

# Monitoring Bus Load of an Open Bus Standard-based Prosthetic Limb System

by

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

**TR 11-212, November 30, 2011**

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](mailto:fcs@unb.ca)  
<http://www.cs.unb.ca>

## **Abstract**

Having multiple talkers on a bus system rises the bandwidth on this bus. To monitor the communication on a bus, tools that constantly read the bus are needed. This report shows an implementation of a monitoring system for the CAN bus utilizing the Altera DE2 development board. The Biomedical Institute of the University of New Brunswick is currently developing together with different partners a prosthetic limb device, the UNB hand. Communication in this device is done via two CAN buses, which operate at a bit-rate of 1 Mbit/s. The developed monitoring system has been completely designed in Verilog HDL. It monitors the CAN bus in real-time and allows monitoring of different modules as well as of the overall load. The calculated data is displayed on the built-in LCD and also transmitted via UART to a PC. A sample receiver programmed in C is also given. The evaluation of this system has been done by using the Microchip CAN Bus Analyzer Tool connected to the GPIO port of the development board that simulates CAN communication.

## Contents

<b>1</b>	<b>Introduction and Motivation</b>	<b>1</b>
<b>2</b>	<b>Basics</b>	<b>3</b>
2.1	FPGA . . . . .	3
2.2	HDL . . . . .	6
2.3	IP Cores . . . . .	8
2.4	CAN Bus . . . . .	9
2.5	Real-time . . . . .	15
2.6	Floating Point Numbers Representation . . . . .	15
2.7	UART . . . . .	17
<b>3</b>	<b>Related Work</b>	<b>19</b>
3.1	Bus Monitoring . . . . .	19
3.2	Existing IP Cores . . . . .	20
3.3	Problem Formulation . . . . .	21
<b>4</b>	<b>Design</b>	<b>23</b>
4.1	Requirements . . . . .	23
4.2	Design Decisions . . . . .	24
4.2.1	Hardware . . . . .	24
4.2.2	Software . . . . .	26
<b>5</b>	<b>Implementation</b>	<b>29</b>
5.1	Usage of IP Cores . . . . .	29
5.2	Architecture . . . . .	33
5.2.1	Overview . . . . .	33
5.2.2	CAN Driver . . . . .	34
5.2.3	Load Calculation . . . . .	36
5.2.4	Output Conversion . . . . .	38
5.2.5	Display Module . . . . .	40
5.2.6	UART Module . . . . .	41
5.2.7	PC Client . . . . .	44
5.3	Interaction of the Modules . . . . .	45

<b>6</b>	<b>Evaluation</b>	<b>49</b>
6.1	Approach . . . . .	49
6.2	Output Module . . . . .	49
6.2.1	Verification of the LCD . . . . .	49
6.2.2	Verification of the UART Module . . . . .	50
6.2.3	Verification of the Floating Point Division . . . . .	52
6.3	Monitor Module . . . . .	52
6.3.1	Verification of the CAN Driver . . . . .	52
6.3.2	Verification of the Load Calculation Module . . . . .	54
6.4	Results . . . . .	55
<b>7</b>	<b>Conclusion and Outlook</b>	<b>59</b>
<b>A</b>	<b>Appendix</b>	<b>62</b>
A.1	Data Frames on the CAN Bus . . . . .	62
A.2	Opening the UART Port in C++ . . . . .	64
A.3	The Selftest Module . . . . .	65
A.4	Groups of Messages Used for the Evaluation . . . . .	67
<b>B</b>	<b>Declaration of Authorship</b>	<b>72</b>

## List of Figures

1.1	The UNB Hand . . . . .	2
2.1	Programming Technologies for FPGAs and PLDs . . . . .	4
2.2	Basic Layout of a Logic Block . . . . .	5
2.3	The Interconnection of Logic Blocks in an FPGA . . . . .	5
2.4	Different Layers of Abstraction in an HDL . . . . .	6
2.5	Simple HDL-based FPGA Flow . . . . .	7
2.6	Comparison of Verilog and VHDL with Advantages and Dis- advantages . . . . .	8
2.7	Integration of IP into an FPGA Design . . . . .	9
2.8	The Interconnection of the 7 Layers of the ISO OSI Reference Model . . . . .	10
2.9	The Control Flow for the Transmission of a Frame on the example of an Ethernet Connection . . . . .	12
2.10	Bit Timing on the CAN Bus . . . . .	16
2.11	Transmission of a Byte . . . . .	18
3.1	The Front Page of the IP Store in the Altera Quartus II PLD Design Software . . . . .	20
4.1	The Altera DE2 Development and Education Board . . . . .	24
4.2	The Microchip CAN BUS Analyzer Tool . . . . .	25
4.3	The Wiring of the FTDI TTL-232R-3V3 Cable . . . . .	26
4.4	A Snippet of the Microchip CAN BUS Analyzer Tool Software	27
5.1	The LCD Instance in the RTL Viewer . . . . .	30
5.2	The UART Module Instance in the RTL Viewer . . . . .	32
5.3	The Floating Point Division Module Instance in the RTL Viewer	32
5.4	The State Machine for the CAN Driver . . . . .	35
5.5	The CAN Driver Instance in the RTL Viewer . . . . .	36
5.6	The State Machine for the Load Calculation Algorithm . . . .	38
5.7	The Conversion of an Integer Value into the IEEE754 Format	40
5.8	The Basic Layout of the PC Client . . . . .	46
5.9	An UML-like Class Diagram for the Interaction of the Moni- tor and Output Part . . . . .	46

*LIST OF FIGURES*

---

5.10 An UML-like class diagram for the Interaction Inside the Monitor Part . . . . .	47
5.11 An UML-like Class Diagram for the Interaction Inside the Output Part . . . . .	47
6.1 The Output on the LCD . . . . .	56
6.2 The Output on the PC . . . . .	57
A.1 A Standard Data Frame on the CAN Bus . . . . .	62
A.2 An Extended Data Frame on the CAN Bus . . . . .	63

**List of Tables**

1	Comparison of Different Programming Technologies on FPGAs	4
2	The Truth Table for the Logical AND Operator . . . . .	11
3	The First Group of CAN Messages Used for the Evaluation .	56
4	The Second Group of CAN Messages Used for the Evaluation	67
5	The Third Group of CAN Messages Used for the Evaluation .	67
6	The Fourth Group of CAN Messages Used for the Evaluation	68

## List of Acronyms

<b>ASIC</b>	Application-Specific Integrated Circuit
<b>CAN</b>	Controller Area Network
<b>CPLD</b>	Complex PLD
<b>CRC</b>	Cyclic Redundancy Check
<b>CSMA/CD</b>	Carrier Sense Multiple Access with Collision Detection
<b>DSP</b>	Digital Signal Processing
<b>EDA</b>	Electronic Design Automation
<b>EEPROM</b>	Electrically Erasable Programmable ROM
<b>FPGA</b>	Field Programmable Gate Array
<b>FSM</b>	Finite State Machine
<b>GPIO</b>	General Purpose Input/Output
<b>HDL</b>	Hardware Description Language
<b>IC</b>	Integrated Circuit
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>IP</b>	Intellectual Property
<b>ISO</b>	International Organisation for Standardization
<b>JTAG</b>	Joint Test Action Group
<b>LCD</b>	Liquid Crystal Display
<b>LED</b>	Light Emitting Diode
<b>LUT</b>	Lookup Table
<b>PLD</b>	Programmable Logic Device



*LIST OF ACRONYMS*

---

<b>RAM</b>	Random Access Memory
<b>ROM</b>	Read-Only Memory
<b>RTL</b>	Register Transfer Level
<b>SPLD</b>	Simple PLD
<b>SRAM</b>	Static RAM
<b>UART</b>	Universal Asynchronous Receiver / Transmitter
<b>VHDL</b>	VHSIC Hardware Description Language
<b>VHSIC</b>	Very High Speed Integrated Circuit

---

## 1 Introduction and Motivation

The UNB hand is an anthropomorphic prosthetic device developed by the Institute of Biomedical Engineering and the Applied Nanotechnology Lab at the University of New Brunswick, the thin film research group at the University de Moncton and the Biomechatronics Development Lab at the Rehabilitation Institute of Chicago. It is funded by the Atlantic Canada Opportunities Agency. The hand itself features a movable thumb and fingers. It comes in different sizes for either a left hand or right hand configuration and is capable of performing different grasp patterns and wrist motions. Built up out of three components, the hand features a three-motor mechanical hand and control system, the glove as a cover for the hand, which also shields the interior, and the sensors that measure pressure, to make sure that the hand does not put too much pressure on the object, e.g. on a glass or an egg. An actual image of the hand can be seen in Figure 1.1.

The hand itself utilizes two CAN buses to pass messages along, one bus for the sensors and one for the actuators. On top of the CAN bus the newly proposed Prosthetic Device Communication Protocol (PDCP) has been placed, which operates mainly on layer 3 in the ISO OSI reference model. As CAN is the underlying communication system it has a maximum throughput of 1 Mbps. Logging and filtering of the messages is done with an FPGA. Debugging and decoding is mostly done with CAN analyzer tools, but they lack either the possibility to decode data captures in real-time or the possibility to monitor load that each module puts on the bus. Furthermore both buses are running almost at its limit of bandwidth.

To be able to detect what module puts how much load on the bus and to be able to see the traffic in real-time a new tool is proposed. This tool needs to be able to constantly monitor the bus without interfering communication on it while dynamically monitoring all the modules that are talking on the bus. For every module the load that it puts on the bus will be calculated and then displayed. Furthermore also the overall load is of interest; the ability to see, when the bus is actually running near its limit.

---

This report is structured as follows: in Chapter 2 the fundamental knowledge to understand this report is given. This includes an introduction to FPGAs, HDLs, IP cores, the CAN bus, the term real-time, the digital representation of floating point numbers, and UART. Chapter 3 deals with other research that has been conducted on the field of bus monitoring, not only limited to FPGAs. The design of the HDL model that has been created during this work is introduced in Chapter 4 and starts with the requirements. After that the specification of this solution is given, as well as the design decisions that have been made during the development stage. The following chapter, Chapter 5, describes the tools that have been used and explains in detail what has been done in order to achieve the desired goal. As the implementation has to be evaluated Chapter 6 shows how the whole HDL model has been tested. This report ends with the conclusion and outlook in Chapter 7 where also is explained what is left to do. Bigger Figures that influence the reading flow, as well as bigger code excerpts, appear in the appendix.



**Figure 1.1:** The UNB Hand [1].

---

## 2 Basics

### 2.1 FPGA

Maxfield describes in [2] a Field Programmable Gate Array (FPGA) as a digital integrated circuit (IC) which “contains configurable (programmable) blocks of logic along with configurable interconnects between these blocks” (p. 1). Besides FPGAs there is a big variety of digital ICs, ranging from microprocessors to programmable logic devices (PLDs), to application-specific integrated circuits (ASICs) and FPGAs. In the mid-1980s FPGAs have been primarily used to implement glue logic, which is some small logic that is used to connect two large logical blocks together [2]. Today FPGAs are used for a variety of tasks, which include [2]:






- ASIC functionality
- Digital Signal Processing (DSP)
- Microcontroller functionality
- Physical layer communication
- Reconfigurable computing

Creating an ASIC has two big negative points, first the logic that has been implemented cannot be changed after the creation and second ASICs are very expensive, as ASIC vendors do not sell single ASICs but large amounts. FPGAs can be used to implement the same designs that are running on an ASIC. DSP has been done using dedicated microprocessors, so called digital signal processors. But FPGAs can also contain embedded multipliers and accumulators and with the parallelism of an FPGA it can outperform traditional DSP chips by a factor of 500 or more [2]. Due to the large capacity of logic gates, FPGAs are also capable of running soft processor cores, which is a processor that has been realized using logic gates. Thus also embedded microcontroller applications are suitable for an FPGA. The ability of hardware accelerating software algorithms with an FPGA leads to reconfigurable computing, as e.g. it is possible with an FPGA to have hardware encryption.

## 2.1 FPGA

Feature	SRAM	Antifuse	EEPROM / FLASH
Technology node	State-of-the-art	One or more generations behind	One or more generations behind
Reprogrammable	Yes (in system)	No	Yes (in system or offline)
Reprogramming speed	Fast	—	3x slower than SRAM
Good for prototyping	Yes (very good)	No	Yes (reasonable)
Power consumption	Medium	Low	Medium

**Table 1:** Comparison of different programming technologies on FPGAs [2].

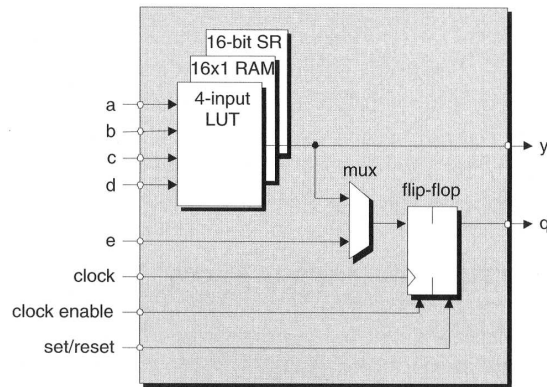
Technology	Symbol	Predominantly associated with ...
Fusible-link		SPLDs
Antifuse		FPGAs
EPROM		SPLDs and CPLDs
E <sup>2</sup> PROM/FLASH		SPLDs and CPLDs (some FPGAs)
SRAM		FPGAs (some CPLDs)

**Figure 2.1:** Programming technologies for FPGAs and PLDs [2].

There are different programming technologies available for FPGAs and PLDs, as Figure 2.1 shows. PLDs can be divided into Simple PLDs (SPLDs) and Complex PLDs (CPLDs). The difference between them is the amount of logic gates available. For FPGAs either SRAM, anti-fuses, or EEPROM / FLASH is used, the advantages and disadvantages of them are shown in Table 1.

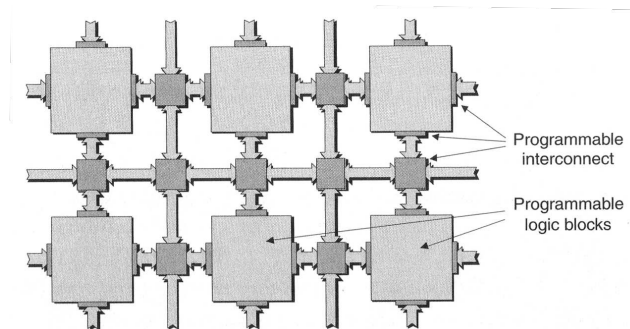
Logic blocks are named differently by each vendor. The biggest two vendors are Xilinx and Altera. Xilinx calls them Logic Cells (LCs), whereas Altera calls them Logic Elements (LEs) [2]. Each logic block contains a lookup ta-

ble (LUT), which is triggered by a different amount of inputs, a multiplexer and a D-FlipFlop, as can be seen in Figure 2.2.



