

C^3 : CONFIGURABLE CAN FD CONTROLLER: DESIGN, IMPLEMENTATION
AND EVALUATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MEHMET ERTUĞ AFŞİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2018

Approval of the thesis:

**C^3 : CONFIGURABLE CAN FD CONTROLLER: DESIGN, IMPLEMENTATION
AND EVALUATION**

submitted by **MEHMET ERTUĞ AFSİN** in partial fulfillment of the requirements
for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Tolga Çiloğlu
Head of Department, **Electrical and Electronics Engineering** _____

Prof. Dr. Şenan Ece Güran Schmidt
Supervisor, **Electrical and Electronics Eng. Dept., METU** _____

Assoc. Prof. Dr. Klaus Werner Schmidt
Co-supervisor, **Electrical and Electronics Eng. Dept., METU** _____

Examining Committee Members:

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering Department, METU _____

Prof. Dr. Şenan Ece Güran Schmidt
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Department, METU _____

Prof. Dr. İlkey Ulusoy
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. Orhan Gazi
Electronic and Communication Eng. Dept., Çankaya University _____

Date:

February 7, 2018

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: MEHMET ERTUĞ AFŞİN

Signature :

ABSTRACT

C^3 : CONFIGURABLE CAN FD CONTROLLER: DESIGN, IMPLEMENTATION AND EVALUATION

AFŞİN, Mehmet Ertuğ

M.S., Department of Electrical and Electronics Engineering

Supervisor : Prof. Dr. Şenan Ece Güran Schmidt

Co-Supervisor : Assoc. Prof. Dr. Klaus Werner Schmidt

February 2018, 120 pages

CAN FD (Controller Area Network with Flexible Data Rate) is a new communication standard, compatible with CAN. Different from CAN, CAN FD switches to high data rate during data transmission and allows payloads up to 64 bytes. In this thesis, we propose C^3 : Configurable CAN FD Controller which features up to fully configurable 96 TX and 96 RX buffers organized as mailboxes. Each RX buffer has dedicated acceptance filters. The host MCU sees C^3 as a memory mapped device and interfaces with it via SPI protocol which is designed and developed in the scope of this thesis. Different from existing CAN FD Controllers, C^3 provides run time configurable number of buffers and individual buffer sizes which makes it best use of a single hardware for every application. Furthermore, it provides efficient and flexible usage of a limited embedded memory. C^3 is implemented on a Xilinx Virtex 5 FPGA demo board as an IP Core and its functions are verified at 2 Mbps and the response time measurements are performed to evaluate the timing performance.

Keywords: CAN, CAN FD , CAN FD Controller, FPGA, Buffer Organization

ÖZ

C^3 : AYARLANABİLİR CAN FD KONTROLCÜSÜ: TASARIM, GERÇEKLEŞTİRİM VE DEĞERLENDİRME

AFŞİN, Mehmet Ertuğ

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi : Prof. Dr. Şenan Ece Güran Schmidt

Ortak Tez Yöneticisi : Doç. Dr. Klaus Werner Schmidt

Şubat 2018 , 120 sayfa

CAN FD, CAN ile uyumlu yeni bir haberleşme protokolüdür. CAN protokolünden farklı olarak, CAN FD veri gönderim fazında yüksek hıza çıkmakta olup, 64 bayta kadar faydalı yük taşıyabilmektedir. Bu tez kapsamında posta kutusu formatında ayarlanabilir boyutlarda 96 adede kadar gönderici ve 96 adede kadar alıcı ara belleğine sahip C^3 : Ayarlanabilir CAN FD kontrolcüsü sunulmaktadır. Her bir alıcı arabelleği için mesaj tanımlama filtresi bulunmaktadır. Kullanıcı mikro denetleyici, C^3 'ü hafıza haritalı bir cihaz olarak görmektedir ve SPI haberleşme protokolü ile C^3 'e erişmektedir. Bu tez kapsamında mikro denetleyici ve C^3 arasındaki SPI protokolü de tasarlanmıştır. Mevcut CAN FD kontrolcülerinden farklı olarak C^3 uygulama zamanında ayarlanabilir sayıda arabellek ve ayarlanabilir boyutlarda arabellek yapısı sunmaktadır. Bu esneklik sayesinde tek bir kontrolcü tüm uygulamalar için kullanılabilir. Ayrıca kısıtlı olan gömülü hafızadan en verimli şekilde faydalanılmaktadır. C^3 , Xilinx firmasının Virtex 5 FPGA geliştirme kartında IP çekirdeği olarak gerçekleştirilmiştir. C^3 'ün tüm özellikleri 2 Mbps hızda doğrulanmış olup tepki süreleri de ölçülerek zamanlama performansı değerlendirilmiştir.

Anahtar Kelimeler: CAN, CAN FD, CAN FD Kontrolcüsü, FPGA, Ara Bellek Organizasyonu

To My Family

ACKNOWLEDGMENTS

I would like to express my great appreciations to my supervisor Prof. Dr. Şenan Ece Güran Schmidt and co-supervisor Assoc. Prof. Dr. Klaus Werner Schmidt for their support and guidance throughout this thesis. I am thankful for their guidance, which was very helpful in my research and writing of the thesis.

I wish to thank ASELSAN A.Ş. for giving me the opportunity of continuing my education and providing financial support for my conference attendances. I wish to thank my colleagues and seniors in the hardware design department. This thesis work was supported by the Middle East Technical University as a Scientific Research Project with contract number of BAP-03-01-2017-002. I would like to thank Empa Electronics for their donated transceiver components.

I would like to express my special appreciation to Onur Aktop for his contributions to improve my engineering skills. I would like to express special thanks to my family. Finally, I would like to express my appreciation to my friend, Mehmet Ufuk Büyükaşahin. During thesis work, he helped me in every matter and he didn't leave me alone.

TABLE OF CONTENTS

| | |
|--|-------|
| ABSTRACT | v |
| ÖZ | vii |
| ACKNOWLEDGMENTS | x |
| TABLE OF CONTENTS | xi |
| LIST OF TABLES | xiii |
| LIST OF FIGURES | xiv |
| LIST OF ABBREVIATIONS | xviii |
| CHAPTERS | |
| 1 INTRODUCTION | 1 |
| 2 IN-VEHICLE NETWORKS | 5 |
| 2.1 CAN | 6 |
| 2.2 CAN FD | 8 |
| 2.3 CAN/CAN FD Controllers | 12 |
| 3 PREVIOUS WORK ON CAN/CAN FD CONTROLLERS | 15 |
| 4 C ³ (CONFIGURABLE CAN FD CONTROLLER) ARCHITECTURE | 23 |
| 4.1 Hardware Blocks: Memory Mapped Register Block | 25 |
| 4.2 Hardware Blocks: SPI Protocol Control Block | 31 |
| 4.3 Hardware Blocks: Interrupt Control Block | 35 |

| | | |
|-----|---|-----|
| 4.4 | Hardware Blocks: Transmitter Module | 36 |
| 4.5 | Hardware Blocks: Receiver Module | 42 |
| 4.6 | Configuration Phase | 46 |
| 4.7 | Data Phase and Timing | 47 |
| 4.8 | FPGA Implementation Results | 52 |
| 5 | EVALUATION OF C ³ (CONFIGURABLE CAN FD CONTROLLER) | 61 |
| 5.1 | Development and Test Environment | 61 |
| 5.2 | Host Simulator Implementation | 66 |
| 5.3 | Transmit Buffer Configuration and Transmission Tests | 74 |
| 5.4 | Receive Buffer Configuration and Reception Tests | 80 |
| 5.5 | Response Time Measurements | 85 |
| 5.6 | Interrupt and Error Tests | 95 |
| 5.7 | Arbitration and Other Tests | 101 |
| 6 | CONCLUSION | 113 |
| | REFERENCES | 117 |

LIST OF TABLES

TABLES

| | | |
|-----------|---|----|
| Table 3.1 | CAN FD Controllers Comparison 1 | 19 |
| Table 3.2 | CAN FD Controllers Comparison 2 | 20 |
| Table 3.3 | CAN FD Controllers Comparison 3 | 21 |
| Table 4.1 | Configuration Registers | 29 |
| Table 4.2 | SPI Commands and Responses | 34 |
| Table 4.3 | Interrupt Register Set | 37 |
| Table 4.4 | TX Control Register Set | 39 |
| Table 4.5 | RX Control Register Set | 44 |
| Table 4.6 | TX(ID) Register Set | 48 |
| Table 4.7 | RX(ID) Register Set | 49 |
| Table 4.8 | FPGA Device Utilization Summary | 53 |
| Table 4.9 | FPGA Project Status | 54 |
| Table 5.1 | Xilinx Virtex 5 FPGA Familiy Comparison | 63 |

LIST OF FIGURES

FIGURES

| | | |
|-------------|---|----|
| Figure 2.1 | CAN Base Frame | 7 |
| Figure 2.2 | CAN FD Frames | 10 |
| Figure 4.1 | C^3 Architecture | 23 |
| Figure 4.2 | Top Level Architecture | 24 |
| Figure 4.3 | Memory Mapped Register Block (MMR) | 26 |
| Figure 4.4 | Example Memory organizations: (a) 88 buffers (b) 44 buffers . . . | 30 |
| Figure 4.5 | SPI Protocol Control Block | 31 |
| Figure 4.6 | SPI Timing Diagrams | 33 |
| Figure 4.7 | Interrupt Control Block | 35 |
| Figure 4.8 | TX Control Logic Block (TXCL) | 38 |
| Figure 4.9 | Binary Search Algorithm | 40 |
| Figure 4.10 | CAN FD Transmitter Block (CANFDTX) | 41 |
| Figure 4.11 | RX Control Logic Block (RXCL) | 42 |
| Figure 4.12 | CAN FD Receiver-RX Block (CANFDRX) | 45 |
| Figure 4.13 | Message Filter Block (MF) | 46 |
| Figure 4.14 | FPGA Project Hierarchy | 55 |

| | |
|---|----|
| Figure 4.15 FPGA Implementation Options 1 | 56 |
| Figure 4.16 FPGA Implementation Options 2 | 57 |
| Figure 4.17 FPGA Implementation Options 3 | 57 |
| Figure 4.18 FPGA Implementation Options 4 | 58 |
| Figure 4.19 FPGA Implementation Options 5 | 58 |
| Figure 5.1 Test Setup Block Diagram | 61 |
| Figure 5.2 Test Setup | 62 |
| Figure 5.3 Xilinx ML507 Demoboard | 63 |
| Figure 5.4 CAN FD Transceiver Internal[31] | 64 |
| Figure 5.5 Transceiver Signals | 65 |
| Figure 5.6 CAN FD Transceiver Board | 65 |
| Figure 5.7 PCAN USB FD Hardware | 66 |
| Figure 5.8 Xilinx Platform Cable | 67 |
| Figure 5.9 FPGA Implementation Block Diagram | 67 |
| Figure 5.10 Host Simulator Block Diagram | 69 |
| Figure 5.11 SPI Read Operation | 70 |
| Figure 5.12 SPI Burst Read Operation | 71 |
| Figure 5.13 Initialization Block Ram Data Content for 32 Bit SPI Write Com- mand | 71 |
| Figure 5.14 Initialization Block Ram Data Content for 32 Bit SPI Read Command | 72 |
| Figure 5.15 Initialization Block Ram Data Content for Burst SPI Write Command | 73 |
| Figure 5.16 Initialization Block Ram Data Content for Burst SPI Read Command | 74 |

| | |
|--|----|
| Figure 5.17 First 16 Buffers Configuration | 77 |
| Figure 5.18 First 16 Buffers Transmission Data Content and Request | 78 |
| Figure 5.19 PCAN Data Logging for First 16 Transmissions | 79 |
| Figure 5.20 Transmission Requests for Priority Test | 80 |
| Figure 5.21 PCAN Message Reception for Transmission Priority Test | 81 |
| Figure 5.22 Message Content for Buffer 27 | 84 |
| Figure 5.23 Buffer 27 Read Operations 1 | 86 |
| Figure 5.24 Buffer 27 Read Operations 2 | 87 |
| Figure 5.25 TX Response Time Measurements 1 | 89 |
| Figure 5.26 TX Response Time Measurements 2 | 90 |
| Figure 5.27 FPGA Core Delay | 92 |
| Figure 5.28 ISR Read Delay | 93 |
| Figure 5.29 3x RSR Read Delay | 93 |
| Figure 5.30 Burst SPI Read Delay | 93 |
| Figure 5.31 RSR Clear Delay | 94 |
| Figure 5.32 ISR Clear Delay | 94 |
| Figure 5.33 Total Receive Response Time | 94 |
| Figure 5.34 Interrupt Generation | 95 |
| Figure 5.35 ISR Read Operation | 96 |
| Figure 5.36 PCAN Listen Only Mode Settings 1 | 97 |
| Figure 5.37 PCAN Listen Only Mode Settings 2 | 97 |
| Figure 5.38 TX Ack Error Register Content | 98 |

| | |
|--|-----|
| Figure 5.39 PCAN Error Generator | 99 |
| Figure 5.40 Bit Error Register Content | 99 |
| Figure 5.41 PCAN Error Log | 100 |
| Figure 5.42 RX Stuff Error Register Content | 100 |
| Figure 5.43 PCAN Error Log | 101 |
| Figure 5.44 RX CRC Error Register Content | 102 |
| Figure 5.45 Form Error Register Content | 102 |
| Figure 5.46 PCAN Form Error Log | 103 |
| Figure 5.47 Arbitration Test Setup Block Diagram | 104 |
| Figure 5.48 Arbitration Loss | 106 |
| Figure 5.49 Message Reception Orders | 107 |
| Figure 5.50 Arbitration Setup | 108 |
| Figure 5.51 Overflow Register Content | 109 |
| Figure 5.52 Overflow ISR Read Operation | 110 |

LIST OF ABBREVIATIONS

| | |
|---------|---------------------------------------|
| SOF | Start of Frame |
| RTR | Remote Transmission Request |
| IDE | Identifier Extension |
| DLC | Data Length Code |
| ACK | Acknowledgement |
| EOF | End of Frame |
| EDL | Extended Data Length |
| BRS | Bit Rate Switch |
| ESI | Error State Indicator |
| IFS | Inter Frame Spacing |
| CAN | Controller Area Network |
| CAN FD | Controller Area Network Flexible Data |
| IP Core | Intellectual Property Core |
| VHDL | VHSIC Hardware Description Language |
| ID | Identification |

CHAPTER 1

INTRODUCTION

CAN (Controller Area Networks) is the most widely used communication standard in vehicle networks. ECUs (Electronic Control Units) in the vehicles exchange signals in the form of messages in a CAN network [35]. Due to CAN Bus network topology, multiple ECUs are connected to each other on a single bus. The CAN Bus transmission rates are up to 1 Mbps. However, the practical data rate is much lower than 1 Mbps. The data rate is limited by the arbitration and acknowledgment mechanisms. During arbitration, for the simultaneous transmissions, each signal transmitted by the nodes must reach to each other on time such that bit overwrite mechanism works. During acknowledgment phase, the transmitter's signal must propagate to the the receiving nodes and the nodes' acknowledgment responses must reach to the transmitter in time.

The number of ECUs in vehicles, the number of messages hence the amount of information carried on CAN BUS increase significantly day by day. CAN BUS is becoming slow and inadequate. However due to wide usage of CAN BUS and automotive industry's exacting reliability requirements, it is not easy to start using a completely new communication protocol. For these reasons, Bosch came up with CAN FD (CAN with Flexible Data Rate) protocol [28] in 2011. On the one hand, CAN FD preserves the physical layer of CAN which determines the bus arbitration signaling. On the other hand, CAN FD increases the data rate by simply switching to a high transmission rate of up to 10 Mbps after the arbitration is over. Furthermore it enables transmitting longer payloads of up to 64 Bytes instead of the 8 Byte CAN payload.

Since CAN FD is a new protocol, ECU manufacturers' plan to integrate CAN FD to

their design is at initial stage. The CAN FD Controllers available on the market are implemented as IP Cores [6, 1, 9, 18, 5, 10, 4, 8, 20, 7] and being developed since 2015. These IP cores are commercially sold and not available for academic evaluation and analysis.

The worst-case response times (WCRTs) of messages on CAN/CAN FD are computed assuming that the messages are stored in infinite priority queues. However, in practice it is possible that higher priority messages that are released by the application may get blocked due to the non-availability of transmission buffers in a CAN controller [30]. Hence, the timing and the scheduling of CAN/CAN FD messages improve if the transmit and the receive buffer numbers and sizes are compatible with the messages. On the one hand, the number of messages, their sizes and their priorities depend on the applications that run on the ECUs. The variation in the message sizes is particularly significant for CAN FD with a maximum data size of 64 Bytes. On the other hand, the CAN controllers are embedded devices with hardware resource constraints which do not allow implementing the buffer configurations for all possible message sets.

The CAN FD IP cores currently present in the literature either offer completely fixed size buffer arrangement without configuration capability or allow limited buffer configuration with some constraints. Therefore, it is not possible to have a single buffer configuration which fits to every application. Different buffer configurations for different message sets lead to inefficient memory usage for the controllers without memory configuration capability.

In this thesis, we present a novel CAN FD Controller denoted as C^3 (Configurable CAN FD Controller). It supports non-ISO CAN FD protocol specification. It is implemented as an FPGA IP Core. The interface between the controller and the host MCU is widely used communication protocol called SPI (Serial Peripheral Interface).

Different from existing CAN/CAN FD controllers, C^3 enables the configuration of the transmit and receive buffers via SPI during run time before the main applications on the ECU start to run. Such configuration capability enables appropriate buffer configurations on each ECU in the vehicle according to the messages of the applications running that ECU. Furthermore, it makes it possible to reconfigure the buffers if new

applications with new messages are added to the vehicle. The standard SPI interface of C^3 enables any micro controller to run the applications and use C^3 without any specific interface requirements.

C^3 features up to 96 transmit and 96 receive buffers which can be configured via SPI during the configuration phase before the main ECU application begins to run. The buffers are organized as mailboxes. This configuration capability ensures that controller's memory is efficiently allocated for different message sets used for any CAN network present in vehicles. Moreover, it is quite easy to add new messages to the existing message set in a CAN network by reconfiguring the controller's buffer allocation. The mailboxes are allocated an average payload size of 16 bytes instead of maximum payload size of 64 bytes. Because it is very unlikely that all the messages in a message set of a network application will require 64 bytes payload size. This yields 96x16 bytes memory consumption instead of 96x64 bytes hence less memory usage and more memory utilization. The usage of SPI interface makes the integration of the controller to the host MCUs easy since SPI is present in almost every MCU on the market including the low cost ones.

The first main contribution of this thesis is the detailed design of C^3 . The design is described in detail with functional blocks including the SPI block where a custom communication protocol is designed over SPI. This custom protocol defines the communication between the controller and the MCU. The signaling between the blocks and how they exchange and process data are explained. The second main contribution of this thesis is the transmit and receive response time analysis. The delay components contributing the response time are measured individually and overall response time is evaluated. C^3 is implemented on Xilinx Virtex 5 ML507 demo board and the commercial CAN FD transceivers are used for CAN FD physical layer implementation. The design is verified by performing functional tests by sending and receiving CAN FD messages at 2 Mbps using a professional CAN FD analyzer hardware tool.

The thesis organization is as follows: CAN, CAN FD and CAN/CAN FD Controller basics are covered in Chapter.2. Existing CAN/CAN FD Controllers are discussed in Chapter 3. C^3 hardware blocks are presented with details including the SPI communication protocol in Chapter 4. This chapter explains FPGA Implementation details

and the challenges faced. Furthermore, data phase, configuration phase and timing analysis are covered. The experimental setup, hardware components used, testing methods, functional verifications and the response time performance measurements of C^3 are provided in Chapter 5. Finally, this thesis is concluded and future work is outlined in Chapter 6.

CHAPTER 2

IN-VEHICLE NETWORKS

The contemporary vehicles contain a large number of electronic units which are actuators, sensors and microprocessor based Electronic Control Units (ECU)s. These electronic components and the software running on the microprocessors enable implementing complex functions that improve the safety, comfort and the efficiency of the vehicle. To this end, electronic systems assist the driver to control the vehicle with the functionality related to the steering, traction (i.e., control of the driving torque) or braking including the anti lock braking system (ABS), electronic stability program (ESP), electric power steering (EPS), active suspensions, or engine control. Furthermore employing the vehicle electronics enable the near future technologies such as autonomous driving.

The ECUs run applications which rely on the information coming from the other electronic components in the vehicle. Great majority of these applications are real-time and require deadlines. In other words, in-vehicle networks connect the electronic components together and provide the communication between them by meeting the timing requirements.

The benefits of in vehicle networking can be listed as below:

- The size of the cabling is dramatically reduced and the number of input/output pins required at ECUs is less since the point to point connection is replaced by a bus structure
- Common information such as temperature sensors or speed data is shared with every ECU on the bus.

- It is easy to apply a modification by a software change without changing the hardware. This gives great flexibility to the system designer. For a point to point connections without any networking topology, any added data communication function would introduce a new point to point connections and more input and output pins.

The most frequently used in-vehicle network standard is Controller Area Network (CAN). The focus of this thesis is CAN FD (CAN with Flexible Data Rate) network which is based on CAN. Further in-vehicle network standards include FlexRay, MOST and Ethernet [35].

2.1 CAN

CAN (Controller Area Network) is a serial asynchronous bus network. It connects electronic devices, various sensors and actuators in a system or sub-system for various applications. It is multi master communication protocol. The protocol is developed by Robert Bosch GmbH in 1986. It was designed for automotive applications requiring a reliable communication. The data rate is up to 1 Mbps. Apart from being used in automotive industry, CAN is also used in embedded applications and industrial control systems [16]. CAN (Controller Area Network) is a serial asynchronous bus network. It connects electronic devices, various sensors and actuators in a system or sub-system for various applications. It is multi master communication protocol. The protocol is developed by Robert Bosch GmbH in 1986. It was designed for automotive applications requiring a reliable communication. The data rate is up to 1 Mbps. Apart from being used in automotive industry, CAN is also used in embedded applications and industrial control systems [16].

The CAN protocol defines the physical layer and data link layer specifications. It defines how the frames are formed and how the arbitration mechanism works. CAN is basically an event triggered protocol, there is no time slot mechanism where the messages are supposed to be transmitted, instead, they are transmitted whenever the bus is idle. If two nodes begin transmission at the same time, the one with the ID having more priority takes over the bus and the other node stops transmitting and

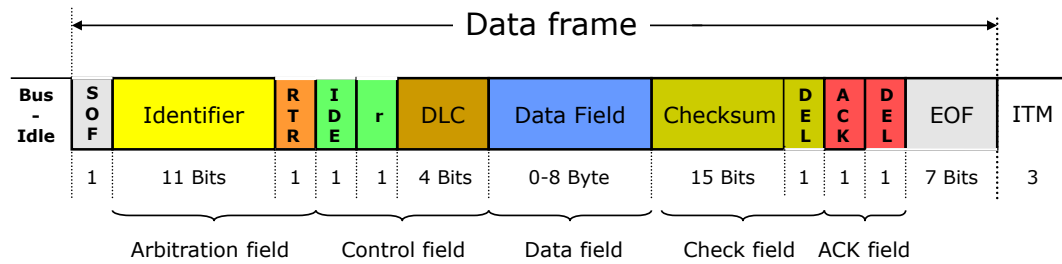


Figure 2.1: CAN Base Frame

attempts to transmit its frame again when the current transmission is over. The priority is determined by the bit values, 0 is a dominant bit whereas 1 is a recessive bit. 0 overwrites 1 when they are transmitted simultaneously. To this end, the message with the smallest ID has the highest priority and always takes over the bus for simultaneous message transmissions [22].

There are two types of IDs, which are base and extended IDs. Base ID data frame is a CAN bus frame with 11 bit ID representation. On the other hand, extended ID data frame is a CAN bus frame with 29 bit ID representation. CAN Bus data frame for Base ID can be seen in Fig.2.1.

Error detection and management is very important for in vehicle networking. There is an acknowledgment mechanism in a CAN frame. Cyclic Redundancy Check (CRC) is performed at the end of the frame. CRC is computed beginning from the start bit of the frame both by the transmitter and the receiver nodes. The CRC computation result calculated by the receiving node must be the same as the one transmitter node places in the frame for a successful communication, otherwise error condition occurs. The nodes which receive a sent frame send a dominant acknowledgment bit to indicate that the frame is received successfully after CRC operation. Bit stuffing provides an additional bit when 5 consecutive same bits are transmitted. Additional bit value is the complement of the value of the previous 5 bits. Bit stuffing is used to ensure that bus does not stay at the same voltage level for a long time and node clocks can synchronize to the bus to have accurate bit sampling point. Lastly, bus monitoring is performed for any bit errors by the transmitter node to check if the transmitted bit value is really on the bus. With all these mechanisms, it is easy to detect any errors and have a very reliable and safe communication with CAN Bus.

The worst case response time (WCRT) of a message is the maximum time between the message is generated at the transmitter node application and it is received at the receiver node application. WCRT depends on the priority levels of the other messages, hence, the messages with higher priority might make a given message m stay in the buffer for a long time. [26, 38, 37] explains the analytical ways to compute WCRT of the messages for transmission. These methods basically assign IDs such that the message is received at its destination before the deadline. In other words, by assigning IDs to the messages, they are given appropriate priority such that WCRT of the messages are smaller than the deadline they are obligated to meet. It is also possible that such WCRT analysis is not necessary for a lightly loaded network.

The data rate for a CAN bus is limited by the arbitration mechanism. During the arbitration phase, the bits transmitted by a node must be received by the other nodes in a single bit time. Therefore, during simultaneous transmissions, it is important that bit overwrite is detected on time such that a node losing arbitration stops transmission. Although the specified maximum bit rate is 1 Mbps, the practical rates are up to 500 kbps. 125 kbps is the mostly used bit rate. The payload of a CAN node is up to 8 bytes. When the other parts of a CAN FD frame are considered, the overhead is too much and it is about 50%. Due to low data rate, large number of messages, small payload size, large overhead and increase in vehicle complexity, the bus load of the networks is between 50% and 95% [33].

2.2 CAN FD

CAN Bus doesn't meet the data rate requirements of the contemporary in vehicle communication anymore. However, CAN is a trusted protocol being used for many years in countless applications in the automotive industry. For these reasons, CAN FD (CAN with Flexible Data Rate) which both offers much higher bandwidth than CAN and backward compatibility with CAN is developed by Bosch [28, 6].

CAN FD operates at two different bit rates within a message frame. It has the same bit rate and the arbitration method as CAN but it switches to a higher bit rate during the data phase. CAN FD payload is up to 64 Bytes. Therefore at a given bus load,

the overhead of the frame decreases down to 15% [33] and theoretical net bit rates about 5 Mbps are possible [6]. Furthermore, the arbitration phase baud rate limits the overall baud rate of the frame. For example, a frame with 64 bytes payload, 11 bit standard ID, 1 Mbps of arbitration phase baudrate, 8 Mbps of dataphase baud rate has net bit rate about 5.9 Mbps [40]. Furthermore, the increase of baud rate is also beneficial for higher layer software protocols.[41] assesses the effectiveness and performance of CAN FD with respect to CAN bus in agricultural systems using higher layer protocols like J1939 and ISOBUS.

According to [3], CAN FD data phase bit rates up to 2 Mbit/s will be used in the first CAN FD systems. The network topology will be like star or hybrid. Later generation CAN FD systems will increase the data rate up to 5 Mbit/s. CAN FD frames can be divided into three parts, which are arbitration phase, data phase and arbitration phase again as can be seen in Fig.2.2. The bit rate switches to higher rate only during data phase and switches back to its old rate when the data phase is over.

Car manufacturers begin to adapt CAN FD in their system design. Toyota, Denso, and Renesas cooperate for autonomous driving system developments. Renesas contributes with micro controllers and System on Chip (SoC) devices featuring CAN FD [17]. According to [14], Mercedes considers introducing CAN FD in their S-class series cars. There are some works to adopt CAN FD to real network systems and CAN FD is considered to be used in the same network with CAN [23], [24].

CAN FD frame format differs from CAN frame format in terms of payload length, Data Length Code (DLC) and CRC computation method. Therefore, some hardware changes are required in the controllers. If the payload size is kept as 8 Bytes as in standard CAN messages, there is no need for any software changes [25]. CAN FD supports payload size of up to 64 Bytes. In such implementations software changes are required. The cost to implement CAN FD is very similar to CAN implementation costs [28].

The frames with the base ID have 11 bits ID representation while the frames with the extended ID has 29 bits ID representation. Furthermore, some control bit values in the frame change according to the ID type. CAN FD data frames for Base and Extended ID can be seen in Fig.2.2a and in Fig.2.2b.

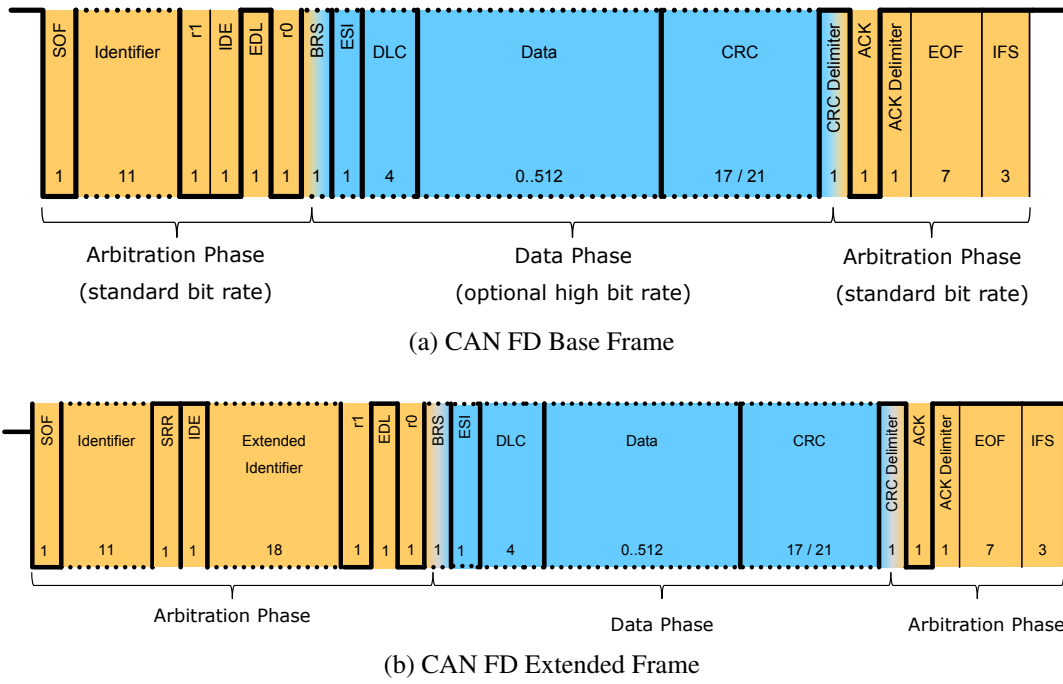


Figure 2.2: CAN FD Frames

Example use cases of CAN FD are listed below:

- Fast software downloads: Reprogramming of ECUs and performing their in-vehicle calibrations are typical cases requiring high data rates[27]. The footprint of the software increases day by day. Most of the contemporary ECUs use CAN Bus for software downloads, removing the necessity of additional communication interface like Ethernet, hence reducing the hardware costs. Since CAN FD provides higher bandwidth with 64 byte payload support and increased bit rate when compared to CAN, it is quite faster to download the software. According to [27], CAN FD provides 1.4 to 14 times faster programming times. For example, it takes 4.45 times longer to download 32 byte data via 500 kbit/s CAN bus when compared with 2Mbit/s CAN FD as illustrated in the following example:
 - Time to transmit 4 standard CAN messages with 8 data bytes and 15% stuff bits takes 1021 μ s [32].
 - Time to transmit 1 CAN FD message with 32 data bytes and 15% stuff bits takes 229 μ s [32].

Furthermore, although CAN FD offers better software programming performance when compared to CAN, it should be noted that Flexray and CAN FD

has comparable performance. Under some conditions Flexray is faster and under some other conditions CAN FD is faster according to[29].

- Avoiding split messages: Some information is represented with more than 8 bytes, in order to transmit such information multiple CAN frames are transmitted. This increases the work load of the software. With CAN FD's large payload capability up to 64 bytes, the information does not need to be splitted into several frames hence it is transmitted in a single message. Therefore, the transport layer software management becomes easier.
- Faster Communication: As the features increase in automotive industry, the data exchange between ECUs increases therefore CAN FD can handle the increased traffic with its higher bandwidth.
- Less Bus Load: Due to increase in bandwidth, the bus loading greatly reduces.
- Bus Length: As the number of nodes and the length of the stubs increase, the bit rate reduces. With the bus length of 40 meters, stub lengths of 3 meters and 30 nodes on the bus, SAE J1939-15 states 250 kbit/s bit rate for CAN Bus[32]. With CAN FD, data phase bit rate is independent of cable length therefore, with 250 kbit/s arbitration phase bit rate and 4 Mbit/s data phase bit rate, average bit rate is 810 kbit/s[32]. Therefore, communication speed is accelerated with long cables.

The CAN protocol is defined by the ISO 11898 standard. ISO 11898-1 specifies the Data Link Layer. CAN FD requirements are currently integrated as ISO 11898-1:201. Therefore, the first CAN FD version by Bosch [6] is called non-ISO CAN FD now. Different than the non-ISO CAN FD protocol, ISO CAN FD has different failure detection abilities. 3 bit stuff counter and an extra parity bit are added in the frame and the CRC computation value is modified when compared to non-ISO CAN FD. Therefore, non-ISO and ISO CAN FD protocols are incompatible with each other [11].

2.3 CAN/CAN FD Controllers

In order to participate in an in-vehicle network, an in-vehicle node needs to have micro controller (MCU), a bus controller and a transceiver. The MCU is the unit where the software application runs. The Controller implements the layer 2 protocol of the bus and transmits the data received from the MCU on the bus by executing necessary framing and arbitration. The transceiver connects the node to the physical medium of the Bus. The C^3 controller that we present in this thesis is a CAN FD controller. However, we note that the architecture can be adopted for CAN controllers as well. To this end, we present previous work on CAN/CAN FD controllers in Chapter 3.

The basic functions of the CAN/CAN FD Controllers are to convert the information that comes from the application into CAN/CAN FD frames by following the protocol specifications, extract the related information from the received frames and convey this data to the application. For transmission, the controllers perform physical level bit generation following the timing requirements, generate frames by implementing bit stuffing, perform CRC computation and error detection. For reception, they perform physical level bit sampling according to the timing requirements, apply ID based filtering, perform buffering of the received frames, remove the stuffed bits from the received bit sequence, check the CRC values and perform error detection.

The buffering is one of the most important features of CAN/CAN FD controllers. There are two types of buffer organization, which are FIFO (First In First Out) and mailbox. In FIFO organization, for transmission, what is written to FIFO first is taken out and transmitted first. The disadvantage of this method is that it is not possible to give priority to the messages which are required to be transmitted first. For the reception, the frame received first is written to FIFO first and is taken out by the application first. Similarly, the application should process the messages previously in the FIFO before it can finally reach the message with the high priority. This leads to undesirable delay for the high priority messages. Hardware cost of FIFO implementation is low and the controller design is simple.

In mailbox organization, there is a dedicated buffer for each message. The mailbox

buffers hold a single frame. For transmission, each mailbox has priorities assigned and they are transmitted in the order such that the highest priority mailbox is vacated first. Similarly for reception, the messages are placed in their corresponding mailboxes and the application reads the message with the highest importance first without having to read the others like in the case of FIFO concept. RX buffer organization particularly becomes significant if the MCU is slow at processing the received messages [36]. The hardware cost of the mailbox is higher since more memory is required, the controller design becomes more complex and the timing performance is better when compared to FIFO concept.

In order to reduce the MCU load, the receivers of the controllers have a filtering feature. Only the messages which pass the filters are placed to the buffers. By doing so, the unwanted messages are discarded by the controller so that MCU does not need to process them.

The controllers interface with the transceivers with the digital receive and transmit pins. Moreover, they communicate with the MCUs in two different ways. In the first method, the controller is integrated inside the MCU chip. The communication between the controller and the MCU processor core is the internal system bus which is a shared bus with the other peripheral controllers like UART, SPI, I2C and Ethernet. In the second method, the MCU does not contain the CAN/CAN FD controller inside the chip, instead, the controller is external to the MCU. This is the case for the low cost MCUs where the most of the peripherals already contained in higher cost MCUs are excluded from the chip to reduce the cost and the footprint of the chip. Common interfaces which can be used between the CAN/CAN FD Controller and the MCU are listed and discussed below:

- UART: UART is a serial communication protocol with the most common baud rate of 115.2 kbit/s. There are some instances of UART which are used in high performance MCUs with the baud rates of several Mbit/s. Since our case is for the low cost MCUs, an interface with the bit rate of several hundred kbits is not acceptable when the baud rate of CAN/CAN FD is considered. Interfaces having baud rates lower than CAN/CAN FD baud rate would result in huge response time, thus reducing the timing performance of the controller signifi-

cantly.

