Investigation of a Flexray - CAN Gateway in the Implementation of Vehicle Speed Control

---

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ENGINEERING TECHNOLOGY
OF WATERFORD INSTITUTE OF TECHNOLOGY
IN COMPLETE FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS OF ENGINEERING

Author

Brian Somers

Supervisor

Mr. John Manning

June 2009

Dedicated to:

My Mother: Ann Somers

and

My Father: Tom Somers

Declaration

I hereby declare that the material presented in this document is entirely my own work and has not been submitted previously as an exercise or degree at this or any other establishment of higher education. I, the author alone, have undertaken the work except where otherwise stated.

Signed: _____

Date: _____

Acknowledgements

I hereby acknowledge the contributions to my work and offer my thanks to people who have helped and supported me during the course of this research.

My Supervisor:
Mr. John Manning: I would like to thank John for his continuous supervision, encouragement and invaluable guidance in all areas throughout the course of this research.

My Family and Friends:
I would like to thank my family and friends for their support, encouragement and understanding throughout all of my studies.

The AAEC (Advanced Automotive Electronic Control) Research Group:
I would like to take this opportunity to thank all members of the research group, both past and present, whose assistance, knowledge and support has been first-rate. In particular I would like to thank Henry Acheson and Niall Murphy for their additional support throughout the project.

There are also many more people who have contributed in countless others way and deserve my thanks also - Thank you!

Abstract

As the quantity of in-vehicle electronics has increased, automotive networking has been introduced to replace point to point wiring. Of the automotive networks developed, Controller Area Network (CAN) has become the most widely used. While the use of CAN has has been very useful in the development of automotive electronic systems, easing the implementation of such features as ABS, traction control and on-board diagnostics, it is now becoming a limiting factor in the design of new features such as X-by-wire. This is due to increasing demands on bandwidth, and an increasing need for determinism and fault tolerance. To meet these requirements a communication protocol called FlexRay has been developed, and is set to become the industry standard for advanced automotive communications.

In future automotive systems, FlexRay will be needed to work in parallel with CAN. For effective operation of the devices on the network, the two protocols will need to be able to communicate with one another. Efficient gateways will be needed to enable node to node communications over the different protocols.

This report investigates the design and implementation of an efficient gateway to enable communication between the CAN and FlexRay protocols. To achieve this, a framework for gateway design was developed. The designed gateway was then implemented on a Freescale HCS12X microprocessor. The implemented system obtains data from a FlexRay node via the FlexRay bus and translates the data to the CAN protocol. The data is then displayed on a dash panel along with the error status of the FlexRay bus.

Contents

List of Figures

List of Tables

Abbreviations

ABS	Anti-lock Braking System
AMI	Asynchronous Memory Interface
BM	Bus Minus
BP	Bus Plus
BRP	Baud Rate Prescaler
CAN	Controller Area Network
CC	Communication Controller
CHI	Controller Host Interface
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
EBI	External Bus Interface
ECU	Electronic Control Unit
EOBD	European On-board Diagnostics
FPGA	Field Programmable Gate Array
GmbH	Gesellschaft mit beschrankter Haftung (Ltd)
GUI	Graphical User Interface
IC	Integrated Circuit
ICD	In-Circuit Debugger
ID	Identifier

IDE	Integrated Development Environment
IRQ	Interrupt Request
ISO	International Standards Organisation
LAN	Local Area Network
LED	Light Emitting Diode
LIN	Local Interconnect Network
LLD	Low Level Driver
MMC	Memory Mapping Control
MSCAN	Motorola Scalable Controller Area Network
MT	Macrotick
NIT	Network Idle Time
NRZ	Non-Return-to-Zero
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PIT	Periodic Interrupt Timer
POC	Protocol Operation Control
RISC	Reduced Instruction Set Computer
RTR	Remote Transmit Request
SAE	Society of Automotive Engineers
SCI	Serial Communication Interface
SJW	Synchronisation Jump Width
TDMA	Time Division Multiple Access
TTCAN	Time Triggered Controller Area Network
TTP	Time Triggered Protocol
USB	Universal Serial Bus
VAN	Vehicle Area Network

Chapter 1

Introduction

1.1 Introduction

Since the introduction of electronics in automotive systems its use has increased exponentially. Some examples of this are engine management systems, anti-lock braking systems (ABS) , automatic transmissions and central locking. These electronic control modules typically get their inputs from switches and sensors, compute using the received data and then use actuators to enforce the outputs. For all these components to function properly there needs to be a link between them. For example, to change gear the transmission ECU requests the engine ECU to reduce torque, the transmission ECU then informs the gear shift actuator to change gear. Once the gear change has been made the transmission requests the engine to increase torque again.

If all of these devices and sensors were to be connected together using point-to-point wiring, the cable networks in cars would grow to lengths of several miles. This would add to the overall cost and reliability problems of a car. To overcome this problem a networking system was designed by Robert Bosch GmbH in the 1980s using a serial bus system to connect the various control systems. This system be-

came a standard and is known as CAN (Controller Area Network). By networking using CAN any node (device) can be removed or added to the bus without having to redesign the whole system. This is very useful where cars may have different specifications such as with or without electric windows, particular airbag arrangements etc. The wiring circuit for these cars can be standardised, and the devices can be plugged in as required without disturbing the communication of the other nodes [1]. Today, the majority of cars use the CAN standard.

As automobile manufacturers began investigating new powertrain, chassis, and by-wire control systems, two companies in particular, BMW and Daimler Chrysler, realised that none of the automotive networking standards would meet the needs of these particular innovations. They required a very fast, deterministic, and fault-tolerant protocol that could satisfy the speed, reliability, and safety requirements for applications such as brake-by-wire and steer-by-wire [2]. In 2000 they formed a consortium with other car and electronics manufacturers to address these needs. Over the past nine years they have developed a networking system called FlexRay, which is set to become the new automotive standard [3]. FlexRay has a much higher baudrate than CAN (20Mb/s compared to 1Mb/s), and it is a time triggered protocol which gives it a much higher reliability and fault tolerance [4].

CAN systems however, are not likely to become obsolete in the near future. CAN is able to adequately handle the less demanding communications tasks more cost effectively than FlexRay, and has already been tested and proven in current systems, thus reducing development costs. For these reasons CAN will remain a fundamental network in automotive communications. CAN will be needed to operate in parallel with FlexRay, to handle functions on the car, such as engine management, body electronics etc, while FlexRay networks the more advanced functions such as steer and braking by wire, chassis control etc. As some of the components on the FlexRay

network may need to acquire data from the components on the CAN network, and vice versa, therefore gateways will be required to enable the networks to communicate. A gateway acts as a translator between the two networks. For a gateway to be effective it needs to be able to cope with the needs of the protocols it is interfacing. A FlexRay - CAN gateway will need to be high speed and able to transfer messages between the networks accurately in a predictable and reliable manner.

This research thesis investigates the current standards in automotive networks, with particular attention to inter-protocol network gateway systems. It will then describe the design and implementation of a network gateway to communicate between a CAN and FlexRay network.

1.2 Thesis Contributions

The material and information presented in this thesis has been compiled on the basis of:

i. Literature review

ii. System design

iii. Implementation and Testing

The work presented in this thesis is laid out as follows:

- Chapter Two gives an overview of the most relevant information from all literature reviewed during the research for this thesis. It outlines the protocols, technologies and components researched in order to formulate a suitable methodology for this application.

- Chapter Three explains the requirements of the system and proposes a framework for the design of the system with respect to the findings of the literature reviewed. A real-world application is then presented based on the framework which was proposed.

- Chapter Four describes how the system outlined in Chapter Three was implemented in hardware and software. The chapter then describes the different testing methods and the tools used for verification of the systems operation. The results from this testing are then analysed.

- Chapter Five outlines the conclusions based on the research and system implementation conducted. A discussion regarding further possibilities for research based on findings from this particular study is also provided.

Chapter 2

Literature Review

2.1 Introduction

This chapter outlines the areas of review relevant to the research, from all literature assessed. The topics reviewed include the automotive network protocols Controller Area Network (CAN) and FlexRay, their history and future outlook. Inter-network communication gateways are discussed with regard to these protocols. This chapter also outlines the possible choices available during the design of the proposed system. The information based on the literature review presented in this chapter is laid out as follows:

- Section 2.2 describes the history and fundamental ideas behind the introduction of Local Area Networks and their application in automotive systems. It also discusses how automotive networks are standardised to the OSI (Open Systems Interconnection) model.
- Section 2.3 provides an overview of the Controller Area Network, and discusses the operation of the protocol with regard to the physical layer and data link layer of the OSI model.
- Section 2.4 discusses the limitations of Controller Area Network (CAN) in the

context of next generation networks, and the systems being used to overcome these limitations.

- Section 2.5 provides an overview of the FlexRay protocol, and discusses its operation with regard to the physical layer and data link layer of the OSI model.
- Section 2.6 describes the function and operation of network gateways and discusses different topologies for use in gateway design.
- Section 2.7 provides a brief summary of the information presented in this chapter.

2.2 Automotive Networks

2.2.1 History

Electronics was initially introduced to commercially available automobiles in the late 1950s and early 1960s. These were discontinued soon afterwards, as they were not well received by customers. During the 1970s two major events occurred that restarted the trend toward the use of modern electronics in automobiles, the first being the introduction of government regulations for exhaust emissions and fuel economy, which required better control of the engine than was possible with the methods being used, and the second being the development of relatively low cost per function solid-state digital electronics that could be used for engine control and other applications [5].

To work effectively the new electronic devices needed to be able to retrieve information from other devices and sensors located in the car requiring many devices to be interconnected individually. As the number of sensors increased, the wiring

increased vastly. For example, todays high-end automobiles may have more than 4 kilometres of wiring compared to 45 meters in vehicles manufactured in 1955. In July 1969, Apollo 11 used a little more than 150 kbytes of on board memory to go to the moon and back. Currently a family car might use 500 kbytes to keep the CD player from skipping tracks [6].

The increased wiring introduced problems such as an increase in vehicle weight, (for example, in an average well-tuned vehicle, every extra 50 kilograms of wiring, or extra 100 watts of power, increases fuel consumption by 0.2 litres for each 100 kilometres travelled). The extra wiring lead to a rise in the number of connectors, causing more possible failure points which affected the reliability of the systems. Also, complex wiring harnesses took up large amounts of vehicle volume, limiting expanded functionality. Eventually, the wiring harness became the single most expensive and complicated component in vehicle electrical systems [6].

To overcome the problems associated with the vast amounts of wiring which would be required for future electronic systems which would be implemented in automobiles, it was decided to use Local Area Networks (LAN) to connect a vehicles electronic components. Motorola reported that replacing wiring harnesses with LANs in the four doors of a BMW reduced the weight by 15 kilograms while enhancing functionality [6]. Figures 2.1 and 2.2 illustrate and compare point to point wiring to using a LAN.

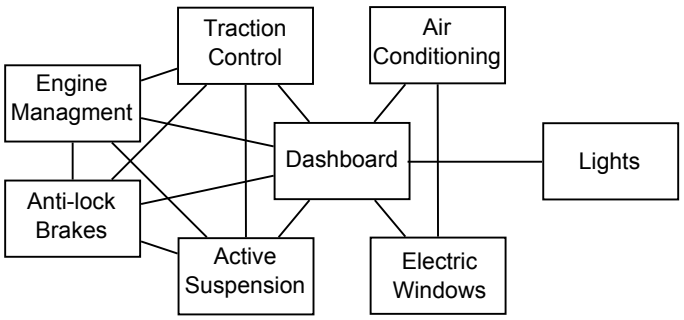

Figure 2.1: Point-to-Point Wiring System

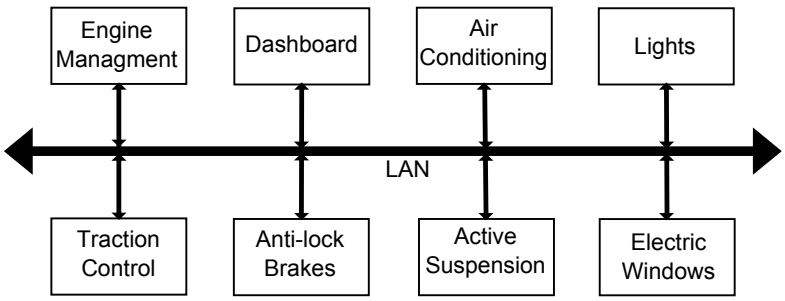

Figure 2.2: Local Area Network

To incorporate LANs in automotive subsystems the car manufacturing companies initially developed their own LAN technologies. Some examples are Controller Area Network (CAN) by Robert Bosch GmbH, Vehicle Area Network (VAN) by Renault and Peugeot, Automobile Bus (ABUS) by Volkswagen, and J1850 by GM, Ford and Daimler Chrysler. However, as many of the automotive vendors share subcontractors there was a need for standardisation. One system that became increasing popular in the beginning of the 1990s was the Controller Area Network and it soon became the most used LAN in the automotive industry. It is estimated that the number of CAN nodes in 2005 was 400 million, and experiencing 30 percent growth in sales yearly [7][8]. Due to its popularity, and the fact that CAN only comprises layer 1 and 2 of the Open Systems Interconnection (OSI) model, several higher layer protocols have been developed on top of CAN over the years, including CAN Kingdom , CANopen and DeviceNet. These higher layer protocols simplify the usage of CAN in terms of development and maintenance, by their capabilities

and tool support [3]. More recently LIN (Local Interconnect Network) has been developed as a cheaper, simpler subsystem to be used in conjunction with CAN for applications such as lights, climate control etc.

Although CAN is a robust, cost-effective general control network, certain newly advanced applications demand more specialised control networks. For example, X-by-wire (i.e. Brake by wire and steer by wire) systems use electronics, rather than mechanical or hydraulic means, to control a system. These systems require highly reliable networks [6]. This need has given rise to a number of new protocols such as Time Triggered CAN (TTCAN), Time Triggered Protocol (TTP), and FlexRay which are fault tolerant, deterministic and have the bandwidth necessary for the next generation of automotive applications. Out of these FlexRay seems set to become the de-facto standard [9].

2.2.2 Vehicle Applications

The use of electronics in automobiles has brought many benefits to the consumer. These range from central locking to European On-board Diagnostics (EOBD) to driver assist systems.

Diagnostic systems are used to provide information about the status of the vehicle. Initially, this was driven by legal requirements which commanded monitoring of emission related components under EOBD. It also now plays a large role for mechanics in the fault finding process. Each workshop has tooling which can be plugged into the network and will give the user feedback on where a problem lies. Another advantage is the increased perceived quality of an automobile by the customer. For example sensors on the car may detect when a component is starting to operate outside its limits and the user is alerted of this before it becomes a critical issue. Ideally

this would lead to the customer never having a critical issue with the automobile [10].

At the moment steer and brake by wire are not widely available on commercial automobiles, but with the development of FlexRay it is due to arrive over the coming years. There are many consumer safety advantages from having drive by wire systems. For example, removal of the steering column will improve driver safety in collisions [6], suitable sensors will track nearby cars preventing several types of accidents from occurring by applying the brakes or steering automatically to avoid collision [11]. There are many advantages to the automotive producers also. The removal of the steering column allows new styling freedom, while also simplifying production of left and right-hand models [6]. Using brake by wire reduces assembly costs, makes the car lighter, more environmentally friendly, and allows more advanced functionality to be implemented [3].

2.2.3 OSI Model

The ISO (International Standards Organisation) has created a layered model called the OSI (Open Systems Interconnect) model to describe defined layers in a network operating system. The purpose of the layers is to provide clearly defined functions to improve internetwork connectivity between computer manufacturing companies. Each layer has a standard defined input and a standard defined output [12].

There are 7 Layers of the OSI model:

7. Application Layer (Top Layer)
6. Presentation Layer
5. Session Layer
4. Transport Layer
3. Network Layer

2. Data Link Layer
1. Physical Layer (Bottom Layer)

The design of relevant automotive networking models will be explained in terms of the OSI layers in the following sections.

2.3 CAN: Controller Area Network

2.3.1 Overview

CAN (Controller Area Network) is a serial bus system, which was originally developed for automotive applications in the early 1980's. The CAN protocol is internationally standardised in ISO 11898-1 and comprises the data link layer and components of the physical layer of the 7-layer ISO/OSI reference model. CAN, which is now available from more than 50 semiconductor manufacturers in hardware, provides two communication services: the sending of a message (data frame transmission) and the requesting of a message (remote transmission request, RTR) [13].

A typical vehicle can contain two to five separate CAN networks operating at different transmission rates. A low-speed CAN network running at less than 125 kbps usually connects vehicle comfort electronics, like seat and window movement controls and other user interfaces. Generally, control applications that are not real-time critical use this low-speed network segment. Low-speed CANs have an energy-saving sleep mode in which nodes stop their oscillators until a CAN message awakens them. Sleep mode prevents the battery from running down when the ignition is turned off. A higher-speed CAN interconnects the more real-time-critical functions such

as engine management, anti-lock brakes, and cruise control. Although capable of a maximum baud rate of 1 Mb/s, the electromagnetic radiation on twisted-pair cables that results from a CANs high-speed operation makes providing electromagnetic shielding in excess of 500 kb/s too expensive [6].

CAN is a message based protocol. This means that devices connected to a CAN network do not have unique addresses, but instead the message that a device sends onto the network possesses a unique ID number, according to its content (e.g. engine temperature). As a result, each device on the network listens to every message transmitted on the bus and determines what action, if any, it needs to take.

2.3.2 CAN and the OSI Model

As discussed in Section 2.2.3, the CAN protocol conforms to the OSI model (Figure 2.3). The CAN protocol defines the Data Link Layer and parts of the Physical Layer. The communication medium portion of the model was purposely left out of the Bosch CAN specification to enable system designers to adapt and optimise the communication protocol on multiple media for maximum flexibility (twisted pair, single wire, optically isolated, RF, IR, etc.). With this flexibility however, comes the possibility of interoperability issues. To ease some of these concerns, the International Standards Organisation and Society of Automotive Engineers (SAE) have defined some protocols based on CAN that include the Media Dependant Interface definition such that all of the lower two layers are specified. ISO11898 is a standard for high-speed applications, ISO11519 is a standard for low-speed applications, and J1939 (from SAE) is targeted for truck and bus applications. All three of these protocols specify a 5V differential electrical bus as the physical interface. The rest of the layers of the ISO/OSI protocol stack are left to be implemented by the system software developer [14].

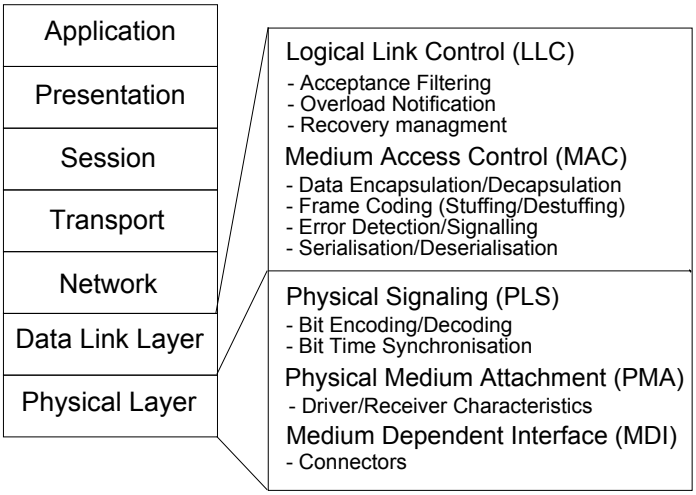

Figure 2.3: CAN and the OSI Model

2.3.3 Physical Layer

The physical layer shall be discussed under the following headings:

- Bit Encoding/Decoding
- Bit Timing and Synchronisation
- Physical Medium
- Data Rate vs Bus Length

2.3.3.1 Bit Encoding/Decoding

The CAN protocol uses Non-Return-to-Zero (NRZ) bit coding. With NRZ bit coding the signal level remains constant over the bit time so just one time slot is required for the representation of a bit (See Figure 2.4). The signal level can remain constant over a longer period of time; therefore measures must be taken to ensure that the maximum permissible interval between two signal edges is not exceeded. This is

important for synchronisation purposes, and therefore 'Bit Stuffing' is used. Bit stuffing is applied by inserting a complementary bit after five bits of equal value. At the receiving end, the receiver has to un-stuff the stuff-bits so that the original data content is processed [13].

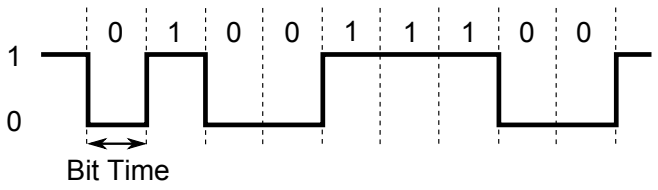

Figure 2.4: NRZ Waveform

2.3.3.2 Bit Timing and Synchronisation

CAN uses synchronous bit transmission which enhances its transmitting capacity but also means that a sophisticated method of bit synchronisation is required. While bit synchronisation in an asynchronous transmission is performed upon the reception of the start bit available with each character, in a synchronous transmission protocol there is just one start bit available at the beginning of a frame. To enable the receiver to correctly read the messages, continuous resynchronisation is required. Phase buffer segments are therefore inserted before and after the sample point within a bit interval.

The CAN protocol regulates bus access by bit-wise arbitration, so the signal propagation from sender to receiver and back to the sender must be completed within one bit time. For synchronisation purposes a further time segment, the propagation delay segment, is needed in addition to the time reserved for synchronization. The propagation delay segment takes into account the signal propagation on the bus as well as signal delays caused by transmitting and receiving nodes [13].

There are two types of synchronisation used: hard synchronisation at the start of a frame and resynchronisation within a frame.

- After a hard synchronisation the bit time is restarted at the end of the sync segment. Therefore the edge, which caused the hard synchronization, lies within the sync segment of the restarted bit time.
- Resynchronisation shortens or lengthens the bit time so that the sample point is shifted according to the detected edge. The amount by which this can be shifted at any one time is set by the SJW (Synchronisation Jump Width) parameter.

These various timing parameters can be adjusted by the system designer to suit the current application. The CAN bit timing segments are shown in Figure 2.5.

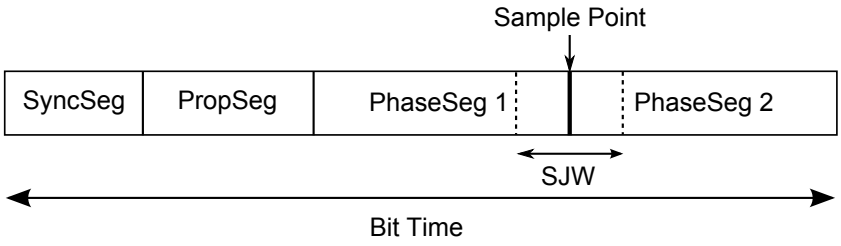

Figure 2.5: CAN Bit Time Segments

2.3.3.3 Physical Medium

Although the physical medium is not defined in the CAN protocol, a medium must be chosen that is capable of transmitting the two possible bit states 'dominant' and 'recessive'. One of the cheapest and most common ways is to use a twisted wire pair. The bus lines are then called 'CAN High' and 'CAN Low'. The two lines are driven by the nodes with a differential signal which is defined in the protocol. This signal is illustrated in Figure 2.6. A recessive bit is represented by both lines being

driven to a level of 2.5V, resulting in a differential voltage of 0V. A dominant bit is represented by CAN High going to 3.5V and CAN Low going to 1.5V, resulting in a differential voltage of 2V [1]. For the node to read the bus level correctly it is important that signal reflections are avoided. This is achieved by terminating the bus line with a termination resistor of 120Ω at both ends of the bus.

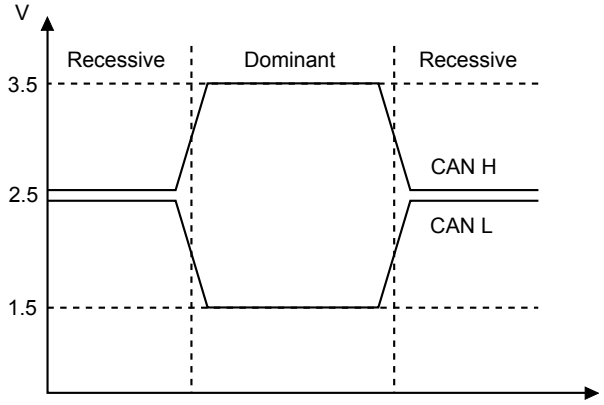

Figure 2.6: CAN Bus Levels

2.3.3.4 Data Rate vs. Bus Length

Depending on the size of the propagation delay segment the maximum possible bus length at a specific data rate (or the maximum possible data rate at a specific bus length) can be determined (See Table 2.1). The signal propagation is determined by the two nodes within the system which are furthest apart from each other. It is the time that it takes a signal to travel from one node to the other (taking into account the delay caused by the transmitting and receiving node), synchronisation and the signal from the second node to travel back to the first one. Only then can the first node decide whether its own signal level is the actual level on the bus or whether it has been replaced by another node. This is important for bus arbitration [15].

Bit Rate (kb/s)	Bus Length (m)
1000	30
500	100
250	250
125	500
62.5	1000

Table 2.1: Data Rate vs. Bus length

2.3.4 Data Link Layer

The data link layer shall be discussed under the following headings:

- Message Framing
- Arbitration
- Error Detection and Handling

2.3.4.1 Message Framing

Once the raw data has been accepted from the processor, it is then bundled into a predefined structure called a frame bt the CAN controller. The CAN protocol defines four different types of frames [13]:

- **Data Frame:** A data frame is generated by a CAN node when the node wishes to transmit data. This is received by all other nodes on the bus.
- **Remote Frame:** A remote frame is generated by a destination CAN node to request data from another node on the network.
- **Error Frame:** An error frame is generated by a node when it detects a protocol error.

- **Overload Frame:** An overload frame is generated if a node wishes to request more time to process received information.

2.3.4.2 Arbitration

The CAN communication protocol uses a CSMA/CD process. CSMA is an abbreviation for Carrier Sense Multiple Access. This means that every node on the network must monitor the bus for a period of no activity before trying to send a message on the bus (Carrier Sense). Also, once this period of no activity occurs, every node on the bus has an equal opportunity to transmit a message (Multiple Access). The CD stands for Collision Detection. If two nodes on the network start transmitting at the same time, the nodes will detect the collision and take the appropriate action. In CAN protocol, a non destructive bitwise arbitration method is utilised. This allows all messages to remain intact after arbitration is completed even if collisions are detected. All of this arbitration takes place without corruption or delay of the higher priority message [14].

