to configure the signal set.

• Optimize the system in terms of modularity, performance, cost and maintainability.

and to construct:

• A simple prototype that exemplifies the research done.

1.5 Delimitation

This study does not cover the following:

• Develop software for recording and displaying in the diagnose programme (SPD3).

• Develop software for signal filtering.

2 Theory

2.1 OSI model

A communication system is a collection of individual communication networks, relay stations, transmission systems and data terminal equipment combined to an integrated whole. The Open System interconnection model (OSI) characterize and standardizes the internal functions of a communication system in form of a conceptual model. This means that the OSI model does not have any function, but makes it easier to understand the complex interactions that are happening.

The model consist of seven layers. Control is passed from the top layer and proceeds down to the bottom layer, over the physical link to the next station and upward the hierarchy again.

[15]:

1. **Physical layer**- handles the transmission and reception of the unstructured raw bit stream over a physical medium. This means that the first layer describes the electrical, optical, mechanical and functional interfaces to the physical medium and carries the signals for all of the higher layers.

2. **Data link layer**- package raw bits from the physical layer into frames. A frame can be seen as logical, structure packet for data. Because the frames are made in here this layer is also responsible for transferring from one computer to another without errors. The data link layer waits for an acknowledgment after sending away a frame from the receiving part.
3. **The Network layer** - handles the routing of the data. Such as routing from the source to the destination computer and handles traffic problems (flow control). It also has a key word in addresses, the layer addresses messages and translate logically the names into physical addresses.

4. **The Transport layer** - guarantee complete data transfers. It manages the end to end control (determines if an packet has arrived etc.) and error checking. This layer also include error recognition and recovery.

5. **The Session layer** - allows two or more computers to establish, use and end a connection/session. During a session the layer regulates which side transmits and for how long.

6. **The Presentation layer** - is sometimes called a syntax layer. The layer converts incoming and outgoing data from one presentation format to another. Also manages security issues by encryption and compression functions. This layer may often be included in the operating system. §

7. **Application layer** - The top layer where communication partners and quality of service is identified. Also user authentication and privacy are regarded. Any constraints on data syntax are identified. The layer represents the services that directly support application such as software for file transfers, database access and email.

2.2 CAN

The digital CAN (Controlled area network) bus is often used in vehicles as a standard bus protocol. The bus protocol lets microcontrollers and devices communicate with each other without having a host computer. The reason the CAN bus protocol is so attractive to use in vehicles, robotics, factory controls and medical devices as its embedded system solution is because of its low cost, light protocol management and error detection/re-transmission feature built into the system. The CAN system is standardized for different application speeds, but is designed to operate at speeds from 20 kbit/s to 1 Mbits/s. The transmission rate depends on the physical bus length and transceiver speed [7].

2.2.1 Architecture

The CAN bus is a serial bus with many bus masters, also known as an multi-master serial bus. Every electronic control unit is connected to the same bus with each two nodes. This makes it impossible to send a message to just a specific node, all the nodes picks up all traffic and listens to all transmissions. Each node connected to the bus requires three main components [3]:

- **Host processor in form of a:** CPU, microcontroller or host computer. The "brain" of the node that decides what the received message mean and what messages it wants to transmit. This is the place where sensors, actuators and control devices are connected in the node.

- **CAN controller.** When receiving serial bits from the bus, the controller stores it all until an entire message is available and then the host processor triggered by an interrupt fetches the message. When sending a message,
the CAN controller gets an message from the host processor and transmits the bits serially into the bus whenever the bus is free. Usually this component is integrated into the microcontroller.

- **Transceiver.** When receiving data stream from the CAN bus the transceiver converts layer levels from the bus to levels that the CAN controller uses. When transmitting data stream from the CAN controller the transceiver converts from CAN controller level to CAN bus levels. The transceiver is defined by ISO 11898 standard.

![Figure 1: The ISO 11898 network CAN with nodes [3].](image)

### 2.2.2 Physical layer

The original CAN specification developed by Robert Bosch Gmbh in 1986 describes the CAN bus protocol at the physical and data link layer of the ISO/OSI model. The physical layer is used to define how electrical signals are encoded as bits and how the bits are transmitted from one node to another. The Bosch CAN standard however omits the specification of the physical transmission medium, connectors, physical wiring and driver/receiver stages and instead limits itself to specifying only bit timing, bit encoding and synchronization [7, p. 1].

**Encoding** The stream of bits on the CAN bus line is coded with the Non Return to Zero method (NRZ). In the NRZ method, ones are represented by a significant condition such as positive voltage and the zeros are represented by another different significant condition such as negative voltage. This means that in NRZ there is no neutral or rest state. The two states are divided into different definitions, the recessive and the dominant state. Due to that the CAN protocol allows multi-master access to the bus and if multiple masters try to drive the bus state at the same time the dominant configuration prevails the recessive. [7, p. 3] This method has both advantages and disadvantages. A good thing is to require only a minimum bandwidth of the signal transmission. But during long periods of constant values problems with synchronization leads to faulty bit detection since each node in the network has its own clock generator. To prevent the disadvantages, the technique of bit-stuffing is introduced to the bit stream. [6, p. 4]
Synchronization  In order for transmission to work all nodes on the bus needs to agree on the current value of the bit being transmitted and this requires that they are all synchronized on the bit edges. This is done by using a protocol that keeps the receiver’s bit rate aligned with the rate of the transmitted bits. Synchronization starts with the transition of a recessive to dominant bit level, IE the bus goes from idle to the start bit of the frame. Each node resynchronizes on every recessive to dominant transmission during the frame.

To make sure that synchronization is maintained during transmission bit stuffing is used. Bit stuffing ensures that for every five consecutive bits of the same type (bit level), a complementary or inverse bit is automatically inserted by the transmission node. This bit is removed by the receiving node before it processes the frame. All fields in the frame are stuffed with the exception of the CRC delimiter, ACK field and end of frame they have a fixed size and are not stuffed. Whenever six consecutive bits of same type occurs a node can send an active error flag [7, p. 2-5].

2.2.3 Data link layer

CAN Frame  CAN uses four different types of frames each designed for a specific purpose: Data frames which carry the data from a source node on to the bus, where the frame is either broad-casted to all receivers on the bus or targeted at a particular node using an identifier. Data frames are the only type of frame which carry actual data. Remote frames are used to request data from a source node. Error frames are sent whenever any node on the bus detects an error. Lastly, the overload frames are used for flow control and synchronization.

A frame consists of several fields with specific length and purpose, Figure 2 depicts a complete CAN frame. Each frame starts with a Start-of-frame bit which is always a dominant 0. Following the start bit is the arbitration field containing the identifier (encoded message priority) and RTR (remote transmission request) which indicates whether the message is a remote or data frame. The control field consists of the IDE bit (identifier extension bit) which is used to declare if the frame is of the extended format with a 29 bit identifier or the standard 11 bit, a reserved bit (set to dominant 0) and the DLC (data length code) which indicates how many bytes of data the message consists of the 4 bit field. The actual payload or data of the message is in the data field and can be up to 64 bits long (0-8 bytes). The CRC field (cyclic redundancy check) contains the checksum for the preceding bits and used for fault detection. The CRC is 16 bits long where the last bit is used as a delimiter and must be recessive (1). The ACK field contains the ACK (acknowledgment) slot and a
delimiter (must be recessive), the transmitter sets the ACK to 1 (recessive) and expects that any node receiving the message will overwrite the recessive level to dominant and thus confirming that a correct CAN frame has been received. Lastly the EOF field (end of frame) is sent and consists of 7 bits, all recessive which is an intended breach of the bit-stuffing mechanism to mark that this is the end of the data frame. The following 3 bits after the EOF is called IS (idle space) or IF (interframe bits) and they are used to separate a frame from another one. [7, p. 13-16] [6, p. 4-8]

**Bus Arbitration**  The CAN protocol is priority based, with the message priority encoded into the frame. Access to the bus is given to the node with the highest priority (lowest ID) and once it starts transmitting it cannot be preempted. Whenever a node wanting to transmit finds the bus in idle state it can start the arbitration phase by sending a start of frame bit. At this stage any other node also wanting to transmit can start competing for the bus. After synchronizing on the start of frame bit they each starting transmitting their identifiers (priorities) while listening to the bus. The identifiers are transmitted serially with the MSB (most significant bit) first. The CAN bus can be thought of as a wired AND channel connecting all the nodes, this means that if two nodes are sending their identifiers and both of their first bit is 1 (recessive) the level on the bus will be 1, but if one of them sends a 0 (dominant bit) the level on the bus will be 0. The node that sent a 1 will notice that the level of the bus is not the same as the one it sent realizes that it lost the arbitration and will go back into listening mode instead. [7, p. 17-18] [6, p. 8-9]

### 2.3 OBD-II and EOBD

OBD stands for onboard diagnostics, its a system designed to give a vehicle owner or repair technician access to the status of various vehicle subsystems. This diagnostic status can be error codes generated by a sensors which has detected one or several malfunctions in various vehicle parts or it could be real time data such as the current vehicle speed. The amount of information available through OBD has varied greatly since its conception and there has been several standards. This report will focus mainly on OBD-II and the European version, EOBD (European onboard Diagnostics) which essentially has the same technical implementation. The OBD-II standard defines a list of DTC’s (Diagnostic Trouble Code), the type of physical connector to use as well as the pinout, the electrical signaling protocol and the message format. Although the OBD-II standardization originally was prompted by emission requirements, nowadays most vehicles use the OBD-II DLC (Data Link Connector) connector as a port to all of the vehicles system where they can be diagnosed and programmed and not only to the required emission related codes [21].

#### 2.3.1 Diagnostic connector

The OBD-II standard requires the connector to be a female 16 pin of type J1962 and it needs to be located within 2 feet of the steering wheel. The pinout of the connector is specified according to Table 1, roughly half of the signals are left to the manufacturers discretion [21].
Table 1: OBD-II Diagnostic connector pinout

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unassigned</td>
</tr>
<tr>
<td>2</td>
<td>Bus Positive Line of SAE J1850 PWM and VPW</td>
</tr>
<tr>
<td>3</td>
<td>Unassigned</td>
</tr>
<tr>
<td>4</td>
<td>Chassis ground</td>
</tr>
<tr>
<td>5</td>
<td>Signal ground</td>
</tr>
<tr>
<td>6</td>
<td>CAN-High (ISO 15765-4 and SAE J2284)</td>
</tr>
<tr>
<td>7</td>
<td>K-Line of ISO 9141-2 and ISO 14230-4</td>
</tr>
<tr>
<td>8</td>
<td>Unassigned</td>
</tr>
<tr>
<td>9</td>
<td>Unassigned</td>
</tr>
<tr>
<td>10</td>
<td>Bus Negative Line of SAE J850 PWM Only (not SAE J1850 VPW)</td>
</tr>
<tr>
<td>11</td>
<td>Unassigned</td>
</tr>
<tr>
<td>12</td>
<td>Unassigned</td>
</tr>
<tr>
<td>13</td>
<td>Unassigned</td>
</tr>
<tr>
<td>14</td>
<td>CAN-Low (ISO 15765-4 and SAE J2284)</td>
</tr>
<tr>
<td>15</td>
<td>L-Line of ISO 9141-2 and ISO 14230-4</td>
</tr>
<tr>
<td>16</td>
<td>Battery voltage</td>
</tr>
</tbody>
</table>

2.3.2 Signal protocols

The OBD-II standard permits five signaling protocols:

- SAE J1850 PWM, uses pulse-width modulated signals with a data rate of 41.6 kB/s
- SAE J1850 VPW, uses variable pulse-width modulated signals and has a data rate of 10.4 or 41.6 kB/s
- ISO 9141-2, uses a single bi-directional line for asynchronous serial data communication with a data rate of 10.4 kB/s.
- ISO 14230 KWP2000 (Keyword Protocol), uses bi-directional serial communication on a single line called K-line with an additional line called L-line for wakeup, it has a data rate between 1.2 and 10.4 kilobaud.
- ISO 15765 CAN (see page 8 for more info)

See Table 1 on page 12 to see on which pins each protocol can operate on the OBD-diagnostic connector [21].