- I2C: I2C is a serial communication protocol with the most common baud rate up to 400 kbit/s. Due to the same reasons listed for UART, I2C is not a suitable interface for CAN/CAN FD Controllers
- PCIe: PCIe is a serial communication protocol with bit rates in the range of Gbit/s. Since low cost MCUs do not have PCIe interface, this protocol is not suitable for standalone CAN/CAN FD controllers.
- SPI: SPI is a serial protocol with the most common baud rate of 10 Mbit/s. This baud rate is higher than CAN/CAN FD baud rates. When high baud rates of CAN FD like 4 Mbit/s is implemented, higher SPI baud rates would be used to get better response time. Serial communication is better than parallel when the I/O pin utilization of MCU is considered. Low cost MCUs have limited amount of I/Os and any unused peripheral I/Os can be used for other purposes as there is I/O multiplexing for unused pins. SPI seems to be the best communication interface for CAN/CAN FD controllers.

CHAPTER 3

PREVIOUS WORK ON CAN/CAN FD CONTROLLERS

[34] covers CAN Controllers and their features. It presents a controller which has a maximum of 32 mailbox TX (Transmit) buffers. Regarding configurable CAN Controllers,[12] is a CAN Controller IP Core. TX buffers are organized as one high priority buffer and a FIFO with a configurable depth up to 64 message objects. RX (Receive) buffers are organized as FIFO with a configurable depth up to 64 messages. It features user configurable acceptance filters for the received messages, the number of the filters can be up to 4. The communication interface with the MCU is PLB v4.6 bus standard. [13] is an external CAN controller with SPI interface, it features two receive buffers with prioritized message storing. There are six 29 bit filters and two 29 bit masks for the received messages. It supports three transmit buffers with prioritization. The communication interface with the MCU is SPI. Since it is an ASIC (Application Specific Integrated Circuit), the depth of the buffers is fixed and not configurable like the controllers which are realized as IP Cores.

This thesis focuses on CAN FD Controller implementation. Bosch, the company that invented CAN and CAN FD protocols, has two CAN FD controllers which are realized as FPGA IP Cores. The first one is C_CAN FD8. It supports CAN FD messages with the payload up to 8 bytes. It contains 32 message objects and ID masks for each of the objects. The message objects can also be programmed as FIFO and they are used for both the transmitted and the received messages. The user interface is AMBA APB bus for ARM processors and Avalon bus for ALTERA FPGAs. The purpose of this IP core is to maintain the compatibility with the existing Bosch CAN controller as CAN bus message payload is also up to 8 bytes. The second IP core by Bosch is M_CAN, which supports both ISO 11898-1:2015 and non-ISO CAN FD imple-

mentation [2] (2015). The payload of CAN FD frames supported is up to 64 bytes. The memory where the buffers are located is not internal to the controller, instead it uses the existing single or dual port memory inside the MCU. The interface to the external memory is 32 bit generic master interface. It features two configurable Receive FIFOs with up to 64 message objects with filtering capability. Furthermore, it supports 32 TX buffers whose message size can be configured. However, the configuration sizes for all of the buffers are the same, therefore, the buffer sizes should be configured according to the longest message payload size. This leads to inefficient memory allocation especially for heterogeneous message sizes. TX handler block inside the controller picks the message with the highest priority to transmit among all the buffers. The controller interfaces with the MCU via its 8/16/32 bit generic slave interface.

[1, 9, 18] are the CAN FD IP cores developed in 2015. [1] supports non-ISO CAN FD implementation. Total buffer size is synthesis time configurable. Buffers are organized as transmit buffer, high-priority transmit buffer and receive buffer whose depths are individually configurable by the MCU. There are up to 16 acceptance filters. The interface to the MCU is AHB-Lite slave interface. [9] supports both non-ISO and ISO CAN FD implementation. It features synthesis time configurable depths for receive and transmit FIFOs. It contains 256 message filters for the received messages. It interfaces with the MCU with 8/16/32/64 bit system bus. [18] also supports non-ISO and ISO CAN FD. The size of the transmit and the receive buffers are configurable during synthesis time. The interface to the MCU is via the system bus.

[5, 10, 4] are the CAN FD IP cores on the market since 2016. [5, 10, 4] support ISO and non-ISO CAN FD formats. [5] features transmit buffers with up to 32 message objects. Receive buffers support up to 48 message objects with ID filtering featuring. The buffers can be configured as FIFO or mailbox. Transmit handler selects the highest priority message to begin transmitting. The number of the buffers is fixed and not configurable. The MCU interface is AXI4-Lite bus. [10, 4] are IP Cores similar to each other. [10] implements two types of transmit buffers. One is the high priority primary transmit buffer, the other one is the lower priority secondary transmit buffer. The high priority transmit buffer can store only one message. However, the depth of

the lower priority buffer is synthesis time configurable. The size of the receive FIFO can be configured during the synthesis and there are up to 16 independently programmed filters for the received messages. The MCU interface options are generic 32-bit host controller interface, AHB, APB (32 bit), generic 8 or 16 bit. Similarly, [4] offers one high priority transmit buffer and configurable number of low priority buffers. Moreover it supports RX FIFO buffering with up to 29 bit acceptance filtering. The configuration is done during synthesis. The interface to the host MCU is generic 8 bit host controller, 8/16/32 bit AMBA-APB or 32 bit AHB-Lite.

[8, 20, 7] don't have non-ISO CAN FD support. [8] has fixed size 128 byte receive buffer and transmit buffer. It supports message filtering. The communication between the controller and the MCU is 8/16/32-bit CPU slave interface. [20] offers synthesis time configurable mailboxes. The mailboxes can be used for both the transmitting and the receiving functions. Furthermore, there is a RX FIFO which can store 6 frames. The MCU communication interface is on-chip system bus. [7] CAN FD IP Core has TX and RX FIFOs with synthesis time configurable sizes. There are user configurable acceptance filters. MCU interface is AMBA-AXI4-Lite interface (32-bits) or standard address/data configuration Interface.

[39] is a recent paper which presents CAN FD IP Core with SPI interface. There are no details about the interface between the controller and the host MCU. The interface is described very briefly. The buffers have no configuration capability. The design is explained very coarsely with very little implementation detail. The design is only verified by oscilloscope signal inspection.

To sum up, to the best of our knowledge C^3 is the first CAN FD IP core that offers very flexible and efficient configuration capability to the software developer during application run time (after synthesis), with a total number of 192 mailboxes. Mailboxes are organized as 96 transmit and 96 receive buffers. The mailboxes are allocated an average payload size of 16 bytes instead of maximum payload size of 64 bytes. Because it is very unlikely that all the messages in a message set of a network application will require 64 bytes payload size. This yields 96x16 bytes memory consumption instead of 96x64 bytes hence less memory usage and more memory utilization. It is the first controller with SPI interface whose response time is measured, academically

evaluated, tested and analyzed. Different from other controllers, C^3 supports up to 96 filters for each RX buffer for the received messages.

The comparison of the CAN FD IP cores mentioned in this chapter can be seen in Table 3.1 , Table 3.2 and Table 3.3. We define the performance criteria for comparison as follows:

- Year: Release year of the controllers
- Transmit Buffer Type: Buffer organization structure for the transmit messages, either FIFO or mailbox
- Transmit Buffer Configuration: Configurability properties of the transmit buffers
- Receiver Buffer Type: Buffer organization structure for the received messages, either FIFO or mailbox
- Receive Buffer Configuration: Configurability properties of the receive buffers
- Buffer Configuration Time: The time when the configuration takes place, either synthesis time or application time. Synthesis time configuration is one time configuration before the application runs and does not give the user the ability to reconfigure the buffers during run time while application time configuration gives the user the ability to reconfigure the buffers during run time for different applications.

Table3.1: CAN FD Controllers Comparison 1

| | [12] | [10] | [4] | [1] |
|--------------------------------------|--|--|--|--|
| Year | 2016 | 2016 | 2016 | 2015 |
| Transmit Buffer Type | FIFO or mailbox with 32 elements | FIFO | FIFO | FIFO |
| Transmit Buffer Configuration | Transmit buffers' total size is configurable | Primary ve secondary transmit buffers. Total size of the secondary transmit buffer is configurable | Primary ve secondary transmit buffers. Total size of the secondary transmit buffer is configurable | Primary ve secondary transmit buffers' total size is configurable |
| Receiver Buffer Type | FIFO or mailbox with 48 elements | FIFO | FIFO | FIFO |
| Receive Buffer Configuration | Receive buffers' total size is configurable | Receive buffers' total size is configurable | Receive buffers' total size is configurable | Receive buffers' total size is configurable |
| Buffer Configuration Time | Synthesis time | Synthesis time | Synthesis time | Total memory size is synthesis time, each FIFO size is application time configurable |
| User Interface | CPU Interface, AXI-4 Lite Bus | 8/16/32 bit CPU Interface | CPU Interface, AMBA, APB or AHB-Lite Slave Interface | CPU Interface, AHB-Lite Slave Interface |

Table3.2: CAN FD Controllers Comparison 2

| | [2] | [8] | [20] | [9] |
|--------------------------------------|---|--|---|---|
| Year | 2016 | 2016 | 2016 | 2015 |
| Transmit Buffer Type | FIFO or mailbox with 32 elements | FIFO | Mailbox | FIFO |
| Transmit Buffer Configuration | All mailboxes can be configured but the configuration size have to be the same for each mailbox | 128 byte transmitter FIFO is fixed size | Total number of mailboxes can be configured | Transmit buffers' total size is configurable |
| Receiver Buffer Type | FIFO or mailbox with 64 elements | FIFO | Mailbox or FIFO | FIFO |
| Receive Buffer Configuration | All mailboxes can be configured but the configuration size have to be the same for each mailbox | 128 Byte Receive FIFO is fixed size | Total number of mailboxes can be configured | Receive buffers' total size is configurable |
| Buffer Configuration Time | Application time | Synthesis time | Unknown | Synthesis time |
| User Interface | 8/16/32 bit CPU Interface | CPU interface, 8/16/32-bit CPU slave interface | CPU Interface, Peripheral BUS Interface | CPU Interface, 8/16/32/64-bit CPU slave interface |

Table3.3: CAN FD Controllers Comparison 3

| | [18] | [7] | C ³ : Configurable CAN FD Controller |
|-------------------------------|--|--|---|
| Year | 2015 | 2016 | 2017 |
| Transmit Buffer Type | FIFO | FIFO | Mailbox |
| Transmit Buffer Configuration | Transmit buffers' total size is configurable | Transmit buffers' total size is configurable | Configurable total size of memory. Each mailbox can be configured with desired size and the number of the mailboxes can be configured up to 96 |
| Receiver Buffer Type | FIFO | FIFO | Mailbox |
| Receive Buffer Configuration | Receive Buffers' total size is configurable | Receive Buffers' total size is configurable | Total size of memory is configurable. Each mailbox can be configured with the desired size and the number of the mailboxes can be configured up to 96 |
| Buffer Configuration Time | Synthesis time | Synthesis time | Total memory size is synthesis time, all the other settings are application time configurable |
| User Interface | CPU Interface, System Bus Interface | CPU Interface, System Bus Interface | SPI |

CHAPTER 4

C^3 (CONFIGURABLE CAN FD CONTROLLER) ARCHITECTURE

This chapter presents the hardware architecture of our proposed C^3 Configurable CAN FD Controller. A brief presentation for the implementation details, evaluation and the response time measurements of C^3 : Configurable CAN FD Controller can be found in [22], [21] which are presented in the Appendix of this thesis .

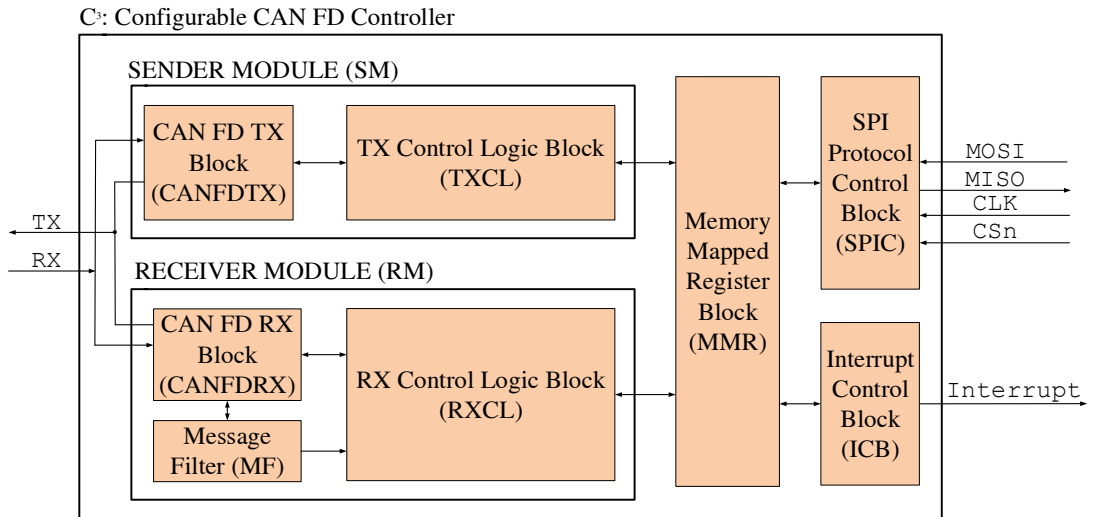


Figure 4.1: C^3 Architecture

C^3 implements the CAN FD Base & Extended Format Data Frame according to the non-ISO specification [6] as we introduce in Section 2.2. Our proposed hardware architecture features a TX Buffer memory and an RX Buffer memory each with a fixed size. The MCU programmer can organize these memory areas into respective TX and RX buffers with desired message size and message count. In our implementation, a maximum of 96 TX buffers and 96 RX buffers can be configured in mailbox form. C^3 transmits the messages in the TX buffers according to the priority order. Each RX

buffer has an ID-Mask pair for message filtering. The interface between C^3 and the host MCU is through SPI (Serial Peripheral Interface).

We present the C^3 FPGA IP Core block architecture together with the host MCU and a transceiver in Fig.4.2. Transceiver is used to convert controller's single ended TTL/CMOS level signals to differential ended CAN BUS physical layer signals, CAN High and CAN Low.

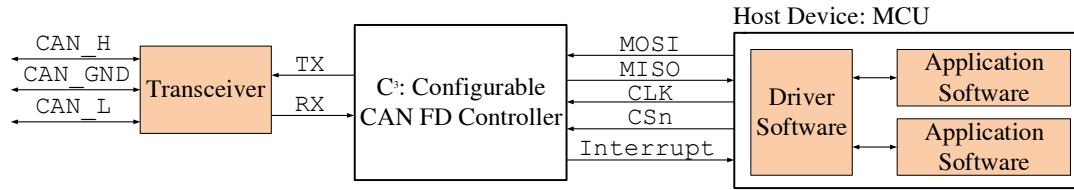


Figure 4.2: Top Level Architecture

The top level hardware blocks of C^3 are; Sender Module (SM), Receiver Module (RM), and the user interface blocks which are the Memory Mapped Register Block (MMR), SPI Control Block (SPIC) and Interrupt Control Block (ICB). The CAN standard requires sending an acknowledgment after each message reception. To this end, the SM and RM are both connected to the physical layer TX/RX lines on the transceiver. The RX and TX buffers are realized on Prioritized Flexible TX Memory Block (PFTM) and Flexible RX Memory Block (FRM) in MMR respectively as in Fig.4.3.

The hardware resources of C^3 are configured by writing the register array implemented in the MMR. These registers are accessible by the C^3 *Driver software* using the standard SPI protocol signals MISO (Master In Slave Out), MOSI (Master Out Slave In), CSn, CLK and interrupt signal. After the configuration, the *application software* starts to run on the host MCU which generates and consumes the data carried in the CAN FD message payload. Host CPU sees the controller as a set of registers which can be read and be written to according to the controller's protocol requirements. The application software sends and receives message data together with their CAN IDs through the driver software. C^3 runs at 100 MHz. We provide implementation details in Section 4.8 together with the evaluation results in Section

5.

4.1 Hardware Blocks: Memory Mapped Register Block

The host MCU communicates with C^3 for configuration and data transmission using a memory mapped architecture that is realized with a set of registers in MMR. The addresses of these registers are known to the Driver Software. The host device reads and writes these registers via our SPI protocol signals.

The block diagram of Memory Mapped Register Block (MMR) can be seen in Fig.4.3.

There are 96 sets of TX registers and 96 sets of RX registers to support up to 96 TX and RX buffers respectively.

Each TX register set consists of

- TX ID Register
- TX DLC (Data Length Code) Register
- TX Data Register

Each RX register consists of

- RX ID Register
- RX Mask Register
- RX DLC (Data Length Code) Register
- RX Data Register

MMR also features the following registers: Transmit Control & Status Registers

- Transmission Request Register (TRR)
- Transmit Message Status register (TMSR)

Receive Control & Status Registers

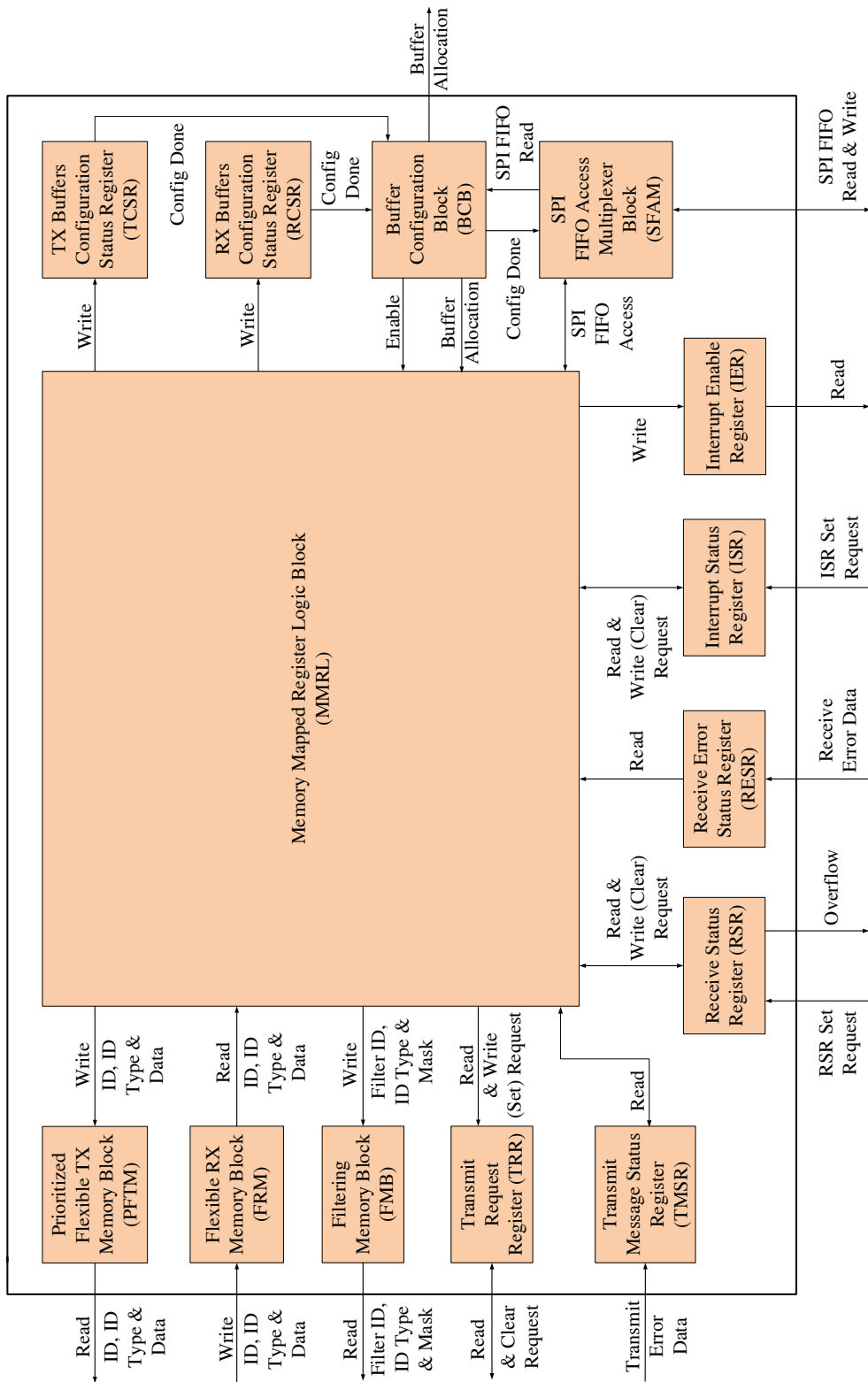


Figure 4.3: Memory Mapped Register Block (MMRL)

- Receive Status Register (RSR)
- Receive Error Status Register (RESR)

Interrupt Control & Status Registers

- Interrupt Enable Register (IER)
- Interrupt Status Register (ISR)

Buffer Configuration Control Registers

- RX Buffers Configuration Status Register (TCSR)
- RX Buffers Configuration Status Register (RCSR)

For the rest of the thesis we write **TX(ID)** and **RX(ID)** to indicate a specific register set for a CAN ID. The content of the data register of **TX(ID)** is the payload of the message with the specified CAN ID.

SPI FIFO Access Multiplexer Block (SFAM): As described in detail in Sec.4.2, Host MCU interfaces C^3 with SPI, and MMR communicates SPCB with FIFO accesses. SFAMB multiplexes FIFO signals between MMR Logic Block (MMRL) and Buffer Configuration Block (BCB). Due to the nature of hardware design, a signal cannot be driven by two sources, therefore for situations where multiple drivers are required, a multiplexing method is used. This multiplexer first gives access to BCB, after the buffer allocation is done, access is given completely to MMRL.

Buffer Configuration Block (BCB): This Block manages the **TX DLC (Data Length Code) Register** and **RX DLC (Data Length Code) Register**. **DLC Registers** must be configured to arrange the size of the buffers. The content of a **DLC Register** can be seen in Fig.4.6 and Fig.4.7

Host MCU has to configure the registers one by one in an ordered way. First TX buffers then RX buffers configuration should be done. After TX buffer sizes are configured, host should write **TX Buffers Configuration Status Register (TCSR)** to indicate TX buffer configuration is finished so that RX buffer configuration can start.

Same process is also repeated for RX buffers. Finally, **RX Buffers Configuration Status Register (RCSR)** is written by the host to indicate whole configuration process is finished. As the configuration finishes, the buffer allocation information, the address and size of each buffer in the Prioritized Flexible TX Memory Block (PFTM) and Flexible RX Memory Block (FRM), is kept as registers and used by Sender Module (SM) and Receiver Module (RM) and MMR Logic Block (MMRL). Furthermore, after the whole buffer configuration process is complete, SFAM block is informed so that SPI signals are routed to MMRL block. The content of **TCSR** and **RCSR** are shown in Table 4.1.

Prioritized Flexible TX Memory Block (PFTM): The host MCU can partition PFTM to implement up to 96 TX buffers in desired sizes. This block is designed as a 1920 bytes block RAM. However, the size of PFTM can be increased during synthesis. There are 480 addressable locations each with a depth of 32 bits. We implement the buffers in the mailbox architecture where each TX buffer is allocated for a specific CAN ID and can store the payload for a single frame. The memory is organized with an appropriate size to store 4 byte ID registers and an average of 16 bytes data for each of the 96 buffers. $(96 \times 4 + 96 \times 16) = 384 + 1536 = 1920$ bytes) Using payload size average of 16 bytes per mailbox instead of allocating 64 bytes for each mailbox yields less memory consumption and more memory utilization. Two example buffer configurations that fully utilize 1536 bytes data area are depicted in Fig.4.4

The first buffer in the memory has the highest priority. This information is known to host MCU and configuration is done with taking this information into account. MMRL writes ID & Data to be transmitted to the related buffer location when a write request comes from Host MCU. TX Control Logic Block (TXCL) has read access to this memory block to get the ID and data to transmit.

Flexible RX Memory Block (FRM): This block is designed as a 1920 bytes block ram. There are 480 addressable locations each with a 32 bit depth. Memory is organized as large as it can store 4 byte ID registers and 16 bytes data for each 96 buffers. $(96 \times (4 + 16)) = 1920$ bytes). Using payload size average of 16 bytes per mailbox instead of allocating 64 bytes for each mailbox yields less memory consumption and more memory utilization. RX Control Logic Block (RXCL) has write access to this

Table4.1: Configuration Registers

| | Address (0x) | Register Name | Read (R)/ Write (W) | Description |
|-------------------------------|--------------|--|------------------------|---|
| Config Status Register Set | 0 | TX Buffers Configuration Status Register (TCSR) | W | Host should write to this register once the configuration is completed Register[0] => 1 => Transmit Buffers Initialization is completed Register[0] => 0 => Transmit Buffers Initialization is not completed Register[7:1] => # of Transmit Buffers whose Initialization is completed, e.g. => 23 means buffers 1 to 23 are initialized |
| | 4 | RX Buffers Configuration Status Register (RCSR) | W | Host should write to this register once the configuration is completed Register[0] => 1 => Receive Buffers Initialization is completed Register[0] => 0 => Receive Buffers Initialization is not completed Register[7:1] => # of Receiver Buffers whose Initialization is completed, e.g. => 23 means buffers 1 to 23 are initialized |

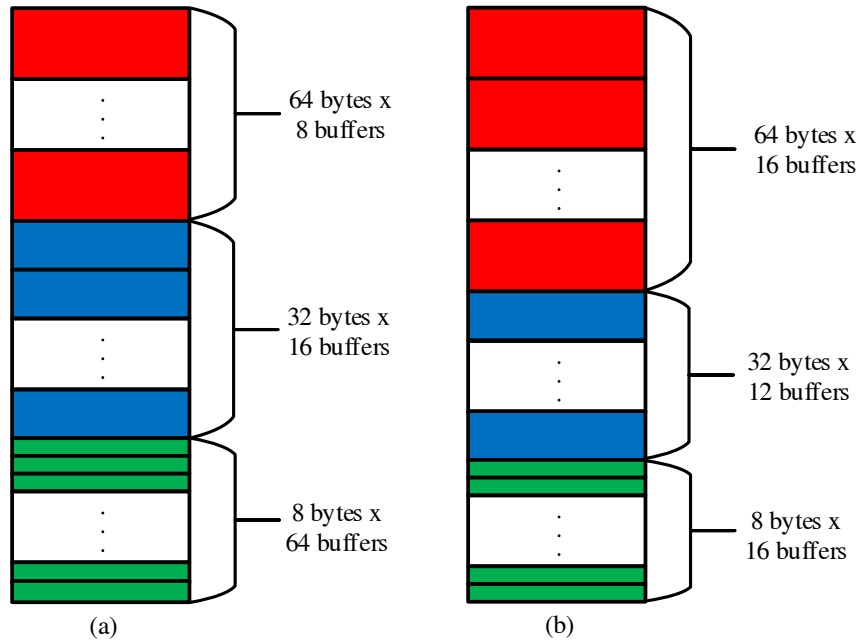


Figure 4.4: Example Memory organizations: (a) 88 buffers (b) 44 buffers

memory block to put the ID and data of the received frames in the related location. As read request comes from the Host MCU, MMRL reads the related information from the related buffer address and responds to the request.

Filtering Memory Block (FMB): The Memory consists of 32 bits of ID & Mask pairs for each 96 buffers. There are 192 locations with depth of 32 bits. During configuration, Host MCU programs ID & Mask pair for each buffer to be used. MMRL has write access to this memory block, it writes ID & Mask pairs as requested by the Host MCU. Message Filter (MF) reads from this memory during filtering process.

MMR Logic Block (MMRL): After buffer allocation is done, this block takes over the SPI FIFO access from BCB. All of the CPU's write and read requests are handled here. This logic block has control for all of the registers except the ones with buffer size configuration (**TX DLC (Data Length Code) Register**, **RX DLC (Data Length Code) Register**, **TX Buffers Configuration Status Register (TCSR)** and **RX Buffers Configuration Status Register (TCSR)**). The state machine here polls the SPI instruction FIFO to see if there are any pending requests. The machine has two different state sets, one for write requests and one for read requests. The write state set gets the address of the register from the address FIFO in SPCB, gets the burst

data size from burst data FIFO in SPCB and finally gets the data part from RX Data FIFO in SPCB. The amount of data to be read from the RX data FIFO depends on the instruction type and burst data size. For the write requests, either a related memory address is written or related register is written. The addresses read from Address FIFO is mapped to PFTM, FMB memory addresses and directly to some registers as seen in the Fig.4.3.

Similarly, the read state set gets the address of the register from the address FIFO in SPCB, gets the burst data size from burst data FIFO in SPCB. The addresses read from Address FIFO are mapped to FRM memory addresses and directly to some registers as seen in the Fig.4.3. Since the read requests are non posted requests, MMRL gets the data from the related mapped address or from a register and writes the data to the TX Data FIFO quickly such that when SPB reads the data to respond, the data is present in the FIFO.

The mechanism of each register accessed by MMRL will be described in the related block descriptions.

4.2 Hardware Blocks: SPI Protocol Control Block

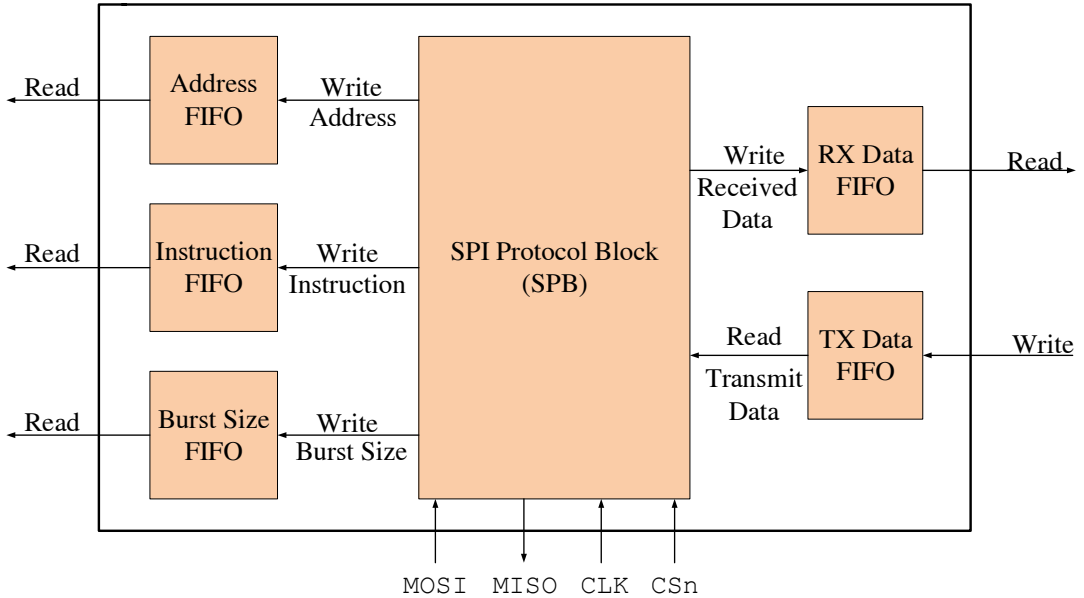


Figure 4.5: SPI Protocol Control Block

We develop and implement the SPI (Serial Peripheral Interface) control block in the

scope of this thesis work. The block diagram of SPI Protocol Control Block (SPCB) can be seen in Fig.4.5. The SPI standard defines the physical layer of the interface, timing of the signals and the roles of the master and the slave devices. The SPI physical layer signals going out of the controller are as follows:

- CS_n (Chip Select)
- CLK (Clock)
- MOSI (Master Out Slave In)
- MISO (Master In Slave Out)

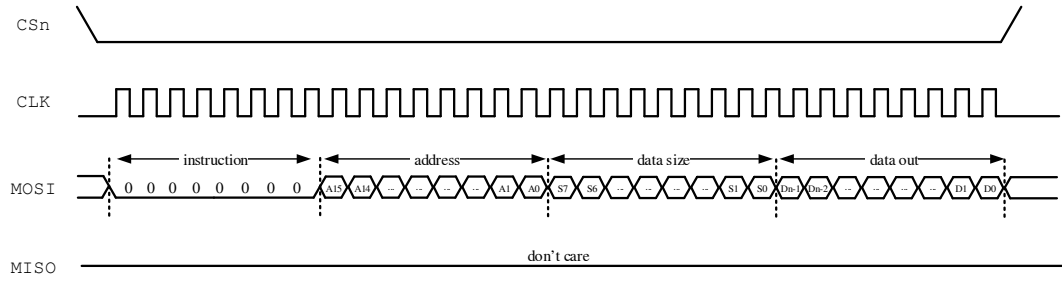
Master device has the following tasks:

- It controls CS_n , MOSI, CLK signals
- It asserts CS_n along with the CLK and data on MOSI line
- It deasserts CS_n to indicate that the communication is over

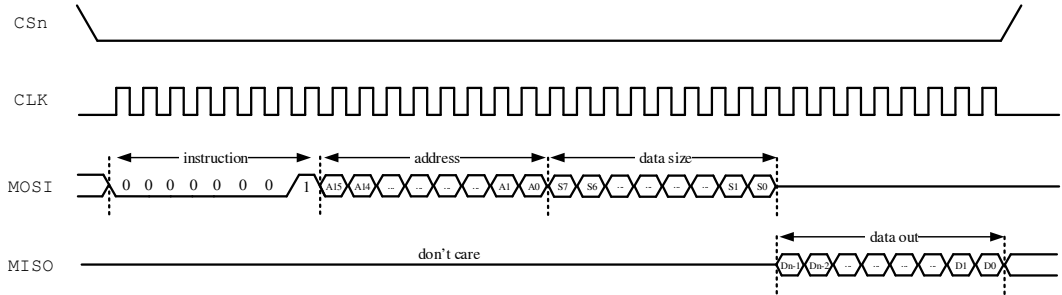
Slave device has the following tasks:

- When CS_n is asserted by the host, it starts to sample data at the rising edges of the clock.
- It gets the data on MOSI line and responds on MISO line if a response is required. If not, it just gets the data until the CS_n is deasserted

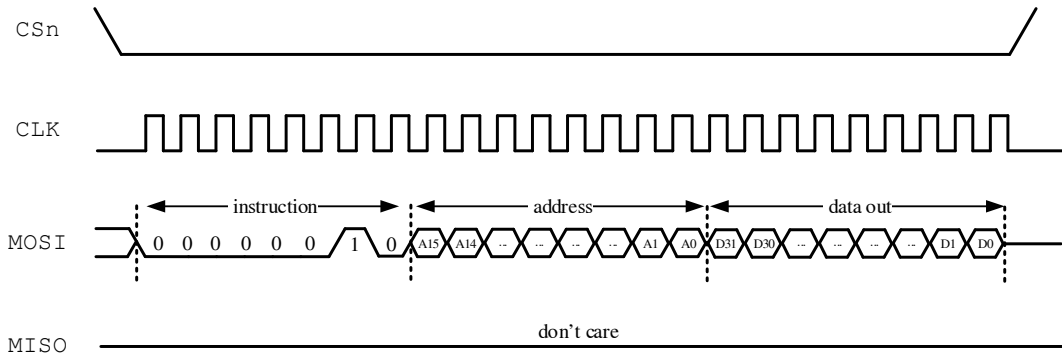
The SPI protocol only defines the mechanisms listed above. The custom protocols using SPI define the data amount and data content which the master device sends on MOSI line and the data content to which slave needs to send a response, the content and the amount of the data slave puts on MISO line, the time when the slave begins to respond when the slave response is required. Accordingly, we designed and implemented our own communication protocol with SPI. Our communication protocol over SPI between the host MCU and C^3 is described in Table 4.2.



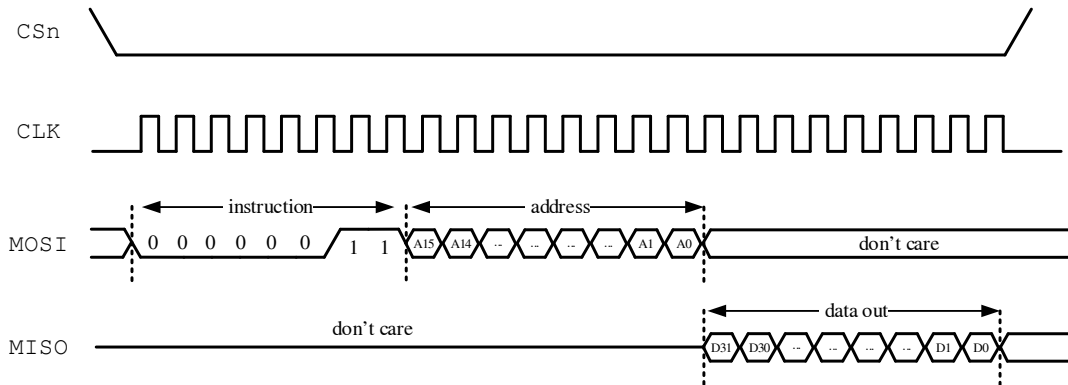
(a) Burst SPI Write



(b) Burst SPI Read



(c) 32 Bit SPI Write



(d) 32 Bit SPI Read

Figure 4.6: SPI Timing Diagrams

Table4.2: SPI Commands and Responses

| Access Type | Host Request Inst. Code | Host Data | C^3 Response |
|------------------|-------------------------|---|----------------|
| Burst SPI Write | 0x00 | Address (16 bits) + Data size (8 bits) + Burst data | — |
| Burst SPI Read | 0x01 | Address (16 bits) + Data size (8 bits) | Burst data |
| 32 bit SPI Write | 0x02 | Address (16 bits) + Data (32 bits) | — |
| 32 bit SPI Read | 0x03 | Address (16 bits) | Data (32 bits) |

The host requests are on the `MOSI` line and the C^3 response is on the `MISO` line. The request types are defined with the instruction codes of 8 bits. After the instruction, regardless of its type, master device puts the 16 bit address of the register on `MOSI`. Depending on the instruction type, host might go on transmitting data or slave responds to the request. During Burst Read or Write instructions, the amount of data is also provided to the slave with 8 bits of burst data size. The burst data transfers are in multiples of 4 bytes. Fig.4.6 depicts the signaling of our protocol for the 32 bit and burst instructions.