There are a couple of things that are required to support non-destructive bitwise arbitration. Firstly logic states need to be defined as dominant or recessive. Secondly, all transmitting nodes must monitor the state of the bus to see if the logic state it is trying to send actually appears on the bus. A dominant bit state will always win arbitration over a recessive bit state, therefore the lower the value in the message identifier (the field used in the message arbitration process), the higher the priority of the message. For example, if two nodes are trying to transmit a message at the same time, each node will monitor the bus to make sure the bit that it is trying to send actually appears on the bus. The lower priority message will at some point try to send a recessive bit and the monitored state on the bus will be a dominant. At that point this node loses arbitration and immediately stops transmitting. The higher priority message will continue until completion and the node that lost arbi-

tration will wait for the next period of no activity on the bus and try to transmit its message again [14].

2.3.4.3 Error Detection and Handling

To ensure the integrity of messages, CAN nodes have the ability to determine fault conditions and transition to different modes based on the severity of problems being encountered. They also have the ability to distinguish short disturbances from permanent failures and modify their functionality accordingly. CAN nodes can transition from functioning as a normal node (being able to transmit and receive messages normally), to shutting down completely (bus-off) based on the severity of the errors detected. This feature is called Fault Confinement. No faulty CAN node or nodes should be able to use all of the bandwidth on the network as any faults will be confined to the faulty nodes which will shut off before bringing the network down.

2.4 Next Generation Automotive Networks

At present CAN is without doubt the most widely used in-vehicle network. It was designed by Bosch in the mid 1980s for multiplexing communication between ECUs (Electronic Control Units) in vehicles. This decreased the length and number of dedicated wires in the wiring harness. For example, the number of wires has been reduced by 40 percent, from 635 to 370, in the Peugeot 307, in comparison with the non-multiplexed Peugeot 306 [7]. While the use of CAN has has been very useful in the development of automotive electronic systems, easing the implementation of such features as ABS, traction control and on-board diagnostics, it is now becoming a limiting factor in the design of certain new innovations. This is due to increasing demands on bandwidth, and an increasing need for deterministic and fault tolerant

networks. Fault tolerance of a system is a property which allows a system to operate properly in the event of a failure of one or more of its components. A deterministic communication system is one which can guarantee transmission times of a message. To overcome the determinism problem in CAN another protocol version was released called TTCAN (Time Triggered CAN). TTCAN messages are sent on a time triggered basis as opposed to event driven in standard CAN. Each message has a specific time slot for transmission on the network. This ensures that all nodes get a chance to transmit on the bus, which is essential for safety critical systems such as airbags which need to react in real time, and cannot wait to gain access to the bus. While TTCAN has addressed some of CAN's limiting factors, it still has not achieved the performance necessary for future automotive systems [16].

The next generation of automotive electronics are showing a trend towards the further integration of current systems, with the result of increasing their performance. For example, it has been determined that ABS and traction control systems provide better performance if the values of vertical acceleration (traditionally needed only by the suspension control system) are also made available to them [11]. This however, leads to a much higher load on the network, that may not be satisfied by conventional CAN. One option to overcome this is to use multiple CAN networks connected via gateways. This is not regarded as the optimal solution however, due to both the cost of the gateways and the additional time delays they introduce. There is now a movement towards reducing the number of different networks that are used in a car although it is not generally possible to rely on one network only, especially in luxury cars [11].

Another direction automotive networks are moving towards is the implementation of drive by wire systems. As discussed in Section 2.2.2, there are a number advantages in terms of safety and flexibility to both the customer and the manufacturer of

having brake and steer by wire systems on a vehicle. These systems however require large bandwidth as well as determinism and fault tolerance. While these systems could possibly be implemented using CAN, the lack of fault tolerance means they could not be used commercially unless they have some form of back up. This is seen in the current 'wet' steer by wire systems on some car models, which use electric motors to steer but still keep a mechanical link to the steering wheel.

A number of other protocols have been developed to address the current automotive needs such as TTP (Time Triggered Protocol), TTCAN, Byteflight and FlexRay. FlexRay seems set to become the industry standard, as an increasing number of manufacturers are joining the FlexRay Consortium. FlexRay has the capabilities to meet the requirements of the future networks, due to its high bandwidth, deterministic architecture, fault tolerance and flexibility to adapt to different needs [9]. The next section shall discuss the FlexRay protocol.

2.5 FlexRay

2.5.1 Overview

The introduction of advanced control systems, which combine multiple sensors, actuators, and electronic control units (ECU), has begun to place boundary demands on the existing CAN communication bus found in most of todays automobiles [17]. As a result, initiatives by automobile manufacturers and suppliers have led to the creation of FlexRay. FlexRay is intended to be a new standard which will meet and exceed future requirements for a deterministic and fault-tolerant bus system with high data rates for advanced automotive control applications . The leaders in this initiative, called the FlexRay Consortium which started in 2000, were BMW, Daimler Chrysler, Philips Semiconductor and Freescale Semiconductor. Since then,

they have been joined by over 100 companies from different areas of the automotive and semiconductor sectors.

The consortium finalised the FlexRay Communications System Specifications Version 2.0 in the summer of 2004 and has since made them available to the general public [17]. The FlexRay protocol is internationally standardised in ISO 10681. FlexRay uses the OSI-model as a reference model. The layers that FlexRay uses are the Application, Data Link and Physical layers [18].

2.5.2 FlexRay Networks

FlexRay is a dual channel, high speed protocol with data rates of up to 20Mb/s (10Mb/s per channel) [19][20]. It is fault-tolerant and deterministic, and is aimed at advanced applications such as X-by-Wire. Unlike CAN, there is no single FlexRay topology. Instead networks can be configured in a number of ways such as bus, star or combinations of these (see Figures 2.9 to 2.11). To further add to the layout options, FlexRay has two channels (A and B) which can be configured separately. This allows data to be transmitted on one channel without the other being used, thus saving bandwidth. For safety critical applications, messages can be transmitted simultaneously on both channels giving a built in redundancy to the network. If one channel gets damaged, transmission will continue without interruption on the other channel. This is essential if drive by wire applications are to be implemented, so that, in the event of a failure on a channel the driver would still be able to have full control over the vehicle's brakes and steering. While drive by wire systems have not been implemented in commercially available automobiles at present, FlexRay forms the networking on the Active Suspension system in the BMW X5 series [21].

Data is transmitted on the FlexRay bus in both timed and event driven manner.

Each message is divided into two sections called the Static Segment and the Dynamic Segment. The static segment is defined during the configuration of the application and transmits the data on a TDMA (Time Division Multiple Access) basis. The Dynamic segment of the message handles data on an event triggered basis.

2.5.3 FlexRay and the OSI Model

As discussed earlier, the FlexRay protocol complies with the OSI model (Figure 2.7). The protocol defines parts of the Physical Layer, Data Link Layer, Presentation Layer and Application Layer [22]. For the purpose of this literature review, the Data Link and Physical Layers will be investigated. Figure 2.8 shows some of the sub-layers of the OSI model in the context of a FlexRay communication controller.

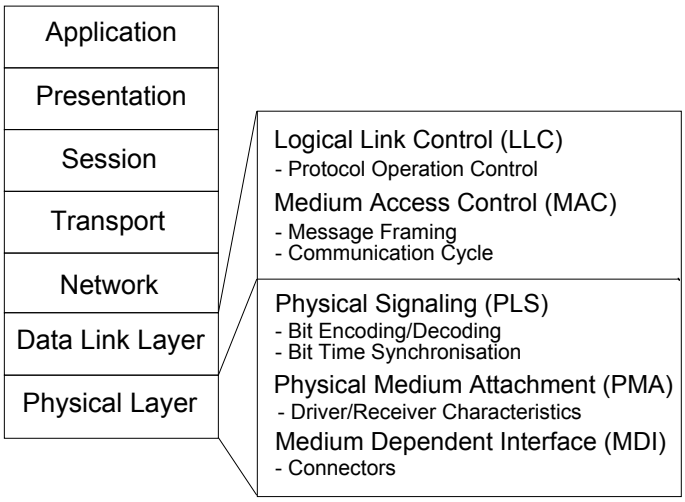

Figure 2.7: FlexRay and the OSI Model

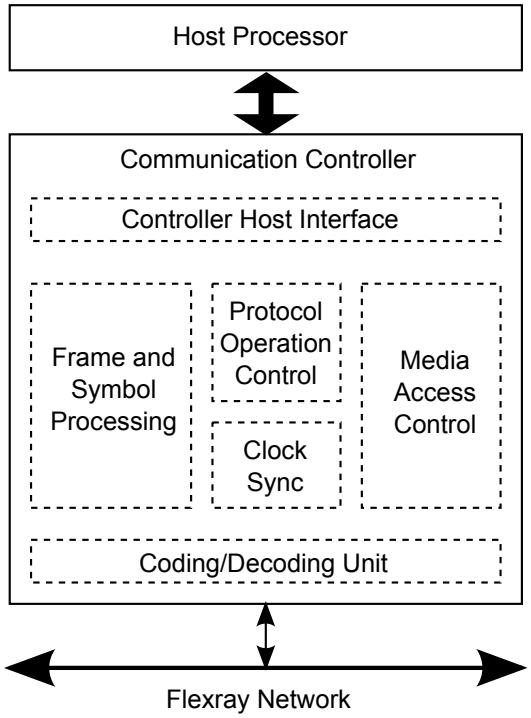

Figure 2.8: Architecture of Communication Controller

2.5.4 Physical Layer

The FlexRay physical layer shall be discussed under the following headings:

- Network Topologies
- Transmission Medium
- Signal Levels and Bit Representation
- Bit Coding and Decoding
- Synchronisation

2.5.4.1 Network Topologies

As previously discussed there are several options for the layout of a FlexRay network. It can be configured as a single-channel or dual-channel bus network, a single-channel or dual-channel star network, or in various combinations of bus and star

topologies. A FlexRay network consists of a maximum of two channels, Channel A and Channel B. Each node on the network can be connected to either or both of these channels. This flexibility in configuration may be used to increase bandwidth and/or introduce redundancy in to the system to increase its level of fault tolerance. The various topologies are described in more detail below:

- **Bus:** This configuration is similar to that used in CAN networking. A node can be connected to Channel A and B, or just to one of these channels. The distance between any two nodes on the bus cannot exceed 24 metres and a maximum of 22 nodes can be supported on any one channel. An example of a FlexRay bus is shown in Figure 2.9(a). Note that not every node is connected to both channels.

- **Passive Star:** In a passive star configuration, all nodes on a channel are connected to a single point. Each channel will have its own star. As with the bus configuration, distances between any two nodes cannot exceed 24 metres, and the number of nodes cannot exceed 22. An example of a passive star configuration is shown in Figure 2.9(b).

- **Active Star:** The active star network uses point-to-point connections between star couplers and nodes. When a data stream is received on one branch of the active star it is re-sent immediately on all other branches of the star. The star coupler has a transmitter and receiver circuit for each branch. Therefore the branches of the star are electrically decoupled from each other. This can be an advantage in a situation where a short circuit occurs on the network, as the fault will be confined to just that branch and the other branches can work as normal. For an active star configuration a maximum of 16 branches is allowed on any one star. The maximum branch length is 24 metres. An example of a dual channel single star topology is shown in Figure 2.10(a). Active stars can

also be cascaded as shown in Figure 2.10(b)

- **Hybrid Configurations:** In addition to topologies that are composed either entirely of a bus topology or entirely of a star topology, it is possible to have hybrid topologies that are a mixture of these configurations. The FlexRay system will support hybrid topologies once the limits applicable to each individual topology are not exceeded. There are countless variations of configurations which can be implemented, an example is shown in Figure 2.11

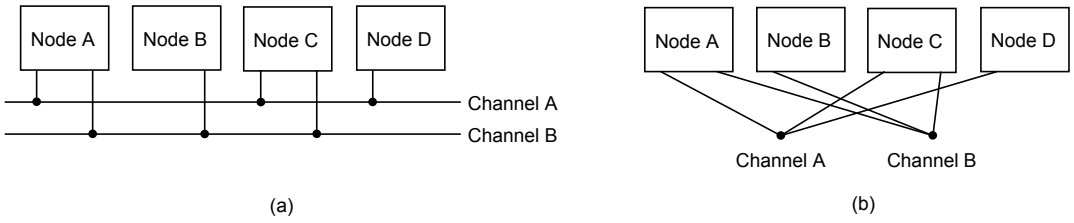

Figure 2.9: (a)Dual Channel Bus (b) Dual Channel Star

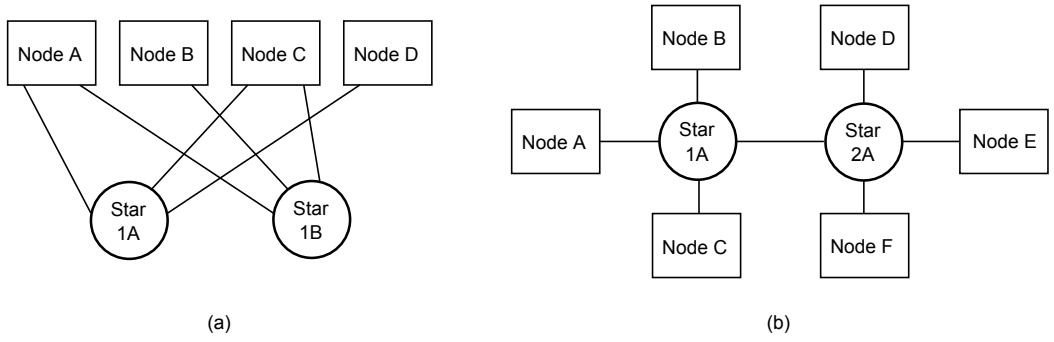

Figure 2.10: (a)Dual Channel Single Star (b)Single Channel Cascaded Star

2.5.4.2 Transmission Medium

The FlexRay protocol specification does not define cable types to be used, but does stipulate their electrical specifications. The medium in use for FlexRay busses may be shielded or unshielded cables, as long as they provide the following characteristics: Impedance of 80 - 110Ω at a frequency of 10MHz, maximum line delay of 10ns/m

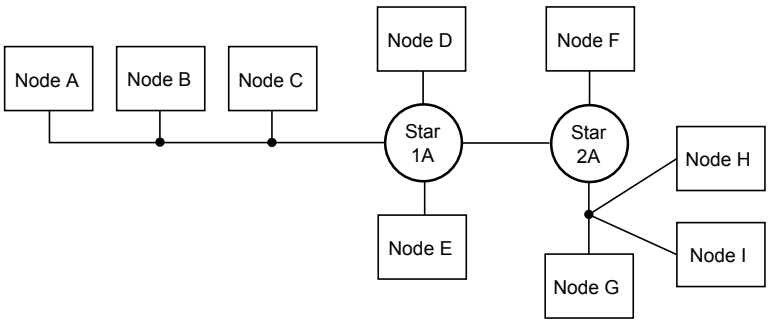

Figure 2.11: Single Channel Combined Bus and Star

and a maximum cable attenuation of 82dB/km at a frequency of 5MHz. These cable requirements are similar to that of CAN. A twisted-wire pair is generally used, as it helps to prevent electromagnetic interference from other electrical devices in the vicinity affecting the network. Each channel uses two wires to connect to the bus, labelled BP (Bus Plus) and BM (Bus Minus). The cables need to be terminated at each node, and at either end of a bus. This is achieved by connecting a termination resistor, R_T, in the region of 100Ω between the BP and BM wires (See Figure 2.12). This prevents the signal being reflected back through the bus once it reaches the end of the system.

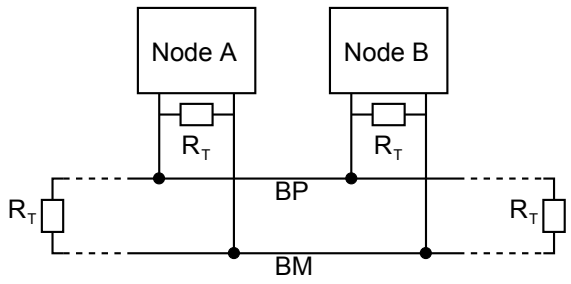

Figure 2.12: FlexRay Bus with Terminating Resistors

2.5.4.3 Signal Levels and Bit Representation

The bus communicates using two signals BP and BM. The differential voltage between the signals, V_{diff}, is used to represent the four different states which can occur

on the bus: Idle LP, Idle, Data 1, Data 0. The various states and their voltages (measured to ground) are shown in Figure 2.13. The differential voltage on the bus is designed as follows:

$$V_{diff} = V_{BM} - V_{BL} \tag{2.1}$$

- When the bus is in Idle LP (Low Power) there is no current being driven to either BP or BM and the bus driver biases both to ground.
- When the bus is in Idle there is also no current being driven to BP or BM, but as we can see from Figure 2.13 however, the connected nodes bias both BP and BM to 2.5 volts
- To drive the bus to Data 1, the bus driver increases the voltage on BP by 600mV and decreases BM by 600mV. This gives us a differential voltage of 1.2V. Data 1 represents a logical HIGH
- To drive the bus to Data 0, the bus driver decreases the voltage on BP by 600mV and increases BM by 600mV. This gives us a differential voltage of -1.2V. Data 0 represents a logical LOW

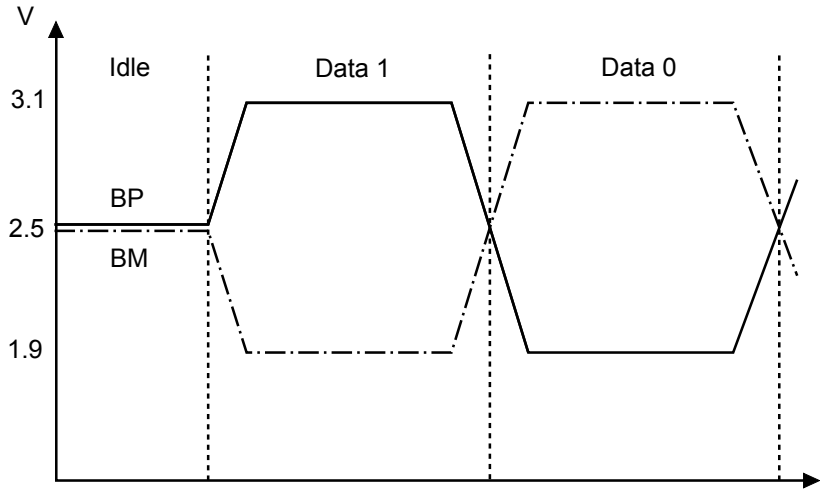

Figure 2.13: FlexRay Bus Levels

2.5.4.4 Bit Coding and Decoding

The decoding process samples the incoming data at eight times the rate of the bit clock. These samples are forwarded to a majority voting process, which analyses the last five samples received. If at least three samples are HIGH the process outputs a value of HIGH for that bit, otherwise it outputs a value of LOW. This voting process is used to suppress glitches in the received signal, provided that the duration of the glitch is less than three samples [23]. This process is shown in Figure 2.14. FlexRay nodes use a non-return to zero (NRZ) signalling method for coding and decoding of signals. This means that the generated bit level is either LOW or HIGH during the entire bit time. In order to support two channels each node must implement two sets of independent coding and decoding processes, one for channel A and another for channel B.

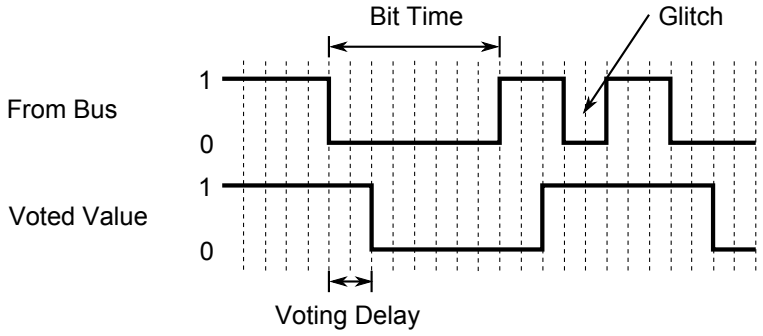

Figure 2.14: FlexRay Bit Decoding

2.5.4.5 Synchronisation

FlexRay is a time triggered networking system. Media access is time controlled and unlike CAN there is no collision detection or resolution mechanism in case of collisions but instead a mechanism for prevention of collisions. All nodes must be synchronised for successful and accurate communication. The clocks of the communication controllers in the network, however, can be influenced by temperature

and voltage fluctuations, or production tolerances of the oscillator. This leads to differing internal time bases. To offset this, the FlexRay protocol uses a distributed clock synchronisation mechanism, i.e. there is no single physical reference clock. Instead, each node individually synchronises itself to the network by observing the timing of transmitted synchronisation frames from other nodes. From this a virtual reference clock is established using a distributed fault-tolerant clock synchronisation algorithm. The deviation to this reference clock is then periodically measured in regard to phase and frequency deviation in order to ensure offset and rate correction respectively. If necessary, the clock is adjusted accordingly.

2.5.5 Data Link Layer

The FlexRay data link shall be discussed under the following headings:

- Message Framing
- Communication Cycle
- Static Segment
- Dynamic Segment
- Protocol Operation Control
- Controller Host Interface

2.5.5.1 Message Framing

The FlexRay frame consists of three segments, these are the header segment, the payload segment, and the trailer segment, as shown in Figure 2.15.

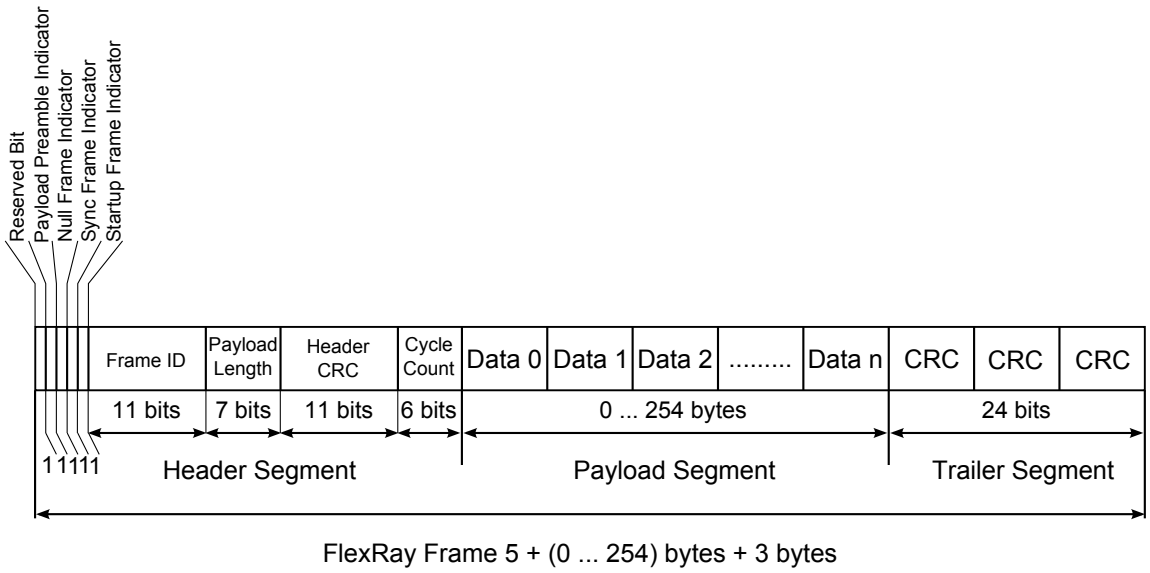

Figure 2.15: FlexRay Frame Format

Header Segment: The FlexRay header segment consists of five bytes. These bytes contain a reserved bit, the payload preamble indicator, null frame indicator, sync frame indicator, startup frame indicator, frame ID, payload length, header CRC, and the cycle count.

- **Reserved Bit (1 Bit):** This bit is not currently used by the protocol, and has been reserved for further use.
- **Payload Preamble Indicator (1 Bit):** This bit specifies the existence of vector information in the payload segment of the frame. In the static frame it indicates a Network Management Vector and in a dynamic frame it indicates Message ID.
- **Null Frame Indicator (1 Bit):** This bit designates whether or not the frame is a null frame, i.e. a frame that contains no usable data in the payload segment of the frame.
- **Sync Frame Indicator (1 Bit):** This indicates whether or not the frame is a sync frame, i.e. a frame that is utilised for system wide synchronisation of

communication.

- **Startup Frame Indicator (1 Bit):** Shows whether or not the node sending frame is the start-up node.
- **Frame ID (11 Bits):** The frame ID defines the slot in which the frame should be transmitted. An ID is assigned to each node at system designing. Valid frame IDs range from 1 to 2047.
- **Payload Length (7 Bits):** This states the data length of the payload segment.
- **Header CRC (11 Bits):** This is the CRC calculation value of Sync Frame Indicator, Startup Frame Indicator, Frame ID, and Payload Length which is calculated by the host.
- **Cycle Count (6 Bits):** This indicates the value of the cycle counter on the transmitting node at the time of frame transmission.

Payload Segment: The FlexRay payload segment contains 0 to 254 bytes of data. The bytes are identified numerically, starting at Data 0 for the first byte after the header segment and increasing by one with each subsequent byte.

For frames transmitted in the static segment the first 0 to 12 bytes of the payload segment may optionally be used as a network management vector. The payload preamble indicator in the frame header indicates whether the payload segment contains the network management vector. The length of the network management vector can be configured from 0 to 12 bytes.

For frames transmitted in the dynamic segment the first two bytes of the payload segment can be used as a message ID field, allowing the receiving nodes to filter data

based on the contents of this field. The payload preamble indicator in the frame header indicates whether the payload segment contains the message ID.

Trailer Segment: The FlexRay trailer segment contains a single 24 bit field. This has the CRC calculation values which have been calculated by the host for all fields in the header and payload segments of the frame.

2.5.5.2 Communication Cycle

Media access control is based on a recurring communication cycle. Communication cycles are executed periodically, and are of a constant time duration. There are 64 communication cycles, numbered from 0 to 63, which are executed sequentially. The schedule for each communication cycle can be different, i.e. for the same slot, different frames can be transmitted in different cycles. This makes efficient use of the bandwidth available by increasing the number of different signals that can be transmitted on the system. Within one communication cycle FlexRay offers the choice of two access schemes. These are a static time division multiple access (TDMA) scheme, and a dynamic mini-slotting based scheme. The communication cycle is defined by a timing hierarchy consisting of four levels as shown in Figure 2.16.