2.4 Modular Performance Analysis with Real-Time Calculus

Modular Performance Analysis (MPA) is a method for analysis the best and worst case of real time systems. It uses Real-Time Calculus (RTC) as mathematical basis [17].

This section will provide a brief description of the basics of MPA.
2.4.1 Curves

In a distributed embedded system components communicate via message passing. This data message exchange between components can be abstracted by means of event streams. An input event for a component represents the arrival of a message and an output event the sending of a message. In Real-Time Calculus, event streams are abstracted from the time domain and represented in the interval domain by means of so-called arrival curves.

**Definition 1.** [Arrival Curve] Let \( R_e(s, t) \) denote the number of events that arrive on an event stream in the time interval \([s; t)\). Then, the corresponding upper and lower arrival curves are denoted as \( \alpha^u \) and \( \alpha^l \), respectively, and satisfy

\[
\alpha^l(t - s) \leq R_e[s,t] \leq \alpha^u(t - s), \forall s < t
\]

where \( \alpha^u(0) = \alpha^l(0) = 0 \).

In MPA an event stream is modeled with a PJD model, where \( p \) is the period and \( j \) is the maximum jitter with respect to the ideal periodic arrival time and a minimum inter-arrival distance \( d \) between any two events. The upper and lower arrival curves of \( \alpha^u \) and \( \alpha^l \) of a PJD event stream model with parameters \( p, j \) and \( d \) are computed as follows

\[
\alpha^l(\Delta) = \left[\Delta - \frac{j}{p}\right], \alpha^u = \min\left\{\left[\Delta + \frac{j}{p}\right], \left[\frac{\Delta}{d}\right]\right\} \Delta > 0
\]

In a similar way, the availability of processing or communication resources is represented by means of service curves:

**Definition 2.** [Service Curve] Let \( C[s, t) \) denote the number of events that a resource can process in the time interval \([s,t)\). Then, the corresponding upper and lower service curves are denoted as \( \beta^u \) and \( \beta^l \), respectively, and satisfy

\[
\beta^l(t - s) \leq C[s,t] \leq \beta^u(t - s), \forall s < t
\]

where \( \beta^u(0) = \beta^l(0) = 0 \).

It’s important that \( \alpha \) and \( \beta \) are expressed in the same unit, the occurring of events \( R \) and the availability of resources \( C \) can either be expressed in event units or as resource units. The simplest resource model is a fully available resource which corresponds to a pair of service curves \( \beta^l \) and \( \beta^u \) with slope 1.

This thesis only focuses on communication tasks and therefore events mean messages and resources denote available bandwidth.

2.4.2 Components

In real-time systems, tasks process event streams by using resources. In MPA tasks are represented as RTC components, which take arrival curves \( \alpha \) and a service curve \( \beta \) as inputs and calculate the outgoing arrival curves \( \alpha' \) and an outgoing service curve \( \beta' \). There are several components defined within the MPA framework, in this thesis however only one component is used, which is the Greedy Processing Component (GPC).
**Greedy Processing Component**  The GPC component is the simplest component, it has a single arrival and service curve as input. It models the case where only one task is processed at the time using only a single resource. It processes incoming events in a greedy fashion in first-in-first-out order while being bounded by the resource availability described by the service curve $\beta$. Its used as a building block for many other components.

![Figure 3: Greedy processing component [17].](image)

2.4.3 Analysis

To model a larger system, the output of one component can be provided as input to the next component and build a task sequence where each event takes its toll on the available resource [17].

2.5 SPI

Serial Peripheral interface (SPI) is another type of serial communication protocol. The SPI bus interface consist of four logic signals and uses an principle of master and slave devices. The master device initiates and dictates the connection and controls it. The neat thing with with Master/slave architecture is that to every master devices plural slave devices can be connected. After the initiation both the master and the slave/slaves can transmit and receive data. SPI devices communicate in Full duplex mode. The SPI uses four pins for communication.

- MOSI-Master output, Slave in.
- MISO-Slave input, Master output.
- SCK - Serial clock, output from Master.
• SS - Slave select, output from Master.

This is later used and mentioned in section 7.2.3 and figure 18.

3 Scania

This section is dedicated to describe how certain things are implemented at Scania or describe Scania specific systems or components.

3.1 SPD3

Scania Diagnos and Programmer 3 has been developed to support communication with Scania vehicles and Scania industrial and marine engines. The program is used for troubleshooting, adjusting customer parameters, calibrations, conversions affecting the electrical system and updating software in control units. In order to use the program a USB-key and a VCI is needed.

3.1.1 BasAPP

SPD3 is created by several sections within Scania, BasAPP is the name of the group of people who are responsible for the GUI and framework of SPD3.

3.1.2 Scomm

Scomm is created by another group that works with SPD3 and they create a communication interface between the vehicle and the PC it is hooked up to. Scomm is used by several different internal programs at Scania.

3.1.3 Vehicle computer interface

The Vehicle computer interface VCI is the interface that is used between the vehicle or industrial and marine engine and the computer. VCI stands for (Virtual CAN interface) and is a product made for Scania by a company called Kvaser. The VCI has an OBD port in one end and a USB port in the other.

3.2 CAN

Scania’s trucks and buses like all modern vehicles, uses the internal network CAN bus for all the systems of the car to communicate with each other, this means for example that the breaks can talk to the engine for stability control etc. Everything connected to the CAN bus can send information and are always listening to the CAN bus with help of their own ECU. To say that every system are connected to each other is not quite true and may be an simplification. The fact is that Scania’s different systems are connected and categorized in three different CAN buses, red, green and yellow, see Figure 4. Each of the three CAN buses are then connected to the same Coordinator, so even if all the sensors in the truck may not be connected to each other they are connected by indirect means to each other. The Coordinator receives all the messages from every CAN bus and sends away requests from one systems to another. The red CAN bus handles the most important stuff such as Brake systems and Gearbox...
systems. All the ECU’s on the red CAN bus has the highest priority, that is the lowest ID. After that in priority comes the yellow CAN bus that handles systems as the locking and alarm systems and all wheel drive systems. The lowest priority CAN bus to the coordinator is the green one. Here systems as heating systems and climate control systems exist.

Figure 4: Scania CAN bus architecture

3.3 Common sensors

There exist a lot of different types of sensors in Scania’s vehicles that are monitoring different systems. By focusing on the cases in section 4 this project main focus lays in three types of sensors. Temperature sensors, pressure sensors and NOx sensors.

3.3.1 Temperature sensors

Maybe not all temperature sensors in Scania’s vehicles are the same, but all the sensors covered in this report consist of NTC temperature sensors. An NTC temperature sensor consist of pure metal oxides. They respond quickly to temperature changes and even a small temperature increase causes their resistance to decrease significantly. By placing an NTC temperature sensor in a potential divider circuit with an pull up resistor an analog voltage value can be read and then converted into temperature.

3.3.2 Pressure sensor

All pressure sensors involved in this project are absolute pressure sensors. Which means the sensors measures pressure relative perfect vacuum. This means that the higher pressure that are induced on the sensor the lower their resistance. By placing a pressure sensor in at potential divider circuit with pull down resistors an analog voltage value can be read and then converted.
3.3.3 NOx sensors

NOx sensors (see Figure 5) are primarily found in the exhaust chain to control the proper emission and their function is to monitor the NOx conversion efficiency and regeneration of the catalyst material. The sensors consist of two chambers that read the oxygen concentration with help of electrodes and then convert the NOx to nitrogen. The sensors send an output voltage signal that are directly proportional to NOx concentration in the sensor [1].

![NOx Sensor](image)

Figure 5: An NOx sensor used to control NOx conversion efficiency and regeneration of catalyst material.

3.3.4 Other smart sensors

Besides the sensors mentioned above there also exist sensors that are "smart". A smart sensor consists of many different kinds of sensors (for example a sensor that can measure both pressure and temperature) implemented into one and have a own microcontroller inside of it that converts the output value to voltage.

4 Background interviews

This section covers information obtained through interviewing people at Scania who are in charge of the diagnostic part of various systems. This research was conducted to get a deeper understanding of the underlying problem.

4.1 Case I - Marine and Industrial Engine

When installing a new marine or industrial Scania engine a lot of measurements have to be made in order to make sure that the engine has been installed correctly and functions properly. Many of these measurements require external sensors and the solution for implementing these are usually Ad hoc. This means that the data extraction requires at least two systems, one being SPD3 and the other being an Ad hoc solution. It would be preferable if the external and internal sensors could be synchronized and displayed in SPD3.

With a working product the customer should be able to troubleshoot the engine by themselves and should not be dependent on Scania. The external sensor should be able to measure values in places where there are no internal sensors. This will also help during troubleshooting. Optimal additional insertions/attachments should be added to the engine so external sensors easily can
be attached during test runs. About 20 different places needs additional sensors to measure values. The prioritized external receivers is the temperature and pressure sensors. Wireless connection is not necessary, but would be nice.

4.2 Case II - Exhaust after-treatment system

Scania's trucks uses Selective catalytic reduction (SRC) in their exhaust chain. In a brief explanation of the SCR system, a reductant substance (Adblue in this case) is added to the exhaust gas and is adsorbed onto a catalyst and makes the byproduct carbon dioxide. This of course, for environment reasons. When the reductant substance is added to the exhaust, the mixture goes through a catalyst. During the hole exhaust process, the temperature and pressure in the system is essential for the system to work. The Exchaust and after-treatment system uses CAN sensors to measure NOx levels.

To more easily understand and explain, the system can be divided into two parts. The first is the exhaust engine control system which calculate the dosage of Adblue that should be outputted to the system. The second system is the exhaust value measuring system were all the NOx sensors is located.

4.2.1 Exhaust emission control system problems

First thing off, only to get to the exhaust emission control system is generally quite difficult and many other components of the truck has to be removed just to reach it. The problem in the exhaust emission control system is that it only exist one internal sensor that can measure the pressure in the dosage unit. If the sensor alarms that it is a problem in the system, it is quite hard to locate where in the system the problem is right away. Today workshops have different workshop methods they preform in a specific order to exclude components until they reach the problem. But even with the workshop methods its hard to narrow down the problem to exactly what seems to be the problem with the failing component of the system. This often leads to the whole component being replaced instead for example only the filter in the component.

4.2.2 Exhaust measuring system

The problem with the exhaust measuring system is that the NOx sensors are very fragile. This leads to that sometimes that broken sensor will measure wrong values and indicate a problem in the system even if there is no problem. The NOx sensors are also expensive so workshops tend to troubleshoot the whole system before replacing the NOx sensor.

