

CHALMERS



Low-Response-Time PC Interface for Real-Time Analysis of Embedded Systems

Master of Science Thesis in Embedded Electronic System Design

MALLIGARAJ MALLESWARAN

VIKTOR JARENFOR

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden September 2015

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Low Response Time PC Interface for Real-Time Analysis of Embedded Systems

Malligaraj Malleswaran, Viktor Jarenfors

© Malligaraj Malleswaran, September 2015.

© Viktor Jarenfors, September 2015.

Examiner: Per Larsson-Edefors

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden September 2015

Abstract

Automotive industries are constantly increasing the number of electronically controlled units in a vehicle to provide performance, comfort and extra features to the user. Communication between these different electronic devices commonly occur through the Controller Area Network (CAN) bus. Due to the presence of a large number of nodes in the network, testing different nodes can be cumbersome. So to make the testing and verification efficient and precise, a system that acts as an interface between a Personal Computer (PC) and the CAN network has been designed in this project. Since CAN messages can be scheduled within 1 ms time frames at maximum baud rate, the response time of this interface must be of the same order to avoid missing any messages. With the help of this interface, specific test sequences can be injected to a particular node and the response can be observed. It can also be used to monitor the messages in the network at frame level. A prototype of such an interface has been designed and implemented using STM3240G-Eval Evaluation board. Ethernet and Universal Serial Bus (USB) are used as the physical interface between PC and Microcontroller. An evaluation of different physical interfaces were performed and it was found that Ethernet had a response time of 0.4 ms making it suitable for this application. On the other hand USB in high-speed mode had a very low response time of 0.08 ms and so can be used for applications that require very high performance.

Acknowledgements

We would like to thank the following people for their help and support throughout the project,

- Per Larsson-Edefors, Examiner at Chalmers University of Technology for his support.
- Lena Peterson, Supervisor at Chalmers University of Technology for her valuable feedback and discussion during the project.
- Shwan Ciyako, Supervisor at QRTECH AB for his guidance at the company.
- Linus Lundin, Nikola Vorkapic, Lars-Åke Johansson, Joakim Plate, Anders Sandblad at QRTECH AB for their technical insights and suggestions.

The Authors, Gothenburg, September 9, 2015

Abbreviations

ACK	Acknowledgement
AMP	Arbitration on Message Priority
APB	Advance Peripheral Bus
API	Application Programming Interface
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information and Interchange
CAN	Controller Area Network
CD	Collision Detection
CDC	Communication Device Class
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
DB	D-Sub-miniature
DLC	Data Length Code
DLL	Dynamic Link Library
ECU	Electronic Control Unit
EOF	End of Frame
FCS	Frame Check Sequence
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FS	Full Speed

GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HID	Human Interface Devices
HS	High Speed
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IFS	Inter-Frame Space
IO	Input/Output
IP	Internet Protocol
ISO	International Standards Organization
LAN	Local Area Network
LLC	Logic Link Control
LSB	Least Significant Bit
LwIP	Low weight Internet Protocol
MAC	Media Access Control
MCU	Microcontroller
MSC	Mass Storage Class
NRZ	Non-Return to Zero
OS	Operating System
OSI	Open Systems Interconnections
OTG	On-the-Go
PC	Personal Computer
PIC	Programmable Integrated Circuit
RAM	Random Access Memory
RS232	Recommended Standard 232
RTR	Remote Transmission Request
RTS	Real Time System
RTT	Round Trip Time
SoC	System on Chip

SOF	Start of Frame
SRAM	Static Random Access Memory
SRR	Substitute Remote Request
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WSA	Windows Sockets API

Contents

1	Introduction	1
1.1	Goal	2
1.2	Scope and Limitation	2
1.3	Report Outline	3
2	Technical Background	4
2.1	Open System Interconnection (OSI) Model	4
2.2	Controller Area Network (CAN)	5
2.3	Ethernet	8
2.3.1	Transmission control protocol (TCP)	9
2.3.2	User Datagram Protocol (UDP)	9
2.4	Universal Serial Bus (USB)	10
2.5	RS232	12
3	Literature Review	13
3.1	Similar Systems	15
3.2	QRCAN	15
4	Design and Implementation	16
4.1	System Overview	16
4.2	STM3240G - EVAL Evaluation Board	17
4.3	Ethernet Communication	18
4.3.1	Light-Weight Internet Protocol (LwIP)	18
4.3.2	Transmission Control Protocol (TCP)	19
4.3.3	User Datagram Protocol (UDP)	21
4.3.4	Routing Server	22
4.4	USB Communication	24
4.4.1	USB Host	24
4.4.2	USB Device	24
4.5	CAN Implementation	25

4.6	PC-MCU Communication Protocol	26
4.7	Scheduler	27
4.8	BUSMASTER	28
5	Evaluation	30
5.1	Ethernet Performance	30
5.2	Routing Server	31
5.2.1	Threaded Routing Server	31
5.2.2	Event-driven routing server	32
5.3	USB Performance	33
5.4	Response Time Comparison	33
6	Discussion and Future Work	35
6.1	Physical Interface	35
6.2	Routing Server	35
6.3	Scheduler	36
6.4	BUSMASTER Plug-in	36
7	Conclusion	37
	Bibliography	38
	Appendices	
A	Response Times Plots	

1

Introduction

Electronic Control Units (ECUs) are used to control almost every aspect of the vehicle. Automotive systems use these ECUs in engine control, active safety, driver assistance, comfort etc. The basic necessity of any system is a reliable communication network between different components. With a common communication network one does not have to change the wiring between components for every different configurations of various components [1]. It also gives the flexibility of adding and removing devices from the existing network without affecting the communication between other devices. The data that is transferred in this communication network are to a large extent safety critical. For example in a brake by wire system, any corruption of data might result in fatality. So the time and integrity at which the messages reach different destinations is vital. The communication must be highly stable and reliable to not degrade the performance of the entire system.

With the help of a CAN network, different ECUs in a vehicle can communicate with each other without a host computer [2]. The CAN protocol is a broadcast protocol, this means that all the nodes in the network receive every message, but nodes that are not expecting any messages just discard them. The CAN protocol facilitates the detection of collision in messages and allows transmission of data based on priority of the nodes.

Some ECUs are used in applications such as transmission control, anti-lock braking system, fuel injection etc. that are safety-critical, hence testing and verification of all components of these systems is crucial. The complete system can be tested by sending test data to all the components through the communication network and analyzing the responses or each component can be tested individually. For doing such testing of individual components, a high-performance interface to a PC is necessary. Now the rest of the CAN network can be simulated in the PC and data sequences corresponding to a particular node can be sent from the PC. The basic block diagram of the system is shown in Figure 1.1.

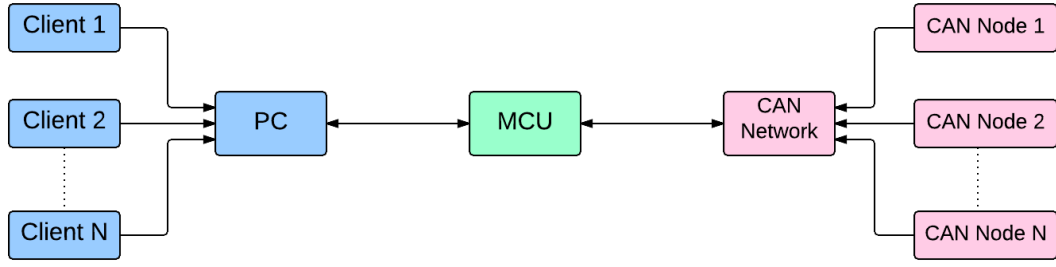


Figure 1.1: An overall block diagram with the components from PC, Interface and CAN network

As shown in Figure 1.1, the device under test could be one of the nodes in the CAN network and different applications on PC could send test sequences through the Microcontroller (MCU) that acts as an interface.

1.1 Goal

The main goal of this project is to specify and develop a prototype of a high-performance interface between PC and CAN network . This interface can be used to monitor messages on the CAN network. The PC can emulate a node in the network so that it can send/receive customized messages to/from the network through the interface. This prototype has to be developed using an Advanced RISC machine (ARM) microcontroller (MCU), as the ARM architecture provides high performance necessary for this kind of applications. An optimal physical interface between the MCU and PC has to be chosen and an efficient communication protocol has to be developed. A routing server mechanism that allows multiple application clients on PC to communicate with MCU has to be designed. Further the response time of the interface should be in the order of 1 ms to avoid dropping frames at the maximum baud rate of the CAN network.

1.2 Scope and Limitation

This project is focused on the technical aspects of the prototype, so that parameters like performance and reliability are covered. All the implementations on the PC side are designed based on Windows because of its prominent use in industry. Thus, there is no guarantee that the same implementation will work on other Operating Systems (OSs).

1.3 Report Outline

The rest of the report is organized as follows. In Chapter 2 the necessary technical background required to understand different design choices to the project is given. Chapter 3 provides a literature review and a review of similar solutions available on market to indicate the need to develop such a prototype. The design and implementation of the prototype is given in Chapter 4 followed by the evaluation of implementation in Chapter 5. Thereafter a discussion of the previously evaluated results and further improvements that can be added is given in Chapter 6. Finally Chapter 7 summarizes and concludes the project.

2

Technical Background

This chapter provides the required knowledge to understand the design choices and implementation of the prototype. Relevant material regarding various physical mediums used in the project such as Ethernet, USB and CAN are provided. The information found here is limited to the project area and does not cover all aspects of each medium. With the basic understanding of how these components, media, protocols, algorithms and functions operate, ease of insight and understanding will follow.

2.1 Open System Interconnection (OSI) Model

OSI model was developed by International Standards Organization (ISO). It describes how data from an application on one computer is transferred to an application on another computer. The model contains seven different layers as shown in Figure 2.1 [3].