Our implementation is a high-speed SPI with 10 MHz clock frequency.

During master write operations, SPI Protocol Block (SPB) samples the bits at the rising edges of the clock when the `CSn` signal is low. It first gets 8 bit instruction and writes this data to the Instruction FIFO, then gets 16 bit address and writes the address into the Address FIFO. Depending on the instruction type, it either gets 8 bits of burst data size and writes it to the burst size FIFO or just skips to the data part. In the data phase, the data is written to RX Data FIFO in multiples of 32 bits. If it's a 32 bit SPI Write, only 32 bit data is written to the FIFO, otherwise the quantity of data to be written to the FIFO is determined according to the amount of data contained in the burst data size field. During master read operation, 8 bit instruction and 16 bit address is fetched by the SPB and written to the respective FIFOs. If the instruction is 32 bit SPI Read, SPB gets the data from TX Data FIFO and responds to the master on `MISO` line. Otherwise, SPB fetches burst data size and writes this information to Burst Size FIFO and gets as much as data as indicated in burst data size from TX

Data FIFO and responds to the master on MISO line. In summary, SPCB decodes the received frames and extracts the fields and puts the data contained in the fields to the related FIFOs and during read operations where response is required, it gets the data from the FIFO and responds. Memory Mapped Register Block (MMR) in C^3 interfaces SPCB with FIFO control signals therefore, SPCB is interpreted as a black box. SPCB provides read access for the Address, Instruction, Burst Size and RX Data FIFOs and write access for the TX Data FIFO for MMR.

4.3 Hardware Blocks: Interrupt Control Block

Interrupt Control Block (ICB): The block diagram of this block can be seen in Fig.4.7. There are four types of interrupt sources, which are

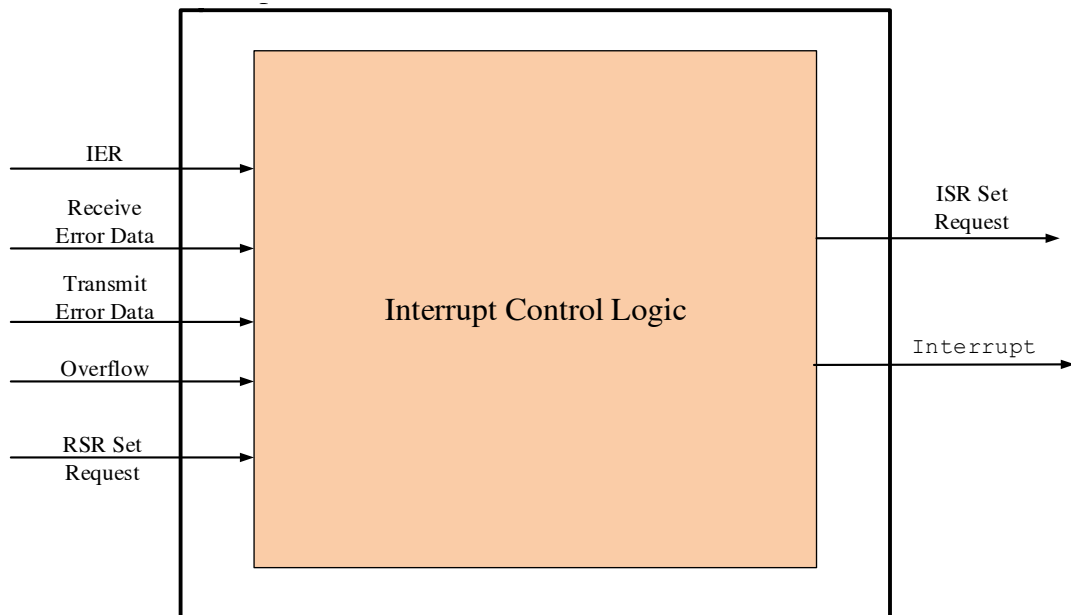


Figure 4.7: Interrupt Control Block

- Transmit Error
- Receive Error
- Receive Buffer Overflow
- Message Reception Event

Any of the interrupt sources can be enabled by writing to the **Interrupt Enable Register (IER)** shown in Table 4.3.

The block gets **IER** from MMR. Receive Error Data & Transmit Error data is obtained from CANFDRX and CANFDTX blocks which are polled for any error occurrences and if enabled, an interrupt is generated. Furthermore, overflow information is received from **RSR** management and if enabled, an overflow interrupt is generated. Similarly, when a message is received and placed in a buffer then **RSR** is requested to be set. When this signal is received, an interrupt is generated if enabled.

The interrupt is generated as an active low pulse for a duration of 1 ms.

For the Host MCU to learn the source of the interrupt, the source information of the interrupt is put in the **Interrupt Status Register (ISR)** after the processing the data obtained from other blocks for a host to read. Even if the the interrupt is not enabled for a source, related **ISR** bit is set if it occurs but an interrupt is not generated. When the host MCU reads **ISR**, it may also get additional interrupt information, which can be discarded if any action is not needed.

4.4 Hardware Blocks: Transmitter Module

TX Control Logic Block (TXCL):

TXCL block diagram can be seen in Fig.4.8. This block determines the message to be transmitted. Transmit Request Register shown in Table 4.4, holds the information of the pending transmission requests. When host MCU wants to transmit frames, it makes a request by writing to **TRR**. There are 3 **TRR** each holding 32 buffer transmission requests.

TXCL finds the buffer with the highest priority to transmit among the buffers waiting for transmission by using binary search algorithm. The algorithm is depicted in Fig.4.9. Each step in the Fig.4.9 takes a single clock cycle. Any pending request is held as logic **1** in **TRR**. The algorithm first checks if any one of the three of the **TRR** (3 x 32 buffers) is not equal to all **0s**. In the first step, it determines the **TRR** which is not all **0s**, if more than one **TRR** satisfies the condition, the one having the

Table4.3: Interrupt Register Set

| | Address (0x) | Register Name | Read (R)/ Write (W) | Description |
|---------------------------|--------------|---------------------------|------------------------|--|
| Interrupt Register Set | 28 | Interrupt Enable Register | W | <p>Interrupt Enable Control Register</p> <p>Register[x] = 1 => Interrupt is enabled</p> <p>Register[x] = 0 => Interrupt is disabled</p> <p>Register[0] => Transmit Error Enable, read Transmit Message Status Register for details</p> <p>Register[1] => Receive Error Enable, read Receive Message Status Register for details</p> <p>Register[2] => Receive Buffer Overflow Interrupt Enable</p> <p>Register[3] => Message received Interrupt Enable</p> <p>Register[31:4] => Reserved</p> |
| | 2C | Interrupt Status Register | R/W | <p>Interrupt Status Register, Write '0' to related bits to clear related bits of this register</p> <p>Register[x] = 1 => indicates interrupt is generated from x source</p> <p>Register[x] = 0 => indicates interrupt is not generated from x source</p> <p>Register[0] => Transmit Error, perform reading of Transmit Message Status Register for details</p> <p>Register[1] => Receive Error, perform reading of Receive Message Status Register for details</p> <p>Register[2] => Receive Buffer Overflow Interrupt</p> <p>Register[3] => Message reception Interrupt Enable, perform reading of Receive Status Register</p> <p>Register[10:4] => Buffer index of the overflow event</p> |

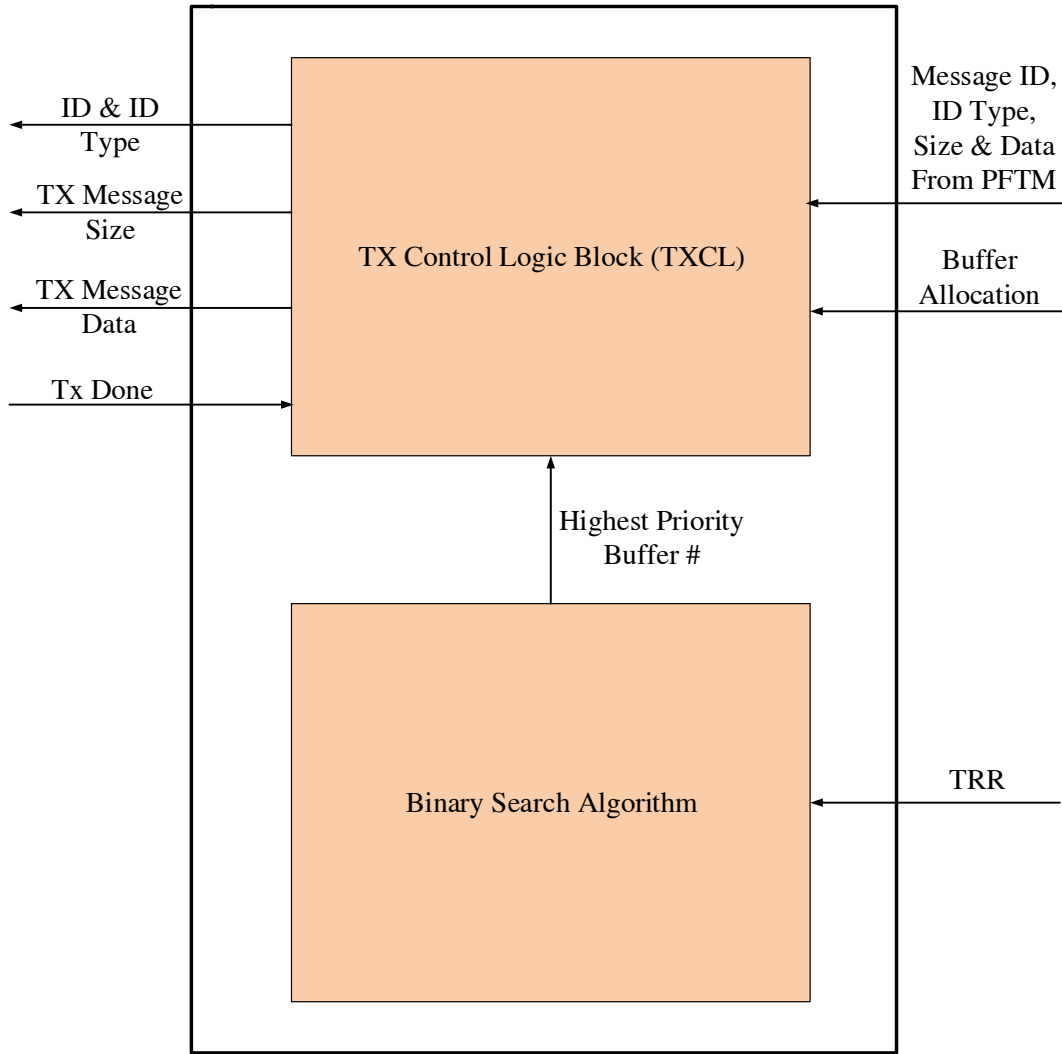


Figure 4.8: TX Control Logic Block (TXCL)

lower indexes is chosen. In the next step, algorithm divides the 32 bit **TRR** in two and checks if any of the partitions is not all **0s**. If more than one partition satisfies the condition, the one with the lower indexes is chosen. The algorithm proceeds as we describe until the last step and the buffer number is determined. The whole process takes 6 clock cycles. The algorithm is depicted in Fig.4.9.

Determining the buffer number means, storing the address and size of the buffer in PFTM because buffer allocation information is shared with TXCL. Therefore, TXCL gets the ID and data from the related part of PFTM. It writes the data to the TX Data FIFO of CANFDTX and the size to the TX Size FIFO. It loads the ID to CANFDTX. When the transmission is successful, CANFDTX indicates that TX is done and TXCL begins searching for a new pending transmission request if there is any. If it finds a

Table4.4: TX Control Register Set

| | Address (0x) | Register Name | Read (R)/ Write (W) | Description |
|-------------------------------------|--------------|-------------------------------------|------------------------|---|
| Transmit Control Register Set | 8 | Transmit Request Register 1 | R/W | Host writes to related field to initiate transmission, core clears when the transmission is completed successfully Register[31] => Priority Index 32 Transmission Request Register[30] => Priority Index 31 Transmission Request |
| | C | Transmit Request Register 2 | | |
| | 10 | Transmit Request Register 3 | | Register[0] => Priority Index 1 Transmission Request Register [31:3] => 29 Bit ID of the last transmission |
| | 14 | Transmit Message Status Register | R | Register[2] => ID Type of the frame, 0 => Base ID, 1 => Extended ID Register[1:0] => 2 bit code of the last reception Status Register[1:0] => "00" => No Error Register[1:0] => "01" => Bit Error Register[1:0] => "10" => Ack Error Register[1:0] => "11" => Reserved |
| | | | | |

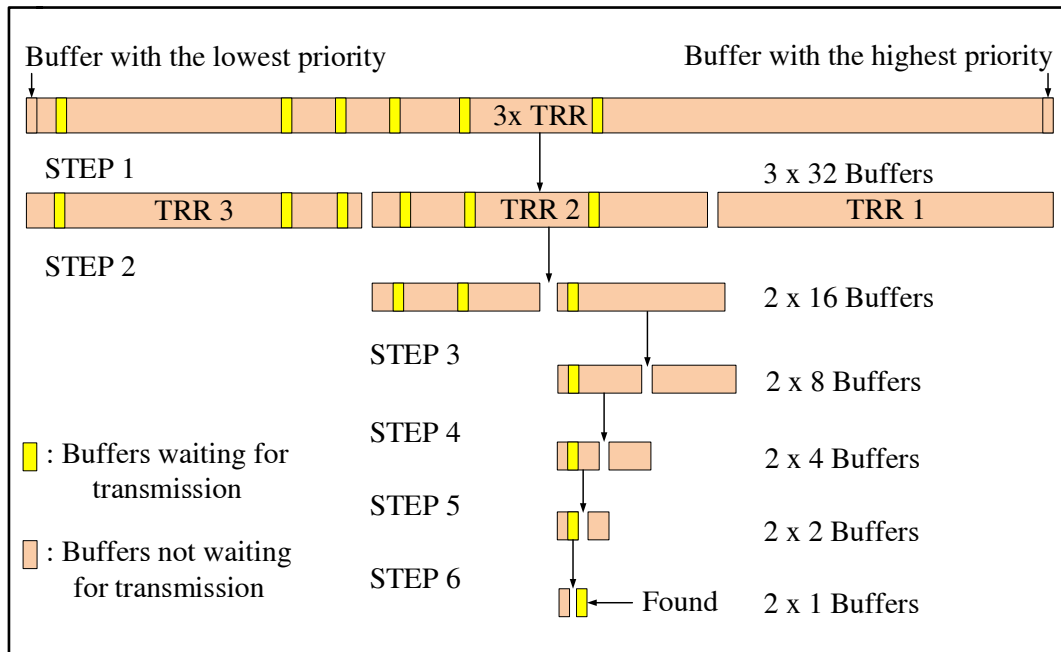


Figure 4.9: Binary Search Algorithm

pending request, it repeats the whole process described here.

CAN FD Transmitter Block (CANFDTX):

This block creates properly formatted CAN FD frames according to CAN FD protocol specification. Its block diagram can be seen in Fig.4.10. CANFDTX polls TX Size FIFO, if the FIFO is not empty, it reads the size of the message and starts transmitting. CANFDTX gets the payload from TX Data FIFO. It inserts CAN ID and other control information such as DLC at the beginning of the frame. It computes the CRC by communicating CRC Calculator Block (CCB), it inserts it at the end of the frame and performs the necessary bit stuffing of the constructed frame as defined in the standard. CANFDTX executes the CAN arbitration when the frame is ready to transmit and switches to the high CAN FD bit rate when the frame successfully completes the arbitration phase. Data and Size FIFOs do not hold more than 1 frame information. CRC Calculator Block calculates either 17 bit or 21 bit CRC depending on the message payload size as defined in the standard.

When the frame is transmitted successfully, TRR Clear Block requests to clean the related bit in **TRR**. Furthermore, after **TRR** is cleared, CANFDTX indicates that the transmission is done and it is ready for a new transmission.

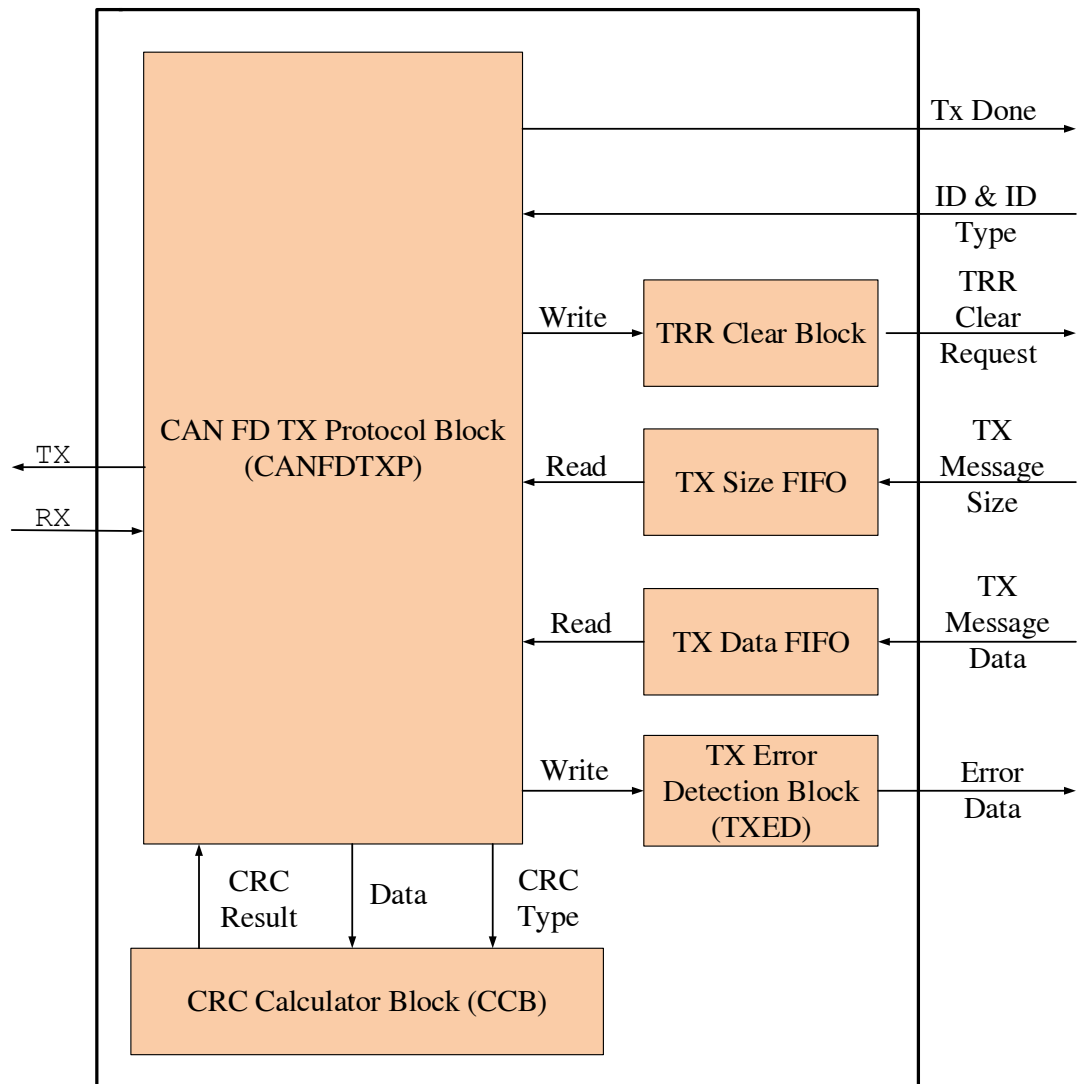


Figure 4.10: CAN FD Transmitter Block (CANFDTX)

The controller detects that an arbitration is lost if a different bit received is on the line than what is transmitted. In that case, CANFDTX attempts to send the frame again when the bus becomes idle until it is successfully transmitted. If an error occurs during transmission, data FIFO is reset so that the FIFO becomes empty for a fresh transmission, the current transmission is abandoned and transmission is indicated as over. Since the related **TRR** bit is not cleared, TXCL will attempt to transmit the frame again when Binary Search Algorithm decides. TX Error Detection Block (TXED) provides error data to **TMSR**. If no error occurred, **TMSR** is also updated to indicate no error occurred. TXED checks for errors such as bit error and ack error and ID information is attached to the error type to form the **TMSR**. **TMSR** content

can be seen in Fig.4.4.

4.5 Hardware Blocks: Receiver Module

Most functionality of the Receiver Module sub-blocks are either analogous or reversing the actions taken by the corresponding blocks of the Transmitter Module.

RX Control Logic Block (RXCL):

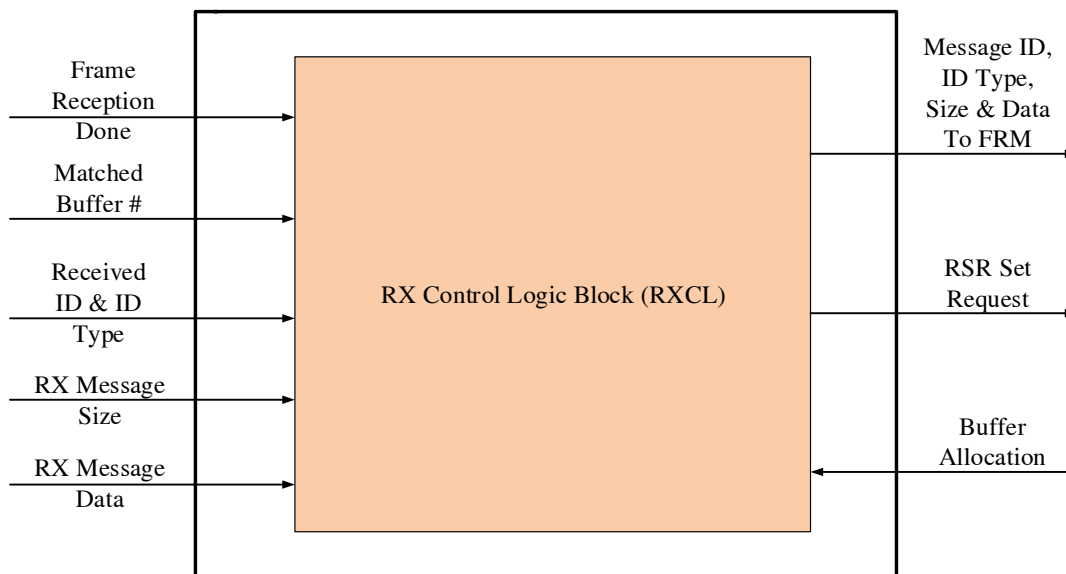


Figure 4.11: RX Control Logic Block (RXCL)

When a message is received successfully, RXCL gets frame reception done information with the buffer number information which comes from MF. Having the buffer number information means knowing the address of the related buffer in FRM where the data will be written. RXCL reads the ID and ID type from CANFDRX, payload size and data from FIFOs of CANFDRX. This data is written to the related address of FRM. Furthermore, **Receive Status Register (RSR)** is requested to be set. If the bit of **RSR** where it is requested to be set is already set, overflow condition occurs. New data is over written hence the buffer holds the new frame data and overflow interrupt is generated if enabled. There are total of 3 **RSR** for 96 RX buffers. **RSR** has the information of the buffer numbers which hold the received messages. This is an indication for the Host MCU to learn the buffers with the received frames. When the host reads the messages from the related buffers, it requests to clear **RSR**. The content of

RSR is shown in Table 4.5 The block diagram of RXCL can be seen in Fig.4.11

CAN FD Receiver-RX Block (CANFDRX)

The block diagram of CANFDRX can be seen in Fig.4.12. CANFDRX receives the frames that are formatted according to CAN FD standard. It performs the CRC control and other checks on the received CAN FD frames. When a message is successfully received and matches any one of the filters, the size of the payload is written to RX Size FIFO and the payload is written to RX Data FIFO. Frame reception done information is provided to indicate that a message is received and ready to be read from the FIFOs. If an ID match not found signal is received, after the whole frame is received, the FIFOs are reset and the frame reception done is not provided. In other words, the received frame is discarded.

CRC Calculator Block calculates either 17 bit or 21 bit CRC depending on the message payload size as defined in the standard. If an error occurs during reception, current reception is abandoned, the data and size FIFOs are reset so that the FIFOs become empty for a fresh reception, and the bus is waited to be idle for a new reception. RX Error Detection Block (RXED) provides error data to **Receive Message Status Register (RMSR)**. If no error occurs, **RMSR** is also updated to indicate no error occurred. RXED checks for errors such as bit error, stuff error, CRC error and form error and the ID information is attached to the error type to form **RMSR**. RMSR content can be seen in Fig.4.5.

Furthermore, this block has an interface with the physical layer CAN FD signals, namely, TX and RX. The block diagram of CANFDRX can be seen in Fig.4.12. Since CAN FD has to send an acknowledgment bit during reception, it has an access to TX physical layer signal as well.

Message Filter Block (MF):

MF block diagram is shown in Fig.4.13. MF gets the ID and ID type (either base or extended) along with the ID ready information. Filter Control block inside MF gets programmed ID and Mask pairs of each RX buffers from FMB and feeds this data to Comparator block one by one. Comparator block compares the mask applied received message ID with the mask applied ID data that come from FMB for each of the 96

Table4.5: RX Control Register Set

| | Address (0x) | Register Name | Read (R)/ Write (W) | Description |
|------------------------------------|--------------|-------------------------------|------------------------|---|
| Receive Control Register Set | 18 | Receive Status Register 1 | R/W | When a message is received, core updates the related bit as 1 When the host reads a message from any buffer, it should write 1 to the related bit to clear Only one bit should be cleared at a write, Dont perform another read before clearing related rsr bit Register[31] => Buffer 32 message reception status Register[30] => Buffer 31 message reception status Register[0] => Buffer 1 message reception status |
| | 1C | Receive Status Register 2 | | |
| | 20 | Receive Status Register 3 | | |
| | 24 | Receive Error Status Register | R | Register [31:3] => 29 Bit ID of the last reception Register[2] => ID Type of the frame, 0 => Base ID, 1 => Extended ID Register[1:0] => 2 bit code of the last reception Status Register[1:0] => "00" => No Error Register[1:0] => "01" => Stuff Error Register[1:0] => "10" => CRC Error Register[1:0] => "11" => Form Error |

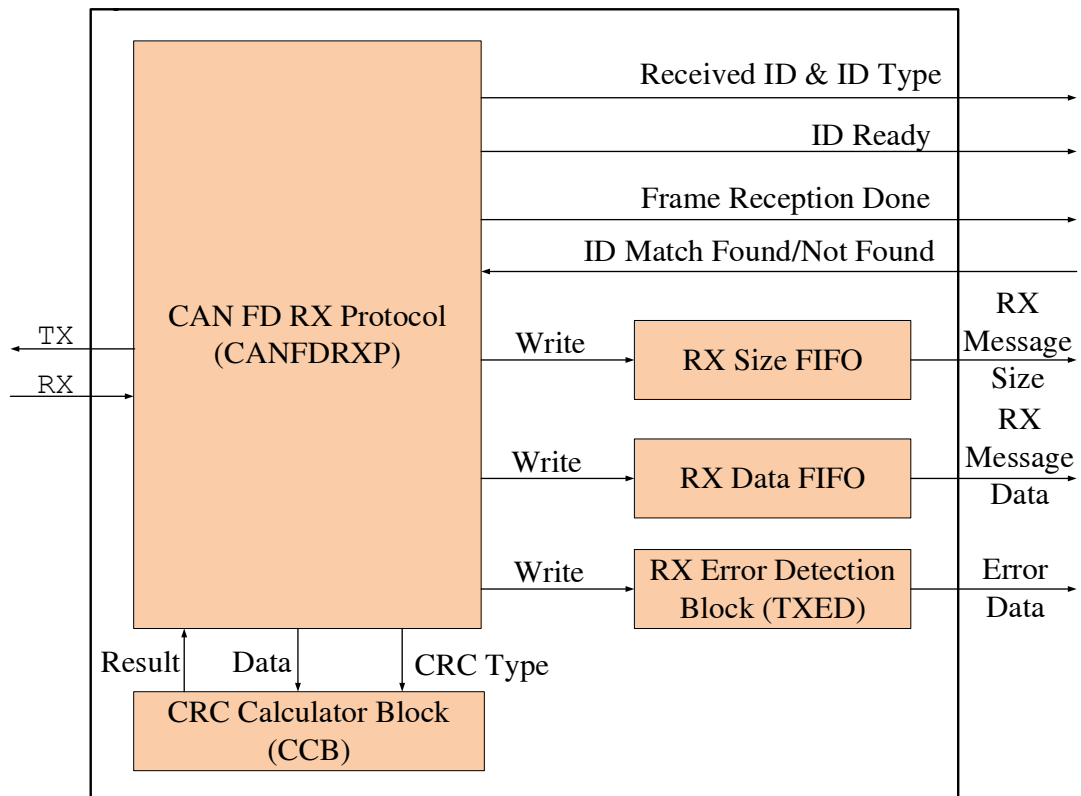


Figure 4.12: CAN FD Receiver-RX Block (CANFDRX)

RX buffers' programmed ID & Mask pair. If it finds a match, it indicates that ID match is found and provides the buffer number whose filtering is passed. If a match is not found for any one of the 96 RX buffers, it is indicated that ID match is not found. This means that the received frame does not pass the filtering. The messages that do not pass any of the RX filters are discarded. Messages may pass more than one filter, for this case, the first filter is taken into account and the buffer number is provided accordingly. The filtering process begins when ID ready information is received from CANFDRX, that is, during the reception process, MF does not wait for whole frame to be received, instead, only receiving the ID part of the frame is enough to perform the filtering operation. Therefore, by the time the frame is received, filtering process is already finished. For the worst case (a match is found at 96th buffer or no match is found after 96 steps), the filtering takes 496 clock cycles, which is approximately 5 μ s. For 2 Mbit/s bit rate, this time corresponds to 10 bits time for a CAN FD frame.

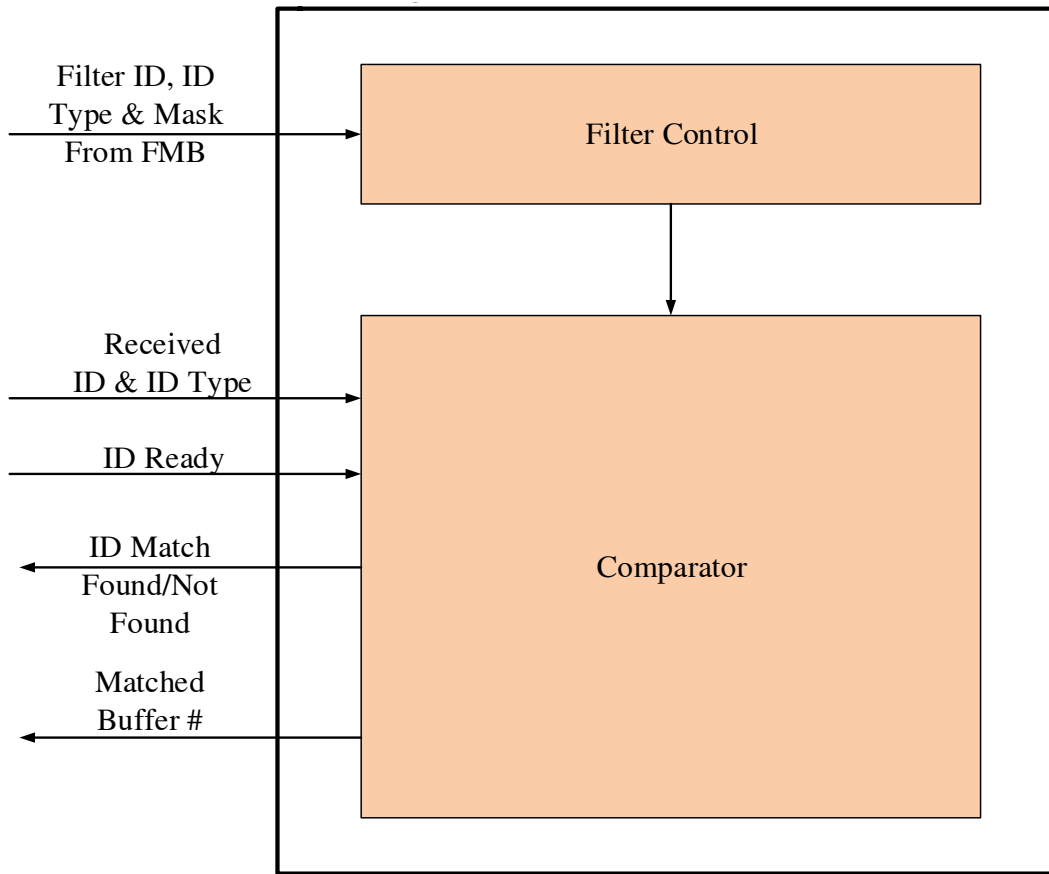


Figure 4.13: Message Filter Block (MF)

4.6 Configuration Phase

The C^3 driver software that should run on the MCU executes the configuration process of the controller before the main application of the host MCU starts. The registers requiring configuration on MMR are programmed by the C^3 driver software. For this purpose, for each TX buffer, the size of the payload in the **DLC Registers** are configured over SPI. Accordingly, BCB in MMR allocates and assigns the required memory for the payload of each transmitted CAN FD message on PFTM. Similarly for each RX buffer, **DLC Registers** are configured by the driver over SPI. To this end, BCB in MMR allocates and assigns the required memory space on FRM according to the payload of each CAN FD message to be received. After both of the buffers are configured, BCB state machine stays in a dead state and reprogramming of the buffers is not possible. In other words, when the Host MCU application begins running, the buffers can not be configured again. **ID Registers** in TX Register sets are configured

for the buffers which are going to be used. Similarly, **RX ID** and **RX Mask Registers** are programmed. This process is also called the initialization of the controller.

TX(ID) and **RX(ID)** register sets can be seen in Table 4.6 and Table 4.7. An example configuration can be seen below:

- **TX ID Register:** For 29 bit extended ID of 03A66A30 for TX Buffer 1, 0x1D335184 is written to this register at 0x0100 address
- **TX DLC Register:** For 64 Byte payload configuration for TX Buffer 1, 0x00000040 is written to this register at 0x0104 address
- **RX ID Register:** For 11 bit Base ID of 081 for RX Buffer 1, 0x10200000 is written to this register at 0x1000 address
- **RX DLC Register:** For 32 Byte payload configuration for RX Buffer 1, 0x00000020 is written to this register at 0x1004 address
- **RX Mask Register:** To filter out the messages which do not have the same first most significant 9 bit ID bits for RX Buffer 1, 0xFF800000 is written to this register at 0x1008 address

4.7 Data Phase and Timing

The Data Phase consists of message transmit and receive actions for the applications that run on the host MCU. Here we note that the clock cycle for C^3 is denoted as $c_{C^3} = 10$ nsec while the SPI clock cycle is denoted as $c_{SPI} = 100$ nsec [22].