**Figure 2.2:** Basic layout of a logic block [2].

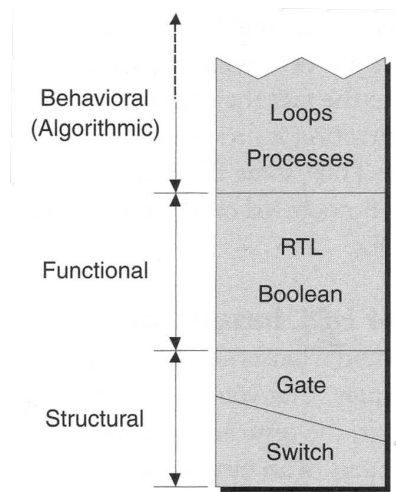
All logic blocks are arranged in a two-dimensional  $N \times M$  matrix. Each of the logic blocks is connected with an interconnect to its four direct neighbours, namely the one above, below, to the left, and to the right of it. The interconnections can be disabled and enabled as wanted and interconnects are interconnected to other interconnects. Thus it is possible for one logic block to communicate with other logic blocks that are not its direct neighbours. Figure 2.3 shows this.



**Figure 2.3:** The interconnection of logic blocks in an FPGA [2].

## 2.2 HDL

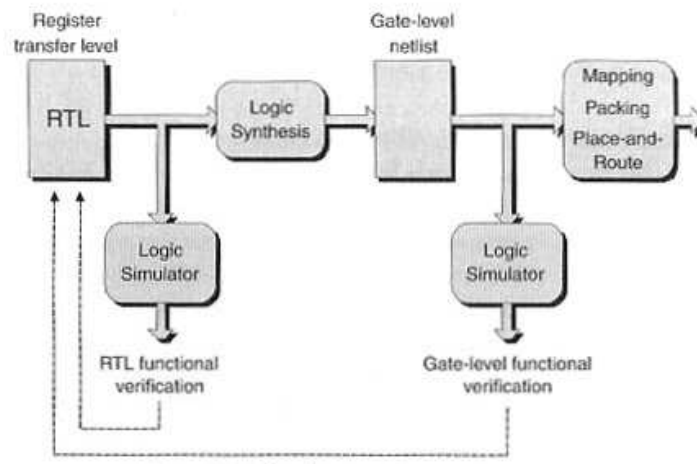
To be able to program an FPGA a hardware description language (HDL) is needed. While there are numerous languages available VHDL and Verilog are the most widely used ones. An HDL can be used to, as the name implies, describe hardware. In an HDL the term hardware “refers only to the electronic portions (components and wires) of ICs and printed circuit boards” [2]. In the beginning of electronics, most vendors who created electronic design automation (EDA) tools also created their own HDLs, some of them being analog HDLs, others digital [2].



**Figure 2.4:** Different layers of abstraction in an HDL [2].

In digital designs the functionality is represented in different layers of abstraction, as Figure 2.4 shows. As can be seen, the lowest layer, the structural layer, is made up of the gates and switches, which might also be addressed in an HDL. The middle layer, the functional layer, contains the register transfer level (RTL) representations, as well as boolean logic. RTL covers a “multitude of manifestations, [...] [the] concept is to consider a design formed from a collection of registers linked by combinational logic” [2]. The highest layer, the behavioural or algorithmic layer, consists of loops and processes, which might be needed to describe a design. This also includes algorithmic elements, like adders and multipliers.

The HDL-based FPGA flow from the RTL layer of abstraction to the finished HDL model can be seen in Figure 2.5. The RTL representation is converted to a gate-level netlist by the synthesis tool. Meanwhile the logic simulator verifies the RTL functionality. The gate-level netlist is a functional verification from the logic simulator and is mapped, packed, and placed-and-routed to the final HDL model.



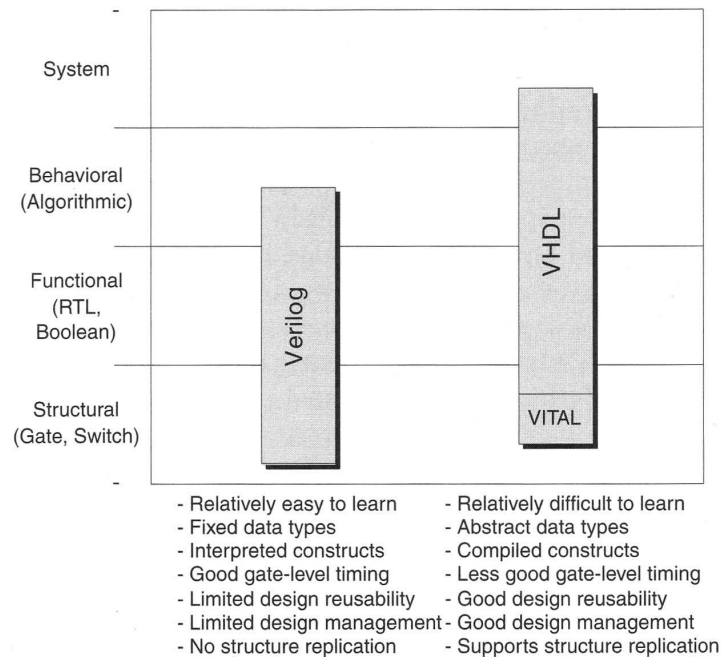
**Figure 2.5:** Simple HDL-based FPGA flow [2]

VHDL and Verilog are the most popular HDLs. Verilog was created in the mid-1980s by Phil Moorby. In 1985 his company, Gateway Design Automation, released the language to the market as well as the logic simulator Verilog-XL [2]. Verilog is able to work on all three layers of the layer of abstraction model introduced before.

VHDL on the other hand was also released in 1985, as in 1980 “the US Department of Defense launched the very high speed integrated circuit (VHSIC) program, whose primary objective was to advance the state of the art in digital IC technology” [2]. Out of this program in 1981 the VHSIC HDL (VHDL) project was established. VHDL is strong on the functional layer, but rather weak in the structural layer [2]. Furthermore it also supports



some system-level design constructs. Figure 2.6 shows a comparison of Verilog and VHDL with some advantages and disadvantages.



**Figure 2.6:** Comparison of Verilog and VHDL with advantages and disadvantages [2].

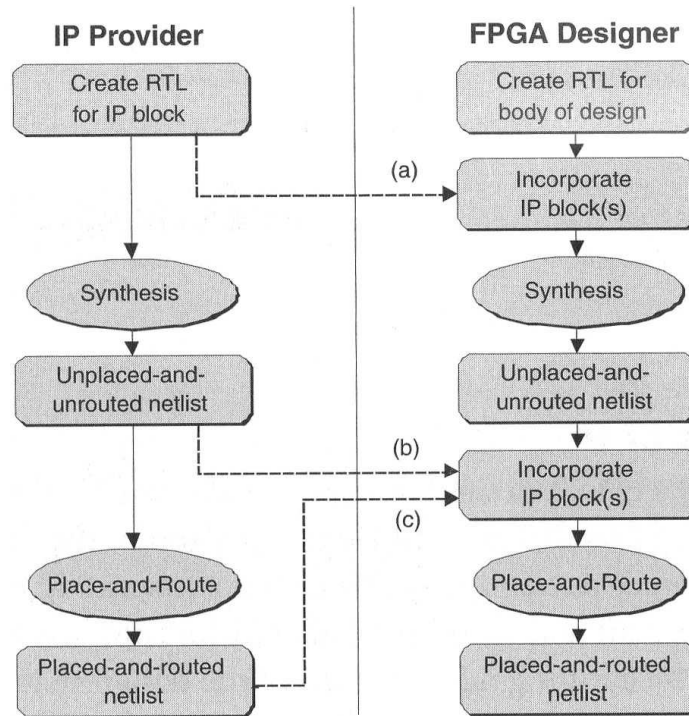
## 2.3 IP Cores

As FPGA designs can be very big and complex it is impractical to always start from scratch. Thus reusability of existing parts is needed. This can be either done by integrating previous implementations into the current HDL model or by using intellectual property (IP). There are three main sources for IP:

- Previous designs
- FPGA vendors
- Third-party IP providers

IP can be distributed encrypted or unencrypted, depending on whether the provider wants the FPGA designer to see the actual implementation. Figure

2.7 shows how IP can be integrated into the FPGA design.



**Figure 2.7:** Integration of IP into an FPGA design, (a) unencrypted, (b) encrypted at the unplaced-and-unrouted netlist level, and (c) encrypted at the placed-and-routed netlist level [2].

If the vendor delivers the IP unencrypted the FPGA designer can directly integrate the RTL into their code, as seen in Figure 2.7 (a). The most common part for designers to purchase IP is at the encrypted and unplaced-and-unrouted netlist level, they can be integrated into the FPGA design as seen in Figure 2.7 (b) [2]. Sometimes only placed-and-routed netlist IPs are available. They are integrated at the same part in the FPGA design as the unplaced-and-unrouted netlists (Figure 2.7 (c)).

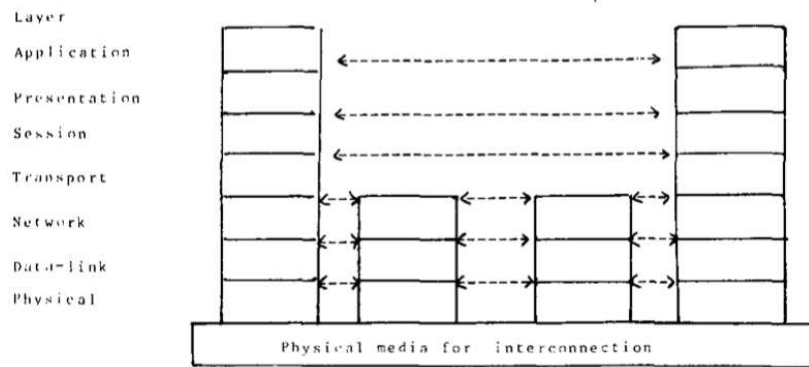
## 2.4 CAN Bus

The Controller Area Network (CAN) bus was an internal project from Bosch, whose development started in 1983 and was officially introduced in 1986. In

1987 the first CAN controller chips were released by Intel and Philips Semiconductors. Bosch released the updated specification 2.0 in 1991 [3].

The primary goal of the CAN project was to make automobiles “more reliable, safe and fuel-efficient, while decreasing wiring harness weight and complexity” [4]. The first automotive manufacturer that adopted the CAN bus technology has been Mercedes in the 1991 released S-class, other manufacturers were then following, including Volvo, Saab, BMW, and Volkswagen [5]. By 2004 there were at least “50 different microprocessor families with on-chip CAN capability” [5]. By 2007 almost every new car sold in Europe had at least one CAN bus onboard and by 2008 the US Environmental Protection Agency mandated that every newly built car or truck has to be equipped with a CAN bus for on-board diagnostics [5].

The ISO OSI Reference Model as proposed in [6] has a layered approach for system implementation. The model itself is divided into seven layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application. The interconnection of these layers can be seen in Figure 2.8. The CAN bus implements most of the two bottom layers, namely the Phys-



**Figure 2.8:** The interconnection of the 7 layers of the ISO OSI Reference Model [6]

ical Layer and the Data Link Layer. The specification of the transmission medium was left out on purpose to allow system designers to use their own medium for transmission [4].

AND	0	1
0	0	0
1	0	1

**Table 2:** The truth table for the logical AND operator.

As communication on a bus can always result in multiple nodes trying to access and write on the bus at the same time and therefore cause a collision, a protocol is needed to avoid this unwanted behaviour. Therefore the CAN bus makes use of the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol [4], as originally defined in the IEEE 802.3 standard, which handles Ethernet communication as well as the CSMA/CD protocol for collision detection [7]. The control flow for the transmission of a frame on the example of an ethernet connection can be seen in Figure 2.9.

The transmission of bits on the bus is based on dominant and recessive bits which are connected with the logical AND operator. A dominant bit is the logical 0 and a recessive bit is the logical 1. Table 2 shows the truth table for the logical AND operator, which shows why 0 is the dominant bit and 1 the recessive bit. When there is no communication on the bus a constant stream of recessive bits is sent.

The CAN bus is a message-based protocol, which means that messages contain an identifier, which also serves as the priority, as well as the data itself [4]. All nodes connected to the CAN bus receive all messages and the nodes themselves have to decide whether to keep the message for further processing or not. There are four different types of messages defined in the CAN protocol:

- Data Frame
- Remote Frame
- Error Frame

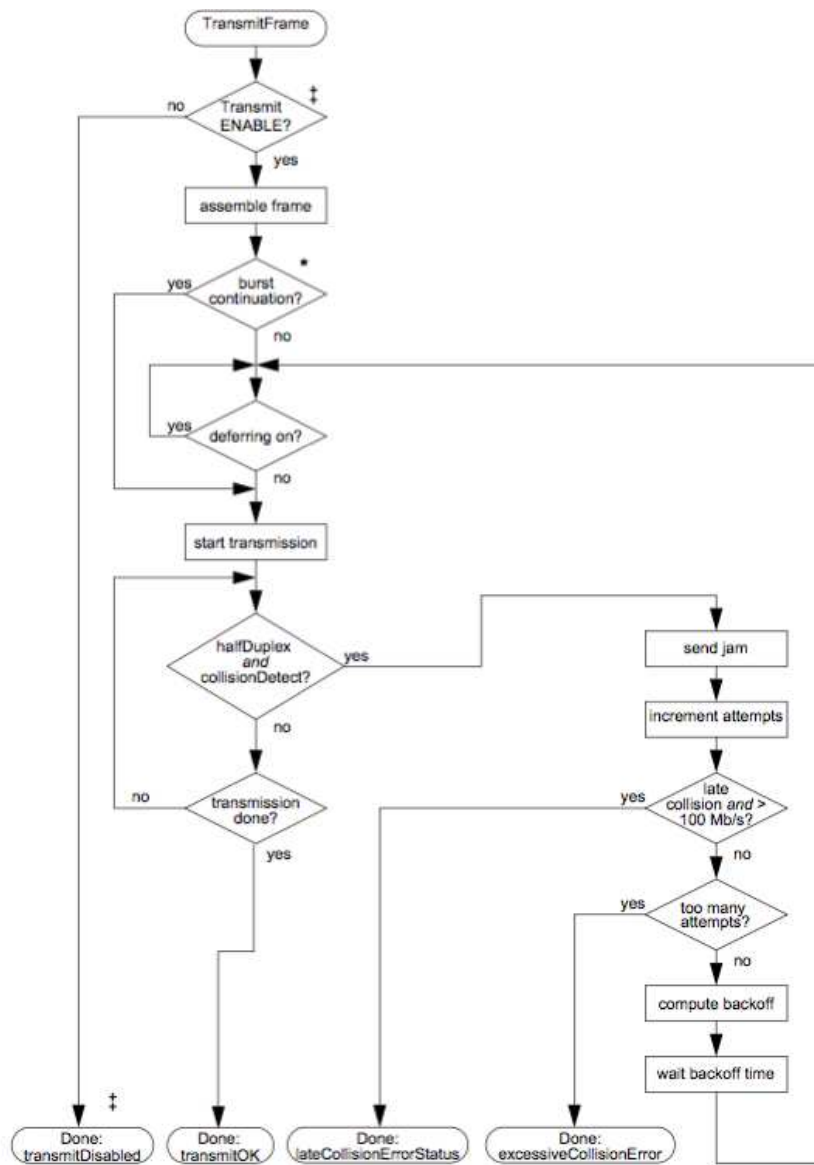


Figure 2.9: The control flow for the transmission of a frame on the example of an ethernet connection [7].