Each layer works independent of each other and data is transferred between different layers. The bottom most layer is the physical layer that does the actual transmission of data over a medium of choice (Eg. Ethernet, WiFi, Optical Fiber Cable etc). The digital transmission occur through signals that represent low and high voltage levels. The next layer is the data link layer containing the basic functions that are necessary for the transfer of data from one point to another. The data link layer takes care of error detection, framing and addressing of data. Further the data layer provides access permissions to the device in the network to transmit data. The network layer is responsible for routing of data based on the source and destination addresses (Eg. Internet Protocol (IP)). The transport layer controls the reliability of a given link between a source and destination. The transport layer also handles acknowledgements, congestion and collision control between different nodes in the network. Example transport layer protocols are Transmission Control Protocol (TCP), User Datagram Protocol (UDP)

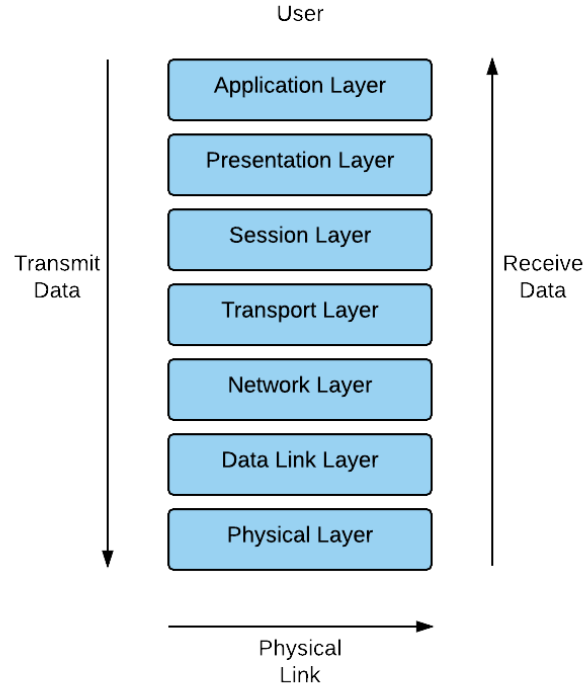


Figure 2.1: OSI Reference Model [3]

etc. The next layer called the sessions layer is responsible for opening and closing of a connection. It also takes care of the direction of the data flow. The presentation layer encodes and decodes the data between the application layer and all other layers below it. It converts the data into human readable form for the applications running on a computer. Finally the application layer is closest to the user (Eg. Web Browser, File transfer applications, Mail clients like Outlook, Thunderbird etc.) and can be used to initiate the data transfer process.

2.2 Controller Area Network (CAN)

Controller Area Network (CAN) bus is the most commonly used vehicle bus standard that allows communication between the different ECUs in a vehicle without the help of a host computer [2]. It was developed by Bosch and is a twisted pair cable multi-master asynchronous serial communication protocol. The CAN 2.0 standard has a maximum baud rate of 1Mbit/sec and the bit rate depends on the length of the bus.

The CAN communication protocol can be broken down into different abstraction layers

as explained in Section 2.1. Since CAN is a broadcast protocol appropriate filtering is done at the application layer of each node to accept messages that are addressed to that particular ECU. The next layer is the transport layer that handles the main processes of communication such as error detection, acknowledgement, message framing etc. It is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (Carrier Sense Multiple Access (CSMA)/Collision Detection (CD) + Arbitration on Message Priority (AMP)). In case of a collision CSMA is responsible for making different devices in the network to wait for a random period of idleness before sending a message. CD+AMP takes care of collisions with a bit-wise arbitration based on a static priority assigned to each message. The CAN uses “dominant” bits denoting a logical 0 and “recessive” bits denoting a logical 1. An idle state is represented by a recessive bit. So when a node transmits a dominant bit and another node transmits a recessive bit, the contention is solved by giving priority to the node that transmitted the dominant bit. The lowest layer is the physical layer that is a twisted-pair wire with 9-pin D-sub type connector. The messages are transmitted and received serially in this layer using non-return-to-zero (NRZ) format.

The CAN messages are sent in the form of frames and are of four types:

- Data Frame** A frame that contains the data to be transmitted.
- Remote Frame** A frame that is used to request retransmission from a specific node.
- Error Frame** A frame that is transmitted by any node that detects an error.
- Overload Frame** A frame that is used to inject a delay between data and/or remote frame

Data Frame Format

There are two types of CAN data frames namely Standard CAN and Extended CAN. The difference is that the standard version uses 11-bit identifiers whereas the extended version uses 29 bits. The Standard message format is shown in Figure 2.2.

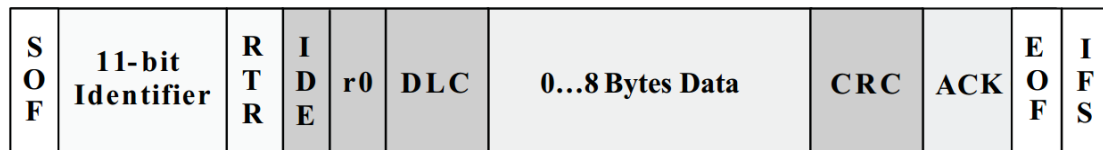


Figure 2.2: Standard CAN Message Format [4]

The different fields in the message are explained below.

- **SOF** - The Start of frame is used to synchronize the nodes on the bus after being idle.

- **Identifier** - It is used to denote the priority of the message; the lower the value of this, higher the priority.
- **RTR** - The Remote transmission request bit is made dominant when information is required from another node. The identifier determines a specific node and is the same case while receiving the response as well.
- **IDE** - A dominant bit in Identifier extension means that a standard CAN identifier with no extension is being transmitted.
- **r0** - Reserved bit
- **DLC** - Data length code is 4 bits long and denotes the number of bytes of data that is being transmitted
- **Data** - Upto 8 bytes of data can be transmitted
- **CRC** - A 16-bit Cyclic redundancy check contains the checksum of the preceding data for error detection.
- **ACK** - This bit is changed to a dominant bit if no error is detected in the message that was sent. But if the receiving node detects an error, it leaves this bit to be recessive and the sending node re-sends the data after arbitration.
- **EOF** - The 7-bit End of frame field marks the end of a CAN frame.
- **IFS** - This 7-bit Inter-frame space provides the time required by the controller to move a correctly received frame to its proper position in the message buffer.

The Extended message format is shown in Figure 2.3.

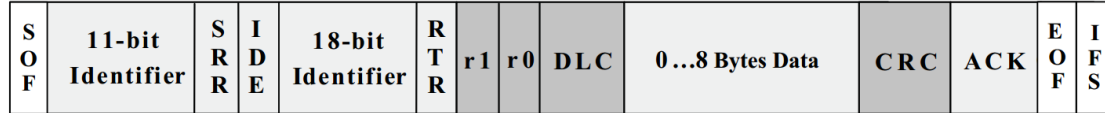


Figure 2.3: Extended CAN Message Format [4]

The extended message format has the same structure except for a few extra fields that are listed below.

- **SRR** - The Substitute remote request replaces the RTR of the standard frame as a placeholder in the extended format
- **IDE** - A recessive bit in this field indicates that there are more identifier bits to follow.
- **r1** - An additional reserve bit.

A message is considered error free when the last bit of the EOF field in a message is received in an error-free recessive state. A dominant bit in the EOF will cause an error

and the transmitter should repeat the transmission. The CAN protocol is considered very robust due to three message level and two bit level error checking procedures. The error checking is ensured by the use of CRC and ACK slots in the CAN frame.

2.3 Ethernet

Ethernet is a physical-layer Local Area Network (LAN) technology and has become the most prominently used LAN technology due to its speed, cost and ease of installation. It was invented by Robert Metcalfe in 1979 and is used to connect multiple computers in a building using hardware [5]. It was standardized in 1983 as IEEE 802.3 and has since then been refined to support higher bit rates and longer link distances. An Ethernet LAN typically uses coaxial cable or special grades of twisted pair wires. Systems communicating over Ethernet divide a stream of data into smaller pieces called frames. According to the Open Systems Interconnection (OSI) model Figure 2.1, Ethernet provides services up to and including the data link layer. The data link layer is responsible for access control, flow control and error correction during the communication. The structure of an Ethernet frame is shown in Figure 2.4.

Preamble	SOF	Destination Address	Source Address	Length	LLC	Data	Pad	FCS
----------	-----	---------------------	----------------	--------	-----	------	-----	-----

Figure 2.4: Structure of Ethernet Frame [5]

The different fields in the Ethernet frame structure are,

- **Preamble** - It signals that a frame is being sent and sets bit timing.
- **SOF** - An 8-bit sequence indicating the Start of frame
- **Destination Address** - It is a 48-bit hardware Media access control (MAC) address of the destination.
- **Source Address** - 48-bit MAC address of the transmitting device
- **Length** - Indicates the length of the data field
- **LLC** - The Logical link control governs the assembly of data at the data link layer of the Open Systems Interconnection (OSI) model of the communication system.
- **Data** - It can hold 46 to 1500 bytes of data.
- **Pad** - A number of bits with the value 0 are added to the end of data field if there are fewer than 46 bytes of data to have a packet of minimum size.
- **FCS** - The Frame check sequence detects transmission errors and provides quality of service at the receiving end.

The layer above the data link layer would be the Network layer that manages multi-node network by addressing and routing the packets. Internet Protocol (IP) is the mainly used protocol in this layer. The IP has the main task of delivering a packet from a particular source to a specific destination based on the IP addresses. Every device connected to the network is assigned an IP address to identify itself in the network. It is with these IP addresses the transactions occur in a point to point fashion. The next layer is the transport layer comprised of protocols like Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

2.3.1 Transmission control protocol (TCP)

The TCP is a standard that defines how to establish and maintain a network conversation through which application programs can exchange data. It works with IP, which defines how computers send packets of data between each other. TCP is a connection-oriented protocol, which means a connection is established between the sender and receiver and maintained until the application programs at each end have finished exchanging messages. It determines how to break application data into packets that networks can deliver [6].

An application does not need to know any particular mechanism for sending data via a link to another host. The TCP at transport layer handles all the handshaking and transmission details and presents an abstraction of the network connection to the application. At the lower levels of the protocol stack the packets may be lost or delivered out of order due to network congestion or traffic load balancing. TCP detects these problems and requests retransmission of lost data, rearranges out-of-order data and even helps to minimize network congestion to reduce the occurrence of other problems. A technique called positive acknowledgement with retransmission is used to ensure reliability of packet transfer. So the receiver responds to each data received with an acknowledgement message and the sender keeps track of each packet it sends. Further the sender logs the time of each message sent and does a retransmission is done if the timer expires before the message is acknowledged. TCP is optimized for accurate delivery rather than timely delivery.

