Modularity of a heater simulator and implementation in a prototype HIL-system

CHRISTOFFER LINDAHL
MATTIAS NILSSON
Abstract

The thesis is about a modular low budget prototype hardware-in-the-loop system for testing diesel heaters. A modular plant model of the heater has been developed using model-based development in Simulink. The model has been derived mathematically from tests and empirical investigations. C-code was generated from the plant model and implemented in a microcontroller. The microcontroller simulates the model and sends the data values to different sensor emulators so that the electrical control unit is tricked into thinking it is connected to a real heater.

The modularity is implemented in both software and hardware of the HIL-system. The plant model is modular by using different subsystems for the different parts of the heater. The hardware was also designed to be modular, in the form of "fake" sensors that can be changed and also in the form of the different loads that represent the actuators, such as fan, in the heater. These loads can be changed in size depending on what is needed.

An important aspect was to have the HIL-rig as low budget as possible. The tests show that the HIL-rig is suffering from bad resolution on the temperature during higher temperatures. This is due to the fact that the digital potentiometer used had a fairly low resolution. This performance could be improved using potentiometers with a higher resolution, but it would increase the price.
Sammanfattning


Modulerad systemet är implementerat i både mjukvaran och hårdatvaran hos HIL-system. System-modellen är modulär genom att använda sig av olika undersystem för de olika delar av värmaren. Hårdatvaran var också designad att vara modulär, i formen av ”simulerade” sensorer som kan bytas ut samt i formen av olika laster som representerar de olika aktuatorer, så som fläkten i värmaren. Dessa laster kan bli förändrade i storlek beroende vad som behövs.

Acknowledgements

The authors would like to thank the Reformtech supervisors Sebastian Wätz and Andreas Eriksson for the help, support and guidance through this project. As well as the supervisor, examiner and coordinator at the university, Bengt Eriksson, Lei Feng and Damir Nesic.
Contents

1 Introduction 1
  1.1 Background ...................................................... 1
  1.2 Purpose and definition ............................................ 1
  1.3 Research Questions ............................................... 3
  1.4 Method .......................................................... 3
  1.5 Thesis responsibilities .......................................... 4

2 Frame-of-reference 5
  2.1 HIL ............................................................. 5
  2.2 Modularity ......................................................... 6
    2.2.1 Modular models in simulink for code generation ............. 8
    2.2.2 Hardware ..................................................... 9
  2.3 Continuous flame heater ......................................... 9
  2.4 Oxygen measurement ............................................ 10
    2.4.1 Stoichiometric ratio ....................................... 10
  2.5 Heat exchange ................................................ 11

3 Implementation 13
  3.1 Heater analysis ................................................ 13
    3.1.1 NTC Temperature sensors .................................... 15
    3.1.2 Nozzle thermocouple sensor ................................ 15
    3.1.3 Fuel pressure sensor ....................................... 16
    3.1.4 Wideband lambda sensor ................................... 16
    3.1.5 Fan feedback sensor ....................................... 17
    3.1.6 Fan .......................................................... 19
    3.1.7 Fuel-pump ................................................... 19
    3.1.8 Lambda heater ............................................... 19
    3.1.9 Other actuators ............................................ 19
  3.2 Heater model ................................................... 20
    3.2.1 Model development ......................................... 20
    3.2.2 Simulink and code generation ............................... 23
    3.2.3 Main Model ................................................ 24
    3.2.4 Air Fan Model ............................................... 24
    3.2.5 Coolant Pump Model ....................................... 26
    3.2.6 Fuel Pump Model ........................................... 26
    3.2.7 Nozzle Model ............................................... 27
    3.2.8 Fuel Combustion Model .................................... 27
    3.2.9 Heat Exchange and Cooling Model ............................ 29
  3.3 HIL environment ................................................ 30
    3.3.1 Microcontroller ............................................ 30
    3.3.2 I/O ports .................................................. 30
    3.3.3 Main TivaC program ....................................... 31
  3.4 Sensor Emulation ................................................ 32
    3.4.1 Lambda module ............................................. 33
    3.4.2 Thermistor module ........................................ 35
3.4.3 Nozzle thermocouple module ........................................ 35
3.4.4 Fuel pressure module .................................................. 35
3.4.5 Sensor module development .......................................... 36
3.5 Cooling of the power sinks ............................................. 38
3.6 Complete HIL-system .................................................... 41

4 Analysis/Result .............................................................. 47
4.1 How the test was made ................................................... 47
4.2 Results ......................................................................... 47
4.3 Budget/Cost .................................................................. 53

5 Conclusion ................................................................. 55

6 Discussion/Recommendation ............................................... 57
6.1 General Discussion ....................................................... 58
6.2 Future work ............................................................... 58

A Main Model ................................................................. 64
A.1 Air Fan Model ............................................................ 65
  A.1.1 Motor Model: ....................................................... 66
  A.1.2 Fan Model: ......................................................... 68
A.2 Coolant Pump Model .................................................... 69
A.3 Fuel Pump Model .......................................................... 70
A.4 Nozzle Model ............................................................. 72
A.5 Fuel Combustion Model ................................................ 73
A.6 Heat Exchange and Cooling Model .................................. 76

B Code Listings ............................................................... 78
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A general HIL-system implementation</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Component swapping modularity</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Component sharing modularity</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Fabricated to fit modularity</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Bus modularity</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>A functional overview of the heaters component</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>A functional NTC overview</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>Main functionality overview of LSU 4.9</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>Single channel encoder</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>First draft of Plant Model</td>
<td>21</td>
</tr>
<tr>
<td>11</td>
<td>Main draft of Plant Model</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>Duty cycle to ramp validation</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>Motor validation of RPM</td>
<td>25</td>
</tr>
<tr>
<td>14</td>
<td>Fuel Pump Gain Validation</td>
<td>26</td>
</tr>
<tr>
<td>15</td>
<td>Nozzle flow validation</td>
<td>27</td>
</tr>
<tr>
<td>16</td>
<td>Lambda output logic</td>
<td>28</td>
</tr>
<tr>
<td>17</td>
<td>Fuel Energy output logic</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>Verification of the model against the real system</td>
<td>29</td>
</tr>
<tr>
<td>19</td>
<td>TivaC Connector View</td>
<td>31</td>
</tr>
<tr>
<td>20</td>
<td>Main function flowchart</td>
<td>32</td>
</tr>
<tr>
<td>21</td>
<td>Circuit of the lambda sensor simulator chip</td>
<td>34</td>
</tr>
<tr>
<td>22</td>
<td>Pressure Circuit Layout</td>
<td>36</td>
</tr>
<tr>
<td>23</td>
<td>Sensor Emulation Boards View One</td>
<td>37</td>
</tr>
<tr>
<td>24</td>
<td>Sensor Emulation Boards View Two</td>
<td>38</td>
</tr>
<tr>
<td>25</td>
<td>The resistor connection</td>
<td>39</td>
</tr>
<tr>
<td>26</td>
<td>Thermal resistance of the components</td>
<td>41</td>
</tr>
<tr>
<td>27</td>
<td>HIL-system functionality</td>
<td>42</td>
</tr>
<tr>
<td>28</td>
<td>The HIL rig</td>
<td>43</td>
</tr>
<tr>
<td>29</td>
<td>Coolant tank and power sinks</td>
<td>44</td>
</tr>
<tr>
<td>30</td>
<td>Power sinks</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>Temperatures in the real system in a normal test</td>
<td>48</td>
</tr>
<tr>
<td>32</td>
<td>Temperatures in the model during a similar test as in the real world</td>
<td>49</td>
</tr>
<tr>
<td>33</td>
<td>Temperatures in the HIL system during the same test</td>
<td>50</td>
</tr>
<tr>
<td>34</td>
<td>Fan speeds from the real system, HIL system and the model compared to each other</td>
<td>51</td>
</tr>
<tr>
<td>35</td>
<td>Fan speeds and duty cycle from real system and HIL system compared to each other</td>
<td>52</td>
</tr>
<tr>
<td>36</td>
<td>Pump PWMs in the real system, HIL system and the model compared to each other</td>
<td>53</td>
</tr>
<tr>
<td>37</td>
<td>Main Model Simulink</td>
<td>64</td>
</tr>
<tr>
<td>38</td>
<td>Air Fan Model Simulink</td>
<td>65</td>
</tr>
<tr>
<td>39</td>
<td>Motor Model Simulink</td>
<td>67</td>
</tr>
<tr>
<td>40</td>
<td>Fan Model Simulink</td>
<td>68</td>
</tr>
<tr>
<td>41</td>
<td>Coolant Pump Model Simulink</td>
<td>69</td>
</tr>
</tbody>
</table>
List of Tables

1. Table of molar mass ................................................. 11
2. Heater models input and output signals .......................... 22
3. Variables in the heat exchanger model ............................. 29
4. Load on the resistors ................................................. 40
5. Thermal resistances .................................................. 40
6. Components ......................................................... 54
7. Motor models variables .............................................. 66
8. Fuel pump polynomial coefficients ................................. 70
9. Fuel pump PI controller ............................................. 70
10. Fuel flow polynomial coefficients ................................. 72

List of Listings

1. Sample from "Motor.h" file ......................................... 78
2. Sample from "Motor.c" file .......................................... 78
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIL</td>
<td>Hardware-In-the-Loop</td>
</tr>
<tr>
<td>ECU</td>
<td>Electrical Control Unit</td>
</tr>
<tr>
<td>HWIL</td>
<td>Hardware-In-the-Loop</td>
</tr>
<tr>
<td>AFR</td>
<td>Air-Fuel Ratio</td>
</tr>
<tr>
<td>FAME</td>
<td>Fatty Acid Methyl Esters</td>
</tr>
<tr>
<td>RME</td>
<td>Rapeseed Methyl Ester</td>
</tr>
<tr>
<td>GUI</td>
<td>Grapical User Interface</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>NTC</td>
<td>Negative Temperature Coefficient</td>
</tr>
<tr>
<td>MBD</td>
<td>Model-Based Development</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital To Analog Conversion</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog To Digital Conversion</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>LHV</td>
<td>Lower Heating Value</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>B100</td>
<td>100% Bio Fuel</td>
</tr>
<tr>
<td>HEX</td>
<td>Heat Exchanger</td>
</tr>
<tr>
<td>LSU</td>
<td>Lambda Sensor Unit</td>
</tr>
<tr>
<td>ZrO₂</td>
<td>Zirconium Dioxide</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse-width Modulation</td>
</tr>
<tr>
<td>RPM</td>
<td>Revolution Per Minute</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>CCS</td>
<td>Code Composer Studio</td>
</tr>
<tr>
<td>PCB</td>
<td>Prototype Circuit Board</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>CLK</td>
<td>Clock Timing</td>
</tr>
<tr>
<td>SS</td>
<td>Slave Select</td>
</tr>
<tr>
<td>MOSI</td>
<td>Master Out Slave In</td>
</tr>
<tr>
<td>MISO</td>
<td>Master In Slave Out</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
</tbody>
</table>

**TivaC**  The microcontroller used in the thesis
1 Introduction

This section describes the thesis background of a low-budget HIL-system for testing of an open-flame heater. The HIL-system was developed together with a small company with a limited budget. Therefore cost efficiency was a priority. To be able to ensure a low cost, the HIL-system was developed from scratch. The thesis was limited to a prototype, since it was in a first development cycle. The thesis was also designed to check if modularity could be implemented in the HIL-prototype.