The communication cycle level contains the static segment, the dynamic segment, the symbol window and the network idle time (NIT). The static and dynamic segments will be discussed in greater detail later in this section. For the purpose of this research the symbol window is not required. The network idle time is a communication free period which concludes each communication cycle and is used by each node to calculate and apply clock correction. The arbitration grid level forms the backbone of FlexRay media arbitration. In the static section it consists of consecutive time intervals called static slots, and in the dynamic section it contains consecutive intervals called minislots. The next level is the macrotick level which is defined by

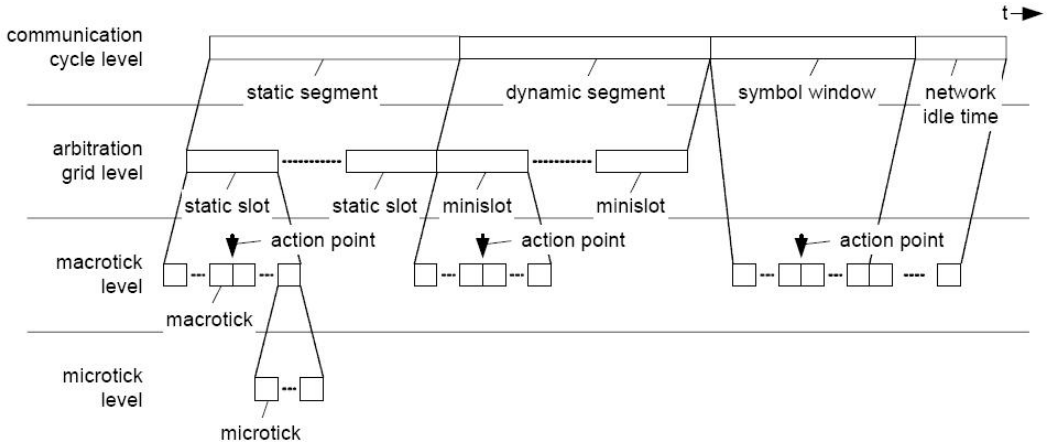

Figure 2.16: Communication Cycle Timing Heirarchy

the macrotic. A macrotick is a time slot the duration of which is an integer number of microticks. The lowest level in the hierarchy is the microtick. Microticks are time units derived directly from the communication controller's oscillator clock tick, which optionally can make use of a prescaler.

2.5.5.3 Static Segment

The communication cycle always contains a static segment, containing at least two static slots. The static segment of the communications cycle is used for scheduling time-triggered messages and reserved for synchronous communications [17]. This segment contains a configurable number of static slots. All static slots consist of the same number of macroticks. The length of the static slot must be configured to handle the amount of data which is to be transmitted in one communication cycle. The segment timing is exactly the same on both channels. An example of a static segment is shown in Figure 2.17.

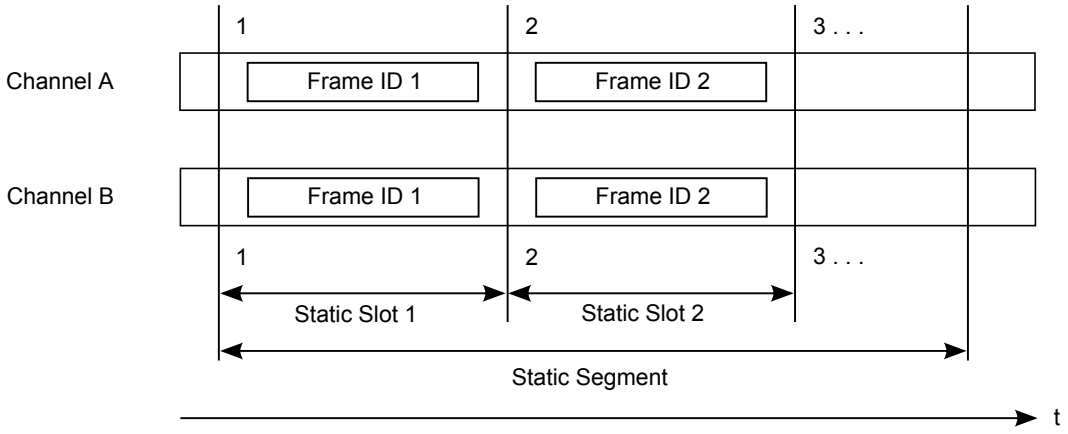

Figure 2.17: Structure of the Static Segment

2.5.5.4 Dynamic Segment

The communication cycle may also contain a dynamic segment. The dynamic segment is used for event-based messages that may emerge during run time and require varying bandwidths. Within the dynamic segment, devices compete for bandwidth using a priority driven scheme which assigns priority based on a message's Frame ID [17]. The dynamic segment consists of a configurable number of minislots consisting of an identical number of macroticks. If no dynamic segment is required, then the number of minislots can be set to zero. Each minislot is owned by exactly one (or none) node per channel and cycle. If a node wishes to communicate it must wait for its minislot to occur. If transmission does not occur in a minislot then all nodes increment their slot counter and the next node can begin transmission. The size of the communication slots in the dynamic segment may vary to accommodate frames of different length (see Figure 2.18), but data will only be sent if there is enough time left in the dynamic segment. Like the CAN system, this is priority driven where the message with the lowest ID has the highest priority.

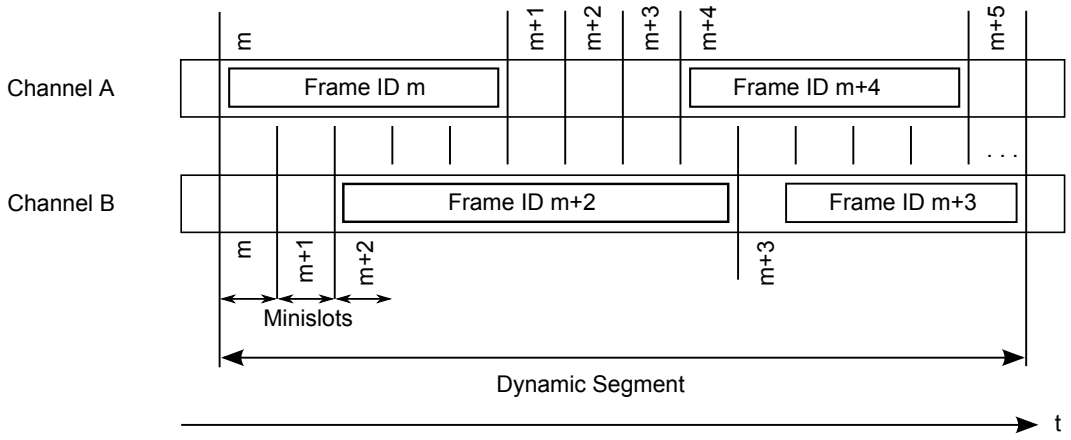

Figure 2.18: Structure of the Dynamic Segment

2.5.5.5 Protocol Operation Control

The main purpose of the protocol operation control (POC) is to react to commands from the host processor, or to protocol conditions such as errors. The POC sets the other components in the controller to the appropriate operation mode and its operation state represents the different controller states. An operation overview is shown in Figure 2.19. After power-up, the POC starts in the default configuration state, before it proceeds to the config state. The controller is only configurable via the config port in these two states. After configuration, the POC goes to the ready state. From there it depends on the configuration and the communication channel activity whether it proceeds to wakeup or startup. If a connected communication channel is idle and the ECU is allowed to wakeup a channel, the control goes to wakeup until the idle channel is ready to work. Then, in the startup state, the controller integrates into the cluster. As long as no errors occur, the controller stays in normal active state. In the case of an error, the POC changes to normal passive and tries to reintegrate. On a fatal error, the bus controller stops operation in the halt state. The host also has the possibility to change the controllers state. If the bus controller receives a command from the host, it has to decide whether the command is allowed in the current state and when it is to be applied. For example, a halt

command sent by the host is only allowed in the normal active or normal passive state and it is processed at the end of one communication round. A communication round is one execution of the whole schedule [24].

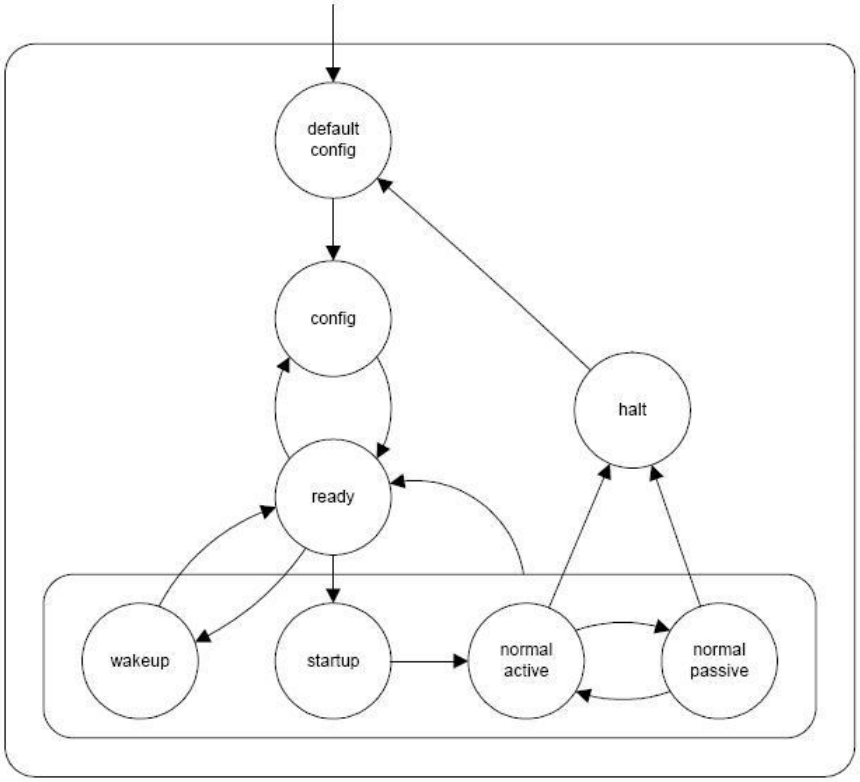

Figure 2.19: Overview of Protocol Operation Control

2.5.5.6 Controller Host Interface

The controller host interface (CHI) manages the data and control flow between the host processor and the FlexRay protocol engine within each node. The CHI contains two main interface blocks, the protocol data interface and the message data interface. The protocol data interface transfers protocol related control, configuration and status data between the host and the FlexRay protocol. The message data interface transfers messages and message related control, configuration, and status data between the host and the FlexRay protocol [25]. The architecture of the

controller host interface is shown in Figure 2.20.

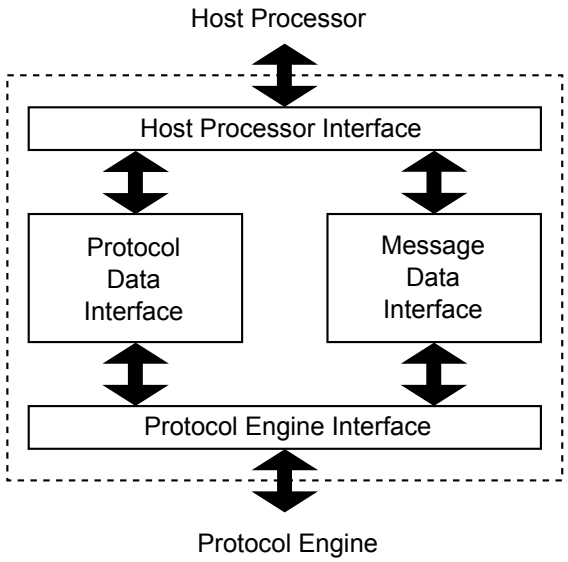

Figure 2.20: Controller Host Interface

2.6 Gateways

2.6.1 Introduction

A gateway is a device used to provide communication between networks. This can be between networks using the same protocol, or between networks running on different protocols. Gateways between networks on the same protocol can be used in the case of the network being operated at different configurations, e.g. different speeds, or if it would exceed the parameters of the system to have all the traffic on one network, i.e. data bus too long, or too much traffic on the bus. A gateway can also be used as a translator between two protocols, in this case the gateway converts the data frame format of one protocol to that of another protocol. This is the type of gateway which this section will investigate in detail.

It has been discussed in the earlier sections, that there are several different network systems in use in today's automobiles, depending on the nature of their application. Therefore a gateway system is necessary to allow these protocols communicate with one another to support the needs of the system [26]. An obvious suggestion would be to make all components compatible with one network, however if the communication network protocols can be made compatible with each other, the automotive industry will be able to advance more rapidly. For example, a CAN networked engine system can be installed in the FlexRay networked chassis of a vehicle [27]. This reduces development time and manufacturing costs by using technology which is already in place.

2.6.2 Function of a Gateway

Each gateway is unique in its application, but generally all gateways have the same basic functions. A gateway has to provide message translation, message routing, message monitoring, bus error management, and network management, all in real-time. A gateway is typically designed to minimise transmission latency (delays) and to minimise lost messages (overruns), with a the minimum demand on the CPU [26][28].

Message Translation: This is the conversion of data from one protocol format to an other. If the data contained on the message is larger than the data segment on the protocol it is being transferred to, then the message may have to be divided in to smaller sections. For example, when a 40-byte message from the FlexRay protocol must be transmitted to the CAN protocol, the gateway system will divide the single 40-byte message into five 8-byte message frames [27].

Message Routing: This allows the transferring of the message to the correct network. The routing of messages will be predetermined by the algorithm implemented by the designer. For example, all data coming in on Slot 4 on the FlexRay bus is transmitted on to the CAN bus with identifier 7.

Message Monitoring: The gateway needs to monitor the message buffers to check if messages have been received, and then act accordingly.

Error Management: Most protocols have their own built in error handling systems. However these error handling systems may not be compatible with other protocols. The gateway should be able to diagnose an error as it occurs and deal with it appropriately to stop the error propagating throughout the other networks in the system. [28]

Network Management: The gateway needs to be able to send and receive messages as they occur on either network. It has to be able to deal with the requests on one network but still stay within the parameters of the other networks on the gateway. For example, if the gateway is handling a message received from the FlexRay bus, it would need to be able to handle an RTR from the CAN bus looking for a status update on a speed sensor located on the FlexRay bus.

2.6.3 Gateway Layout

There are many ways to structure networks and their gateways, depending on the requirements of the application. However, these have been broken down in to three main categories. These are Central Gateways, Backbone Gateways and Daisy Chain Gateways [29][30]. Depending on the complexity of the system a combination of any of these types may be used. Each layout is described below highlighting the advantages and possible uses of each. For simplicity the following figures present the

physical topology only as a simple bus line, but in fact it depends on the type of communication network. For example, as we have discussed in Section 2.5.4.1, FlexRay networks can be implemented using a bus or star topology.

Central gateway: A central gateway system is shown below in Figure 2.21. In this case the gateway is being used to connect a high speed CAN, low speed CAN and FlexRay network. A typical application of this would be in an automobile, connecting the ABS sensors, brake lights, and brake by wire system. Each of these systems would require information from the other networks. All the protocol translation in this system is performed by one gateway CPU. Therefore the data will be only translated between protocols a maximum of once between nodes, reducing message latency and also the possibility of errors. A disadvantage of this system is, if this gateway fails none of the nodes will be able to communicate with a node on a different bus, also these gateways can become very complicated when dealing with a larger number of networks. This layout is best suited for when there are just two networks to be connected.

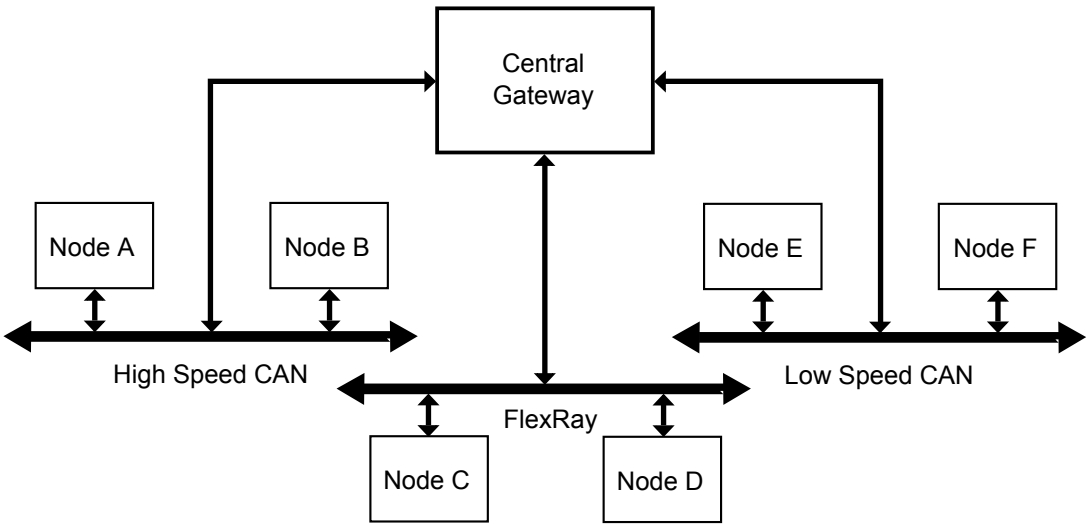

Figure 2.21: Central Gateway

Backbone Gateway: The backbone gateway system is shown in Figure 2.22. In this case the gateway is being used to connect a high speed CAN, low speed CAN and TTCAN network. Using this layout, all data passes through the backbone network. For this reason the backbone protocol must be compatible with the other protocols in the system. In this system we are using high speed and low speed CAN which are event triggered, and TTCAN which is a time triggered protocol. FlexRay supports both of these formats, so is suitable to use as the backbone network. There are a number of gateways used in this system, which helps spread the translation load amongst the gateway CPUs, thus reducing latency and error possibilities. Due to the layout of the system, any data will not be translated more than twice between nodes, further helping reduce latency and errors. An advantage of this system is that if one of the gateways fail, this will not bring down the entire system, the other networks should still be able to communicate amongst one another. This layout is best suited for where high speeds may be necessary for multiple buses.

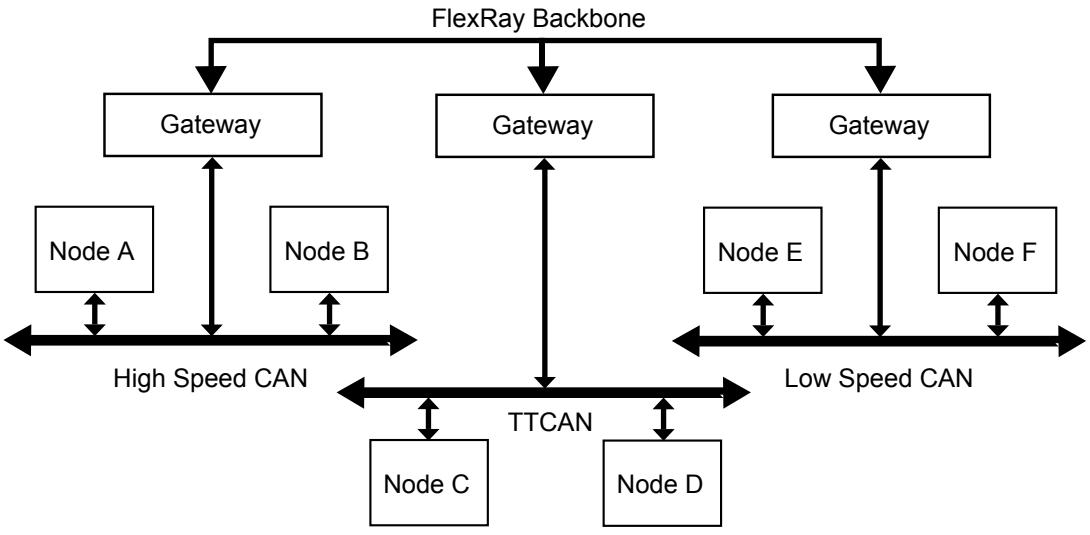

Figure 2.22: Backbone Gateway

Daisy Chain Gateway: The daisy chain gateway system is shown in Figure 2.23. In this case the gateway is being used to connect a high speed CAN and two low speed CAN networks. Each individual gateway in this system is similar to the central gateway mentioned above, but these are linked together in chain format. This system is easier to expand without having to configure an entire system, it is just a matter of adding another link to the chain. While these systems can be simpler to implement they have some disadvantages, especially as the system size and data rates increase. These systems can have high latencies between either end of the system, as the data has to pass through multiple gateways between source and destination. Also the data transfer rate can only be as fast as the slowest network in the chain. Depending on the system which is implemented, the CPUs in the central links of the system may be put under significant load as they have to deal with a large amount of traffic, which in turn may lead to errors. If one gateway fails it can isolate the networks either side of it from each other. This system is best suited for linking multiple networks with low data rates.

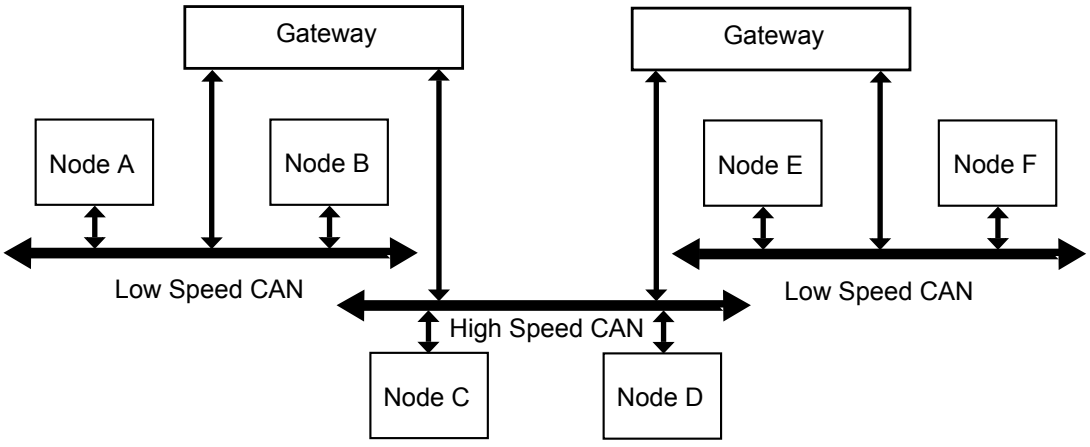

Figure 2.23: Daisy Chain Gateway

2.7 Summary

The main points outlined in this chapter were as follows:

- Local area networks allow for increased functionality while considerably reducing the cabling requirements of a system, when compared to point to point wiring.
- CAN is a well established protocol and is used on most modern cars. It is however approaching its limits as car electronic systems advance.
- FlexRay is a relatively new protocol which is aimed to meet the requirements of the next generation of automotive electronics such as steer and brake by wire. It offers high bandwidth, determinism and fault tolerance.
- FlexRay is set to become the standard for high speed fault tolerant communications in automotive networks. However CAN will still remain in use for applications such as power train and body electronics control.
- With the increasing number of networks on cars, gateways have become indispensable components within automotive networks.

The next chapter will discuss the design of a FlexRay - CAN gateway to be implemented for a vehicle speed control system.

Chapter 3

Gateway Design

3.1 Introduction

This chapter explains the methods used in designing an inter-protocol communication gateway for communication between a CAN and FlexRay network. This stage of the research was completed after the literature review and all choices made and methods used were based on the findings from the literature review. To explain the system design process undertaken for this study the chapter is divided into the following sections:

- Section 3.2, with regard to the literature review, outlines the criteria an automotive gateway must meet.
- Section 3.3 describes the physical system with which it is proposed to implement the gateway.
- Section 3.4 outlines a framework for the design of an inter-protocol gateway to communicate between a CAN and FlexRay network. This section discusses the operation of the system and the considerations designers need to take when implementing such a system.
- Section 3.5 provides a summary of the information presented in this chapter.

3.2 Gateway Requirements

Based on the findings of the literature review it is clear that the FlexRay and CAN communication protocols will be primarily used in future in-vehicle networks. Therefore FlexRay-CAN gateways are an important and indispensable component for automotive networks [31]. This chapter proposes a design for a gateway to allow communication between a CAN network and a FlexRay network. The following factors need to be considered when designing the system:

- The purpose of a gateway is to allow the transfer of data between two networks. This transfer must be performed as efficiently as possible, and with minimal latency.
- The gateway must transfer the data in a reliable and predictable manner, and must be able to deal with any sequence of events which may occur on either network.
- The gateway must be able to translate the data with 100 percent accuracy
- Any error which occurs in one network must not be able to propagate to the other network.
- The gateway must be prioritised, i.e. high priority messages must be processed first by the gateway.

3.3 Proposed Physical System

Now that the requirements have been specified, the hardware for the implementation is to be defined. It is proposed to use a gateway to interface a CAN network with N number of nodes and a FlexRay network with M number of nodes. It has been decided to use a central gateway architecture as there are just two networks to be interfaced (see Section 2.6.3). Using the central approach to designing the

gateway means any one message on either network will be translated a maximum of once, which will keep the latency of the system to a minimum and help reduce the possibility of errors. Using this approach also leaves the maximum possibility for expansion of the system. If a developer needed to extend the system it would be possible to either daisy-chain the systems or use FlexRay as the backbone of the system. The same framework could be used for each new gateway added and would not need to effect the operation of the existing gateway.

The gateway will connect to one node on the CAN bus and one node on the FlexRay bus. The configuration of the system is kept relatively basic for ease of expansion of the system for further applications. The layout of the gateway is shown in Figure 3.1.