4.3 Case III - Fuel system

Like most of the system in Scania's vehicles, the exhaust system is very complex. One of the key components to the system that is not really in the system is the accumulator. The accumulator helps keeping two types of pressures in the system line: High pressure from the fuel tank to the fuel injector and lower pressure leading the fuel back again if any remaining fuel that were not absorbed. If the accumulator fails the whole vehicle gets problems due to abrupt lack of fuel flow, so every accumulator have to clear a performance test before the
vehicle is shipped from the factory. One initialization test the accumulator have
to clear is that the leakage from the system have to be under 300 bar after 30
seconds, an healthy accumulator leaks about 25 bar. The troubleshooting tree
is very general and it is hard to locate the problem. The troubleshooting tree
have become more and more confusing over time because of additions to the
tree have been made sense the fuel system have become more advanced over the
years. The troubleshooting tree is now at the stage that its so confusing, most
people tend to avoid it.

To simplify the problem, Scania is now under development of an new addi-
tional troubleshooting tool called the mini-rail. The mini-rail consists of safety
valves and pressure sensors that are added to the vehicle during the troubleshoot
testing and the mini-rails purpose is to divide the high and low pressure side of
the system line to located the problem faster.

Even if the mini-rail cuts the troubleshooting process in half. There are
still room for improvement in the troubleshooting process. Like in the previous
cases, additional sensors would help to locate the problem faster. To make
the most optimal troubleshooting system, a collaboration between additional
pressure sensors and the mini-rail is necessary. There has to be a good way to
compound the gathered data from the collaboration.

One thing that simplifies this is that the concerning locations, that would
improve from additional sensors are relative easy to reach. Currently it takes
approximately five minutes to hook up the mini-rail. This means that a wireless
connection to the sensors is unnecessary.

4.4 Requirements

Summarizing the interviews there are two things that are especially important
to deal with and that is:

- Measure analog signals from temperature and pressure sensors.
- Communicate with CAN sensors.

In one of the cases there was also a wish that the sensors could be hooked up
via a wireless connection. With the project scope in mind and in particular
"investigate the possibilities of synchronizing the external and internal signals"
some questions were asked regarding the real time demands of these sensors:

- **Temperature**: The real time requirements of this kind of measurement
  is really low, synchronous around a second would be good.
- **Pressure**: The real time demands of a pressure sensor depends on where
  the sensor is deployed but it has a higher demand than the temperature
  sensor. Data should be synchronized in the millisecond range.

5 Ad hoc or Existing solution?

5.1 Key factors

The idea is to find a solution which satisfies the goals presented in section
1.3. The solutions investigated during this project will be analyzed and judged
according to these four key factors:
• **Modularity**: The solution should be able to cope with hardware changes, for example a new sensor being added to the system.

• **Performance**: The solutions performance should at least be of the same quality in terms of speed as the system being diagnosed IE the solution should not introduce another bottleneck in the diagnostic process.

• **Cost**: The solution should be cost effective, while no max cost of the solution has been mentioned this factor will have a heavy weight when determining what solution to go forward with.

• **Maintainability**: The solution should be maintainable in the future, its important to investigate and make sure that key elements such as hardware components and software API’s are available and working in the future as well.

One aspect of this project is that there already exist a market of products that easily could fit as a solution for computing external signals. The drawback with buying an already existing solution is that all the four key factors mentioned are not matched or are matched but not optimized for all the factors. More about the disadvantages in section 5.2.4. The benefit with an Ad hoc solution is that it can be modified to be optimized in terms of the four key factors.

### 5.2 Already existing solution

As mentioned above, a market of products that easily would handle the task at hand already exist. Listed below are some examples of products and a short consideration how this product holds to the four key factors.

#### 5.2.1 Ipetronik

Ipetronik [5] is a German company that is one of the leading suppliers of mobile measurement technology. They have several hardware product that have analog sensor inputs and measurement outputs to CAN. The concepts of Ipetronik is that every product is embedded in a hard-wearing box that also is completely galvanic isolated. The concept of the different tools is that they are easily combined together to match the required measurement tools. The products from Ipetronik differ in size and functionality. The most advanced hardware product is the MultiDAQ that has 42 analog inputs and a CAN-bus. The analog inputs are fitted for temperature sensors, voltage sensors and frequency sensors. The CAN-bus measurement data outputs according to ISO 11898-2. All of Ipetroniks hardware products are implemented with Ipetroniks own free software - Ipemotion 2015. The software have many functionalities, such as Digital filters, fast fourier transform, analysis editor and CAN traffic logger etc.

The obvious good reason to use Ipetronik products for our purpose is that it is more or less designed to do exactly what we are after. The smaller measurement boxes from Ipetronik has either temperature sensor inputs or measure voltage and current. Many of the cases described above are in need of being able to measure pressure, temperature etc all at once. This means that many different Ipetronik boxes are needed to be combined during a test. A major
disadvantage for Ipetronik is that it uses resistive temperature detectors (RTD) and not negative temperature coefficient (NTC) described in section 3.3.1. If Scania would implement this tool in the future, every Scania workshop would have to have two types of sensors. NTC- sensors as spare part for the truck and RTD sensors for the tool. A more convenient thing would be to have the same type of sensors for the measurement tool as the sensors already existing in the truck. The biggest drawback with using Ipetronik as an solution is the cost of the products. One of their simplest measurement tools called M-RTD2 which has four temperature inputs and one CAN-bus cost about 20 000 SEK. Looking back at the four key factors the cost of Ipetronik does not make it a good contender for a solution. Even if the product itself would serve as a good solution for this project the cost to have Ipetronik devices on every Scania workshop is unacceptable.

Figure 6: A hardware product from Ipetronik called M-RTD2.

5.2.2 Dewe

Dewe is a world wide company that also work with producing mobile measurement technology. Dewe and Ipetronik products works quite the same, Dewe are using the function of combining multiple measurement units to one computer. Just as Ipetronik products from Dewe would also fulfill the needs of the project but are too expansive. Dewe comes with its own software, the software and one of the smallest measurement tools from Dewe cost about 35 000 SEK inclusive the software for the program.

5.2.3 Cheaper already existing solutions

Both Ipetronik and Dewe make sophisticated solutions and are really good at what they do, but the sophisticated solution comes with a prize. There should exist cheaper options for this problem, but no serious company with products could be found. This means that even if there exist fitting solutions, on for example ebay by an single individual this would not be an option for Scania to invest in. Because of one of the key factors: maintainability, if a person sells products on ebay there is no guarantee the person will do it tomorrow. Of course the same thing can happen to a company as Ipetronik, but more money invested in a company makes it a safer bet. The only cheaper solutions that were products of a company were either CAN transmitters, logic analyzers or
products with analog inputs. A solution with both a CAN bus interface and analog inputs could not be found.

5.2.4 Disadvantages using existing solutions

To sum up the main disadvantage using existing solutions mentioned in 5.2.1 to 5.2.3 is that they are not optimized for the key factors mentioned in 5.1. Besides the cost factor, using an already existing solution is kind of dependent of the maintainability of the product. For example an implication will occur if the third party are launching an new API in the near future and the existing API that would satisfies this project will vanish or be changed. If the API is changed this would not be a huge problem on a small scale, but if this solution will exist on every Scania repair shop this makes a big problem. The solution have to be able to work as simple as possible as long as possible.

Another factor, that is not part of the four key factors, is that already existing solutions such as Ipetronik usually has an introduction course you have to take to understand how to work with the product. The best thing would be if the solution is as simple as possible and a simple handbook would do for a new person to understand how to use it.

5.3 Ad hoc solutions

From the conducted research we ended up with four alternatives that would satisfy the need on the end product solution. Three of the four solutions include an external CAN bus that do not interact with the internal CAN bus of the vehicle hence takes away the prospect of overloading the BUS and messing up the priority of the messages on the BUS. Therefore an external CAN bus is consider to be a good aspect for the solution.

5.3.1 VCI-VCI

Kvaser is the name of the company that makes the VCI's(Virtual CAN Interface) for Scania and they are either USB on one end and OBDII port on the other or wireless and OBDII. The idea here is to skip Scania's own API(Application Program Interface) SCOMM and connect and talk directly to the onboard CAN bus of the vehicle.

By doing this the diagnose protocol KWP would be circumvented and some overhead would be reduced and the round-trip time will most likely be shortened.

The other VCI would be connected to an external CAN bus or Scania sensor and data from both sources would be gathered and displayed, logged or plotted in a custom made software created during the project.

This solution will be considered as very modular, it has few if any hardware requirements besides the VCI which is an integral part of Scania's troubleshooting process already. Thus it should be easy to maintain as well. This would also most likely be the solution with the highest performance in terms of speed since it has the least overhead and the cost would be the cost of buying a second VCI.

Although this might be the fastest solution in terms of latency, it would require a lot of insight into the Scania CAN architecture and knowledge of Kvaser's CAN library. This solution would not support analog sensors, unless
they are connected to an ECU of some kind. This would be the most complex solution of the ones looked into during this project. An overview of the setup can be seen in Figure 7.

Figure 7: VCI-VCI setup. Internal CAN bus on the right and external to the left.

5.3.2 VCI-SCOMM

This idea is similar to the VCI-VCI setup but here the Scania API would be used to communicate with the internal CAN bus instead. The performance gain in terms of latency, mentioned in the previous solution would obviously be lost, but the communication with the internal CAN bus would be much simpler. This solution would share all aspects of the key factors with the previous solution except for the performance aspect. The main drawback here is that analog sensors would need their own ECU to be connected. Figure 8 illustrates the setup, at first glance it looks exactly the same as figure 7 the only difference is what API is used to connect with the internal CAN bus.

5.3.3 MCU/MCPU-SCOMM

This solution would use SCOMM to communicate with the internal CAN bus and a micro-controller or a microprocessor to communicate with external sensors. An external CAN bus controller needs to be added and connected with the MCU/MCPU to allow communication with external CAN sensors. The MCU or MCPU would act as a relay for all devices connected to it, both analog and CAN and simply forwards each sensor data to the computer running the diagnose program. By doing this the burden of modularity and maintainability is placed on the computer side, where it more easily could be updated. Performance wise it would be similar to the VCI-SCOMM setup, depending on how many external sensors are connected. The added cost of an MCU/MCPU, CAN
5.3.4 Benefits and problems using an Ad hoc solution

The most convenient thing when using an ad hoc solution would be to be able to use Scania’s sensors (some of them described in section 3.3). Then the Scania workshop that want to use this ad hoc solution would be able to use a redundant bus controller on a custom printed circuit board is estimated to be less than that of the cost of buying a second VCI.

This is the only solution which allows easy connection of analog sensors and is the solution which corresponds mostly with the authors field of study.
Table 2: A comparison between different existing solutions and Ad hoc solutions. Notation: * has either CAN input or analog input, not both. ** Modular if ECU is updated as well.

<table>
<thead>
<tr>
<th>Solution/factors</th>
<th>Modularity</th>
<th>Performance</th>
<th>Cost</th>
<th>Maintainability</th>
<th>CAN/Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ipetronik</td>
<td>-</td>
<td>+++</td>
<td>-</td>
<td>+</td>
<td>Both</td>
</tr>
<tr>
<td>Dewe</td>
<td>-</td>
<td>+++</td>
<td>-</td>
<td>+</td>
<td>Both</td>
</tr>
<tr>
<td>Cheap solution</td>
<td>+</td>
<td>++</td>
<td>+++</td>
<td>-</td>
<td>C or A*</td>
</tr>
<tr>
<td>VCI-VCI</td>
<td>+,**</td>
<td>+++</td>
<td>++</td>
<td>+++</td>
<td>CAN</td>
</tr>
<tr>
<td>VCI-SCOMM</td>
<td>+,**</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>CAN</td>
</tr>
<tr>
<td>MCU/MCPU-SCOMM</td>
<td>+++</td>
<td>++</td>
<td>+++</td>
<td>++</td>
<td>Both</td>
</tr>
</tbody>
</table>

sensor that are just laying around in the workshop to hook in to the ad hoc solution and get data from it. The problem with this is that the different sensors described in section 3.3 use different type of connections (some are analog sensors, some are connected to the CAN bus) and they are also using different values on pull up resistors and pull down resistors. When creating an ad hoc solution, these problems have to be considered and dealt with.