When the application has data to send, the driver locates the corresponding **TX(ID)** register, the address of the register. Then, the driver writes the payload data on the **Data Register** of **TX(ID)** using the Burst SPI Write command which takes

$$8 + 16 + 8 + B \cdot 8 c_{SPI} = 3.2 + 0.8 \cdot B \mu s \text{ for } B \text{ byte payload.}$$

The **Data Register** of the first TX buffer can be seen in Fig.4.6

Table4.6: TX(ID) Register Set

| | Address (0x) | Register Name | Read (R)/ Write (W) | Description |
|--|--------------|------------------|------------------------|---|
| Transmit Buffer 1 Register Set TX(ID) | 100 | TX ID Register | W | Register [31:21] => 11 bit Standard Message ID, ID[28:18] field of CAN FD Frame |
| | | | | Register [20:3] => Extended Message ID. ID[17:0] field of CAN FD Frame, valid for only extended CAN FD frames |
| | | | | Register [2] => IDE, Identifier Extension Flag field of CAN FD Frame 1 => indicates Extended Message Identifier 0 => indicates Standard Message Identifier |
| | 104 | TX DLC Register | W | Register[1:0] => Reserved |
| | | | | Register[31:7] => Reserved |
| | 108 | TX Data Register | W | Register[6:0] => Data Length of CAN FD Frame, in terms of bytes (1 to 64) Data Field of CAN-FD Transmit Buffer |

Table4.7: RX(ID) Register Set

| | Address (0x) | Register Name | Read (R)/ Write (W) | Description |
|---|--------------|------------------|------------------------|---|
| Receive Buffer 1 Register Set RX(ID) | 1000 | RX ID Register | R/W | Register [31:21] => 11 bit Standard Message ID, ID[28:18] field of CAN FD Frame |
| | | | | Register [20:3] => Extended Message ID. ID[17:0] field of CAN FD Frame, valid for only extended CAN FD frames, Dont care for base ID |
| | | | | Register [2] => IDE, Identifier Extension Flag field of CAN FD Frame, this bit is only for the received messages, i.e. Read only 1 => indicates Extended Message Identifier 0 => indicates Standard Message Identifier |
| | | | | Register [1:0] => Reserved |
| | 1004 | RX Mask Register | W | Register [31:21] => 11 bit Standard Message ID Mask, ID[28:18] field of CAN FD Frame Mask, 1=> must comply, 0 => dont care |
| | | | | Register [20:3] => Extended Message ID Mask. ID[17:0] field of CAN FD Frame, valid for extended CAN FD frames, Write 0 for Base ID 1=> must comply, 0 => dont care |
| | | | | Register [2] => Reserved and it is '1' |
| | | | | Register [1:0] => Reserved |
| | 1008 | RX DLC Register | W | Register[31:9] => Reserved |
| | | | | Register[8] => EDL, Extended Data Length field of CAN FD Frame 1 => Frame is in CAN FD frame format 0 => Not Applicable |
| | | | | Register[7] => BRS, Bit Rate Switch field of CAN FD Frame 1=> Bit rate is switched to 2 Mbps in data phase of CAN FD Frame 0=> Not Applicable |
| | | | | Register[6:0] => Data Length of CAN FD Frame, in terms of bytes (1 to 64) |
| | 100C | RX Data Register | R | Data Field of CAN-FD Receive Buffer |

When the data write is complete, the MCU asserts bit(s) in the **Transmit Request Register (TRR)** enabling the indicated TX Buffers for transmission, which takes

$$(8 + 16 + 32) c_{SPI}$$

There are two methods for transmission

- Host MCU can write the payload of a frame to data register of a single buffer first then set only one bit of **TRR** then repeat this process for each of the buffers that are desired to be transmitted.
- Host MCU can write the payload of the frames to data registers of multiple buffers first, then set multiple bits of **TRR** at a single write.

The host MCU may program the **Interrupt Enable register (IER)** to get notified when a transmit error occurs. If an interrupt is received, the host MCU reads the **Interrupt Status Register (ISR)** to determine the source of the interrupt. It then reads the **Transmit Message Status Register (TMSR)** for the error source and the ID of the message. Finally, the MCU clears the **ISR** after handling the interrupt. C^3 polls **TRR** for pending transmission requests. If there are any bits set in **TRR**, C^3 selects the message with the highest priority and begins transmitting. It takes $6 \cdot c_{C^3} = 60$ nsec to determine the message with the highest priority, which is implemented as binary search algorithm. If no errors occur during the transmission, the C^3 clears the related bit of the **TRR**. In case of an error, C^3 determines the cause of the error and updates the **TMSR** with the ID of the message and the error type. C^3 retransmits the message if the previous attempt has error [22].

Total transmission delays can be formulated as below [22]. In the formula below, x denotes the FPGA core delay and it is approximately $1 \mu s$ for $B = 64$. This value completely depends on the implementation and it is not possible to estimate the value theoretically instead practical value obtained from real hardware implementation is used. More information can be found in Sec.5.5.

$$T_{TX(B)} = x + 3.2 + 0.8 \cdot B \mu s + 64 c_{SPI} \approx (4.2 + 0.8 \cdot B + 64 \cdot c_{SPI}) \mu s.$$

In our implementation, the significant C^3 core transmit delays add up to less than $62 \mu s$ using the formula above and taking $B = 64$. This time is very small compared

to expected application delays and jitters on the MCU which should be bounded by the message periods. Here we note that the lowest message periods in vehicle applications is close to 5 ms [22].

Regarding the reception, the host MCU polls the **Receive Status Register (RSR)** or enables the interrupt. If there is a new message, the corresponding bit to the buffer ID is set in one of the three **RSRs**. The MCU reads all of the three **RSR** and determines the buffer holding the received message. The corresponding **RX(ID)** data register is read using the Burst SPI Read command. The MCU is notified with an interrupt with the source of error indicated in the **ISR** if an error occurs. After successful reception or after an error, the related bit of **RSR** which corresponds to the read message is cleared. Messages with an error are discarded [22].

A frame with CAN FD ID is first fully received by the CANFDRX and then placed in the **RX(ID)**. Hence, C^3 and the host MCU have to finish processing the frame and the MCU has to complete reception before the following frame is ready at CANFDRX. In our current implementation, the MCU and C^3 Driver software is emulated with a hardware block on the FPGA for evaluation purposes. To this end, we estimate the receive processing time of C^3 without driver and MCU overhead. Furthermore, we assume that all CAN FD frames with a given ID have the same payload length of B Bytes. We also assume that the CRC sequence is 21 bits for all frame sizes, whereas it is 17 bits for frames with payload less than 20 bytes. The transmission at standard CAN rate is 1 Mbps, CAN FD Data rate is 2 Mbps [22].

We denote the shortest time between two consecutive CAN FD transmissions for a CAN FD frame with B byte payload by $T_{Bus(B)}$ which consists of the time to transmit a frame and the inter-frame gap of $3\mu s$. During frame transmission, SOF and arbitration fields (BASE ID + SRR + IDE + EXTENDED ID + r1 + EDL + r0: $29 + 6 = 35$ bits) at the beginning of the frame and ACK+EOF ($3 + 7 = 10$ bits) at the end of the frame are transmitted at 1 Mbps. Control Field (BRS+ESI + DLC: 6 bits), Payload Data ($B \cdot 8$ bits), CRC + CRC delimiter ($21 + 1 = 22$ bits) are transmitted at 2 Mbps. Hence, $T_{Bus(B)} = (59 + B \cdot 4 + 3) \mu s = (B \cdot 4 + 62) \mu s$ [22].

We next compute $T_{RX(B)}$ which denotes the estimated time for the host MCU to receive a frame with B bytes payload from C^3 . The SPI cycle times are computed

according to Table 4.2. Reception of the frame starts by writing B bytes in FRM. Then **ISR** is updated and an interrupt is generated for the host MCU which takes y . The host MCU gets the interrupt and reads the **ISR** with the 32 bit SPI Read operation ($56 \cdot c_{SPI}$). Then, the host MCU reads 3 **RSR**, where each register has 32 bits to represent 96 RX buffers ($3 \cdot 56 \cdot c_{SPI}$). The host reads the **Data Register** of **RX(ID)** with Burst SPI Read ($32 + B \cdot 8 \cdot c_{SPI}$) and clears ISR with a 32 bit SPI Write ($56 \cdot c_{SPI}$). Finally the operation is completed by clearing **RSR** with a 32 bit SPI Write ($56 \cdot c_{SPI}$) [22].

Accordingly, $T_{RX(B)} = y + (368 + B \cdot 8) \cdot c_{SPI} = (y + 36.8 + B \cdot 0.8) \mu s \approx (B \cdot 0.8 + 38.2) \mu s$ where y is approximated as 1.4μ for $B = 64$. Theoretical estimation of y is not possible because it depends on the hardware implementation hence the practical value from the real hardware is used here. More information can be found in Sec.5.5.

Hence, under our assumptions, $T_{RX(B)}$ is significantly smaller than $T_{Bus(B)}$. That is, C^3 and the host MCU indeed complete processing each frame before the next frame can be transmitted[22].

4.8 FPGA Implementation Results

Xilinx Virtex 5 FPGA is used for the implementation. More information about the FPGA is provided in Sec.5.1 FPGA device utilization and project status are depicted in Table 4.8 and Table 4.9. FPGA code includes both the application and CAN FD Controller. The application acts like a host MCU simulator. More details are given in Sec.5.2. Clock frequency is **100 MHz**.

SPI Data is sampled at the rising edge and put on the line at falling edge of SPI CLK signal. Therefore, effective SPI clock frequency seen by the FPGA is **20 MHz**. **100 MHz** FPGA clock frequency is suitable enough to handle effective **20 MHz** SPI clock frequency.

Xilinx FIFO, Block Memory Generator and PLL IP Cores are used. All other modules including CAN FD protocol and SPI protocol are written in VHDL manually and implemented.

Table4.8: FPGA Device Utilization Summary


| Device Utilization Summary | | | |  |
|---|--------|-----------|-------------|---|
| Slice Logic Utilization | Used | Available | Utilization | Note(s) |
| Number of Slice Registers | 5,029 | 44,800 | 11% | |
| Number used as Flip Flops | 5,029 | | | |
| Number of Slice LUTs | 11,541 | 44,800 | 25% | |
| Number used as logic | 11,331 | 44,800 | 25% | |
| Number using O6 output only | 9,088 | | | |
| Number using O5 output only | 854 | | | |
| Number using O5 and O6 | 1,389 | | | |
| Number used as exclusive route-thru | 210 | | | |
| Number of route-thrus | 1,098 | | | |
| Number using O6 output only | 1,064 | | | |
| Number using O5 output only | 34 | | | |
| Number of occupied Slices | 4,522 | 11,200 | 40% | |
| Number of LUT Flip Flop pairs used | 13,250 | | | |
| Number with an unused Flip Flop | 8,221 | 13,250 | 62% | |
| Number with an unused LUT | 1,709 | 13,250 | 12% | |
| Number of fully used LUT-FF pairs | 3,320 | 13,250 | 25% | |
| Number of unique control sets | 494 | | | |
| Number of slice register sites lost to control set restrictions | 867 | 44,800 | 1% | |
| Number of bonded IOBs | 6 | 640 | 1% | |
| Number of LOCed IOBs | 6 | 6 | 100% | |
| Number of BlockRAM/FIFO | 14 | 148 | 9% | |
| Number using BlockRAM only | 14 | | | |
| Number of 36k BlockRAM used | 6 | | | |
| Number of 18k BlockRAM used | 10 | | | |
| Total Memory used (KB) | 396 | 5,328 | 7% | |
| Number of BUFG/BUFGCTRLs | 2 | 32 | 6% | |
| Number used as BUFGs | 2 | | | |
| Number of PLL_ADVs | 1 | 6 | 16% | |
| Average Fanout of Non-Clock Nets | 4.28 | | | |

Table4.9: FPGA Project Status

| TopModule Project Status (11/11/2017 - 12:02:21) | | | |
|--|---|------------------------------|---|
| Project File: | CAN_FD.xise | Parser Errors: | No Errors |
| Module Name: | TopModule | Implementation State: | Placed and Routed |
| Target Device: | xc5vfx70t-1ff1136 | • Errors: | |
| Product Version: | ISE 14.7 | • Warnings: | |
| Design Goal: | Balanced | • Routing Results: | All Signals Completely Routed |
| Design Strategy: | Xilinx Default (unlocked) | • Timing Constraints: | All Constraints Met |
| Environment: | System Settings | • Final Timing Score: | 0 (Timing Report) |

Design module hierarchy is in Fig.4.14. The functions of each file is listed below:

- CANFD_Controller.vhd: module includes Memory Mapped Register Block (MMR), RX Control Logic (RXCL), Message Filter (MF), TX Control Logic (TXCL) and Interrupt Control Blocks
- CANFD_Receiver.vhd: Module includes CAN FD RX Block (CANFDRX)
- CANFD_Transmitter.vhd: Module includes CAN FD TX Block (CANFDTX)
- CRC_17.vhd: 17 bit CRC Calculator Block (CCB)
- Crc_gen_21bit.vhd: 21 bit CRC Calculator Block (CCB)
- SPI_Controller.vhd: Module includes SPI Protocol Control Block (SPCB)
- Host_Application.vhd: Module includes Host Simulator
- SPI_Master_Controller.vhd: Module includes Master SPI Control Block (MSPIC)
- TopModule.vhd: Provides connection between CAN FD Controller and host simulator.
- Debouncer.vhd: Provides debouncer for button press on FPGA demoboard.
- CANFD_Constraints.ucf: Includes the timing and pin location constraints
- PLL.xav: 100MHz clock is put into PLL to generate a clock with less jitter and more stability.

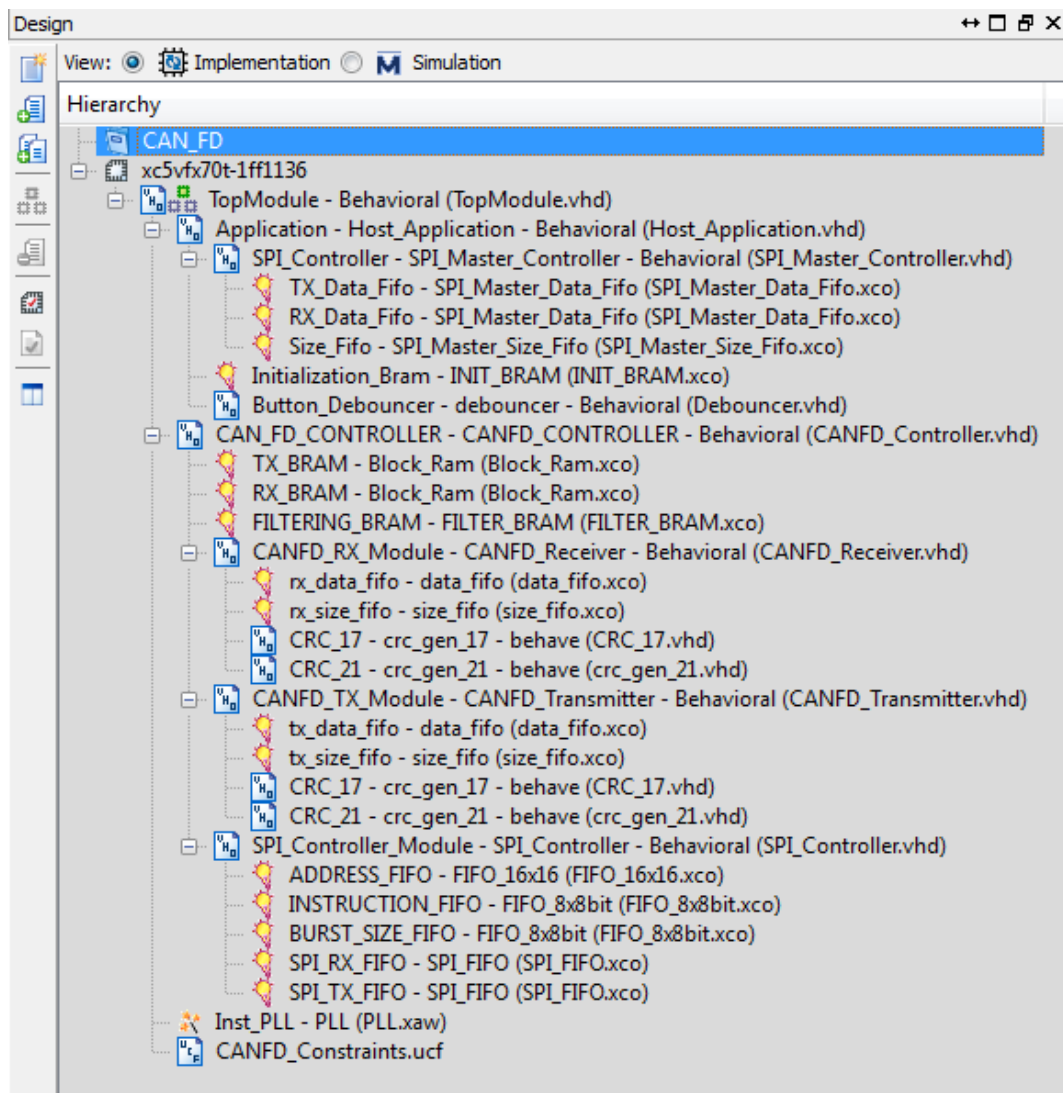


Figure 4.14: FPGA Project Hierarchy

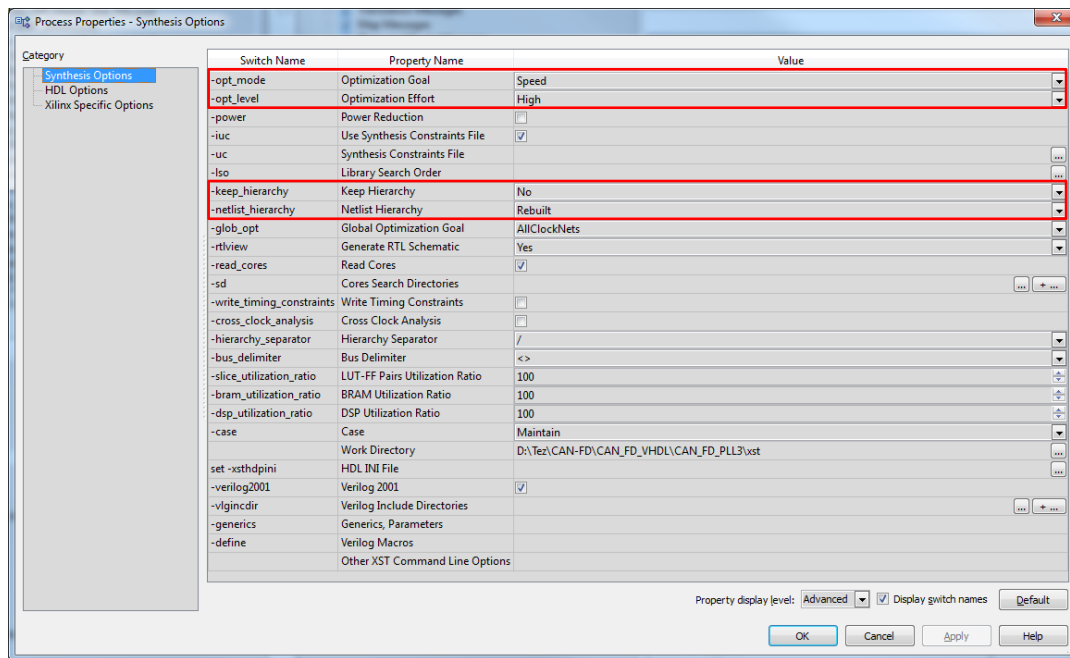


Figure 4.15: FPGA Implementation Options 1

One of the biggest challenges of the implementation was to meet the timing. Since there are so many registers and internal signals for 96 RX and 96 TX buffers, the design didn't meet the timing at the first trial. It is not possible to reduce **100 MHz** FPGA clock frequency due to effective **20 MHz** SPI Clock frequency. Therefore, to improve timing two strategies have been developed:

There are many synthesis, HDL, Xilinx Specific, Map, Place & Route options. By changing the options, its possible to have better timing performance. For this purpose, the following settings in Fig.4.15, Fig.4.16, Fig.4.17, Fig.4.18 and Fig.4.19 are applied.

The other strategy is to increase the steps to perform the memory mapping operation. Since the number of the registers is high for 96 TX and RX buffers. There are also much more signals related to the buffers than the buffers' own registers, hence huge cascaded multiplexers have to be used during implementation. Cascaded structures are the enemies of timing since the signals are serialized and serialized operation must be completed in a single clock cycle. For this purpose, configuration mapping is done in 12 steps, SPI writes are done in 15 steps and SPI reads are done in 12 steps. Since SPI reads require quick response, steps to perform register read requests must

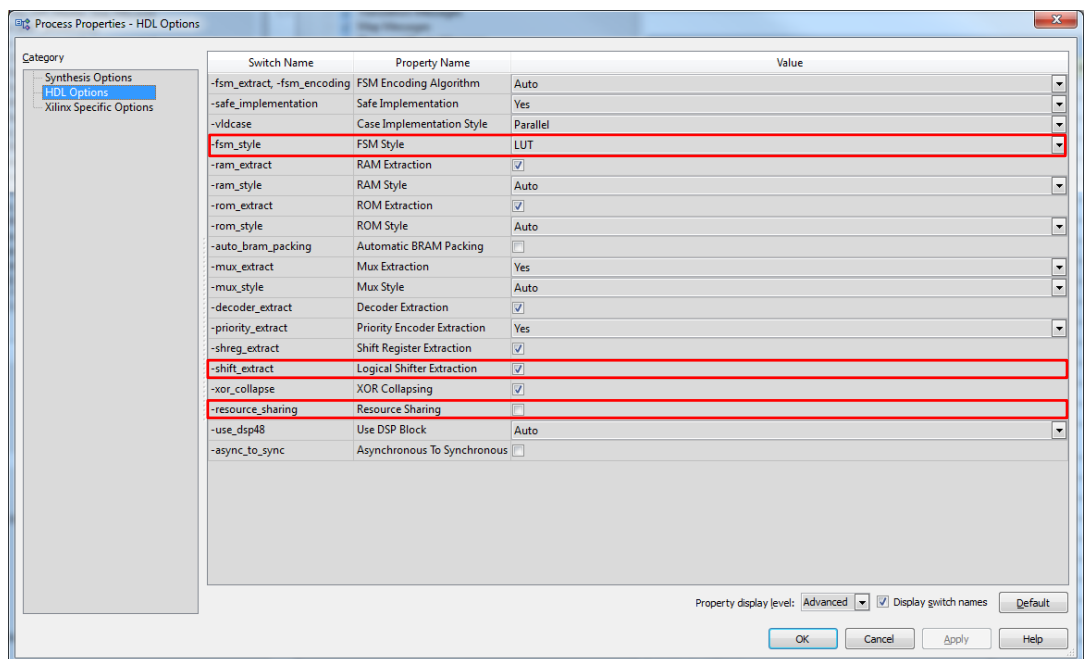


Figure 4.16: FPGA Implementation Options 2

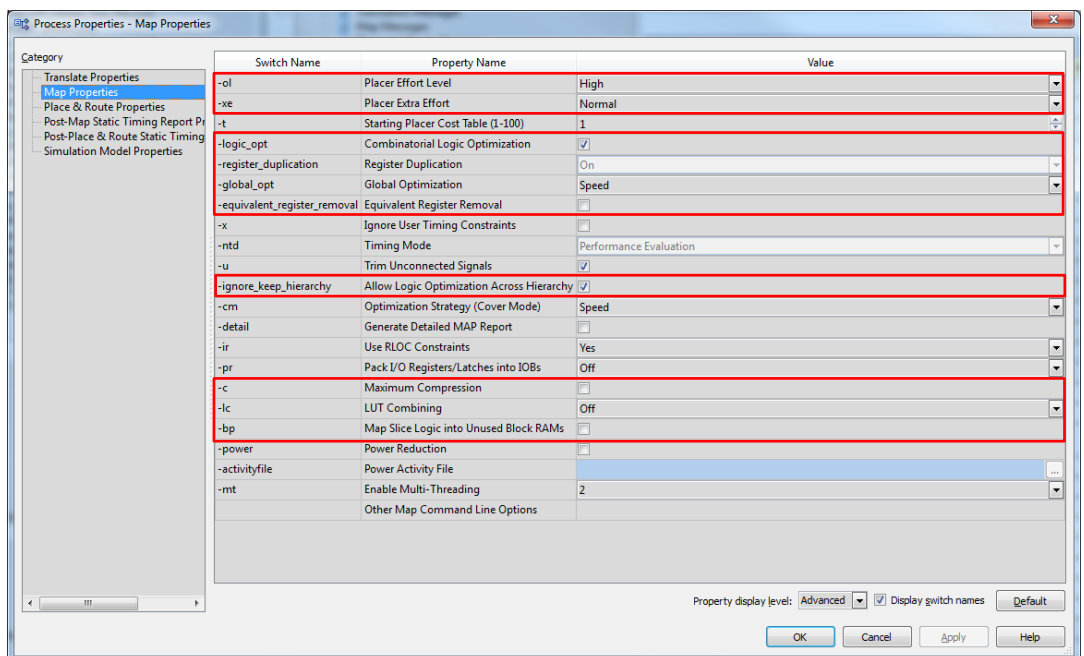


Figure 4.17: FPGA Implementation Options 3

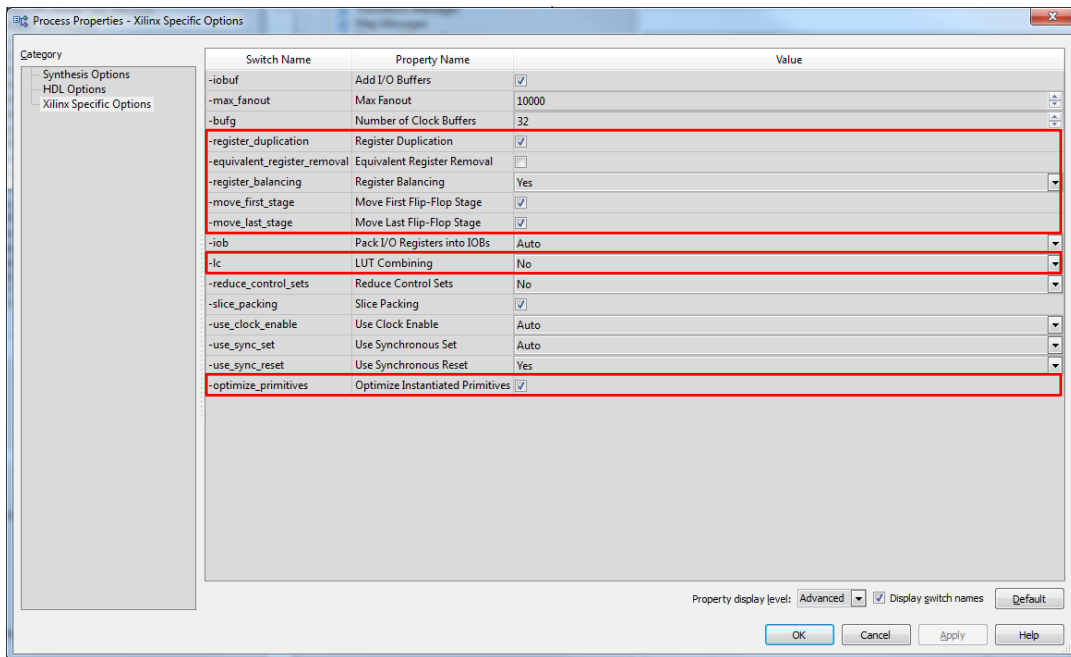


Figure 4.18: FPGA Implementation Options 4

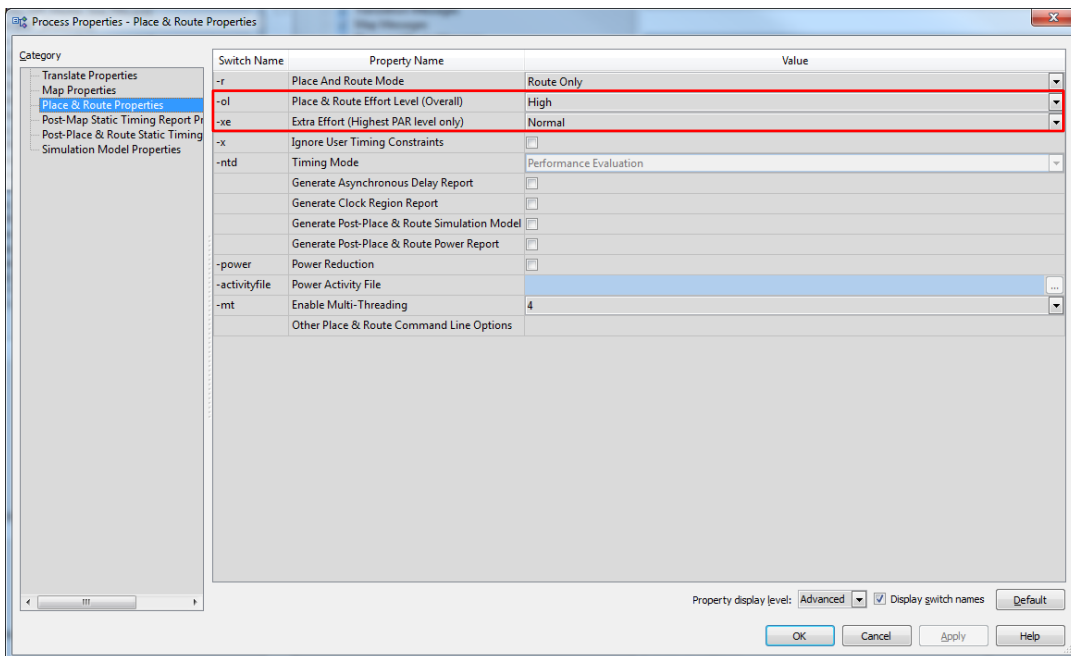


Figure 4.19: FPGA Implementation Options 5

be completed fast. 12 steps are quick enough for CAN FD Controller to perform the mapping and put the required data on MISO line on time. By increasing the mapping steps, cascaded multiplexers are divided into smaller cascaded units hence being able to complete the operation on time.

By these two adjustments, timing score was improved greatly and could be pulled down to 0 and timing is met. Furthermore, besides the general VHDL coding techniques, following strategies are followed for more optimized implementation.

Using Case Statements Instead of Nested If Else Statements: Case statements have been used for all of the state machines and most of the conditional logic having more than two conditions. Use of case statement yields faster implementation. Nested if else structure uses priority encoders for implementation therefore more combinational logic is used hence the hardware becomes slower.

Arithmetic Operations: Instead of using multipliers or dividers for some arithmetic operations, following strategies are used:

- For multiplying with 2, shifting the logic vectors one bit left
- For dividing by 2, shifting the logic vectors one bit right

Therefore, by using simple shift registers, usage of complex multipliers or dividers are avoided. This yields lower resource consumption.

Furthermore, grouping arithmetic statements gives great timing advantage. For example, the first statement below is implemented as 4 serialized adders while the second statement below is implemented as 2 parallel adders. The latter one gives better timing performance.

- $C \leq C1 + C2 + C3 + C4$
- $C \leq (C1 + C2) + (C3 + C4)$

Using Constants: Use of constant statement instead of vector or integer variables provides more optimized implementation.

Avoiding Latches: Latches are created when incomplete if-else statement is used. In other words, “if” part of the condition is specified but “else” part is not specified. If the target FPGA does not have latch units, latches are created from multiplexers and some other logic elements hence increasing resource consumption and hence reducing timing performance.

Using Synchronous Reset: Most FPGAs have logic elements with synchronous reset therefore when asynchronous reset is used, some extra logic elements are used to implement asynchronous structure. To minimize resource consumption, a reset signal synchronous to the clock signal is used throughout whole design.

Using Distributed or Block Memory: Instead of using huge number of registers for memory purposes, it is good practice to use embedded memory. Because embedded memory only uses the existing memory block inside the FPGA. Using embedded memory makes the design quite simple and more efficient.

State Machine Implementation Types: State machine implementations are categorized in several ways. Some of them require encoding/decoding while some does not require. For example, the sequential state machine implementation type performs binary encoding for state mapping. It uses less logic resource but since it requires encoding/decoding, it is slow and the timing performance is bad. On the other hand, one hot encoding requires no encoding/decoding since each state is represented by a single flip flop. Therefore one hot encoding consumes more resources but it is fast and yields better timing performance. For this design, the choice is left to the program automatically but it might also have been forced to one hot encoding for good timing.

Using Synchronous Processes: Using synchronous processes and avoiding combinational logic as much as possible yields better timing performance. Combinational logic is the enemy of the timing. For this implementation, mostly synchronous logic is used as much as possible.

CHAPTER 5

EVALUATION OF C^3 (CONFIGURABLE CAN FD CONTROLLER)

In this chapter, the development environment, the Host Simulator implementation details, response time measurements and functional tests are covered in detail. The purpose is to verify the design functions and measure the response time performance of the CAN FD Controller.

5.1 Development and Test Environment

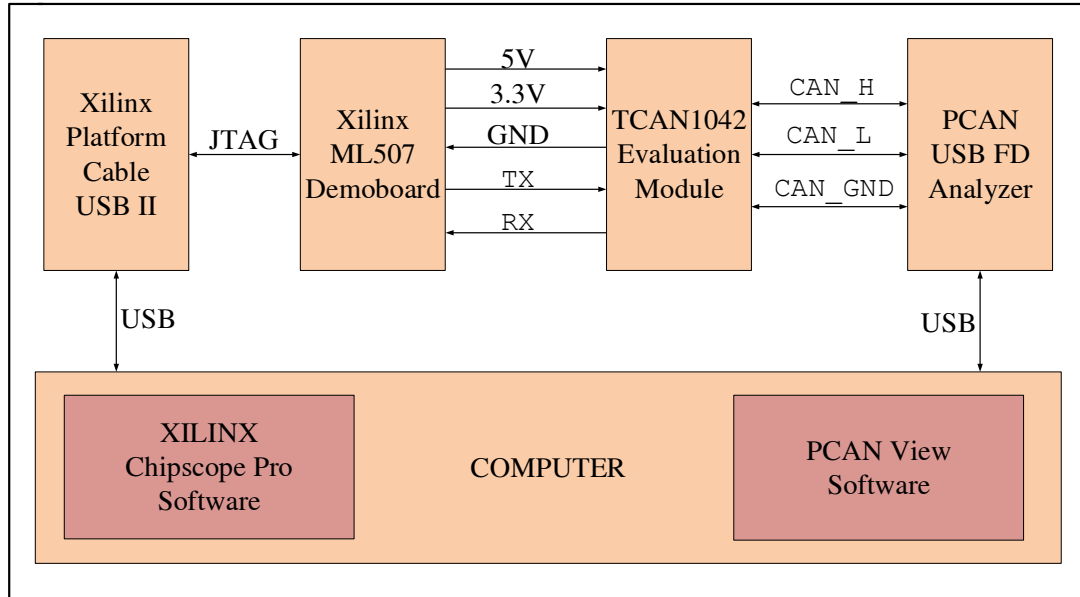


Figure 5.1: Test Setup Block Diagram

The test setup block diagram is depicted in Fig.5.1. We implement our C^3 CAN FD Controller and a host simulator on the FPGA demo board. CAN FD transceiver module is connected to FPGA demo board to meet the physical layer requirements of CAN FD protocol. The transceiver module connects C^3 CAN FD Controller to CAN

FD Bus. PCAN analyzer is connected to CAN FD Bus to send CAN FD frames to the controller and monitor the bus for the frames that the controller sends.

Xilinx Platform Cable USB II is connected to FPGA demo board via USB for two purposes. One is to program the FPGA. The other is to monitor the FPGA signals to verify the functions.

Xilinx ISE 14.7 development platform is used for FPGA hardware design. Code is written in VHDL. Synthesizing and hardware implementation of the design is done with this platform. Synthesizing and hardware implementation of VHDL code is dependent on the FPGA used. Therefore, Xilinx ISE platform is used.

Chipscope Pro software tool is used to debug and verify the design.

The test setup can be seen in Fig.5.2.