2.3.2 User Datagram Protocol (UDP)

UDP is a connectionless message based transmission model with no handshaking or error corrections. In UDP the data is transferred in the form of small chunks called datagram using a source and destination address. In this protocol the receiver does not generate an acknowledgement of packet received and in turn the sender does not wait for any acknowledgement making it unreliable. Retransmission and the need to reorder packets after they arrive can introduce latency in a TCP stream. Highly time-sensitive applications like Voice over IP and streaming video generally rely on a transport like

UDP. These applications can afford to lose a few packets than to have a delayed reception of packets. Like TCP, UDP also works with the IP to get a packet from one node to another node. The application uses the software port number to send its data. Different network applications running on the PC are differentiated by the port numbers. If a higher reliability is necessary and TCP protocol is not an option, implementations in the upper layers must be made.

2.4 Universal Serial Bus (USB)

The Universal Serial Bus is a high-speed wired serial communication standard (similar to Ethernet) between electronic components. It is a host controlled protocol and allows a single host in a system. The USB is based on “Tiered Star Topology” in which a single host can be connected to up to 127 devices acting as slaves and works based on different abstraction layers shown in Figure 2.1. The slave devices connect to the host through a physical medium referred to as Ports. The USB 1.1 standard supports two speeds namely Low speed (1.5 Mbits/s) and Full speed (12 Mbits/s) and the upgraded version USB 2.0 supports an additional High speed (480 Mbits/s) [7]. Only the host can initiate a communication and so no two slave devices can communicate with each other without involving the host. In the same way if a device needs to transfer some data to the host it has to wait until the host has requested the data. There is an additional tack in the USB specification called “On-the-Go” that has implemented the host negotiation protocol. With the help of the host negotiation protocol two devices can negotiate for the role of host. The slave devices can also draw power (5 V, 100 mA) from the host and it is also possible to remotely cut power to a particular device [8].

USB Packet

USB is made up of different layers of protocol but the USB controllers take care of the lower layers of it and make it invisible to the end designer. The endpoint in an USB communication denotes either a source or sink for data. The OUT endpoint means from host to device and similarly IN endpoint means from device to host. A pipe is a logical data connection between the host and a particular endpoint. The data to be sent are in the form of packets and it could be considered as the smallest element in the communication. The USB protocol sends the data with Least Significant Bit (LSB) first. There are four types of packets namely Token Packet, Data Packet, Handshake Packet, Start of Frame Packet. The basic structure of a packet is shown in Figure 2.5.



Figure 2.5: Structure of a USB Packet [8]

The USB packets start with a Sync field followed by the data bytes that can have the fields PID, ADDR, ENDP, CRC and ends with a EOP field.

Sync

Every packet begins with a sync field and is 8 bits long for Low and Full speed or 32 bits long for High speed modes. It is used to synchronize the clock of the receiver with that of the transmitter.

PID

The PacketID field is used to identify the type of packet that is being sent. The PID is 4 bits long but in order to ensure that it is correctly received the 4 bits are complemented and repeated to make it 8 bits in total.

ADDR

This field specifies the destination device for which the packet is intended. It is 7 bits long, thereby allowing 127 devices to be connected to a host. Address 0 is invalid as it is used by any device that has not been allocated an address and has to respond to packets sent to that address.

ENDP

The Endpoint field is 4 bits long providing 16 possible endpoints.

CRC

The Cyclic Redundancy Checks (CRC) are done on the data within the payload of the packet. It is 5 bit long (CRC5) for token packets and 16 bit long (CRC16) for data packets.

EOP

The End of Packet field marks the end of packet and is signalled by Single ended zero (SE0) for approximately two bit times.

2.5 RS232

The RS232 is a serial communication standard found commonly in computer serial ports used in the data link layer shown in Figure 2.1 [9]. The RS232 port was once a feature of personal computer used for communicating with mouse, printers, modem etc. But it is not used prominently nowadays due to its low data transmission speed, high voltage swing and size of the physical connector. The serial ports on a PC are full duplex and uses separate lines for transmitting and receiving data. The communication starts with a start bit followed by the actual data and the frame is terminated using the stop bit. The number of data bits and a baud rate is set between the transmitter and received. In addition to start and stop bits that are used for synchronization, parity bits can be added to the data to check data integrity. The RS232 uses a DB9 or DB25 connector as the physical medium.

3

Literature Review

This chapter provides an overview of previous research that has been done to design such an interface. It also explains the drawbacks of existing solutions and provides the necessary data to design and implement an interface between a PC and CAN network.

Wang and Guo [10] developed a CAN protocol converter based on Microchip's PIC18F2580 microcontroller with an integrated CAN module designed for RS232 and CAN protocol converters to facilitate the direct communication between a computer and a CAN network. The physical interface used was RS232, but nowadays this is being replaced by the more advanced physical interfaces such as USB or Ethernet. The drawback of the above system is that modern laptops no longer have RS232 connectors, so an external converter to USB or Ethernet is required. Also, the communication speed of RS232 is lower than that of USB or Ethernet, making it a less favorable option.

A hardware-based CAN bus simulation system has been implemented on a PC using the Freescale's automotive microcontroller MC9S12XEP100 with on-chip CAN controller by Luo and Liu [11]. A software simulating multiple CAN buses to support the above hardware has been developed. The software can be programmed with simulation scripts that enable the emulated nodes to react to bus events. This response to bus events might be an issue if the physical medium is chosen to be USB. Because in USB all communications are initiated by the host. So if the host has to be aware of any data that is sent by the MCU, the host has to poll the port continuously to ensure that the data is received.

The CAN bus is also used in the field of automation, as it is cheap, immune to electrical interference, is able to self-diagnose and repair data errors. Due to the critical nature of the applications, it is very important to test data on frame level and bit level in the CAN bus network. Further during testing it has become a necessity to further inject data into the CAN network. So a RS232 based CAN bus network analyzer has been designed by

Kumar [12]. The interface is designed based on ARM7 (LPC2129) microcontroller based on acceptance filter concept. The CAN Bus Analyzer tool is used to monitor the CAN network with the help of its Graphical User Interface (GUI) and interpret bus traffic.

Nedeoglo et al. [13] test their developed control system using the CAN bus adapter. The adapter is based on Philips SJA1000 CAN bus controller that provides fast access to the CAN network through direct memory mapping. A monitor program that listens and visualises traffic of CAN bus and allows users to send manually composed messages over the network was also developed. The system is supported under Linux OS.

Kashif et al. [14] developed a CAN-bus analyzer that can monitor and inject data in a CAN network. The analyzer is developed on a System on Chip (SoC) and verified by simulation and implementation on a Field Programmable Gate Array (FPGA) board. The system uses SoC for implementing the CAN controller, a microcontroller, an error injector and a serial interface.

Mostafa, Shalan and Hammad [15] propose a similar approach for testing a CAN bus at the bit level. The approach depends on the generation of bus errors to cover crucial corner cases, which in turn makes it possible to go beyond frame level testing that is provided by many commercial tools. The system also allows bit-stream level testing and data injection.

An analysis of possibilities available for testing the industrial distributed systems and their components is done by Novak et al. [16]. It mainly focuses on erroneous state handling tests caused by external electromagnetic disturbances. A so called Test Generator was developed to meet all specific test requirements. The main reason behind development of such a system is to model the errors caused by the random disturbances and use them to test the communication system and make it reliable.

Novak and Kocourek [17] design and implement an automated test site to avoid human influence in testing. It realizes more than 50 physical, link and application protocol layer tests. The tests of electronic components can be divided into several groups, according to the development phase in which they are applied. One of them is the functional testing focused on the accurate acquisition of input data and in-time evaluation of correct outputs used either directly for system control or as the inputs for other components in the system. The test site uses an FPGA for performing the tests on the ECU at both hardware and software level.

The bottleneck in such high-speed networking systems would be the operating system of the PC. The speed of the main memory and the IO devices are much lower than that of the CPU. The cache systems developed in modern computers try to bridge the bandwidth gap but the accesses generated to the IO buffers do not have sufficient locality to allow the cache system to reduce the memory traffic. This leads to a drop in IO performance and reduces the throughput thereby degrading the response time.

3.1 Similar Systems

In addition to the above research approaches there are a few commercial products that serve as interface between PC and CAN. One of the most commonly used product is CANCaseXL from Vector Informatik [18]. There are other similar systems such as PCAN of Peak Systems [19], ValueCAN from Intrepid Control Systems and CANUSB. In the automotive industry, every person developing software for an ECU will need such a system on their desk as this is the only way through which they can monitor the operation of the ECU. So any automotive development industry will require several of these interfaces. One drawback of these systems is that the user does not have any control over the way in which data is being handled inside the hardware. Further these commercial systems are expensive and commercial software to support these interfaces on the PC makes it even more expensive.

3.2 QRCAN

In an effort to make a cheaper CAN interfaces for the PC, QRTech AB [20] developed the so called QRCAN interface. This interface was used to support their Test Engine which generated various test sequences to verify product requirements. But there were a few drawbacks with the system. The response times for the messages sent from the PC were around 25 ms which was not fast enough to be used in testing CAN networks in real time. The long response time was primarily due to the fact that the PC was running Windows 7 which is not a real time OS and has several other system processes running in background. So the response time spiked when the PC was operated in heavy load. So an interface with shorter response times that are independent of PC's load had to be designed.

4

Design and Implementation

This chapter explains design and implementation of the interface system. An overview of the implementation is given followed by a short description of the hardware platform used to develop the prototype. Then the different libraries used and implementation procedure of different blocks of the system is explained.

4.1 System Overview

With the existence of QRCAN it was possible to get some motivation and inspiration on where to begin with development of such a prototype. Since the main impact on the response time of the previous system was the applications and OS of PC, all influence from the PC must be removed with the new design. One approach was to send all the messages in a burst from the PC and store them in the RAM of the MCU and execute them. To realize this approach, timestamps are required. The timestamps will represent the execution time for each message and will therefore make it possible to predict and verify message forwarding. Now the messages could be sent from the PC in bursts and executed without the influence of the PC. The idea was to send all the messages in the test sequence and perform the test from within the MCU. But the RAM available on the MCU is 8 Kbytes which is too low for this purpose.

