# Analysis of PDCP Data on a Prosthetic Limb System

by

M. Dombrowski, Y. Losier, K. Kent, R. Herpers

**TR 12-216, June 1, 2012**

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

**Abstract**

The abstraction from bus systems greatly enhances the flexibility in designing modules for a network as different virtual channels can be created for communication between modules. The Prosthetic Device Communication Protocol (PDCP) is such a protocol. To be able to evaluate the messages that are passed using the PDCP additional tools are needed. This report shows the design, implementation, and testing of an evaluation tool for the PDCP which is an open protocol and is featured in the University of New Brunswick's most recent prosthetic limb research project, the UNB Hand System. This prosthetic device utilizes the CAN bus hardware with the PDCP for passing command and data messages between modules within the prosthetic limb system. To be able to analyze communication in the CAN layers as well as in the PDCP layer this report shows a solution utilizing an FPGA for CAN bus bandwidth load monitoring and a microcontroller for PDCP monitoring and analysis.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **ADC** | Analog to Digital Converter |
| **BA** | Bus Arbitrator |
| **CAN** | Controller Area Network |
| **CRC** | Cyclic Redundancy Check |
| **CSMA/CD** | Carrier Sense Multiple Access with Collision Detection |
| **DSP** | Digital Signal Processing |
| **FPGA** | Field Programmable Gate Array |
| **GND** | Ground |
| **GPIO** | General Purpose Input/Output |
| **HDL** | Hardware Description Language |
| **IDE** | Integrated Development Environment |
| **IEEE** | Institute of Electrical and Electronics Engineers |
| **IP** | Intellectual Property |
| **ISO** | International Organisation for Standardization |
| **ISR** | Interrupt Service Routine |
| **IVT** | Interrupt Vector Table |
| **I/O** | Input/Output |
| **LED** | Light Emitting Diode |
| **LCD** | Liquid Crystal Display |
| **MIPS** | Million Instructions per Second |
| **MC** | Microcontroller |

| | |
|---|---|
| **PDCP** | Prosthetic Device Communication Protocol |
| **PWM** | Pulse Width Modulation |
| **RAM** | Random Access Memory |
| **ROM** | Read-Only Memory |
| **RX** | Receive |
| **SFR** | Special Function Register |
| **SoC** | System on a Chip |
| **SPI** | Serial Peripheral Interface |
| **SRAM** | Static RAM |
| **TX** | Transmit |
| **UART** | Universal Asynchronous Receiver / Transmitter |
| **VHDL** | VHSIC Hardware Description Language |
| **VHSIC** | Very High Speed Integrated Circuit |

# 1 Introduction and Motivation

The University of New Brunswick's (UNB) Institute of Biomedical Engineering (IBME) is developing together with UNB's Applied Nanotechnology Laboratory, the Rehabilitation Institute of Chicago's BioMechatronics Development Laboratory, the Université de Moncton's Thin Films and Photonics Research Group, and Liberating Technologies Inc. a prosthetic hand system, termed the UNB Hand System. For the design of the hand not only the overall cost was relevant, but also the controllability and design modularity. Thus the UNB Hand is capable of recognizing different patterns, with an emphasis on the movements that are controllable by most amputees. As every user has their own requirements the UNB Hand is also modular to fit different sizes[1]. An image of the hand can be seen in Figure 1.1.

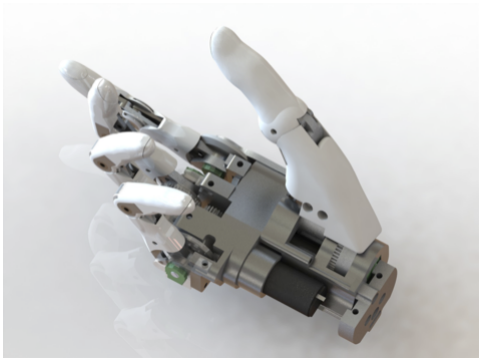

**Figure 1.1:** The UNB hand.

Inside the hand different sensors for measuring the pressure and motors for moving the thumb and fingers are working. These modules are connected via a CAN bus network (ISO OSI layer 1 and 2). On top of this bus system the Prosthetic Device Communication Protocol (PDCP) has been built[2]. It allows an abstraction of the underlying bus system to give engineers more flexibility when designing new components for the hand. The PDCP itself acts on ISO OSI layer 3 and is used for information exchange. The PDCP will be explained in Chapter 2.5. As both CAN buses are operating nearly at their bandwidth limit tools are needed to monitor the CAN bus. In order

to keep track of different nodes talking to each other high-level interpretation of these messages is needed as well.

During the implementation stage of additional modules for the UNB hand, tools are needed to analyze the data on the bus to ensure that messages are transmitted and received properly before these modules are used in a live system. Furthermore, in a live system the engineers need to be able to gather information on how the prosthetic device is used by different patients. Thus a tool is needed which is able to monitor and analyze the communication on the bus online and offline.

Chapter 2 introduces and explains the concepts of a microcontroller with the Microchip dsPIC33E family, which will be used for this project. Furthermore, this chapter introduces a daughterboard for this microcontroller family, the Altera DE0-nano board as an FPGA board, and it briefly explains the CAN bus as well as the PDCP. Chapter 3 shows different approaches on monitoring a bus system and gives the problem formulation for this project. The design of this project is explained in Chapter 4, with the implementation details in Chapter 5. The project is evaluated in Chapter 6 and in Chapter 7 the project will be summarized and an outlook on future work is given.

# 2 Basics

This chapter splits the necessary background information in order to understand the remaining chapters of this report. It will start with hardware basics in microcontrollers, then the Microchip Multimedia Expansion Board as a daughterboard, and the Altera DE0-nano board as an FPGA board. After that details about the CAN bus, the UART, and the PDCP are be given.

## 2.1 Microcontroller

To be able to explain the concept of a microcontroller a short introduction is given first. After that, the Microchip dsPIC33E family microcontrollers are introduced. As every microcontroller has Special Function Registers, interrupts, and timers, these concepts are explained as well.

### 2.1.1 Introduction

A microcontroller (MC) consists of a processor with different pins attached to it. Each pin can be configured to have different functionality. This is achieved by wiring each pin to other modules. These modules can include a liquid crystal display (LCD), different bus systems, such as the Controller Area Network (CAN) or Inter-integrated Circuit (I2C), or light emitting diodes (LEDs). Each microcontroller has furthermore its own memory. Contrary to a system on a chip (SoC) MCs usually have less memory and are single chip solutions. MCs can include digital signal processing (DSP) functionality, pulse width modulation (PWM) modules, interrupts, and timers. To be able to address these functions special function registers (SFRs) are needed. The functionality of SFRs will be introduced in Subsection 2.1.3. Interrupts and timers will be introduced in Subsection 2.1.4.

Microcontrollers are usually found in embedded systems. These systems serve a specific purpose. The term embedded implies that it is part of a bigger system, e.g. in a washing machine[3]. For academic and education purposes embedded systems can also consist of different parts. The microcontroller is usually found on a motherboard, which most of the time

feature beside the MC either a JTAG interface or a USB connector for programming the microcontroller. Furthermore these motherboards feature expansion connectors, so that it is possible to connect daughterboards with extended functionality to it. Subsection 2.1.2 introduces the Microchip dsPIC33E USB Starter kit as a motherboard and Section 2.2 introduces the Microchip Multimedia Expansion Board as a daughterboard. These two boards are also the boards that have been used during the development of this project.

### 2.1.2 dsPIC33E USB Starter Kit

The Microchip dsPIC33E USB Starter Kit features the 16 bit Microchip dsPIC33EP512MU810 microcontroller, which is capable of performing a maximum of 60 million instructions per second (MIPS) and can operate at a frequency of 120 MHz. The architecture of this CPU uses a 24 bit wide program space and a 16 bit wide data space and is a modified Harvard architecture, which means that data can also be present in the program space[4].

The block diagram for this CPU can be seen in Figure 2.1. The CPU has a C compiler optimized instruction set which allows writing of performant code in the high level programming language C. Onboard are seven PWM generators and each generator can output two PWM signals, which for instance can be used to drive motors. Furthermore the MC features two independent analog to digital converter (ADC) modules, nine 16 bit timers/counters, a hardware real-time clock, and a peripheral pin select. The latter one can be used to remap ports to be used for different functionality. An image of the dsPIC33E Starter kit can be seen in Figure 2.2.

### 2.1.3 Special Function Registers

A Special Function Register (SFR) is a register inside a microcontroller that can have various purposes, which include ports, timers, and interrupts. SFRs are mapped to different addresses inside a microcontroller and are usually defined by the microcontroller vendor and are different for each manufacturer and might also be different for different product lines. The vendor usually

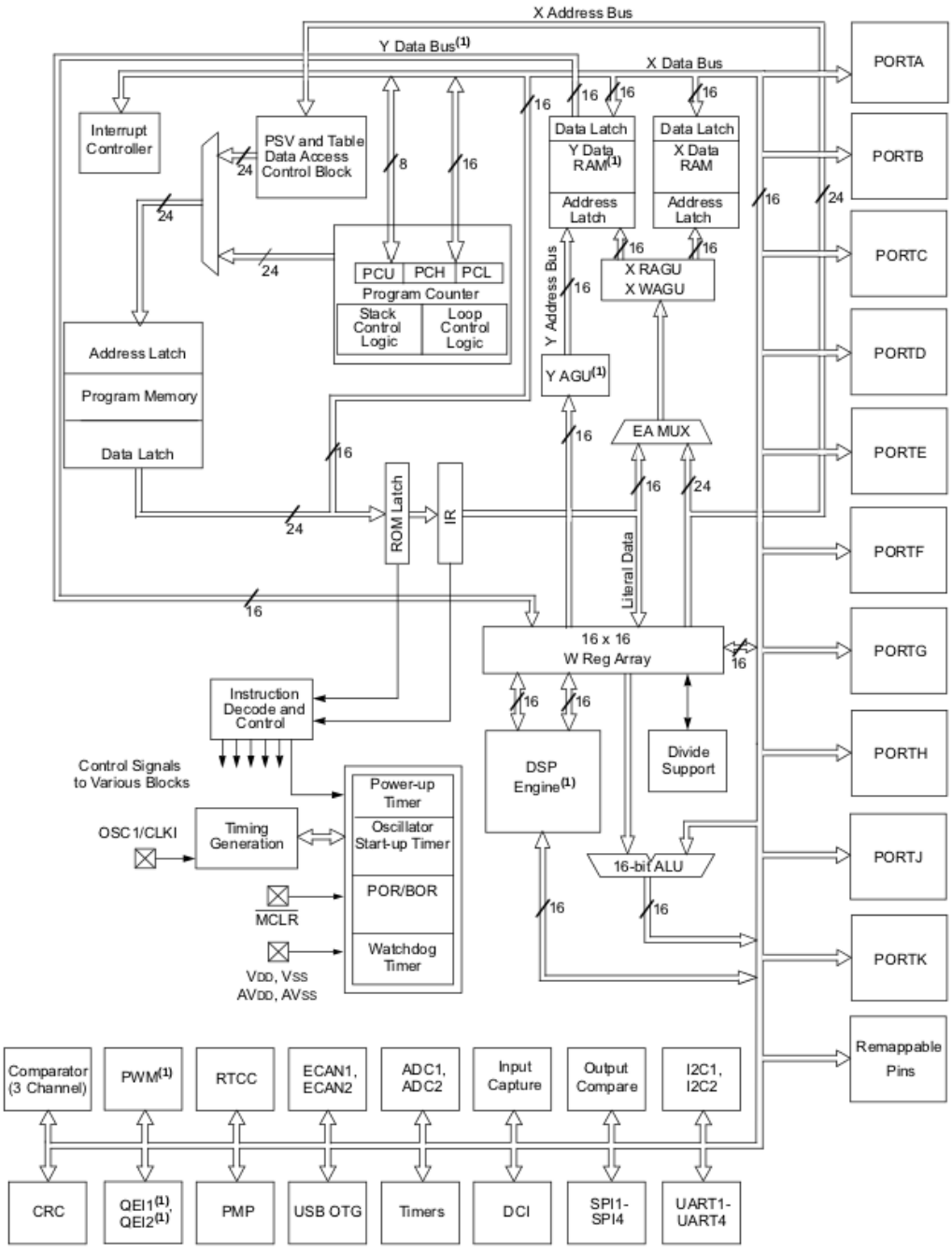


**Figure 2.1:** The CPU block diagram for the dsPIC33EP512MU810[4].

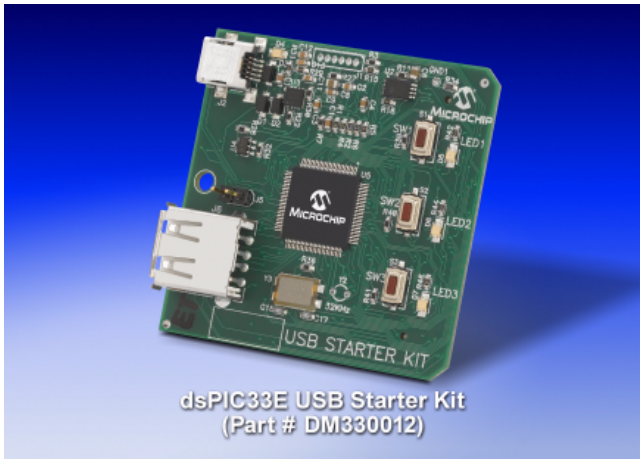


**Figure 2.2:** The Microchip dsPIC33E USB Starter Kit[5].

provides the necessary include files for addressing these SFRs. When programming in a high level language, e.g. in C, the developer needs to include the corresponding header file for their project in order to address the SFRs. SFRs can be either input ports (read only), output ports (write only), or inout ports (read and write).

