Designing a generic fieldbus framework and refactoring legacy code for multiple platforms

at Beijer Electronics

Simon Hillbom
Robin Lindberg

2015

LUND
UNIVERSITY

Master's Thesis

Electrical Measurements

Faculty of Engineering LTH
Department of Biomedical Engineering

Supervisors:
Christian Antfolk
Christian Thorell

Abstract

When working with large programming projects, one of the major problems is to write the code so it can be maintained. If you write code that is easy to maintain it is also easier to extend, change and fix bugs in the code.

In this project two large, similar projects were investigated to find similarities in the code. If the similarities could be brought together in a generic solution, the code would be easier to maintain, understand and develop. The two projects are separate projects for communication with field buses, running different fieldbus protocols on different hardware platforms. The common factors in these projects are that both need to communicate over a fieldbus, somehow store the information received and then be able to communicate with a PC to send the gathered data or receive commands to send on the fieldbus.

By adapting modern refactoring methods to an older programming standard, code written in C (C89) can be refactored to a more modular and extensible code standard.

Sammanfattning

När man jobbar med stora pogrammeringsprojekt är ett av de stora problemen att skriva koden så den går att underhålla. Skriver man kod som är lätt att underhålla är det också enklare att göra tillägg, ändringar och åtgärda problem.

I detta projekt har två stora, liknande projekt undersökts för hitta likheter i koden. Om likheterna kan sammanföras till en generisk lösning kommer koden att vara lättare att underhålla, förstå och utveckla. De två projekten är två olika projekt för att kommunicera med fältbussar, som kör olika fältbussprotokoll på olika platformar. De gemensamma faktorerna i projekten är att båda måste kommunicera över en fältbuss, på något vis lagra informationen och därefter kunna kommunicera med en PC för att skicka över fältbussdata eller ta emot kommandon för att skicka data ut på fältbussen.

Genom att anpassa moderna refaktoriseringsmetoder till en äldre programmeringstandard kan även kod skriven i C (C89) refaktoriseras till en mer modern modulär och förändringsbar kodstandard.

Preface

This project took place during the spring of 2015 at Beijer Electronics in Malmö, Sweden. The project aimed to solve the company's problem with time consuming support and maintenance of firmware for a multitude of protocols for fieldbusses. The goal was to bring the firmwares together to a more generic solution which covers all the protocols.

Acknowledgements

We would like to thank Christian Antfolk and Christian Thorell for their great help and support as advisers to this thesis process. We would also like to thank Stefan Klein, Carl Andersson and Beijer Electronics, as a whole, for making this project possible.

Contents

1 | Introduction

The project took place at Beijer Electronics. Beijer Electronics HMI's (Human Machine Interface) use different modules to connect to fieldbusses. This way the HMI may be replaced while keeping the module, and the same HMI hardware may be used on many different fieldbus protocols. A typical setup can be seen in figure 1.1, where different industrial hardware could be connected to the bus.

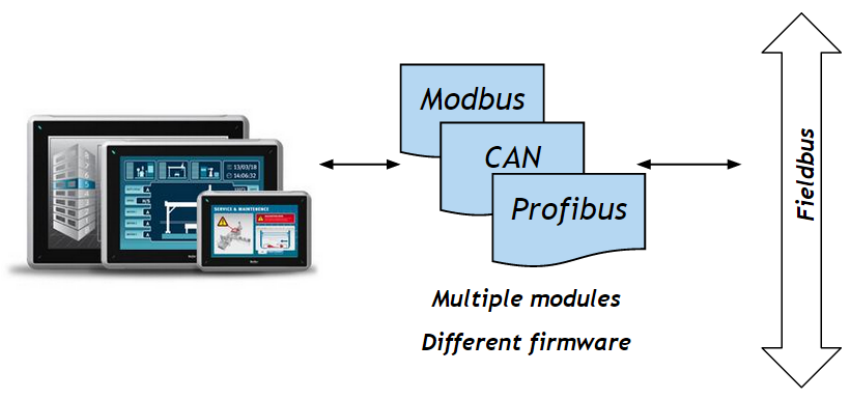

Figure 1.1.: The Beijer Electronics HMI connects to the fieldbus using a module for the specific bus type.

All these different modules use different firmware to handle the communication. The main problems all the modules solve is to lift data from the bus to the module and to send data from the module onto the bus. Since this is very similar for all the firmwares it becomes interesting to investigate if a common, generic firmware can be created.

This posed the following question: Is it possible to create a generic driver for several fieldbus protocols and different hardware platforms?

1.1. Goals

When creating the project plan some main goals and milestones were defined.

Research protocols and investigate similarities Research what protocols are most widely used and have similar properties. The generic driver should be of large value to Beijer Electronics and should also be possible to develop.

Investigate feasibility of generic driver engine Test and find out if a driver is possible. Is it possible to break apart the entire code, or some smaller piece. Can a part like hardware communication be made into a generic framework.

Build generic driver Develop the part that was evaluated as a feasible part. The second task will decide the scope of the project and what can be done in the generic driver software.

Integrate driver into Beijer Electronics device firmware Integrating the firmware so that the generic driver firmware can replace the current solution.

1.2. Beijer Electronics

Beijer Electronics is a global company with offices for development in several countries. The main focus of the company is automation and communications solutions. With powerful and compatible solutions industrial companies can improve performance and efficiency while keeping the costs down.

1.2.1. HMI

One of the core products at Beijer Electronics is the HMI's or panels. Most of Beijer Electronics HMI's feature a touch screen interface while some have keyboards or keypads for navigation. The HMI's are equipped with

Ethernet and serial connectors for communication purposes. The main goal of the HMI is to represent the states of a machine to a human. Gauges and meters can represent pressures, flows or other states that the machine can measure. On the HMI the human operator could monitor and control these parameters.

In order to allow the HMI's to communicate on the vast amount of fieldbus protocols that is supported by Beijer Electronics, a small module is connected to the back of the HMI. More information about fieldbusses are available in chapter 2.1. The module can be equipped with one or more ports to connect with onto the bus. Communication between the module and the HMI is done over USB/Serial.

The module handles all the communication on the network and the panel only needs to ask for information, or submit information to be sent on the bus. This communication, between module and panel, is handled through a generic interface and driver engine built by Beijer Electronics.

1.2.2. iX

The main software that Beijer Electronics develops is iX developer. Using iX a developer or customer can create custom interfaces that runs on a Beijer HMI. Since Beijer has control of the top level software running on the HMI's Beijer has control over and develops the drivers that communicates with the fieldbus modules.

2 | Background

This chapter contains some information and explanation of the different hardware, software and methods encountered in the project.

2.1. Fieldbus protocols

A fieldbus is a type of communication standard that is widely used in automation, industry, cars and other similar applications. A typical use for a field bus in industry would be to connect several machines and sensors with PLCs (Programmable Logic Controller) that communicate on the bus, an operator could then control and monitor the system on a PC or some kind of panel.

There are a large variety of fieldbus protocols, cables and connectors. Many companies develop their own standard due to lack of features, security, speed or other requirements that are not available in the more common protocols.

Some of the more widely used protocols are:

- CAN
- PROFIBUS
- Modbus
- Interbus

Each of the protocols have their own limitations and communication standards, some can also have a variety of sub protocols. CAN is often used as CANopen or J1939, depending on the application. These different

standards may make it easier for engineers to find good default settings to start with on a new application. For example, J1939 uses the most fault tolerant and secure settings available in CAN, and is recommended for automotive use. There are also some standard ID's (identifiers) that can be used for certain temperatures, pressures or similar sensors.

2.1.1. Bus compared with point to point

The main difference when comparing a field bus for communication (such as CAN) with a more normal communication standard such as Ethernet (TCP/IP) is that on a bus all or some devices are connected on the same cable. When using Ethernet each device is connected to only one port or device.

For example a computer may be connected to a switch, which is connected to two more computers, the switch will then route the data to the correct receiver. Such an example can be seen in figure 2.1. In this case, computer C will have no idea that communication occurs between computer A and B. This is very important, especially in large networks (the Internet would work poorly if all data was transmitted to everyone) or secure applications.

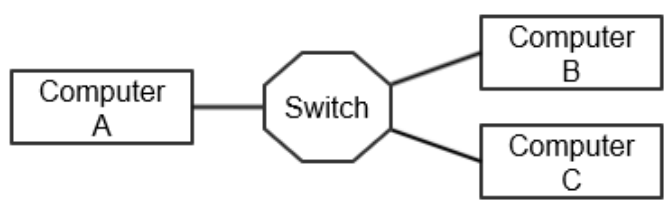

Figure 2.1.: Three computers and a switch in a point to point topology.

On the other hand, when using bus communication all data is transmitted to everyone which presents a lot of interesting problems and solutions. In figure 2.2 an example setup of a bus is shown. In this case if PLC A sends a message to B, both PLC C an D will receive the message and discard it.