1.1 Background

Hardware-In-the-Loop (HIL) testing is an established and well used process in industries. A HIL-system can help test and verify the Electric Control Unit (ECU) and controller software. A HIL-system allows for testing the control logic without the need for a real world test that can be time consuming and expensive. A general HIL-system consists of hardware and software. The hardware can be described as a HIL-rig and consist of Input/Output (I/O) ports, sensor/actuator emulators and real-time processor. The software consists of the real-time operating system and a model over the real world product. This model that simulates the product is generally called a "plant model".

ReformTech is a company that develops heaters. The company wanted to evaluate if a HIL-system would be a good way to replace or complement the current testing process. The company did not have the budget to buy an industrial general purpose HIL-rig, and therefore wants to develop an own HIL-prototype rig. The prototype HIL rig was to be designed with the specific ECU in mind, and not to be a general purpose rig. ReformTech is developing continuous flame heaters for the automotive industry. The heaters produce different effects depending on the components and fuel that are used. This gives the product a range of options and qualities.

1.2 Purpose and definition

The purpose of this thesis was to develop a low budget HIL-system and investigate the possibility of modular design.

To test and verify the control software for the heaters in a HIL-test, plant model of the heaters was needed. It’s time consuming and expensive to develop a specific model for each and every product version in the product range. The plant model was developed as a modular model that was parametrized to cover several of the products and easily be expanded by having addable, removable and/or interchangeable subsystems. The plant model was created in simulink and converted to C-code and then implemented on the prototype HIL-rig. The modular design of the plant model was needed to keep its modularity when the C-code had been generated from
Simulink. The future use of the HIL-system will only use C-code, both for running
the test and improving the plant model. The Simulink design approach was used
to create a good foundation to start the development of the plant model, since it
allows for logic testing during the development. It also allows for a model based
development process.

The HIL project had a limited budget of around 10 000 SEK, so it was not feasible to
buy multipurpose hardware/software products from companies like dSpace/"National
Instruments". The HIL-rig that the model was applied on, was designed with an
economical cost in mind. Developing the HIL-rig hardware with a low budget was
vital of the project.

To ensure the success of the project, limitations were defined.

The minimum updating frequency was set to 100 Hz, due to the ECU sample rate
of the sensors in the heater.

One of the actuators has the ability to self regulate the power consumption de-
pending on its temperature as long as it’s supplied by a power source. One of the
limitations of this thesis was to not automatically control the actuator’s power con-
sumption. If needed the user has to do this manually.

The HIL-system needs to run one normal operation cycle of the heater. During
the cycle, no errors can be triggered in the ECU. A cycle can be summarised to four
major states.

The restrictions were set up as bullet-points:

- The updating frequency needs to be minimum 100 Hz.
- The actuator currents needs to be drawn.
- No automatic control of the actuator currents.
- Not trigger any ECU errors.
- The HIL-system needs to be able to run a normal test cycle.

  1. Initialize state
  2. Pre-heating state
  3. Normal operation state
  4. Cooling down state

And if it was time, the following should be implemented in the second major iteration

- Error injection in the model
- LabView GUI for control of the HIL-system.
1.3 Research Questions

The research questions for this thesis were formulated as following:

- How to create a modular model of a heater?
- How can the model be parameterized in order to support different heater models?
- Can the model be created to have interchangeable/addable/removable subsystems?
- What are the limitations of a non-expensive rig?
- How will a test of the parameterized models in the prototype HIL-system compare to a test of the heater regarding performance and accuracy?
- Can the model be used to simulate a behaviour that is not possible with the real heater? If yes, how?

1.4 Method

The plant model of the heater was developed using model-based development that fit under a quantitative paradigm. Mathematical models of several physical domains were used as a base for the plant model where signal and energy flows was studied and applied to the model. The model was derived from the heater and measurement taken from it in a controlled lab. The plant model corresponds to the lab tests, and not a heater that was mounted in an vehicle. This decision was done, since there was no data from that environment available.

The HIL system takes both discrete-time and continuous-time into account since it has both hardware and software interactions. Model-based development is a useful tool when working with signals passing between different time frames of digital and analog systems. The plant model developed in Simulink runs on a processor that is in discrete-time, therefore the plant model was developed for a discrete-time operation. The main code that controls input and output from the HIL-rig and sends the data to the plant model operates with both continuous- and discrete-time.

The major work was done together by the authors, but the responsibilities was divided up between them. Each worked and developed parts separate, but still with input from the other person. The report was written together, but the authors was responsible for different chapters that corresponds to the responsibility of each author.
1.5 Thesis responsibilities

Mattias Nilssons responsibilities was the general modularity structure and code generation of the models, the emulation of the lambda sensor, the main code of the HIL-system and several submodels.

Christoffer Lindahls responsibilities was the hardware (cables, power consumption, power sinks and cooling) of the HIL-system. The heat exchanger model, emulation of actuator signals to the HIL-rig.
2 Frame-of-reference

This chapter describes relevant information for the thesis.

2.1 HIL

A HIL system is a test system that is built to emulate a real world system/product. A plant model is created of the product (plant) and implemented in the HIL-system. The ECU that controls the plant can instead be connected to the HIL-System, as seen in figure 1. The HIL is operating in real-time and tricks the ECU to "think" it’s controlling the plant, since the ECU still receives the same kind of signals. A HIL-system can be designed to simulate a whole product/system or a specific subsection.

Benefits of a HIL system can be cost effectiveness(time/money), increased safety and quality [1] by enable testing continuously during the development process. HIL can be used for tests of new software, without having to put the software in the real plant. Depending on what the system is, a real-world test of new software can be expensive and/or dangerous.
There are companies that develop tools for HIL-systems, such as MathWorks[2]. There are also companies that develop parts for and/or complete HIL-systems for customers. Some of those companies are dSPACE[3], National Instruments[4], and Opal-Rt[5]. HIL-systems can help save money during development, but buying a HIL-system is a big investment. The hardware and software can cost a lot of money that a small company can’t afford.

HIL-systems are not limited to big and expensive equipment, there are works that has developed a simple low-cost HIL platforms[6] for testing and verifying the control strategy. Model based development process with Matlab/Simulink has been used to generate code for a Raspberry Pi based real-time hardware testbed [7].

A HIL-system must imitate the plant as close as possible, including sending the right signals at the right time, be fast enough and drawing the right amount of current in the right places. A good control software checks for all of these things, and has fail/error states that triggers if something is wrong. The HIL rig has a limited budget and therefore it’s limited in what it can do and many of the design decisions are based on what is the cheapest way but also has a decent quality.

2.2 Modularity

The word "Modularity" can have different meanings in different fields, it depends on the context where the word is used. Examples of fields where the term "modularity" can have different meanings can be "product development", "organizational structures" and "company strategies". The word "modularity" is not clearly defined, and neither are its interaction with other aspects. A good example of this is how it interact with integrality. The traditional view is that modularity and integrality are at opposite ends of the same spectrum. On one end there is modularity with interchangeable components/systems, and the other end is integrality with integrated components/systems that can not be switched. A different approach is suggested in [8], where the relation between modularity and integrality is developed and new definition of both words are suggested. The paper focus on the organizational structures of modularity, but [9] discuss it with design in mind and the benefits product development.

This thesis will focus on modularity in product development with hardware/software aspects. In this thesis, modularity is used in the context of modular design (modularity in design). Modular design is a method of thinking that a system can be divided in to smaller parts, called "modules". The modular system consist of parts that is developed independently, but combined functions as a whole complete system [8]. The modules can be created/changed independently from one another since pre-determined interfaces is used.

In modular design, a products modularity can be represented in four ways according
to [10].

The first is **component-swapping modularity**, see figure 2. It’s when one kind of base uses different kinds of addons on the same "port". An example of this can be a screwdriver with exchangeable heads.

![Figure 2: Component swapping modularity](image)

The second is **component-sharing modularity**, see figure 3. It’s when different kinds of bases uses the same type of addon. An example of this is how different types of smartphones uses the same type of charger port.

![Figure 3: Component sharing modularity](image)

The third type is **fabricate-to-fit modularity**, see figure 4. It’s when two or more standard designed interfaces uses a variable additional component. A good example of this is a cable with connectors at either end. The wire can be of a variable length, but the connectors are the interfaces.
The last type is **bus modularity**, see figure 5. It’s when different modules can be matched/placed in different volumes and places. An example of this is how a computer can use different amounts of Random Access Memory (RAM) components in different slots.

The different types of product modularity can help set up boundaries with what type of modularity is useful for a product/project. This can be used when designing a modular system. The system is analyzed by using "structural decomposition"[10], to identify the subsystems and the subsystems components. With this information a hierarchical structure can be viewed/observed.

### 2.2.1 Modular models in simulink for code generation

Simulink is a good tool to use for model based development developed by MathWorks[2]. It’s graphical programming environment that allows the user to develop and simulate models. It’s well used in education/industries and has a lot of third party adaptations that can be used together with it. Example of this can be HIL systems developed by companies like dSPACE[3], as mention earlier. The HIL-system then runs directly on the simulink model, since dSPACE has developed third party tools for Simulink.

For a low budget HIL prototype, that type of expensive software can’t be used.
When running the HIL-system, the model must be in C-code. This means that the simulink model must be translated into C-code. This can be done through automatic code generation in the simulink program. The problem with code generation is that often the generated code is not good. The generated code can be hard for a human to read or the code executes slowly. A normal made simulink model might not translate into good code, since variables might not behave the same when generated from a graphical programming language to a code language.

In "modular code generation", code should be able to be generated independently from context, and the "main block" should have minimal knowledge of its sub-blocks[11]. The paper also discuss how the term "modularity" is connected to "re-usability". That "modularity" can be measured in number of interfaces, where the fewer interfaces a module has the higher degree of modularity it posses, but this gives the module a lower re-usability, It's a trade of. The paper goes into modularity vs code size, of how modularity can improve the code size. But the optimal size of the generated code does not corresponds to the highest degree of modularity.

2.2.2 Hardware

To be able to emulate the sensor signals going back into the ECU, module boards for sensor emulation is needed. The sensor emulation boards will translate the calculated digital value that is returned from the plant model into an analog signal that is sent to the ECU. The boards will need to convert the signal using cheap component and a predetermined interface. The goal is to have the sensor emulation boards as modular as possible.

A good solution would be to have a "Field Programmable Gate Array"(FPGA) to control where the signal is sent. But due to budget and time consideration, the boards will be developed without it.

2.3 Continuous flame heater

The heater is a continuous flame heater, also called an indirect fuel-fired heater. It works on the same principle as a boiler or a tank-less water heating system for houses. A energy source warms up a liquid medium.