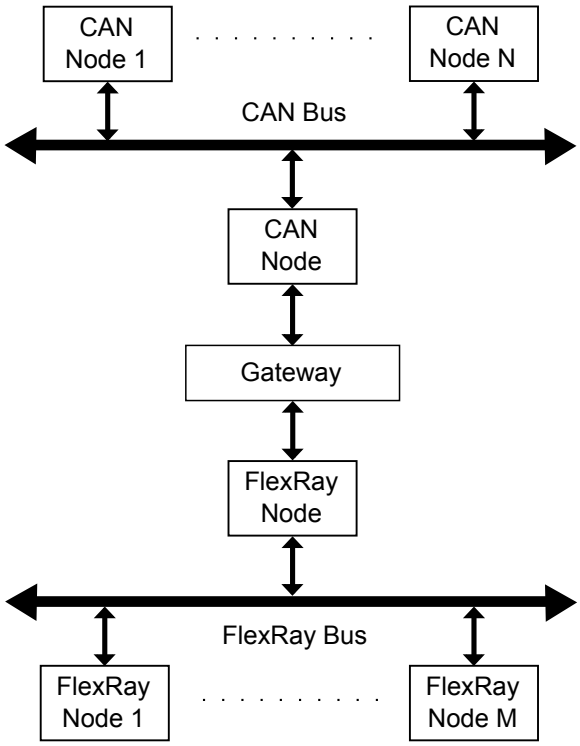

Figure 3.1: Block Diagram of Proposed System

3.4 Framework for Solution

This section describes a framework for the implementation of a gateway between FlexRay and CAN. This framework is not hardware specific, instead it is based on the topology outlined in the block diagram of the proposed system (Figure 3.1) with reference to the factors outlined in Section 3.2. The gateway consists of a standard processor, internal memory and the relevant communication controllers

When designing this framework there were two system architecture levels which were considered at which to perform protocol translation. The first option was to design at the service level where the networks communicate by directly mapping the services of one protocol to the other. For each decoded message frame it receives, the gateway simply has to issue the corresponding message frame to the service at the other side for coding and retransmission to the receiving network. The advantage of working at this level is that the gateway is easier to implement. However this is offset by a lack of flexibility of the system, as the data cannot be manipulated as it passes though the gateway. The second option was to design at a lower level, called the PDU (Protocol Data Unit), or message level, where processed messages are adapted to a suitable format for inter-network communications. For example, a FlexRay message whose payload is larger than 8 bytes would need to be broken in to smaller segments, and given the appropriate identifier before being transmitted on a CAN network. The main advantages to working at the PDU level are efficiency of data transfer and the flexibility it offers in the manipulation and optimisation of data during protocol translation. It has disadvantages however, such as being more complex to specify, and more difficult to implement than a service level gateway [32]. From the literature review it has been found that reducing processor overheads in a gateway is one of the most important requirements in a gateway meeting its goals. If a processor becomes overloaded at any time, this can itself introduce errors in to

the system. Based on the aforementioned advantages it was decided to implement the gateway at the PDU level. The architecture of a PDU level gateway is shown in Figure 3.2.

In a PDU level gateway each message is taken in from the bus and is first de-

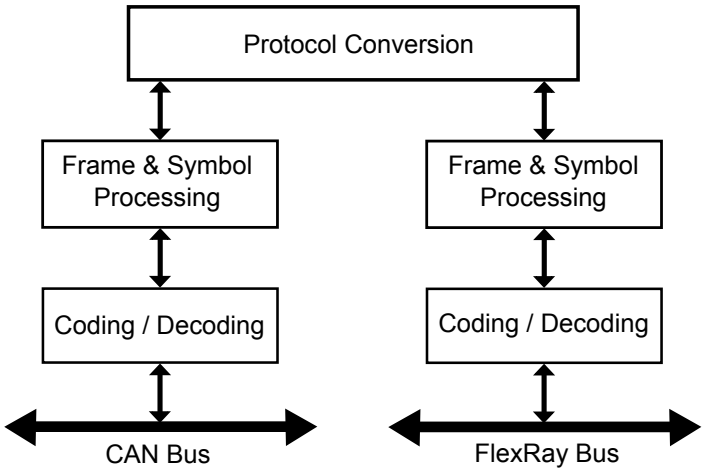

Figure 3.2: PDU Level Gateway Architecture

coded. The message frame is then processed to get the raw data from the message. The protocol conversion is then undertaken. The raw data is inserted into an appropriate frame and then coded for transmission on to the receiving bus.

3.4.1 Operation of System

This section outlines the steps the gateway takes in the transferring of a message from one network to another. The transfer process is almost identical in both directions, except where noted. The operation of the system is as follows:

- Initially the CPU waits for a message to occur on the bus, the CPU shall either poll the message buffers for change in status or have an interrupt configured to alert it of a received message. Once the message arrives to the message buffer,

the CPU then takes the information from the message buffer and stores it on its on-board memory. The source of the message shall be distinguished by the interrupt or function which initiates the process.

- The CPU will then extract the message data from the data frame and store the different fragments individually. The identifier on the received message will determine the course of action the CPU will take. If it is a CAN message destined for a FlexRay network, the CPU will convert the message ID to the format of the FlexRay network, and load all or part of the message and its identifier to the transmit message buffer.

- If it is a FlexRay message being transmitted to a CAN network there is the possibility that the message may be too large to fit in to a CAN message frame, for example a 64 byte message would be too large to fit in the 8 byte CAN frame. In this case the system will check is the message compatible for transmission on the network, i.e. less than or equal to 8 bytes. If the message meets the criteria it will then be loaded to the message buffer. If the message is too large, the CPU will send the message to the transmit buffer sequentially in blocks of 8 bytes until there is no remaining data to be transmitted. The operation of the system is illustrated in Figure 3.3.

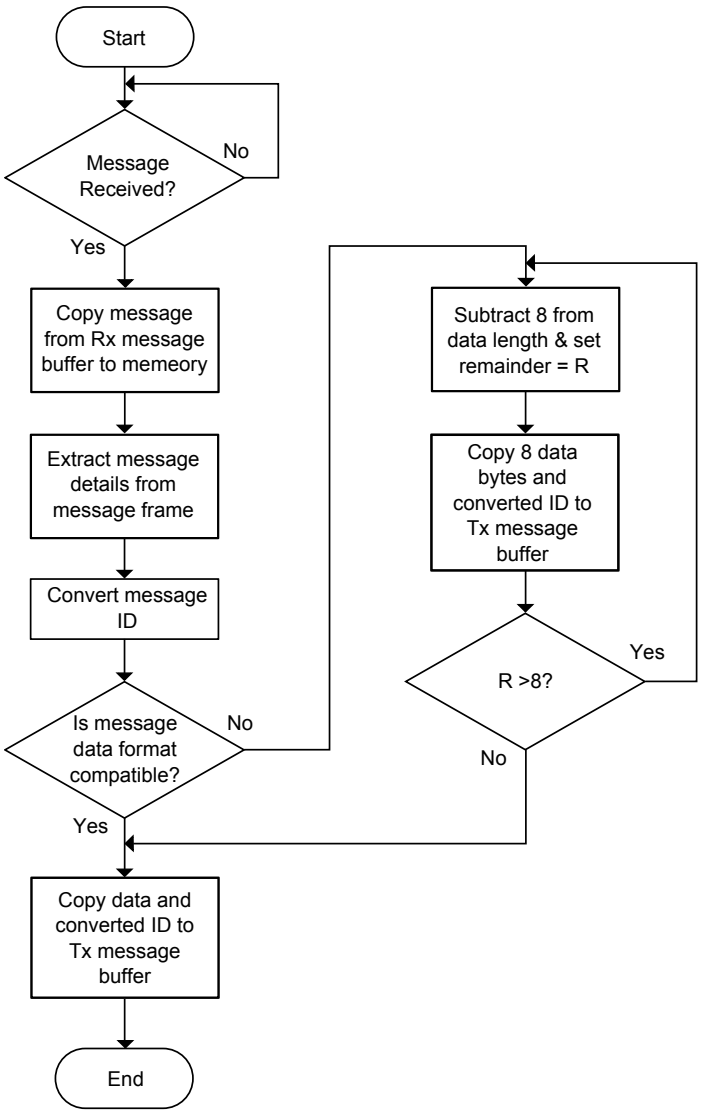

Figure 3.3: Flowchart for Operation of Gateway

3.4.2 Considerations for System Configuration

When configuring the system, care must be taken to meet the needs of both protocols, but also not to exceed their limitations. As has been discussed in Chapter Two, FlexRay has much higher data capabilities than CAN (20Mb/s compared to 1Mb/s), therefore CAN's limitations shall require the greater attention. Assuming the load on the FlexRay network is not near a critical level, the FlexRay network should be able to handle any information CAN sends through the gateway. How-

ever, as FlexRay is capable of passing up to 20 times CAN's capacity to the gateway, care must be taken during configuration to ensure that FlexRay does not pass an excessive load to the CAN network. If FlexRay continually sends messages to the CAN network at a higher rate than the CAN network is able transmit, this will lead to message buffer overruns and potential data losses. This can be avoided by keeping the data rate within CAN limits, and distributing the data destined for the CAN bus evenly among the communication cycles to allow maximum time between messages. The maximum FlexRay message payload is 32 times the size of the maximum CAN payload. Sending large chunks of data in any one message should be avoided, as these large blocks of information will need to be broken down in blocks of 8 bytes, thus causing extra latency as the message passes through the gateway.

Using the system outlined in Section 3.4, it is not necessary to implement a prioritising system within the gateway, as the communication controllers on the nodes on either side of the gateway will handle this according to their respective protocols. What is necessary however, is a correspondence in priority levels between the two networks as the gateway converts the identifiers. For example, a high priority message coming in from the CAN network should be given a relatively high priority, or possibly a slot in the static segment, when being sent to the FlexRay network. Also as the FlexRay network may be sending a large amount of data through the gateway, it will need to be given high priority on the CAN network so that the CAN message buffers will be kept available for new data coming from the FlexRay network.

The communication controllers for each network will also manage any errors which occur in their network. This means an error on one network should be contained by the communication controller and will not effect transmission in the other network. What can be beneficial however is to monitor the error status of either network

using the gateway and pass this information to the other network for subsequent use.

3.4.3 Potential Application of Gateway

A potential application of this could be the implementation of a FlexRay networked traction control engine management system. In this case the traction control system would need access to the ABS (Anti-lock Braking System) wheel speed data on the CAN bus so it can determine any speed differentials between the wheels of the car and act accordingly. This application would not exceed the limits of the gateway, as the high bandwidth of the FlexRay network should easily cope with the relatively small loading by the CAN network. For a system such as this the FlexRay network would need to relay back very little information to the CAN bus.

3.5 Summary

In this chapter the system design was taken from the problem definition and requirements, to a final gateway design. The main points covered in this chapter were as follows:

- Section 3.2 outlined the requirements for an effective gateway design, based on the findings from the literature reviewed.
- Section 3.3 proposed a physical configuration for the system based on the central gateway outlined in Chapter Two. The system was kept to a basic configuration to allow more adaptability for further applications.
- Section 3.4 described a framework for implementing a gateway in the system described in Section 3.3. This section defined the gateway architecture and the operation of the system. Considerations for designers were then discussed.

A FlexRay - CAN gateway based on the framework described in this chapter was fully implemented to verify the research. The implementation was performed using the Freescale microcontroller based Softec development board. Chapter Four shall describe the steps involved in the implementation and testing of of this system.

Chapter 4

Gateway Implementation and Testing

4.1 Introduction

This chapter will outline and explain all methods used during the system implementation and testing stage of this study. This chapter is divided into the following sections:

- Section 4.2 outlines the choices available when selecting a processing system for this project and discusses the factors which were considered for each processor.
- Section 4.3 describes the methods used in the implementation and testing of a stand alone CAN network based on the hardware described in Section 4.2
- Section 4.4 describes the implementation and testing of a FlexRay network based on the hardware described in Section 4.2
- Section 4.5 describes the hardware and software configurations of an inter-protocol communication gateway based on the framework discussed in Chapter 3, to communicate between the systems implemented in Sections 4.3 and 4.4. It also describes the testing methods.

- Section 4.6 provides a summary of the information presented in this chapter.

4.2 Processor Selection

This project involved the implementation of a gateway between a CAN and FlexRay network, thus a processor with the capability to handle these protocols needed to be selected. While a number of manufacturers offered processors which had the capability to run a FlexRay and CAN network, most involved a large amount of work interfacing their hardware with the networks, and also a large amount of configuration of the processor. After extensive research, three companies were found to offer viable options for use in FlexRay prototyping. These were:

- Fujitsu FlexRay FPGA Evaluation Kit 369 [33][34]
- Softec Microsystems - SK-S12XDP512-A Development Board containing Freescale HCS12X Processor [35][36]
- Hitex Infineon TC179x Starter Kit [37][38]

The processors to be used needed to meet the requirements of the project specification, and also to be able to operate in the harsh automotive environment. When deciding on which processor to use, the following factors were considered by the author.

- Suitability to Automotive Environment
- Support for the CAN Protocol
- Support for the FlexRay Protocol
- Programming Environment

4.2.1 Suitability to Automotive Environment

As this project is of an automotive application, the hardware must be capable of dealing with the conditions this environment contains such as vibrations, g-forces, humidity, and temperature extremes. As the components of these boards are all solid state i.e. no moving parts, they are not subject to problems associated with vibrations and g-force. To counter any problems due to humidity, the systems will need to be suitably enclosed for the environment they are placed in. In the automotive environment, ambient operating temperatures can range from -40°Cto +125°C[39]. To ensure the processors could work in this environment, their operating temperature ranges were investigated. The results are shown in Table 4.1.

As can be seen from the table, both the Infineon and Freescale processors are

Processor	Temperature Range °C
Fujitsu MB91F369	-40 to +105
Freescale MC9S12XDP512	-40 to +125
Infineon TC1796	-40 to +125

Table 4.1: Ambient Operating Temperatures of Processors

designed to operate over the full automotive temperature range, while the Fujitsu processor does not meet the upper extreme.

4.2.2 Support for the CAN Protocol

CAN support was essential for this project, Table 4.2 shows the CAN capabilities of the selected development boards. All of the processors examined have internal CAN controllers. Having internal controllers reduces processor overheads and propagation delays when compared to the alternative of using an SPI link to communicate between the CPU and an external CAN controller. The CPU in this case will out-

Development Board	Integrated CAN Controller	No. of CAN Controllers	No. of CAN Transceivers
Fujitsu	Yes	2	2
Softec	Yes	5	5
Hitex	Yes	2	2

Table 4.2: CAN Capabilities of Development Boards

put a CAN messages directly, to an on-board transceiver. The development boards contain one transceiver for each CAN controller on the processor.

4.2.3 Support for the FlexRay Protocol

FlexRay support was also essential for this project, and was the main contributing factor on the selection of a development board. Table 4.3 shows the FlexRay capabilities of the selected development boards.

In contrast to the CAN modules, all of the development boards have their FlexRay controllers separate from the main processor. These communication controllers are on daughter boards which attach to the development board. Each of the boards has one FlexRay controller and transceiver, and come with driver libraries for interfacing the controllers to the main CPU.

4.2.4 Programming Environment

The programming environment consists of an Integrated Development Environment (IDE), which is used to write the programs, debugging software, and a hardware interface for transferring the program on to the processor. Table 4.4 shows the features

Development Board	FlexRay Controller	Controller Implementation	No. of FlexRay Transceivers	FlexRay Driver Library
Fujitsu	1 x Altera EP1S25F672CFPGA	Daughter Board	1	Yes
Softec	1 x Infineon CIC310	Daughter Board	1	Yes
Hitex	1 x Freescale MFR4300	Daughter Board	1	Yes

Table 4.3: FlexRay Capabilities of Development Boards

of the environments supplied with the development boards: The Fujitsu develop-

Development Board	Programming Interface	Integrated Development Environment	Programming Language	In-Circuit Debugger
Fujitsu	USB / Serial Interface	Softune / Quartus II	C / C++ & VHDL	Yes
Softec	USB	CodeWarrior Board	C / C++ Assembly	Yes
Hitex	USB	None	C / C++ Assembly	Yes

Table 4.4: Programming Environments of Development Boards

ment board consists of two processors which need to be programmed separately, the main processor is programmed using C / C++ while the FPGA FlexRay controller is programmed using VHDL. The other two development kits may be programmed using just one language, thus reducing development time. Both the Fujitsu and the Softec development kits come supplied with an IDE as standard.

4.2.5 Synopsis of Reviewed Processors

On review of the specifications of the three development boards investigated, it was found that all three options should be capable of implementing the project. However, it was concluded from Table 4.5 that the Fujitsu development board was not best suited for this project primarily due to its programming environment. It was also decided not to use the Hitex development board as it was not supplied with an IDE or ICD, and compatibility problems may be encountered using a generic substitute. The Softec development board was chosen for a number of factors, foremost was its programming environment. The kit came supplied with an easy to use interface for programming and debugging the hardware. On inspection of the user manuals they also seemed very concise, and the FlexRay driver library was well laid out. The development board also had extra CAN modules that would be useful in further expansions of the project.

	Automotive Environment Suitability	CAN Protocol Support	FlexRay Protocol Support	Programming Environment
Fujitsu	Moderate	Sufficient	Sufficient	Moderate
Softec	Sufficient	Excellent	Sufficient	Excellent
Hitex	Sufficient	Sufficient	Sufficient	Moderate

Table 4.5: Synopsis of Reviewed Processors

4.3 CAN Implementation

4.3.1 CAN Hardware

As discussed in Section 4.2 the Freescale HCS12XDP512 processor contains five MSCAN modules (protocol controllers). The development kit contains five CAN transceivers; 2 x Motorola MC33388, 2 x Philips PCA82C250 and 1 x Motorola MC33989. These transceivers are an interface between the CAN protocol controllers and the physical bus. The devices provide differential transmit and receive capability for the CAN protocol controller at the voltage levels required for transmission on the CAN bus. They also act as a buffer between the CAN controller and the high-voltage spikes that can be generated on the CAN bus by outside sources.

The three transceiver types available on the board share similar characteristics but it was decided to use the PCA82C250 as it handles CAN transmission rates up to 1

Mbaud, has two readily usable transceivers on the development board [40][41][42].

Figure 4.1 shows the physical configuration of the CAN network which was set

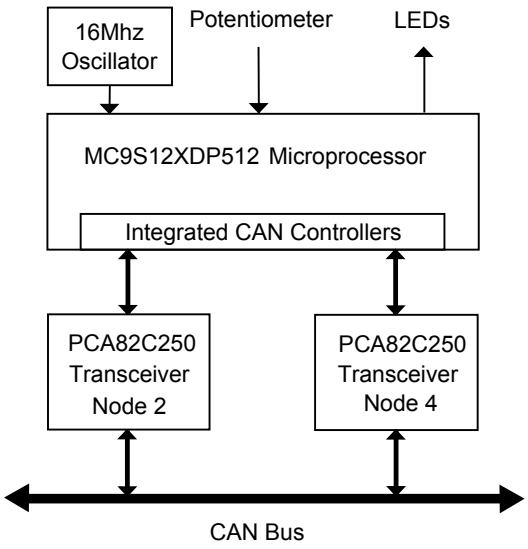

Figure 4.1: Block Diagram of CAN Hardware

up on the development system. The CAN bus was made using a twisted pair of cables with 120Ω impedance. Each node was already terminated with 120Ω resistors so it was not necessary to place these at each end of the bus. An on-board potentiometer was used to mimic a sensor input and LEDs were used as an output to monitor the activity on the bus.

4.3.2 CAN Software

The function of the first software application is to transmit a value taken from the A-D converter on the HCS12X to the CAN bus, to receive it on another CAN node and then display the value on LEDs on the development board (See Figure 4.1). This software process is illustrated in the flowchart shown in Figure 4.2. The Freescale CodeWarrior Development Studio was used to program the HCS12X microcontroller. The software contains the necessary header files and functions to help

configure the CAN modules

4.3.2.1 Analogue-to-Digital Conversion

The A-D converter on board the HCS12X is first configured to sample the voltage level coming from the potentiometer on the board. It is set for continuous conversion using register `ATD1CTL5` and eight bit resolution using register `ATD1CTL4`.

Once configured, the A-D converter is continuously polled, and after checking that the converter is not mid sequence, the value in register `ATD1DR0H` is copied to the variable `potentiometer_value` for use later when transmitting a CAN message [36]. A flowchart for the program is shown in Figure 4.2.

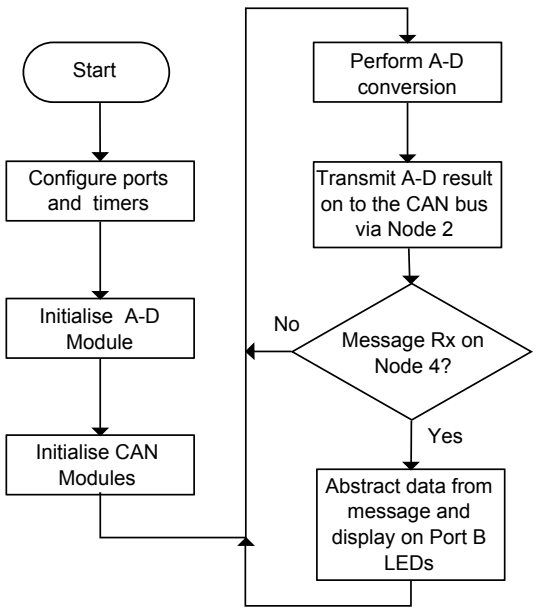

Figure 4.2: Flowchart of CAN Software Operation

4.3.2.2 CAN Message Transmission/Reception

Before communication may commence the CAN modules need to be initialised and configured (see Section 2.3.3.2). Firstly the CAN module is enabled and entered into initialisation mode. It is then set up to use the oscillator clock using the `CANCTL1` register. The CAN baud rate is set to 125 kbaud. The SJW and BRP are set to 2Tq and 4 respectively using register `CANBTR0`. The sampling rate and TSEG1 and TSEG2 are then set to one sample per bit, 4Tq and 3Tq respectively on register `CANBTR1`.[36] The module is then exited from initialisation mode and entered into operation mode.

The CAN module is set to run on a timed basis, using the Periodic Interrupt Timer (PIT) on the processor. When sufficient time has elapsed, the message information is loaded to the appropriate variables. This includes the value for the identifier field of the message, the actual data to be transmitted (in this case the value from the potentiometer), the number of bytes in the data field, the RTR status, and the message priority.

```
msg_send.id = 4;
msg_send.data[0] = potentiometer_value;
msg_send.len  = 1;
msg_send.RTR = FALSE;
msg_send.prty = 1;
```

Once these values have been loaded the function, `(void)MSCANSendMsg(MSCAN_2, msg_send)` is called to transmit the data on to the bus for reception by the other nodes.

After the message is sent CAN Node 4 is checked to see if a message has been received (`if(MSCANCheckRcvdMsg(MSCAN_4))`). If a message was received, the mes-

sage is checked to see if it has the correct identifier and is not an RTR (`if(msg_get.id == 4 && msg_get.RTR == FALSE)`). If the message meets the criteria, the value on `data[0]` is output to the LED display on PORT B. The steps involved in the transmission and reception of messages are illustrated in Figure 4.3

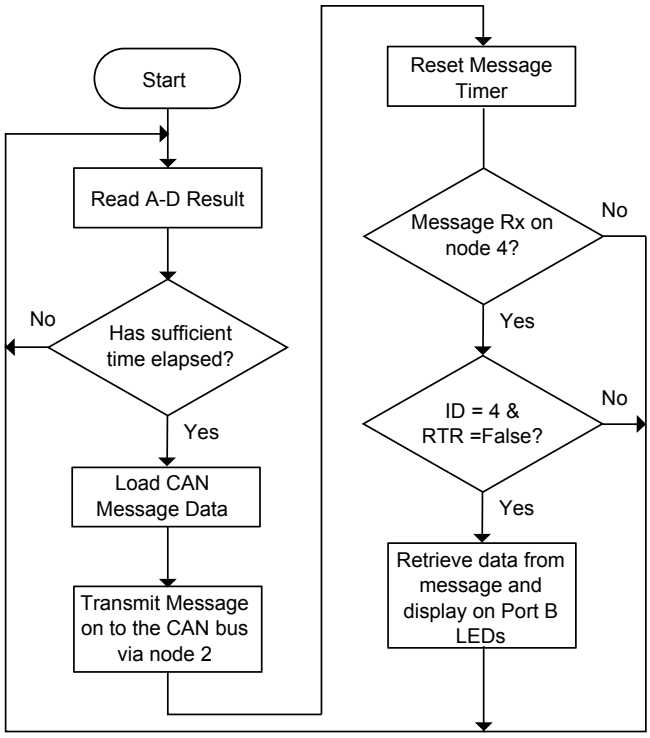

Figure 4.3: Flowchart of CAN Message Transmisson/Reception

4.3.3 Testing of the CAN Implementation

Initially the circuit was tested using the on board potentiometer as an input and the onboard LEDs as visual outputs to display the value from the potentiometer. This was principally performed to confirm that the two nodes were communicating.

To check that the program was operating within the parameters specified in the code, some further testing was necessary. It was now required to monitor the activity on the CAN bus to see exactly what traffic was being sent on the bus and what

it consisted of. To achieve this, CANalyzer network analysis tool was used. CANalyzer allows a PC/Laptop to function as a CAN node. It can transmit messages on to the bus or can just listen without interference to the bus and monitor what data the other nodes are transmitting and receiving. The tool consists of a CAN node connected to a GUI (Graphical User Interface) on a computer via a USB interface(Figure 4.4). From the GUI the user can configure the node so that it is able to synchronise with the other nodes on the bus. The GUI is then able to display numerous factors such as bus traffic and loading statistics, a history of messages transmitted/received, time-stamp information and data content for received/transmitted messages.

The CANalyzer software was set up for the configuration described in Section

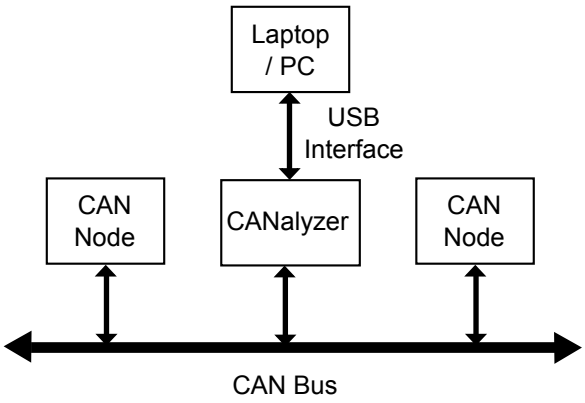

Figure 4.4: CANalyzer being used to monitor the CAN Bus

4.3.2.2, and was set to listen only mode. The following output, shown in Figure 4.5, was observed. From this it could be confirmed that the messages were being transmitted every 500ms with identifier 4. It was also observed that the full scale of the A-D was being utilised i.e. 0 - 255.