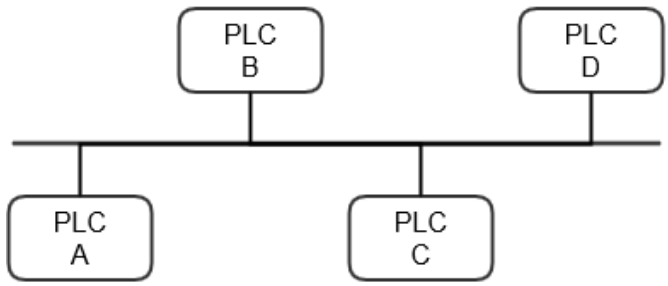

Figure 2.2.: Four PLCs connected to a common bus.

2.2. Physical bus connections

The physical media and connector used to communicate varies as much as the protocols. In one application the datagrams may be transferred over Ethernet and IP between the different units, another application may use CAN and two dedicated wires. Some application trade more wires and higher voltage for increased speed, while another protocol may utilize the improved signal quality for better fault tolerance. In a car it might be preferable to use a system with very high fault tolerance.

In this project two of the more popular protocols were chosen as targets for a generic driver design.

2.2.1. Bus protocols explained with the OSI model

The Open System Interconnection model (OSI model) is handy when describing the differences in the protocols and a short explanation should suffice. More in depth information about the OSI model is available in the book "Data Communications and Networking[8, p. 30].

2.2.1.1. The OSI model

The OSI model is described by the following seven layers:

1. **Physical** Describes the cable (how many pins, what connector etc.) and what represents a logical 1 or 0 (such as bit encoding, voltages etc.)
2. **Data link** Transmission of data frames onto the network. In CAN a data frame is called a telegram. The data link layer also describes

what low level error detection and error correction to use.

3. **Network** Defines how different devices address and connect to each other on a multi-node network. IP and routing is located in this layer.

4. **Transport** Defines how to ensure reliable transmission for the Network layer datagrams/packets. The transport layer could specify that packages should be numbered and the receiver should receive all packets in order, or that dropped packages does not matter as in UDP.

5. **Session** Handles the connected session for continuous communication.

6. **Presentation** When a complete file or message has been transferred the presentation layer describes the file presentation. For an image the presentation layer could be the PNG format, an audio file could be mp3 or a text message could be UTF 8.

7. **Application** The high-level application that works with data in the presentation layer. A computer could send a message to a PLC controller or a web browser could request a HTML document using HTTP.

Since fieldbus communication is mostly low-level most protocols operate on layer 1 and 2. Sometimes layer 3 is used to connect between buses. In many applications the layer 3 data is used directly by the application, bypassing some of the session and translation features in the higher level layers.

A CAN controller could receive a telegram and if it is addressed to that specific controller the message could be copied over serial or USB to a PC that extracts the interesting data from the telegram and presents it on a monitor.

2.2.1.2. CAN

CAN (Controller Area Network) is a real time protocol with very high security and reliability. CAN is most often used in automotive applications and supports a bitrate of up to 1 Mbit/s. To describe the CAN protocol the pysical, data link and application layer can be used.

2.2.1.2.1. Physical Layer

The physical layer may be defined by the application or engineer as the standard covers the handling of the sent datagrams. With that known, the most usual application is two wire communication with regular serial port connectors. In this standard setup one wire drives a low voltage and is called CAN Low and the other one is a higher voltage called CAN High.

The differential signal can be described as follows, with the resulting signal V_{out}:

$$CAN_{high} - CAN_{low} = V_{out} \tag{2.1}$$

By using a differential signal an increase in reliability is achieved. If a arbitrary disturbance e is introduced on both CAN_{high} and CAN_{low} it can be shown that the measured signal remains the same:

$$(CAN_{high} + e) - (CAN_{low} + e) = CAN_{high} - CAN_{low} = V_{out} \tag{2.2}$$

When the bus is idle both CAN High and CAN Low rests at 2.5 V. If a signal is sent CAN High rises to 3.75 V and CAN Low drops down to 1.25 V as seen in figure 2.3. The resulting 2.5 V differential is the measured signal.

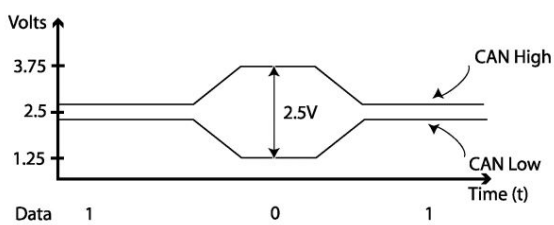

Figure 2.3.: CAN uses a differential signal to increase transmission quality. Image from [6].

The use of a differential signal makes the bus insensitive to electrical fields and other noise. Since the bus is often used in vehicles it is important to use a transfer method like this. Sometimes power is transferred on other

wires in the CAN cable, generating an electrical disturbance that could have caused interference on the bus during power surges.

To make sure that signals are not sent at the same time, the node with the lowest ID will always have priority. If two nodes starts to transmit at the same time the one with the higher ID will stop and let the lower ID node continue.

2.2.1.2.2. Data link layer

The CAN bits are ordered into telegrams or data frames as seen in figure 2.4. Most of the CAN standard and rules are applied to the data link layer. For example error detection, message validation and acknowledgment rules are defined.

There are four types of telegrams that can be sent:

Regular this telegram contains data being sent. A temperature report may be sent in this frame.

Remote are used to request a data transmission.

Error is sent if an error is detected on the node.

Overload are used to put delays between other frames.

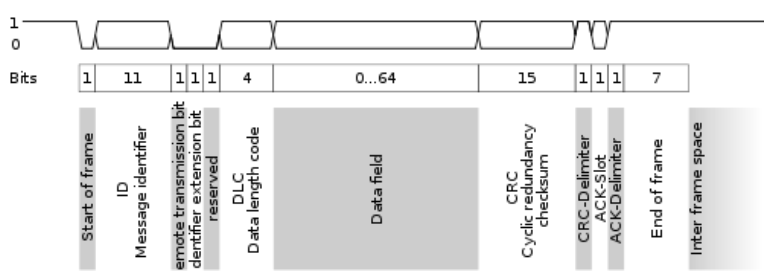

Figure 2.4.: A CAN telegram containing ID, data and CRC, which is used for error detection. Image from [12].

As this is a bus, addressing is also done in the data link layer. Addressing is done by assigning ID's to the different nodes, so that each node knows what messages it is supposed to receive.

2.2.1.2.3. Application Layer

On the CAN application layer, messages can be filtered and handled. For example a node could receive a message that requires a temperature to be set. The node could then gather data from other nodes and send data to the actuators so that the temperature is changed correctly.

2.2.1.3. PROFIBUS

PROFIBUS is a standard for fieldbus communication, often used in automation technology. There are two variants of PROFIBUS, DP (Decentralized Peripherals) and PA (Process Automation). This section focuses on the former, which is more widely used. PROFIBUS DP is a master-slave-protocol and supports bitrates from 9.6 kbit/s up to 12 Mbit/s. It was implemented with the OSI-model as a basis, though it only uses layer 1,2 and 7.

2.2.1.3.1. Physical Layer

PROFIBUS specifies three different connection methods, RS-485, optical transmission and MBP (Manchester Bus Powered) transmission. Optical transmission is used for long distance connections and MBP is used in explosion-hazardous environments. RS-485 is the most common connection method, using a twisted, shielded copper cable with a pair of wires.

The two twisted data lines, often referred to as line A and line B, carry different signals. The receiver can determine if a binary 1 or 0 was sent by calculating the difference in voltage between the two wires according to the following equations:

$$V_A - V_B < -0.2V = 1 \tag{2.3}$$

$$V_A - V_B > 0.2V = 0 \tag{2.4}$$

An example of a signal can be seen in figure 2.5. By comparing the signals, problems such as several volts difference in the ground level of transmitter and receiver are bypassed. Also, by using twisted cables, noise from electromagnetic radiation is reduced as seen in figure 2.6.

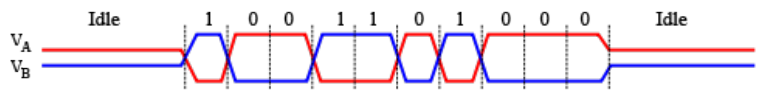

Figure 2.5.: An example of a signal sent in RS-485.

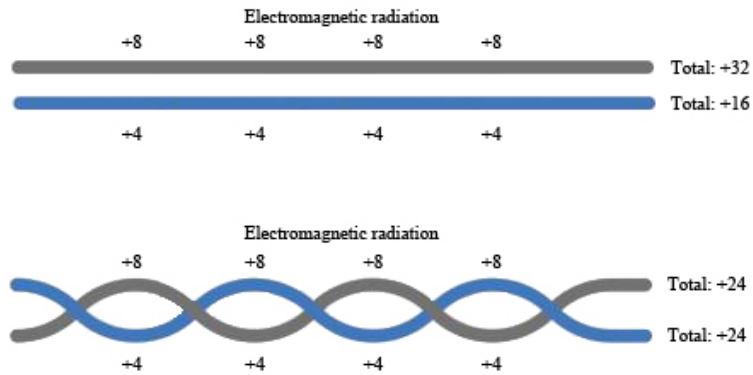

Figure 2.6.: Cables with normal and twisted pair of wires are affected by radiation and noise differently.

