# INTER-CONNECTED FLEXRAY AND CAN NETWORKS FOR IN-VEHICLE COMMUNICATION: GATEWAY IMPLEMENTATION AND END-TO-END PERFORMANCE STUDY

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

BY

MELİH ALKAN

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

MAY 2010

Approval of the thesis:

**INTER-CONNECTED FLEXRAY AND CAN NETWORKS FOR IN-VEHICLE COMMUNICATION: GATEWAY IMPLEMENTATION AND END-TO-END PERFORMANCE STUDY**

submitted by **MELİH ALKAN** 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. Canan Özgen __________

Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. İsmet Erkmen __________

Head of Department, **Electrical and Electronics Engineering**

Asst. Prof. Dr. Şenan Ece Schmidt __________

Supervisor, **Electrical and Electronics Engineering Dept., METU**

**Examining Committee Members:**

Prof. Dr. Semih Bilgen ____________________

Electrical and Electronics Engineering Dept., METU

Asst. Prof. Dr. Şenan Ece Schmidt ____________________

Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Özgür Barış Akan ____________________

Electrical and Electronics Engineering Dept., METU

Asst. Prof. Dr. Cüneyt Bazlamaçcı ____________________

Electrical and Electronics Engineering Dept., METU

Emrah Yürüklü, M.Sc. ____________________

TOFAŞ Turkish Automobile Factory A.Ş.

**Date:** 13.05.2010

# ABSTRACT

## INTER-CONNECTED FLEXRAY AND CAN NETWORKS FOR IN-VEHICLE COMMUNICATION: GATEWAY IMPLEMENTATION AND END-TO-END PERFORMANCE STUDY

Alkan, Melih

M. S., Department of Electrical and Electronics Engineering
Supervisor: Asst. Prof. Dr. Ece Ş. Güran Schmidt

May 2010, 265 pages

The increasing use of electronic components in today's automobiles demands more powerful in-vehicle network communication protocols. FlexRay protocol, which is expected to be the de-facto standard in the near future, is a deterministic, fault tolerant and fast protocol designed for in vehicle communication. The current de-facto in-vehicle communication standard, CAN, and the future in-vehicle communication standard FlexRay will exist together in future cars. Data exchange between these two standards will be performed via Gateway units. In this thesis, end-to-end performance of a FlexRay-CAN network connected by a Gateway is evaluated as well as Gateway functionality and processing delay. The results of the experiments, which are performed for a realistic message set with various scheduling schemes, are presented and discussed.

Keywords : in-vehicle communication, FlexRay, Gateway, end-to-end performance

# ÖZ

ARAÇ İÇİ HABERLEŞME İÇİN BİRBİRİNE BAĞLI FLEXRAY VE CAN AĞLARI: AĞ GEÇİDİ (GATEWAY) UYGULAMASI VE UÇTAN UCA BAŞARIM ÇALIŞMASI

Alkan, Melih

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü

Tez Yöneticisi : Y. Doç. Dr. Ece Ş. Güran Schmidt

Mayıs 2010, 265 sayfa

Günümüz otomobillerinde artan elektronik birim kullanımı daha güçlü araba içi haberleşme protokollerine olan ihtiyacı doğurmaktadır. FlexRay protokolü ortaya çıkan bu ihtiyacı karşılayabilecek özelliklere sahip, kararlı, hatalara dayanıklı ve hızlı bir haberleşme protokolüdür. Bugünün defakto araç içi haberleşme protokolü CAN ve geleceğin defakto araç içi haberleşme protokolü FlexRay gelecekte, otomobillerde eş zamanlı olarak yer almaları beklenmektedir. Bu iki ağ arasındaki veri haberleşmesi ağ geçiti (gateway) birimleri ile gerçekleştirilecektir. Bu tezde, Ağ Geçidi ile bağlanmış FlexRay-CAN ağlarının uçtan uca başarımı ve aynı zamanda Ağ Geçidi işlem süresi ve çalışırlığı değerlendirilmiştir. Gerçekçi mesaj kümesi ile çeşitli çizelgeleme yaklaşımlarına göre gerçekleştirilen deneylerin sonuçları sunulmuş ve tartışılmıştır.

Anahtar Kelimeler: araç içi haberleşme, FlexRay, Ağ Geçidi, uçtan uca başarım

*To My Family*

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Asst. Prof. Dr. Ece Ş. Güran Schmidt for her guidance, advice, criticism, encouragement, endless patience and insight throughout the completion of the thesis.

I wish to thank my company ASELSAN A.Ş for giving me the opportunity of continuing my thesis study and supporting me in my efforts to get the Master degree.

I owe a debt of gratitude to TOFAŞ Türk Otomobil Fabrikaları A.Ş. for their support and providing the hardware and the software that I used throughout the thesis.

I would like to forward my appreciation to all my friends and colleagues for the contribution to my thesis with their continuous guidance, advice, encouragement and for expanding my horizons.

Finally, my family. No word can suffice to express how I am grateful to my parents but at least I can mention my sincere gratitude to them for their unwavering support, continual confidence and endless and gratis love. I also wish to thank my beloved sister for her support, criticism and help on the format of my thesis.

# TABLE OF CONTENTS

# LIST OF FIGURES

## FIGURES

# LIST OF TABLES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ABS | : Antilock Braking System |
| API | : Application Programming Interface |
| ASC | : Automatic Stability Control |
| CAN | : Controller Area Network |
| CC | : Communication Controller |
| CCAL | : Communication Controller Abstraction Layer |
| CHI | : Controller Host Interface |
| ECU | : Electronic Control Unit |
| EPS | : Electric Power Steering |
| ESP | : Electronic Stability Program |
| FHAL | : FlexRay Hardware Abstraction Layer |
| FID | : FlexRay Identification |
| FTDMA | : Flexible Time Division Multiple Access |
| HAL | : Hardware Abstraction Layer |
| IPC | : Instrument Panel Cluster |
| LIN | : Local Interconnect Network |
| LIP | : Linear Integer Programming |
| MOST | : Media Oriented System Transport |
| NIT | : Network Idle Time |
| NRZ | : Nonreturn-to-Zero |
| PFR | : Port Function Register |
| QoS | : Quality of Service |
| SAS | : Steering Angle Sensor |
| TDMA | : Time Division Multiple Access |

# CHAPTER 1

# INTRODUCTION

Today, in modern automobiles, the most widely used network protocol for in-vehicle communication is the Controller Area Network (CAN) [31]. Although other communication schemes along with the CAN bus are also used, the CAN bus constitutes the main communication backbone in current automobiles. The requirement for a variety of distinct communication protocols arises from the necessity of running applications with different needs in terms of delay, jitter, bandwidth, message loss, integrity and Quality of Service (QoS). The controlling of wipers, lights, doors and windows, telematic functions such as car radio, DVD, navigation systems and rear seat entertainment, the functions for the electronic control of the engine such as ABS, ESP, ASC and numerous safety-critical functions to provide the control of suspension, steering and braking can be counted as the applications running in the modern cars so as to emphasize the diversity. Although CAN is well suited to be the main communication network for in-vehicle communication with such applications, CAN is evaluated to fall short for very near future applications such as x-by-wire which can, in short, be defined as the replacement of mechanical and hydraulic systems by completely electronic ones, due to its data rates between 50 Kbit/s and 1 Mbit/s and its event-triggered arbitration mechanism. The emerging FlexRay protocol with a much higher bandwidth of 10 Mbit/s and support for both time-triggered and event-triggered message traffic is expected to replace CAN as the new de-facto standard for in-vehicle communication [1]. However, since the technology transition from CAN to

FlexRay is not anticipated to happen at once, in the near future both network protocols are expected to appear together in an automobile. In this scheme, the applications requiring lower speed will still be carried out by CAN bus while new high-speed functionality which necessitates real time response will be implemented on FlexRay network. Consequently, this situation imposes the existence of a Gateway unit which facilitates the inter-communication among the nodes on CAN and FlexRay networks. As the CAN and the FlexRay networks are expected to exist together for a long time, so does the Gateway. Therefore, attention should be paid for the design of an efficient and high-performance Gateway. In this respect, a Gateway has to perform fast and correct protocol conversion between both networks. It must have the capability of processing the payload of the messages in the signal level and the gateway processing delay should be bounded with low variance.

## 1.1 TERMINOLOGY

Throughout the thesis, the terms that define the data carrying entities, *signal*, *message* and *frame* are used with different meanings.

*Signal* is the smallest meaningful piece of data exchanged in a network. Speed data, information from the sensors, indications of the display panel of an automobile can be given as the examples to the signals. In this context, the length of the signals can vary from 1-bit to tens of bytes depending on the application.

*Message* indicates the entire data that is exchanged in a single transmission. In this sense, the messages are composed of signals. Depending on the application, the messages can contain tens of signals as well as they might be composed of a single signal.

*Frame* is the protocol data exchange format for both CAN and FlexRay networks. The data payload of the frame is the message. Note that when the message in encapsulated in the frame the necessary encoding and bitstuffing operations are carried out.

## 1.2 CONTRIBUTIONS OF THESIS

The first contribution of this thesis is the design, implementation and performance evaluation of a FlexRay-CAN Gateway that is realized by microcontroller programming. The implementation is carried out on an evaluation board with Fujitsu microcontroller and built-in FlexRay and CAN controllers. We demonstrate that the Gateway can perform the correct protocol conversion by testing it in an interconnected FlexRay-CAN network where the CAN nodes are the components of a real vehicle and the FlexRay nodes are realized by evaluation boards with FlexRay hardware. In addition we demonstrate the capability of the Gateway to map the signals in the incoming messages to outgoing messages according to a given configuration.

The second contribution of the thesis is the experimental performance study of FlexRay and CAN networks that are interconnected by the designed Gateway. This study involves both local timing measurements on the Gateway node to calculate its processing delay and end-to-end delay and jitter where jitter indicates the deviation of the periodicity for the signal transmission from source node to destination node which are on different networks. The scheduling approaches followed for both CAN and FlexRay Networks affect the timing performance of the signals that are sent within a single network as well as the signals that are sent to the other network in the entire inter-connected network. In this respect we first investigate the timing performance of different scheduling algorithms for CAN and FlexRay networks. Next we demonstrate the end-to-end timing performance of selected scheduling approaches on a 7 node FlexRay-CAN network inter-connected by the Gateway.

In the literature, there is a small number of studies on FlexRay-CAN Gateway. However, these studies focus on the demonstration of the correct protocol conversion and measuring the Gateway processing time. The arbitrary signal mapping capability of the Gateway is not implemented. Furthermore to the best of our knowledge there is no previous study that presents the end to end delay and jitter measurements on an interconnected FlexRay-CAN network.

## 1.3 THESIS ORGANIZATION

The thesis is organized as follows. CHAPTER 2 describes the evolution of in-vehicle communication, the basic characteristics of CAN and FlexRay protocols together with some other in-vehicle networking schemes and discusses the previous work made on Gateway. In CHAPTER 3, we explain the design phases of a FlexRay-CAN Gateway. While we introduce the development and test tools that we have used throughout the study in CHAPTER 4, the implementation details of the Gateway, design of which is described in CHAPTER 3, are discussed in CHAPTER 5. Our experimental data and the complete discussion of the results are reported in CHAPTER 6. Finally, CHAPTER 7 summarizes the entire thesis work and provides concluding remarks.

# CHAPTER 2

# BACKGROUND

In the beginning of their invention, automobiles were described to be composed of purely mechanical and hydraulic systems. This description was valid until the beginning of 1970s [1]. Parallel with the developments in electronics after the introduction of transistor, share of the electronic parts and systems in the automobiles rapidly increased. The growing reliability and the performance of the electronic hardware components enabled improving the in-vehicle comfort as well as the safety. The very first adopted electronic components were mainly about comfort and utilities, like wipers, tapes, electric windows, lights and so on. As the time passed, safety-critical functions were implemented in the car, while more and more electronic components for entertainment and comfort were continued being introduced. Today, in most modern cars, safety-critical electronic systems and components like Antilock Braking System (ABS), Electronic Stability Program (ESP), Electric Power Steering (EPS), Airbags, Active Suspension, Engine Control and so forth, exist in their base configuration. As a result of this "*electronization*", signal exchange traffic in the automobiles is enormously increased. In today's luxury cars, up to 2500 signals are exchanged by up to 70 "Electronic Control Units" (ECU) [22].

In the beginning of automotive electronics, each function was implemented by stand alone ECUs which had their own microcontroller, sensors and actuators. Until the beginning of 1990s, signal exchange between ECUs was provided by point-to-point links. According to this strategy, $n^2$ communication channels were required, assumed that all ECUs interconnected and number of ECUs is *n*. In other words, the number of communication links were to grow with $n^2$. However, as the number of ECUs were increasing very rapidly by time, this approach fell short due to problems of weight and cost of ECUs and reliability, power consumption and complexity because of the interconnection of huge numbers of wires and connectors. The solution to these problems was to use communication networks in the automobiles to multiplex numerous signals over a shared medium between different ECUs. This new approach was effective and really flourished. It was mentioned that replacing the wiring in the four doors of BMW with Local Area Network reduced the weight by 15 kilograms [2]. On the other hand, when communication network was used, it was seen that the amount of required wires was reduced 40% from 635 to 370 in Peugeot 307 with regard to non-multiplexed Peugeot 306 [3]. Besides the reduction in weight and the number of wires required, it is obvious to conclude that, this wiring replacement would also reduce power consumption and space allocation for the wiring harness and improve reliability and complexity.

In the beginning, every manufacturer was developing its own network in its automobile. However, as the role of external component suppliers became important in automotive industry, cost of integration of new components to the in-vehicle communication networks increased and reliability deteriorated. This is because, external suppliers were required to adapt their components to different communication networks for different manufacturer's car. So, standardized and widely accepted defining rules, i.e protocols for signal exchange through a shared medium were needed to be defined. This need was fulfilled in mid-1980s by Bosch which developed Controller Area Network (CAN) [1]. Popularity of CAN increased very rapidly beginning with its first use in Mercedes cars in early1990s. Designed specifically for automotive applications, CAN became to be used in other areas such as industrial automation and medical equipment. In 2005 it is estimated that about 400 million CAN nodes (all application fields) were being sold per year [4]. Today, CAN has become most widely used communication network in automotive industry.

Since the introduction of CAN, new components, technologies and needs continued to emerge in automotive industry. All these new components were requiring new networking approaches. In modern cars, while for some applications, bounded delay and real time performance is required (like x-by-wire), for some other applications large bandwidth allocation is needed (like media applications). Since the performance demands and safety needs of all the functions embedded in the car are not the same, different Quality of Service (QoS like, bounded delay, jitter, bandwidth, redundancy of communication channels, efficient error detection mechanisms, so forth) are expected from communication networks. Therefore, today, apart from CAN bus, different communication network protocols exist together in a car. Among the some others, LIN (Local Interconnect Network), MOST (Media Oriented System Transport), FlexRay and CAN can be counted as the basic communication network system in a modern car.

An in-car embedded electronic system, typically, is divided into four functional domains which have different features, structure, QoS requirements and constraints, namely, powertrain, chassis, body and telematics [5].

Powertrain is mainly responsible for controlling the engine of the automobile. Powertrain domain involves with real time control functions and performs safety-critical operations. In order to cope with diversity of critical tasks to be performed which are concerned with the most important component of a car, engine, multitasking is required and stringent time scheduling constraints are imposed on the tasks. Furthermore, powertrain domain requires frequent data exchange with other car domains such as the chassis (ABS, ESP, ASC), and the body (dashboard, climate control).

Chassis domain is the other domain which is concerned with safety-critical functions in the car. The main function of the chassis domain is to provide the control of suspension, steering and braking. Some of the functions that are gathered under the chassis domain are ASC (Automatic Stability Control), ESP, ABS, 4WD (4 Wheel Drive) and so forth. Chassis domain is little bit more critical than powertrain in safety standpoint since functions in this domain have a stronger impact on vehicle's stability, dynamics and agility. Moreover, the x-by-wire technology, which is a generic term and means the replacement of mechanical and hydraulic systems by completely electronic ones, is being introduced to perform steering or braking functions. Although the x-by-wire technology has been being used in avionics industry, it is an emerging technology for automotive industry. Studies on this new technology leads to new design methods for developing the x-by-wire functions safely [6] and for preventing the interferences between functions [7]. Chassis domain mainly exchange signal with powertrain. Implementation of both powertrain and chassis domains moves toward a time-triggered approach rather event-triggered ones which will enhance the deterministic real-time behaviour of the system [8] [9]. As a result, CAN bus usage in these domains is being given up in favor of TTCAN or new emerging technology FlexRay.

Body domain composed of systems such as dashboard, wipers, lights, doors, windows, seats, mirrors, climate control and so on which are more and more began to be controlled by software based systems. Main characteristic of this domain is that the components of the system require to exchange numerous of messages with small piece of data and that the functions are mainly triggered by the passengers' or the driver's solicitation. Despite the event-triggered characteristics of the body functions, since most of the nodes do not require large bandwidth through communication, which is offered by CAN bus, a new low-cost network is required to be designed, namely Local Interconnect Network (LIN), to satisfy the requirements of the body domain.

Telematics is another domain existing in modern cars. Number of signals exchanged, number of functions implemented and bandwidth used in this domain have been increasing more and more rapidly nowadays. Functions like hands-free phones, car radio, CD, DVD, navigation systems, rear seat entertainment and so forth are, all, becoming standard in the automobiles today. Common property of these functions is that they require to exchange big amount of data within the car, even with the external world, when compared to other domains [10]. In this domain, rather than the messages subject to strict deadline scheduling, multimedia data streams, multimedia QoS, bandwidth sharing, integrity are considered to be more important. Both the data rate of CAN and LIN are inadequate to satisfy the requirements of telematics. Moreover, in this domain, once functions activated, signals are exchanged periodically instead of event-triggered signal exchange. Therefore, as the requirements of this domain are different than the other domains, data communication structure should also be different. In the modern automobiles, the telematics functions are exchanged through Media Oriented System Transport (MOST) networks.

## 2.1 IN-VEHICLE COMMUNICATION NETWORKS

In the following sections, the basic operating principles of some important communication networks implemented in modern cars will be described and their usage in automobile networks will be explained briefly. Although we, mainly, are interested in CAN and FlexRay network protocols in the scope of this study, the basic characteristics of LIN and MOST networks are also included by keeping the discussion short for the sake of the completeness of the literature.

### 2.1.1 CAN (Controller Area Network)

The CAN bus was developed by Robert Bosch GmbH as a multi-master, message broadcast system that specifies a maximum signaling rate of 1 Mbits/s [11]. It was designed initially to be used in automotive industry. While, today, CAN is the most popular field bus used in various fields like electric power, petroleum, chemical, metallurgical, steel and transport industry, CAN is widely applied in automotive industry, aviation industry, industrial control and security protection [12].

As mentioned above CAN communication is a multi-master protocol. That is to say, every individual node connected to CAN bus can send message whenever they find the medium idle. With this regard, CAN communication is attributed as event-triggered communication. Conflicts may occur, as accessing the shared medium, if two or more nodes try to send message at the same time when they see that the medium is idle. In the case of conflict, arbitration is done via the identifiers of the messages. The identifiers specify the priority of the message. The lower the identifier is, the higher the priority. This scheme is the result of the arbitration mechanism performed throughout the bus. According to this arbitration mechanism, in the physical layer of CAN protocol, logic (HIGH or LOW) provided by each node are ANDed and this ANDed logic is put on the wire of CAN bus. Therefore, if a node sending a HIGH logic level (which is recessive) sees a LOW logic level on the bus, the node understands that there exists at least an other node with a smaller identifier which means higher priority and immediately stops its transmission. CAN arbitration procedure relies on the fact that the sending node monitors the bus while transmitting. The Figure 2-1 is illustrating the CAN arbitration mechanism.

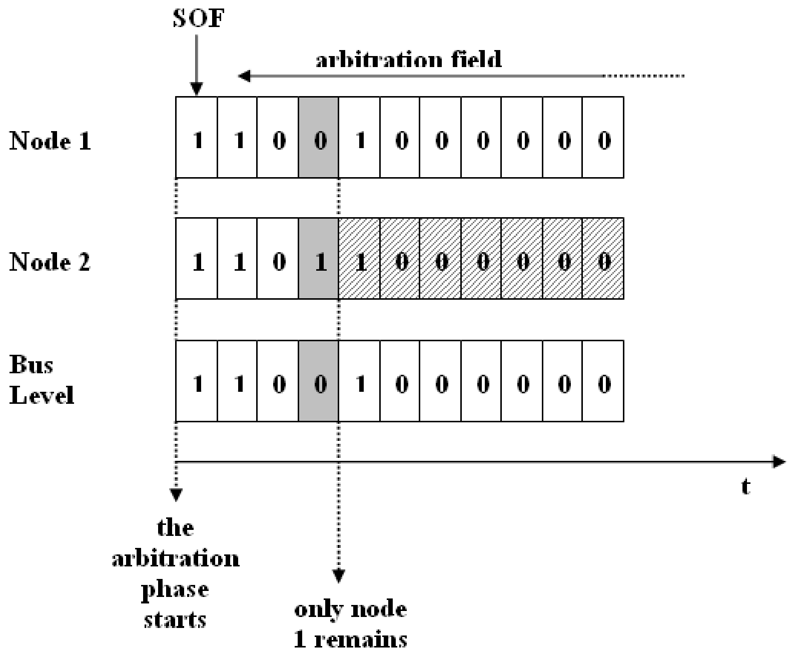


**Figure 2-1** CAN Bus Arbitration Scheme[14]

This arbitration mechanism affects the feasible communication data rate and communication distance. Throughout the arbitration process, before a bit value is decided along the bus, the signal must propagate to the most remote node and return back. Therefore, by taking the speed of signal through the wires into consideration, 1 Mbps rate is achieved on a 40m bus at maximum, whereas 250 kbps rate is feasible over 250m.

CAN with a physical layer implementation on a twisted pair of copper wires became an ISO standard in 1994 [13] and found a wide range of application area in automotive industry due to its low cost, robustness and fairly bounded delay for its applications [4]. There exists, today, two versions of CAN protocol differing in size of identifiers. CAN 2.0A is the *"Standard"* CAN protocol with 11-b identifier and CAN 2.0B is the *"Extended"* CAN protocol with 29-b identifier. Up to now, Standard CAN protocol was sufficient for in-vehicle communication since it facilitated $2^{11}$ messages to be sent via CAN bus. However, developments in automotive electronics and upcoming needs were taken into consideration, the usage of Extended CAN in automobiles will be a requirement in near future.

CAN protocol permits maximum 8 bytes of data to be sent through the bus. When all the protocol overheads considered, a Standard CAN frame can contain at maximum 135 bits. CAN frame format is depicted in Figure 2-2.

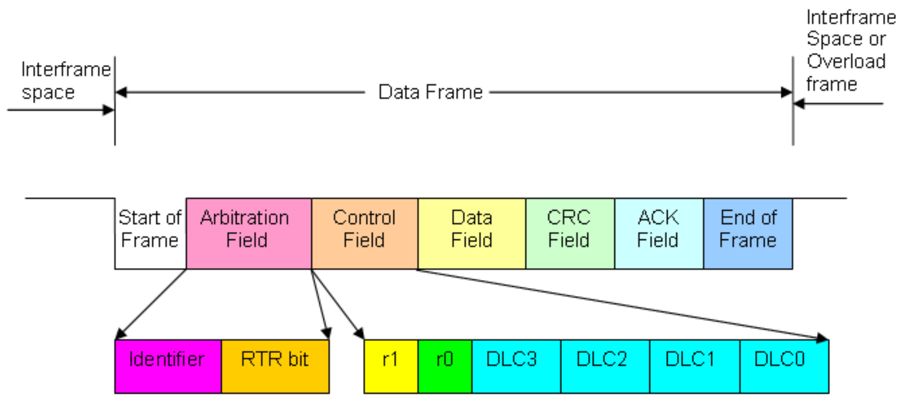


**Figure 2-2** CAN Frame Format

A CAN frame is composed of 7 fields.

- the start of frame
- the arbitration field
- the control field
- the data field
- the CRC sequence
- the ACKnowledgement field
- the end of frame

An 8th area, called interframe space, forms an integral part of the frame to bind it to the next frame. In the following parts the content of these fields will be briefly dissected.

*Start of frame.* Start of frame (SOF) field consists of a single dominant bit signaling that  the data exchange starts.

*Arbitration field.* This field consists of standard identifier (11-b) and a bit called Remote Transmission Request (RTR). A valid identifier can not have all of its most significant bits to be recessive (1). Therefore maximum number of valid identifier combination is $2^{11} – 2^4 = 2032$. On the other hand, in a data frame RTR bit must be dominant (0).

*Control field.* Control field consists of 6 bits. The first 2 bits are reserved bits to ensure the future upward compatibility. The last 4 bits of the control field is called Data Length Code (DLC) and indicates the number of bytes contained in the data field of the CAN frame. DLC field can be between 1 and 8 since CAN protocol permits maximum 8 bytes of data to be send in a single CAN frame.

*Data field.* Data field is the field which consists of useful data of CAN message exchange. Data to be transferred via CAN frame can be any number of bytes between 0 and 8, both inclusive. Since 9 possible values exist to indicate the number of data to be sent, to represent these values, 4 bits are used in DLC in "Control Field".

*CRC field.* CRC stands for "Cyclic Redundancy Code". This field consists of the CRC sequence area followed by a CRC delimiter.

The receivers check whether the transmitted message contains error by the help of CRC sequence sent by the transmitter. If CRC sequence and the data are not compatible the transmitted message is considered to be erroneous and rejected.

CRC sequence is maximum 15 bits. Including the CRC delimiter, CRC Field can be maximum 16 bits.

*Acknowledgement field.* This field consists of two bits, namely ACK slot and ACK delimiter. During these two bits time, transmitter sends two recessive (1) bits along the bus (in practice, the sender leaves the bus free and switches itself to listening or 'receiver' mode) [14]. If a receiver in the network receives the message with "no transmission error" including CRC, the received message is considered to be valid and receiving node acknowledges this message by sending dominant bit (0) in ACK slot time. Since the logic level along the bus is the result of AND operation of individual nodes' output, the transmitting node will see a dominant bit in ACK slot although it has sent a recessive bit. This signifies that the transmitted message is acknowledged by a receiver node. Although the nodes connected to the CAN network are not interested in the transmitted message, they must acknowledge it if they received it with "no transmission error". With this sense, acknowledgement of the transmitted message does not mean that the message is being used by one of the nodes, it simply means that the message is received correctly by at least one of the nodes connected to the network. If the received message contains one or more transmission errors then the receiving node must issue an error frame. On the other hand, ACK delimiter slot must be always recessive (1). Therefore, ACK slot lies between two recessive (1) bits, namely CRC delimiter and ACK delimiter.

*End of frame.* The data frame is terminated by a flag consisting of a sequence of 7 recessive (1) bit which is longer than the standard length of bit stuffing.

*Interframe space.* After the transmitting frame ends, another node can not start a new frame immediately. According to protocol, time between two consecutive frames must be at least 3-bit time.

In the physical layer, CAN uses nonreturn-to-zero (NRZ) bit representation with a bit stuffing length of 5. Bit stuffing is an important feature of CAN. In order to count the bit time, stations need to resyncronize periodically. However, if signal along the bus is recessive/dominant for a very long time, the stations can not find the opportunity to resyncronize themselves. In another words, station needs to see edges (changes from 0 to1 or 1 to 0) in the signal. Because of this requirement, according to CAN protocol, more than 5 consecutive equal-level (all-recessive or all-dominant) bits can not exist. If six or more equal-level bits are required to be sent by a transmitter, transmitter stuffs an opposite-level bit after 5 consecutive equal-level bits. On the other hand, receiver destuffs the sixth bit coming after 5 equal-level bit and obtains the original message sent by the transmitter.

As explained above in the CAN frame fields, when an error is detected by a node, an “error flag”, which consists of six consecutive dominant bits, is send to make the network be aware of the fault. Error recovery time in CAN, defined as the time from detecting of an error until the possible start of a new frame, is 17-31 bit time long. Since the corrupted frame reenters into the next arbitration phase, the additional delay, which is at least as long as error recovery time, may cause the frame to miss its deadline. Although CAN possesses some fault-confinement mechanisms based on error counters which are aimed at identifying permanent failures due to hardware disfunctioning at the level of the microcontroller, communication controller or physical layer, the main drawback of CAN is that a node must diagnose itself [1]. This requirement may lead to the nondetection of some critical error. Besides, for instance, a single node can perturb the functioning of the whole network by sending messages outside their specification (i.e., length and period of the frames). Some mechanisms were proposed to increase the reliability of CAN-based networks [15] [16] [17]. However, it is pointed out that although each of them solves a particular problem, none of them proposes a complete remedy [18]. Therefore, in the light of the above discussion, and by taking the inadequate communication rate of CAN, it can be said that CAN is not suited for safety-critical applications such as some future x-by-wire systems. Today, in modern cars, CAN is used in domains named powertrain, chassis and body. Although the former two domains are concerned with real-time control and safety of the vehicle's behaviour, currently CAN meets the requirements fairly well. However, with the introduction of x-by-wire applications and with the increase in the requirements of safety-critical and real time applications, CAN will fall short in a very near future. The gap in this domain will be satisfied with an emerging bus architecture named FlexRay™.

### 2.1.2 FlexRay

FlexRay protocol is the product of the exhaustive studies of a consortium whose core members are BMW, Bosch, DaimlerChrysler, General Motors, Motorola, Philips, and Volkswagen. The aim of the consortium, which was signed in 2000, was to conduct technical analyses of the existing networks used or available for use in the car industry, namely CAN, TTCAN, TCN, TTP/C and Byteflight to discover whether any one of them was capable of meeting all the technical requirements like high data rate, redundant channel support, deterministic delay, optical transmission demanded by modern automobiles when some near-future requirements were also taken into consideration. All of them were found to be insufficient at some point.

The shortcomings of these networking protocols, according to the study of the consortium are summarized below.

- CAN is not fast enough for the new applications required. Also, CAN does not utilize redundant channel and making CAN truly deterministic is difficult due to its event-trigger nature.

- TTCAN is inevitably not fast enough since it is somehow modified version of CAN. Although, due to its time triggered nature, it facilitates to make the transmission deterministic, still it does not support the redundant transmission channel. Also, it fails to provide support for optical transmission and a bus guardian.

- TTP/C frame size is considered to be too small for new applications. In spite of the use of TDMA for bus access, TTP/C provides no flexibility when compared to FlexRay. Support of the combination of synchronous and asynchronous transmission sections, the multiple transmission slots for a single node in the synchronous section and the nodes acting on single, double or mixed channels can be counted as the facilities of the TDMA in FlexRay which do not exist in TTP/C.

- Byteflight can be considered to be the subset of FlexRay. The asynchronous mode of FlexRay is functionally compatible with byteflight. Therefore Byteflight does not offer enough functionality for demanding applications.

As a result of this picture, consortium began to work on a new communication protocol, named FlexRay, which will be remedy for the insufficiencies of the existing networking protocols. The FlexRay protocol is aimed to operate at high frequencies so as to fill the gap in the applications where the bit rate of CAN falls short. FlexRay is designed to be capable of implementing X-by-Wire applications and providing redundancy. Some of the new and promising facilities that FlexRay provides is given below which signify that the FlexRay protocol is defined to be capable of serving all near-future electronic functions.

FlexRay;

- supports single channel or two channel (redundant) communication topology (nodes using single channel communication or two channel communication may exist in the same network),
- provides gross data rate of 10 Mbps,
- transmits data in synchronous and asynchronous modes and the length of these modes can be adjustable,
- provides deterministic data transmission with pre-known and guaranteed latency and jitter,
- detects the signal errors very quickly,
- withstands synchronization errors of the global time base,
- provides an error management mechanism via an independent "bus guardian" (bus guardian is optional and network still works without a centralized bus guardian),
- permits the addition of new nodes to an existing system without the need to reconfigure the existing nodes,
- avoids collision for bus access,
- provides a robust system against transient faults and external radiation.

Having included these facilities, FlexRay working group created by the consortium, published the first publicly available protocol specification in 2004 [19]. The final version of the FlexRay is "Version 2.1" and released in December 2005 [19]. The above mentioned parameters and facilities will enable future requirements to be met for three classes of application not yet covered by CAN or by other existing protocols. These are communication with high bandwidth, deterministic communication with high bandwidth and deterministic and redundant communication with high bandwidth. Since FlexRay protocol is just a recently emerged protocol, it will take about 10 year of time before industry has widely implemented FlexRay protocol in automobiles. When the FlexRay is commonly being used in automotive industry in near future, CAN will be used as a sub-bus of FlexRay and LIN as a sub-bus of CAN [14].

### 2.1.2.1 Protocol Properties

Bus access in FlexRay is implemented via TDMA structure. As a natural consequence of TDMA, at least one time slot is assigned for every node constituting the network so that they can transmit their frames at these time slots exclusively. Since the synchronization is well established throughout the network, all nodes can estimate the time slot which they are in with fairly small and acceptable differences. As each node sends their messages in the time slots dedicated to them, no collision arises when accessing the bus except that the collisions which may occur during the starting phase of the network. One whole cycle of the FlexRay network is divided into two separate parts namely Static Part and Dynamic Part. Static Part is the part of the FlexRay cycle where frames are sent according to TDMA structure as defined above. Whereas, Dynamic Part adds event-triggering nature to the FlexRay cycle and properties of Dynamic Part will be explained in the following chapters more detailed. It is worth mentioning that the proportion of Static Part to Dynamic Part in one FlexRay cycle is configurable and may change from a network to other. Even in some configurations one of the parts may not exist at all. This concept is illustrated in Figure 2-3

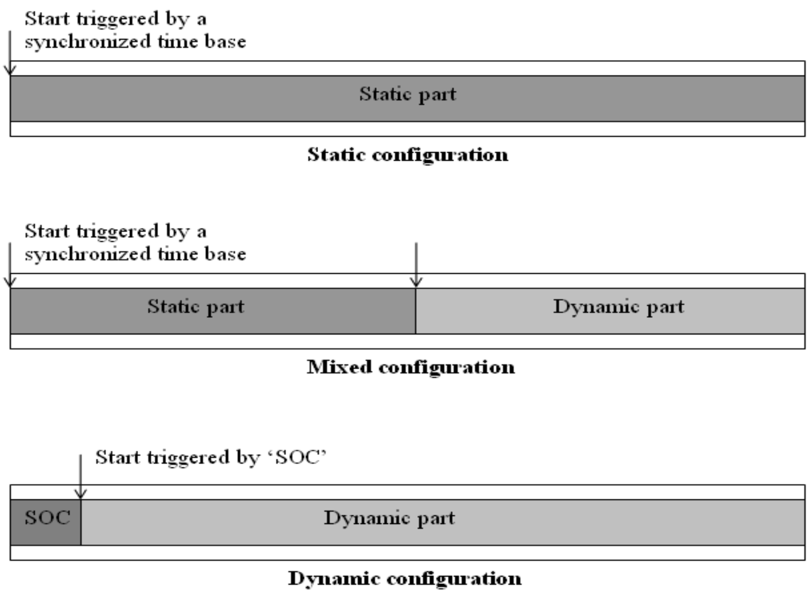


**Figure 2-3** FlexRay TDMA Structure[14]

In order to understand FlexRay synchronization, protocol properties and TDMA structure better, some terms regarding the timing hierarchy of the FlexRay protocol are given below.

*Communication Cycle:* Communication in FlexRay takes place with the aid of recurring communication cycles. These communication cycles are composed of "static segment", "dynamic segment", "symbol window" (optional) and a phase in which the network is in idle mode, called the Network Idle Time (NIT). This is illustrated in Figure 2-7.

Once the FlexRay Bus is configured, every node knows in which time slots to transmit their frames throughout the cycle. In all cycles, those time slots are dedicated to the very same nodes. However, nodes may decide to send different messages in the same time slot in seperate cycles or not to send any frames at all. For example, Node A, to whom, let's say, 15th time slot is allocated in the cycle, may be configured to send frame1 in every 2 cycles and frame2 in every 4 cycles in the 15th slot by giving frame2 an offset. This is similar to the "Matrix Structure" where rows of the Matrix are the FlexRay cycles and the columns are time slots in a cycle. This scheme is illustrated in Figure 2-4 for 15th time slot. In this “Matrix Structure”, number of row, i.e number of cycles can be 64 at maximum according to FlexRay protocol.

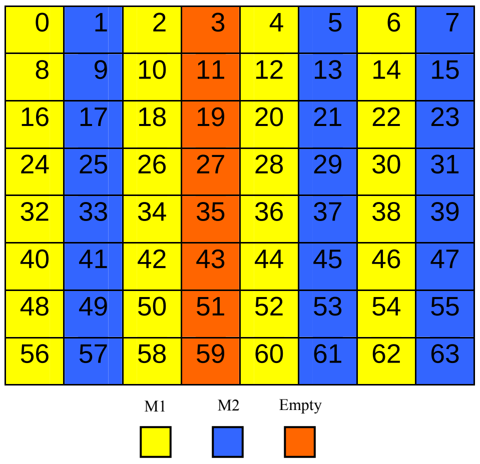


**Figure 2-4** Scheduling Structure: (Msg, Freq, Offset): (M1, 2, 0) and (M2, 4, 1)

*Macrotick:*

Macrotick is the smallest unit of global time granularity of the network [19]. Macrotick time interval is derived from the cluster-wide clock synchronization algorithm. Therefore, macrotick duration is same for all nodes comprising the network. Macrotick duration is calculated by an algorithm synchronization routine and is composed of whole number of microticks.

*Microtick:*

Microtick, which constitutes the macrotick, is even smaller time unit in FlexRay timing hierarchy. Main difference with macrotick is that microtick duration is not common for all nodes throughout the network. Rather, they are created locally node by node and are, directly, derived from CC’s (Communication Controller) oscillator possibly through the use of a prescaler.

The timing hierarchy of FlexRay from communication cycle level to microtick level is illustrated in Figure 2-5.

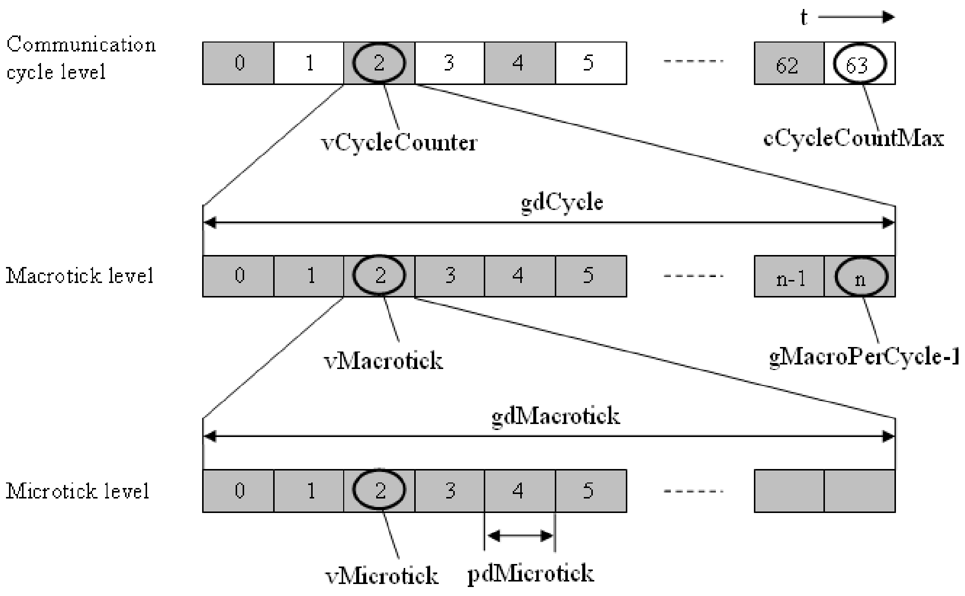


**Figure 2-5** FlexRay Timing Hierarchy [19]

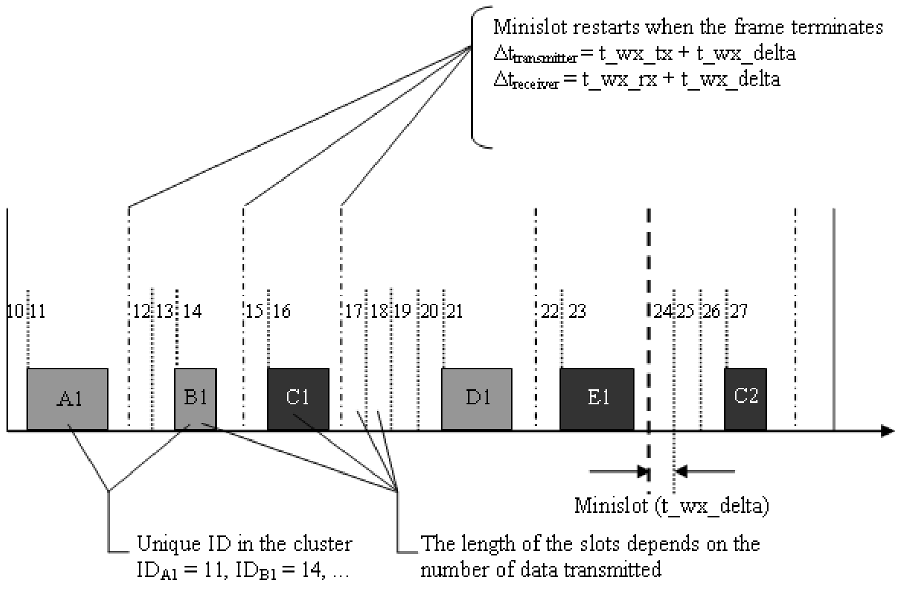
### 2.1.2.2 Medium Access

FlexRay cycle is composed of two different segments where FlexRay frames containing the payload are transmitted. These segments are called Static Segment and Dynamic Segment. Although there are two other segments that are composing the FlexRay cycle, namely “Symbol Window” and “Network Idle Time (NIT)”, these segments are not used for exchanging payload, rather they are used for synchronization and other purposes.

Medium access in Static Segment and Dynamic Segment are different from each other and are summarized in the following parts.

*Static Segment:* Medium access in Static Segment is purely TDMA. According to this structure, the segment is divided into equal time slots including silence spaces. The whole segment is well structured and start and end times are precisely defined cluster-wide. Each time slots are assigned a unique identifier. When FlexRay network is established this time slots are allocated by certain nodes. Therefore, this structure enables to have a media free from collisions, to have a real time network due to the known latencies and to have a system with known bandwidth for a given bit time.

*Dynamic Segment:* Medium access in Dynamic Segment can be defined as Flexible TDMA (FTDMA) as in Byteflight. According to this FTDMA structure, Dynamic Segment is divided into equal minislots. Each minislot is identified by a unique identifier similar to the slot identifiers in the Static Segment. However, in Dynamic Segment, duration of minislots is much smaller than the duration of the slots in Static Segment, as the name "mini"slot implies. Therefore, it is not possible to transmit a regular FlexRay frame which can be 254-byte long through this minislot. When it is taken into consideration that Dynamic Segment of FlexRay cycle is used to send the event-triggered and spontaneous data, it is obvious that many nodes may not have any frame to send when their time has come and it is logical to design minislot duration to be small so that not much bandwidth is wasted when the node has nothing to send in its minislot time. For instance, assume, the network is at the time of minislot "*m*". If the node, which is supposed to send the frame with identifier "*m*" if there exists a frame to send, does not have any frame to send at that time instant, nothing is sent and after one minislot time the network becomes to be at the time of minislot "$m+1$". But, if at that instant, that node does have a frame to send with identifier "*m*", then the node transmits its frame by extending the minislot duration so that the duration is sufficient to send the whole frame properly. After the frame with identifier "*m*" is sent, then network becomes to be at the time of minislot "$m+1$". This procedure goes like this till to the end of the Dynamic Segment. As it can easily be understood from the architecture of Dynamic Segment, while in a cycle Dynamic Segment ends, say, with $k^{th}$ minislot, in another cycle it ends, say, with nth minislots where "*k*" and "*n*" may be different from each other. This may happen because in some cycles quite a lot frames may be ready to be sent when their minislot time has come which prevents minislot counter from increasing much whereas in other cycles few frames may be ready to be sent when their minislot time has come and since the frames are passed idle, minislot counter counts more. As a result of this, a node having a frame to send with a large frame identifier, may not send its frame if there are fairly many frames with smaller identifier to be sent in that cycle. Therefore, this may lead that frame to miss its deadline. To avoid this, the designers of FlexRay network should allocate identifiers to the frames inversely proportional to their priorities. With the introduction of this FTDMA structure to FlexRay network cycle, it became possible to send spontaneous and event-triggered messages, provide burst transmission, easily manage diagnostic data and, in general, transfer all kinds of messages in an ad hoc way. The FTDMA structure of FlexRay is depicted in Figure 2-6



Minislot restarts when the frame terminates
$\Delta t_{transmitter}$ = t_wx_tx + t_wx_delta
$\Delta t_{receiver}$ = t_wx_rx + t_wx_delta
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
A1
B1
C1
D1
E1
C2
Minislot (t_wx_delta)
Unique ID in the cluster
$ID_{A1}$ = 11, $ID_{B1}$ = 14, ...
The length of the slots depends on the number of data transmitted

**Figure 2-6** FlexRay FTDMA Structure [14]

What is explained up to this point about the timing hierarchy and the protocol structure of FlexRay protocol can be illustrated by Figure 2-7

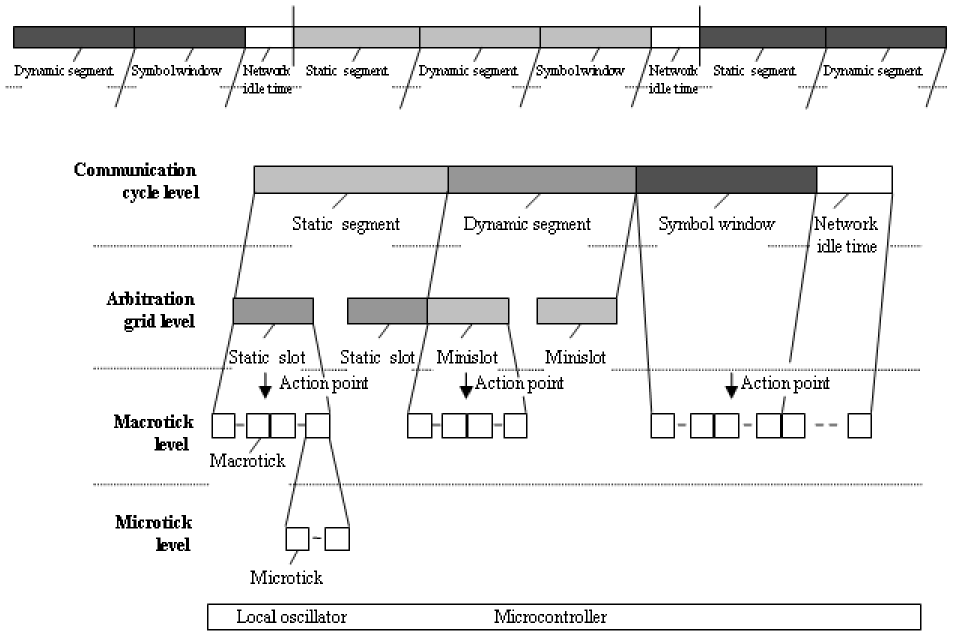


**Figure 2-7** FlexRay Timing Hierarchy [19]

### 2.1.2.3 Physical Characteristics

As opposed to CAN protocol, in FlexRay protocol specification, the physical layer of the protocol is not explicitly defined. Therefore, there may exist various physical layer possibilities from differential pair wire to optical fiber.

A very distinctive feature of FlexRay protocol is that physical layer must consist of two completely independent channels. While these channels may be used separately from each other for data transmission so as to increase data rate up to 10 Mbps gross bandwidth, the channels can also be used completely equivalent to each other so that functional redundancy can be achieved throughout the network providing reliability and enhancing the fault-tolerant aspect of the system.

The bit coding of FlexRay network is “non-return to zero” (NRZ 8N1) coding, which means that the transmission of each byte (8 bit) is framed by a start bit and a stop bit and that the value of the physical signal does not change during the whole of the bit time.

According to the FlexRay 2.0 specification [19], the electrical values of the binary elements 1 and 0 are both represented by dominant states. Because the differential electrical levels alternate (change their sign) to represent logical level “1” and “0”. As it is seen from the Figure 2-8, there are two other electrical levels defined in the protocol. Both of these recessive levels are reserved for the idle modes of the bus except that one of them is for “low power down” idle mode. Since the differential voltage level between the dominant levels is in the order of 0.7V, signals do not cause any significant electromagnetic radiation throughout the network.

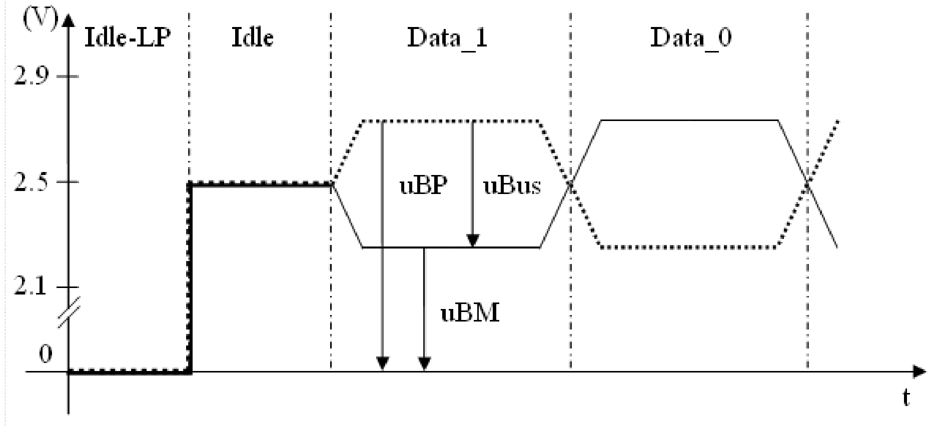


**Figure 2-8** FlexRay Electrical Levels [14]

The bit time and the propagation time bound is strongly related in CAN network. This is not the case for the FlexRay bus as the medium access is realized with time sharing principle. Gross bit rate for FlexRay protocol is 10 Mbps which is implying a bit time of 100ns. According to the protocol, regardless of the FlexRay network topology (direct line, active stars, repeaters, etc.), the maximum propagation time must not exceed 2500ns.

## 2.1.2.4 Communication Frame Format

FlexRay frame consists of 3 separate data fields, namely, header, payload and trailer. The whole picture of a FlexRay frame is shown in Figure 2-9

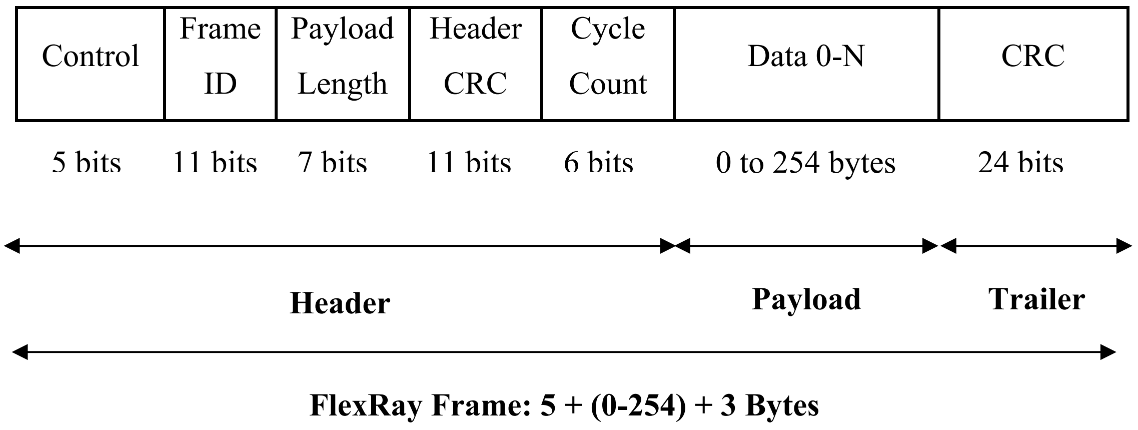


**Figure 2-9** FlexRay Frame Formats

*Header*

Every frame starts with a Frame Starting Sequence (FSS) consisting of 8 bits of '0' without a start or stop bit. FSS situates between two consecutive frames. After FSS, comes header segment which is encoded in 40 bits. The first 5-bit area is the control area in header segment. The name of the bits in this 5-bit area in order of appearance is given below:

- reserved = 1
- payload preamble indicator
- null frame indicator
- synchronization frame indicator

    - "0" means normal frame
    - "1" means sync frame for synchronizing the clocks
- start-up frame indicator

The next field in Header Segment is Frame ID area. Frame ID consists of 11 bits ranging from 1 to 2047, "0" being illegal. In a cycle, Frame ID determines the slot number in Static Segment. In other words, a frame is transmitted only at the slot time equal to its Frame ID. Frame ID also determines the priority of a frame in dynamic segment as a low identifier value indicating high priority.

The length of the message to be transmitted is indicated in Payload Length field in terms of double byte. Since the payload length can be 127 at maximum, 254 bytes of data can be transmitted with a single FlexRay frame. The data up to this point is protected with Header CRC field which is encoded in 11 bits with a Hamming distance of 6. The last field in Header Segment is Cycle Counter field. Cycle Counter is 6-bit field indicating the number of the current communication cycle. The communication cycle number can not be increased indefinitely and is bounded with the maximum number of 64 as explained in the preceding chapters. Frame allocations for all nodes repeat itself with the periodicity of this maximum number of communication cycle value.

*Payload*

Payload Segment is the field where useful data of the length as indicated in Payload Length field in Header Segment is transmitted.

*Trailer*

At the very end of the frame comes another CRC field encoded in 24 bits which protects the integrity of the whole frame with a Hamming distance of 6.

### 2.1.3 LIN (Local Interconnect Network)

In modern cars today, CAN, with its variants namely High Speed CAN, Low Speed CAN and Low Speed CAN Fault Tolerant, is used as the primary communication network. However, among the numerous functionalities existing in a modern automobile, some of them requires much less capacity than CAN offers. Therefore, for those functions those require less bandwidth and capacity, a new serial communication system to be used as SAE Class A network, namely LIN (Local Interconnect Network), has been designed by the consortium comprised of the companies, Audi, BMW, Daimler Chrysler, Volkswagen, Volvo Car Corporation and Motorola. The final specification of the LIN protocol (rev 2.0) was issued in September 2003 [20]. The primary and original purpose of LIN [21] is to provide a "sub-bus" for CAN, with reduced functionality and lower costs, in other words to provide an economical solution when the requisite performance level is not high. Sun roof (open, close, inclination, etc.), rain detector, automatic headlight switch-on, seats (all seat adjustments and functions) are some of the network nodes/participants which work satisfactorily in a LIN bus.

LIN serial network works on "single master multiple slaves" concept. According to this concept, one of the nodes in LIN network operates as the master node and all the other nodes in the network are the slaves. The LIN link is based on asynchronous communication. No nodes are required to keep a supplementary clock for the operation. The traffic on the bus is initialized and controlled solely by the "master task" of the network which sits in the master node. The master node invites the slave nodes to communicate on the bus according to the scheduling table in its "master task". The node for whom the communication is granted, which can be the master node itself, sends its frame through the network. Thus, neither the arbitration for accessing the shared media nor a supplementary clock to track the network time is needed in LIN network. Since the slaves cannot supply data unless they have been invited on the basis of a scheduling table established by the master, it enables us, by examining the scheduling table, to predict the moment when a message will be supplied on the bus. So LIN protocol provides us a degree of determinism.

A LIN communication frame consists of a header provided by the master task and a response provided by a slave task. In other words, master nodes invites a slave task to talk in the header part of the communication frame and the invited slave task supplements its data in the data part of the very same frame. LIN communication frame is depicted in Figure 2-10.

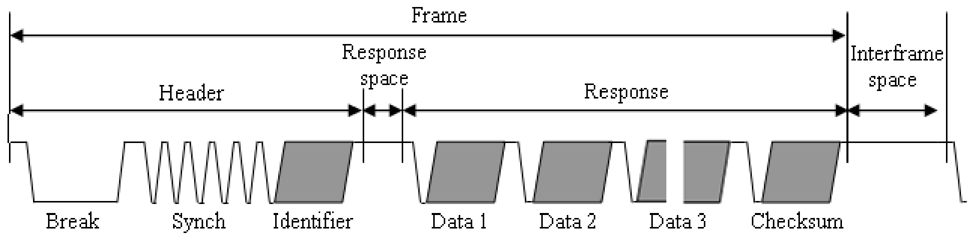


**Figure 2-10** LIN Communication Frame [14]

*Header*

The frame starts with a header field, consisting of three main parts, all transmitted by the master, as mentioned above. The first field in header is “break field” which consists of at least 13 dominant bits and followed by “break delimiter” which is at least 1 recessive bit long. Then comes the synchronization field. So as to facilitate the evaluation of the bit rate of the bus, synchronization field has the conventional value of “0101 0101” (hex 0x55), framed by the dominant start bit ‘1’ and recessive stop bit ‘0’, making numerous bit transitions available. After synchronization field, there exists a 10-bit field where the identifier of the message is defined. This 10-bit header field starts with a dominant start bit and ends with a recessive stop bit. 2 parity bits are added just before the stop bit. The remaining 6 bits represents the identifier of the message providing 64 identifiers. The 6-bit identifier field also can be divided into two. While the first 4 bits define the identifier, the other 2 bits specify the length of the data field in 4 different values, 1 byte, 2 bytes, 4 bytes and 8 bytes.

*Data*

The frame is followed by the “response” of one of the slaves (or the master task). So as to decode and process the header field, a space called “response space” is granted to the slave tasks (or master task) between the header and the data field. The data field consists of 1 to 8 bytes data as required in the header section plus 3 command and security bytes. The checksum field which is sent after the data terminates the LIN frame. The consecutive frames are separated by a time interval called the ‘interframe’.