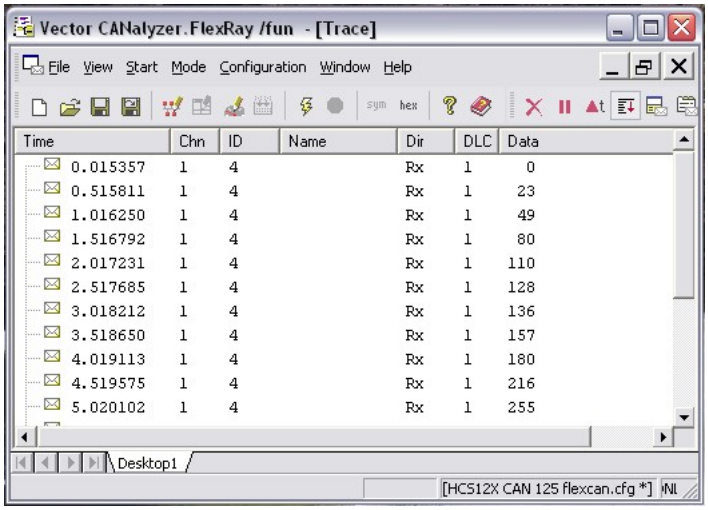

Figure 4.5: CANalyzer Screen Output

4.4 FlexRay Implementation

4.4.1 FlexRay Hardware

The MC9S12XDP512 has a built in CAN module but no built in FlexRay module. This means a separate IC is needed to convert the necessary information to the FlexRay protocol. The IC that was used for this was the Freescale MFR4300 Communication Controller [43], which in turn communicates with the Philips TJA1080 FlexRay Transceiver, to transmit / receive messages on the FlexRay bus.

Figure 4.6 shows the physical configuration of the FlexRay network that was implemented. Each node consisted of a HCS12X microcontroller connected to the FlexRay communication controller (CC) which is connected to the FlexRay Transceiver. For this implementation the CC and transceiver are mounted on one module called a daughter card. This is connected to the development board via 2 x 50 pin sockets. The daughter card also contains the various switches, jumpers and other components necessary for interfacing with the FlexRay bus.

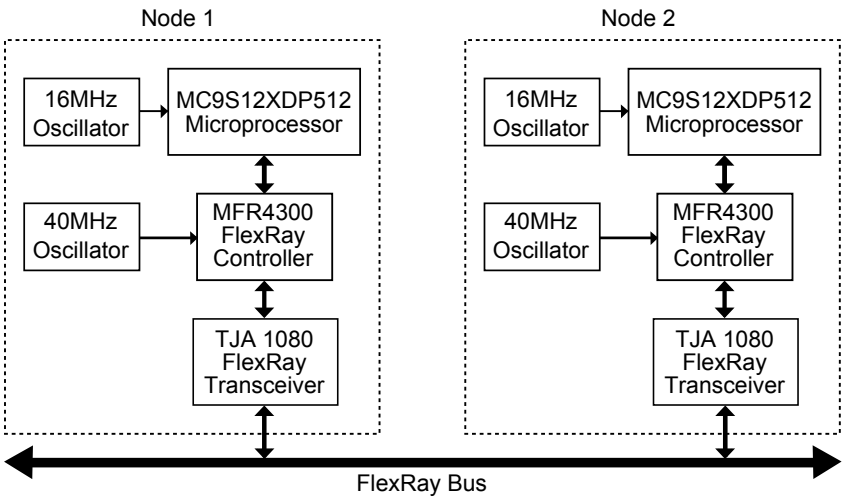

Figure 4.6: Block Diagram of FlexRay Hardware

4.4.2 FlexRay Software

Initially a test program was created by the author to verify the operation of the FlexRay hardware before it is further developed to be used in a gateway application. The function of this first FlexRay software application was to periodically transmit a value from Node 2 onto the FlexRay bus using Slot 1 in the static segment, receive this message on Node 1 where it is incremented, and return the edited value back onto the bus using Slot 4 in the static segment. From there it is received again by Node 2.

The FlexRay network was configured with the values shown in Table 4.6:

Bit Rate	10 Mb/s
Macrotick Length	1μs
Communication Cycle Length	5000MT
Static Segment	3000MT
No. of Static Slots	60
Static Slot Length	50MT
Dynamic Segment	880MT
Max. No. of Minislots	22
Minislot Length	40MT

Table 4.6: FlexRay Configuration

4.4.2.1 Node 1 Software Implementation

The operation of Node 1 software is described in Figure 4.7. The node is operated on a poll driven basis to transmit and receive messages on the FlexRay bus. For ease of testing the data is to be incremented by 100 as it is received. The incremented value is then transmitted back on the FlexRay bus.

Initialisation and Configuration: Firstly the External Bus Interface (EBI) was enabled and configured in order for the daughter card to communicate with the main processor. The Memory Mapping Control (MMC) Module was enabled in normal mode to allow the peripherals of the HCS12X to gain access to the memories available. Next the FlexRay CC was enabled and configured for the message set being used. Message Buffer 1 is used for transmission of messages and Message Buffer 2 is used for receiving messages.

Message Transmission/Reception: The data for this program is to be transmitted in message slot 4 and received from message slot 1. Node 1 is to be operated in poll driven mode. For this the program needs to continuously poll the various registers for changes in their status. This is achieved by using `if` statements within a `while(1)` loop. The code for transmitting messages is as follows:

```
tx_status = Fr_check_tx_status(TX_SLOT_4);
if(tx_status == FR_TRANSMITTED)
{
   tx_data_4[0] = rx_data_1[0] + 100;
   tx_return_value = Fr_transmit_data(TX_SLOT_4, &tx_data_4[0], 16);
}
```

The status of transmission slot 4 is checked using the `Fr_check_tx_status()` function. If the function returns that there has been a successful transmission, the data to be transmitted is given the value of the latest received message and incremented by 100. The message buffer is then updated with the new data. The code for receiving messages is as follows:

```
rx_status = Fr_check_rx_status(RX_SLOT_1);
if(rx_status == FR_RECEIVED)
{
    rx_return_value = Fr_receive_data(RX_SLOT_1,
    &rx_data_1[0], rx_data_length, &rx_status_slot);
}
```

Next the status of `RX_SLOT_1` is checked using the function `Fr_check_tx_status`. If the status of this shows that a message was received, the data from the message is copied into the array `rx_data_1[0]` where it can be used at a later stage.

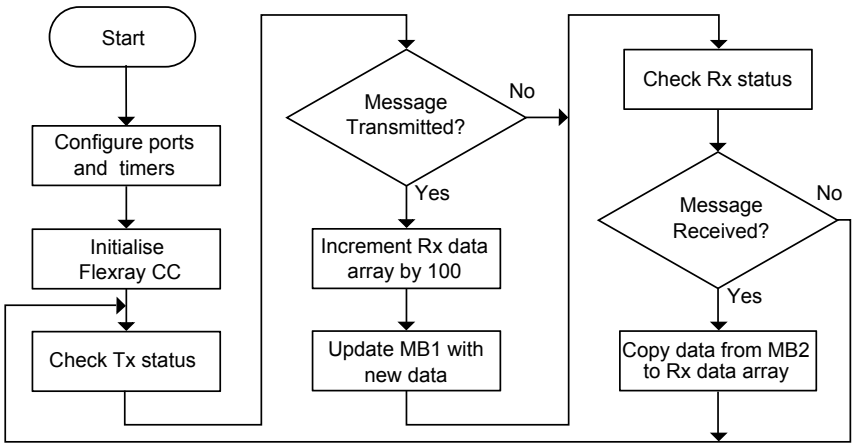

Figure 4.7: Flowchart of FlexRay Node 1 Software Operation

4.4.2.2 Node 2 Software Implementation

A flowchart of the software used in Node 2 is shown in Figure 4.8. This is run on an interrupt driven basis. It receives data from the bus and retransmits it on the same bus, but in a different slot in the dynamic segment.

Initialisation and Configuration: Node 2 is configured similarly to Node 1 as the hardware involved is identical.

Message Transmission/Reception: The data for this program is to be transmitted in message Slot 1 and received from Slot 4, both of which are part of the Static Segment. Message Buffers 0 and 1 are used for transmission, and Message Buffer 3 is used for reception of data.

Node 2 was set up to send and receive messages on an interrupt driven basis. When the status of the transmit message buffer changes, the Transmit Buffer Interrupt Flag (TBIF) sends an interrupt to the processor. From here a predefined function from the LLD is called (`Fr_set_MB_callback`), which will call a function defined in main (`CC_interrupt_slot_1`). The transmit function is as follows:

```
void CC_interrupt_slot_1(uint8 buffer_idx)
{
    tx_return_value = Fr_transmit_data(TX_SLOT_1, &tx_data_1[0], 16);
    Fr_clear_MB_interrupt_flag(TX_SLOT_1_TRANSMIT_SIDE);
    if(tx_return_value == FR_TXMB_UPDATED)
    {
        tx_data_1[0] = rx_data_4[0];
    }
}
```

The buffer number is passed to the function from the `Fr_set_MB_callback` function. Firstly the function updates the commit side of the double transmit MB with new data, it is then necessary to clear the interrupt flag on the transmit side. If the MB updating was successful then the data to be transmitted is updated with the latest received data. This will be output to the FlexRay bus on the next cycle.

When the status of the receive message buffer changes the following function is called:

```
void CC_interrupt_slot_4(uint8 buffer_idx)
{
    rx_return_value = Fr_receive_data(buffer_idx, &rx_data_4[0],
    &rx_data_length, &rx_status_slot);
}
```

The buffer number is passed to the function from the `Fr_set_MB_callback` function. The received data is then copied in to the given array for later use [44].

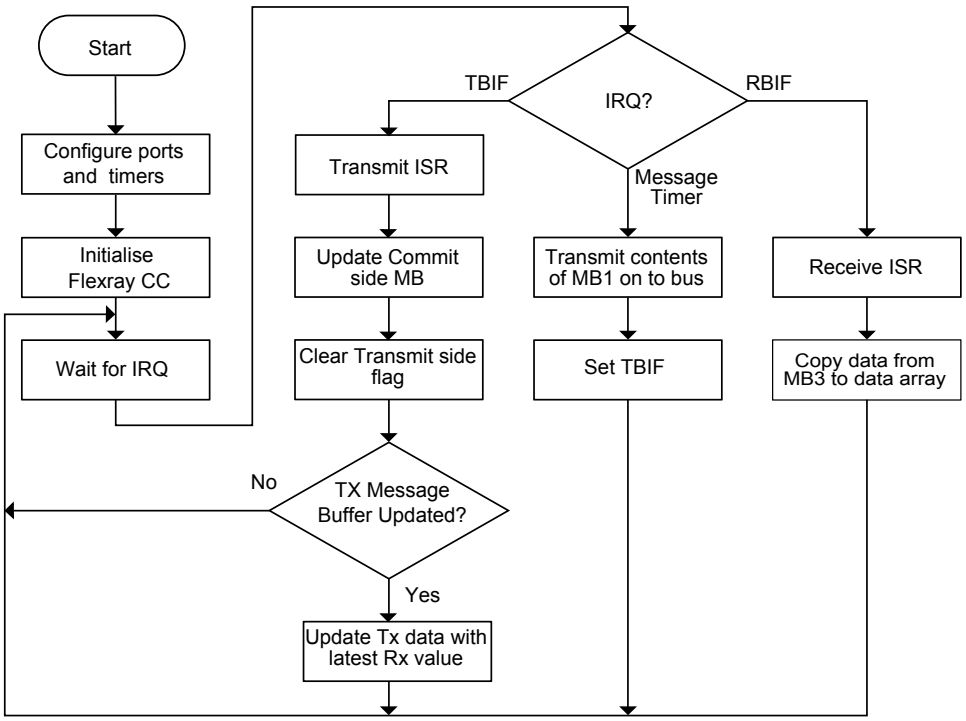

Figure 4.8: Flowchart of FlexRay Node 2 Software Operation

4.4.3 Testing of the FlexRay Implementation

A tool called FreeMASTER was used to monitor activity on the FlexRay bus. FreeMASTER is a GUI developed by Freescale, for use with their processors, to display real time values from various registers within a microprocessor onto the screen of a PC / Laptop.

To set up an application, a user must first place a driver in the embedded code which allows the GUI access to the registers that are to be monitored. The information is exported from the HCS12X via a SCI (Serial Communication Interface) link to an RS232 transceiver, from where it is then transmitted to the Serial Port of the PC (See Figure 4.9). The GUI also needs to be configured to take the information from the relevant registers within the HCS12X. Once this is complete the user can manipulate the received data to display it on the screen in numerical, graphical or message based form. [45] For this test application it was required to read the values

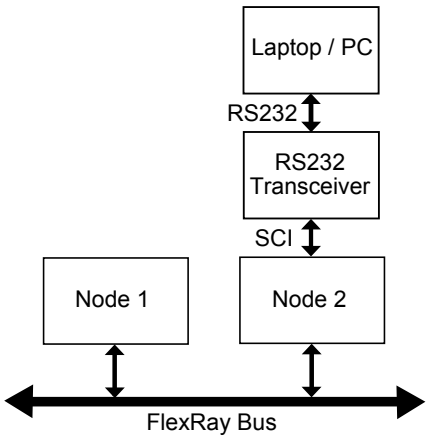

Figure 4.9: FreeMASTER being used to monitor variables on Node 2

transmitted and received on the FlexRay bus by Node 2. It was expected to see a difference of 100 between the data received and the data transmitted, this would show that the data has in fact been incremented as it passed through Node 1. For this, access to the variables `tx_data_1[0]` and `rx_data_4[0]` was required. To enable access to these variables from the FreeMASTER software, the SCI on the HCS12X needed to be enabled. This was configured to 19200 baud using register `SCI0BDL`. The FreeMASTER GUI had to be configured to connect to the serial port on the PC, and to extract the relevant data from the registers on the development board. There were initially some communication errors when connection was attempted. It was discovered that the default setting was for the GUI to connect to the COM5 port on the PC, once this was changed to COM1 port error free communication was achieved. The variables were output to the screen in graphical and numerical format, see Figure 4.10.

The output observed on the screen was of two sawtooth waveforms, these were caused by the variables incrementing from 0 up to 65535 and once they reached this maximum value the data rolls over to zero and begins again. When zoomed out from the signal it is quite difficult to see the difference between the two signal traces, but

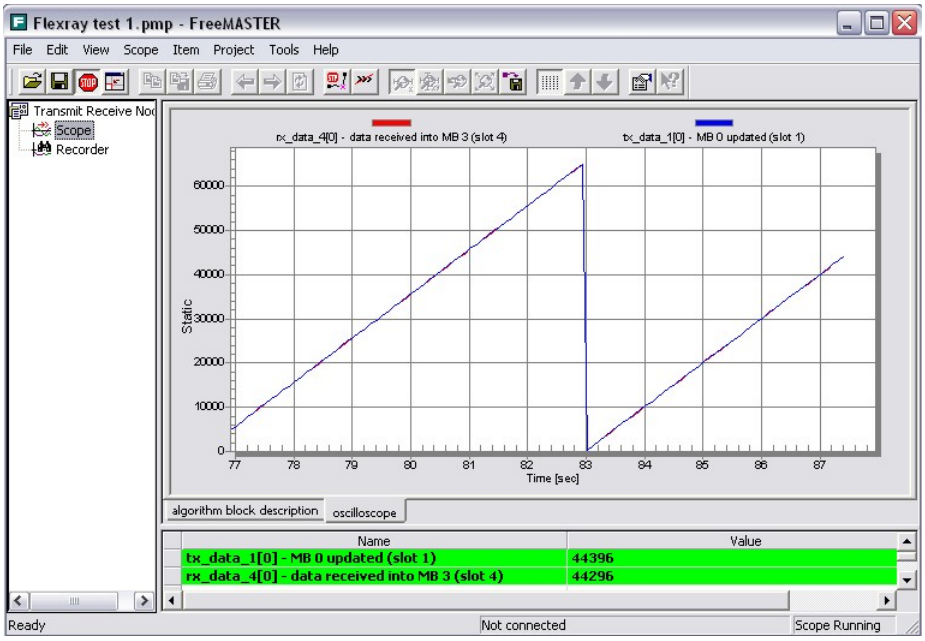

Figure 4.10: FreeMASTER Screen Output

when zoomed in as in Figure 4.11 we can see, as expected, that the received value is the transmitted value but offset by 100. This can also be seen in the numerical representation of the data on the bottom of the screen.

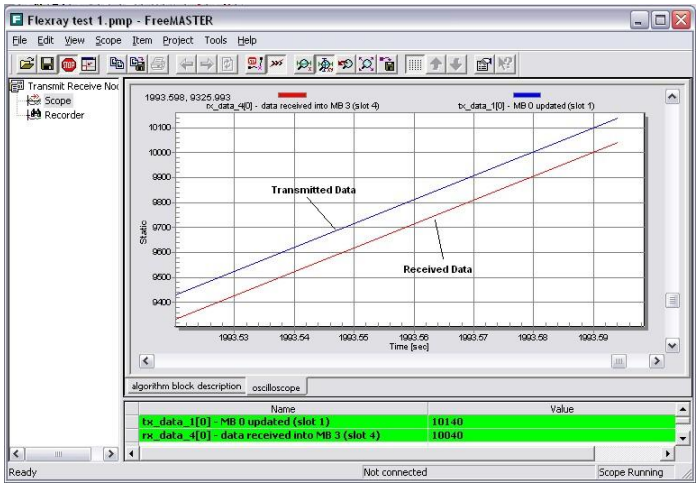

Figure 4.11: FreeMASTER Screen Output (time scaled)

4.5 Gateway Implementation

4.5.1 Gateway Hardware

The general layout of this application consists of two FlexRay nodes (1 and 2) which are operated by Freescale HCS12X microcontrollers. These connect to the network via FlexRay communication controllers and transceivers mounted on daughter card modules. The processor on FlexRay Node 2 also acts as a gateway between the FlexRay bus and a CAN bus. This processor has an internal CAN controller and connects to the CAN bus via a transceiver mounted on the development board. Figure 4.12 shows the physical configuration of the hardware. The sensors are connected to the microcontroller by point to point wiring. The sensor data is passed to the FlexRay bus by FlexRay Node 1. FlexRay Node 2 receives this data and passes it to CAN Node 2 via a FlexRay - CAN gateway. The data is then transferred on to the CAN bus for subsequent use. The data flow through the system is shown in Figure 4.13

4.5.2 Gateway Software

The function of the software in this application is to use a HCS12X processor as a gateway to transfer data between a FlexRay network and a CAN network. Information from sensors is taken in by the processor controlling FlexRay Node 1. It is then broadcast on to the FlexRay bus where it is received by FlexRay Node 2. The processor controlling FlexRay Node 2 receives the FlexRay message, extracts the data from the message and subsequently translates the data to CAN message format for broadcasting on to the CAN bus by CAN Node 2. From here the data is taken and output on a visual display. As we have discussed in Section 2.6.2, the system also needs to be able to manage bus error issues as they occur. The FlexRay

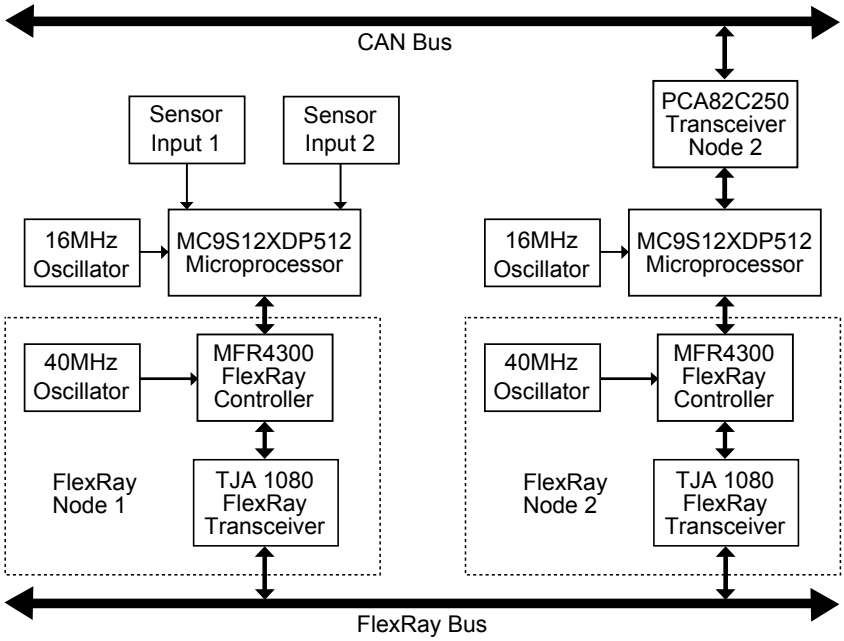

Figure 4.12: Block Diagram of Gateway Hardware

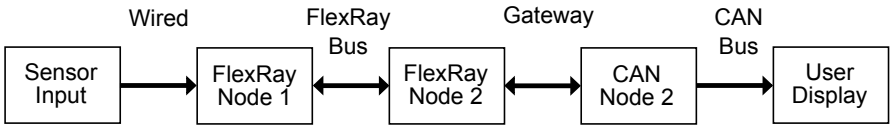

Figure 4.13: Data Flow through System

nodes output their error status to the FlexRay bus from where it is transferred to the CAN bus for displaying to the user. As the CAN bus is relying on the FlexRay network to supply it with information, the CAN messages are no longer sent on a timed basis. The messages are now event driven. When the processor receives the relevant information from the FlexRay bus the CAN send functions are invoked. Therefore the gateway only adds traffic to the bus when needed, making it more efficient on bandwidth. The gateway software section is broken in to two parts, data handling and error handling.

The FlexRay and CAN networks were configured with the values shown in Table 4.7:

Flexray		CAN	
Bit Rate	10 Mb/s	Bit Rate	125kb/s
Macrotick Length	1μs	SJW	2Tq
Communiaction Cycle Length	5000MT	BRP	4
Static Segment	3000MT	TSEG1	4Tq
No. of Static Slots	60	TSEG2	3Tq
Static Slot Length	50MT		
Dynamic Segment	880MT		
Max. No. of Minislots	22		
Minislot Length	40MT		

Table 4.7: FlexRay and CAN Configuration

4.5.2.1 Data Handling

Node 1: The purpose of this node is to retrieve data from two sensor inputs and periodically transmit the data on to the FlexRay network. It is to transmit the data using Slot 4 using Message Buffer MB 1. It is operated on a polled basis using `if` statements within a `while(1)` loop as follows:

```
tx_status = Fr_check_tx_status(TX_SLOT_4);
if(tx_status == FR_TRANSMITTED)
{
    tx_data_4[0] = sensor1data;
    tx_data_4[1] = sensor2data;
    tx_return_value = Fr_transmit_data(TX_SLOT_4, &tx_data_4[0], 16);
}
```

As we can see from the flowchart in Figure 4.14, the software first checks the status of transmission slot 4 using the function `Fr_check_tx_status()`. If the function

returns that there has been a successful transmission, then the data to be transmitted is given the values of the most recent data taken in from the sensors. The message buffer is then updated with the new data.

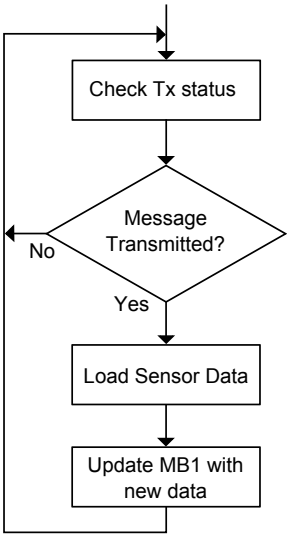

Figure 4.14: Flowchart of FlexRay Node 1 Software Operation

Node 2: The purpose of Node 2 is to receive messages from the FlexRay bus which have been transmitted by Node 1 in Slot 4. It is then to parse the data from the FlexRay message and transfer it to a CAN message and transmit this message on to the CAN bus via CAN Node 2. This data can be used by other nodes on the CAN network. The FlexRay messages on Node 2 are handled by interrupts, while the CAN messages are handled on a polled basis. For ease of understanding, the flow chart in Figure 4.15 below has been broken up in to two parts, the interrupt driven FlexRay section, and the poll driven CAN section.

The FlexRay receive code is as follows:

```
void CC_interrupt_slot_4(uint8 buffer_idx)
{
```

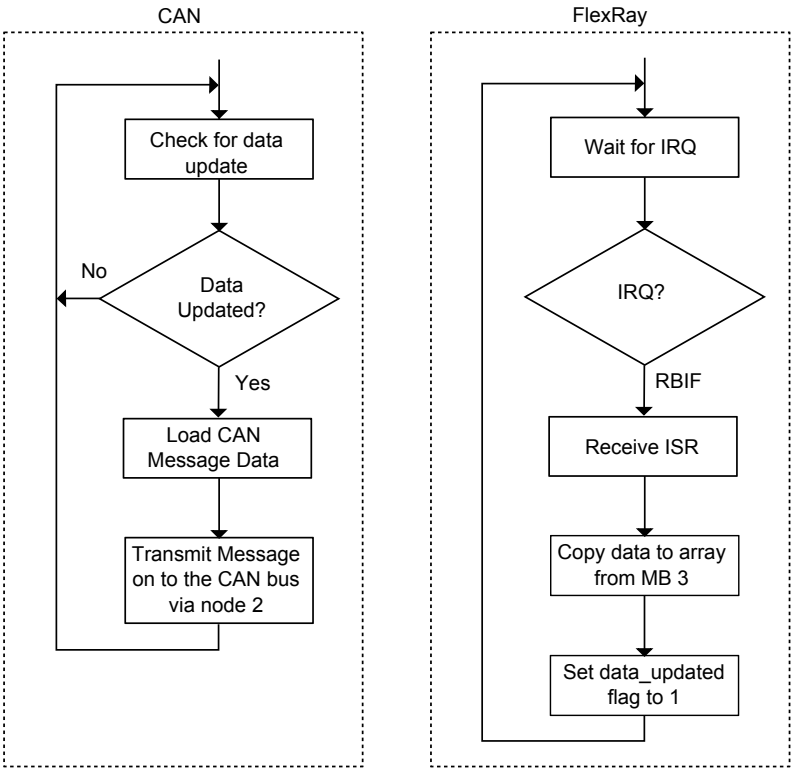

Figure 4.15: Flowchart of FlexRay Node 2 Software Operation

```
    rx_return_value = Fr_receive_data(buffer_idx, &rx_data_4[0],
    &rx_data_length, &rx_status_slot);
    cantx0 = rx_data_4[0];
    cantx1 = rx_data_4[1];
    data_updated = 1;
}
```

When the status of the receive message buffer changes, the Receive Buffer Interrupt Flag (RBIF) sends an interrupt to the processor which results in the `CC_interrupt_slot_4()` function being called. The buffer number is passed to the function from the `Fr_set_MB_callback` function. The received data is then copied to the `rx_data_4` array. From here the relevant data is copied to variables for use by the CAN code. The `data_updated` flag is then set to 1. The `rx_data_4` array acts as a buffer, avoiding CAN code access to the FlexRay variables. If the CAN

code has a FlexRay variable locked for a read as the FlexRay software wished to update the variable, it may cause data corruption.