When addressing a port on the Microchip dsPIC33E family (e.g. the Microchip dsPIC33EP512MU810 which is used for this project) the developer can address the port as shown in Figure 2.3. With the tristate switch `TRIS` the data direction can be specified, with 1 being input and 0 being output, `PORT` allows the reading and writing of the current port bit, the latch `LAT` has in output configuration the same functionality as `PORT` but for input configuration the bit in the latch will be read. Some ports also have an open-drain control `ODC` for generating voltages higher than 5V on a 5V tolerant pin by using pull-up resistors and some can be set to receive analog signals with `ANSEL` to be able to receive values between $V_{OH}$ and $V_{OL}$. The address mapping of the SFRs on the Microchip dsPIC33EP512MU810 MC can be seen in Figure 2.4.

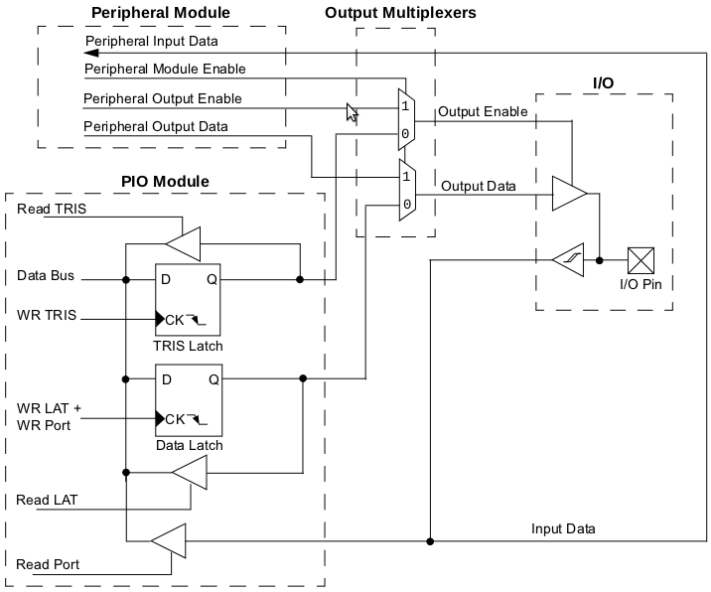


**Figure 2.3:** The shared port structure on the Microchip dsPIC33E family[4].

### 2.1.4 Interrupts and Timers

A microcontroller usually does not run an operating system. Thus the only program that is running is the user's program, starting from the main method. To be able to catch external (e.g. flags from other hardware modules) or internal (e.g. timer) events the flow of the program needs to be interrupted and the occured event needs to be handled. This is achieved using interrupts. The developer needs to declare every interrupt that needs to be handled. Every interrupt has a so-called interrupt service routine (ISR) which will be called when the interrupt occurs. After successfully handling the ISR the main program will be resumed. Most microcontrollers use for the interrupts special registers which hold the corresponding interrupt flags. This list is also called the interrupt vector table (IVT). After handling the ISR for a specific interrupt the flag needs to be unset to allow the interrupt event to occur again. A compact overview of the IVT of the Microchip dsPIC33EP512MU810 can be found in the Appendix A.2. The full IVT can usually be found in the data sheet of the microcontroller.

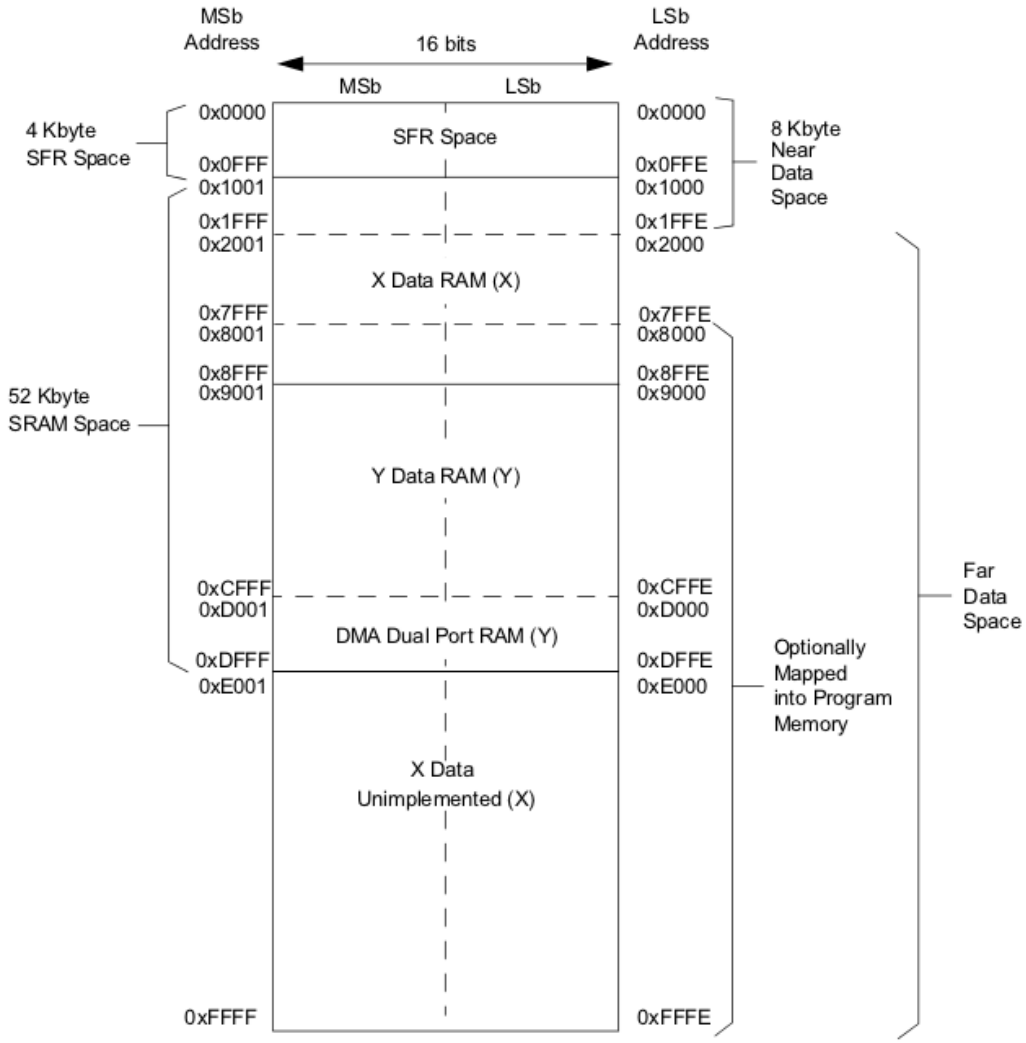


**Figure 2.4:** The data memory mapping of the special function registers on the Microchip dsPIC33EP512MU810[4].

Timers are used for either waiting for a specific amount of time or to count the occurences of a specific event, which can also be an external event. The first purpose is actually called the timer, while the latter one is referred to as a counter. Timers are bound to a clock and can either be driven by an external clock input or also by the device internal clock[4]. When in timer mode a specific amount of clock ticks is counted before the interrupt flag will be raised. This results for a 16 bit timer in a maximum of 65,535 clock ticks that can be counted. This would mean for a clock frequency of 120 MHz the timer would be toggled 1,831.05 times a second when counting to 65,535 clock ticks. Furthermore it is possible to have a prescaler for a timer. A prescaler divides the clock so that it is possible to count a multiple of the available clock ticks. E.g. a 16 bit timer with a prescaler of 1:256 running in a microcontroller with a clock frequency of 120 MHz would result in the interrupt flag being toggled 7.15 times per second. Figure 2.5 shows the block diagram for a timer in the Microchip dsPIC33E family.

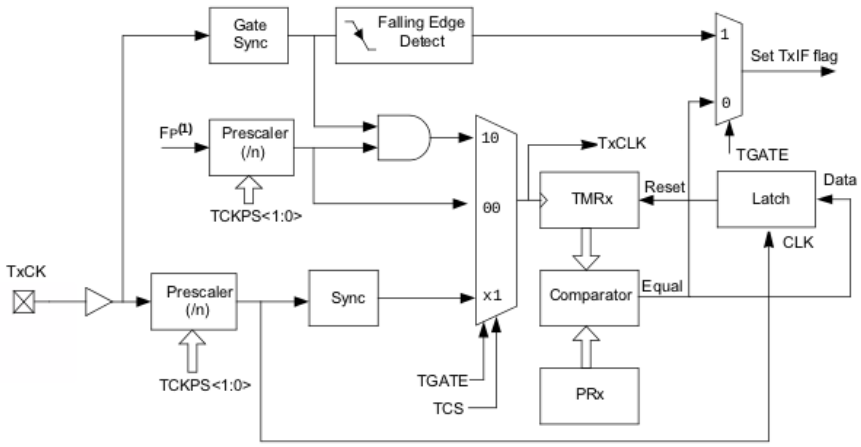


**Figure 2.5:** The timer block diagram for the dsPIC33E family[4].

## 2.2 Multimedia Expansion Board

The Microchip Multimedia Expansion Board (MEB) is a daughterboard for the Microchip PIC32, dsPIC33E, or PIC24E Starter Kits. It is powered by

the motherboard and features besides a LCD with touchscreen capabilites different LEDs, sound input and output jacks, a microSD card connector, an accelerometer, and a WiFi module. An image of the MEB can be seen in Figure 2.6.

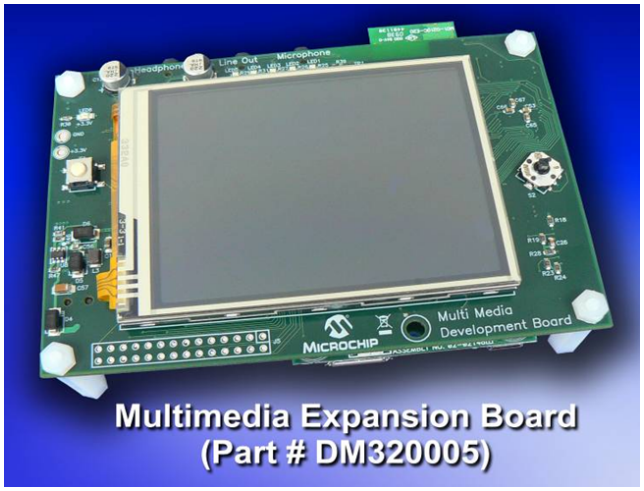


**Figure 2.6:** The Microchip Multimedia Expansion Board[6].

To be able to address the LCD or the microSD card, communication with the onboard Solomon SSD1926 graphics controller is required. The communication is done via either a 8 bit or 16 bit wide bus which is dependant on the starter kit that is used. For the dsPIC33E Starter Kit only the 8 bit wide bus can be used. Before the microSD card can be accessed via the Solomon SSD1926 it needs to be configured with the onboard CPLD. The CPLD stores the information for wiring the different components. It allows the usage of either a serial peripheral interface (SPI) bus[7], WiFi[8], or ZigBee[9]. For using the microSD card the CPLD needs to be configured to work on SPI.

## 2.3 Altera DE0-nano Board

The Altera DE0-nano board features a Cyclone IV EP4C22 FPGA with 22,320 logic gates. It operates at a clock frequency of 50 MHz and features 32 MB SDRAM, as well as 2 kB EEPROM[10]. Furthermore it has 2 general

purpose input/output (GPIO) expansion headers, an eight channel 12 bit resolution ADC and an 13 bit resolution accelerometer. An image of the Altera DE0-nano board can be seen in Figure 2.7.

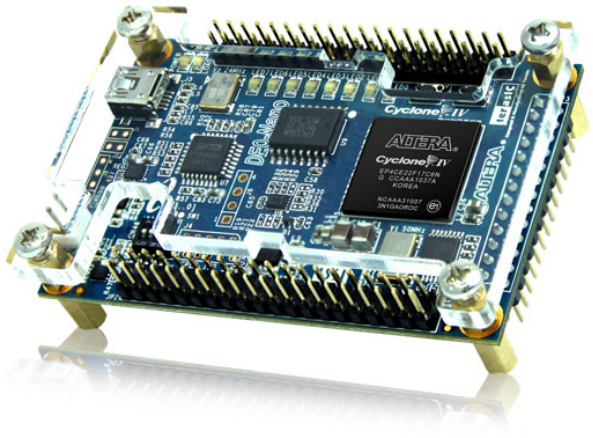

**Figure 2.7:** The Altera DE0-nano Board[10].

As with most Altera development boards the DE0-nano board can be programmed via a USB blaster interface. Via this USB connection the DE0-nano board also receives its power. All of the input, output, and inout pins can be configured to run at different voltages: 1.2V, 1.5V, 1.8V, 2.5V, 3.0V, and 3.3V.

## 2.4 CAN Bus and UART

As mentioned in [11] the CAN bus is operating on layers 1 and 2 of the ISO OSI Reference Model, as seen in Figure 2.8. The specification itself is not limited to a specific physical medium, so the engineers can decide for themselves what type of medium they want to use.

For collision detection the CAN bus utilizes the CSMA/CD protocol[13], which is also used by the IEEE 802.3 standard[14]. The transmission of bits on the bus is done by using dominant and recessive bits, where a logical 1 is recessive and a logical 0 dominant. On an idle bus there is always a logical

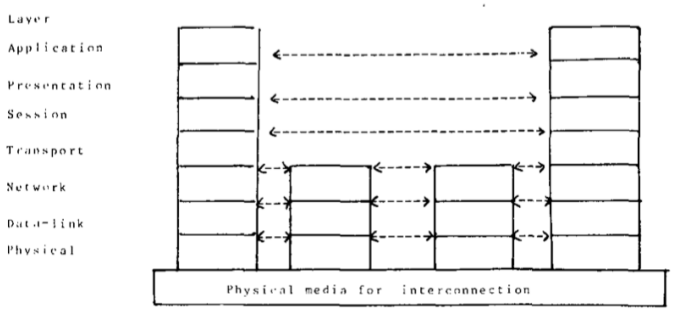


**Figure 2.8:** The ISO OSI Reference Model[12].

1. This is recessive because if a node tries to send it has to write a logical 0 on the bus. Looking at combining the idle logical 1 with the logical 0 that has to be written the AND operation is used.