The slave nodes do not send any acknowledgement of correctly received messages in LIN. The consistency of the network traffic is checked by the master node. If there is an inconsistency (no response from the slave, incorrect checksum, etc.), the master can retransmit the message. On the other side if the slave detects an inconsistency, the slave controller sends it to the master in the form of diagnostic data.

The maximum bus bit rate of LIN network is 20 kbps. Because this low bit rate and low cost, LIN is suitable for the low bit-rate applications where using CAN would be waste of both bandwidth and money.

The bit encoding of LIN is NRZ (non return to zero) and the termination resistance of master node and the slave nodes are 1 kΩ and 20-47 kΩ, respectively. The length of the wire link of a network must not exceed 40m and the maximum recommended number of nodes in a sub-network should not exceed 16 even though the protocol can support a maximum of 63.

### 2.1.4 MOST (Media Oriented System Transport)

Changing applications throughout the years necessitate imposing new requirements for in-vehicle communication networks. As the share of the telematics and media functions increased in automobiles by time, conventional buses like CAN bus began to fall short to meet the requirements of these applications. Media and telematics functions were requiring larger bandwidth than CAN could provide and were necessitating working synchronously which is not compatible with the event-trigger nature of CAN bus. Because of these motivations, the MOST Cooperation was set up in 1998 by BMW, Daimler Chrysler, Harman/Becker and OASIS Silicon Systems, with the aim of standardizing the communication technology of the MOST concept. The MOST bus is designed to provide links between radios, navigation controllers and associated systems, displays (on the instrument panel, at seats, etc.), CD players and changers (audio and video CD, DVD, CD-ROM, etc.), voice recognition systems, mobile telephony, active in-car sound distributors and so forth. Although synchronous mode communication is dominant, MOST enables to communicate in both synchronous mode and asynchronous mode. In Table 2-1 the main properties of most of the signals that can travel on a MOST bus are summarized.

**Table 2-1** Signals Travelling on a MOST Bus

| Signal Type | Signals | Bit Rate | Format |
|---|---|---|---|
| Control Signals | | 125/250 kbps | Async |
| Data Signals | Digital Audio:<br>- Uncompressed audio CD<br>- MPEG compressed audio | 1.41 Mbps<br>128/384 kbps | Sync<br>Async |
| | Digital Video:<br>– Uncompressed CCIR 601/4:2:0<br>– Compressed MPEG1<br>– Compressed MPEG2 | 249 Mbit s-1<br>1.86 Mbit s-1<br>2/15 Mbit s | Sync<br>ASync<br>A Sync |
| | Navigation:<br>– Data carrier<br>– MPEG1 video<br>– Vice | 250 kbps<br>1.4 Mbps<br>1.4 Mbps | Async<br>Sync<br>Sync |
| | Data Communications | Several bytes | Async |

In Figure 2-11, the most unfavorable situation of a conventional configuration for audio and video signal distributions in a motor vehicle is given.

| | | |
|---|---|---|
| Four-Channel Stereo Audio | (4x2) x 1.4 Mbps | 11.2 Mbps |
| Multiplexed Video | 2.8 to 11 Mbps<br>+ reserve of 4 Mbps | +2.8 to 11 Mbps<br>+ 4 Mbps |
| **Making Total of** | | **18 to 26.2 Mbps** |

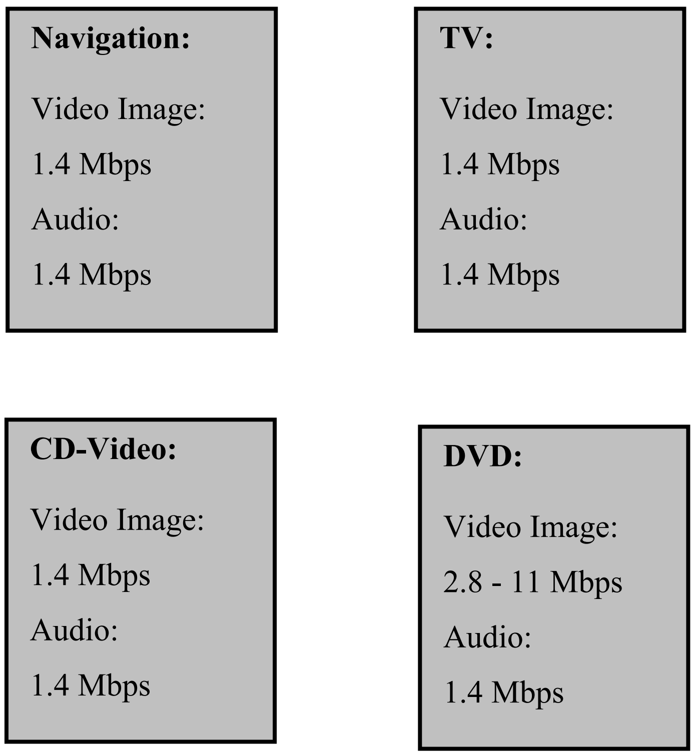


**Figure 2-11** Conventional Configuration of Audio and Video Signals

Therefore, the MOST bus must provide a data rate in the order of 20 Mbps to satisfactorily meet the requirements of a modern car. In fact, the theoretical gross data transfer rate of the MOST bus is 25.46 Mbps in synchronous mode. However, the MOST networks currently operate at around 8-10 Mbps gross for video and audio applications. On the other hand, MOST networks operate at 14.4 Mbps at maximum in asynchronous mode which can be used for example, to transmit short bursts of signals such as those corresponding to voice signals for navigation assistance and other driver assistance messages.

Based on the D2B solution, the MOST is initially designed to carry digital audio CD data at a fixed rate. This working mode is the “synchronous” mode of the MOST system. In synchronous mode, a master supplies a clock signal so that all the other network participants synchronize themselves to this clock. MOST also supports the presence of several masters in a single network and the maximum number of participants is 64. The digital data is transferred in frames with 44.1 kHz (the rate of digital Audio CD) rate. Each frame consists of 60 channels of 1 byte each. Therefore, the theoretical maximum data rate of the bus is calculated as follows:

$$(60 \times 8) \times 44100 = 21.268 Mbps \tag{2-1}$$

In fact, because of the format of the byte transmitted (in 10 bits, 8N1) in this mode, the maximum gross data transfer rate is:

$$(60 \times 10) \times 44100 = 26.585 Mbps \tag{2-2}$$

The working principle of the MOST network is simple. 60 bytes of the frame is considered as 60 channels and these channels are multiplexed with nodes or applications. For example, as shown in the Figure 2-12, 6 channels of the frame, making up 2.1168 Mbps, can be reserved for audio transmission and 29 channels of the frame, making up 10.231 Mbps, can be allocated for carrying video signals. Thus, taking the number and the quality of the applications (how many audio signals, how many video signals and what quality) to be transmitted through the MOST bus into consideration, sufficient number of channels are assigned to each application. Along with these data channels, some communication channels are dedicated to the transfer of commands, which indicates the command information of the data send by the transmitter. Consequently, once the channel allocation is done, when the connection has been established, the digital data stream can be transmitted without being formed into packets.

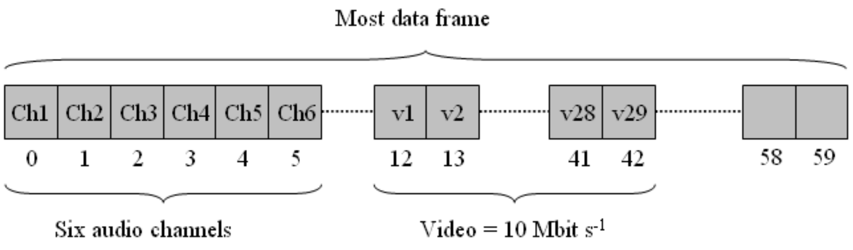


**Figure 2-12** Example of a MOST Frame [14]

The MOST physical layer was originally designed around a 'copper' twisted pair wire link. However, now, it has evolved to its present form, supported by a fiber optic medium. The MOST bus with optical fiber physical medium provides both a wider range of applications and greater immunity to external parasitic signals, while avoiding interference by radiation with the immediate environment. The application topology of this bus is often in the form of a ring network.

## 2.2 GATEWAY NODE FOR INTERCONNECTED INVEHICLE NETWORKS

As explained in the preceding sections, various network protocols, from LIN to FlexRay, have emerged throughout the evolution of the in-vehicle electronics and communication. This is due to the fact that, numerous applications, whose performance and quality requirements differentiate significantly from each other, are to run together in modern cars today. While for some applications, bounded delay and real time performance is required, for some other applications providing a large bandwidth is nothing but waste of sources. Among the diverse automotive networking protocols, each of which are developed to suit to a specific requirement, we can easily say that CAN bus is still the most important networking protocol and constitutes the backbone of the in-vehicle communication. However, this will not be the case in near future. New technologies such as x-by wire technology, which is a generic term and means the replacement of mechanical and hydraulic systems by completely electronic ones, is being introduced step by step to perform steering or braking functions. Since such emerging technologies perform safety-critical tasks which require real time communication, they require much more bandwidth than CAN provides. Therefore a new network protocol, FlexRay, has recently been developed to be used in x-by-wire applications where the CAN bus falls short in satisfying the requirements. It is expected that in long term, the FlexRay protocol will completely replace the CAN bus for high-speed applications and become the main communication backbone in automobiles. CAN is expected to stay around for a long time for relatively lower speed applications due to its legacy status. Hence, both during the transition from CAN to FlexRay for high-speed applications and afterwards both protocols will exist together in the car performing the tasks of different characteristics. The co-existence of FlexRay and CAN in the car imposes a Gateway unit which provides an interface between the both networks and facilitates the inter-communication without compromising the overall performance of the inter-connected network in terms of delay and jitter.

The previous work on Gateway design and implementation for in-vehicle networks include Gateway implementation on FPGA which focus on the hardware performance and timing properties [24], [25]. In addition, implementations by micro-controller programming in [26], [27], [28] and [29] demonstrate that the gateway correctly converts the messages between protocols. Among these studies, [29] implements the Gateway unit in a Hybrid Electrical Vehicle test bench and report experimental results for the achieved data rate of 0.9285 Mbps data rate on CAN and 4.3478 Mbps on FlexRay. To the best of our knowledge all of the previous Gateway implementations convert the messages from one protocol to another without any processing of the message payload in the signal level. Furthermore there is no study of the end-to-end network performance study for FlexRay-CAN networks connected by a Gateway. In this thesis, we both process the message payload in the signal level in addition to the protocol conversion and examine the end-to-end network performance as well as the Gateway processing delay for the Gateway implemented. Being able to process the messages in signal level is very important for a full functional Gateway. Because according to the configuration of the networks and the operational scenario of the Gateway, it might be required to fragment the incoming messages into their signals and pack the signals that will be sent in a single message according to the application requirements and to potentially increase the network efficiency. One example could be an ECU on FlexRay which requires multiple signals collected from different sensors on CAN network to complete a certain task. If the periods of the CAN signals are appropriate the gateway unit can put the signals in individual CAN messages into a single FlexRay message and send it to the ECU. Similarly the ECU in FlexRay might generate multiple signals at the same time to be sent to a number of different actuators on the CAN bus. In that case it sends a single FlexRay message which carries these multiple signals. The gateway fragments the message and sends each signal to a different CAN node with different priorities as required. The design, implementation and the performance analysis of the Gateway are discussed respectively in the following chapters.

# CHAPTER 3

# FLEXRAY-CAN GATEWAY DESIGN

A Gateway, as explained above, is the unit which has interface with several different networks in a bigger network and provides data transfer between the nodes in distinct networks. These networks might employ different communication protocols. In this context, FlexRay-CAN Gateway properties, design requirements and the functional design of the Gateway are discussed in this chapter.

The operational requirements and performance metrics that we consider in the design are the correct *protocol conversion* between FlexRay and CAN, bounded *gateway processing delay* and *delay variance* and the *flexibility of the configuration*. We first eloborate these design considerations and then describe the Gateway functionality to transfer messages between FlexRay and CAN networks.

First of all, the Gateway has to perform protocol conversion which includes extracting the payloads of the incoming messages and then adding the correct protocol headers before sending them to their destination network.

Since FlexRay and CAN communication protocols are very different from each other conceptually, FlexRay-CAN Gateway must comply with some number of requirements. The Gateway design must take the basic differences between the FlexRay and the CAN Networks into consideration which are the payload difference, bit-rate difference and the difference in the arbitration scheme of the protocols. While the maximum payload which can be transferred through CAN network is 8 bytes, FlexRay frame might be up to 254 bytes long. Regarding the arbitration schemes, CAN is an event-triggered network and uses the bus access is granted to frames based on their priorities. FlexRay is a time-triggered network. There is no conflict for medium access in FlexRay network since FlexRay uses TDMA and flexible TDMA structure. Also bit rates of these two protocols are very different from each other. As FlexRay bit rate is 10 Mbps, CAN network bit rate can vary from several tens of kilobits to 1 Mbps at maximum. This difference in transfer data rate of the two networks must be compensated and some precautions should be taken in the Gateway so that no conflict occurs and no data is lost in communication. The impacts of these differences of the FlexRay and CAN networks on the design of the Gateway are explained below in 2 parts: FlexRay-to-CAN Functionality Design and CAN-to-FlexRay Functionality Design.

Needless to mention, a very important performance metric for a Gateway is the processing delay which is defined as the time difference between the transmit time of the signal from the Gateway and the receive time of the same signal in the Gateway. The processing delay is a component of the end-to-end delay of the signals carried in the message. Hence it should be bounded by a maximum value to be able to compute an upper bound for the signals that are transmitted end-to-end over the Gateway. Obviously, the lesser the delay that the signals experience in the Gateway, the better performance the Gateway has. The experimental data showing that the Gateway processing delay is bounded is given in section 6.3.4.3. Finally the processing of the Gateway is also related with the hardware used in it. For a better performance, as mentioned previously, a high speed microcontroller unit is used in the design of the Gateway. A further metric that is related to processing delay is the variation of it. This variation particularly increases the jitter of the periodic signals that are transmitted from FlexRay to CAN.

The messages that arrive at the Gateway might contain more than one signal. In that case the Gateway should be able to map these signals to different outgoing messages in a flexible way as required by the operation of the vehicle.

Our Gateway design is realized by software on a microcontroller. We took the software complexity into consideration in the implementation. Low software complexity is important for both keeping the processing delay low and the easy adaptation of the Gateway code to the new emerging requirements of the node.

## 3.1 FLEXRAY-to-CAN GATEWAY FUNCTIONALITY DESIGN

FlexRay-to-CAN functionality mainly focuses on the fragmentation of the FlexRay message and queuing the incoming messages to transfer them in order through FlexRay network since FlexRay frame and data rate can be much greater than CAN frame and data rate. The tasks that are performed in FlexRay-to-CAN functionality are explained below in more detail.

*Fragmentation*

An average FlexRay frame is expected to be longer than the longest CAN frame which is 8 bytes long. This is because the maximum FlexRay frame length is 254 bytes. Therefore, for FlexRay frames longer than the biggest CAN frame, FlexRay-to-CAN functionality has to fragment the incoming message into pieces and send them in consecutive CAN messages. If the incoming message is fragmented, the Gateway must put a header to each message piece so as to relate these fragments with the entire message on the other side. CAN node on the other side can easily reassemble the messages coming from the Gateway and build up the whole message. To handle the incoming FlexRay messages in the Gateway and making them wait to be sent through CAN bus, Gateway needs to have some dedicated buffers or a queue structure to store the fragments of the FlexRay message. Since the data rate of the FlexRay is greater than the bit rate of CAN bus, after a while, the fragments waiting to be sent on CAN bus might be accumulating in the Gateway. So, to prevent any loss of data, the length of buffer allocated for the fragments must be considered properly to meet the requirements of the Gateway.

*Message Elaboration with Signal Mapping (Processing)*

Very often, multiple signals of the vehicle such as temperature and pressure values can be carried in a single message. Hence these signals might be destined to different nodes. In addition, a certain signal might be received by multiple nodes. The different message routing options that the Gateway must handle are summarized below in bullets:

- Multiple fragments of the FlexRay message (each carrying a different signal), each is sent with distinct CAN ID.
- Some fragments of the FlexRay Messsage might be multiplexed to different CAN nodes and the others, each, is sent to single distinct CAN ID. E.g Flexray message : 20 B, Fragment 1 (F1) : 6 B, Fragment 2 (F2) : 6 B, Fragment 3 (F3) : 5 B and Fragment 4 (F4) : 3 B. F1 goes to CAN ID 1, CAN ID 3 and CAN ID 4, F2 goes to CAN ID 2 and CAN ID 7, F3 goes to CAN ID 6 and F4 goes to CAN ID 5
- Further, some fragments of the FlexRay message might be directed to the same CAN ID. E.g F1 and F5 directed to CAN ID 10.
- As a situation related with fragmentation, if multiple fragments of the FlexRay Message are sent by single CAN ID in the CAN bus, the total length of the fragments may exceed 8 B. Then the message must be fragmented and headers must be put properly. E.g F2(4B), F3(5B), F4(3B) and F7(2B) wanted to be sent by CAN ID 5 then the fragments can be packed as F2-F4 and F3-F7 so that total length of data becomes 7B and there still remains 1B space to put header into the message.

As seen, Gateway must be able to handle very different signal mapping situations and process the incoming data accordingly. Even, some of the possibilities described above in bullets, might exist at the same time. In this case the buffer management is very significant when processing the FlexRay frame. For instance, if the message to be transmitted via CAN is more than 8 bytes, the message needs to be transmitted in parts and the waiting parts must be stored in buffers. Therefore, the more "greater than 8 byte messages" to be transmitted through CAN bus, the more buffers the Gateway needs. Moreover, if the priority of the CAN message that Gateway transmits is low (with greater ID) and the production rate of the message in FlexRay network is high then it is probable that before all parts of the message have been sent via CAN network, new messages arrive from FlexRay network. This situation leads to the increase in the number of buffers needed. The same problem may occur, though less probable, when the length of the message to be transmitted through CAN is less than 8 bytes. Therefore, FlexRay ID numbers and CAN priorities should be assigned properly by taking the specific mapping situations and the data rates into consideration so that minimum number of buffers is needed in Gateway and no data is lost. As a result, message processing functionality of the Gateway is highly dependent on the network configuration. Besides this, software protocol for the data exchange is also important. While, as one possibility, according to the protocol running, Gateway might, for example, be required to add header before each signal when multiple signals are to be sent in a single CAN frame, as another possibility, the places of all signals are predefined in software and zero is sent for the signal that are not present. To sum up, in message processing task, Gateway must take, application dynamics, network configuration and software protocol into account all together.

*Queuing*

Queuing is a very important functionality of the Gateway. The queuing requirement arises from the bit rate difference of FlexRay and CAN network. Therefore, Gateway must create and handle a proper queue structure and manage the incoming messages by the help of this queue.

As discussed above, main functionalities of the Gateway can not be considered independent of network configuration and software protocol running on Gateway. Therefore, for the Gateway to function properly, queue requirements and structure, FlexRay IDs and CAN priorities must be selected accordingly. As a consequence of this, Gateway performance will be increased and data loss will be avoided.

Gateway functionality is depicted in Figure 3-1.

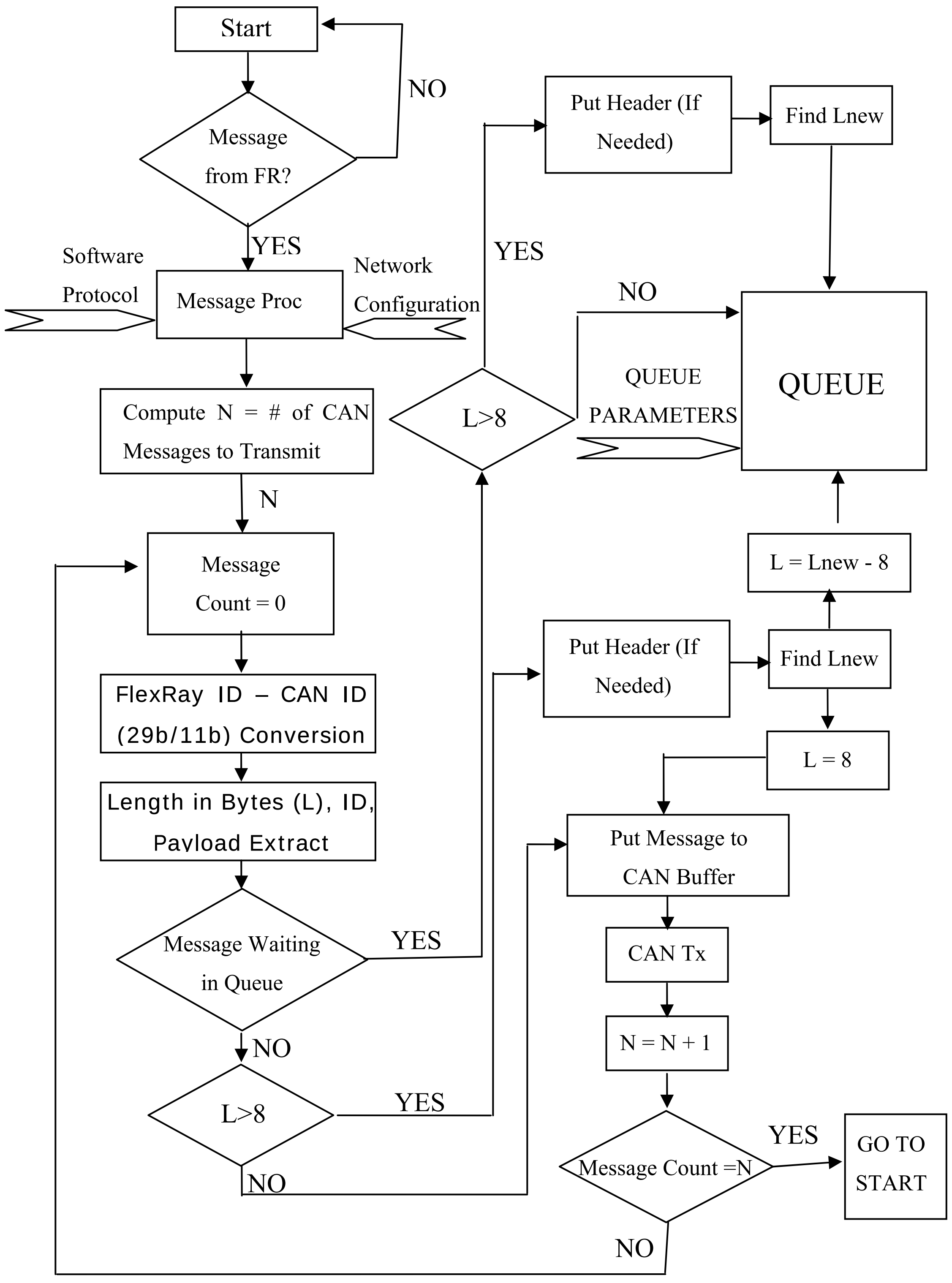


**Figure 3-1** FlexRay-to-CAN Gateway Functional Diagram

## 3.2 CAN-to-FLEXRAY GATEWAY FUNCTIONALITY DESIGN

CAN-to-FlexRay part of the Gateway is fairly straightforward when compared to FlexRay-to-CAN direction since the data rate and the frame length of FlexRay are greater than those of CAN bus. Therefore, Gateway does not have much difficulty in processing the burst of CAN data and worry about how to store and where to store the incoming data. CAN-to-FlexRay functionality performs basically CAN-FlexRay ID conversion according to the signal mapping in the network. On top of this, Gateway mainly focuses on elaborating the incoming CAN signals and packing them properly so that the network sources is used efficiently without compromising the delay requirements of the CAN signals. In a network composed of both FlexRay and CAN nodes, lots of the signals passing through the Gateway are addressed to the same FlexRay node. Therefore, it is wise for the Gateway to combine the signals properly such that not distinct FlexRay slots are allocated to each incoming CAN signal. This way network sources are utilized efficiently and CAN signals experience less delay. This packing mechanism in the Gateway is strongly related with network configuration. The factors such as length of the static slot in static segment, CAN signal frame length, CAN signal priority, arrival rate of the incoming CAN signals, mapping of the network and so forth, all, affect the way how the incoming signals will be elaborated in the Gateway. Buffer management and queuing requirements in FlexRay-to-CAN functionality of the Gateway are not likely to be needed in this part of the Gateway unless the CAN traffic is too crowded or Gateway structure is established badly. As a result, it can be said that the CAN-to-FlexRay Gateway, while, forwards signals according to the CAN-to-FlexRay signal mapping, it also accomplishes the signal transfer between the FlexRay and the CAN network by making use of the network parameters as described above.

Functional description of the Gateway is given in Figure 3-2.

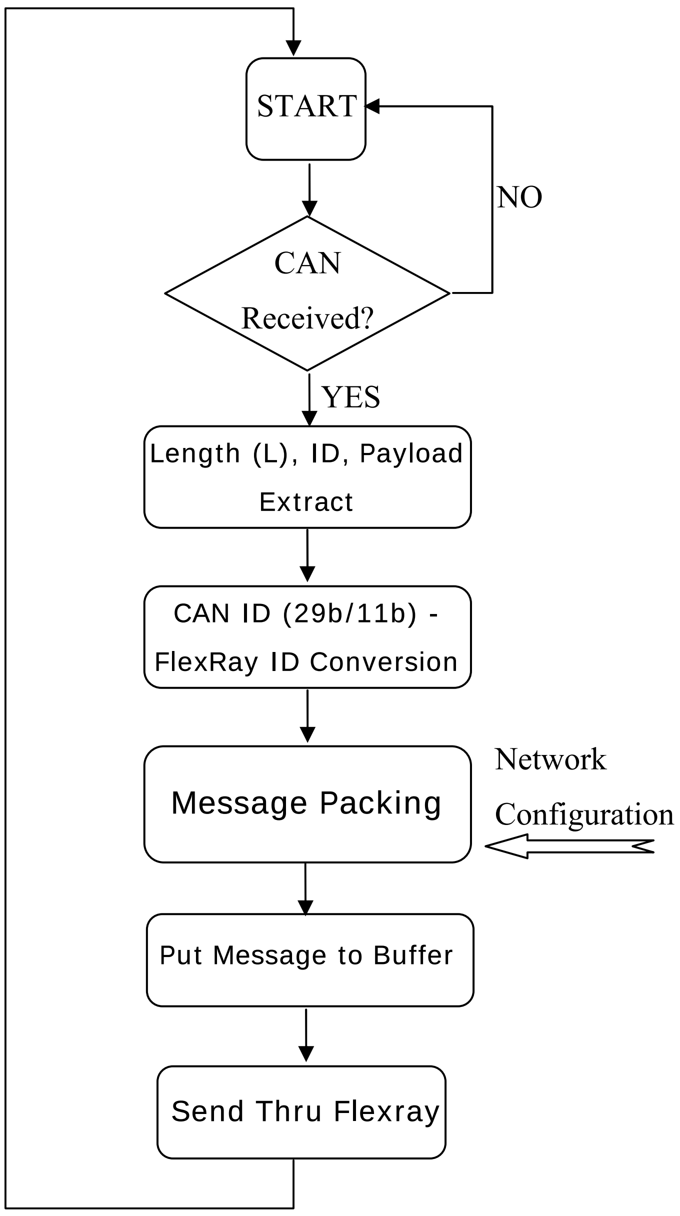


**Figure 3-2** CAN-to- FlexRay Gateway Functional Diagram

# CHAPTER 4

# DEVELOPMENT AND TEST ENVIRONMENT

The entire development, debugging, testing and the experimentation phases of the Gateway are done using hardware and software tools that are compliant with the automotive standards. All of these tools are mainly designed to be used for automotive applications, particularly for FlexRay. Though all of the tools are very new, since FlexRay protocol is recently maturing, they are fairly stable and worked well throughout the studies. In order to provide a better understanding about how the Gateway is developed and the experimentations are held, the tools and the hardware used during the studies will be explained briefly in the following sections.

## 4.1 SK-91465X-100MPC FUJITSU FLEXRAY EVALUATION BOARD

The SK-91465X-100MPC is a multifunctional evaluation board for the Fujitsu 32-bit Flash microcontroller series MB91F465XA (CPU) which is very efficient to be used in automotive applications. This Fujitsu FlexRay Evaluation Board is the main building block of the network composed of CAN nodes, FlexRay nodes and the Gateway node. In the experiments, this hardware is used as a distinct node as one of the three options, namely, FlexRay Node, CAN Node, Gateway Node. Besides the FlexRay and CAN support, the evaluation board is also compatible with LIN. As a whole, it has 2 FlexRay Channels, 2 CAN Channels, 2 LIN/UART Channels and 1 dedicated UART Channel.

FlexRay Channels of the board are the redundant Channels, namely Channel A and Channel B. Physical layer of the FlexRay via these channels is implemented by AMS8221B transceiver. As opposed to CAN, data to the transceiver is not delivered directly by the Microcontroller, MB91F465XA. Communication with the transceiver is provided by the MB88121 series Standalone Communication Controller. The function of the CPU is to control and configure the Communication Controller.

MB91F465XA supports up to 6 different CAN connections. However, only 2 of these connections are used in the Evaluation Board. So, it is possible to connect the Fujitsu FlexRay Evaluation board to two different CAN networks at the same time via the dedicated CAN channels located on it. TLE6250GV33 high speed transceivers are used on the board for the CAN communication.

Figure 4-1 shows the SK-91465X-100MPC multifunctional evaluation board.

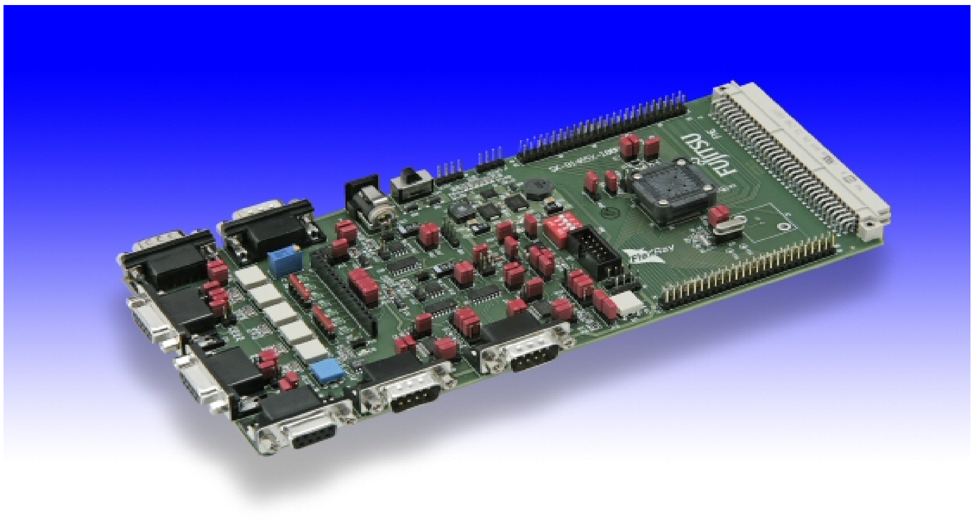
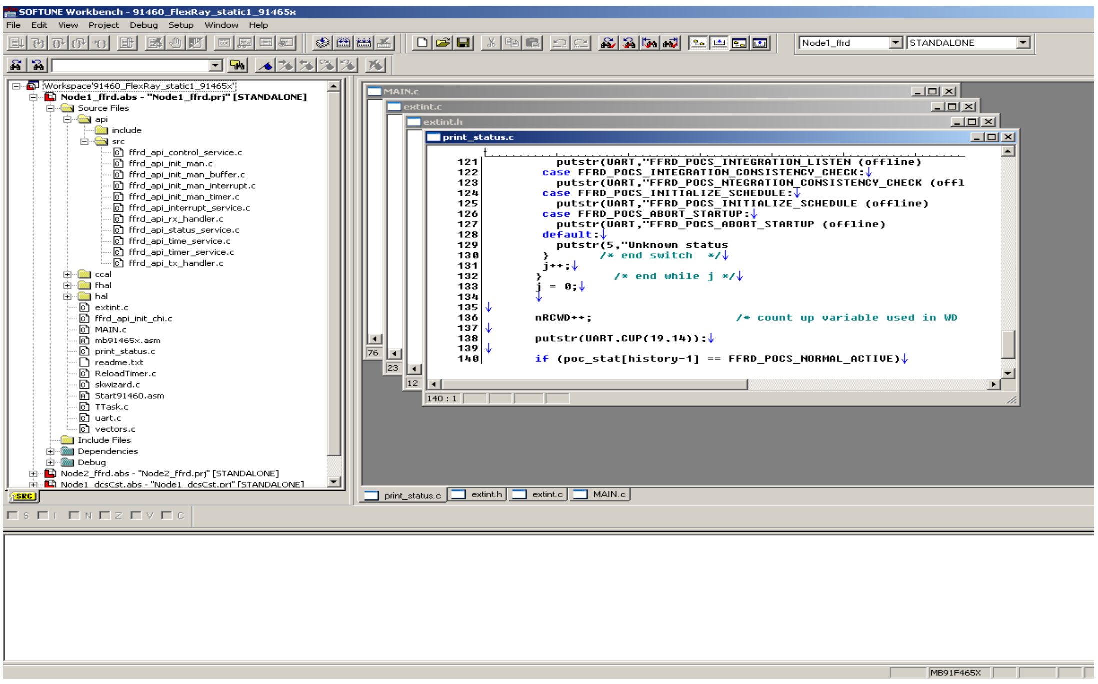


**Figure 4-1** SK-91465X-100MPC Evaluation Board

## 4.2 SOFTUNE WORKBENCH SOFTWARE DEVELOPMENT ENVIRONMENT

Softune Workbench, which is the propriety of FUJITSU, is the development environment for FR Family Microprocessors. The projects to be finally downloaded into the Fujitsu Microcontrollers are created, developed, manipulated, built and stored in Softune Software Development Environment. The Gateway and all the experiments are developed by the V60L06 version of the Softune Workbench. After compilation of the developed project, Softune Workbench creates a *.mhx file as the output. This *.mhx file can be directly downloaded into the flash memory of the CPU (MB91F465XA). Softune also creates a *.abs file which includes necessary information for the debugging process. Figure 4-2 shows a view from the Softune Workbench Development Environment.



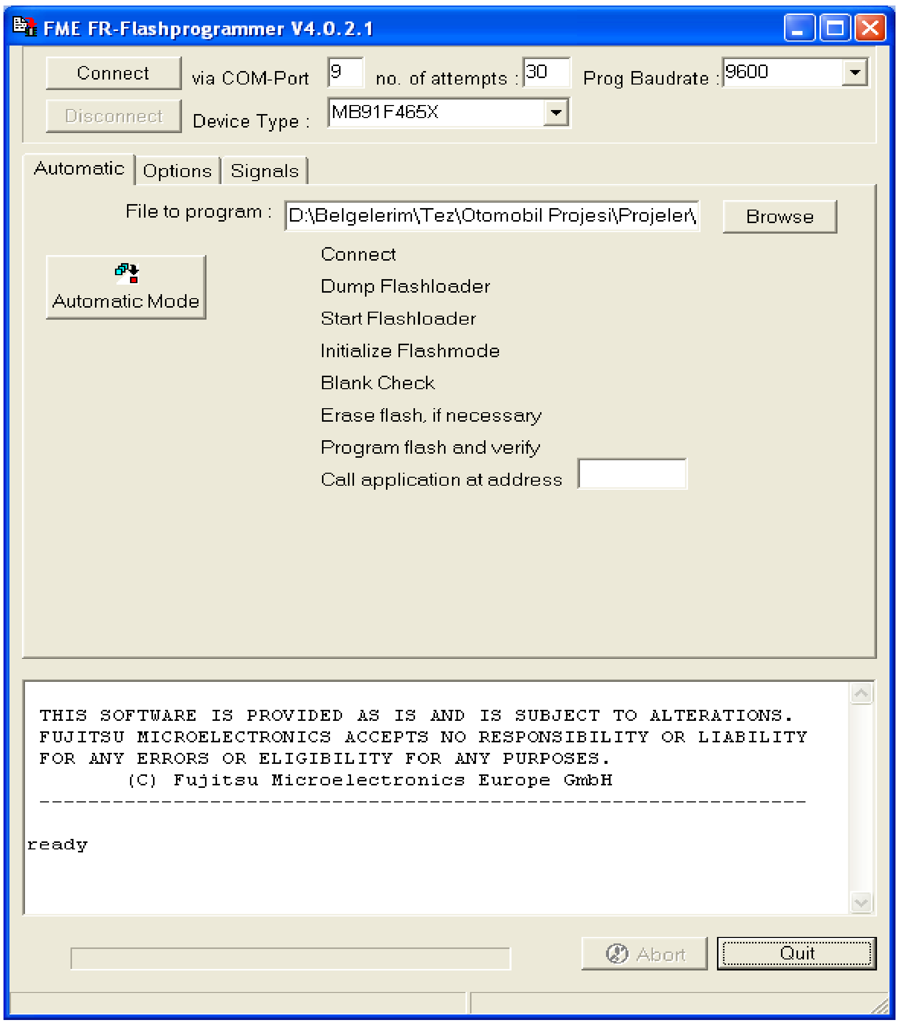
```
Workspace'91460_FlexRay_static1_91465x'
  Node1_ffrd.abs - "Node1_ffrd.prj" [STANDALONE]
    Source Files
      api
        include
        src
          ffrd_api_control_service.c
          ffrd_api_init_man.c
          ffrd_api_init_man_buffer.c
          ffrd_api_init_man_interrupt.c
          ffrd_api_init_man_timer.c
          ffrd_api_interrupt_service.c
          ffrd_api_rx_handler.c
          ffrd_api_status_service.c
          ffrd_api_time_service.c
          ffrd_api_timer_service.c
          ffrd_api_tx_handler.c
      ccal
      fhal
      hal
      extint.c
      ffrd_api_init_chi.c
      MAIN.c
      mb91465x.asm
      print_status.c
      readme.txt
      ReloadTimer.c
      skwizard.c
      Start91460.asm
      TTask.c
      uart.c
      vectors.c
    Include Files
    Dependencies
    Debug
  Node2_ffrd.abs - "Node2_ffrd.prj" [STANDALONE]
  Node1_dcsCst.abs - "Node1_dcsCst.prj" [STANDALONE]
```

```
          putstr(UART,"FFRD_POCS_INTEGRATION_LISTEN (offline)
        case FFRD_POCS_INTEGRATION_CONSISTENCY_CHECK:
          putstr(UART,"FFRD_POCS_NTEGRATION_CONSISTENCY_CHECK (offl
        case FFRD_POCS_INITIALIZE_SCHEDULE:
          putstr(UART,"FFRD_POCS_INITIALIZE_SCHEDULE (offline)
        case FFRD_POCS_ABORT_STARTUP:
          putstr(UART,"FFRD_POCS_ABORT_STARTUP (offline)
        default:
          putstr(5,"Unknown status
        }        /* end switch  */
        j++;
      }          /* end while j */
      j = 0;


      nRCWD++;                          /* count up variable used in WD

      putstr(UART,CUP(19,14));

      if (poc_stat[history-1] == FFRD_POCS_NORMAL_ACTIVE)
```

**Figure 4-2** FR Family SOFTUNE Workbench V60L06

## 4.3 FR-FLASH PROGRAMMER

FME FR-Flash Programmer V4.0.2.1, the propriety Fujitsu Microelectronics Europe GmbH, is used to download the projects to the flash memory of the microcontroller without requiring an emulator. FR-Flash Programmer facilitates to bury the code into the all kind of FR Family microcontroller via RS-232 serial port. It connects to the correct memory area (FLASH) of the CPU, goes in flash mode, erases the existing code in the flash and programs the flash with the machine code located in *.mhx file. A view from the flash programmer which has many additional advanced features is shown in Figure 4-3



via COM-Port: 9  no. of attempts: 30  Prog Baudrate: 9600

Device Type: MB91F465X

File to program: D:\Belgelerim\Tez\Otomobil Projesi\Projeler\

Connect
Dump Flashloader
Start Flashloader
Initialize Flashmode
Blank Check
Erase flash, if necessary
Program flash and verify
Call application at address

```
  THIS SOFTWARE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS.
  FUJITSU MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY
  FOR ANY ERRORS OR ELIGIBILITY FOR ANY PURPOSES.
         (C) Fujitsu Microelectronics Europe GmbH
  ----------------------------------------------------------------

ready
```

**Figure 4-3** FME FR-Flash Programmer V4.0.2.1

## 4.4 FLEXRAY COMMUNICATION CONTROLLER DRIVER

In the SK-91465X-100MPC message handling and communication tasks for the FlexRay bus are not performed by the CPU but rather these tasks are handled by a specific Communication Controller located on the board. The Communication Controller used in the board is Bosch ERay series Standalone Communication Controller [32]. The MCU's mission is to configure this Communication Controller and to read/write from/to the registers of the controller. Therefore, Fujitsu Microelectronics Europe GmbH offers a FlexRay Communication Controller Driver to perform all necessary communication between the MCU and the Communication Controller and to provide software to facilitate the evaluation of the FlexRay. The aim of Fujitsu in providing such a tool is to save the users from dealing with the dedicated registers in the beginning of their evaluations since getting familiar with a new bus system like FlexRay bus requires quite a time.

Besides, FlexRay Communication Controller Driver provides an environment for the user to easily configure the Communication Controller, which is Bosch ERay module in the Evaluation Board, it also includes numerous of API (Application Programming Interface) functions to evaluate the FlexRay network. The configuration of the Communication Controller with the driver can be performed in two ways. The first way is to manually program the dedicated registers in Bosch ERay via the driver software. The other and more user friendly option is to use a dedicated program, called FlexConfig, to configure FlexRay bus. This program, which will be explained in the following chapter, outputs a *.chi file which is recognized and can be directly used by FlexRay Communication Controller Driver. Therefore, once a proper *.chi file describing the configuration of the network is included in the project, FlexRay Driver automatically uses the file and makes the necessary settings according to the configuration of the network. FlexRay Communication Controller Driver is built upon a layered architecture which contains four layers named as Application Programming Interface (API), Communication Controller Abstraction Layer (CCAL), FlexRay Hardware

Abstraction Layer (FHAL) and Hardware Abstraction Layer (HAL) as shown in Figure 4-4.

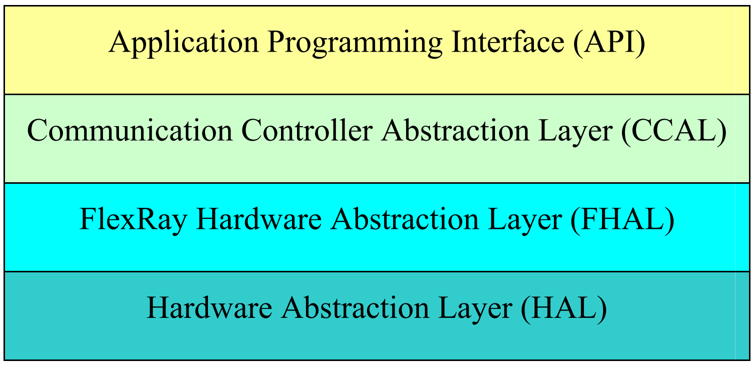

Application Programming Interface (API)
Communication Controller Abstraction Layer (CCAL)
FlexRay Hardware Abstraction Layer (FHAL)
Hardware Abstraction Layer (HAL)

**Figure 4-4** FlexRay Communication Controller Driver Layer Concept [33]

API layer as its name implies provides variety of functions to the user to evaluate the FlexRay bus. In other words it can be said that API layer is the user interface of the Fujitsu FlexRay Driver. API layer of the driver provides the user with more than 90 functions which can be categorized in various services as listed below.

- Initialization Services (by "*.chi" files or manually)
- Control Service
- Interrupt Services
- Reception (Rx) Services
- Status Information Services
- Time Services
- Timer Services
- Transmission (Tx) Services

CCAL layer contains the routines for the driver while FHAL is the layer where the FlexRay hardware description (Bosch ERay) is done. Finally the read/write operations via the control hardware (CPU, MB91F465XA) are defined in the HAL layer. The relations of these 4 layers between each other are strongly related with the working principle of the driver. According to the architecture of the driver, the user application calls one of the API functions (ffrd_api_functionname) for the Fujitsu FlexRay Driver. This function evaluates and calls pertain routine ffrd_ccal_functionname ( ). This layer includes all routines for computing values, register settings, buffer requests and interrupt routines etc.

The ffrd_ccal_functionname ( ) calls the Macro from ffrd_fhal_functionname ( ). In this layer the address offset for the E-Ray address is added.

In ffrd_hal_function ( ) the macros for different MCU-FlexRay Controller access placed.

The Fujitsu FlexRay Driver is developed flexible to be used for various hardware combinations therefore files, macros and functions are included in the software if needed only. The principle driver architecture is shown in the Figure 4-5.

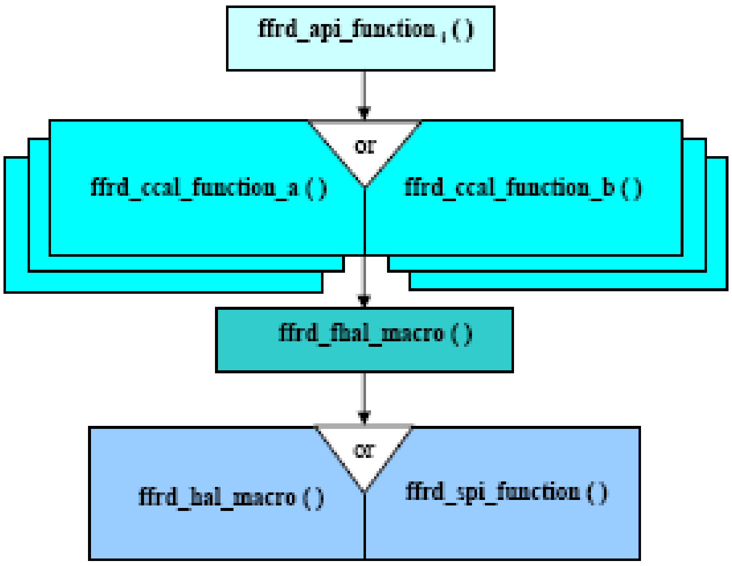
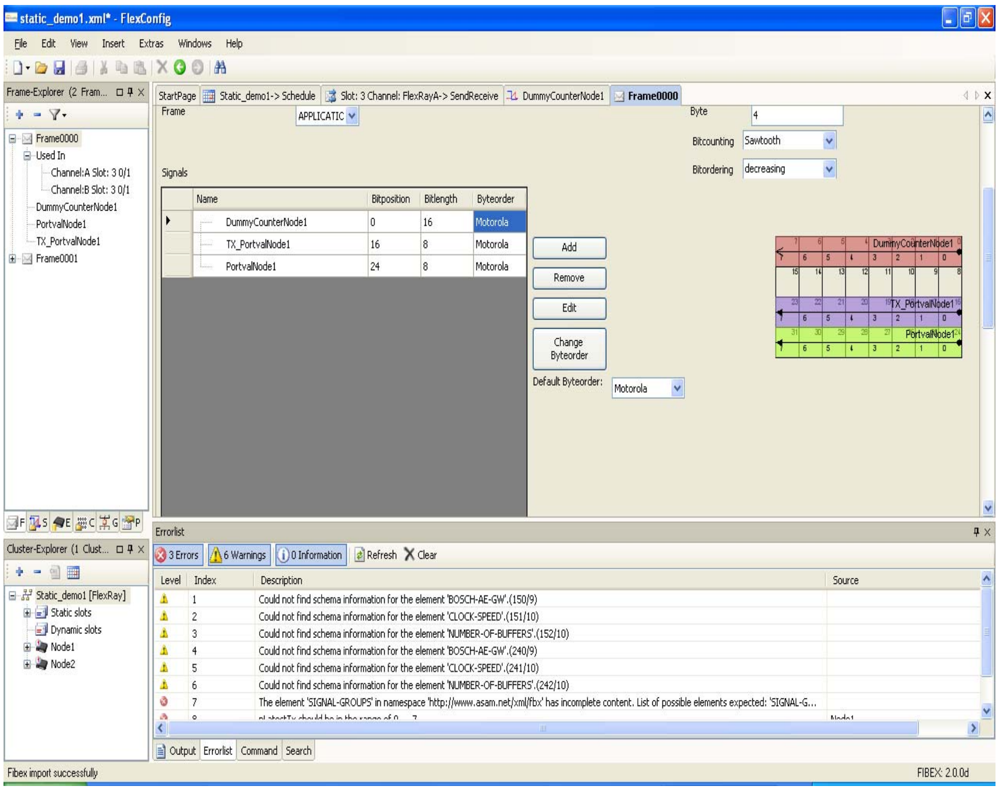


**Figure 4-5** FlexRay Communication Controller Driver Architecture [33]

## 4.5 FLEXCONFIG™ DEVELOPER – UNIVERSAL FLEXRAY CONFIGURATION TOOL

As discussed in FlexRay Communication Controller Driver part, FlexRay network parameters can be configured either manually or automatically by means of *.chi file. FlexConfig™ Developer is a software which facilitates to configure the numerous parameters of the FlexRay network with its user friendly interface and outputs a corresponding *.chi file to be used in the FlexRay Driver. Version S3V0-F of the FlexConfig™ Developer is used throughout the all development and the experimentation phases of the study. By means of FlexConfig™ Developer all possible network parameters of the FlexRay bus defined in the FlexRay protocol specification can be set and modified very easily. Therefore, it is not possible, by using the FlexConfig™, to leave a network parameter unconfigured which might lead the network to undetermined and unstable states. Moreover, using this tool in the configuration of the FlexRay network is less prone to errors when compared to configuring it by handling all the dedicated registers manually. Because FlexConfig™ checks all the parameters entered by the user to identify if there exists any non-conformances and, if there is, indicates these in the form of warnings and errors. Even before the user sets any parameter, FlexConfig™ guides the user about the limit values of the field to be entered, by taking all the other current parameter values of the network into account. Also, the software shows to the user which parameters of the network would be directly affected if the parameter pointed out by the cursor was changed which provides the user a broader perspective about the network. Scheduling, Frame ID allocation, fixing the Static Slot length, choosing the macrotick duration, deciding the payload amount, defining the synchronization and the start-up nodes are just a few examples among the many others a user can set by using the FlexConfig™.

A view from the user interface of the FlexConfig™ is shown in Figure 4-6.



Frame: APPLICATIC
Byte: 4
Bitcounting: Sawtooth
Bitordering: decreasing

Signals

| Name | Bitposition | Bitlength | Byteorder |
|---|---|---|---|
| DummyCounterNode1 | 0 | 16 | Motorola |
| TX_PortvalNode1 | 16 | 8 | Motorola |
| PortvalNode1 | 24 | 8 | Motorola |

Default Byteorder: Motorola

Errorlist: 3 Errors, 6 Warnings, 0 Information

| Level | Index | Description | Source |
|---|---|---|---|
| ⚠ | 1 | Could not find schema information for the element 'BOSCH-AE-GW'.(150/9) | |
| ⚠ | 2 | Could not find schema information for the element 'CLOCK-SPEED'.(151/10) | |
| ⚠ | 3 | Could not find schema information for the element 'NUMBER-OF-BUFFERS'.(152/10) | |
| ⚠ | 4 | Could not find schema information for the element 'BOSCH-AE-GW'.(240/9) | |
| ⚠ | 5 | Could not find schema information for the element 'CLOCK-SPEED'.(241/10) | |
| ⚠ | 6 | Could not find schema information for the element 'NUMBER-OF-BUFFERS'.(242/10) | |
| ● | 7 | The element 'SIGNAL-GROUPS' in namespace 'http://www.asam.net/xml/fbx' has incomplete content. List of possible elements expected: 'SIGNAL-G... | |

Fibex import successfully
FIBEX: 2.0.0d

**Figure 4-6** FlexConfig™ User Interface

## 4.6 FLEXCARD CYCLONE II SE

FlexCard Cyclone II SE is the network analyzer hardware used for the evaluation of the performance metrics of both the FlexRay bus and the CAN bus. It is 32-bit CardBus Card which is interfaced with a personal computer through PCMCIA slot. FlexCard Cyclone II SE supports two redundant FlexRay channels as well as two high speed CAN channels. Therefore, with this hardware, it is very easy to make a performance analysis of a Gateway network which includes both CAN and FlexRay network since the FlexCard Cyclone facilitates the user to use its CAN and FlexRay interfaces at the same time. Apart from listening to the bus connected to it, FlexCard Cyclone II SE can also send messages to both CAN and FlexRay networks. While Bosch E-Ray communication controller, which is the communication controller used in this study, core is included in it, FlexCard Cyclone has 2MB memory for buffering the incoming data which makes it very precise in tracking and logging the network traffic. A picture of the FlexCard Cyclone II SE hardware is given in Figure 4-7.

**Figure 4-7** FlexCard Cyclone II SE

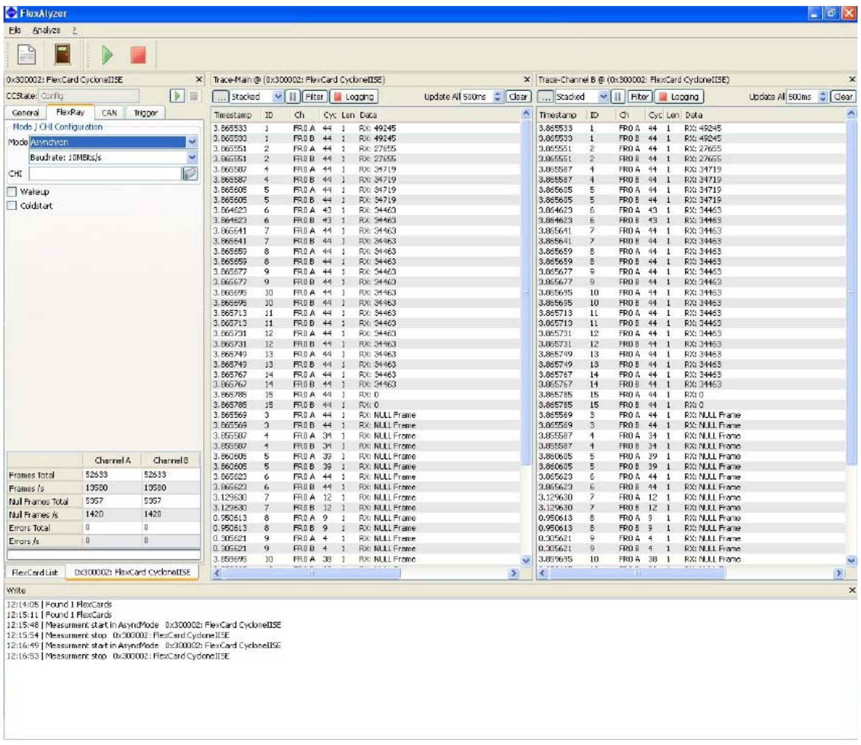
## 4.7 FLEXALYZER

The FlexAlyzer is the software which is developed to operate in accordance with the FlexCard Cyclone II SE to monitor and analyze FlexRay and CAN network traffic. Both of FlexRay and CAN network traffic can be tracked and analyzed by FlexAlyzer software at the same time. Monitoring in FlexRay bus can be performed in synchronous and asynchronous mode. The synchronous mode, which is used throughout this study, gives out more accurate results with respect to asynchronous mode. To work in synchronous mode, FlexAlyzer software requires a dedicated *.chi file which describes the scheduling of the FlexRay bus and when to send/receive for the FlexCard Cyclone according to this scheduling. Since the FlexCard Cyclone has the Bosch E-Ray communication controller core in it, the registers of it can be configured by means of a *.chi file. In this context, FlexCard can be seen as a distinct node in the network which can send/receive messages to/from the network. Therefore, when FlexRay network is configured by FlexConfig™, the time slots, generally the all, where the FlexAlyzer needs to monitor the bus must be marked as receive slots for FlexCard. On top of it, if in some time slots, FlexCard is required to send data, then those slots must be chosen as transmit slot for the FlexCard. However, throughout this study, no data is sent by FlexCard. After the configuration has been finished, FlexConfig outputs a dedicated *.chi file for each of the nodes connected to the network, one of which is FlexCard. When the *.chi file for the FlexCard Cyclone is included in the FlexAlyzer, the software begins to operate properly. The FlexAlyzer software shows in the monitor the incoming payload, its data length, frame id and the receive cycle for both Channel A and Channel B as well as the diagnostic data like, network info, flags, CRC, errors and so forth. FlexAlyzer is capable of working with multiple FlexCards at the same time. In such a case, the software gives the monitoring information of each hardware in distinct windows.

FlexAlyzer software can also be used to monitor and analyze the CAN bus. FlexCard can be used to transmit data for CAN network too. For the FlexCard to be ready to operate on a CAN bus, the communication data rate must be set correctly and the CAN bus must be terminated with 120 ohm. Once these are done, the software will immediately begin to monitor the CAN bus. FlexCard has two interfaces on it which can be used for either CAN or FlexRay according to the application. Therefore, if required, both of the interfaces can be used to monitor two different CAN buses with different data rate. In total, FlexAlyzer software is capable of monitoring 8 different CAN buses at the same time and giving the information to the output window. The FlexAlyzer, when monitoring the CAN bus, shows in the monitor the local time stamp for the receive time, the CAN id and the payload.

The FlexAlyzer software has a user friendly architecture which provides user with various flexibilities. Filtering the data to show in the monitor according to numerous different criteria, taking log of the network traffic without any time limitation, displaying the data in decimal or hexadecimal format are some of the examples to the user friendly structure of the FlexAlyzer software.

A view from FlexAlyzer user interface is shown in Figure 4-8.