2.2.1.3.2. Data link Layer

PROFIBUS calls its data link layer FDL(Fieldbus Data Link). This layer uses a combination of the master slave method and the token passing method along with a collection of telegrams for different kind of data packages. In the master slave method one device is in control of all other devices. In the token passing method only the device who holds the token may take action or communicate on the bus. The token is passed on to the next in line when the one currently holding it is finished. In PROFIBUS these two are mixed in to a hybrid method where it is possible to have multiple masters. In order to not have several masters giving orders simultaneously, the token method is applied to them. This way only the master who is currently holding the token may command the others. In

a PROFIBUS network the PLCs or controllers are the masters and the sensors, HMI's and other hardware are the slaves.

2.2.1.3.3. Application Layer

The application layer is home to an interface to the PROFIBUS DP protocol and thus serves as a link between communication and application. PROFIBUS DP is defined in three different service levels, each with a specific area of use. DP-V0 is used for cyclic exchange of data and diagnosis. DP-V1 is used for acyclic data exchange and alarm handling. DP-V2 is used for isochronous mode and slave to slave communication.

The actual application lies above the application layer (layer 7) and is not part of the OSI model.

2.3. Hardware platforms

Beijer Electronics has developed several modules for different protocols and HMI's. A module is a kind of backpack that is attached to the back of the HMI to allow the HMI to connect to different fieldbus networks.

This chapter describes some of the more important components of the PCB for each protocol.

2.3.1. CAN

The Beijer Electronics CAN module[1], figure 2.7, features a serial over USB connection and dual CAN ports. New firmware can be programmed either over the USB connection using the custom bootloader or a complete firmware may be flashed using a special adapter and a JTAG interface. The special adapter used is the KEIL ULINK 2[2].

The main processor on the module is the NXP LPC2388[11].

2.3.1.1. LPC2388

For the application the LPC2388 is a decent chip. The comparatively low cost and the available features makes it a suitable product. Notably there

[1] `http://beijerinc.com/product/industrial/can.php`
[2] `http://www.keil.com/ulink/`

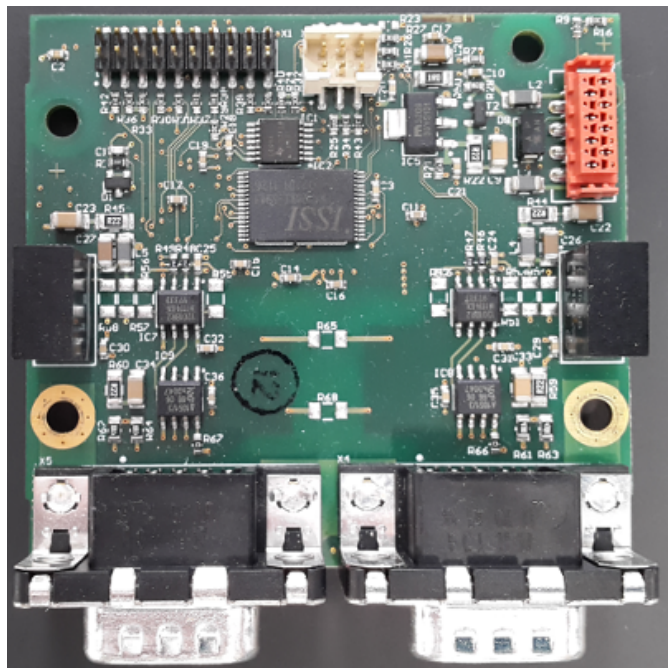

Figure 2.7.: The Beijer Electronics CAN module featuring an NXP LPC2388 processor.

is hardware support for dual CAN connections and UART/USB connections for communication. There is also support for hardware and software interrupts.

The single purpose of the CPU and the high timing requirements in the application gives little time for other tasks. This means that runtime logging is a bad idea as it simply will take to much time that the CPU needs to use for other things.

As comparison the popular Raspberry Pi 2 is a very powerful multipurpose product and would have no problem logging to a website and plain serial logging would be a breeze.

This means that debugging and testing the code will be difficult, both due to the rather slow hardware and that the protocol is real time. With the real time protocol, step through debugging will not always work as the system expects immediate replies to sent packages.

2.3.2. Profibus

The Beijer Electronics DP module[3], figure 2.8, allows the HMI to communicate over PROFIBUS. It has one serial connector and three led lights on its front panel. On the board itself there are several components. The C8051F380, the blue square in figure 2.8, from Silicon Laboratories is the MCU (micro controller) which contains the main code. There is also a MPI12x chip, the yellow square in figure 2.8, from Profichip which contains PROFIBUS logic and is used by the C8051F380. Two of the led lights are connected to the MPI12x and the third to the MCU. This third light can be controlled from C-code on the MCU. In order to put new firmware on the MCU one can flash it using JTAG and a piece of hardware from Silicon Laboratories called USB Debug Adapter[3].

2.3.2.1. C8051F380

C8051F380 from Silicon Laboratories has performance similar to the LPC2388, as stated earlier this adds constraints on what and how often tasks can be done. The module also sports a MPI12X profichip to handle the physical layer communication, as this is not built into the main CPU.

The C8051F380 suffers from the same limitations when it comes to debugging and logging as the LPC2388. When not doing excessive logging the C8051F380 combined with the MPI12X is quick enough to handle plenty of PROFIBUS communication.

2.3.3. Common properties

The two platforms have some common properties. For example both modules use C89, also called ANSI C. This is an older version of the programming language C with some not commonly known constraints. In C89 any declaration of variables must happen in the beginning of a code block. Also the usual double slash comments are not supported. Keil does not support compiling C99 for the C8051F380, so C89 was used as a common language. Other similarities are that both modules have JTAG capabilities and serial connectors for the fieldbus cable.

[3] `http://beijerinc.com/product/industrial/cix-dp.php`

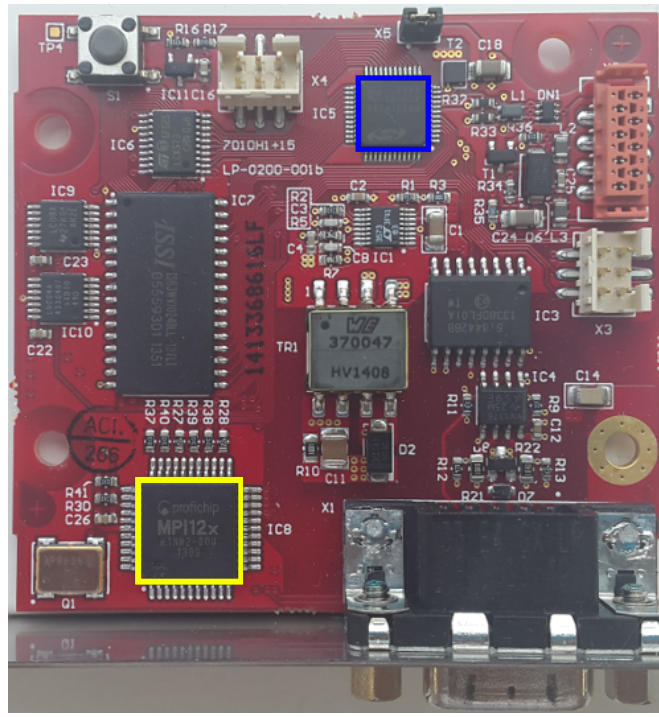

Figure 2.8.: The Beijer Electronics DP module featuring the C8051F380 processor (blue) and the profichip MPI12x (yellow).

2.4. Legacy C Code

Jumping into a large project can always be a new experience. There are a plethora of tools available to make the transition smoother. Sadly, this project did not use any of these tools.

The early plan was to build upon and refactor the existing code until a generic driver was reached.

2.4.1. Existing driver design

Designing the code before writing is very important as this can help the developer to maintain principles such as single responsibility and a general sense of what file should contain a specific piece of code. Code in

need of refactoring or replacing can often be identified by different ”code smells”[10]. These ”smells” are can for example be a bad pattern or a ill chosen variable name.

2.4.1.1. Existing frameworks

The existing design is based on a demo project. Some parts are written by NXP, some are from other demos from NXP developers but most of the code is built in-house. This way the code contains many different design patterns so that some calls are polling even as there are interrupts available. Interrupts are used in some cases with different types of calls for similar methods, a send method may use a buffer pointer whereas a receive method may put the data in a global variable.

2.4.2. Existing coding style

Moreover the code has a very varying quality/readability standard. While some functions may almost follow a nice code standard[4, chap. 6] some functions grow over 1000 lines with nested if cases several levels deep and gotos that can take the reader anywhere in the code.