There are different frames possible to be sent over the CAN bus: data frame, remote frame, error frame, and overload frame. For this project it suffices to only look at the data frames. The full specification of this frame can be found in [15] or in [13]. When creating a data frame the message itself contains also an error checking method for the bits in the data field. This method is called the CRC-15-CAN. The polynomial for calculating this sum is:

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1 \tag{2.1}$$

In hexadecimal representation this polynomial is `0x4599`. Bosch specified in [15] the pseudocode for calculating this polynomial using shift registers:

```
1 CRC_RG = 0; // initialize shift register
2 REPEAT
3   CRCNXT = NXTBIT EXOR CRC_RG(14);
4   CRC_RG(14:1) = CRC_RG(13:0); // shift left by
5   CRC_RG(0) = 0; // 1 position
6   IF CRCNXT THEN
7     CRC_RG(14:0) = CRC_RG(14:0) EXOR (4599hex);
8   ENDIF
9 UNTIL (CRC SEQUENCE starts or there is an ERROR condition)
```

`CRC_RG` is the 15 bit shift register, `CRCNXT` a temporary variable, `NXTBIT` the next bit of the data field of the CAN data frame, and `EXOR` the exclusive-OR operation.

Chu described in [16] the Universal Asynchronous Receiver and Transmitter (UART) as "a circuit that sends parallel data through a serial line. UARTs are frequently used in conjunction with the EIA (Electronic Industries Alliance) RS-232 standard, which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment". There are different baud rates available for UART and the configuration of a UART can be set to using a parity bit, or to one or two stop bits. Every UART message has 8 bits of data and as there is no clock on the UART line both the receiver and the transmitter have to agree on a clock before starting the communication. Figure 2.9 shows the transmission of a byte via UART.

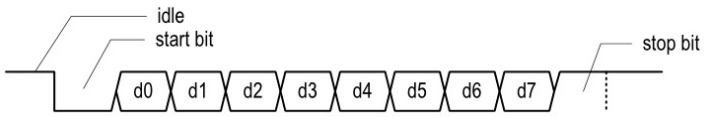


**Figure 2.9:** Transmission of a byte via UART.

## 2.5 PDCP

The Prosthetic Device Communication Protocol (PDCP) was developed by UNB and is set on layer 3 of the ISO OSI Reference Model, which is shown in Figure 2.8. Initially designed for abstraction of the lower 2 layers the PDCP enables the design of a device not to be bound to its underlying system. Thus, once configured, any device connected to the PDCP must be able to store its configuration, so that reconfiguration of the device at a later point in time is easier.

To be able to achieve this abstraction the PDCP uses a Bus Arbitrator

(BA) to assign every connected node a node ID. The BA is also the first device that needs to be connected to the bus, as all other nodes try to ask the BA for a node ID during startup. To be able for the BA to assign a node an ID each node needs to be uniquely identifiable. This is done using the devices vendor ID, product ID, and serial number. During a bind request the requesting node transmits those IDs and the BA then assigns this node the next available node ID if the BA does not recognize the calling node, or if recognized it transmits the previous assigned node ID to the node. Before initialization the topology of the setup can be as shown in Figure 2.10. Once configured the topology can be as shown in Figure 2.11.

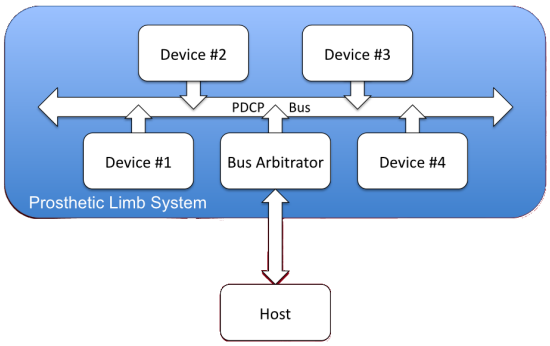


**Figure 2.10:** The underlying architecture before initialization.

Figure 2.11 also shows that nodes can have multiple inputs and outputs. Each node who wants to communicate with another node on this PDCP can ask the BA for a so called data channel link to another node. Every data channel link receives its own ID, making it easier for the sending and receiving devices to communicate with each other. This means, that e.g. a node with ID `0x02` and a node with ID `0x03` may have a data channel link with the ID `0xA0`. If node `0x02` wants to send data to `0x03` then it wraps a CAN message with the ID `0xA0` to directly communicate with node `0x03`. This has the advantage that node `0x03` directly knows where the message is coming from.

When wrapping the CAN message the 11 bit Standard Identifier field is seg-

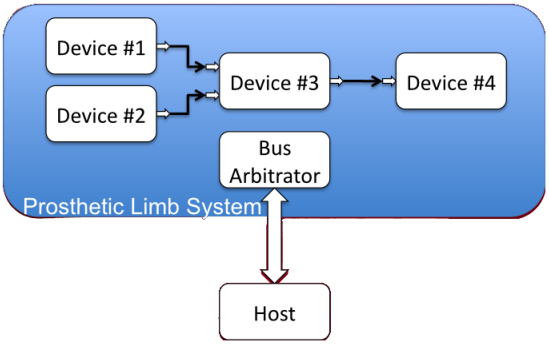


**Figure 2.11:** A possible topology after initialization.

mented into three parts: the first two most significant bits for the message priority, one bit for the message mode and eight bits for the node ID. Figure 2.12 shows this. The message priority is either set to $00_2$ for high priority, $01_2$ for normal priority, $10_2$ for low priority or $11_2$ for bind requests. The message mode is always $1_2$ for the BA and $0_2$ for all other devices. The full table of the available commands in PDCP can be found in the Appendix A.1. The full specification of how these commands have to be used can be found in [17].

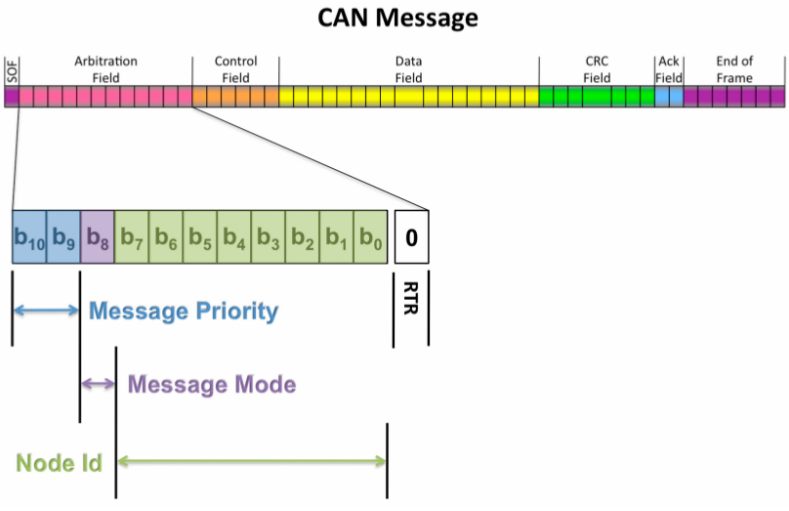


**Figure 2.12:** The segmentation of the Arbitration Field in PDCP.

# 3 Related Work

The field of this project is bus monitoring using hardware devices such as FPGAs and microcontrollers. As there are other existing solutions available, these solutions need to be introduced as well. Furthermore, a problem formulation is given in order to show why the existing solutions are not sufficient.

## 3.1 Bus Monitoring

Much research has been conducted in the field of bus monitoring, and not only limited to the CAN bus. Bochem et al showed in [18] an approach for monitoring the CAN bus using an Altera DE2 development board with two different CAN controllers connected to it. This design has been done in Verilog using different IP cores.

Kashif et al showed in [19] an implementation of a CAN bus analyzer using Verilog on Spartan 3E and Vertex 2 Pro FPGAs. They implemented an 8051 microcontroller with external RAM and two Philips SJA1000 standalone CAN controllers. Furthermore they wanted to be able to inject further data on the bus with their solution.

Li et al showed in [20] a design for monitoring the CAN bus as well. They utilized for their solution two PCs and a "USB-CAN smart card" [20]. This embedded device interfaces with two nodes to the CAN bus. Incoming or outgoing messages are processed by an onboard microcontroller and then sent to the bus or to the PC, respectively.

Another FPGA based softcore processor implementation on monitoring the CAN bus has been done by Mostafa et al in [21]. Their system could be controlled and configured using a RS232 compatible UART block. Furthermore, they also provided the ability to inject errors onto the bus.

Using a CPLD to monitor the CAN bus Yang et al showed in [22] their approach to monitor the CAN bus. The CPLD is used for logic control

and for the redundancy strategy, which interfaces to the CAN bus with two transceivers. These formatted messages are then passed to an ARM microcontroller which does further processing of the data. As they are using their approach on a space robot arm the whole embedded block is shielded against electromagnetic waves and ions[22]. For their implementation they used VHDL.

Other approaches using an FPGA to monitor a system have been shown by Mendoza-Jasso et al in [23] and by Zamantzas et al in [24] who were monitoring the Large Hadron Collider's (LHC) beam loss in CERN in real-time. As the PDCP is currently only being used in the UNB hand there is no related work on monitoring this protocol.

The main difference between this project's solution and the shown solutions is that this approach is able to monitor the bus load on the CAN bus in real-time with the ability to monitor different modules simultaneously, creating only bus load statistics for these nodes. Furthermore, this solution is able to interpret the messages according to a protocol that is built atop the CAN bus and is also able to configure different parameters depending on the devices that are on the bus.

## 3.2 Problem Formulation

There are already existing solutions to monitor the CAN bus using a microcontroller as shown in Chapter 3.1. In [25] a solution has been shown to monitor the CAN bus using a FPGA. All of the solutions are only monitoring the first two layers of the ISO OSI reference model. The PDCP as introduced in Section 2.11 sets atop of the CAN layer and needs its own monitoring as it provides transparency for the underlying bus system. As it is possible to create data and channel links a solution is needed that monitors these specific channels. Thus it is needed to start listening on the bus before other nodes register, so that the solution can identify the modules with their corresponding links.

The PDCP furthermore specifies different functions which also need to be logged and analyzed. Therefore an implementation of the full PDCP is needed as well. The proposed solution will feature a microcontroller that is connected to the CAN bus with the ability to receive CAN messages and interpret these messages according to the PDCP. Furthermore the MC will receive the bus load data from a FPGA that is connected via UART to the MC. With the PDCP messages and the bus load data the MC will create usage statistics of the system. The solution will also be capable of reading a user-specified configuration from a microSD card. All output data files will be stored in a human-readable way on the microSD card as well.

# 4 Design

To be able to specify the design for this project at first the requirements need to be formulated. With these requirements it is then possible to make decisions for the hardware design. After that it is possible to decide on the software to use.

## 4.1 Requirements

For this project the requirements are as follows:

- Usage of a microcontroller and an FPGA board.
- The microcontroller code has to be programmed in C.
- The code from the previous project needs to be ported to a smaller FPGA board.
- The FPGA board should send the CAN bus load data to the microcontroller via UART.
- The microcontroller should be able to read and write on the CAN bus.
- The PDCP should be implemented in the microcontroller.
- Output files should be generated for the analysis of the PDCP and CAN bus.

In the previous project[11] the CAN bus monitor has been developed on the Altera DE2 Development and Education Board[26]. As this board is too large for everyday use, it has too many unused functionality on the board itself, the project needs to be ported to a smaller FPGA board. For the analysis of the PDCP and CAN bus data a microcontroller needs to be used and it should be programmed in C. This has the advantage that C is a high level language and can be more easily extended than other microcontroller languages, such as Assembler.

The FPGA board itself should send the bus load data according to the protocol specified in [11] to the microcontroller. Thus the microcontroller

needs to follow the same protocol. Furthermore it should be able to read and write to the CAN bus. When being connected to the PDCP setup the microcontroller needs to register itself with the BA. Therefore a write method for the CAN bus is also needed. The PDCP itself needs to be implemented, so that it is possible for the microcontroller to interpret the messages accordingly. Finally, output files need to be generated, so that it is possible for a user to view the activity on the bus on a PC in a human-readable way.

## 4.2 Design Decisions

With the requirements shown in Section 4.1 the hardware to use can be narrowed down. As hardware limits the range of software which can be used the software decisions can only be made after the hardware is decided.

### 4.2.1 Hardware

As this project consists of two different parts, which have been separated by the hardware itself, design decisions needed to be made individually. For the FPGA part the Altera DE0-nano[27] board will be used, which has been briefly introduced in Section 2.3. In comparison to the Altera DE2 board the DE0-nano board features a Cyclone IV FPGA with 22,320 logic gates. These are less gates than the FPGA on the DE2 board, but the project only utilizes about 16,000 logic gates, so it can easily be fitted into the FPGA of the DE0-nano board. Furthermore this board lacks most of the hardware functionality that the DE2 board has, thus an external LCD will be connected to one of the GPIO ports to be able to display the output messages.

The second part of this project is done on a microcontroller. It will be done on the Microchip dsPIC33E USB Starter Kit which has been introduced in Section 2.1.2. The dsPIC33E board will be connected to the Microchip Multimedia Expansion Board that was introduced in Section 2.2. For debugging purposes the onboard LCD screen of the MEB will be used. The MEB itself has a microSD card slot which will be used for storing the analyzed data and for reading the user configuration.