0x300002: FlexCard CycloneIISE

CCState: Config

Mode / CHI Configuration

Mode: Asynchron

Baudrate: 10MBits/s

CHI:

☐ Wakeup

☐ Coldstart

| | Channel A | Channel B |
|---|---|---|
| Frames Total | 52633 | 52633 |
| Frames /s | 13580 | 13580 |
| Nul Frames Total | 5357 | 5357 |
| Nul Frames /s | 1420 | 1420 |
| Errors Total | 0 | 0 |
| Errors /s | 0 | 0 |

Trace-Main @ (0x300002: FlexCard CycloneIISE)

Update All 500ms

| Timestamp | ID | Ch | Cyc | Len | Data |
|---|---|---|---|---|---|
| 3.865533 | 1 | FR0 A | 44 | 1 | RX: 49245 |
| 3.865533 | 1 | FR0 B | 44 | 1 | RX: 49245 |
| 3.865551 | 2 | FR0 A | 44 | 1 | RX: 27655 |
| 3.865551 | 2 | FR0 B | 44 | 1 | RX: 27655 |
| 3.865587 | 4 | FR0 A | 44 | 1 | RX: 34719 |
| 3.865587 | 4 | FR0 B | 44 | 1 | RX: 34719 |
| 3.865605 | 5 | FR0 A | 44 | 1 | RX: 34719 |
| 3.865605 | 5 | FR0 B | 44 | 1 | RX: 34719 |
| 3.864623 | 6 | FR0 A | 43 | 1 | RX: 34463 |
| 3.864623 | 6 | FR0 B | 43 | 1 | RX: 34463 |
| 3.865641 | 7 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865641 | 7 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865659 | 8 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865659 | 8 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865677 | 9 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865677 | 9 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865695 | 10 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865695 | 10 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865713 | 11 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865713 | 11 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865731 | 12 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865731 | 12 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865749 | 13 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865749 | 13 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865767 | 14 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865767 | 14 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865785 | 15 | FR0 A | 44 | 1 | RX: 0 |
| 3.865785 | 15 | FR0 B | 44 | 1 | RX: 0 |
| 3.865569 | 3 | FR0 A | 44 | 1 | RX: NULL Frame |
| 3.865569 | 3 | FR0 B | 44 | 1 | RX: NULL Frame |
| 3.855587 | 4 | FR0 A | 34 | 1 | RX: NULL Frame |
| 3.855587 | 4 | FR0 B | 34 | 1 | RX: NULL Frame |
| 3.860605 | 5 | FR0 A | 39 | 1 | RX: NULL Frame |
| 3.860605 | 5 | FR0 B | 39 | 1 | RX: NULL Frame |
| 3.865623 | 6 | FR0 A | 44 | 1 | RX: NULL Frame |
| 3.865623 | 6 | FR0 B | 44 | 1 | RX: NULL Frame |
| 3.129630 | 7 | FR0 A | 12 | 1 | RX: NULL Frame |
| 3.129630 | 7 | FR0 B | 12 | 1 | RX: NULL Frame |
| 0.950613 | 8 | FR0 A | 9 | 1 | RX: NULL Frame |
| 0.950613 | 8 | FR0 B | 9 | 1 | RX: NULL Frame |
| 0.305621 | 9 | FR0 A | 4 | 1 | RX: NULL Frame |
| 0.305621 | 9 | FR0 B | 4 | 1 | RX: NULL Frame |
| 3.859695 | 10 | FR0 A | 38 | 1 | RX: NULL Frame |

Trace-Channel B @ (0x300002: FlexCard CycloneIISE)

Update All 500ms

| Timestamp | ID | Ch | Cyc | Len | Data |
|---|---|---|---|---|---|
| 3.865533 | 1 | FR0 A | 44 | 1 | RX: 49245 |
| 3.865533 | 1 | FR0 B | 44 | 1 | RX: 49245 |
| 3.865551 | 2 | FR0 A | 44 | 1 | RX: 27655 |
| 3.865551 | 2 | FR0 B | 44 | 1 | RX: 27655 |
| 3.865587 | 4 | FR0 A | 44 | 1 | RX: 34719 |
| 3.865587 | 4 | FR0 B | 44 | 1 | RX: 34719 |
| 3.865605 | 5 | FR0 A | 44 | 1 | RX: 34719 |
| 3.865605 | 5 | FR0 B | 44 | 1 | RX: 34719 |
| 3.864623 | 6 | FR0 A | 43 | 1 | RX: 34463 |
| 3.864623 | 6 | FR0 B | 43 | 1 | RX: 34463 |
| 3.865641 | 7 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865641 | 7 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865659 | 8 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865659 | 8 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865677 | 9 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865677 | 9 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865695 | 10 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865695 | 10 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865713 | 11 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865713 | 11 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865731 | 12 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865731 | 12 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865749 | 13 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865749 | 13 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865767 | 14 | FR0 A | 44 | 1 | RX: 34463 |
| 3.865767 | 14 | FR0 B | 44 | 1 | RX: 34463 |
| 3.865785 | 15 | FR0 A | 44 | 1 | RX: 0 |
| 3.865785 | 15 | FR0 B | 44 | 1 | RX: 0 |
| 3.865569 | 3 | FR0 A | 44 | 1 | RX: NULL Frame |
| 3.865569 | 3 | FR0 B | 44 | 1 | RX: NULL Frame |
| 3.855587 | 4 | FR0 A | 34 | 1 | RX: NULL Frame |
| 3.855587 | 4 | FR0 B | 34 | 1 | RX: NULL Frame |
| 3.860605 | 5 | FR0 A | 39 | 1 | RX: NULL Frame |
| 3.860605 | 5 | FR0 B | 39 | 1 | RX: NULL Frame |
| 3.865623 | 6 | FR0 A | 44 | 1 | RX: NULL Frame |
| 3.865623 | 6 | FR0 B | 44 | 1 | RX: NULL Frame |
| 3.129630 | 7 | FR0 A | 12 | 1 | RX: NULL Frame |
| 3.129630 | 7 | FR0 B | 12 | 1 | RX: NULL Frame |
| 0.950613 | 8 | FR0 A | 9 | 1 | RX: NULL Frame |
| 0.950613 | 8 | FR0 B | 9 | 1 | RX: NULL Frame |
| 0.305621 | 9 | FR0 A | 4 | 1 | RX: NULL Frame |
| 0.305621 | 9 | FR0 B | 4 | 1 | RX: NULL Frame |
| 3.859695 | 10 | FR0 A | 38 | 1 | RX: NULL Frame |

```
12:14:05 | Found 1 FlexCards
12:15:11 | Found 1 FlexCards
12:15:48 | Measurment start in AsyncMode   0x300002: FlexCard CycloneIISE
12:15:54 | Measurment stop   0x300002 : FlexCard CycloneIISE
12:16:49 | Measurment start in AsyncMode   0x300002: FlexCard CycloneIISE
12:16:53 | Measurment stop   0x300002 : FlexCard CycloneIISE
```

**Figure 4-8** FlexAlyzer User Interface

# CHAPTER 5

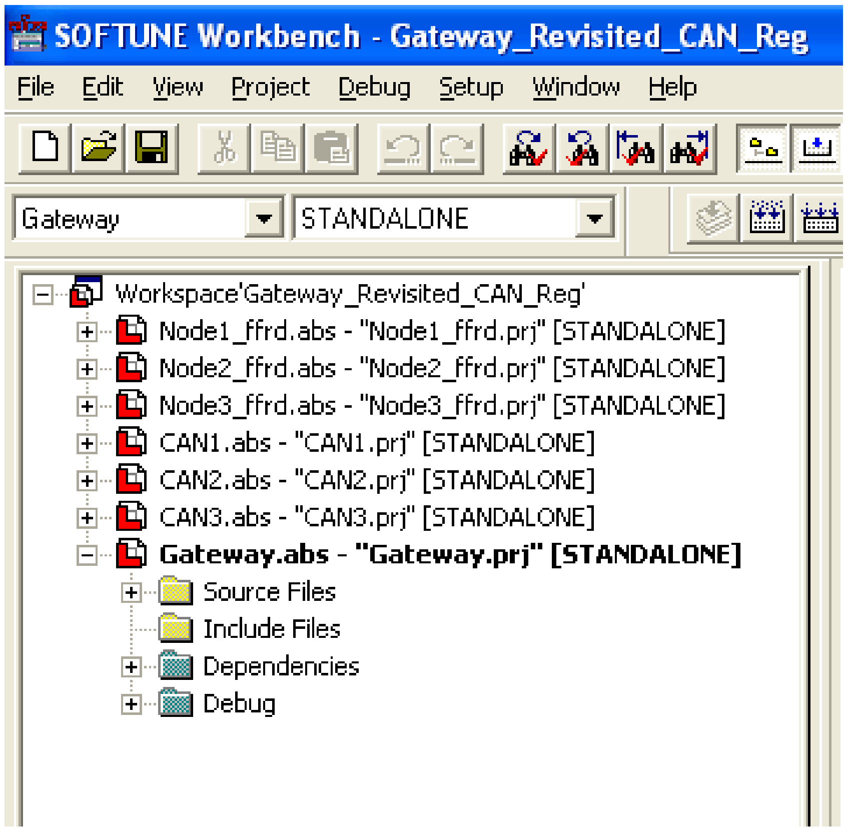
# FLEXRAY-CAN GATEWAY IMPLEMENTATION

In this chapter we provide the details of all design and development phases of the FlexRay-CAN Gateway. In this context, the discussion of the details of the FlexRay-CAN Gateway design can be categorized under three sub-titles which includes the considerations specific to the Gateway experiment, critical issues to be taken into account about CAN and FlexRay protocols and the general structure of the workflow in the software.

In order to clearly explain all the design considerations without missing any detail, after having gone over the general structure of the design, the critical parts of the FlexRay-CAN Gateway project will be handled exclusively. Finally, the remarks about the FlexRay Communication Controller Driver, the experiences obtained while working on auxilary hardware and software such as analysis tools, and additional design considerations will be presented.

## 5.1 GENERAL ARCHITECTURE OF THE GATEWAY

A Softune Workbench workspace consists of the collection of a number of projects each of which stands for a distinct node of the network. In this context, as it will be detailed in Section 6.3, the workspace of the FlexRay-CAN Gateway experiment is composed of 7 projects for 3 FlexRay nodes, 3 CAN nodes and a Gateway node. This is shown in Figure 5-1 below. All the tasks such as message receiving, message sending, time management, buffer handling and so forth for the FlexRay nodes are covered in the projects named "Node1_ffrd.prj", "Node2_ffrd.prj" and "Node3_ffrd.prj". Similarly, the tasks for CAN nodes are included in the CAN projects named "CAN1.prj", "CAN2.prj" and "CAN3.prj". "Gateway.prj" is the project where the Gateway tasks are implemented. After being developed and compiled, *.mhx file for each of the project is downloaded into the distinct node of the network via FR-Flashprogrammer.


SOFTUNE Workbench - Gateway_Revisited_CAN_Reg

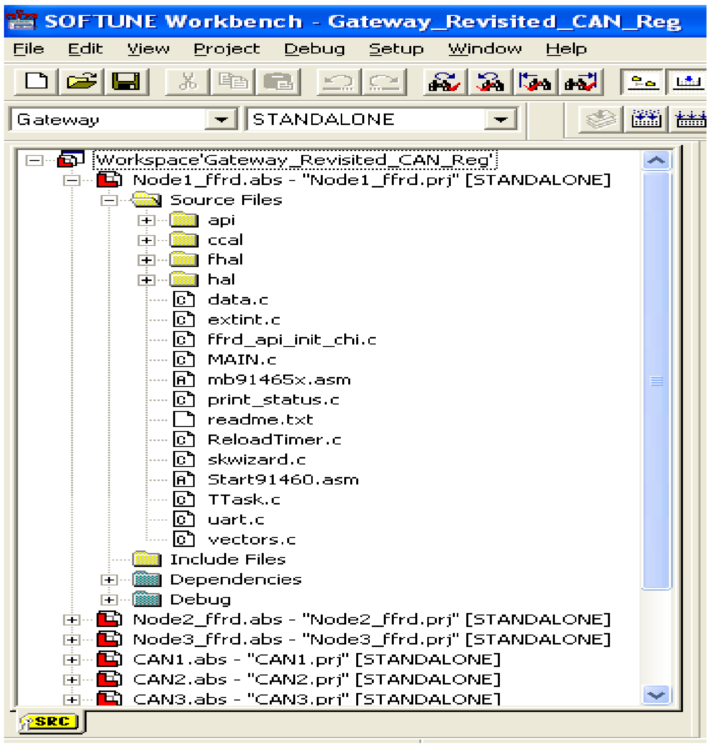
Gateway | STANDALONE

```
Workspace'Gateway_Revisited_CAN_Reg'
  Node1_ffrd.abs - "Node1_ffrd.prj" [STANDALONE]
  Node2_ffrd.abs - "Node2_ffrd.prj" [STANDALONE]
  Node3_ffrd.abs - "Node3_ffrd.prj" [STANDALONE]
  CAN1.abs - "CAN1.prj" [STANDALONE]
  CAN2.abs - "CAN2.prj" [STANDALONE]
  CAN3.abs - "CAN3.prj" [STANDALONE]
  Gateway.abs - "Gateway.prj" [STANDALONE]
    Source Files
    Include Files
    Dependencies
    Debug
```

**Figure 5-1** Projects in the FlexRay Gateway Workspace

The coding structure of the projects is as shown in Figure 5-2. Top-most folders starting with the name "api" and ending with "hal" are the places where the FlexRay Communication Controller Driver software is located. The FlexRay Driver is required to facilitate the communication with the FlexRay Communication Controller and the MCU. Therefore, it is needed to be used, naturally, in FlexRay projects and also in the Gateway project since the Gateway also interfaces with the

FlexRay bus during its operation. In order to make correct time measurements and obtain reliable values for the performance metrics, CAN nodes are to be synchronized to FlexRay network. Hence, in all experiments, FlexRay software driver is, also, included in the CAN projects. The source files which perform the main functions of the node are located under the FlexRay software driver folders. While the header files of the software driver are located automatically in the "Dependencies" folder, the user-created header files might be located in either "Include" folder or the "Dependencies" folder.


SOFTUNE Workbench - Gateway_Revisited_CAN_Reg
File Edit View Project Debug Setup Window Help
Gateway
STANDALONE
SRC

```
Workspace'Gateway_Revisited_CAN_Reg'
  Node1_ffrd.abs - "Node1_ffrd.prj" [STANDALONE]
    Source Files
      api
      ccal
      fhal
      hal
      data.c
      extint.c
      ffrd_api_init_chi.c
      MAIN.c
      mb91465x.asm
      print_status.c
      readme.txt
      ReloadTimer.c
      skwizard.c
      Start91460.asm
      TTask.c
      uart.c
      vectors.c
    Include Files
    Dependencies
    Debug
  Node2_ffrd.abs - "Node2_ffrd.prj" [STANDALONE]
  Node3_ffrd.abs - "Node3_ffrd.prj" [STANDALONE]
  CAN1.abs - "CAN1.prj" [STANDALONE]
  CAN2.abs - "CAN2.prj" [STANDALONE]
  CAN3.abs - "CAN3.prj" [STANDALONE]
```

**Figure 5-2** Coding Structure of the Projects

Some of the source files namely, Start91460.asm, ffrd_api_int_chi.c, mb91465x.h, vector.c and print_status.c are common to all projects. These files are important and fundamental for the projects to work properly.

To begin with, Start91460.asm makes all the necessary settings of the 91460 series Fujitsu chips. When powered on, CPU first runs this code and adjusts its own settings. In another saying, Start91460.asm is the boot-up code of the 91460 series Fujitsu microcontroller. Via the Start91460.asm file, the user can select the controller device to work as the CPU from the options of the 91460 series Fujitsu chips, perform all the memory management tasks, initialize the stacks, make the Boot-Flash security settings, determine the clock speed of all kind of clocks exist in the CPU, such as the peripheral clock, main clock, external clock, CAN clock an so forth and perform many other task to make the CPU ready-to-go after power up. Although Start91460.asm has never been modified during this study, it is required to be modified accordingly when a project is to be debugged via Accemic Debugger tool for Fujitsu Microcontrollers.

The file named ffrd_api_int_chi.c is also common to all projects. This source file is used to configure the FlexRay network automatically via a corresponding *.chi (controller host interface) file. This source code is called only if the initialization mode of the FlexRay Software driver, which is "FFRD_INIT_MODE", is defined as "CHI". This definition and many other settings about the FlexRay Communication Controller Driver are located in the header file named "ffrd_api_global_def.h". Another option to be selected for the configuration of the FlexRay network in "ffrd_api_global_def.h" is "MAN" which stands for the manual configuration of the FlexRay Communication Controller meaning to input all the dedicated register values by hand.

If "CHI" is selected as "FFRD_INIT_MODE" in the "ffrd_api_global_def.h" file, then the name of the *.chi file must be entered in the proper place of the code by the user as shown below.

```
void ffrd_api_include_chi ( void )
{
#include "Controller_Name.chi" /* add your *.chi file */
}
```

The *.chi file whose name is added in the source code should also be included in the project. The Fujitsu FlexRay Software Driver searches for the *.chi file in the following address: Root\Generated_files\src_FlexConfig. The "Root" in the address is the place where the workspace file, *.wsp, is located. Having performed these steps, as a result of the compilation of the project, the *.chi file, which is included in the above address and whose name is added in ffrd_api_include_chi.c, appears under the "Dependencies" folder of the project automatically and the configuration of the Communication Controller has been successfully achieved.

"mb91465x.h" is a very important header file which has to be used in all projects. Because all of the registers constructing the microcontroller, MB91465XA, are defined in this file according to their memory addresses in the CPU. Moreover, by making use of the "union" structure of "C", all registers are structured to be accessible in bit, byte, half-word (16-bit) or word (32-bit) level as whichever of those are useful. Therefore, mb91465x.h header file has to be included in all of those source files and the header files where a register of the microcontroller, MB91465XA, is to be used.

Finally, "vector.c" and "print_status.c" are the files common in all experiments that worth making a few words about. Vector.c is a very fundamental source file which is responsible for the Interrupt management in the project. Though, the "interrupt sub-routines" for specific interrupt sources might be located in different source files of the project, the necessary settings about the interrupts are handled in this file. In Vector.c, priority levels of the interrupt sources are arranged, the interrupt sub-routines are defined and the vector address, which is the memory address where the running code would jump and execute the interrupt subroutine, of the interrupt sources are set.

Print_status.c is the file which includes the idle task running in the experiments. As it will be explained in more details later, in all experiments, there exists an idle task, of which name is "printFlexRayStatus", running all the time in an infinite "for" loop. Via "printFlexRayStatus" routine, the standing status of the node in FlexRay bus is exported as ASCII characters through one of the RS-232 interfaces of the SK-91465X-100MPC evaluation board. Therefore, by connecting the PC to the serial interface of the evaluation board through the HyperTerminal software, the user can observe the standing status of the node, which can be "Online", "Offline", "Wakeup Listen", "Wakeup Standby" and so on.

Having explained the general architecture of the Gateway workspace and given detailed information about the fundamental source codes which are included in all of the projects in all experiments, in the following chapters, the tasks performed in the FlexRay nodes, CAN nodes and the Gateway node will be discussed, in order, more deeply.

## 5.2 FLEXRAY PROJECT DETAILS

The FlexRay projects in the Gateway experiment are "Node1_ffrd.prj", "Node2_ffrd.prj" and "Node3_ffrd.prj" as shown in Figure 5-1. Since they are built on the same architecture and only their related messages and their scheduling differ from each other, "Node1_ffrd.prj" is selected as a generic FlexRay project to explain in detail.

In the broadest sense, in the FlexRay projects two tasks are executed. Idle task is the task which is always running in the project. "printFlexRayStatus" function, which is the function to send FlexRay standing status as explained in the previous chapter, is realized in the Idle task. The FlexRay task which is responsible for the handling of the FlexRay message exchange, is executed upon a dedicated interrupt and is the second task running in the project. The main structure of the code in the FlexRay project is built on this "idle task" - "interrupt" cycle. In fact, this structure also applies to CAN projects and the Gateway project which will be detailed in the following chapters.

### 5.2.1 Tasks Executed In MAIN.c

The steps that have been taken during the design of the FlexRay node will be explained and discussed beginning with the "MAIN.c" file of the project. If the "MAIN.c" file, which is included in APPENDIX A, is examined, it can be seen that, first, all interrupts and ports are enabled since, after power-on, the default attribute for the interrupts and the ports is "disabled". Also the hardware watchdog is cleared to initialize the code.

```
        __EI();              /* enable interrupts */
        __set_il(31);        /* allow all levels */
        HWWD = 0x10;         /* clear HW watchdog */
        PORTEN = 0x3;        /* enable I/O Ports */
```

Then, in the flow of the main(), some CPU registers that are not set in Start91460.asm and the interrupt vector table are initialized by InitCPUExtraRegs() and InitIrqLevels(), respectively. InitIrqLevels() is a routine in the file named vector.c. As explained in the previous chapter, priority levels of the interrupts and the vector addresses of the interrupt sub-routines are set in "vector.c".

On the other hand, port 16 and port 27, which are connected to the LEDs in the starter kit, are set to I/O port, UART4 of the evaluation board is initialized to be used as the serial communication channel with the personal computer and a number of "reload timers" are set to their initial values in InitCPUExtraRegs(). Six distinct reload timers are used in the FlexRay tasks. One of them is used to clear watchdog periodically and one another is loaded to issue interrupts to produce FlexRay static messages. The remaining four "reload timers" are used as time ticks for the generation of the sporadic dynamic segment messages. In the main flow, after InitCPUExtraRegs(), all those 6 reload timers are started so as to periodically interrupt the code for their individual tasks. Meanwhile, FlexRay driver and Communication Controller are initialized by ttStartupHook() function. So that the sporadic messages can be produced properly, "C" command, for the random number generation, "srand(nTime2)" is used where the nTime2 is the seed of the random number generator obtained by taking the network time of the FlexRay bus. The script for the random number generation is given below.

```
nTime2 = ffrd_api_get_mtick();
srand(nTime2);
```

Up to this point, various registers for the operation of the program have been set and initialized, the reload timers, which will supply time tick for the main tasks of the FlexRay node, are loaded and started and the hardware to perform all these tasks is initialized for the proper operation. Therefore, after all those preparations have been done, the idle task of the FlexRay project, "runTask()", is called to start the operation. The idle task is essentially an infinite loop in which the standing status of the FlexRay network is transferred to a PC via the RS-232 interface. In the normal flow of the project, this loop is interrupted by one of the reload timers so that the task corresponding to that reload timer can be performed. Once the task is successfully performed after the issue of the interrupt, the code returns to the infinite loop of idle task, performing the "printFlexRayStatus()" and waiting for another interrupt to jump into. As a consequence, throughout the operation of the project, the program can never complete the idle task, "runTask()". If, somehow, the code achieves to get rid of the idle task trap, this means that the FlexRay project does not function properly and something goes wrong. So, to prevent the program from ending in an undesirable state, after the idle task, a function named, ttShutdownHook(0), is added which shuts the FlexRay driver down and ends the operation of the code.

Next, we look into the details of the tasks performed in "MAIN.c". Mainly, in the file, MAIN.c, the production of the periodic and the sporadic messages is realized. As mentioned previously, the structure of the code is built on the generation of the interrupts for distinct tasks in a timely manner and the time ticks for the tasks are supplied by the distinct "Reload Timers". Next we explain the details of the "Reload timers" subroutine which performs the FlexRay tasks.

To begin with, when the interrupt for the static slot message generation is issued, the necessary arrangements for the production of the static segment message is made in the subroutine of the interrupt named IsrReloadTimer1(). "Reload timer1" is programmed to issue interrupt with the period of 5 ms which is the cycle length of the Gateway experiment. The period of the "reload timer" may be altered in different experiments, according to the cycle length of the network or the specific message generation requirement of the experiment.

The goal of the periodic message generation is to generate the entire set of the static segment messages in the very beginning of the FlexRay cycle. There are three related issues. First of all, when a node is turned on, it synchronizes itself to the already running FlexRay network and begins generation FlexRay messages with the period of 5 ms. However, it is very likely, when it is synchronized to the network, that the network is at an arbitrary time of the cycle which is beyond the time where the FlexRay messages are meant to be generated. Second, even the static segment message generation is in the very beginning of the cycle, still, after some time, message generation begins to be realized in an undesired portion of the cycle due to the drift of the local clock of the node from the global time tick of the network. Finally, the generation time of the "reload timer" interrupt with 5 ms period may also drift with respect to the global network time which results in the same undesired consequence with the previous two situations.

To overcome the above problems and stabilize the message generation to the beginning portion of the cycle time the following procedure is followed [23]. In the interrupt subroutine, the network time of the FlexRay network is obtained in macroticks via the function "ffrd_api_get_mtick()". This is a utility function supplied by the FlexRay Software Driver in its time services. Therefore, to be able to use this function, "ffrd_api_time_service.h" header file must be included at the beginning of the corresponding file. "ffrd_api_get_mtick()" gives the current macrotick value of the network in 16-bits. The aim is to issue the "reload timer" interrupt, so that it always falls within the macrotick values from 0 to TASK_OFFSET where the TASK_OFFSET is the user defined macrotick value defining the limit of the "beginning of the cycle" term quantitatively. As observed in APPENDIX A, the TASK_OFFSET is defined to be 100 for the Gateway experiment which means that the timer interrupt for the periodic message generation is to be realized within the first 100 macroticks of the FlexRay cycle. Therefore, in the "Reload Timer1" interrupt subroutine, the network time obtained by "ffrd_api_get_mtick()" is compared with the "TASK_ OFFSET" so as to determine whether the interrupt generation falls in the limits. If not, the "Reload Timer" is loaded with a reload value which corresponds to a smaller period than 5 ms so as to adjust the interrupt issuing time. Unless the "reload timer" is loaded with a specific reload value, "reload timer" continues to issue interrupt according to the last reload value. Since the last value after the adjustment is the reload value which corresponds to a smaller period than 5 ms, this reload value causes the interrupts to be issued in an earlier network macrotick each time. As a consequence, another user defined variable TASK_OFFSET_MIN is defined. When the interrupt issuing time happens to be a smaller value than TASK_OFFSET_MIN, then the reload value corresponding to 5ms period is begun to be used again which prevents the further interrupts from being issued earlier. Therefore, the "reload timer" interrupt issuing time for the generation of the messages fluctuate between the macrotick values TASK_OFFSET and the TASK_OFFSET_MIN which stabilizes the message generation to happen, fairly, in the beginning of the FlexRay cycle. The script achieving this stabilization task is given below.

```
__interrupt void IsrReloadTimer1(void)        // 5ms
{
     /* get FlexRay ClusterTime */
     nTime = ffrd_api_get_mtick();
     /* correct host offset */
```

```
            if (nTime >= TASK_OFFSET)
            {
               TMRLR1 = 2490u; //reload value 4.98 ms
            }
            if (nTime <= TASK_OFFSET_MIN)
             {
             TMRLR1 = 2500u; //reload value 4.98 ms
             }

         TMCSR1_UF = 0; /*Reset Timer, clear interrup flag*/
         ......
}
```

It is important here to notice that the case only in one direction where the interrupt issued after the "TASK_OFFSET" is handled in the code. The reason for this is that the "reload timer" of the node always retards with respect to the global network time. So, the time drift happens only in one direction and it is not required to consider the situation where the interrupt is issued earlier than 5 ms.

Having the periodic FlexRay messages generated at a known time, particularly, in the beginning of the cycle is important. By doing so, more controllable and robust results are obtained. For example, by generating all messages in the beginning of the cycle, it is guaranteed that the messages are sent through FlexRay in that cycle unless any specific constraints because of the repetition and the offset values apply. In other words, if the messages were generated at arbitrary times in the cycle, it would be possible for a message to be created at a time after its sending slot time which would cause the message to wait for an about the cycle time unnecessarily. To conclude, if the stabilization of the generation of the messages was not done, it would get harder to make conclusions about the behavior of the FlexRay network or figure out the reasons of the possible errors by examining, the jitter and the end-to-end delay values of the messages since the arrival of the messages is chaotic or, at least, drifts in time uncontrollably.

Moreover, the kick off for all of the FlexRay messages must be realized beginning from the very same FlexRay cycle so that the more controlled results can be obtained from the hardware experiment. This common cycle where all FlexRay signals are begun to be generated is selected to be the 0th FlexRay cycle. However, unless some precautions are taken, the nodes shall not start the message generation beginning from the 0th cycle. Even, all nodes will begin to generate the messages in different FlexRay cycles since they will be synchronized to the FlexRay network in different cycles with respect to each other. To solve this problem, each node finds out the current FlexRay cycle by utilizing the FlexRay Driver routine, "ffrd_api_get_cycle()". Unless the FlexRay cycle they are in is not the 0th cycle, they do not generate their initial message. Moreover, not any of the nodes begins to send FlexRay messages once they are in 0th cycle after powered on, but, rather, all of the nodes wait for the 0th cycle to have passed 150 times before sending the initial static segment FlexRay message. The first reason for this wait time is that the FlexRay node can not immediately be synchronized to the FlexRay network and when a node is not synchronized then the output of the "ffrd_api_get_cycle()" routine is always zero. So, before the synchronization is established, the node finds itself to be in 0th cycle numerous times although in fact it is not. Besides, although one of the nodes may have been synchronized and ready to generate messages, it should still wait some time for all the other nodes to be synchronized. As a result, to compensate for these effects, each node starts their message generation after they have counted the 0th FlexRay cycle 150 times. The script for this cycle management is given below.

```
if(start<=150)

     {
```

```
     task_Node1();

      cycle_no = ffrd_api_get_cycle();

         if (cycle_no == 0)

         {

       start++;

         }

     }


    if(start > 150)

    {

      ........

    }
```

At this point of the interrupt subroutine, IsrReloadTimer1(), for the Reload Timer1, it is guaranteed that the first message is generated in the 0th cycle of the FlexRay network and all the messages are produced in the very beginning of each cycle by compensating for the drifts in the local clock. Once this infrastructure has been provided, the messages are generated according to their periods. So as to generate a message in the subroutine, a flag for the message is set to be processed in the function task_Nodex() which is the function where the FlexRay task is handled. After the flags for the messages to be produced in that specific cycle have been set, the FlexRay task, task_Nodex(), is called in the interrupt subroutine so that the messages got prepared in the correct FlexRay buffer.

Another task handled in the MAIN.c is the clearance of the hardware watchdog. Hardware watchdog of the project is cleared in every 500ms the time tick of which is provided by reload timer0. By clearing the hardware watchdog periodically, the node is prevented from being reset by the watchdog after certain amount of time.

Generation of the sporadic messages is performed in four different interrupt subroutines. The tasks held in all of four subroutines are the same. When an interrupt for one of the sporadic messages is issued, it means that the time to generate the message has come. As opposed to the static message generation case, the payload for the message is immediately put into the FlexRay buffers. It should be remembered that in the static segment message generation, the flags for the messages to be produced were set and the necessary data were put into the buffer by a FlexRay task function, task_Nodex(), which is called periodically at the end of the IsrReloadTimer1() interrupt subroutine. As a result, the messages were achieved to be produced periodically, which is the goal of the static segment message generation. However, the goal of the sporadic message generation is to produce all the sporadic messages with random interarrival times. Therefore, it is not possible to assign a flag to each sporadic message and fill the corresponding FlexRay buffers with the necessary data in a single task according to the flags. As a consequence of this, for each sporadic message, a specific reload timer is assigned to determine the sporadic generation time and the message data is immediately transferred to the corresponding buffer in the very interrupt subroutine of the message. Having had the sporadic message generated, load value of the reload timer is determined for the next generation of the interrupt. Unless loaded with a specific value, reload timers are programmed to be loaded with the last reload value that they operated. So, not reloading the reload timer means generating the reload interrupt with a period of its reload value which is not desired in sporadic message generation. In order to kill this periodicity in the sporadic message generation, a random number is produced via standard "C" function rand(). As mentioned previously, while the main code is being prepared for the proper operation, a seed for the random number generation process is produced via "srand(nTime2)" where "nTime2" is the network macrotick time of the moment when the command is called. The random number generated by the "rand()" function is used as the new reload value of the reload timer which determines the time after which the timer issues the interrupt. Since after each time the interrupt is generated, reload value is determined anew randomly, sporadic message generation is achieved. What is important here is that the random numbers as the result of the "rand()" command that are below some certain value are not allowed to be used as the new reload value of the reload timer. This is because, besides, the messages are to be generated with random interarrival times, the interarrival time of the generated messages must, also, be greater than some certain value. Therefore, by ignoring the values smaller than the limit and iterating the random number generation until a value satisfying the requirement has been obtained, the interarrival time between the consecutive message generation is guaranteed to be greater than some certain value. The minimum interarrival time is not unique and the value changes depending on the sporadic message. The script for the random number generation is provided below.

```
do {
r_number = rand();
}while (r_number <= 5000);//min. interarrival limit
```

As a final remark, the message handling through the FlexRay buffers is performed in the dynamic segment message generation process. To elaborate the FlexRay buffers in the MAIN.c, buffer handling and reading/writing services of the FlexRay Software Driver must be utilized. This means that the corresponding header files must also be included in the MAIN.c as well as TTask.c.

### 5.2.2 Tasks Executed In TTask.c

In the previous section, FlexRay message generation for both the periodic messages in the Static Segment and the sporadic messages in the Dynamic Segment is explained. As mentioned above, the periodicity of the static segment messages is provided in "MAIN.c" and those messages are filled into the buffers via the FlexRay task in the "TTask.c" to be sent through the FlexRay bus. This FlexRay task, namely, "task_Nodex()", will be explained in this section. Apart from the function,"task_Nodex()", there, also, exists other functions named "ttStartupHook()", "ttErrorHook()" and "ttShutdownHook()".FlexRay driver and Communication Controller are initialized via "ttStartupHook()". As its name implies, "ttShutdownHook()" stops the operation of the FlexRay driver. This function is called after the idle task, runTask(), has been run. Since the runTask() is, in fact, an infinite loop, if program tried to execute the next command after runTask(), this means that something has gone wrong and the FlexRay operation should be terminated so as not to give way to the unstable situations.

The most important task existing in TTask.c is the task named "task_Nodex()" where the message exchange through the FlexRay buffers is handled. This function is originally called in the subroutine named "IsrReloadTimer1()" where the static segment messages are generated. As mentioned previously, in this interrupt subroutine, the flags for the messages whose periods have arrived in that specific cycle are set and task_Nodex() is called. Therefore, in every cycle, which is 5 ms for the Gateway experiment, task_Nodex() is called periodically and the messages whose flags have been set are put into the corresponding FlexRay buffers. Since throughout the tasks handled in the "TTask.c", details of which will be covered shortly, Reception (Rx) Services, Transmission (Tx) Services, Status Information Services, Control Services, Time Services and Initialization Services of the FlexRay Communication Controller Driver are used, the following header files must, additionally, be included in the beginning of the TTask.c.

```
    #include <ffrd_api_init_chi.h>
    #include <ffrd_api_control_service.h>
    #include <ffrd_api_tx_handler.h>
    #include <ffrd_api_rx_handler.h>
    #include <ffrd_api_status_service.h>
    #include "ffrd_api_time_service.h"
```

Every time task_Nodex() is called, first of all, status of the Communication Controller is checked and if it is not working in the normal way and still not halted, then the Communication Controller is forced to be started through the "COLDSTART" method.

After the status check, all the messages to be sent in that cycle are filled into the FlexRay buffers one by one via the script equivalent to the following.

```
if (tx8_flag){

 buffer8.Port = tx8_data;

 buffer8.period = tx8_period;

 buffer8.m_counter = ffrd_api_get_mtick();

 buffer8.c_counter = ffrd_api_get_cycle();

 statusTx8=ffrd_api_tx_handler_buffer((FFRD_UINT32)
 &buffer8, 10, 3, FFRD_CHANNEL_A_B);

 tx8_flag = 0;

}
```

Actually, the above code is just an example of the scripts doing the same job for the other static segment messages. For instance, that is the code to fill the necessary FlexRay buffer for the periodic message whose message number is 8. It should be noted that when the message number 8 is generated in the MAIN.c part, then the flag for that message is set and after all flags have been set, the function task_Nodex() is called. The above script is the place where this flag is processed in the task_Nodex() routine. In task_Nodex(), all the flags for all the messages are checked individually to determine whether the messages will be sent in that cycle or not. If the flag is set then the corresponding buffer, which is buffer8 in the above example, is filled with the payload to be sent through FlexRay bus. As seen from the above code, for the Gateway experiment, the data consists of the payload, period, cycle number and macrotick time of the network. All these elements composing the whole data to be written into the buffer are the elements of the C structure which is created for all of the buffers. The buffer structure designed for the Gateway experiment is given below.

```
typedef struct{

     uint16_t Port;

     FFRD_UINT16 c_counter;

     FFRD_UINT16 m_counter;

     uint16_t period;

     uint16_t empty[1];

}data_content;
```

Although 8 bytes were enough for the data, the structure was created as 10 bytes long since the length of the FlexRay frame to be sent in the Gateway experiment was 10 bytes long.

Note that in the C structure of the buffers that in the Fujitsu CPU, MB91F465XA, the memory is reached as 16-bit or 32-bit. 8 bits of data can not be read/written from/to the CPU memory. Therefore, the members in the C structure with the length of 8 bit will be read/written from/to the memory as 16 bits. This may cause to the loss of data in the situation where the structure includes members with length of 8 bits and no unused structure members are included. The situation can be illustrated by the following example C structure.

```
typedef struct{

   uint8_t mem1;

   uint8_t mem2;

   uint16_t mem3;

   uint16_t mem4;
```

```
      }data_content;
```

In the above structure, structure length is 6 bytes. When 6 bytes of data beginning from the first member of the structure is to be written to the memory of the CPU, the last member, mem4, of the structure can not be written to the memory since the first two members of the structure occupy 4 bytes of memory instead of 2 bytes which results in the loss of the last member of the structure. Because of this reason, to track the data length of the structure easily, the data type of the variable "c_counter" was chosen as "FFRD_UINT16" instead of FFRD_UINT8 though the maximum value for the cycle number is 255.

Once the members of the buffer structure are properly assigned, the data is copied to Communication Controller hardware by means of the ffrd_api_tx_handler_buffer() routine. This function is the member of the Transmission (Tx) Service in the FlexRay Software Diver. What ffrd_api_tx_handler_buffer() function actually does can, more easily, be explained by going over the above example for the message number 8. This message handling function looks like the following with its arguments included.

```
statusTx8=ffrd_api_tx_handler_buffer((FFRD_UINT32)
&buffer8, 10, 3, FFRD_CHANNEL_A_B);
```

The first argument of the function signifies the address of the buffer8. The second argument means the data length whereas the third argument indicates the Communication Controller buffer number. So, in the light of these definitions, the function ffrd_api_tx_handler_buffer() copies 10 bytes of data from the beginning of buffer8 to the 3rd buffer of the Communication Controller. Last argument of the function means that the data will be transmitted from both of the FlexRay channels. Sending the data from only Channel A or only Channel B are the other remaining options for the last argument of the function. After this ffrd_api_tx_handler_buffer() function has been handled, the data is put into the Communication Controller buffer. From this point on, the job of sending the message through FlexRay bus belongs to the Communication Controller and the

FlexRay transceiver. According to the FlexRay IDs (FID) of its buffers, Communication Controller waits for the slot time to come for each of the buffers and sends the message of the buffer whose time has come to the transceiver so that the message is put onto the wire on time. As a result of this mechanism, the TDMA structure of the FlexRay network is successfully achieved.

The buffer where to store the static segment messages is determined according to the network configuration. As mentioned before, the FlexRay network is configured via the FlexConfig™ program which provides a graphic user interface to facilitate the network configuration without any error. The FlexConfig™ outputs a *.chi file for each node in the network describing the scheduling of the node and the register settings of the Communication Controller. The scheduling related part of the *.chi file of the FlexConfig™ for one FID is given below as an example.

```
/*  Tx  Buffer  5  (Frame  Id:  19,  Payload  length  5,
FlexRayAB, Base 0, Rep. 2)*/

     WAIT_TILL_CLEARED32(0x80000000, 0x00000514);

     WRITE32(0x17020013, 0x00000500); /* WRHS1 */

     WRITE32(0x000500be, 0x00000504); /* WRHS2 */

     WRITE32(0x0000007c, 0x00000508); /* WRHS3 */

     WRITE32(0x00000001, 0x00000510); /* IBCM */

     WRITE32(0x00000005, 0x00000514); /* IBCR */
```

As seen from the example, the Communication Controller is configured by the FlexConfig™ to store the message to be sent in the 19th static slot in its 5th buffer. Similarly, the information about which message belongs to which buffer is included in the *.chi file for all of the messages one by one. Therefore, the developer programming the task_Nodex() routine has to use the *.chi file corresponding to the very node he is working on and assign the buffer argument of the ffrd_api_tx_handler_buffer() routine accordingly. Doing the job of mapping of the buffer number and the static slot via the *.chi file by hand is cumbersome and prone to errors. To eliminate these problems and realize the buffer number and the static slot mapping automatically, a code parsing program is composed and the structure of the project is changed accordingly. The details of this program are discussed in section 5.5.

## 5.3 CAN PROJECT DETAILS

In the Gateway experiment, there exist three CAN projects of which structures are very similar to each other as shown in Figure 5-1. Without loss of generality, the tasks held on the CAN projects will be discussed by projecting the discussion to one of the projects namely "CAN1.pjt".

As in the FlexRay project, the tasks performed in the CAN projects can, also, be combined in two different files. These are MAIN.c and CAN.c. Therefore, these files will be explained in the coming two sections to figure out, more detailed, what is being done in the CAN projects.

### 5.3.1 Tasks Executed In MAIN.c

In MAIN.c, the time ticks are provided so that the CAN messages are generated according to the desired periodicity. The structure of MAIN.c in the CAN project is very similar to that of the FlexRay project. Therefore, instead of discussing all the details in the file, the tasks held in MAIN.c will be tried to be explained through the differences/similarities from/to the MAIN.c of the FlexRay project.

First of all, it should be noted that the CAN nodes are also connected to the FlexRay network in addition to the CAN bus to have synchronization with the other nodes and get the network time to be used for measurements. As a result, all the necessary arrangements, settings and the initialization which have been done for the FlexRay bus in the FlexRay nodes should also be done in the CAN nodes except the message receive and transmission functions.

The flow of the code in main() for CAN project is essentially the same as that of the FlexRay project except for the two differences. The first difference is in the InitCPUExtraRegs() function where some registers used throughout the project are initialized. In CAN project additionally, InitCANCtrl0() function is added in InitCPUExtraRegs(). Like, the FlexRay driver is initialized via ttStartupHook(), CAN engine in the CPU also must be initialized to be ready for a proper operation. InitCANCtrl0() is the function which initializes one of the 6 CAN controllers existing in the CPU and the details for this initialization function will be given under the next chapter.

The other difference is that there are 2 reload timers in CAN project instead of 6 reload timers which is the case in the FlexRay project. One of the reload timers is used to clear the hardware watchdog periodically and the other reload timer is used to provide time tick for the periodic CAN message production. The four reload timers, which account for the difference between the two projects, are each used for the sporadic message generation in the FlexRay project. Since there exist no sporadic messages in the CAN project, no additional reload timers are needed.

Under IsrReloadTimer3() subroutine which is the interrupt subroutine of the reload timer responsible for the periodic message generation, essentially the same tasks are performed as in the static slot message generation of the FlexRay project. The CAN messages are generated in the very beginning of each FlexRay cycle and the first CAN message generation takes place at the 0th FlexRay cycle. These tasks are achieved as they are achieved in the FlexRay project. In the end of the subroutine CAN0_SendMessage function which is responsible for sending the CAN messages is called. CAN0_SendMessage routine fills the applicable registers of the CAN engine in the CPU as required and the CAN engine sends the message to the CAN transceiver so that the message is physically put on the CAN bus. The details of the CAN0_SendMessage function will be presented in the following section.

## 5.3.2 Tasks Executed In CAN.c

In the file CAN.c 5 different tasks are executed. These are InitCANCtrl0, CAN0_STATUS_ISR_Handler, CAN0_ReadMessageBuffer, CAN0_SendMessage and CAN0_ISR.

In InitCANCtrl0 routine, the registers of the 0th CAN Controller of the CPU are initialized so that the CPU can process the incoming and the outgoing messages in a desired way. The CPU, MB91465XA, supports 6 different CAN Controller in its peripherals. However, only two of them are interfaced to the CAN transceivers and the rest are used as general purpose in the Fujitsu evaluation board, SK-91465X-100MPC. Used CAN controllers in the SK-91465X-100MPC are CAN0 and CAN4 and InitCANCtrl0 routine initializes the former one. Since the CAN0 transceiver is connected to the 23rd port of MB91465XA, applicable bits of the PFR23 (Port Function Register 23) must be set for CAN receive and CAN transmit functions. The following arrangements are made for this purpose.

```
        PFR23_D0 = 1;                 /* RX */
        PFR23_D1 = 1;                 /* TX */
```

Configuration of the CAN0 controller is also done in InitCANCtrl0. The CAN baud rate determination, deciding on which type of interrupts to be issued and making the buffer arrangements according to the application are some examples to the configuration options of the CAN controller. The CAN controller must be disabled so that some of the configurations may be done and after the configurations have been set, the CAN controller is enabled again. The script for this operation and the setting of the CAN bit rate is given below.

```
CTRLR0_CCE = 1;              /* enable cfg change */
BTR0 = BTR_16M_500k_16_68_3; /* BTR config 500 kBaud */
CTRLR0_CCE = 0;              /* disable cfg change */
```

```
CTRLR0_EIE = 1;              /* enable error interrupt */
CTRLR0_SIE = 1;      /* enable status change interrupt */
CTRLR0_IE = 1;  /*enable interrupt generation for CAN0*/
CTRLR0_Init = 0;   /* complete init, start CAN0 */
```

Message buffers in the CAN Controller must also be configured in the InitCANCtrl0 routine. There exist 32 message buffers in each CAN controller of the CPU. Each of the buffers can be set to be used either as transmit buffer or receive buffer according to the application. All the necessary settings to properly arrange the message buffers are done in InitCANCtrl0. Defining the ID to receive/send, deciding to use the extended ID mode or not, managing the interrupt settings are some of the examples of the buffer configuration which are covered in InitCANCtrl0. One last thing to be taken into account when configuring the message buffers of the CAN Controller of the MB91465XA is that the CAN buffers greater than 16 in number can not be configured to be used as receive buffer whereas all the buffers from 1 to 32 can be used as transmit buffer. In other words, it must be avoided to configure the buffers from 17 to 32 as CAN receive buffer.

In the operating CAN scenario, CPU is informed when the CAN messages are received via interrupts instead of polling the CAN controller message buffers. Because of this reason, the routines, CAN0_ReadMessageBuffer, CAN0_ISR and CAN0_STATUS_ISR_Handler functions in cooperation with each other. The latter two of them are responsible for the interrupt handling. These functions determine the source of the interrupt, clear the necessary flags and call the CAN0_ReadMessageBuffer function if the interrupt source is one of the receive message buffers. CAN0_ReadMessageBuffer function is called in the interrupt subroutine by the argument which is the number of the buffer causing the interrupt. Therefore what should be done first in the CAN0_ReadMessageBuffer routine is to transfer the payload and the other necessary data from the message buffer to the interface registers of the CPU so that the CPU can easily process the incoming data. This is done by the following code.

```
    IF1CREQ0 = buffer; // buffer number to be transferred
```

This command transfers all of the content of the buffer which is not masked, to the interface registers of the CPU. From this point on the CPU is able to read and manipulate, the ID, the payload or the any other information about the message transfer of the incoming message.

The last task held in CAN.c is sending the CAN messages via CAN0_SendMessage function. This function is called in the reload timer subroutine where the CAN messages are generated. In the interrupt subroutine, CAN0_SendMessage function is called with 5 arguments. Two of the arguments are the 4-byte data pieces which form the 8 byte payload of the CAN message. The others are the ID of the message, data length in bytes and the buffer number from which the message is to be transmitted. While storing a message to the message buffer, the opposite procedure is followed when compared to reading the message from the buffer. First of all, the CPU transfers the data to the applicable interface registers as required with the arguments of the function and then executes the following command.

```
    IF1CREQ0 = buffer; // buffer number to be transferred
```

This time the above command stores all the content of the interface registers which are not masked, to the specified message buffer so that it is put on the CAN bus via CAN protocol engine. Since in the Gateway experiment, message trip time throughout the network is measured so as to evaluate the performance metrics of the Gateway, timestamp is put on the outgoing CAN message. This timestamp obtained via the time service functions of the FlexRay Software driver. Therefore, so as to use the applicable functions of the time service routines of the FlexRay driver, "ffrd_api_time_service.h" is included in CAN.c.  After the timestamps have been obtained, they are written into the CPU interface registers which store the payload of the outgoing message. Then, the content of the interface registers is transferred to the message buffer as described above and the CAN message with the timestamp on it, starts its journey on the CAN bus. The code for the timestamps is given below.

```
        nCycle = ffrd_api_get_cycle();
        mtick = ffrd_api_get_mtick();
        IF1DTA120 = nCycle + (mtick<<8) + (data1<<24);
```

## 5.4 GATEWAY PROJECT DETAILS

Gateway project also has very big similarities with the CAN and the FlexRay project since Gateway is somehow the combination of both projects. In the coming sections, only, the considerations and the approaches that differ from the preceding projects will be discussed and the things which are repeated in the Gateway project will simply be omitted. The tasks held in the Gateway will be discussed under 3 chapters; Main tasks, FlexRay tasks and CAN tasks.

### 5.4.1 Tasks Executed In MAIN.c

The tasks performed in MAIN.c are very much the same with that of, CAN project and the FlexRay project. The structure of the code is the same. First of all, the registers are configured and set, second reload timers are started and finally the idle task of the project begins to run waiting the interrupts to be issued so that the main tasks of the project can be performed.

It should be noted that the CAN Controller of the CPU must be initialized also for the Gateway node. This is because the Gateway node performs the tasks of both FlexRay and CAN nodes. Therefore, the initialization routine for the CAN Controller, InitCANCtrl0(), is called in the InitCPUExtraRegs() function. In addition to this, CAN interrupt vector and the CAN interrupt priority must also be defined in vector.c by the user so that the Gateway node can receive CAN messages through interrupt mechanism.

One of the reload timers is used to call the FlexRay task in every 5 ms. Since the Gateway has no message generation requirement, in the interrupt subroutine, no

FlexRay message is prepared. Instead of message generation, this reload timer is used for message polling of the incoming FlexRay messages. The incoming messages are polled with the period of 5 ms in MAIN.c and if some messages are received, they are processed and routed through the CAN network in the TTask.c. One more thing to add about this reload timer subroutine and the polling mechanism is that the polling time of the node is adjusted to occur just in the middle time of the whole cycle, which is 2.5 ms. Adjusting of the polling time just in the middle of the cycle is specific to the Gateway experiment. As mentioned in the previous chapters, it is possible and desirable to generate the reload timer interrupts in the very beginning of the FlexRay cycle for the FlexRay and the CAN projects. The mechanism to stabilize the interrupt issuing time to the beginning of the FlexRay cycle is discussed in section 5.2.1. Moreover, the reasons for adjusting the message generation time this way is also explained clearly. Because of the parallel reasons but with a different point of view, the FlexRay messages in the Gateway must be polled in the middle of the FlexRay cycle. The reason for this is that, as it is mentioned, the FlexRay messages are being generated in the beginning of the FlexRay cycle. Therefore, the messages do not miss their sending time slot if they are to be sent in that very cycle. On the other hand, if the receiving node, which is the Gateway, for those messages is programmed to poll the incoming buffers in the very beginning of the cycle, naturally, the receiving node will see that either the buffers are empty or the messages sitting there belong to the messages of the previous cycle. In other words, the Gateway will receive nearly all of the messages with one whole cycle delay. Because the messages were not generated yet and they could only be sent after their defined slot time has come. Therefore, so as to avoid this phase delay between the generated Flexray messages and the received ones, the Gateway polls the receive buffers if there is a new message arrived just in the middle of the cycle. The middle of the FlexRay cycle is an important time spot, because, according to the scheduling the FlexRay network that the Gateway is connected to, it is guaranteed that all the generated messages are achieved to be transmitted to the other node until the mids of the FlexRay cycle. The stabilization of the reload timer interrupt to the mids of the FlexRay is essentially the same thing as the stabilization of the messages to the very beginning of the cycle which is explained in section 5.2.1 in details. The only difference with that case is that the TASK_OFFSET value is defined to be 2500 which signifies the 2.5 ms time difference instead of using the previous value which was 100 and indicated the beginning of the cycle.

### 5.4.2 Tasks Executed In TTask.c

As mentioned previously, task_Nodex routine in TTask.c is used for polling the FlexRay receive buffers to see if new messages have come or not. The function task_Nodex is not used to send FlexRay messages through the FlexRay network though the Gateway has to perform this task also. This sending task is held in CAN.c and the details of the task will be explained in the following chapter.

Before performing the receive task, in the beginning of the function, task_Nodex checks the status of the Communication Controller and, if it is needed to, follow the procedure to start the Controller as described in section 5.2.2. After the status check and the necessary arrangements, the FlexRay task does check all the receive buffers one by one to see if there are new received messages. The function named ffrd_api_rx_handler_buffer(), whose structure is very similar to the message transmit handling function, is used to receive messages from FlexRay bus. While the routine ffrd_api_rx_handler_buffer must be called twice separately for each redundant channel, calling the message transmitting routine only once is enough. Because there exist two distinct channels, Channel A and Channel B, to receive the FlexRay messages and although, the sending side of the message has send the message through both channels it is possible that the message might be received at only one channel due to the hardware problems, software bug or electromagnetic interference. On the other hand, to send a message through redundant channels, filling only one buffer is enough. The Communication Controller copies the content of this transmitting buffer when to send the data to the FlexRay transceivers of both Channels. Parallel to this structure, when the FlexRay network is scheduled by defining the receiving and the transmitting time slots via the FlexConfig™, the program outputs a *.chi file where only a single buffer is allocated for each transmitted signal and two buffers for each received signal. An example to this situation from a *.chi file is given below.

```
/* Tx Buffer 5 (Frame Id: 34, Payload 5, FlexRayAB*/
WAIT_TILL_CLEARED32(0x80000000, 0x00000514);
WRITE32(0x17030022, 0x00000500); /* WRHS1 */
..........................................


/* Rx Buffer 8 (Frame Id: 68, Payload 4, FlexRayA */
..........................................


/* Rx Buffer 9 (Frame Id: 68, Payload 4, FlexRayB*/
..........................................
```

The script to check the receive buffers to determine if some messages have come is given below.

```
statusRx1=ffrd_api_rx_handler_buffer((FFRD_UINT32)&sRx1,
&header_rx1,        10,        0,         FFRD_CHANNEL_A,
ffrd_api_new_rx_data_buffer(0));

statusRx2=ffrd_api_rx_handler_buffer((FFRD_UINT32)&sRx1,
&header_rx1,        10,        1,         FFRD_CHANNEL_B,
ffrd_api_new_rx_data_buffer(1));
```

The first 2 arguments of the total 5 arguments of the function ffrd_api_rx_handler_buffer() are the outputs and the rest of them are the inputs of the routine. First argument of the function is the payload of the received message and the second argument is the header. The input arguments of the ffrd_api_rx_handler_buffer() function which are the 3rd, 4th and the 5th arguments stand for the data length in bytes, the buffer number and the receiving channel, respectively. Therefore, to combine it all, ffrd_api_rx_handler_buffer() routine reads the content of the receive buffer, which is given in the 4th argument, for the specified channel and writes the payload and the header of the received message to the addresses defined by the 1st and the 2nd arguments, respectively. The length of the data that will be written beginning with the memory address specified in the first argument is given in the 3rd argument. As it is in the message transmit case, in ffrd_api_rx_handler_buffer() function also, the user must enter carefully the buffer numbers that will be read by the function, by checking it from the corresponding *.chi file.

In task_Nodex(task), the message receiving task for the periodic messages and that of the sporadic messages are handled separately although it does not have to. Because, since the structure of dynamic segment messages and the static segment messages are different from each other, it is more convenient to process those messages separately.

It is considered that a new message has arrived to the node if the return value of the ffrd_api_rx_handler_buffer() function is FFRD_OKAY for at least one of the redundant channels. The function ffrd_api_rx_handler_buffer() does not return FFRD_OKAY value again after it has read a buffer until a new message arrives to the receive buffer. When it is determined, as just explained, that a new message has arrived, the incoming message is processed and forwarded through the CAN bus according to the requirements of the Gateway experiment. The details of the task_Nodex() function for the Gateway project is given in APPENDIX C.

### 5.4.3 Tasks Executed In CAN.c

The tasks held in CAN.c is very similar those of the CAN project which is discussed previously. Also in the Gateway project, the same tasks namely, InitCANCtrl0, CAN0_STATUS_ISR_Handler, CAN0_ReadMessageBuffer, CAN0_SendMessage and CAN0_ISR are held.  Only the applications coded for the tasks differ.

In the beginning of the CAN.c the routine named InitCANCtrl0 is implemented. In InitCANCtrl0, CAN Network and the CAN Controller is configured. The tasks such as configuring the message buffers and determining the CAN baud rate are held in the function as discussed under CAN project. It should be noted that although the InitCANCtrl0 routine is defined and implemented in CAN.c, the function is called in MAIN.c during the configuration of the necessary registers of the CPU. In addition, so as to be able to receive messages via CAN bus, the CAN interrupt vector and the CAN interrupt priority must also be defined in vector.c

The CAN0_SendMessage function is an other task held in the CAN.c the structure of which is also very much the same with that of the CAN project. The function CAN0_SendMessage is called from TTask.c when a FlexRay message is received. After a FlexRay message has been received, the Gateway node processes the incoming message according to the application and sends the message through CAN bus by calling the CAN0_SendMessage function. Inside the function, timestamp, which signifies the leaving time of the message from the Gateway, is added in the payload so that the performance metrics of the network can easily be measured and then the message is put into the buffers to be transmitted through the CAN bus.

The other important task held in CAN.c is CAN0_ReadMessageBuffer routine. The CAN0_ReadMessageBuffer function works in concordance with the interrupt related routines, namely CAN0_STATUS_ISR_Handler and CAN0_ISR. When a new CAN message is received, an interrupt is issued by the CAN Controller indicating the message buffer number in which the incoming message is stored. CAN0_ReadMessageBuffer is called in the interrupt subroutine with an argument which stands for the buffer to read.