5.4 Which solution is the most suitable?

In Table 2 all the different solutions for this project are listed, both already existing solutions and ad hoc solutions. The awarding of points in the table comes from evaluated thoughts on the key factors and other aspects that will influence the final solution. The four key factors are essential but are also not all equally prioritized. For example, the performance of Ipetronik may be the best but may also have unnecessary good performance, as long as an solution equals SDP3 in performance other key factors are more important.

6 Implementation

6.1 Setup

The setup chosen for this implementation will be the ad hoc solution described in section 5.3.3 and the implementation will henceforth be divided into two parts, hardware and software where each author is responsible for one part. The chosen setup can be seen in in Figure 10. This section will include everything that is shared between the two implementations such as the communication interface between MCU and PC and how to interpret analog data.

The intended final solution is shown in Figure 10. Described in 2.2.3 ECUs connected to the CAN bus have an ID value which gives the ECU a priority. If two identical sensors hence identical ECUs are needed this will create a problem when they both have the same ID and prioritization. The MCU are connected to two separate external CAN Buses to easily sidestep this problem. The MCU also has ADC’s which provide analog sensors to be directly connected to the MCU without the sensors belonging ECU. Both using CAN buses and analog inputs makes this solution very generic and puts sensors laying around in workshops easily to use.
6.1.1 Communication protocol interface

A common interface is needed in order to communicate between the MCU and PC. A simple protocol was constructed early on so that each part of the project could be worked on simultaneously. The idea was to keep the messages sent to the microcontroller simple, the reason for this is to keep the message parsing in the MCU as simple as possible. So each message to the MCU will be one character and each character will represent a certain request.

The left hand side of the dash is the character sent and the right hand side is the MCU’s response. The letter \( n \) corresponds to the number of sensors and \( p \) is port number, \( d \) is data and \( i \) is CAN identifier.

- "a" - "1;n:p:.." : Analog Info Information about how many analog sensors are connected and their port number.
- "b" - "2;n:p:d:.." : Analog Data Port number and data of each analog sensor.
- "c" - "3;n:i:.." : CAN Info Information about how many CAN sensors are connected as well as their identifiers.
- "d" - "4;n:p:d[0-7]:.." : CAN Data Port number and data of each CAN sensor.

The first character on the right hand side indicates what type of message the transmission contains, the semicolon (;) is used to signify end of type section. The reason for using this is to be able to write a specific parser for each type of message on the PC side. Colon(:) is used to separate everything else. Figure 11 and 12 illustrates the process of a typical request for analog info and data being made on the PC.
6.1.2 Analog data interpretation

Depending on which MCU is chosen, the MCU contains 6-8 channels that are 10-bit analog to digital converters (Arduino has 6, Atmega644 in section 7.1.1 has 8). This means that it will map input voltage between 0 and 5 volts into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts/1024 units hence 0.0049 volts (49mV) per unit.

6.2 MCU or MCPU

An Ad hoc platform solution can take many shapes and forms. But to make it more easily accessible to SDP3 in the future the Ad hoc solution should be able to measure data and send it to a computer while optimizing the four key factors in section 1.4. To make this happen the Ad hoc solution need some kind of processor to control the incoming data and transmit measurements to a computer. There are many different products that would fit this project.

Microcontroller  The code that is written in the microcontrollers IDE is the only code that runs on the chip, which means no interpreter, no operating system and no firmware. The C code is translated into machine language and then runs on the microcontroller chip itself, this is as barebone as it gets.

Arduino  Arduino is a physical computing platform based on a simple microcontroller board. The Arduino board consist of preconfigured components such as a power jack, voltage regulators, ceramic resonator and a USB connection (components may vary depending on which Arduino board that is used), this saves a lot of time and work when starting a project. Like the standalone microcontroller the Arduino also has several of I/O connections.
The Arduino can be used to develop interactive objects, taking inputs from sensors and switches and controlling physical outputs. The main core of every Arduino is a microcontroller from the family Atmega. The neat thing when programming in Arduinos own programming environment is that there already exists predefined functions that makes it easier to program the microcontroller on the Arduino. The Arduino programming language is an implementation of a open source programming framework called Wiring, which can both interpret C and C++ scripts. Due to this the Arduino is at good start when testing a hypothesis if a microcontroller project can work. Writing a program with predefined functions usually makes the work easier and if the Arduino preforms as whished its guaranteed that a standalone microcontroller also will preform as whished. However the program script has to be translated into entirely C code for it to work on the standalone microcontroller.

Raspberry pi  The Raspberry pi may look like the Arduino and share some of the fundamental principles but is actually a SBC. The core of the board is a 32 bit microprocessor that supports ports for ethernet, audio, video, USB, SD cards and HDMI. The board also have some GPIO pins that are similar to Arduinos IO pins. Actually the Raspberry pi have more things in common with a computer than it has with a Arduino. The operating system of a raspberry pi is typically some linux distribution. When both the Microcontroller and Arduino takes the code to the hardware directly, the programming of the raspberry pi is actually writing programs within the operating system.

6.2.1 Difference between Microcontroller and Microprocessor

When describing the different Ad hoc platforms in section 5.3.3 both microcontrollers and microprocessors where in consideration. It is hard to compare the two things with each other and deciding which is the best platform. An easier

Figure 12: Analog Data request communication example.

PC

MCU

RequestAnalogData()

switch(2)

case dataA:

SendAnalogData()

switch(“b”)

case dataA:

SendAnalogData()
A question to answer is which platform is the best for the concerning project. Comparing the Raspberry pi with an Arduino (which is based on a microcontroller) both have a similar structure. Both have a CPU that executes the instructions, timers and memory. Comparing an Arduino Uno and a Raspberry pi Mod 2 in the same price range they have the following specifications, see Table 3.

Table 3: An comparison between Arduino Uno and Raspberry Pi mod 2 specifications.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Arduino Uno</th>
<th>Raspberry Pi mod 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price (SEK)</td>
<td>215</td>
<td>300</td>
</tr>
<tr>
<td>Processor</td>
<td>AVR, Atmega328p</td>
<td>ARM1176JZFS</td>
</tr>
<tr>
<td>Clockspeed</td>
<td>16 MHz</td>
<td>700 MHz</td>
</tr>
<tr>
<td>Register width</td>
<td>8-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>RAM</td>
<td>2kb</td>
<td>512 MB</td>
</tr>
<tr>
<td>I/O pins</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>I/O Current max</td>
<td>40mA</td>
<td>5-10 mA</td>
</tr>
<tr>
<td>Power consumption (no external hardware included)</td>
<td>175 mW</td>
<td>700 mW</td>
</tr>
<tr>
<td>Operating System</td>
<td>None</td>
<td>Linux and others</td>
</tr>
</tbody>
</table>

When first glancing on Table 3 the Raspberry pi looks better, but considering what the Ad hoc solution should do, the microcontroller and Arduino may be more proper for the project at hands. Projects that consist of controlling videos and cameras, complex math and graphic interface are more suited for the Raspberry pi. But projects that consist of controlling motors and sensors are better suited for a microcontroller.
6.2.2 Temporary prototype

Best suited for the problem at hand were the microcontroller over the raspberry pi. To be able to test the solution fast and easy, a temporary prototype using an Arduino Uno was made. The reason for this approach is the unique connection between Arduino and the MCU microcontroller. As mention in 6.2 section, the Arduinos main core is a microcontroller from the family Atmega. In section 9.1 after testing the Arduino proved itself of working as an solution for the project, a standalone microcontroller was chosen to become the final Ad hoc solution. The advantages of using a standalone microcontroller instead of the Arduino platform is that the Ad hoc solution can be slimmed down even further and loose unnecessary components that are included on the Arduino board that are not used. This makes the final Ad hoc solution smaller, cheaper and even more suited for the project than the Arduino solution.

6.2.3 MCU, platform of choice

When all the comparisons and considerations on which solution to use were completed. The standalone Microcontroller became the top pick. Looking at the key factors again in 5.1, the microcontroller looks like the most optimized solution for the job.

Modularity: The microcontroller always has the option of being connected through SPI-serial communication to be reprogrammed. Although reprogramming the microcontroller is a bit restricted if hardware changes are necessary.

Cost: It is hard to find a cheaper solution than using a microcontroller. All of the reasonable microcontroller suited for this project cost between 8 SEK up to 200 SEK.

Performance: If 8 MHz, that most the microcontrollers can produce, is not enough one can always use an external oscillator in the hardware setup to reach the necessary performance frequency. The dissolution provided by the internal ADC on a microcontroller should be enough to get a accurate measurement from sensors.

Maintainability: Since the microcontroller is programmed by our self the maintainability of it will be as long as one wish. As for the Modularity, the access to reprogram the microcontroller if needed is also plausible.
7 Hardware

As mentioned in section 6.2, the Arduino proved to work as a good Ad hoc solution and hence a this section will describe the work done on the hardware when finding the next step solution with a microcontroller.

7.1 Hardware Components

7.1.1 MCU

On deciding which microcontroller to use two things were important factors. The microcontroller should have many ADC pins and should also have one or more SPI interface pins to be able to connect to the CAN bus components described in 7.1.2. One other thing that also makes the testing and programming of the chip easier was if the chip was in a DIP case. The biggest microcontroller from Atmega that had DIP casing and the most ADC pins were the Atmega 664p.

7.1.2 CAN bus components

To create a standalone CAN bus the only thing needed is to have two wires - CAN High and CAN low and a 220 ohm resistor between them. To be able to connect and communicate over the CAN bus the microcontroller have to be able to transmit and receive data over the CAN bus. Two common components to use with microcontrollers to achieve this is MCP2551 and MCP2515.

**MCP2551** is a high speed CAN transceiver made by the company Microchip. The device serves as the interface between a CAN protocol controller and the physical bus. The MCP2551 device provides differential transmit and receive capability for the CAN protocol controller, and is fully compatible with the ISO-11898 standard, including 24V requirements. It will operate at speeds up to 1 Mb/s [9].

**MCP2525** is a stand-alone CAN controller with SPI interface that can be connected to the microcontroller. The MCP2515 includes a CAN module that handles all functions for receiving and transmitting messages on the CAN bus. So by connecting this chip between the MCP2551 and the microcontroller, we develop a gateway for the microcontroller to send and receive messages over the CAN bus. [8]

7.1.3 DC/DC converter

The idea of this Ad hoc solution is that it should be easy to hook up to the truck and do some testing. When it comes to power supply, the most comfortable way to give a current would be by connecting it to the powerjack output in the trucks cockpit. The problem is that the power supply from the truck has 24 volts. The microcontroller and the rest on the circuit only need about 5 volts. So a DC/DC converter is needed in the circuit. When looking for a suitable DC/DC converter one should take the opportunity to look for a very convenient feature that takes care of galvanic isolation. Galvanic isolation is a principle of isolating sections of the electrical system to prevent current flow. This means no direct conduction
path is permitted. This is an effective method of breaking ground loops by preventing unwanted high currents from flowing into a section that has much lower ground potentials.