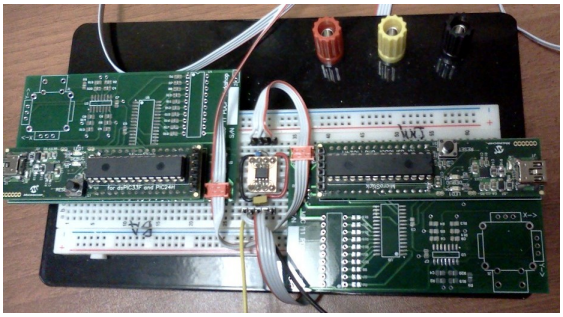

**Figure 4.1:** The bus arbitrator and the data monitor.

To be able to evaluate the working of the CAN and PDCP code external hardware will be used. This hardware setup has been created by the Institute of Biomedical Engineering (IBME) at the University of New Brunswick and it contains the BA, a data monitor, and two electrodes. The BA and the data monitor can be seen in Figure 4.1. The electrodes contain a module which wraps the read values in a PDCP message and sends them over the CAN bus. An example of the electrode board can bee seen in Figure 4.2. All of the modules are connected via one CAN bus. The CAN RX and TX wire, as well as the GND wire, will be connected to the expansion header of the MEB. Furthermore the hardware setup provides two additional wires for UART TX and GND for the FPGA board. These two wires will also be connected to the MEB. As the pins on the expansion header are used for different functionality than in their specification, these pins need to be reconfigured before addressing them in the code.

### 4.2.2 Software

For the FPGA part the Altera Quartus II[28] IDE will be used as this is the best supported IDE for programming Altera FPGAs. The version used for this project is Altera Quartus II 11.1 on a Ubuntu 11.10 64-bit machine. Quartus supports the most common HDLs Verilog and VHDL, as well as the Altera-proprietary AHDL. Thus, only the code from the Altera DE2 board programmed in Altera Quartus II 11.0 has to be ported to the DE0-nano board with pin and voltage modification.

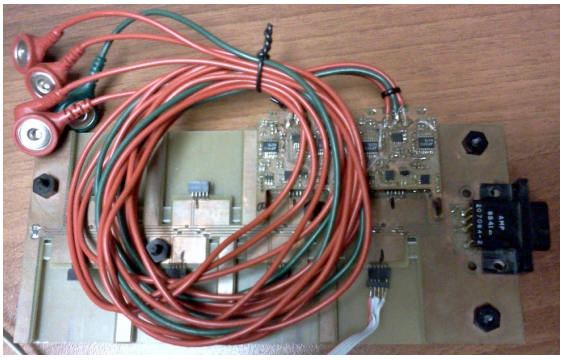

**Figure 4.2:** Electrodes that operate using the PDCP.

The microcontroller can be programmed by using the Microchip MPLAB C30 compiler[29], which is a compiler toolchain for the Microchip PIC24, dsPIC30F and dsPIC33F families. On top of this the Microchip MPLAB X IDE v1.0[30] will be used. The programming of the microcontroller will be done on a Ubuntu 11.10 64-bit machine as well as on a Mac OS X 10.7.2 64-bit machine. Figure 4.3 shows a screenshot of the MPLAB X IDE.

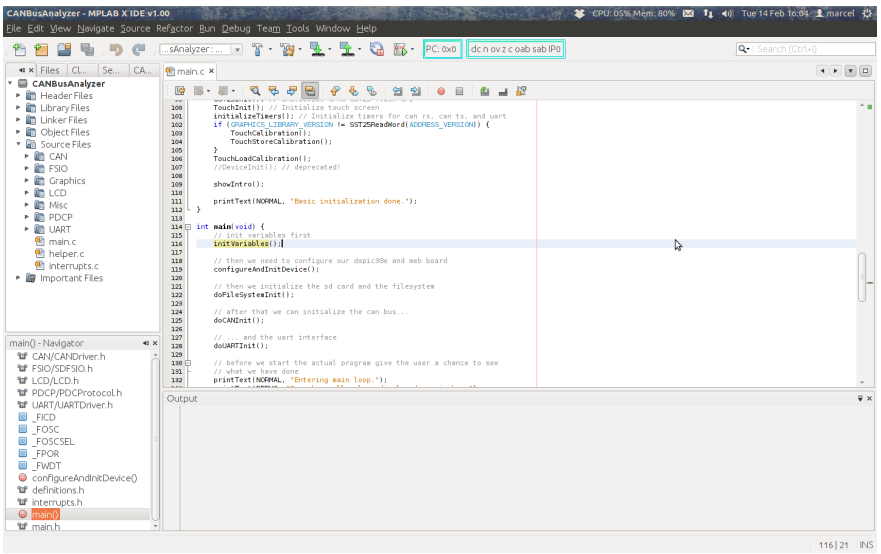


**Figure 4.3:** Basic layout of the MPLAB X IDE.

The architecture of this project will be shown in Subsection 5.3.1. The project has been designed top-down, meaning that every problem has been split into smaller problems as often as possible. Afterwards the implementation has been done bottom-up, which means that starting from the smallest problem the implementation has been done in a hierarchical way to the topmost problem. But before actually being able to debug the code a driver for the LCD needed to be implemented. With this driver and simple display printing routines the debugging of the solution of the problems can be simplified. The next chapter will explain what code has been reused, as well as how the porting of the code to another FPGA board has been achieved, and finally give implementation details about the different problems and how these modules interact with each other.

# 5 Implementation

The first thing to do when implementing a bigger project is to look for code which can be reused, so that not every part of the code needs to be written from scratch. This applies to the microcontroller part as well as to the FPGA part. After that the porting of the code from the Altera DE2 board as shown in [11] to the Altera DE0-nano board is shown. Following this the architecture and the details of the microcontroller implementation is given as well as the interaction of the modules inside the microcontroller.

## 5.1 Usage of Existing Code

Microchip provided demo code for programming the MEB using the dsPIC33E USB Starter Kit. This includes a demo for the onboard LCD as well as for the accelerometer and the SD card. The LCD code has been modified to fit this project's purpose of debugging the code quicker than with the built-in debugging tools. With this modified code it is possible to display messages on the LCD which reflect what happens when the code is running. Figure 5.1 shows the debug output for the config file. The SD card demo code is not working out-of-the-box with the dsPIC33E board, as it has been written for the Microchip Memory Disk Drive File System (MDDFS) which is not compatible with the dsPIC33E board[31]. Thus the accelerometer demo code has been used and modified to work with the SD card demo code for the MDDFS. For this to work the CPLD onboard the MEB needed to be configured first:

```
1     CPLDInitialize();
2     CPLDSetGraphicsConfiguration(GRAPHICS_HW_CONFIG);
3     CPLDSetSPIFlashConfiguration(SPI_FLASH_CHANNEL);
```

The first command initializes the CPLD by setting the corresponding pins for the MEB to allow the configuration of the CPLD. In order to write to the LCD the second line is needed, which tells the onboard Solomon SSD1926 controller to allow writing to the LCD. The last line tells the Solomon SSD1926 controller to use SPI for addressing the microSD card.

For writing to the LCD the provided libraries from Microchip have been

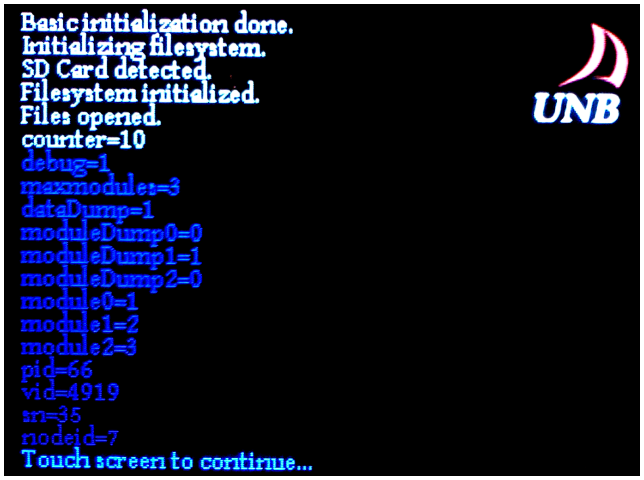


**Figure 5.1:** Debug messages displayed during startup.

used and extra functionality has been added. This includes the writing of strings in different output formats: `NORMAL`, `DEBUG`, `INFO`, `WARNING`, and `ERROR`. The output formats `NORMAL`, `INFO`, and `WARNING` display the output on the LCD and then continue with the program. Their only difference is in the colour that is used for displaying messages. The format `DEBUG` only prints the text if the debug switch has been set in the configuration file. `ERROR` displays the error that occurred and halts the program. This is used if needed functionality cannot be provided and the program has no option to continue, e.g. if the SD card is missing. During the evaluation of this project the LCD will be referenced more often, as it sometimes has been crucial to see what was happening without having to look at the internal registers and memory in the debugger view. Furthermore the LCD methods were enhanced to halt the program flow temporarily when too many messages were displayed, so that the user has to touch the screen to continue with the program.

## 5.2 Porting the Code from the DE2 to the DE0-nano Board

The FPGA code as shown in [11] has been developed for the Altera DE2 Development and Education board. As the Altera DE0-nano board uses a different FPGA changes in the code needed to be made. This also includes

| Node Name | Old Pin Assignment | New Pin Assignment | Voltage |
|---|---|---|---|
| CLOCK_50 | PIN_N2 | PIN_R8 | 2.5V |
| LCD_DATA[7] | PIN_H3 | PIN_E7 | 2.5V |
| LCD_DATA[6] | PIN_H4 | PIN_D8 | 2.5V |
| LCD_DATA[5] | PIN_J3 | PIN_C8 | 2.5V |
| LCD_DATA[4] | PIN_J4 | PIN_E6 | 2.5V |
| LCD_DATA[3] | PIN_H2 | PIN_A7 | 2.5V |
| LCD_DATA[2] | PIN_H1 | PIN_C6 | 2.5V |
| LCD_DATA[1] | PIN_J2 | PIN_B7 | 2.5V |
| LCD_DATA[0] | PIN_J1 | PIN_D6 | 2.5V |
| LCD_EN | PIN_K3 | PIN_B6 | 2.5V |
| LCD_POWER | N/A | PIN_F13 | 3.3V |
| LCD_RS | PIN_K1 | PIN_A5 | 2.5V |
| LCD_RW | PIN_K4 | PIN_A6 | 2.5V |
| TxDWire | PIN_E25 | PIN_T11 | 2.5V |

**Table 1:** The pin assignment and the voltages for the Altera DE0-nano board.

changes in the pin assignment, as the DE0-nano board does not feature a LCD and an external LCD had to be connected via one of the GPIO boards.

The pin assignment as well as their corresponding voltages can be seen in Table 1. All of the pins except the LCD_POWER pin run at a voltage of 2.5V. The LCD_POWER pin needs to run at 3.3V in order for the data pins to work at 2.5V[32]. The LCD itself features the same controller as the LCD on the Altera DE2 board, thus the existing code can be reused without modification. The LCD on the DE2 board does not need a power connection as it automatically receives power from the board.

Furthermore the implementation for the seven segment LEDs is not needed, as the DE0-nano board does not feature any seven segment LEDs. In order to be able to select the module that has to be displayed two of the push buttons on board of the DE0-nano board have been used. Their wiring is as shown in Table 2.

To now be able to select a module the push buttons have been implemented

| Node Name | Pin Assignment | Voltage |
|---|---|---|
| KEY[1] | PIN_E1 | 2.5V |
| KEY[0] | PIN_J15 | 2.5V |

**Table 2:** The pin assignment and the voltages for the push buttons.

so that pressing button KEY[1] increments the module counter and KEY[0] decrements the module counter. This has been realized using a state machine:

```
1  always@(posedge clock)
2  begin
3    case(inputButtons)
4      2'b10: // key 1 pressed
5      begin
6        if (!buttonPressed)
7        begin
8          outputNumber = outputNumber - 1;
9          if (outputNumber > 32)
10           outputNumber = 32;
11         buttonPressed = 1'b1;
12       end
13     end
14
15     2'b01: // key 0 pressed
16     begin
17       if (!buttonPressed)
18       begin
19         outputNumber = outputNumber + 1;
20         if (outputNumber > 32)
21           outputNumber = 0;
22         buttonPressed = 1'b1;
23       end
24     end
25
26     2'b00: // no button pressed
27       buttonPressed = 1'b0;
28
29     // this implementation ignores the case when both buttons
         are pressed
```

```
30    endcase
31  end
```

The advantage of using a state machine is that each of the four possible cases can be handled individually: button 1 is pressed, button 0 is pressed, no button is pressed, both buttons are pressed. Furthermore a register is used which only increments or decrements the value once as long as a button is being pressed. If no button is pressed this flag is cleared and pressing another button will then do its corresponding function.

## 5.3 Architecture

The architecture itself is first explained in an overview. Succeeding, the low level and high level functions are explained.

### 5.3.1 Overview

The project has been divided into several parts. Thus it contains different modules according to their functionality. For the software part the GRASP pattern of high cohesion and low coupling according to [33] has been used. This results in different modules with every module being responsible for their own part of functionality. These modules are:

1. SD Card Module
2. CAN Bus Module
3. UART Module
4. PDCP Module
5. Statistics Module

The first three modules contain the low level drivers in order to access the SD card for (1), accessing the CAN bus for (2), and interfacing via UART for (3). Furthermore it has been decided to include the high level part of the software also in these modules, so that each module is closed for their functionality. Thus it is easier to access the different functionality, as well as to add more functionality, e.g. new hardware or software methods. As this

project has been realized using the MEB the advantage of having a LCD has been used as well. Thus the high level LCD methods are also included in the statistics module.

The implementation details of the SD card module are discussed in Subsection 5.3.2, the CAN module is discussed in Subsection 5.3.3, the UART module in Subsection 5.3.4, the PDCP module in Subsection 5.3.5, and finally the statistics module as well as the LCD methods in Subsection 5.3.6. After explaining the implementation details of every module the interaction of every module will be shown in Section 5.4.

### 5.3.2 SD Card Driver

As shown in Section 5.1 the CPLD needs to be initialized before being able to address the microSD card on the MEB. After this has been done the low level functions provided by Microchip in their MDDFS example can be used. These functions include opening and closing a file, changing, creating, and removing a directory, reading, writing, and seeking in a file, as well as flushing the data that is still in the buffers.