- Overload Frame

Every frame starts with the “Start of Frame” bit, which is a dominant bit, to signal the nodes that a new frame is being sent. The data frame is the most important frame. After the “Start of Frame” field the “Arbitration Field” is sent, which consists of 12 bits, where the first 11 bits are the identifier in least significant bit (LSB) endianness and the last bit is the “Remote Transmit Request” (RTR) bit, which is set if a node is requesting another node to transmit. If this bit is set, the frame is a remote frame. After the “Arbitration Field” the “Control Field” is sent. The first bit is the “Identifier Extension Bit” (IDE), which is set to a recessive bit if the node ID needs more than 11 bits. If the IDE is set, the RTR bit also has to be set to the dominant bit and serves as the “Substitute Remote Request” (SRR) bit. The RTR bit will follow then after the “Extended Identifier”. The whole frame then will be an extended data frame. In the data frame the bit after the IDE is a reserved bit (RB0) and has to be sent dominant, but also recessive is accepted. The last 4 bits of the “Control Field” make up the “Data Length Code” (DLC) in which is stored how many byte data will be sent. Values of 0 to 8 describe the corresponding bytes of data and values greater than 8 will result in 8 bytes of data. The next 0 to 64 bits then will make up the “Data Field”. Depending on the DLC, 0 to 64 bits (or 8 bytes) will be transmitted. After the “Data Field” 16 bits for the “CRC Field” are following. The first 15 bits make up the Cyclic Redundancy Check (CRC) bits, which are computed by the CRC-15-CAN algorithm after the following polynomial:

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

With this polynomial it is possible to detect, not correct, up to 5 single bit errors in the “Start of Frame”, “Arbitration Field”, “Control Field”, and “Data Field”. After the CRC field a one bit delimiter is sent, which has to be recessive. The next bit is sent dominant, but the receiver has to acknowledge reception by setting it recessive. The following bit is a delimiter for the acknowledge and has to be sent recessive. The last 7 bits of the CAN message are the end of frame and have to be sent all recessive ([4], pp. 2 - 7).

The extended data frame has between the IDE and the RTR bit 18 additional bits for the ID. This makes up a total of 29 bits for the node ID. After the RTR bit and before the DLC will then follow two reserved bits (RB1 and RB0), which are dominant. In total an extended data frame can contain 64 to 128 bits, depending on the DLC and the data that has to be sent. For the standard data frame the total length is between 44 and 108 bits.

The error frame is used to tell other nodes that an error has occurred in either the CRC, a missing acknowledge, a form error, a bit error, or a stuff error. The last frame, the overload frame, is used to tell other nodes that the sending node is currently busy handling new frames [4].

At the end of the transmission of each data or remote frame a so called “Inter-Frame Space” has to be sent, which consist of three recessive intermission (INT) bits and the recessive bus idle bits. A complete message can look like this in the hexadecimal system:

```
00 21 00 20 40 60 80 A0 C0 E1 18 B3 2F F
```

This message is a data frame, sent from the node with the ID 2, it has a DLC of 8 and the 8 bytes are: 1, 2, 3, 4, 5, 6, 7, 8. An overview of the data frame and the extended data frame can be found in the appendix at Figure A.1 and Figure A.2.

As communication on the bus has to be synchronized and because every node can have a different clock, bit stuffing has been introduced. Bit stuffing happens between the “Start of Frame” field until the first 15 bits of the “CRC Field” [4]. After five consecutive bits with the same polarity (either dominant or recessive) a stuff bit is sent, which is of opposite polarity. This bit can be discarded for the message but has to be taken into consideration for synchronisation. Synchronisation happens at every rising or falling edge in the bit stream.

### 2.5 Real-time

In [8] Liu and Layland describe a real-time system as a system which is able to control or monitor equipment at the same time when a new event occurs. “Each function to be performed has associated with it a set of one or more tasks. Some of these tasks are executed in response to events in the equipment controlled or monitored by the computer. The remainder are executed in response to events in other tasks. None of the tasks may be executed before the event which requests it occurs. Each of the tasks must be completed before some fixed time has elapsed following the request for it” [8]. They differentiate between hard real-time and soft real-time. Hard real-time has to guarantee that an action is taken in this fixed time span, whereas in soft real-time it is also possible to calculate with a belated answer.

Applied to the CAN bus introduced in Chapter 2.4 hard real-time has to be guaranteed. Each bit that is transmitted on the bus has to be read and interpreted at almost the same time it occurs. If a bit gets sampled too late the following bit may already have arrived and be read instead. Sampling too early could possibly result in sampling the non-stable state of the bit (see Figure 2.10), where the signal is in the process of getting established. Sampling in this phase is critical as there is a probability of sampling a dominant bit and a probability of sampling a recessive bit.

Soft real-time does not work for a bus system as reading and interpreting the bit has to be done exactly at the sample point. If out of a read bitstream one bit is interpreted at a later point than all other bits the resulting bitstream might have switched bits in it.

### 2.6 Floating Point Numbers Representation

Digital numbers are a mapping of a voltage to the logical states of 1 and 0. The representation of integer numbers is a stream of 0s and 1s in the binary system. For instance the number 42 in the decimal system is 101010 in the binary system. To show that a number is in the decimal system the base  $_{10}$  is added to the number. For the binary system the base  $_2$  is added,



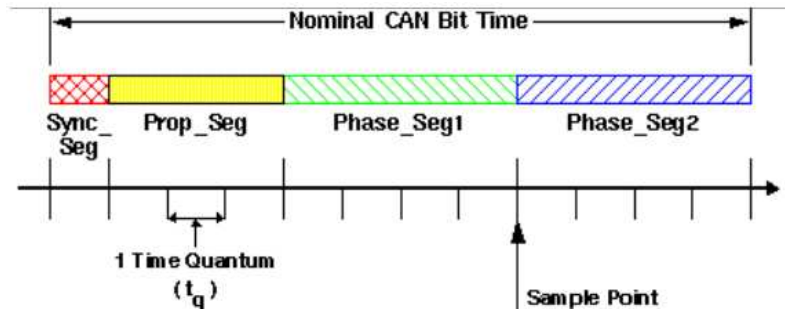


Figure 2.10: Bit timing on the CAN bus [9].

hexadecimal usually has a <sub>16</sub> or a *H* as a base, and octal a <sub>8</sub>. If no base for a number is given then it is assumed that the number is a decimal number.

To be able to represent floating point numbers (such as e.g. 3.141) a different approach is needed. Different forms of this representation are handled in the IEEE 754 norm or in the IEEE 754r norm [10]. The first one handles the representation of floating point numbers with single (32 bit) and double (64 bit) precision. The latter one has representations for 16, 32, 64, and 128 bit.

In order to convert a floating point number the number has to be decomposed as follows:

$$(2.2) \quad x = s * m * b^e,$$

where *s* is the sign bit, *m* the mantissa, *b* the base, and *e* the exponent. The first step is to determine the sign bit. If the number is a positive number the sign bit is 0, else 1. After that the first part of the number up to the comma is converted into binary, e.g. 5<sub>10</sub> is 101<sub>2</sub>. For the part after the comma the first digit in binary represents 2<sup>-1</sup>, the second 2<sup>-2</sup>, etc. Thus a 0.375 in decimal would be represented as a 011 (0 \* 2<sup>-1</sup> + 1 \* 2<sup>-2</sup> + 1 \* 2<sup>-3</sup>). Now both parts have to be put together with the comma between both parts, e.g. 101,011. Then the comma has to be shifted that there is only one 1 in front of the comma, in the previous case this would be 1,01011. The part right of the comma forms up the mantissa, which is in single precision 23 bit. The

remaining bits of the mantissa are filled up with 0's (e.g. 0 1011 would be 010 1100 0000 0000 0000 0000). The shift of 2 positions then has to be added to the exponent. The exponent also contains a bias, in single precision mode it is a bias of 127. Thus in the previous example the exponent would be  $127 + 2 = 129$ . Shifting to the left results in addition and shifting to the right in subtraction. The next step is to convert the exponent into the binary form, e.g.  $129_{10}$  is  $10000001_2$ . The final step is to put everything together after the following pattern:

$$(2.3) \quad \text{IEEE754 number} = \text{Signbit} \circ \text{Exponent} \circ \text{Mantissa},$$

where  $\circ$  is the concatenation operation. Thus  $5.375_{10}$  would be in IEEE754 single precision notation 0100 0000 1010 1100 0000 0000 0000 0000.

To convert a number from IEEE 754 notation into a floating point number again the following equation is needed:

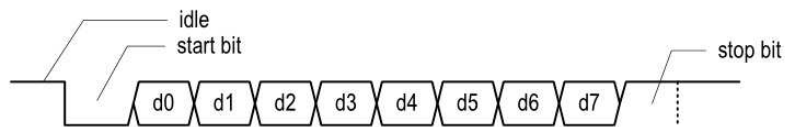
$$(2.4) \quad \text{Float} = (-1)^{\text{Signbit}} * \left(1 + \sum_{i=0}^{22} (\text{Mantissa}[22-i] * 2^{-(i+1)})\right) * 2^{\text{Exponent} - 127},$$

where **Signbit** is the first bit of the IEEE 754 number, **Exponent** the following 8 bits, and **Mantissa** the remaining 23 bits. The IEEE 754 single precision number 0100 0000 1010 1100 0000 0000 0000 0000 is segmented as follows: **Signbit** = 0, **Exponent** = 1000 0001, **Mantissa** = 0101 1000 0000 0000 0000 000. Again, the 127 is the bias. This example would give after inserting the numbers  $(-1)^0 * (1 + (1 * 2^{-2}) + (1 * 2^{-4}) + (1 * 2^{-5})) * 2^{129-127} = 5.375$ , which is the original number.

## 2.7 UART

The universal asynchronous receiver and transmitter (UART) is “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” [11]. A UART consists of a transmitter and receiver. The transmitter is “a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate” [11].

The receiver works analogously to the transmitter by shifting data bit by bit in. The transmission always starts with the start bit, a logical 0, followed by the data bits and an optional parity bit and ends with stop bits, a sequence of 1s. Six to eight bits can be sent in one message [11]. The transmission of a byte can be seen in Figure 2.11.



**Figure 2.11:** “Transmission of a byte” [11].

In a UART transmission no clock signal is sent, thus receiver and transmitter have to agree on some parameters in advance. These include:

- Baud rate (e.g. 115,200 bauds)
- Number of data bits
- Number of stop bits
- Whether to use the parity bit or not

A baud is the number of symbols that can be sent in a second. As only one line is available and only two voltages are sent (either +5V or 0V) one baud equals one bit. Consequently a baud rate of 115,200 baud equals 115,200 bit/s or 112,5 kBit/s.

---

## 3 Related Work

### 3.1 Bus Monitoring

There has been lots of research done in the field of bus monitoring, not only limited to the CAN bus. Bochem et al showed in [12] an approach of monitoring the CAN bus using an Altera DE2 development board and two different CAN controllers that are connected to the development board. The design has been done in Verilog using different IP cores.

Kashif et al showed in [13] 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 stand-alone CAN controllers. Furthermore they wanted to be able to inject further data on the bus with their solution.

Li et al showed in [14] a design for monitoring the CAN bus as well. They utilized for their solution two PCs and a “USB-CAN smart card” [14]. 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.

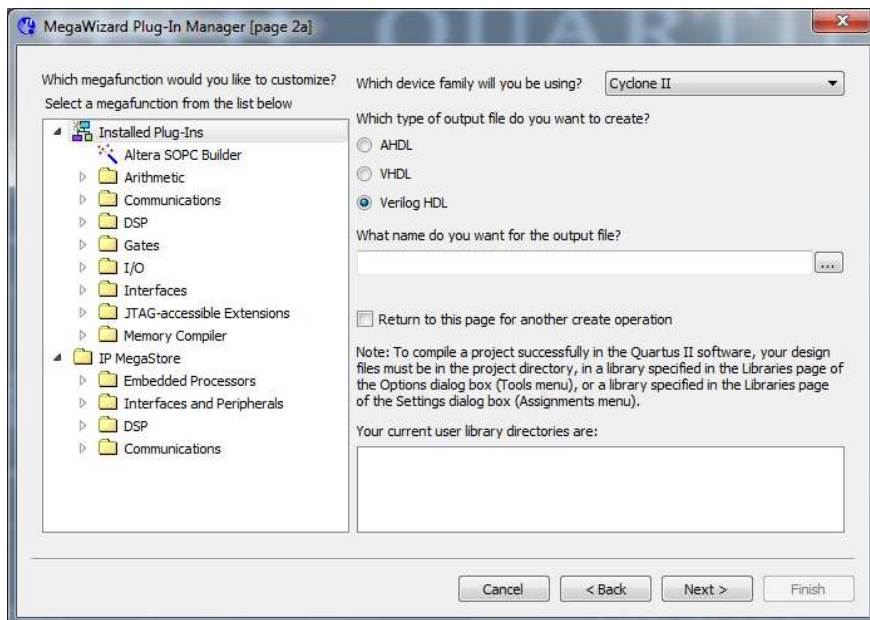
Another FPGA based softcore processor implementation on monitoring the CAN bus has been done by Mostafa et al in [15]. Their system could be controlled and configured using a RS232 compatible UART block. Furthermore they also provided the ability to inject errors onto the bus.