A continuous flame heater uses a fan, and a swirler to blow in air with a turbulent flow into the mixing chamber/area. A fuel nozzle is mounted after the swirler but before the mixing area, and disperse the fuel as a fine spray into the turbulent air. The air/fuel mixture is then ignited by a glow plug. The heat that is generated and contained in the exhaust is then transfers to a continuously flowing liquid medium by a heat exchanger. The liquid is then used to heat the intended target.

2.4 Oxygen measurement

Lambda sensors is used to measure the oxygen proportion in a gas medium. The proportion of oxygen in the exhaust corresponds to the air-fuel ratio (AFR) that
enters the combustion area.

\[ AFR = \frac{m_{air}}{m_{fuel}} \]  \hspace{1cm} (2.1)

Each type of fuel has a Stoichiometric ratio \((AFR_{stoich})\) that corresponds to the amount of air that is needed to completely burn a certain amount of fuel.

The lambda value \((\lambda)\) is used to see if the combustion is rich or lean.

\[ \lambda = \frac{AFR}{AFR_{stoich}} \]  \hspace{1cm} (2.2)

When \(\lambda = 1\) the combustion is at a perfect ratio. The fuel reacts with all the oxygen atoms. \(\lambda < 1\) the combustion is rich. There is more fuel than oxygen. \(\lambda > 1\) the combustion is lean. There is more oxygen than the fuel can bind.

### 2.4.1 Stoichiometric ratio

The word Stoichiometry can be viewed as the balancing of an equation for a chemical reaction. It’s a extension of the law of conservation for mass[12] which itself is an version of the first law of thermodynamics, the conservation of energy[13]. To get a stoichiometric ratio of a chemical reaction, the molar balance between the ingredients must be calculated, not the mass. To calculate the stoichiometric ratio for the two fuels, a formula that calculates the molar balance of the combustion can be used. It’s a general approximation that only uses the average hydrocarbon atom-number of the fuel-mix and a simplified version of air. The air is approximated into 21% air and 79 % nitrogen. By using Stoichiometric balancing, an equation can show how much air is needed to balance one mole fuel[14], as can be seen in

\[ C_nH_m + \lambda \cdot (n + \frac{m}{4})[O_2 + \frac{0.79}{0.21}N_2] \rightarrow n_2 + \frac{m}{2} \cdot H_2O + \frac{0.79}{0.21} \cdot (n + \frac{m}{4})N_2 \]  \hspace{1cm} (2.3)

Where \(n\) is the average number of carbon atoms and \(m\) is the average number of hydrogen atoms in the fuel-mix. The equation above gives the molar basis ratio, but the mass ratio is needed for \(AFR_{stoich}\). This can be calculated by multiplying each of the atoms molar mass[15] as can be seen in table 1.

If \(\lambda\) is set to one, the stoichiometric value can be calculated. For one part fuel the following air is needed.

\[ AFR_{stoich} = \frac{(n + \frac{m}{4})(O_2) + \frac{0.79}{0.21}(N_2)}{M(C_n) + M(H_m)} \]  \hspace{1cm} (2.4)

"\(M(\text{Atom}_x)\)" indicates the atomic weight of an atom. The letter \(x\) indicates a number of atoms, and can be extracted to multiply the Atomic weight.

\[ M(\text{Atom}_x) = x \cdot M(\text{Atom}) \]  \hspace{1cm} (2.5)
Table 1: Table of molar mass

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Atomic weight [g/mol]</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>carbon</td>
<td>12.0107</td>
</tr>
<tr>
<td>H</td>
<td>hydrogen</td>
<td>1.00794</td>
</tr>
<tr>
<td>O</td>
<td>oxygen</td>
<td>15.9994</td>
</tr>
<tr>
<td>N</td>
<td>nitrogen</td>
<td>14.0067</td>
</tr>
</tbody>
</table>

The product will use two types of fuel:

**Diesel:**
The diesel that is used is petrodiesel and is approximated to $C_{13.883}H_{24.053}$, which is a value used internally in the company, and is taken from [16]. The molar balance equation is:

$$C_{13.883}H_{24.053} + 19.896O_2 + 74.848N_2 \rightarrow 13.833CO_2 + 12.026H_2O + 74.848N_2$$  \hspace{1cm} (2.6)

This gives the $AFR_{stoich} = 14.3118$

It’s close to what the average diesel stoichiometric value are considered to be around 14.5[17]

**Biodiesel:**
The biodiesel that is used is Rapeseed Methyl Ester(RME). RME is one type of biodiesel that can be named as Fatty Acid Methyl Esters(FAME). The fuel is not mixed with anything and is therefore consider to be B100( 100% Biofuel). Biodiesel is usually considered of having FAME ranging between $C_{12}$ to $C_{22}$[18]. No useful average data about the FAME in the biodiesel has been found, so the stoichiometric value that is used is 12.5[17].

2.5 Heat exchange
The heat exchanger is the part that transfers heat from the open flame burner into a liquid medium that makes it easier to distribute the heat evenly in a target area( such as the driver cabin of a vehicle). The heat exchanger is neither of the 3 simple forms of heat exchangers (parallel flow, where the two mediums are flowing along each other, counter flow, where the mediums are flowing against one another, strict cross flow, where the mediums crosses one another). It can be seen as a more complex version of cross flow heat exchanger, but because of this the thermodynamic flow calculations would be a lot more complex. The more complex calculations would be heavier for the simulation and would probably not yield a better result. So instead of using those the equations for the efficiency, or $\eta$, is taken from previous tests made by Reformtech. This will yield a simpler, yet still accurate model. The heat
transferred, $\dot{Q}_{th}$, can be calculated from the chemical heat flow, $\dot{Q}_{ch}$. So the heat transferred from the burner to the coolant mixture can be calculated from

$$\dot{Q}_{th} = \eta \dot{Q}_{ch} \quad (2.7)$$

Using the calculated $\dot{Q}_{th}$ the temperature difference, $\Delta T$ can be calculated with

$$\dot{Q}_{th} = \dot{m} c_p \Delta T \quad (2.8)$$

From $\Delta T$ and with a known temperature on the coolant flowing in, $T_{in}$, the temperature on the coolant flowing out, $T_{out}$ can be calculated with

$$\Delta T = T_{out} - T_{in} \quad (2.9)$$

The mass flow in the system can be calculated from the density and the volume flow of the system. Since the volume flow is known and the density is also known the equation will be

$$\dot{m} = \rho \dot{v} \quad (2.10)$$

However, both the density, $\rho$, and the specific heat capacity, $c_p$, will vary based on the temperature of the medium. The flowing medium used is a mixture of glycol and water with 60% glycol. $c_p$ is based on the temperature, $T$, (here $T = T_{in}$, $T$ is in degrees Celsius). The formula for specific heat capacity for the mixture is

$$c_p(T) = 3.8815T + 3112.8 \quad (2.11)$$

and the density for water and glycol are

$$\rho_{water}(T) = 1000.4 - 0.0036T^2 - 0.0656T \quad (2.12)$$

and

$$\rho_{glycol}(T) = 1130.4 - 0.6833T \quad (2.13)$$

The resulting temperature can be calculated from equations 2.8 and 2.9. Then the heated water is mixed with the rest following the standard formula of mixing

$$T = \frac{m_1 c_{p1} T_1 + m_2 c_{p2} T_2}{m_1 c_{p1} + m_2 c_{p2}} \quad (2.14)$$

which gives the resulting temperature in the system.
3 Implementation

This section describes how the HIL-prototype was developed. It contains most of the work in this thesis.

3.1 Heater analysis

The heater has eight sensors and eight actuators that combined controls the heater. Sensors:

- 5 Temperature sensors
- 1 Fuel pressure sensor
- 1 Fan feedback sensor
- 1 lambda sensor

Actuators:
- Fan
- Fuel pump
- Nozzle heater
- Glow-plug
- Coolant pump
- Fuel filter heater
- Fuel valve
- Lambda heater

The sensor and actuators are shown in figure 6. The figure is a functional overview of the heater and does not represent the physical design of the heater.
The fuel pump sucks the fuel through a fuel filter that has a heater inside it. The ECU always supplies voltage to the fuel-filter heater, but the fuel-filter has its own internal temperature sensor and control system that turns the heater on/off depending on the temperature. After the fuel have passed the pump, it has a higher pressure. The fuel enters into the nozzle area, where another heater called the nozzle heater warms it and the surrounding area. This nozzle heater is controlled by the ECU, that uses the data from the nozzle temperature sensor to control the heater. The high pressure fuel then reaches the nozzle where an ECU controlled actuator valve controls if the fuel flows through the nozzle and sprays into the mixing chamber/area.

The fan sucks air in from the surrounding and pushes it past a swirler which makes the air flow turbulent when it reach the mixing chamber/area. The fuel gets sprayed out from the nozzle into the turbulent flowing air. The fuel and air mixes and flows forward until it reaches the glow-plug. The ECU controls the glow-plug and ignites the fuel mixture. The glow-plug is only needed when igniting a new flame, if it already is burning the continuous flowing air mixture will ignite on the existing flame.

The hot exhaust from the chemical reaction keeps moving through a heat exchanger(HEX) where it gets cooler. Then the exhaust moves past the lambda sensor and out from the heater. The lambda sensor detects the oxygen content of the exhaust and sends the data back to the ECU.

The coolant pump sends the coolant liquid from the coolant tank past the first temperature sensor into the HEX, where it starts to absorb heat. Inside the HEX Figure 6: A functional overview of the heaters component
the coolant passes another temperature sensor that is mounted close to the inner
wall of the HEX. The coolant liquid continue flowing forwards and exits the HEX
where it passes the last coolant temperature sensors and flows back into the coolant
tank.

This was the laboratory setup, from which the data has been collected. The model
was created to simulate this setup. The coolant gets taken from and put back into
the same tank, the tank has another external HEX that passes the energy from the
coolant liquid to the air. This was a separated system that was not modeled.

3.1.1 NTC Temperature sensors

There are four temperature sensors of the type Negative Temperature Coefficient (NTC).
The sensors are mounted in the coolant input, heat exchanger, coolant output and
at the air input as can be seen in figure 7. The data from the sensors can be used as
a safeguard against overheating and mode control. NTC sensors internal resistance
is temperature dependent. The resistance gets lower the higher the temperature is.

![Figure 7: A functional NTC overview](image)

3.1.2 Nozzle thermocouple sensor

On the nozzle block inside the heater a thermocouple sensor of type J is mounted.
It’s a temperature sensor that work different from a NTC sensor mention above.
A thermocouple consist of two different types of wires connected in a junction at
one end. The other ends is connected by extension wire to a external voltage mea-
urement device. The junction is the point that is exposed to the temperature that
needs to be measured. The temperature energy generate a voltage in the circuit
because the two different types of wire. The voltage strength and direction changes
depending on the temperature. The voltage difference is in the mV range.