High level functions have been put atop of these, which are:

```
1 void doFileSystemInit();
2 int getCounter();
3 char *getConfigItem(char *item, char *config);
4 void readConfig(FSFILE *pointer);
5 void writeFile(FSFILE *pointer, const char *text);
6 void saveToStatistics(const char *data);
7 void saveToLog(const char *data);
8 void saveMessage(BYTE_BITS message[16]);
```

The only methods that need to be called by other modules are `doFileSystemInit()`, `saveToStatistics(...)`, `saveToLog(...)`, and `saveMessage(...)`. The first method initializes the filesystem by reading the config files from the microSD card. For this two different files need to be present on the microSD card: `config.txt` and `counter.txt`. A possible `config.txt` can look like this:

```
1  debug=1
2  dataDump=1
3  maxModules=3
4  module0=1
5  module1=2
6  module2=3
7  moduleDump0=0
8  moduleDump1=1
9  moduleDump2=0
10 pid=66
11 vid=4919
12 sn=35
13 nodeid=7
```

The first item toggles debug messages on and off. With the option `dataDump` it can be toggled that every CAN message that arrives will be saved in the log file. `maxModules` specifies how many modules will be monitored. After that for the amount of modules specified two options are expected to be present: `module`$x$ ($0 \leq x <$ `maxModules`) expects the node ID of the module that needs to be monitored and `moduleDump`$x$ ($0 \leq x <$ `maxModules`) specifies if every message with this node ID needs to be logged. The options `pid`, `vid`, and `sn` specify the devices product ID, vendor ID, and serial number, respectively. Furthermore it is also possible to assign a node ID to this device with the option `nodeid`, but this node ID might be changed by the BA during runtime.

The second configuration file that is needed is `counter.txt`, which only needs to include a positive integer variable, in order for the program to create independent logging and statistics files for every time this device is used. These files are `log_`$x$`.txt` and `stat_`$x$`.txt`, with $x$ being the value from `counter.txt`.

With the methods `saveToStatistics(...)` and `saveToLog(...)` a string can be written to the statistics and logging file, respectively. In order to save a message to the logging file the method `saveMessage(...)` is utilized, which takes `BYTE_BITS[16]` as an input and converts these 128 bits to a hexadecimal number and puts it into a string.

The method `getCounter()` reads from the file `counter.txt` and returns the read integer. `readConfig(...)` reads the configuration from the specified parameter. In order to look for specific options in a configuration file the method `getConfigItem(...)` searches in the variable `config` for the item `item` and returns its value.

Finally, the method `writeFile(...)` writes `text` to the specified file `pointer` which needs to be opened before calling this method. The methods `saveTo Log(...)` and `saveToStatistics(...)` make use of this method.

### 5.3.3 CAN Driver

The CAN RX and TX wire have been connected to the `C2OUT/AN9` port for CAN RX and to the `RA10` port for CAN TX of the MEB according to [34]. These pins map to `RPI41` for CAN RX and `RA10` of the Microchip dsPIC33EP512MU810 microcontroller. Before being able to reprogram the pins all reprogrammable pins need to be unlocked with the following statement:

```
1 __builtin_write_OSCCONL(OSCCON & ~(1 << 6));
```

This is an internal function that unlocks the reprogrammable pins according to [4]. The reprogramming of the CAN RX pin can then be done as follows:

```
1 RPINR26bits.C1RXR = 0x2900;
```

This maps the `C1RX` pin of the on-board ECAN module to pin `RPI41`. After the necessary pins have been reprogrammed the reprogrammable pins need to be locked again:

```
1 __builtin_write_OSCCONL(OSCCON | (1 << 6));
```

The CAN Module itself contains the driver for the CAN bus as two interrupts, one for RX and one for TX. Furthermore it contains high level methods for generating statistics, wrapping of data into CAN messages, calculating CRC, interpreting CAN messages, as well as the initialization:

```
1 void doCANInit();
2 void BARegister();
```

```
3  void generateCANStatistics(unsigned short msgId, unsigned short
       length);
4  void calculateCRC(BYTE_BITS message[8], BYTE_BITS dlc);
5  void createCANMessage(BYTE_BITS message[8], BYTE_BITS dlc,
     BYTE_BITS priorityAndMode);
6  void sendMessage(BYTE_BITS message[8], BYTE_BITS dlc, BYTE_BITS
       priorityAndMode);
7  void interpretCANMessage();
8  void __T1_ISR _T1Interrupt(void);
9  void __T5_ISR _T5Interrupt(void);
10 void fakeCANMessage();
```

The methods `doCANInit()`, `BARegister()`, and `sendMessage(...)` are the only methods that need to be accessed from other modules. All other methods are internal methods. `doCANInit()` resets the variables for this module, sets the mode for PDCP to be unregistered with the BA, and reconfigures the CAN RX pin. The activation of the interrupts for CAN RX and CAN TX could have been done in there, but it has been decided to put them in the basic system device initialization. In there all of the different interrupts are initialized at once. This method can be found in `interrupts.c` and is called `initializeTimers()`.

To be able to register this device with the BA in a PDCP system the method `BARegister()` needs to be triggered by the main loop. It acts as a state machine depending on the current state the registration process is in: `BIND_REQUEST`, `RETRY_BIND_REQUEST`, `ACK`, `WAIT_FOR_RESPONSE`, and `SUCCESS`. The first two states have the same behaviour, as in there a Bind Device Request message for the BA will be created according to the device's vendor ID, product ID, and serial number. At first a predefined node ID will be used to send this message. If the BA gave the device a different node ID this ID will be used for retrying the Bind Device Request. After sending the message the state changes to `WAIT_FOR_RESPONSE`. If the BA has responded with a node ID for this device, the PDCP interpretation module changes the state of this state machine either to `RETRY_BIND_REQUEST` or `ACK` depending on the outcome of the message. If the state `ACK` has been reached an internal flag is toggled to signal the microcontroller that it is

successfully bound to the BA and the state changes to `SUCCESS`. This is an empty state which cannot be left.

When a new CAN message has arrived the method `interpretCANMessage` is triggered. In there the message is split into message mode, message priority, and message ID. After that it is checked if this module needs to be monitored. If the module needs to be monitored or if a data dump happens, the length of the message is then added to the corresponding module load variable by the method `generateCANStatistics(...)`. Furthermore this CAN message is passed on to the PDCP interpretation module, which is explained in Subsection 5.3.5. For instance, if the following CAN message is received:

`00 70 e0 22 66 e0 08 40 04 60 00 2f f,`

the interpretation of this message would only lead to message is from ID 7, it has a DLC of 7, and the data is 01 13 37 00 42 00 23. Any interpretation of what this actually means is done in the PDCP interpretation module.

During debugging and evaluation of this project an additional method has been used: `fakeCANMessage()`. This method emulates the reception of a predefined CAN message. The result of this is that the main loop then thinks a new CAN message has arrived and interprets this message.

### 5.3.4 UART Driver

For this project it was only necessary to be able to receive messages via UART. Thus only the following methods were implemented:

```
1 void doUARTInit();
2 void __T2_ISR _T2Interrupt(void);
3 void fakeUARTMessage();
```

As explained in Subsection 5.3.3 all interrupts are activated in `interrupts.c`. This also applies to the UART RX interrupt. The method `doUARTInit()` resets the variables needed for this module and reprograms the pin for the internal UART module. The UART RX wire has been connected to pin `SDI1A` on the expansion header of the MEB. This pin maps to `RP98` on the

dsPIC33EP512MU810 microcontroller. The following code shows how to reprogram the `U1RX` to this pin:

```
1 RPINR18bits.U1RXR = 0x6200;
```

This remapping also requires the pins to be unlocked before reprogramming and locking them again after being done.

The ISR for UART RX waits for a 0 at the beginning, as a 0 signalizes that a new UART frame is coming. After that it saves the next eight bits. In [11] the UART has been configured to operate at a bit rate of 115,200 kbps with two stop bits and no parity bits. Thus this ISR is triggered every 1042 microcontroller clock ticks. When a full UART frame has been received the ISR signals the main loop that a new message is waiting. It has been decided that the interpretation of this message has been put into the statistics generation module, as all UART messages are already interpreted by the FPGA and can be directly used for updating the statistics.

The UART module, as well as the CAN bus module, includes a method for emulating the reception of a predefined UART message. This has been used in order to easily debug and evaluate the working of this module.

### 5.3.5 PDCP Interpreter

After the CAN module as shown in Subsection 5.3.3 has interpreted the basic layout of the CAN message it passes the message over to the PDCP interpreter, which segments the message according to the PDCP. For this it uses a state machine according to the first byte of the data field of the message. The full list of implemented functions is shown in Table A.2. The PDCP function table in the Appendix shows all functions that are specified. Some of them are not used anymore.

Depending on the function code zero to seven bytes of additional data have to be interpreted. As the implemented project needs to be able to connect and register itself with the BA the function for Bind Device Request Response have to check if the device itself is meant:

```
if (checkIfBytesAreEqual(data[2], vendorID[1]) &&
 checkIfBytesAreEqual(data[3], vendorID[0]) &&
 checkIfBytesAreEqual(data[4], productID[1]) &&
 checkIfBytesAreEqual(data[5], productID[0]) &&
 checkIfBytesAreEqual(data[6], serialNumber[1]) &&
 checkIfBytesAreEqual(data[7], serialNumber[0]) &&
 !successfullyBound) {
  if (checkIfBytesAreEqual(data[1], nodeID))
    BARegisterState = ACK;
  else {
    nodeID.Val = data[1].Val;
    BARegisterState = RETRY_BIND_REQUEST;
  }
}
```

| PDCP Function Code | Description |
|---|---|
| 0x01 | Bind Device Request |
| 0x03 | Get Device Parameter |
| 0x04 | Set Device Parameter |
| 0x08 | Suspend Device |
| 0x09 | Release Device |
| 0x0A | Device Beacon |
| 0x0B | Reset Device |
| 0x0C | Configure Get Bulk Data Transfer |
| 0x0D | Configure Set Bulk Data Transfer |
| 0x0E | Bulk Data Transfer |
| 0x0F | Update Data Channel |
| 0x81 | Bind Device Request Response |
| 0x83 | Get Device Parameter Response |
| 0x84 | Set Device Parameter Response |
| 0x88 | Suspend Device Response |
| 0x89 | Release Device Response |
| 0x8B | Reset Device Response |
| 0x8C | Configure Get Bulk Data Transfer Response |
| 0x8D | Configure Set Bulk Data Transfer Response |
| 0x8E | Bulk Data Transfer Response |
| 0x8F | Update Data Channel Response |

**Table 3:** The implemented PDCP functions.

As the Bind Device Request Response includes the vendor ID, as well as the product ID and serial number, during interpretation it needs to be checked if they are the device's IDs. The method `checkIfBytesAreEqual(...)` takes two `BYTE_BITS`, which are 8 bits wide, as an input and compares them against each other as there is no method implemented to compare them by e.g. `if (byte1 == byte2)`:

```
1  short checkIfBytesAreEqual(BYTE_BITS byte1, BYTE_BITS byte2) {
2    if (byte1.bits.b0 == byte2.bits.b0 &&
3     byte1.bits.b1 == byte2.bits.b1 &&
4     byte1.bits.b2 == byte2.bits.b2 &&
5     byte1.bits.b3 == byte2.bits.b3 &&
6     byte1.bits.b4 == byte2.bits.b4 &&
7     byte1.bits.b5 == byte2.bits.b5 &&
8     byte1.bits.b6 == byte2.bits.b6 &&
9     byte1.bits.b7 == byte2.bits.b7)
10      return 1;
11
12   return 0;
13 }
```

If for instance the message `00 70 e0 22 66 e0 08 40 04 60 00 2f f` is received the PDCP interpreter will recognize the node ID as 7, with the message having a priority of 0 and being sent in mode 0 with the message to be a Bind Device Request by a node with vendor ID `0x1337`, product ID `0x0042`, and serial number `0x0023`. After interpreting the message the PDCP module generates output messages that then will be saved to the logging and statistics file. If the device is known to the program the output message will be changed accordingly to the known devices, e.g. `can bus analyzer: bind device request` instead of `0x23: bind device request`.

### 5.3.6 Statistics Generation

The statistics generation module is coupled with the LCD, as the LCD is only being used for displaying statistics. Therefore this module includes the low-level LCD driver, as well as high-level LCD functions. As shown in Section 5.1 a method named `printText(...)` has been implemented to put strings

on the display. Furthermore, the methods `putStatisticsOnDisplay()` and `showIntro()` belong to the LCD high-level functions. The second method only shows some pre-defined strings at the startup of the program, which can be freely changed in order to e.g. display a disclaimer. The first method prints the outline of the main interface on the LCD:

```
1  SetColor(LIGHTGRAY);
2  SetFont((void *) &GOLSmallFont);
3  while (!OutTextXY(0, 0, "CAN␣Bus␣Analyzer␣Tool"));
4  while (!OutTextXY(0, 12, identification));
5  while (!OutTextXY(0, 40, "Last␣CAN␣Message:"));
6  while (!OutTextXY(0, 80, "Last␣UART␣Message:"));
7  while (!OutTextXY(0, 120, "Bus␣Load:"));
8  while (!OutTextXY(0, 160, "Last␣Action:"));
9
10 SetColor(BLUE);
11 SetFont((void *) &GOLSmallFont);
12 while (!OutTextXY(0, 228, "(c)␣2011-2012␣UNB"));
```

The method `OutTextXY(...)` has been implemented by Microchip and allows the printing of text onto the display on a position $(x, y)$ that is given as the first two parameters. The functions `SetColor(...)` and `SetFont(...)` are responsible for changing the colour and the font, respectively. A colour can be chosen by converting the tuple $(R, G, B)$ with $0 \leq R, G, B < 2^8$ to a colour value, with $R$ being responsible for the red part of the colour, $G$ for green, and $B$ for blue. The bit value can then be converted by using the Microchip implemented macro `RGB565CONVERT(...)`, e.g. the colour red would be implemented as `RGB565CONVERT(255, 0, 0)`. Microchip provided different font types for the font used for displaying text. They also provided a tool that allows the conversion of user-installed fonts from the host PC into a format that is compatible with the Microchip dsPIC33E MC family.