Using a CPLD to monitor the CAN bus Yang et al showed in [16] 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 [16]. For their implementation they used VHDL.

Other approaches using an FPGA to monitor a system have been shown by Mendoza-Jasso et al in [17] and by Zamantzas et al in [18] who were monitoring the Large Hadron Collider's (LHC) beam loss in CERN in real-time.

### 3.2 Existing IP Cores

When designing software or HDL models reusability of existing algorithms or functions play a big role, so that the engineer does not have to start completely from scratch. Using HDLs such as Verilog or VHDL this is done relatively easy, using IP cores, as introduced in Chapter 2.3. Altera provides for their FPGAs a store with IP ranging from Altera provided IP to vendor provided IP. Figure 3.1 shows the front page of this store with the topmost categories in the Altera Quartus II PLD design software, where the user can select between the different IP. In the IP MegaStore, vendor IP can be purchased.



**Figure 3.1:** The front page of the IP store in the Altera Quartus II PLD design software

Relevant for this project were IP for the following devices:

- CAN bus
- Floating point division
- LCD
- UART

Igor Mohor developed his own CAN bus controller in Verilog, called “CAN Protocol Controller”[19]. The controller utilizes 12,000 logic blocks. Bosch and CAST, Inc., provide their CAN controllers in the IP MegaStore in Altera Quartus II. This IP has to be licensed in order to use it in a project. The Bosch CAN controller for instance has a licensing fee of 10,200 Euro for the first 100,000 CAN products (further information are in [20]).

Most arithmetic operations are already provided by Altera free of charge. Thus floating point division is also included and the IP is called `ALTFP_DIV` [21]. When using this IP the engineer has to specify the precision for the division, which can be 32 bits, 64 bits, or with single extended precision ranging from 43 bits to 64 bits. As input it takes besides the clock, two numbers which already have to be in the correct precision in floating point notation. The output consequently is also a floating point number.

As the Altera DE2 board is not only a development board but also an education board implementations for most on-board devices are also given. This includes the LCD as well as the UART, for receiving and transmitting.

### 3.3 Problem Formulation

As shown in Chapter 3.1 there are already existing solutions for monitoring the load on a CAN bus using an FPGA. But almost none of them used a complete HDL written design but rather using a system on a chip (SoC) solution with a softcore processor. This has the advantage over microcontrollers that everything is directly on the chip, consequently this design is

### *3.3 Problem Formulation*

---

faster than an implementation on a microcontroller. But they are wasting a lot of potential of the FPGA with this solution, as having a HDL model is much faster than using a softcore processor.

It has also been shown that an implementation in Verilog without using softcore processors exists, but this solution is using an external CAN controller to read and write messages from and to the bus. Thus everything that is CAN bus based is in the responsibility of the external CAN controller.

The proposed solution features a HDL model implemented in Verilog that directly connects to the CAN bus. Therefore it is completely responsible for handling CAN messages.

---

## 4 Design

### 4.1 Requirements

The requirements for this project are:

- Usage of an FPGA development board
- Design has to be implemented in an HDL
- No softcore processors should be used
- Data should be directly acquired from the CAN bus
- Data acquisition has to happen in real-time
- Load values have to be calculated depending on the maximum throughput of the CAN bus
- Load values have to be displayed on the board itself
- Load values have to be transmitted to the PC via UART

As shown in Chapter 3.1 there are already existing solutions for monitoring the CAN bus with an FPGA. Most of them use a softcore processor for their approach. This project needs to implement an HDL model without using a softcore processor, to make use of all the advantages of FPGAs over microcontrollers. Thus the design needs to be implemented in an HDL, which either has to be Verilog or VHDL.

Furthermore the implementation has to acquire data directly from the CAN bus without using a dedicated CAN controller. Thus the implementation needs a driver for the CAN bus as well. As there is no clock on the CAN bus the driver needs to be able to resynchronize itself. This is possible due to the bit timing specified in the CAN protocol, as shown in Chapter 2.5. To achieve this bit synchronization the implementation should be able to operate in real-time.

After capturing the data from the CAN bus the design needs to interpret the

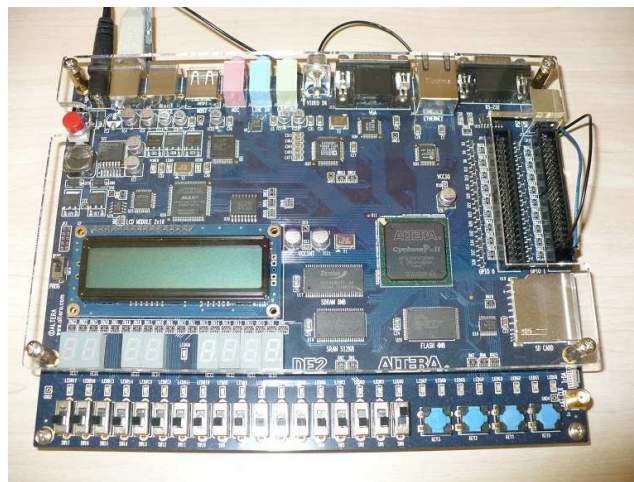


messages in order to calculate the load for the modules as well as the overall load. As all CAN data frames contain identifier bits the messages can be interpreted by looking at the ID bits. The calculated load values then need to be displayed directly on-board depending on the maximum throughput of the bus. Furthermore the load values also have to be transmitted to the PC via UART to be able to further process the information on a PC.

### 4.2 Design Decisions

#### 4.2.1 Hardware

For the development platform the Altera DE2 development and education board has been chosen. The board features an Altera Cyclone 2 FPGA with approximately 35,000 logic elements. It can be directly programmed from a PC via the USB blaster interface. For data display it features a two line 16 character LCD, as well as eight seven-segment LEDs. User input for choosing the module that can be monitored on the built-in display can be either done by the 18 on-board toggle switches or the four push-button switches. Via two extension headers additional peripherals can be connected. A picture of the Altera DE2 development board can be seen in Figure 4.1.



**Figure 4.1:** The Altera DE2 development and education board [22].

The Microchip CAN BUS Analyzer Tool supports the CAN specification

---

## 4.2 Design Decisions

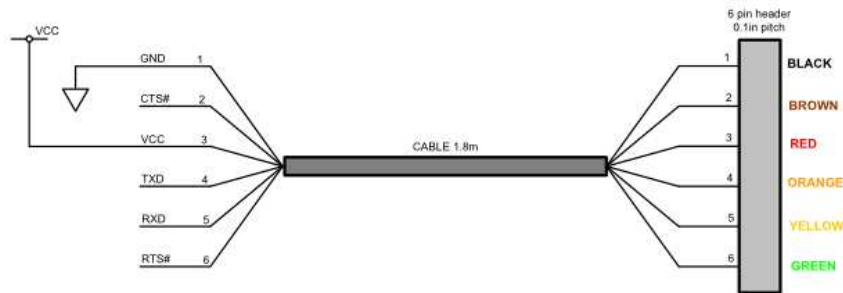
---

2.0b and can be used in a CAN network for development and debugging [23]. To establish a network at least two nodes are needed. Furthermore to be able to directly capture the data from the bus using the CAN BUS analyzer wires can be directly connected to the device which forward the CAN signals to an external device. The CAN BUS Analyzer by Microchip can be seen in Figure 4.2.



**Figure 4.2:** The Microchip CAN BUS Analyzer Tool [23].

The last hardware device used is the FTDI TTL-232R-3V3 cable, which allows the connection of a PC to an external device using the UART interface. The one end of the cable is a USB type A plug, the other end features a six pin header with wires for ground (GND), clear to send (CTS), VCC, transmission (TXD), reception (RXD), and request to send (RTS). The wiring can be seen in Figure 4.3.



**Figure 4.3:** The wiring of the FTDI TTL-232R-3V3 cable [24]

### 4.2.2 Software

For the software side the Altera Quartus II design software has been chosen as this is the best supported design software for Altera FPGAs. The versions used were Altera Quartus II 10.1sp1 and Altera Quartus II 11, both operating on a Windows 7 machine with 64 bit. With Quartus II it is possible to design in either Verilog or VHDL, as well as in other HDLs. In Chapter 2.2 the advantages of Verilog and VHDL is shown. Thus the project will be implemented mostly in Verilog. To be able to send messages on the CAN bus Microchip provided a tool which is able to command the Microchip CAN bus analyzer to send and receive messages. The tool can be seen in Figure 4.4. For receiving data over the UART interface to the PC a PC running Mac OS X 10.7 has been chosen, as it is capable of compiling and running C programs written for Linux.

The complete architecture of this project will be shown in Chapter 5.2. Everything was designed top-down, meaning the beginning problems were divided into smaller problems and these smaller problems then were divided for as long as they can be divided.

This lead to the project consisting of two big parts: the monitoring part and the output part. The monitoring part reads the input from the CAN bus, calculates the load values and stores everything for the output part. Furthermore the monitor part also consists of the CAN driver. The output

## 4.2 Design Decisions

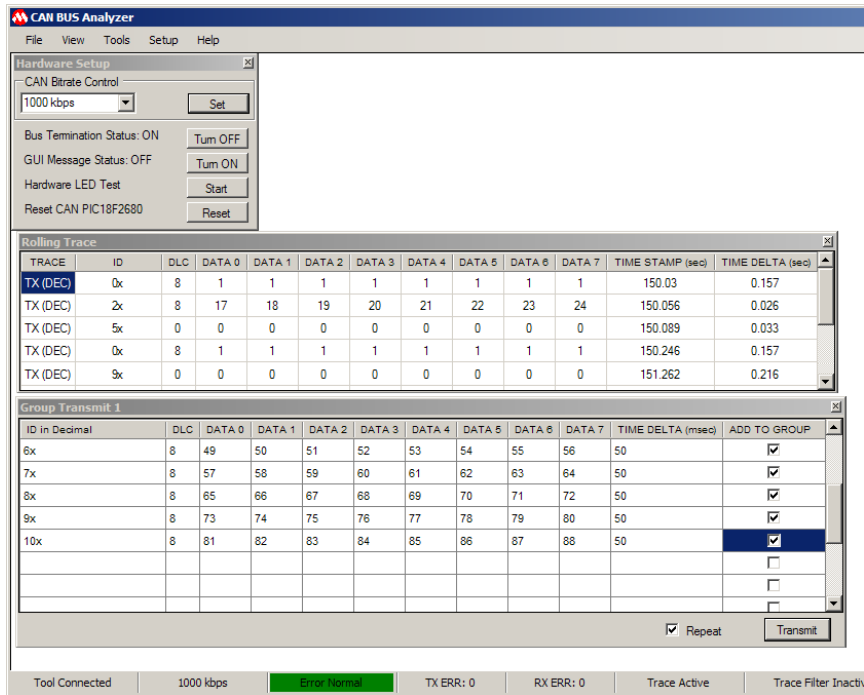


Figure 4.4: A snippet of the Microchip CAN Bus Analyzer Tool software.

part takes the calculated values, puts them into relation with the maximum throughput of the bus and displays the values according to the user selected module on the built-in LCD of the Altera DE2 board. It also disables the eight seven-segment LEDs as by default they are set to always on. Furthermore the output part is responsible for transmitting the load values via UART. The protocol for this will be shown in Chapter 5.2.6.

For the project design decisions include that only the FPGA internal logic blocks are used for storing the data, as saving data in the on-board SDRAM would lead to a loss in speed due to the distance of the FPGA to the SDRAM chip. Thus this project will include an implementation for 32 modules, which fit in the FPGA internal logic blocks. In Chapter 6.4 it will be shown, that more modules are also possible. In order to evaluate this project the load values are sampled once a second. This improves the comparison of the FPGA calculated values with the correct values, as well as the human readability of the data on the LCD. Furthermore the UART interface will only

be used in a one-way direction. This lowers the amount of logic needed for the FPGA but requires a strict protocol that has to be used on both the FPGA and PC side.

After the design stage the implementation had to be completed. This has been achieved using the bottom-up approach, which means that all of the problems that have been divided during the top-down design were implemented and after their implementation have been connected together. The next chapter will present IP that has been used and then explain what is implemented in each module.

---

## 5 Implementation

### 5.1 Usage of IP Cores

For this project only IP from Altera has been used:

- LCD
- UART Transmitter
- Floating Point Division

The reason for using these IP is that they provided the cleanest implementations found for their corresponding purposes. For the LCD only minor changes had to be made to make it function correctly. The provided LCD Verilog code was not able to alter the output on the display once it was set. It has been modified as follows:

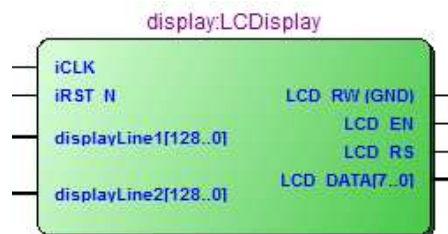
```
1  always@(posedge iCLK or negedge iRST_N)
2  begin
3      if(!iRST_N)
4          begin
5              [...]
6          end
7      else
8          begin
9              if(LUT_INDEX < LUT_SIZE)
10             begin
11                 case(mLCD_ST)
12                     0:      begin
13                             [...]
14                         end
15                     1:      begin
16                             [...]
17                         end
18                     2:      begin
19                             [...]
20                         end
21                     3:      begin
22                             [...]
23                         end
24                 endcase
```

```

25         end
26         else
27             LUT_INDEX      <= LCD_INTIAL+4;
28         end
29 end

```

The change that has been done occurs in line 27 of the above code, because after having reached the end of the display command array there was no possibility for the driver to go into a command state which accepts new inputs. Thus the display is put into state `LCD_INTIAL+4`, which is the command `9'h080`, after having written the last symbol on the display. This state resets the display. Then the driver can write more text on the display. As there is constant input from the output module to the LCD the display can be reset after having written the text. The instance of this module in the RTL viewer can be seen in Figure 5.1. The modified code takes two input string with the size of 129 bit, whereas the first bit is only used internally and does not need to be set. The remaining 128 bits represent each of the 16 symbols on one line of the display.