In CAN0_ReadMessageBuffer routine first of all the network time is obtained by using the FlexRay Driver API functions. The obtained values signify the receive time of the CAN messages in the Gateway. After the timestamp has been obtained, the data in the message buffer is transferred to the interface registers so that the CPU easily processes the incoming data. Then the payload and the ID of the received message is read and processed. Since the incoming message contains the timestamp indicating the transmitting time of the message, macrotick value and the cycle number are separately extracted to be sent through FlexRay. The script extracting the timestamp components from the payload and reading the ID from the corresponding register is given below.

```
can_send_cycle = 0x000000FF&IF1DTA120;
can_send_mtick = (0x00FFFF00&IF1DTA120)>>8;
ID = 0x1FFFFFFF&IF1ARB120;
```

The remaining code in the CAN0_ReadMessageBuffer routine forwards the received CAN messages to the FlexRay network according to the CAN-to-FlexRay mapping of the Gateway.

It should be noted that as opposed to the FlexRay message generation case in the FlexRay project, here, a-flag-for-each-message structure is not applied. Instead of setting a flag for each CAN to be sent through the FlexRay network and processing all the flags periodically and sending the messages whose flags are set, in CAN0_ReadMessageBuffer, the FlexRay buffers are filled with the corresponding CAN messages immediately. In other words, ffrd_api_tx_handler_buffer() function is directly called under CAN0_ReadMessageBuffer whenever needed. If the ffrd_api_tx_handler_buffer() routine was not immediately called and the CAN messages are put into the FlexRay buffers all together periodically by the call of the task_Nodex function, then the CAN messages would experience extra delay by waiting the period of the task_Nodex to come. Even as the worst case, it is possible that a CAN message is received just after the call of the task_Nodex. In this case the

CAN message will suffer in the Gateway for a time of a whole cycle which is 5 ms for the Gateway experiment. For the case of 5 ms task_Nodex polling period, the mean time the CAN messages will stay additionally in the Gateway is 2,5 ms which is a very large amount of time for the performance of the Gateway. However, the aim of the design of the Gateway is to keep the incoming messages in the Gateway with minimum of time and forward them to the destination network as soon as possible. Therefore, so as to improve the delay performance of the Gateway, ffrd_api_tx_handler_buffer() function is called and the FlexRay buffers are filled with the messages immediately in the CAN.c. Also, since the buffer handling, time and control services of the FlexRay Software driver are used throughout the coding in CAN.c, the driver header files listed below must be included in the beginning of the file.

```
#include "ffrd_api_time_service.h"
#include <ffrd_api_tx_handler.h>
#include <ffrd_api_status_service.h>
#include <ffrd_api_control_service.h>
```

## 5.5 OTHER DEVELOPMENT ACTIVITIES

As it is explained in the previous sections, while using the ffrd_api_tx_handler_buffer() and the ffrd_api_rx_handler_buffer() functions, the arguments of the functions, particularly the buffer number, have to be entered manually by the programmer. This is done by checking the mapping of the Gateway and the corresponding *.chi file of the Gateway where the buffer allocation for each FlexRay ID is given. Since all those checking of the files are done by hand, the task of filling the buffer number of the ffrd_api_tx_handler_buffer() and the ffrd_api_rx_handler_buffer() functions is cumbersome and carries the risk of making errors. Therefore, so as to do the mapping and filling the buffer numbers in the corresponding FlexRay functions automatically, a program is written in

Windows environment and the coding structure of the applicable files are a little bit changed.

According to the scenario which will carry out the above mentioned process automatically, first of all, a *.txt file named "config.txt" is created. In the "config.txt" file, CAN-to-FlexRay mapping, FlexRay-to-CAN mapping information and the name of the *.chi file created for the Gateway node by the FlexConfig™ is entered by the user. An example for the "config.txt" file is given in APPENDIX E. The "config.txt" file and the *.chi file created for the Gateway node are both included in the project directory of the program which is coded by the MS Visual Studio 6.0 in C language. This program parses both the config.txt file and the *.chi file of the Gateway and outputs two files named gateway.c and gateway.h where the FlexRay buffers are mapped with the FlexRay IDs and the CAN messages. The content of the files, "gateway.c" and "gateway.h", are given in APPENDIX F and APPENDIX G, respectively. As the file "gateway.c" is examined, it will be observed that all the FlexRay IDs and the CAN IDs, which are received and transmitted during the operation of the Gateway, are stored in the file as distinct arrays. The name of those arrays are CANtx[],CANrx[],FRtx[] and FRrx[]. In addition to this, the mapping information is kept in two different arrays, namely CAN2FR[] and FR2CAN[]. CAN2FR[] is the array that indicates the FlexRay IDs to which the incoming CAN signals will be mapped by storing the index number of the FRtx[]. Likely, the array FR2CAN[] indicates the CAN IDs to which the incoming FlexRay signals will be mapped by storing the index number of the CANtx[]. This concept can be illustrated by the following script which is taken from a real "gateway.c".

```
int CANtx[15] = {1, 0, 19, 18, 17, 13, 12, 11, 10, 9,
8, 5, 4, 3, 2};

int CANrx[15] = {123, 400};

int FRtx[2] = {6, 9};

int FRrx[15] = {7, 8, 16, 17, 19, 20, 26, 28, 31, 33,
36, 37, 41, 47, 48};
```

```
int CAN2FR[2] = {0, 1};

int FR2CAN[15] = {0, 2, 1, 3, 4, 10, 6, 7, 5, 9, 8,
14, 12, 13, 11};
```

When FRCAN[] is examined, for example, the 6th element of the array, which is FR2CAN[5], is found to be 10. This means that the 6th element of the FRrx[] array which stands for the incoming FlexRay signals is mapped to the 10th element of the CANtx[] array signifying the outgoing CAN messages. Therefore, FID 26 is mapped to CAN ID 12 according to the above script. The same logic applies to the CAN2FR array also.

Finally, in the gateway.c, the buffer allocations for the FlexRay IDs for both receive and the transmit buffers are in Tx_Buffer[],Rx_Buffer_A[] and Rx_Buffer_B[] arrays. In those arrays, buffer numbers to which the FlexRay IDs are mapped are stored. Similar to the previous logic, the number corresponding to an element of the, say, Tx_Buffer[], indicates that the buffer with that number is allocated to the FlexRay ID corresponding to that index number in the FRtx[] array. The same condition also applies to the Rx_Buffer_A[], Rx_Buffer_B[] and FRrx[] triplets. Note that two distinct buffers are allocated to each FlexRay ID. Therefore, there exist two different buffer mapping array in the "gateway.c" for the received signals in the FRrx[].

Once the files gateway.c and the gateway.h are created by the program coded in Windows environment as explained above, the next step is to adapt these files to the Gateway project and change, if needed, the structure of the Gateway code accordingly so that the FlexRay buffer numbers are automatically entered with less effort and without any error.

In fact, no drastic, changes are done in the structure of the Gateway coding. Including the gateway.h file in the corresponding Gateway file is enough to be able to use the arrays in the gateway.c. If the gateway.h file is observed in APPENDIX G, it will be seen that, what is done basically in the file is to render the arrays existing in the gateway.c to be used in external files by the C command "extern".

The reason that the gateway.h is created together with gateway.c is to obtain a more flexible structure and to be able to use the gateway.c in any files without any compilation error by simply including the gateway.h file in the corresponding file.

Therefore, in the light of the above information, to be able to use the files gateway.c and gateway.h, first of all, these files are included in the working directory of the Gateway project. After that, the gateway.h is included in all of the files where the ffrd_api_tx_handler_buffer() and the ffrd_api_rx_handler_buffer() functions are used. Finally, depending on the function used, one of Tx_Buffer[], Rx_Buffer_A[] or Rx_Buffer_B[] arrays is used in the argument of the function which stands for the buffer number. As a result of the usage of Tx_Buffer[], Rx_Buffer_A[] and Rx_Buffer_B[] arrays, the buffer numbers for the ffrd_api_tx_handler_buffer() and the ffrd_api_rx_handler_buffer() functions are automatically entered, the probability of making error in filling those arguments decreases and the cumbersome task of checking both the network scheduling and the corresponding *.chi file to match the buffer allocation one by one is avoided.

# CHAPTER 6

# EXPERIMENTAL PERFORMANCE ANALYSIS AND RESULTS

In this chapter, we present the experimental performance evaluation of our designed Gateway node as well as a performance evaluation of an interconnected FlexRay-CAN network. Before going over each experiment individually, first of all the performance metrics investigated in the experiments are explained. Next, general description of the experiments is presented, the issues common to all of the experiments are discussed and the time measuring mechanism is explained in details including the discussion about the effects of possible errors. Finally, each experiments held during the study is discussed under separate title. Our selected results and discussions are also presented in [34].

## 6.1 PERFORMANCE METRICS

In this work, we first investigate the *end-to-end delay* (worst case response time) and *jitter* of messages in individual FlexRay and CAN networks as well as end-to-end delay and jitter of the signals that go through the Gateway. In addition we consider the effect of different scheduling approaches on the timing of the messages and the efficiency of the bandwith use particularly for the FlexRay network.

The end-to-end delay and jitter of the signals that are transmitted through the Gateway depend on the performance of our Gateway design and implementation.

Hence, we investigate the *correctness of the protocol conversion*, *signal mapping* and *the processing delay* for the Gateway.

Next, we define these metrics in detail.

The *end-to-end delay* is defined as the time difference between the transmit time of the message and the receive time of the same signal. Here, the transmit time, theoretically, is defined as the time that the message is put on the wire for the FlexRay node and that the message is tried to be put on the wire for the first time for the CAN node. In both of the definitions, the time stamp for the transmit time must be put into the message just at the end of the respective transceivers of the CAN and the FlexRay node. Therefore, let us call this time as the hardware transmit time. However, the hardware transmit time is not measurable practically since the transceivers are not programmable and the timestamps are put into the payload via the software running on the CPU. As a result of this, the transmit time for both the CAN nodes and the FlexRay nodes is considered to be the time where the message is put into the sending buffers of the nodes in the CPU. Let us call this time as software transmit time.

Similarly the same logic applies to the receive time for both the CAN and the FlexRay nodes. So, the hardware receive time is the time where the incoming signal is received from the wire via the transceiver. The software receive time is the time where the CPU is, for the first time, able to process the message.

To combine it all, the end-to-end delay can be formulated as follows.

$$\Delta T_{end-to-end,hw} = t_{rx,hw} - t_{tx,hw} (hardware) \tag{6-1}$$

$$\Delta T_{end-to-end,sw} = t_{rx,sw} - t_{tx,sw} (software) \tag{6-2}$$

In the experiments, end-to-end delay is calculated by using the formula for the software end-to-end delay although this value is greater than the actual delay experienced by the messages. The difference between the software end-to-end delay and the hardware end-to-end delay is discussed in details in the chapters where the experiments are individually handled by displaying the order of the deviation quantitatively and explaining the reasons of the deviation.

*Jitter* is the second performance metric which is measured in the experiments. The jitter that the signals experience is easily calculated once the receive time of those messages is properly measured. As defined formerly, jitter is the time deviation from the periodicity and is, obviously, applicable to the periodic signals only. Also, it should be noted that this jitter definition is applicable per signal rather than the group of signals. The average jitter of a signal can be formulated as follows.

$$Jitter\_Avg = \frac{1}{N} \times \sum_{n=1}^{N} t_{rx,sw}(n) - t_{rx,sw}(n-1) - signal\_period \qquad (6\text{-}3)$$

According to this formula the jitter is calculated in all experiments for the periodic signals and the jitter results are discussed under the corresponding chapter where the experiment is explained in details.

*Protocol conversion correctness* is the main performance metric for the functionality of the Gateway. If verified, this performance metric means that the Gateway performs its very basic functionality. In this context, the protocol conversion includes being able to receive messages from both networks, map the incoming messages to be sent in the other network according to the network scheduling configuration and writing the payload to be sent to the right buffer while satisfying the specific requirements of both networks.

*Signal mapping* is another performance metric which is about the Gateway functionality. This metric covers the capability of the Gateway to process the received messages in the signal level. While the Gateway is able to segment an incoming message into pieces to transmit each in distinct messages possibly together with other signals, it is also able to assemble several signals in order to send them in a single message.

*Processing delay* is simply defined as the time during which a signal stay in the Gateway while crossing it. Therefore, this metric is the measure of the quality of the Gateway design and the smaller the processing delay is, the better performance the Gateway exhibits.

## 6.2 OVERVIEW OF THE EXPERIMENTS

The basic aim of this study is to design and implement a high performance FlexRay-CAN Gateway which satisfies all the functionalities that are discussed in CHAPTER 3 and conduct an end-to-end performance analysis of the inter-connected FlexRay and CAN networks in terms of delay and jitter. The performance of the Gateway mainly signifies the processing delay of the node and its contribution to the overall jitter. In this context, the performance of the Gateway solely depends on its design and implementation. On the other hand, end-to-end performance analysis of the interconnected network, which is composed of FlexRay network, CAN Network and the Gateway node, depends on the performances of both networks and the Gateway unit separately. This is because, a signal that is generated from one end of the network experience a delay in both FlexRay and CAN networks and a processing delay in the Gateway during its transmission to the other end. The variations in these delay components also cause the signal to experience a jitter if it is a periodic signal. Therefore, the goal in designing the entire interconnected network is to minimize the end-to-end delay and the jitter that the signals experience as well as guaranteeing that all of the signals are delivered to the destination within their deadlines. The relationship between the deadline of the signals which cross the Gateway and the delay components that those very same signals experience in each network are given in (6-4).

$$d_S > d_C + d_{FR} + t_{GW} \qquad (6\text{-}4)$$

where $d_S$ is the deadline of the signal, $d_C$ is the time duration that the signal passes in CAN network, $d_{FR}$ is the delay that the signal experiences in FlexRay network and $t_{GW}$ is the processing delay of the Gateway.

Before setting up a network, the first task is to compute the message schedule for FlexRay, CAN networks and the Gateway node so that all of the messages meet their deadline requirements. In the context of message scheduling, the deadlines $d_C$ and $d_{FR}$ in (6-4) are free parameters that have to be chosen such that their sum is smaller than the signal deadline, $d_S$, reduced by the processing delay of the Gateway delay as shown in (6-5).

$$d_S - t_{GW} > d_C + d_{FR} \tag{6-5}$$

It is readily observed that the deadlines of the CAN messages sent by the Gateway can only be evaluated if the deadlines of the corresponding FlexRay messages are known and vice versa. Therefore, it becomes impossible to determine a scheduling scheme for either network since the deadlines, before which the signals must be delivered, can not be fixed. In order to break this cyclic dependency, one of $d_C$ or $d_{FR}$ shall be fixed so that the other variable can be determined as in (6-5). It is proposed in [30] to first compute a CAN priority assignment such that the messages passing the Gateway have the shortest possible worst-case response times on CAN since the uncertainty in the worst-case response time and the message jitter is introduced by the slower and event-triggered CAN network. After the CAN deadline is computed for each of the messages, the FlexRay deadline can easily be calculated as in (6-5) and the FlexRay network is scheduled accordingly.

In [30] three different scheduling schemes are proposed and discussed for the CAN network. On the other hand, two priority assignment schemes are suggested for the FlexRay network.

It is obvious that the scheduling schemes that the networks utilize directly affect the delay that the messages experience on those networks. Since one of our goals is to study the end-to-end performance of an inter-connected FlexRay and CAN networks in terms of delay and jitter, we analyze the performances of the sole CAN network and sole FlexRay network with the scheduling schemes proposed in [30]. Moreover, having examined the behaviour of the individual CAN and FlexRay networks, we will be able to understand and exhibit the performance of the Gateway unit better while performing experiments on the whole network where there exist CAN network, FlexRay network and the Gateway node.

Apart from these experiments, where the performance of CAN and FlexRay networks are analyzed according to the scheduling schemes suggested in [30], also several other experiments are held to verify and test the Gateway functionality and the performance. During the testing of the Gateway functionality and the performance, a bigger network, in which there also exist CAN and FlexRay networks, is established.

All of the experiments held during this study are listed below in order to provide a more comprehensive picture.

- Experiments to investigate the impact of scheduling
    - CAN Experiments
        - Conventional Scheduling
        - Prioritized Scheduling
        - Scheduling with Fixed Priority
    - FlexRay Static Segment Experiments
        - Scheduling Without Jitter
        - Scheduling With Minimum FID
    - FlexRay Dynamic Segment Experiments

        - 18 Minislots
        - 19 Minislots
        - 20 Minislots

- Overall network performance: Experiments with the Gateway
    - Gateway Functionality
        - Protocol Conversion
        - Signal Mapping
    - Gateway Performance
        - Real-Time measurements
        - Effect of polling frequency

### 6.2.1 Experiment Set-Up

The structure of the experiment set-up for all of the experiments are similar to each other. The set-up consists of SK-91465X-100MPC evaluation boards, CAN and FlexRay PCB busses, cables with 1-to-1 D-Sub-9 female connectors at both end, FlexCard Cyclone II SE network analyzer card and a PC. Each SK-91465X-100MPC starter kit is used as an individual node composing the network. Since SK-91465X-100MPC has both CAN and FlexRay interfaces on it, it can be used as a CAN node, a FlexRay node or the Gateway node in the experiments. The nodes composing the network, i.e the SK-91465X-100MPC starter kits, are connected to each other via hardware busses. The hardware busses used in the experiments are the PCBs with D-Sub-9 Male connectors mounted on it. All of 9 pins of those connectors are connected to each other through the PCB to enable several distinct nodes communicate each other through the PCB. A maximum number of 9 nodes can be connected via the PCB. The photograph of the PCB bus used in the experiments is shown in the Figure 6-1.

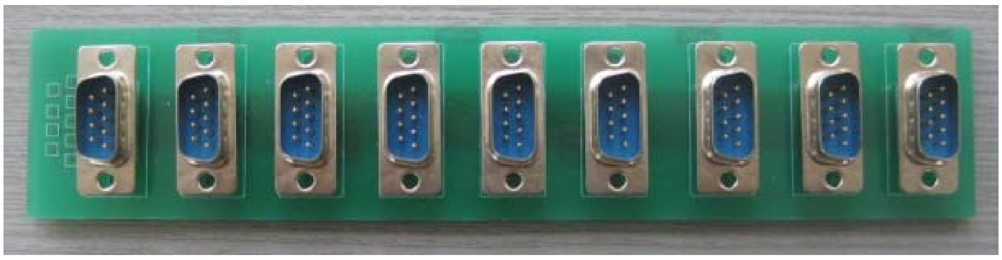


**Figure 6-1** The PCB Bus Used For Both FlexRay and CAN Bus

While used as FlexRay bus, two distinct PCB busses are used in the experiments, one for each FlexRay channel. The FlexRay bus and the CAN bus are physically identical to each other except that there are termination resistors on the CAN bus. Two parallel 120 Ω resistors are welded on the two different connectors of the PCB bus between the live pins, namely CAN_L and CAN_H to provide proper termination. Without the termination is provided, CAN communication can be established between the SK-91465X-100MPC nodes up to 100 kbps data rate. Beyond this bit rate, SK-91465X-100MPC requires termination for the CAN communication. However, without termination, FlexCard analyzer is not able take any CAN measurement even for the bit rates smaller than 100 kbps. Therefore, throughout the experiments with all CAN data rates, smaller or greater than 100 kbps, the PCB CAN bus is terminated to properly log and analyze the bus.

FlexCard Cyclone II SE is used in all of the experiments as a FlexRay node or a CAN node or both at the same time depending on the experiment. Although it is possible to use the FlexCard Cyclone II SE to send messages in the network, it is only used as a receive node to monitor the data exchange through the CAN and the FlexRay busses. While, to monitor the CAN bus, no arrangement is required to be done on FlexCard Analyzer except the CAN data rate, in order to, properly, analyze the FlexRay bus, the *.chi file produced by the FlexConfig™ for the FlexCard must be included in the FlexAlyzer software. Therefore, while scheduling the FlexRay network via FlexConfig™, FlexCard is also defined to be a receiving FlexRay node for all of the time slots.

The final equipment used in all of the experiments is a PC. Via the PC, the microcontrollers of the nodes are programmed by using the FME FR-Flash Programmer V4.0.2.1. Also, the data exchange through the network is monitored and logged by the FlexAlyzer software which is installed in the PC. Finally the logged data is parsed offline by using the parsing program, that we wrote in the PC, to obtain the results of the performance metrics.

Illustration and the photograph of the Gateway network are given in Figure 6-2 and Figure 6-3, respectively.

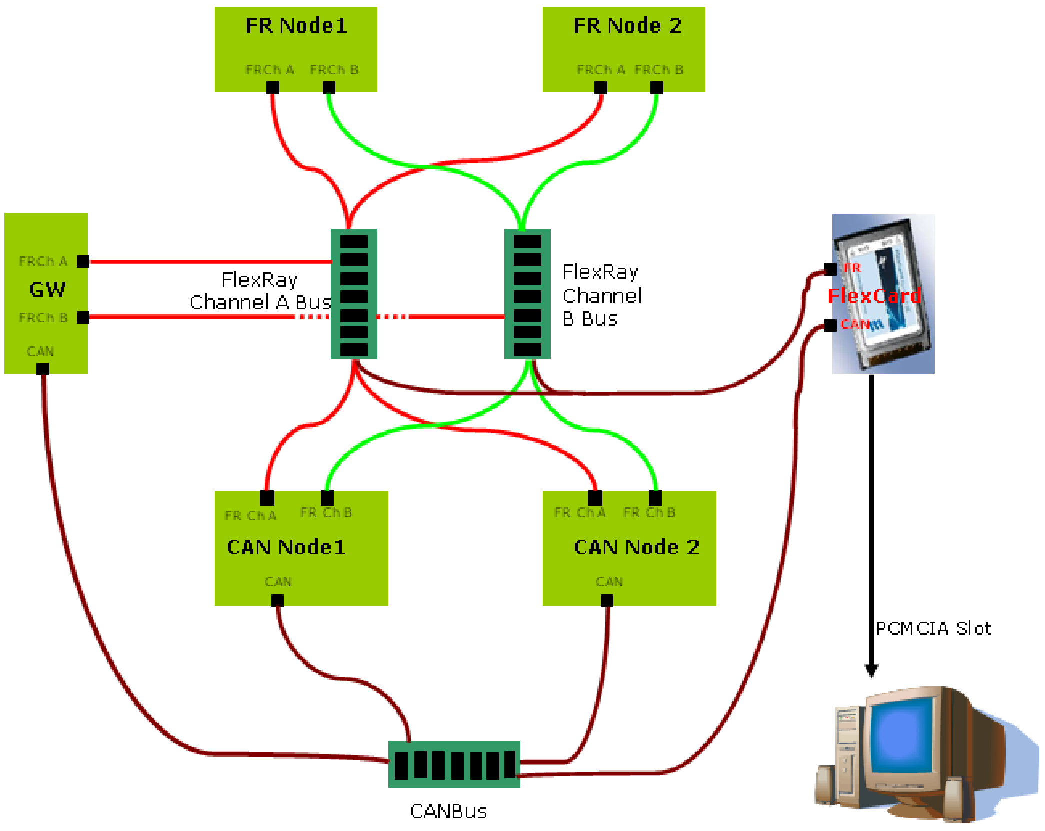


**Figure 6-2** The Gateway Network Illustration

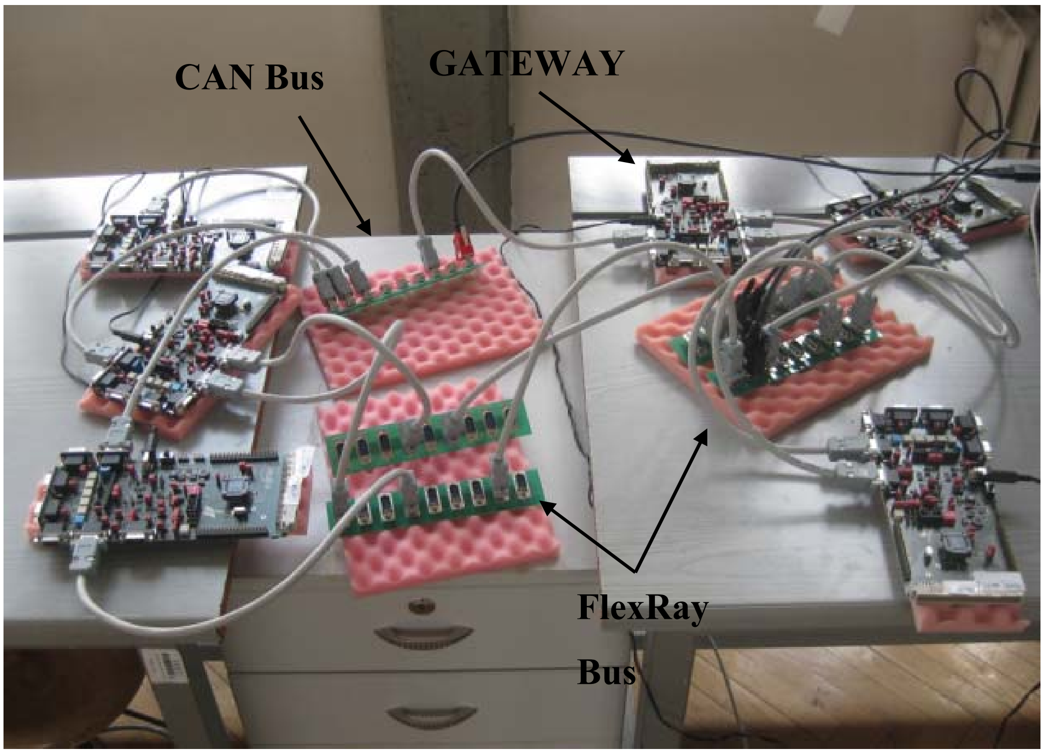


**Figure 6-3** The Gateway Network Photograph

When Figure 6-2 and Figure 6-3 are examined, it is seen that CAN nodes are also connected to the FlexRay network. The reason for this connection is that so as to take the time measurement that is valid throughout the whole network, all the nodes composing the network must be synchronized to each other. CAN bus is an event trigger bus without any synchronization. Because of this reason the synchronization among all nodes is established through the FlexRay bus via the FlexRay interfaces on SK-91465X-100MPC. This way, the transmit and the receive time of all of the messages, whether from CAN node or FlexRay node, can easily be tagged in order to be used in the analysis of the network. The considerations taken into account during the time measurements and the logging are explained in the following chapter in more detail.

### 6.2.2 Time Measurements

The experimental evaluation of the timing performance requires the correct time measurements. As introduced in the preceding chapter, the all nodes in the network must be synchronized to each other through the FlexRay network so that the all of the signals exchanged via the network can be time tagged properly. An experiment set-up consisting of a Gateway node, CAN nodes which are connected to also the FlexRay network, FlexRay nodes and the FlexCard which is used to monitor the whole network traffic, is illustrated Figure 6-4.

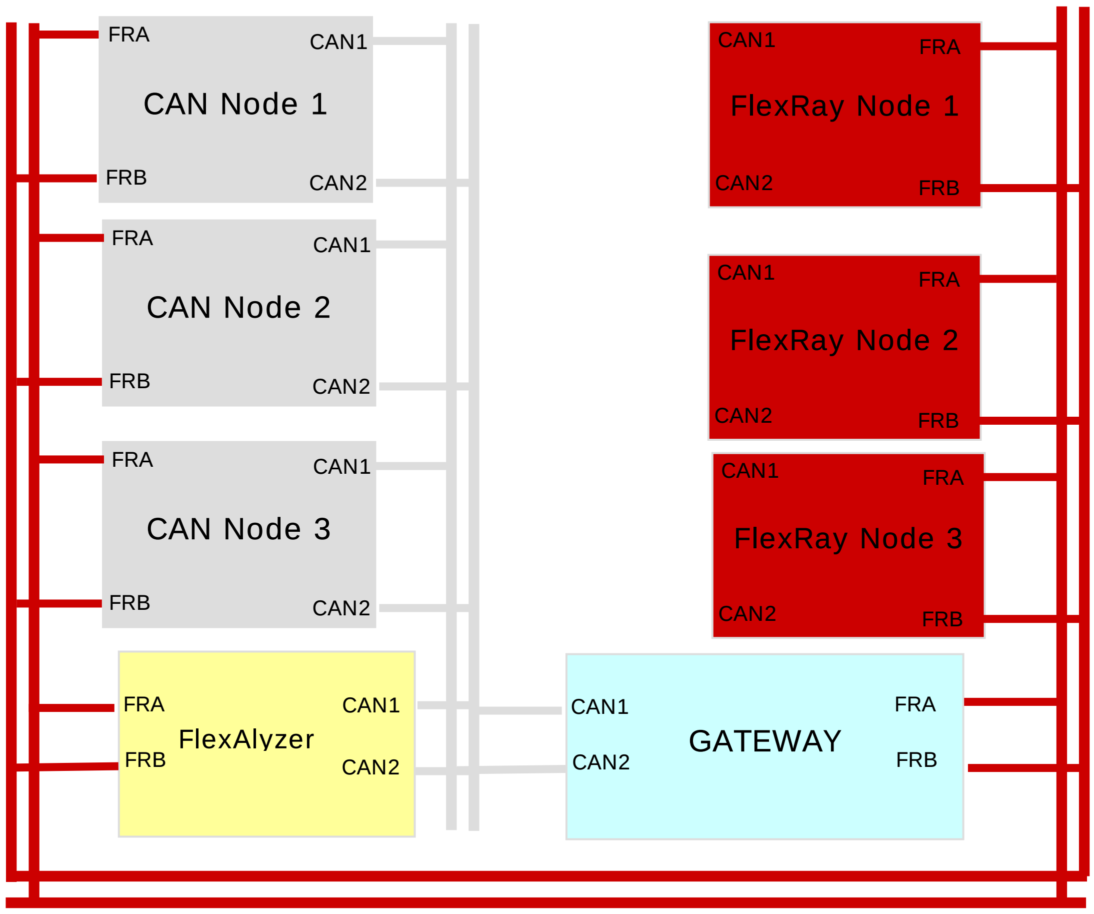


**Figure 6-4** The Gateway Network

Once all of the nodes connected to the network are synchronized to each other, four different timestamps are included, at four distinct points of the network, into the payload of the message traveling from the originating node to the destination. The spots, in the network, that the timestamps are added into the payload are the exit point from the CAN/FlexRay node, the arrive point in the Gateway, the exit point from the Gateway and the arrive point in the FlexRay/CAN node. By means of these 4 timestamps it becomes possible to measure the end-to-end delay and the jitter that a signal experiences exclusively, in CAN network, in Gateway, in FlexRay network and throughout whole of its travel.

If we consider, first, the travel of a packet from CAN to FlexRay bus the scenario is as the following. The message packet originates from a CAN node and it is sent on the CAN bus. Then the message is received by the Gateway via its CAN interface. After the Gateway has processed and mapped the signal to a FlexRay signal, the signal is transmitted through the FlexRay network. Finally the signal arrives to its final destination, a FlexRay node. In all of the experiments, FlexCard Cyclone II SE is used as the FlexRay node which receives all signals coming from the Gateway. (See Figure 6-5)

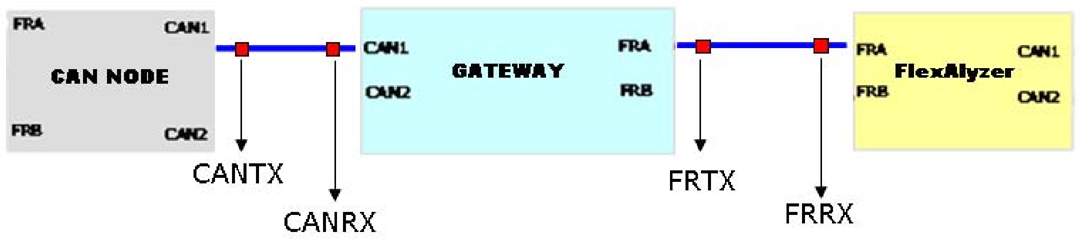


**Figure 6-5** Time Tagging: CAN2FR

The timestamp CANTX is obtained in CAN0_SendMessage routine, which is explained in CHAPTER 5, just before loading the payload to the outgoing message buffer of the CAN Controller. Next, when this CAN message packet is received by the Gateway, the timestamp CANRX is obtained just at the beginning of the CAN0_ReadMessageBuffer function which is called in the CAN interrupt subroutine. Finally, the Gateway obtains the FRTX timestamp indicating the leaving time of the packet from the Gateway just before storing the payload to the FlexRay sending buffers via ffrd_api_tx_handler_buffer() function. As introduced in section 6.1, those timestamps obtained at some certain points of the network do not represent the actual time that they refer to. Rather, they are either obtained a little bit earlier or a little bit later. The source of these errors is discussed in section 6.2.3 and the impact of the difference, which is no significant, is discussed in more detail for each experiment in further sections. The timestamp FRRX is obtained by the FlexAlyzer showing the receive time of the packet in the FlexCard. Therefore FFRX does not come in the payload of the signal but rather it is created and logged by the FlexAlyzer analyze software.

The timestamps, which are included to the payload of the message, are obtained via two different time service functions of the FlexRay Software driver. These functions return the network time by means of the FlexRay network fundamental time units which are the cycle number and the macrotick count. Since the time service functions return the cycle number and the macrotick count in 8-bits and 16-bits, respectively, each timestamp engraved in the packets are 3 bytes long. Therefore, the time tags except FRRX, namely CANTX, CANRX and FRTX, shown in Figure 6-5 are all 3 bytes long. As a consequence of this, the length of the FlexRay message at the end of the Gateway becomes to 9 bytes long. Since the maximum payload length allowed in FlexRay protocol is 254 bytes, 9 bytes of timestamp data is easily be sent through FlexRay bus to be received by the FlexCard.

As mentioned, all of the messages are finally received by the FlexCard. The incoming network traffic to the FlexCard is logged by the FlexAlyzer software. Since the all 3 timestamps are already in payload of the incoming traffic and the FlexAlyzer itself tags the $4^{th}$ timestamp in the log as the receive time of the messages, in the log file extracted by the FlexAlyzer, all four timestamps shown in Figure 6-5 are present together. In order to examine those log files offline to measure the performance metrics, distinct text parsing scripts are written for each of the experiments.

For FlexRay to CAN communication the signal which is produced by a FlexRay node sinks at the Gateway node. After being processed in the Gateway, the message is transmitted via CAN bus. Again, the final destination of the message has to be the FlexCard where the network traffic is monitored and analyzed. The timestamps must also be included in the payload as shown in Figure 6-6, so as to be able to analyze the network performance afterwards depending on the data log stored by the FlexAlyzer.

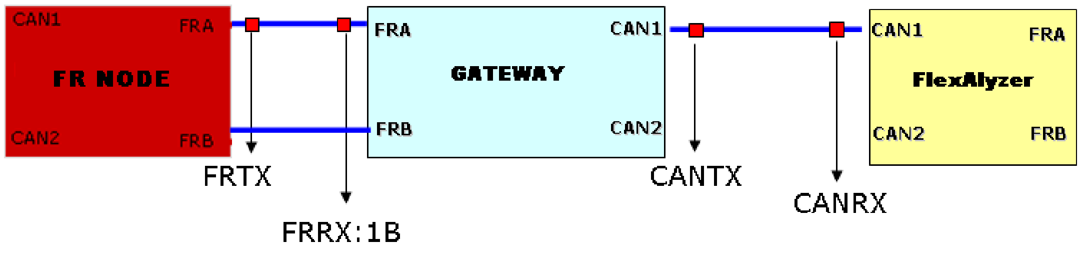
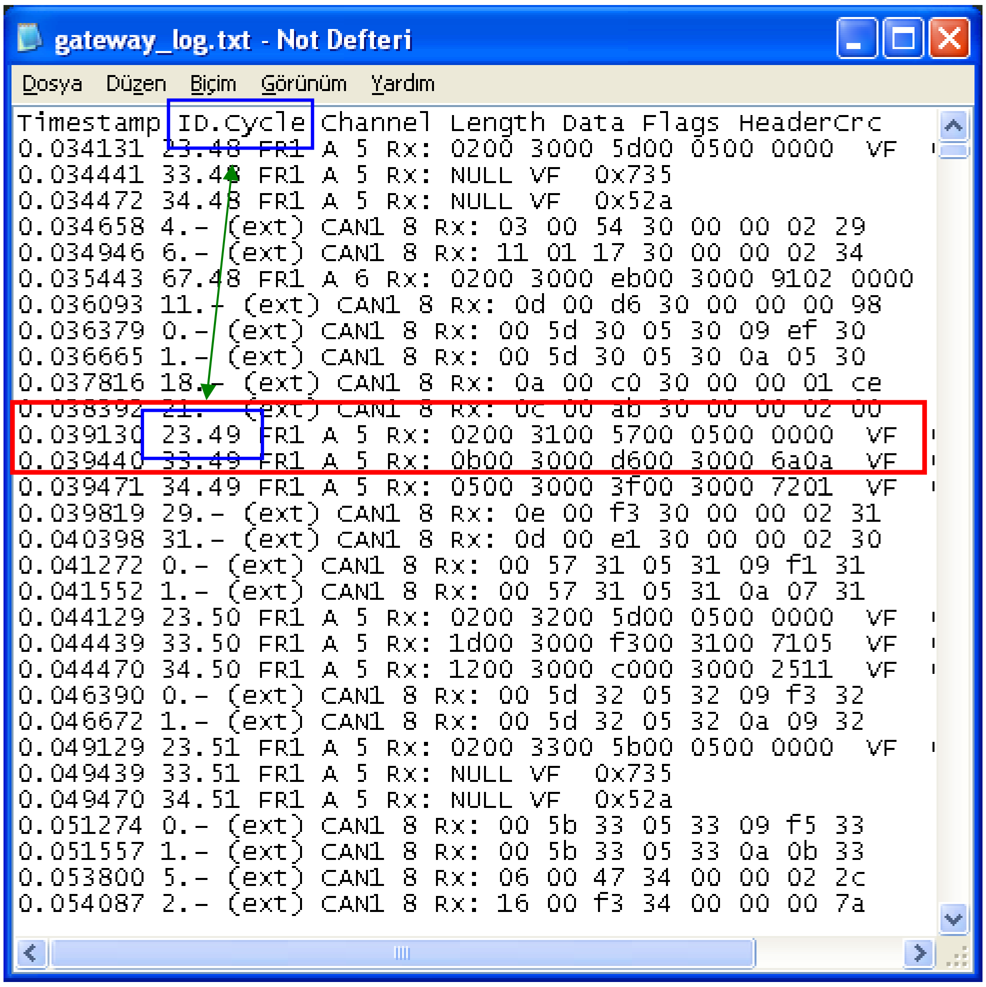


**Figure 6-6** Time Tagging: FR2CAN

Although each timestamp must be 3 bytes long, the timestamp, FFRX, is only composed of the cycle number which is 1 byte long as seen from the Figure 6-6. Because, if all the timestamps were, as in the previous case, 3 bytes long, the Gateway would have to send 9 bytes of data through the CAN bus after having included the timestamp, CANTX. However, it is impossible to transmit a message bigger than 8 bytes through CAN bus in a single packet. On the other hand, if the timestamp data were to be transmitted as two CAN frames, this would impair the analysis results of the experiments. Therefore, one of the FlexRay network time units, namely, cycle number and macrotick count, would have to be omitted from one of the timestamps. It is seen that the omission of the macrotick count from the timestamp, FRRX, has no impact on the end-to-end delay and the jitter calculations. The reason for this is that the receive time of a static slot message is exactly determined in terms of macrotick due to the TDMA structure of the FlexRay network. Besides, the receive cycle number must, still, be included in the timestamp since the FlexRay messages might be sent with variety of repetition and offset settings which makes the receive time of the message ambiguous.

As a result of this small change in the time tagging structure, the total length of the timestamps in the payload happens is 7 bytes long. This 7 byte time data is received by the FlexCard in the end. The FlexCard, also, puts the CANRX time tag and stores all the time data in a log file. As explained previously, those log files are analyzed by means of the text parsing codes and the results about the performance of the network is this way obtained.

As mentioned above, the timestamps FRRX and CANRX are obtained via the FlexCard itself in the two opposite communication directions of the Gateway. So that the reliable results can be obtained as a result of parsing the log file created by the FlexAlyzer software, those timestamps, FRRX and CANRX, must also be in the same time units as the other time tags. A view from the log file exported by the FlexAlyzer software is displayed in Figure 6-7.


gateway_log.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım

```
Timestamp ID.Cycle Channel Length Data Flags HeaderCrc
0.034131 23.48 FR1 A 5 Rx: 0200 3000 5d00 0500 0000  VF
0.034441 33.48 FR1 A 5 Rx: NULL VF  0x735
0.034472 34.48 FR1 A 5 Rx: NULL VF  0x52a
0.034658 4.- (ext) CAN1 8 Rx: 03 00 54 30 00 00 02 29
0.034946 6.- (ext) CAN1 8 Rx: 11 01 17 30 00 00 02 34
0.035443 67.48 FR1 A 6 Rx: 0200 3000 eb00 3000 9102 0000
0.036093 11.- (ext) CAN1 8 Rx: 0d 00 d6 30 00 00 00 98
0.036379 0.- (ext) CAN1 8 Rx: 00 5d 30 05 30 09 ef 30
0.036665 1.- (ext) CAN1 8 Rx: 00 5d 30 05 30 0a 05 30
0.037816 18.- (ext) CAN1 8 Rx: 0a 00 c0 30 00 00 01 ce
0.038392 21.- (ext) CAN1 8 Rx: 0c 00 ab 30 00 00 02 00
0.039130 23.49 FR1 A 5 Rx: 0200 3100 5700 0500 0000  VF
0.039440 33.49 FR1 A 5 Rx: 0b00 3000 d600 3000 6a0a  VF
0.039471 34.49 FR1 A 5 Rx: 0500 3000 3f00 3000 7201  VF
0.039819 29.- (ext) CAN1 8 Rx: 0e 00 f3 30 00 00 02 31
0.040398 31.- (ext) CAN1 8 Rx: 0d 00 e1 30 00 00 02 30
0.041272 0.- (ext) CAN1 8 Rx: 00 57 31 05 31 09 f1 31
0.041552 1.- (ext) CAN1 8 Rx: 00 57 31 05 31 0a 07 31
0.044129 23.50 FR1 A 5 Rx: 0200 3200 5d00 0500 0000  VF
0.044439 33.50 FR1 A 5 Rx: 1d00 3000 f300 3100 7105  VF
0.044470 34.50 FR1 A 5 Rx: 1200 3000 c000 3000 2511  VF
0.046390 0.- (ext) CAN1 8 Rx: 00 5d 32 05 32 09 f3 32
0.046672 1.- (ext) CAN1 8 Rx: 00 5d 32 05 32 0a 09 32
0.049129 23.51 FR1 A 5 Rx: 0200 3300 5b00 0500 0000  VF
0.049439 33.51 FR1 A 5 Rx: NULL VF  0x735
0.049470 34.51 FR1 A 5 Rx: NULL VF  0x52a
0.051274 0.- (ext) CAN1 8 Rx: 00 5b 33 05 33 09 f5 33
0.051557 1.- (ext) CAN1 8 Rx: 00 5b 33 05 33 0a 0b 33
0.053800 5.- (ext) CAN1 8 Rx: 06 00 47 34 00 00 02 2c
0.054087 2.- (ext) CAN1 8 Rx: 16 00 f3 34 00 00 00 7a
```

**Figure 6-7** A view from the log file exported by FlexAlyzer

A received FlexRay message is encircled in Figure 6-7 to give an example for the CAN2FR communication direction of the Gateway. In this direction, a message originated from the CAN bus is finally received by the FlexCard through the FlexRay bus. Since all the three timestamps, until arriving in the FlexCard hardware, namely CANTX, CANRX and FRTX, are described in terms of the cycle number and the macrotick count, the timestamp, FRRX, which is obtained by the FlexCard, must also be in the same form. While the former three timestamps exist in the payload of the received message, the time tag, FRRX, is encircled by a blue rectangle in Figure 6-7. As seen, the FlexAlyzer gives this time tag in the form of the receive cycle and the receive ID which is not exactly the same as the form of the other three timestamps. However, this time tag can easily be converted to the conventional form since the receive ID indicates the receiving time slot of the signal and the length of a timeslot is fixed and known in terms of macrotick. Therefore, all 4 timestamps can be considered to be in the same form and the end-to-end delay that a message experiences from the CAN node to the FlexRay node can be formulated as in (6-6).

$$\begin{aligned}&(FRRX.Cycle - CANTX.Cycle) \times Cycle\_Length\_in\_MT + \\ &\left[(IDRx - 1) \times SS\_Length\_in\_MT + Action\_Offset\_in\_MT\right] \\ &- CANTX.Macrotick\end{aligned} \tag{6-6}$$

where

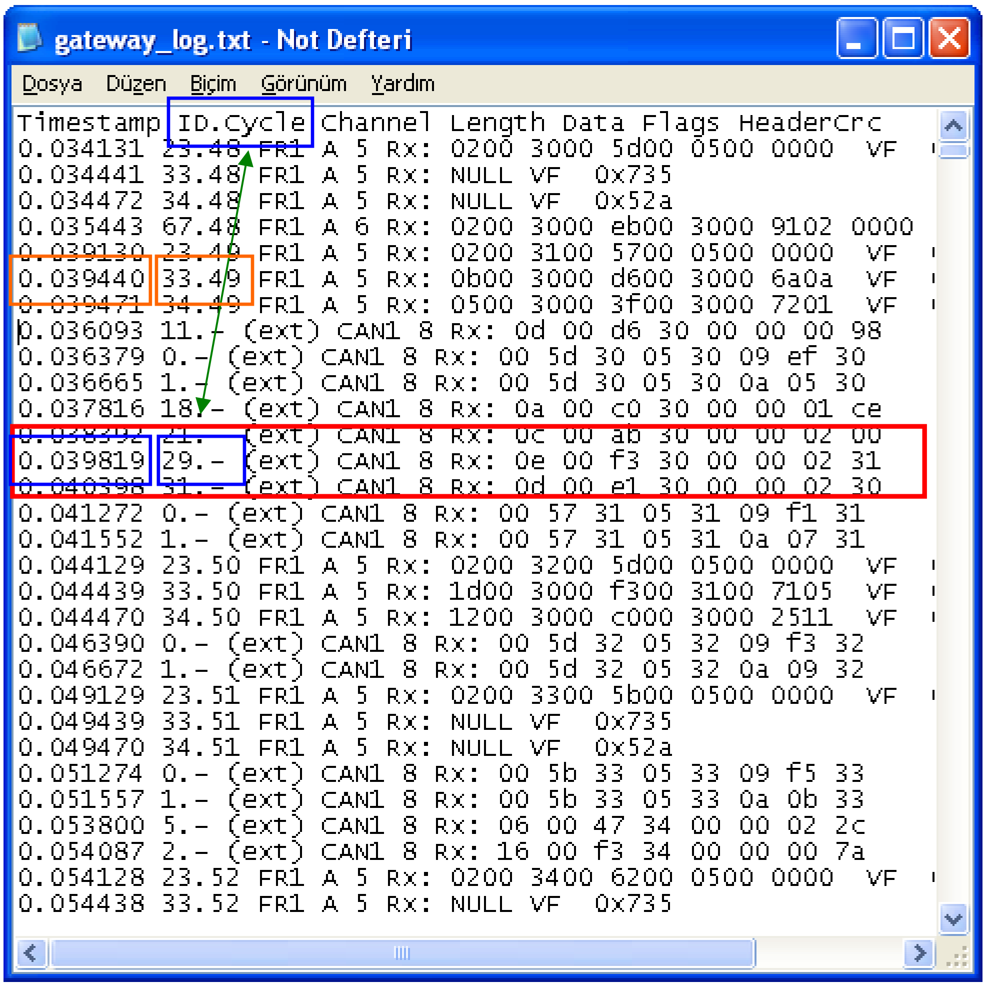
FRRX.Cycle and the CANTX.Cycle are the cycle numbers of the timestamps FRRX and CANTX, respectively,

Cycle_Length_in_MT, SS_Length_in_MT and Action_Offset_in_MT are the matrotick correspondences of the network parameters the Cycle Length, the Static Slot Length and the Action Offset Length respectively,

IDRx is the ID of the received FlexRay message via Flexcard and

CANTX.Macrotick is the macrotick count of the timestamp CANTX.

The task of obtaining the timestamp is not that straightforward for the FR2CAN direction as the CAN2FR communication direction of the Gateway. To illustrate the situation in the FR2CAN direction, a view from the log file of an experiment is given Figure 6-8.


gateway_log.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım

```
Timestamp ID.Cycle Channel Length Data Flags HeaderCrc
0.034131 23.48 FR1 A 5 Rx: 0200 3000 5d00 0500 0000  VF
0.034441 33.48 FR1 A 5 Rx: NULL VF  0x735
0.034472 34.48 FR1 A 5 Rx: NULL VF  0x52a
0.035443 67.48 FR1 A 6 Rx: 0200 3000 eb00 3000 9102 0000
0.039130 23.49 FR1 A 5 Rx: 0200 3100 5700 0500 0000  VF
0.039440 33.49 FR1 A 5 Rx: 0b00 3000 d600 3000 6a0a  VF
0.039471 34.49 FR1 A 5 Rx: 0500 3000 3f00 3000 7201  VF
0.036093 11.- (ext) CAN1 8 Rx: 0d 00 d6 30 00 00 00 98
0.036379 0.- (ext) CAN1 8 Rx: 00 5d 30 05 30 09 ef 30
0.036665 1.- (ext) CAN1 8 Rx: 00 5d 30 05 30 0a 05 30
0.037816 18.- (ext) CAN1 8 Rx: 0a 00 c0 30 00 00 01 ce
0.038392 21.- (ext) CAN1 8 Rx: 0c 00 ab 30 00 00 02 00
0.039819 29.- (ext) CAN1 8 Rx: 0e 00 f3 30 00 00 02 31
0.040398 31.- (ext) CAN1 8 Rx: 0d 00 e1 30 00 00 02 30
0.041272 0.- (ext) CAN1 8 Rx: 00 57 31 05 31 09 f1 31
0.041552 1.- (ext) CAN1 8 Rx: 00 57 31 05 31 0a 07 31
0.044129 23.50 FR1 A 5 Rx: 0200 3200 5d00 0500 0000  VF
0.044439 33.50 FR1 A 5 Rx: 1d00 3000 f300 3100 7105  VF
0.044470 34.50 FR1 A 5 Rx: 1200 3000 c000 3000 2511  VF
0.046390 0.- (ext) CAN1 8 Rx: 00 5d 32 05 32 09 f3 32
0.046672 1.- (ext) CAN1 8 Rx: 00 5d 32 05 32 0a 09 32
0.049129 23.51 FR1 A 5 Rx: 0200 3300 5b00 0500 0000  VF
0.049439 33.51 FR1 A 5 Rx: NULL VF  0x735
0.049470 34.51 FR1 A 5 Rx: NULL VF  0x52a
0.051274 0.- (ext) CAN1 8 Rx: 00 5b 33 05 33 09 f5 33
0.051557 1.- (ext) CAN1 8 Rx: 00 5b 33 05 33 0a 0b 33
0.053800 5.- (ext) CAN1 8 Rx: 06 00 47 34 00 00 02 2c
0.054087 2.- (ext) CAN1 8 Rx: 16 00 f3 34 00 00 00 7a
0.054128 23.52 FR1 A 5 Rx: 0200 3400 6200 0500 0000  VF
0.054438 33.52 FR1 A 5 Rx: NULL VF  0x735
```

**Figure 6-8** Illustration for the task of obtaining the CANRX

As seen from the Figure 6-8, the time tag obtained by the FlexCard is not in the form of the cycle count and the macrotick count since it receives the data from the CAN interface. On the other hand the time tag that the FlexAlyzer includes in the beginning of every line of the log file is the local time of the FlexCard indicating the amount of time that has passed since the beginning of the measurements. Therefore, this local time is not in the conventional form either and has no chance of being directly used. To overcome this problem, the log data regarding the FlexRay messages is used to make a connection to the time domain of the synchronization, which is the FlexRay time domain, as follows. First of all, we make use of the fact that the time delay to a received FlexRay message in the FlexCard can be easily calculated as described in the previous case. Secondly, it is known that in the beginning of every line of the log file the FlexCard puts the local time as the receive time of the corresponding message. Therefore, in order to find the delay that a message experiences from a FlexRay node to a CAN node, first of all, the delay up to a received FlexRay packet is calculated. Then, the time offset between the local time of the received CAN signal and the local time of the received FlexRay signal up to which the delay component has been measured, is added to the already calculated delay and this way, the total delay that the signal experiences is found.

This calculation can easily be illustrated on the Figure 6-8. For instance, suppose that the end-to-end delay between the production time of a FlexRay signal and the CAN signal encircled with a blue rectangle in Figure 6-8 is required to be calculated. Production time of the FlexRay message is already in the payload of the message. As described above, first of all, the time delay to an arbitrary FlexRay message, which is called "reference FlexRay message", is calculated. In Figure 6-8, "reference FlexRay message" is shown in orange rectangle. The local receive time of the FlexRay message is 0.039440 and this local time is called as "reference time". The time delay from the original FlexRay message to this "reference FlexRay message" is calculated as in the case of the CAN2FR communication direction of the Gateway. Having calculated this delay, it is known from the Figure 6-8 that the CAN message, which is encircled with a blue rectangle, is received at the local time 0.039819 which is (0.039819 - 0.039440) second later than the "reference FlexRay message". Therefore, this time offset is added upon the, previously, calculated delay value and this way the desired end-to-end delay is found. The end-to-end delay that a message experiences from the FlexRay node to the CAN node is formulated as follows.

$$\begin{bmatrix} (FRRX.Cycle\_ref - FRTX.Cycle) \times Cycle\_Length\_in\_MT + \\ \left[(IDRx\_ref - 1) \times SS\_Length\_in\_MT + Action\_Offset\_in\_MT\right] \end{bmatrix} \qquad (6\text{-}7)$$
$$- FRTX.MT + (CANRX.reftime - RfFR.reftime)$$

where

FRRX.Cycle_ref and the FRTX.Cycle are the cycle numbers of the timestamps "Reference FlexRay Message" and FRTX, respectively,

Cycle_Length_in_MT, SS_Length_in_MT and Action_Offset_in_MT are the matrotick correspondences of the network parameters the Cycle Length, the Static Slot Length and the Action Offset Length, respectively,

FRTX.MT is the macrotick number of the timestamp FRTX,

IDRx_ref is the ID of the "Reference FlexRay Message" and

CANRX.reftime and RfFR.reftime are the local receive time of the received CAN message and the "Reference FlexRay Message" in the FlexCard.

### 6.2.3 Quantitative Analysis of the Time Measurement Errors

In order to examine the end-to-end delay and the jitter values that the signals experience, throughout the experiments, we take measurements at certain points of the network as described in section 6.2.2. Obviously, the calculations of the performance metrics such as end to-end delay and jitter directly depend on that we take the time measurements accurately. However, it is inevitable that the time measurements taken in the experiments deviate for certain amount from the real time values for several reasons. The reasons for the deviation are listed below from the CAN bus perspective.

1. In the experiments, the time stamps are put on the messages before they, actually, are put on the wire. After the timestamp has been tagged in the software, the software continues to run for a certain time before the message

with the timestamp is transferred to the transmit buffers in the CAN Controller of the CPU. The first component of the deviation of the timestamps from the actual values is this software processing time. Transferring the message to the transmit buffer does not mean that the message is put on the wire. In the buffers, the messages experience certain delay before they are, first, passed to the CAN transceiver and then put on the CAN bus physically. This delay experienced in the hardware is the second component of the deviation of the timestamps from the actual values.

2. Unlikely, on the CAN receive side of the messages, the timestamps are tagged later than the actual receive time of the messages. The components of the deviations are as follows. When the packet is first received in the CAN node, the unit that welcomes the message is the transceiver. Then the transceiver passes the message to the CAN engine of the CPU. After some time in the buffer of the CAN Engine, the CAN Controller issues an interrupt alerting the software about the receive of a new CAN packet. The time up to this point is the time passed in the hardware. From the issuing time of the interrupt to the time where the application code running on the CPU reads the incoming message and obtains the timestamp, CPU processes the interrupt in the software. This time delay is the software component of the entire deviation.

3. When two nodes communicate through the CAN bus, the above discussions explain the deviations of the delay that the CAN signals experience from the actual delay values assuming that the network time of the communicating nodes are exactly the same. However, in reality, the clocks of the nodes in a network drift with respect to each other. Therefore, network time of the communicating nodes always differ for certain amount. At this point assume that the clock of the receiving node is ahead of that of the transmitting node. In such a situation, the receiving node tags the received message by a greater time value with the amount equal to the difference of the clocks of the nodes

than the time value that the node would tag the receiving message if there were no clock deviation between the nodes. Therefore, in addition to the first two factors, the clock difference between the nodes might also cause the calculated time to deviate from the real time.

4. The last error factor is the FlexAlyzer. The timestamps put on the log file of the FlexAlyzer have also error in it. The timestamps in the log file are used twice for the cases where the CAN messages are received via the FlexCard. In such a case, timestamps used in the calculations are the time tags for the "reference FlexRay message" and the received CAN signal. For the situations where the FlexAlyzer measurements are used twice, time deviations from the actual values get bigger.

As mentioned, the above explanations are written for the CAN communication time measurement errors. Although the very same factors affect the time measurements in the FlexRay network in the same manner, the errors do not appear in the FlexRay results. The reason for this is that the delay values of the FlexRay messages occur in the multiple of the FlexRay cycle which is 5ms since the FlexRay network contention is provided via TDMA. Since the time measurement errors are much smaller than the FlexRay cycle length, these errors do not affect the results obtained for the FlexRay bus. On the other hand, since the CAN bus is an event triggered bus, the time measurement errors appear in the CAN bus experiment results. For this reason, the above explanations which account for the deviation of the time measurements taken in the experiments from the actual time values are given from the CAN bus perspective.