If the sample period has passed an interrupt is called which raises a flag for the statistics module in order to sample the current values and update the output files. The method that is called is `sampleAndUpdate()`. The sample rate of this method has been set to one second, but it can be changed by setting the prescaler and ticks for timer 3 to different values in the file

`interrupts.c`:

```
1     PR2 = 5000;
2     T3CONbits.TCKPS = 3
```

The prescaler has been set to 256 with the ticks counter at 5000. The clock frequency of this project is 32 MHz. This results in the timer 3 to toggle an interrupt $32 * 10^6/(256 * 5000) = 25$ times a second. In order to achieve a sample rate of one second the `sampleAndUpdate()` method needs to become active every 25 times the interrupt has been triggered. In order to achieve this behaviour the ISR for timer 3 checks if the variable `displayUpdateFlag` is set to 25 and then toggles the `updateDisplay` variable which signals the statistics module to become active:

```
1 if (displayUpdateFlag++ == 25) {
2   displayUpdateFlag = 0;
3   updateDisplay = 1;
4 }
```

The statistics module then updates the values on the display (see Appendix Chapter A.3 for the corresponding code) and calculates the overall bus load depending on the raw value received by the FPGA:

```
1 load = 1.0 * overallLoad / 1000;
2 loadPercent = 1.0 * overallLoad / MAXLOAD / SAMPLERATE;
```

As `overallLoad` is an integer an explicit type-cast has to be done in order to receive floating point numbers for the division. `MAXLOAD` is set to 1000000, which is the bitrate of the CAN bus and `SAMPLERATE` is set to 1, as the method is triggered once a second.

As the PDCP interpreter already saved a string depending on the PDCP message received the statistics module can directly use this string and write to the output file. In the statistics output file at every sampling point the overall bus load as well as the bus load of the monitored modules is written:

```
1 sprintf(text, "overall=%8l␣", overallLoad);
2 saveToStatistics(text);
3 overallLoad = 0;
4 for (i = 0; i < maxModules; i++) {
5   sprintf(text, "%02x:␣load=%08d", moduleNumber[i], moduleLoad[
        i]);
```

```
6     saveToStatistics(text);
7     moduleLoad[i] = 0;
8   }
```

The received CAN message as well as its interpretation by the PDCP is written to the logging file:

```
1 saveToLog(oldCANMessage);
2 saveToLog(action);
```

## 5.4 Interaction of the Modules

A diagram containing all the modules in the microcontroller can be seen in Figure 5.2. This diagram also shows how the FPGA is connected to the microcontroller and how it interacts with the CAN bus and the microcontroller. The full architecture of the FPGA is shown in [11].

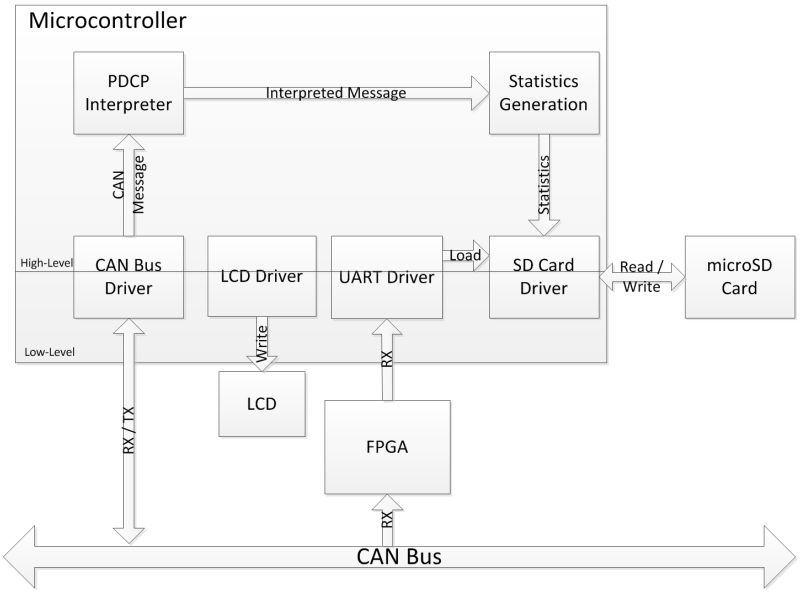


**Figure 5.2:** The microcontroller architecture design.

The MC is directly connected to the CAN bus via the CAN Bus Driver and is able to receive and transmit messages from and to the bus. The

FPGA on the other hand only has receiving capabilities as it does not need to be able to write anything on the bus. With the help of the ECAN module inside the dsPIC33EP512MU810 MC most low-level features are already implemented in hardware. The high-level part of the CAN Bus Driver receives messages from the CAN bus and passes them on to the PDCP Interpreter.

The PDCP Interpreter only includes high-level functions and after it is done with the calculation it sends the Statistics Generation module the interpreted message. Via the UART Driver the MC is capable of receiving messages via UART from the FPGA (see [11] for protocol details). The Statistics Generation module takes the received values from the UART Driver and the interpreted messages from the PDCP Interpreter and creates statistics, which then are written by the SD Card Driver to the microSD card. The SD Card Driver is also responsible for reading files from the microSD card.

Not shown in Figure 5.2 is the interrupts module which has been left out of the diagram to increase the readability. This module is responsible for configuring and activating the interrupts. It does not include the ISRs itself, as these are being handled by their corresponding modules.

# 6 Evaluation

The evaluation of this project is done by first explaining the approach. Then the FPGA part and the microcontroller part are evaluated and finally the results are explained.

## 6.1 Approach

To be able to evaluate this project the evaluation has been split according to its functionality. After every part has been evaluated on its own the whole project can be evaluated. For the evaluation of the low-level part of the microcontroller as well as for the FPGA a digital oscilloscope and logic analyzer will be used to analyze the signals received and transmitted. The tool that is used is the USBee DX and can be seen in Figure 6.1.


**Figure 6.1:** The USBee DX oscilloscope and logic analyzer[35].

At first the FPGA part will be evaluated. The evaluation of the porting of the design from the Altera DE2 to the Altera DE0-nano board is shown in Section 6.2. This is done by looking at the CAN driver, the UART driver, the LCD driver, and the load calculation.

After that it is possible to evaluate the microcontroller. As the microcontroller architecture has been split into low-level drivers and high-level functionality the evaluation will look at both aspects separately at first.

The low-level drivers are evaluated in Subsection 6.3.1, the high-level functions evaluation is shown in Subsection 6.3.2. This chapter concludes with the evaluation of the whole project in Section 6.4.

## 6.2 The DE0-nano Code

The evaluation of the project on the Altera DE2 board has been shown in [11]. Following this evaluation approach at first the LCD will be evaluated, then the UART module, the CAN driver, and finally the load calculation module.

The LCD connected to the Altera DE0-nano features a HD44780 compatible controller[36], which is the same controller as on the Altera DE2 board. In order to make sure that the LCD is running correctly the correct input voltages have to be chosen. The datasheet for the LCD specifies that it can be run at either $+5V$ or $+3.3V$[36]. The Altera DE0-nano board features on both GPIO ports pins with $V_{CC\_SYS} = +5V$ and $V_{CC3P3} = +3.3V$. Unfortunately all of the data pins can only be driven with a maximum of $+3.3V$ which disallows the usage of the $+5V$ operation mode, as the input high voltage has to be at least $+3.5V$[36]. Thus the data pins have to be set to be between $+2.3V$ and $+3.3V$ for high voltage and below $+0.6V$ for low voltage. The voltages have been set as shown in Table 1. After the voltages have been set accordingly the LCD code can be evaluated. As the controller of the LCD is fully compliant to the controller of the Altera DE2 board the code was working directly with the same behaviour as on the Altera DE2 board.

In order to be able to evaluate the UART driver the USBee DX oscilloscope and logic analyzer has been connected to the UART TX pin of the Altera DE0-nano board. As the board has to follow the same protocol as shown in [11] it has to be made sure that it is configured at a baudrate of 115.200 kbps with two stop bits and no parity bit. The protocol for sending the data is as follows[11]:

1. Wait for data to be ready

2. Send start signal
3. Send the module id number
4. Send the corresponding load
5. Repeat 3 and 4 for all 32 modules
6. Signal that the overall load is sent
7. Send overall load
8. Set the data ready variable to false
9. Go to 1

The software for the USBee DX allows the configuration of the UART wires with different settings, so it has been configured to run at these settings. Figure 6.2 shows partially the received data stream on the UART RX wire of the USBee DX. Two wires are shown in this Figure. The top one represents the interpreted UART signals while the lower one shows the data stream. As can be seen on this figure the FPGA sends the start signal, which is `00 10 00 01` followed by the package for module `0` which is `00 00 00 00` and the remaining packages for module `1` to `8`. Module `9` to `31` and the overall load has been cropped out of the image.

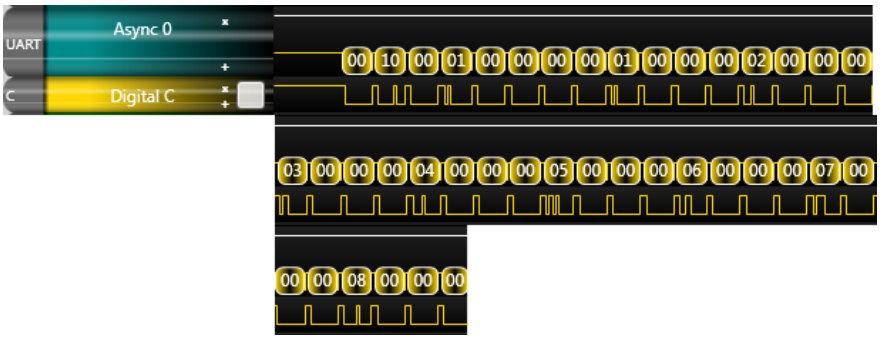


**Figure 6.2:** The UART data stream captured by the USBee DX.

For the CAN driver evaluation the same approach as for the evaluation of the CAN driver on the Altera DE2 board has been used. The DE0-nano

board has been configured to display every CAN message received on the LCD and received CAN messages have been compared with the CAN messages that were sent on the bus. With this approach it can be verified that the CAN driver is working correctly.

Finally, the load calculation module has to be evaluated. For this the Altera DE0-nano board as well as the Altera DE2 board have been hooked up to the CAN bus. Then the load values calculated by both boards have been compared against each other to ensure that the calculation of the DE0-nano board is working correctly.

## 6.3 Microcontroller Evaluation

In order to efficiently evaluate the microcontroller part at first the correct functionality of the low-level methods of the project needs to be ensured, as the high-level methods rely on these. Thus, the evaluation starts with the low-level part and is then followed with the high-level part.

### 6.3.1 Low-Level Part

As the low-level part of the microcontroller consists of the LCD driver, the CAN driver, the UART driver, and the microSD card driver each driver will be evaluated separately, starting with the LCD driver. As Microchip provided a fully functional library for using the LCD via the Solomon SSD1926 graphics chip on the Microchip MEB which is connected to the Microchip dsPIC33E USB Starter Kit it only needs to be made sure that their methods are working according to their specification. After that printing text on the LCD and clearing the LCD screen could be evaluated. Figure 6.3 shows sample output on the LCD. As can be seen on this image the program waits for the user to touch the screen after it has been filled with messages.

For the evaluation of the CAN driver the USBee DX has been used. After configuring the ECAN module in the MC, different pre-defined messages have been sent over the CAN TX wire and the captured waveform has been compared against the message that was sent. Figure 6.4 shows the wave-

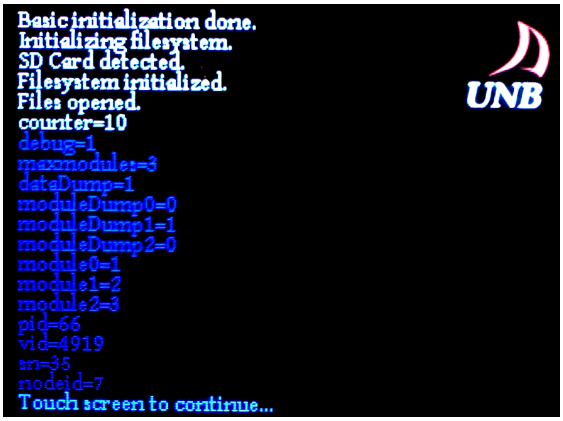


**Figure 6.3:** Messages displayed on the LCD.

form captured by the USBee DX when the MC sent a message with node ID `0x05`, a DLC of `1` with data `0x11`.


**Figure 6.4:** CAN Message received by the USBee DX.

Once the evaluation of the transmission of messages using the ECAN module was done the reception of messages can be evaluated. This has been done by connecting different nodes to the CAN bus. At first only the BA has been attached to the bus. As the BA sends a beacon every $500ms$ this beacon also has to be captured by the MC every $500ms$. The BA has the node ID `0x01` and the beacon has a DLC of `1` with data `0x0A`. This message is captured by the ECAN module and with the help of the LCD the received CAN messages can be displayed in order to compare them against the transmitted message. After these messages were received correctly other modules were connected to the bus and the correct reception of their messages have been checked.

The UART driver has been evaluated by sending pre-defined messages via UART to the UART RX pin of the MC. At first it had to be made sure that

the UART RX wire was configured correctly. After that messages could be received by the UART wire. This has been tested by sending different 8 bit values from a PC over the UART interface to the MC. For this the program used to test UART reception in [11] has been changed to send values. Then the MC was connected via UART to the FPGA and it was checked if the values received corresponded to the values transmitted by the FPGA. The evaluation of the protocol will be shown in Subsection 6.3.2.