This mix of design made it incredibly time consuming to work with the code. For example, some of the NXP USB code could look like the code shown in example 1, a part of a 280 lines long method.

```
1  if (disr & EP_SLOW_INT) {
2          episrCur = 0;
3          episr    = EP_INT_STAT;
4          for (n = 0; n < USB_EP_NUM; n++) {
5                  if (episr == episrCur) break;
6                  if (episr & (1 << n)) {
7                  episrCur |= (1 << n);
8                  m = n >> 1;
9                  EP_INT_CLR = (1 << n);
10
11                 while ((DEV_INT_STAT & CDFULL_INT) == 0);
12                 val = CMD_DATA;
13                 if ((n & 1) == 0) {
14                         if (n == 0) {
15                                 if (val & EP_SEL_STP) {
16                                         if (USB_P_EP[0]) {
17                                                 USB_P_EP[0](↩
                                                       USB_EVT_SETUP);
18                                                 continue;
19                                         }
20                                 }
21                         }
22                         if (USB_P_EP[m]) {
23                                 USB_P_EP[m](USB_EVT_OUT);
24                         }
25                 } else {
26                         if (USB_P_EP[m]) {
27                                 USB_P_EP[m](USB_EVT_IN);
28                         }
29                 }
30         }
31 }
```

Example 1: An example of what some of the badly designed code looks like.

On the other hand some code was clearly written, with explanatory function and variable names, as seen in example 2.

```
1  int CDC_WrOutBuf (const char *buffer, int *length) {
2          int bytesToWrite, bytesWritten;
3
4          /* Write *length bytes */
5          bytesToWrite = *length;
6          bytesWritten = bytesToWrite;
7
8          /* Copy Data to buffer  */
9          while (bytesToWrite) {
10                 CDC_BUF_WR(CDC_OutBuf, *buffer++);
11                 bytesToWrite--;
12         }
13         return (bytesWritten);
14 }
```

Example 2: An example of what some of the better written code could look like.

So, while the first piece of code may do what it should perfectly, it does not make it easy for the reader to understand how. By describing the key errors by line of code this becomes clear:

line 1 The developer is using & to decide what to do. This might be regular in C, but the normal method of using == is much more clear. A much better way would be to use a method to decide what interrupt has occurred, so that a switch case could be used as in example 3.

```
1  switch(getInterruptType(disr){
2          case EP_SLOW_INT:
3                  ...
4                  break;
5          case ...
6  }
```

Example 3: An example of a switch case implementation.

line 2, 3 The use of short variable names makes it difficult to understand what is happening. Compared with the variables in the other part of the code, bytesToWrite and bytesWritten, which clearly indicates what they represent, episr is on the verge of obfuscating the code.

line 4 A new global variable `USB_EP_NUM` is suddenly introduced. To improve testing and understanding this variable should have been in the method call.

line 13 - 29 The entire if nesting can be severely reduced without affecting functionality. With this change the setup, in and out events are more separated and the setting part will not have to be duplicated. How this would look is shown in example 4.

```
1  int event = -1;
2  if((n & 1) == 0){
3      if((n == 0) && (val & EP_SEL_STP) && (USB_P_EP[0]){
4          event = USB_EVT_SETUP;
5          m = 0;
6      } else if (USB_P_EP[m]) {
7          event = USB_EVT_OUT;
8      }
9  } else if(USB_P_EP[m]){
10         event = USB_EVT_IN;
11 }
12
13 if(event != -1)
14     USB_P_EP[m](event);
```

Example 4: An example of a refactorization of the if nesting.

Another big miss in quality standard was the use of somewhat german names of variables, methods and sometimes also in comments. How this could look can be seen in example 5. The mixed languages impeded un-

```
1 //if(check_reply_telegrams(index,kanal,dat) >0) ;//goto auftragfertig;
2 check_reply_telegrams(index,kanal,dat);
```

Example 5: An example of english mixed with german in variables and comments.

derstanding as not only the code in itself was difficult to understand, the words could not simply be read without translating them beforehand. Some words were also a made up mix of german and english word, making it even more difficult.

2.4.3. Refactor or Replace?

After working with the existing code for a while it had to be decided if the existing codebase could be fixed, if some part could be saved or it was time to let it go and start anew. In order to know if refactoring is a valid option, one first have to know what approaches can be taken.

3 | Methods

This chapter contains the methods, approaches, software and hardware used throughout the project.

3.1. Software and hardware

The practical parts of the thesis involved several software and hardware products. In this section the use of these will be explained.

3.1.1. IDE

There are many IDEs (Integrated Development Environment) to choose from. The CAN source code was written in Keil µVision[1], whilst the PROFIBUS source code was written in Silicon Laboratories[2]. Silicon Laboratories was lacking basic functions for speeding up and aiding code development. For example it does not support finding the definition or declaration of variables and methods. This is used regularly by a developer. These functions did however exists in Keil µVision. µVision was chosen as the projects IDE and the PROFIBUS code would be migrated to it.

In order for µVision to be able to build the PROFIBUS firmware it needed the Silicon Laboratories device driver and compiler. It also needed to be able to use the USB Debug Adapter. For further information about how to program for the C8051F380 using Keil µVision 5, see Appendix B.

3.1.2. Debugging

Keil is well equipped for debugging purposes. One can choose to run a simulation or choose a suitable driver and run the firmware on chip. Both

cases allow for line to line debugging with breakpoints, memory display and watch windows for viewing variable values in runtime.

Although, since real time protocols are handled, debugging can sometimes prove to be difficult. For example when debugging a USB connection the connection is not made since the software is to slow to react during the handshake. There are several instances where similar time limits exists which makes debugging in such places troublesome.

In order to debug the software a testing environment for each protocol had to be arranged.

3.1.2.1. PROFIBUS test environment

PROFIBUS test setup consisted of a PLC connected to the Beijer Electronics DP module using a PROFIBUS cable. The PLC, a Siemens Simatic S7-300[1], shown to the left in figure 3.2, was acting as a master on the network. It could also communicate to the PC via an Ethernet cable. The DP module was acting as a slave and was connected to the PC via the USB Debug Adapter, which can be seen to the right in figure 3.1.

Beyond the hardware, the test setup also consisted of a number of software components. First of all, as mentioned earlier, one could line-to-line debug the code from µVision using the USB Debug Adapter. In order to test the PROFIBUS functionality a software called OPC server was used alongside a Beijer Electronics iX project. Section 1.2.2 gives a short explanation to what iX is. OPC server is a software developed by Beijer Electronics for monitoring and manipulating PLC values. The iX project simply had the functionality to, using the PROFIBUS protocol, request values from the PLC and set values on the DP module. These tools allowed for a PROFIBUS communication loop. The loop consisted of five editable values both in iX and in OPC Server. The values in iX were monitored and displayed in OPC Server and vice versa. If a value was edited in iX and the corresponding value was updated in OPC Server, then the PROFIBUS communication worked in that direction. Similarly, if a value was edited in OPC Server and the corresponding value was updated in the iX project, then the PROFIBUS communication worked in that direction also.

[1] `http://w3.siemens.com/mcms/programmable-logic-controller/en/advanced-controller/s7-300/pages/default.aspx`

In order to see how the PROFIBUS communication behaved in more detail, a tool called ProfiTrace could be used. It is a two part tool set. One part is the ProfiTrace software and the other is the ProfiCore Ultra, seen to the left in figure 3.1.By using it, one could see each package sent over the bus and detect any errors that occurred.

Figure 3.1.: The ProfiCore Ultra and the USB Debug Adapter respectively.

3.1.2.2. CAN test environment

To test the CAN module an iX project was set up that used both CAN ports on the module. The test was built so that data could be entered into a field, sent over the bus and displayed in another field. If the module worked then the data would be updated as expected.

For debugging and flashing purposes the ULINK2 adapter was used. There was also equipment from ixxat available, with CAN analyzing software. The ixxat USB to CAN debugger was used to do package inspection.

There was also the logic analyzer as seen in figure 3.3. The logic analyzer could be attached on exposed pins on the CAN bus and with it very low level debugging could be performed.

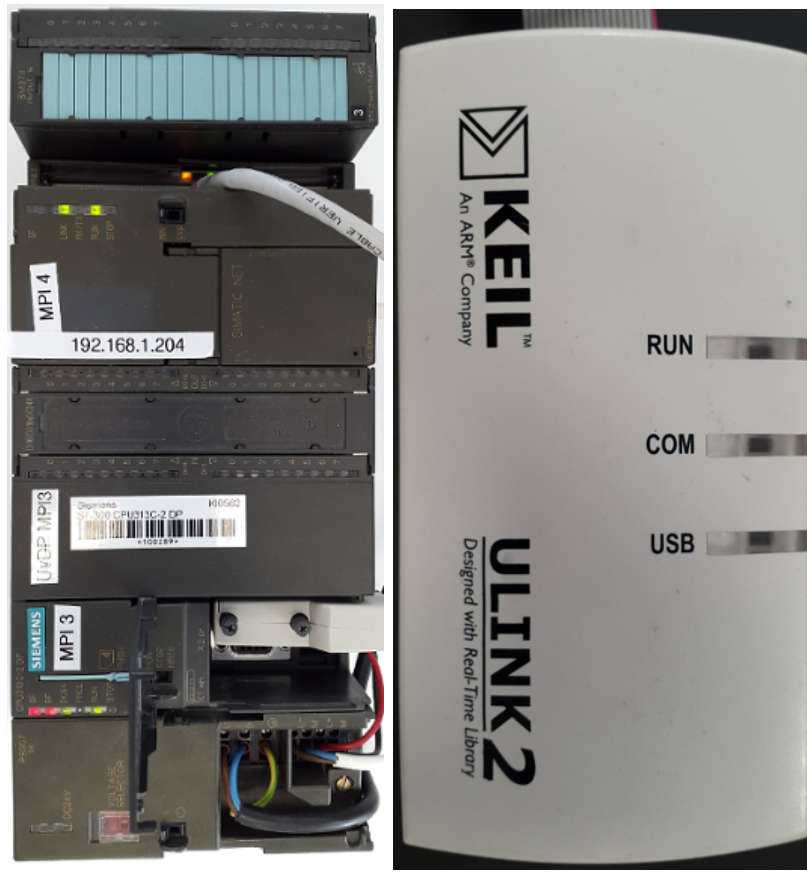

Figure 3.2.: The Siemens S7-300 PLC and the KEIL ULINK2 respectively.