**TEL 3-2011** Is a DC/DC converter with galvanic isolation from the company Traco Power. The converter has a voltage input span between 10 to 30 volt and gives a steady voltage output of 5 volts with the effect magnitude of 3 watt. With this component the solution becomes more versatile and the circuit is running a smaller risk of becoming burnt from a too high input current [18].

### 7.1.4 Other necessary components

**D connector 25 pins** Due to that the main idea of this solution is to be able to connect sensors to inputs means that is going to be a lot of wires going out from the circuit. Because the circuit probably going to be protected in a casing all the outgoing pins from the circuit is connected to a 25 pins D connector. See section 7.2.3 and Figures 19 and 20 for a better explanation.

**2 pin header for resistors** As mentioned in section 5.3.4 sensors that are connected to the analog inputs have different values on their pull up and pull down resistors depending on which type of temperature sensor etc they are. To make the circuit more versatile and to not lock one input to a specific type of temperature sensor 2 pin headers will replace the pull down/up resistor. With this way the resistor magnitude is changeable and makes the hole circuit more versatile to more types of pressure and temperature sensors, see Figure 14

![Figure 14: 2 pin headers to the left makes it possible to change resistors instead of having resistors directly connected to the board as on the right.](image)

**6 pin header connected to the MCU** In case the microcontroller have to be reprogram a 6 pin header can be used in the circuit so one can easily connect the microcontroller to a computer via SPI communication for instance by using an USBasp.

### 7.2 PCB layout design

A natural step when having a functional circuit is to design a PCB layout for the circuit. A PCB makes the circuit smaller and more compact and also removes
all the wires that are in danger of being disconnected and are replaced with traces on the PCB. Another benefit of having a PCB of the circuit is that it is easier to attend the circuits problems with EMC and EMI.

7.2.1 EMC and EMI

The EMC of a circuit is primarily determined by how the components are located to each other and by how electrical connections are made between components. Every current flowing in a trace generates a current of the same magnitude flowing in the return trace. This trace line loop creates an antenna that can radiate electromagnetic energy whose magnitude is determined by the geometrical area of the loop, amplitude of the current and the repetition frequency. Trace line that contribute to undesirable radiation can be categorized in different classes, each with varying degrees of radiation [4].

Supply lines Since the power consumption of the circuit is not constant all over the board but depends on its instant state area, then all the frequency components generated in the individual parts of the system are represented in this category. Supply lines have relatively high impedance, fast current changes cannot be suppressed while active. To prevent this blocking capacitors are added to the circuit.

Current peaks that occur for example when transistor is switched are one of the most significant causes of EMI. Every time an output is switched a corresponding pulse of current flows along the supply trace line. The problem is worsen when the output are switched at a high repetition rate. To reduce this problem additional decoupling capacitors at about the magnitude of 100 nF are necessary as close as possible to the switching component. This technique is effective and ensures that with the expected load charges, no undesired supply voltage changes can occur.

Signal and control lines Loops between components and to ground/supply voltage can be categorized as an own class. These loops often transmit signals at high frequencies but when designed small usually does not radiate much. The most cost effective and technically effective method consist of simply keeping the trace lines as short at possible. To minimize the antenna effect when small circuits are not possible is to run the return trace line parallel to the signal trace line. However this is automatically ensured with a multi layer PCB that have ground level under the signal line.

Oscillator circuit External frequency-determining components creates its own category. Usually the highest frequency is found in these loops and special care have to be taken with the design of the circuit to avoid unnecessary interference voltages. Also the routing of the connecting lines have to be in consideration to minimize the effective areas of the antennas. Due to the oscillator circuit usually holds the highest frequency, the trace line of an external oscillator for example has top priority to have the smallest trace line. When using a multi layer PCB design there should not even have to exist any trace line from the external oscillator instead an direct connection to the capacitors and thenceforth with a via to the desired location (often to ground plane).
7.2.2 Additional rules to the PCB design

Beside the measures described in 7.2.1 to optimize the EMC and reduce the EMI, some additional rules were followed when designing the PCB.

Using guidelines for circuits with frequencies under 50 MHz  In a circuit that does not involve frequencies over 50 MHz traditional decoupling methods are effective. These guidelines involve using decoupling capacitors (highest magnitude 10 μF) placed as close as possible to the power and ground pins.

Multilayer board  To shrink down the size of the board and to more easily shrink down the trace lines a multilayer PCB is used. The amount of layers on the pcb can vary from 2 up to 11. This pcb is not that complicated and only has four layers. This provides two layers for components and traces, one ground layer/polygon for both analog ground and digital ground and one power supply layer/polygon. The order of the layers from the top are 1: Components, traces, analog power supply trace and ground polygons. 2: Ground layer. 3: Power supply. 4: Components, traces and ground polygons, see Figure 15.

![Figure 15: The layer stackup, between the layers the distance is typed out.](image)

Rules for the vias  The design only uses vias that are drilled through the hole board. A different type of vias also exist where the drill hole is only as deep as the concerned layers. This makes the board more optimized but also more expensive due to more advance developing methods being used. Due to this pcb does not really have a size restriction, the advanced vias are unnecessary for this project, see Figure 16 Another rule for the vias is that the design should avoid concentration of vias. A concentration of vias tend to make power and ground planes have higher impedance and a lot of noise generates from areas with high concentration of vias.

![Figure 16: Different types of vias is shown in this picture that both are connected to layer 2 and 4. This pcb design only use the left type.](image)
The 3-W Rule  Cross talk between two traces can cause EMC problems. This usually occurs when two traces are too near each other and the current through one of the traces influence the other with EMI. To prevent this, long traces should be separated with at least three times the width of the trace width. This is not always possible, but one should at least have it in mind when designing a PCB and try to follow the 3-W rule as much as possible.

Keep grounds separated  Separate grounding for analog and digital portions of circuitry is one of the simplest and most effective methods for noise suppression. Analog ground should only be connected to analog circuits and digital ground to digital circuits. In this design it does not exist a lot of analog circuitry and therefore only a small portion of the ground plane is analog ground polygon. The polygon is then connected with a small bridge to the ground plane.

Multi-point grounding  Due to that the circuit contains frequencies over 1 MHz the design will be using multi-point grounding. This means that every pin that are supposed to be connected to ground has its own via to the ground plane. Another approach for circuits with frequencies under 1 MHz is to use single point grounding. In single point grounding all the pins that are supposed to be connected to ground should be traced to one central via that are connected to the ground plane.

7.2.3 The schematic of the circuit for the PCB design

After following the rules and guidelines in section 7.2.1 and 7.2.2 a schematic could be made. The schematic became quite big and therefore is divided into sections. In figure 17 the schematic shows one of the two CAN buses that are created from the CAN bus transmitter MCP2515 and CAN bus receiver MCP2551. Like in figure 17, figure 18 shows the other CAN bus connected to the microcontroller Atmega644p. Notice also that the SPI pins from the microcontroller is connected to a 6 pinned header. This makes it possible to reprogram the microcontroller with an USBasp if necessary, this also mentioned in section 7.1.4.

Figure 19 show the power jack for the PCB that are connected to the DC/DC converter described in 7.1.3. Figure 19 also shows how to connect all the ADC pins from the microcontroller to the D connector. Note that every ADC pin connected to the D connector are also connected to a female 2 pin header. This is so one can easily change the pull up or pull down resistor depending on which kind of sensor one is using, see section 7.1.4.

Besides the PCB schematic, figure 20 shows the schematic how to connect the other end of the D connector cable to the sensor output/inputs. For instance, this can be located on top of the casing of the PCB and near where sensors are connected. See section 9.1 to see how the first prototype box was built. In that case everything in Figure 20 would be located under the lid of the box.
Figure 17: Schematic of MCP2510 and MCP2551 which forms one of the CAN buses
Figure 18: Schematic of the Atmega644p with one of the CAN bus communication
Figure 19: The powerjack for the board are connected to the TEL-3 DC/DC converter. Also in the schematic the output D-connector from the PCB are shown.

Figure 20: This is not part of the PCB rather how to connect the other end of the D connector to the sensor inputs/outputs.
7.2.4 The designed PCB

Using the schematic from section 7.2.3 and everything else from 7.2 in consideration a design of the PCB was made. As described before, the PCB is a four layer board and has the width about 15.7 cm and height 9.6 cm. Hence it would fit in the box used in the first testing, see section 9.1.

**Layer 1** See Figure 21. Almost all of the components are located on the top layer. Except the components and all the tracing there is a large ground polygon surrounding everything. This is to strengthen the signal integrity by shielding the components and traces from cross talk. Note that every component mark with a P for example PP1, PP2 etc are 2 pin female headers. See section 19 for the reason for this instead of resistors.

![Figure 21: The top layer of the designed PCB, almost all of the components are located here. The red polygon is ground.](image)

**Layer 2** See Figure 22, this is the ground layer. A small part of the ground plane is cut out and filled with analog ground polygon and only a small bridge connects the two ground polygons.

**Layer 3** See Figure 23, this layer is the power supply plane. The DC/DC converter has a via directly from it that supplies 5 V to the hole plane.

**Layer 4** See Figure 24. This is the bottom layer and contains one capacitor that did not fit on the top layer. This plane also have some tracing to optimize
the tracing route. This layer also contains a large ground polygon that covers everywhere there is no tracing, via or components.

### 7.2.5 Saving clause on PCB design

To design a PCB and optimize the EMC can be very complex. Decisions I have taking when designing the PCB are only of an theoretical stand of view and only gets the design some steps in the right direction. For instance, decoupling capacitors on the supply loops is only a very limited reduction of EMI. To achieve a significant improvement it its necessary to analyze the complete circuit and its parasitic components. Another thing that should be fixed for the next pcb prototype is that one should try to use surface mount components instead of through hole components. For instance, the microcontroller Atmega644p and the CAN communication components mcp2551 and mcp2515 all exist as surface mount components and are smaller than the through hole components used in this design. During the design phase not all footprints for the components were accessible which lead to the usage of through hole components. However, this project does not really have any limitations on the size of the PCB. So it does not make that big of a difference if the components are through hole or surface mounted. More about improvements of the PCB in Section 12.
Figure 23: The third layer of the PCB is a power supply plane.

Figure 24: The bottom layer which is the fourth and last layer of the PCB. This layer contains one capacitor, tracing and a large ground polygon everywhere else.
8 Software

This section will describe the work done on the computer side or PC as portrayed in section 6.1 figure 10, as the title suggest all of this work is software.

The main purpose of this software is to communicate with the hardware (MCU), internal CAN bus and interpret and log the data gathered from these sources. How to interpret this data differs a lot depending on the type of sensor and whether its analog or digital (CAN). Because of the need to keep things modular IE a new sensor is added or changed, how to interpret data or sensor configurations cannot be hard coded into the program. This has resulted in several programs being created, a main program that communicates with the hardware and logs data and two programs that each lets the user create sensor configurations for analog and digital sensors respectively. One of the goals of the project has also been to investigate delays that occur from request to actually obtaining data in the internal CAN bus and how this might affect synchronization between external and internal sensors. To that end an analytical and experimental approach have been taken where two programs was created. All programs created will be described in the following sections. Figure 25 displays an overview of how the programs work together.

![Diagram of program interactions](image)

Figure 25: Overview of how the different programs work together.