**Figure 5.1:** The LCD instance in the RTL viewer.

Altera provided also code for the transmission and reception via the UART protocol. But only the transmitter has been used for this project. The whole protocol for transmitting data to the PC is shown in Chapter 5.2.6. The code contains a baud rate generator, a start bit, eight data bits, and two stop bits. Following is the code for switching the states:

```

1 always @(posedge clk)
2 case(state)
3     4'b0000: if(TxD_start) state <= 4'b0100;

```

```
4   4'b0100: if(BaudTick) state <= 4'b1000; // start
5   4'b1000: if(BaudTick) state <= 4'b1001; // bit 0
6   4'b1001: if(BaudTick) state <= 4'b1010; // bit 1
7   4'b1010: if(BaudTick) state <= 4'b1011; // bit 2
8   4'b1011: if(BaudTick) state <= 4'b1100; // bit 3
9   4'b1100: if(BaudTick) state <= 4'b1101; // bit 4
10  4'b1101: if(BaudTick) state <= 4'b1110; // bit 5
11  4'b1110: if(BaudTick) state <= 4'b1111; // bit 6
12  4'b1111: if(BaudTick) state <= 4'b0001; // bit 7
13  4'b0001: if(BaudTick) state <= 4'b0010; // stop1
14  4'b0010: if(BaudTick) state <= 4'b0000; // stop2
15  default: if(BaudTick) state <= 4'b0000;
16  endcase
```

The data bits contain a preceding one, whereas the stop bits and the start bit have a preceding zero. This is needed when wanting to send the data, as the muxbit depends on the last three bits of the state:

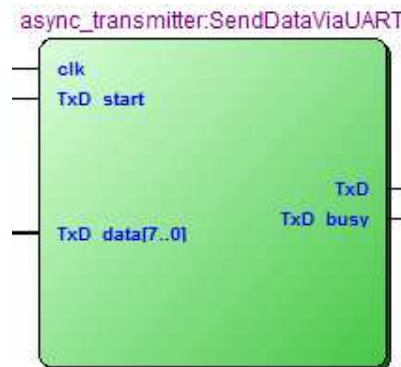
```
1  reg muxbit;
2  always @(state[2:0] or TxD_data)
3  case(state[2:0])
4    0: muxbit <= TxD_data[0];
5    1: muxbit <= TxD_data[1];
6    2: muxbit <= TxD_data[2];
7    3: muxbit <= TxD_data[3];
8    4: muxbit <= TxD_data[4];
9    5: muxbit <= TxD_data[5];
10   6: muxbit <= TxD_data[6];
11   7: muxbit <= TxD_data[7];
12  endcase
```

To now be able to distinguish actual data, the muxbit and the fourth bit of the state are combined by a logical AND operator and the result from this gets combined by a logical OR with the expression `state<4` which is one if the state number is less than four, else it is zero:

```
1  always @(posedge clk) TxD <= (state<4) | (state[3] & muxbit);
```

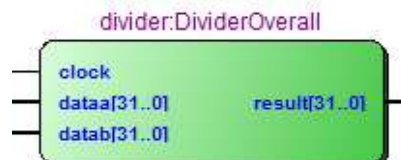
This results in the start and stop bits to always be a logical one that is sent on the bus and the data bits being the actual data that have to be sent. The instance of this module can be seen in Figure 5.2





**Figure 5.2:** The UART module instance in the RTL viewer.

The last IP that has been used is the floating point division called ALTFP\_DIV. This is the only IP where no code was available. In Altera Quartus II a wizard tool is able to configure this module the way the user requires. This module is capable of handling single precision, double precision and single extended precision floating point division. For this project single precision suffices the purposes of calculating load data. The module takes as the first input, `dataa`, the integer load count converted to the IEEE754 single precision format and as the second input, `datab`, the maximum load that can occur on the bus in the given timespan in the IEEE754 single precision format. For one second the value is  $46\ 23\ D7\ 0A_H$ , as the bus has a bitrate of 1 Mbit/s which is  $2^{20}$  bit/s. This number has to be divided by 100 to put this in the percentage interval of  $[0, 100]$  and then has to be converted into the IEEE754 single precision format. The instance of the floating point division module can be seen in figure 5.3.



**Figure 5.3:** The floating point division module instance in the RTL viewer.

## 5.2 Architecture

### 5.2.1 Overview

The complete project has been divided into two main parts: the monitor part and the output part. The monitor part itself consists of the CAN bus driver module and an algorithm for calculating the load data for 32 modules and the overall load. The CAN driver itself is explained in Chapter 5.2.2, the algorithm for calculating the load in Chapter 5.2.3.

The bigger part of this project is the output part. It consists of the conversion of integers to IEEE754 numbers, the division of two floating point numbers, the conversion of numbers to strings, the LCD driver, the UART driver and the protocol for sending data over UART. The conversion and division is discussed in Chapter 5.2.4, the LCD driver in Chapter 5.2.5, the UART driver in Chapter 5.2.6 and the PC client for receiving data over UART in Chapter 5.2.7. This chapter concludes with the interaction of all modules in Chapter 5.3.

For this project the wiring has been done in the assignment editor for the following pins:

- 18 toggle switches: SW[0] to SW[17]
- 8 seven-segment LEDs: HEX $n$ [0] to HEX $n$ [6], with  $0 \leq n \leq 7, n \in \mathbb{N}$
- The FPGA clock generator for 50 MHz: CLOCK\_50
- The LCD command and data pins: LCD\_RW, LCD\_EN, LCD\_RS, LCD\_ON, LCD\_BLON, LCD\_DATA[0] to LCD\_DATA[7]
- The CAN bus wire on GPIO\_1: CANbusWire
- The UART TxD Wire on GPIO\_0: TxDWire

The ports have been declared as follows:

```
1 input          CLOCK_50 ;
2 input          [17:0] SW ;
3 output         [6:0] HEX0 ;
```

```
4 output      [6:0] HEX1;
5 output      [6:0] HEX2;
6 output      [6:0] HEX3;
7 output      [6:0] HEX4;
8 output      [6:0] HEX5;
9 output      [6:0] HEX6;
10 output     [6:0] HEX7;
11 inout      [7:0] LCD_DATA;
12 output     LCD_ON;
13 output     LCD_BLON;
14 output     LCD_RW;
15 output     LCD_EN;
16 output     LCD_RS;
17 output     TxDWire;
18 input      CANbusWire;
```

### 5.2.2 CAN Driver

The CAN driver has been implemented as a state machine. Every 48 clock ticks it takes a sample from the bus and after every edge it resynchronizes itself. The value is 48 because the FPGA clock divided by the bus bitrate is about 47.68, which, rounded, is 48. The algorithm for calibrating the FPGA clock to the CAN clock is as follows in pseudocode:

```
1  if (oldbit != newbit)
2    wait 31 FPGA clock ticks
3    to be between phase 1 and 2
4    reset sample rate and take a sample
```

As shown in Chapter 2.5 sampling has to occur between phase 1 and 2 which occurs after 31 FPGA clock ticks. The state machine has four states: interframespace, idle, message transfer and end of message. In the beginning the state machine starts in the idle state as there might be a transmission in progress. Thus the first message received might be a wrong message as there is no possibility to check if the received bits belong to the interframe space, the message or the bus idle. The state machine is illustrated in Figure 5.4.

As long as the bus is idle it sends logical ones. If a logical zero is received it sets the first bit of the CAN message accordingly, resets all other variables,

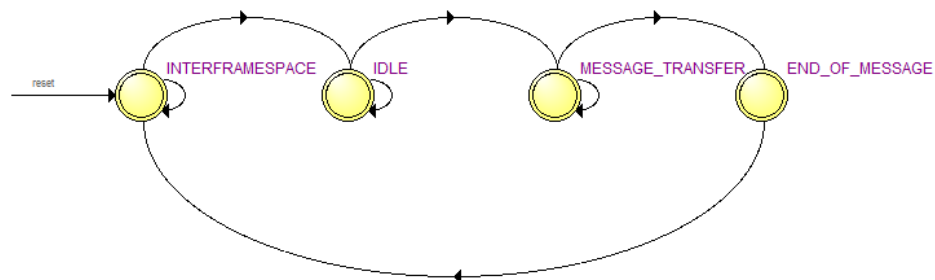


Figure 5.4: The state machine for the CAN driver

and does a state transition from idle to message transfer. In this state the data is stored for as long as there is no stuff bit or the end of message detected. The stuff bit gets detected as follows:

```

1   if (stuffBit != actual read bit && stuffBitCounter > 0)
2       reset stuffBitCounter
3   stuffBit = actual read bit
4   stuffBitCounter++
5   if (stuffBitCounter == 5 && read bit belongs to
6       bit stuffing area)
7       next bit is a stuff bit and has to be discarded
  
```

The bit stuffing area can be determined after the DLC is read. This can be done after having received 40 bits, because if the message is an extended data frame the DLC is at a different position than in the standard data frame. The bit stuffing area can then be calculated like this:

```

1   if (extended)
2       bit stuffing area = 128 - (39 + DLC * 8 + 15)
3   else
4       bit stuffing area = 128 - (19 + DLC * 8 + 15)
  
```

As the message gets filled from bit 127 to bit 0 the 128 has to be subtracted. The end of message can be determined by a similar algorithm:

```

1   if (extended)
2       end of message = 64 - (DLC * 8)
3   else
4       end of message = 84 - (DLC * 8)
  
```

This calculation is already factored out. Until the end of message is reached every bit is stored unless it is a stuff bit. If the end of message is reached a transition from the message transfer state to the end of message state is done.

In this state the output variables are set to the message, as well as the length of the message. Furthermore a flag signalling that a complete message has been received is sent to the main module. Then a transition from the end of message state to the interframe space state is done. This allows the main module enough time to capture the message and do the needed calculation. In the interframe space state the flag for having received a complete message is set to false and three successive logical ones are awaited. After that the transition to the bus idle state is done. Figure 5.5 shows the in- and outputs of the CAN driver module.



**Figure 5.5:** The CAN driver instance in the RTL viewer.

### 5.2.3 Load Calculation

After capturing messages from the CAN bus they need to be interpreted for calculating the load values. This is done in a state machine, as there are four possible situations that can occur when looking at if a new message is ready and if we have hit the sample rate. Thus there are four situations that occur and they can be classified as follows:

1. No new message, sample rate not hit
2. No new message, sample rate hit
3. New message, sample rate not hit
4. New message, sample rate hit

The first state is also called the idle state as in this state the program does not have to do anything, because no new data has arrived and the sample rate has not been hit. The second state is called the sample state, as no new data has arrived but the sample rate has been hit. In this state the program has to generate a freeze of the current values for the module load and the overall load, as well as to signal that new data is ready to send and display. Furthermore the program has to reset the working variables. The freeze and reset is done as follows:

```
1      // generate a freeze of the current variables
2      overallFreeze    = overallLoad;
3      moduleFreeze     = load[moduleNumber];
4      for (i = 0; i < 32; i = i + 1)
5          loadFreeze[i] = load[i];
6
7      // reset the variables
8      for (i = 0; i < 32; i = i + 1)
9          load[i] = 32'b0;
10     overallLoad = 32'b0;
```

The third state occurs when the CAN driver signals that a new message has been received and if the sample rate has not been hit. This state is called the update state. In this state the length of the CAN message gets added to the overall load:

```
1      overallLoad = overallLoad + messageLength;
```

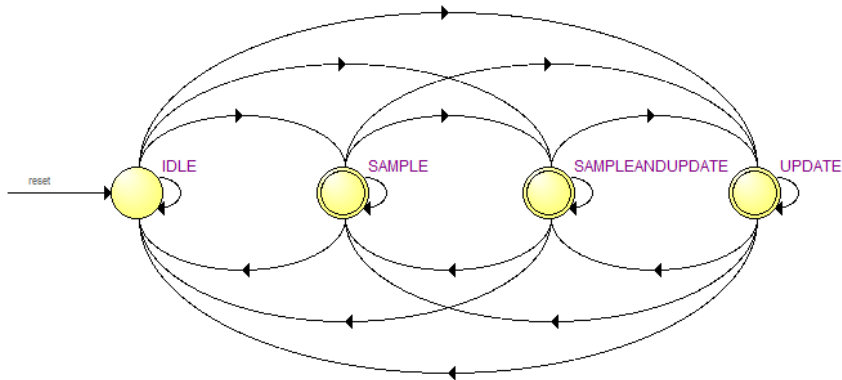
Furthermore for being able to add the length of the message to the corresponding module it needs to be checked if an extended data frame has been received and then the identification pointer needs to be set accordingly:

```
1      if (extended)
2          loadID = the 29 bits of the ID
3      else
4          loadID = the 11 bits of the ID
```

After that the value of the module load variable can be set accordingly:

```
1      load[loadID] += messageLength
```

The last state occurs if the sample rate has been hit and a new message has been received by the CAN driver. This state is called the sample and update state and is a mixture of the second and third state. First the update is done as described in state three and then the sampling is done as described in state two. Figure 5.6 shows the complete state machine.



**Figure 5.6:** The state machine for the load calculation algorithm.

### 5.2.4 Output Conversion

As stated in the requirements in Chapter 4.1 the solution has to be able to display its results on both the built-in LCD, as well as on an external PC screen. As described in Chapter 5.2.3 the HDL model has direct access to the stored load values, which datatype is similar to that of an integer, with the exception that only 29 bits are reserved to store the value.

In order to divide this value based on the maximum load a floating point representation is needed, as not only integer values are the result of a division. Before the load values can be passed to this core the values have to be converted. The following Verilog code shows how this is done:

```
1      if (inputInteger < 0)
2          outputIEEE754[31] = 1;
```

```
3     else
4         outputIEEE754[31] = 0;
5     position = 0;
6     for (i = 0; i < 22; i = i + 1)
7         if (inputInteger[i])
8             position = i;
9     exponent = 127 + position;
10    outputIEEE754[30:23] = exponent;
11    for (i = 0; i < 22; i = i + 1)
12        if (i < position)
13            outputIEEE754[23 - position + i] =
14                inputInteger[i];
```

This code directly follows the formula from Equation 2.2 introduced in Chapter 2.6. In line 1 to 4 the sign bit is determined. Lines 5 to 9 determine the exponent by iterating through the binary representation of the load value searching for the leftmost 1. This is only possible when converting integer values. If a floating point number has to be converted the part after the decimal comma has to be considered, too. The exponent then is the bias plus the determined position of the leftmost 1. A subtraction of 1 for this position is not necessary, as the iteration through the number stops right at the current location of the leftmost 1 and the counting starts from 0. For instance, the binary number  $0011\ 0100_2$  has its leftmost 1 at position 5 considering the counting from right starting from 0. But it is the 6th number.

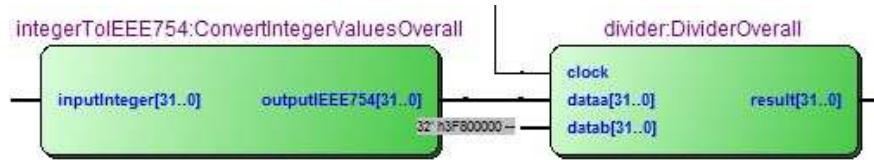
The 23 bit mantissa can be determined by using the bits right from the leftmost 1, e.g. in the above example  $0011\ 0100_2$  would result in the mantissa  $1010\ 0000\ 0000\ 0000\ 0000\ 000_2$ . Lines 11 to 14 show how to iterate through this selection and copy bitwise these bits into the mantissa.