The J type wires consist of iron in one and copper-nickel in the other[19].
3.1.3 Fuel pressure sensor

The pressure sensor acts as a variable resistance that changes depending on the pressure. Higher pressures give lower resistance.

3.1.4 Wideband lambda sensor

The sensor that is used in the heater is a Bosch lambda sensor LSU 4.9 [20]. It’s a wideband lambda sensor that works together with a control unit called CJ125 [21]. The control unit is integrated into the ECU, while the sensor is connected by cabling to the heater.

The LSU 4.9 sensor is designed to have a measurement chamber with a small opening towards the exhaust gas. A small amount of exhaust enters the chamber and moves through a diffusion barrier. Inside the barrier there is a metal coating that acts as a catalyst between the oxygen ions and any unconsumed fuel. This coating acts as a common ground both electrically and chemically for two different cells. Each cell consist of two contact areas/nodes and a layer of Zirconium Dioxide ($\text{ZrO}_2$) between them [22]. When the $\text{ZrO}_2$ is heated up it gets the ability to move oxygen ions through it. The cells acts as electrochemical cells, and creates a voltage difference over the nodes. Figure 8 shows a representation of this.

The first cell can be called the reference/concentration cell. It uses the voltage difference between the node inside the measurement chamber and a node only connect to an electrical wire. This voltage difference changes depending on what state the exhaust inside the diffusion barrier of the measurement chamber is in.

The second cell is called the pump cell, which is between the exhaust gas and the measurement chamber. The same coating exist on the outside against the exhaust gas. Oxygen ions can travel through and from the measurement chamber contact area to the exhaust gas contact area.

In older versions, the second node of the reference cell was against outside air but that was not reliable in harsh environments. The LSU 4.9 instead uses a measurement voltage of 450 mV that corresponds to lambda = 1 (perfect combustion). This voltage it subtracted to the voltage of the reference cell. If the difference between the voltages is positive, the exhaust gas is rich and if the difference is negative the exhaust gas is lean. The control circuit checks this value and uses it to control the value and direction of the current to the pump cell. The current forces the oxygen ions through the $\text{ZrO}_2$ layer, the cell acts as a pump for the oxygen hence the name. The control unit also check if current flows through the pump cell and thereby use that information to control the heater.
3.1.5 Fan feedback sensor

The fan motor has an incremental encoder that sends data back to the ECU. The encoder currently used, has six measurement points (often called ticks) per revolution, this is shown in figure 9. The encoder has only one signal channel.
When the motor is rotating, measurement points passes by the encoder sensor. The sensor detects the measurement point, and the output signal goes high. When the measurement point passes by, the sensor can’t detect anything and the signal goes low. This creates a signal that can be viewed as a PWM signal. By measuring the period of the signal, the frequency can be determined by using the following equation

\[ \text{Frequency} = \frac{1}{\text{period}} \]  

Since the TicksPerRevolution is known, the RPM can be calculated by using

\[ \text{RPM} = \frac{60}{\text{TicksPerRevolution}} \]  

this gives the measured motor speed. But the information can only tell what speed the motor has, not the direction it’s spinning, or the position/angle it’s in. The old encoder had two ticks per revolution. This was a point that could easily be parametrised between different heater models.
Encoders work by having ticks mounted on the motor axis. The resolution of the encoder depends on how many ticks it has mounted on it. There are versions of encoders that have better/worse resolution and/or can determine the motor's angular position and/or direction. This type of encoder has more than one measurement channel.

### 3.1.6 Fan

The fan motor is a 24 volt brushless dc motor. The ECU supplies the motor with 24 volt and sends a separate PWM signal to control the motor. The motor has its own driver that converts the duty cycle of the PWM signal to a corresponding speed.

### 3.1.7 Fuel-pump

The fuel pump is controlled by a high/low side PWM signal. The PWM is both the control and voltage supply to the motor. It's on/off regulated by the PWM signal. Both duty cycle and frequency affect the fuel pressure.

### 3.1.8 Lambda heater

Heats the lambda sensor by receiving a PWM signal from the ECU. The duty cycle determines the average current that passes through the heating-element inside the lambda sensor. The lambda sensor needs to be at a certain temperature before it can read any oxygen content.

### 3.1.9 Other actuators

The rest of the actuators, which can be seen in figure 6, are all on/off controlled by the ECU. The ECU either supplies zero or 24 volts. Two of them are actuators that heats certain components. One is the glow-plug that ignites the fuel. Another is the coolant-pump, it runs only at full power. The last is a fuel valve that opens and closes. These actuators are all on/off controlled by the ECU's supplied voltage.

- **Nozzle heater:** Heats nozzle area when supplied by voltage.
- **Glow-plug:** Heats quickly to a high temperature to ignite the fuel mixture.
- **Fuel filter heater:** Internally controlled as long as it's supplied by 24 volt.
- **Coolant pump:** When supplied by voltage, runs on full power.
- **Fuel valve:** When supplied by voltage, opens the valve and lets fuel pass through the nozzle.
3.2 Heater model

The main simulink model and all the sub model is described in more detail in appendix A.

The HIL-systems heater model of the heater was created in Simulink and then generated into c-code. The model structure was first developed by studying the heater, then it went over a number of iterations of improvement. The model was developed with test inputs signal to allow testing during the development. The HIL-rig receives the ECU:s output signals that should be controlling the actuators, and sends it in to the plant model. Therefor the inputs into the plant model are the outputs of the ECU. The outputs from the plant model are converted by the HIL-rig and sent to the ECU input ports as sensor signals.

The plant model was developed by looking at the heater system, its measurement data and using the information that was learned during the pre-study, as can be seen in chapter 2.

For several components there was limited technical data available such as datasheets. To overcome this limitation, some of the functions were developed from measurement data.

3.2.1 Model development

The goal was to develop a modular model, that had interchangeable parts. The first step was to define the interfaces and structure of the model. The interfaces externally and internally will have a major effect on the possibility of the modularity. A main model containing sub-models was the target, this would be a small modular hierarchical structure. The main model would have the interface outwards to the TivaC, and inwards to the sub models. Then one or more sub-model could change without problems with the rest of the system. This would be true if the main model would be swapped, but the sub-models would be the same.

To identify the interfaces needed, a first draft of the model was developed by identifying the inputs and outputs of the heater. This is both the flowing of medium/energy (air, fuel, heat etc...) and electrical signals between the heater and the ECU. The signals are the information and power that is sent between the ECU and the heaters actuators/sensors.

The different medium flows in the heater was taken into account. How air and fuel flows combine into exhaust in the fuel combustion, and how the coolant medium flows. This gave a good overview for how the plant model should work. This can be seen in figure 10 where the black boxes in the figure shows the major functions that was identified. The blue boxes is the output data, and was designed in the first draft to be both a function and a output. The red arrows is the input data, orange arrows is the medium flow and the black arrows is the internal data between the functions. The functions that was identified, corresponds to actuators or parts in
the heater where different medium flows, combines and/or change energy state.

During development, it was realised that not all actuator signals of the ECU was needed. One of the limitation of the project was "No automatic control of the actuator currents" and the fuel filter component has its own logic for when the filter heater is turned on/off. The lambda sensor needs heating to work, but the logic of lambda calculation did not need it. The task of lambda heating control was given to the TivaC main code. The first version of the Plant model did not take the pre-heating of the fuel or the heating of the lambda sensor into account. The other actuator signal was decided to be necessary for the plant model. It was determined that all the sensor signals was needed to be calculated and sent.

Of the major function in the heater model, only two of them receives PWM signals. The signal to the fan is a control signal only. The signal to the real fuel pump, is a high current PWM signal that both controls and delivers the power to it. It was decided to split this signal into two inputs, one duty cycle and one on/off signal that represents the power supply. The rest of the actuators are controlled by on/off signals.

The sensors listen to eight signals. Five temperatures, one pressure, one RPM
and one lambda signal. The outputs from the plant model are digital values.

The plant model has seven inputs and eight outputs, as seen in table 2. The inputs and outputs are the external interface of the plant model. While the internal structure was created by observing the heaters function.

Table 2: Heater models input and output signals

<table>
<thead>
<tr>
<th>I/O</th>
<th>Function name</th>
<th>Signal type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>Fuel_Valve_IN</td>
<td>on/off</td>
</tr>
<tr>
<td></td>
<td>Nozzle_Heater_IN</td>
<td>on/off</td>
</tr>
<tr>
<td></td>
<td>Fuel_Pump_PWM</td>
<td>duty cycle</td>
</tr>
<tr>
<td></td>
<td>Fuel_Pump_IN</td>
<td>on/off</td>
</tr>
<tr>
<td></td>
<td>Glow_Plug_IN</td>
<td>on/off</td>
</tr>
<tr>
<td></td>
<td>Coolant_Pump_IN</td>
<td>on/off</td>
</tr>
<tr>
<td></td>
<td>Air_Fan_PWM</td>
<td>duty cycle</td>
</tr>
<tr>
<td>Outputs</td>
<td>RPM_OUT</td>
<td>Digital</td>
</tr>
<tr>
<td></td>
<td>NTC1_OUT</td>
<td>Digital</td>
</tr>
<tr>
<td></td>
<td>NTC2_OUT</td>
<td>Digital</td>
</tr>
<tr>
<td></td>
<td>NTC3_OUT</td>
<td>Digital</td>
</tr>
<tr>
<td></td>
<td>NTC4_OUT</td>
<td>Digital</td>
</tr>
<tr>
<td></td>
<td>Pressure_OUT</td>
<td>Digital</td>
</tr>
<tr>
<td></td>
<td>Lambda_OUT</td>
<td>Digital</td>
</tr>
<tr>
<td></td>
<td>NT_OUT</td>
<td>Digital</td>
</tr>
</tbody>
</table>

The plant model design was redone, as can be seen in figure 11. The medium flow was removed, since it does not have a corresponding function in the Simulink model. But the underlying thought behind it and the design is still there. The blue boxes only represent an output signal of the plant mode, the black boxes is now the only functions in the plant model. This meant moving the lambda calculation into the fuel combustion. The filter heater function was removed as mention above. The signal to and from the NTC4 sensor was added, it represent the air temperature. It was decided that it would be a constant, that can in future development get its own function where it can change temperature.
3.2.2 Simulink and code generation

The model was developed in Simulink in iterations. The model was created with code generation in mind.

During the beginning of the development, normal "constant" and "Math Operations" blocks was used to combine and calculate values. The generated code didn’t generated them as variables, but as number instead. This made it hard to read and understand the code without observing the original model. Since the future development of the plant model would only be in C-code, the need of a good and understandable code was vital. A solution to this was to use "Data Store Memory"[23]. An example of the generated code can be seen in listings 1 and listings 2 in appendix B. It’s part of the code that’s been generated from the simulink model that can be seen in figure 39. The generated variables comes from the "Data Store Memory" and "Discrete-Time Integrator" blocks. The variables are globally declared and stored in the random access memory(RAM). This type of code that’s been generated has a
risk of memory overflow. The TivaC Microcontroller[24] has 256 kb SRAM memory. For this project, it was an acceptable risk since the microcontroller has a large SRAM memory.