CAN message transmission is performed by the following code:

```
if( data_updated == 1 )
{
 do{
  msg_send.id = 2;
    msg_send.data[0] = cantx0;
    msg_send.data[1] = cantx0 >> 8;
    msg_send.data[2] = cantx0 >> 16;
    msg_send.data[3] = cantx0 >> 32;
    msg_send.data[4] = cantx0 >> 40;
    msg_send.data[5] = cantx0 >> 48;
    msg_send.data[6] = cantx0 >> 56;
    msg_send.data[7] = cantx0 >> 64;
    msg_send.len = 8;
    msg_send.RTR = FALSE;
    msg_send.prty = 1;
    (void)MSCANSendMsg(MSCAN_2, msg_send);
    cantx0 = cantx0 >> 64;
   } while(cantx0_length != 0);
   do{
  msg_send.id = 3;
    msg_send.data[0] = cantx1;
    msg_send.data[1] = cantx1 >> 8;
    msg_send.data[2] = cantx1 >> 16;
    msg_send.data[3] = cantx1 >> 32;
```

```
    msg_send.data[4] = cantx1 >> 40;
    msg_send.data[5] = cantx1 >> 48;
    msg_send.data[6] = cantx1 >> 56;
    msg_send.data[7] = cantx1 >> 64;
    msg_send.len = 8;
    msg_send.RTR = FALSE;
    msg_send.prty = 1;
    (void)MSCANSendMsg(MSCAN_2, msg_send);
    cantx1 = cantx1 >> 64;
   }while(cantx1_length != 0);
   data_updated = 0;
} ;
```

This code is run within a `while(1)` loop, so was continuously polled while no ISRs are being executed. Firstly the code checks the `data_updated` flag to establish if any relevant messages have been received from the Flexray bus since the last CAN message was transmitted. On confirmation of this, a `do while` loop is used to transmit the data to the CAN bus. The loop executes once to load the information to the appropriate variables and then calls the function `(void)MSCANSendMsg(MSCAN_2, msg_send)` to transmit the data to the CAN bus via Node 2. If there is still data remaining from the FlexRay message, the loop runs again until all the information has been transmitted. There is a separate loop for each variable received from the FlexRay network. Once the data has been transmitted, the `data_updated` flag is reset to 0. This flag may only be reset in the `CC_interrupt_slot_4` function, so the above routine will not be entered again until another FlexRay message is received.

4.5.2.2 Bus Error Handling

In the previous section transmission of data between the FlexRay bus and the CAN bus using a gateway was outlined. This section discusses the implementation of an error handling system which will alert the user of various errors which may occur on the FlexRay bus by transmitting the details of the error on to the CAN bus. The processor will wait for a change in error status on the FlexRay network, and once this change occurs it is passed to the gateway, from where it is passed on to the CAN network.

Node 1: For Node 1 two different error states were investigated: CHI (Controller Host Interface) Error, and Message Buffer Access Error. The CHI Error Flag (`CHIERF`) can be set for various reasons including: a Protocol Operation Control (POC) command being ignored due to being busy executing another command, FIFO overrun error, System Bus communication error, frame id error, Message Payload errors, Network Management errors and illegal memory access errors [43]. The operation of Node 1 is illustrated in Figure 4.16. To determine if a CHI Error has occurred the function `Fr_check_CHI_error()` is called. This returns the value in the `CHIERFR` register. This register may contain one or more flags due to the different errors mentioned above. If this function returns a non-zero value the variable `chi_error` is incremented. The CHI error state is checked once during each iteration of the `while(1)` loop.

```
if(Fr_check_CHI_error() != 0) chi_error++;
```

A message buffer access error occurs when the the invoked data read/write function is unable to lock the message buffer for use. The function in this case is `Fr_transmit_data()`. If it is unable to lock the message buffer it will return the parameter `FR_TXMB_NO_ACCESS`. At the end of each message receive or transmit at-

tempt this value is checked. If it shows that the function was unable to lock the message buffer, the variable `mb_access_error` will be incremented.

```
if(tx_return_value == FR_RXMB_NO_ACCESS) mb_access_error++;
```

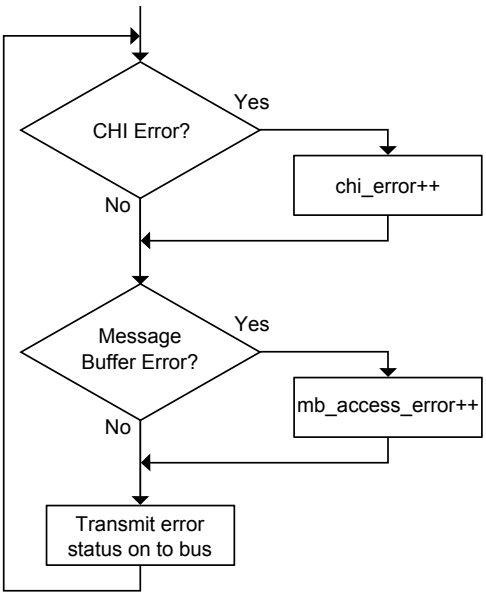

Figure 4.16: Flowchart of FlexRay Node 1 Error Detection

The following code transmits the error data on to the FlexRay bus:

```
tx_status = Fr_check_tx_status(TX_SLOT_4);
if(tx_status == FR_TRANSMITTED)
{
      tx_data_4[2] = chi_error;
      tx_data_4[3] = mb_access_error;
      tx_return_value = Fr_transmit_data(TX_SLOT_4,
      &tx_data_4[0], 16);
}
```

Node 2: Node 2 is programmed to receive any changes in error states from Node 1 and pass them to the CAN bus. It also monitors any POC errors which may occur.

The reason this is done by Node 2 is that the protocol manager monitors the state of the whole bus, and keeps all nodes up to date. Also if the protocol went into error on the node it would still be possible to output this to the CAN network. If this is monitored in Node 1, the communication between FlexRay nodes may have been lost before the error state could be passed on. The following code was used to implement the POC error check and pass the information to the CAN bus:

```
if(Fr_check_protocol_state_changed())
  {
    protocol_state = Fr_get_POC_state();
    if (protocol_state == FR_POCSTATE_CONFIG)
      flexpoc_state = 0;
    else if (protocol_state == FR_POCSTATE_DEFAULT_CONFIG)
      flexpoc_state = 1;
    else if (protocol_state == FR_POCSTATE_HALT)
      flexpoc_state = 2;
    else if (protocol_state == FR_POCSTATE_NORMAL_ACTIVE)
      flexpoc_state = 3;
    else if (protocol_state == FR_POCSTATE_NORMAL_PASSIVE)
      flexpoc_state = 4;
    else if (protocol_state == FR_POCSTATE_READY)
      flexpoc_state = 5;
    else if (protocol_state == FR_POCSTATE_STARTUP)
      flexpoc_state = 6;
    else if (protocol_state == FR_POCSTATE_WAKEUP)
      flexpoc_state = 7;
    else
      flexpoc_state = 8;
```

```
    msg_send.id = 17;
    msg_send.data[0] = flexpoc_state;
    msg_send.len = 1;
    msg_send.RTR = FALSE;
    msg_send.prty = 1;
    (void)MSCANSendMsg(MSCAN_2, msg_send);
} ;
```

Firstly the `Fr_check_protocol_state_changed()` function is used to check whether the protocol state has been changed. The `Fr_get_POC_state()` function then queries the current value of the Protocol Status Register (PSR0) and returns the current protocol state. This value is passed to the variable `protocol_state`, which in turn, uses an if-else-if chain to change the character string result from the function into integer format. Once this is complete the message is then transmitted on the bus for use by the end user. The flowchart for this operation is shown in Figure 4.17.

The controller host interface and message buffer errors are passed on to the CAN

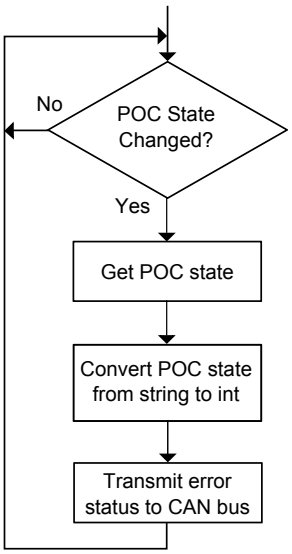

Figure 4.17: Flowchart of FlexRay Node 2 Error Detection

bus in a similar fashion as with the protocol operation control error. These messages do not contain details of an error state, but just inform the user that an error state has actually occurred. Each time that FlexRay Node 2 receives an incremented CHI or MB error value it will transmit a message to the CAN bus, otherwise the transmit function will not be called.

4.5.3 Testing of FlexRay-CAN Gateway Operation

In previous sections of this chapter the correct operation of the FlexRay and CAN busses have been outlined. This section will concentrate on the transfer of data and error information through the gateway on Node 2. A visual interface is to be connected to the CAN bus and used to monitor data traffic on the bus and also display bus errors which have been passed through the gateway from FlexRay.

The visual interface used for testing this application was LabVIEW from National Instruments. LabVIEW, an industry standard tool, is a graphical programming environment used for designing test, measurement and control systems. It uses graphical icons and wires to create its programs in a style similar to that of a flowchart [46] (see Figure 4.18).

It was decided to use LabVIEW to test the gateway application as it has specific hardware and software interfaces for communicating with a CAN network. Its was chosen to perform the gateway testing instead of CANalyzer as it has the capability of building user friendly data interfaces, such as dials, gauges etc. for displaying data received from the bus, rather than simply relying on the user to interpret the data. The LabVIEW tool is connected to the gateway hardware as shown in Figure 4.19

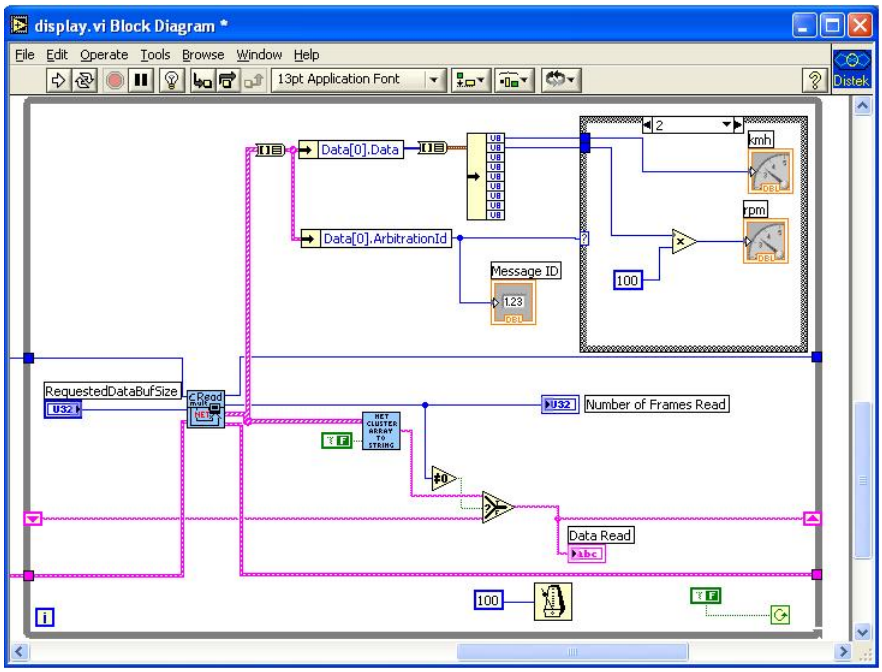

Figure 4.18: Example of LabVIEW Code

The LabVIEW code was built using function blocks which were part of the ap-

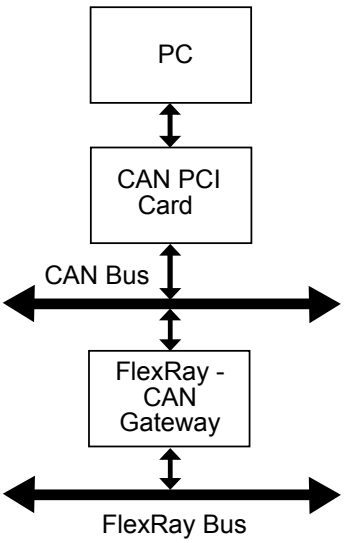

Figure 4.19: LabVIEW monitoring Data and Error Status from the Gateway

plication software. A block was set to configure the CAN-PCI card so that it could communicate with the bus. The next block read the messages from the bus, and sorted them by their identifiers. The inputted data was then displayed on the Lab-

VIEW front panel. The final block was used to handle any errors which occurred on the CAN network. The display was set up as shown in Figure 4.20. It was designed to look similar to an instrument display panel in a car, which would be familiar to most users. The data from the two sensors was displayed on dials and represented vehicle speed in km/h and engine speed in rpm. The error state of the FlexRay network was displayed using LEDs, green representing normal operation and red representing an error having occurred. As there were a number of different POC error states, the state number for a POC error was also displayed. During testing the results observed were as expected, the dials varied linearly with the sensor inputs across the full scale of the potentiometers. The error state LEDs stayed green under normal operation, but the POC error turned to red and gave us an error number 4 when an error was introduced on to Node 1 by temporarily resetting the processor.

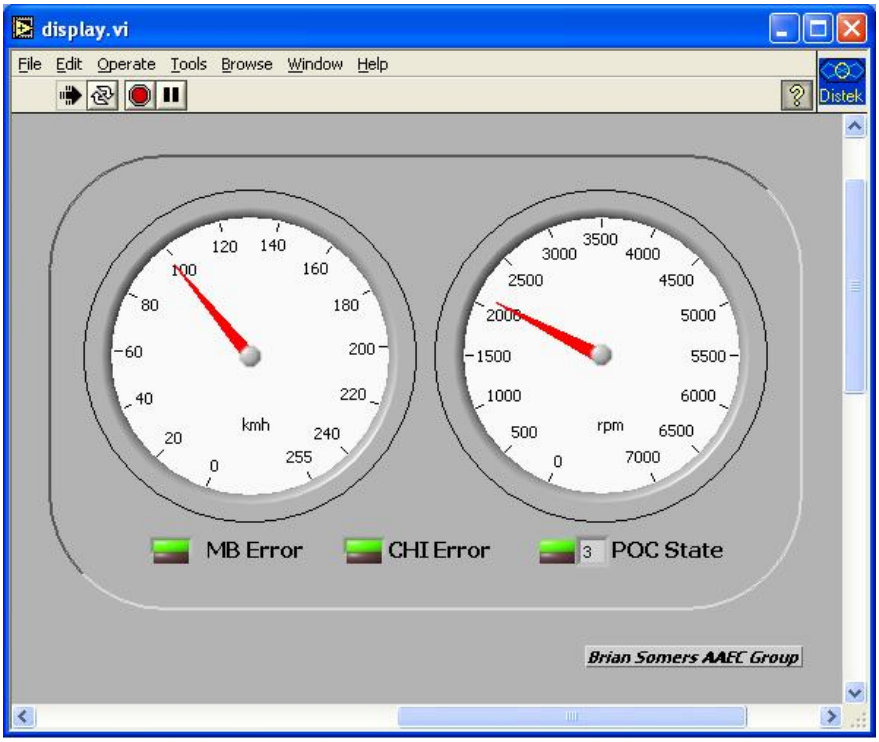

Figure 4.20: LabVIEW Dash Panel

4.6 Summary

This chapter described the methods employed and choices made when implementing a inter-network communication gateway using the framework proposed in Chapter 3. The application of the gateway was to monitor two inputs representing engine and road speed on the FlexRay network and transmit this data aswell as the error status of the FlexRay network to a dashboard display communicating with the CAN bus.

The implementation was divided into three sections as follows:

- Section 4.2 reviewed the hardware options available for the implementation of this project and found the Softec Microsystems - SK-S12XDP512-A Development Board to be most suitable.

- Section 4.3 described the configuration, implementation and testing of a CAN network. The aim of this application was to verify the correct operation of the CAN hardware for later implementation in the gateway system. This network transmitted data from a potentiometer via the CAN bus to a receiving node where the data was represented graphically with LEDs. The system parameters were then tested using the network analysis tool CANalyzer.

- Section 4.4 described the configuration, implementation and testing of a FlexRay network which passed data between two FlexRay nodes. Each time the data passed through Node 1 the data was incremented. The system was tested using the FreeMASTER monitoring tool, which found the network to be working successfully. This once again verified the correct operation of the hardware which was to be used in the next stage of the project.

- Section 4.5 outlined the steps involved in creating a gateway between a CAN and FlexRay network. The gateway was used to transmit data from two sen-

sors on the FlexRay bus to a dashboard communicating via the CAN network. The gateway also monitored the error status of the FlexRay network and also transmitted this data to the dashboard via the CAN bus.

Chapter 5

Conclusions

5.1 Introduction

This chapter outlines the results and conclusions that have been drawn from the project and offers suggestions on further possibilities of research for this project.

- Chapter Two outlined the relevant information from the literature reviewed during the course of this research. It gave an overview of automotive networks, paying particular attention to the CAN and FlexRay protocols and inter-protocol gateways. The process involved in selecting a suitable processor for this project was also described.

- Chapter Three, with regard to the literature reviewed in Chapter Two, described a framework for the implementation of an inter-protocol gateway between a CAN and FlexRay network. This chapter discussed the requirements and the factors which needed to be considered when designing such a system. The application of this framework in a real world system was then discussed.

- Chapter Four discussed the implementation in hardware and software of a system based on the framework laid out in Chapter Three. The system was

an inter-protocol gateway which was used to transmit vehicle speed, engine speed and error data from a FlexRay network to a dashboard communicating with a CAN network. The development of the system was defined in three stages, the implementation of a CAN network, a FlexRay network, and finally the inter-protocol gateway system. The chapter also outlined the different testing methods and the tools used to verify the systems operation.

5.2 Conclusions

Due to the increase in electronics requirements in modern vehicles, the CAN protocol is starting to reach its operational limits. FlexRay is set to become the standard for communication for more advanced networking applications such as drive by wire. As CAN will still be used for body electronics, dash panel and engine management networking etc., inter-protocol gateways will be required to allow communication between these protocols.

The aim of this research thesis was to investigate an inter-protocol gateway for communication between a CAN and FlexRay network. The author described a framework for the implementation of such a system, with the the aim of reducing CPU loading by the gateway, and maximising flexibility of implementation. The framework was then implemented in a real application. The function of the application gateway was to monitor the data flow and error status on the FlexRay bus and transmit the relevant data in real time to a dashboard via a CAN bus. The data on the FlexRay bus consisted of information from two sensors connected to one node on the FlexRay bus.

After a review of the development boards available for use with this project, the Softec development environment containing a Freescale HSC12X processor was cho-

sen for use in the implementation of the system, as it adequately met all of the key requirements for the system. The CodeWarrior Development Studio was used for the coding of the application, and programming of the HCS12X processor. The operation of vehicle speed sensors was imitated by using potentiometers connected to a FlexRay Node. The dash panel was implemented using LabVIEW as it offered the flexibility to design a system specific interface. The layout of the hardware for the system is shown in Figure 5.1

As mentioned above, the data on the FlexRay bus was represented on a dash

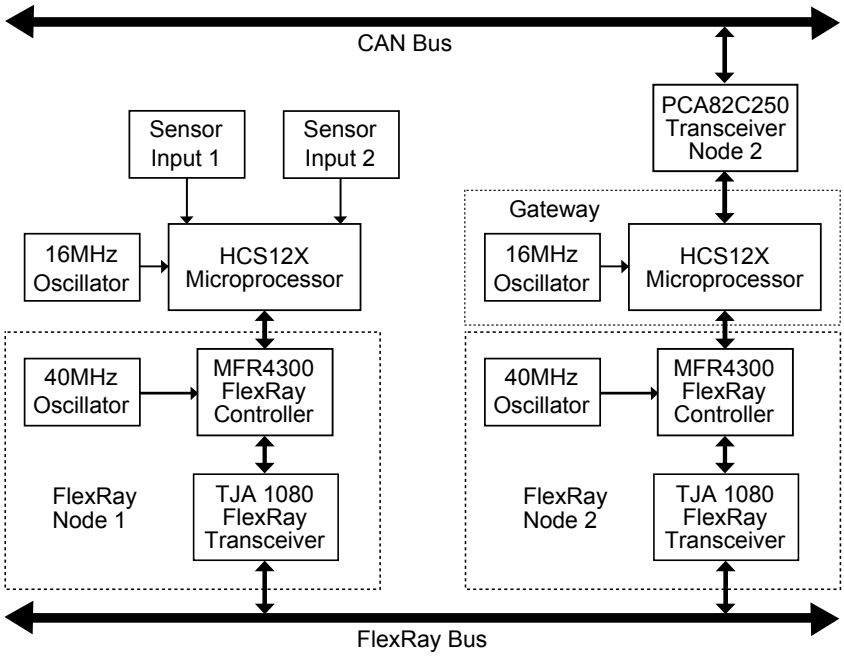

Figure 5.1: Gateway Hardware Layout

panel. The gauges, which represented the data on the FlexRay bus, updated in synchronisation with the rotation of the potentiometers. The LEDs which represented the error state of the FlexRay network on the dashboard also changed state as errors were introduced on to the network. Therefore real time representation of data through the network was achieved. Real time operation is paramount in automotive systems as the devices (or operator) in the vehicle have to be able to

respond immediately to any changes which may occur in the system.

The system implemented in this thesis used the inputs of two sensors as a data source on the FlexRay bus in order to prove the concept. Extra sensors could be added to this system using the same methodology. The gateway linked one CAN network with one FlexRay network using the framework outlined in Chapter three. This can be expanded to encompass a number of networks using the framework described, and allowing various topologies to be used.

To gain exposure to a broader range of different network analysis tools, the author used three different tools, CANalyzer, FreeMASTER and LabVIEW, for the analysis of the designed system. In hindsight it may be have been a better option to use just one package such as LabVIEW which is capable of handling data from all parts of the system. This would make the comparison and timing analysis of the systems simpler and more efficient.

5.3 Further Research

The gateway described in this research links a CAN and a FlexRay network. A further expansion of this system would be to design a capability to communicate with other protocols which are used in automotive networks such as LIN (Local Interconnect Network) or TTP (Time Triggered Protocol). Although TTP is not as commonly used as CAN or FlexRay, LIN is increasingly being used as a cheaper alternative to CAN for communication systems with lower demands.

The processor which was used for this project is the Freescale HCS12X. This processor has an on-board co-processor called XGATE which runs at twice the frequency of the CPU. The XGATE is a RISC (Reduced Instruction Set Computing) processor

which is optimised to process large amounts of smaller commands to take some of the processing load away from the main CPU, in particular the processing of interrupts [36][47]. The system outlined in Chapter Four could be designed to use the interrupt handling capabilities of the XGATE. The XGATE could be used for direct memory accesses while an interrupt which requires a higher level of processing will be passed to the CPU. This will help in reducing the overheads on the gateway CPU.

As has been previously discussed, FlexRay has a larger capacity than CAN and is capable of overwhelming the CAN network. A statistical scheduler may be devised to optimise message availability to the CAN network on transfer from the FlexRay network. The scheduler would ensure all messages get from the FlexRay bus to the CAN bus, but would also need to ensure that the other CAN nodes have access to the bus when needed.

Bibliography

[1] Siemens-Microelectronics. Canpres, October 1998.

[2] Roman Nossal and Roland Lang. Model-Based System Development: An Approach to Building X-by-Wire Applications. *IEEE Micro*, 22(4):56–63, 2002.

[3] Lucia Lo Belloz Thomas Noltey, Hans Hanssony. Automotive Communications - Past, Current and Future. *10th IEEE International Conference on Emerging Technologies*, 2005.

[4] http://www.flexray.com, 2009.

[5] William B. Ribbens and Norman P. Mansour. *Understanding Automotive Electronics*. Newnes, 2003.

[6] Gabriel Leen and Donal Heffernan. Expanding Automotive Electronic Systems. *Computer*, 35(1):88–93, 2002.

[7] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in Automotive Communication Systems. 93(6):1204–1223, June 2005.

[8] Olaf Pfeiffer. Betting on CAN and CANopen. *Embedded Systems Academy*, 2002.

[9] Traian Pop, Paul Pop, Petru Eles, and Zebo Peng. Bus access optimisation for FlexRay-based distributed embedded systems. pages 51–56, 2007.

[10] Jakob Axelsson, Joakim Frberg, Hans Hansson, Kristian S, and Bjrn Villing. Correlating bussines needs and network architectures in automotive applications a comparative case study. *IFAC*, 2003.

[11] G. Cena, A. Valenzano, and S. Vitturi. Advances in automotive digital communications. *Computer Standards & Interfaces*, 27(6):665 – 678, 2005.

[12] Eugene Blanchard. *Introduction to Data Communications*. 1999.

[13] CAN in Automation (CiA). http://www.can-cia.org/, 2008.

[14] Keith Pazul. Controller Area Network (CAN) Basics. *AN713*, 1999.

[15] Pat Richards Microchip Technology Inc. Understanding Microchips CAN Module Bit Timing. *AN754*, 2001.

[16] Thomas Nolte, Hans Hansson, Lucia Lo Bello, and Mlardalen. Implementing Next Generation Automotive Communications. 2004.

[17] Joel Shapiro. Flexray: The Next Generation In-Vehicle Network. *Evaluation Engineering*, March 2005.

[18] Sev Gunes-Lasnet and Olivier Notebaert. Prototype Implementation of a Routing Policy using FlexRay Frames Concept over a Spacewire Network. 2008.

[19] FlexRay Communications System Electrical Physical Layer Specification Version 2.1b.

[20] FlexRay Communications System Protocol Specification Version 2.1a.

[21] Hanser Automotive. XCP on FlexRay at BMW. 2006.

[22] Suk-Hyun Seo, Sang won Lee, Sung-Ho Hwang, and Jae Wook Jeon. Development of Network Gateway Between CAN and FlexRay Protocols For ECU Embedded Systems. *SICE-ICASE, 2006. International Joint Conference*, pages 2256–2261, Oct. 2006.

[23] C. Heller, J. Schalk, S. Schneele, and R. Reichel. Approaching the limits of flexray. *Network Computing and Applications, 2008. NCA '08. Seventh IEEE International Symposium on*, pages 205–210, July 2008.

[24] Peter Bohm. Introduction to FlexRay and TTA. 2005.

[25] Bill Rogers and Stefan Schmechtig. Flexray Message Buffers - The Host View of a FlexRay System. 2006.

[26] *A Reliable Gateway for In-vehicle Networks*, Seoul, Korea, 2008. The International Federation of Automatic Control (IFAC).