The floating point division core needs two IEEE754 numbers in input, depending on the configuration in either single precision or double precision configuration, and delivers as a result another IEEE754 number with the same precision. As the sampling rate of this HDL model is currently 1 Hz the denominator  $d$  is calculated as follows:

$$(5.1) \quad d = 1 * 1024 * 1024 / 1 / 100.$$



The first part up to the first division is the conversion of the upper cap of the CAN bus from Megabit into bit, the 1 is the sampling rate, and the 100 is because the result should be in percent. The whole process can be seen in Figure 5.7.



**Figure 5.7:** The conversion of an integer value into the IEEE754 format with the division of this number with the maximum load in that sampling period.

### 5.2.5 Display Module

The Altera DE2 board features two different kinds of display options: numbers can either be displayed on the eight seven-segment LEDs or on the LCD, which features two lines with 16 characters each. As the requirement is to use the LCD, the LEDs have to be turned off first. This is done with the following Verilog code:

```
1  oSEGO = 7'b1111111;
```

This line is wired to the first seven segment digit and disables it. The remaining seven digits can be disabled analogously. The LCD driver has been explained in detail in Chapter 5.1. As an input it expects two 16 byte strings, which represent the two lines on the display.

For the output on the display this module displays the following base layout, which gets filled by the conversion module:

```
1 Overall:      , %
2 Mod         :      , %
```

The numbers get passed to the conversion module as integer numbers and get parsed character wise and put into a string. To be able to look at each digit of the number itself usage of the integer division is made and looking

at the remainder gives the number at that particular position. Following is the equation for looking digit wise at a number:

$$(5.2) \quad d_p = n/10^{(p-1)}\%10,$$

where  $d$  is the digit at position  $p$ ,  $n$  is the integer number that needs to get passed, and  $\%$  is the modulo operation. For instance the number 123 would get parsed into  $d_1 = 123/10^{(1-1)}\%10 = 3$ , which would be the first digit,  $d_2 = 123/10^{(2-1)}\%10 = 2$ , and the third digit  $d_3 = 123/10^{(3-1)}\%10 = 1$ . After having found the digit as an integer number from a lookup table the corresponding string value is selected:

```

1  case(digit)
2      0: character = "0";
3      1: character = "1";
4      2: character = "2";
5      3: character = "3";
6      4: character = "4";
7      5: character = "5";
8      6: character = "6";
9      7: character = "7";
10     8: character = "8";
11     9: character = "9";
12 endcase;
```

Preceding zeros of the number are truncated after the conversion. After conversion the number is displayed on the display with the following code:

```

1      displayLineString1[63:8] <= overallString;
2      displayLineString2[63:8] <= moduleString;
```

where `displayLineString $n$`  ( $n \in \{1,2\}$ ) represents the corresponding line on the display and `overallString` and `moduleString` the relative bus load of the overall load and the module load, respectively.

### 5.2.6 UART Module

The UART module has been implemented in a one way direction from the Altera DE2 board to the PC. For sending data the following protocol is used:

1. Wait for data to 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 we are sending the overall load
7. Send overall load
8. Set the data ready variable to false
9. Go to 1

To be able to follow this protocol the limitations of the UART core by Altera has to be taken into consideration. This core is able to only send one byte at once. The load values as well as the start signal are longer than one byte and thus have to be split accordingly.

The start signal is  $10\ 0001_H$ , which is  $(2^{20} + 1)$ , a value which can never be reached by the calculation part, as the maximum throughput of the CAN bus is 1 Mbit/s, which is  $2^{20}$  bit/s. The start signal is split into three parts, making 3 bytes, the first byte sent is  $01_H$ , the second  $00_H$  and the last byte is  $10_H$ . To split sending values a state machine is used, the following Verilog code illustrates this:

```
1 assign sendUART = (state == WRITE) ? 1'b1 : 1'b0;
2 assign transmissionDone = (state == IDLE) ? 1'b1 : 1'b0;
3
4 always@(posedge clock)
5 begin
6     case(state)
7         IDLE:
8             if (dataReady)
9                 begin
10                    dataStore = dataIn;
11                    bytePosition = 0;
```

```
12     state = WAIT;
13     end
14
15     WAIT:
16     begin
17         if (!TxDBusy)
18         begin
19             case (bytePosition)
20                 0: dataOut = dataStore[31:24];
21                 1: dataOut = dataStore[23:16];
22                 2: dataOut = dataStore[15:8];
23                 3: dataOut = dataStore[7:0];
24                 4: state = IDLE;
25             endcase
26             if (bytePosition != 4)
27                 state = WRITE;
28             end
29         end
30
31     WRITE:
32     begin
33         bytePosition = bytePosition + 1;
34         state = WAIT;
35     end
36 endcase
37 end
```

The first `assign` handles the transmission over UART and is set to a logical 1 if the state machine is in the `write` state. When the transmission is done the state machine goes into the `idle` state where it waits for new data. While in the `idle` state a flag is set to a logical 1 telling the output part that the UART module is waiting for new input. When sending data the protocol is to iterate through the four bytes of the data from left to right, thus sending the bits 32 to 25 at first, then bits 24 to 17, and the remaining two bytes analogously.

After the start signal has been sent the IDs and the values for the modules is transmitted. The data is packed as follows:

```
1 dataToSend = {id, three stuff bits (0), load[id]};
```

After all 32 module IDs and load values have been sent the overall load is sent. As the project is capable of sending 32 modules, ranging from 0 to 31, the ID for the overall load is 32, which has to be read and interpreted by the PC client to be the value for the overall load. After this has been done the communication channel is held by setting the data ready flag to false. The full communication protocol in pseudocode is as follows:

```
1 always at a rising edge on the FPGA clock
2   if (sample rate is hit)
3     reset all variables needed for communication
4     go to state WAIT
5
6   send-stateMachine:
7     WAIT:
8       protocol-stateMachine:
9         0: data = 00 10 00 01
10        1-32: data = {id, three stuff bits (0), load[id]}
11        33: data = {32, three stuff bits (0), overallLoad}
12        34: set state to IDLE
13        if (protocol is not 34 and communication channel is free)
14          set state to SEND
15
16      SEND:
17        set the protocol-stateMachine to the next state
18        set the send-stateMachine to WAIT
```

If all data has been sent there is no communication on the UART channel until the sample rate has been hit the next time, thus the state is changed to idle which is an empty state.

### 5.2.7 PC Client

As a requirement of this project is to be able to receive the load values on the PC a client that receives values has also been developed. This has been achieved using C++ and is natively running under Linux and Mac OS. Windows support is also possible, but Cygwin[25] has to be installed and configured to make this program run. Furthermore the `ncurses` library is used in this program and is needed to make this program run. It can be installed via the packet manager, which is in Ubuntu Linux `aptitude`, under

Mac OS `macports`, and under Windows `Cygwin`.

First the communication port has to be opened. The full code for this is shown in the appendix in Section A.2. A file descriptor is needed as in an UNIX environment devices are declared as special files. Under Linux the file that has to be opened is by default `/dev/ttyS0`. Mac OS maps the device by default to `/dev/tty.usbserial-FTELTQ86`. After the file has been opened the port has to be configured. The baudrate that has been used is 115,200 Baud/s, the port has been set to read/write with non-blocking communication, no parity bit, but two stop bits, as defined by the Altera UART core.

The `ncurses` library has been used as it makes console programming more comfortable. Thus a layout can be printed to the screen and refreshed without needing to clear the console by hand. Reading from the port can be done as follows:

```
1     read(fd, value, 4);
```

where `value` has been declared as an `unsigned char[4]`, which has in C++ a length of four bytes [26]. After having read from the communication channel the same protocol as introduced in Chapter 5.2.6 has to be implemented. This is done as follows in pseudocode:

```
1 do forever
2   read four bytes from the UART interface
3   if receivedNumber is (2^20 + 1)
4     startFlagReceived = true
5   if startFlagReceived
6     value[id of receivedNumber] = receivedNumber
7     if 33 4-byte tuples have been received
8       startFlagReceived = false
9   update console with the received values and 33 being
10  the overall load value
```

The layout of the PC client can be seen in Figure 5.8.

### 5.3 Interaction of the Modules

The UML-like class diagram in Figure 5.9 shows the interaction of the output and module part. The monitor is fed by the CAN bus wire and the

### 5.3 Interaction of the Modules

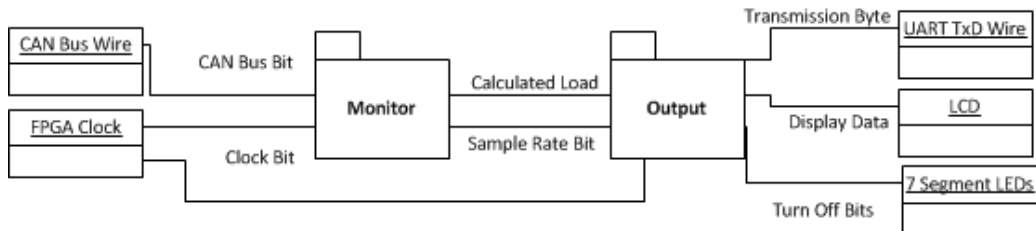
```
can bus monitor - uart receiving tool

      make sure the usb cable is plugged in and the driver is working
      press ctrl-c to quit

overall load:      0 bit/s, 0.00%

module   0:      0 bit/s, 0.00%
module   1:      0 bit/s, 0.00%
module   2:      0 bit/s, 0.00%
module   3:      0 bit/s, 0.00%
module   4:      0 bit/s, 0.00%
module   5:      0 bit/s, 0.00%
```

**Figure 5.8:** The basic layout of the PC client that is able to receive data from the Altera DE2 board. Only the first six modules are shown for illustration purposes.



**Figure 5.9:** An UML-like class diagram for the interaction of the output and the monitor part.

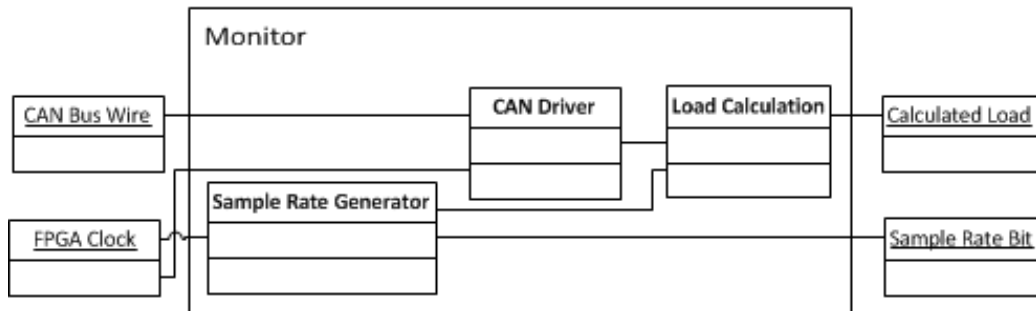
FPGA clock. Both inputs generate a constant bit stream. Internally the monitor acts as shown in Figure 5.10.

The FPGA clock is handled by the sample rate generator, which generates a clock of 1 Hz. This sample rate is then passed over to the load calculation part and to the output part. The CAN driver operates at 50 MHz due to resynchronization and bit timing. It takes as input besides the FPGA clock the CAN bus wire. The output is used for the load calculation, which calculates the overall and module load values. These values are then passed over to the output part as well.

The output part is illustrated in Figure 5.11. The UART send protocol needs the calculated load from the monitor part as well as the sample rate bit, to be able to start its transmission whenever the sample rate has been

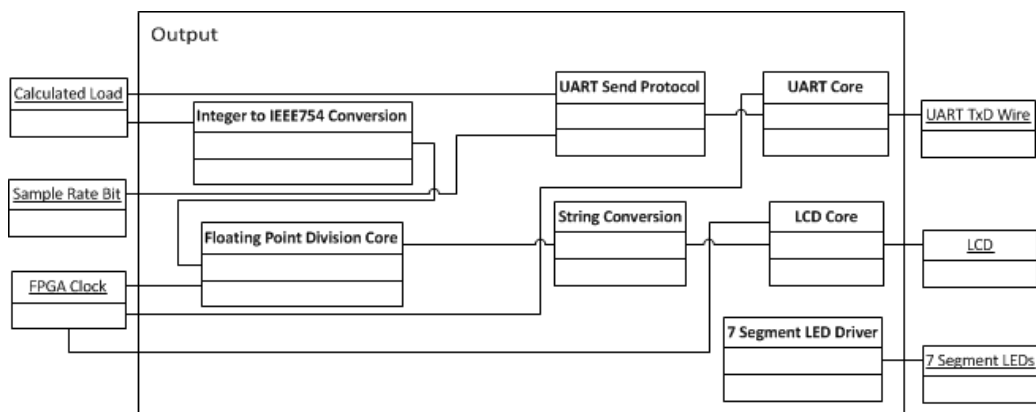
### 5.3 Interaction of the Modules

---



**Figure 5.10:** An UML-like class diagram for the interaction of the modules inside the monitor part.

hit. As an output it delivers data in one byte length which is sent by the UART core over the UART TxD wire to an external device.



**Figure 5.11:** An UML-like class diagram for the interaction of the modules inside the output part.

The calculated load is also used for displaying the values on-board. For that the calculated load is passed over to the integer to IEEE754 conversion module which generates a floating point number that is passed over to the floating point division core. As an additional input the floating point division core needs the FPGA clock. The output of this core is then parsed into a string in the string conversion module. It is then displayed via the LCD core on the built-in LCD. The seven segment LED driver is only responsible



### 5.3 *Interaction of the Modules*

---

for switching off the built-in seven segment LEDs.

---

## 6 Evaluation

### 6.1 Approach

To be able to evaluate this project each of the subparts have to be evaluated individually to ensure that they are working correctly. After all of them have been evaluated the whole project itself can be evaluated.

The first part to be evaluated is the output part as the output is necessary in order to evaluate the internal parts, such as the CAN driver and the load calculation. Thus the evaluation will start with the LCD and the UART module. After this the floating point division will be evaluated. This will also conclude the evaluation of the output part.

The monitor part consists of the CAN driver and the load calculation module. Thus at first the CAN driver is evaluated and then the load calculation. This chapter concludes with an overall evaluation in Chapter 6.4.