The STM3240G-EVAL Evaluation Board was chosen to be the hardware platform of the prototype with the main reason being 192 kB of on-chip RAM and 16 Mb of external RAM that can be used. Furthermore, it also has other features that are necessary for implementing the prototype and STMicroelectronics have a very good documentation of all their products. The overview of the system is shown in Figure 4.1.

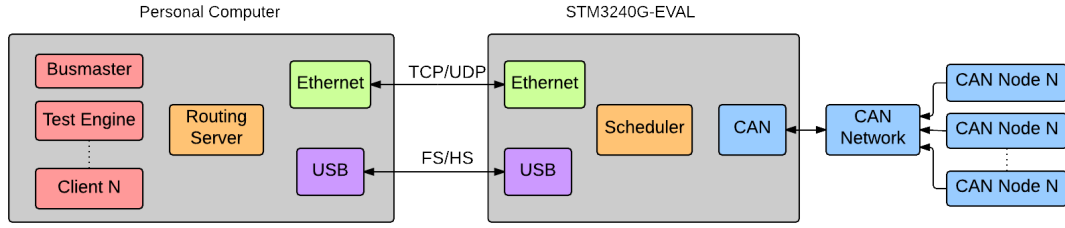


Figure 4.1: An outline of the full system with PC on one side and CAN network on the other side is shown. The evaluation board works as an interface in the middle. The different applications on a PC is used to test the CAN node.

The CAN network may contain a number of nodes that communicate with the evaluation board using a CAN physical connector. The PC side has to support multiple applications that communicate with CAN network in parallel. The support for multiple applications on the PC can be provided with the help of a routing server mechanism that routes the packets from different applications to the MCU. The physical medium between the PC and MCU could be either USB or Ethernet. USB is a good option as all modern laptops have USB ports in them that could be used to connect this device locally. But the ability to connect the device using Ethernet opens up a new possibility of having the device in a network and any PC or laptop connected to the network may use the device enabling resource sharing which is inevitable in industries nowadays.

4.2 STM3240G - EVAL Evaluation Board

The STM3240G-EVAL evaluation board is a complete demonstration and development platform for the STM32F4 series microcontroller. The key features of the evaluation board that are of importance in this application are:

- STM32F407IGH6 high-performance ARM Cortex M4 32-bit microcontroller
- 192 + 4 Kbytes of on-chip SRAM
- 16 Mbit of external SRAM
- IEEE 802.3-2002 compliant Ethernet connector
- Two CAN 2.0 A/B channels on the same DB connector
- USB OTG (HS and FS) with Micro-AB connector

4.3 Ethernet Communication

The communication between the PC and MCU with Ethernet as media is implemented using the TCP and UDP protocols. Both protocols utilize sockets to communicate with other programs using the standard Unix file descriptors. A socket can be configured to act as a server and listen for incoming connections or connect to other servers applications as a client. So a server or a client running in a computer has a socket bound to a specific port number. For the implementation in this project in both TCP and UDP, the MCU is made as the server and the PC acts as a client. The TCP/IP and UDP/IP protocol stack is accomplished in the evaluation board using the so called software architecture called Light-Weight IP.

4.3.1 Light-Weight Internet Protocol (LwIP)

The focus of the LwIP stack is to reduce memory usage and code size, making it suitable for using it in small clients with very limited resources such as embedded systems. In order to reduce processing and memory demands, LwIP uses a tailor made API that does not require any data copying. The ability to have wired or wireless networks in smaller embedded systems like sensors enable it to be connected to an existing network infrastructure such as the global Internet and can be monitored from anywhere. LwIP consists of different protocols such as TCP, UDP, IP etc. The implementation of different protocols are done in layers, where each layer solves a separate part of the communication problem. The process model used to implement the communication protocol is used to let each protocol run as a stand alone process. Further LwIP is implemented as an user space process rather than in operating system kernel, making it portable across different platforms [21].

Data Structure

In LwIP a packet is internally represented using a structure called Packet buffer (pbuf). The pbuf structure has support for both allocating dynamic memory to hold packet contents and for letting packet data reside in static memory. The pbufs can be linked together in a list called pbuf chain to accommodate large packets that may span over several pbufs. The structure of a pbuf chain is shown in Figure 4.2.

The pbuf structure consists of two pointers, two length fields, a flag field and a reference count. The *next* field is a pointer to the next pbuf in case of a pbuf chain. The *payload* pointer points to the start of the data in the pbuf. The *len* field contains the length of the data stored in that specific pbuf. The *tot_len* contains the sum of the *len* fields of all pbufs in the pbuf chain. The *flags* indicate the type of pbuf and *ref* field contains a reference count.

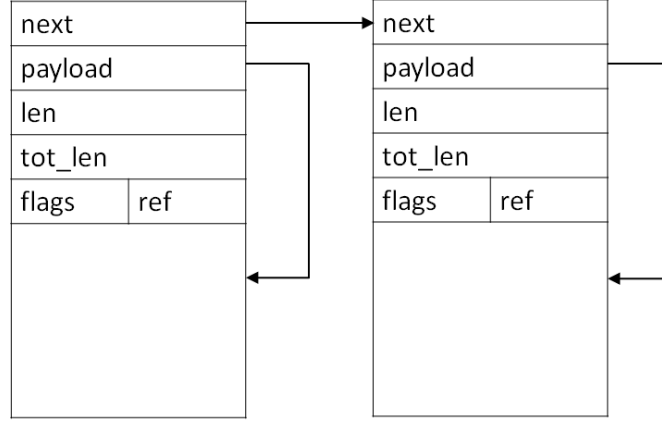


Figure 4.2: Structure of a pbuf Chain [21]

4.3.2 Transmission Control Protocol (TCP)

The implementation of TCP is made using the Socket Application Programming Interface (API) that provides the functions required to establish a communication between a client and a server. The Windows OS version of that API is known as the Winsock or Windows Sockets API (WSA). The work-flow of a TCP/IP Client-Server is shown in Figure 4.3.

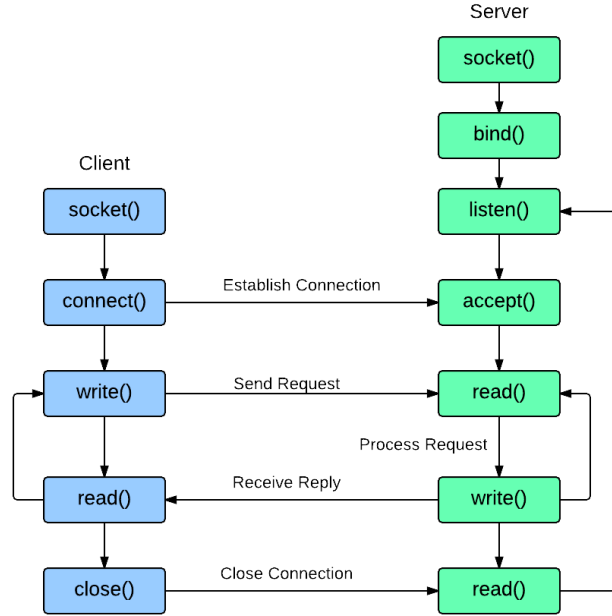


Figure 4.3: TCP/IP Client-Server Work-flow [22]

TCP Client

A client creates a socket using `socket()` that returns a socket descriptor and connects to a particular server with the IP address and port number using `connect()`. After the connection is established it sends the data request using `write()` and reads the response from the server using `read()`. Finally after completing the data transfer, the client closes the connection using `close()`. We have implemented the client on the PC using socket module of Python and time module is used for response time measurements. The time module uses the system clock of the PC to measure the time interval between two events. So a nano-second precision can be obtained using `time()`.

TCP Server

To start the communication process the server calls `socket()` which returns a socket descriptor. Then a call to `bind()` is made with an IP address and TCP port number. This makes the server bind to that port and it starts listening for connections on the port given after calling `listen()`. The server can stay here infinitely until it receives a request from a client. When the server receives a request from a client it accepts the request using `accept()`.

We have implemented the TCP server with the help of the LwIP stack. The processing of a TCP connection is shown in Figure 4.4.

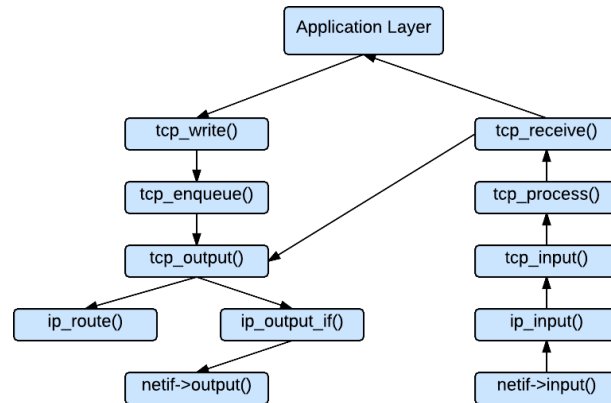


Figure 4.4: Control Flow in TCP Processing [21]

After a connection has been established between the server and a client, data can be transferred in both directions. When the server wants to send data it calls `tcp_write()`. The `tcp_write()` command transfers the control to `tcp_enqueue()` that breaks the data into appropriate sized TCP segments and holds them in a queue of the connection.

Then the function *tcp_output* will check if it is possible to send the data by checking the available space in the receiver window and sends the data using *ip_route()* and *ip_output_if()*.

In the same way if a packet is received from the client, the process begins when *ip_input()* hands over a segment to *tcp_input()*. An initial checksum verification is done and then the segment is processed by *tcp_process()* to find the TCP connection to which the segment belongs. Then a TCP state machine to transfer control between different stages is done. The *tcp_receive()* is called if the connection is in a state to accept data from the network and pass the segment up to the an application program. An ACK for the received packet is sent using the *tcp_output()*.