[27] Suk-Hyun Seo, Sang won Lee, Sung-Ho Hwang, and Jae Wook Jeon. A Gateway System for an Automotive System: LIN, CAN, and FlexRay. *IEEE International Conference on Industrial Informatics*, 2008.

[28] Jin-Ho Kim Sung-Ho Hwang Jae Wook Jeon Tae-Yoon Moon, Suk-Hyun Sec. Gateway System with Diagnostic Function for LIN, CAN and FlexRay. *International Conference on Control, Automation and Systems*, 2007.

[29] John P. Slone. *Local Area Network Handbook*. CRC Press, 1999.

[30] Alessandro Pinto, Luca P. Carloni, and Alberto L Sangiovanni-Vincentelli. A Communication Synthesis Infrastructure for Heterogeneous Networked Control Systems and Its Application to Building Automation and Control. *EMSOFT*, 2007.

[31] *Web-Based Distributed Embedded Gateway System Design*, Washington, DC, USA, 2006. IEEE Computer Society.

[32] G.V. Bochmann. Deriving protocol converters for communications gateways. *Communications, IEEE Transactions on*, 38(9):1298–1300, Sep 1990.

[33] Fujitsu Microelectronics Europe. *FLEXRAY-FPGA-EVA-KIT-369 User Guide*.

[34] Fujitsu Semiconductor. FR50 MB91360G Series Microcontroller Data Sheet. 2006.

[35] SofTec Microsystems. *SK-S12XDP512-A User Guide*.

[36] Freescale Semiconductor Inc. MC9S12XDP512 datasheet. Rev. 2.17, 2007.

[37] Infinion Technologies AG. TC1796 Data Sheet. 2006.

[38] Hitex Development Tools, http://www.hitex.com, 2006.

[39] H. Minorikawa and S.S. Sawa. Current Status and Future Trends of Electronic Packaging in Automotive Applications. *Transportation Electronics, 1990. Vehicle Electronics in the 90's: Proceedings of the International Congress on*, pages 155–163, 1990.

[40] Freescale Semiconductor Inc. MC33388 Datasheet. Rev 2.2, 2000.

[41] Philips Semiconductors. PCA82C250 Datasheet. 2000.

[42] Freescale Semiconductor Inc. MC33989 Datasheet. Rev 4.9.1, 2002.

[43] Freescale Semiconductor Inc. MFR4300 datasheet. Rev. 3, 2007.

[44] Freescale Semiconductor Inc. Flexray Unified Driver User Guide. Rev 1.2, 2006.

[45] Freescale Semiconductor Inc. FreeMASTER for Embedded Applications. Rev. 2.0, 2007.

[46] National Instruments. http://www.ni.com/labview, 2009.

[47] Ross Mitchell. Introducing the XGATE Module to Consumer and Industrial Application Developers. *AN3224*, 2006.

Appendix A

CAN Source Code

A.1 main.c

```
/************************************************************************
**
** File: main.c
**
************************************************************************/

#include <hidef.h>
#include "mc9s12xdp512.h"
#include "mscan.h"
#include "mscan.c"
#pragma LINK_INFO DERIVATIVE "mc9s12xdp512"

/************************************************************************
**
** Defines and variables
**
************************************************************************/

unsigned char potentiometer_value;
unsigned int can_delay = 1000;

/************************************************************************
**
** Peripheral Initialization
**
************************************************************************/

void PeriphInit(void)
{
```

```
    // Configures PORTA[6..0] as outputs
    PORTA = 0x7F;
    DDRA = 0x7F;

    // Configures PORTB[3..0] as outputs and PORTB[7..4] as inputs
    PORTB = 0x00;
    DDRB = 0x0F;

    // Enables pull-ups on PORTB
    PUCR |= 0x02;

    // Configures PORTC[4..0] as outputs
    PORTC = 0x00;
    DDRC = 0x1F;

    // Configures PORTD[4..0] as outputs
    PORTD = 0x00;
    DDRD = 0x1F;

    // Configures the A-D Converter
    // (16 conversions per sequence, 8 bit resolution, wrap around
    channel, continuous conversion)
    ATD1CTL3 = 0x38;
    ATD1CTL4 = 0x80;
    ATD1CTL0 = 0x05;
    ATD1CTL2 = 0x80;
    ATD1CTL5 = 0x32;

    // Configures the PIT (Periodic Interrupt Timer) on Channel 0
    // to generate a periodic interrupt of 500us

    PITCE = 0x01;
    PITINTE = 0x01;
    PITLD0 = 1000;
    PITCFLMT = 0xA0;

    MSCANInit(MSCAN_2);
    MSCANInit(MSCAN_4);

    EnableInterrupts;
}

/*********************************************************************
**
```

```
** main
**
************************************************************************/

void main(void)
{
    struct can_msg msg_send, msg_get;

    PeriphInit();

    while(1)
    {
        // Reads the ADC channels
        while(!(ATD1STAT0 & 0x80))
             ;

       potentiometer_value = ATD1DR0H;

        // Resets SCF flag
        ATD1STAT0 = 0x80;

        // If sufficient time has elapsed
        if(!can_delay)
        {
            msg_send.id = 4;
            msg_send.data[0] = (unsigned char)(potentiometer_value);
            msg_send.len = 1;
            msg_send.RTR = FALSE;
            msg_send.prty = 1;
            (void)MSCANSendMsg(MSCAN_2, msg_send);
            can_delay = 1000;
        }

        // Checks if a message is received from MSCAN4
        if(MSCANCheckRcvdMsg(MSCAN_4))
        {
            if(MSCANGetMsg(MSCAN_4, &msg_get))
            {
                if(msg_get.id == 4 && msg_get.RTR == FALSE)
                PORTB = msg_get.data[0];
            }
        }
    }
}
```

```
/*****************************************************************
**
** PIT0 Interrupt Service Routine
**
*****************************************************************/

#pragma CODE_SEG __NEAR_SEG NON_BANKED

// Decrement can_delay and reset interrupt flag
interrupt void PIT0_ISR(void)
{
    --can_delay;
    PITTF = 0x01;
}

#pragma CODE_SEG DEFAULT
```

A.2 mscan.c

```
/*****************************************************************
**
** File: mscan.c
**
*****************************************************************/

#include "mc9s12xdp512.h"
#include "mscan.h"

/*****************************************************************
**
** Defines and Variables
**
*****************************************************************/

unsigned char *can_periph[5] = {
    &CAN0CTL0,
    &CAN1CTL0,
    &CAN2CTL0,
    &CAN3CTL0,
    &CAN4CTL0
};

/*****************************************************************
```

```
**
** MSCANInit
**
*****************************************************************/

void MSCANInit(unsigned char can_num)
{
    unsigned char *can_pt;

    can_pt = can_periph[can_num];

    // If MSCAN peripheral is not in Initialization Mode,
    // enables the Inizialization Mode Request
    if(!(can_pt[CANCTL1]&CANCTL1_INITAK_MASK))
        {
        can_pt[CANCTL0] = CANCTL0_INITRQ_MASK;
        while(!(can_pt[CANCTL1]&CANCTL1_INITAK_MASK))
            ;
        }

    // Enables MSCAN peripheral and chooses Oscillator Clock,
    // Loop Disabled and Normal Operation
    can_pt[CANCTL1] = 0x80;

    // Sets SJW to 2Tq and BRP  to 4
    can_pt[CANBTR0] = 0x83;

    // Sets one sample per bit, TSEG1 = 4, TSEG2 = 3
    can_pt[CANBTR1] = 0x23;   //can_pt[CANBTR1] = 0x25;

    // Disables all the Filters
    can_pt[CANIDMR_1B+0] = 0xFF;
    can_pt[CANIDMR_1B+1] = 0xFF;
    can_pt[CANIDMR_1B+2] = 0xFF;
    can_pt[CANIDMR_1B+3] = 0xFF;
    can_pt[CANIDMR_2B+0] = 0xFF;
    can_pt[CANIDMR_2B+1] = 0xFF;
    can_pt[CANIDMR_2B+2] = 0xFF;
    can_pt[CANIDMR_2B+3] = 0xFF;

    // Restarts MSCAN peripheral, waits for Initialization Mode exit
    can_pt[CANCTL0] = 0x00;
    while(can_pt[CANCTL1]&CANCTL1_INITAK_MASK)
        ;
```

```
    // Waits for MSCAN synchronization with the CAN bus
    while(!(can_pt[CANCTL0]&CANCTL0_SYNCH_MASK))
        ;
}

/*****************************************************************
**
** MSCAN Send Message Routine
**
*****************************************************************/

Bool MSCANSendMsg(unsigned char can_num, struct can_msg msg)
{
    unsigned char n_tx_buf = 0, i;
    unsigned char *can_pt;

    can_pt = can_periph[can_num];
    if(msg.len > 8)
        return(FALSE);
    if(!(can_pt[CANCTL0]&CANCTL0_SYNCH_MASK))
        return(FALSE);
    while(!(can_pt[CANTFLG]&MaskOR(n_tx_buf)))
        n_tx_buf = (n_tx_buf == MAX_TX_BUFFERS)?
        0: (unsigned char)(n_tx_buf + 1);
    can_pt[CANTBSEL] = MaskOR(n_tx_buf);
    can_pt[CANTXIDR+0] = (unsigned char)(msg.id>>3);
    can_pt[CANTXIDR+1] = (unsigned char)(msg.id<<5);
    if(msg.RTR)
        can_pt[CANTXIDR+1] |= 0x10;
    for(i = 0; i < msg.len; i++)
        can_pt[CANTXDSR+i] = msg.data[i];
    can_pt[CANTXDLR] = msg.len;
    can_pt[CANTXTBPR] = msg.prty;
    can_pt[CANTFLG] = MaskOR(n_tx_buf);
    return(TRUE);
}

/*****************************************************************
**
** MSCAN Get Message Routine
**
*****************************************************************/
```

```
Bool MSCANGetMsg(unsigned char can_num, struct can_msg *msg)
{
    unsigned char i;
    unsigned char *can_pt;

    can_pt = can_periph[can_num];
    if(!(can_pt[CANRFLG]&CANRFLG_RXF_MASK))
        return(FALSE);
    if(can_pt[CANRXIDR+1]&0x08)
        return(FALSE);
    msg->id = ((can_pt[CANRXIDR+0]<<3)&0x0700) |
     (unsigned char)(can_pt[CANRXIDR+0]<<3) |
     (unsigned char)(can_pt[CANRXIDR+1]>>5);
    if(can_pt[CANRXIDR+1]&0x10)
        msg->RTR = TRUE;
    else
        msg->RTR = FALSE;
    msg->len = can_pt[CANRXDLR];
    for(i = 0; i < msg->len; i++)
        msg->data[i] = can_pt[CANRXDSR+i];
    can_pt[CANRFLG] = CANRFLG_RXF_MASK;
    return(TRUE);
}

/*********************************************************************
**
** MSCAN Check for Received Message Routine
**
*********************************************************************/

Bool MSCANCheckRcvdMsg(unsigned char can_num)
{
    unsigned char *can_pt;

    can_pt = can_periph[can_num];
    if(can_pt[CANRFLG]&CANRFLG_RXF_MASK)
        return(TRUE);
    return(FALSE);
}
```

Appendix B

FlexRay Source Code

B.1 FrUnifiedCfg.c

```
/*********************************************************************
**
** File: Fr_Unified_Cfg.c
** FlexRay High-Level Driver Implementation
** Modified version of Freescale's 'Fr_Unified_Cfg.c'
**
*********************************************************************/
#include "Fr_UNIFIED.h"
/*********************************************************************
**
** Global variables
**
*********************************************************************/
// Hardware configuration structure
// Number of MB in Segment 1:   11
// Number of MB in Segment 2:   8
// FIFO Depth:                  0 (not configured)
const Fr_HW_config_type Fr_HW_cfg_00 =
{
    0x140000,     // FlexRay module base address
    0x140800,     // FlexRay memory base address
    FR_MFR4300,     // Type of Freescale FlexRay module
    FALSE,          // Synchronization filtering
    FR_EXTERNAL_OSCILLATOR,
    0,              // Value of the PRESCALE or BITRATE bit field
    16,             // Data size - segment 1
    8,              // Data size - segment 2
    10,             // Last Message Buffer in segment 1
    18,             // Last individual MB
```

```
    19,             // Total number of used MB
    TRUE,           // Allow coldstart
    0,               // Offset of the Sync Frame Table
    FR_DUAL_CHANNEL_MODE    /* Single channel mode disabled */
};

// Transmit MB configuration structure
// Slot 4, payload length 16 Words, Single buffered MB,
// State transmission mode, interrupt disabled,
// channel AB, filtering disabled
const Fr_transmit_buffer_config_type Fr_tx_buffer_slot_04_cfg =
{
   4,                          // Transmit frame ID
   1747,                       // Header CRC
   16,                         // Payload length
   FR_SINGLE_TRANSMIT_BUFFER,  // Transmit MB buffering
   FR_STATE_TRANSMISSION_MODE, // Transmission mode
   FR_STREAMING_COMMIT_MODE,   // Transmission commit mode
   FR_CHANNEL_AB,              // Transmit channels
   FALSE,                      // Payload preamble
   FALSE,                      // Transmit cycle counter filter enable
   0,                          // Transmit cycle counter filter value
   0,                          // Transmit cycle counter filter mask
   FALSE,                      // Transmit MB interrupt enable
   FALSE                       // Used only for double buffer
};

// Receive MB configuration structure
// Slot 1, channel A, filtering disabled, interrupt enabled
const Fr_receive_buffer_config_type  Fr_rx_buffer_slot_01_cfg =
{
   1,                           // Receive frame ID
   FR_CHANNEL_A,                // Receive channel enable
   FALSE,                       // Receive cycle counter filter enable
   0,                           // Receive cycle counter filter value
   0,                           // Receive cycle counter filter mask
   FALSE                        // Receive MB interrupt enable
};

// Configuration data for Shadow Message Buffers
const Fr_receive_shadow_buffers_config_type Fr_rx_shadow_cfg =
{
    TRUE,       // Rx shadow buffer for channel A, seg 1 - enabled?
    TRUE,       // Rx shadow buffer for channel A, seg 2 - enabled?
```

```
    TRUE,       // Rx shadow buffer for channel B, seg 1 - enabled?
    TRUE,       // Rx shadow buffer for channel B, seg 2 - enabled?
    8,          // Ch A, seg 1 - current index of the MB header field
    17,         // Ch A, seg 2 - current index of the MB header field
    9,          // Ch B, seg 1 - current index of the MB header field
    18          // Ch B, seg 2 - current index of the MB header field
};

// Following array is used to determine which message buffers
// defined in Fr_buffer_cfg_xx structure will be used for the
// FlexRay CC configuration
const Fr_index_selector_type Fr_buffer_cfg_set_00[] =
{
    0, 1, 3, 5, 6, 7, FR_LAST_MB
};

// Array of structures with message buffer configuration information
// The MBs 6 and FIFO A will not be configured
const Fr_buffer_info_type Fr_buffer_cfg_00[] =
{ /* Buffer type         Configuration structure ptr MB index
xx = configuration index used by Fr_buffer_cfg_set_xx */
    {FR_TRANSMIT_BUFFER, &Fr_tx_buffer_slot_04_cfg,     1},    // 00
    {FR_RECEIVE_BUFFER,  &Fr_rx_buffer_slot_01_cfg,     2},    // 01
    {FR_RECEIVE_SHADOW,  &Fr_rx_shadow_cfg,             0},    // 05
};

/* Structure of this type contains configuration
   information of the one low level parameters set */
const Fr_low_level_config_type Fr_low_level_cfg_set_00 =
{
    10,         /* G_COLD_START_ATTEMPTS */
    3,          /* GD_ACTION_POINT_OFFSET */
    83,         /* GD_CAS_RX_LOW_MAX */
    0,          /* GD_DYNAMIC_SLOT_IDLE_PHASE */
    40,         /* GD_MINISLOT */
    3,          /* GD_MINI_SLOT_ACTION_POINT_OFFSET */
    50,         /* GD_STATIC_SLOT */
    13,         /* GD_SYMBOL_WINDOW */
    11,         /* GD_TSS_TRANSMITTER */
    59,         /* GD_WAKEUP_SYMBOL_RX_IDLE */
    50,         /* GD_WAKEUP_SYMBOL_RX_LOW */
    301,        /* GD_WAKEUP_SYMBOL_RX_WINDOW */
    180,        /* GD_WAKEUP_SYMBOL_TX_IDLE */
    60,         /* GD_WAKEUP_SYMBOL_TX_LOW */
```

```
    2,          /* G_LISTEN_NOISE */
    5000,       /* G_MACRO_PER_CYCLE */
    10,         /* G_MAX_WITHOUT_CLOCK_CORRECTION_PASSIVE */
    14,         /* G_MAX_WITHOUT_CLOCK_CORRECTION_FATAL */
    22,         /* G_NUMBER_OF_MINISLOTS */
    60,         /* G_NUMBER_OF_STATIC_SLOTS */
    4920,       /* G_OFFSET_CORRECTION_START */
    16,         /* G_PAYLOAD_LENGTH_STATIC */
    5,          /* G_SYNC_NODE_MAX */
    2,          /* G_NETWORK_MANAGEMENT_VECTOR_LENGTH */
    FALSE,      /* G_ALLOW_HALT_DUE_TO_CLOCK */
    20,         /* G_ALLOW_PASSIVE_TO_ACTIVE */
    FR_CHANNEL_AB,  /* P_CHANNELS */
    300,        /* PD_ACCEPTED_STARTUP_RANGE */
    1,          /* P_CLUSTER_DRIFT_DAMPING */
    56,         /* P_DECODING_CORRECTION */
    1,          /* P_DELAY_COMPENSATION_A */
    1,          /* P_DELAY_COMPENSATION_B */
    401202,     /* PD_LISTEN_TIMEOUT */
    601,        /* PD_MAX_DRIFT */
    0,          /* P_EXTERN_OFFSET_CORRECTION */
    0,          /* P_EXTERN_RATE_CORRECTION */
    4,          /* P_KEY_SLOT_ID */
    TRUE,       /* P_KEY_SLOT_USED_FOR_STARTUP */
    TRUE,       /* P_KEY_SLOT_USED_FOR_SYNC */
    1747,       /* P_KEY_SLOT_HEADER_CRC */
    21,         /* P_LATEST_TX */
    5,          /* P_MACRO_INITIAL_OFFSET_A */
    5,          /* P_MACRO_INITIAL_OFFSET_B */
    23,         /* P_MICRO_INITIAL_OFFSET_A */
    23,         /* P_MICRO_INITIAL_OFFSET_B */
    200000,     /* P_MICRO_PER_CYCLE */
    1201,       /* P_OFFSET_CORRECTION_OUT */
    600,        /* P_RATE_CORRECTION_OUT */
    FALSE,      /* P_SINGLE_SLOT_ENABLED */
    FR_CHANNEL_A,   /* P_WAKEUP_CHANNEL */
    16,         /* P_WAKEUP_PATTERN */
    40,         /* P_MICRO_PER_MACRO_NOM */
    8           /* P_PAYLOAD_LENGTH_DYN_MAX */
};
```

B.2 Node 1 - main.c

```
/*****************************************************************
```

```
**
** File: main.c
**
***************************************************************************/

#include <hidef.h>
#include <mc9s12xdp512.h>
#include "Fr_UNIFIED_types.h"
#include "Fr_UNIFIED.h"
#include "Fr_UNIFIED_cfg.h"
#pragma LINK_INFO DERIVATIVE "mc9s12xdp512"

/***************************************************************************
**
** Defines, variables and prototypes
**
***************************************************************************/

#define TX_SLOT_4               1
#define RX_SLOT_1               2

Fr_return_type return_value;
uint8 current_cycle;

uint16 tx_data_4[16] = {0};
Fr_tx_MB_status_type tx_return_value;
Fr_tx_status_type tx_status;

uint16 rx_data_1[16] = {0};
uint8 rx_data_length = 0;
uint16 rx_status_slot = 0;
Fr_rx_MB_status_type rx_return_value;
Fr_rx_status_type rx_status;

void InitMCU(void);
void StartupPLL(void);
void Failed(uint8 number);

/***************************************************************************
**
** Failed
**
***************************************************************************/
```

```
void Failed(uint8 number)      // Function for debugging
{
    // CC should be restarted
    while(1);
}

/*****************************************************************
**
** main
**
*****************************************************************/

void main(void)
{
    boolean cycle_starts = FALSE;

    // Disable interrupts on S12X
    DisableInterrupts;

    // Enable the PLL
    StartupPLL();

    // Initialize S12X MCU
    InitMCU();

    // Enable the FlexRay CC and force it into FR_POCSTATE_CONFIG
    return_value = Fr_init(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
    if(return_value == FR_NOT_SUCCESS) Failed(1);
    // Call debug function in case of any error

    // Initialization of FlexRay CC with protocol config parameter
    Fr_set_configuration(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);

    // Initialization of all message buffers, receive shadow buffers
    // and FIFO storages
    return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0],
    &Fr_buffer_cfg_set_00[0]);
    if(return_value == FR_NOT_SUCCESS) Failed(0xFF);

    // Leave FR_POCSTATE_CONFIG state
    return_value = Fr_leave_configuration_mode();
    if(return_value == FR_NOT_SUCCESS) Failed(2);

    // Retrieve the wakeup state
```

```
 wakeup_status = Fr_get_wakeup_state();

 // Check whether a wakeup pattern has been received
 if(wakeup_status == FR_WAKEUPSTATE_UNDEFINED)
 {   // No wakeup pattern has been received
     // Initiate wakeup procedure
     return_value = Fr_send_wakeup();
     if(return_value == FR_NOT_SUCCESS) Failed(3);
 }

 // Initialize startup
 return_value = Fr_start_communication();
 if(return_value == FR_NOT_SUCCESS) Failed(4);

 // The first initialization of Message Buffer 1
 tx_return_value = Fr_transmit_data(TX_SLOT_4, &tx_data_4[0], 16);
 if(tx_return_value == FR_TXMB_NO_ACCESS) Failed(4);

// Load current wakeup status
wakeup_status = Fr_get_wakeup_state();

 while(1)
 {
  // Check whether or not the communication cycle has been started
     cycle_starts = Fr_check_cycle_start(&current_cycle);
     if(cycle_starts)
     {
  // TRANSMIT SINGLE BUFFER - update transmit MB 1 with new data
  // Check whether data has been transmitted
         tx_status = Fr_check_tx_status(TX_SLOT_4);
         if(tx_status == FR_TRANSMITTED)
         {
          //increment value from received message
             tx_data_4[0] = rx_data_1[0] + 100;
             // Update transmit MB with new data
             tx_return_value = Fr_transmit_data(TX_SLOT_4,
             &tx_data_4[0], 16);
      }

  // RECEIVE BUFFER - copy received data from receive MB 2
  // Check whether the MB has been updated
         rx_status = Fr_check_rx_status(RX_SLOT_1);
         if(rx_status == FR_RECEIVED)
         {
```

```
                // Copy data into given data array
                rx_return_value = Fr_receive_data(RX_SLOT_1,
                &rx_data_1[0], &rx_data_length, &rx_status_slot);
            }
        }
    }
}

/************************************************************************
**
** StartupPLL
**
************************************************************************/

void StartupPLL(void)
{
    // CRG module configuration
    CLKSEL_PLLSEL = 0;          // System Clock = OSCCLK
    PLLCTL = 0xE1;              // CME, PLLON, AUTO, SCME
    REFDV = 3;                  // REFDV = 3
    SYNR = 24;                  // SYNR = 24
    while(!CRGFLG_LOCK);        // Wait for PLL VCO in desired range
    CLKSEL_PLLSEL = 1;          // System Clock = PLLCLK
}

/************************************************************************
**
** InitMCU
**
************************************************************************/

void InitMCU(void)
{
    // EBI (External Bus Interface) module configuration
    EBICTL0 = 0x2D;             // DATA[15:8], ADDR[12:1], UDS
    EBICTL1 = 0x02;             // EXSTR[2:0]=2, 3 stretch cycles

    // MMC (Memory Mapping Control) module configuration
    MODE    = 0xA0;             // Normal Expanded Mode
    MMCCTL1 = 0x01;             // ROMON
    MMCCTL0 = 0x04;             // CS[3:0]=4, CS2 enabled

    // COP module configuration
    COPCTL = 0x00;              // COP disable
```

```
    IRQCR = 0x00;             // Disable IRQ interrupt pin
}
```