### 6.2 Output Module

#### 6.2.1 Verification of the LCD

As the LCD implementation was made by Altera it has been tested to display a message at first. The display has been initialized in the project by:

```
1 display LCDisplay(CLOCK_50, DLY_RST, displayLineString1,  
2   displayLineString2, LCD_DATA, LCD_RW, LCD_EN, LCD_RS);
```

The `displayLineString $n$`  ( $n \in \{1, 2\}$ ) have been set to a predefined message that have been displayed correctly. When wanting to change the message after a predefined amount of time the text on the display had not changed. Thus a modification has been done in Altera's implementation to allow the display driver to change the text on the display. This has been tested by showing a default message which is changed after one second. A small implementation, called `selftest`, has been implemented which changes messages after one second according to a protocol. The `selftest` implementation can be found in the appendix in Chapter A.3. It displays some information on the LCD and then displays the numbers from zero to

nine. This output can be activated by setting **SW17** to on.

The next thing to be evaluated is to display the basic layout on the LCD with the numbers on it. Thus the numbers needed to be parsed from integer values to string-type values. In order to achieve the correct outputs different numbers have been hard-coded into the program that needed to be displayed. This has been achieved in the `conversion.v` file. After the output for an integer number was working correctly the module for converting IEEE754 numbers to a string had to be implemented. A couple of predefined floating point numbers have been calculated according to the algorithm defined in Chapter 2.2 and were hard-coded into the `conversion.v` as well.

The conversion from these IEEE754 numbers to strings have altered precision, as for displaying the numbers on the LCD only two post-decimal digits are used. Thus only the first ten bits of the mantissa have been taken into consideration as from that point on the remaining bits of the mantissa do not influence the result of the conversion significantly. This precision alteration lead to an error of approximately 0.002%, which can result in the second post-decimal place to differ by one from the true number.

### 6.2.2 Verification of the UART Module

The UART module has been evaluated with a similar approach. At first the UART module has been instantiated:

```
1 async_transmitter SendDataViaUART(clock, sendUART, dataOut,  
2   TxDWire, TxDBusy);
```

The `sendUART` variable is set to one if the data in the variable `dataOut` has to be sent over the UART TXD wire `TxDWire`. When the TXD wire is busy the variable `TxDBusy` will read a one, which means that no data can be sent right now over the wire. Keeping this in mind the first thing to do was to send static data to the PC. Thus a predefined number has been set for `dataOut` and every second `sendUART` was set to one with the next FPGA clock edge disabling `sendUART` again. This lead to the FPGA sending once

a second this predefined number. The PC client was always listening on the UART interface and waiting for new data. Every second it received the byte from the FPGA and displayed its number on the PC screen. In the beginning there were issues with the reception as sporadically the number was shifted a couple of bits resulting in a constant stream of the wrong number, as the PC client was not configured correctly. After this has been fixed the correct number is always received.

The next step was to display changing numbers. Therefore the sending part was configured to increment the number every second and then send it to the PC. Thus a numerical sequence was received by the PC. Now the protocol needed to be tested. Before that can be achieved it was needed to make sure that a four byte packet was received correctly. Therefore the data packet was increased to four bytes with a predefined 32 bit message. This packet has been sent to the PC once a second. The PC client needed to make sure that it always reads four bytes. Different predefined four byte packages have been tested to make sure that this was also working.

Finally, the whole UART packaging and sending part could be tested and evaluated in whole. For this, the whole protocol has been activated, but the messages that have to be received by the PC were predefined. This was the overall load as well as the different load values. They have been set to a constant number to make the comparison of the received number to the hard-coded number possible. After this the hard-coded numbers have been removed and the calculated numbers from the calculation part have been used.

As the LCD was working correctly the received numbers have been compared with the ones displayed on the LCD to make sure that this part was working as well. The sending currently faces one issue: when sending the load for the 0th module occasionally the wrong number is received. This is due to the fact that at least 11 consecutive zeroes are sent, leading to a loss of synchronisation on either the PC or the FPGA side.

### 6.2.3 Verification of the Floating Point Division

The conversion of numbers into the IEEE754 format has been evaluated in Chapter 6.2.1. Now the division of these numbers with another IEEE754 number had to be evaluated as well. This has been done using the `ALTFP_DIV` core with hard-coded numbers at first. Different IEEE754 representations have been chosen for the dividend and divisor. The quotient then has been displayed on the LCD and transmitted via UART to make sure that the division is working correctly.

After the successful testing of the division of two hard-coded numbers the divisor has been set to the value needed for the project. This value is  $2^{20}/100$  as the monitor samples the bus once every second. Represented in the IEEE754 format this number is 0100 0110 0010 0011 1101 0111 0000 1010<sub>2</sub> or 46 23 D7 0A<sub>H</sub>. Again hard-coded values have been used for the dividend to make sure that the quotient is correct. Lastly the module is given the values calculated from the calculation module.

## 6.3 Monitor Module

### 6.3.1 Verification of the CAN Driver

The evaluation of the CAN driver has been the most difficult part, as the CAN bus features different kinds of messages, as well as different message lengths, a synchronisation algorithm, and parts with bit stuffing in order to not lose synchronisation. At first the reception of single bits has been tested. This was done by constantly reading the input from the CAN bus wire. After this has been successfully completed the next step was to only read bits when the Microchip CAN Bus Analyzer Tool puts a new bit on the bus.

Therefore a sample rate generator has been implemented. To test this generator a method has been designed to read a continuous stream of 128 bits. The code for reading a stream of 128 bits in Verilog is as follows:

```
1 always@(posedge nextBit)
2 begin
```

```
3   data[counter] = CANMessageBit;
4   if (counter == 128)
5     begin
6       done = 1'b1;
7       counter = 0;
8     end
9   else
10    begin
11      done = 1'b0;
12      counter = counter + 1;
13    end
14  end
```

Every time a new bit is available on the bus (signalized by `nextBit`) this method is triggered. At the next available position it saves the current read bit `CANMessageBit`. If 128 bits have been read it signals the monitor part that a complete message has been received. The monitor then can use this stream for further processing, which in this evaluation case is to display the message in hexadecimal on the LCD. Furthermore for this case, messages consisting only of ones were discarded and the capturing only started when the start bit was received. This was to make sure that mostly real messages were captured. The CAN Bus Analyzer Tool was configured to only send one predefined message to allow the comparison with the message displayed on the LCD. Furthermore it needed to be taken into consideration for the comparison that the received messages included the bit stuffing bits. Different messages have been compared to make sure that the implementation was working correctly. These messages included standard and extended data frames, various DLC and data bits.

As the evaluation of reading messages from the bus was done the next thing to evaluate is bit stuffing. The evaluation has been done the same as without bit stuffing. After the implementation of the bit stuffing protocol the received messages again were displayed on the LCD and compared to the original sent messages. Again, different messages have been chosen to make sure that the implementation was capable of receiving different messages correctly.

At last the synchronisation algorithm needed to be evaluated. This was done in comparison to the implementation without synchronization. Without synchronization approximately 20 out of 100 messages were received incorrectly. This is due to the facts that the FPGA clocks at a much higher rate than the CAN bus and that the FPGA clock is not a multiple of the CAN bus clock. This has a result that a resynchronisation needs to be done at every signal edge on the CAN bus wire. The implemented synchronization algorithm is not an optimal implementation as it still has messages received incorrectly. The implemented synchronization lead to approximately 3 out of 200 messages to be received incorrectly.

#### 6.3.2 Verification of the Load Calculation Module

The calculation of the load has also been evaluated partially. At first an algorithm that counts the message lengths over time has been implemented:

```
1  always@(posedge sampleFlag or posedge done)
2  begin
3      if (sampleFlag)
4          begin
5              overallFreeze = overallLoad;
6              overallLoad = 0;
7          end
8
9      if (done)
10         overallLoad = overallLoad + messageLength;
11 end
```

This implementation counted the length of the messages in the variable `overallLoad` whenever a new message is ready. When the sample rate has been hit it generates a freeze of the current value of the `overallLoad` in the variable `overallFreeze` which is used by the output part to display the load on the display. For this purpose the CAN Bus Analyzer Tool has been set to repeatedly send a message over the CAN bus. According to the times this message has been sent in a second, the length of the message, and the maximum bits that can be received in a second the relative load could be calculated by hand and then compared against the value on the LCD.

The next thing that needed to be evaluated was the calculation of different module IDs. For this purpose the CAN Bus Analyzer Tool was configured to send messages with low IDs repeatedly in order to monitor its output on the LCD. Then the values for the different module IDs have been compared against the values calculated by hand.

### 6.4 Results

The last part was to put all evaluated parts together and verify the working of the HDL model. This has also been done partially at first. The monitor part has been completely evaluated with the verification of the load calculation module. For the output part different hard-coded values have been used for the load values and the overall load value. Then the output module had to correctly calculate the corresponding relative load, display it on the LCD and send the raw values via UART to a PC, which had to display the values correctly.

After this has been achieved the system had been put through an overall test. The CAN Bus Analyzer Tool has been configured to display different sets of messages, Tables 3 to 6 show the four different configurations that have been used for evaluation, with  $D_n$  ( $0 \leq n \leq 7, n \in \mathbb{N}$ ) being the  $n$ -th data byte in the message and an  $x$  in the ID field specifies whether it is an extended ID or not. Tables 4 to 6 can be found in the appendix in Section A.4.

Each of these four message group configurations consist of eleven messages with either shortest or longest length:

- Standard data frame with zero bytes data, a total of 44 bits
- Standard data frame with eight bytes data, a total of 108 bits
- Extended data frame with zero bytes data, a total of 64 bits
- Extended data frame with eight bytes data, a total of 128 bits

These four different configurations have been chosen to verify the working of extended identifiers, because the module ID  $0x$  in an extended data frame is



## 6.4 Results

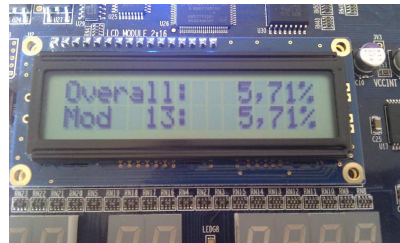
---

ID	DLC	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	T <sub>Δ</sub>
0x	8	1	2	3	4	5	6	7	8	50
1x	8	9	10	11	12	13	14	15	16	50
2x	8	17	18	19	20	21	22	23	24	50
3x	8	25	26	27	28	29	30	31	32	50
4x	8	33	34	35	36	37	38	39	40	50
5x	8	41	42	43	44	45	46	47	48	50
6x	8	49	50	51	52	53	54	55	56	50
7x	8	57	58	59	60	61	62	63	64	50
8x	8	65	66	67	68	69	70	71	72	50
9x	8	73	74	75	76	77	78	79	80	50
10x	8	81	82	83	84	85	86	87	88	50

**Table 3:** The first group of CAN messages used for the evaluation. Each message has a length of 128 bits.

the same as the module id 0 in a standard data frame. Furthermore different message lengths needed to be verified as well.

To verify that the LCD with the FPGA floating point division and the UART output with the floating point division on the PC side is working correctly different messages have been sent and checked for correctness. Figure 6.1 and 6.2 show the same output when sending a constant stream of standard data frames with the ID 13 and eight bytes of data 50 times a second. The difference of 0.01% is as discussed in Chapter 6.2.1 due to the rounding in the IEEE754 conversion part of the output part.



**Figure 6.1:** The output on the LCD when sending 50 standard data frames a second with the message ID 13 and eight bytes of data.

```
can bus monitor - uart receiving tool

make sure the usb cable is plugged in and the driver is working
press ctrl-c to quit

overall load:      59950 bit/s, 5.72%
module   0:        0 bit/s, 0.00%
module   1:        0 bit/s, 0.00%
module   2:        0 bit/s, 0.00%
module   3:        0 bit/s, 0.00%
module   4:        0 bit/s, 0.00%
module   5:        0 bit/s, 0.00%
module   6:        0 bit/s, 0.00%
module   7:        0 bit/s, 0.00%
module   8:        0 bit/s, 0.00%
module   9:        0 bit/s, 0.00%
module  10:        0 bit/s, 0.00%
module  11:        0 bit/s, 0.00%
module  12:        0 bit/s, 0.00%
module  13:      59950 bit/s, 5.72%
module  14:        0 bit/s, 0.00%
module  15:        0 bit/s, 0.00%
module  16:        0 bit/s, 0.00%
module  17:        0 bit/s, 0.00%
module  18:        0 bit/s, 0.00%
module  19:        0 bit/s, 0.00%
module  20:        0 bit/s, 0.00%
module  21:        0 bit/s, 0.00%
module  22:        0 bit/s, 0.00%
module  23:        0 bit/s, 0.00%
module  24:        0 bit/s, 0.00%
module  25:        0 bit/s, 0.00%
module  26:        0 bit/s, 0.00%
module  27:        0 bit/s, 0.00%
module  28:        0 bit/s, 0.00%
module  29:        0 bit/s, 0.00%
module  30:        0 bit/s, 0.00%
module  31:        0 bit/s, 0.00%
```

**Figure 6.2:** The output on the PC console when sending 50 standard data frames a second with the message ID 13 and eight bytes of data.

The whole project needs about 16,037 logic elements with the ability to monitor 32 modules simultaneously. Each additional module needs approximately 88 logic elements. Thus a design with the ability to monitor 200 modules needs about 30,821 logic elements, which is compared to the maximum amount of available logic elements on the Altera DE2 board 92.79%.

If more modules need to be monitored a different implementation needs to be chosen that stores the load values in the on-board SDRAM. The SDRAM

## 6.4 Results

---

on the Altera DE2 board has a width of eight megabytes. As each additional module has a width of 21 bits all  $2^{19}$  modules would need 11,010,048 bits of space, which is 10.5 MB and more than the available space in the SDRAM. In the SDRAM would fit approximately 399,457 out of the 524,288 modules.

---

## 7 Conclusion and Outlook

This report has shown an implementation of a CAN bus monitor on the Altera DE2 platform. It has been completely realized in the Altera Quartus II design software in Verilog HDL with the use of IP cores for the floating point division and the transmission of data over UART. The project has been split into two parts for abstraction, the monitoring part and the output part. The monitoring part handles the CAN driver and the load calculation, whereas the output part handles the display driver for the LCD with the conversion of numbers into strings and into the IEEE754 format, the UART driver and the transmission protocol for the data to a PC.

The HDL model was designed to monitor 32 modules and stores the data directly in the FPGA without using the on-board SDRAM, which has mainly been chosen because of speed issues. It is extensible for up to 200 modules, that will directly fit into the registers. Any number of modules above 200 need an alternative implementation using the SDRAM, but due to the eight MB limitation of the SDRAM only 399,457 out of the 524,288 possible modules can be stored.