4.3.3 User Datagram Protocol (UDP)

The implementation of UDP is also made using the Socket API. For a TCP implementation a stream socket was created but an UDP implementation requires datagram socket. The work-flow of a UDP/IP Client-Server is shown in Figure 4.5

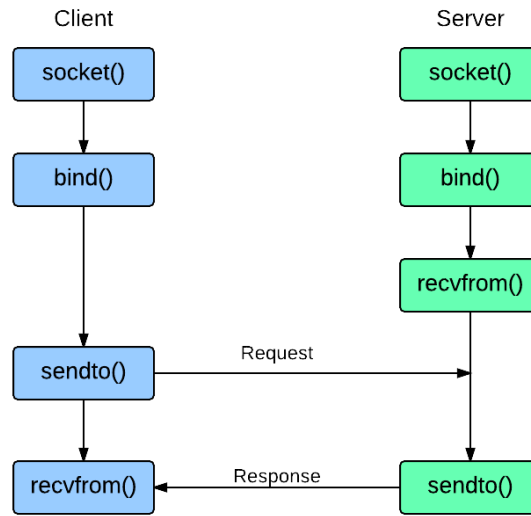


Figure 4.5: UDP/IP Client-Server Work-flow [22]

UDP Client

On the client side a socket is created using *socket()* and bound to a particular IP address and port number using *bind()*. Then the client sends the required data request using *sendto()* which is processed by the server to provide appropriate response that is read

using *recvfrom()*. This implementation of the UDP on the PC side is done with Python script. Time module is used here as well for the response time measurement.

UDP Server

A communication is initialized by the creation of a datagram socket using *socket()* by the server. The server then binds to an IP address and a port number using *bind()*. The server now waits in this state from any data request from a client. The UDP packets are handled using pbuf and the processing of UDP is shown in Figure 4.6 [21].

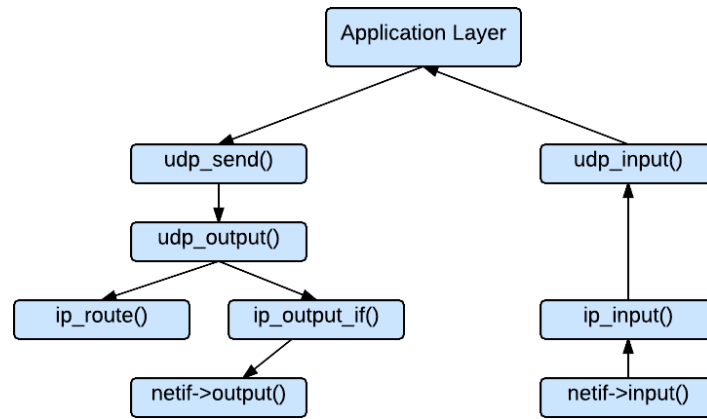


Figure 4.6: Control Flow in UDP Processing [21]

To send data, the application program calls *udp_send()* which in turn calls *udp_output()*. Then the check-summing is done and header fields are filled. Since the UDP packet contains the IP source address of the packet, the function *ip_route()* is called to find the network interface to which the packet is to be transmitted. Then the *ip_output_if()* is used for the actual transmission of the packet. When the UDP datagram arrives, the IP layer calls the *udp_input()* function and checksum verification is done. Finally the *recv()* is called to handle the data contained in the datagram.

4.3.4 Routing Server

We designed an initial routing server C++ using Windows sockets and threads. It was designed to accept all incoming connections and creating a separate thread for each connection. These threads were tied to a thread handler which served as the communication handler between the client applications and target device. A modified point-to-point protocol was used to transmit packages to its targeted destination. We

performed a test with two computers hosting one server each where a client sent a message to be routed to the other server using Ethernet as media. The result was an average Round trip time (RTT) of 0.25 ms which met the requirement. However, an issue arose when the server had multiple clients accessing the same socket with block receive. The result was that only one of the threads received the message and this thread was not necessarily the intended one. So we designed a new server using events to trigger on receive and accept flags. This event-based communication in combination with the broadcasting of all incoming messages allowed multiple clients to communicate with the same socket without erroneous behaviours. The RTT result was equal to that of its predecessor. A diagrammatic representation of the routing server is given in Figure 4.7.

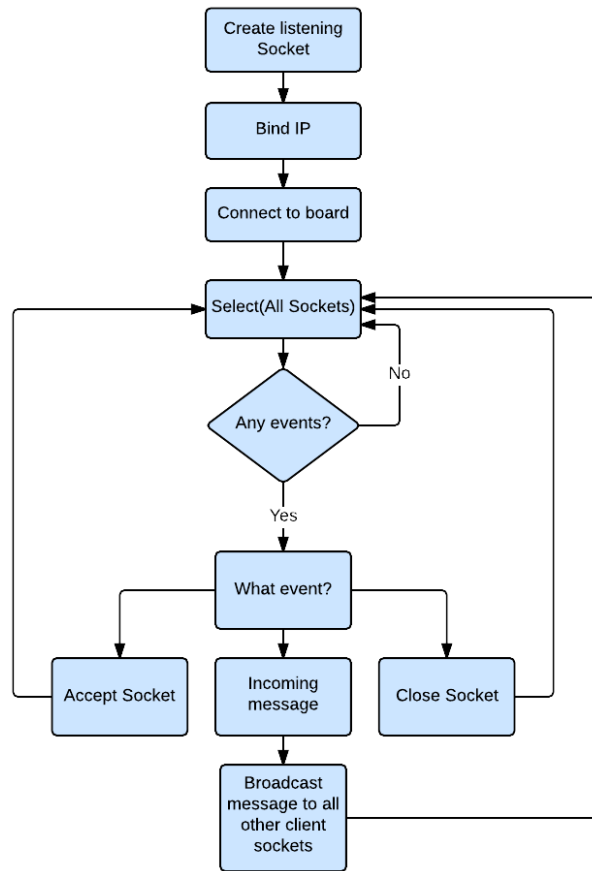


Figure 4.7: Control Flow in Routing Server

The routing server creates a socket that is bound to an IP and listens for any incoming connections from clients. The MCU becomes a fixed client to this server. Then it waits for a connection from other PC applications and broadcasts the message to all the

connections in the network.

4.4 USB Communication

The second way of communicating with the MCU is using USB. The MCU supports both High Speed (HS) and Full Speed (FS) modes of operation. USB is a host initiated communication and a single host supports multiple devices connected to it. The implementation procedure of USB communication is given below.

4.4.1 USB Host

The PC acts as the USB host which is implemented with Python. The PC application is made to communicate with the MCU using the virtual COM port at 115.2 kbps implemented using the serial module. The data is sent using *write()* and read using *read()*. The response time measurement is done using the time module.

4.4.2 USB Device

The USB communication on the MCU is implemented using the STM32Cube USB device library and Hardware Abstraction Layer (HAL) driver. This library contains functions for most common USB device classes like Human Interface Device (HID), Mass Storage Class (MSC), Audio, Communication Device Class (CDC) etc. based on USB device stack that supports all STM32 microcontroller series [23]. The USB device library is generic for all STM32 microcontrollers and is present on top of the HAL driver and offers all the APIs required to develop a USB device application. The drivers support multi packet transferring features which allows for sending big amounts of data without splitting it into max packet size transfers. It also provides configuration files to change the core and the library configuration without changing the library code. The structure of the USB Device library is shown in Figure 4.8 [24].

The application is developed on top of those layers. The USB device library layer contains the core and class drivers. The core consists of full set of APIs to manage the internal USB device library state machine and call back processes from USB interrupts. It also handles the interrupt requests and I/O requests. The USB core uses a control transfer state machine that has four different states namely Default, Addressed, Configured and Suspended.

For the implementation of the communication with PC, CDC class of drivers were used. The CDC core uses two endpoint/transfer types,

- Bulk endpoints for data transfers (1 OUT endpoint and 1 IN endpoint)
- Interrupt endpoints for communication control (CDC requests, 1 IN endpoint)

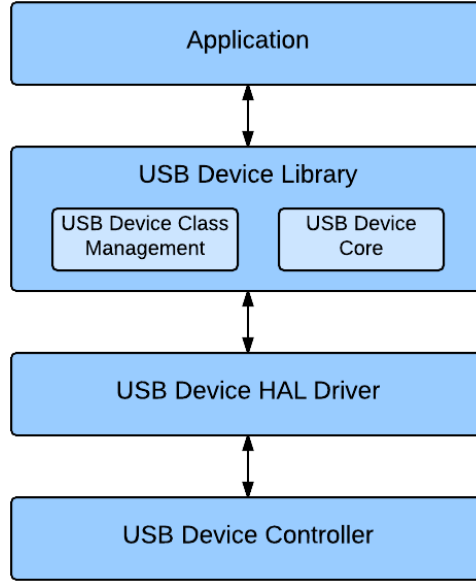


Figure 4.8: A set of library and driver functions that are used to develop user application is present in the form of four layer abstraction [24]

The data transfer from the evaluation board to the PC is managed periodically depending on host request. We use a circular buffer for storing data until the PC requests for the data. The driver calls the lower OUT endpoint and waits until the function is completed before allowing new transfers through the endpoint. The library uses the callback function mechanism for the implementation of the application. The driver structure is mapped to different functions in the application layer that can be used to transfer data.

4.5 CAN Implementation

The STM3240G-Eval evaluation board has two CAN bus channels based on a DB9 connector. It supports the CAN protocol version 2.0 A and B. The CAN controller can support bit rates up to 1 Mbit/s and allows different types of messages like application messages, network management and diagnostic messages. So an advanced filtering mechanism is available. Our application program on the evaluation board uses different registers to request transmissions, handle reception, manage interrupts and configure CAN parameters like baud rate, filter specification etc. It operates in different modes namely Initialization, Sleep, Normal, Loopback, Silent and Silent Loopback [25].

We implemented the interface between the software and hardware for CAN messages

using mailboxes. A mailbox contains all the information related to a message such as identifier, data, control, status and time stamp. In order to transmit a message the application must select an empty mailbox and set up the Identifier, Data Length Code (DLC) and the actual data before requesting for transmission. Then the CAN scheduler schedules the mailbox with highest priority. The priority of the mailboxes is provided based on the identifier. The message with the lowest identifier value has the highest priority according to the arbitration of CAN protocol and if the identifier values are equal the lower mailbox number is scheduled first. The mailbox goes to scheduled state after it is scheduled and becomes empty after the transmission is completed.