In the Simulink models in this project, the "data store blocks" have declaration blocks(colored orange) and read blocks(colored yellow). The "data store blocks" also have a write block function, but that was not used in the plant model. The input to the module are colored red and the outputs are colored teal. This coloration was to help the reader identify the different blocks. Each sub-model gets it own variable, but there was no "global variable" for the whole plant model that was generated. That was something that needs to be written by hand in the future. The code generation was using the "Simulink Embedded Coder" system target file "ert.tlc", with C as the target language. The tool-chain was "MINGW64 v4.x | gmake (64-bit Windows)" and the build configuration was "Faster Builds". The code generations prioritized objectives are first Execution efficiency and then RAM efficiency. The device type was set to ARM Cortex to get the right type/number of bits for the processor.

The code generation creates the main file and its corresponding header. The main file contains a step function, and when that function was called one "step" of the model was run. One step simulates one sampling time of the model. The simulink model was created with 100Hz in sampling time, this comes from the limitation of the project, see chapter 1.2. This means that the step needs to be called 100 times per second. The main file also contains a initialize function that initialize the whole model.

3.2.3 Main Model

The general structure of the main simulink model was developed with each of the sub-modules as reference models. This means that the sub-models was their own simulink models that was referenced in the main model. The reference models can be worked on separately, as long as they have the same input/outputs as defined. This makes the simulink model a partly modular model, where a sub-module can be improved or changed without it causes problem for the whole plant model.

3.2.4 Air Fan Model

The air fan model that can be seen in 11 was divided into two sub models. The motor model calculates the RPM and the fan model translates the RPM to the mass air flow leaving the fan. This was done since the motor calculation might be needed to change.

Motor Model:
The motor is a brushless DC motor and a simulation of such a motor can be numerically heavy. Since it was unknown how much calculation could be done on the TivaC, it was decided to develop a model of a normal DC motor.
The motor model was validated by checking the output RPM against measured data from a real motor test. The motor model was given a duty cycle input that was designed to be similar to the duty cycle input of the test log. As can be seen in figure 12, the left graph is the duty cycle input of the test, and the right is the input signal of the model. The right one goes in scale of 0-0.55 but the left is in 0-55. It’s the same duty cycle, but the test logs saves the value by scaling it by 100. The ramp up time is around 7 seconds, it continues to give the maximum value 0.55 afterwards for the rest of the simulation.

![Duty cycle to ramp validation](image)

**Figure 12:** Duty cycle to ramp validation

This input gives the output RPM as can be seen in figure 13. The left graph is the RPM from the test log and the right graph is the RPM that the model simulates. Both graph shows the duration about 20 seconds, and the RPM is settling on similar values given the similar input. The spike in the beginning of the left graph is considered to be because of the big step in the first input value.

![Motor validation of RPM](image)

**Figure 13:** Motor validation of RPM

**Fan Model:**
The fan model converts the fan motor RPM to the air mass flow.
3.2.5 Coolant Pump Model

The motor in the coolant pump is controlled by on/off signal. When the pump is on it deliver 52 liters per minute in flow.

3.2.6 Fuel Pump Model

The fuel pump model has two inputs, but this version was only using one. The company had no documentation about the pump or it’s characteristics and there where no spare pump do take apart and do measurement on. The model where therefore developed using existing measurements of the pump and the heater. The pump PWM signals from the ECU was translated into a target pressure using a polynomial equation that was fitted from measurement data.

The equation was developed by looking at the fuel pressure in the test data, and dividing it with the fuel pumps duty cycle. The test data did not contain many different data sets since only a few pressure set points was used. It’s worth noting that in the data set the fuel pressure is saved in mBar and the duty cycle is in 0.1%. This gain data that can be seen in figure 14 was then used to create a function that can translate every duty cycle to a corresponding gain value. This value is then later used to calculate the fuel pressure. With a duty cycle bigger than 0.3 the fuel pressure reaches the maximum value of 12 bar, the function does not take this into account. The model needs to do it instead.

![Figure 14: Fuel Pump Gain Validation](image)
3.2.7 Nozzle Model

The nozzle model converts the fuel pressure input into a fuel flow output. The Fuel valve controls if there is a fuel flow exiting the model. The nozzle heater inputs is used to calculate the temperature of the nozzle. The temperature is then outputted from the model.

The equation to calculate the fuel pressure to the fuel flow derived from measurements and can be seen in figure 15. The input pressure should only be between 0-12 bar. The model must handle that case when the input is outside that range.

![Figure 15: Nozzle flow validation](image)

3.2.8 Fuel Combustion Model

The fuel combustion model calculates the combustion of the heater. The equation (2.1) shows the Air-Fuel Ratio(ARF) but it can also be written as

$$AFR = \frac{m_{air}}{m_{fuel}} = \frac{\dot{m}_{air}}{\dot{m}_{fuel}}$$  \hspace{1cm} (3.3)

where instead of masses, there was mass flow. It’s the ratio that important. As written in the equation (2.2) lambda is the difference between the AFR and the sto-
ichiometric ratio($AFR_{stoch}$) that in the model was named Air_Fuel_Ratio. Therefore

$$\lambda = \frac{\text{AirFlow}}{\text{FuelFlow}}$$

(3.4)

was used to calculate the lambda value in the model. There is a control logic that outputs a boolean “Burning ON/OFF” to control combustion in the model. If it was no combustion, the maximum lambda value was returned. This logic can be seen in figure 16.

![Lambda output logic](image)

**Figure 16: Lambda output logic**

The fuel flow was also used to calculate the energy that was released in the combustion. The Fuel_Energy constant depends on what fuel was used, it was the lower heating value(LHV) of the fuel. LHV is the amount of energy in the fuel after all water in it has been vaporized. The fuel flow and the constant determines the maximum possible energy generated during the combustion. If lambda was lower than one, the energy generated was lower than the possible maximum since not all of the fuel has been combusted. This logic can be seen in figure 17.

![Fuel Energy output logic](image)

**Figure 17: Fuel Energy output logic**

The generated energy from the combustion was then sent out from the model.
3.2.9 Heat Exchange and Cooling Model

The Heat Exchange and Cooling Model takes the combustion energy and coolant flow and calculates three separate temperatures of the coolant water that passes by certain points of the heat exchanger. The equation derived from the prestudy

\[
T_{\text{out}} = \frac{\eta Q_{\text{ch}}}{\dot{v}\rho(T_{\text{in}})c_p(T_{\text{in}})} + T_{\text{in}}
\] (3.5)

was used in the model. The models variables can be seen in table 3.

Table 3: Variables in the heat exchanger model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\eta)</td>
<td>Efficiency</td>
</tr>
<tr>
<td>(Q_{\text{ch}})</td>
<td>Chemical Heat Flow</td>
</tr>
<tr>
<td>(\dot{v})</td>
<td>Volume Flow</td>
</tr>
<tr>
<td>(\rho)</td>
<td>Density</td>
</tr>
<tr>
<td>(c_p)</td>
<td>Specific Heat Capacity</td>
</tr>
</tbody>
</table>

The model was then verified against the real system as can be seen in figure 18.

![Verification of model to real system](image_url)

**Figure 18:** Verification of the model against the real system
3.3 HIL environment

To run the HIL-system a microcontroller was needed. The TivaC Series TM4C1294 Evaluation Kit was chosen since it’s fast and capable microcontroller with a low price. It has several Serial Peripheral Interface(SPI) and Inter-Integrated Circuit(I2C) channels, several PWM channels, lots of General-purpose input/output(GPIO) pins and has a development environment available. It can also be upgraded different modules such as increased RAM. The integrated development environment(IDE) used was Texas Instruments Code Composer Studio[25]. It’s free software that has compatibility with the evaluation kit and uses an Eclipse software framework.

3.3.1 Microcontroller

- Evaluation Kit: EK-TM4C1294XL [26]
- Microcontroller: TM4C1294NCPDT [24]
- Processor: 32-bit ARM® Cortex™-M4F, 120 MHz

3.3.2 I/O ports

The TivaC was mounted on an evaluation board, it was removable and only connected by PCB connectors. The TivaC was mounted on the same structure as the sensor modules. The connections from the sensor and actuators to the TivaC can be seen in figure 19. The numbers in light blue in the figure represents the pins on the TivaC. The letters and numbers below the dark green fields was the pin port and number of the microcontroller "tm4c1294ncpdt". The fields below the pink fields are the functions of the pins and the texts below the dark-yellow fields was the sensor emulator or actuator unit the TivaC was sending data to or receiving data from.

The TivaC uses SPI communication to control parts of the sensor emulators. SPI communication uses 4 types of pins. Clock timing (CLK), SlaveSelect (SS), Master Output Slave Input (MOSI) and Master Input Slave Output(MISO). The TivaC only uses three types of signals (CLK, SS and MOSI) since it only talks to and not listen to the components.

The "PWM measurement" takes two inputs of the same PWM signal. The two inputs are interrupts that trigger on the incoming signal, but on different edges. The interrupts calls functions that uses internal clocks to calculate the duty cycle, it only calculates the duty cycle, not the frequency. The pins that are marked "I/O signal" are inputs that checks if the port was high or low. The "PWM Generation" pin generates a PWM signal with the same duty cycle but with different frequencies.
3.3.3 Main TivaC program

The main program uses a timer interrupt that was set to run at 100 Hz. It was called 100 times a second and it runs a set of steps that reads all input, calls the plant model code, and sends the signals to the ECU. The general logic of this function can be seen in figure 20, a simple flow chart that describes the main function that was called every 0.01 seconds.
The first function reads the ECU:s output signals. It reads only the I/O signals since the PWM measurement occurs continuously in the background. The second functions saves the data to the variables that the plant model can use. To have these variable separated was a good way to keep it modularized, it was designed this way to make it easy to develop separately. It also provides the possibility to implement error injection in the "actuator signals" without having to change the plant model. The third function calls the plant model step function once, and all the calculation takes place. The fourth function gets the calculated data. The fifth function start the process of converting and then sending out the calculated data to the sensor emulator modules. Then the function checks if it should send the data over serial. This occurs two times per second because it’s not needed very often. When the frequency of the data being sent was increased, the computer could not write out the data fast enough on the computer screen. This is another reason to why it was sent two times per second. This corresponds to every 50th iteration if the function gets called every 0.01 seconds. If the answer was yes, it sends the data over serial as a string format and then goes to the end of the function. If the answer was no it goes straight to the end. When the function was finished, it waits to be called again the next time the interrupt trigger.

3.4 Sensor Emulation

This section describes how the sensor emulation was done. The goal of the emulation was to translate the calculated sensor values from the plant model into the correct type of signals that the ECU usually receives from the real-world heater. The
emulation of sensors consist of both software and hardware. The software part was responsible for translating the values calculated in the plant model to the correct output from the TivaC to the sensor modules. The output of the signals are in the format of SPI communication and I/O signals for controlling relays. The translation functions are created from measurements during development. The following module sections describes the hardware part of the sensor emulation.