3.2. Generic Driver Design

Designing code from the beginning is very important. In some cases a good design improves performance, but for the most part it is done to decrease the time it takes to make changes to the code later on. If a project contains unclear or badly designed code, even a small change can take huge amounts of time to implement. Developers will have to hunt through the code until they finally find the piece of code that performs the desired function. Then another investigation must start as changing a piece of nested code may influence other methods and behaviors.

Figure 3.3.: The Saleae Logic 16 logic analyzer.

In other projects where the code is well designed, a developer could quickly find the code and visualize the scope of the change. The change could be done in a fraction of the time.

As Michael C. Feathers writes [7, p. 85] in his book, writing more interfaces, classes and files may take time and increase compile and implementation time. But since a developer can choose what parts to compile in a well structured system, total compile time can actually decrease. These changes, if correctly implemented, will only be written once, and the benefits they bring will always be there.

3.2.1. Defining a generic driver

In order to know what the goal is, a generic driver would have to be defined. There are many different methods and designs that are viable. In order to get improved, more easily understood and more independent code the generic driver had the following main goals:

100% generic framework With a generic framework that does not contain any code for the specific hardware or protocol it will be easier to add support for a new protocol or hardware platform since the existing code can be used to get started.

No global variables in the generic part Global variables makes the code difficult to work with as intent or function can be difficult to decipher. Global variables such as buffers, list, error variables or other things

that may come up can be used in the non generic parts of the driver. Usage of the global variables should happen in methods that clearly states the purpose of what is happening.

No sharing of non generic code The different platforms or protocols should not share code. If a piece of code can be shared with the other requirements in mind the code should be in the generic part.

Follow a code standard When the logic of the code is written in a structured way the code itself should also be formatted in a certain way or standard. Embracing a code standard has many benefits. Looking at change logs in version control software will make more sense and the code will be easier to read. Some code standards could also force developers to avoid certain ways of writing code, as example using long indentation would make many nested cases look ridiculous, therefore making the developer realize the mistake. Among the various standards the "Linux kernel coding style" [4] was selected.

3.2.2. Design diagram

In figure 3.4 an overview of the generic design can be seen. In this example the LPC2388 hardware platform is used. In order to switch platform one could keep the core, 'generic_Driver', parts and only connect new files containing the specific code for the new platform. This is possible since the code is decoupled and independent. This means the code gets a 100% generic framework, as explained in the goals. This layout also simplifies the goal of no sharing of non generic code. Any code which can be shared by several protocols or platforms can simply be moved into the generic block at the appropriate place.

More code could of course be put in the generic part, sorting algorithms, package detection and other methods could be put there if they are generic enough. This part only focuses on communicating with the module hardware.

3.3. Refactoring

In order to modify the existing driver to work with the generic framework some refactoring methods had to be investigated, evaluated and modified.

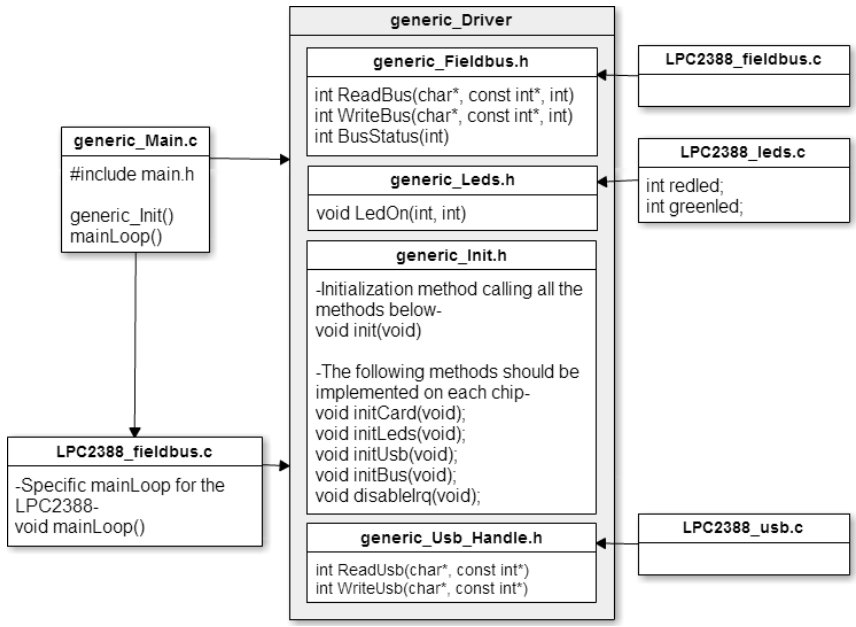

Figure 3.4.: An overview of the generic design, using the LPC2388 platform.

When applying code refactoring to a project the main goal is to increase code quality so that the code is easier to maintain, understand and improve.

In this chapter the techniques used in this project will be described and some changes made in the techniques to alter patterns used for C# or Java to C, as used in the project. SourceMaking[5] covers many patterns and methods, but for higher level languages where concepts such as classes are available.

3.3.1. Patterns for refactoring

The main goal with patterns is to help the developer approach code and use some kind of method to break down methods, classes or similar into smaller, more manageable parts.

Since many of the techniques are meant for languages with classes and interfaces an adaptation for C, which is not object oriented, is needed. .c files and .h files will be called and treated as classes and interfaces respectively.

3.3.1.1. Decorator

By applying interfaces to a class, supported parameters or methods can be applied to similar classes. Both classes may support fetching data, the decorator interface would then describe the different data sources.

The wrapping could be done on a high level, wrapping down on several parts.

- Datasender
  - FileDataSender
    - BusFileDataSender
    - EthernetStreamDataSender
  - StreamDataSender
    - BusStreamDataSender

The decorator pattern is often used to add methods or parameters to an object during runtime.

In this project a kind of decorator pattern was used to break out constant fields so that a developer could access the correct data while writing code.

3.3.1.1.1. Problem example

In 6 the two pieces of code are accomplishing the same thing, all leds are activated. By breaking these methods down into a single one and decorating the respective `.h` files with the available leds a more generic approach can be reached.

3.3.1.1.2. Solution

A more generic solution to the previously shown code snippets can be seen in example 7. The led on/off variable was available for both systems and is therefore available in the `leds.h` file, as shown in example 8. The method to control the leds have been extracted and is available on both platforms, in the same way. The leds on each systems is defined in the specific files `CAN_leds.h` and `Profibus_leds.h`.

The implementation of the `SetLed` method is done in the specific way for each platform.

```
1 //CAN file
2 RED1_LED_ON();
3 RED2_LED_ON();
4 GRN1_LED_ON();
5 GRN2_LED_ON();
6 YEL_LED_ON();
```

```
1 //Profibus file
2 Err_led_ON();
```

Example 6: Two sets of code turning on the leds of the respective module.

```
1 //CAN file
2 #include "CAN_leds.h"
3 SetLed(RedLed1, LedOn);
4 SetLed(GreenLed1, LedOn);
5 SetLed(GreenLed2, LedOn);
6 SetLed(YellowLed1, LedOn);
7 SetLed(YellowLed2, LedOn);
```

```
1 //Profibus file
2 #include "Profibus_leds.h"
3 SetLed(RedLed1, LedOn);
```

Example 7: More generic solutions for turning on the modules leds.

3.3.1.2. Chain of responsibility

By using chain of responsibility, the pattern of returning to the same file to handle returned event can be broken.

Applying a chain of responsibility will help improve single responsibility, making the code easier to understand. It can also help make the code more modular, as links in the chain could be swapped for other with the same input/output.

```
1 //leds.h
2 #define LedOn = 1;
3 #define LedOff = 0;
4 void SetLed(int, int);
```

```
1 //CAN_leds.h
2 #include "leds.h"
3 #define RedLed1 0
4 #define GreenLed1 1
5 #define YellowLed 2
6 #define RedLed2 3
7 #define GreenLed2 4
```

```
1 //Profibus_leds.h
2 #include "leds.h"
3 #define RedLed1 0
```

Example 8: The modules h-files and the leds.h-file for the generic solution.

3.3.1.2.1. Problem example

In this example the initialization methods in the main CAN program, shown in example 9, will be investigated. In the initialization method there are a lot of things going on. Variables are set, some globals are defined or initialized and some methods that does half the work are called. On line 3 `InitLeds()` is called but leds cannot be used until line 7-9 have finished. Then some things are set and and after a while the initialization continues.