B.3 Node 2 - main.c

```
/*********************************************************************
**
** File: main.c
**
*********************************************************************/

#include <hidef.h>
#include <mc9s12xdp512.h>
#include "Fr_UNIFIED_types.h"
#include "Fr_UNIFIED.h"
#include "Fr_UNIFIED_cfg.h"
#pragma LINK_INFO DERIVATIVE "mc9s12xdp512"
#include "freemaster.h"

/*********************************************************************
**
** Defines, variables and prototypes
**
*********************************************************************/

#define TX_SLOT_1                   0
#define TX_SLOT_1_TRANSMIT_SIDE     1
#define RX_SLOT_4                   3

Fr_return_type return_value;
Fr_POC_state_type protocol_state;
Fr_wakeup_state_type wakeup_status;
uint8 current_cycle;
uint16 current_macrotick;
uint16 tx_data_1[16] = {0};
Fr_tx_MB_status_type tx_return_value;
uint16 rx_data_4[16] = {0};
uint8 rx_data_length = 0;
uint16 rx_status_slot = 0;
Fr_rx_MB_status_type rx_return_value;

// FreeMASTER TSA support
```

```
    FMSTR_TSA_TABLE_BEGIN(first_table)
        FMSTR_TSA_RO_VAR(tx_data_1[0], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(rx_data_4[0], FMSTR_TSA_UINT16)
    FMSTR_TSA_TABLE_END()

    FMSTR_TSA_TABLE_LIST_BEGIN()
        FMSTR_TSA_TABLE(first_table)
    FMSTR_TSA_TABLE_LIST_END()

void InitMCU(void);
void StartupPLL(void);
void Failed(uint8 number);
void CC_interrupt_slot_1(uint8 buffer_idx);

/*****************************************************************
**
** Failed
**
*****************************************************************/

void Failed(uint8 number)          // Function for debugging
{
    // CC should be restarted
    while(1);
}

/*****************************************************************
**
** CC_interrupt_slot_1
**
*****************************************************************/

void CC_interrupt_slot_1(uint8 buffer_idx)
{
    // Update double transmit MB with new data (commit side)
    tx_return_value = Fr_transmit_data(TX_SLOT_1, &tx_data_1[0], 16);

    // Clear the flag at transmit side
    Fr_clear_MB_interrupt_flag(TX_SLOT_1_TRANSMIT_SIDE);

    if(tx_return_value == FR_TXMB_UPDATED)
    {
    // Increment received data by 100
        tx_data_1[0] = rx_data_4[0] + 100;
```

```
    }
}

/*****************************************************************
**
** CC_interrupt_slot_4
**
*****************************************************************/

void CC_interrupt_slot_4(uint8 buffer_idx)
{
    // Copy received data into given array
    rx_return_value = Fr_receive_data(buffer_idx, &rx_data_4[0],
     &rx_data_length, &rx_status_slot);
}

/*****************************************************************
**
** CC_interrupt_cycle_start
**
*****************************************************************/

void CC_interrupt_cycle_start(void)
//called when cycle start interrupt is generated
{
// Get the global time
    Fr_get_global_time(&current_cycle, &current_macrotick);
    // Store current cycle value
    tx_data_1[11] = current_cycle;
}

/*****************************************************************
**
** main
**
*****************************************************************/

void main(void)
{
    /* Disable interrupts on S12X */
    DisableInterrupts;

    /* Enable the PLL */
    StartupPLL();
```

```
/* Initialize S12X MCU */
InitMCU();

// Enable the FlexRay CC and force it into FR_POCSTATE_CONFIG
return_value = Fr_init(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
// Call debug function in case of any error
if(return_value == FR_NOT_SUCCESS) Failed(1);

// Init the FlexRay CC with protocol configuration parameter
Fr_set_configuration(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);

// Init all message buffers and receive shadow buffers
return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0],
 &Fr_buffer_cfg_set_00[0]);
if(return_value == FR_NOT_SUCCESS) Failed(0xFF);

// Set callback function in case interrupt from MB 0 occurs
// Interrupt is enabled for transmit side of the double MB
Fr_set_MB_callback(&CC_interrupt_slot_1, TX_SLOT_1_TRANSMIT_SIDE);

// Set callback function in case an interrupt from MB 3 occurs
Fr_set_MB_callback(&CC_interrupt_slot_4, RX_SLOT_4);

// Initialization of the timers
Fr_timers_init(&Fr_timers_cfg_00_ptr[0]);

// Set callback function in case cycle start interrupt occurs
Fr_set_protocol_0_IRQ_callback(&CC_interrupt_cycle_start,
 FR_CYCLE_START_IRQ);

EnableInterrupts;
// Enable IRQ interrupt pin on S12X
IRQCR = 0x40;

// Leave FR_POCSTATE_CONFIG state
return_value = Fr_leave_configuration_mode();
if(return_value == FR_NOT_SUCCESS) Failed(2);

// Retrieve the wakeup state
wakeup_status = Fr_get_wakeup_state();

// Check whether a wakeup pattern has been received
if(wakeup_status == FR_WAKEUPSTATE_UNDEFINED)
```

```
    {   // No wakeup pattern has been received
        // Initiate wakeup procedure
        return_value = Fr_send_wakeup();
        if(return_value == FR_NOT_SUCCESS) Failed(3);
    }

    // Initialize startup
    return_value = Fr_start_communication();
    if(return_value == FR_NOT_SUCCESS) Failed(4);

    // The first initialization of the MB 0
    tx_return_value = Fr_transmit_data(TX_SLOT_1, &tx_data_1[0],16);
    if(tx_return_value == FR_TXMB_NO_ACCESS) Failed(5);

    wakeup_status = Fr_get_wakeup_state();

    // Enable appropriate interrupts

    Fr_enable_interrupts((FR_MODULE_IRQ | FR_PROTOCOL_IRQ |
    FR_FIFO_A_IRQ | FR_RECEIVE_IRQ | FR_TRANSMIT_IRQ),
    (FR_TIMER_1_EXPIRED_IRQ | FR_TIMER_2_EXPIRED_IRQ |
    FR_CYCLE_START_IRQ), 0);

    // Start timers T1 and T2
    Fr_start_timer(FR_TIMER_T1);
    Fr_start_timer(FR_TIMER_T2);

    while(1)
    {
        // FreeMASTER routine
        FMSTR_Poll();
    }
}

/*********************************************************************
**
** StartupPLL
**
*********************************************************************/

void StartupPLL(void)
{
    // CRG module configuration
    CLKSEL_PLLSEL = 0;        // System Clock = OSCCLK
```

```
    PLLCTL = 0xE1;            // CME, PLLON, AUTO, SCME
    REFDV = 3;                // REFDV = 3
    SYNR = 24;                // SYNR = 24
    while(!CRGFLG_LOCK);      // Wait for PLL VCO is in desired range
    CLKSEL_PLLSEL = 1;        // System Clock = PLLCLK
}

/*********************************************************************
**
** InitMCU
**
*********************************************************************/

void InitMCU(void)
{
    // EBI  module configuration
    EBICTL0 = 0x2D;    // DATA[15:8], ADDR[12:1], UDS*  enabled
    EBICTL1 = 0x02;    // EXSTR[2:0]=2, 3 stretch cycles

    // MMC  module configuration
    MODE    = 0xA0;    // Normal Expanded Mode
    MMCCTL1 = 0x01;    // ROMON
    MMCCTL0 = 0x04;    // CS[3:0]=4, CS2 enabled

    // SCI (Serial Communication Interface) module configuration
    SCI0BDH = 0x00;   // IR disabled, transmitter narrow pulse 3/16
    SCI0BDL = 0x51;   // 25 MHz / (16*SCIBR)Baud Rate = 19200

    // FreeMASTER initialization
    FMSTR_Init();

    IRQCR = 0x00;             // Disable IRQ interrupt pin
}

/*********************************************************************
**
** FLEXRAY_ISR
**
*********************************************************************/

#pragma CODE_SEG NON_BANKED
interrupt 6 void FLEXRAY_ISR(void)
{
    // Call FlexRay driver interrupt service routine handle
```

```
    Fr_interrupt_handler();
}
#pragma CODE_SEG DEFAULT

/*****************************************************************
**
** FREEMASTER_ISR
**
*****************************************************************/

#pragma CODE_SEG NON_BANKED
interrupt 20 void FREEMASTER_ISR(void)
{
   // Call FreeMASTER service routine
   FMSTR_Isr();
}
#pragma CODE_SEG DEFAULT
```

Appendix C

Gateway Source Code

C.1 Node 1 - main.c

```
/*****************************************************************
**
** File: main.c
**
*****************************************************************/

#include <hidef.h>
#include <mc9s12xdp512.h>
#include "Fr_UNIFIED_types.h"
#include "Fr_UNIFIED.h"
#include "Fr_UNIFIED_cfg.h"
#pragma LINK_INFO DERIVATIVE "mc9s12xdp512"

/*****************************************************************
**
** Defines, variables and prototypes
**
*****************************************************************/

#define TX_SLOT_4                   1
#define RX_SLOT_1                   2
#define TX_SLOT_5                   4
#define TX_SLOT_5_TRANSMIT_SIDE     5

unsigned char potentiometer_value_1;
unsigned char potentiometer_value_2;

Fr_return_type return_value;
Fr_POC_state_type protocol_state;
```

```
Fr_wakeup_state_type wakeup_status;
uint8 current_cycle;

uint16 tx_data_4[16] = {0};
uint16 tx_data_5[16] = {0};
Fr_tx_MB_status_type tx_return_value;

Fr_tx_status_type tx_status;
uint16 rx_data_1[16] = {0};
uint8 rx_data_length = 0;
uint16 rx_status_slot = 0;
Fr_rx_MB_status_type rx_return_value;
Fr_rx_status_type rx_status;

uint16 mb_access_error = 0;
uint16 chi_error = 0;
boolean protocol_error = FALSE;

void InitMCU(void);
void StartupPLL(void);
void Failed(uint8 number);

/*****************************************************************
**
** Failed
**
*****************************************************************/

void Failed(uint8 number)
{
    while(1);
}

/*****************************************************************
**
** main
**
*****************************************************************/

void main(void)
{
    DisableInterrupts;

    StartupPLL();
```

```
InitMCU();

// Enable the FlexRay CC and force it into FR_POCSTATE_CONFIG
return_value = Fr_init(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
if(return_value == FR_NOT_SUCCESS) Failed(1);

Fr_set_configuration(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);

return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0],
&Fr_buffer_cfg_set_00[0]);

if(return_value == FR_NOT_SUCCESS) Failed(0xFF);

// Leave FR_POCSTATE_CONFIG state
return_value = Fr_leave_configuration_mode();
if(return_value == FR_NOT_SUCCESS) Failed(2);

// Retrieve the wakeup state
wakeup_status = Fr_get_wakeup_state();

// Check whether a wakeup pattern has been received
if(wakeup_status == FR_WAKEUPSTATE_UNDEFINED)
{   // No wakeup pattern has been received
    // Initiate wakeup procedure
    return_value = Fr_send_wakeup();
    if(return_value == FR_NOT_SUCCESS) Failed(3);
}

// Initialize startup
return_value = Fr_start_communication();
if(return_value == FR_NOT_SUCCESS) Failed(4);

// The first initialization of the MB 1
tx_return_value = Fr_transmit_data(TX_SLOT_4,
&tx_data_4[0], 16);
if(tx_return_value == FR_TXMB_NO_ACCESS) Failed(4);

 wakeup_status = Fr_get_wakeup_state();

while(1)
{
    // Check if communication cycle has been started
    cycle_starts = Fr_check_cycle_start(&current_cycle);
```

```
        if(cycle_starts)
        {
            // Check whether data has been transmitted
            tx_status = Fr_check_tx_status(TX_SLOT_4);
            if(tx_status == FR_TRANSMITTED)
            {
                // Load data to variables
                tx_data_4[0] = potentiometer_value_1;
                tx_data_4[1] = potentiometer_value_2;
                tx_data_4[2] = chi_error;
                tx_data_4[3] = mb_access_error;
                // Update transmit MB with new data
                tx_return_value = Fr_transmit_data(TX_SLOT_4,
                &tx_data_4[0], 16);

if(tx_return_value == FR_TXMB_NO_ACCESS) mb_access_error++;
             }

            // Check whether data has been transferred or transmitted
            tx_status = Fr_check_tx_status(TX_SLOT_5);
            // Update commit side of double MB in case that Internal
            // Message Transfer has been performed
            if((tx_status == FR_TRANSMITTED) ||
            (tx_status ==  FR_INTERNAL_MESSAGE_TRANSFER_DONE))
            {
             tx_return_value = Fr_transmit_data(TX_SLOT_5,
             &tx_data_5[0], 16);
             // Increment variable in case of MB access error
        if(tx_return_value == FR_TXMB_NO_ACCESS) mb_access_error++;
            }

            // Check has MB has been updated in last matching slot
            rx_status = Fr_check_rx_status(RX_SLOT_1);
            if(rx_status == FR_RECEIVED)
            {
                // Copy data into given data array
                rx_return_value = Fr_receive_data(RX_SLOT_1,
                &rx_data_1[0], &rx_data_length, &rx_status_slot);
                tx_data_5[2] = rx_status_slot;
         if(rx_return_value == FR_RXMB_NO_ACCESS) mb_access_error++;
            }

            tx_data_5[4] = current_cycle;
```

```
            // Check whether a CHI related error has occurred
            // Increment variable if CHI error occurred
            if(Fr_check_CHI_error() != 0) chi_error++;
            tx_data_5[6] = chi_error;
        }

        // Check whether or not the protocol engine has
        // detected an internal protocol error

        protocol_error = Fr_check_internal_protocol_error();

        tx_data_5[5] = mb_access_error;
       }
}

/*********************************************************************
**
** StartupPLL
**
*********************************************************************/

void StartupPLL(void)
{
    /* CRG module configuration */
    CLKSEL_PLLSEL = 0;       // System Clock = OSCCLK
    PLLCTL = 0xE1;           // CME, PLLON, AUTO, SCME
    REFDV = 3;               // REFDV = 3
    SYNR = 24;               // SYNR = 24
    while(!CRGFLG_LOCK);     // Wait for PLL VCO is in desired range
    CLKSEL_PLLSEL = 1;       // System Clock = PLLCLK
}

/*********************************************************************
**
** InitMCU
**
*********************************************************************/

void InitMCU(void)
{
    // EBI module configuration
    EBICTL0 = 0x2D;     // DATA[15:8], ADDR[12:1], UDS
    EBICTL1 = 0x02;     // EXSTR[2:0]=2, 3 stretch cycles
```

```
    // MMC module configuration
    MODE    = 0xA0;     // Normal Expanded Mode
    MMCCTL1 = 0x01;     // ROMON
    MMCCTL0 = 0x04;     // CS[3:0]=4, CS2 enabled

    IRQCR = 0x00;             // Disable IRQ interrupt pin
}
```

C.2 Node 2 - main.c

```
/*****************************************************************
**
** File: main.c
**
*****************************************************************/

#include <hidef.h>
#include <mc9s12xdp512.h>
#include "mscan.h"
#include "Fr_UNIFIED_types.h"
#include "Fr_UNIFIED.h"
#include "Fr_UNIFIED_cfg.h"
#include "freemaster.h"
#pragma LINK_INFO DERIVATIVE "mc9s12xdp512"

/*****************************************************************
**
** Defines, variables and prototypes
**
*****************************************************************/

#define TX_SLOT_1                   0
#define TX_SLOT_1_TRANSMIT_SIDE     1
#define RX_SLOT_4                   3

unsigned char rxdata_value0;
unsigned char rxdata_value1;
unsigned char cantx0;
unsigned char cantx1;
unsigned char data_updated;
unsigned char chi_n2;
unsigned char mb_acc_n2;
unsigned char flexpoc_state;
```

```
Fr_return_type return_value;
Fr_POC_state_type protocol_state;
Fr_wakeup_state_type wakeup_status;
uint8 current_cycle;
uint16 current_macrotick;

uint16 tx_data_1[16] = {0};
Fr_tx_MB_status_type tx_return_value;

uint16 rx_data_4[16] = {0};
uint8 rx_data_length = 0;
uint16 rx_status_slot = 0;
Fr_rx_MB_status_type rx_return_value;

// FreeMASTER TSA support
    FMSTR_TSA_TABLE_BEGIN(first_table)
        FMSTR_TSA_RO_VAR(tx_data_1[0], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(tx_data_1[1], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(tx_data_1[5], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(tx_data_1[6], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(tx_data_1[7], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(tx_data_1[8], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(tx_data_1[9], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(rx_data_4[0], FMSTR_TSA_UINT16)
        FMSTR_TSA_RO_VAR(protocol_state,
        FMSTR_TSA_USERTYPE(Fr_POC_state_type))
        FMSTR_TSA_RO_VAR(wakeup_status,
        FMSTR_TSA_USERTYPE(Fr_wakeup_state_type))
    FMSTR_TSA_TABLE_END()

    FMSTR_TSA_TABLE_LIST_BEGIN()
        FMSTR_TSA_TABLE(first_table)
    FMSTR_TSA_TABLE_LIST_END()

void InitMCU(void);
void StartupPLL(void);
void Failed(uint8 number);
void CC_interrupt_slot_1(uint8 buffer_idx);
void CC_interrupt_slot_4(uint8 buffer_idx);
void CC_interrupt_timer_1(void);
void CC_interrupt_timer_2(void);
void CC_interrupt_cycle_start(void);
void CC_interrupt_FIFO_A(uint16 header_idx);
```

```
/*************************************************************************
**
** Failed
**
*************************************************************************/

void Failed(uint8 number)
{
    while(1);
}

/*************************************************************************
**
** CC_interrupt_slot_1
**
*************************************************************************/

void CC_interrupt_slot_1(uint8 buffer_idx)
{
    // Update double transmit MB with new data (commit side)
    tx_return_value = Fr_transmit_data(TX_SLOT_1, &tx_data_1[0], 16);

    // Clear the flag at transmit side
    Fr_clear_MB_interrupt_flag(TX_SLOT_1_TRANSMIT_SIDE);

    if(tx_return_value == FR_TXMB_UPDATED)
    {
        //transmit the received value from CAN_4
        tx_data_1[0] = rxdata_value0;
        tx_data_1[2] = rxdata_value1;
    }
}

/*************************************************************************
**
** CC_interrupt_slot_4
**
*************************************************************************/

void CC_interrupt_slot_4(uint8 buffer_idx)
{
    // Copy received data into given array
    rx_return_value = Fr_receive_data(buffer_idx, &rx_data_4[0],
    &rx_data_length, &rx_status_slot);
```

```
    tx_data_1[1] = rx_status_slot;
    cantx0 = rx_data_4[0];
    cantx1 = rx_data_4[1];
    chi_n2 = rx_data_4[2];
    mb_acc_n2 = rx_data_4[3];
    //set data_updated flag
    data_updated = 1;
  }

/*********************************************************************
**
** CC_interrupt_timer_1
**
*********************************************************************/

/void CC_interrupt_timer_1(void)
{
    Fr_get_global_time(&current_cycle, &current_macrotick);
    tx_data_1[14] = current_macrotick;
    tx_data_1[15] = (uint16)(current_cycle);
}

/*********************************************************************
**
** CC_interrupt_timer_2
**
*********************************************************************/

void CC_interrupt_timer_2(void)
{
    Fr_get_global_time(&current_cycle, &current_macrotick);
    tx_data_1[12] = current_macrotick;
    tx_data_1[13] = (uint16)(current_cycle);
}

/*********************************************************************
**
** CC_interrupt_cycle_start
**
*********************************************************************/

void CC_interrupt_cycle_start(void)
{
    Fr_get_global_time(&current_cycle, &current_macrotick);
```

```
    tx_data_1[11] = current_cycle;
}

/*********************************************************************
**
** main
**
*********************************************************************/

void main(void)
{

    struct can_msg msg_send, msg_get;

    DisableInterrupts;

    StartupPLL();

    InitMCU();

    // Enable the FlexRay CC and force it into FR_POCSTATE_CONFIG
    return_value = Fr_init(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);
    if(return_value == FR_NOT_SUCCESS) Failed(1);

    Fr_set_configuration(&Fr_HW_cfg_00, &Fr_low_level_cfg_set_00);

    // Initialise all message buffers, receive shadow buffers
    return_value = Fr_buffers_init(&Fr_buffer_cfg_00[0],
    &Fr_buffer_cfg_set_00[0]);

    if(return_value == FR_NOT_SUCCESS) Failed(0xFF);

     // Enable interrupt for transmit side of the double MB
    Fr_set_MB_callback(&CC_interrupt_slot_1,TX_SLOT_1_TRANSMIT_SIDE);

    // Set callback function in case an interrupt from MB 3 occurs
    Fr_set_MB_callback(&CC_interrupt_slot_4, RX_SLOT_4);

    // Initialise the timers
    Fr_timers_init(&Fr_timers_cfg_00_ptr[0]);

    // Set callback function in case an interrupt from timer 1 occurs
    Fr_set_protocol_0_IRQ_callback(&CC_interrupt_timer_1,
    FR_TIMER_1_EXPIRED_IRQ);
```

```
// Set callback function in case an interrupt from timer 2 occurs
Fr_set_protocol_0_IRQ_callback(&CC_interrupt_timer_2,
FR_TIMER_2_EXPIRED_IRQ);

// Set callback function in case a cycle start interrupt occurs
Fr_set_protocol_0_IRQ_callback(&CC_interrupt_cycle_start,
FR_CYCLE_START_IRQ);

EnableInterrupts;
IRQCR = 0x40;        // Enable IRQ interrupt pin on S12X

// Leave FR_POCSTATE_CONFIG state
return_value = Fr_leave_configuration_mode();
if(return_value == FR_NOT_SUCCESS) Failed(2);

// Retrieve the wakeup state
wakeup_status = Fr_get_wakeup_state();

// Check if a wakeup pattern has been received
if(wakeup_status == FR_WAKEUPSTATE_UNDEFINED)
{   // No wakeup pattern has been received
    // Initiate wakeup procedure
    return_value = Fr_send_wakeup();
    if(return_value == FR_NOT_SUCCESS) Failed(3);
}

// Load the current POC state
protocol_state = Fr_get_POC_state();

// Wait until the FR CC is not in the FR_POCSTATE_READY
while(Fr_get_POC_state() != FR_POCSTATE_READY)
{
    protocol_state = Fr_get_POC_state();
    FMSTR_Poll();
}

// Initialize startup
return_value = Fr_start_communication();
if(return_value == FR_NOT_SUCCESS) Failed(4);

protocol_state = Fr_get_POC_state();
while(Fr_get_POC_state() != FR_POCSTATE_NORMAL_ACTIVE)
{
```

```
    protocol_state = Fr_get_POC_state();
    FMSTR_Poll();
}

protocol_state = Fr_get_POC_state();

tx_return_value = Fr_transmit_data(TX_SLOT_1,
&tx_data_1[0], 16);
if(tx_return_value == FR_TXMB_NO_ACCESS) Failed(5);

wakeup_status = Fr_get_wakeup_state();
// Enable appropriate interrupts
Fr_enable_interrupts((FR_MODULE_IRQ | FR_PROTOCOL_IRQ |
FR_FIFO_A_IRQ | FR_RECEIVE_IRQ | FR_TRANSMIT_IRQ),
(FR_TIMER_1_EXPIRED_IRQ | FR_TIMER_2_EXPIRED_IRQ |
FR_CYCLE_START_IRQ), 0);

Fr_start_timer(FR_TIMER_T1);
Fr_start_timer(FR_TIMER_T2);

while(1)
{

 FMSTR_Poll();

    // Check if a message is received from MSCAN4
    if(MSCANCheckRcvdMsg(MSCAN_4))
    {
        if(MSCANGetMsg(MSCAN_4, &msg_get))
        {
            if(msg_get.id == CAN_MSG_ID_RX &&
             msg_get.RTR == FALSE)
            {
             //load received data to variables
              rxdata_value0 = msg_get.data[0];
              rxdata_value1 = msg_get.data[1];
            }
         }
     }

    // Check has data been updated
    if( data_updated == 1 )
    {
        // Transmit data on CAN Node 2
```

```
    msg_send.id = 2;
    msg_send.data[0] = cantx0;
    msg_send.data[1] = cantx1;
    msg_send.data[2] = chi_n2;
    msg_send.data[3] = mb_acc_n2;
    msg_send.len = 4;
    msg_send.RTR = FALSE;
    msg_send.prty = 1;
    (void)MSCANSendMsg(MSCAN_2, msg_send);
    // reset data_updated flag
    data_updated = 0;
} ;

// Check has protocol state changed
if(Fr_check_protocol_state_changed())
{
flexpoc_state = 9;
  protocol_state = Fr_get_POC_state();

  if (protocol_state == FR_POCSTATE_CONFIG)
        flexpoc_state = 0;
  else if (protocol_state == FR_POCSTATE_DEFAULT_CONFIG)
        flexpoc_state = 1;
  else if (protocol_state == FR_POCSTATE_HALT)
        flexpoc_state = 2;
  else if (protocol_state == FR_POCSTATE_NORMAL_ACTIVE)
        flexpoc_state = 3;
  else if (protocol_state == FR_POCSTATE_NORMAL_PASSIVE)
        flexpoc_state = 4;
  else if (protocol_state == FR_POCSTATE_READY)
        flexpoc_state = 5;
  else if (protocol_state == FR_POCSTATE_STARTUP)
        flexpoc_state = 6;
  else if (protocol_state == FR_POCSTATE_WAKEUP)
        flexpoc_state = 7;
  else
        flexpoc_state = 8;
    // Transmit protocol state on CAN Node 2
    msg_send.id = 17;
    msg_send.data[0] = flexpoc_state;
    msg_send.len = 1;
    msg_send.RTR = FALSE;
    msg_send.prty = 1;
    (void)MSCANSendMsg(MSCAN_2, msg_send);
```

```
        } ;
    }
}

/*************************************************************************
**
** StartupPLL
**
*************************************************************************/

void StartupPLL(void)
{
    // CRG module configuration
    CLKSEL_PLLSEL = 0;       // System Clock = OSCCLK
    PLLCTL = 0xE1;           // CME, PLLON, AUTO, SCME
    REFDV = 3;               // REFDV = 3
    SYNR = 24;               // SYNR = 24
    while(!CRGFLG_LOCK);     // Wait for PLL VCO is in desired range
    CLKSEL_PLLSEL = 1;       // System Clock = PLLCLK
}

/*************************************************************************
**
** InitMCU
**
*************************************************************************/

void InitMCU(void)
{
    // EBI module configuration
    EBICTL0 = 0x2D;    // DATA[15:8], ADDR[12:1], UDS
    EBICTL1 = 0x02;    // EXSTR[2:0]=2, 3 stretch cycles

    // MMC module configuration
    MODE    = 0xA0;    // Normal Expanded Mode
    MMCCTL1 = 0x01;    // ROMON
    MMCCTL0 = 0x04;    // CS[3:0]=4, CS2 enabled

    COP module configuration
    COPCTL = 0x00;     // COP disable

    SCI0BDH = 0x00;    // IR disabled, transmitter narrow pulse 3/16
    SCI0BDL = 0x51;    // 25 MHz / (16*SCIBR)Baud Rate = 19200
```

```
    FMSTR_Init();      // Initialise Freemaster

    IRQCR = 0x00;      // Disable IRQ interrupt pin

    // Configure PB[3..0] as output and PB[7..4] as input
    PORTB = 0x00;
    DDRB = 0x0F;

    // Enables pull-ups on PB port
    PUCR |= 0x02;

    // Configures PD[4..0] port as output
    PORTD = 0x00;
    DDRD = 0x1F;

    // Configures the ATD peripheral
    // (16 conversions per sequence, 8 bit resolution, wrap around
    // channel, continuous conversion)
    ATD1CTL3 = 0x38;
    ATD1CTL4 = 0x80;
    ATD1CTL0 = 0x05;
    ATD1CTL2 = 0x80;
    ATD1CTL5 = 0x32;

    MSCANInit(MSCAN_2);
    MSCANInit(MSCAN_4);
}

/*********************************************************************
**
** FLEXRAY_ISR
**
*********************************************************************/

#pragma CODE_SEG NON_BANKED
interrupt 6 void FLEXRAY_ISR(void)
{
    // Call FlexRay driver interrupt service routine handle
    Fr_interrupt_handler();
}
#pragma CODE_SEG DEFAULT

/*********************************************************************
**
```

```
** FREEMASTER_ISR
**
*****************************************************************/

#pragma CODE_SEG NON_BANKED
interrupt 20 void FREEMASTER_ISR(void)
{
    // Call FreeMASTER service routine
    FMSTR_Isr();
}
#pragma CODE_SEG DEFAULT
```