While not one of the project goals, it has been requested that these tools to handle external sensors where also implemented into Scania’s diagnose program SPD3, this process will also be described briefly.
8.1 External Sensor Module

The external sensor module is the main program, it is used to connect to external and internal sensors and log their data. This section will describe the program and how the code was constructed. The program is written in Microsoft Visual Studio Ultimate 2013 using the presentation system WPF [13]. The code consists of four files: ExternalSensor.cs, SensorData.cs, MainWindow.xaml.cs and MainWindow.xaml. In MainWindow.xaml the visual contents of the program is defined, IE the GUI and the cs file is its associated code-behind file. See [10] for more information about WPF. Figure 26 depicts dependencies between all classes created in this program.

Figure 26: Dependency graph for the for the ExternalSensorModule namespace, generated in Visual Studio

8.1.1 ExternalSensor.cs

ExternalSensor.cs contains the main logic of the program, all methods have been grouped into regions according to their functionality. Each region will be explained in more detail in the upcoming sections.

Members Global variables, get and set functions and the constructor of the class are defined here. A lot of lists are defined here and most of em are initialized
as empty in the constructor but one of them, called ImpEcuList is imported from a file called "CanSensor.xml" and it contains all data created with the XML builder, see section 8.2 for more information.

**State** Perhaps poorly named but the State region contains two methods: `CheckState` and `MessageParse`. `MessageParse` is the first method to be called when a serial port message has been received. It begins by dequeuing the latest message from a FIFO buffer and then splits the message into two strings, the message is split where the delimiter semicolon is found. The first string will contain the type of the message which will determine how the second string should be interpreted, see section 6.1.1 for communication protocol. The second string contains the port numbers and the actual data. The first and second string is fed into the `CheckState` method where a switch statement will determine which method should be used to parse the message. The next two paragraphs will explain in detail how each message parse method works.

**CAN message parse** This region contains two methods for parsing the serial messages: `ParseCanInfoMsg` and `ParseCanDataMsg`.

The purpose of `ParseCanInfoMsg` is to parse the incoming message and create a list of ECU’s according to the information in the message. The message is split with colon as a delimiter into an array of strings. The first string holds the number of sensors connected to the microcontroller. The second and any string after, if they exist holds the unique identifier of a CAN message. So for all sensors connected a CanMsg object is created and a look-up into the ImpEcuList is done and when a match is found configuration data is copied from the list into the newly created CanMsg object. Each object will also be given a port number, this port number is used instead of the long 29-bit identifier when sending actual CAN data. Because CanMsg is a mid level class object another look-up needs to be done so it can be inserted into the right ECU’s Can message list. Once all CanMsg’s are inserted a ECU list is created.

The `ParseCanDataMsg` method is simpler, once again the first string holds the number of sensors and the following strings are port and then data, index 0 to 7. The serial message port number is matched with the previously mentioned ECU list’s CanMsg list’s port number and data is copied into its variable RawData.

See section 8.1.2 for more information on the ECU and CanMsg classes and figure 37 for a graphical view of the class hierarchy.

**Analog message parse** Just like the CAN message parse, the analog parse region consists of of two methods called `ParseAnalogInfoMsg` and `ParseAnalogDataMsg`. InfoMsg also works in a simular fashion, a number of AnalogSensor class object is created depending on the amount of analog sensors connected to the microcontroller. The port number is set according to the serial message and the AnalogSensor TypeList is imported from an external file. The type of sensor needs to be set manually for an analog sensor. See section 3.3 for types of analog sensors at Scania.

`ParseAnalogDataMsg` method parses the serial message and matches the port number with the list created by `ParseAnalogInfoMsg` and updates its Rawdata variable accordingly. It also runs the `UpdateSensor` method wich will update
the objects Data value according to what type has been chosen. See section 8.1.2 for more information about the AnalogSensor class.

**Serialport** All methods regarding the creation and handling of the serial port is grouped here: **SerialPortThread**, **DataReceivedHandler**, **StartSerialPort** and **GetPortName**.

*SerialPortThread* is once again a poorly named method, because it does not actually start a new thread. But it did at one point and since the creation of this program has been an iterative progress some methods have been copied from earlier programs. The method, which is based on [11] sets all parameters for the serial port such as: Serial Port name, baud rate, stop bit, parity bit, data bits, handshake, read and write timeouts. It opens the serial port and subscribes to the SerialDataReceivedEventHandler DataReceived handler. An event is raised when data has been received on the serial port object. When the event is raised the handler *DataReceived* will be called and it will queue the message into a FIFO buffer.

The method *GetPortName* is based on [16] and its using Windows Management Instrumentation to search for all connected ports and add their device id and caption. This creates a list of com ports that looks similar to that which is seen in the Windows device manager. As an input to the method a string is provided, this string could be "Profilic" or "Arduino" or any other USB serial adapter. The output of the method will be the name of the serial port which contains the caption provided by the input string. This method exists so the user does not have to select the com port manually at start-up.

*StartSerialPort* will call upon *GetPortName* and use its output as input for *StartSerialPortThread*. It will toogle a boolean to indicate that the Serial port is up and running.

**Serialport requests** Each serial port request looks the same except for the letter that is written into the serial port. Each letter represents a certain request to the microcontroller and they are as follows:

1. a : Analog Info
2. b : Analog Data
3. c : Can Info
4. d : Can Data
5. e : All Info

All requests will try to open the serial port if it is not open already and they are embedded into try and catch statements with a message box appearing if the serial port could not be opened.

**XML** There are two methods here: **SerializeToXML** and **DeserializeFromXML**. The first one is not used in the final version of the program but **DeserializeFromXML** is used as the name suggests to import data from an xml file by de-serializing it into its original form. This method imports CAN configuration data saved by the XML builder (section 8.2). It takes a file name as input and outputs a list of ECU class objects which is later used to compare with incoming CAN identifiers and configure its setup accordingly.
Scomm  The Scomm region also has two methods: ScommConnect and ScommDispose. ScommConnect is used to create a an Scomm product which in turn is used to connect to the internal CAN network and connect with all its ECU’s. ScommDispose is simply used to dispose of the previously created product.

8.1.2 SensorData.cs

SensorData.cs contains helper classes and functions regarding how to interpret sensor raw data. The classes are: ECU, CanMsg, Io, AnalogSensor and SensorType. The first three are all used to interpret CAN messages and the second two are used to configure and interpret analog data.

ECU  The ECU class as seen in figure 27 have four properties, name, time to connect, average response time and a list of CanMsgList objects. The reason for having this class are for time synchronization purposes, the properties response time and time to connect are set by running the program described in section 8.3.2. Each property have a get and set method.

![Figure 27: Dependency graph for the ECU class, generated in Visual Studio](image)

CanMsg  The CanMsg class have six properties, see figure 28. This level in the class hierarchy represents the same level that messages are sniffed in the external CAN bus connected to the microcontroller. Each message contain a 29-bit identifier and 8 bytes of raw data. Using the identifier a look-up can be done to determine the name of the message as well as which io data the raw data actually contains. As described in earlier sections, port is there to reduce the message length between the MCU and computer.

Io  The Io class is the biggest of the CAN classes, this is where the CAN message’s raw data is interpreted into actual values. Figure 29 depicts all prop-
properties and the overloaded method `UpdateSensor` which exists in the Io class. The method takes either int or an array of bytes as input and updates the Data value of the class depending on how the other properties have been set. The properties: Name, NamePresentation and Unit are all used for logging purposes, as an identifier for each column. The other properties are all used to interpret raw data. See section 8.2 XML builder for more information on what they each represent.

Figure 28: Dependency graph for the CanMsg class, generated in Visual Studio. The arrows toward RawData and IoList indicate that these properties are set in the constructor of the class.

Figure 29: Dependency graph for the Io class, generated in Visual Studio.
AnalogSensor  The AnalogSensor class is the entry point of analog sensors, as seen in figure 26 it’s in the top level of the analog class hierarchy. In figure 30 the properties of the class can be seen. Port and Rawdata are configured automatically when a message from the microcontroller is received but since neither the microcontroller or computer has any idea of what type of sensor is connected, the Type property has to be set manually by the user by choosing the correct sensor from TypeList. The TypeList is an imported list where the user has specified the name and type of the sensor as well as a look-up table for its characteristics. The Delay property is there for time synchronization purposes. The Type and TypeList properties are both objects of the SensorType class.

![Dependency graph for the AnalogSensor class, generated in Visual Studio.](image)

SensorType  As seen in figure 31 the SensorType class has five properties and two methods. The Type property unlike the Type property of the AnalogSensor class is a simple string, Name and Type are there for logging information purposes. The mysteriously named X and Y properties are the x and y axis of a sensor look-up table and they are used by the method InterpolateLUT to get an interpolated value. The method uses linear interpolation [2]. The UpdateSensor method uses almost all of the class members, it begins by converting the RawData value into a voltage and then into a resistance, this resistance will vary because of the sensor type, a temperature and pressure sensor have different pull-up and down resistors. After resistance has been calculated it will call upon InterpolateLUT and then update the Data property.
Figure 31: Dependency graph for the SensorType class, generated in Visual Studio.
8.1.3 MainWindow.xaml.cs

As mentioned in the beginning of this chapter the GUI has been written using WPF, a combination of XAML and any .NET language. In this case the .NET language is C#. The MainWindow.xaml.cs file is where all events of the GUI is handled such as button presses, checkboxes and expanders. The event handlers and methods are grouped into regions according to their functionality.

The class above all else is called MainWindow, it is a partial class because it is combined with the XAML file at runtime to produce the entire window. In the constructor of the class a method call to InitializeComponent is done, in short this is where the linking between the XAML and code-behind is being done. This is also where a class object of the ExternalSensor class is created and where the "DataContext" of the XAML file is set, see section 8.1.4 for more information on DataContext.

Figure 32: A simple WPF Expander, ExpandDirection Down

Expanders An Expander is a control with a header which can be expanded to display additional content or collapsed into its original state. The Expander has an "Expanded" event which occurs when the "IsExpanded" property changes from false to true. By subscribing to this event and specifying an event handler the behavior of the program when this button is pressed can be defined.

In this program there are three expander buttons: CAN_Expanded, Analog_Expanded and Scomm_Expanded. These three handlers are very short, the CAN expander simply calls RequestCanData, the Scomm expander calls ScommConnect. The Analog expander is actually blank, the corresponding call to RequestAnalogData is done at a later stage. There is actually several layers of expanders where the previously mentioned handlers all subscribe to the "Expanded" event caused by clicking on the top layer expander. When clicked the expander, as the name implies also expands into further controls and in this case into more expanders. In the case of the CAN expander the amount of sub level expanders are linked to a list called "ExtEcuList" which is created after a CAN Info Message has been received. The name of each expander is bound to the "EcuName" class property. The next sublevel is linked to "CanMsgList" and its property "CanName" is displayed. At the bottom level everything is bound to "IoList" and here there are no more expanders, just checkboxes where the "Name" property of each IO is displayed. The Scomm and Analog expander works in a similar fashion but they have less layers. As been described before the analog is a special case where the user manually has to select what type of sensor it is.

Checkboxes At the lowest level we have the checkboxes, here the user selects which sensors to add the logging list and in the analog sensor case this is also where its type is selected. For each type of sensor IE CAN, Analog and Scomm there are two types of checkbox handlers: checked and unchecked. They are
surprisingly named Analog_Checkbox_Checked, Analog_CheckBox_Unchecked and vice verse for CAN and Scomm.