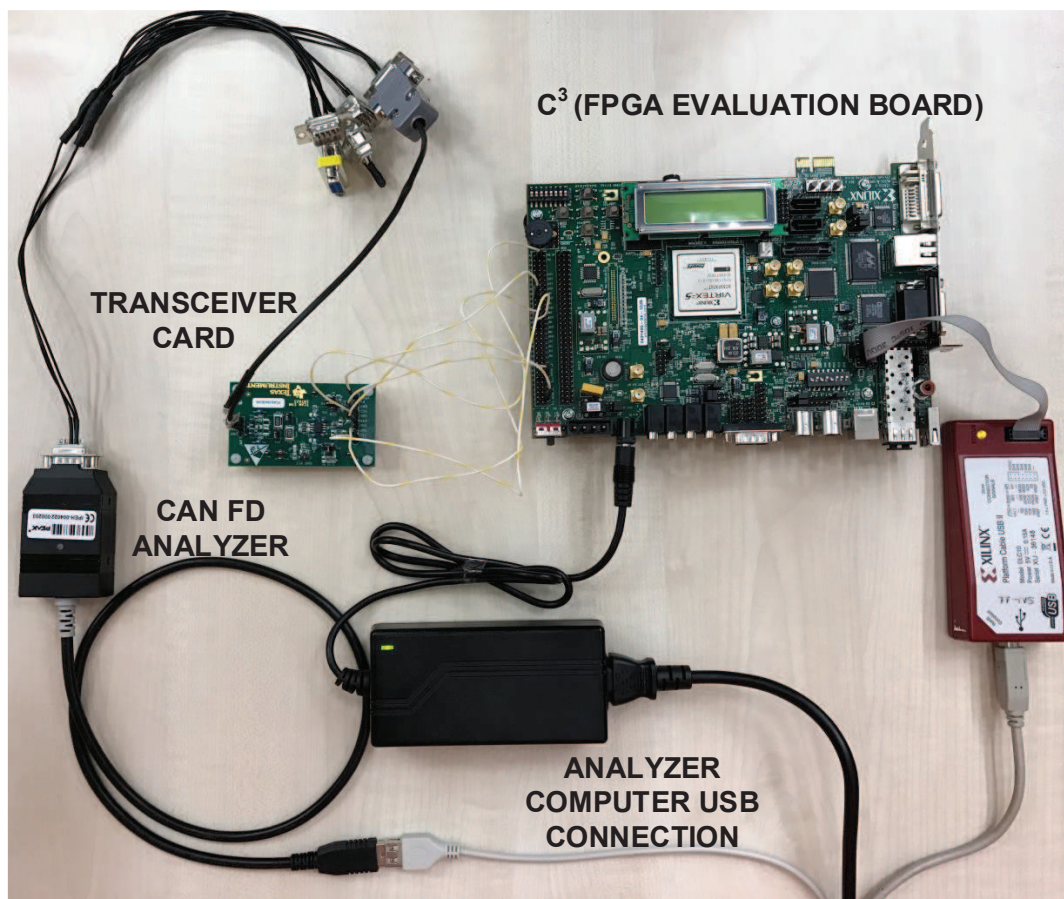


Figure 5.2: Test Setup

FPGA Demoboard: CAN FD Controller is implemented in VHDL and realized in

Xilinx Virtex5 ML507 demo board [15]. The pictures of the board can be seen in Fig.5.3

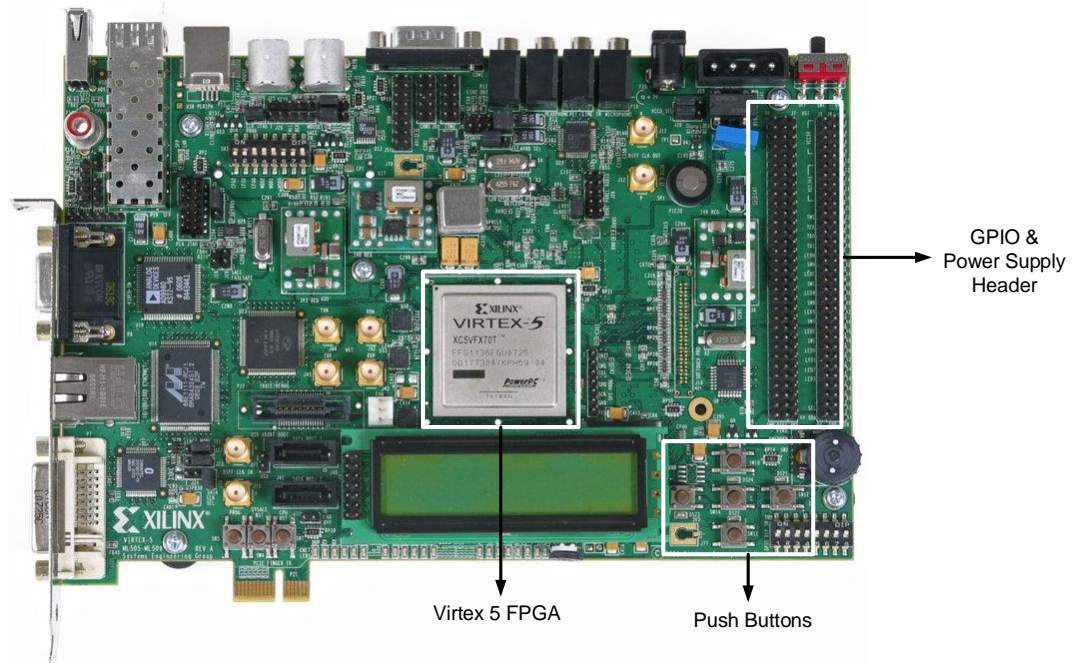


Figure 5.3: Xilinx ML507 Demoboard

Virtex 5 FPGA with the part number of XC5VFX70T-1FFG1136 has the following logic, memory and interface capacities which can be seen in Fig.5.1. The comparison of the FPGA used with the other FPGAs belonging to Virtex-5 family is shown also in Table 5.1.

Table5.1: Xilinx Virtex 5 FPGA Family Comparison

| Device | Configurable Logic Blocks (CLBs) | | | DSP48E Slices ⁽²⁾ | Block RAM Blocks | | |
|------------|----------------------------------|--------------------------------|--------------------------|------------------------------|----------------------|-------|----------|
| | Array (Row x Col) | Virtex-5 Slices ⁽¹⁾ | Max Distributed RAM (Kb) | | 18 Kb ⁽³⁾ | 36 Kb | Max (Kb) |
| XC5VFX30T | 80 x 38 | 5,120 | 380 | 64 | 136 | 68 | 2,448 |
| XC5VFX70T | 160 x 38 | 11,200 | 820 | 128 | 296 | 148 | 5,328 |
| XC5VFX100T | 160 x 56 | 16,000 | 1,240 | 256 | 456 | 228 | 8,208 |
| XC5VFX130T | 200 x 56 | 20,480 | 1,580 | 320 | 596 | 298 | 10,728 |

Some important features of the demo board is as follows:

- Two Xilinx XCF32P Platform Flash PROMs (32 Mb each) for storing large device configuration
- 100 MHz oscillator
- General purpose LEDs, pushbuttons
- Expansion header with 32 single-ended I/O, 16 LVDS-capable differential pairs, 14 spare I/Os shared with buttons and LEDs, power, JTAG chain expansion capability, and I2C bus expansion

CAN FD Transceiver: The signals named as **TXD** and **RXD** shown in Fig.5.4 are the TTL level single ended signals connected to CAN FD Controller. **CANH** and **CANL** are the differential ended signals connected to CAN FD bus.

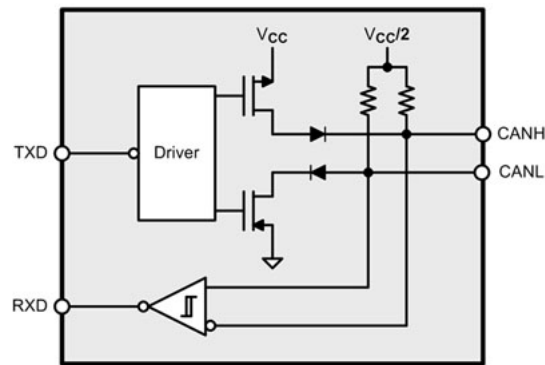


Figure 5.4: CAN FD Transceiver Internal[31]

Transceivers contain both the transmitter and the receiver in a single chip. There are open-drain output transistors with internal pull-up resistors connected to half the power supply voltage ($V_{CC}/2 \pm 10\%$) to generate a differential signal at **CANH** and **CANL**.

When a dominant bit is required to be transmitted, both the open drain transistors conduct resulting in $V_{CC} - 0.9V$ at **CANH** and $1.5V$ at **CANL**. The resulting voltage is a logic low hence a dominant bit.

When a recessive bit is required to be transmitted, both of the open drain transistors are put in High-Z state. By the help of pull up resistors $V_{CC}/2$ voltage is applied hence both **CANH** and **CANL** are at logic high, hence a recessive bit [31].

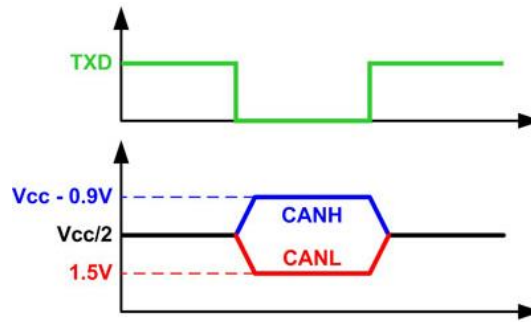


Figure 5.5: Transceiver Signals

Voltage levels of **TXD** and the corresponding **CANH** and **CANL** signals can be seen in Fig.5.5.

For the experiments Texas Instrument's TCAN Evaluation module is used. On the module board, there is a TCAN1042-Q1 Automotive Fault Protected CAN Transceiver with CAN FD.[19]. The picture of the transceiver board can be seen in Fig.5.6. FPGA demo board interfaces this board with TXD and RXD signals. Furthermore, 5V, 3.3V and GND signals are also provided by the FPGA board. These signals are available at the connector on the left side of the board in Fig.5.6. The connections are made between FPGA and transceiver board through this connector. Furthermore, on the right side of the board, CAN FD bus signals are available on the connector. In order for the voltage be formed properly on **CANH** and **CANL**, 120 ohm resistor should be connected between **CANH** and **CANL** lines. For this purpose, there is 120 ohm resistor available on the board.

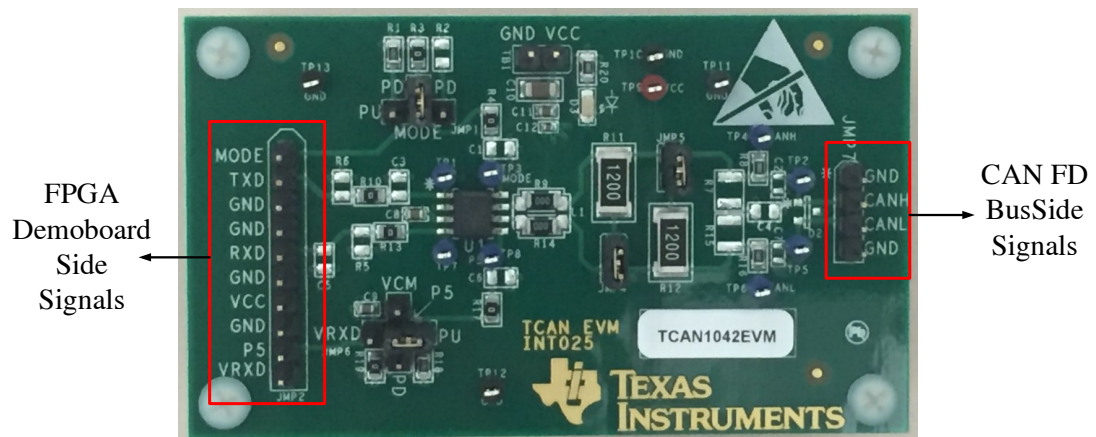


Figure 5.6: CAN FD Transceiver Board

CAN FD Analyzer: CAN FD Analyzer provides connection between CAN FD bus and the computer. Computer connection is done with USB interface. A computer Graphical User Interface called PCAN View is used to generate the desired CAN FD messages, monitor CAN FD bus and generate error. This adapter is used to debug and verify the functionality of the CAN FD Controller developed. Analyzer hardware can be seen in Fig.5.7.



Figure 5.7: PCAN USB FD Hardware

Platform Cable USB II: Platform Cable USB II provides both the hardware and the software to provide high- performance, reliable and easy to perform configuration of Xilinx devices. Platform Cable USB II connects to user hardware for the purpose of configuring Xilinx FPGAs, programming Xilinx PROMs and CPLDs. In addition, the cable is also used for indirectly programming of Platform Flash, third party SPI flash devices, and third-party parallel NOR flash devices via JTAG interface. Furthermore, Platform Cable USB II is a tool for debugging the embedded devices when used in conjunction with the tools such as Xilinx Embedded Development Kit and ChipScope Pro Analyzer. The platform cable can be seen in Fig.5.8.

5.2 Host Simulator Implementation

CAN FD Controller and Host Simulator have been implemented in the FPGA. The interface between the CAN FD Controller and the Host Simulator is SPI. SPI signals between these two are connected inside the FPGA as can be seen in Fig.5.9.

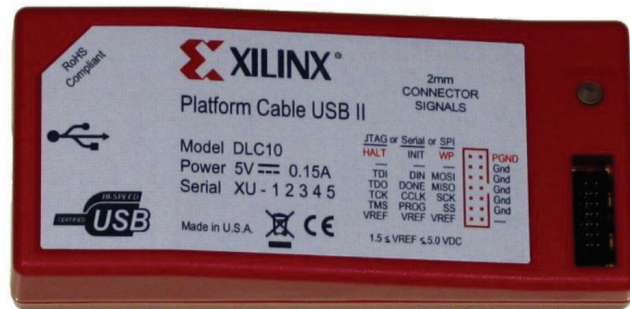


Figure 5.8: Xilinx Platform Cable

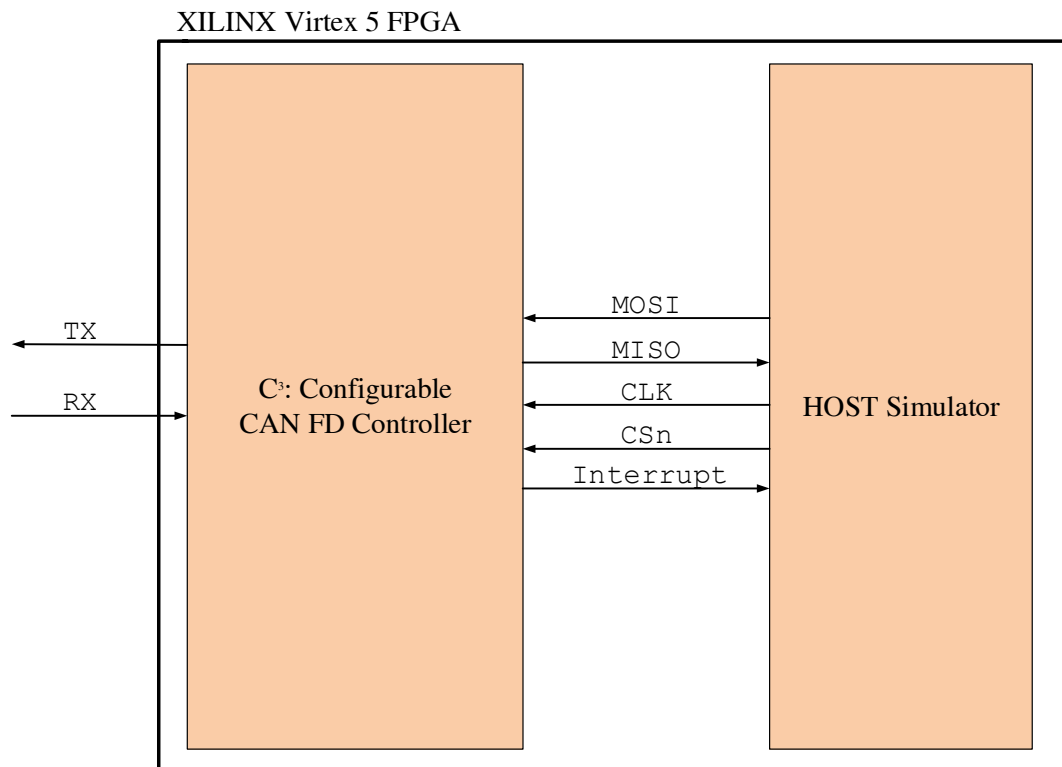


Figure 5.9: FPGA Implementation Block Diagram

Host simulator configures the controller, requests transmissions and reads received frames. In other words, it acts like a MCU. MCU includes a driver written special to the controllers and software application is abstracted from the controller's register sets. Driver handles all of the register management. It knows the addresses of the registers and maps the requests coming from the application to the corresponding register addresses in the controller and performs register read and write operations. Host simulator is implemented in an FPGA and since there is no MCU is involved, driver concept mentioned here makes no sense. Instead, different practical approach which is going to be described in this section is designed to make the Host Simulator act like a MCU. The block diagram of the Host Simulator is in Fig.5.10.

Master SPI Control Block (MSPIC): Host needs an SPI interface to communicate with the controller. As already mentioned in Sec.4.2, CAN FD Controller has a SPI slave implementation. Host has SPI master role. For this purpose, SPI master protocol is designed and implemented.

This block gets frame size in terms of bytes from TX Size FIFO, then gets as much data as read from TX Size FIFO from the TX Data FIFO and begins transmitting SPI frames. As already mentioned in Sec.4.2, `MOSI` line is used by the master and `MISO` line is used by the slave. While transmitting data on `MOSI` line, SPIC block records all of the data and puts it in the RX Data FIFO.

For Burst SPI Write and 32 bit SPI write, the data in the RX Data FIFO makes no sense since no response is required from the slave. To illustrate this, discarded data on `MISO` line is shown in Fig.5.11 for 32 bit SPI write command.

For Burst SPI Read and 32 bit SPI read operations, only some portion of the data makes sense, rest is discarded. To illustrate this, discarded data on `MISO` line is shown in Fig.5.12 for 32 bit SPI read command. For the portion of the frame where the slave responds, do not care part on `MOSI` line in Fig.5.12, as many 0xFF bytes as the amount of the response from the slave are transmitted to hold the `MOSI` line logic 1 and hold the `CSn` signal asserted for the whole duration of the frame.

To illustrate an example SPI frame transmission operation, the steps below are followed by SPIC:

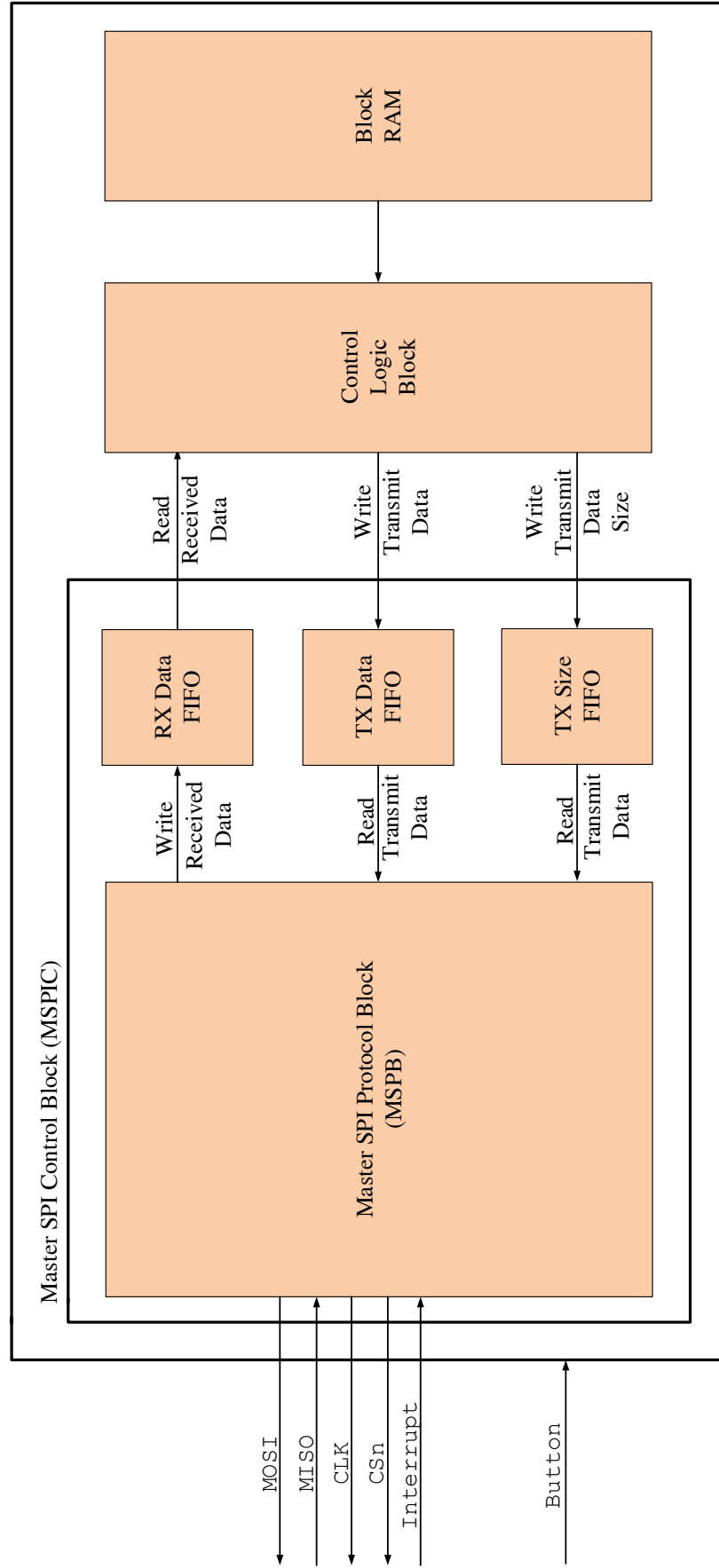


Figure 5.10: Host Simulator Block Diagram

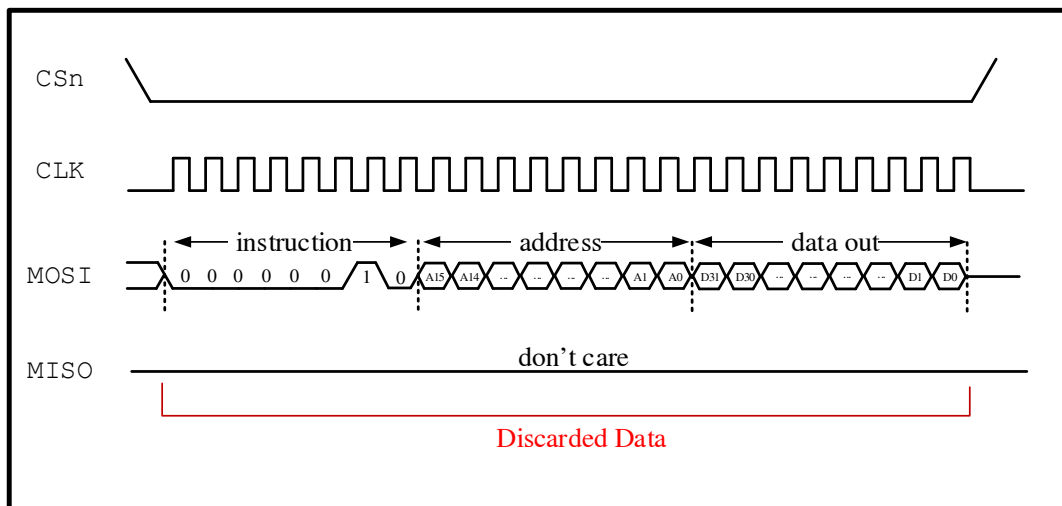


Figure 5.11: SPI Read Operation

- SPIC reads 8 from TX Size FIFO
- Then it reads 8 bytes from the RX data FIFO and forms a frame from this 8 bytes information and begins transmitting.
- While transmitting it records 8 bytes on MISO line and puts them into the RX Data FIFO byte by byte.

Control Logic Block: This block is connected to a block ram. The block ram holds the information in the following format.

Frame Size in terms of bytes + Bytes to be transmitted

Block RAM IP Core has the option to have initialization values when FPGA powers up. An initialization file is presented to the IP Core to define the initial content of each memory location. When the core is generated to be used in project, this file is loaded to block RAM through the block memory generator GUI. Every time this file is modified, whole FPGA code must be resynthesized and implemented to have the changes take effect. For each of four SPI commands, block ram data content is shown below:

32 bit SPI write operation to set **TX buffer 1 DLC (Data Length Code) Register** to 64 byte (0x.. represents hex format):

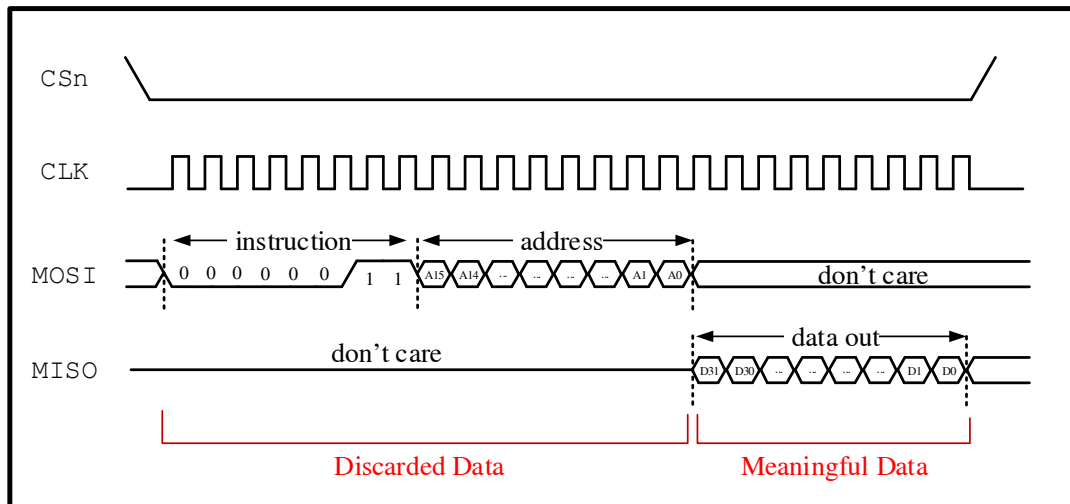


Figure 5.12: SPI Burst Read Operation

```

BRAM_INIT_20170723_18_33.txt - Notepad
File Edit Format View Help
; Sample initialization file for a
; 8-bit wide by 8192 deep RAM
memory_initialization_radix = 16;
memory_initialization_vector =
; set tx buffer 1 dlc=64
07, 02, 01, 04, 00, 00, 00, 40,
; set tx buffer 2 dlc=48
07, 02, 01, 10, 00, 00, 00, 30,
; set tx buffer 3 dlc=32
07, 02, 01, 1C, 00, 00, 00, 20,

```

Figure 5.13: Initialization Block Ram Data Content for 32 Bit SPI Write Command

- Instruction Code (1 byte): 0x02
- Register Address (2 bytes): 0x0104
- Data (32 bit-4 bytes): 0x00000040
- Frame Size in terms of bytes = 1 + 2 + 4 = 7
- When all of this information is put in the block ram, following content is formed: 0x07, 0x02, 0x01, 0x04, 0x00, 0x00, 0x00, 0x40
- The data content of this SPI frame in block ram can be seen in Fig.5.13.

32 bit SPI read operation to read **RSR 1 (Receive Status Register 1)** at address 0x0018 (0x.. represents hex format):

```

BRAM_INIT -20170802_10_45.txt - Notepad
File Edit Format View Help
:
:Clear RSR 12
07, 02, 00, 18, 00, 00, 08, 00,
:Read RSR 1
07, 03, 00, 18, FF, FF, FF, FF,
:Read ID 13
07, 03, 13, 90, FF, FF, FF, FF,
:Read Received Data 13
24, 01, 13, 9C, 20, FF, FF, FF, FF, FF,
:Clear RSR 13
07, 02, 00, 18, 00, 00, 10, 00,

```

Figure 5.14: Initialization Block Ram Data Content for 32 Bit SPI Read Command

- Instruction Code (1 byte): 0x03
- Register Address (2 bytes): 0x0018
- Data (32 bit-4 bytes): 0xFFFFFFFF
- Frame Size in terms of bytes = 1 + 2 + 4 = 7
- When all of this information is put in the block ram, following content is formed: 0x07, 0x03, 0x00, 0x18, 0xFF, 0xFF, 0xFF, 0xFF
- For the 32 bit SPI read, the slave responds to this request with 4 bytes (32 bit) of data, for this purpose, the data part of the frame is transmitted as 4 times 0xFF bytes to keep CSn signal asserted hence the slave is given enough time to provide the required 4 bytes of data. The data content of this SPI frame in block ram can be seen in Fig.5.14.

Burst SPI write to Buffer 1 **TX Data Register** (0x.. represents hex format):

- Instruction Code (1 byte): 0x00
- Register Address (2 bytes): 0x0108
- Burst Size (1 byte): 0x40
- Data (64 bytes): 0x01, 0x02, 0x03... 0x63, 0x64
- Frame Size in terms of bytes = 1 + 2 + 1 + 64 = 68 = 0x44

```

BRAM_INIT -20170802_10_45.txt - Notepad
File Edit Format View Help
:
:
:
: TX Buffer 1 Data Write 64x
44, 00, 01, 08, 40,
01, 02, 03, 04, 05, 06, 07, 08,
09, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32,
33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48,
49, 50, 51, 52, 53, 54, 55, 56,
57, 58, 59, 60, 61, 62, 63, 64,
:
:
: TX Buffer 2 Data Write 48x
34, 00, 01, 14, 30,
00, 02, 04, 06, 08, 10, 12, 14,
16, 18, 20, 22, 24, 26, 28, 30,
32, 34, 36, 38, 40, 42, 44, 46,
48, 50, 52, 54, 56, 58, 60, 62,
64, 66, 68, 70, 72, 74, 76, 78,
80, 82, 84, 86, 88, 90, 92, 94,
:
:

```

Figure 5.15: Initialization Block Ram Data Content for Burst SPI Write Command

- When all of this information is put in the block ram, following content is formed: 0x44, 0x00, 0x01, 0x08, 0x40, 0x01, 0x02, 0x03 ... 0x64
- The data content of this SPI frame in block ram can be seen in Fig.5.15.

32 bytes Burst SPI read from Buffer 11 **RX Data Register** (0x.. represents hex format):

- Instruction Code (1 byte): 0x01
- Register Address (2 bytes): 0x1304
- Burst Size (1 byte): 0x20
- Data (64 bytes): 0xFF, 0xFF, 0xFF ... 0xFF
- Frame Size in terms of bytes = 1 + 2 + 1 + 32 = 38 = 0x24
- When all of this information is put in the block ram, following content is formed: 0x24, 0x01, 0x13, 0x04, 0x20, 0xFF, 0xFF, 0xFF ... 0xFF
- For 32 byte Burst SPI read, the slave responds to this request with 32 bytes of data, for this purpose data part of the frame is transmitted as 32 times 0xFF

```

BRAM_INIT -20170802_10_45.txt - Notepad
File Edit Format View Help
07, 03, 12, AC, FF, FF, FF, FF,
;
;Read Received Data 10
34, 01, 12, B8, 30, FF, FF, FF,
FF, FF, FF, FF, FF, FF, FF, FF,
FF, FF, FF, FF, FF, FF, FF, FF,
FF, FF, FF, FF, FF, FF, FF, FF,
FF, FF, FF, FF, FF, FF, FF, FF,
FF, FF, FF, FF, FF,
;
;Clear RSR 10
07, 02, 00, 18, 00, 00, 02, 00,
;
;Read RSR 1
07, 03, 00, 18, FF, FF, FF, FF,
;
;Read ID 11
07, 03, 12, F8, FF, FF, FF, FF,
;
;Read Received Data 11
24, 01, 13, 04, 20, FF, FF, FF,
FF, FF, FF, FF, FF, FF, FF, FF,
FF, FF, FF, FF, FF, FF, FF, FF,
FF, FF, FF, FF, FF, FF, FF, FF,
FF, FF, FF, FF, FF,
;
;Clear RSR 11
07, 02, 00, 18, 00, 00, 04, 00,

```

Figure 5.16: Initialization Block Ram Data Content for Burst SPI Read Command

bytes to keep CS_n signal asserted hence slave is given enough time to provide the requested 32 bytes of data. The data content of this SPI frame in block ram can be seen in Fig.5.16.

The control logic block gets the frames one by one from the initialized block ram. It writes the frame size to the Size FIFO and the data to the Data FIFO of SPIC and waits for the transmission to be completed. For a new frame to be transmitted, it either waits for a trig (button press or an interrupt) or begins transmitting the new frame without waiting until a determined number of frame transmissions is reached. This depends on the application. For each application developed for CAN FD Controller verification, the content of Block RAM initialization file and the control logic block application code is modified.

5.3 Transmit Buffer Configuration and Transmission Tests

The purpose of this test is to verify the transmission functionality of CAN FD Controller. By doing this test, following functions are verified:

- 96 x Transmit Register Set functionality
- **TRR** functionality
- Each of 96 x Buffers functionality
- Buffer memory configurability
- Buffer priority functionality
- Transmission of the frames according to CAN FD protocol specification

Test setup in Fig.5.2 is used. Host Simulator's block ram initialization file is modified such that:

- TX buffers are configured such that the following number of messages with indicated payload size is stored. Total of 96 buffers are used and the total size of the buffers is 1392 bytes.
 - 64 bytes x 4 buffers
 - 48 bytes x 6 buffers
 - 32 bytes x 6 buffers
 - 24 bytes x 6 buffers
 - 20 bytes x 6 buffers
 - 16 bytes x 6 buffers
 - 12 bytes x 6 buffers
 - 8 bytes x 6 buffers
 - 7 bytes x 6 buffers
 - 6 bytes x 6 buffers
 - 5 bytes x 6 buffers
 - 4 bytes x 6 buffers
 - 3 bytes x 6 buffers
 - 2 bytes x 6 buffers
 - 1 byte x 14 buffers

- ID of the TX mailboxes are configured
- **Data** and **TRR (Transmit Request Register)** writes have been performed one by one to transmit the messages in an ordered way. For example:
 - The message content has been written to CAN FD Controller first mailbox buffer and
 - 0x00000001 was written to **TRR1** to initiate the transmission of the first message.
- Similarly 96 other mailboxes are requested to transmit their frames one by one following the procedure above. The messages are transmitted each time the push button is pressed. The message data content and the IDs of the messages received by the CAN FD analyzer are checked and compared with the ones on the FPGA side. The contents of the messages are verified to be correct.
- All of the data is written one by one to the mailboxes. Then 3 consecutive **TRR (TRR 1, TRR 2, TRR 3)** writes in order to request all of the messages to be transmitted at once have been performed. The messages have been transmitted in correct priority order and the message data content and ID of the messages received by the CAN FD analyzer is checked and compared with the ones on the FPGA side. The content of the messages are verified to be correct. The time gap between SPI frames is 1 second in the host application.

First 16 buffers configuration, ID assignment can be seen in Fig.5.17.

First 16 buffers' data content and transmission requests can be seen in Fig.5.18.

The message content (ID, data, message size and timestamp) received at CAN FD Analyzer can be seen in Fig.5.19. First 16 buffers are put in a red rectangle.

- Another experiment regarding the priority verification is done by using the same host application block memory initialization file as in the previous test. However, the order of the three **TRR** writes has been changed as in Fig.5.20. Furthermore, the time gap between SPI frames in the host application is reduced to 10 μ s. By doing this, we expect to see that the frame in the buffer number 33 is received first since it is requested first by **TRR 2** write. Then

```

; Sample initialization file for a
; 8-bit wide by 256 deep RAM
memory_initialization_radix = 16;
memory_initialization_vector =
;
; set tx buffer 1 dlc=64
07, 02, 01, 04, 00, 00, 00, 40,
;
; set tx buffer 2 dlc=48
07, 02, 01, 10, 00, 00, 00, 30,
;
; set tx buffer 3 dlc=32
07, 02, 01, 1C, 00, 00, 00, 20,
;
; set tx buffer 4 dlc=24
07, 02, 01, 28, 00, 00, 00, 18,
;
; set tx buffer 5 dlc=20
07, 02, 01, 34, 00, 00, 00, 14,
;
; set tx buffer 6 dlc=16
07, 02, 01, 40, 00, 00, 00, 10,
;
; set tx buffer 7 dlc=12
07, 02, 01, 4C, 00, 00, 00, 0C,
;
; set tx buffer 8 dlc=8
07, 02, 01, 58, 00, 00, 00, 08,
;
; set tx buffer 9 dlc=7
07, 02, 01, 64, 00, 00, 00, 07,
;
; set tx buffer 10 dlc=6
07, 02, 01, 70, 00, 00, 00, 06,
;
; set tx buffer 11 dlc=5
07, 02, 01, 7C, 00, 00, 00, 05,
;
; set tx buffer 12 dlc=4
07, 02, 01, 88, 00, 00, 00, 04,
;
; set tx buffer 13 dlc=3
07, 02, 01, 94, 00, 00, 00, 03,
;
; set tx buffer 14 dlc=2
07, 02, 01, A0, 00, 00, 00, 02,
;
; set tx buffer 15 dlc=1
07, 02, 01, AC, 00, 00, 00, 01,
;
; set tx buffer 16 dlc=1
07, 02, 01, B8, 00, 00, 00, 01,

```

```

; write TX Buffers Configuration Status Register
07, 02, 00, 00, 00, 00, 00, 09,
;
-----
; set rx buffer 1 dlc=64
07, 02, 10, 08, 00, 00, 00, 40,
;
; set rx buffer 2 dlc=12
07, 02, 10, 54, 00, 00, 00, 0C,
;
; set rx buffer 3 dlc=7
07, 02, 10, A0, 00, 00, 00, 07,
;
; set rx buffer 4 dlc=1
07, 02, 10, EC, 00, 00, 00, 01,
;
; write RX Buffers Configuration Status Register
07, 02, 00, 04, 00, 00, 00, 09,
;
-----
; set TX Buffer 1 ID Register BASE
07, 02, 01, 00, 6D, E0, 00, 00,
;
; set TX Buffer 2 ID Register BASE
07, 02, 01, 0C, 6E, 00, 00, 00,
;
; set TX Buffer 3 ID Register BASE
07, 02, 01, 18, 6E, 20, 00, 00,
;
; set TX Buffer 4 ID Register BASE
07, 02, 01, 24, 6E, 40, 00, 00,
;
; set TX Buffer 5 ID Register BASE
07, 02, 01, 30, 6E, 60, 00, 00,
;
; set TX Buffer 6 ID Register BASE
07, 02, 01, 3C, 6E, 80, 00, 00,
;
; set TX Buffer 7 ID Register BASE
07, 02, 01, 48, 6E, A0, 00, 00,
;
; set TX Buffer 8 ID Register BASE
07, 02, 01, 54, 6E, C0, 00, 00,
;
; set TX Buffer 9 ID Register BASE
07, 02, 01, 60, 6E, E0, 00, 00,
;
; set TX Buffer 10 ID Register BASE
07, 02, 01, 6C, 6F, 00, 00, 00,
;
; set TX Buffer 11 ID Register BASE
07, 02, 01, 78, 6F, 20, 00, 00,
;
; set TX Buffer 12 ID Register BASE
07, 02, 01, 84, 6F, 40, 00, 00,
;
; set TX Buffer 13 ID Register BASE
07, 02, 01, 90, 6F, 60, 00, 00,
;
; set TX Buffer 14 ID Register BASE
07, 02, 01, 9C, 6F, 80, 00, 00,
;
; set TX Buffer 15 ID Register BASE
07, 02, 01, A8, 6F, A0, 00, 00,
;
; set TX Buffer 16 ID Register BASE
07, 02, 01, B4, 6F, C0, 00, 00,

```

Figure 5.17: First 16 Buffers Configuration

| | | |
|---|--|---|
| TX Buffer 1 Data write 64x 44, 00, 01, 08, 40, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, | TX Buffer 5 Data write 20x 18, 00, 01, 38, 14, 00, 05, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, | TX Buffer 12 Data write 4x 08, 00, 01, 8C, 04, 00, 12, 24, 36, |
| TX Buffer 2 Data write 48x 34, 00, 01, 14, 30, 00, 02, 04, 06, 08, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, | TX Buffer 6 Data write 16x 14, 00, 01, 44, 10, 00, 06, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 92, | TX Buffer 13 Data write 3x 08, 00, 01, 98, 04, 00, 13, 26, 00, |
| TX Buffer 3 Data write 32x 24, 00, 01, 20, 20, 00, 03, 06, 09, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, | TX Buffer 7 Data write 12x 10, 00, 01, 50, 0C, 00, 07, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, | TX Buffer 14 Data write 2x 08, 00, 01, A4, 04, 00, 14, 00, 00, |
| TX Buffer 4 Data write 24x 1C, 00, 01, 2C, 18, 00, 04, 08, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84, 88, 92, | TX Buffer 8 Data write 8x 0C, 00, 01, 5C, 08, 00, 08, 16, 24, 32, 40, 48, 54, | TX Buffer 15 Data write 1x 08, 00, 01, 80, 04, 00, 00, 00, 00, |
| TX Buffer 5 Data write 20x 18, 00, 01, 38, 14, 00, 05, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, | TX Buffer 9 Data write 7x 0C, 00, 01, 68, 08, 00, 09, 18, 27, 36, 45, 54, 00, | TX Buffer 16 Data write 24x 08, 00, 01, BC, 04, FF, 00, 00, 00, |
| | TX Buffer 10 Data write 6x 0C, 00, 01, 74, 08, 00, 10, 20, 30, 40, 50, 00, 00, | TRR 1 tx request all 07, 02, 00, 08, FF, FF, FF, FF, |
| | TX Buffer 11 Data write 5x 0C, 00, 01, 80, 08, 00, 11, 22, 33, 44, 00, 00, 00, | TRR 2 tx request all 07, 02, 00, 0C, FF, FF, FF, FF, |
| | | TRR 3 tx request all 07, 02, 00, 10, FF, FF, FF, FF, |

Figure 5.18: First 16 Buffers Transmission Data Content and Request

after the transmission is over, Buffer 1 takes over the transmission priority. Because by the time **TRR 1** is written, Buffer 33 is the buffer with the highest priority. Therefore, after the buffer 33 frame transmission, we expect to see buffer 1 transmission, buffer 2 transmission, buffer 3 transmission and so on.