Several experiments are run in order to quantitatively figure out the order of these errors listed above. Before continuing with the experiments, the error components, which are explained in the first two bullets above, are exhibited below in Figure 6-9.

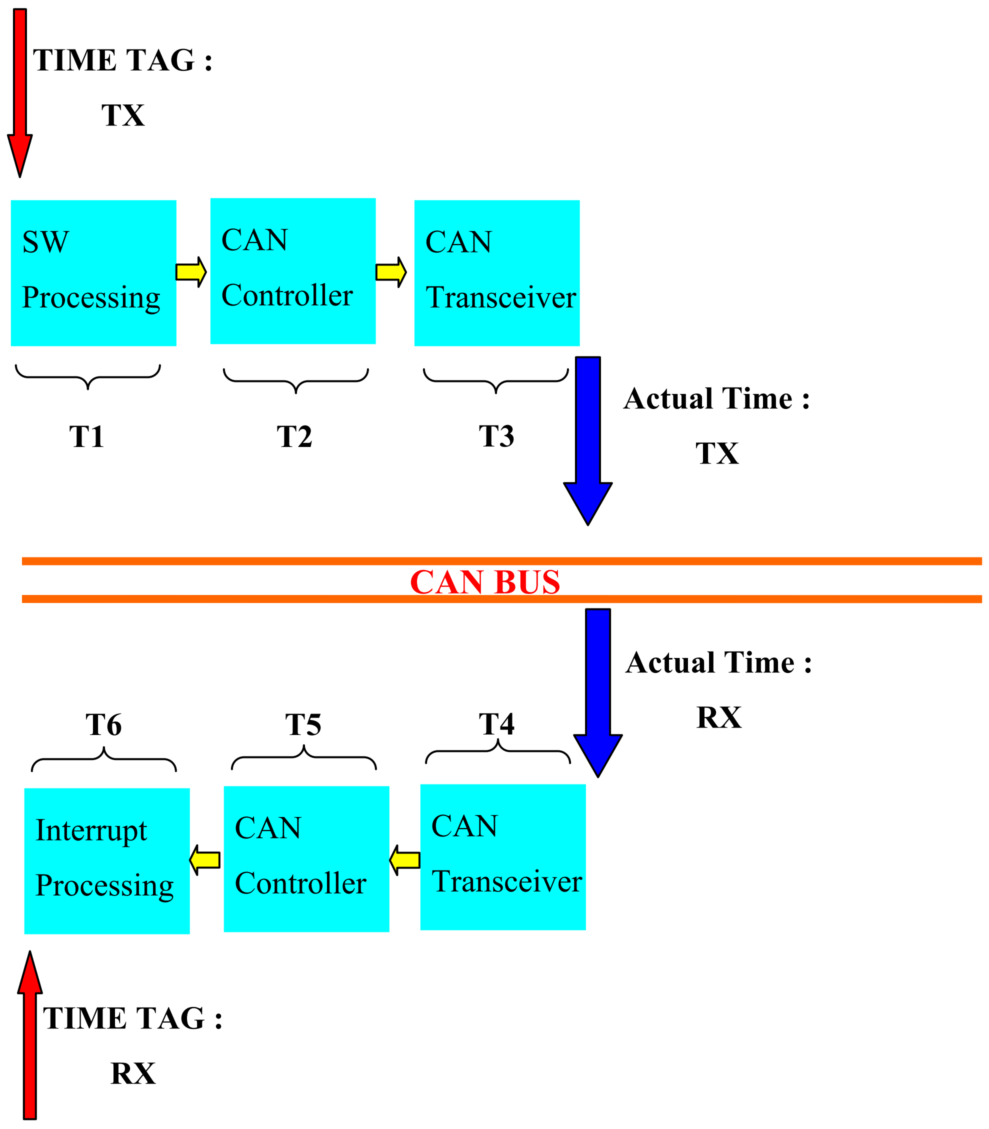


**Figure 6-9** Time Stamp Deviation from the Actual Time

Out of the 4 issues explained above, different ones are effective in different experiments when calculating the end-to-end delay that a signal experiences through the CAN bus. In fact, exploring the behavior of the error components under two cases will be enough. The first case comprises of the experiments where the both timestamps are tagged via the CPU in the delay calculations. In this context, the delay calculations for the CAN2CAN experiments and for the signals that traverse the Gateway in CAN2FR direction can be included under the first case. In such situations, the articles number 1, 2 and 3 are effective in the calculation of the error between the measured time and the actual time. On the other hand, the second case includes the experiments where one of the timestamps is obtained in the CPU and the other time tag is put via the FlexAlyzer. The delay calculations for the signals that travel in the Gateway in the FR2CAN direction can be the example for this case. The articles number 1, 3 and 4 are effective in the calculation of the error between the measured time and the actual time in the second type of situations.

To begin with the first case where the both timestamps are tagged via the CPU in the delay calculations, it is already mentioned that the articles number 1, 2 and 3 are effective in the calculation of the error between the measured time and the actual time. In order to calculate quantitatively the impact of these factors to the time measurement errors, the following simple experiment is set up. In the experiment, only one CAN message is sent through the CAN bus so that the packet does not lose any time for the contention. Therefore, the time that the message spends in the bus is supposed to be 270 µs where the bit rate of the CAN bus is 500 kbps and the message length is 8 B. The very same message is sent periodically in every 5 ms and the both transmit time and the receive time of the messages are tagged via the CPU. After the receive time tag have been put on the signal, the message is sent via FlexRay network where all those FlexRay messages are received and logged by the FlexAlyzer analyzer. When the log file that is output by the FlexAlyzer is processed by the text parsing program the maximum end-to-end delay that the messages experience is found to be 316 µs. This means that the time measurements taken by the CPU in the CAN2CAN experiments cause the end-to-end delay calculations to deviate about 46 µs from the actual end-to-end delay that the CAN packets experience. In this 46 µs delay error, all of the 3 articles, article number 1, 2 and 3, are accounted. Therefore, the sum of the time components from T1 to T6 in Figure 6-9 plus the time deviation caused because of the instability of the clocks of the nodes with respect to each other equals to 46 µs. However, in the real experiments there exist tens of messages rather than single message. As explained above, if single message existed in the network, the end-to-end delay of the message would deviate from the actual value with 46 µs. When there are multiple messages exchanged through the network, the messages might not be able to directly be passed through the CAN controller by issuing interrupt. Because at the time that a packet is received via CAN Controller and it is stored in the message buffers, there might exist some packets already have been waiting to be passed to the software via interrupt. Therefore, the message might wait, in addition to the time components depicted in Figure 6-9, for the messages whose priority is greater than itself to be processed. This process time is not the software processing time of the interrupt subroutine as shown in Figure 6-9 with T6 but the time for the processing time of the messages in CAN0_ReadMessageBuffer(). To sum up, in order to have an idea about the time quantity that a message waits in the CAN Controller for the situation where the multiple messages are exchanged via CAN bus, the duration of the routines named CAN0_ReadMessageBuffer, CAN0_ISR() and CAN0_STATUS_ISR_Handler() must also be known. Another experiment is set up to measure the duration of these routines. In the experiments, we make use of the FlexRay driver time service functions, namely, ffrd_api_get_cycle() and ffrd_api_get_mtick(). So as to calculate the duration of the routines, in the very beginning and at the very end of the functions, the network time is obtained via those time service functions. Then the time data obtained with these functions are sent through the FlexRay bus so that the data are logged via the FlexAlyzer software into a log file. Finally the duration of these functions are easily calculated by properly processing the log file. The results obtained for the functions are as follows.

T[CAN0_ReadMessageBuffer] = 35 µs

T[CAN0_ISR()] = 4 µs

T[CAN0_STATUS_ISR_Handler()] = 2 µs

Apart from this, a message packet in the CAN controller might also wait for the interrupt subroutines, IsrReloadTimer1() and IsrReloadTimer3(), which are used to clear the watchdog and to perform the FlexRay task, respectively. The reason for this is that the priorities of both interrupt subroutines are greater than that of CAN0_ISR(). Although the probability of this to occur is fairly small, it should be taken into consideration in the worst case delay calculations since it is enough for such a situation to happen only once to affect the calculation. Below listed are the results.

T[IsrReloadTimer1()] = 2 µs

T[IsrReloadTimer3()] = 10 + $n$*20 µs

where "$n$" is the number of CAN messages sent by the node.

Therefore, while analyzing the possible errors between the measurement time and the actual time, all of the factors summarized above should be considered for the first case.

The situation in the second case is a little bit different from the first case as previously described. In this case, while one of the timestamps is obtained in the CPU and the other time tag is put via the FlexAlyzer. It is more difficult in this case to find out the error in the delay calculations since the calculations involve the FlexCard hardware in which we have no control. Either, there exists no information in the data sheets of the FlexCard and the FlexAlyzer software about the measurement errors of the tools. As mentioned, the error components effective in the second case are the 1st, 3rd and the 4th articles discussed above. 1st article comprises of the error components T1, T2 and T3 as shown in Figure 6-9. It is logical to accept this error component as the half of the error which is depicted in Figure 6-9 and calculated to be 46 µs above. Therefore, the error component for the 1st article is found to be 23 µs. The error components discussed in the 3rd and the 4th articles are related to each other and can be figured out by examining the behavior of the FlexAlyzer. In order to measure the error in the FlexAlyzer, a real scenario is run and the log file obtained by the FlexAlyzer is observed. The receive time of the CAN packets are recorded in the log file. It is seen that the consecutive CAN messages are reported to be received with 295 μs apart. This means that when a burst of CAN messages with N members is received in the FlexCard, the last received message will be reported with an error of (N-1)*25 μs.

The possible the error components discussed so far which might occur during the end-to-end delay calculations are summarized quantitatively in Table 6-1.

**Table 6-1** Error Components in the CAN2CAN Delay Calculations

| MEASUREMENT ERRORS | |
|---|---|
| Error Components | Duration (μs) |
| TX Error (T1 + T2 + T3) | 23 |
| RX Error (T4 + T5 + T6) | 23 |
| Hardware Error (T1 + T2 + T3 + T4 + T5 + T6) | 46 |
| T[CAN0_ReadMessageBuffer] | 35 |
| T[CAN0_ISR()] | 4 |
| T[CAN0_STATUS_ISR_Handler()] | 2 |
| T[IsrReloadTimer1()] | 2 |
| T[IsrReloadTimer3()] (for *n* CAN messages) | 10 + *n**20 |
| FlexAlyzer Error (for a burst of N packets) | (N-1) * 25 |

## 6.3 DISCUSSION OF THE EXPERIMENTS

The list of the experiments held throughout this thesis study is given in Section 6.2. In this section, each experiment will be discussed separately in more detail. The message sets used in the experiments, the measurements, the set-up changes done specific to the experiments, the results obtained and the discussions about the results including the possible measurement faults are the topics that are explained in the following sections.

### 6.3.1 CAN Experiments

As mentioned previously, the main goal of this thesis is to design and implement a high performance FlexRay-CAN Gateway and make an end-to-end performance analysis of the inter-connected FlexRay and CAN networks in terms of delay and jitter. In this section, we examine the performance of CAN network with respect to different scheduling schemes proposed in [30]. The delay and the jitter values experienced by the signals in the CAN network directly affect the end-to-end delay and the jitter that the signals experience in the inter-connected network. Therefore, examining the behavior of the CAN network is also useful to visualize the bigger picture where both CAN and FlexRay networks and the Gateway unit exist.

Three experiments with distinct CAN scheduling schemes are held as described in [30]. The CAN network is composed of 3 distinct nodes with the data rate of 500 kbps. The aim of the experiments is to measure the delay and the jitter that the CAN signals experience when the priorities of the CAN messages are assigned according to respective scheduling algorithm proposed in [30]. The experimental results are compared with the worst case response times of the messages which are computed with an analytical approach in [30].

#### 6.3.1.1 Conventional Scheduling CAN Experiment

A well known scheduling approach for real time systems is simply to assign the priorities to the CAN messages increasing with the decreasing deadlines of the messages. Therefore, in Conventional Scheduling, smaller the deadline that the signal has, higher the priority that is assigned to the signal.

The message set which is used in the experiment and the priority assignment for the messages as per the Conventional Scheduling algorithm are given in Table 6-2 and Table 6-3, respectively.

**Table 6-2** Message Set for CAN Scheduling

| Signal | Period | Deadline | Length | Message | Period | Deadline | Length |
|---|---|---|---|---|---|---|---|
| C1 | 10 ms | 2.5 ms | 8B | C14 | 20 ms | 20 ms | 8B |
| C2 | 5 ms | 5 ms | 8B | C15 | 20 ms | 20 ms | 8B |
| C3 | 5 ms | 5 ms | 8B | C16 | 20 ms | 20 ms | 8B |
| C4 | 10 ms | 5 ms | 8B | C17 | 20 ms | 20 ms | 8B |
| C5 | 10 ms | 5 ms | 8B | C18 | 20 ms | 20 ms | 8B |
| C6 | 10 ms | 7 ms | 8B | C19 | 20 ms | 20 ms | 8B |
| C7 | 10 ms | 7.6 ms | 5B | C20 | 20 ms | 20 ms | 8B |
| C8 | 10 ms | 10 ms | 8B | C21 | 20 ms | 20 ms | 8B |
| C9 | 10 ms | 10 ms | 8B | C22 | 20 ms | 20 ms | 8B |
| C10 | 10 ms | 10 ms | 8B | C23 | 20 ms | 20 ms | 8B |
| C11 | 10 ms | 10 ms | 8B | C24 | 20 ms | 20 ms | 8B |
| C12 | 10 ms | 10 ms | 8B | C25 | 20 ms | 20 ms | 8B |
| C13 | 10 ms | 10 ms | 8B | C26 | 20 ms | 20 ms | 8B |

**Table 6-3** Priority Assignment Using Conventional CAN Scheduling

| Signal | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 |
|---|---|---|---|---|---|---|---|---|---|
| Priority | 1 | 0 | 4 | 3 | 2 | 5 | 6 | 12 | 11 |
| Signal | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 |
| Priority | 10 | 9 | 8 | 7 | 25 | 24 | 23 | 22 | 21 |
| Signal | C19 | C20 | C21 | C22 | C23 | C24 | C25 | C26 | |
| Priority | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | |

The message parameters in the Table 6-2 are inspired from the message set of a real automotive company and only minor modifications are done in producing the message set. The 26 CAN signals are distributed to 3 CAN nodes as shown in Table 6-4.

**Table 6-4** Distribution of the Signals in CAN Nodes

| CAN Node | Signals |
|---|---|
| CAN Node 1 | C5, C7, C8, C16, C17, C18, C19, C20, C26 |
| CAN Node 2 | C1, C2, C3, C4, C6, C9, C10, C11, C12, C13, C14, C15, C21, C22, C23 |
| CAN Node 3 | C24, C25 |

The CAN nodes produce the signals as shown in Table 6-4 according to the message set and the priority assignment as explained. The aim in performing this experiment is to generate all 26 CAN messages at the same time in the very beginning of the experiment and continue to generate the signals according to the periods of the messages afterwards. However, in reality, all of the nodes, which are actually SK-91465X-100MPC, can not start to operate at the same time. Therefore, the message generation times have offset with respect to each other. It should be remembered that although this experiment is held for a purely CAN network, for time measurements purposes as explained before, all of the CAN nodes are also synchronized to each other via FlexRay network. Since all of the CAN messages have the period which is multiple of 5 ms, the cycle length of the FlexRay network is chosen to be 5 ms and all the CAN messages are generated in the beginning of the FlexRay cycles as explained in section 5.2.1. However, since the nodes can not be synchronized to the network at the same time, every time the experiment is run different results are obtained. The solution to this problem is to have all the nodes wait for a time so that all the nodes can be synchronized and after that have all 3 nodes start to operate in the same cycle. This solution is also explained in section 5.2.1 in details. Having taken this measure, the Conventional Scheduling CAN experiment is run with the signal set and the assignments shown above for about 120 seconds. The results obtained for the performance metrics, namely maximum end-to-end delay and jitter, are given in Table 6-5.

**Table 6-5** CAN Conventional Scheduling: End-to-End Delay and Jitter

| Signal | Priority | Delay (μs) | Jitter (μs) | Period (ms) | Jitter (%) |
|---|---|---|---|---|---|
| C1 | 1 | 557 | 73.25 | 10 | 0.73 |
| C2 | 0 | 590 | 137.98 | 5 | 2.76 |
| C3 | 4 | 1159 | 765.68 | 5 | 15.31 |
| C4 | 3 | 831 | 73.93 | 10 | 0.74 |
| C5 | 2 | 880 | 133.61 | 10 | 1.34 |
| C6 | 5 | 1394 | 75.46 | 10 | 0.75 |
| C7 | 6 | 1444 | 289.21 | 10 | 2.89 |
| C8 | 12 | 3444 | 93.42 | 10 | 0.93 |
| C9 | 11 | 3042 | 92.46 | 10 | 0.92 |
| C10 | 10 | 2768 | 91.79 | 10 | 0.92 |
| C11 | 9 | 2499 | 90.93 | 10 | 0.91 |
| C12 | 8 | 2230 | 89.91 | 10 | 0.90 |
| C13 | 7 | 1958 | 88.99 | 10 | 0.89 |
| C14 | 25 | 7581 | 42.46 | 20 | 0.21 |
| C15 | 24 | 7309 | 40.42 | 20 | 0.20 |
| C16 | 23 | 7072 | 38.56 | 20 | 0.19 |
| C17 | 22 | 6803 | 36.70 | 20 | 0.18 |
| C18 | 21 | 6533 | 34.80 | 20 | 0.17 |
| C19 | 20 | 6269 | 34.90 | 20 | 0.17 |
| C20 | 19 | 6000 | 34.84 | 20 | 0.17 |
| C21 | 18 | 5299 | 35.93 | 20 | 0.18 |
| C22 | 17 | 4738 | 33.90 | 20 | 0.17 |
| C23 | 16 | 4182 | 76.36 | 20 | 0.38 |
| C24 | 15 | 4341 | 75.61 | 20 | 0.38 |
| C25 | 14 | 4357 | 727.56 | 20 | 3.64 |
| C26 | 13 | 3710 | 73.14 | 20 | 0.37 |

Also the figures in Figure 6-10 and Figure 6-11 illustrate the results obtained for the Conventional CAN Scheduling with respect to the increasing priority values of the CAN signals. The bars with light blue color in the figures indicate the signals that pass through the Gateway. While in Figure 6-10, the end-to-end delay values that the signals experience are shown in milisecond, in Figure 6-11 the jitter values are shown in microsecond.

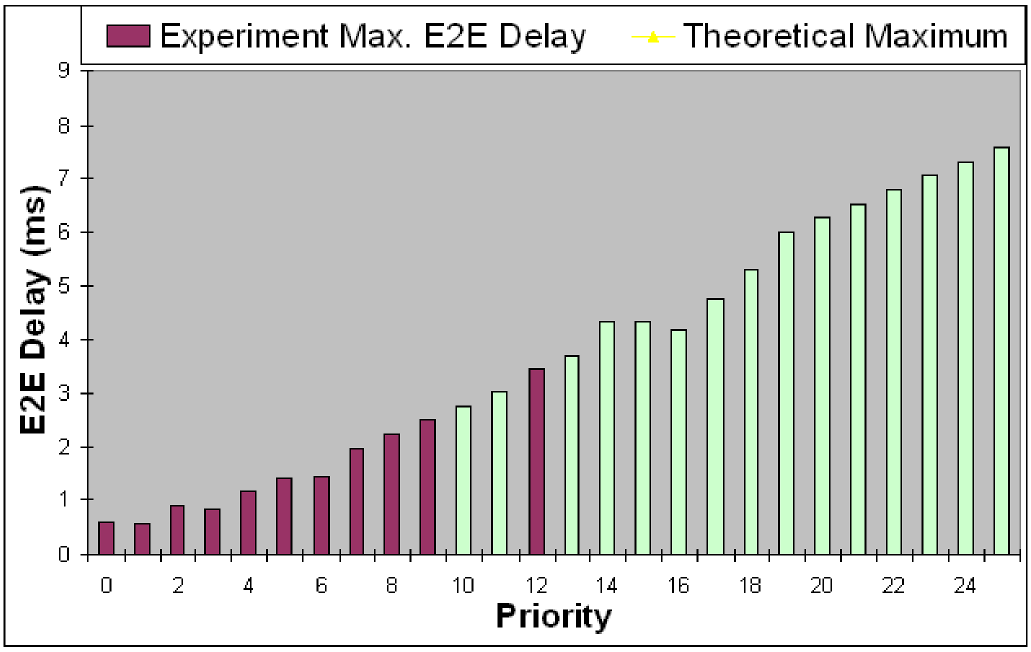


**Figure 6-10** End-to-End Delay vs Priorities: Conventional Scheduling

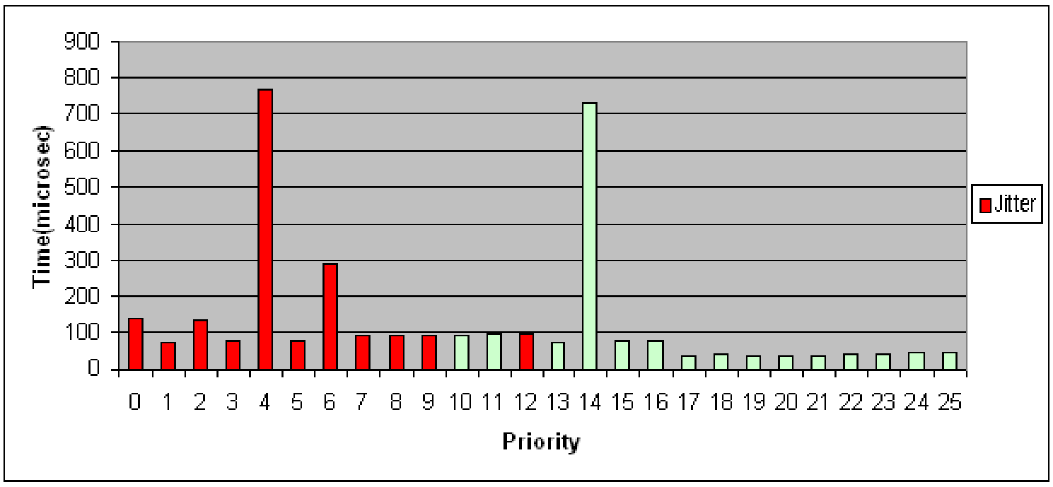


**Figure 6-11** Jitter vs Priorities: Conventional Scheduling

In [30], the theoretical maximum delay that the signals defined in Table 6-2 with the priority assignment in Table 6-3 could experience is also calculated.

Comparative results with respect to this experiment and the theoretical values are given below in Table 6-6 and Figure 6-12 where the light blue bars indicates the signals passing the Gateway. In Figure 6-12, the end-to-end delay values that the signals experience are shown together with the theoretical maximum values which can be observed in the network.

**Table 6-6** End-to-End Delay vs Theoretical Maximum

| Signal | Priority | Delay in Experiment (ms) | Theoretical Max. Delay (ms) |
|---|---|---|---|
| C1 | 1 | 0.557 | 0.96 |
| C2 | 0 | 0.590 | 0.64 |
| C3 | 4 | 1.159 | 1.92 |
| C4 | 3 | 0.831 | 1.6 |
| C5 | 2 | 0.880 | 1.28 |
| C6 | 5 | 1.394 | 2.24 |
| C7 | 6 | 1.444 | 2.5 |
| C8 | 12 | 3.444 | 4.42 |
| C9 | 11 | 3.042 | 4.1 |
| C10 | 10 | 2.768 | 3.78 |
| C11 | 9 | 2.499 | 3.46 |
| C12 | 8 | 2.230 | 3.14 |
| C13 | 7 | 1.958 | 2.82 |
| C14 | 25 | 7.581 | 7.82 |
| C15 | 24 | 7.309 | 7.82 |
| C16 | 23 | 7.072 | 7.5 |
| C17 | 22 | 6.803 | 7.81 |
| C18 | 21 | 6.533 | 6.98 |
| C19 | 20 | 6.269 | 6.66 |
| C20 | 19 | 6.000 | 6.34 |
| C21 | 18 | 5.299 | 6.14 |
| C22 | 17 | 4.738 | 5.82 |
| C23 | 16 | 4.182 | 5.18 |
| C24 | 15 | 4.341 | 4.92 |
| C25 | 14 | 4.357 | 4.8 |
| C26 | 13 | 3.710 | 4.62 |

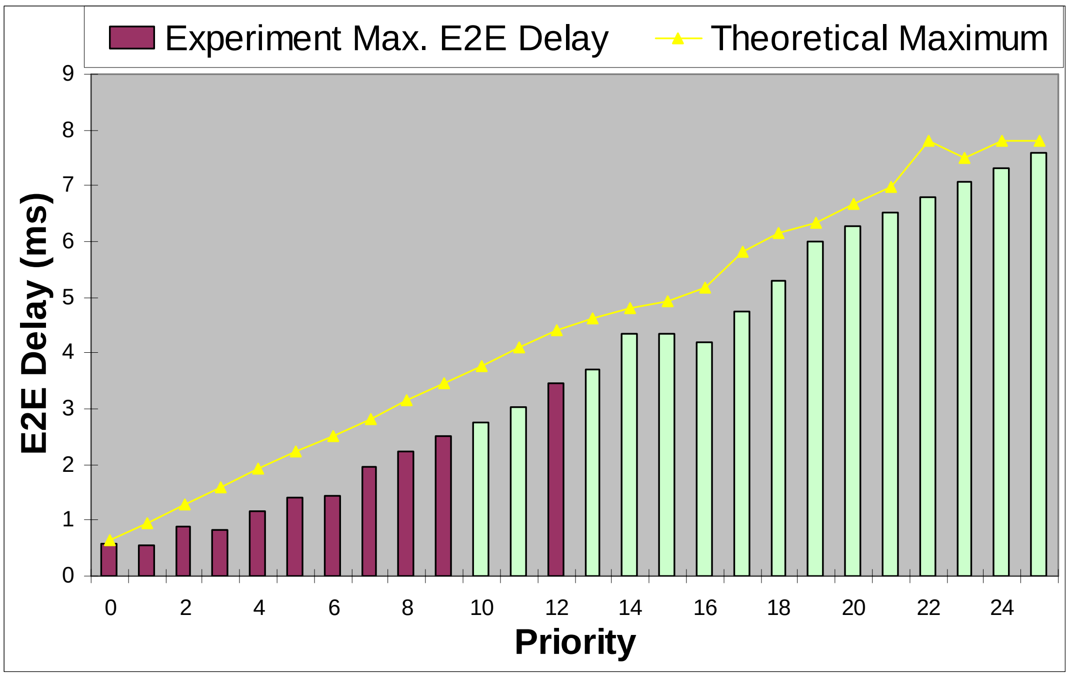


**Figure 6-12** End-to-End Delay and the Theoretical Maximum Values

When Table 6-6 and the Figure 6-12 are observed, it is found out that the end-to-end delay that the CAN signals experience in the experiment stays within the theoretical limits. Note that the theoretical worst case limit computation in [30] assumes that all CAN signals are generated at the same time. However, in the hardware, although the CAN signals are intended to be generated at the same time, it is never possible to make a fine tuning of the generation time of the signals. The possible factors which might affect the generation time of the signals can be the FlexRay network time stability of the nodes, being a slave or master in the network or the volume of the application running on the node.

The following example demonstrates the situation. Consider the messages C20 and C21 which are generated by CAN Node1 and CAN Node2, respectively. Priorities of those messages are given as 19 and 18. Both CAN Node1 and CAN Node2 begin to generate their CAN messages in the beginning of the FlexRay cycle. There are also other signals assigned to CAN Node1 and CAN Node2. Assume that, either because of the reasons mentioned so far or due to the other factors, CAN Node1 achieved to generate C20 whose priority is 19, before CAN Node2 manages it. In such a situation a message with a lower priority goes earlier than a signal of higher priority since the former one produced earlier. Because of these kind of impurities existed in the hardware environment, the end-to-end delay values of the signals may display variation form node to node and signal to signal.

The other performance metric calculated in the experiment is the jitter that the CAN packets experience. In order to understand the jitter behavior of the network, the total of 26 CAN signals exchanged throughout the network should be divided into three set of messages according to their message generation periods. So, in this context, CAN messages C2 and C3 form the first group whose period is 5 ms. C1 and all messages from C4 to C13, inclusive, form the group with 10 ms period. The third group whose period is 20 ms is composed of the messages from C14 to C26, inclusive. Since all of the messages are begun to be generated in the very same cycle at the same time, there happens to be only three scenarios. Either only the first group signals are generated or the second group signals are also generated together with the first group signals in the cycles that are multiples of 2 or finally the third group signals are added on top of the first two signal groups in the cycles that are multiples of 4. Therefore, first group messages sometimes fight only against each other for the medium access. In these situations, the end-to-end delay they experience is the minimum. In the cycles that are multiples of 2, these signals try to grab the medium also against additional second group. So, in those cycles the end-to-end delay they experience is greater than the previous case. As a worst case, in the cycles that are multiples of 2, all CAN signals fight against each other for contention. Obviously, the first group signals experience the maximum delay in the medium. Therefore, those three different delay order causes the first group signals to experience jitter. If Table 6-5 is examined, it is observed that the first group CAN signals, which are C2 and C3, experience fairly big jitter values when compared to other CAN signals. The same logic applies to the second group signals also. While, they, half of the time, fight against the CAN signals of the first group and the second group, in the other half of its time, they try to grab the medium against all of the signals. Therefore, they experience two different delay values in the consecutive generation time which, obviously, means the jitter.

Also, due to the fact that some messages might be ready to be put on the wire before the other opponents due to the differences in the hardware as explained in the previous paragraph, some of the signals may have to wait additionally for the lower priority signals which appear on the bus earlier and this situation causes jitter.

### 6.3.1.2 Prioritized Scheduling CAN Experiment

According to this priority ordering explained in [30], the CAN messages which carry the signals destined to FlexRay network are given higher priorities with respect to the messages which are sent within the CAN network only. The analytical worst case response time calculation for each message with the given priority ensures that the entire CAN message set meets their deadlines. The CAN messages to be sent through the Gateway are C9, C10 and the signals from C14 to C26, both inclusive. In the light of this information, the priority assignment used in this experiment is given in Table 6-7.

**Table 6-7** Priority Assignment Using Prioritized CAN Scheduling

| Signal | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 |
|---|---|---|---|---|---|---|---|---|---|
| Priority | 7 | 6 | 16 | 15 | 14 | 20 | 21 | 25 | 1 |
| Signal | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 |
| Priority | 0 | 24 | 23 | 22 | 19 | 18 | 17 | 13 | 12 |
| Signal | C19 | C20 | C21 | C22 | C23 | C24 | C25 | C26 | |
| Priority | 11 | 10 | 9 | 8 | 5 | 4 | 3 | 2 | |

The message IDs sent by the nodes do not change and are the same as the Table 6-4. The experiment is run for 2 minutes just same as the previous experiment except the configuration changes explained above. The results obtained are given in Table 6-8.

**Table 6-8** CAN Prioritized Scheduling: End-to-End Delay and Jitter

| Signal | Priority | Delay (μs) | Jitter (μs) | Period (ms) | Jitter (%) |
|---|---|---|---|---|---|
| C1 | 7 | 2013 | 946.01 | 10 | 9.46 |
| C2 | 6 | 2328 | 660.50 | 5 | 13.21 |
| C3 | 16 | 4923 | 2778.39 | 5 | 55.57 |
| C4 | 15 | 4020 | 2874.75 | 10 | 28.75 |
| C5 | 14 | 4359 | 2584.48 | 10 | 25.84 |
| C6 | 20 | 6033 | 4311.29 | 10 | 43.11 |
| C7 | 21 | 6950 | 4312.19 | 10 | 43.12 |
| C8 | 25 | 8081 | 4315.65 | 10 | 43.16 |
| C9 | 1 | 576 | 10.25 | 10 | 0.10 |
| C10 | 0 | 553 | 9.15 | 10 | 0.09 |
| C11 | 24 | 7135 | 4314.77 | 10 | 43.15 |
| C12 | 23 | 6863 | 4313.81 | 10 | 43.14 |
| C13 | 22 | 6593 | 4312.93 | 10 | 43.13 |
| C14 | 19 | 5600 | 32.66 | 20 | 0.16 |
| C15 | 18 | 5329 | 30.64 | 20 | 0.15 |
| C16 | 17 | 4261 | 29.48 | 20 | 0.15 |
| C17 | 13 | 3408 | 69.67 | 20 | 0.35 |
| C18 | 12 | 3138 | 68.21 | 20 | 0.34 |
| C19 | 11 | 2869 | 67.67 | 20 | 0.34 |
| C20 | 10 | 2603 | 67.61 | 20 | 0.34 |
| C21 | 9 | 2444 | 67.58 | 20 | 0.34 |
| C22 | 8 | 2176 | 66.08 | 20 | 0.33 |
| C23 | 5 | 1031 | 13.02 | 20 | 0.07 |
| C24 | 4 | 1159 | 116.46 | 20 | 0.58 |
| C25 | 3 | 1463 | 220.13 | 20 | 1.10 |
| C26 | 2 | 820 | 10.36 | 20 | 0.05 |

The results are also illustrated in Figure 6-13 and Figure 6-14 with respect to the increasing priority values of the CAN signals. Similar with the previous experiment, the bars with light blue color shows the signals that pass the Gateway. In Figure 6-13, the end-to-end delay values that the signals experience are shown in milisecond for all of the signals. Similarly, in Figure 6-14, the jitter values that the signals experience are exhibited in microsecond.

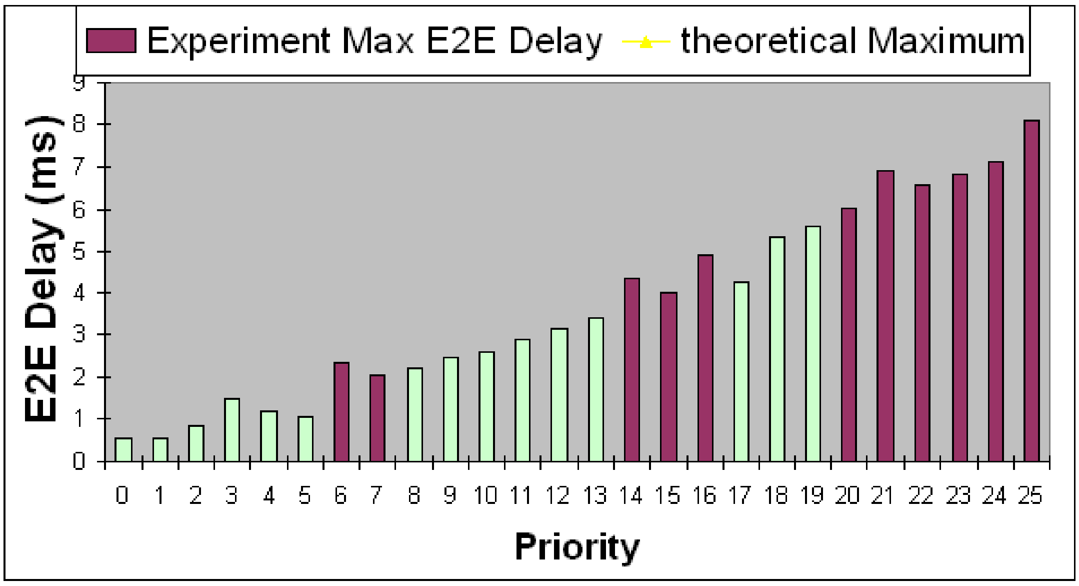


**Figure 6-13** End-to-End Delay vs Priorities: Prioritized Scheduling

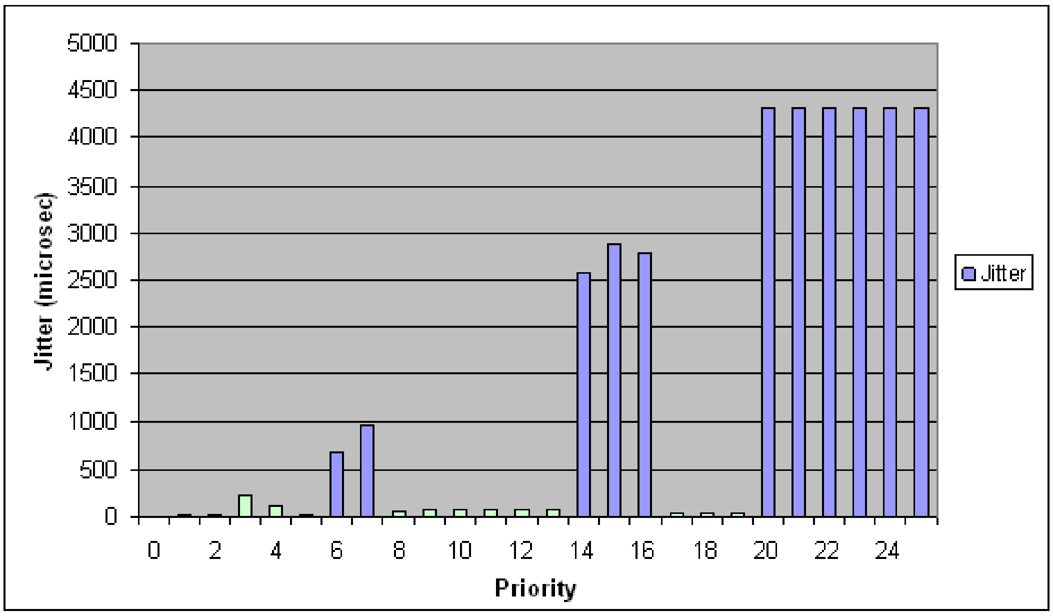


**Figure 6-14** Jitter vs Priorities: Prioritized Scheduling

When these results are compared against the theoretical maximum values computed in [30], the comparative results are demonstrated in Table 6-9 and Figure 6-15. In Figure 6-15, the bars indicate the individual signals and the line draws an envelope for the corresponding signals such that no signals can experience an end-to-end delay beyond this envelope line. The bars of light blue color in Figure 6-15 indicate the signals passing the Gateway.

**Table 6-9** End-to-End Delay vs Theoretical Maximum Values

| Signal | Priority | Delay in Experiment (ms) | Theoretical Max. Delay (ms) |
|---|---|---|---|
| C1 | 7 | 2.013 | 2.36 |
| C2 | 6 | 2.328 | 2.04 |
| C3 | 16 | 4.923 | 5 |
| C4 | 15 | 4.020 | 4.68 |
| C5 | 14 | 4.359 | 4.36 |
| C6 | 20 | 6.033 | 6.6 |
| C7 | 21 | 6.950 | 6.86 |
| C8 | 25 | 8.081 | 7.82 |
| C9 | 1 | 0.576 | 0.96 |
| C10 | 0 | 0.553 | 0.64 |
| C11 | 24 | 7.135 | 7.82 |
| C12 | 23 | 6.863 | 7.5 |
| C13 | 22 | 6.593 | 7.18 |
| C14 | 19 | 5.600 | 6.28 |
| C15 | 18 | 5.329 | 5.96 |
| C16 | 17 | 4.261 | 5.64 |
| C17 | 13 | 3.408 | 4.04 |
| C18 | 12 | 3.138 | 3.84 |
| C19 | 11 | 2.869 | 3.52 |
| C20 | 10 | 2.603 | 3.2 |
| C21 | 9 | 2.444 | 3 |
| C22 | 8 | 2.176 | 2.68 |
| C23 | 5 | 1.031 | 1.64 |
| C24 | 4 | 1.159 | 1.54 |
| C25 | 3 | 1.463 | 1.34 |
| C26 | 2 | 0.820 | 1.16 |

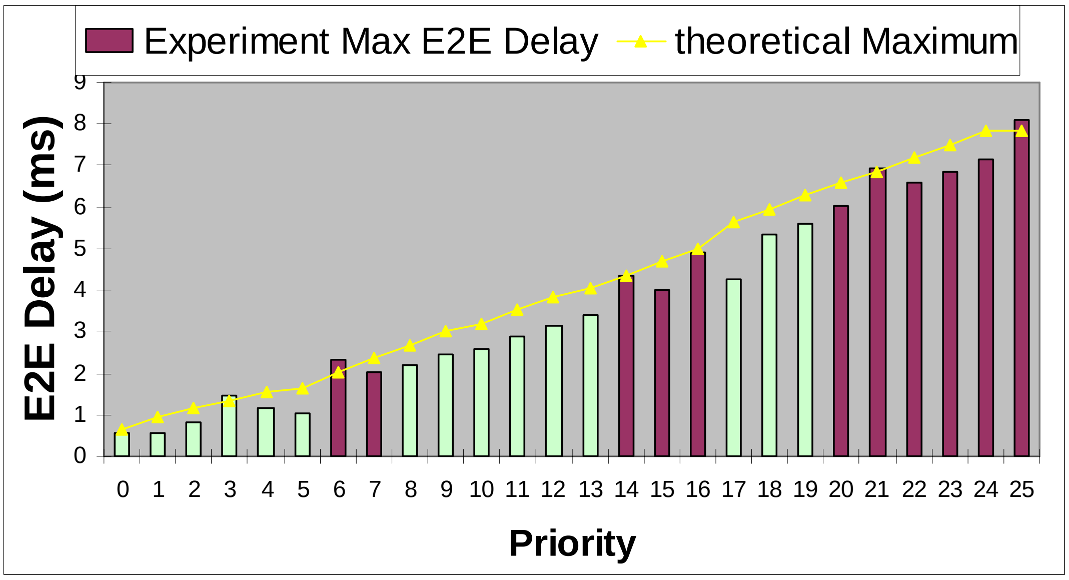


**Figure 6-15** End-to-End Delay and the Theoretical Maximum Values

As Table 6-9 and Figure 6-15 are examined, although generally the end-to-end delay that the CAN signals experience in the experiment stays within the theoretical limits, some CAN packets, which are highlighted in Table 6-9, namely C2, C7, C8 and C25, exceed the theoretical end-to-end delay limit for themselves. This is rooted from the fact that the time measurements taken for the experiments are not exact as discussed in section 6.2.3 in details. It should be noted that the situation in this experiment falls into the second case in section 6.2.3 where one of the timestamps is tagged via the CPU and the other time tag is put in the FlexAlyzer for the delay calculations. If we focus on the magnitude of the errors, the deviations in the end-to-end delay calculations are found to be 288 μs, 90 μs, 251 μs and 123 μs for the C2, C7, C8 and C25, respectively. Referring the details to the section 6.2.3, the deviation of the end-to-end calculations from the actual end-to-end delay can be formulated as follows.

$$Error = TX\_Error + (N-1) \times 25 \tag{6-8}$$

where *N* is the number of the messages in a CAN burst received by the FlexCard up to the CAN packet whose end-to-end delay is calculated. The variable TX Error is defined in section 6.2.3. Although the number of the messages received in a burst before the signal whose end-to-end delay is to be calculated can not be known exactly, since in the worst case all the signals are sent through the CAN bus, it is logical to accept *N* to be the priority of the CAN signal. Still, some messages with lower priority might be received earlier than the higher priority messages since the generation time of the messages can not be exactly fixed and controlled, as discussed previously. In the light of this and the above formula the maximum possible errors are calculated to be 223 μs, 573 μs, 673 μs and 123 μs for the signals *C2*, *C7*, *C8* and *C25*, respectively. According to the results, while for the signals *C7*, *C8* and *C25*, the deviation in the end-to-end delay calculation are explained with the above formula, the deviation that occurs for the signal *C2* can not be accounted for. It should be remembered that there exists an interrupt subroutine whose priority is greater than that of the interrupt subroutine which is responsible for the generation of *C2*. This interrupt subroutine must also be taken into account since there is a certain probability, though small, that this interrupt is issued while *C2* is being generated. If the length of this interrupt subroutine is also considered, the deviation in the end-to-end delay calculation can be explained for the signal *C2*.

The other performance metric of the experiment, which is jitter, is shown in Table 6-8 and Figure 6-14. As the jitter values in Figure 6-14 are examined together with the signal periods that are included in Table 6-8, it is observed that the priorities at which the signals experience significant jitter belong to the signals whose periods are 5 ms and 10 ms. On the other hand, signals with 20 ms period experience fairly no jitter. These results are expected due to the reasons discussed in section 6.3.1.1 in the paragraph where the jitter results are explained.

### 6.3.1.3 CAN Scheduling with Fixed Priority

The scheduling used in this experiment is more convenient for practical purposes. In reality, a network is not established at once at a given time. Rather, new signals are required to be added to the network gradually. By using the CAN Scheduling with Fixed Priorities algorithm it is possible to assign priorities only to the CAN signals which are to be added to the network and keep the priorities of the existing signals as they are. Therefore, this scheduling scheme saves the engineers from the obligation of configuring the CAN network from the beginning every time a new signal is to be added. The algorithm presented in [30] assigns the priorities to the new messages and checks the schedulability of the entire message set.

The priority assignment according to the CAN Scheduling with Fixed Priorities algorithm is given in Table 6-10.

**Table 6-10** Priority Assignment Using CAN Scheduling with Fixed Priorities

| Signal | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 |
|---|---|---|---|---|---|---|---|---|---|
| Priority | 7 | 3 | 10 | 12 | 17 | 28 | 30 | 20 | 1 |
| Signal | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 |
| Priority | 0 | 58 | 22 | 14 | 31 | 29 | 21 | 19 | 18 |
| Signal | C19 | C20 | C21 | C22 | C23 | C24 | C25 | C26 | |
| Priority | 13 | 11 | 9 | 8 | 6 | 5 | 4 | 2 | |

The mapping of the signals to each node does not change with respect to the previous two experiments and given in the Table 6-4. The experiment is run for about 120 seconds and end-to-end delay and jitter values are obtained as the outcome of the experiments. The results are shared in Table 6-11.

Table 6-11 CAN Scheduling with Fixed Priorities: End-to-End Delay and Jitter

| Signal | Priority | Delay (μs) | Jitter (μs) | Period (ms) | Jitter (%) |
|---|---|---|---|---|---|
| C1 | 7 | 2587 | 1120.90 | 10 | 11.21 |
| C2 | 3 | 1461 | 491.49 | 5 | 9.83 |
| C3 | 10 | 3477 | 1738.48 | 5 | 34.77 |
| C4 | 12 | 4024 | 1983.89 | 10 | 19.84 |
| C5 | 17 | 4654 | 1966.85 | 10 | 19.67 |
| C6 | 28 | 6900 | 3738.37 | 10 | 37.38 |
| C7 | 30 | 7532 | 4024.14 | 10 | 40.24 |
| C8 | 20 | 6064 | 3446.74 | 10 | 34.47 |
| C9 | 1 | 793 | 10.11 | 10 | 0.10 |
| C10 | 0 | 480 | 9.62 | 10 | 0.10 |
| C11 | 58 | 7531 | 4312.82 | 10 | 43.13 |
| C12 | 22 | 6069 | 3737.49 | 10 | 37.37 |
| C13 | 14 | 4025 | 2270.66 | 10 | 22.71 |
| C14 | 31 | 7190 | 42.90 | 20 | 0.21 |
| C15 | 29 | 6592 | 39.05 | 20 | 0.20 |
| C16 | 21 | 6332 | 33.20 | 20 | 0.17 |
| C17 | 19 | 5154 | 29.32 | 20 | 0.15 |
| C18 | 18 | 4843 | 27.50 | 20 | 0.14 |
| C19 | 13 | 4241 | 69.09 | 20 | 0.35 |
| C20 | 11 | 3634 | 67.88 | 20 | 0.34 |
| C21 | 9 | 2520 | 66.44 | 20 | 0.33 |
| C22 | 8 | 2213 | 64.82 | 20 | 0.32 |
| C23 | 6 | 1613 | 61.85 | 20 | 0.31 |
| C24 | 5 | 2041 | 305.28 | 20 | 1.53 |
| C25 | 4 | 1733 | 66.00 | 20 | 0.33 |
| C26 | 2 | 999 | 11.66 | 20 | 0.06 |

The results are also illustrated in Figure 6-16 and Figure 6-17 with respect to the increasing priority values of the CAN signals. The signals passing the Gateway are shown with light blue color in the figures. While in Figure 6-16, the end-to-end delay values that the signals experience are shown in milisecond, in Figure 6-17 the jitter values are shown in microsecond.

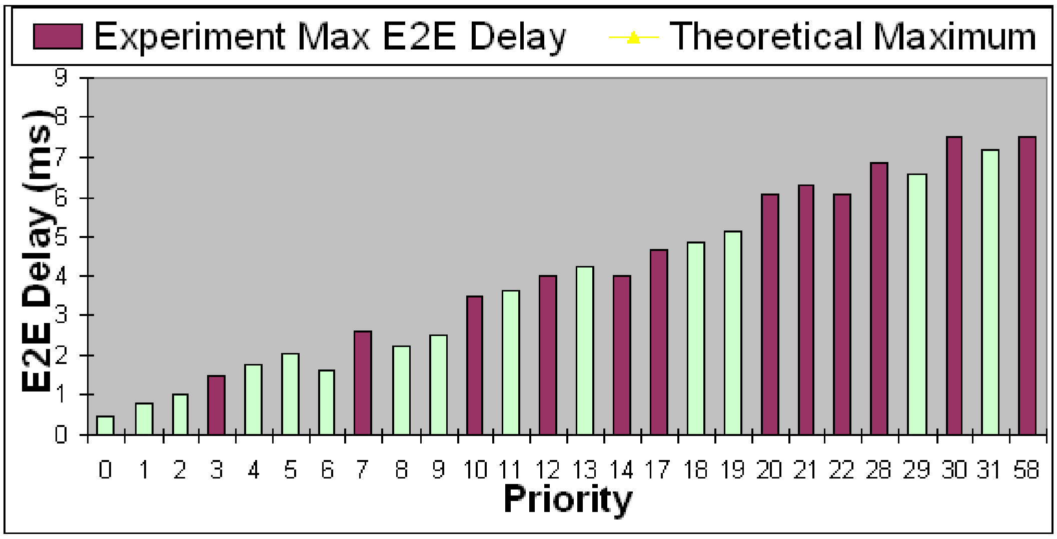


**Figure 6-16** E2E Delay vs Priorities: CAN Scheduling with Fixed Priorities

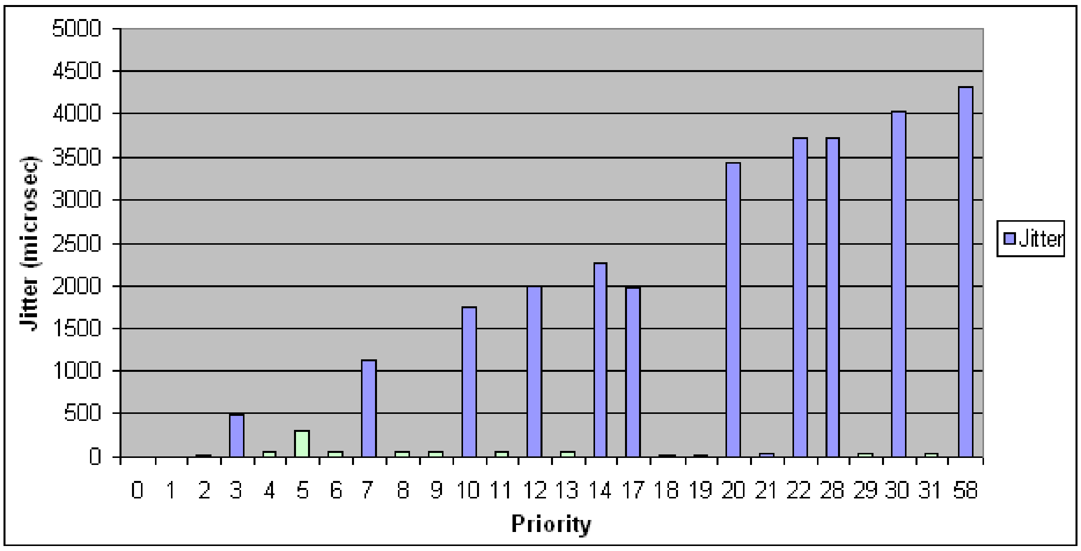


**Figure 6-17** Jitter vs Priorities: CAN Scheduling with Fixed Priorities

As done in the previous two experiments, the results obtained in the experiment are compared against the maximum theoretical values for each of the signals in Table 6-12 and Figure 6-18 where the bars with light blue colour indicate the signals that pass the Gateway. The envelope line given in Figure 6-18 indicates the theoretical maximum values.

**Table 6-12** End-to-End Delay vs Theoretical Maximum

| Signal | Priority | Delay in Experiment (ms) | Theoretical Max. Delay (ms) |
|---|---|---|---|
| C1 | 7 | 2.587 | 2.36 |
| C2 | 3 | 1.461 | 1.44 |
| C3 | 10 | 3.477 | 3.32 |
| C4 | 12 | 4.024 | 3.84 |
| C5 | 17 | 4.654 | 4.8 |
| C6 | 28 | 6.900 | 6.92 |
| C7 | 30 | 7.532 | 7.5 |
| C8 | 20 | 6.064 | 5.96 |
| C9 | 1 | 0.793 | 0.96 |
| C10 | 0 | 0.480 | 0.64 |
| C11 | 58 | 7.531 | 7.82 |
| C12 | 22 | 6.069 | 6.6 |
| C13 | 14 | 4.025 | 4.48 |
| C14 | 31 | 7.190 | 7.82 |
| C15 | 29 | 6.592 | 7.24 |
| C16 | 21 | 6.332 | 6.28 |
| C17 | 19 | 5.154 | 5.64 |
| C18 | 18 | 4.843 | 5.12 |
| C19 | 13 | 4.241 | 4.16 |
| C20 | 11 | 3.634 | 3.52 |
| C21 | 9 | 2.520 | 3 |
| C22 | 8 | 2.213 | 2.68 |
| C23 | 6 | 1.613 | 2.04 |
| C24 | 5 | 2.041 | 1.84 |
| C25 | 4 | 1.733 | 1.66 |
| C26 | 2 | 0.999 | 1.16 |

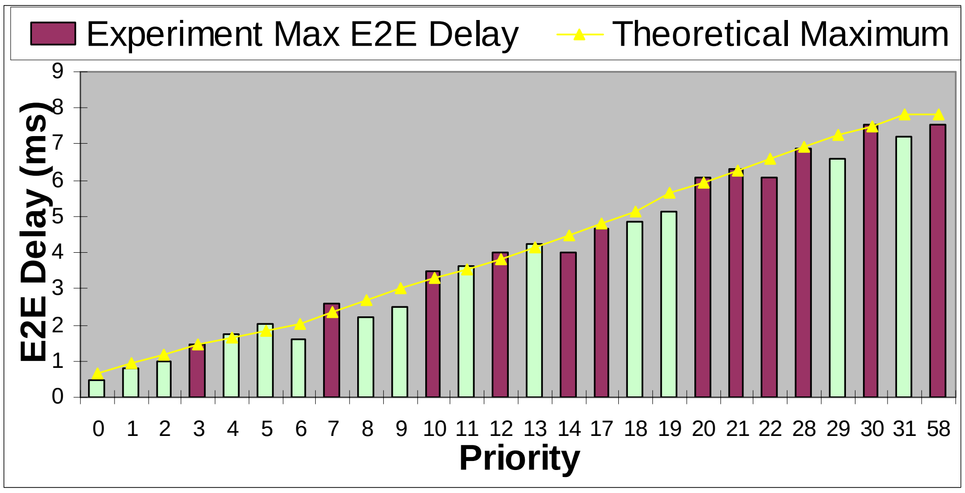


**Figure 6-18** End-to-End Delay and the Theoretical Maximum Values

When the Table 6-12 and Figure 6-18 are observed, it is seen that the delay that some of the CAN signals experience does not stay within the theoretical limits. These CAN messages are highlighted in Table 6-12. Referring the details of the reasons for this deviation to the discussion done for the previous experiment about the end-to-end delay and to the section 6.2.3, we will elaborate only the signals which exceed the theoretical limits with a significant error, by using (6-8). These signals are C1, C4 and C24 with the deviations of 227 μs, 184 μs and 201 μs, respectively. The maximum possible errors expected in the delay calculations are found to be 223 μs, 373 μs and 173 μs using (6-8) for the signals C1, C4 and C24, respectively. Therefore, errors that occur in the end-to-end delay calculation of these signals are accounted by this quantitative analysis except for several microseconds which can be acceptable for such hardware applications. On the other hand, as mentioned above, the delay deviations of the other signals from the theoretical maximum values are not even calculated as they are far small with respect to those 3 signals.

As Table 6-11 and Figure 6-17 are examined it is observed that the jitter values are greater for the messages with the period of 5 ms and 10 ms than the messages whose period are 20 ms. This result is already anticipated according to the discussion done about the jitter behavior of the signals in section 6.3.1.1.

#### 6.3.1.4 Overall Discussion of the CAN Experiment Results

For the preceding CAN experiments, the performance metrics namely, maximum end-to-end delay and jitter are compared against each other for only the CAN signals destined to FlexRay network. The reason for this is that the scheduling schemes that are used in the CAN experiments are created so that the delay values that those signals experience is decreased. In this context, the observations about the results obtained in the CAN experiments can be expressed as follows. It is readily seen from the results that the signals that pass the Gateway experience the biggest maximum end-to-end delay values in CAN Conventional Scheduling Experiment. This result is expected since the priorities are assigned in the Conventional Scheduling such that they increase with the decreasing deadlines of the messages. Therefore, no particular arrangement is done for the signals destined to the FlexRay network. On the other hand the maximum end-to-end delay values that the signals passing the Gateway experience are observed to be the smallest in the CAN Prioritized Scheduling Experiment since in this scheduling scheme, the CAN signals destined to FlexRay network are given higher priorities with respect to the signals which are sent within the CAN network only. For the CAN Scheduling with Fixed Priorities Experiment, the maximum end-to-end delay values that the signals, which cross the Gateway, experience fall in between the two experiments, expectedly. Also, in the CAN Scheduling with Fixed Priorities, the signals which are destined to the FlexRay network are given higher priorities. However, in this scheme, we can not assign the priorities to those signals as we wish. Because, some of the CAN IDs are assumed to be already existing and assigned to certain signals. Therefore, the signals that pass the Gateway are scheduled without using the CAN IDs which are already assigned. Although, due to the reasons explained above, the CAN Scheduling with Fixed Priorities scheme observes greater maximum end-to-end delay than CAN Prioritized Scheduling does, the former scheduling scheme is the most convenient for practical purposes among the three scheduling schemes.