This makes it really difficult to find out what parts of the code are used to setup the hardware and what code is used for other things. It is also difficult to discern what code is used for leds, bus communications or other setup.

```
1  //main.c
2  ...
3  InitLeds();              //Leds initialisieren
4  ...
5  WDMOD=  0x0;             //watchdog aus
6  //Addressumschaltung aktivieren ext Mem RAM2
7  PINSEL3  &=   0xFFC0FFFF;        //Pin 1.24-P1.26=0 (16...21=0)  GPIO ↩
       Pins
8  PINMODE3 |=   0x003F0000;        //Pin 1.24-P1.26=0 (16...21=0) pull ↩
       down resistor an
9  IODIR1   |= 0x07000000;
10
11 selectbank(1);           //ext Memory 0 üfr QL[]
12 init_timer();                                    // Initialize Timer
13 ...
14 ...
15 #ifdef serdebug
16          init_serial();      // Initialize Serial Interface
17 #endif
18 ...
19 init_watchdog(400);              //4 Sekunden Watchdog Timeout
20 WDFEED = 0xaa;
21 WDFEED = 0x55;
22 ...
```

Example 9: A selection of lines from the initialization of the CAN module.

3.3.1.2.2. Solution

A solution to the problem in the previous section is shown in code example 10. The files `main.c` and `init.c` does not contain any platform specific code. In this case the chain of responsibility have been applied to the problem. It becomes clear what happens during initialization and when the initialization is done. While this is only one level of the chain it can be applied in the same way on bigger methods, chaining the methods even deeper.

This does not only make it easier for the reader, it also improves testing and measuring. A developer could measure the time for a specific method or a group of calls. The module specific implementation would be done in one or similar other classes. The linking of the methods is done in the modules interfaces.

```
1  //main.c
2  int main(void)
3  {
4    init();
5    ...
6  }
```

```
1  //init.c
2  void init(){
3    disableInterrupts();
4
5    initCard();
6    initLeds();
7    initUsb();
8    initBus();
9
10   enableInterrupts();
11 }
```

Example 10: A generic design to the main initialization method.

The card specific implementation may also be more structured, but with the low level bitmasking there is not much that can be done, except commenting the code.

The functionality of the original code is preserved, but the readability is much improved. This is shown in example 11.

3.3.1.3. Extract method / Strategy

The strategy pattern can be seen in the specific implementation of the chain of responsibility. This pattern is used to be able to interchange the implementation of a method. This way something that is happening in a similar way can be handled by different classes.

3.3.1.3.1. Problem example

A common example is logging, which can be implemented as in example 12. In this example the logger can select a method to call for the logging

```
1  //LPC2388_init.c
2  ...
3  void initCard(){
4    PINSEL10 = 0;          // Disable ETM interface
5    FIO2DIR  = LEDMSK;     // LED's defined as Outputs
6
7    PINSEL3  &=   0xFFC0FFFF;      //Pin 1.24-P1.26=0 (16...21=0)  GPIO ↩
         Pins
8          PINMODE3 |=   0x003F0000;        //Pin 1.24-P1.26=0 (16...21=0) ↩
               pull down resistor an
9          IODIR1  |= 0x07000000;
10 }
11
12 void initLeds(){
13   int timeout = 65536; //Small delay to make all leds blink on init
14
15         //Initializing leds turns led on
16   PINSEL4  &= ~(MASK_PINSEL_LED_GRN1 | MASK_PINSEL_LED_GRN2 |↩
         MASK_PINSEL_LED_RED1 |MASK_PINSEL_LED_RED2 | MASK_PINSEL_LED_YEL)↩
         ;
17   FIO2DIR  |=  (MASK_LED_GRN1_DIR    | MASK_LED_GRN2_DIR    |↩
         MASK_LED_RED1_DIR    |MASK_LED_RED2_DIR | MASK_LED_YEL_DIR);
18   FIO2MASK &= ~(MASK_LED_GRN1_DIR    | MASK_LED_GRN2_DIR    |↩
         MASK_LED_RED1_DIR    |MASK_LED_RED2_DIR | MASK_LED_YEL_DIR);
19
20   while(timeout != 0)timeout--;
21
22   //Turn all leds off
23   LedOn(0, 0);
24   LedOn(1, 0);
25   LedOn(2, 0);
26   LedOn(3, 0);
27   LedOn(4, 0);
28 }
29 ...
```

Example 11: A generic design to the main initialization method.

purpose. This approach is not so pretty, as code is commented in the file.

In the CAN code an attempt to log in the main file is made. The implementation of this attempt can be viewed in example 13. In this case the logging is not even in a method, changing the type of logging would therefore be very time consuming.

```
1 //example.c
2 void addData(int data){
3         AddDataToList(data);
4     FileLogger(data);
5     //NetworkLogger(data);
6 }
```

Example 12: One example of how to implement logging.

```
1         //logging ?
2 //      if(logging_is_on >2) {          //==1 -> ausgeschaltet
3         if(logging_is_on >0) {          //angeschaltet
4                 dp=set_log_poi();
5                 if(dp==0) goto rx1_exit;
6                 ...
7         ...
```

Example 13: The original way of logging.

3.3.1.3.2. Solution

The solution, as shown in example 14, could be use to do logging in a development environment. When switching to production, logging is turned off automatically. When all the code has been disconnected from the other parts and put into interfaces the existing example code could be simplified, as seen in example 15. This way the developer could always log interesting data, without worrying about removing the code when releasing to the customer. It also makes it simple to add a new method for logging.

3.3.1.4. Extract global variable

How to use global variables can be debated in several ways. Looking at it from a performance perspective it can almost always be the correct path. From a design or readability perspective it almost always makes the code more difficult to understand.

When attempting to extract a global variable it does not necessarily

```
//logger.h
#ifdef production
        #include "NoLog.h"
#else
        #include "FileLog.h"
#endif

void log(int data);
```

```
//NoLog.c
void Log(int data){return;}

//FileLog.c
void Log(int data){
        //Perform logging
        ...
}
```

Example 14: An implementation of logging in a development environment.

```
//example.c
void addData(int data){
        AddDataToList(data);
        Log(data)
}
```

Example 15: A simplified version of the original code in example 12.

mean removing the global variable from the code. In order to preserve performance and improve readability, the method using the global variable should receive all used variables as parameters. By doing so the global variable can still be used, by calling the method with the global variable as a parameter. This approach is similar to dependency injection[9] when using classes.

By injecting the variable instead of always using the same, the code can be tested with mock objects. It also makes it clearer when running the call

what is going to happen.

3.3.1.4.1. Problem example

```
1  //cdcuser.c
2  void CDC_BulkIn(void) {
3    int tmpw=0;
4          //wenn Daten zum senden da und
5          if(outUSBcnt >0 && CDC_DepInEmpty == 1) {
6                  if(outUSBcnt>64) {
7                          tmpw=64;
8                          leersend =0;
9                  }
10                 else {
11                         tmpw=outUSBcnt;
12                         if (tmpw==64) leersend =1;
13                         else leersend =0;
14                 }
15                 CDC_DepInEmpty =0;
16         }
17   /* send over USB */
18   if (tmpw > 0) {
19         USB_WriteEP (CDC_DEP_IN, outUSBpoi, tmpw);
20         outUSBpoi += tmpw;
21         outUSBcnt -= tmpw;
22         if(outUSBcnt==0) outUSBpoi = &outUSB[0];            //↩
               ausgangspoi üzurcksetzen
23   }
24   else {
25         if(leersend > 0) {
26                 leersend=0;
27                 USB_WriteEP (CDC_DEP_IN, outUSBpoi, 0);
28         }
29     CDC_DepInEmpty = 1;
30   }
31 }
```

Example 16: The original code for sending data over Serial/USB to the connected PC.

In example 16, `CDC_BulkIn` does a lot of things. The main goal is to send data over serial/USB to the connected PC. The data to send is preselected and always in the same buffer. Some other global variables like `outUSBcnt` and `CDC_DepInEmpty` are also used to control if data should be sent or not.

In this case `outUSBcnt` could have been checked before calling the method, while `CDC_DepInEmpty` is a variable set by the system that declares that the built in functions are ready to send. That kind of global variable cannot be moved and it also does fill a valid purpose.

3.3.1.4.2. Solution

```
1  //LPC2388_Usb.c
2  int WriteUsb (char *buffer, const int *length){
3    if (length > 0) {
4      if (CDC_DepInEmpty) {
5        CDC_DepInEmpty = 0;
6        return USB_WriteEP (CDC_DEP_IN, (BYTE*)buffer, *length);
7      }
8    }
9    return 0;
10 }
```

Example 17: A simplified and more clear implementation of how to send data over Serial/USB to the connected PC.

As shown in example 17, by refactoring and placing the low level handling of large telegrams in a lower level method the `WriteUSB` method can be called with buffers. One could still use the globally declared buffer but it is a lot clearer what is going on. The code has also lost dependencies to some variables. Looking at the generic implementation some code could be reused on both platforms.