The messages are received in the expected order as can be seen in Fig.5.21 and the priority of the buffers is verified.

- Both of the frame types, which are with the base and the extended ID are verified. PCAN shows the 29 bit and 11 bit representation of the IDs while the memory initialization file shows the 32 bit representation of the IDs as described in ID register. Therefore, when 32 bit ID in the initialization file is converted to 11 bit Base or 29 bit Extended ID format, it is verified that the same ID number is seen at the PCAN analyzer as the FPGA side.
- Buffer sizes are arranged such that they include every possible message size CAN FD protocol defines therefore different message sizes and two types of CRC calculation methods depending on the message size are verified.

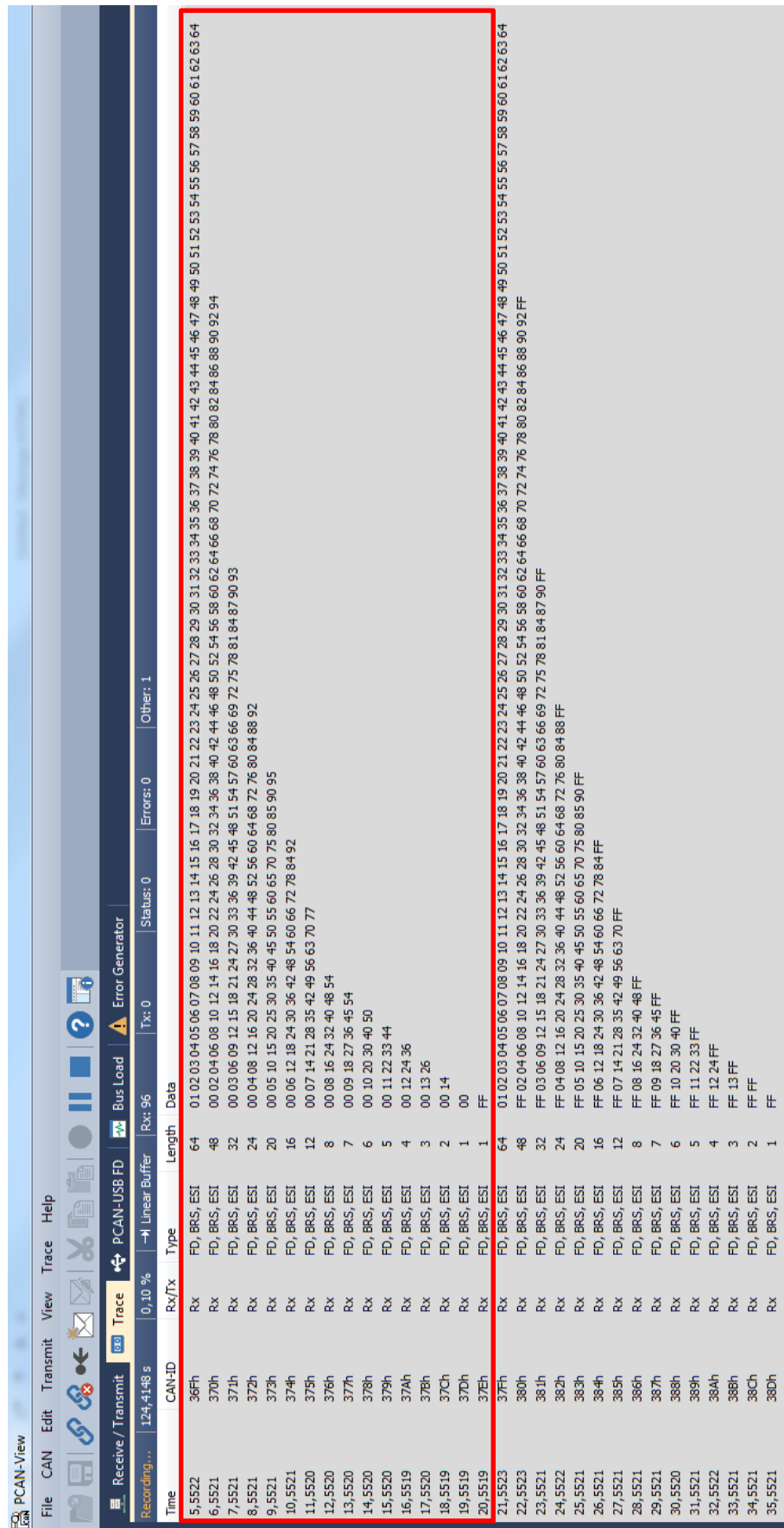


Figure 5.19: PCAN Data Logging for First 16 Transmissions

```

; TRR 2 tx request all
07, 02, 00, 0C, FF, FF, FF, FF,
;
; TRR 1 tx request all
07, 02, 00, 08, FF, FF, FF, FF,
;
; TRR 3 tx request all
07, 02, 00, 10, FF, FF, FF, FF,
;

```

Figure 5.20: Transmission Requests for Priority Test

5.4 Receive Buffer Configuration and Reception Tests

The purpose of this test is to verify the reception functionality of the CAN FD Controller. By doing this test, following functions are verified:

- RX buffers are configured such that the following number of messages with the indicated payload are stored. Total of 96 buffers are used and the total size of the buffers is 1392 bytes.
 - 64 bytes x 4 buffers
 - 48 bytes x 6 buffers
 - 32 bytes x 6 buffers
 - 24 bytes x 6 buffers
 - 20 bytes x 6 buffers
 - 16 bytes x 6 buffers
 - 12 bytes x 6 buffers
 - 8 bytes x 6 buffers
 - 7 bytes x 6 buffers
 - 6 bytes x 6 buffers
 - 5 bytes x 6 buffers
 - 4 bytes x 6 buffers
 - 3 bytes x 6 buffers
 - 2 bytes x 6 buffers

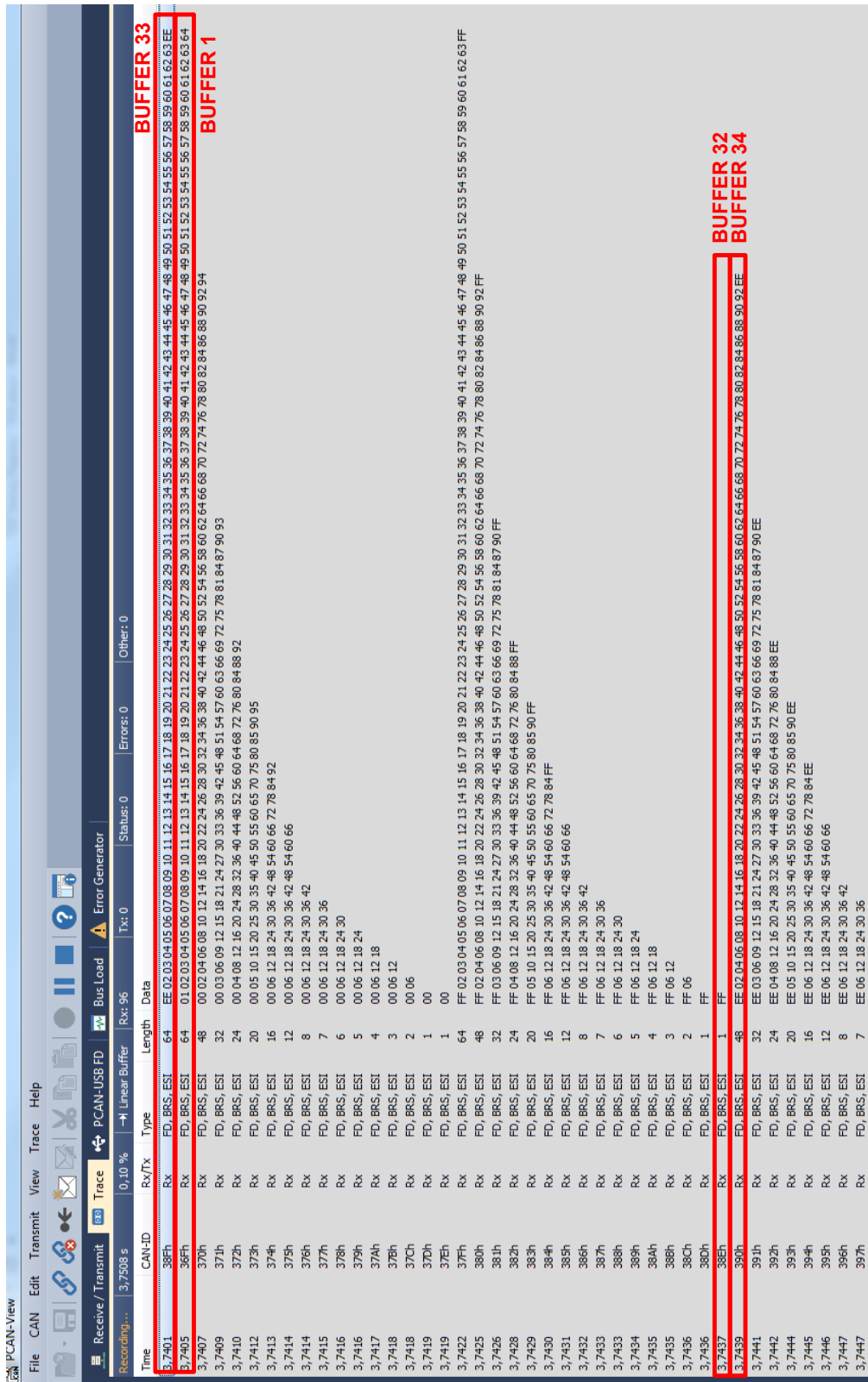


Figure 5.21: PCAN Message Reception for Transmission Priority Test

– 1 byte x 14 buffers

- ID and Mask Registers are configured
- For each buffer, a frame with the ID that can pass the buffer's filter and a frame with an ID that can't pass the buffer's filter are prepared. Therefore, $96 \times 2 = 192$ frames are going to be transmitted via CAN FD Analyzer's PCAN View to CAN FD Controller.
- The frames that are in the passing group can only pass one buffer's filter.
- The data content of the frames that can't pass the filter is filled with all 0xFFs.
- These 192 message frames are transmitted by the analyzer.
- CAN FD Controller receives them all and should place the frames that pass the filter into the related buffers in FRM and ID and data content should be the same as the ones on the PCAN View transmitted. Furthermore, data content should not be all 0xFFs.
- Since all of the 96 RX buffers are filled with the received data, the content of **RSR** is expected to be all logic 1.
- Host performs read operations and puts the received data to its RX Data FIFO as already explained in Sec.4.5. ChipScope is used to monitor the FIFO signals. ChipScope is like an oscilloscope inside the FPGA. It monitors the real time data and shows this information to the user for debugging, verification and testing purposes. It has to be given a trigger so that the desired data is captured when the trigger condition occurs. It is possible to have as many trigger occurrences as desired in a single capture. In order to observe the read data from SPI, trigger condition is set to rising edge of `rx_data_fifo_wr_en` signal. This signal is the FIFO write enable strobe. SPIC pulses this FIFO Write Enable signal along with the received byte to put the received byte in RX Data FIFO. Each time this signal is pulsed, the new data is written to the FIFO. Therefore, by setting the trigger condition to rising edge of this signal and trigger occurrence count to 96, it is possible to monitor all of the read data from CAN FD Controller. Even for the frame with 64 byte payload 96 trigger

occurrences are enough to capture every single received byte from CAN FD Controller signals.

- An application is designed for this test such that whenever the button on the demo board is pressed, a new SPI read or write request from Host simulator to CAN FD Controller takes place.
- Before the button is pressed, ChipScope is commanded to wait for trigger. When the button is pressed, data is captured.
- The following procedure is repeated by 96 times, starting from the Buffer 1.
 - One of the three **RSR** is read by Host Simulator to learn the message mailbox number holding the message to verify that the message passes the filter correctly and its stored in the correct mailbox.
 - ID of the message received is read by Host Simulator and compared with PCAN View data so that the received message ID is verified
 - The message content of the received frame is read by Host Simulator and the payload of the received frame is verified by comparing with the data on PCAN View
 - Related **RSR** bit is cleared by the Host Simulator by performing **RSR** write
- By performing this test,
 - Both of the frame types, which are with the base and the extended ID are verified.
 - 96 x RX buffer functionality is verified
 - The message reception according to CAN FD protocol specification is verified
 - All possible message sizes and CRC calculation according to the message size are tested and verified
 - **RSR** functionality is verified
 - Buffer configurability is verified

- The reception procedure described here is illustrated as follows and Buffer 27 is chosen as a sample.
 - PCAN messages to be transmitted by PCAN with matching and non-matching IDs for Buffer 27 can be seen in Fig.5.22.

PCAN-View

FileCANEditTransmitViewTraceHelp

Receive / TransmitTrace

Receive

| CAN-ID | Type | Length | Data | Cycle Time | Count |
|---------|------|--------|------|------------|-------|
| <Empty> | | | | | |

Transmit

| CAN-ID | Type | Length | Data | Cycle Time | Count |
|--------|--------|--------|---|------------|-------|
| 2A1h | FD BRS | 20 | 76 74 72 70 68 66 64 62 60 58 56 54 52 50 48 46 44 42 40 38 | 100 | 0 |
| 2A2h | FD BRS | 20 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 2A5h | FD BRS | 20 | 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 | 100 | 0 |
| 2A6h | FD BRS | 20 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 2A9h | FD BRS | 20 | 88 86 84 82 80 78 76 74 72 70 68 66 64 62 60 58 56 54 52 50 | 100 | 0 |
| 2AAh | FD BRS | 20 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 2ADh | FD BRS | 20 | 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 | 100 | 0 |
| 2AEh | FD BRS | 20 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 2B1h | FD BRS | 16 | 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 | 100 | 0 |
| 2B2h | FD BRS | 16 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 2B5h | FD BRS | 16 | 00 03 06 09 12 15 18 21 24 27 30 33 36 39 42 45 | 100 | 0 |
| 2B6h | FD BRS | 16 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 2B9h | FD BRS | 16 | 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 | 100 | 0 |
| 2BAh | FD BRS | 16 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 2BDh | FD BRS | 16 | 90 88 86 84 82 80 78 76 74 72 70 68 66 64 62 60 | 100 | 0 |
| 2BEh | FD BRS | 16 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 481h | FD BRS | 16 | 01 02 01 99 96 93 90 87 84 81 78 75 72 69 66 63 | 100 | 0 |
| 482h | FD BRS | 16 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 485h | FD BRS | 16 | 96 94 92 90 88 86 84 82 80 78 76 74 72 70 68 66 | 100 | 0 |
| 486h | FD BRS | 16 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 489h | FD BRS | 12 | 01 02 03 04 05 06 07 08 09 10 11 12 | 100 | 0 |
| 48Ah | FD BRS | 12 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 48Dh | FD BRS | 12 | 02 04 06 08 10 12 14 16 18 20 22 24 | 100 | 0 |
| 48Eh | FD BRS | 12 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 691h | FD BRS | 12 | 49 48 47 46 45 44 43 42 41 40 39 38 | 100 | 0 |
| 692h | FD BRS | 12 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |
| 695h | FD BRS | 12 | 98 96 94 92 90 88 86 84 82 80 78 76 | 100 | 0 |
| 696h | FD BRS | 12 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF | 100 | 0 |

Filter Matching Frame for Buffer 27

Filter Not Matching Frame for Buffer 27

Figure 5.22: Message Content for Buffer 27

- **RSR1** is read as 0xFC000000 and can be seen in Fig.5.23a

RSR1[31:0] = 0xFC000000 means:

- * RSR1[25:0] -> All logic 0s
- * RSR1[31:26]-> All logic 1s.

Since Buffers are read one by one and the related bits of **RSR** are cleared. 26 other **RSR** bits are already cleared. And remaining 6 bits of **RSR 1** are logic 1 and will be cleared in the remaining part of the test.

- ID of the message read as in Fig.5.23b. 0x55200000 -> 2A9h
- The payload of the message is as in Fig.5.24a.
- **RSR 1** after related buffer 27 bit is cleared is read as 0xF8000000 and can be seen in Fig.5.24b.
 - * RSR 1[31:0] = 0xF8000000 means:
 - RSR 1[26:0] -> All logic 0s
 - RSR 1[31:27]-> All logic 1s.
 - * Buffers are read one by one and related bits of **RSR** are cleared. 26th **RSR 1** bit is cleared with this operation hence total of 27 bits of **RSR 1** are cleared by now.

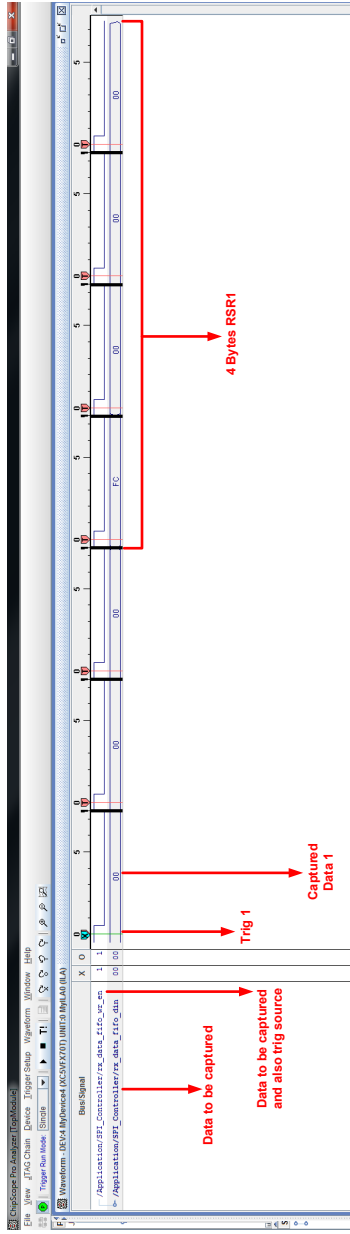
5.5 Response Time Measurements

Response time measurements are carried out for transmission and reception separately.

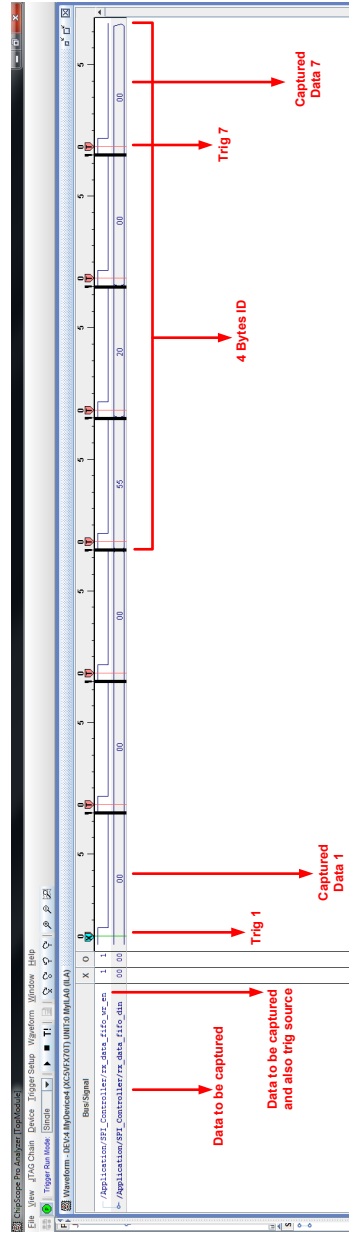
Transmission Response Time: Host writes data through SPI to the related mailbox and writes to **TRR** register to initiate transmission. Host Simulator is arranged to fulfill this purpose. FPGA Core polls **TRR** to detect any pending request, determines the number of the pending buffer having the highest priority by binary search algorithm. Core reads the data from the mailbox and writes this data to CAN FD TX Data FIFO and message size is already known by the controller and it is also written CAN FD TX Size FIFO and the transmission begins. Response time consists of all of these processes. The response time delay components are as follows:

- FPGA Core Delay:
 - Binary search to determine the highest priority message to transmit
 - Memory read operation to get the transmit data from the related mailbox
 - Memory write operation to the CAN FD Transmitter FIFOs

The measurement is done between SPI frame is received completely (the time when SPI CS_n signal is deasserted) and CAN FD frame begins transmission (start bit of canfd_tx_int signal)

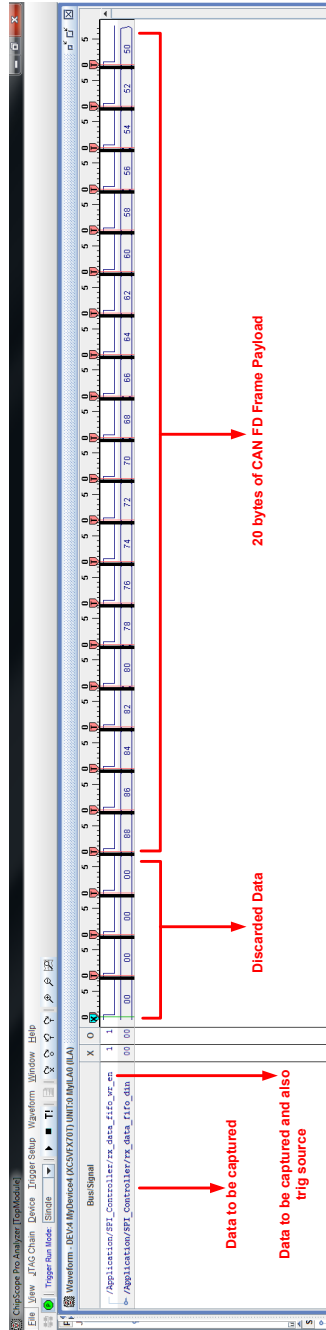


(a) RSR1 Read Operation

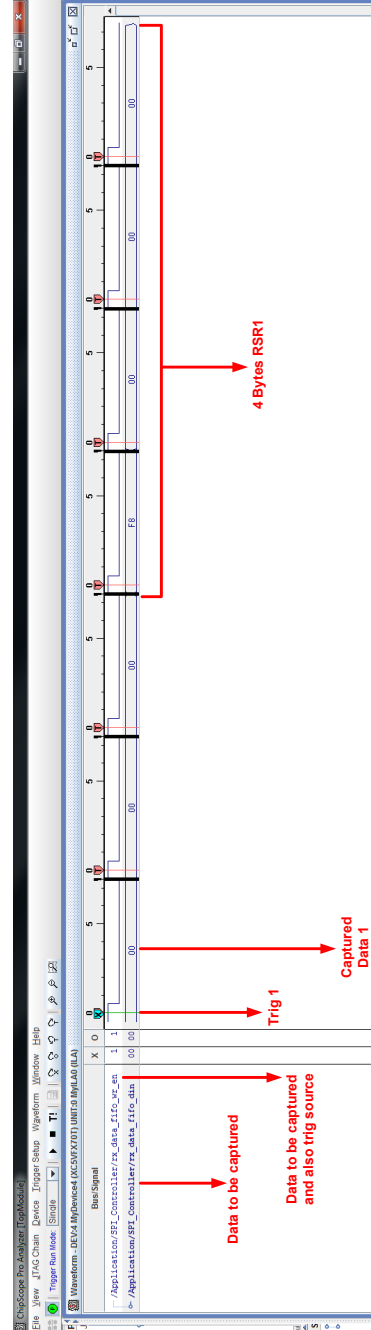


(b) 27th Buffer ID Register Read Operation

Figure 5.23: Buffer 27 Read Operations 1



(a) 27th Buffer Data Register Read Operation



(b) RSR1 Read Operation After Clearing

Figure 5.24: Buffer 27 Read Operations 2

Chipscope Result: 113 FPGA clock cycles = 1,13 μ s as can be seen in Fig.5.25a.

- SPI Burst write to the mailbox frame data can be seen in Fig.5.25b.

Theoretical Result: $(8 + 16 + 8 + B \cdot 8) \cdot c_{SPI}$, For $B = 64$, the delay is 54,4 μ s as already explained in Sec.4.7.

Chipscope Result: 54,51 μ s

The difference between the theoretical result and the practical result is due to the fact that $\overline{CS_n}$ signal is kept asserted for some additional clock cycles before and after the signals are active.

- SPI write to **TRR** can be seen in Fig.5.26a.

Theoretical Result: $(8 + 16 + 32) \cdot c_{SPI}$, the delay is 5,4 μ s as already explained in Sec.4.7.

Chipscope Result: 5,72 μ s

The difference between the theoretical result and the practical result is due to the fact that $\overline{CS_n}$ signal is kept asserted for some additional clock cycles before and after the signals are active.

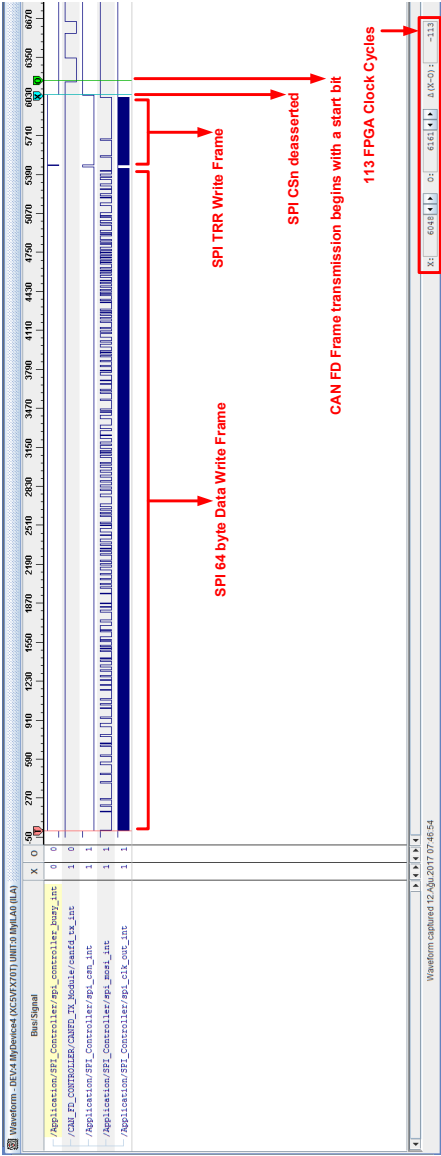
- Over all response time measurement can be seen in Fig.5.26b.

Theoretical Result: 54,4 μ s + 5,4 μ s = 59,8 μ s as explained in Sec.4.7.

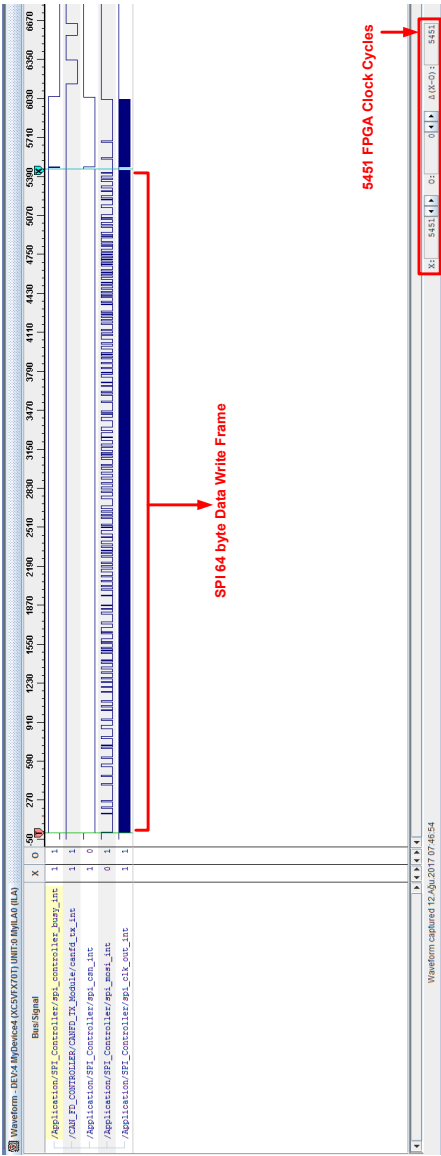
Chipscope Result: 61,48 μ s

Reception Response Time: CAN FD message is received by the controller. An interrupt is generated to notify the Host Simulator. When the Host Simulator gets the interrupt it begins reading **ISR (Interrupt Status Register)** to learn the source of the interrupt. Then after learning message reception event, it reads three **RSR** namely **RSR1**, **RSR2**, **RSR3** to identify the buffer number holding the received message. Then it reads data register of the related buffer. It clears **ISR** and **RSR**. Response time consists of all of these processes. The response time delay components are as follows:

- FPGA Core Delay:
 - FPGA writing the received data to related buffer memory



(a) FPGA Core Delay



(b) SPI Burst Write Delay

Figure 5.25: TX Response Time Measurements 1

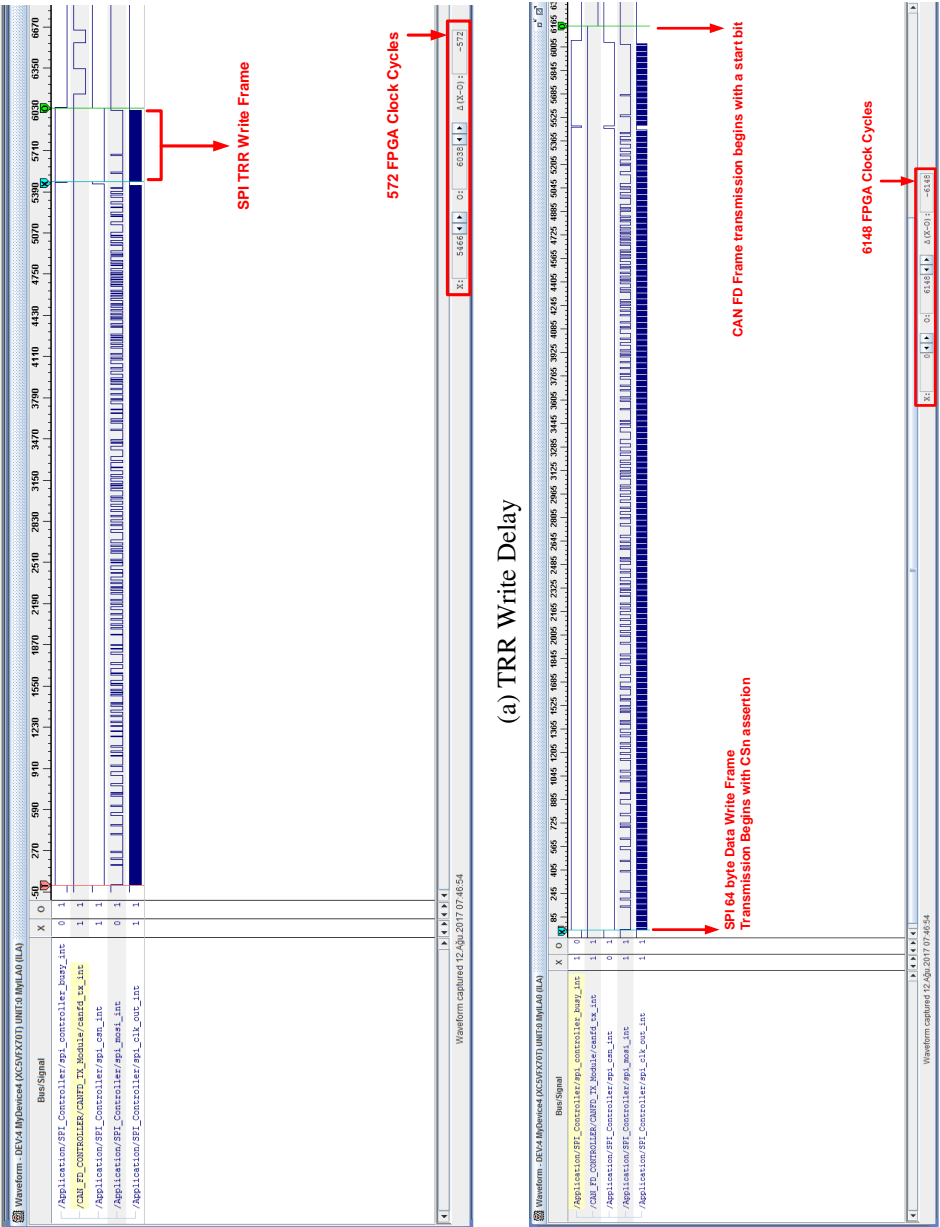


Figure 5.26: TX Response Time Measurements 2

- FPGA updating **RSR** and generating interrupt

The time between the CAN FD frame is received and the interrupt is generated is measured in Fig.5.27.

Chipscope Result: 1,41 μs as can be seen in Fig.5.27.

- SPI Read from **ISR** by the host to determine the source of the interrupt can be seen in Fig.5.28.

Theoretical Result: $(8 + 16 + 32) \cdot c_{SPI}$, the delay is 5,4 μs as already explained in Sec.4.7.

Chipscope Result: 5,72 μs

- 3x SPI Read from **RSR** (**RSR 1**, **RSR 2** and **RSR 3**) can be seen in Fig.5.29

Theoretical Result: $3 \times (8 + 16 + 32) \cdot c_{SPI}$, the delay is 16,2 μs as already explained in Sec.4.7.

Chipscope Result: $3 \times 5,72 \mu s = 17,46 \mu s$

- Burst SPI Read of the payload of the frame can be seen in Fig.5.30.

Theoretical Result: $(8 + 16 + 8 + B \cdot 8) \cdot c_{SPI}$, For $B = 64$, the delay is 54,4 μs as already explained in Sec.4.7.

Chipscope Result: 54,52 μs

- SPI Write to clear the related **RSR** can be seen in Fig.5.31.

Theoretical Result: $(8 + 16 + 32) \cdot c_{SPI}$, the delay is 5,4 μs as already explained in Sec.4.7.

Chipscope Result: 5,72 μs

- SPI Write to clear **ISR** can be seen in Fig.5.32.

Theoretical Result: $(8 + 16 + 32) \cdot c_{SPI}$, the delay is 5,4 μs as already explained in Sec.4.7.

Chipscope Result: 5,72 μs

- Overall response time measurement be seen in Fig.5.33

Theoretical Result: 89,4 μs as explained in Sec.4.7.