3.4.1 Lambda module

The prototype HIL-rig needs to be able to simulate lambda from 0.65 to 5 and send the equivalent electric signals to the lambda sensor control chip in the ECU. The LSU 4.9 sensor does not give the value in one signal, but uses four signals. The sensor works together with a control unit that interprets the analog values of the four signals to a corresponding lambda value. The CJ125 control chip is mounted in the ECU, so the HIL-rig needs to communicate to it first, and through it simulate lambda.

One possible solution would be to have the real LSU 4.9 sensor inside a small test chamber where the oxygen concentration of an exhaust gas could be precisely controlled. This was not possible, as this thesis did not have the resources or expertise to build a test chamber like that.

The other solution was to replace the LSU 4.9 sensor with a specific designed hardware that can simulate the four different signal voltage. This hardware also need to be able to fool the diagnostics inside the CJ125 control chip.

The lambda sensor has six wires. Two wires are for the heater, and four are for the measurement of the lambda value. The circuit that simulate the lambda uses only four wires, they are stated below with the different name and corresponding color which can be seen in figure 21.

- RE/UN (Black): Nerst voltage
- VM/IPN (Yellow): Virtual ground
- IP/APE (Red): Pump current
- IA/RT (Green): Trim resistor
When the sensor gets hot enough the gas helps move the current through the Nernst cells. The control unit CJ125 uses this property to check if the sensor is hot enough, and use that data to control the heater. This was done by checking the impedance between RE(black) and VM(Yellow) cables. When it was in the right value range the CJ125 tells the rest of the ECU that it’s ok, else it sends an error message. By using a resistor of 300 ohms, the CJ125 was fooled. No diodes that control the direction of the current can be used between the RE and VM.

The simulation can be split in to two cases, $\lambda \geq 1$ and $\lambda < 1$.

When lambda goes below 1, the current change direction in all the cables. Instead of exiting in the VM cable, the current enters from it and exits in the rest. When lambda is lower than one, not all fuel has been combusted and there is no oxygen in the exhaust. The pump cell starts to transport oxygen ions the other way. This introduce a voltage direction change. To simulate this a voltage of 1-1.5 V must be added between RE and VM with the positive terminal to the RE cable. This was done by having a KA317 linear voltage regulator, that was using its lowest available increase of 1.25V. No diode can be used to control/limit the current direction since it would interfere with the impedance. Two relays was used to control where the current goes the right direction and don’t short-circuit the component when
the lambda goes over/under one. When both relays were set so no connection was made between VM and RE, the CJ125 chip read it as the sensor was not hot enough.

In the sensor the pump cell controls the current and voltage. In the simulator circuit the current was controlled through a digital potentiometer, then the resistance was changed, the lambda value changes. The digital potentiometer is 8 bit, 50K ohm. When the resistance was small the λ value was close to 1 and when the resistance was big the λ was high(5)/low(0.65) depending on what state the relays are in.

3.4.2 Thermistor module

The NTC module are a simple digitally controlled potentiometer that can have resistance between 0-50K ohms. Input are +5 volts. The higher the resolution of the potentiometer, the more accurate value the HIL-system can simulate.

The potentiometer that was in use is a SPI controlled linear quad channel potentiometer with a resolution of $2^8$ per channel. This was quite low, and gives big steps at higher temperatures. A low resistance corresponds to higher temperature measurements for the ECU. To fix this, a higher resolution digital potentiometer was needed, but that can be expensive. A logarithmic potentiometer (also called audio potentiometer) can be a better choice, since it has better resolution for lower resistance values.

A solution to the bad resolution was to use a relay to parallel connect a normal resistor with the value of 2K ohm to lower the overall resistance. This gives the option to have a little better resolution when needed. This gave a better smaller steps at higher temperature, but it was still bad. Since this was a prototype, it was seen as acceptable in the first iteration.

3.4.3 Nozzle thermocouple module

The ECU measures the difference in mV between two channels of the J-type thermocouple. To simulate this, two Digital to Analog Conversion (DAC) components with the resolution of $2^{16}$ was used. A digital value was transformed into a voltage signal. One step in digital value was a change in mV as an analog signal. By setting one DAC as a reference point, and changing the other, it was easy to control the temp sensor. The Circuit was built so both DAC can be changed if needed. The DAC:s are controlled by SPI.

3.4.4 Fuel pressure module

The sensor uses two channels to communicate data. Two analog data and one ground. The first data channel has output 5V, the other data channel is input. The receiving voltage on the second channel goes from around 5 to around 4.5 volt for pressure between 0-12 bar. This was simulated by a circuit as seen in figure 22.
The circuit consist of two resistor in series, and a measurement point between the resistors.

![Circuit Diagram]

**Figure 22:** Pressure Circuit Layout

The first resistor is a digital potentiometer called R1 that can go between 0-50K ohm and a resistor R2 of 5K ohm.

### 3.4.5 Sensor module development

The sensor emulators was developed on RE220-HP prototype boards. The evaluation boards was made to be stack-able and easily removed to fit the modularity approach. This can be seen in figure 23 and 24.
The evaluation board gave the ability to select and group the cables and easily connect them to the right input port on the TivaC. The sensor modules can be easily replaced and the cables are easily removable by a single pull. The information about what cables corresponds to what signal was marked on each sensor board, this was done to make future development easier.
3.5 Cooling of the power sinks

The real system has a lot of high power components such as fuel pump, coolant pump, nozzle heater, fuel filter heater and so on. If these aren’t connected or responding the ECU will give an error code. Thus the HIL-rig needs to be able to consume a lot of power to trick the ECU into thinking that it’s connected to a real system and not have faulty components or other errors. The best and easiest way to do this is to have power resistors that swallow the load. Because of the power dissipation the resistors needs to be cooled somehow. To cool them they need to be placed on a heat sink. There are two main types of heat sinks which are air cooled heat sinks or liquid cooled heat sinks. The difference between these are which medium that transfers the heat away from the heat sink. The air cooled heat sinks can be left as they are or be placed within a forced air flow, usually made by fans. The forced air flow variance is better at transferring away the heat but the fans can be very loud which isn’t ideal since the rig is supposed to be placed in an office environment. The liquid cooling system will however require tight seals so that they don’t leak any liquid. Which is especially important when electricity is involved. It will also require a pump and preferably some sort of water tank. The pump can generate some noise but it’s usually more silent than several fans.
Since the total power consumption is about 1kW the fans and heat sink for air cooling would have to be quite large, therefore expensive, and loud. The decision was made to go with a liquid cooling system, and proper heat sinks was purchased.

For the rig to draw the required power the resistors will be connected in parallel and series as follows to spread out the power dissipation to lessen the strain on the individual resistors. This can be seen in figure 25.

\[
R_{tot} = \left( \frac{1}{R_1 + R_2} + \frac{1}{R_1 + R_2} \right)^{-1}
\]

and the power on the resistor pair will then be

\[
\frac{U}{R_{tot}^2} = 36.9W
\]

The power on the different setups will then be as seen in table 4

According to [27] the internal temperature of the resistors, \(T_j\), the formula

\[
T_j = P_d(\Theta_{jc} + \Theta_{cs} + \Theta_{sa}) + T_a
\]

is used. Where \(\Theta_{jc}\) is the thermal resistance of the junction (resistors), \(\Theta_{cs}\) is the thermal resistance of the insulator (thermal paste), \(\Theta_{sa}\) is the thermal resistance of the heat sink, \(P_d\) is the power dissipation and \(T_a\) is the ambient temperature. The resistors is mounted on the heat sink as shown in figure 26. The values of the thermal resistances are found in the components datasheet/website [28] [29] [30] and
Table 4: Load on the resistors

<table>
<thead>
<tr>
<th>Number of resistor pairs</th>
<th>Total resistance [(\Omega)]</th>
<th>Total Power [W]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.6</td>
<td>36.9</td>
</tr>
<tr>
<td>2</td>
<td>7.8</td>
<td>73.8</td>
</tr>
<tr>
<td>3</td>
<td>5.2</td>
<td>110.7</td>
</tr>
<tr>
<td>4</td>
<td>3.9</td>
<td>147.7</td>
</tr>
<tr>
<td>5</td>
<td>3.12</td>
<td>184.6</td>
</tr>
<tr>
<td>6</td>
<td>2.6</td>
<td>221.5</td>
</tr>
<tr>
<td>7</td>
<td>2.23</td>
<td>258.5</td>
</tr>
</tbody>
</table>

can be seen in table 5. The power of the two series resistors will always be the same as long as the voltage is the same. If there needs to be a larger load more resistors can be added in parallel.

The power in a resistor is max 25.6 W. The calculation of the temperature on the resistor will then be

\[
T_j = P_d(\Theta_{jc} + \Theta_{cs} + \Theta_{sa}) + T_a
\]  
(3.9)
This gives a final temperature in the resistor of about 78 degrees Celsius. Which is an acceptable temperature. Holes are drilled through the heat sink and the resistors are mounted with screws on it with thermal paste in between to make the heat transference better. The cables in between the resistors are fitted with flat pin connectors so that they can easily be changed or rerouted if needed. The setup of the resistors and the wiring of the cables makes the setup modular. It can be seen as a bus-type modularity as described in chapter 2.2. This will be beneficial if the company decides to change a part to another one with a different load.

Hoses are then connected so that the heat sinks are connected in series. These are connected to a pump and water tank via quick release connectors. This will make it so that if the company wants to use their larger cooling system, i.e. if they want larger loads or run it for a very long period, they can do so.

3.6 Complete HIL-system
The HIL-system can be divided into three parts. The microcontroller TivaC, the sensor emulators and the power sinks. The HIL-system functionality can be seen in figure 27. It shows how the signals are connected between the different parts. The figure includes the ECU to show how it interacts with the HIL-system.
The biggest physical part of the HIL-system is the power load and cooling systems. It’s large because it has to be able to handle a big amount of current and heat. The TivaC and sensor emulators are compact, mounted together and can be easily handled. The cables between the systems takes up space, they could be shorter and better constructed if there was a permanent place for the HIL-rig. Currently the HIL-system is mounted on a carriage to be easily moved as can be seen in figure 28. On the top shelf the ECU, "Module emulator stack", coolant tank and coolant pump was placed. On the bottom shelf the power sinks and power supply.

Figure 27: HIL-system functionality
Figure 28: The HIL rig

The coolant liquid is pumped from the tank down to into the power sinks, circulated through them and then pumped back up into the coolant tank. This can be seen in figure 29. The pump was supplied from the same power source as the HIL-rig and was therefore always on when the HIL-rig was on.
The power sinks were mounted on a wooden rack. It’s used to separate the sinks from each other and hold them in place as can be seen in figure 30. The heat sinks are easily removable, and can be accessed without problem. The resistors on the heat sinks are mounted in groups and clearly marked with which "actuator" they are connected to.
Figure 30: Power sinks
4 Analysis/Result

This section handles the analysis and result of the thesis.