3.3.2. Dependency breaking

By applying the methods mentioned above a lot of dependencies can be broken and a lot of code made more generic.

One of the goals with breaking dependencies is to improve testing. Use of global variables and dependencies within the code makes testing certain parts very difficult as there is no entry point that can access the piece of code in need of testing. When dependencies are successfully broken a single piece of code can easily be tested, such as blinking the leds or simulating

a PROFIBUS datagram.

When testing, it is important to be able to use mock or simulated data. This way a known input should always produce a known output. If the code is built so that a method cannot be called with parameters as global parameters are used, it becomes difficult to test with mock objects.

3.3.3. Generic driver patterns

The most important pattern in order to build the generic driver is the `Extract method` pattern. As the goal is to have different implementations with the same method call this fits very well with the design of the generic driver.

If there is some generic property, like led control, it should be called in the same way on both platforms. That is the goal of the strategy pattern.

The `chain of responsibility` patterns is more targeted at cleaning up the code and getting a ”red thread” to follow when examining the code. Therefore chain of responsibility will be used more to improve readability, and not building the generic driver.

Another powerful pattern is the `Decorator` pattern. As seen in example 7, the decorator pattern combined with the strategy pattern can allow one to refactor into a totally generic driver.

4 | Result

The described methods was applied to the legacy driver with some success. A generic framework was built for both PROFIBUS and CAN. The generic framework only covers communication with the hardware part of the both modules.

As in the design from figure 3.4 the framework is rather high-level in the generic methods. The module specific implementation is done in separate projects for each module.

4.1. Framework demo

As a result a simple demo which did the following could be built with a few lines of code:

- Read data from USB serial
- If data was received, blink led
- Send the received data back over USB serial
- Send a sample CAN telegram

This program covers most of the hardware functionality and uses a mix of interrupts driven event handling and polling. The main method is called using the generic framework after all the initialization has been successfully run. The module specific main code for the program can be found in example 18. Most of the code (line 16 - 26) is just to keep the led on long enough when data is transmitted / received. The demo shows that the concept is viable and that using a more generic approach to programming is possible at this level.

```
1  void mainLoop(){
2    int receivedChars;
3    int ledTimer = 0;
4    int buffer = 32;
5
6    //The buffer used for reading and writing, alla data
7    //written to the buffer is read back to the USB
8    char readBuffer[32] = {0};
9    UBYTE test[8] = {0};
10
11   while(1){
12     receivedChars = ReadUsb(readBuffer, &buffer);
13     WriteUsb(readBuffer, &receivedChars);
14     sendTelegram(123, 8, test, 2, 0,0);
15
16     if(ledTimer == -1 && receivedChars > 0){
17       ledTimer = 100;
18     }
19
20     if(ledTimer == 100){
21       SetLed(YellowLed, LedOn);
22       ledTimer--;
23     } else if(ledTimer > 0){
24       ledTimer--;
25     } else if(ledTimer == 0){
26       SetLed(YellowLed, LedOff);
27       ledTimer--;
28     }
29   }
30 }
```

Example 18: The main loop if the generic framework for the Beijer Electronics CAN module.

4.2. Fieldbus research

The research, as well as the produced framework, indicates that it is indeed possible to create a more generic solution for the protocols. The telegram handling is not very similar, as seen in table 4.1. For example the channel access in CAN is done according to priority of the ID, whilst in PROFIBUS the master polls slaves for status and reacts to their replies. There is a fundamental and programmatical difference between the two. As such the telegram handling part should not be made generic, as the resulting

solution would become very convoluted.

	PROFIBUS	CAN
Network	Master/Slave	Multi Master
Communication	Cyclic	Event driven
Channel Access	Token	Priority

Table 4.1.: A comparison of some properties of the CAN and PROFIBUS protocols.

4.3. Feasibility of generic driver engine

Instead an approach where the data is collected by a non generic part and stored in a more generic way is proposed. The storage of data could definitely be made generic. The data could be stored in a database, like MySQL or similar. There could also be a custom database implementation used on several platforms. Combining this with a new generic iX driver could enable a much smaller driver code base with a common syntax for the different commands.

Interfacing with the hardware is made generic, as seen in example 18. By making it easier to understand how the data on the bus was read and how data was sent much of the code could be refactored.

With this implementation almost the whole chain could be made generic, except for one part. With the refactoring methods in mind and by using the other parts described there is a chance that some of the telegram handling can be made generic.

4.4. Unmet goals and change of scope

The goals to build a fully functional generic driver and integrate it into the Beijer Electronics device firmware were not met. The scope of the project was decreased and a new goal to build a generic framework was introduced and achieved.

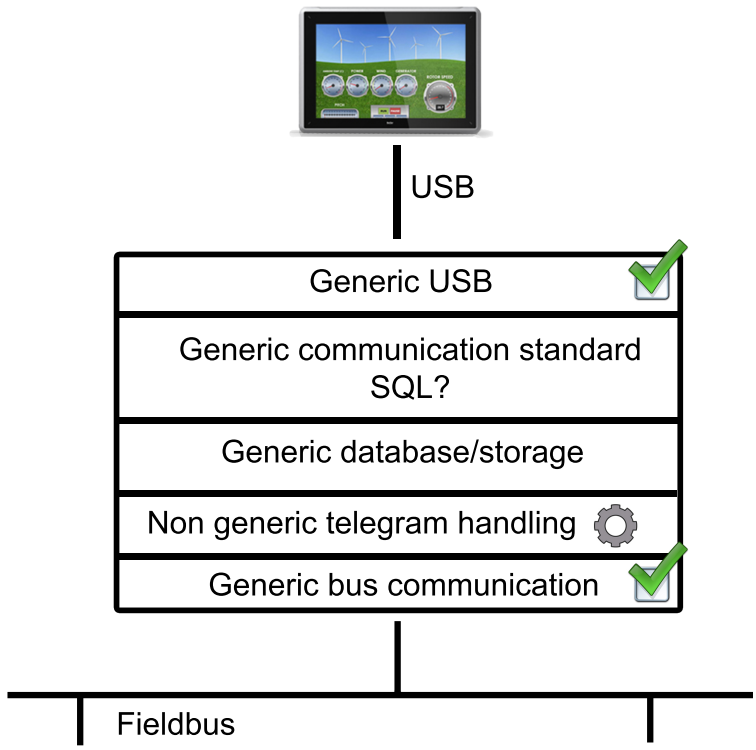

Figure 4.1.: An overview of the implemented features and the feature possibilities.

4.5. Beijer Electronics code documentation

As a bonus for Beijer Electronics, a lot of documentation and investigation was done on the legacy code. Documents on how to set up the IDE, test and a lot of other information were created. Some of the documentation is available in the appendix of this thesis.

5 | Discussion and conclusions

During this project many problems and possibilities presented themselves and several decisions were made. These led to the result presented in chapter 4. In this chapter a critical discussion will be held regarding these factors and how they impacted the result of the project.

5.1. Methods

In this section the methods used, as explained in chapter 3, as well as obstacles overcome in order to complete the project, will be discussed.

5.1.1. Software

The licensing of µVision was the first issue encountered. The PROFIBUS project used the compiler and license from the free KEIL PK51 Developer's Kit discussed in appendix A. This compiler could not be used in the CAN project and hence a real license was required. With a trial license there was a size limit on projects allowed to be compiled. This license was, just as the hardware, used at the german office. In order to get it the german developer had to be contacted. The license is bound to one specific PC at a time. The german developer had to detach the license from his instance of µVision and send the license key. µVision would now allow a new PC to be specified for the license. The issue here was mainly that the german developer needed the license for some time in his work before he could transfer it to Malmö. This meant delays for us.

The fact that the original PROFIBUS project was developed in Silicon Laboratories presented a problem. In order for development to run smoothly a common IDE was required and migrating to KEIL µVision 5

proved troublesome. As seen in appendix A, the process of migrating is not straightforward. Since little help was to be found online it took quite some time to crack the puzzle.

5.1.2. Hardware

Setting up the test environments and being able to debug and analyze the setup took some time. One issue was that the intended debugging hardware, the Saleae Logic 16, did not support the PROFIBUS DP protocol. To analyze PROFIBUS other tools were needed. Beijer had ProfiTrace tool set, which was located at the german office. The hardware is expensive and a very limited number of people within the company were working with it. Hence the company only owned one instance and it had to be sent to the office in Malmö as soon as possible.

Another hindrance was the fact that documentation for both the Beijer Electronics CAN- and DP module is almost nonexistent. This meant that there were no place to find information about key properties such as memory or registry addresses, clock frequencies and RAM size. This lead the work to a stop since Keil µVision requires some of these properties to be set in order for it to be able to flash the firmware of the module. In other words no progress or testing could be made without these settings being correct. This lead to some unplanned delays.

In order to solve this the developer at the german office, responsible for the modules' firmware, was invited to come to Malmö. During a few days the knowledge of the modules was transferred and documented. Some breakthroughs were made regarding USB communication between the modules and a PC.