The purpose of the checkboxes is similar for all three sensor types, to add and remove sensors from the sensor logging list but the code to implement this varies a little bit in each case. The simplest case is the CAN_CheckBox_Checked where the "Tag" property of the checkbox control is cast as an IO class object. This object is then added to the "CanLoglist". The Tag property is a way to link an object with a certain control. What is stored is a reference to the object. The code for unchecked checkbox handler looks the same but it does a remove instead of add from the CanLoglist.

The analog checkbox handler needs to link two references together, SensorType and AnalogSensor. The AnalogSensor object is referenced with the Tag property and the SensorType object is passed along as a "CommandParameter". After a match is found between the selected port number and the ones found iterating through the "AiSensorList" the Type and Name are copied into the AiSensorList’s AnalogSensor object and then added into the "AiLogList".

The Scomm checkbox handler also requires two objects, the io name as a string and and and Scomm ICCommonEcu object as CommandParameter. The string is converted into an Scomm ICUserData object and then stored into a list called IoLogList and the ICCommonEcu object is stored into a list called "EcuLogList". An ICCommonEcu object is needed to read an io and an ICUserData object is the io to be read.

Buttons There are two buttons in this program called start and stop. The start button when pressed initiates a "SaveFileDialog" where the user can enter the name of the log file. After that ExtWriteHeader is called and the program starts writing header information about the selected sensors into the newly created file. The program has a slider for selecting the frequency of the sensor data requests which operate between 1 and 10 Hz. The value of the slider is read and a new thread is created which runs with a timer based on the slider value. A single method runs in this thread called RequestDataCallBack, see next paragraph for more information.

The Stop_Button_Click event handler ends all processes started by its nemesis the start button. It terminates and disposes of the previously created callback thread and closes the file writer and starts a garbage collector. In short, it stops the logging process.
Callback  The Callback region only has one method, the \textit{RequestDataCallback} mentioned in the previous paragraph. The whole method is encapsulated in a do while loop where the condition is: run while the boolean ”TerminateCallbackThread” is false. This is where program requests sensor data from the microcontroller or the internal CAN bus for Scmm sensors. Time stamps are created and added to a list and a call to \textit{ExtLogToFile} is made in this method.

Logging  Two methods belong here, \textit{ExtWriteHeader} and \textit{ExtLogToFile}. \textit{ExtWriteHeader} writes header information such as: sensor name, type and unit with a TAB character in between. In the \textit{ExtLogToFile} method the actual sensor data is written, along with current time stamp and time elapsed since last time stamp. All entries are separated with a TAB here as well. After a successful logging the data in the log file can easily be copied into Excel or Matlab for plotting or analysis.

8.1.4 MainWindow.xaml

This is where the visual presentation is defined, the borders, panels, expanders, checkboxes etc and their shapes, height, placement etc. XAML stands for “Extensible Application Markup Language” and its an XML-based markup language developed by Microsoft [12]. The visual content have to some extent already been described in the previous section but there are some important aspects of XAML that deserves a little more explaining without getting into too much details.

DataContext  The DataContext is a perfect example of such. It is the link between the user interface and code-behind. The DataContext is like the root or home folder, by setting it, access is given to to all sub-directories. The DataContext can be set at an application level, at the window level or for a control. In this program the DataContext is set for the window and bound to the class object of ExternalSensor created in the constructor of the MainWindow class.

Binding  By setting the DataContext we can then easily bind to objects or properties within that class, such as a name or a list of items.

ItemsControl  The ItemsControl coupled with ItemTemplate is heavily used within this project to display a list of items. A good example is the code for the external CAN expander in figure 36. The first four rows creates a number of expanders based on how many objects that exists within ExtEcuList
and the header of all these will display their property EcuName respectively. DataContext is always inherited from their parent control, so by binding the ItemsSource of the ItemsControl to ExtEcuList the DataContext for all subsequent controls is now ExtSensor.ExtEcuList (ExtSensor is the ExternalSensor class object created in the constructor of the MainWindow class). In the fifth row another ItemsControl appear and its ItemsSource is bound to CamMsgList. This changes the DataContext to ExtSensor.ExtEcuList.CanMsgList for all subsequent controls. The following expanders all bind their headers to the property CanName. Lastly in the 9th row the DataContext is changed to ExtSensor.ExtEcuList.CanMsgList.IoList and number of selectable checkboxes appear depending on how many Io's exists within IoList. The Tag property was mentioned in the previous section and in figure 36 at the second to last line we can see how the Tag property is set for the checkbox. It is set to "Binding" which means that its bound to the object in the DataContext.

8.2 XML builder

In order to make any of the previously mentioned programs modular in terms of external sensor data gathering, sensor-data cannot be hard-coded into the program. Instead there needs to be a way for the method engineer to add, change, remove and load sensor configurations as new sensors are added or a sensor configuration changes. one way of doing so is to export sensor-configurations into files and load them again during run-time. XML provides an easy way to do this while keeping the format readable and changeable by humans as well as machines. Figure 37 displays the hierarchy used in all external programs created. This hierarchy was chosen because it best represents the CAN bus of a truck or bus. The various buses are populated with a number of ECU’s and each ECU sends one or more messages and each message contains data from one or more Io’s(sensors).
8.2.1 CAN library

The XML builder will be used to make lists of Io’s and CAN messages and link these together. A list of ECU’s can be imported from either a previously saved file or imported from the measurement program, see section 8.3.2. ECU’s will be linked with CAN messages according to name and CAN messages will be linked with Io’s according to CAN identifier. The idea is that a method engineer should be able to add the calibration data found in the datasheet specifying all messages for that particular ECU.

8.2.2 Lookup-Table Maker

the LUTMaker is a program designed to assist in creating analog sensor configurations. The idea is that the user should be able to look into a data sheet and find a chart with sensor characteristics and then add these values to a table and save it with the sensor name, type and the unit of the table data. There is also an option to insert a value pull-up,down resistors which is important to include if the sensor characteristics describe the output value as a function of resistance. In which case the sensor most likely needs to be coupled with a voltage divider circuit. For instance all NTC temperature sensors needs to be connected this way. In figure 39 on the right side the characteristics of a pressure sensor is shown. The output is a voltage signal as a function of pressure with the unit bar. When the output is a voltage signal like this one, voltage should be selected as the input type.
Figure 38: How to insert data into the XML builder from CAN datasheet.

Figure 39: How to insert data into the lookup-table maker from a sensor chart.
8.3 Synchronization

In section 1.4, one of the project goals is to investigate possible delays that may occur within internal and external signaling. To clarify, the word internal will always be a reference to the system residing within the truck i.e the CAN bus and external is a reference to the Ad hoc solution created by the authors of this thesis. The word delay can mean many things but here its meaning is the deviation from a fixed round-trip time. The primary focus of this chapter will be to investigate and list possible sources of delay along the path of a diagnose data request message. A diagnose request message starts in the diagnose program(SDP3) after its construction its transmitted through a cable or through wifi into the VCI. The VCI converts the message into a CAN frame and from there it is then transmitted onto the green CAN bus. Depending on the target ECU, the frame will either remain on the green bus or it will be placed in a FIFO-buffer and then gated to the red or yellow bus by the COO. The signal delay from computer to CAN bus is assumed to be static and its contribution to the overall delay can be measured. The CAN bus however is priority based and non-preemptive(section 2.2.3) making it non-deterministic. The diagnose message is of the lowest priority, which means that its round trip time will always depend on the current bus load. By analyzing the best and worst response times of each ECU, upper and lower bounds of the signal delay could be obtained. This could be done either experimentally or analytically. The experimental approach would be to measure the round-trip time of a ECU data request and repeat this process until a decent average is obtained. Analytically this could be done using modular performance analysis(MPA) with real-time calculus(RTC) and in the next section an attempt to model the CAN bus will be made using this method.

8.3.1 Analytic approach

To get the min-maximum response times of a diagnose message we need to find the outgoing arrival curve for the cyclic message with the lowest priority. In order to do that we need to find its incoming service curve. The service curve offered to a message will depend on all messages that come before it, IE any message with a higher priority. So a list of all ECU priorities and cycle times needs to be created. Such a list was found in the appendix of internal document [14] and it contains message name, CAN identifiers, cycle time as well as their respective contribution to the bus load for all ECU’s connected to the red bus. By using MPA and the Matlab RTC toolbox [19] we can now model the red bus. First we sort the list according to priority and then we loop through list and build our arrival curves with the cycle time as input and their resource demand as input. A CAN frame varies in size depending on the amounts of stuff-bits needed (130-155 bits) but in this experiment the packet size is fixed at 155 bits per frame. Figure 40 depicts how the model was created.

After the initial service curve $\beta$ is created it is fed into the greedy processing component along with the arrival curve $\alpha$ of the CAN message with the highest priority. The output arrival curve $\alpha'$ is then saved into an array while the output service curve $\beta'$ is fed into the next GPC component along with the arrival curve of the message with the second highest priority, and so it goes on until all 92 messages have been processed. It should be noted that in order to get this
Figure 40: Overview of the MPA RTC model

simulation to work the cycle time of all messages exceeding 500 milliseconds had their cycles time reduced to 500 milliseconds. Nearly half of the messages where affected by this and this will create a very pessimistic result where the bus load is increased significantly.
8.3.2 Experimental approach

The idea behind the experimental approach is like the analytical one, to get upper and lower bounds on response times for each ECU data query. A simple program will be constructed to do these measurements and a flowchart of the program can be seen in figure 41.

![Flowchart of measurement program.](image)

While the program is working through the list of ECU’s another list will be made, a list of ECU class objects which will have a name and the minimum and maximum response times. This list will be exported to xml and subsequently imported by the main program. It will then be used to match incoming CAN messages from the external CAN bus with its corresponding ECU. To get a decent average each IO reading will be done 500 times.
9 Testing

9.1 Testing and verification

In this subsection we describe how our first prototype worked with an actual Scania truck (model Euro 6). During the test we tested both a NOx sensor which is connected through our external CAN bus and a pressure sensor with an analog input connection to our prototype b1. Three different runs were done were we compared the trucks internal sensors with the external sensors connected to prototype b1 and hooked to the truck externally. The external pressure sensor were connected to the truck through a bridge connected on top of the internal pressure sensor that measures the loading pressure of the air intake system to the turbo. Which means they should measure roughly the same pressure. The CAN NOx sensor was placed in the exhaust pipe.

Figure 42: Upper left corner shows the NOx sensor, upper right corner shows the internal pressure sensors and the tube going to the external one. The bottom pictures are of the prototype box and how its internally connected.
9.2 Real testing. Exhaust emission control system

After the first verification testing a second test were done on the Exhaust emission control. This time the aim was to create a problem in the system and see if the prototype could be used to detect the problem. This would simulate how the prototype actually would work in a real work scenario.

9.2.1 Test rig Setup

To be able to simulate a problem a test rig of the Exhaust emission control system was used, see Figure 43. The system mainly consist of three parts. A tank that contains the liquid Adblue, a pump that produces pressure and a dosage unit that injects the calculated ad blue as an output to the rest of the system. To be able to create an error in the system a valve was added between the pump and the dosage unit. To quickly summarize how the system works: The pump creates a suction from the tank with Adblue and send the Adblue through to the dosage unit. The dosage unit also has an injector that outputs the right amount of ad blue out of the system. The dosing unit want to have the pressure of 9 bar when the outputs occur to give out the right amount of Adblue. The Adblue that are not sent out continue the path forward back to the tank.