When we examine the jitter that the signals destined to FlexRay network experience, we observe that the values in the CAN Conventional Scheduling are greater than the other two scheduling schemes, while the experienced jitter in these scheduling schemes are almost equal to each other.

These three scheduling schemes are also simulated in the [30] using the very same settings and the messages sets with those of the experiments. When the experiment results are compared against the simulation results, we observe that the experiment results, which are exhibited and discussed in the preceding chapters, agree with those of the simulation in the sense of maximum end-to-end delay and jitter.

### 6.3.2 FlexRay Static Segment Experiments

FlexRay network composes the one part of the bigger inter-connected network where the other part is the CAN network. As discussed in section 6.3.1, the delay and the jitter values experienced by the signals in solely FlexRay network directly affect the end-to-end delay and the jitter experienced by the signals in the inter-connected network. Therefore, by studying the performance of the FlexRay network according to different scheduling schemes, we will have a strong grounding about the behavior of the overall network where both CAN and FlexRay networks and the Gateway exist.

The experiments held in this chapter are FlexRay FID Allocation Without Jitter and FlexRay FID Allocation With Minimum FID. The network configurations are common to both experiments and shown in Table 6-13.

**Table 6-13** Network Configurations for the FlexRay Static Slot Experiments

| Network Parameter | Value |
|---|---|
| FlexRay Cycle Length | 5 ms |
| Static Slot Length | 31 μs |
| Static Slot Number | 64 |
| Dynamic Segment Length | 0 |

The discussions about these experiments are done in the following sections.

### 6.3.2.1 FlexRay FID Allocation Without Jitter

This experiment is held on pure FlexRay network which is composed of 3 distinct nodes, Fujitsu SK-91465X-100MPC. The aim of the experiment is to measure the end-to-end delay and the jitter that the FlexRay messages experience when the FIDs of the FlexRay messages are allocated so that the messages are delivered without any jitter. The message set used in the experiment are comprised of 41 messages which are used by an automotive company in real applications. The messages used in this experiment are given in Table 6-14. FID allocation, which is demonstrated in Table 6-15, for these signals is taken from [30] where the messages are scheduled so that they experience no jitter. In Table 6-15, the messages of node 1, 2 and 3 are characterized by a white, light gray and dark gray background, respectively. It can be observed from the table that each message is scheduled with the largest possible repetition that is a divisor of the message period. Hence, the smallest possible FID count without introducing jitter, which is the optimal solution, is achieved. The formal discussion for the scheduling scheme is given in [30].

Note, also, that the signals that are used in this experiment are sent in the FID values with an offset of 16 slots with respect to the FID values mentioned in [30]. The reason for this is that all of the FlexRay messages are started to be generated in the beginning of the cycle. Since the process of the message generation, which includes the storing of the messages to the applicable FlexRay buffers, takes certain time, the FlexRay messages with low FID can not be sent for sure in the very cycle that they are generated. From the FID allocation in [30], it is seen that all of the messages are to be transmitted within the first 16 FlexRay slots. Although this may cause no difficulty for simulation purposes, it means that almost all of the messages are to be sent in the next cycle in the FlexRay bus, if they are generated in real hardware. Because of this reason the FID allocation for the FlexRay signals are shifted by 16 static slots so that each node is able to prepare all of its messages before their slot time has passed.

**Table 6-14** FlexRay Message Set

| Signal | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 |
|---|---|---|---|---|---|---|---|---|---|
| Period/ms | 10 | 5 | 20 | 10 | 10 | 10 | 10 | 10 | 10 |
| Node | 2 | 2 | 2 | 2 | 1 | 2 | 1 | 1 | 2 |
| Signal | P10 | P11 | P12 | P13 | P14 | P15 | P16 | P17 | P18 |
| Period/ms | 10 | 10 | 20 | 10 | 20 | 10 | 10 | 10 | 10 |
| Node | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 |
| Signal | P19 | P20 | P21 | P22 | P23 | P24 | P25 | P26 | P27 |
| Period/ms | 100 | 50 | 100 | 100 | 100 | 250 | 500 | 250 | 10 |
| Node | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 1 | 2 |
| Signal | P28 | P29 | P30 | P31 | P32 | P33 | P34 | P35 | P36 |
| Period/ms | 100 | 100 | 100 | 2000 | 2000 | 1000 | 1000 | 20 | 2000 |
| Node | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 |
| Signal | P37 | P38 | P39 | P40 | P41 | | | | |
| Period/ms | 2000 | 2000 | 2000 | 2000 | 100 | | | | |
| Node | 1 | 2 | 2 | 3 | 1 | | | | |

**Table 6-15** FlexRay FID Allocation Without Jitter

| Signal | P1 | P2 | P3 | P4 | P5 | P6 | P7 |
|---|---|---|---|---|---|---|---|
| Period/ms | 10 | 5 | 20 | 10 | 10 | 10 | 10 |
| FID | 24 | 23 | 28 | 24 | 22 | 25 | 17 |
| Repetition/ms | 10 | 5 | 20 | 10 | 10 | 10 | 10 |
| Offset | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| Signal | P8 | P9 | P10 | P11 | P12 | P13 | P14 |
| Period/ms | 10 | 10 | 10 | 10 | 20 | 10 | 20 |
| FID | 17 | 25 | 26 | 26 | 28 | 27 | 29 |
| Repetition/ms | 10 | 10 | 10 | 10 | 20 | 10 | 20 |
| Offset | 1 | 1 | 0 | 1 | 3 | 0 | 0 |
| Signal | P15 | P16 | P17 | P18 | P19 | P20 | P21 |
| Period/ms | 10 | 10 | 10 | 10 | 100 | 50 | 100 |
| FID | 27 | 18 | 18 | 19 | 20 | 19 | 29 |
| Repetition/ms | 10 | 10 | 10 | 10 | 20 | 10 | 20 |
| Offset | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| Signal | P22 | P23 | P24 | P25 | P26 | P27 | P28 |
| Period/ms | 100 | 100 | 250 | 500 | 250 | 10 | 100 |
| FID | 29 | 29 | 31 | 31 | 21 | 28 | 20 |
| Repetition/ms | 20 | 20 | 10 | 20 | 10 | 10 | 20 |
| Offset | 2 | 3 | 0 | 3 | 0 | 0 | 1 |
| Signal | P29 | P30 | P31 | P32 | P33 | P34 | P35 |
| Period/ms | 100 | 100 | 2000 | 2000 | 1000 | 1000 | 20 |
| FID | 20 | 20 | 22 | 22 | 21 | 21 | 31 |
| Repetition/ms | 20 | 20 | 80 | 80 | 40 | 40 | 20 |
| Offset | 2 | 3 | 1 | 3 | 3 | 7 | 1 |
| Signal | P36 | P37 | P38 | P39 | P40 | P41 | |
| Period/ms | 2000 | 2000 | 2000 | 2000 | 2000 | 100 | |
| FID | 22 | 22 | 30 | 30 | 32 | 21 | |
| Repetition/ms | 80 | 80 | 80 | 80 | 80 | 20 | |
| Offset | 5 | 7 | 0 | 1 | 0 | 1 | |

According to the message set and the scheduling shown in Table 6-14 and Table 6-15, respectively, at each node, the FlexRay signals are generated in the beginning of the FlexRay cycles as explained in section 5.2.1. The experiment is run for 2 minutes and the network traffic is logged via FlexAlyzer so as to obtain the performance of the FlexRay network with respect to the performance metrics, end-to-end delay and the jitter. Below, the results obtained for the end-to-end delay and the jitter are given in Table 6-16.

**Table 6-16** FID Allocation Without Jitter: End-to-End Delay and Jitter

| Signal | FID | Delay (µs) | Jitter (µs) | Period (ms) | Jitter (%) |
|---|---|---|---|---|---|
| P1 | 24 | 634 | 1.21 | 10 | 0.01 |
| P2 | 23 | 633 | 0.65 | 5 | 0.01 |
| P3 | 28 | 5516 | 2.43 | 20 | 0.01 |
| P4 | 24 | 5607 | 1.21 | 10 | 0.01 |
| P5 | 22 | 458 | 1.21 | 10 | 0.01 |
| P6 | 25 | 611 | 1.21 | 10 | 0.01 |
| P7 | 17 | 440 | 1.21 | 10 | 0.01 |
| P8 | 17 | 5413 | 1.21 | 10 | 0.01 |
| P9 | 25 | 5585 | 1.21 | 10 | 0.01 |
| P10 | 26 | 589 | 1.21 | 10 | 0.01 |
| P11 | 26 | 5562 | 1.21 | 10 | 0.01 |
| P12 | 28 | 15490 | 2.43 | 20 | 0.01 |
| P13 | 27 | 567 | 1.21 | 10 | 0.01 |
| P14 | 29 | 494 | 2.43 | 20 | 0.01 |
| P15 | 27 | 5540 | 1.21 | 10 | 0.01 |
| P16 | 18 | 417 | 1.21 | 10 | 0.01 |
| P17 | 18 | 5391 | 1.21 | 10 | 0.01 |
| P18 | 19 | 395 | 1.21 | 10 | 0.01 |
| P19 | 20 | 370 | 12.14 | 100 | 0.01 |
| P20 | 19 | 5368 | 6.07 | 50 | 0.01 |
| P21 | 29 | 5466 | 12.14 | 100 | 0.01 |
| P22 | 29 | 10439 | 12.14 | 100 | 0.01 |
| P23 | 29 | 15412 | 12.14 | 100 | 0.01 |
| P24 | 31 | 911 | 30.34 | 250 | 0.01 |
| P25 | 31 | 15857 | 60.69 | 500 | 0.01 |
| P26 | 21 | 401 | 30.35 | 250 | 0.01 |
| P27 | 28 | 544 | 1.21 | 10 | 0.01 |
| P28 | 20 | 5343 | 12.14 | 100 | 0.01 |
| P29 | 20 | 10316 | 12.14 | 100 | 0.01 |
| P30 | 20 | 15290 | 12.14 | 100 | 0.01 |
| P31 | 22 | 5187 | 242.75 | 2000 | 0.01 |
| P32 | 22 | 15160 | 242.76 | 2000 | 0.01 |
| P33 | 21 | 15238 | 121.38 | 1000 | 0.01 |
| P34 | 21 | 35212 | 121.38 | 1000 | 0.01 |

**Table 6-16 (Continued)**

| | | | | | |
|---|---|---|---|---|---|
| P35 | 31 | 5911 | 2.43 | 20 | 0.01 |
| P36 | 22 | 25134 | 242.75 | 2000 | 0.01 |
| P37 | 22 | 35107 | 242.75 | 2000 | 0.01 |
| P38 | 30 | 416 | 242.76 | 2000 | 0.01 |
| P39 | 30 | 5389 | 242.75 | 2000 | 0.01 |
| P40 | 32 | 859 | 242.76 | 2000 | 0.01 |
| P41 | 21 | 5294 | 12.14 | 100 | 0.01 |

Also the figures in Figure 6-19 and Figure 6-20 illustrate the results obtained for the FID Allocation Without Jitter with respect to the increasing message IDs of the FlexRay messages. While in the figures, in x-axis, the FIDs of the signals which are used in the experiment are shown, in y-axis the values obtained in the experiment are exhibited.

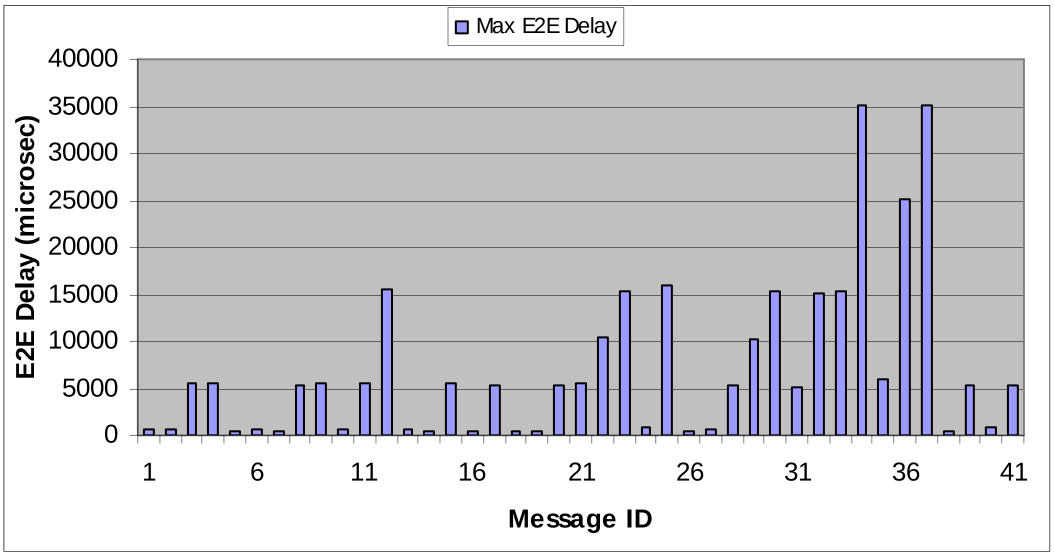


**Figure 6-19** End-to-End Delay vs Message ID: FID Allocation Without Jitter

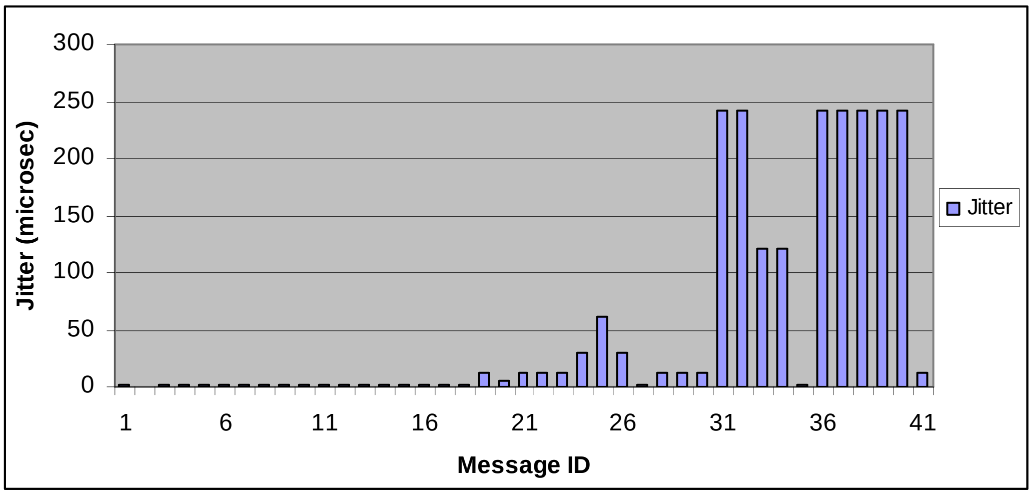


**Figure 6-20** Jitter vs Message ID: FID Allocation Without Jitter

In order to check if the experiment results exhibited in Table 6-16, Figure 6-19 and Figure 6-20 are correct, we can compute the maximum end-to-end delay and the jitter values that the signals experience. Since the FlexRay signals are only exchanged in their dedicated time slots and repetition and offset values are known for the entire signal set, it is easy to figure out the delay and the jitter values that the signals experience without doing complex algebra. We can understand the amount of delay that the signals experience through an example more clearly for the signal set and the scheduling scheme given in Table 6-14 and Table 6-15. Consider the signal *P12*. This signal is generated with a period of 20 ms, i.e. 4 cycles, and the repetition and the offset values for the signal are 20 ms and 3 cycles, respectively. Since all of the signals in Table 6-15 are begun to be produced simultaneously from the 0th cycle, it is obvious that *P12* is generated in the cycles that are multiples of 4, i.e. 0, 4, 8 and so on. On the other hand, the signal is scheduled to be exchanged through the network in the cycles which are equal to three in modulo 4, i.e. 3, 7, 11 and so forth. Therefore, it is readily observed that *P12* experiences a delay of 3 cycles which is equal to 15 ms. When this computed value is compared with the maximum end-to-end delay that *P12* experiences in the experiment and shown in Table 6-16, it is observed that the delay value obtained in the experiment, which is

15.49 ms, agrees with this computed value. The additional fraction value which exists in the experiment result is due to the generation time of *P12* in the hardware which occurs earlier than its dedicated FlexRay slot with the amount of this additional fraction. To sum up, the computation of the end-to-end delay values can be generalized as follows. As the signals are scheduled with the repetition values which are a divisor of the signals' period, all of the signals are transmitted in one of their assigned slots except the offset values. Therefore, the signals experience a delay which is equal to their schedule offset multiplied by the cycle length in milisecond. When the Table 6-16 is examined anew, the maximum end-to-end delay values that the signals experience are observed to be equal to their offset values in milisecond.

On the other hand, the jitter is defined, in short, as the deviation from the periodicity. It is computed above that all the signals are transmitted through the network in their time slots which appear later than the generation time with the amount which exactly equals to the offset count. Therefore, the transmit times of the signals are also strictly periodic. As a result, we conclude that in this scheduling scheme signals are to experience no jitter. This conclusion is verified with the experiment results demonstrated in Table 6-16 where the experienced jitter values are found to be 0.01% in percentage. These very small but non-zero jitter values might be experienced due to the instability of the clocks of the nodes which cause small fluctuations in the generation time of the signals.

#### 6.3.2.2 FlexRay FID Allocation With Minimum FID

In this experiment the impact of the FID assignment, which will provide minimum number of static slot allocation, to the performance metrics is examined. All other configurations and arrangements including the FlexRay message set are same as the previous FlexRay static segment experiment. As its name implies the aim in this FID allocation scheme is to assign all of the messages to the minimum number of time slots by properly selecting the scheduling parameters. In this scheduling, as opposed to the previous one, the rule of choosing the signal repetition values that are a divisor of the signal period is broken in favor of the minimum bandwidth usage. Consequently, by using this scheduling scheme, while some of the signals in Table 6-14 are allowed to experience jitter, the number of the allocated FIDs is decreased from 16 to 12 when compared to FlexRay FID Allocation Without Jitter. The details of the scheduling algorithm are discussed in [30].

The FID assignment used for these FlexRay messages, as it was the case in the previous experiment, are taken from [30] except that all the FID values are used with an offset of 16 slots. The reason for the offset of 16 time slots is discussed in the previous experiment. The FlexRay FID Allocation with Minimum FID is given in Table 6-17. Similar to the experiment, FID Allocation Without Jitter, the messages of node 1, 2 and 3 are also characterized by a white, light gray and dark gray background, respectively in this experiment.

**Table 6-17** FlexRay FID Allocation With Minimum FID

| Signal | P1 | P2 | P3 | P4 | P5 | P6 | P7 |
|---|---|---|---|---|---|---|---|
| Period/ms | 10 | 5 | 20 | 10 | 10 | 10 | 10 |
| FID | 22 | 21 | 26 | 23 | 17 | 22 | 17 |
| Repetition/ms | 2 | 1 | 4 | 2 | 2 | 2 | 2 |
| Offset | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| Signal | P8 | P9 | P10 | P11 | P12 | P13 | P14 |
| Period/ms | 10 | 10 | 10 | 10 | 20 | 10 | 20 |
| FID | 18 | 23 | 24 | 24 | 26 | 25 | 27 |
| Repetition/ms | 2 | 2 | 2 | 2 | 4 | 2 | 4 |
| Offset | 0 | 1 | 0 | 1 | 3 | 0 | 0 |
| Signal | P15 | P16 | P17 | P18 | P19 | P20 | P21 |
| Period/ms | 10 | 10 | 10 | 10 | 100 | 50 | 100 |
| FID | 25 | 18 | 19 | 19 | 20 | 20 | 27 |
| Repetition/ms | 2 | 2 | 2 | 2 | 16 | 8 | 16 |
| Offset | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| Signal | P22 | P23 | P24 | P25 | P26 | P27 | P28 |
| Period/ms | 100 | 100 | 250 | 500 | 250 | 10 | 100 |
| FID | 27 | 17 | 28 | 28 | 20 | 26 | 20 |
| Repetition/ms | 16 | 16 | 32 | 64 | 32 | 2 | 16 |
| Offset | 2 | 3 | 1 | 2 | 6 | 0 | 2 |
| Signal | P29 | P30 | P31 | P32 | P33 | P34 | P35 |
| Period/ms | 100 | 100 | 2000 | 2000 | 1000 | 1000 | 20 |
| FID | 20 | 20 | 20 | 20 | 20 | 20 | 28 |
| Repetition/ms | 16 | 16 | 64 | 64 | 64 | 64 | 4 |
| Offset | 3 | 4 | 10 | 11 | 7 | 9 | 0 |
| Signal | P36 | P37 | P38 | P39 | P40 | P41 | |
| Period/ms | 2000 | 2000 | 2000 | 2000 | 2000 | 100 | |
| FID | 20 | 20 | 27 | 27 | 28 | 20 | |
| Repetition/ms | 64 | 64 | 64 | 64 | 64 | 16 | |
| Offset | 12 | 13 | 5 | 7 | 3 | 5 | |

The experiment is run for about 120 seconds using the FlexRay FID Allocation With Minimum FID. The network traffic is logged via FlexAlyzer and the results obtained by parsing the log files are exhibited in Table 6-18.

**Table 6-18** FID Allocation With Minimum FID: End-to-End Delay and Jitter

| Signal | FID | Delay (ms) | Jitter (ms) | Period (ms) | Jitter (%) |
|---|---|---|---|---|---|
| P1 | 22 | 0.45 | 0.001 | 10 | 0.012 |
| P2 | 21 | 0.45 | 0.001 | 5 | 0.013 |
| P3 | 26 | 5.33 | 0.002 | 20 | 0.012 |
| P4 | 23 | 0.43 | 0.001 | 10 | 0.012 |
| P5 | 17 | 0.32 | 0.001 | 10 | 0.012 |
| P6 | 22 | 5.42 | 0.001 | 10 | 0.012 |
| P7 | 17 | 5.29 | 0.001 | 10 | 0.012 |
| P8 | 18 | 0.29 | 0.001 | 10 | 0.012 |
| P9 | 23 | 5.40 | 0.001 | 10 | 0.012 |
| P10 | 24 | 0.40 | 0.001 | 10 | 0.012 |
| P11 | 24 | 5.38 | 0.001 | 10 | 0.012 |
| P12 | 26 | 15.30 | 0.002 | 20 | 0.012 |
| P13 | 25 | 0.38 | 0.001 | 10 | 0.012 |
| P14 | 27 | 0.31 | 0.002 | 20 | 0.012 |
| P15 | 25 | 5.35 | 0.001 | 10 | 0.012 |
| P16 | 18 | 5.27 | 0.001 | 10 | 0.012 |
| P17 | 19 | 0.27 | 0.001 | 10 | 0.012 |
| P18 | 19 | 5.24 | 0.001 | 10 | 0.012 |
| P19 | 20 | 65.22 | 30.010 | 100 | 30.010 |
| P20 | 20 | 30.25 | 15.001 | 50 | 30.002 |
| P21 | 27 | 65.28 | 30.010 | 100 | 30.010 |
| P22 | 27 | 70.25 | 30.010 | 100 | 30.010 |
| P23 | 17 | 75.23 | 30.010 | 100 | 30.010 |
| P24 | 28 | 155.69 | 78.751 | 250 | 31.500 |
| P25 | 28 | 310.63 | 157.501 | 500 | 31.500 |
| P26 | 20 | 150.08 | 78.755 | 250 | 31.502 |
| P27 | 26 | 0.36 | 0.001 | 10 | 0.012 |
| P28 | 20 | 70.19 | 30.010 | 100 | 30.010 |
| P29 | 20 | 75.17 | 30.010 | 100 | 30.010 |
| P30 | 20 | 60.14 | 30.010 | 100 | 30.010 |
| P31 | 20 | 289.95 | 121.315 | 2000 | 6.066 |
| P32 | 20 | 294.92 | 121.315 | 2000 | 6.066 |
| P33 | 20 | 315.01 | 70.758 | 1000 | 7.076 |
| P34 | 20 | 284.98 | 70.758 | 1000 | 7.076 |
| P35 | 28 | 0.69 | 0.002 | 20 | 0.012 |
| P36 | 20 | 299.90 | 121.315 | 2000 | 6.066 |
| P37 | 20 | 304.87 | 121.315 | 2000 | 6.066 |
| P38 | 27 | 265.20 | 121.315 | 2000 | 6.066 |
| P39 | 27 | 275.17 | 121.315 | 2000 | 6.066 |
| P40 | 28 | 255.61 | 121.315 | 2000 | 6.066 |
| P41 | 20 | 65.11 | 30.010 | 100 | 30.010 |

Also the figures in Figure 6-21 and Figure 6-22 illustrate the results obtained for the FID Allocation Without Minimum FID with respect to the increasing message IDs of the FlexRay messages. The x-axis of the figures indicates the FID assignment of the experiment, while in y-axis, the results obtained in the experiment are exhibited.

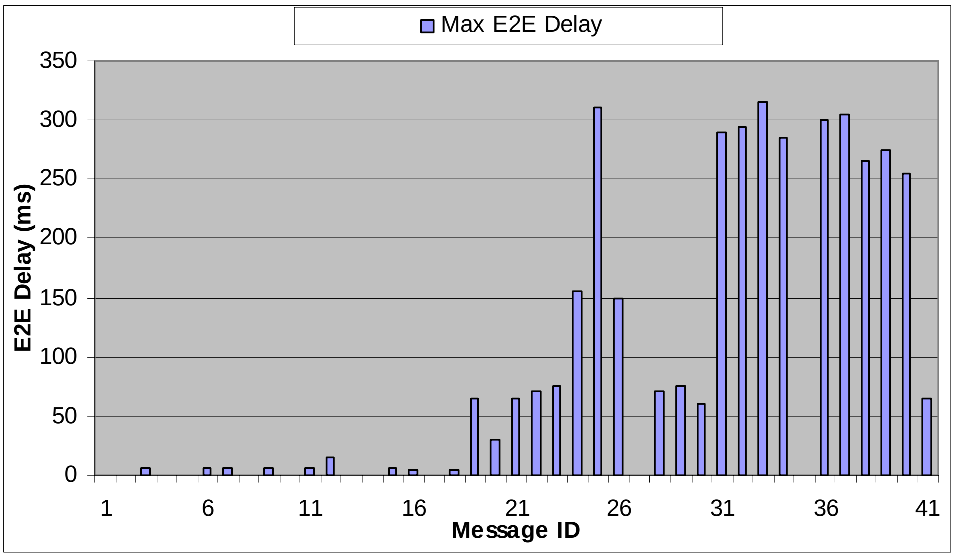


**Figure 6-21** E2E Delay vs Message ID: FID Allocation With Minimum FID

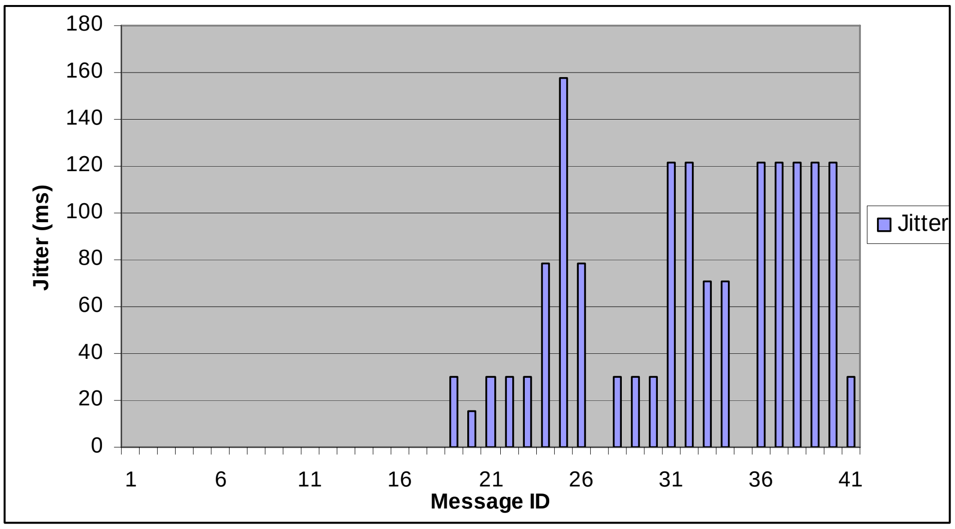


**Figure 6-22** Jitter vs Message ID: FID Allocation With Minimum FID

The results demonstrated in Table 6-18 can also be theoretically computed and the behaviours of the performance metrics can be better explained by means of examples.

To begin with the worst case end-to-end delay, consider, the FlexRay signal *P26*. The period, repetition and the offset values of the signal, *P26*, are 250, 32 and 6, respectively, as shown in Table 6-17. This means that the signal is generated at every 250 ms and it is transmitted either in the cycle 6 or 38 as there exist 64 cycles totally. The generation time and the transmit time of P26 in terms of the FlexRay cycle number are illustrated in Figure 6-23 beginning from the 0th cycle.

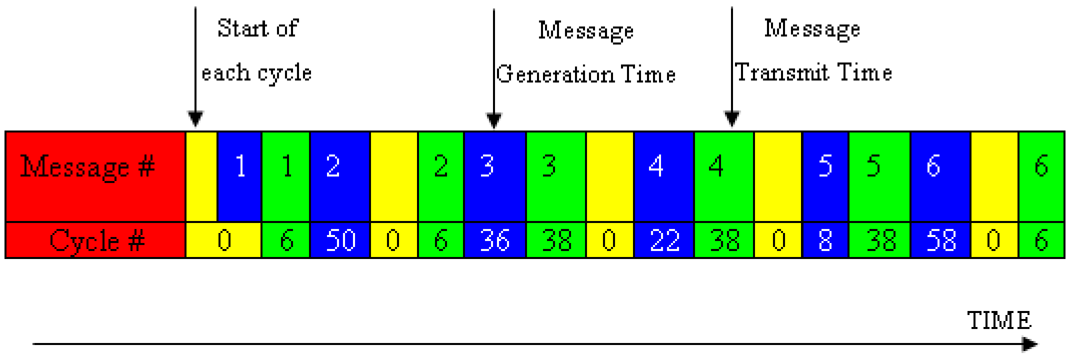


**Figure 6-23** Illustration of End-to-End Delay of P26

As seen from the figure while the cells with blue shading indicates the message generation time of the signal in terms of the cycle number, the green shaded cells show the transmit time of the messages. Moreover, numbers in the upper row of the figure indicate the number of the message and the lower row numbers show the number of the cycle where the generation or the transmit of the signal takes place. So, in Figure 6-23, for example, the second message is depicted to be generated in 50th cycle. Since the repetition value of *P26* is 32, this means that the message *P26* can not be transmitted at all cycles through the FlexRay bus but rather the transmit resolution of the signal is 32 cycles. It is already mentioned that the transmit cycles for *P26* are either cycle number 6 or cycle number 38. As a consequence, the second message, which is generated in 50th cycle can hardly be sent in 6th cycle in its time slot experiencing a delay of 100 ms. Similarly, when all the consecutive messages of *P26* are examined this way, it is calculated that the maximum end-to-end delay that the message *P26* can experience with these settings, is 30 cycles which corresponds to 150 ms delay. This theoretical approach is also verified with this experiment such that the corresponding maximum end-to-end delay is found to be 150.08 ms in Table 6-18 for *P26*. It is possible to apply this example to all 41 signals to determine the maximum delay values that each signal experiences and verify the experiment theoretically.

The Figure 6-23 can also be used to explain the jitter behavior of the signals which are exchanged in this experiment. When we examine the jitter that *P26* experiences, we find out that the delay between the receive times of the consecutive P26 signals are 320 ms, 160 ms, 320 ms, 320 ms, 160 ms and so forth. In fact, this scheme explains the reason for the jitter. In order not to experience jitter, the receive time of the consecutive *P26* signals must, all, be 250 ms apart which means 50 cycles. This could be achieved if the repetition value of the signal was 50 or one of the divisors of 50. However, 32 is not the divisor of 50 and expectedly the signal *P26* experiences jitter. According to the above explanations the amount of jitter that *P26* experiences is found to be 76.7 ms theoretically. When this value is compared with the jitter value that *P26* experiences in Table 6-18, it is observed that the jitter value obtained in the experiment is 78.76 ms and agrees with the value computed theoretically. In a similar manner with this example, all the jitter values that are obtained in the experiment are verified with those which are computed theoretically and the results obtained in the experiment are observed to be in agreement with the theoretically computed jitter values.

### 6.3.3 FlexRay Dynamic Segment Experiments

In this section, the impact of the length of the FlexRay dynamic segment to the end-to-end performance of the interconnected FlexRay and CAN networks is studied.

The length of the dynamic segment directly affects the schedulability of the messages and the delay that the sporadic messages experience. As described in section 2.1.2.2, the dynamic segment is composed of minislots. All of the sporadic signals are granted an FID number at which they can be transmitted. If the sporadic signal is not ready to go at the time of its FID, only one minislot time passes, FID counter is increased by one and the other signal to which the next FID is assigned is checked to be sent. On the other hand, when the slot time comes for a particular signal that is ready to be sent, the content of the signal is transmitted. In this case, depending on the length of that particular signal, some number of minislots is consumed by the signal leaving less number of minislots for the remaining signals and again FID counter is only increased by one. As a result of this scheme, if the length of the dynamic segment is not big enough, a sporadic signal, to which a greater FID is assigned (low priority), may suffer for a long time before being transmitted or even may not be able to be sent at all. In order to avoid such situations the length of the dynamic segment should be chosen properly together with the FID assignments of the sporadic messages.

Together with the length of the dynamic segment, the FID assignments for the sporadic signals are also important. In order to assign FID values to the signals, a worst-case delay analysis is done. For a signal, *S*, to experience a worst case delay, *S* should arrive right after its dedicated time slot. The delay component from this arrival time to the end of the cycle composes the one part of the total delay and is called as *initial delay*. Next, a linear integer programming (LIP) problem is formulated that tries to fill the dynamic segment of the following *N* FlexRay cycles with signals that have a smaller FID than *S*. If this is achieved, then the worst-case response time of *S* is larger than the sum of the *initial delay* and the time for the *N* FlexRay cycles. Moreover, the same analysis has to be carried out for *(N+1)* FlexRay cycles. On the other hand, if *N* FlexRay cycles can not be filled with the signals that have a smaller FID, the worst-case response time of *S* can be computed from the longest delay *S* experiences within *N* FlexRay Cycles. Using this algorithm, the schedulability analysis is done for a particular message set and FIDs are assigned to each signal so that they experience a worst case delay that is smaller than their deadlines. When this algorithm is applied for the message set given in Table 6-20, starting from the dynamic segment length of 8 minislots, the algorithm determines that the message set is schedulable for the dynamic segment length of, at least, 19 minislots. The algorithm is discussed in more detail in [30].

We examine the effect of changing the length of the dynamic segment to the schedulability by conducting three dynamic segment experiments where the dynamic segment lengths are chosen to be 18 minislots, 19 minislots and 20 minislots. We use the scheduling assignment as per [30] and the network parameters which are set for all of the 3 experiments are as follows.

**Table 6-19** Configuration Parameters for the Dynamic Segment Experiments

| Network Parameter | Value |
|---|---|
| FlexRay Cycle Length | 4 ms |
| Static Slot Length | 31 μs |
| Macrotick Length | 1 μs |
| Minislot Length | 5 MT |
| SymbolWindow | 100 MT |
| NIT | 800 MT |
| DynamicSlotIdlePhase | 1 MS |

The details of the experiments and the discussions about the results are given in the following sections.

### 6.3.3.1 Dynamic Segment with the Length of 18 Minislots

The aim of this experiment is to determine whether the dynamic segment signals exchanged in the experiment are delivered within their deadlines when the dynamic segment length is 18 minislots. The signals in this experiment are also exchanged by three nodes as it is the case in the previous FlexRay experiments. In the experiment, only the dynamic segment of the FlexRay cycle is used, i.e. no static segment message is exchanged. The message set used in this dynamic segment experiment is given in Table 6-20.

**Table 6-20** Message Set for the Dynamic Segment Experiments

| Signal | | D1 | D2 | D3 | D4 | D5 |
|---|---|---|---|---|---|---|
| Period (ms) | | 10 | 10 | 20 | 20 | 25 |
| Deadline (ms) | | 5 | 10 | 15 | 15 | 18 |
| Length (B) | | 18 | 12 | 8 | 12 | 4 |
| # of Minislot | | 8 | 7 | 6 | 7 | 5 |

The period in the above table means the minimum inter-arrival time between the consecutive sporadic messages. The priority assignments for these messages, which are shown in Table 6-21, are done according to the scheduling algorithm which is introduced in section 6.3.3 and discussed in detail in [30].

**Table 6-21** Priority Assignment

| Signal | D1 | D2 | D3 | D4 | D5 |
|---|---|---|---|---|---|
| Priority | 1 | 2 | 3 | 4 | 5 |

The experiment is run for this message set and the corresponding priority assignment with the network configurations described in section 6.3.3 for about 2 minutes. So as to increase the probability to create the worst case scenario, consecutive sporadic messages are generated within 3 ms after the minimum inter-arrival time. The network traffic is analyzed via the FlexAlyzer to obtain the end-to-end delay values that the sporadic messages experience in the network to check whether the message set is schedulable with the dynamic segment of 18 minislots as in the theoretical computation. In this experiment, no jitter value is calculated since, by definition, the performance metric jitter is only applicable to the periodic messages. The results obtained for the worst case end-to-end delay is given in the Table 6-22 and Figure 6-24. In Figure 6-24, the dynamic segment priorities of the signals and the end-to-end delay values that those signals experience are depicted.

**Table 6-22** End-to-End Delay: Dynamic Segment with 18 Minislots

| Signal | Priority | Delay (ms) | Deadline (ms) |
|---|---|---|---|
| D1 | 1 | 4.034 | 5 |
| D2 | 2 | 4.067 | 10 |
| D3 | 3 | 8.022 | 15 |
| D4 | 4 | 8.059 | 15 |
| D5 | 5 | 19.804 | 18 |

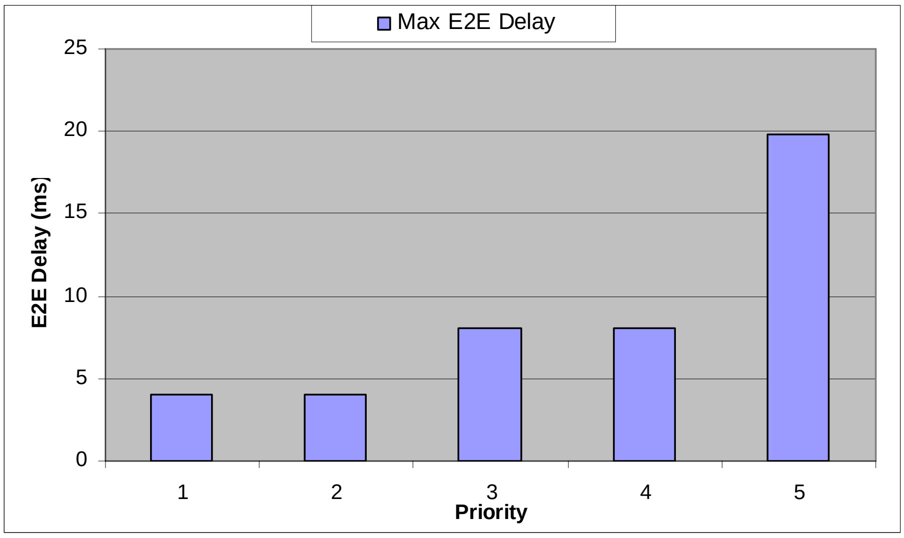


**Figure 6-24** End-to-End Delay vs Priority: Dynamic Segment with 18 Minislots

In [30], the maximum time duration that a dynamic segment message has to wait before being delivered is calculated theoretically according to the message set, priority assignment and the minislot number. The results obtained in the experiment are compared with the theoretical maximum values in Table 6-23 and Figure 6-25.

The envelope line that is observed in Figure 6-25 represents the theoretical maximum values that the signals can ever experience in the network.

**Table 6-23** E2E Delay Comparison Against the Theoretical Maximum Values

| Message | Priority | Delay in Experiment (ms) | Theoretical Max (ms) |
|---|---|---|---|
| D1 | 1 | 4.034 | 4.04 |
| D2 | 2 | 4.067 | 4.07 |
| D3 | 3 | 8.022 | 8.035 |
| D4 | 4 | 8.059 | 8.065 |
| D5 | 5 | 19.804 | (−) |

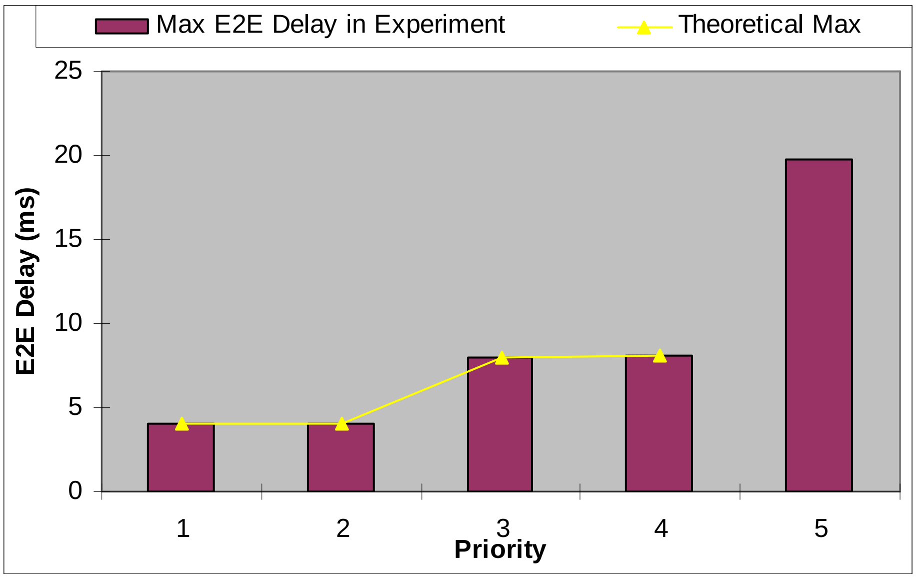


**Figure 6-25** Maximum End-to-End Delay and Theoretical Maximum

When the Table 6-23 and Figure 6-25 are observed, it is found out that the dynamic signal with the lowest priority, D5, is delivered beyond its deadline as expected. Because when the algorithm, details of which are discussed in [30], is run for the dynamic segment of length 18 minislots, this experiment's message set is found to non-schedulable. This theoretical expectation mentioned in [30] is verified via this dynamic segment experiment.

### 6.3.3.2 Dynamic Segment with the Length of 19 Minislots

In this experiment, the number of minislots composing the Dynamic Segment is increased from 18 to 19. Also, the priority assignment is changed and given in Table 6-24. Apart from these two changes, no other modification is made on the previous experiment. The aim of this experiment is to obtain the worst case end-to-end delay values that the sporadic signals experience and decide whether the message set is schedulable with the dynamic segment of 19 minislots. After having run the experiment for 2 minutes, the obtained results are exhibited in Table 6-25 and Figure 6-26 together with the theoretical the maximum time delay that a dynamic segment message experiences according to [30]. In Figure 6-26, the bars represent the end-to-end delay values of the corresponding signals and the envelope line shows the maximum delay limit that those signals can experience in the network.

**Table 6-24** Priority Assignment

| Signal | D1 | D2 | D3 | D4 | D5 |
|---|---|---|---|---|---|
| Priority | 1 | 2 | 4 | 3 | 5 |

**Table 6-25** End-to-End Delay: Dynamic Segment with 19 Minislots

| Signal | Priority | Delay (ms) | Theoretical Max (ms) |
|---|---|---|---|
| D1 | 1 | 4.034 | 4.04 |
| D2 | 2 | 4.067 | 4.07 |
| D3 | 4 | 8.053 | 8.065 |
| D4 | 3 | 8.034 | 8.035 |
| D5 | 5 | 15.943 | 16.025 |

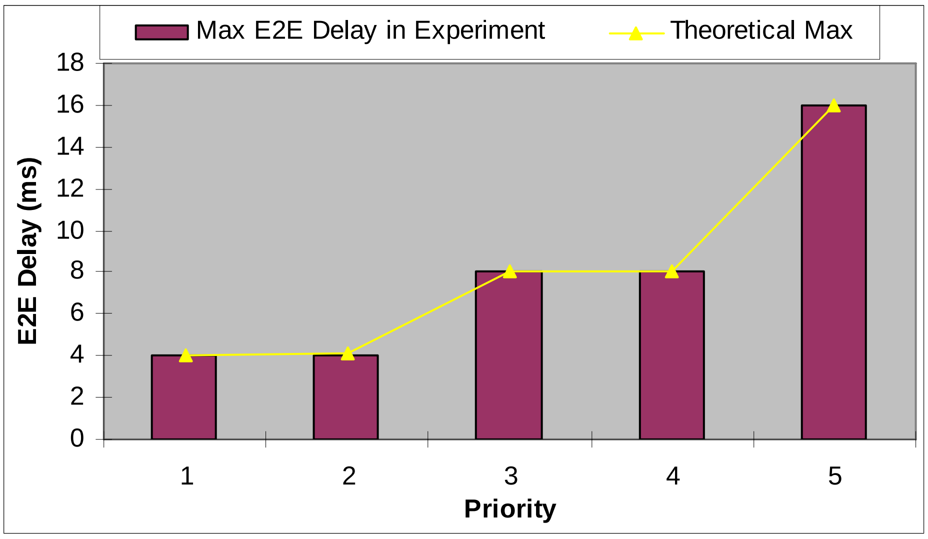


**Figure 6-26** End-to-End Delay and Theoretical Maximum

In agreement with the computations of the algorithm in [30], the results exhibited in Table 6-25 and Figure 6-26 shows that the message set is schedulable for the dynamic segment of 19 minislots and all the messages can be delivered within a bounded time without violating their deadline requirements.

#### 6.3.3.3 Dynamic Segment with the Length of 20 Minislots

This is the last dynamic segment experiment where the length is increased to 20 minislots. The priority assignment for the experiment is the same as the experiment with 18 minislots and given in Table 6-26. All of the other configurations are done as described in section 6.3.3. The aim of this experiment, which is same as the previous experiments, is to obtain the worst case end-to-end delay values that the sporadic signals experience and check whether the message set is schedulable with the dynamic segment of 20 minislots. The experiment is run for about 120 seconds and the network is traffic is monitored and logged by FlexAlyzer. The results obtained after parsing the log file is shown below in Table 6-27 and Figure 6-27. Similar to the previous dynamic segment experiments, also in Figure 6-27, the theoretical maximum end-to-end delay values that the sporadic signals can experience are shown with an envelope line.

**Table 6-26** Priority Assignment

| Signal | D1 | D2 | D3 | D4 | D5 |
|---|---|---|---|---|---|
| Priority | 1 | 2 | 3 | 4 | 5 |

**Table 6-27** End-to-End Delay and Theoretical Maximum

| Signal | Priority | Delay (ms) | Theoretical Max (ms) |
|---|---|---|---|
| D1 | 1 | 4.034 | 4.04 |
| D2 | 2 | 4.067 | 4.07 |
| D3 | 3 | 8.02 | 8.035 |
| D4 | 4 | 8.06 | 8.065 |
| D5 | 5 | 14.82 | 16.025 |

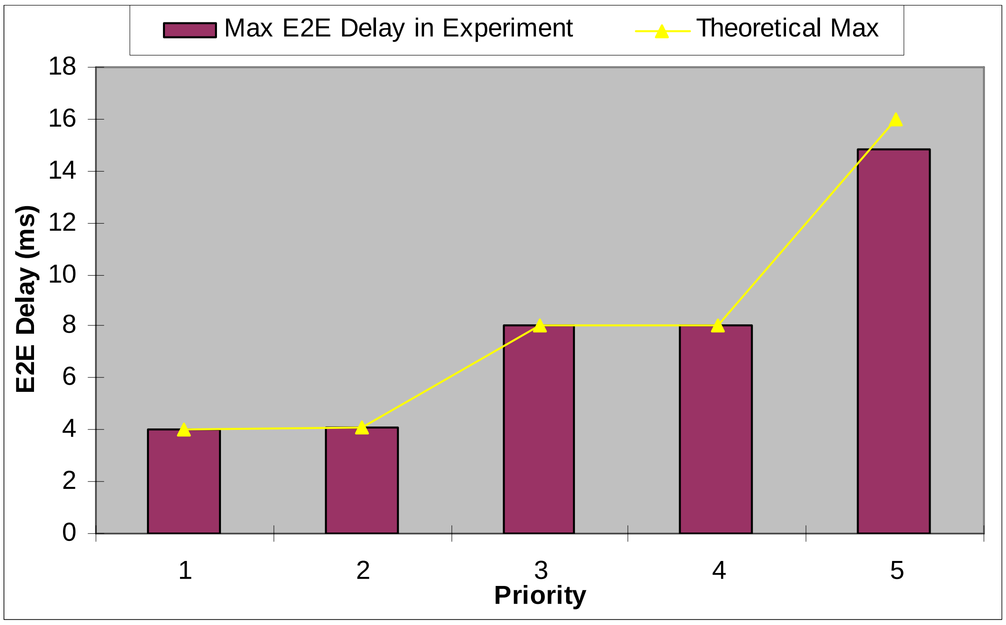


**Figure 6-27** End-to-End Delay vs Theoretical Maximum

Therefore, the results shown in Table 6-27 and Figure 6-27 verify the algorithm computations made in [30] that the dynamic segment with the length of 20 minislots is schedulable for this message set and the configuration and that all the messages can be delivered within a bounded time.

### 6.3.4 Gateway Experiments

The experiments which are performed in this section can be grouped under two subsections. In the first group of experiments, the Gateway functionality is verified and tested. By means of the other group of experiments, the end-to-end performance analysis of the inter-connected FlexRay and CAN networks is performed for the performance metrics of delay and jitter, as well as the performance analysis of the Gateway node in terms of Gateway processing delay. The experiments discussed in this chapter are, namely, Gateway Protocol Conversion Functionality, Gateway Signal Mapping Functionality, Gateway Performance Measurements and The Effect of Polling.

#### 6.3.4.1 Gateway Functionality: Protocol Conversion

The aim of this experiment is to verify that the designed Gateway performs its basic functionality, which is the protocol conversion, and operates according to the requirements that it is programmed to. What makes this experiment more valuable is that this verification is held on the real hardware of an automotive company. The utilized real components which exist in a real automobile are Instrument Panel Cluster (IPC) and Steering Angle Sensor (SAS). In the experiment all of the capabilities of the Gateway are tested. For this purpose, the FlexRay messages with different periods are exchanged through the Gateway. Besides, Gateway is scheduled to use its dynamic segment. On top of these, in this experiment, Gateway is tested to realize the protocol conversion while it is connected to two distinct CAN busses with different bit rates, namely, 50 kbps and 500 kbps. The network topology and the experiment set-up are illustrated in Figure 6-28 and the photograph of the real experiment environment is show in Figure 6-29.

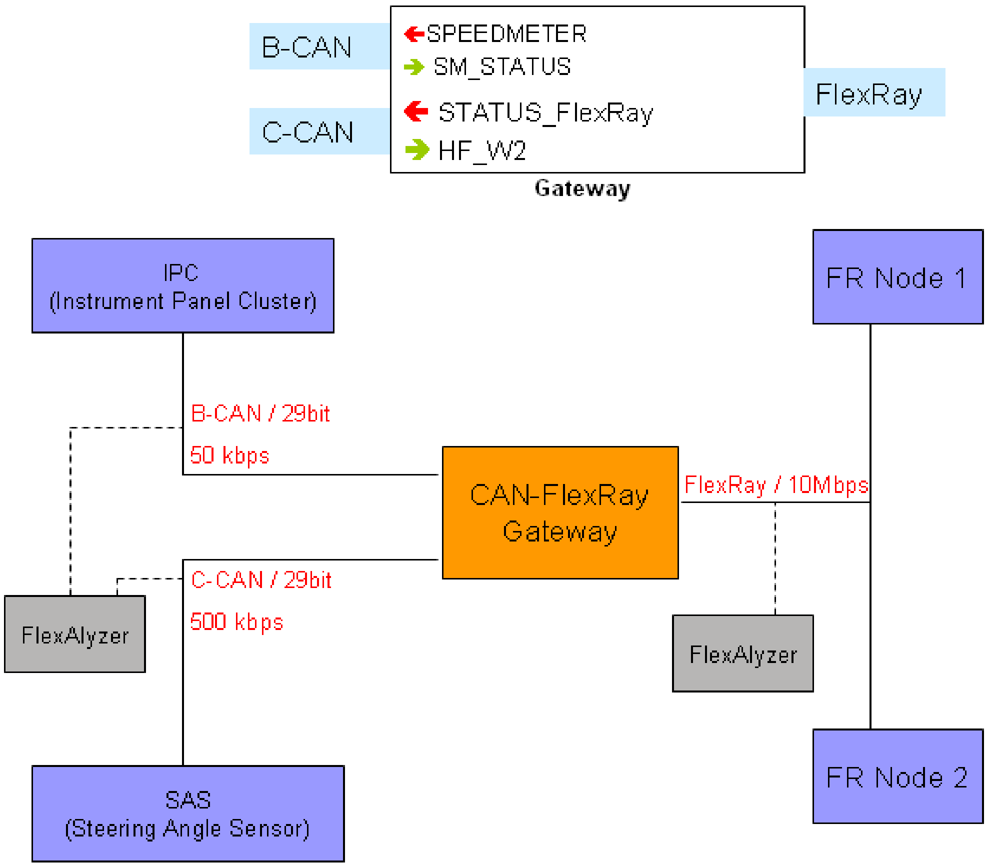


**Figure 6-28** Protocol Conversion Experiment Network Topology

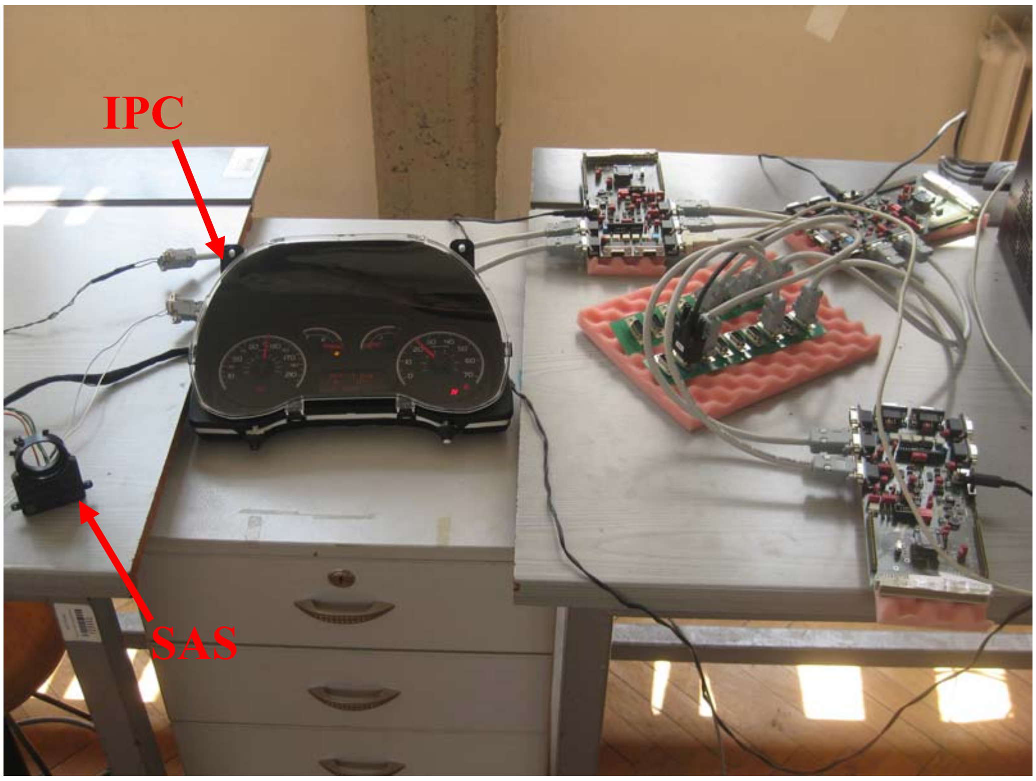


**Figure 6-29** Photograph of the Protocol Conversion Experiment

The configuration parameters for the network and the message set together with signal mapping are given in Table 6-28 and Table 6-29, respectively.