Microchip provided filesystem functions for using a SD card connected to the dsPIC33E USB Starter Kit. But these functions were not compatible with the MEB, as the Solomon SSD1926 graphics controller is responsible for handling file I/O operations. Thus, at first a connection with the graphics controller needed to be established. As the accelerometer example included the connection with the graphics controller this code has been used for this project. After the connection was established at first read and write operations were tested if they are working:

```
1  FSFILE *file;
2  char *testString = "test";
3  char *testString2;
4
5  // writing to a file
6  file = FSfopen("test.txt", "w");
7  FSfwrite(testString, sizeof (char), sizeof (testString), file);
8  FSfclose(file);
9
10 // reading from a file
11 file = FSfopen("test.txt", "r");
12 FSfread(testString2, 4, 1, file);
13 FSfclose(file);
14 printText(DEBUG, testString2); // print text on the LCD
```

Lines 6-8 open a file on the microSD card for write access and write the string `test` in the file `test.txt`. The write operation is then tested by reading from the same file in lines 11-14 and displaying the read string on the LCD. After successful reading and writing from the microSD card high-level functions as reading and interpreting a configuration file could be evaluated as will be shown in Subsection 6.3.2.

### 6.3.2 High-Level Part

As the CAN driver, the UART driver, and the SD card driver also provide high-level functions these have to be evaluated as well as the PDCP interpretation and statistics generation module. The CAN driver is capable of interpreting received CAN messages. For the evaluation purpose of this method the device beacons sent by the BA were used for testing this method as well as the binding request of different nodes. Once a complete CAN message is received a flag is toggled to signal the CAN module to interpret this message. To make sure that the interpretation method is called a display output has been generated when the method is called:

```
1 void interpretCANMessage() {
2   printText(DEBUG, "CAN message received");
3
4   ...
5 }
```

Whenever a new CAN message is received a new message appears on the screen. Once this was working the segmentation of the CAN message into the node ID, DLC, and data has been tested. For this an LCD output has also been generated:

```
1 char debugMessage[80];
2 sprintf(debugMessage, "ID=%02x, DLC=%01x, DATA=%02x %02x %02x
    %02x %02x %02x %02x %02x", msgId, dlc, data[0], data[1],
    data[2], data[3], data[4], data[5], data[6], data[7]);
```

This `debugMessage` then has been displayed on the LCD. After this was working the filter of the CAN interpreter has been evaluated. Furthermore, when no FPGA is present, the bus load calculation of the MC had to be tested as well:

```
1 // check for dump
2 if (dataDump) {
3   saveMessage(newCANMessage);
4   interpretMessageFlag = 1;
5 }
6
7 // check if we have to dump this module, if we do a datadump
    then this is obsolete
```

```
8  if (!dataDump)
9    for (i = 0; i < maxModules; i++)
10     if (moduleNumber[i] == msgId && moduleDump[i]) {
11       saveMessage(newCANMessage);
12       interpretMessageFlag = 1;
13     }
14
15 // find id
16 for (i = 0; i < maxModules; i++)
17   if (moduleNumber[i] == msgId) {
18     found = i;
19     break;
20   }
21
22 // add can message to corresponding module
23 moduleLoad[found] += length;
```

Lines 2-13 show the code for checking if the message needs filtering or if a data dump has to be created, while lines 16-23 show the load calculation. For evaluation of this the values of the variables have been displayed on the LCD. If the `interpretMessageFlag` has been raised the CAN message needs interpretation by the PDCP interpreter module. This has also been tested by displaying output on the LCD:

```
1 if (interpretMessageFlag) {
2   printText(DEBUG, "Calling␣PDCP␣interpreter");
3   interpretPDCP(data, msgPriority, msgMode, msgId, dlc);
4   generateCANStatistics(msgId, length);
5 }
```

The UART module has been evaluated in a similar way: as it only has to capture four consecutive UART frames it is waiting for the start signal first. If the start signal was received an output on the LCD has been generated. After that 32 four byte tuples have been captured and after every four byte tuple a flag for the statistics module has been raised:

```
1 if (UARTMessagePointer == 4) {
2   printText(DEBUG, "4␣bytes␣received");
3   UARTMessagePointer = 0;
4   UARTMessageArrived = 1;
5 }
```

As the SD card driver provides high-level functions for reading config files these needed to be evaluated as well. The file `config.txt` contains an option for setting the maximum amount of modules to be watched, `maxModules`, its successful reading from the file has been tested as follows:

```
1 // check how many modules should be watched
2 maxModules = getConfigItemInt("maxmodules", text);
3 sprintf(debug, "maxmodules=%d", maxModules);
4 printText(DEBUG, debug);
```

The variable `text` contains the contents of the file `config.txt`, while the method `getConfigItemInt(...)` calls the method `getConfigItem(...)` and converts the returned string to an integer. The method `getConfigItem(...)` can be seen in the Appendix A.4 and searches through an input string for a given sequence.

Once the PDCP interpreter module is toggled by the CAN module it has to segment the messages according to the PDCP. The first byte of the data field is the corresponding PDCP function:

```
1 function = convertBitsToShort(data[0]);
```

To be able to evaluate this the function code has also been displayed on the LCD, e.g. when the BA sends a device beacon the display should read `0A` as this is the PDCP function code for a device beacon. After this was working the next step was to evaluate different messages: at first the Bind Device Request was evaluated. For this function six bytes of data has been sent along the function code. The first two bytes are the vendor ID, the second two bytes the product ID, and the last two bytes the serial number:

```
1 switch (function) {
2         case 0x01: // Bind Device Request
3             // device vendor id
4             i1 = (int) convertBitsToShort(data[1]) << 8 | (int)
                  convertBitsToShort(data[2]);
5             // device product id
6             i2 = (int) convertBitsToShort(data[3]) << 8 | (int)
                  convertBitsToShort(data[4]);
7             // device serial number
8             i3 = (int) convertBitsToShort(data[5]) << 8 | (int)
                  convertBitsToShort(data[6]);
```

```
9
10              sprintf(action, "%s: Bind Device Request VID=%x,
                    PID=%x, SN=%x", getDevice(msgId), i1, i2, i3);
11              break;
12
13          ...
14  }
```

In order to not waste too much memory only four integer variables have been reserved for the interpretation of all PDCP functions. Therefore every function code makes use of the same four variables `i1`, `i2`, `i3`, and `i4`. As the vendor ID, product ID, and serial number are 16 bits wide the first byte needs to be shifted eight bits to the left and the results need to be casted to an integer, as a short is only eight bits wide. The method `getDevice(...)` looks up the sender's node ID in a table that has been created by the user as well as the program if the node IDs are unknown and returns their name as a string if the device is known. If the device is not known it returns its node ID in hexadecimal format. The output of this interpretation is then saved into the `action` string which is displayed by the statistics module on the LCD. The remaining PDCP function codes were evaluated in similar manner.

When the sample period is hit a flag is raised by the interrupt to signal the statistics module to sample the data and create statistics as well as the output. This has been tested by displaying a message on the LCD whenever the sample time has been hit. After that it formats the last CAN and UART messages into a string for LCD output and calculates the overall bus load during that sample period. After that it displays the last CAN and UART message, as well as the overall bus load and the last PDCP action that happened on the bus on the LCD. This behaviour has been used to evaluate the correctness of the reception of CAN and UART messages, as well as for the calculation of the overall load and the PDCP interpretation.

The final step for the high-level interpretation is writing of the statistics. In the file `log_`$x$`.txt` (with $x$ being an integer number) the dump of the CAN

messages and the PDCP interpretation is done and in the file `stat_`$x$`.txt` (with $x$ being an integer number) the bus load statistics with overall bus load and individual module load are written. This has been checked for correctness by putting different modules on the bus and checking the output files according to the modules.

## 6.4 Results

After every part has been evaluated separately the whole system can be evaluated. For this purpose the FPGA was hooked up to the CAN bus with the BA, a data monitor, and two electrodes connected. The board containing the BA and the data monitor can be seen in Figure 6.5. The board has been designed to be directly hooked up to the expansion header of the MEB as well as to hook up the FPGA via UART.

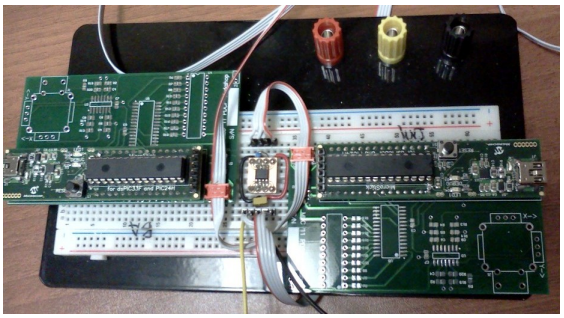

**Figure 6.5:** The Bus Arbitrator (left) and a data monitor (right) connected via a CAN bus network.

During startup of the components every component registers itself with the BA to receive its node ID. Thus these messages needed to be received by the FPGA and the MC. Furthermore, every node sends a device beacon every $500ms$ which has to be captured as well. The FPGA had to calculate the bus load data according to these messages and the MC has to interpret the messages and write the corresponding statistics and receive messages via the UART from the FPGA. To help verify the correctness of the calculations of both the MC and the FPGA the USBee DX has been used.

To ensure the correctness of the FPGA and the MC the USBee DX has been set to capture the first second after startup by setting a trigger to start capturing once the first signal fall occurs on the CAN bus. At the same time the display output on both the FPGA and the MC has been watched. After that the bus load has been manually calculated by checking every message on USBee DX's output and then compared against the output on both LCDs.

Furthermore, different tests have been conducted in the middle of the runtime. Every test captured one second, as this is the sample rate for the FPGA and the MC. The results were then compared against the manually calculated values as well. Finally, the log and statistics files had to be evaluated on a live system. For this purpose the above test scenarios were used again. After every test the microSD card has been removed from the MC and has been connected to a PC to review the written files. During manual review of the files it has been checked that every message that has been sent on the bus and that needed monitoring and interpretation was actually written in the logging file and that every second the overall bus load and the individual module's bus load was written. An image of the running program in normal execution mode can be seen in Figure 6.6.


**Figure 6.6:** The microcontroller program during normal execution.

# 7 Conclusion and Outlook

This report has shown the successful implementation of a MC that is capable of reading and writing on the CAN bus. This MC is also capable of interpreting the PDCP, which is a high-level protocol. The CAN bus itself operates on ISO OSI layer 1 and 2, while the PDCP can be set on any bus as it operates on ISO OSI layer 3. For the MC the Microchip dsPIC33E USB Starter Kit with the Microchip MEB attached to it has been used. The MC design and implementation has been done completely in Microchip MPLAB X. Furthermore, this report has shown the porting of the FPGA design from the Altera DE2 board to the Altera DE0-nano board. The FPGA can be used for monitoring the CAN bus and can be configured to monitor different modules simultaneously.

As most bus analyzers are only capable of displaying raw message data this project introduced an analyzer which can be configured depending on the nodes that are connected to the system as well as on the messages that are sent on this network. This analyzer displays messages during runtime on a built-in LCD and saves the messages received in the network according to pre-defined filters on a microSD card, so that the logged data can later be viewed offline on a PC.

This project can be further extended in terms of configurability, so that it is possible for the engineer to specify the CAN bus bitrate in the configuration file, as well as to set the system clock differently, as using different bus systems might require a system clock higher than 32 MHz. Furthermore, also having an option to specify which bus system is used might be helpful, so that the project does not need to be rebuilt every time a different bus system is used.

As the PDCP has been designed to be an open protocol other applications could also implement the PDCP as well. For instance in the automotive industry the PDCP could be used to make it easier for engineers to establish communication between different nodes. Nodes could create communication

channels with every node that they need to talk to and then communicate only using these channel IDs. With this the receiving node directly knows which node has sent the message. Using the tools shown in this report the engineer can directly connect these devices to the CAN bus and specify the configuration parameters as well as the list of the devices on the bus. The engineer would then be able to see online what is happening on the bus in real-time and if the devices are known also see which devices are communicating. Furthermore different messages can also be specified, so that if a node sends a specific bit combination in the data field it is displayed as e.g. `proximity_sensor1 to braking_system: obstacle in 20m`. As every action can be logged it is also possible to have different runs either in simulation or in real-life usage and then review the communication as CAN messages as well as interpreted messages offline on a PC for tuning system or node parameters.

The PDCP is not only limited to the CAN bus, but can be implemented atop other bus systems as well, e.g. for I2C. The Microchip dsPIC33E features a I2C controller, but at this time no driver is given for this bus. Thus, if needed, a driver needs to be implemented first. If then a new I2C message arrives it could directly be passed to the PDCP interpreter with no further changes. Also other bus systems can be used, which do not necessarily need to have a controller in the microcontroller. Then the input and output pins need to be read and written by the software engineer according to the underlying bus system. This can be achieved using interrupts on the microcontroller.