The received CAN messages are stored in mailboxes organized as a First In First Out (FIFO) and the application accesses the messages using the mailboxes. A received message is valid when the message is received correctly according to the CAN protocol and passes through a filter for message identifier successfully. Once a message is received completely and stored in the FIFO mailbox, an interrupt request is generated for the application program to process on it.

The configurations to the CAN related registers can be made only in the initialization mode avoiding any disturbance to the communication that may be caused by changing hardware configuration.

4.6 PC-MCU Communication Protocol

An efficient communication protocol is necessary to reduce the response time between the PC and MCU. The protocol includes all possible types of data that could be transferred. A basic protocol with Start, Data and End was available from QRCAN. But we made a few improvements to that to accommodate efficient scheduling and access to multiple applications on PC. The improvements were to add Time stamp, data length and CRC fields to the protocol. The structure of a frame in the communication protocol is shown in Figure 4.9.

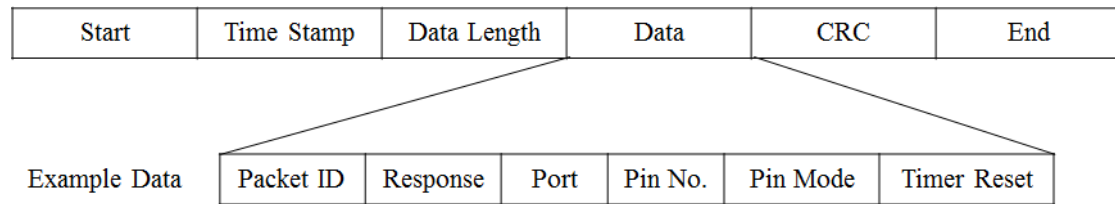


Figure 4.9: Structure of a frame in Communication Protocol between PC and MCU

The description of different fields is given in Table 4.1

Table 4.1: Description of fields in Communication Protocol

Start	Indicates the start of frame (8 bits)
Time Stamp	Relative time delay from the previous message (32 bits)
Data Length	Length of the data (16 bits)
Data	Payload of the frame (depends on Packet ID)
CRC	Cyclic Redundancy Check (32 bits)
End	Indicates the end of frame (8 bits)

Each frame starts with a Start byte that could be a fixed value (Eg. 11111111) followed by a 32-bit time stamp. The Time stamp can have a resolution up to a few microseconds. It is placed in the beginning of the message to provide an opportunity for the scheduler to schedule messages even without knowing the content of the message. The CRC field is provided to check the integrity of the message and the packet ends with a End byte with a fixed value (Eg. 00000000).

The example data field in Figure 4.9 is a Digital IO message from the PC that signals the MCU to set a specific pin. The type of message is specified in Packet ID followed by the Response field that could be used by the MCU to send a response upon successful completion of execution. Then the other fields specify the details of the pin that must be set or reset.

4.7 Scheduler

The way in which messages are handled in the MCU is a crucial factor that determines the efficiency of the protocol. The messages are stored in RAM and a scheduler has to schedule all the messages according to the time stamp sent from the PC. The timestamp is a relative delay from the previous message. We decided to use the relative delay over absolute delay to reduce the load on the MCU for reordering the out of order messages and scheduling them. The relative delay between the messages can be as short as 10 μ s. The delay before execution of different messages in the MCU is obtained by counting the clock pulses operated by a 168 MHz clock, thus once every 5.95 ns.

An implementation of driving the Digital IO pin in the MCU in accordance with the time stamps from the PC is shown in Figure 4.10. The highlighted portion of the image shows the precise delay generated by the MCU.

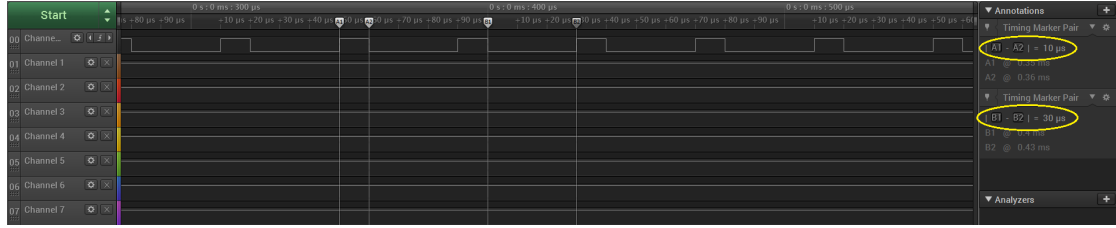


Figure 4.10: Example waveform generated by Digital IO pin. The high pulse is $10\ \mu\text{s}$ long (shown in top circle) and the low pulse is $30\ \mu\text{s}$ long (shown in bottom circle)

4.8 BUSMASTER

BUSMASTER is an open source software tool that runs on Windows OS and helps in monitoring, analyzing and simulating CAN bus network [26]. It has the functionality to analyze the data bytes in raw, physical or logical format. The BUSMASTER has features to display message content, message information, data logging, time stamping, message filters etc. [27]. Further it allows multiple USB-CAN hardware to be connected to it through USB or Ethernet. BUSMASTER uses an American Standard Code for Information and Interchange (ASCII) based communication protocol to transfer information about CAN messages. A window showing the CAN messages displayed in BUSMASTER is shown in Figure 4.11.

Time	Tx/Rx	Channel	Msg Type	ID	Message	DLC	Data Byte(s)
10:25:42:5125	Rx	1	s	0x149	0x149	8	01 01 01 01 01 01 01 01
10:25:42:6125	Rx	1	s	0x149	0x149	8	02 02 02 02 02 02 02 02
10:25:42:7124	Rx	1	s	0x149	0x149	8	03 03 03 03 03 03 03 03
10:25:42:8124	Rx	1	s	0x149	0x149	8	04 04 04 04 04 04 04 04
10:25:42:9124	Rx	1	s	0x149	0x149	8	05 05 05 05 05 05 05 05
10:25:43:0124	Rx	1	s	0x149	0x149	8	06 06 06 06 06 06 06 06
10:25:43:1124	Rx	1	s	0x149	0x149	8	07 07 07 07 07 07 07 07
10:25:43:2124	Rx	1	s	0x149	0x149	8	08 08 08 08 08 08 08 08
10:25:43:3124	Rx	1	s	0x149	0x149	8	09 09 09 09 09 09 09 09
10:25:43:4125	Rx	1	s	0x149	0x149	8	00 00 00 00 00 00 00 00
10:25:43:5124	Rx	1	s	0x149	0x149	8	01 01 01 01 01 01 01 01
10:25:43:6124	Rx	1	s	0x149	0x149	8	02 02 02 02 02 02 02 02
10:25:43:7124	Rx	1	s	0x149	0x149	8	03 03 03 03 03 03 03 03
10:25:43:8124	Rx	1	s	0x149	0x149	8	04 04 04 04 04 04 04 04
10:25:43:9124	Rx	1	s	0x149	0x149	8	05 05 05 05 05 05 05 05
10:25:44:0124	Rx	1	s	0x149	0x149	8	06 06 06 06 06 06 06 06
10:25:44:1124	Rx	1	s	0x149	0x149	8	07 07 07 07 07 07 07 07
10:25:44:2124	Rx	1	s	0x149	0x149	8	08 08 08 08 08 08 08 08
10:25:44:3123	Rx	1	s	0x149	0x149	8	09 09 09 09 09 09 09 09
10:25:44:4124	Rx	1	s	0x149	0x149	8	00 00 00 00 00 00 00 00
10:25:44:5124	Rx	1	s	0x149	0x149	8	01 01 01 01 01 01 01 01
10:25:44:6124	Rx	1	s	0x149	0x149	8	02 02 02 02 02 02 02 02
10:25:44:7124	Rx	1	s	0x149	0x149	8	03 03 03 03 03 03 03 03
10:25:44:8123	Rx	1	s	0x149	0x149	8	04 04 04 04 04 04 04 04
10:25:44:9123	Rx	1	s	0x149	0x149	8	05 05 05 05 05 05 05 05
10:25:45:0123	Rx	1	s	0x149	0x149	8	06 06 06 06 06 06 06 06
10:25:45:1124	Rx	1	s	0x149	0x149	8	07 07 07 07 07 07 07 07
10:25:45:2123	Rx	1	s	0x149	0x149	8	08 08 08 08 08 08 08 08
10:25:45:3123	Rx	1	s	0x149	0x149	8	09 09 09 09 09 09 09 09
10:25:45:4124	Rx	1	s	0x149	0x149	8	00 00 00 00 00 00 00 00
10:25:45:5123	Rx	1	s	0x149	0x149	8	01 01 01 01 01 01 01 01
10:25:45:6123	Rx	1	s	0x149	0x149	8	02 02 02 02 02 02 02 02
10:25:45:7123	Rx	1	s	0x149	0x149	8	03 03 03 03 03 03 03 03
10:25:45:8123	Rx	1	s	0x149	0x149	8	04 04 04 04 04 04 04 04
10:25:45:9123	Rx	1	s	0x149	0x149	8	05 05 05 05 05 05 05 05
10:25:46:0123	Rx	1	s	0x149	0x149	8	06 06 06 06 06 06 06 06
10:25:46:1123	Rx	1	s	0x149	0x149	8	07 07 07 07 07 07 07 07
10:25:46:2123	Rx	1	s	0x149	0x149	8	08 08 08 08 08 08 08 08
10:25:46:3122	Rx	1	s	0x149	0x149	8	09 09 09 09 09 09 09 09
10:25:46:4123	Rx	1	s	0x149	0x149	8	00 00 00 00 00 00 00 00
10:25:46:5123	Rx	1	s	0x149	0x149	8	01 01 01 01 01 01 01 01
10:25:46:6123	Rx	1	s	0x149	0x149	8	02 02 02 02 02 02 02 02
10:25:46:7123	Rx	1	s	0x149	0x149	8	03 03 03 03 03 03 03 03
10:25:46:8123	Rx	1	s	0x149	0x149	8	04 04 04 04 04 04 04 04
10:25:46:9123	Rx	1	s	0x149	0x149	8	05 05 05 05 05 05 05 05
10:25:47:0123	Rx	1	s	0x149	0x149	8	06 06 06 06 06 06 06 06
10:25:47:1123	Rx	1	s	0x149	0x149	8	07 07 07 07 07 07 07 07
10:25:47:2122	Rx	1	s	0x149	0x149	8	08 08 08 08 08 08 08 08
10:25:47:3122	Rx	1	s	0x149	0x149	8	09 09 09 09 09 09 09 09
10:25:47:4123	Rx	1	s	0x149	0x149	8	00 00 00 00 00 00 00 00
10:25:47:5122	Rx	1	s	0x149	0x149	8	01 01 01 01 01 01 01 01

Figure 4.11: CAN Messages displayed in BUSMASTER

From the figure, it can be seen that multiple nodes in different CAN channels can be monitored simultaneously. It is also possible to configure CAN filtering, baud rate etc. from BUSMASTER. We decided to use BUSMASTER mainly to reduce the cost of developing a custom software. We use it to monitor the messages on the CAN network in real time and also to configure different parameters of the CAN network. Further an emulation of CAN node was done to send specific test sequences to the device under test.