**Table 6-28** Experiment Configuration Parameters

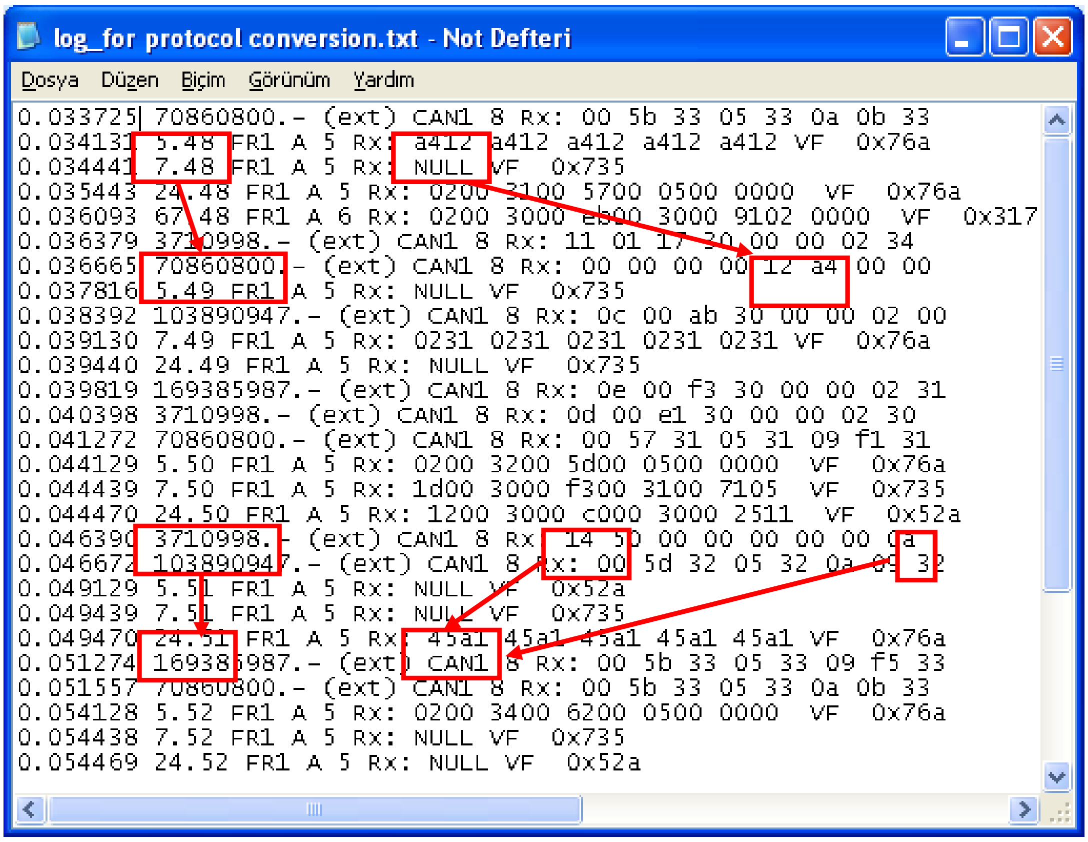
| Network Parameter | Value |
|---|---|
| FlexRay Cycle Length | 5 ms |
| Static Slot Length | 31 μs |
| Static Slot Number | 64 |
| Minislot Length | 5 μs |
| Minislot Number | 20 |
| B-CAN Data Rate | 50 kbps |
| C-CAN Data Rate | 500 kbps |

**Table 6-29** Signal Mapping for the Experiment

| CAN | | | Gateway | FlexRay | |
|---|---|---|---|---|---|
| Sending Node | CAN ID | Direction | Gateway Send FID | Direction | Receiving Node |
| SAS | 0x27ABDC | → | 24 | → | FlexCard |
| IPC (Status) | 0x6336983 | → | 67 | → | FlexCard |

| CAN | | Gateway | FlexRay | | |
|---|---|---|---|---|---|
| Receiving Node | Direction | Gateway Send ID | Direction | FID | Sending Node |
| IPC (Speed Odometer) | ← | 0xABFD123 | ← | 5 | FR Node 1 |
| CAN Analyzer | ← | 0xA0CA246 | ← | 7 | FR Node 2 |

The experiment is run with this configuration and the mapping on a real hardware as mentioned above. As a result, it is verified that the Gateway performs the protocol conversion task successfully and no problem occurs during the operation of the Gateway on the hardware of an automotive company. In Figure 6-30, a snapshot from the log file of the experiment exhibits how the Gateway handles the protocol conversion.


log_for protocol conversion.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım

```
0.033725 70860800.- (ext) CAN1 8 Rx: 00 5b 33 05 33 0a 0b 33
0.034131 5.48 FR1 A 5 Rx: a412 a412 a412 a412 a412 VF  0x76a
0.034441 7.48 FR1 A 5 Rx: NULL VF  0x735
0.035443 24.48 FR1 A 5 Rx: 0200 3100 5700 0500 0000  VF  0x76a
0.036093 67.48 FR1 A 6 Rx: 0200 3000 eb00 3000 9102 0000  VF  0x317
0.036379 3710998.- (ext) CAN1 8 Rx: 11 01 17 30 00 00 02 34
0.036665 70860800.- (ext) CAN1 8 Rx: 00 00 00 00 12 a4 00 00
0.037816 5.49 FR1 A 5 Rx: NULL VF  0x735
0.038392 103890947.- (ext) CAN1 8 Rx: 0c 00 ab 30 00 00 02 00
0.039130 7.49 FR1 A 5 Rx: 0231 0231 0231 0231 0231 VF  0x76a
0.039440 24.49 FR1 A 5 Rx: NULL VF  0x735
0.039819 169385987.- (ext) CAN1 8 Rx: 0e 00 f3 30 00 00 02 31
0.040398 3710998.- (ext) CAN1 8 Rx: 0d 00 e1 30 00 00 02 30
0.041272 70860800.- (ext) CAN1 8 Rx: 00 57 31 05 31 09 f1 31
0.044129 5.50 FR1 A 5 Rx: 0200 3200 5d00 0500 0000  VF  0x76a
0.044439 7.50 FR1 A 5 Rx: 1d00 3000 f300 3100 7105  VF  0x735
0.044470 24.50 FR1 A 5 Rx: 1200 3000 c000 3000 2511  VF  0x52a
0.046390 3710998.- (ext) CAN1 8 Rx: 14 50 00 00 00 00 00 0a
0.046672 103890947.- (ext) CAN1 8 Rx: 00 5d 32 05 32 0a 09 32
0.049129 5.51 FR1 A 5 Rx: NULL VF  0x52a
0.049439 7.51 FR1 A 5 Rx: NULL VF  0x735
0.049470 24.51 FR1 A 5 Rx: 45a1 45a1 45a1 45a1 45a1 VF  0x76a
0.051274 169385987.- (ext) CAN1 8 Rx: 00 5b 33 05 33 09 f5 33
0.051557 70860800.- (ext) CAN1 8 Rx: 00 5b 33 05 33 0a 0b 33
0.054128 5.52 FR1 A 5 Rx: 0200 3400 6200 0500 0000  VF  0x76a
0.054438 7.52 FR1 A 5 Rx: NULL VF  0x735
0.054469 24.52 FR1 A 5 Rx: NULL VF  0x52a
```

**Figure 6-30** Log File of the Experiment

Via the Figure 6-30, the Gateway is verified to perform the task of protocol conversion. Also from the Figure 6-30, it is observed that the Gateway can extract the CAN data partially and map it to the FlexRay message. Similarly, it can map the incoming FlexRay message to the specified part of the CAN signal. This capability of the Gateway will be examined in more details in the following chapter.

### 6.3.4.2 Gateway Functionality: Signal Mapping

Another important functionality of the Gateway is tested in this experiment. This functionality has two folds. The first one is the ability of the Gateway to segment any incoming message and forward them to the other side as separate signals. The other fold of the functionality is just the opposite. It is the ability of the Gateway to combine several incoming signals so as to send them to the other network in a single message. In order to test this capability of the Gateway this experiment is established. According to the experiment, 3 CAN nodes send 10 CAN messages to the Gateway to be transferred to the FlexRay network. The Gateway combines two CAN signals into one FlexRay message and sends 10 CAN messages in 5 FlexRay signals through the FlexRay network. Besides, it segments the incoming FlexRay messages into several CAN signals and transmits each of segmented CAN signals separately. The Gateway capability to be tested in this experiment is illustrated in Figure 6-31.

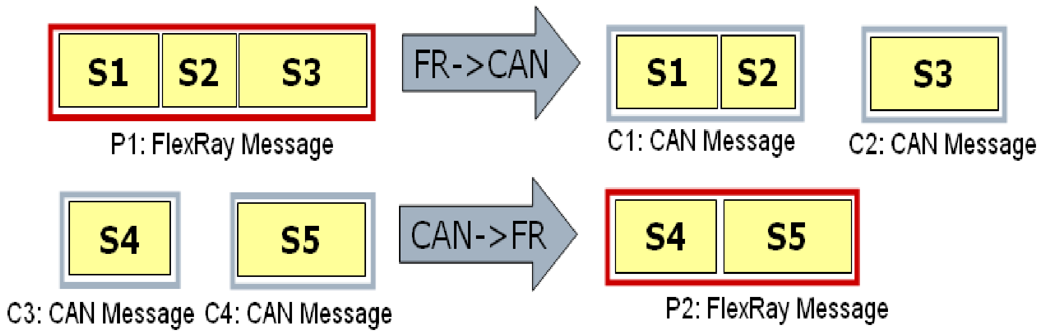


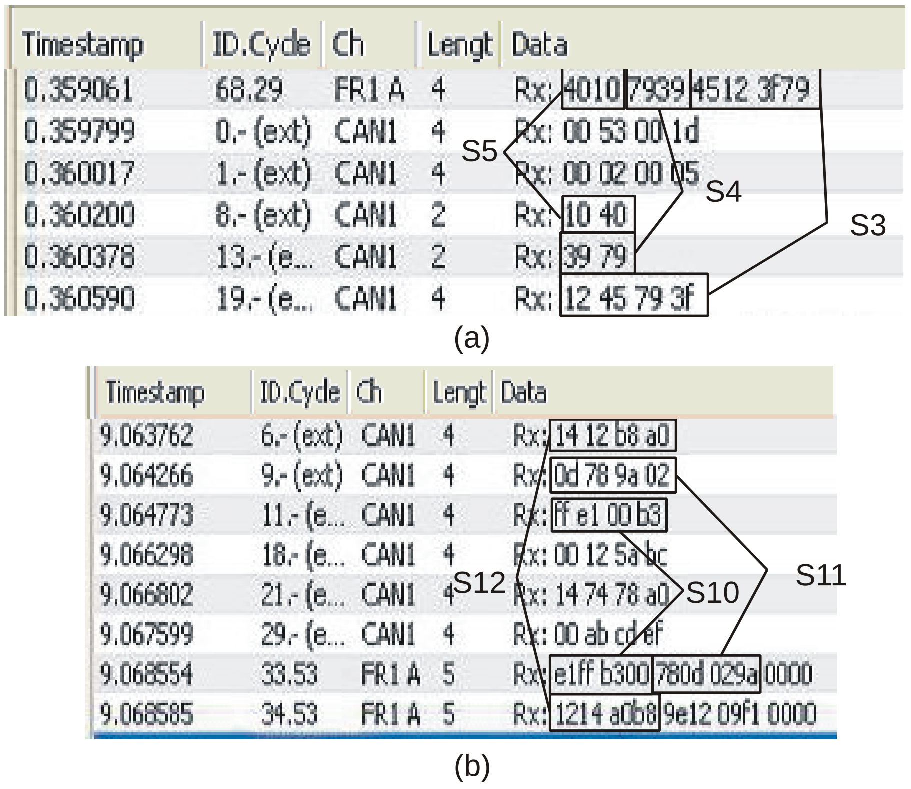
**Figure 6-31** Signal Mapping Functionality

The experiment is set up on seven distinct nodes. Out of seven nodes, three nodes are used as CAN nodes, another three nodes are FlexRay nodes and the remaining node is the Gateway node. Although numerous other signals are exchanged in CAN bus and in FlexRay network, we are only concerned about the messages on which the Gateway performs the signal mapping functionality. Those signals of concern and their mapping are given in Table 6-30.

**Table 6-30** Experiment Signal Mapping Scheme

| Combination | | Segmentation | |
|---|---|---|---|
| CAN ID | FlexRay ID | FlexRay ID | CAN ID |
| 31 | 33 | 23 | 1 |
| 29 | | | 0 |
| 21 | 34 | 68 | 19 |
| 18 | | | 13 |
| 11 | 33 | | 8 |
| 9 | | | |
| 6 | 34 | | |
| 5 | | | |
| 4 | 67 | | |
| 2 | | | |

After the experiment has been run and the network traffic is monitored via FlexAlyzer, a log file is obtained showing the behaviour of the network. When the monitor screen of the FlexAlyzer, a snapshot of which is included in Figure 6-32, is observed, it is clearly seen that the Gateway fully satisfies the signal mapping functionality.


```
Timestamp   ID.Cycde    Ch      Lengt  Data
0.359061    68.29       FR1 A   4      Rx: 4010 7939 4512 3f79
0.359799    0.- (ext)   CAN1    4      Rx: 00 53 00 1d
0.360017    1.- (ext)   CAN1    4      Rx: 00 02 00 05
0.360200    8.- (ext)   CAN1    2      Rx: 10 40
0.360378    13.- (e...  CAN1    2      Rx: 39 79
0.360590    19.- (e...  CAN1    4      Rx: 12 45 79 3f
```

```
Timestamp   ID.Cycde    Ch      Lengt  Data
9.063762    6.- (ext)   CAN1    4      Rx: 14 12 b8 a0
9.064266    9.- (ext)   CAN1    4      Rx: 0d 78 9a 02
9.064773    11.- (e...  CAN1    4      Rx: ff e1 00 b3
9.066298    18.- (e...  CAN1    4      Rx: 00 12 5a bc
9.066802    21.- (e...  CAN1    4      Rx: 14 74 78 a0
9.067599    29.- (e...  CAN1    4      Rx: 00 ab cd ef
9.068554    33.53       FR1 A   5      Rx: e1ff b300 780d 029a 0000
9.068585    34.53       FR1 A   5      Rx: 1214 a0b8 9e12 09f1 0000
```

**Figure 6-32** a) Segmentation b) Combination

### 6.3.4.3 Gateway Performance: Real-Time Measurements

We have verified, thus far, the basic functionality of the Gateway in the previous experiments, one of which is held on a real hardware. In this experiment, the performance of the designed Gateway will be examined with respect to the performance metrics, namely, worst case end-to-end delay and jitter. So as to have a complete picture together with the experiments held in sections 6.3.1, 6.3.2 and 6.3.3, we have decided to set the Gateway experiment up on the message sets examined in the previous FlexRay and the CAN experiments. The messages are generated with the same configuration that they are used in sections 6.3.1.3 and 6.3.2.1. That is to say, both CAN and FlexRay message sets are generated in 3 distinct nodes and the mapping of the signals to the nodes are as it is in the corresponding chapters. The message sets for the CAN signals and the FlexRay signals are given in Table 6-2 and Table 6-14, respectively. Since there exists a

Gateway unit in this experiment set up, some of the signals from both CAN and the FlexRay message sets are chosen to be passed to the other network through the Gateway.

Coming to the scheduling schemes, there exists more than one possibility for both CAN and FlexRay messages. For the CAN message set to be used in the Gateway performance experiment, the CAN Scheduling with Fixed Priorities is decided to be assigned. The reason for choosing this scheduling assignment is that it has more practical concern with respect to the other two priority assignment schemes discussed under section 6.3.1. On the other hand, the FID allocation without jitter scheme is chosen for the FlexRay message set since we want to decrease the overall jitter value that the messages experience. Moreover, economizing the bandwidth for such a case where only several tens of signals are exchanged, has no practical value. The priority assignment for the CAN messages which travel only in CAN bus, FID allocation for the FlexRay messages that are exchanged only in FlexRay network and the priority assignment of both CAN and FlexRay messages that cross the Gateway are given in Table 6-31, Table 6-32 and Table 6-33, respectively. The CAN and the FlexRay signals, which cross the Gateway, are highlighted in Table 6-31 and Table 6-32 so as to provide a better understanding. One thing to be noted about the Table 6-32 and Table 6-33 is that the FID allocations for the P5, P7, P10 and P16 are different in Table 6-33 than the original FID assignment located in Table 6-32. The reason for this is that these FlexRay messages share the same time slot with other FlexRay signals. For instance, the FlexRay signals P7 and P8 are both to be sent in the 17th static slot. If the Gateway was also to transmit in the same time slots, the two distinct nodes would happen to share the same time slot which violates the FlexRay protocol. Therefore, the FID assignment for the signals P5, P7, P10 and P16 are defined as in Table 6-33. However for P2, the FID allocation is not changed. This is because; Gateway is not the sender node for P2 but only the receiver node.

On the other hand, in order to make the performance analysis of a complete Gateway, which handles and routes all kinds of signals, the dynamic segment messages are also mixed in the CAN and the FlexRay signals. The message set and the priority assignment for the sporadic messages are same as those of the dynamic segment experiment with 20 minislots in section 6.3.3.3. There are only two differences from the configuration of section 6.3.3.3. The first one is that in this experiment, all of the sporadic messages are generated by only one FlexRay node and the Gateway node. The second difference is done by applying a minor change to the priority assignment scheme. Two out of the total 5 dynamic segment signals are chosen to be passed through the Gateway and shown in Table 6-33. For the sake of completeness the priority assignment of the sporadic messages are given in Table 6-35, shading in color the signals that cross the Gateway.

**Table 6-31** CAN Scheduling with Fixed Priorities

| Signal | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 |
|---|---|---|---|---|---|---|---|---|---|
| Priority | 7 | 3 | 10 | 12 | 17 | 28 | 30 | 20 | 1 |
| Signal | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 |
| Priority | 0 | 58 | 22 | 14 | 31 | 29 | 21 | 19 | 18 |
| Signal | C19 | C20 | C21 | C22 | C23 | C24 | C25 | C26 | |
| Priority | 13 | 11 | 9 | 8 | 6 | 5 | 4 | 2 | |

**Table 6-32** FID Allocation Without Jitter

| Signal | P1 | P2 | P3 | P4 | P5 | P6 | P7 |
|---|---|---|---|---|---|---|---|
| Period/ms | 10 | 5 | 20 | 10 | 10 | 10 | 10 |
| FID | 24 | 23 | 28 | 24 | 22 | 25 | 17 |
| Repetition/ms | 10 | 5 | 20 | 10 | 10 | 10 | 10 |
| Offset | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| Signal | P8 | P9 | P10 | P11 | P12 | P13 | P14 |
| Period/ms | 10 | 10 | 10 | 10 | 20 | 10 | 20 |
| FID | 17 | 25 | 26 | 26 | 28 | 27 | 29 |
| Repetition/ms | 10 | 10 | 10 | 10 | 20 | 10 | 20 |
| Offset | 1 | 1 | 0 | 1 | 3 | 0 | 0 |
| Signal | P15 | P16 | P17 | P18 | P19 | P20 | P21 |
| Period/ms | 10 | 10 | 10 | 10 | 100 | 50 | 100 |
| FID | 27 | 18 | 18 | 19 | 20 | 19 | 29 |
| Repetition/ms | 10 | 10 | 10 | 10 | 20 | 10 | 20 |
| Offset | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| Signal | P22 | P23 | P24 | P25 | P26 | P27 | P28 |
| Period/ms | 100 | 100 | 250 | 500 | 250 | 10 | 100 |
| FID | 29 | 29 | 31 | 31 | 21 | 28 | 20 |
| Repetition/ms | 20 | 20 | 10 | 20 | 10 | 10 | 20 |
| Offset | 2 | 3 | 0 | 3 | 0 | 0 | 1 |
| Signal | P29 | P30 | P31 | P32 | P33 | P34 | P35 |
| Period/ms | 100 | 100 | 2000 | 2000 | 1000 | 1000 | 20 |
| FID | 20 | 20 | 22 | 22 | 21 | 21 | 31 |
| Repetition/ms | 20 | 20 | 80 | 80 | 40 | 40 | 20 |
| Offset | 2 | 3 | 1 | 3 | 3 | 7 | 1 |
| Signal | P36 | P37 | P38 | P39 | P40 | P41 | |
| Period/ms | 2000 | 2000 | 2000 | 2000 | 2000 | 100 | |
| FID | 22 | 22 | 30 | 30 | 32 | 21 | |
| Repetition/ms | 80 | 80 | 80 | 80 | 80 | 20 | |
| Offset | 5 | 7 | 0 | 1 | 0 | 1 | |

**Table 6-33** CAN and FlexRay Messages that Cross the Gateway

| CAN2FR | | | | FR2CAN | | | |
|---|---|---|---|---|---|---|---|
| CAN | | FlexRay | | FlexRay | | CAN | |
| C14 | 31 | 33 | P5 | P2 | 23 | 1 | C9 |
| C15 | 29 | | | | | 0 | C10 |
| C16 | 21 | 34 | P7 | D3 | 68 | 19 | C17 |
| C18 | 18 | | | | | 13 | C19 |
| C20 | 11 | 33 | P10 | | | 8 | C22 |
| C21 | 9 | | | | | | |
| C23 | 6 | 34 | P16 | | | | |
| C24 | 5 | | | | | | |
| C25 | 4 | 67 | D4 | | | | |
| C26 | 2 | | | | | | |

The signal names shown in Table 6-33 are renamed in Gateway perspective so that the results are exhibited without causing any confusion.

**Table 6-34** Signal Names in Gateway

| Name in CAN | Name in Gateway |
|---|---|
| C9 | S1 |
| C10 | S2 |
| C17 | S3 |
| C19 | S4 |
| C22 | S5 |
| C14 | S6 |
| C15 | S7 |
| C16 | S8 |
| C18 | S9 |
| C20 | S10 |
| C21 | S11 |
| C23 | S12 |
| C24 | S13 |
| C25 | S14 |
| C26 | S15 |

**Table 6-35** Dynamic Segment Messages

| Signal | D1 | D2 | D3 | D4 | D5 |
|---|---|---|---|---|---|
| Priority | 65 | 66 | 68 | 67 | 69 |

The Gateway experiment set-up is composed of 7 distinct SK-91465X-100MPC Fujitsu nodes. Out of these 7 nodes, three of them are used as CAN nodes, the other three are used as FlexRay nodes and the remaining node is the Gateway node. The experimentation parameters for the Gateway Performance Analysis experiment can be summarized in Table 6-36.

**Table 6-36** Gateway Experiment Parameters

| Network Parameter | Value |
|---|---|
| FlexRay Cycle Length | 5 ms |
| Static Slot Length | 31 µs |
| Static Slot Number | 64 |
| Minislot Length | 5 µs |
| Minislot Number | 20 |
| CAN Data Rate | 500 kbps |

So as to calculate the end-to-end delay and the jitter that the signals, which cross the Gateway, experience, during the travel of the signals, 4 time stamps are obtained at four different points of the network. Besides the end-to-end delay values, the delay components that the signals experience only in CAN network, only in FlexRay network and only in the Gateway are also calculated via these time stamps. The details about how to obtain those time stamps, how to calculate the delay values and the possible errors in calculations are discussed in sections 6.2.2 and 6.2.3. After having run the Gateway experiment for 2 minutes, the log file obtained by the FlexAlyzer software is analyzed and the results shown in Table 6-37, Figure 6-33 and Figure 6-35 are obtained. In these table and figures, the delay and the jitter values that only 15 signals, which cross the Gateway, experience are exhibited.

**Table 6-37** Experiment Results for the Signals Passing the Gateway

| Signal S | CAN Delay (ms) | GW Delay (ms) | FR Delay (ms) | E2E Delay (ms) | Jitter (ms) |
|---|---|---|---|---|---|
| S1 | 0.840 | 1.912 | 0.607 | 3.325 | 0.11693 |
| S2 | 0.576 | 1.889 | 0.607 | 3.042 | 0.11516 |
| S3 | 2.710 | 0.716 | 9.985 | 11.956 | NA |
| S4 | 1.866 | 0.694 | 9.985 | 11.678 | NA |
| S5 | 1.029 | 0.672 | 9.985 | 11.398 | NA |
| S6 | 8.228 | 0.050 | 4.037 | 10.808 | 0.00499 |
| S7 | 7.104 | 0.050 | 4.580 | 10.789 | 0.00499 |
| S8 | 6.333 | 0.050 | 6.063 | 10.912 | 0.00499 |
| S9 | 4.851 | 0.050 | 6.637 | 10.890 | 0.00499 |
| S10 | 3.662 | 0.050 | 3.314 | 5.837 | 0.01272 |
| S11 | 2.235 | 0.050 | 3.886 | 5.771 | 0.01272 |
| S12 | 1.563 | 0.050 | 4.487 | 5.784 | 0.00499 |
| S13 | 1.473 | 0.050 | 5.623 | 5.978 | 0.00499 |
| S14 | 1.163 | 0.050 | 6.034 | 6.936 | 0.06371 |
| S15 | 0.475 | 0.050 | 6.318 | 6.827 | 0.06188 |

In Figure 6-33, all of the delay components that the signals experience are shown together in one figure. These delay components are, end-to-end delay, CAN delay, FlexRay delay and the Gateway processing delay.

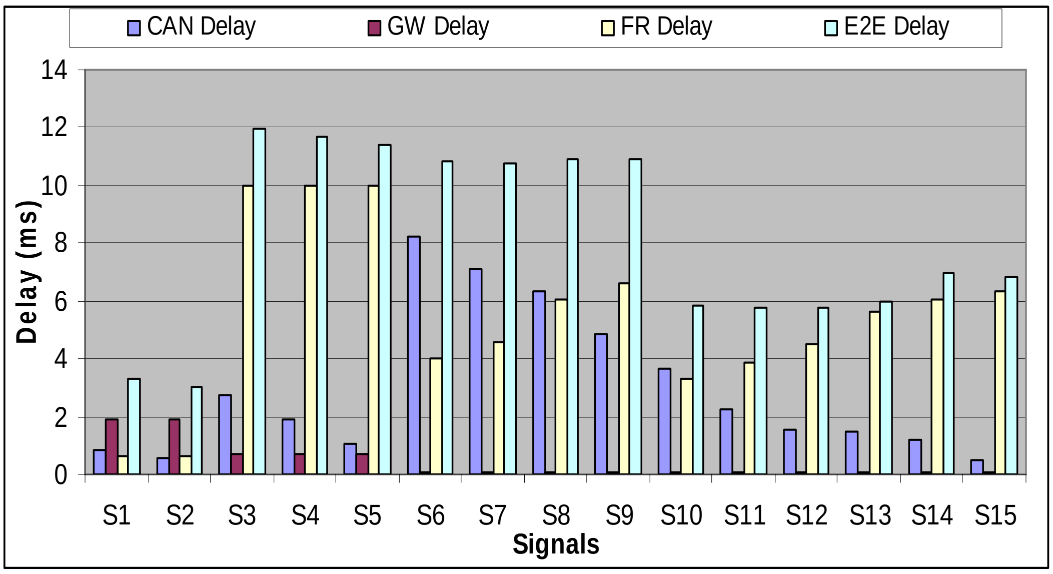


**Figure 6-33** Delay Decomposition of the Gateway Experiment

In order to better visualize the weight of the Gateway processing delay, the Figure 6-34 which shows the delay components of the signals in percentage is also included below. The delay components shown in Figure 6-33 except the end-to-end delay are represented in Figure 6-34

**Figure 6-34** such that the individual weight of the delay components to the total delay are exhibited and their summation makes 100%.

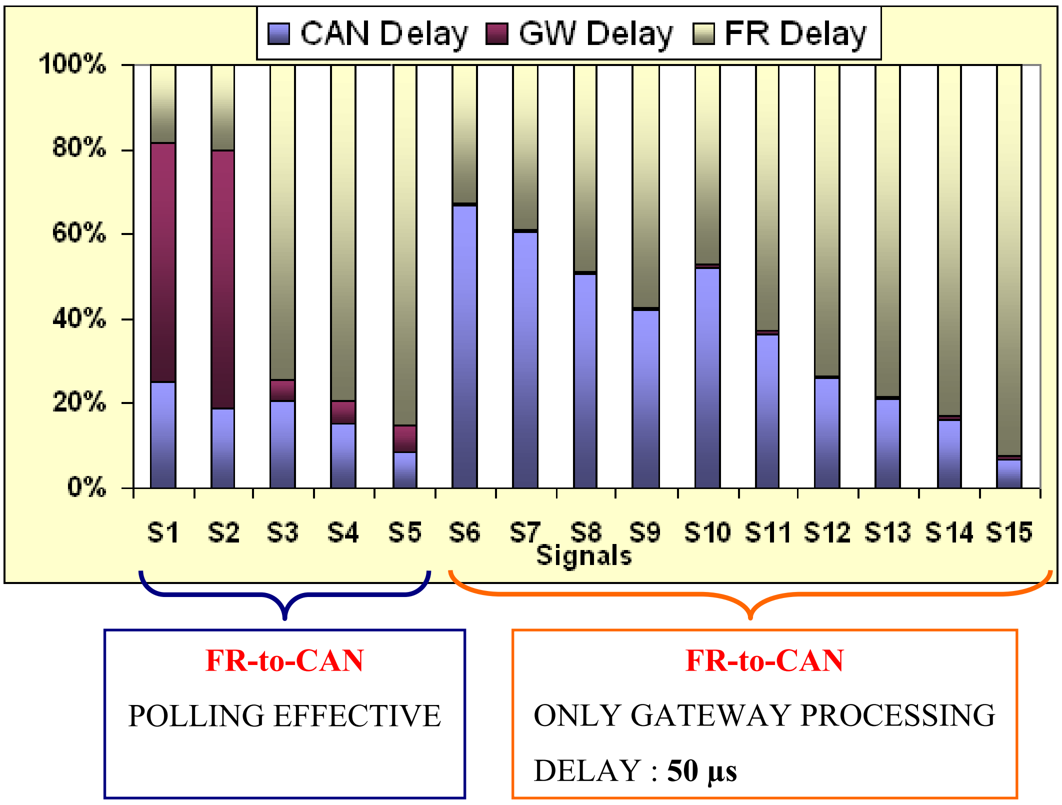


**Figure 6-34** Delay Decomposition of the Gateway Experiment in Percentage

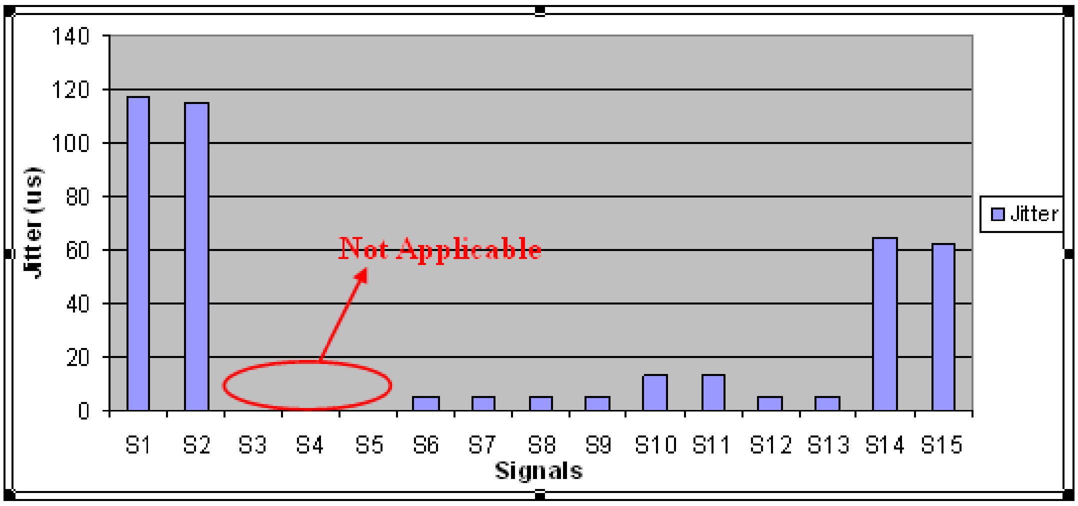


**Figure 6-35** Jitter Values in Gateway Experiment

Let us, first of all, analyze the delay behavior of the Gateway by means of the results exhibited in Table 6-37, Figure 6-33 and Figure 6-35. When the CAN delay that the signals experience is observed in Table 6-37 and compared against the delay values obtained in the CAN experiment with the same CAN scheduling scheme which is shown in Table 6-11, it is found out that, more or less, the all results except several agree with each other. This situation might arise from the fact that though, in both experiments, the same scheduling is applied, by the inclusion of the Gateway node, the signals are distributed among four nodes instead of three in the Gateway experiment. Moreover, since the Gateway node generates the CAN signals whenever it receives the corresponding FlexRay packet, the message generation pattern also differs from that of the CAN experiment. Therefore, due to these two factors, the deviations in the CAN Delay values in both experiments with respect to each other can be considered to be acceptable.

Coming to the FlexRay delay analysis of the Gateway, the FlexRay FID allocation in both dynamic segment and the static segment are changed with respect to the previous FlexRay experiments except the message P2 in Table 6-16. Therefore, no comparison can be done for the FlexRay delay values against any previous experiments except the delay that the signal P2 experiences. The FlexRay delay that P2 experiences in the FlexRay and the Gateway experiment are observed to be in agreement with each other.

When the Table 6-37 is examined to observe the time duration that the signals experience in the Gateway, it is found out that while the signals travel in the CAN-to-FR direction experience very little amount of time, the signals that cross the Gateway in the FR-to-CAN direction experience a considerable amount of time. The situation can be visualized in Figure 6-34 better. From the figure, it is seen that the Gateway delay has almost no contribution to the end-to-end delay for the signals passing the Gateway in CAN-to-FlexRay direction. The time duration that the signals stay in the Gateway in this direction is about 50 µs. Therefore, the Gateway performance in this direction is perfectly good. However, in the reverse direction, the Gateway delay is observed to be fairly high. The reason behind it lies in the polling mechanism which is used to receive the FlexRay packets. In the Gateway, the FlexRay messages are received through polling the incoming message buffers in every 5 ms as opposed to CAN receive mechanism. The Gateway is informed of the arrival of a CAN message through the interrupts issued by the CAN Controller. Thus, although a FlexRay message is received and put into the receive buffers of the Gateway, the CPU does not extract the buffer content before the polling time comes. This forms the biggest part of the time that the signals pass in the Gateway. The amount of the time delay caused by the polling mechanism can be discussed quantitatively in the following manner. In Table 6-33, it is seen that the Gateway receives only two FlexRay signals, one in 23rd and the other in 68th time slots. The polling time of the incoming buffers is chosen to be in the very middle of the whole cycle in order to decrease the time that the messages waste in the FlexRay receive buffers. For this purpose, the very same mechanism which is used to generate the messages in the very beginning of the FlexRay cycle and explained in details in section 5.2.1 is used to provide the time ticks as the polling time in the middle of the cycle, i.e at 2500th macrotick. Therefore we can assume that the CPU reads the FlexRay buffer content in 2500th macrotick. However, the messages from the 23rd and the 68th time slots are found to be received in macrotick 709 and 2070, respectively by using the following formula.

$$\begin{aligned} &FlexRay\_RX\_Macrotick(FID) = (FID-1)\times SS\_Length\_in\_MT + \\ &Action\_Offset\_in\_MT \end{aligned} \tag{6-9}$$

where SS_Length_in_MT and Action_Offset_in_MT are the matrotick correspondences of the network parameters the the Static Slot Length and the Action Offset Length, respectively and the FID is the slot number of the received FlexRay message.

As a result, the messages received from 23rd and the 68th time slots are read by the CPU 1800 µs and 430 µs later than the time they are actually stored in the received buffers. So, without the polling mechanism, the Gateway processing delay is found to be approximately 100 μs and 250 μs for the messages received from 23rd and the 68th time slots referring the Table 6-37. This result is in the order of the delay that the messages experience in the CAN-to-FlexRay direction.

The Gateway delay in the FlexRay-to-CAN direction can be decreased in two ways. The first solution is to use the interrupt mechanism also in receiving the FlexRay messages instead of polling. However this solution is not feasible due to two reasons. First of all, the FlexRay messages are received via a dedicated FlexRay Communication Controller. As mentioned previously, the application running on the CPU controls the Communication Controller by means of the Fujitsu FlexRay Software Driver. Therefore, so as to receive the FlexRay messages through interrupt mechanism, the Communication Controller must be checked if it supports interrupt processing. Even the Communication Controller supports the interrupt processing, it will take more time in the CPU to process the interrupts which are fed outside. Secondly, assume that, somehow, interrupt mechanism could be used to receive the FlexRay messages and the number of the messages exchanged are much greater. In such a situation, significant number of the available slots would be allocated. Due to the TDMA structure of the FlexRay protocol, once the messages begin to be received, CPU will be occupied, for a long while, by the interrupts issued consecutively. This would prevent the CPU from handling the other tasks that it should perform including the CAN receive interrupt processing. Moreover, the time available for the interrupt processing of the received FlexRay signals is only 31 μs which is the length of the static slot. In such a situation, the received messages would still be made wait in the buffers for the processing of the preceding interrupts and this would also cause significant amount of delays. Therefore, because of the reasons discussed above, the polling mechanism is considered to be more feasible with respect to the interrupt processing.

The other solution to decrease the Gateway delay in the FlexRay-to-CAN direction is to utilize the polling mechanism more than once throughout the cycle. This solution is tried in section 6.3.4.4 as a separate experiment. Referring the details to section 6.3.4.4, we can say that by increasing the polling frequency twice, it is observed that the Gateway processing delay is decreased by certain amount.

To sum up the discussion about the delay that the signals experience in the Gateway, we can say that the Gateway processing delay is bounded to 50 μs and the delay in the FlexRay-to-CAN direction is sourced from the polling mechanism as discussed above. The histograms of the delay that the signals experience in the Gateway in both directions are given in Figure 6-36 and Figure 6-37, respectively.

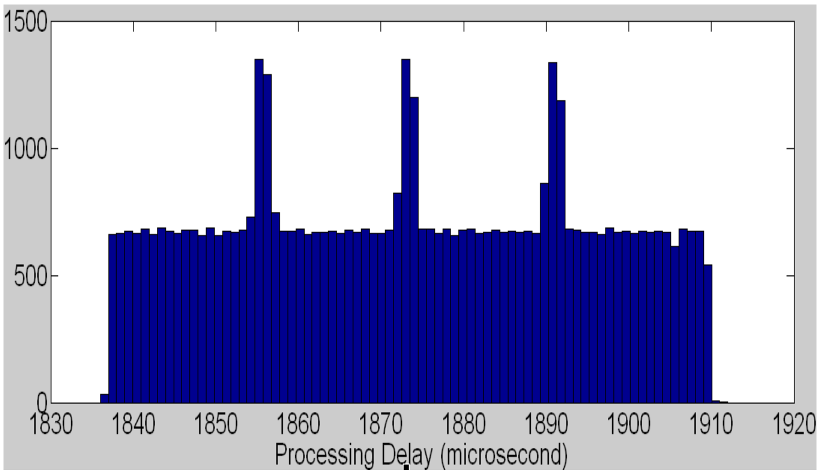


**Figure 6-36** Delay Histogram in FlexRay-to-CAN Direction

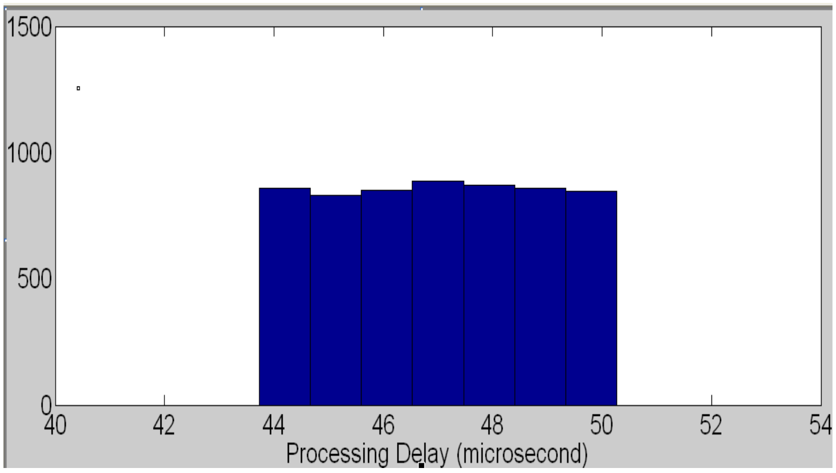


**Figure 6-37** Delay Histogram in CAN-to-FlexRay Direction

When we examine Figure 6-36 and Figure 6-37, we observe that the variance of the Gateway delay is fairly small. This means that the Gateway contribution to the end-to-end jitter is low. On the other hand, if the Table 6-37 is examined for the purpose of observing the jitter behavior of the network, it is seen that the jitter that the signals experience is considerably small such that they can be assumed to be zero. Although the CAN signals experience fairly big jitter values in percentage as seen in section 6.3.1.3, it can be said that because of the TDMA nature of the protocol, FlexRay regulates the signal flow such that the end-to-end jitter values appear to be fairly small. Therefore, the success of the network in jitter performance should mostly be attributed to the FlexRay network while considering the low-jitter behavior of the Gateway processing delay.

Finally, the results of the experiment are compared against the theoretical maximum values which are computed in [30] for both of the networks and demonstrated in Table 6-38.

**Table 6-38** Comparative Results for the Experiment and Theoretical Maximum

| Signal | CAN Delay (ms) (Exp/Max) | | FlexRay Delay (ms) (Exp/Max) | | E2E Delay (ms) (Exp/Max) | |
|---|---|---|---|---|---|---|
| S1 | 0.84 | 0.96 | 0.607 | 5 | 3.325 | 6.46 |
| S2 | 0.576 | 0.64 | 0.607 | 5 | 3.042 | 6.14 |
| S3 | 2.71 | 5.04 | 9.985 | 10 | 11.956 | 14.04 |
| S4 | 1.866 | 4.16 | 9.985 | 10 | 11.678 | 13.52 |
| S5 | 1.029 | 2.68 | 9.985 | 10 | 11.398 | 12.68 |
| S6 | 8.228 | 7.82 | 4.037 | 10 | 10.808 | 17.22 |
| S7 | 7.104 | 7.24 | 4.58 | 10 | 10.789 | 17.24 |
| S8 | 6.333 | 6.28 | 6.063 | 10 | 10.912 | 16.28 |
| S9 | 4.851 | 5.12 | 6.637 | 10 | 10.89 | 15.12 |
| S10 | 3.662 | 3.52 | 3.314 | 10 | 5.837 | 13.52 |
| S11 | 2.235 | 3 | 3.886 | 10 | 5.771 | 13 |
| S12 | 1.563 | 2.04 | 4.487 | 10 | 5.784 | 12.04 |
| S13 | 1.473 | 1.84 | 5.623 | 10 | 5.978 | 11.84 |
| S14 | 1.163 | 1.68 | 6.034 | 10 | 6.936 | 11.68 |
| S15 | 0.475 | 1.16 | 6.318 | 10 | 6.827 | 11.16 |

As seen from Table 6-38, the experiment results lie within the theoretical limits.

### 6.3.4.4 Gateway Performance: Effect of Polling

In this experiment, the solution that is proposed in section 6.3.4.3 to decrease the time duration that the signals stay in the Gateway is performed. The suggested solution is to utilize the polling mechanism more than once throughout the cycle. For this purpose two experiments are set up. While one of the experiments is utilizing the polling mechanism once in a cycle, in the other experiment, which is of the very same configuration with the first one, the incoming buffers are polled once in every 2.5 ms. The goal in performing these experiments is to compare the experiments against each other and visualize the effect of polling to the Gateway processing delay. The message set used in this experiment is same as that of the CAN Conventional Scheduling Experiment which is shown in Table 6-2. The priority assignment for the CAN signals are also the same and shown in Table 6-39.

**Table 6-39** CAN Priority Assignment for the Experiment

| Signal | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 |
|---|---|---|---|---|---|---|---|---|---|
| Priority | 1 | 0 | 4 | 3 | 2 | 5 | 6 | 12 | 11 |
| Signal | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 |
| Priority | 10 | 9 | 8 | 7 | 25 | 24 | 23 | 22 | 21 |
| Signal | C19 | C20 | C21 | C22 | C23 | C24 | C25 | C26 | |
| Priority | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | |

In this experiment, 3 CAN nodes and the Gateway node exchange the CAN messages that are shown in Table 6-39. The signals named C9, C10 and the signals from C14 to C26 are generated by the Gateway. These messages are sent to the Gateway through the FlexRay network from a single FlexRay node. Gateway does not forward any received CAN messages to the FlexRay network. Therefore, the

Gateway operates only in one direction namely, FlexRay-to-CAN. The mapping of the signals to the nodes is shown in Table 6-40.

**Table 6-40** Signal Distribution With Respect to the Nodes

| NODES | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| CAN1 | | CAN2 | | CAN3 | | Gateway | | FlexRay | |
| S | P | S | P | S | P | S | P | S | P |
| C1 | 1 | C2 | 0 | C3 | 4 | C9 | 11 | FR1 | 7 |
| C4 | 3 | C5 | 2 | C6 | 5 | C10 | 10 | FR2 | 8 |
| C7 | 6 | C8 | 12 | C11 | 9 | C14 | 25 | FR3 | 16 |
| C12 | 8 | C13 | 7 | | | C15 | 24 | FR4 | 17 |
| | | | | | | C16 | 23 | FR5 | 19 |
| | | | | | | C17 | 22 | FR6 | 20 |
| | | | | | | C18 | 21 | FR7 | 26 |
| | | | | | | C19 | 20 | FR8 | 28 |
| | | | | | | C20 | 19 | FR9 | 31 |
| | | | | | | C21 | 18 | FR10 | 33 |
| | | | | | | C22 | 17 | FR11 | 36 |
| | | | | | | C23 | 16 | FR12 | 37 |
| | | | | | | C24 | 15 | FR13 | 41 |
| | | | | | | C25 | 14 | FR14 | 47 |
| | | | | | | C26 | 13 | FR15 | 48 |

The other configurations that are used in the experiments are summarized in Table 6-41. Two distinct experiments are run with these parameters for about 120 seconds. The only difference between the experiments is in the polling mechanism such that, while in one of the experiments the message buffers are polled once in a cycle, in the second experiment the polling frequency is increased two times so as to see the effect of the polling in the delay that the signals experience in the Gateway. The comparative results for the end-to-end delay that the signals passing the Gateway experience is given in Table 6-42 and Figure 6-38.

**Table 6-41** Experiment Parameters

| Network Parameter | Value |
|---|---|
| FlexRay Cycle Length | 5 ms |
| Static Slot Length | 31 μs |
| Static Slot Number | 64 |
| Dynamic Segment Length | 0 |
| CAN Data Rate | 500 kbps |

**Table 6-42** E2E Delay for Polling: @ 5ms and @ 2.5 ms

| Signal | Delay @ 5 ms Polling (ms) | Delay @ 2.5 ms Polling (ms) | Delta in Delay (ms) |
|---|---|---|---|
| C9 | 6.784 | 4.217 | 2.567 |
| C10 | 6.602 | 4.189 | 2.413 |
| C14 | 5.703 | 3.27 | 2.433 |
| C15 | 5.819 | 3.328 | 2.491 |
| C16 | 5.821 | 3.275 | 2.546 |
| C17 | 5.822 | 3.354 | 2.468 |
| C18 | 5.999 | 3.526 | 2.473 |
| C19 | 6.089 | 3.584 | 2.505 |
| C20 | 6.151 | 3.557 | 2.594 |
| C21 | 6.29 | 3.61 | 2.68 |
| C22 | 6.3 | 3.787 | 2.513 |
| C23 | 6.38 | 3.841 | 2.539 |
| C24 | 6.39 | 3.811 | 2.579 |
| C25 | 6.4 | 3.871 | 2.529 |
| C26 | 6.445 | 4.043 | 2.402 |

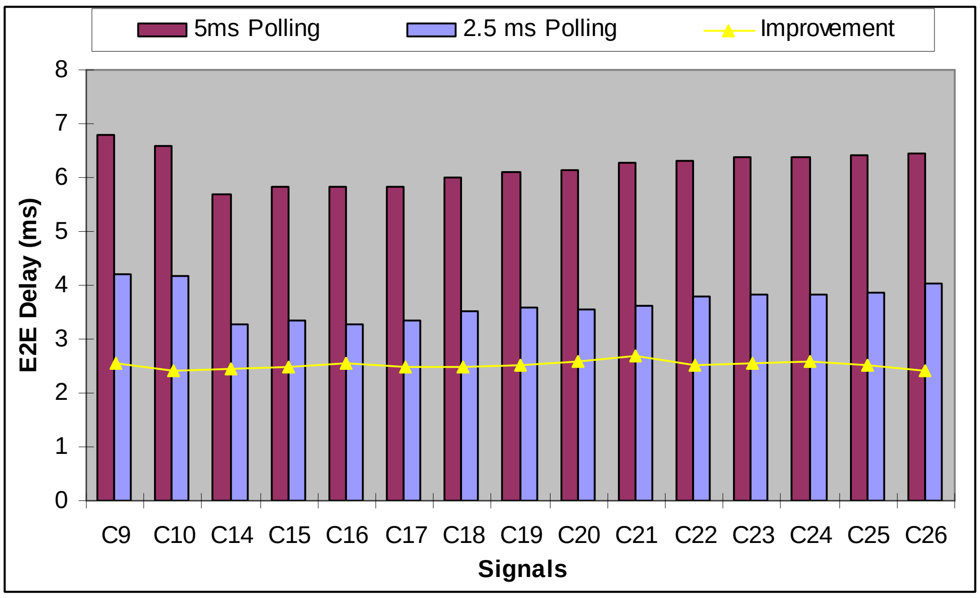


**Figure 6-38** Effect of Polling: @ 5ms and @ 2.5 ms

The results exhibited in Table 6-42 and Figure 6-38 were already expected. Because in the experiment where the incoming FlexRay buffers are polled once in a cycle, polling is done in the very beginning of the cycle. At the time of polling, buffers are observed to be empty since no data could be generated from the FlexRay node. Therefore, for this case, all the time when the buffers are polled, the data extracted from the buffers belong to the FlexRay messages which are generated in the previous cycle. As a result of this scheme, the FlexRay messages are received in the Gateway with an approximate delay of 5 ms when the polling is performed with a period of 5 ms. On the other hand, in the experiment where the polling is performed once in every 2.5 ms, the incoming buffers are polled once in the beginning of the cycle and for a second time, in the very middle. So, although the first polling sees the buffers empty, during the second polling the message data are extracted from the incoming buffers. Consequently, the messages in the Gateway which applies the polling with a period of 2.5 ms experience a delay about 2.5 ms. To sum up, this experiment verifies the suggestion proposed in section 6.3.4.3 that increasing the polling frequency in the Gateway improves the Gateway processing delay.

# CHAPTER 7

# CONCLUSION

In this thesis study, the performance of FlexRay-CAN networks inter-connected by a Gateway unit for in-vehicle communication is experimentally evaluated and the obtained results are exhibited with detailed discussions. Apart from the verification of the Gateway functionality, in particular, we focus on the flexibility of the Gateway implementation regarding the mapping of signals to messages and the worst-case response times encountered by signals that pass the Gateway including a Gateway processing delay. The end-to-end delay and the jitter that the signals experience, particularly, in FlexRay network, CAN network and the Gateway unit are examined by experiments with variety of different scheduling schemes each of which exhibits distinct characteristics in terms of performance metrics and practical applicability. All of these experiments are realized in real time hardware environment with realistic message sets depending on the message set provided by an automotive company.

We first focused on the behavior of the individual CAN and FlexRay networks so as to show the impact of scheduling of these networks to the overall performance in the sense of end-to-end delay and jitter. Therefore, CAN and FlexRay networks are elaborated respectively for different scheduling schemes with realistic message sets in order to exhibit the delay and the jitter performance.

The characteristics of CAN network is evaluated via three experiments, namely CAN Conventional Scheduling Experiment, CAN Prioritized Scheduling

Experiment and CAN Scheduling with Fixed Priorities Experiment. We show that the messages that pass the Gateway experience the biggest delay values in CAN Conventional Scheduling Experiment as expected since no special arrangement is made in this experiment to improve the performance of the signals crossing the Gateway. While the signals passing the Gateway experience the smallest delay in CAN Prioritized Scheduling Experiment, the performance of CAN Scheduling with Fixed Priorities Experiment falls between the two experiments in terms of experienced delay. The reason for this is that in CAN Prioritized Scheduling the signals crossing the Gateway are granted higher priorities (lower IDs), i.e. they are prioritized, with respect to the remaining CAN messages. On the other hand in the CAN Scheduling with Fixed Priorities Experiment, the signals that pass the Gateway are given higher priorities while some of the CAN signals are assumed to have fixed priorities. Because of this reason, the delay performance of the CAN Scheduling with Fixed Priorities is a little worse than that of the CAN Prioritized Scheduling. When the performances of the scheduling schemes are examined in the sense of experienced jitter, in spite of fluctuations, generally the jitter values that the signals passing the Gateway are greater in the CAN Conventional Scheduling than the other two scheduling schemes while the jitter in these two scheduling schemes are moreorless the same.

Then, we study the impact of scheduling in FlexRay network. The experiments held in FlexRay network are decomposed into two experiment sets since the FlexRay arbitration structure is composed of two distinct schemes, namely Static Segment and Dynamic Segment. In Static Segment experiments we evaluate the Static Segment performance in terms of delay and jitter with two scheduling schemes where the message set used is derived from the signals in a real vehicle. We apply the scheduling algorithms named FID Scheduling Without Jitter and FID Scheduling with Minimum FID. As its name implies, in FID Scheduling Without Jitter Experiment, the exchanged signals experience, fairly, no jitter. On the other hand, the signals that are exchanged in the FID Scheduling with Minimum FID experiment, experience jitter. However, in this experiment, the signals are achieved to be allocated in a number of FID which is much less than the FID number used in FID Scheduling Without Jitter Experiment. Therefore the bandwidth is utilized more efficiently in the FID Scheduling with Minimum FID experiment.

In Dynamic Segment experiment, we examine the schedulability of the dynamic segment with respect to the length of it in minislots by measuring the delay that the signals experience. We perform 3 experiments to see if the dynamic segment is schedulable with the minislot counts of 18, 19 and 20, respectively. We show that the message set used is not schedulable when the minislot number of the dynamic segment is 18 since the worst case delay that one of the messages experiences is greater than its deadline. On the other hand, when the experiments are held with the dynamic segment of 19 and 20 minislots respectively, it is observed that all of the messages are sent within their deadlines which signifies that the messages sets that are used in these experiments are schedulable.

Finally we conduct the experiments where the Gateway unit is also included. Out of four Gateway experiments, we examine the Gateway performance quantitatively in two experiments while the other two experiments are held qualitatively as the proof of concept of different functionalities of the Gateway. First of all, we verify the main functionality of the Gateway which is the protocol conversion between both networks. Second, we demonstrate the signal mapping capability of the Gateway which is the processing of the messages in signal level including the functionalities of message fragmentation and signal assembly. Next, we perform the first quantitative experiment where FlexRay-CAN networks are inter-connected via the Gateway unit. In this experiment, we show the end-to-end worst-case response time of the signals in the overall network with a large message set that is derived from the signals in a real vehicle as well as the Gateway processing delay. The worst-case end-to-end delay values are found to be within the signal deadlines and smaller than the theoretical maximum values. Also we show that the processing delay of the Gateway is 50 μs at maximum. The previous work on FlexRay-CAN gateway design mostly focus on the minimizing of the processing delay. However, we observe that although processing delay can have impact on the end-to-end delay values, the scheduling of the messages on both networks has the most significant effect.

Finally, we conduct an experiment to examine the time duration that the signals stay in the Gateway with respect to two different polling frequencies. We demonstrate that when the Gateway polling period is decreased from 5 ms to 2.5 ms, the duration that the signals stay in the Gateway decreases about 2.5 ms.

In this thesis, we provide the groundwork for the anticipated in-vehicle network architecture in the near future. That is to say, we develop a verified Gateway unit and the experimental performance analysis of possible scheduling approaches for interconnected FlexRay and CAN networks. The next stage in our research aims at developing selected x-by-wire applications such as automatic parking, steer-by-wire and break-by-wire. For this purpose, first the required FlexRay ECU's, their respective signals and messages and the appropriate scheduling approach will be determined. Then the signal exchange between the FlexRay ECUs and CAN ECUs for the envisaged application will be carried out via our Gateway unit.

# REFERENCES

[1] N. Navet, Y. Song, F. Simonot-Lion and C. Wilwert, “Trends in Automotive Communication Systems," Proceedings of the IEEE, vol. 93, no. 6, pp. 1204 -1223, June 2005.

[2] G. Leen and D. Heffernan, “Expanding Automotive Electronic Systems,” IEEE Comput., vol. 35, no. 1, pp. 88–93, Jan. 2002.

[3] Y. Martin, “L’avenir de l’automobile tient à un fil,” L’argus de l’automobile, vol. 3969, pp. 22–23, Mar. 2005.)

[4] K. Johansson, M. Törngren, and L. Nielsen, "Handbook of Networked and Embedded Control Systems," D. Hristu-Varsakelis and W. S. Levine, Eds. Boston, MA: Birkhäuser, 2005.

[5] F. Simonot-Lion, “In-car embedded electronic architectures: how to ensure their safety,” presented at the 5th IFAC Int. Conf. Fieldbus Systems and Their Applications (FeT 2003), Aveiro, Portugal, 2003.

[6] C.Wilwert, N. Navet, Y.-Q. Song, and F. Simonot-Lion, “Design of automotive X-by-Wire systems,” in The Industrial Communication Technology Handbook, R. Zurawski, Ed. Boca Raton, FL: CRC, 2004.

[7] M. Ayoubi, T. Demmeler, H. Leffler, and P. Köhn, “X-by-Wire functionality, performance and infrastructure,” presented at the Convergence Conf. 2004, Detroit, MI.

[8] J. Rushby, “A Comparison of Bus Architecture for Safety-Critical Embedded Systems,” NASA/CR, Tech. Rep. NASA/CR-2003- 212161, Mar. 2003.

[9] S. Poledna, W. Ettlmayr, and M. Novak, “Communication bus for automotive applications,” presented at the 27th Eur. Solid-State Circuits Conf., Villach, Austria, 2001.

[10] K. Ramaswamy and J. Cooper, “Delivering multimedia content to automobiles using wireless networks,” presented at the Convergence Conf. 2004, Detroit, MI.

[11] "CAN Specification 2.0," 2003, retrieved from http://www.can-cia.org/can/, Oct 18th, 2009.

[12] L. Hui, Z. Hao, P. Daogang, H. Wen, "Design and Application of Communication Gateway based on FlexRay and CAN," 2009 International Conference on Electronic Computer Technology, pp. 664-668, 2009.

[13] Road Vehicles—Interchange of Digital Information—Controller Area Network for High-Speed Communication, ISO 11 898, 1994.