5.1.3. Development

The development phase of the project contained its own collection of problems. The main issue slowing down progress was that the legacy code was both poorly documented and written, whilst at the same time much of the code used german function names, variable names and comments. Since there was a lack of design and the code was full of global variables it was extremely difficult to understand it. A lot of time had to be put into simply breaking it apart and, as a first milestone, only locating the key compo-

nents to control the led lights. During the entire course of the project the legacy code proved to be the single most time consuming element.

The issue of the legacy code was of course detected early on, but it affected the entire project's schedule greatly. The sheer amount, complexity and state of legacy code had been underestimated. Since there had been no opportunity to view the code before hand, all these factors had had to be estimated. And even though the scheduled time was greater than the estimation, it was still far below the actual time it would consume. As an effect of this, the scope of the project had to be altered to a more realistic one. The primary aim of the project became to design and implement a working generic framework for both the CAN and the PROFIBUS protocol. This replaced the two last goals, found in section 1.1.

Encountering this legacy code base also shifted the scope of the project. It became interesting to see if refactoring could be made in order to straighten up the existing code. Examining this presented many interesting opportunities.

5.1.4. Planning

Having a well thought through plan with milestones and a scheduled date for each task was a great help, even as the scope changed. It made it easy to know if the project was running on time and how much work needed to be done. Having planned well before hand was what made it possible to know that the scope needed to be change in order to finish on time.

A decision made early on was to split the work so that one would research and work with the CAN protocol and the other with the PROFIBUS protocol. This turned out to work well and the project progressed at a good pace thanks to it.

5.2. Result

The resulting generic framework for the fieldbus protocols is a clear step forward. The successful demo of functionality indicates that a fully functional implementation should be possible. The demo code also follows the main design goals, set in section 3.2.1, for the generic driver. The result does not, however, reach the goals set for the project, as seen in section 1.1. As mentioned in the previous section, the scope of the project was changed.

Instead of creating a generic driver engine and integrate it into the Beijer Electronics device firmware, the goal was to design and implement a generic framework for the driver. This goal was met.

As a result of the change of scope, the project managed to stay on schedule and finish one week ahead of it. This can be seen as a clear indicator that the decision to limit the scope was a correct one. If it had not been done the project would have clearly overshot the schedule and perhaps have had no reasonable results when the time was up.

In hindsight one could argue that, had it been known that a fully functional implementation could not be done on time, the time spent on researching the fieldbusses could be used on other things. This knowledge of CAN and PROFIBUS of course came to some use while debugging and using the hardware, but not to the extent which was expected. The fieldbusses played a big part in the project, but are not so apparent in the result. Of course this was difficult to know at the time. Since focus was on learning the legacy code, learning the protocols was thought to be needed. Also there was the issue of getting access to the legacy code. In the early stages of the project research was the only option available. When the code later became accessible, the research was already far gone and it seemed logical and important to continue and get a good grasp of the matter at hand. Perhaps, if time would not have been spent on the protocols, more time could have been spent on developing the generic driver.

Some minor tests were made to see that the performance of the code was acceptable. There were no errors or wrongful behavior made by the code in these tests. There was a possibility that the increase of function calls would impair the performance. This seems, however, to not be the case. The increase of functions could also allow the compiler more flexibility when optimizing the code for performance and size.

5.3. Future

Since the design of the framework follows the design goals, it allows added support for a new protocol or hardware platform in the future. The lack of global variables means programmers will have an easier time understanding and utilizing it. Following a code standard further increases the ease of reading and understanding the code. As a whole, the design is seen as a

success.

An effect of the design is increased maintainability. Since programmers have an easier time reading and understanding, as stated above, the code it is also easier to maintain. This saves a lot of work. Fixing a bug in the generic code will now fix it for all protocols at once, instead of having to do it for each protocol separately. With a large number of protocols, this saves a huge amount of time.

There are clear possibilities for the framework to be extended to more protocols and for each protocol to get a fully functional implementation. Having this generic framework is a crucial steppingstone to developing a generic driver engine.

A | Migrating from Silicon Laboratories to Keil

This appendix explains how to migrate a project for C8051F380 from Silicon Laboratories to Keil µVision 5. Since µVision does not recognise the C8051F380 chip from Silicon Laboratories, the first step is to get a Keil license and install support for the device in µVision. This is done by registering for a free license for Keil PK51 Developer's Kit[1]. This kit includes the compiler/linker/assembler for use with Silicon Laboratories 8-bit MCU. This does, however, install µVision 4 instead of 5. Luckily the license also works in µVision 5 so one can simply install µVision 5 and enter retreived LIC(License ID Code) as a new license in the License Management window found in the file-menu tab. In order to migrate the Silicon Labs 8-bit MCU support altering the programs TOOLS.INI file is required. This file is found in each programs respective program folder. The TOOLS.INI file is built up of blocks. From the block called [UV2], copy the line containing 'silabs.cdb'. Paste this line at the end of the same block in the TOOLS.INI file of µVision 5. Now copy and paste the entire [C51] block from the µVision 4 file to the µVision 5 file. Now µVision 5 has knowledge of the Silicon Labs 8-bit MCUs. A bonus feature of installing the Developers' Kit is that it includes support for the Silicon Labs USB Debug Adapter.

With µVision 5 installed and modified for use with the C8051F380, the next step is to setup the project. This step is explained in Appedix B.

[1] `http://pages.silabs.com/lp-keil-pk51.html`

B | Keil 5 and C8051F380

In order to build a project for the C8051F380 with Keil µVision some configuration is required. Table B.1 shows the required settings. Before applying these settings one is required to have followed the steps in Appendix A.

Device tab	Device	C8051F380
	Toolset	C51
Target tab	Xtal(MHz)	48.0
	Memory Model	Large: variables in XDATA
	Code Rom Size	Large: 64K program
Output tab	Create HEX File	Checked
	HEX Format	HEX-80
A51 tab	Include Paths	'Sil Labs folder'\MCU\INC
BL51 Locate tab	Code Range	0X0000-0XFFFF
	Xdata Range	0X0000-0X7FFF
Debug tab	Use	Silicon Labs C8051Fxxx Driver
Settings window	USB Debug Adapter	Checked
Utilities tab	Dropdown list	Silicon Labs C8051Fxxx Driver

Table B.1.: Keil µVision 5 settings used when programming for C8051F380.

After applying these settings to the project, Keil µVision 5 is ready to build for the C8051F380 platform.

C | Keil 5 and LPC2388

Setting up keil is not as straightforward as it could be. To develop and program the LPC2388 (and possibly others) installing two versions of keil is required. Installing MDK474.exe installs a newer version of keil, this version can communicate with the ULINK2 adapter. Installing MDK403a.exe and adding the licence code enables you to compile files >32 kB, this version cannot communicate with the ULINK.

It might be interesting to install the MDK79V514.exe since it adds legacy support including some code examples for LPC2388 etc. To make everything work install both products. This should create the two folders: `C:/Keil` and `C:/Keil_v5`

In the folder `C:/Keil_v5/ARM/BIN` the 2 relevant DLL files are located: UL2ARM.dll and ULP2ARM.dll. Copy these files to the folder `C:/Keil/ARM/BIN` and overwrite or save a backup of the originals, to enable communication with the ULINK2 adapter. To launch Keil run `C:/Keil/UV4/Uv4.exe`

To allow compiling large files a license have to be added. The system is built around that a PSN (Product Serial Number) is bought. From the PSN and a CID (Computer ID) a LIC (License ID Code) can be generated. The PSN is then bound to that computer/LIC. Uninstalling the LIC on the internet enables installing a new LIC on a new computer.

Bibliography

[1] Keil uvision. http://www.keil.com/uvision/. Accessed 2016-05-19.

[2] Silicon laboratories. https://www.silabs.com/products/mcu/Pages/8-bit-microcontroller-software.aspx. Accessed 2016-05-19.

[3] Usb debug adapter. https://www.silabs.com/products/mcu/Pages/USBDebug.aspx. Accessed 2016-05-19.

[4] Linux kernel coding style. https://www.kernel.org/doc/Documentation/CodingStyle, 2015. Accessed 2016-05-11.

[5] Marina Pavlova Alexander Shvets, Gerhard Frey. Sourcemaking. https://sourcemaking.com/, 2015. Accessed 2016-05-18.

[6] Axiomatic. What is can? http://www.axiomatic.com/whatiscan.pdf, 2006. Accessed 2016-05-06.

[7] Michael C. Feathers. *Working Effectively with Legacy Code*. 2008.

[8] Behrouz A. Forouzan. *Data Communications and Networking -4th ed.* 2007.

[9] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html. Accessed 2016-05-24.

[10] Kent Beck Martin Fowler. Code smells. https://sourcemaking.com/refactoring/bad-smells-in-code, 2015. Accessed 2016-05-18.

[11] NXP. Nxp lpc2388. `http://www.nxp.com/documents/data_sheet/LPC2388.pdf`, 2013. Accessed 2016-05-19.

[12] Fröstel (S.G.). A can base frame telegramm according to specification v2.0a. `https://commons.wikimedia.org/wiki/File:CAN_telegramm_2.0A.svg`, 2011. Accessed 2016-06-16.