Thus the different components of the system was implemented both on the PC and MCU. A performance measurement of the different components are given in the next chapter.

5

Evaluation

This chapter presents an evaluation of the project with respect to the requirements. The main evaluation parameter is the response time and design choices were made according to the response time results. The response time values for different physical media with different software configurations are given in this chapter. The response time measurement was mainly done using the time module of Python and system clock of the PC.

5.1 Ethernet Performance

Ethernet uses UDP/IP or TCP/IP protocol for transferring data. This section shows the performance results with Ethernet for different transport layer protocols. The purpose of the test was to measure the response times with the Ethernet media using TCP and UDP protocol.

Setup

The MCU was programmed as a server which will receive a connection from a Test client application on PC. The application will transfer 10000 test cases and measure response times for each case. The response times are saved to a file.

Results

Table 5.1: Response Times of Ethernet using TCP and UDP

	TCP	UDP
Average Time	0.39 ms	0.41 ms
Shortest Time	0.29 ms	0.25 ms
Longest Time	8.93 ms	14.94 ms

A graph showing the response times of all 10000 messages can be found in Appendix A.

Conclusion

The conclusion drawn from these tests is that the transport layer communication protocol used has little impact on the response time for this application. This is because, TCP consumes more time only if there is a disturbance in the network causing it to resend the packet. Since the test was conducted with just two devices on the network, there was no disturbance in the network resulting in an almost equal response times for the two protocols. Thus the choice of protocol was made solely on implementation complexity. Since this application has important data transferred, TCP is chosen as the main transport layer protocol due to its high reliability.

5.2 Routing Server

The various tests performed to check the performance of the routing server mechanisms is given below.

5.2.1 Threaded Routing Server

A test to check the performance of the threaded routing server was done.

Setup

The setup of using a multi-threaded routing server with TCP connections was tested with a client, routing server and an echo server. The servers were bounded to different IPs after which the client would request a connection to the route server, start a timer and send a message containing the destination address of the echo server. The echo server sends back every incoming message to the socket from which it was received. The

route server then forwards the message back to the client and the client measure time between sending and receiving the message. 100000 messages were sent and RTTs were saved. This was done both internally within a PC and over LAN using Ethernet.

Results

100% of messages accepted with an average RTT of 0.015ms internally within a PC and 0.25ms over LAN (Ethernet).

Conclusion

The measured RTT leaves 0.75ms for application layer protocol handling and extra features. But handling of multiple clients requesting the same socket have proved troublesome.

5.2.2 Event-driven routing server

A test to check if response time requirement was satisfied for the communication interface while allowing easier multiple point-to-point connection was performed.

Setup

The setup for using an event driven routing server with TCP connections was tested with a client, routing server and an echo server. The servers were bounded to separate IPs after which the client would request a connection to the route server, start a timer and send a message which then will be broadcasted to all connected sockets except the sender and the routing server. The echo server sends back every incoming message to the socket from which it was received. The route server broadcasts the message in the same manner and the client measure time between sending and receiving the message. 100000 messages were sent and RTTs were saved. This was done both internally within a PC and over LAN using Ethernet.

Results

100% of messages accepted with an average RTT of 0.015ms internally and 0.25ms over LAN (Ethernet).

Conclusion

The event driven routing server fulfills the time requirement while enabling a better solution for the multiple point-to-point problem.

5.3 USB Performance

The tests that were performed to measure the performance of USB using FS and HS are given below.

Setup

The MCU was programmed as an USB device that will connect to the USB host i.e. PC. The PC transfers data using the virtual serial port available. The PC transfers test cases and saves the response times to a file.

Results

Table 5.2: Response Times of USB using FS and HS

	FS	HS
Average Time	0.99 ms	0.08 ms
Shortest Time	0.15 ms	0.05 ms
Longest Time	8.75 ms	0.64 ms

A graph showing the response times of 10000 messages can be found in Appendix A.

Conclusion

The High Speed protocol is ten times faster than the Fast Speed protocol and meets the response time requirement. Hence the HS was the obvious choice between the two for high performance.

5.4 Response Time Comparison

A comparison between response times of different modes in the physical interface is shown in Figure 5.1. As seen from the figure the USB-FS has the highest response time

close to 1 ms making it impossible to use form this application. But on the other hand, USB-HS is the fastest among the four and can be used for applications that require very high performance. Both TCP and UDP have a have a response time of around 0.4 ms and can be used for applications that are not performance critical.

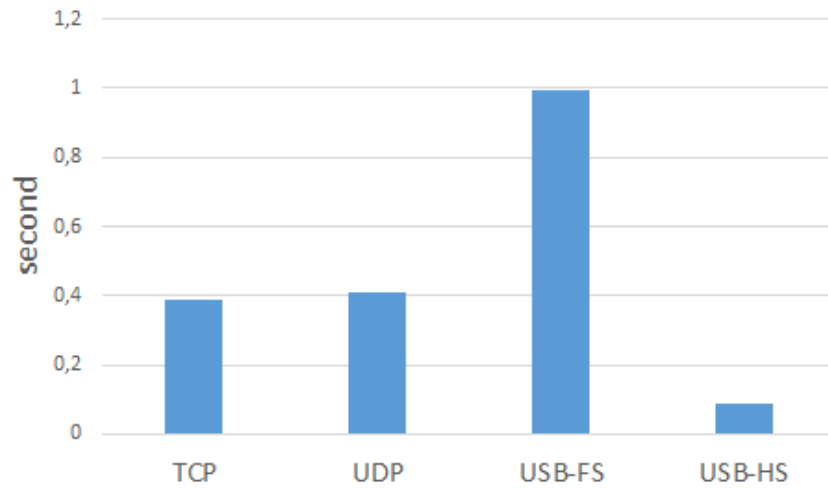


Figure 5.1: Response Time Comparison

6

Discussion and Future Work

6.1 Physical Interface

One of the central concerns throughout the project was directed towards the physical media used between the PC and the MCU. The combination of media and protocols available for each of them had an impact on the performance. There are pros and cons with each medium that led to enabling of both modules depending on the situation. Ethernet provides the possibility of connecting multiple PCs to the network to the device thereby enabling resource sharing. This is a desired feature in such a device. The downside of having only Ethernet is that, modern laptops today have only USB ports. Hence the need to have the USB interface operational to be used locally with a single PC. From the evaluation results, it can be seen that USB HS protocol had the best performance.

Other possible physical media options could be WiFi, RS232, USB 3.0. From a general point of view, Wi-Fi has a lower performance than Ethernet making it a less likely choice. Moreover handling Wi-Fi data on the MCU side increases the complexity of both the hardware and software. As mentioned earlier, RS232 is slower than USB and its physical connector is no longer available in modern PCs. USB 3.0 could be a good option to increase the performance, but it increases the complexity and price of the MCU.

6.2 Routing Server

The route server application works with Ethernet and focuses on forwarding messages with minimal delay while maintaining scalability. The existing design has a basic func-

tionality of continuously accepting connections and forwarding messages. But this server could be improved for logging all data traffic and detecting faults. Further a similar way or routing messages from different applications must be done for USB.

6.3 Scheduler

The current design of scheduler handles messages based on relative delay between messages. The MCU starts execution of messages either after a buffer overflow or after receiving all the messages from the PC which is denoted by an end of message frame. So messages can be transmitted from the PC in bursts. But the relative delay mechanism has a few disadvantages. It is difficult to maintain relative delay to previous message if multiple applications on the PC try to send messages at the same time.

So a solution for allowing multiple applications to use time stamp could be to design the scheduler based on absolute delay. This approach would require the clock of both the PC and MCU to be synchronous. But the disadvantage with this approach is that, more number of bits are required to send absolute time from PC making the frame size in the communication protocol bigger. There are two approaches for the synchronization of clocks. One way is to synchronize the clock without accounting for the time it takes to transfer the reset-clock instruction to the receiving device. So the initiating device will always experience a delay when logging messages. The advantage of choosing this design is that the receiving device will on average have the same time that the sending device had when setting time stamp, thus the processing and transferring time is already accounted for and needs no further consideration when setting the time stamp. The other approach is to account for transfer time in the calibration stage and thus provides more accurate logging information since both devices are operating closer to each others clocks. Since messages are now stored in a priority array and executed whenever their time stamp is met, each message has to consider the processing and transferring time between the PC to the MCU.

6.4 BUSMASTER Plug-in

The current design of this plug-in is a basic version and misses CAN data frames at maximum baud rate. It is also good to have full control of the way in which data is handled in BUSMASTER. So a solution to improve the performance would be to develop a custom plug-in that suits the performance of the MCU. Some initial research has been done for the implementation and it was found that a separate project can be created for the plug-in and integrated with the existing main BUSMASTER repository.

7

Conclusion

This report summarizes the implementation of an interface between PC and CAN network based on an ARM microcontroller using USB or Ethernet as the physical interface. An open source software supports the hardware and acts as an user interface from the PC. The current version of the interface is fully functional, but a few improvements can be done to make it suitable for complex applications. Future work considers the custom-made BUSMASTER plug-in to enable traffic logging in real-time and verifying application testing features with timestamped message priority. The protocol to establish this communication has to be implemented on both ends. The designed scheduler using timestamped priority can be implemented to replace the current relative delay scheduler. A routing server mechanism can be implemented for USB to support multiple applications on PC.