Chipscope Result: 91,26 μs

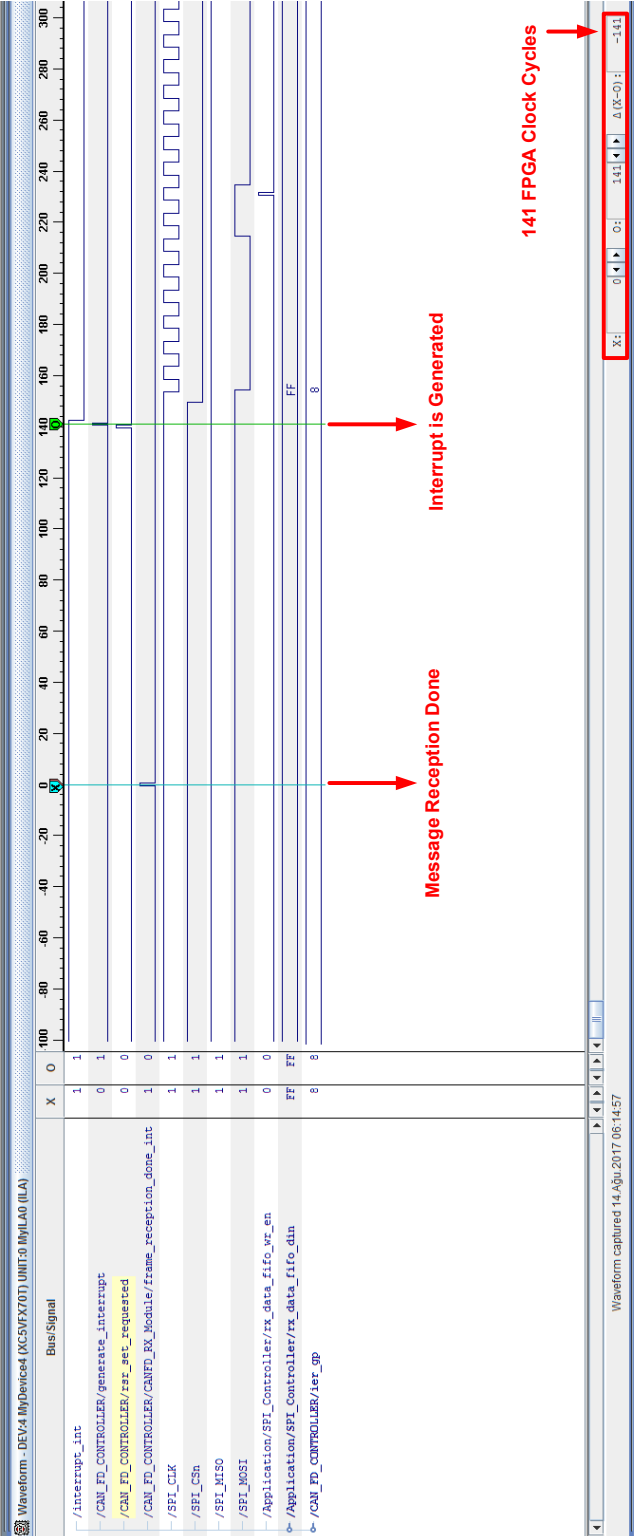


Figure 5.27: FPGA Core Delay

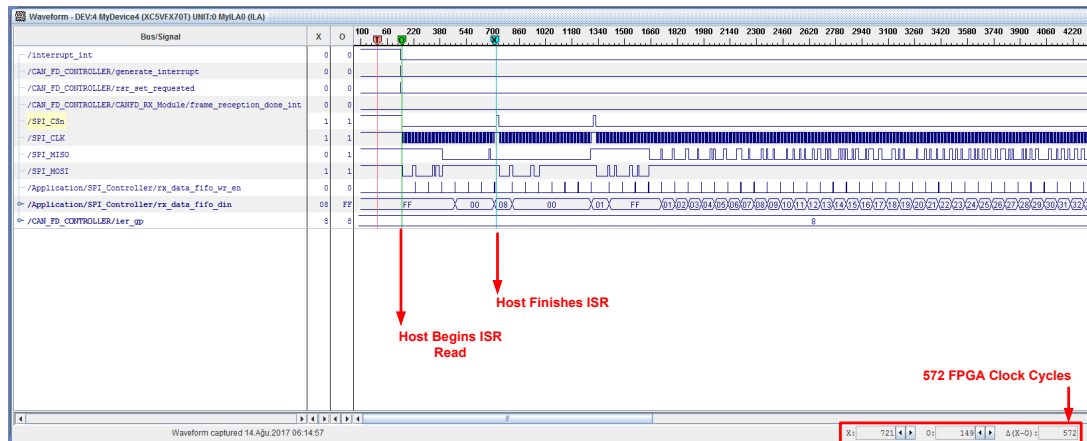


Figure 5.28: ISR Read Delay

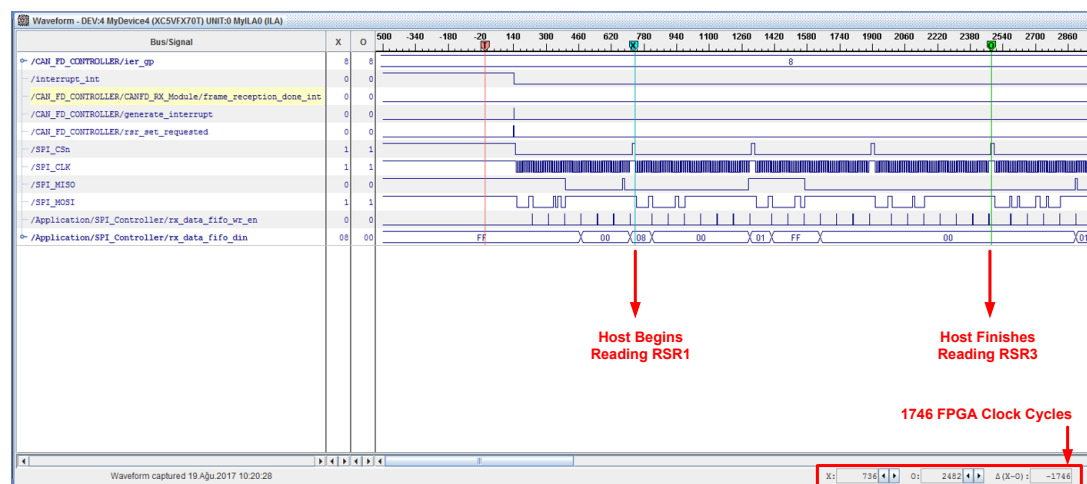


Figure 5.29: 3x RSR Read Delay

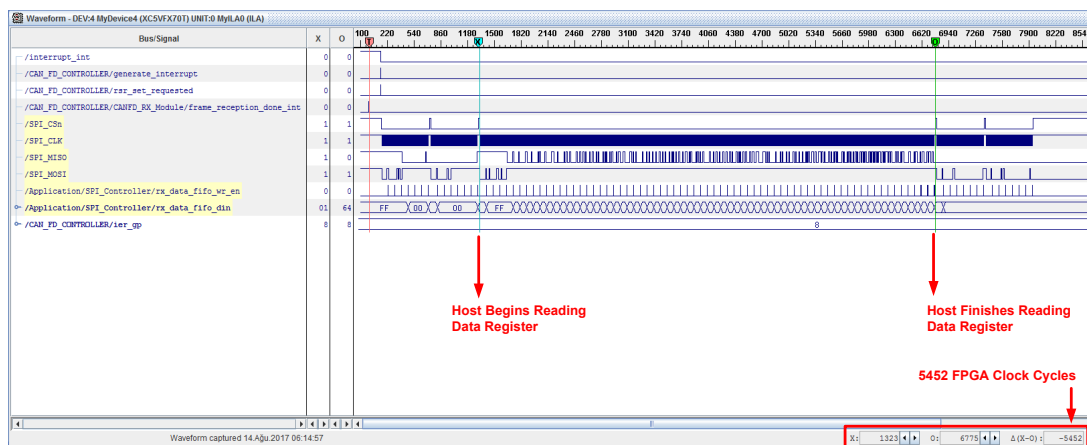


Figure 5.30: Burst SPI Read Delay

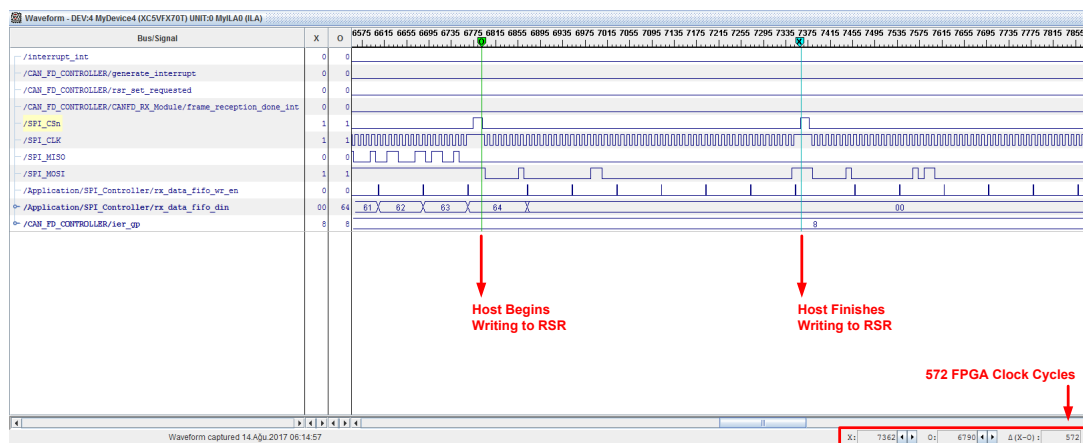


Figure 5.31: RSR Clear Delay

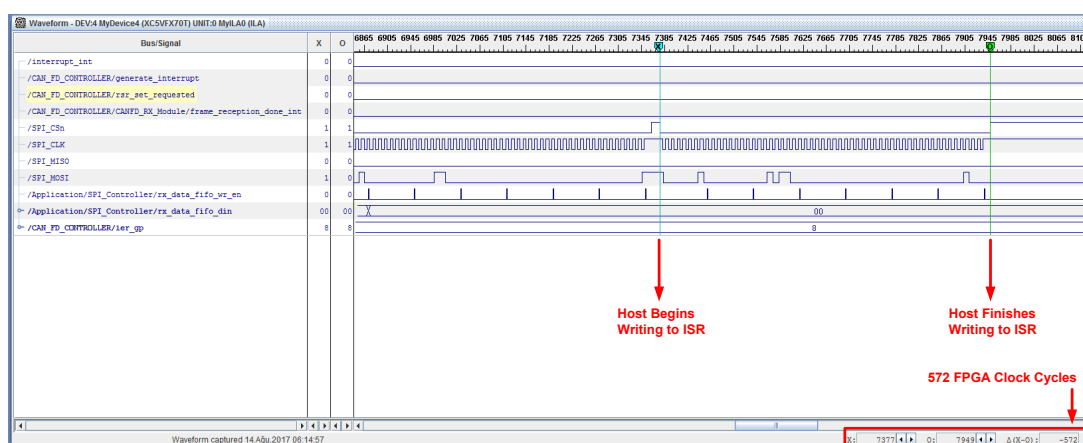


Figure 5.32: ISR Clear Delay

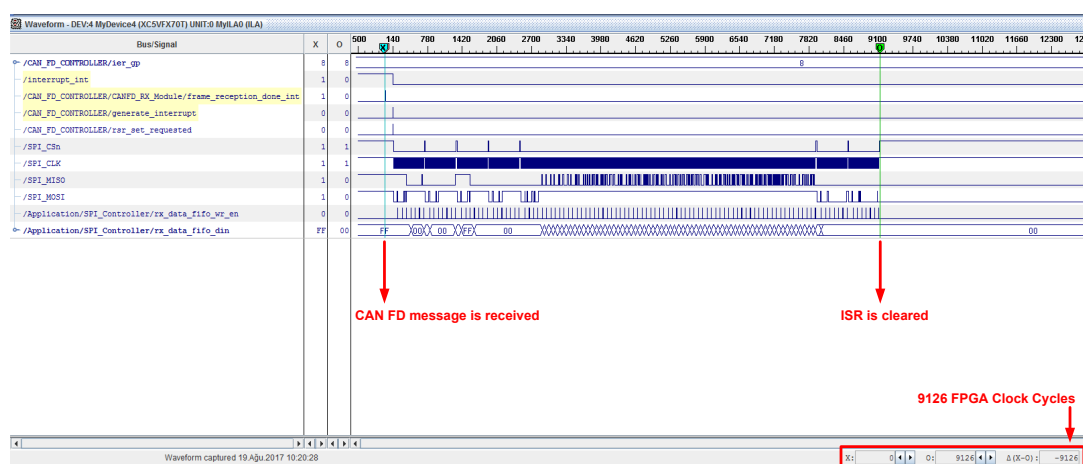


Figure 5.33: Total Receive Response Time

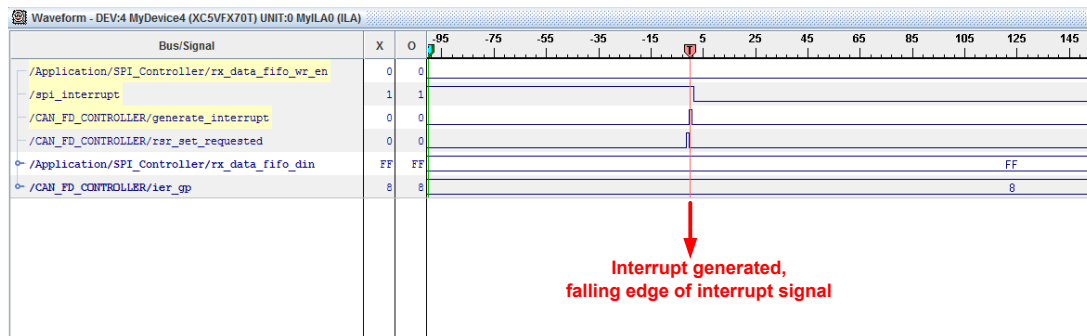


Figure 5.34: Interrupt Generation

5.6 Interrupt and Error Tests

Interrupt Test: When a message is received and it passes the filter, an interrupt is generated successfully and **ISR** is set accordingly. Interrupt is generated as active low pulse, it is verified as in Fig.5.34 with `spi_interrupt` signal.

ISR is read and its verified that **ISR** includes the information that a new message is received, in other words 3th bit of **ISR** is logic **1** as in Fig.5.35

Transmission Error Tests

- Ack Error: PEAK Analyzer is set to *listen only* mode. In this mode, the analyzer only listens to the bus and does not send responses such as acknowledgment to the received messages. In other words, it acts like a passive node. *Listen only* setting can be seen in Fig.5.36 and Fig.5.37. When FPGA transmits a message and does not get an acknowledgment, error condition occurs. For this case, **TMSR (Transmit Message Status Register)** last two bits are read '**10**' and **ISR** last 4 bits are read as '**0001**' as can be seen in Fig.5.38. These values indicate that ack error condition is detected and related registers are correctly modified.
- Bit Error: PEAK Analyzer has a feature called Error Generator. Analyzer puts 6 dominant bits at the indicated position selected in GUI as in Fig.5.39. Transmitter detects the error when its recessive bit is overwritten by a dominant bit. This might take up to 6 bit time. Error generator is set to destroy 40th bit of

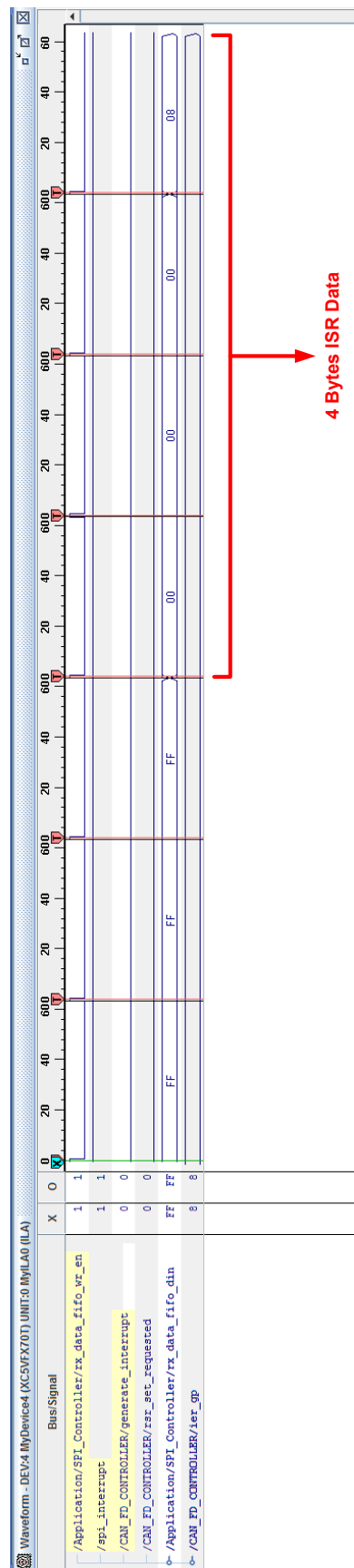


Figure 5.35: ISR Read Operation

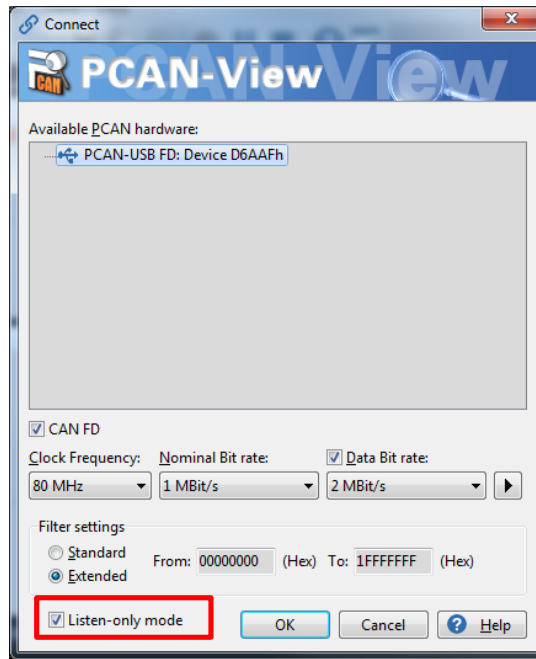


Figure 5.36: PCAN Listen Only Mode Settings 1

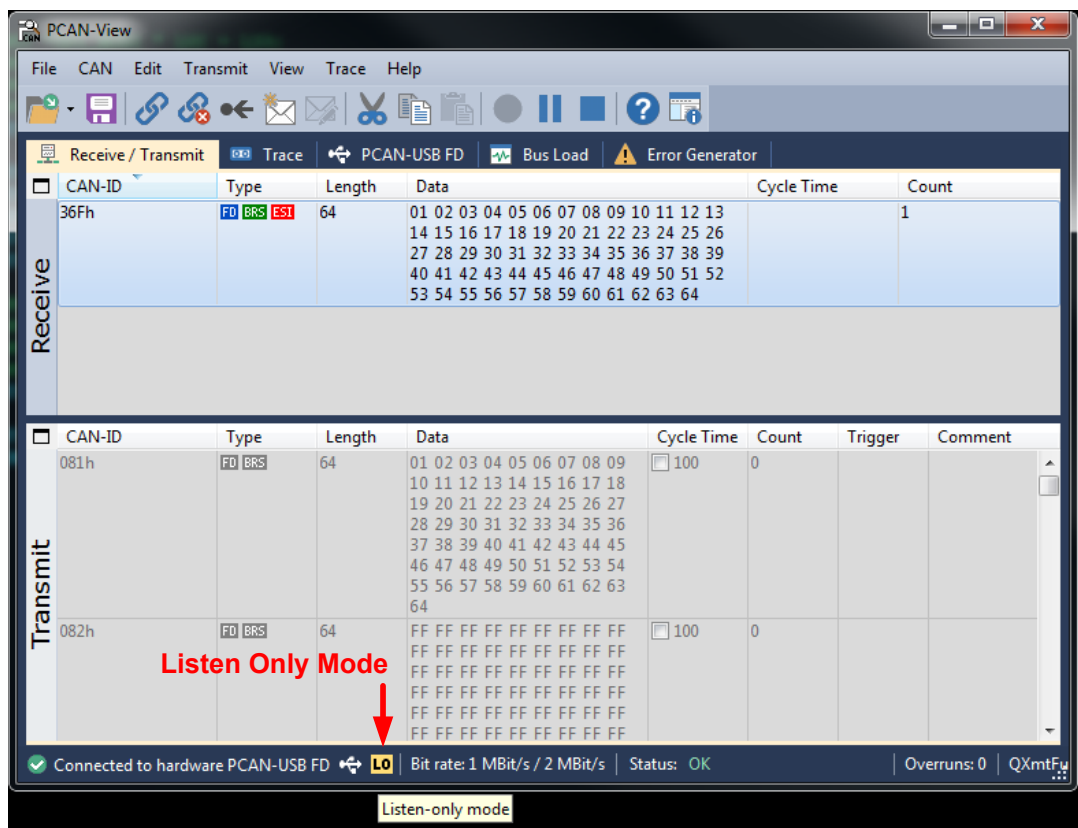


Figure 5.37: PCAN Listen Only Mode Settings 2

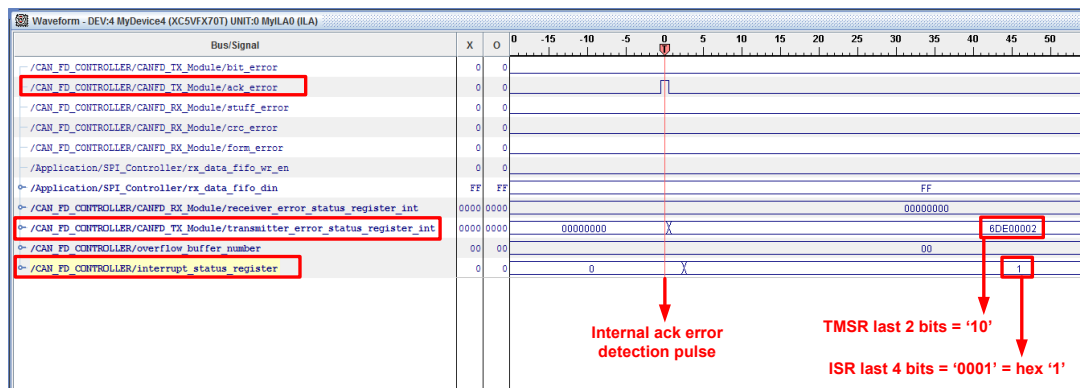


Figure 5.38: TX Ack Error Register Content

the frame. By doing this, bit error is introduced. **TMSR** last 2 bits are set as '01' and **ISR** last four bits are set as '0001' for bit error event as can be seen in Fig.5.40. These values indicate that bit error condition is detected and related registers are correctly modified. Bit error is only for transmission, for the case where 6 dominant bit is intruded by the error generator, it is interpreted as stuff error at the PEAK Analyzer data logger as can be seen in Fig.5.41, because after 5 consecutive dominant bit, 1 recessive bit must be driven. In other words bit error at transmitter side (FPGA) corresponds to stuff error at the receiver side (PEAK Analyzer) due to error generation algorithm of the analyzer.

RECEIVE ERROR TESTS

- **Stuff Error:** Error Generator of PCAN Analyzer is set to generate error starting from 40th bit and driving 6 consecutive dominant bits hence causing stuff error. For this error case, last two bits of **RESR (Receive Error Status Register)** are set as '01' last two bits of **ISR** are set as '10' as can be seen in Fig.5.42. These values indicate that stuff error condition is detected and related registers are correctly modified. Furthermore, stuff error is logged at PCAN's data logger as can be seen in Fig.5.43.
- **CRC Error:** With trial and error, the bit which the error generator will destroy to obtain CRC Error is determined as 550th bit and CRC error is generated. For this error case, last two bits of **RESR** are set as '10' last two bits of **ISR** are set as '10' as can be seen in Fig.5.44. These values indicate that stuff error

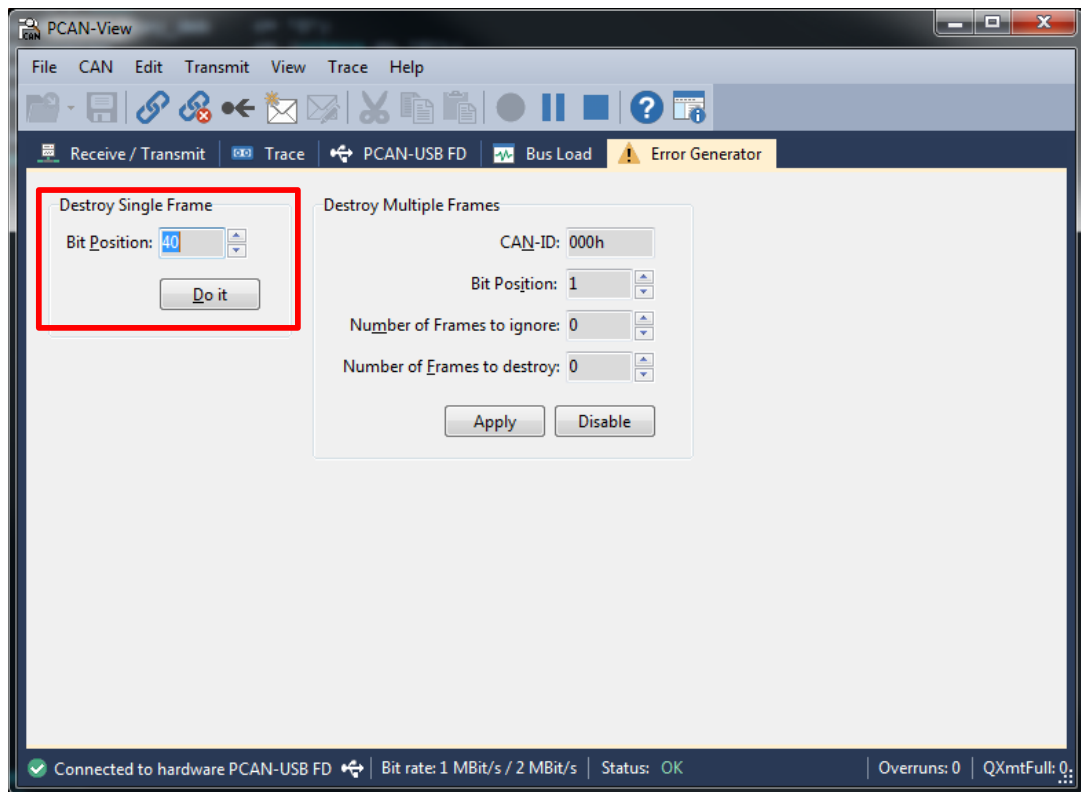


Figure 5.39: PCAN Error Generator

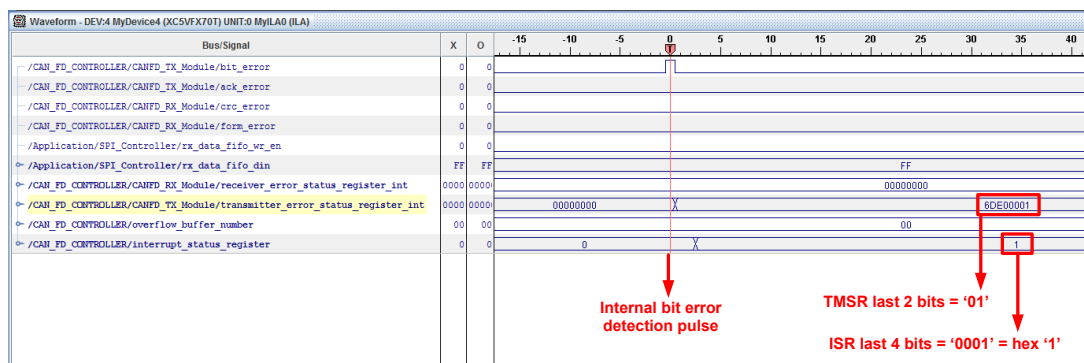


Figure 5.40: Bit Error Register Content

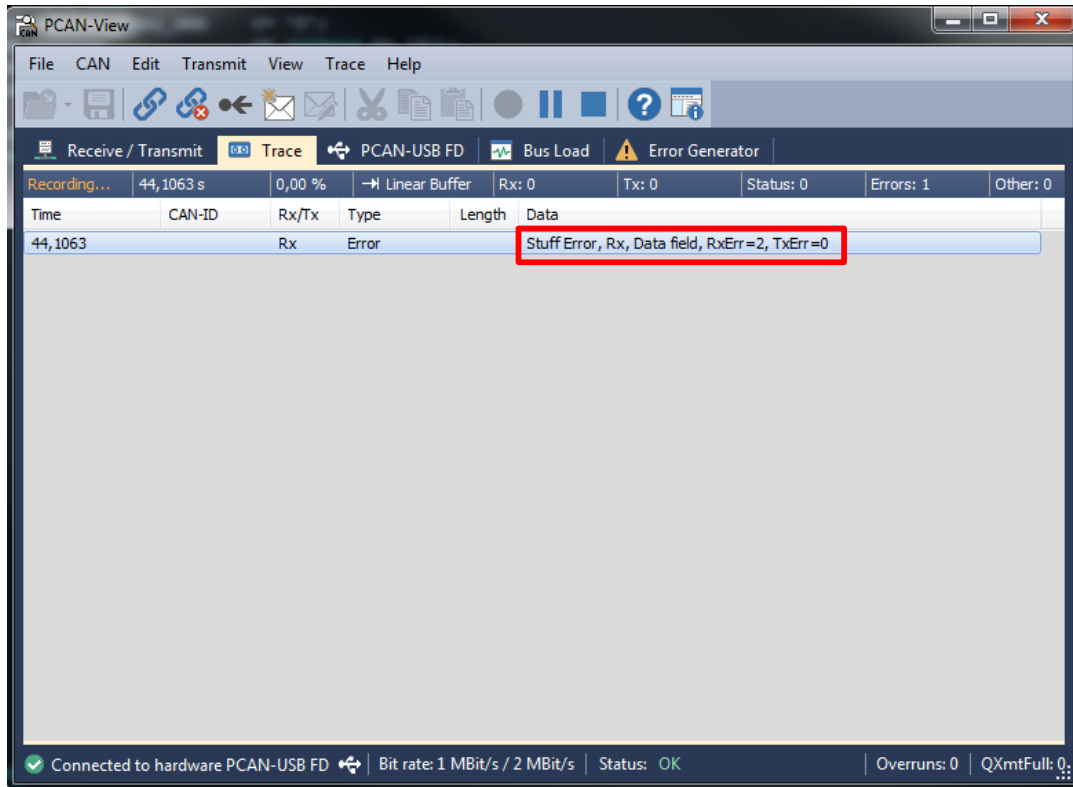


Figure 5.41: PCAN Error Log

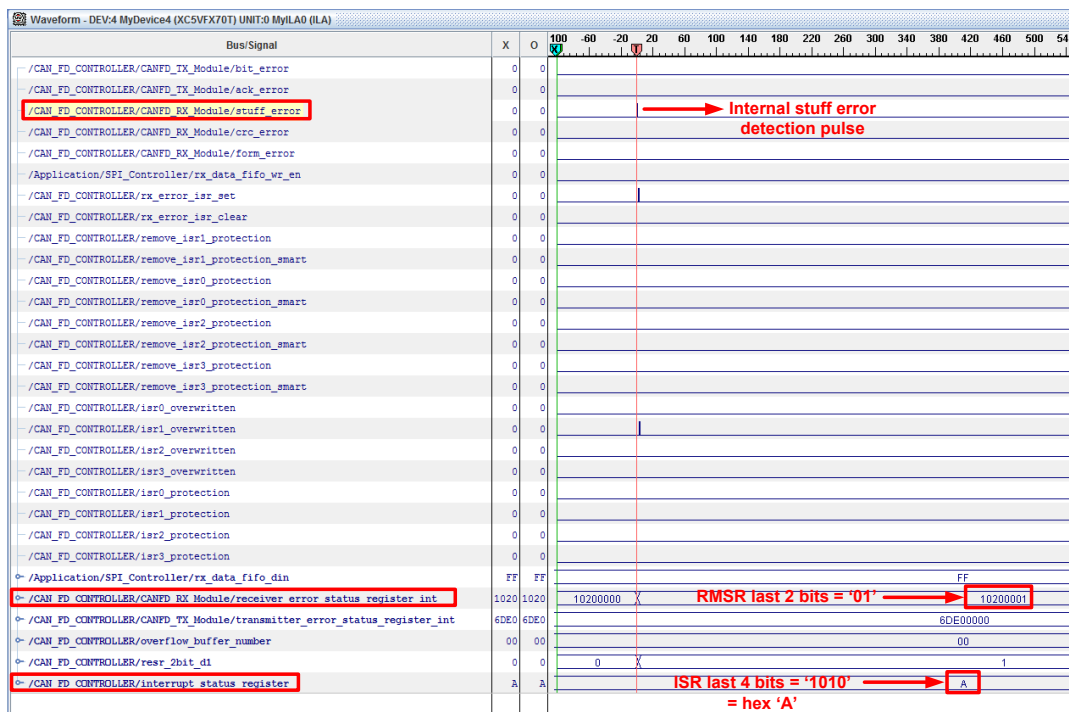


Figure 5.42: RX Stuff Error Register Content

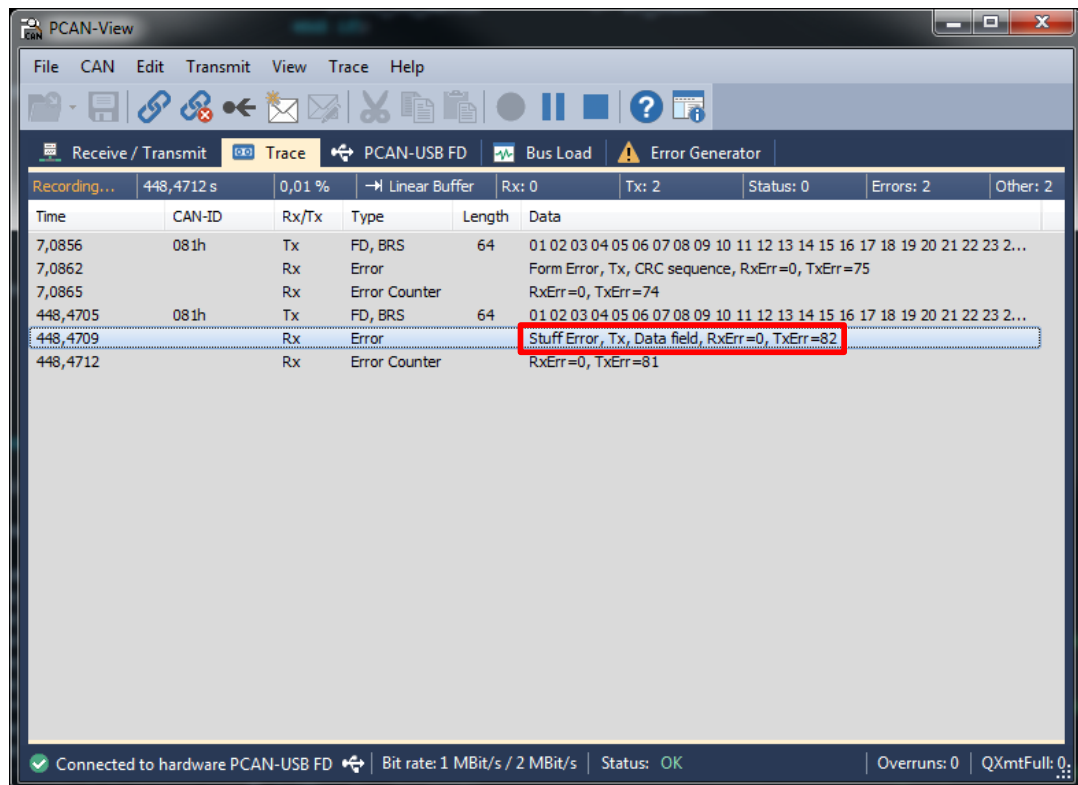


Figure 5.43: PCAN Error Log

condition is detected and related registers are correctly modified.

- **Form Error:** With trial and error, the bit which the error generator will destroy to obtain form error is determined and form error is generated such that acknowledgment delimiter is destroyed. For this error case, last two bits of **RESR** are set as '11' last two bits of **ISR** are set as '10' as can be seen in Fig.5.45. These values indicate that stuff error condition is detected and related registers are correctly modified. Furthermore, CRC error is logged at PCAN's data logger as can be seen in Fig.5.46.

5.7 Arbitration and Other Tests

Arbitration Test:

The setup whose block diagram shown in Fig.5.47 is set.

- 2 x FPGA CAN FD Controllers + Applications are implemented in the FPGA.