# A Appendix

## A.1 PDCP Commands

| Function Code | Function Code Description | Message Size (Bytes) | Sender | Recipient | Response Function Code |
|---|---|---|---|---|---|
| 0x01 | Bind Device Request | 7 | Device | Bus Arbitrator | 0x81 |
| 0x02 | GetDeviceInfo | 2 | Bus Arbitrator | Device | 0x82 |
| 0x03 | Get Device Parameter | 3 | Bus Arbitrator | Device | 0x83 |
| 0x04 | Set Device Parameter | 4-7 | Bus Arbitrator | Device | 0x84 |
| 0x05 | IndGetDeviceParameter | 4 | Device | Bus Arbitrator | 0x85 |
| 0x06 | IndSetDeviceParameter | 5-8 | Device | Bus Arbitrator | 0x86 |
| 0x07 | SetNodeId | 2 | Bus Arbitrator | Device | 0x87 |
| 0x08 | Suspend Device | 3 | Bus Arbitrator | Device | 0x88 |
| 0x09 | Release Device | 1 | Bus Arbitrator | Device | 0x89 |
| 0x0A | Device Beacon | 1 | D or BA | BA or D | None |
| 0x0B | Reset Device | 1 | Bus Arbitrator | Device | 0x8B |
| 0x0C | Configure Get Bulk Data Transfer | 5 | Bus Arbitrator | Device | 0x8C |
| 0x0D | Configure Set Bulk Data Transfer | 5 | Bus Arbitrator | Device | 0x8D |
| 0x0E | Bulk Data Transfer | 3-8 | D or BA | BA or D | 0x8E |
| 0x0F | Update Data Channel | 2 | Device | Bus Arbitrator | 0x8F |
| 0x10 - 0x4F | Reserved for Future System Commands | N/A | N/A | N/A | N/A |
| 0x50 - 0x7F | Reserved for Module-Specific Commands | N/A | N/A | N/A | N/A |
| 0x81 | Bind Device Request Response | 8 | Bus Arbitrator | Device | None |
| 0x82 | GetDeviceInfoAck | 2-6 | Device | Bus Arbitrator | None |
| 0x83 | Get Device Parameter Response | 4-8 | Device | Bus Arbitrator | None |
| 0x84 | Set Device Parameter Response | 5-8 | Device | Bus Arbitrator | None |
| 0x85 | IndGetDeviceParameterAck | 4-7 | Bus Arbitrator | Device | None |
| 0x86 | IndSetDeviceParameterAck | 1 | Bus Arbitrator | Device | None |
| 0x87 | SetNodeIdAck | 1 | Device | Bus Arbitrator | None |
| 0x88 | Suspend Device Response | 1 | Device | Bus Arbitrator | None |
| 0x89 | Release Device Response | 1 | Device | Bus Arbitrator | None |
| 0x8B | Reset Device Response | 1 | Device | Bus Arbitrator | None |
| 0x8C | Configure Get Bulk Data Transfer Response | 6 | Device | Bus Arbitrator | None |
| 0x8D | Configure Set Bulk Data Transfer Response | 6 | Device | Bus Arbitrator | None |
| 0x8E | Bulk Data Transfer Response | 2 | D or BA | BA or D | None |
| 0x8F | Update Data Channel Response | 3 | Bus Arbitrator | Device | None |
| 0x90 - 0xCF | Reserved for Future System Responses | N/A | N/A | N/A | N/A |
| 0xD0 - 0xFE | Reserved for Device-Specific Responses | N/A | N/A | N/A | N/A |

**Figure A.1:** An overview of all available PDCP commands[17].

## A.2 Overview of the IVT of the dsPIC33E Family

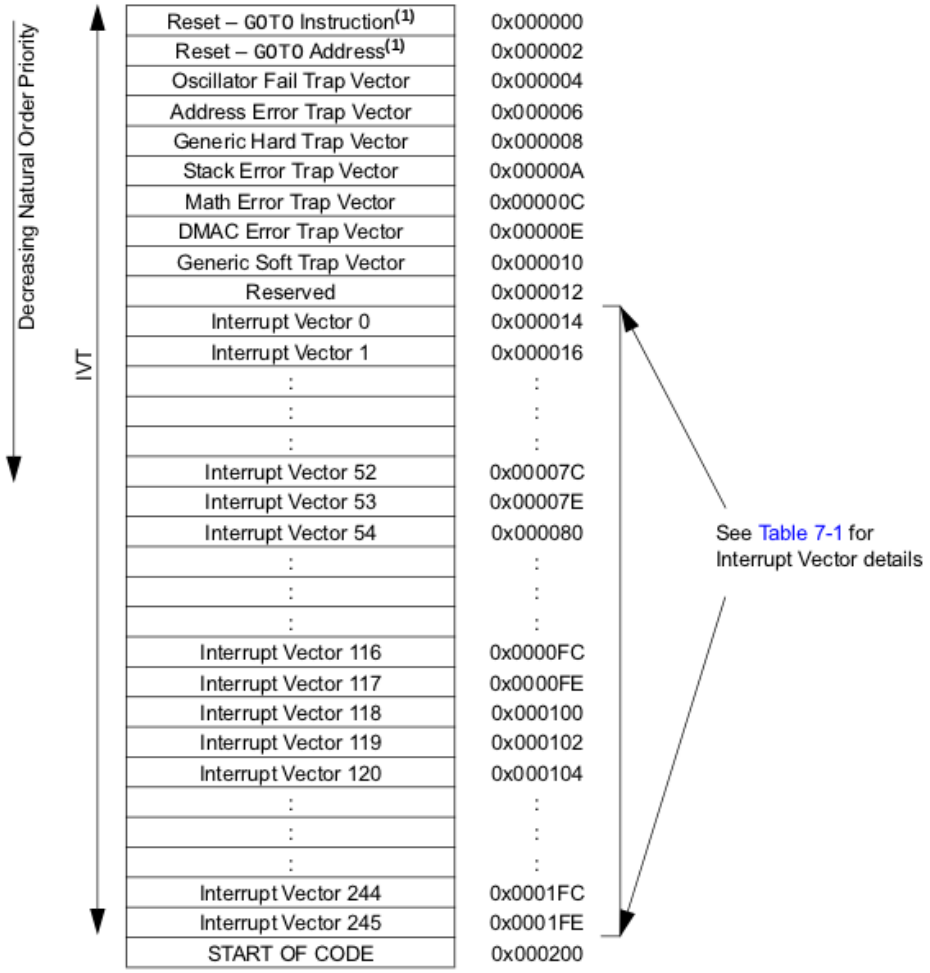


**Figure A.2:** Compact overview of the IVT of the dsPIC33E family[4].

## A.3 Code Excerpt for Updating the Values on the LCD

Code excerpt from `LCD.c`:

```
1  // remove old message
2  SetColor(BLACK);
3  SetFont((void *) &GOLSmallFont);
4  while (!OutTextXY(0, 52, oldCANMessage));
5  while (!OutTextXY(0, 92, oldUARTMessage));
6  while (!OutTextXY(0, 132, busLoad));
7  while (!OutTextXY(0, 172, lastAction));
8
9  // display the messages as hex
10 sprintf(oldCANMessage, "%02x␣%02x␣%02x␣%02x␣%02x␣%02x␣%02x␣%02x
       ␣%02x␣%02x␣%02x␣%02x␣%02x␣%02x␣%02x␣%02x␣%02x␣%02x",
11   convertBitsToShort(newCANMessage[0]),
12   convertBitsToShort(newCANMessage[1]),
13   convertBitsToShort(newCANMessage[2]),
14   convertBitsToShort(newCANMessage[3]),
15   convertBitsToShort(newCANMessage[4]),
16   convertBitsToShort(newCANMessage[5]),
17   convertBitsToShort(newCANMessage[6]),
18   convertBitsToShort(newCANMessage[7]),
19   convertBitsToShort(newCANMessage[8]),
20   convertBitsToShort(newCANMessage[9]),
21   convertBitsToShort(newCANMessage[10]),
22   convertBitsToShort(newCANMessage[11]),
23   convertBitsToShort(newCANMessage[12]),
24   convertBitsToShort(newCANMessage[13]),
25   convertBitsToShort(newCANMessage[14]),
26   convertBitsToShort(newCANMessage[15]));
27
28 sprintf(oldUARTMessage, "%02x␣%02x␣%02x␣%02x",
29   convertBitsToShort(newUARTMessage[0]),
30   convertBitsToShort(newUARTMessage[1]),
31   convertBitsToShort(newUARTMessage[2]),
32   convertBitsToShort(newUARTMessage[3]));
33
34 load = 1.0 * overallLoad / 1000;
35 loadPercent = 1.0 * overallLoad / MAXLOAD / SAMPLERATE;
36 sprintf(busLoad, "%04.2f␣kbit/s,␣%03.2f%%", load, loadPercent);
37 sprintf(lastAction, "%s", action);
```

```
38
39 // put new message on the lcd
40 SetColor(RED);
41 SetFont((void *) &GOLSmallFont);
42 while (!OutTextXY(0, 52, oldCANMessage));
43 while (!OutTextXY(0, 92, oldUARTMessage));
44 while (!OutTextXY(0, 132, busLoad));
45 while (!OutTextXY(0, 172, lastAction));
46
47 // display if we are registered with the ba
48 if (successfullyBound) {
49   SetColor(YELLOW);
50   SetFont((void *) &GOLSmallFont);
51   while (!OutTextXY(0, 206, "Registered␣with␣BA"));
52 }
```

## A.4 Searching for a Substring in a String

Code excerpt from `SDFSIO.c`:

```
1  /*
2   * read the item from the config file
3   * the return value is the value of the item read
4   */
5  char *getConfigItem(char *item, char *config) {
6      int i, k, l, m = 0, n = 0;
7      int j = 0;
8      int size = 0;
9      static char search[255];
10     short found = 0;
11
12     // get size of item
13     for (i = 0; i < 255; i++)
14         if (item[i] == '\0') {
15             size = i;
16             break;
17         }
18
19     for (i = 0; i < 255; i++) {
20         // check for match
21         if (config[i] == item[j]) {
22             if (!j)
23                 n = i;
24             j++;
25         } else
26             j = 0;
27
28         // if we have found the complete string
29         if (j == size && config[i + 1] == '=') {
30             found = 1;
31             break;
32         }
33     }
34
35     if (found) {
36         for (k = n + j; k < 255; k++) {
37             if (config[k] == '\n' || config[k] == '\r') {
38                 for (l = n + j + 1; l < k; l++)
```

```
39                      search[m++] = config[l];
40                  search[m] = '\0';
41                  return search;
42              }
43          }
44      }
45
46      return "NOT FOUND";
47  }
```

# References

[1] Y. Losier, A. Clawson, and A. Wilson, "An Overview Of The UNB Hand System," 2011.

[2] Y. Losier and A. Wilson, "Moving towards an open standard: The UNB prosthetic device communication protocol," *Proc. ISPO World Congr*, 2010.

[3] M. Barr, "Embedded systems glossary," *Netrino Technical Library. Retrieved on*, pp. 04–21, 2007.

[4] Microchip, *dsPIC33EPXXXMU810 Data Sheet*. 2011.

[5] "`http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en554619`," Access Date: 17/01/2012.

[6] "`http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2615&dDocName=en548037`," Access Date: 17/01/2012.

[7] Microchip, "Overview and Use of the PICmicro Serial Peripheral Interface," `http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf`, Access Date: 23/05/12.

[8] "`http://www.ieee802.org/`," Access Date: 23/05/12.

[9] "`http://www.zigbee.org/`," Access Date: 23/05/12.

[10] "`http://www.altera.com/education/univ/materials/boards/de0-nano/unv-de0-nano-board.html`," Access Date: 29/12/2011.

[11] M. Dombrowski, Y. Losier, K. B. Kent, and R. Herpers, "Monitoring Bus Load of an Open Bus Standard-based Prosthetic Limb System," UNB Technical Report TR11-212, 2011.

[12] H. Zimmermann, "OSI reference modelThe ISO model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 1980.

[13] K. Pazul, "Controller Area Network (CAN) Basics," *Microchip Technology Inc. Preliminary DS00713A-page*, vol. 1, 1999.

[14] "`http://standards.ieee.org/about/get/802/802.3.html`," Access Date: 27/02/2012.

[15] Bosch, "CAN Specification," 1991.

[16] P. Chu, "FPGA prototyping by Verilog examples: Xilinx Spartan-3 version," *Interface*, pp. 215–234, 2008.

[17] Y. Losier, "Prosthetic Device Communication Protocol for the AIF UNB Hand Project," *Draft Copy*, 2011.

[18] A. Bochem, J. Deschenes, J. Williams, K. B. Kent, and Y. Losier, "FPGA Design for Monitoring CANbus Traffic in a Prosthetic Limb Sensor Network," *RSP paper*, 2011.

[19] H. Kashif, G. Bahig, and S. Hammad, "CAN bus analyzer and emulator," in *Design and Test Workshop (IDT), 2009 4th International*, pp. 1–4, IEEE, 2009.

[20] R. Li and C. Liu, "A design for automotive CAN bus monitoring system," *Vehicle Power and Propulsion Conference*, pp. 1–5, Sept. 2008.

[21] M. Mostafa, M. Shalan, and S. Hammad, "FPGA-Based Low-level CAN Protocol Testing," in *System-on-Chip for Real-Time Applications, The 6th International Workshop on*, pp. 185–188, IEEE, 2006.

[22] J. Yang, T. Zhang, J. Song, H. Sun, G. Shi, and Y. Chen, "Redundant design of A CAN BUS Testing and Communication System for space robot arm," in *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, no. December, pp. 1894–1898, IEEE, 2008.

[23] J. Mendozajasso, G. Ornelasvargas, R. Castanedamiranda, E. Venturaramos, a. Zepedagarrido, and G. Herreraruiz, "FPGA-based real-time remote monitoring system," *Computers and Electronics in Agriculture*, vol. 49, pp. 272–285, Nov. 2005.

[24] C. Zamantzas, B. Dehning, E. Effinger, J. Emery, and G. Ferioli, "An FPGA Based Implementation for Real-Time Processing of the LHC Beam Loss Monitoring System's Data," *2006 IEEE Nuclear Science Symposium Conference Record*, pp. 950–954, Oct. 2006.

[25] M. Dombrowski, K. B. Kent, Y. Losier, A. Wilson, and R. Herpers, "Analyzing Bus Load Data Using an FPGA and a Microcontroller," in *DSD*, unpublished, 2012.

[26] "`http://www.altera.com/education/univ/materials/boards/de2/unv-de2-board.html`," Access Date: 29/06/2011.

[27] "`http://www.altera.com/education/univ/materials/boards/de0-nano/unv-de0-nano-board.html`," Access Date: 07/012012.

[28] "`http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html`," Access Date: 11/03/2012.

[29] "`http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010065`," Access Date: 24/02/2012.

[30] "`http://www.microchip.com/mplabx/`," Access Date: 15/03/2012.

[31] "`http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2680&dDocName=en537999`," Access Date: 01/03/2012.

[32] D. Matrix, L. Crystal, and D. Controller, "Hd44780u (lcd-ii)," pp. 167–226.

[33] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, 1995.

[34] Microchip, *Multimedia Expansion Board User's Guide*. 2010.

[35] "`http://usbee.com/dx.html`," Access Date: 20/03/2012.

[36] 411 Technology Systems, "Data Sheet for SSC2F16DLNW-S," 2008.

# B Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, at this or any other University.

City, Date, Signature Marcel Dombrowski