Bibliography

- [1] Vehicle bus. [Accessed: 15 February 2015]. [Online]. Available: http://en.wikipedia.org/wiki/Vehicle_bus
- [2] Robert Bosch GmbH, “CAN Specification 2.0,” 1991.
- [3] OSI Reference Model. [Accessed: 04 May 2015]. [Online]. Available: <http://www.windowsnetworking.com/articles-tutorials/common/OSI-Reference-Model-Layer1-hardware.html>
- [4] Texas Instruments, Introduction to the Controller Area Network (CAN). [Accessed: 22 January 2015]. [Online]. Available: <http://www.ti.com/lit/an/sloa101a/sloa101a.pdf>
- [5] Fujitsu Ltd., Ethernet Tutorial. [Accessed: 19 March 2015]. [Online]. Available: www.fujitsu.com/downloads/TEL/fnc/.../ethernet-prerequisite.pdf
- [6] TCP Definition. [Accessed: 15 April 2015]. [Online]. Available: <http://www.linfo.org/tcp.html>
- [7] USB in a NutShell. [Accessed: 11 March 2015]. [Online]. Available: <http://www.beyondlogic.org/usbnutshell/usb1.shtml>
- [8] USB made Simple. [Accessed: 17 March 2015]. [Online]. Available: <http://www.usbmadesimple.co.uk/index.html>
- [9] RS-232 Data Interface - A Tutorial on Data Interface and Cables. [Accessed: 07 April 2015]. [Online]. Available: <http://www.arcelect.com/rs232.htm>
- [10] X. Wang and W. Guo, “The Design of RS232 and CAN Protocol Converter based on PIC MCU,” *Computer and Information Science, CCSE*, vol. 2, no. 3, pp. 176–181, 2009.

- [11] F. Luo and C. Liu, "Implementation of CAN Bus Real-Time Simulation Kernel Based on Windows Platform," *Advances in Computer, Communication, Control & Automation, LNEE*, vol. 121, pp. 61–68, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25541-0_9
- [12] G. S. Kumar, "Designing and Development of a CAN Bus Analyzer for Industrial Applications Using ARM and PIC," *International Journal of Computer Science and Information Technologies (IJCSIT)*, vol. Vol. 3, no. 2, pp. 3749–3753, 2012.
- [13] F. Nedeoglo, D. Komissarov, O. Novozhilov, and A. Chepurnov, "Simple CAN- Bus adapter for Accelerator control running under Linux and RTLinux ," *3rd International Workshop on Personal Computer and Particle Accelerator Controls*, Oct 2000.
- [14] H. Kashif, G. Bahig, and S. Hammad, "CAN bus analyzer and emulator," *International Design and Test Workshop, IEEE*, pp. 1–4, Nov 2009.
- [15] M. Mostafa, M. Shalan, and S. Hammad, "FPGA-Based Low-level CAN Protocol Testing," *The 6th International Workshop on System on Chip for Real Time Applications, IEEE*, pp. 185–188, Dec 2006.
- [16] J. Novak, A. Fried, and M. Vacek, "CAN generator and Error Injector," *9th International Conference on Electronics, Circuits and Systems*, vol. 3, pp. 967–970 vol.3, 2002.
- [17] J. Novak and P. Kocourek, "Automated Testing of Electronic Control Units Compatibility in Vehicle CAN Networks," *Proceedings of the IEEE International Symposium on Industrial Electronics, ISIE*, vol. 4, pp. 1423–1428, June 2005.
- [18] CANCaseXL. [Accessed: 16 March 2015]. [Online]. Available: https://vector.com/vi_cancase_xl_log_en.html
- [19] PEAK-System. [Accessed: 6 February 2015]. [Online]. Available: <http://www.peak-system.com/>
- [20] QRTech AB. [Accessed: 29 January 2015]. [Online]. Available: <http://www.qrttech.se/>
- [21] A. Dunkels, "Design and Implementation of the LwIP TCP/IP Stack," *Swedish Institute of Computer Science*, p. 46, February 2001.
- [22] TCP Tutorial. [Accessed: 14 April 2015]. [Online]. Available: <http://vichargrave.com/network-programming-design-patterns-in-c/>
- [23] STM32CubeMX. [Accessed: 04 March 2015]. [Online]. Available: <http://www.st.com/stm32cube>
- [24] *UM1734 STM32Cube USB Device Library User Manual*, STMicroelectronics.

- [25] *RM0090 Reference Manual*, STMicroelectronics.
- [26] BUSMASTER. [Accessed: 6 February 2015]. [Online]. Available: <http://rbeietas.github.io/busmaster/>
- [27] *BUSMASTER User Manual*, Robert Bosch & Business Solutions Limited.

Appendices

A

Response Times Plots

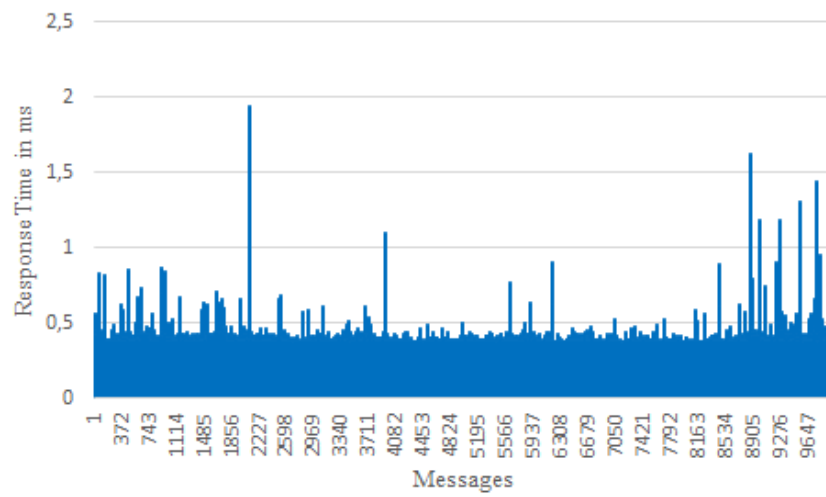


Figure A.1: Response times of 10000 messages using TCP/IP

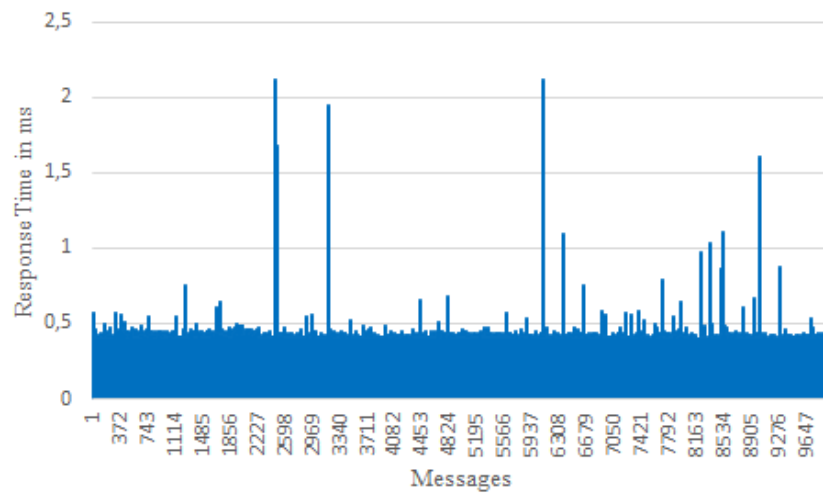


Figure A.2: Response times of 10000 messages using UDP/IP

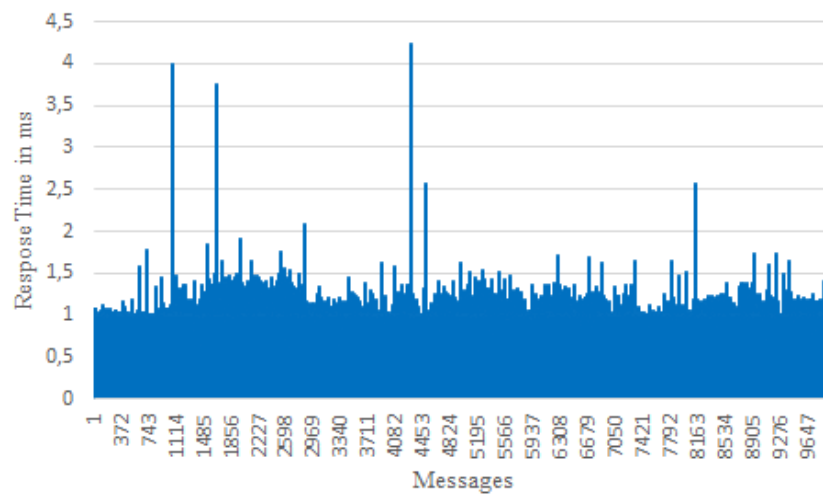


Figure A.3: Response times of 10000 messages using USB-FS

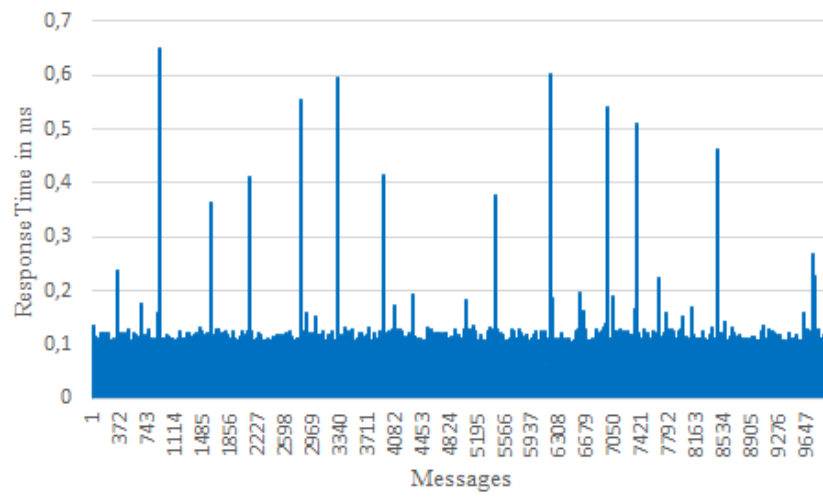


Figure A.4: Response times of 10000 messages using USB-HS