4.1 How the test was made

The company uses a LabView GUI when they do their testing. The GUI communicates with the ECU via CAN. The real world test was done using one heater, test rig and LabView to log the data. It was done under normal lab conditions, at around five degrees ambient temperature. The test with the HIL-system was done in a similar fashion. Using the same or close to the same parameters such as ambient temperature. The same ECU was used with the same code on it. The LabView GUI was used to log data this time as well. The data from the actual simulation was logged by having the TivaC send the data in serial via USB. This data was logged twice every second. The reason why the data was logged with a fairly low frequency was because it was unknown if the TivaC could handle to send data more frequently. It might have caused some disturbances with the actual simulation, and this data was supposed to be used as more of a comparison.

The test sequence went as follows: Plug everything in, start logging, send a start signal to the ECU, let the ECU initialize and start up, ECU checks for any open load errors or other similar errors, start the ignition and the runs until it gets any errors or is sent a shut off signal. The ECU was shut off after about 400s during the real world test and at about 600s in the HIL-test.

4.2 Results

The temperature from the real system can be seen in figure 31 which can be compared to the temperature of the model logged directly from the TivaC to a computer in figure 32 and also the temperature logged from the ECU via LabView when the ECU was connected with the HIL-system in figure 33. The reason why the temperature curve isn’t very smooth in the modeled data was because the data was sent using integers. The data was sent over serial communication from the TivaC to a computer, and the serial can only send whole integers. This was a mistake and should have been done by multiplying the values by 100 before sending them and then divide by 100 after. This won’t change the general shape of the curve though, it will only improve the resolution. The reason why the temperature from the HIL-system looks like it does was because the real NTC sensor is nonlinear but the digital potentiometer used is linear. Even if the non-linearity was modeled, the big problem would be the resolution of the potentiometer. This causes the emulated sensor to work fairly well during low temperatures but when in high temperatures one step on the digital potentiometer represents a much larger step in temperature. This can be fixed by using a potentiometer with a much higher resolution or using a nonlinear potentiometer such as an audio potentiometer or logarithmic potentiometer. This behaviour was noticed too late and there wasn’t enough time to develop a new sensor emulator.
Figure 31: Temperatures in the real system in a normal test
Figure 32: Temperatures in the model during a similar test as in the real world
If the fan speeds in figure 34 are compared, the "model data" and HIL fan speed are very similar. A slight difference that’s likely due to some errors when emulating the encoder signal but it’s nothing significant. The fan speed from the "real world" test shows some differences between it and the model/HIL data. However the data in figure 35 shows the fan speeds in comparison to the PWM inputs, and they are very similar to each other. Therefore it’s likely that the cause of the difference lies elsewhere. It should also be noted that the PWM that LabView logged and the actual PWM sent by the ECU was not the same. This signal was measured by an external oscilloscope and compared to the LabView display. This could cause some of the behaviour seen in the graphs.
Figure 34: Fan speeds from the real system, HIL system and the model compared to each other
As can be seen in figure 36 the fuel pump’s PWM also differs some from the real system. This is likely because of the same reason as the behavior of the fan. The signal was however much noisier which can be caused by the way the signal was read by the TivaC.
Figure 36: Pump PWMs in the real system, HIL system and the model compared to each other

4.3 Budget/Cost

The total cost of all the purchased components can be seen in table 6. These are not the only components that were used in the rig. Several components like cords, pump and connectors were materials that the company already had and were not purchased. The cost of these are roughly estimated to about 1000 SEK. The total cost of all the material therefore lands at about 9000 SEK. If the total number of hours worked was taken into consideration the cost would increase a lot.

The cost for a commercial system would however be a lot higher than this. Even if there would be less work into setting up the commercial system it would still require a lot of time. So the price difference is quite substantial.
Table 6: Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Amount</th>
<th>Price Per</th>
<th>Total Price [SEK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resistor 50 W 10 ohm</td>
<td>27</td>
<td>19.49</td>
<td>526.23</td>
</tr>
<tr>
<td>Resistor 50 W 5.6 ohm</td>
<td>27</td>
<td>21.12</td>
<td>570.24</td>
</tr>
<tr>
<td>Board</td>
<td>1</td>
<td>195.94</td>
<td>195.94</td>
</tr>
<tr>
<td>Heat-sink Water</td>
<td>6</td>
<td>596</td>
<td>3576</td>
</tr>
<tr>
<td>Solid-state Relay</td>
<td>7</td>
<td>206.78</td>
<td>1447.46</td>
</tr>
<tr>
<td>Thermal Paste</td>
<td>2</td>
<td>165</td>
<td>330</td>
</tr>
<tr>
<td>Low Signal Relays</td>
<td>4</td>
<td>17.51</td>
<td>70.04</td>
</tr>
<tr>
<td>Bipolar Transistors NPN</td>
<td>4</td>
<td>3.14</td>
<td>12.56</td>
</tr>
<tr>
<td>Linear Voltage Regulators</td>
<td>4</td>
<td>7.93</td>
<td>31.72</td>
</tr>
<tr>
<td>1N4007 Diode</td>
<td>10</td>
<td>1.4</td>
<td>14</td>
</tr>
<tr>
<td>DAC 16-bit</td>
<td>2</td>
<td>115</td>
<td>230</td>
</tr>
<tr>
<td>DAC</td>
<td>2</td>
<td>44.11</td>
<td>88.22</td>
</tr>
<tr>
<td>Digital Potentiometer ICs Quad</td>
<td>2</td>
<td>72.67</td>
<td>145.34</td>
</tr>
<tr>
<td>Prototypeboard</td>
<td>4</td>
<td>43.9</td>
<td>175.6</td>
</tr>
<tr>
<td>Socket header 2x36</td>
<td>2</td>
<td>49.7</td>
<td>99.4</td>
</tr>
<tr>
<td>Pin header 2x36</td>
<td>2</td>
<td>22.9</td>
<td>45.8</td>
</tr>
<tr>
<td>Hose</td>
<td>1</td>
<td>54.9</td>
<td>54.9</td>
</tr>
<tr>
<td>Faston</td>
<td>10</td>
<td>29.9</td>
<td>299</td>
</tr>
<tr>
<td>Hose Nipple</td>
<td>2</td>
<td>24.9</td>
<td>49.8</td>
</tr>
<tr>
<td>Hose Adapter</td>
<td>2</td>
<td>32.9</td>
<td>65.8</td>
</tr>
<tr>
<td>Other Material</td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Total cost:</td>
<td></td>
<td></td>
<td>9028.05</td>
</tr>
</tbody>
</table>
5 Conclusion

Even if the HIL-system has performance issues with bad resolution, the overall function of the HIL-system is still good. The cost of the HIL-system is a fraction of a commercial system and can be used for testing none vital functions. The company has used the system to run several tests, where a good resolution is not vital to the test. An example of this is verifying if the ECU outputs power to the right components at the correct time. Since the company is quite small they get more value for their money with the custom built prototype system.

The resolution of the HIL-rig depends on the resolution of the digital potentiometers. This gives a good ability to improve the results by using higher resolution digital potentiometers.

The prototype HIL-system can be run a full test without the ECU throwing any errors. The HIL-system fulfills all the goals/limitations of the project, although the accuracy could be improved.
6 Discussion/Recommendation

How to create a modular model of a heater? To create a modular model of a heater, the general structure of the heater must be viewed and analysed. What are its physical, functional properties. How does the data signals, physical media and energy flow through it? Break down it’s structure, and divide it into models. Define the interfaces in and between the models. See if the parts/sub-models can be re-used in different models/areas.

How can the model be parameterized in order to support different heater models? The goal is to identify the differences. If the heater has different motors, can different motor parameters be used to calculate the correct RPM? And by RPM calculate the air flow. Does the heater have different type of nozzle for the fuel? Can you calculate different fuel flow into the combustion area? Are different fuels used? if that is the case, what are the "Air-Fuel Ratio" stoichiometric value? and what are the fuels heating values, especially the "Lower Heating Value"?

The most important part are:

- Air flow.
- Fuel flow
- The fuels AFR-Stoich value
- The fuels LHV

Other parameter

- Encoder Ticks
- Coolant flow
- Motor parameters

Can the model be created to have interchangeable/addable/removable subsystems? Yes, the model can be created to have interchangeable parts as long as you use the same interfaces.

What are the limitations of a cheap rig?

One limitation is the resolution of the data, cheap digital potentiometers have been used, that have a resolution of 8-bit. If higher resolution digital potentiometer is used, the resolution of the HIL-rigs data will improve.

The rig is vulnerable to interference, that can corrupt measurements. One example is that the PWM measurement is at risk of triggering interrupts on the wrong rising/falling flank because it can detect and interpret interference noise/spikes as flanks. A higher budget gives more resources to combat this problem. The interference might for example come from the cheap power supply or coolant motor.
Another limitation is that the rig might not be suitable for long term tests. The testing that was done was done in one full day and then the rig was turned off and cooled down during the night. If one want to run tests that are over the span of a few days the rig will most likely overheat.

**How will the parameterized models in the prototype HIL-system compare to a test of the existing system?**
The result has shown that the resolution on some of the data is very bad, but that comes from the digital potentiometers not the parameterized models. The models give a reasonable resemblance to the real world test, and can trick the ECU into thinking it’s in the real heater.

**Can the model be used to simulate a behaviour that is not possible with the existing system? If yes, how?**
At the moment, no. But it’s the impression of the authors that with fault injection it will be be possible, this is mention in the future work.

### 6.1 General Discussion

The HIL-system is only one small prototype, and will only be used in one location to replace/append a testing case. Therefore the authors believe that the social and ethical implications of this work will be minimal. The prototype will only remove the fuel consumption of one test cell that it replaces. So the environmental impact will also be minimal.

The HIL-rig has in total cost the company about 10,000 SEK which is cheaper than the ones on the market. Even if it doesn’t perform as well as many of the other ones the HIL-rig can still be useful during several tests. Especially ones where the resolution isn’t all that important. The price compared to what it can actually do for the company is therefore quite low.

### 6.2 Future work

A fault injection function would make the HIL-rig more versatile and useful when testing different functions of the heater. This could be used to test that the behaviour of the heater is as expected when some error occurs. To implement fault injection a special LabView GUI, or some other form of user interface, made for the HIL-rig would be an requirement. From that interface the user would be able to add different error cases such as a too low pressure in the fuel line, too hot intake air and so on.

A specially made GUI would also improve the user experience of the HIL-rig. Not only for the fault injection but for other features as well. The GUI can be used for many different analytics tools. Implementing a GUI would require another form of communication with the TivaC other that using the serial bus, such as CAN.
One thing that would increase the performance a lot is the use of a higher resolution digital potentiometer. A better resolution in the same resistor range 0-50 000 ohms would give increased accuracy in the data. Therefore it would be better to test their control algorithm. Since the whole system is modular the change can be done swiftly without having to rebuild the whole system. The only thing needed to change would be that specific circuit as well as some parts in the code. The code changes is the conversion algorithm that the main code in the TivaC does. It’s a separated function in the C-code that can easily be replaced and/or improved. If the new component also uses SPI the change would be easier than if it uses some other communication.