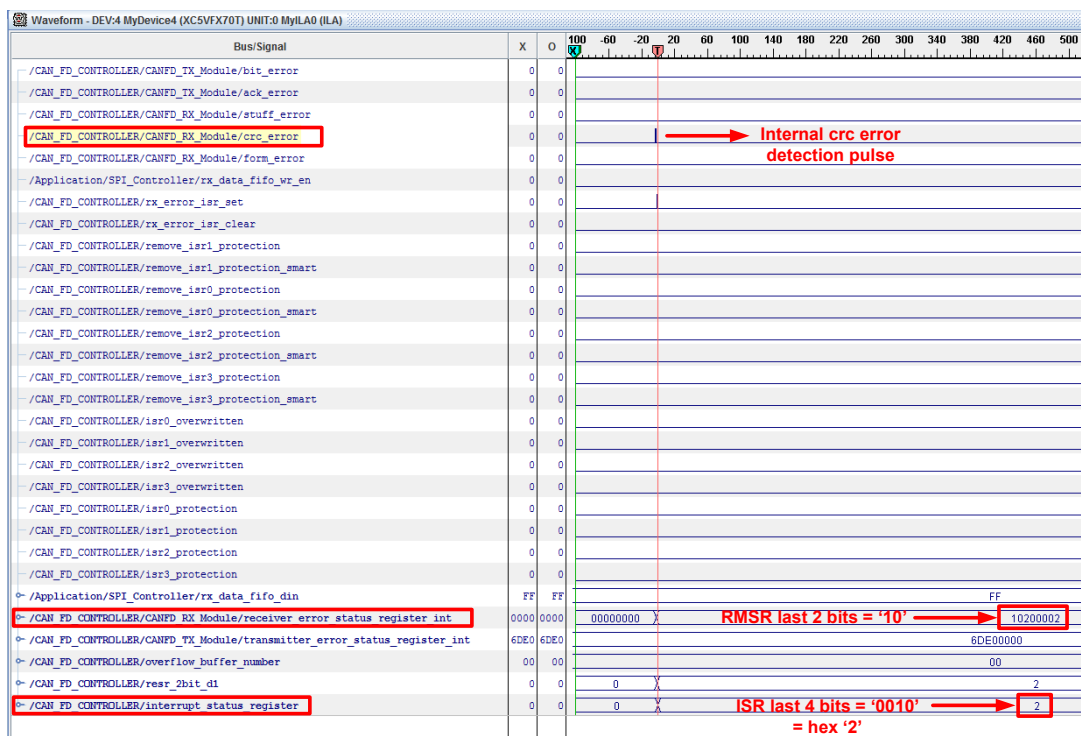


Figure 5.44: RX CRC Error Register Content

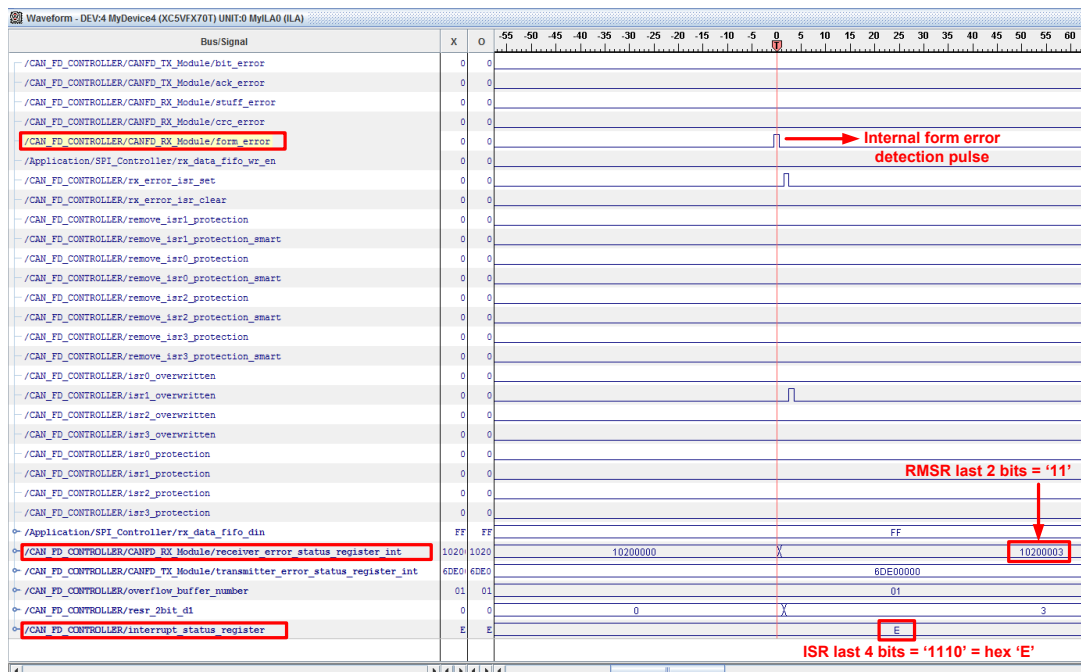


Figure 5.45: Form Error Register Content

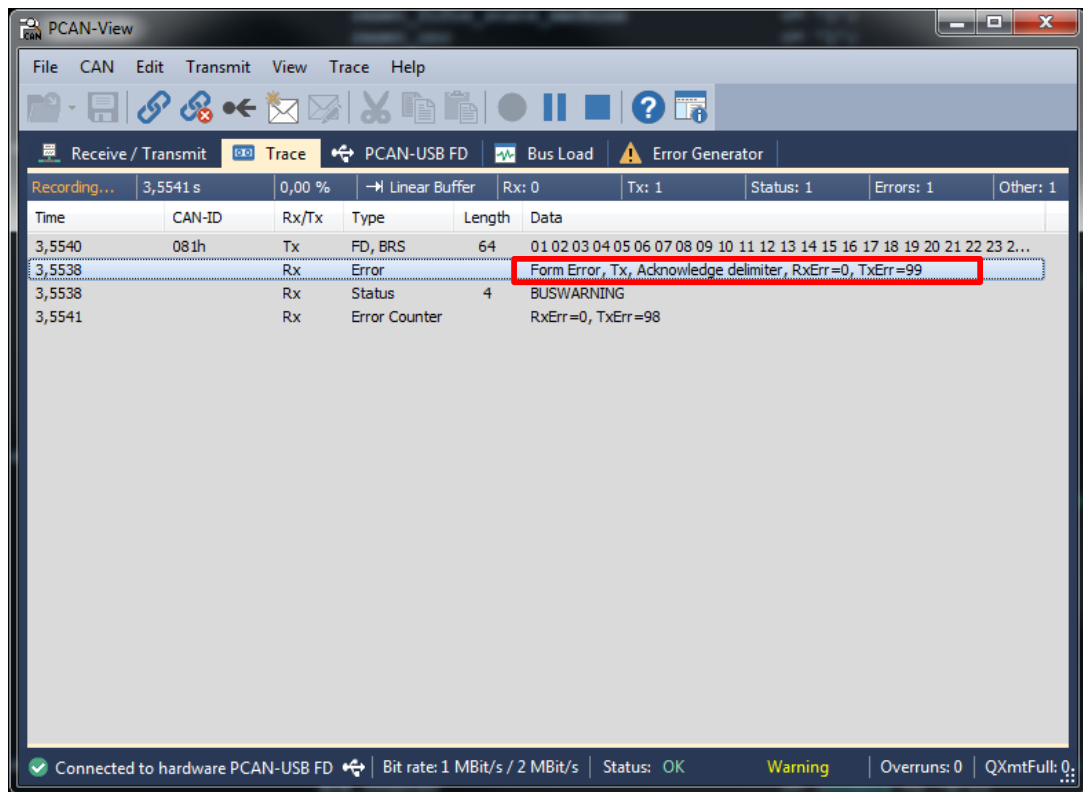


Figure 5.46: PCAN Form Error Log

Each node is connected to a CAN FD Transceiver. These two transceivers and also CAN FD Analyzer are connected to the CAN FD bus.

- A button on the FPGA demo board is configured such that when it is pressed each application will start transmitting messages with 36F and 36E ID respectively such that there is 100 ns time between the transmissions.
- Host Simulators' Initialization files are configured accordingly for this purpose.
- Node 1 has ID **36F** and Node 2 has ID **36E**.
- The cases where the message with the ID **36E** is transmitted first and the message with the ID **36F** is transmitted first are tested. For both of these two cases, the time between the first message and the second message is 100 ns.
- It is observed that the message with the ID **36E** is received first on the analyzer side and, the message with the ID **36F** loses arbitration and is transmitted with the next attempt and is received after the message with the ID **36E** at the analyzer.

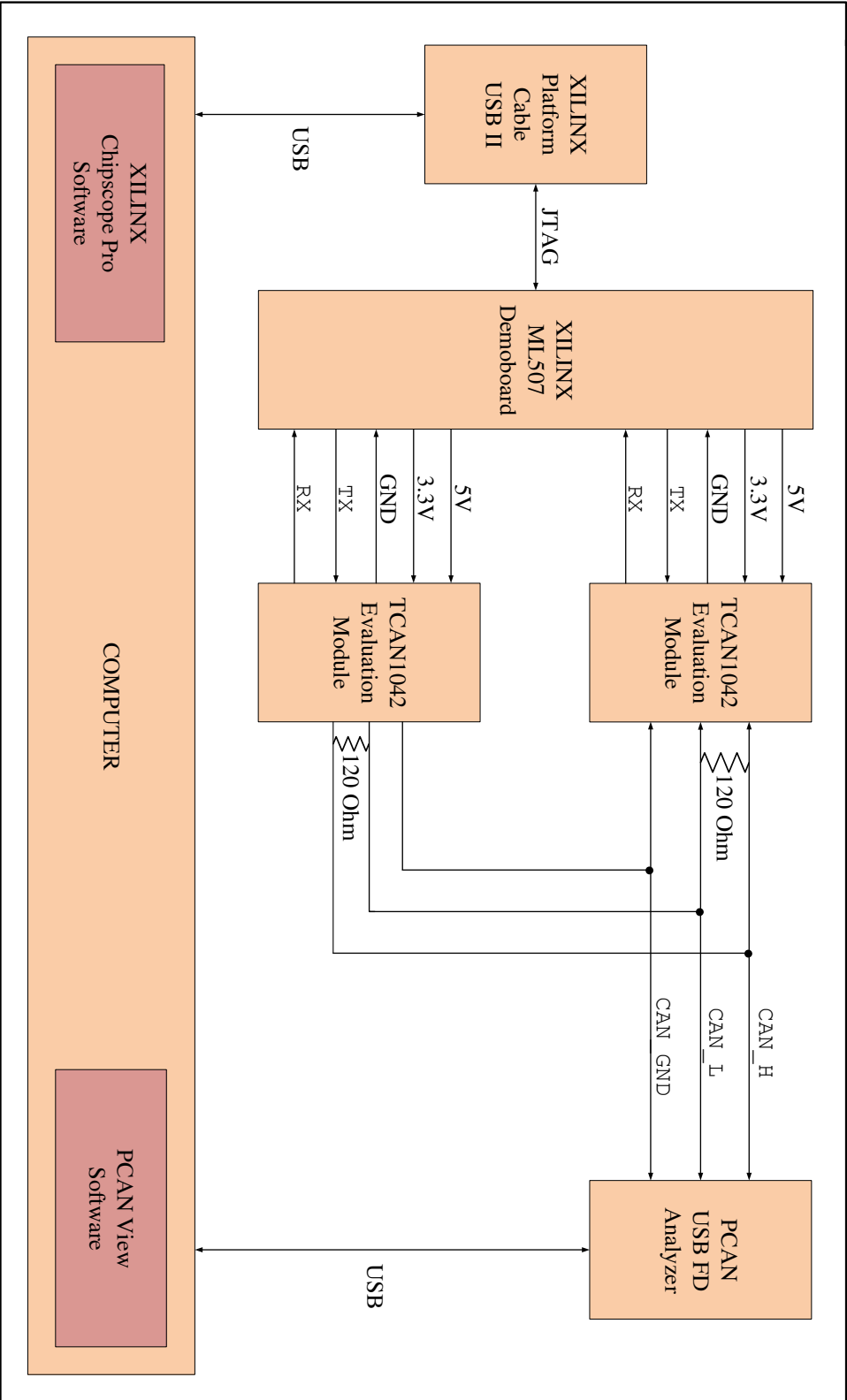


Figure 5.47: Arbitration Test Setup Block Diagram

- Node 1 always loses arbitration as can be seen in Fig.5.48.
- No matter which one is sent first, the message with the ID of **36E** is received first at the analyzer and the message with the ID of **36F** is received later. Two cases are shown in Fig.5.49a and Fig.5.49b. Furthermore, these messages have different data content.
- Arbitration mechanism is tried with both the short cable and 40m cable. The mechanism is verified for both of the cable types.
- Arbitration test Setup with a 40 meters CAN FD Bus cable can be seen in Fig.5.50.

Buffer Overflow Test: A messages which will be placed to Buffer 1 is transmitted twice and overflow condition is generated. **ISR** last 8 bit is read as **0x1C**, which means a message is received, an overflow condition occurred and the number of the buffer where overflow occurred is the buffer number 1. The content of **ISR** during overflow event and the content of **ISR** being read by the host simulator can be seen in Fig.5.51 and Fig.5.52 respectively.

Bus Length Test: CAN FD specification indicates that with 1 Mbit arbitration phase baud rate, bus length can be up to 40 meters. There are some factors which affect bus speed. First of all, CAN arbitration mechanism limits the bus length because during the acknowledgment bit or at the beginning of the arbitration phase where the nodes compete with each other to take over the bus, the bit transmitted by a node should propagate to every other node in the network system and the response of the nodes must propagate back to the transmitter node in a single bit time. Furthermore, every node in a CAN FD network introduces a stub which causes signal reflection and reducing the signal integrity. The number of the nodes and their stub length also affect the baud rate of the CAN FD bus. Moreover, CAN FD data phase baud rate is independent from the bus length since the communication is one way and no response is required from the other nodes like in the arbitration phase. But the arbitration phase baud rate depends on the factors explained here.

In order to verify this bus length requirement, test setup with a 40 meters cable is used. Successful transmissions and receptions as well as arbitration mechanism tests

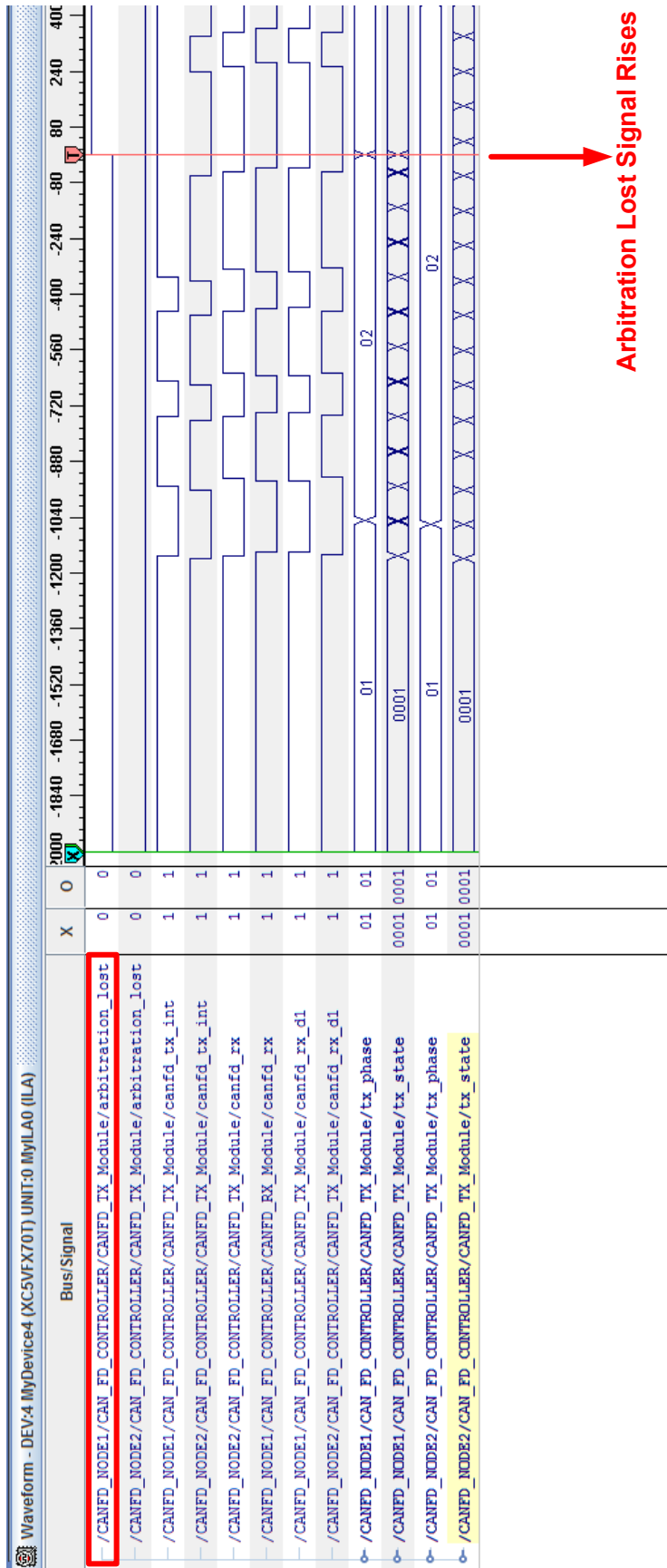
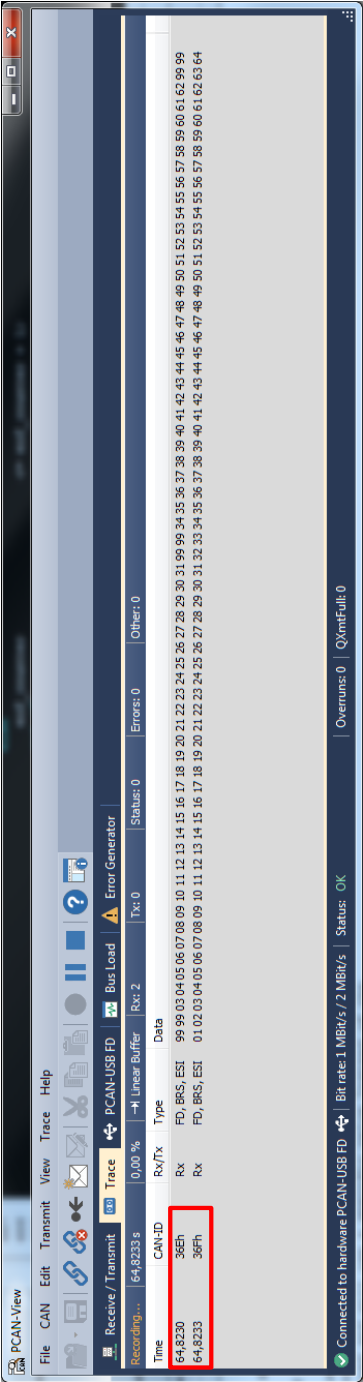


Figure 5.48: Arbitration Loss



(a) Message Reception Order for Case 1



(b) Message Reception Order for Case 2

Figure 5.49: Message Reception Orders

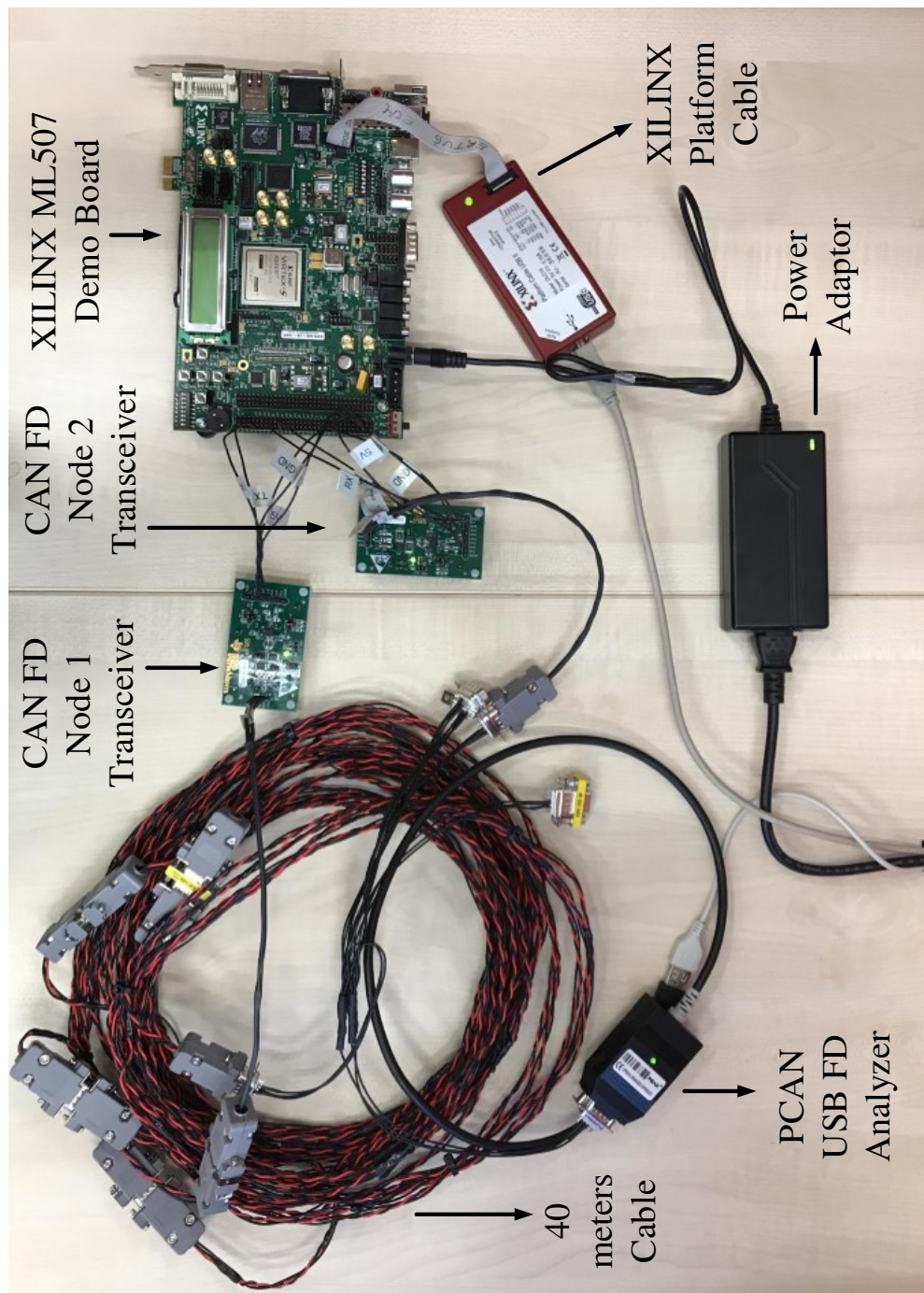


Figure 5.50: Arbitration Setup

| Waveform - DEV4 MyDevice4 (XC5VFX70T) UNIT:0 MyILA0 (ILA) | | | | | | | | | |
|---|------|------|---|----|---|----|---|----|---|
| Bus/Signal | X | O | 0 | 5 | 0 | 5 | 0 | 5 | 0 |
| - /CAN_FD_CONTROLLER/remove_isr0_protection | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/remove_isr0_protection_smart | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/remove_isr2_protection | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/remove_isr2_protection_smart | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/remove_isr3_protection | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/remove_isr3_protection_smart | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr0_overwritten | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr1_overwritten | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr2_overwritten | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr3_overwritten | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr0_protection | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr1_protection | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr2_protection | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/isr3_protection | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/CANFD_EX_Module/reset_fifo... | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/CANFD_EX_Module/bus_idle | 1 | 1 | | | | | | | |
| - /CAN_FD_CONTROLLER/CANFD_EX_Module/frame_recap... | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/CANFD_EX_Module/reset_fifo | 0 | 0 | | | | | | | |
| - /Application/SPI_Controller/rx_data_fifo din | FF | FF | | FF | | FF | | FF | |
| - /CAN_FD_CONTROLLER/CANFD_EX_Module/receiver_er... | 1020 | 1020 | | | | | | | |
| - /CAN_FD_CONTROLLER/CANFD_TX_Module/transmitter... | 6DE0 | 6DE0 | | | | | | | |
| - /CAN_FD_CONTROLLER/overflow_buffer_number | 01 | 01 | | | | | | | |
| - /CAN_FD_CONTROLLER/resr_2bit_d1 | 0 | 0 | | | | | | | |
| - /CAN_FD_CONTROLLER/interrupt_status_register | C | C | | | | | | | |

Figure 5.51: Overflow Register Content

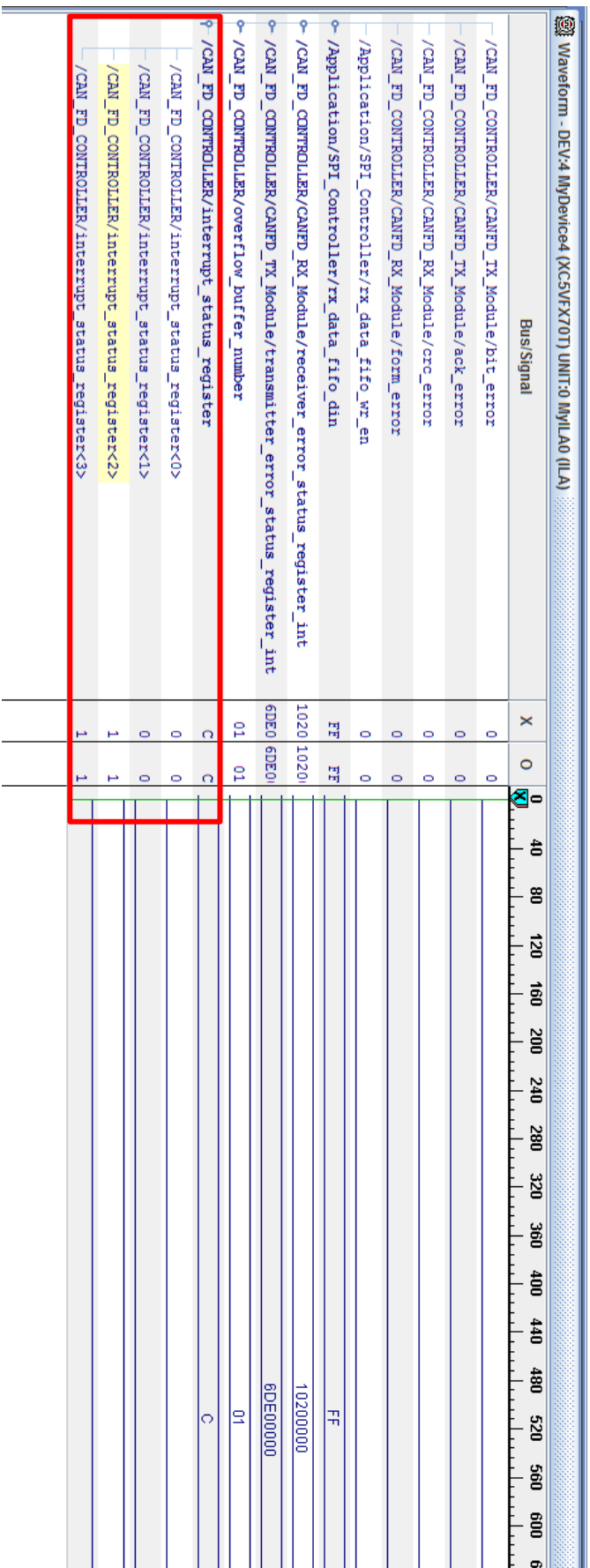


Figure 5.52: Overflow ISR Read Operation

are successfully done. Both of the bus ends should have 120 ohm termination resistors for the bus to function properly.

CHAPTER 6

CONCLUSION

This thesis presents the design, implementation and evaluation of C^3 : Configurable CAN FD Controller. C^3 features a total of 192 buffers which are organized as fully configurable 96 TX and 96 RX buffers. The buffers are organized as mailboxes and the sizes of the mailboxes can be up to 64 bytes. Application time configurable size for each mailbox gives the user great flexibility and convenience during the software development for a CAN FD network. Furthermore, total size of the memory can also be modified during synthesis. Therefore, it can easily be said that the buffers are fully configurable in every way. The mailbox structure used in C^3 instead of FIFO helps the response time to be lower since the higher priority buffers can always be accessed first. This is critical for tightly scheduled networks. Each RX buffer has a dedicated message acceptance filter to reduce the work load of the host MCU. C^3 implements CAN FD frames with both the base and the extended ID according to non-ISO CAN FD protocol specification. The CAN FD baud rate is 2 Mbps.

C^3 is configured and controlled through a defined register set by the host MCU. A communication protocol is developed over SPI for any MCU to be able interface with C^3 easily. To be able to perform the functional verification and timing performance tests of the designed controller hardware, a host MCU is required. For this purpose a simulator which acts like a host MCU is developed in the FPGA platform. Host simulator design includes a SPI master protocol block implementation and a block memory which includes SPI frames data content to configure, control and command C^3 . The block memory cells are initialized with the predefined data. For each test performed, the number of SPI frames in block memory and the SPI frame data content change.

The host simulator gets the data from the block memory and transmits SPI frames according to the designed SPI protocol. The host application varies according to the block memory content. FPGA implementation includes C^3 along with the host simulator. FPGA clock frequency is 100 MHz. Number of slice registers used is 5029 (11% utilization). The number of slice LUTs is 11541 (utilization is 25%) and the number of LUT flip flops is 3320 (utilization is 25%). Overall resource utilization can be considered 25%. Resource consumption is low however due to clock frequency being high, 100 MHz, some timing problems have been encountered during the design phase. With the strategies explained in Sec.4.8, timing problems have been solved and the design is implemented successfully.

The tests are performed to verify the functionality of the controller. The buffer configurability for different CAN FD message sizes, transmission and reception of the messages according to non-ISO CAN FD specification are verified. Furthermore, the host MCU simulator successfully configures the ID and the message size registers for each TX/RX mailbox and the ID & mask register pairs of accepting filters for RX mailboxes. It successfully requests message transmissions, reads the received messages from the desired mailboxes when an interrupt is received. For the performed tests, a professional CAN FD analyzer is used to transmit messages to C^3 or monitor the messages C^3 transmits. Error conditions implemented in C^3 are tested by generating errors via CAN FD analyzer's error generation function. Arbitration mechanism is tested by implementing one more CAN FD controller in the FPGA.

The response time measurements are performed to evaluate the timing performance of the designed controller. Detailed measurements are provided and compared with the theoretical results. Measured transmit response time is 61,48 μ s, which is much smaller than the MCU application delays, jitters and message periods. Furthermore, the lowest message periods in vehicle applications is close to 5 ms as described in Sec.4.7. Measured receive response time is 91,26 μ s, which is quite smaller than the message duration that is 318 μ s. This indicates that the received message is taken from C^3 more than quickly enough before the next message comes. Overall response times for both the receive and transmit cases provide excellent timing performance.

C^3 : Configurable CAN FD Controller is based on Bosch non-ISO CAN FD protocol

specification. Current development is only for academic purposes. C^3 is an open hardware platform and it can be used with any MCU having an SPI interface. SPI interface is the communication protocol which defines the register read and write operations between C^3 and the host MCU. The host simulator hardware is also designed in the scope of this thesis. Our next step is to develop a driver for C^3 and begin software development using this driver to perform the scheduling and frame packing algorithms for this brand new CAN FD protocol.

REFERENCES

- [1] ARASAN CAN FD Controller IP Core. <https://www.arasan.com/products/can-fd/>. Accessed on 2016.
- [2] BOSCHCAN IP Core. http://www.bosch-semiconductors.com/en/automotive_electronics/ip_modules/can_ip_modules/in_vehicle_communication.html. Accessed on 2016.
- [3] CAN 2020: The future of CAN technology. <https://www.can-cia.org/news/cia-in-action/view/can-2020-the-future-of-can-technology/2016/3/21/>. Accessed on 2017.
- [4] CAN-CTRL CAN 2.0 CAN FD Bus Controller Core. <http://www.cast-inc.com/ip-cores/interfaces/can-ctrl/index.html>. Accessed on 2017.
- [5] CAN FD v1.0 LogiCORE IP Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/canfd/v1_0/pg223-canfd.pdf. Accessed on 2017.
- [6] CAN with Flexible Data-Rate Specification version v1.0, Robert Bosch GmbH. <https://can-newsletter.org/assets/files/ttmedia/raw/e5740b7b5781b8960f55efcc2b93edf8.pdf>. Accessed on 2016.
- [7] CAN/CAN FD IP CORE. <http://www.kuantek.com.tr/ipcekirdek/KNTK-IP-CANFD-100-PB.pdf>. Accessed on 2016.
- [8] DCAN FD. <https://www.dcd.pl/ipcore/131/dcan-fd/>. Accessed on 2017.
- [9] IFI CAN FD IP Core. http://www.ifi-pld.de/IP/CANFD/body_canfd.html. Accessed on 2017.
- [10] IPMS CAN ISO CAN FD, CAN 2.0B CONTROLLER CORE. <https://www.ipms.fraunhofer.de/content/dam/ipms/common/products/WMS/canfd-e.pdf>. Accessed on 2017.
- [11] ISO CAN FD OR NON ISO CAN FD. https://can-newsletter.org/engineering/standardization/141209_iso-can-fd-or-non-iso-can-fd. Accessed on 2017.
- [12] LogiCORE IP XPS Controller Area Network (CAN) (v3.01a). https://www.xilinx.com/support/documentation/ip_documentation/xps_can.pdf. Accessed on 2015.
- [13] MCP2515). <http://www.microchip.com/wwwproducts/en/en010406>. Accessed on 2015.

- [14] Mercedes W140: First car with CAN. https://can-newsletter.org/engineering/applications/160322_25th-anniversary-mercedes-w140-first-car-with-can. Accessed on 2017.
- [15] ML505/ML506/ML507 Evaluation Platform User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf. Accessed on 2017.
- [16] NXP In Vehicle Networking. <https://www.nxp.com/docs/en/brochure/BRINVEHICLENET.pdf>. Accessed on 2017.
- [17] Renesas provides chips for Toyota. https://can-newsletter.org/engineering/applications/171114_17-4_renesas-provides-chip-for-toytas-self-driving-cars_renesas. Accessed on 2017.
- [18] Synective Labs IP for CAN and CAN FD. <http://www.synective.se/index.php/solutions/ip-for-can-and-can-fd/>. Accessed on 2017.
- [19] TCAN1042 Evaluation Module. <http://www.ti.com/tool/tcan1042devm>. Accessed on 2017.
- [20] The FlexCAN with CAN-FD. https://www.silvaco.com/products/IP/flexcan_with_can_fd/index.html. Accessed on 2017.
- [21] M. E. Afşin, K. W. Schmidt, and E. G. Schmidt. A Configurable CAN FD Controller: Architecture and Implementation. In *2017 25th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, May 2017.
- [22] M. E. Afsin, K. W. Schmidt, and E. G. Schmidt. C3: Configurable CAN FD Controller: Architecture, Design and Hardware Implementation. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9, June 2017.
- [23] H. S. An and J. W. Jeon. Analysis of CAN FD to CAN message routing method for CAN FD and CAN gateway. In *2017 17th International Conference on Control, Automation and Systems (ICCAS)*, pages 528–533, Oct 2017.
- [24] G. Cena, I. C. Bertolotti, T. Hu, and A. Valenzano. Improving compatibility between CAN FD and legacy CAN devices. In *2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 419–426, Sept 2015.
- [25] B. Cheon and J. W. Jeon. The CAN FD network performance analysis using the CANoe. In *IEEE ISR 2013*, pages 1–5, Oct 2013.
- [26] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, April 2007.

- [27] A. Happel. High-Speed Reprogramming and Calibration with CAN FD: A Case Study. https://can-cia.org/fileadmin/resources/documents/proceedings/2013_decker.pdf. Accessed on 2017.
- [28] F. Hartwich. CAN with Flexible Data Rate. In *The international CAN Conference (iCC)*, 2012.
- [29] S. J. Jang and J. W. Jeon. Software reprogramming performance analysis of can fd and flexray protocols. In *2015 IEEE International Conference on Information and Automation*, pages 2535–2540, Aug 2015.
- [30] D. A. Khan, R. J. Bril, and N. Navet. Integrating hardware limitations in can schedulability analysis. In *2010 IEEE International Workshop on Factory Communication Systems Proceedings*, pages 207–210, May 2010.
- [31] T. Kugelstadt. Isolated CAN Transceiver Assures Robust Fieldbus Design. <https://www.ecnmag.com/article/2009/10/isolated-can-transceiver-assures-robust-fieldbus-design>. Accessed on 2017.
- [32] T. Lindenkreuz. CAN FD – CAN with Flexible Data Rate, Vector Kongress 2012. https://vector.com/portal/medien/cmc/events/commercial_events/VectorCongress_2012/VeCo12_8_NewBusSystems_3_Lindenkreuz_Lecture.pdf. Accessed on 2017.
- [33] R. Lotoczky. CAN-FD Flexible Data Rate CAN An Abbreviated primer. https://vector.com/portal/medien/vector_cantech/Congress2013/1_9_CAN%20FD_Update.pdf. Accessed on 2016.
- [34] Marco Di Natale, Haibo Zeng, Paolo Giusto, Arkadeb Ghosal. *Understanding and Using the Controller Area Network Communication Protocol*. Springer-Verlag New York, 1 edition, 2012.
- [35] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proceedings of the IEEE*, 93(6):1204–1223, June 2005.
- [36] O. Pfeiffer, A. Ayre, and C. Keydel. *Embedded Networking with CAN and CANopen*. Copperhill Media Corporation, rev. 1. ed edition, 2008.
- [37] K. Schmidt, B. Alkan, E. G. Schmidt, D. C. Karani, and U. Karakaya. Controller area network (CAN) with priority queues and fifo queues: Improved schedulability analysis and message set extension. *International Journal of Vehicle Design*, 71(1/2/3/4), 2015.
- [38] K. W. Schmidt. Robust priority assignments for extending existing controller area network applications. *IEEE Transactions on Industrial Informatics*, 10(1):578–585, Feb 2014.
- [39] J. W. Shin, J. H. Oh, S. M. Lee, and S. E. Lee. Live demonstration: CAN FD controller for in-vehicle network. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 748–749, Oct 2016.

- [40] A. K. Sinha and S. Saurabh. CAN FD: Performance reality. In *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*, pages 1–6, Feb 2017.
- [41] G. M. Zago and E. P. de Freitas. A Quantitative Performance Study on CAN and CAN FD Vehicular Networks. *IEEE Transactions on Industrial Electronics*, 65(5):4413–4422, May 2018.