Some aspects of this project can be improved by taking a closer look at the implementation of the synchronization algorithm of the CAN bus driver, the rounding error on displaying the numbers on the built-in LCD, and the transmission of the module with the ID zero via UART to a PC. The synchronization algorithm has been implemented with the calculation of the ticks needed of the FPGA clock to be between phase one and phase two of the bit timing on the CAN bus. As this project uses only the calculated number approximately 3 out of 200 messages are received incorrectly. This number can be tweaked by trying different tick settings for the bit timing to get a more stable result.

As discussed before only the ten first bits of the mantissa are used for the conversion of an IEEE754 number into a string. This can be furthermore tweaked by taking more numbers into consideration for the calculation. The

---

error as of now is about 0.002%, but the error can be removed with higher precision. The last part that can be improved is that sometimes due to the loss of synchronization the transmission via UART of the load value for the module with the ID zero is erroneous. This can be remedied by either introducing stuff bits as well or by setting the three bits between the ID and the load value to a value other than three zeroes.

The developed HDL model has been evaluated using two Microchip CAN Bus Analyzer Tools, which are capable of sending CAN messages. The advantage of using these tools is that they have a deterministic behaviour. This HDL model will be used by the University of New Brunswick for monitoring the communication in their developed prosthetic hand.

The usage of this implementation is not only limited to monitoring the CAN bus system of the UNB hand, but can also be used on other devices that utilize a CAN bus. As this project only focus on the raw CAN bus data it constantly reads CAN messages and only interprets the message IDs and allocates the load date to the corresponding IDs in the project. This results in a tool which can be used for monitoring CAN buses no matter in what application they are used, e.g. in monitoring the bus system in a car. Furthermore only the receive (RX) wire of the CAN bus is connected to the FPGA, so that this implementation does not have a big influence on the working of the CAN bus.

This project can be used for many different applications that use a CAN bus as the underlying bus system. It can be used for instance in research when the engineers try to find out what module is causing how much load on the bus in order to implement further techniques for reducing the bus load. If a module puts a higher load on the bus than it is supposed to it might be an indication that this specific module has not been implemented correctly. Furthermore it can also be used in the industry, e.g. in the automotive industry, to actually look at the bus load during normal operation. In the automotive industry as well as in other areas it is crucial to not utilize the whole bandwidth of the bus in order for some high priority messages to

---

be sent that are time-critical, e.g. the opening signal for the airbag, which is not allowed to arrive late. Thus the engineers can optimize their system using this implementation.

# A Appendix

## A.1 Data Frames on the CAN Bus

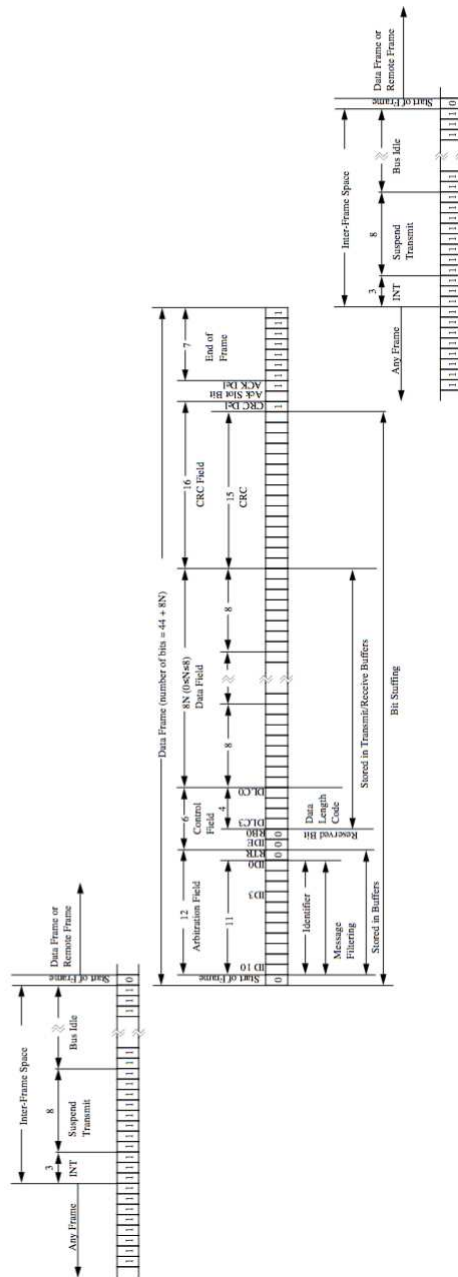


Figure A.1: A standard data frame on the CAN bus [4].

## A.1 Data Frames on the CAN Bus

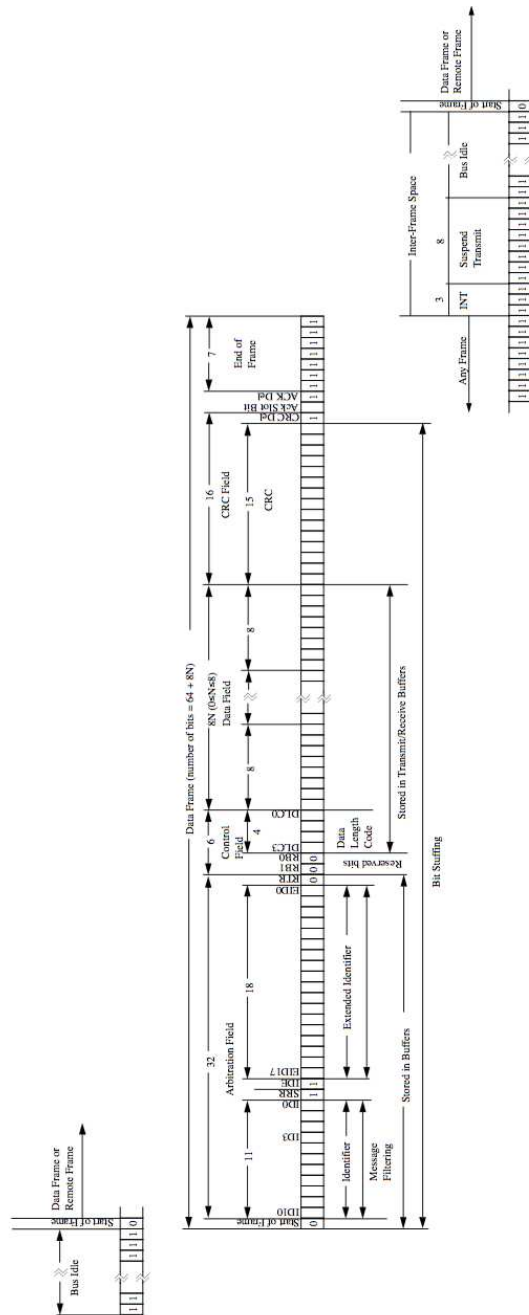


Figure A.2: An extended data frame on the CAN bus [4].



## A.2 Opening the UART Port in C++

```
1 // open the serial port and set to B115200, no parity bits,
2 // and 2 stop bits
3 void open_port(char *port){
4     fd = open(port, O_RDWR | O_NOCTTY | O_NONBLOCK);
5
6     // if open is unsuccessful
7     if(fd == -1)
8     {
9         printf("ERROR! unable to open %s", port);
10        endwin();
11        exit(EXIT_FAILURE);
12    }
13    fcntl(fd, F_SETFL, O_ASYNC);
14
15    // structure to store the port settings in
16    struct termios port_settings;
17    tcgetattr(fd, &port_settings);
18
19    // set baud rates
20    cfsetispeed(&port_settings, BAUDRATE);
21    cfsetospeed(&port_settings, BAUDRATE);
22
23    port_settings.c_cflag |= (CLOCAL | CREAD);
24    // set not parity, 2 stop bits, data bits
25    port_settings.c_cflag &= ~PARENB;
26    port_settings.c_cflag |= CSTOPB;
27    port_settings.c_cflag &= ~CSIZE;
28    port_settings.c_cflag |= CS8;
29    port_settings.c_cflag &= ~CRTSCTS;
30    port_settings.c_lflag &= ~(ICANON | ECHO | ISIG);
31    cfmakeraw(&port_settings);
32    // apply the settings to the port
33    tcsetattr(fd, TCSANOW, &port_settings);
34 }
```

### A.3 The Selftest Module

```
1 always@(posedge sampleRateHit)
2 begin
3     case(state)
4         0, 1, 2, 3:
5             begin
6                 testLine1 = "CAN_Bus_Monitor";
7                 testLine2 = "(c)_UNB_2011";
8             end
9         4, 5, 6, 7:
10            begin
11                testLine1 = "Connect_CAN_wire";
12                testLine2 = "to_GPIO1_2_blue";
13            end
14        8, 9, 10, 11:
15            begin
16                testLine1 = "GPIO1_12_black";
17                testLine2 = "Init_selftest";
18            end
19        12, 13, 14, 15:
20            begin
21                testLine1 = "GPIO0_2-12(even)";
22                testLine2 = "the_UART_plug";
23            end
24        16:
25            begin
26                testLine1 = "0000000000000000";
27                testLine2 = "0000000000000000";
28            end
29        17:
30            begin
31                testLine1 = "1111111111111111";
32                testLine2 = "1111111111111111";
33            end
34        18:
35            begin
36                testLine1 = "2222222222222222";
37                testLine2 = "2222222222222222";
38            end
39        19:
40            begin
```

```
41         testLine1 = "3333333333333333";
42         testLine2 = "3333333333333333";
43     end
44     20:
45     begin
46         testLine1 = "4444444444444444";
47         testLine2 = "4444444444444444";
48     end
49     21:
50     begin
51         testLine1 = "5555555555555555";
52         testLine2 = "5555555555555555";
53     end
54     22:
55     begin
56         testLine1 = "6666666666666666";
57         testLine2 = "6666666666666666";
58     end
59     23:
60     begin
61         testLine1 = "7777777777777777";
62         testLine2 = "7777777777777777";
63     end
64     24:
65     begin
66         testLine1 = "8888888888888888";
67         testLine2 = "8888888888888888";
68     end
69     25:
70     begin
71         testLine1 = "9999999999999999";
72         testLine2 = "9999999999999999";
73     end
74     endcase
75     state = state + 1;
76     if (state > 25)
77         finished = 1;
78 end
```

#### A.4 Groups of Messages Used for the Evaluation

ID	DLC	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	T <sub>Δ</sub>
0x	0									50
1x	0									50
2x	0									50
3x	0									50
4x	0									50
5x	0									50
6x	0									50
7x	0									50
8x	0									50
9x	0									50
10x	0									50

**Table 4:** The second group of CAN messages used for the evaluation. Each message has a length of 64 bits.

ID	DLC	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	T <sub>Δ</sub>
0	8	1	1	1	1	1	1	1	1	50
1	8	1	1	1	1	1	1	1	1	50
2	8	1	1	1	1	1	1	1	1	50
3	8	1	1	1	1	1	1	1	1	50
4	8	1	1	1	1	1	1	1	1	50
5	8	1	1	1	1	1	1	1	1	50
6	8	1	1	1	1	1	1	1	1	50
7	8	1	1	1	1	1	1	1	1	50
8	8	1	1	1	1	1	1	1	1	50
9	8	1	1	1	1	1	1	1	1	50
10	8	1	1	1	1	1	1	1	1	50

**Table 5:** The third group of CAN messages used for the evaluation. Each message has a length of 108 bits.

#### A.4 Groups of Messages Used for the Evaluation

---

ID	DLC	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	T <sub>Δ</sub>
0	0									50
1	0									50
2	0									50
3	0									50
4	0									50
5	0									50
6	0									50
7	0									50
8	0									50
9	0									50
10	0									50

**Table 6:** The fourth group of CAN messages used for the evaluation. Each message has a length of 44 bits.

## References

- [1] “<http://www.smpp.northwestern.edu/research/biomechatronics/news.html>,” 2011.
- [2] C. Maxfield, *The Design warrior’s guide to FPGAs: Devices, tools and flows*, vol. 1. Elsevier, 2004.
- [3] “<http://www.can-cia.de/index.php?id=161>,” 2011.
- [4] K. Pazul, “Controller Area Network (CAN) Basics,” *Microchip Technology Inc. Preliminary DS00713A-page*, vol. 1, 1999.
- [5] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, “Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised,” *Real-Time Systems*, vol. 35, pp. 239–272, Jan. 2007.
- [6] 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.
- [7] “<http://standards.ieee.org/about/get/802/802.3.html>,” 2011.
- [8] J. W. Liu, C.L.; Layland, “Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment Scheduling Algorithms for Multiprogramming,” *Computing*, no. 1, pp. 46–61, 1973.
- [9] F. Hartwich and A. Bassemir, “The configuration of the CAN Bit Timing,” in *6th International CAN Conference*, pp. 1–10, 1999.
- [10] “754-2008 IEEE Standard for Floating-Point Arithmetic,” pp. 1–58, 2011.
- [11] P. Chu, “FPGA prototyping by Verilog examples: Xilinx Spartan-3 version,” *Interface*, pp. 215–234, 2008.
- [12] 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.

## REFERENCES

---

- [13] 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.
- [14] R. Li and C. Liu, "A design for automotive CAN bus monitoring system," *Vehicle Power and Propulsion Conference*, pp. 1–5, Sept. 2008.
- [15] 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.
- [16] 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.
- [17] J. Mendozajasso, G. Ornelasvargas, R. Castanedamiranda, E. Venturaramos, a. Zepedagarrido, and G. Herreraruiiz, "FPGA-based real-time remote monitoring system," *Computers and Electronics in Agriculture*, vol. 49, pp. 272–285, Nov. 2005.
- [18] 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.
- [19] "<http://opencores.org/project,can>," 2011.
- [20] Bosch, "Automotive Electronics License Conditions - CAN Protocol License - CAN IP modules for use in FPGAs," pp. 2–4.
- [21] Altera, "Floating-Point Megafunctions User Guide," *Design*, no. May, 2011.
- [22] "<http://www.altera.com/education/univ/materials/boards/de2/unv-de2-board.html>," 2011.

## REFERENCES

---

- [23] “[http://www.microchip.com/stellent/idcplg?IdcService=SS\\_GET\\_PAGE&nodeId=1406&dDocName=en546534](http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en546534),” 2011.
- [24] F. T. D. I. Limited, “TTL to USB Serial Converter Range of Cables Datasheet,” *Technology*, 2010.
- [25] “<http://www.cygwin.com/>,” 2011.
- [26] P. Prinz and T. Crawford, *C in a Nutshell*. O’Reilly Media, Inc., 2005.



---

## **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