As section 4.2.1 mentions, there only exist one internal sensor in this rig and is located in the dosage unit. The internal sensor in the dosage unit is a pressure sensor that also can read temperature) that measures pressure that are lead from the pump.
9.2.2 Problem setup

A common problem for the Exhaust emission control system is that the pump can not give the desired pressure to the injector. This may be because of different reasons: The pump can be broken or have a leakage, the pressure cable between the components are filled with soil or are leaking or that the filters in either of the components have been clogged. To simulate this, the added valve between the pump and dosage unit can either be open, choked or closed. The rig can be operated to tell the injector in the dosage unit to apply Adblue to the output of the system and when not to do it. The pump can also be operated to be active and inactive.

9.2.3 Using the prototype to solve problem

To solve the problem mentioned above, the prototype was connected (analog input) to an other external pressure sensor that was of the same type of sensor that is used internally in the system. The external sensor were inserted between the pump and valve that leads to the dosage unit, see optional placement for ext. sensor 1 in fig 43. Four different tests should then be made:

1. : Open valve, dosage apply.
2. : Choked valve, no dosage apply.
3. : Choked valve, dosage apply.
4. : Closed valve, no dosage apply.

With the placement of the external sensor and these four different test one should be able to locate were the problem is located. That is, if one already know that the problem is the added closed or choked valve obviously.
Figure 44: Upper left: entire test-rig, Upper right: Prototype box with connections. Lower left picture displays the actual location of the EEC-system and the lower right picture is a picture of the Ad-blue tank. The EEC system is located underneath or behind this tank.
10 Results

10.1 Testing and Verification

Figure 45: Results from testing the prototype on a Euro 6 truck. External Nox sensor (blue) compared to internal upstream Nox sensor (orange).
Figure 46: Results from testing the prototype on a Euro 6 truck. External NOx sensor (blue) compared to the downstream NOx sensor (orange).
Figure 47: Results from testing the prototype on a Euro 6 truck. External pressure sensor (orange) compared to internal pressure sensor (blue) located in the turbo charge.
10.2 Problem testing

Figure 48: Results obtained from test rig. The valve was completely open and no Ad-blue dosing was applied. Internal pressure sensor (blue) compared to external pressure sensor (orange).
Figure 49: Results obtained from test rig. The valve was choked and no Ad-blue dosing applied. Internal pressure sensor (blue) compared to external pressure sensor (orange).

Figure 50: Results obtained from test rig. The valve was choked and Ad-blue dosing was applied. Internal pressure sensor (blue) compared to external pressure sensor (orange).
Figure 51: Results obtained from test rig. The valve was closed and no AdBlue dosing was applied. Internal pressure sensor (blue) compared to external pressure sensor (orange).
10.3 Response time analysis

Figure 52: Matlab MPA model of the CAN bus results.

Figure 53: Results of the response time testing program being used on a Euro 6 Scania Truck. Blue columns indicate the average response time of 500 queries.
11 Discussion

11.1 Test and verification

The idea behind the test was to make sure that both analog and CAN sensors would work. The initial tests indicated that the prototype was working as intended, in Figure 46 we can see that the external CAN NOx sensor follows the internal one most of the time. It should be noted that the external sensor was not mounted at the same location as the internal, in fact they are quite far apart. Also the external sensor was not mounted in perfect alignment of the emission flow, see Figure 44.

Figure 47 displays the results obtained while testing the pressure sensor, once again the setup can be seen in Figure 44. Here we can see that the external sensor follow the internal one perfectly, except for an offset of about 0.2. The reason for this offset is boggling, because what we are looking at are the raw-values of identical sensors with identical setup, in a closed system. Normally the raw-values is not what should be logged but in this case we had a problem in the software with the interpretation of this sensor data. This issue however have been resolved in the current version of software.

11.2 Problem testing

11.2.1 Test 1, open valve and no dosing

This test were only made to see if the internal and external sensor were synced and gave the same values. As Figure 48 shows. Both sensors gave similar values before, during and after the pump was activated. The small difference of one bar lays withing the sensors error band, see Figure 39

11.2.2 Test 2, choked valve and no dosage apply

Choked valve means that pressure is coming to the dosage unit from the pump but not as easily as usual. The internal sensor tells the pump to keep on creating pressure until the dosage unit has 9 bar. When this happens, the pressure before the valve is much higher. This means that the pressure on the external sensor should be higher then the internal sensor, which Figure 49 shows.

11.2.3 Test 3, choked valve and dosage apply

When dosing is applied the pressure in the dosage unit drops. With a choked valve the external sensor should show a steady higher pressure then 9 bar and the internal should first show 9 bar but during the dosage apply drop in pressure. Figure 50 shows that the system works accordantly.

11.2.4 Test 4, closed valve and no dosage apply

In this test it does not matter if the dosage function is activated or not because the dosage unit does not have the pressure to output Adblue. As Figure 51 shows, the pressure on the external sensor before the closed valve builds up the pressure and then shuts down due to a system crash.
11.2.5 What to make from tests and how does the external sensor help

Currently it does not exist an error code for small pressure differences (about 1-2 bar) between the pump and dosage unit. This means that the dosage unit not always get up to 9 bar and can not give away the right amount of Adblue when needed. If not the right amount of Adblue is ejected, an alarm will go off and alarm the driver that something is wrong with the truck and the truck has to be troubleshooting. Usually when this happens the troubleshooting process leads to that something is wrong with the Exhaust emission control system, but what the problem is is still unknown. As mentioned in 4.2.1, to localize the problem today involves steps in workshop methods to exclude which component that does not work correct. The problem is that they are only able to measure the pressure in the dosage unit. A step in the workshop method is to measure the amount of ad blue that are ejected with an measuring glass. This can be tiresome due to one usually have to remove the hole Adblue tank. With our prototype this will not longer be needed. With an external sensor added, one can easily see if there is something wrong with the pump, or if something is wrong with the dosage unit or if its just a cloaked filter. If the external and internal sensors show the same low pressure value when the pump is activated there is something wrong with the pump. If the external sensor shows a higher value than the internal there is nothing wrong with the pump and the problem probably lays in some filter between them or in the pressure hose. We believe that if this prototype will be put in use, Scania is going to save money on not throwing away perfectly good pump unnecessary and will also save time on the troubleshooting process. With the prototype there is no need to remove the hole tank to be able to measure the ejected Adblue.

11.3 Response time testing

The results in Figure 52 tells us that the even though the maximum period length has been set to 500 milliseconds, affecting nearly half of all messages sent on the red CAN bus. The maximum delay is still only 0,5 millisecond. Which is nothing compared to the delay of an average IO request to all connected ECU’s which can be seen in figure 53. One conclusion to be drawn from this is (assuming the MPA model was setup correctly and that the results can be trusted) that the red CAN bus does not contribute much to the overall delay of a diagnostic IO request. This information does not matter much for synchronization purposes between internal and external signals but its important to remember when reading Section 12.3 in future work.

With regards to synchronization figure 53 is very interesting, there we can see that except for BMS (Brake Management System) and DIS (Distance Sensor) the average round trip time is below 30 milliseconds and the lowest being around 8 milliseconds for the SMS (Suspension Management System). Now, the average round trip time of an external CAN sensor request have not been measured but it could be calculated like this assuming a baud rate of 56 kbit, the MCU is not busy and that only one CAN sensor is connected.

\[
\frac{Outgoing\ Packet\ Size(\text{Bits})}{\text{BaudRate}(\text{Bits/second})} + \frac{Incoming\ Packet\ Size(\text{Bits})}{\text{BaudRate}(\text{Bits/second})} = RoundTripTime(\text{milliseconds})
\]

71
which means the round trip time would be:

\[ \frac{16}{56000} + \frac{112}{56000} = 2.29 \text{ms} \]

Connecting another sensor would increase the incoming packet size with 80 which means another 1.43 ms would be added to the total round trip time. So with a response time of roughly 2.3 ms from the external tool and minimum response time of 8 ms from the internal CAN bus the difference is only 5.7 seconds and for the maximum (BMS and DIS excluded) of 30 the difference is 27.7 ms. For a temperature sensor this difference really does not matter in any of the cases, for a pressure sensor it could, depending on the application. All tests done have been without any kind of synchronization and have been done with a frequency of 1 Hz or lower. Also worth mentioning is that when the response time test was done on the euro 6 truck only the first IO of each ECU was queried and the BMS have some IO’s that are turned on only when when requested, this is why the BMS has such a high average response time.
12 Future work

12.1 Hardware
The PCB should have a power supply trace from the power jack directly to the D connector with 24 volt so the CAN bus has current. This was discovered to late in the project and the hole PCB design would have to be redesigned to make the power jack and the D connector much closer to each other to make the trace between them as short as possible. This does not mean that the current PCB does not work. But need an external power supply source of 24 V to power the CAN bus. This can be done the same way as in the first prototype with the Arduino. See section 9.1.

12.2 Software
There are a lot of things that could have been made better and there are things that are not fully implemented yet. Synchronization between internal and external signals for instance is not fully implemented for two reasons. One being that the communication between microcontroller and computer was created without any synchronization in mind. With the current implementation one can only synchronize an internal ECU with all connected CAN sensors and not one internal with one external. This is obviously a problem if synchronization is needed and there are more then one CAN sensor connected to the external module. This could be fixed with a more sophisticated two-way communication protocol between the main program and the MCU, but that is left as future work. The other reason being that synchronization was considered of lower priority then the otherwise overall functionality of the system. Another thing left as future work is to run all internal and external sensor data requests on separate threads with individual timers. This is required if synchronization between external and internal sensors is suppose to work. Lastly, there are some sensors that requires a turn-on message to be sent. During this project only one has been encountered and handled, the NOx sensor. This sensor sends messages with a certain identifier when it’s not turned on and sends message with another id when its turned on. Due to the crude communication protocol currently used this scenario is handled in the MCU. The MCU listens on the CAN bus for the NOx sensors IDLE identifier and if it is detected it will then send the start-up message. It would be better if this was solved in the software running on the computer, IE SPD3 or the External Sensor Module program. To solve this a list of all sensors requiring start-up messages is needed.

12.3 Recommended future setup
While our solution works and has been successfully used to find problems that the internal system could not, we believe there are better ways to do it. An idea we had early on was to have a what we call a ”Diagnostic ECU”, this ECU would have a reserved CAN identifier on the red, green and yellow CAN bus and could be moved between these buses depending on the need. The ECU could be constructed in a similar way to how we created our external module. With one CAN controller talking to the internal CAN bus and one CAN controller for talking with external ECU’s, the analog sensors could be
connected to the analog inputs of the microcontroller. This would solve all synchronization problems since all sensors now reside on the same bus. Another potential problem that we have not taken into account is the various cable length for sensors and the USB-cable that might be needed to troubleshoot the vehicle. With the "Diagnostic ECU" you would only need to connect to the OBD port to request data from both internal and external sensors.

The reason we did not do this ourselves is that this would require modification of SCOMM and a deeper knowledge of the internal CAN bus.

13 Conclusion

During this project we have looked at various ways to connect and obtain data from external sensors and internal sensors. We have investigated potential delays within the internal CAN bus and how these could be handled and how synchronization between internal and external could be achieved. A prototype was created based on the research done during this project. This prototype was successfully used to troubleshoot and find a problem on a test rig where an error had been purposely introduced, reducing the whole troubleshooting process to a fraction of the time it normally takes.
References