[14] D. Paret, "Multiplexed Networks for Embedded Systems," pp. 25-242, John Wiley & Sons, 2007.

[15] G. Lima and A. Bums, “Timing-independent safety on top of CAN,” presented at the 1st Int. Workshop Real-Time LAN’s in the Internet Age, Vienna, Austria, 2002.

[16] J. Ferreira, L. Almeida, J. Fonseca, G. Rodriguez-Navas, and J. Proenza, “Enforcing consistency of communication requirements updates in FTT-CAN,” presented at the Int. Workshop Dependable Embedded Systems, Florence, Italy, 2003.

[17] G. Rodriguez-Navas and J. Proenza, “Clock synchronization in CAN distributed embedded systems,” presented at the 3rd Int. Workshop Real-Time Networks, Catania, Italy, 2004.

[18] G. Rodriguez-Navas, M. Barranco and J. Proenza, “Harmonizing dependability and real time in CAN networks,” presented at the 2nd Int.Workshop Real-Time LANs in the Internet Age, Porto, Portugal, 2003.

[19] FlexRay Consortium. (2004, Jun.) FlexRay Communication System, Protocol Specification, Version 2.0. [Online]. Available: http://www.flexray.com. Accessed on 15 March 2010.

[20] LIN Consortium. (2003, Sep.) LIN Specification Package, Revision 2.0. [Online]. Available: http://www.lin~subbus.org/. Accessed on 20 March 2010.

[21] A. Rajnák, The Industrial Communication Technology Handbook, R. Zurawski, Ed. Boca Raton, FL: CRC, 2005.

[22] A. Albert, “Comparison of event-triggered and time-triggered concepts with regards to distributed control systems,” presented at the Embedded World Conf. 2004, Nürnberg, Germany, 2004.

[23] A. Demirci, "Performance Evaluation of Flexray Networks for In-Vehicle Communication," Master Thesis, Ankara, Nov. 2009.

[24] S. Shaheen, D. Heffernan and G. Leen, "A gateway for time-triggered control networks," Microprocessors and Microsystems, vol. 31, no. 1, pp. 38-50, Agu. 2006.

[25] T. Lorenz, "Advanced Gateways in Automotive Applications," Ph.D. Thesis, Elektrotechnik und Informatik der Technische Universität Berlin, 2008.

[26] L.Hui, Z. Hao, P. Daogang and H. Wen, "Design and application of communication gateway based on FlexRay and CAN," International Conference on Electronic Computer Technology, ICECT 2009, pp. 664-668, 2009.

[27] T. Y. Moon, S. H. Seo and J. H. Kim, "Gateway system with diagnostic function for LIN, CAN and FlexRay," CCAS 2007 - International Conference on Control, Automation and Systems, pp. 2844 – 2849, 2007.

[28] T. Y. Moon, S. H. Seo, J. H. Kim, K. H. Kwon and J. W. Jeon, "A fault-tolerant gateway for in-vehicle networks," IEEE International Conference on Industrial Informatics (INDIN), pp. 1144-1148, 2008.

[29] T. Y. Moon, S. H. Seo, J. H. Kim, K. H. Kwon and J. W. Jeon, "An evaluation of the FlexRay-CAN gateway-embedded system in the HEV test bench," IEEE International Symposium on Industrial Electronics, 2009. ISIE 2009., pp. 664-669, 2009.

[30] E.G. Schmidt and K. Schmidt, "Development of a FlexRay-CAN Gateway," Technical Report, Chair of Automatic Control, University of Erlangen-Nuremberg, 2010.

[31] I. Standard-11898, “Road vehicles-interchange of digital information – Controller Area Network (CAN) for high-speed communication,” International Standards Organisation (ISO), 1993.

[32] (2009) Bosch E-Ray FlexRay IP-Module user’s manual. [Online]. Available: http://www.semiconductors.bosch.de/pdf/ERay Users Manual 1 2 6.pdf. Accessed on 15 January 2010.

[33] Fujitsu Microelectronics Europe (2007) Fujitsu FlexRay Driver Manual V1.3, Langen, Germany.

[34] E. G. Schmidt, M. Alkan, K. Schmidt, E. Yuruklu, and U. Karakaya, “Performance Evaluation of FlexRay/CAN Networks Interconnected by a Gateway,” IEEE - Symposium on Industrial Embedded Systems, SIES 2010 (Submitted).

# APPENDIX A

## GATEWAY EXPERIMENT SOURCE CODE: FLEXRAY.PRJ/MAIN.C

```
/*--------------------------------------------------------
                         MAIN.C
--------------------------------------------------------*/

/*******************@INCLUDE_START********************
#if (EMULATOR == 0)
#include "mb91465x.h"
#else
#include "mb91465x_emulator.h"
#endif
#include "global.h"
#include <ffrd_api_global.h>
#include <ffrd_fhal_read.h>
#include <ffrd_api_status_service.h>
#include "ffrd_api_time_service.h"
#include "ReloadTimer.h"
#include "data.h"
```

```
#include "TTASK.h"

#include "uart.h"

#include "print_status.h"

/**********************@INCLUDE_END******************/


#define TASK_OFFSET     100

#define TASK_OFFSET_MIN  50

#define TASK_OFFSET_MAX 150


/*****************@GLOBAL_VARIABLES_START*************/

FFRD_UINT8 nOSSyncStatus;

uint32_t nRCWD = 0;

uint16_t counter=0;

uint16_t r_number = 0;

FFRD_UINT16 nTime = 0;

FFRD_UINT16 nTime2 = 0;

FFRD_UINT16 cycle_no;

uint8_t start = 0;

static volatile FFRD_RETURN_TYPE statusSx1;

static volatile FFRD_RETURN_TYPE statusSx2;

static volatile FFRD_RETURN_TYPE statusSx3;

static volatile FFRD_RETURN_TYPE statusSx5;


typedef struct{

uint8_t Port;

FFRD_UINT8  c_counter; // cycle counter value ==> Ali
```

```
FFRD_UINT16 m_counter; // macrotic counter value ==> Ali
uint16_t empty[8];
}sporadic_content;

sporadic_content Sbuffer1;
sporadic_content Sbuffer2;
sporadic_content Sbuffer3;
sporadic_content Sbuffer5;
/*******************@GLOBAL_VARIABLES_END**************/

/*****************@FUNCTION_DECLARATION_START**********/
extern  void InitController(void);
static void InitCPUExtraRegs(void);
/*****************@FUNCTION_DECLARATION_END************/

static void InitCPUExtraRegs(void)
{
  HWWD = 0x10;      /* clear HW watchdog of MB91F465X */
  /* Port 16 and 25 are connected to LED at SK-91F467-
FLEXRAY Stareterkit */
  PDR16 = 0x00;   /* clear port data register */
  PFR16 = 0x00;   /* set port function to I/O port */
  DDR16 = 0x0F;   /* data direction 0..3: output */
  PDR27 = 0x00;   /* clear port data register */
  PFR27 = 0x00;   /* set port function to I/O port */
  DDR27 = 0x0F;   /* data direction 0..3: output */
  Init_rldtmr_0(62500u, 0x181A);
  Init_rldtmr_1(2500u, 0x081A);
  Init_rldtmr_2(5000u, 0x081A);  /* D1 10ms */
  Init_rldtmr_3(5000u, 0x081A);   /* D2 10ms */
  Init_rldtmr_4(10000u, 0x081A);  /* D3 20ms */
  Init_rldtmr_6(12500u, 0x081A);  /* D5 25ms */
  InitUart4();
  HWWD = 0x10;      /* clear HW watchdog of MB91F465X */
} /* eof InitCPUExtraRegs */

static void runTask(void)
{
   for (;;)
   {
      printFlexRayStatus();
   }
}

void main(void)
{
   __EI();                     /* enable interrupts */
   __set_il(31);               /* allow all levels */
   HWWD = 0x10;     /* clear HW watchdog of MB91F467D */
   PORTEN = 0x3;               /* enable I/O Ports */
   InitCPUExtraRegs();
   nRCWD = 1;   /* count up variable used in WD ISR */
```

```
    InitIrqLevels(); * init interrupts (intvect table) */
    HWWD = 0x10;     /* clear HW watchdog of MB91F467D */
    start_rldtmr_0();       /* start HW watchdog tick */
    nRCWD = 1;   /* count up variable used in WD ISR */
    ttStartupHook();  /* initialise FlexRay driver */
    nRCWD = 1;
    start_rldtmr_1();           /* start system tick */
    nRCWD = 1;
    start_rldtmr_2();
    nRCWD = 1;
    start_rldtmr_3();
    nRCWD = 1;
    start_rldtmr_4();
    nRCWD = 1;
    start_rldtmr_6();
    nRCWD = 1;
    nTime2 = ffrd_api_get_mtick();
    srand(nTime2);
    runTask();                  /* Idle Task */
    nRCWD = 1;
    ttShutdownHook(0);   /* shutdown FlexRay driver */
}

__interrupt void IsrReloadTimer0(void)       //500ms
{
    if (nRCWD > 0){
         nRCWD = 0;
         HWWD = 0x10;
         }
         TMCSR0_UF = 0;
}

__interrupt void IsrReloadTimer1(void)       // 5ms
{
  /* get FlexRay ClusterTime */
    nTime = ffrd_api_get_mtick();
    /* correct host offset */
    if (nTime >= TASK_OFFSET)
    {
       TMRLR1 = 2490u;
    }
    if (nTime <= TASK_OFFSET_MIN)
     {
     TMRLR1 = 2500u;
     }
     TMCSR1_UF = 0; // Reset Timer _ clear interrup flag

    if(start<=150)
    {
      task_Node1();
      cycle_no = ffrd_api_get_cycle();
      if (cycle_no == 0)
      {
           start++;
      }
    }

    if(start > 150)
    {
           if (counter%2 == 0)
           {
                tx8_data = 8;
                tx8_flag = 1;
                tx8_period = 10;
           }
           if (counter%2 == 0)
           {
                tx17_data = 17;
                tx17_flag = 1;
                tx17_period = 10;
           }
           if (counter%2 == 0)
           {
                tx18_data = 18;
                tx18_flag = 1;
                tx18_period = 10;
           }
```

```
if (counter%10 == 0)
{
     tx20_data = 20;
     tx20_flag = 1;
     tx20_period = 50;
}
if (counter%20 == 0)
{
     tx19_data = 19;
     tx19_flag = 1;
     tx19_period = 100;
}
if (counter%20 == 0)
{
     tx28_data = 28;
     tx28_flag = 1;
     tx28_period = 100;
}
if (counter%20 == 0)
{
     tx29_data = 29;
     tx29_flag = 1;
     tx29_period = 100;
}
if (counter%20 == 0)
{
        tx30_data = 30;
        tx30_flag = 1;
        tx30_period = 100;
}
if (counter%50 == 0)
{
        tx26_data = 26;
        tx26_flag = 1;
        tx26_period = 250;
}
if (counter%20 == 0)
{
        tx41_data = 41;
        tx41_flag = 1;
        tx41_period = 100;
}
if (counter%200 == 0)
{
        tx33_data = 33;
        tx33_flag = 1;
        tx33_period = 1000;
}
if (counter%200 == 0)
{
        tx34_data = 34;
        tx34_flag = 1;
```

```
     tx34_period = 1000;
}
if (counter%400 == 0)
{
     tx31_data = 31;
     tx31_flag = 1;
     tx31_period = 2000;
}
if (counter%400 == 0)
{
     tx32_data = 32;
     tx32_flag = 1;
     tx32_period = 2000;
}
if (counter%400 == 0)
{
     tx36_data = 36;
     tx36_flag = 1;
     tx36_period = 2000;
}
if (counter%400 == 0)
{
     tx37_data = 37;
     tx37_flag = 1;
     tx37_period = 2000;
}
```

```
            counter++;

            if (counter == 400)

            counter = 0;


            task_Node1();      /* start FlexRay Task */


   }//if(start >150) __END

}


__interrupt void IsrReloadTimer2(void)

{

       nRCWD = 1;

       Sbuffer1.Port = 1;  /* D1 */

       Sbuffer1.m_counter = ffrd_api_get_mtick();

       Sbuffer1.c_counter = ffrd_api_get_cycle();

       statusSx1                                        =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer1,   18,
19, FFRD_CHANNEL_A);

       statusSx1                                        =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer1,   18,
20, FFRD_CHANNEL_B);


     do {

     r_number = rand();


     } while (r_number <= 5000);
     TMRLR2 = r_number;

     TMCSR2_UF = 0;

     nRCWD = 1;

}


__interrupt void IsrReloadTimer3(void)

{

       nRCWD = 1;

       Sbuffer2.Port = 2;  /* D2 */

       Sbuffer2.m_counter = ffrd_api_get_mtick();

       Sbuffer2.c_counter = ffrd_api_get_cycle();

       statusSx2                                        =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer2,   12,
21, FFRD_CHANNEL_A);

       statusSx2                                        =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer2,   12,
22, FFRD_CHANNEL_B);


     do {

     r_number = rand();

     } while (r_number <= 5000);


     TMRLR3 = r_number;

     TMCSR3_UF = 0;

     nRCWD = 1;

}


__interrupt void IsrReloadTimer4(void)
```

```
{
       nRCWD = 1;
       Sbuffer3.Port = 3;  /* D3 */
       Sbuffer3.m_counter = ffrd_api_get_mtick();
       Sbuffer3.c_counter = ffrd_api_get_cycle();
       statusSx3                                    =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer3,    8,
23, FFRD_CHANNEL_A);
       statusSx3                                    =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer3,    8,
24, FFRD_CHANNEL_B);

     do {
     r_number = rand();
     } while (r_number <= 10000);

     TMRLR4 = r_number;
     TMCSR4_UF = 0;
     nRCWD = 1;
}

__interrupt void IsrReloadTimer6(void)
{
       nRCWD = 1;
       Sbuffer5.Port = 5;  /* D5 */
       Sbuffer5.m_counter = ffrd_api_get_mtick();
       Sbuffer5.c_counter = ffrd_api_get_cycle();
       statusSx5                                        =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer5,     4,
25, FFRD_CHANNEL_A);

       statusSx5                                        =
ffrd_api_tx_handler_buffer((FFRD_UINT32)&Sbuffer5,     4,
26, FFRD_CHANNEL_B);


     do {

     r_number = rand();

     } while (r_number <= 12500);


     TMRLR6 = r_number;

     TMCSR6_UF = 0;

     nRCWD = 1;

}
```

# APPENDIX B

## GATEWAY EXPERIMENT SOURCE CODE: GATEWAY.PRJ/MAIN.C

```
/*-------------------------------------------------------
  MAIN.C
  -----------------------------------------------------*/

/*********************@INCLUDE_START*****************/
#if (EMULATOR == 0)
#include "mb91465x.h"
#else
#include "mb91465x_emulator.h"
#endif
#include "global.h"
#include <ffrd_api_global.h>
#include <ffrd_fhal_read.h>
#include <ffrd_api_status_service.h>
#include "ffrd_api_time_service.h"
#include "ReloadTimer.h"
#include "TTASK.h"
```

```
#include "uart.h"
#include "print_status.h"
#include "CAN.h"
/*********************@INCLUDE_END*******************/

#define TASK_OFFSET     2500 //for 2,5 ms offset
#define TASK_OFFSET_MIN 2450
#define TASK_OFFSET_MAX 150

/*****************@GLOBAL_VARIABLES_START**************/
FFRD_UINT8 nOSSyncStatus;
uint32_t nRCWD = 0;
FFRD_UINT16 nTime = 0;
/*****************@GLOBAL_VARIABLES_END***************/

/***************@FUNCTION_DECLARATION_START***********/
extern  void InitController(void);
static void InitCPUExtraRegs(void);
/***************@FUNCTION_DECLARATION_END*************/

static void InitCPUExtraRegs(void)
{
  HWWD = 0x10;      /* clear HW watchdog of MB91F465X */
  /* Port 16 and 25 are connected to LED at SK-91F467-
FLEXRAY Stareterkit */
  PDR16 = 0x00;   /* clear port data register */
  PFR16 = 0x00;   /* set port function to I/O port */
  DDR16 = 0x0F;   /* data direction 0..3: output */
  PDR27 = 0x00;   /* clear port data register */
  PFR27 = 0x00;   /* set port function to I/O port */
  DDR27 = 0x0F;   /* data direction 0..3: output */

  Init_rldtmr_1(31250u, 0x181A);
  Init_rldtmr_3(2500u, 0x081A);
  InitUart4();
  InitCANCtrl0();
  HWWD = 0x10;
} /* eof InitCPUExtraRegs */

static void runTask(void)
{
   for (;;)
   {
      printFlexRayStatus();
   }
}

void main(void)
{
    __EI();                       /* enable interrupts */
    __set_il(31);                 /* allow all levels */
    HWWD = 0x10;
    PORTEN = 0x3;                 /* enable I/O Ports */
   InitCPUExtraRegs();
   nRCWD = 1;    /* count up variable used in WD ISR */
   InitIrqLevels(); * init interrupts (intvect table) */
   HWWD = 0x10;
   start_rldtmr_1();
   nRCWD = 1;
   ttStartupHook();  /* initialise FlexRay driver */
   nRCWD = 1;
   start_rldtmr_3();          /* start system tick */
   nRCWD = 1;
   runTask();                 /* Idle Task */
   nRCWD = 1;
   ttShutdownHook(0);      /* shutdown FlexRay driver */
}

__interrupt void IsrReloadTimer1(void)
{
   if (nRCWD > 0){
      nRCWD = 0;
      HWWD = 0x10;
  }
   TMCSR1_UF = 0;        /* clear Interrupt flag */
}
__interrupt void IsrReloadTimer3(void)       // 5ms
{
   /* get FlexRay ClusterTime */
   nTime = ffrd_api_get_mtick();
   /* correct host offset */
   if (nTime >= TASK_OFFSET)
   {
      TMRLR3 = 2490u;
   }
   if (nTime <= TASK_OFFSET_MIN)
    {
      TMRLR3 = 2500u;
    }
     TMCSR3_UF = 0; /* Reset Timer_clear interrup flag*/
     task_Node1();            /* start FlexRay Task */
}
```

# APPENDIX C

## GATEWAY EXPERIMENT SOURCE CODE: GATEWAY.PRJ/TTASK.C

```
/*-------------------------------------------------------
  TTASK.C
  -----------------------------------------------------*/

#include "TTASK.h"
#include <ffrd_api_global.h>
#include <ffrd_api_init_chi.h>
#include <ffrd_api_control_service.h>
#include <ffrd_api_tx_handler.h>
#include <ffrd_api_rx_handler.h>
#include <ffrd_api_status_service.h>
#include "ffrd_api_time_service.h"
#include "global.h"
#include "CAN.h"
#if (EMULATOR == 0)
#include "mb91465x.h"
#else
```

```
#include "mb91465x_emulator.h"
#endif
#include "uart.h"
#define NODE_NAME "Node1"

extern uint32_t nRCWD;
static volatile unsigned int nIdleTaskInvocations;
static volatile unsigned int nTaskInvocations;
static volatile FFRD_RETURN_TYPE statusRx1;
static volatile FFRD_RETURN_TYPE statusRx2;
static volatile FFRD_RETURN_TYPE statusRx3;
static volatile FFRD_RETURN_TYPE statusRx4;
FFRD_RX_BUFFER_HEADER_STRUCT header_rx1;

uint32_t data;
uint16_t period = 10;
uint8_t j = 0;
uint32_t cycle;
uint32_t macro_tick;
uint32_t period_for_shift;
uint32_t port_for_shift;
uint8_t rx_cycle;
FFRD_UINT8 old_c_counter = 0;
FFRD_UINT16 old_m_counter = 0;
int CANtx[5] = {0, 1, 8, 13, 19};
```

```
typedef struct{
uint16_t Port;
FFRD_UINT16 c_counter;
FFRD_UINT16 m_counter;
uint16_t period;
uint16_t  empty[1];
}data_content;

data_content sRx1;

typedef struct{
uint8_t Port;
FFRD_UINT8  c_counter;
FFRD_UINT16 m_counter;
uint16_t empty[8];
}sporadic_content;

sporadic_content Sbuffer1;

ttTASK(Node1)
{
    FFRD_POC_STATUS_TYPE poc_status;
    ++nTaskInvocations;
    ++nIdleTaskInvocations;
    nRCWD = 1;
    if(!(nTaskInvocations%100))
    {
        /* check if FlexRay CC is not sync */
        poc_status = ffrd_api_get_poc_status();
        /* check if FlexRay CC is not sync */
        if (poc_status != FFRD_POCS_NORMAL_ACTIVE)
        {
           if(!ffrd_api_pocs_is_halt())
            {
               /* if not sync, enter HALT state */
               ffrd_api_poc_command(FFRD_POCC_FREEZE);
            }
            /* enter DEFAULT_CONFIG state */
            ffrd_api_poc_command(FFRD_POCC_CONFIG);
            /* enter CONFIG state */
            ffrd_api_poc_command(FFRD_POCC_CONFIG);
            /* enter READY state */
            ffrd_api_poc_command(FFRD_POCC_READY);
            /* enter RUN state */
            ffrd_api_poc_command(FFRD_POCC_RUN);


ffrd_api_poc_command(FFRD_POCC_RESET_STATUS_INDICATORS);
/*   do   a   coldstart   or   integration   start   */
ffrd_api_poc_command(FFRD_POCC_ALLOW_COLDSTART);
        }
    }
     nRCWD = 1;
    /* Receive data */

          nRCWD = 1;

          /*P2*/

statusRx1=ffrd_api_rx_handler_buffer((FFRD_UINT32)&sRx1,
&header_rx1,        10,        0,        FFRD_CHANNEL_A,
ffrd_api_new_rx_data_buffer(0));

statusRx2=ffrd_api_rx_handler_buffer((FFRD_UINT32)&sRx1,
&header_rx1,        10,        1,        FFRD_CHANNEL_B,
ffrd_api_new_rx_data_buffer(1));


if ( statusRx1 == FFRD_OKAY || statusRx2 == FFRD_OKAY )

{

rx_cycle = ffrd_api_get_cycle();

cycle = sRx1.c_counter;

macro_tick = sRx1.m_counter;

period_for_shift = sRx1.period;

data = period_for_shift + (cycle<<8) + (macro_tick<<16);


CAN0_SendMessage(data, rx_cycle, 17, CANtx[0], 8);

CAN0_SendMessage(data, rx_cycle, 18, CANtx[1], 8);

}//end if for P2


/*D3*/


statusRx3=ffrd_api_rx_handler_buffer((FFRD_UINT32)&Sbuff
er1,    &header_rx1,    20,    8,    FFRD_CHANNEL_A,
ffrd_api_new_rx_data_buffer(8));
```

```
statusRx4=ffrd_api_rx_handler_buffer((FFRD_UINT32)&Sbuff
er1,    &header_rx1,    20,    9,    FFRD_CHANNEL_B,
ffrd_api_new_rx_data_buffer(9));


if ( statusRx3 == FFRD_OKAY || statusRx4 == FFRD_OKAY )

{

if(Sbuffer1.c_counter    !=    old_c_counter    ||
Sbuffer1.m_counter != old_m_counter)

{

rx_cycle = ffrd_api_get_cycle();

cycle = Sbuffer1.c_counter;

macro_tick = Sbuffer1.m_counter;

port_for_shift = Sbuffer1.Port;

data = port_for_shift + (cycle<<8) + (macro_tick<<16);


CAN0_SendMessage(data, rx_cycle, 19, CANtx[2], 8);

CAN0_SendMessage(data, rx_cycle, 20, CANtx[3], 8);

CAN0_SendMessage(data, rx_cycle, 21, CANtx[4], 8);


old_c_counter = Sbuffer1.c_counter;

old_m_counter = Sbuffer1.m_counter;

}

}//end if for D3

}


void ttErrorHook( int error )

{
}

void ttStartupHook(void)
{
    FFRD_RETURN_TYPE initController;
    nRCWD = 1;

    #if (EMULATOR == 0)   /* set PLL2 of MB91F465XA */
       PLL2DIVM = 1;
       PLL2DIVN = 0x13;
       PLL2DIVG = 0;
       PLL2MULG = 0;
       PLL2CLKR = 0x04; /* enable PLL, BCLCK & SCLK */
       /* wait for PLL Oscillaition stabilisation */
       TBCR = 0x08;     /* setup Timebase Timer */
       CTBR = 0x00;     /* clear TBT count register */
       while (!TBCR_TBIF)
       nRCWD = 1;   /* wait until timer finished */
       PLL2CLKR |= 0x02;   /* switch to PLL2 clock */
       EPFR31 = 0x77;    // set pin to FlexRay function
       PFR31 = 0x77; // Use FlexRay Function no I/O port
       DDR31 = 0x77;
    #endif
    initController = ffrd_api_init_chi();
    nRCWD = 1;
    if (initController != FFRD_OKAY)
    {
       ttShutdownHook(-1);
       PDR16 = 0xAA;
    }
    nRCWD = 1;
}

void ttShutdownHook( int error )
{
    putstr(4, "\n");
    putstr(4, "Node1 is shut down");
    putstr(4, "\n");
}
```

# APPENDIX D

# GATEWAY EXPERIMENT SOURCE CODE: GATEWAY.PRJ/CAN.C

```
/*******************************************************/
/** \file  CAN.C

/*******************************************************/

/*********************@INCLUDE_START*******************/
#include "CAN.h"
#include "uart.h"
#include "skwizard.h"
#include "global.h"
#include <ffrd_api_global.h>
#include "ffrd_api_time_service.h"
#include "TTASK.h"
#include <ffrd_api_tx_handler.h>
#include <ffrd_api_rx_handler.h>
#include <ffrd_api_status_service.h>
#include <ffrd_api_init_chi.h>
```

```
#include <ffrd_api_control_service.h>
/*********************@INCLUDE_END*******************/

/***************@GLOBAL_VARIABLES_START***************/
int8_t RxOK_Int;
int8_t TxOK_Int;
int8_t LEC_Int;
uint32_t nCycle;
uint32_t mtick;
uint16_t IntPointer = 0x0000;
uint16_t IntBuffer;
uint32_t can_send_cycle;
uint32_t can_send_mtick;
uint32_t ID;
uint32_t fr_rx_cycle_forshift;
unsigned char sth_came_3129=0;
unsigned char sth_came_2118=0;
unsigned char sth_came_1109=0;
unsigned char sth_came_0605=0;
unsigned char sth_came_0402=0;
FFRD_UINT8 prev_cycle3129=0;
FFRD_UINT8 prev_cycle2118=0;
FFRD_UINT8 prev_cycle1109=0;
FFRD_UINT8 prev_cycle0605=0;
FFRD_UINT8 prev_cycle0402=0;
unsigned char token_3129=31;
unsigned char token_2118=21;
unsigned char token_1109=11;
unsigned char token_0605=6;
unsigned char token_0402=4;
static volatile FFRD_RETURN_TYPE statusTx5;
static volatile FFRD_RETURN_TYPE statusTx7;
static volatile FFRD_RETURN_TYPE statusTx10;
static volatile FFRD_RETURN_TYPE statusTx16;
static volatile FFRD_RETURN_TYPE statusTx4;

typedef struct{
uint16_t Port;
FFRD_UINT16 can_send_cycle;
FFRD_UINT16 can_send_mtick;
FFRD_UINT16 can_rx_cycle;
FFRD_UINT16 can_rx_mtick;
}gateway_content;

gateway_content buffer7;
gateway_content buffer5;
gateway_content buffer10;
gateway_content buffer16;

typedef struct{
uint16_t Port;
FFRD_UINT16 can_send_cycle;
```

```
FFRD_UINT16 can_send_mtick;
FFRD_UINT16 can_rx_cycle;
FFRD_UINT16 can_rx_mtick;
uint16_t  empty[5];
}gateway_content_dyn;

gateway_content_dyn buffer4;

/*******************@GLOBAL_VARIABLES_END**************/

void InitCANCtrl0(void)
{
  int16_t bufcnt;
  PFR23_D0 = 1;                /* RX */
  PFR23_D1 = 1;                /* TX */

  CTRLR0_Init = 1;             /* Stop CAN operation */
  IF1ARB120 = 0x00000000;
  IF1MSK120 = 0x00000000;
  IF1MCTR0 = 0x0080;          /* only EOB-Flag is set */
  IF1DTA120 = 0x00000000;
  IF1DTB120 = 0x00000000;

  IF1CMSK0_WR = 1;
  IF1CMSK0_Mask = 1;
  IF1CMSK0_Arb = 1;
  IF1CMSK0_Control = 1;        /* Tx request NOT set */
  IF1CMSK0_TxReq = 0;
  IF1CMSK0_DataA = 1;
  IF1CMSK0_DataB = 1;

  for(bufcnt=1; bufcnt<=MAXBUF; bufcnt++)
  {
    IF1CREQ0 = bufcnt;/*xfer the IF content to buffer */
  }

  CTRLR0_CCE = 1;              /* enable cfg change */
  BTR0 = BTR_16M_500k_16_68_3; /*BTR config 500 kBaud */
  CTRLR0_CCE = 0;              /* disable cfg change */
  CTRLR0_EIE = 1;           /* enable error interrupt */
  CTRLR0_SIE = 1;   /* enable status change interrupt */
  CTRLR0_IE = 1;    /* enable interrupt generation */
  CTRLR0_Init = 0;  /* complete init, start CAN */

/* Config CAN0 Buffer 1-16 as Rx, Rx>16 will not work */
  IF1ARB120 = 2;
  IF1ARB20_Xtd = 1;            /* 29bit ID */
  IF1ARB20_DIR = 0;            /* Rx buffer */
  IF1ARB20_MsgVal = 1;         /* buffer set as valid */

  IF1MSK120 = 0x1fffffff;      /* mask all ID */
  IF1MSK20_MDir = 0;          /* do not mask Dir flag */
```

```
IF1MSK20_MXtd = 1;            /* mask ID type flag */
IF1MCTR0_NewDat = 0;          /* clear NewDat flag */
IF1MCTR0_MsgLst = 0;          /* clear MsgLst flag */
IF1MCTR0_IntPnd = 0;          /* clear IntPnd flag */
IF1MCTR0_UMask = 1;           /* use Mask Filter */
IF1MCTR0_TxIE = 0;
IF1MCTR0_RxIE = 1;
IF1MCTR0_RmtEn = 0;
IF1MCTR0_TxRqst = 0;
IF1MCTR0_EoB = 1;

IF1CMSK0_WR = 1;
IF1CMSK0_Mask = 1;
IF1CMSK0_Arb = 1;
IF1CMSK0_Control = 1;
IF1CMSK0_TxReq = 0;
IF1CMSK0_DataA = 0;
IF1CMSK0_DataB = 0;

IF1CREQ0 = 1;
   …
   The buffer config is same for other Rx buffers, so
   omitted
   …
```

/* Config CAN0 Buffer 17-21 as Tx, Tx>16 will work */

```
IF1ARB120 = MSG2STD(0x02);
IF1ARB20_Xtd = 1;             /* 29bit ID */
```

```
IF1ARB20_DIR = 1;            /* Tx buffer */
IF1ARB20_MsgVal = 1;      /* buffer set as valid */

IF1MSK120 = 0x1fffffff;      /* accept all ID */
IF1MSK20_MDir = 1;           /* mask Dir flag */
IF1MSK20_MXtd = 1;           /* mask ID type flag */
IF1MCTR0_NewDat = 0;         /* clear NewDat flag */
IF1MCTR0_MsgLst = 0;         /* clear MsgLst flag */
IF1MCTR0_IntPnd = 0;         /* clear IntPnd flag */
IF1MCTR0_UMask = 1;          /* use Mask Filter */
IF1MCTR0_TxIE = 0;
IF1MCTR0_RxIE = 1;
IF1MCTR0_RmtEn = 0;
IF1MCTR0_TxRqst = 0;
IF1MCTR0_EoB = 1;
IF1CMSK0_WR = 1;
IF1CMSK0_Mask = 1;
IF1CMSK0_Arb = 1;
IF1CMSK0_Control = 1;
IF1CMSK0_TxReq = 0;
IF1CMSK0_DataA = 0;
IF1CMSK0_DataB = 0;

IF1CREQ0 = 17;
…
  The buffer config is same for other Tx buffers, so
  omitted
```

…

```
void CAN0_ReadMessageBuffer(unsigned char buffer)
{
  nCycle = ffrd_api_get_cycle();
  mtick = ffrd_api_get_mtick();

/* receive Control Info, Msg data and Arbitration
from Msg Buffer */

  IF1CMSK0_WR = 0;
  IF1CMSK0_Mask = 0;
  IF1CMSK0_Arb = 1;
  IF1CMSK0_Control = 1;
IF1CMSK0_CIP = 1; /*clear pending Int by reading */
  IF1CMSK0_TxReq = 1;
  IF1CMSK0_DataA = 1;
  IF1CMSK0_DataB = 1;

  IF1CREQ0 = buffer;        /* start transfer */
  if(IF1MCTR0_MsgLst)       /* in case msg lost */
  {
    IF1MCTR0_MsgLst = 0;
    IF1CMSK0_WR = 1;
    IF1CMSK0_Mask = 0;
    IF1CMSK0_Arb = 0;
    IF1CMSK0_Control = 1;
    IF1CMSK0_CIP = 0;
```

```
            IF1CMSK0_TxReq = 0;
            IF1CMSK0_DataA = 0;
            IF1CMSK0_DataB = 0;
            IF1CREQ0 = buffer;
          }

          can_send_cycle = 0x000000FF&IF1DTA120;
          can_send_mtick = (0x00FFFF00&IF1DTA120)>>8;
          ID = 0x1FFFFFFF&IF1ARB120;

if (ID==31)  //C14 or C15 to P5
{
     if(sth_came_3129==1)
     {
     if(nCycle==prev_cycle3129)//second one has come in 
the same cycle
          {
          sth_came_3129 = 0;
          if(token_3129==31)
          {
          buffer5.Port = 31;
          buffer5.can_send_cycle = can_send_cycle;
          buffer5.can_send_mtick = can_send_mtick;
          buffer5.can_rx_cycle = nCycle;
          buffer5.can_rx_mtick = mtick;
statusTx5=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r5, 10, 2, FFRD_CHANNEL_A_B);
```

```
         token_3129=29;
          }
               else
               {
               token_3129=31;
               }
          }//if(nCycle==prev_cycle) __END

          else
          {
          prev_cycle3129 = nCycle;

               if(token_3129==31)
               {
               buffer5.Port = 31;
          buffer5.can_send_cycle = can_send_cycle;
          buffer5.can_send_mtick = can_send_mtick;
          buffer5.can_rx_cycle = nCycle;
          buffer5.can_rx_mtick = mtick;
statusTx5=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r5, 10, 2, FFRD_CHANNEL_A_B);
          }
          }
  }//     if(sth_came_3129==1) __END

  else
  {
```

```
  sth_came_3129 = 1;

  prev_cycle3129 = nCycle;


    if(token_3129==31)

     {

     buffer5.Port = 31;

    buffer5.can_send_cycle = can_send_cycle;

    buffer5.can_send_mtick = can_send_mtick;

    buffer5.can_rx_cycle = nCycle;

    buffer5.can_rx_mtick = mtick;


statusTx5=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r5, 10, 2, FFRD_CHANNEL_A_B);

    }

 }

}//if (ID==31) __END


       if (ID==29)   //C14 or C15 to P5

       {

          if(sth_came_3129==1)

          {

     if(nCycle==prev_cycle3129)//second one has come in
     the same cycle

          {

          sth_came_3129 = 0;

                    if(token_3129==29)

                    {

     buffer5.Port = 29;
     buffer5.can_send_cycle = can_send_cycle;

     buffer5.can_send_mtick = can_send_mtick;

     buffer5.can_rx_cycle = nCycle;

     buffer5.can_rx_mtick = mtick;

statusTx5=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r5, 10, 2, FFRD_CHANNEL_A_B);

                    token_3129=31;

                    }


                    else

                    {

                    token_3129=29;

                    }

               }//if(nCycle==prev_cycle) __END



  else

       {

       prev_cycle3129 = nCycle;

     if(token_3129==29)

     {

     buffer5.Port = 29;

     buffer5.can_send_cycle = can_send_cycle;

     buffer5.can_send_mtick = can_send_mtick;

     buffer5.can_rx_cycle = nCycle;

     buffer5.can_rx_mtick = mtick;
```

```
statusTx5=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r5, 10, 2, FFRD_CHANNEL_A_B);
     }
     }


    }//if(sth_came_3129==1)
     else
     {
     sth_came_3129 = 1;
     prev_cycle3129 = nCycle;
     if(token_3129==29)
          {
     buffer5.Port = 29;
     buffer5.can_send_cycle = can_send_cycle;
     buffer5.can_send_mtick = can_send_mtick;
     buffer5.can_rx_cycle = nCycle;
     buffer5.can_rx_mtick = mtick;
statusTx5=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r5, 10, 2, FFRD_CHANNEL_A_B);
     }
    }
  }//if (ID==29) __END


  if (ID==21)   //C16 or C18 to P7
  {
     if(sth_came_2118==1)
     {
```

```
     if(nCycle==prev_cycle2118)//second one has come in
the same cycle
          {
          sth_came_2118 = 0;
               if(token_2118==21)
               {
     buffer7.Port = 21;
     buffer7.can_send_cycle = can_send_cycle;
     buffer7.can_send_mtick = can_send_mtick;
     buffer7.can_rx_cycle = nCycle;
     buffer7.can_rx_mtick = mtick;
statusTx7=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r7, 10, 4, FFRD_CHANNEL_A_B);
          token_2118=18;
          }
          else
          {
          token_2118=21;
          }
          }//if(nCycle==prev_cycle) __END


     else
     {
     prev_cycle2118 = nCycle;
     if(token_2118==21)
     {
     buffer7.Port = 21;
     buffer7.can_send_cycle = can_send_cycle;

     buffer7.can_send_mtick = can_send_mtick;

     buffer7.can_rx_cycle = nCycle;

     buffer7.can_rx_mtick = mtick;

statusTx7=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r7, 10, 4, FFRD_CHANNEL_A_B);

          }

     }

    }//if(sth_came_2118==1)


     else

     {

     sth_came_2118 = 1;

     prev_cycle2118 = nCycle;

    if(token_2118==21)

          {

          buffer7.Port = 21;

          buffer7.can_send_cycle = can_send_cycle;

          buffer7.can_send_mtick = can_send_mtick;

          buffer7.can_rx_cycle = nCycle;

          buffer7.can_rx_mtick = mtick;

statusTx7=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r7, 10, 4, FFRD_CHANNEL_A_B);

     }

    }

  }//if (ID==21) __END
  if (ID==18)   //C16 or C18 to P7
 {
     if(sth_came_2118==1)
     {
if(nCycle==prev_cycle2118)//second one has come in the
same cycle
          {
          sth_came_2118 = 0;
          if(token_2118==18)
          {
          buffer7.Port = 18;
          buffer7.can_send_cycle = can_send_cycle;
          buffer7.can_send_mtick = can_send_mtick;
          buffer7.can_rx_cycle = nCycle;
          buffer7.can_rx_mtick = mtick;
statusTx7=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r7, 10, 4, FFRD_CHANNEL_A_B);
     token_2118=21;
       }


     else
          {
          token_2118=18;
          }
          }//if(nCycle==prev_cycle) __END


     else
```

```
          {
          prev_cycle2118 = nCycle;
               if(token_2118==18)
               {
          buffer7.Port = 18;
          buffer7.can_send_cycle = can_send_cycle;
          buffer7.can_send_mtick = can_send_mtick;
          buffer7.can_rx_cycle = nCycle;
          buffer7.can_rx_mtick = mtick;
statusTx7=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r7, 10, 4, FFRD_CHANNEL_A_B);
          }
     }
    }//if(sth_came_2118==1)


     else
     {
     sth_came_2118 = 1;
     prev_cycle2118 = nCycle;
          if(token_2118==18)
          {
          buffer7.Port = 18;
          buffer7.can_send_cycle = can_send_cycle;
          buffer7.can_send_mtick = can_send_mtick;
          buffer7.can_rx_cycle = nCycle;
          buffer7.can_rx_mtick = mtick;
```

```
statusTx7=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r7, 10, 4, FFRD_CHANNEL_A_B);
     }
    }

  }//if (ID==18) __END

  if (ID==11)   //C20 or C21 to P10
    {
     if(sth_came_1109==1)
     {
if(nCycle==prev_cycle1109)//second one has come in the
same cycle
          {
          sth_came_1109 = 0;
               if(token_1109==11)
               {
          buffer10.Port = 11;
          buffer10.can_send_cycle = can_send_cycle;
          buffer10.can_send_mtick = can_send_mtick;
          buffer10.can_rx_cycle = nCycle;
          buffer10.can_rx_mtick = mtick;
statusTx10=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er10, 10, 3, FFRD_CHANNEL_A_B);
          token_1109=9;
          }
          else
          {
```

```
                    token_1109=11;
               }
               }//if(nCycle==prev_cycle) __END
               else
               {
               prev_cycle1109 = nCycle;
               if(token_1109==11)
               {
               buffer10.Port = 11;
               buffer10.can_send_cycle = can_send_cycle;
               buffer10.can_send_mtick = can_send_mtick;
               buffer10.can_rx_cycle = nCycle;
               buffer10.can_rx_mtick = mtick;
statusTx10=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er10, 10, 3, FFRD_CHANNEL_A_B);
               }
          }


         }//if(sth_came_1109==1)


          else
          {
          sth_came_1109 = 1;
          prev_cycle1109 = nCycle;
               if(token_1109==11)
               {
          buffer10.Port = 11;
```

```
      buffer10.can_send_cycle = can_send_cycle;

      buffer10.can_send_mtick = can_send_mtick;

      buffer10.can_rx_cycle = nCycle;

      buffer10.can_rx_mtick = mtick;


statusTx10=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er10, 10, 3, FFRD_CHANNEL_A_B);

      }

     }

  }//if (ID==11) __END


  if (ID==9)   //C20 or C21 to P10

    {

     if(sth_came_1109==1)

     {

if(nCycle==prev_cycle1109)//second one has come in the
same cycle

          {

          sth_came_1109 = 0;

               if(token_1109==9)

               {

          buffer10.Port = 9;

          buffer10.can_send_cycle = can_send_cycle;

          buffer10.can_send_mtick = can_send_mtick;

         buffer10.can_rx_cycle = nCycle;

         buffer10.can_rx_mtick = mtick;

     statusTx10=ffrd_api_tx_handler_buffer((FFRD_UINT32)
     &buffer10, 10, 3, FFRD_CHANNEL_A_B);
```

```
          token_1109=11;
          }
          else
               {
               token_1109=9;
               }
        }//if(nCycle==prev_cycle) __END
   else
          {
          prev_cycle1109 = nCycle;


       if(token_1109==9)
          {
          buffer10.Port = 9;
         buffer10.can_send_cycle = can_send_cycle;
         buffer10.can_send_mtick = can_send_mtick;
         buffer10.can_rx_cycle = nCycle;
         buffer10.can_rx_mtick = mtick;
statusTx10=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er10, 10, 3, FFRD_CHANNEL_A_B);
          }
     }
   }//if(sth_came_1109==1)
  else
     {
     sth_came_1109 = 1;
     prev_cycle1109 = nCycle;
     if(token_1109==9)

     {

     buffer10.Port = 9;

    buffer10.can_send_cycle = can_send_cycle;

    buffer10.can_send_mtick = can_send_mtick;

    buffer10.can_rx_cycle = nCycle;

    buffer10.can_rx_mtick = mtick;

statusTx10=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er10, 10, 3, FFRD_CHANNEL_A_B);

     }

    }

   }//if (ID==9) __END


if (ID==6)   //C23 or C24 to P16

    {

     if(sth_came_0605==1)

     {

if(nCycle==prev_cycle0605)//second one has come in the
same cycle

     {

     sth_came_0605 = 0;

     if(token_0605==6)

          {

          buffer16.Port = 6;

         buffer16.can_send_cycle = can_send_cycle;

         buffer16.can_send_mtick = can_send_mtick;

         buffer16.can_rx_cycle = nCycle;
         buffer16.can_rx_mtick = mtick;

statusTx16=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er16, 10, 5, FFRD_CHANNEL_A_B);

          token_0605=5;

          }

     else

          {

          token_0605=6;

          }

     }//if(nCycle==prev_cycle) __END


     else

          {

          prev_cycle0605 = nCycle;

          if(token_0605==6)

          {

          buffer16.Port = 6;

         buffer16.can_send_cycle = can_send_cycle;

         buffer16.can_send_mtick = can_send_mtick;

         buffer16.can_rx_cycle = nCycle;

         buffer16.can_rx_mtick = mtick;

statusTx16=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er16, 10, 5, FFRD_CHANNEL_A_B);

          }

     }

  }//if(sth_came_0605==1)
```

```
     else
     {
     sth_came_0605 = 1;
     prev_cycle0605 = nCycle;
     if(token_0605==6)
          {
          buffer16.Port = 6;
         buffer16.can_send_cycle = can_send_cycle;
         buffer16.can_send_mtick = can_send_mtick;
         buffer16.can_rx_cycle = nCycle;
         buffer16.can_rx_mtick = mtick;
statusTx16=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er16, 10, 5, FFRD_CHANNEL_A_B);
     }
    }
  }//if (ID==6) __END


if (ID==5)   //C23 or C24 to P16
 {
     if(sth_came_0605==1)
     {
if(nCycle==prev_cycle0605)//second one has come in the
same cycle
          {
          sth_came_0605 = 0;
               if(token_0605==5)
               {
```

```
          buffer16.Port = 5;
         buffer16.can_send_cycle = can_send_cycle;
         buffer16.can_send_mtick = can_send_mtick;
         buffer16.can_rx_cycle = nCycle;
         buffer16.can_rx_mtick = mtick;
statusTx16=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er16, 10, 5, FFRD_CHANNEL_A_B);


          token_0605=6;
     }
          else
          {
          token_0605=5;
          }
       }//if(nCycle==prev_cycle) __END


     else
          {
          prev_cycle0605 = nCycle;


          if(token_0605==5)
          {
               buffer16.Port = 5;
              buffer16.can_send_cycle = can_send_cycle;
              buffer16.can_send_mtick = can_send_mtick;
              buffer16.can_rx_cycle = nCycle;
              buffer16.can_rx_mtick = mtick;
```

```
statusTx16=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er16, 10, 5, FFRD_CHANNEL_A_B);

          }

     }

   }//if(sth_came_0605==1)


     else

     {

     sth_came_0605 = 1;

     prev_cycle0605 = nCycle;


     if(token_0605==5)

          {

          buffer16.Port = 5;

         buffer16.can_send_cycle = can_send_cycle;

         buffer16.can_send_mtick = can_send_mtick;

         buffer16.can_rx_cycle = nCycle;

         buffer16.can_rx_mtick = mtick;

statusTx16=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buff
er16, 10, 5, FFRD_CHANNEL_A_B);

     }

    }

}//if (ID==5) __END


     if (ID==4)   //C25 or C26 to D4

    {

     if(sth_came_0402==1)
     {
if(nCycle==prev_cycle0402)//second one has come in the
same cycle
     {
     sth_came_0402 = 0;
     if(token_0402==4)
          {
     buffer4.Port = 4;
     buffer4.can_send_cycle = can_send_cycle;
     buffer4.can_send_mtick = can_send_mtick;
     buffer4.can_rx_cycle = nCycle;
     buffer4.can_rx_mtick = mtick;
statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 6, FFRD_CHANNEL_A);
statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 7, FFRD_CHANNEL_B);
     token_0402=2;
}


     else
          {
          token_0402=4;
          }
      }//if(nCycle==prev_cycle) __END


else
     {
     prev_cycle0402 = nCycle;
```

```
          if(token_0402==4)
          {
               buffer4.Port = 4;
               buffer4.can_send_cycle = can_send_cycle;
               buffer4.can_send_mtick = can_send_mtick;
               buffer4.can_rx_cycle = nCycle;
               buffer4.can_rx_mtick = mtick;
     statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
     r4, 12, 6, FFRD_CHANNEL_A);
     statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
     r4, 12, 7, FFRD_CHANNEL_B);
         }
     }


     }//if(sth_came_0402==1)


          else
          {
          sth_came_0402 = 1;
          prev_cycle0402 = nCycle;
          if(token_0402==4)
               {
               buffer4.Port = 4;
              buffer4.can_send_cycle = can_send_cycle;
              buffer4.can_send_mtick = can_send_mtick;
              buffer4.can_rx_cycle = nCycle;
              buffer4.can_rx_mtick = mtick;
```

```
statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 6, FFRD_CHANNEL_A);

statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 7, FFRD_CHANNEL_B);

     }

    }


  }//if (ID==4) __END


     if (ID==2)   //C25 or C26 to D4

     {

     if(sth_came_0402==1)

     {

  if(nCycle==prev_cycle0402)//second one has come in the
same cycle

     {

     sth_came_0402 = 0;

     if(token_0402==2)

          {

               buffer4.Port = 2;

              buffer4.can_send_cycle = can_send_cycle;

              buffer4.can_send_mtick = can_send_mtick;

              buffer4.can_rx_cycle = nCycle;

              buffer4.can_rx_mtick = mtick;

statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 6, FFRD_CHANNEL_A);

statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 7, FFRD_CHANNEL_B);
```

```
      token_0402=4;
      }


else
      {
      token_0402=2;
      }
   }//if(nCycle==prev_cycle) __END


else
      {
      prev_cycle0402 = nCycle;
      if(token_0402==2)
            {
            buffer4.Port = 2;
            buffer4.can_send_cycle = can_send_cycle;
            buffer4.can_send_mtick = can_send_mtick;
            buffer4.can_rx_cycle = nCycle;
            buffer4.can_rx_mtick = mtick;
statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 6, FFRD_CHANNEL_A);
statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 7, FFRD_CHANNEL_B);
      }
   }
}//if(sth_came_0402==1)
```

```
     else
     {
     sth_came_0402 = 1;
     prev_cycle0402 = nCycle;
     if(token_0402==2)
          {
          buffer4.Port = 2;
         buffer4.can_send_cycle = can_send_cycle;
         buffer4.can_send_mtick = can_send_mtick;
         buffer4.can_rx_cycle = nCycle;
         buffer4.can_rx_mtick = mtick;
statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 6, FFRD_CHANNEL_A);
statusTx4=ffrd_api_tx_handler_buffer((FFRD_UINT32)&buffe
r4, 12, 7, FFRD_CHANNEL_B);
     }
    }


  }//if (ID==2) __END
}


int CAN0_SendMessage(uint32_t data, uint8_t fr_rx_cycle,
unsigned char buffer, int id, unsigned char dlc)
{
  uint32_t timeout = 0;
     IF1ARB120 = id;
     IF1ARB20_Xtd = 1;
     IF1ARB20_DIR = 1;
```

```
IF1ARB20_MsgVal = 1;


  IF1MSK120 = 0x1fffffff;

  IF1MSK20_MDir = 1;

  IF1MSK20_MXtd = 1;

  IF1MCTR0_NewDat = 0;

  IF1MCTR0_MsgLst = 0;

  IF1MCTR0_IntPnd = 0;

  IF1MCTR0_UMask = 1;

  IF1MCTR0_TxIE = 0;

  IF1MCTR0_RxIE = 0;

  IF1MCTR0_RmtEn = 0;

  IF1MCTR0_TxRqst = 1;

  IF1MCTR0_EoB = 1;


  IF1MCTR0_DLC = dlc;

  IF1DTA120 = data;


  nCycle = ffrd_api_get_cycle();

  mtick = ffrd_api_get_mtick();

  fr_rx_cycle_forshift = fr_rx_cycle;

  IF1DTB120     =     nCycle     +     (mtick<<8)     +
(fr_rx_cycle_forshift<<24);


  IF1CMSK0_WR = 1;

  IF1CMSK0_Mask = 1;

  IF1CMSK0_Arb = 1;
```

```
        IF1CMSK0_Control = 1;

        IF1CMSK0_TxReq = 0;

        IF1CMSK0_DataA = 1;

        IF1CMSK0_DataB = 1;


  IF1CREQ0 = buffer;


while((TREQR120   &   (0x1   <<   (buffer-1))   !=   0)   &&
(timeout++ < TIMEOUT) && (CTRLR0_Init != 1));

if((timeout == TIMEOUT) || (CTRLR0_Init == 1))     /* the
following code clears TxRqst bit */

  {

    IF1CMSK0_WR = 0;

    IF1CMSK0_Mask = 0;

    IF1CMSK0_Arb = 0;

    IF1CMSK0_Control = 1; /* because TxRqst is a Control
bit! (MCTR) */

    IF1CMSK0_TxReq = 0;

    IF1CMSK0_DataA = 0;

    IF1CMSK0_DataB = 0;

    IF1CREQ0 = buffer;

    IF1MCTR0_TxRqst = 0;

    IF1CMSK0_WR = 1;

    IF1CREQ0 = buffer;

    return 0;     /* Tx failed! */

  }

  return 1;       /* Tx succedded */

}
```

```
__interrupt void CAN0_ISR(void)

{

     HWWD_CL = 0;

     IntPointer = INTR0;

if(  (IntPointer  &  0x8000)  ==  0x8000)/*  is  Status
Interrupt */

     {

     CAN0_STATUS_ISR_Handler();

     /* IRQ should be cleared here */

     }

     else /* is message buffer interrupt */

     {

     IntPointer  =  IntPointer  &  0x00FF;/*  use  only  the
lower six bits */

if( (IntPointer >= 1) && (IntPointer <= 0x80) ) /* valid
buffer number */

     {

     IntBuffer = 0x01 << (IntPointer-1);

          {

/* Check whether the interrupt source is a valid buffer
*/

     if( (MSGVAL120 & IntBuffer) != 0) /* message buffer
is valid */

     {

     /* Check whether the interrupt cause is recieve or
transmit */

          if(  (NEWDT120  &  IntBuffer)  !=  0  )  /*  is  a
recieve interrupt */

               {
```

```
     /* call the recieve handler */

     CAN0_ReadMessageBuffer(IntPointer);

     /* Clear Newdat and pending Int */

     }

     else /* is a transmit interrupt */

               {

          /* call the transmit handler */

          } /* end else "is a transmit interrupt" */

          } /* end if "message buffer is valid" */

          }

          } /* end if "valid buffer number" */

} /* end else "is message buffer interrupt" */

}


void CAN0_STATUS_ISR_Handler()

{

     unsigned short int canstatus;

/* Read the Status Register (this operation will clear
pending Status/Error Interrupt ) */

     canstatus = STATR0;

     /* Error Interrupt handling */

     /* BusOff State */

if( (canstatus & 0x80) == 0x80 ) /* C_CAN Channel is in
BusOff state */

     {

     /* Do what has do be done in BusOff state */

putstr(5,CUP(1,27)); /* set Cursor to position 1, line26
*/
```

```
putstr(5, "C_CAN channel 0 is in BusOff state -> System
Halted !!!");

     while(1)

     {

     HWWD_CL = 0; /* endless loop */

     }

}

     /* RX ok */

if( (canstatus & 0x10) == 0x10 ) /* Last Reception ok */

     {

          RxOK_Int = 1;

          /* Clear RxOK Flag in Status register. */

          STATR0_RxOK = 0;

     }

     /* TX ok */

if( (canstatus & 0x08) == 0x08 ) /* Last Transmission ok
*/

     {

          TxOK_Int = 1;

          /* Clear TxOK Flag in Status register. */

          STATR0_TxOK = 0;

     }

     /* Last Error Counter */

if( (canstatus & 0x07) != 0x00 ) /* Show Last Error */

     {

     LEC_Int = 1;

     /* Clear LEC in Status register. */
```

```
        STATR0_LEC = 0;
        }
    }
```

# APPENDIX E

## CONFIG.TXT

```
/* Enter here your CAN IDs and corresponding FID for
each CAN ID*/

/* CANRX FRTX */

CANID 0 = 123 --> FID 0 = 6

CANID 1 = 400 --> FID 1 = 9

/* Enter here your FIDs and corresponding CAN ID for
each FID*/

/* CANTX FRRX */

FID 0 = 7 --> CANID 0 = 1

FID 1 = 8 --> CANID 1 = 0

FID 2 = 16 --> CANID 2 = 19

FID 3 = 17 --> CANID 3 = 18

FID 4 = 19 --> CANID 4 = 17

FID 5 = 20 --> CANID 5 = 13

FID 6 = 26 --> CANID 6 = 12

FID 7 = 28 --> CANID 7 = 11
FID 8 = 31 --> CANID 8 = 10

FID 9 = 33 --> CANID 9 = 9

FID 10 = 36 --> CANID 10 = 8

FID 11 = 37 --> CANID 11 = 5

FID 12 = 41 --> CANID 12 = 4

FID 13 = 47 --> CANID 13 = 3

FID 14 = 48 --> CANID 14 = 2


/* Enter here .chi file of your gateway*/


Gateway Chi File = GW_Controller1.chi
```

# APPENDIX F

## GATEWAY.C

```
int CANtx[15] = {1, 0, 19, 18, 17, 13, 12, 11, 10, 9, 8,
5, 4, 3, 2};

int CANrx[15] = {123, 400};

int FRtx[2] = {6, 9};

int FRrx[15] = {7, 8, 16, 17, 19, 20, 26, 28, 31, 33,
36, 37, 41, 47, 48};

int CAN2FR[2] = {0, 1};

int FR2CAN[15] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14};

int Tx_Buffer[2] = {1, 4};

int Rx_Buffer_A[15] = {0, 2, 4, 6, 8, 10, 13, 15, 17,
19, 21, 23, 24, 26, 28};

int Rx_Buffer_B[15] = {1, 3, 5, 7, 9, 11, 12, 14, 16,
18, 20, 22, 25, 27, 29};
```

# APPENDIX G

## GATEWAY.H

```
#define    fr_rx_count   15
extern int CANtx[15];
extern int CANrx[2];
extern int FRtx[2];
extern int FRrx[15];
extern int CAN2FR[2];
extern int FR2CAN[15];
extern int Tx_Buffer[2];
extern int Rx_Buffer_A[15];
extern int Rx_Buffer_B[15];
```