The future model could be adapted with behaviours that are not available in the real life system. The user could for instance, speed up the initialization and pre-heating states if they are irrelevant for the current test. Change the ambient temperature during a test to see how the heater reacts. Implementing behaviours that doesn’t exist in the real system.

The sub-models could be improved in the future. A good thing about the modular approach is that the models can be improved individually, depending on the need. If a component change in the heater, for example the fan motor, it’s simple to create a new motor model and swap it with the current motor model. But since the modular design of the Air-Fan model, the Fan model can be kept the same, see figure 38 for the design structure.

The main model of the heater might be improved by adding a sub model of the fuel filter heater. The fuel filter heater should turn on/off depending on the ambient temperature, or the temperature of the incoming fuel from the fuel tank. The on/off signal should control if the heater draws current or not. This was removed earlier from the thesis, since the thesis limitation of "No automatic control of the actuator currents".
References


Appendix
A Main Model

The main simulink model can be seen in figure 37, the red part are the inputs, the teal are the outputs and the grey boxes are the sub-models.

Figure 37: Main Model Simulink
A.1 Air Fan Model

The air fan model interface was set to receive two inputs, and send two outputs. The inputs are "Fan PWM" and NTC4 and outputs "Air Mass Flow" and RPM. The air fan model was divided into two reference models itself. A motor part and a fan part, this can be seen in figure 38.

![Figure 38: Air Fan Model Simulink](image)
### A.1.1 Motor Model:

The electric characteristics of a DC-motor can be parametrised where $U$ is the voltage, $i$ is the current and $\rho$ is the angle of the outgoing shaft. The rest of the terminology are described in Table 7.

#### Table 7: Motor models variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Unit</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_m$</td>
<td>0.2</td>
<td>Nm/A</td>
<td>Motor torque</td>
</tr>
<tr>
<td>$R_{motor}$</td>
<td>0.4</td>
<td>Ohm</td>
<td>Motor Resistance</td>
</tr>
<tr>
<td>$K_e$</td>
<td>0.047</td>
<td>V/Rad/s</td>
<td>Back-emf</td>
</tr>
<tr>
<td>$L_{motor}$</td>
<td>0.05</td>
<td>H</td>
<td>Motor inductance</td>
</tr>
<tr>
<td>$J_{tot}$</td>
<td>0.01</td>
<td>kgm$^2$</td>
<td>Rotor inertia</td>
</tr>
<tr>
<td>$d_m$</td>
<td>0</td>
<td>Nms/rad</td>
<td>Viscous friction</td>
</tr>
</tbody>
</table>

From the equations

$$U = R_{motor} \cdot i + L_{motor} \cdot \frac{d}{dt} + k_e \cdot \dot{\rho} \tag{A.1}$$

and

$$J_{tot} \cdot \ddot{\rho} = K_m \cdot i + d_m \cdot \dot{\rho} \tag{A.2}$$

the motor model was developed. The change of current ($di/dt$) is derived from equation (A.1) and integrated over time to get the current, which is then used to calculate the angle acceleration. The angle acceleration ($\dot{\rho}$) is derived from equation (A.2) and integrated to get the velocity. The velocity is then converted to revolution per minute (RPM) by using the following equation

$$RPM = \dot{\rho} \cdot \frac{60}{2 \cdot \pi} \tag{A.3}$$

Using the information gain from the validation as can bee seen in figures 12 and 13 the values for the equation could be determined. The values can be seen in Table 7. The motor model can be seen in figure 39.
Figure 39: Motor Model Simulink
A.1.2 Fan Model:

The fan model takes the density of the air depending on the temperature into account. The conversion factor RPM_to_Air_Constant was used to convert the RPM to the air velocity leaving the fan. RPM_to_Air_Constant = 2.5719e-04. In the model the density of the air and the area of the burner tube was used to calculate the mass flow of air. The model can be seen in figure 40.

Figure 40: Fan Model Simulink
A.2 Coolant Pump Model

The model can be seen in figure 41 shows a simple on/off control logic for 52 liters per minute in flow.

Figure 41: Coolant Pump Model Simulink
### A.3 Fuel Pump Model

The fitting for the model equation was done in Matlab with the help of the "fit" function [31]. It’s a function that fit a curve or surface to data set, and the "fitType" that was used is a quadratic polynomial curve called "poly2". This resulted in a polynomial with the data seen in table 8.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1_Fuel_pump</td>
<td>0.0007148</td>
</tr>
<tr>
<td>P2_Fuel_pump</td>
<td>-0.01506</td>
</tr>
<tr>
<td>P3_Fuel_pump</td>
<td>-0.4001</td>
</tr>
</tbody>
</table>

The PWM_Fuel_Pump_scale is a simple gain to correct the incoming duty cycle value for the polynomial equation. The Max_pressure and Min_pressure is to limit the values the pump model can output. A PI controller was created to simulate the rise time of the pressure, the values can be seen in table 9.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0.0016</td>
</tr>
<tr>
<td>I</td>
<td>0.33</td>
</tr>
</tbody>
</table>

The second input to the model is fuel pump IO, that can be seen in figure 42. It’s not used in this version. It’s a rest component from a decision to split the PWM signal into two signals. It was defined as a interface, therefore the input port has not been removed.
Figure 42: Fuel Pump Model Simulink
A.4 Nozzle Model

The nozzle model has three inputs and two outputs. In the upper half of the model, the calculation of the nozzle heater temperature takes place. It uses the nozzle heater on/off signal to control the nozzle temperature. It’s a simple increase/decreases logic, where the current temperature rises when the input "Nozzle heater" is larger than zero, or the temperature falls if the input is equal to zero. In both cases the change is scaled by the sampling time(Ts). If the temperature reaches the MaxNozzTemp, it stops at that value, as long the input Nozzle heater is larger than zero. If the Nozzle heater input is zero, the temperature will decrease until it reaches the TempNozzleStartValue. The increase/decrease values was matched from test data. The calculated temperature was returned from the model.

In the lower half of the model the calculation of the fuel flow that is leaving the nozzle takes place. The input signal "Pressure" is translated into a fuel flow[g/min] with a polynomial function that was created from measurement data, see figure 15. The Matlab function fit was used with the fitType "poly2". The coefficient can be seen in table 10.

Table 10: Fuel flow polynomial coefficients

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1_Fuel</td>
<td>-0.2519</td>
</tr>
<tr>
<td>P2_Fuel</td>
<td>6.661</td>
</tr>
<tr>
<td>P3_Fuel</td>
<td>-0.8269</td>
</tr>
</tbody>
</table>

The fuel flow is then translated from [g/min] to [kg/s] by multiplying with MinToSeconds and GramToKiloGram.

The input signal "Fuel Valve" controls if the output signal "Fuel Flow" of the nozzle model will be the calculated fuel flow, or if it will be zero. The last step is to divide the outgoing value that is in [kg/s] with the sampling time(Ts) to get the correct amount of flow per iteration [kg/Ts]. This can be seen in figure 43.
Figure 43: Nozzle Model Simulink

A.5 Fuel Combustion Model

The fuel combustion model can be seen in figure 44, has three inputs and two outputs. The inputs are "Air flow", "Fuel flow" and "Glowplug I/O". All three signals are used to calculate the burn-state. The burn-state was used to check and control if there was combustion inside the heater. The fuel and air flows are used to calculate the lambda($\lambda$) value. The input "Air flow" can be written as $\dot{m}_{\text{air}}$ and
the input "Fuel flow" can be written as $\dot{m}_{fuel}$.

It calculates lambda by checking the ratio between the different flows compared to the Air_Fuel_Ratio. The Air_Fuel_Ratio depends on what fuel was used. The lambda values that can be measured by the lambda sensor is between 0.65 to 5 lambda, therefore the minimum and maximum lambda value that can be calculated was limited to the 0.65 - 5 range by the use of the variables Min_Lambda and Max_Lambda.

**Figure 44:** Fuel Combustion Model Simulink
The "BurnStateModel" controls if the calculated lambda value gets returned or not. It’s a function that have three inputs and a true/false output as can be seen in figure 45.

Figure 45: Burner State Model
A.6 Heat Exchange and Cooling Model

The Heat Exchange and Cooling Model (HEX model) have two inputs and three outputs. The inputs are "Fuel Energy" (read combustion energy) and Coolant flow. The outputs of the model are three temperature values NTC1, NTC2 and NTC3. NTC1 was the input temperature of the cooling medium into the HEX. NTC2 was the temperature of the cooling medium exiting the HEX. NTC3 was the temperature of the cooling medium close inside the heat exchanger.

The model was a general HEX equation with delay and cooling between output and input of the cooling medium. The setup of the real world test was to pump the cooling into/out of a tank. The model takes the cooling liquid into account, it was 60% glycol and 40% water. The delay and temperature drop of the cooling medium outside the heater was fitted from data of a heater system in a lab environment. Therefore the model only fits a test system. The equation for the mixture, 2.14, was implemented in the model. Where the volume of the heated liquid was the same volume that passed through the heat exchanger in one iteration of the model.

It was approximated that the mix between the heated liquid and the rest of the liquid happens instantly due to the fact that the flow rate was quite high compared to the whole volume and it would also make the model less complex. The model can be seen in figure 46. To have a better model of this it would require lots of "measurement points" stored in the memory which would slow down the model.
Figure 46: Heat Exchange and Cooling Model Simulink
B Code Listings

Listing 1: Sample from "Motor.h" file
/* Block signals and states (auto storage) for model 'Motor' */
typedef struct {
    real_T DiscreteTimeIntegrator1_DSTATE;
        /* '<Root>/Discrete-Time Integrator1' */
    real_T DiscreteTimeIntegrator_DSTATE;
        /* '<Root>/Discrete-Time Integrator' */
    real_T Jtot;
        /* '<Root>/Jtot Memory' */
    real_T K_e;
        /* '<Root>/K_e Memory' */
    real_T K_m;
        /* '<Root>/K_m Memory' */
    real_T L_motor;
        /* '<Root>/L_motor Memory' */
    real_T R_motor;
        /* '<Root>/R_motor Memory' */
    real_T d_m;
        /* '<Root>/d_m Memory' */
} DW_Motor_f_T;

typedef struct {
    DW_Motor_f_T rtdw;
} MdlrefDW_Motor_T;

Listing 2: Sample from "Motor.c" file
/* Start for referenced model: 'Motor' */
void Motor_Start(DW_Motor_f_T *localDW)
{
    /* Start for DataStoreMemory: '<Root>/Jtot Memory' */
    localDW->Jtot = 0.01;

    /* Start for DataStoreMemory: '<Root>/K_e Memory' */
    localDW->K_e = 0.047;

    /* Start for DataStoreMemory: '<Root>/K_m Memory' */
    localDW->K_m = 0.2;

    /* Start for DataStoreMemory: '<Root>/L_motor Memory' */
    localDW->L_motor = 0.05;

    /* Start for DataStoreMemory: '<Root>/R_motor Memory' */
    localDW->R_motor = 0.4;
}