Waterford Institute of Technology

Ireland

Institiúid Teicneolaíochta Phort Láirge

Éire

A CAN to FlexRay Migration Framework

Richard Murphy B.Sc. (Hons)

M.Sc. Thesis

(Supervisor) Frank Walsh B.A., B.A.I., M.Sc.

Submitted to Waterford Institute of Technology Awards Council, September 2009

Acknowledgments

I would like to thank sincerely the following people, for without their input this thesis would not have been possible.

I would like to thank my supervisor Frank Walsh for his ideas and input especially when things were not going to plan he was always willing to give up his time to discuss any issues. I would also like to thank the group leader, Brendan Jackman for his assistance and input any time it was asked of him. A big thank you must go to Gareth, Rob and Wei Da who helped me any time I asked even while they were busy conducting their own research.

I would like to thank Sumitomo Ystrad. The financial backing of Sumitomo has made this thesis possible. I would like to especially thank Paul, Jim and Eamonn whose hospitality and assistance was second to none during my visits there.

Finally I would like to thank my parents Milo and Kitty who backed me both financially and with encouragement. Without them I would not be where I am now.

Declaration

I Richard Murphy, declare that this thesis is submitted by me in partial fulfilment of the requirement for the degree M.Sc., is entirely my own work except where otherwise accredited. It has not at any other time whole or in part been submitted for any other educational award.

Signature:______________________________

Richard Murphy

9th of October 2009

Abstract

One of the first electronics components used in an automobile was the fuse. Additional features were developed such as engine management systems. These additional features increased the amount of wiring in a wiring harness. This contributed towards the necessity to develop the bus structure in the 1980s. The de-facto bus structure in the automotive industry became CAN (Controller Area Network). By using a bus structure this resulted in less hard wiring being required in the production of an automobile which further lead to a reduction in production cost. CAN is an event-triggered protocol which denotes it is non-deterministic and it has a theoretical bandwidth limit of 1Mbit\s. The practical limit is nearer 500kbit\s. During the 80's and 90's, automotive electronics development increased; this was primarily driven by increased development of safety features such as ABS (Anti-lock Braking System). The increase in the number of features and nodes caused increased traffic on the bus. The CAN protocol will be unable to meet the requirements for the extra applications.

This resulted in the development of FlexRay in 2000 by a consortium originally consisting of amongst others BMW, Daimler Chrysler, Freescale and Philips. This protocol can implement both time-triggered and event-triggered messages, determinism, fault tolerance, redundancy and can operate at 10Mbit\s. Newly developed technologies have high initial costs therefore being initially more expensive than established technologies. This could result in it being financially unworkable to replace a complete automotive bus with FlexRay. FlexRay can possibly be used for mission critical applications such as powertrain applications, while other protocols such as CAN and LIN (Local Interconnect Network) may possibly be used for less critical applications.

The aim of this research is to design and develop a framework that allows the implementation of a CAN application on the FlexRay protocol, without degrading the applications performance.

Table of Contents

Table of Contents

Table of Figures

Table of Figures

Table of Figures

Table of Figures

Table of Figures

Table of Tables

Table of Tables

Table of Equations

Table of Equations

Section I - Thesis Overview

1 Thesis Overview:

1.1 Problem Specification

The aim of this research is to develop a framework to migrate an application from CAN, an event driven network protocol, used principally in automotive applications to FlexRay, a time-triggered protocol. Within the automotive industry the predominant network has been CAN (Controller Area Network) and this is discussed in greater detail in Chapter 3. In the past eight to ten years it has been realised by automobile manufacturers that the CAN protocol will not be sufficient for future application requirements (Thomas Noltey, 2005). This is the main reason behind FlexRay's development. The FlexRay protocol is not intended necessarily to replace CAN. Through being able to operate a single protocol for an application this can reduce the complexity associated with operating multiple protocols via gateway/s. Consideration of various aspects of each protocol are to be take into account such as cost, knowledge of all protocols and the time required to implement these protocols are some examples of some of the possible issues encountered.

As consumers expect higher luxury levels in automobiles, this increases the loads on the current network protocols (Schedl, 2007). This, combined with the ever increasing amount of safety applications being developed, has resulted in predictions by Dr Anton Schedl at the Vector FlexRay Symposium 2007 amongst others, that current network protocols will not be able to cope with this demand in the near future (Schedl, 2007). In 2000 the FlexRay consortium was founded by automobile manufacturers BMW, Daimler Chrysler and semiconductor manufacturers Motorola semiconductors products sector (now Freescale semiconductors) and Philips semiconductors. Other leading companies in the automotive industry soon joined. It is anticipated that

FlexRay will replace CAN (as the de-facto automobile network) for critical and safety applications.

While FlexRay is anticipated as the solution to safety critical high speed applications, it is also anticipated that CAN technology will continue to be used in less critical applications. A migration from the CAN network to the FlexRay network would be required where CAN has reached maximum capacity or new features are added to an application that requires some of the features provided for on the FlexRay network. This research aims to provide a framework that will allow applications that have previously operated on the CAN network, to successfully operate on the FlexRay network.

1.2 Specified Solution

One solution is to develop a framework to migrate from the CAN network to the FlexRay network. To achieve a successful migration a minimum requirement is that the minimum timing parameters in the CAN network are at least achieved or exceeded in the FlexRay network. This requires a detailed understanding of both network technologies, but specifically the FlexRay network as it is more complex.

1.3 Research Questions

Question 1:

What are the benefits of using the migration framework versus the use of a gateway?

Question 2:

What migration techniques used in other or similar protocols are applicable in this research?

Question 3:

What parameters are required in relation to the application and network protocol, for migration to be undertaken?

1.4 Document Layout

The thesis is arranged as follows;

- **Section I** : Thesis Overview
  This section, Thesis Overview presents the problem specification and the research questions.

- **Section II** : Literature Review
  The Literature Review section discusses the background of automotive networks before presenting the CAN and FlexRay protocols in depth. An embedded system overview is discussed before the section concludes with a review of migration procedures carried out by other authors.

- **Section III** : Framework Development
  This section, Framework Development, presents the requirements and methodology for undertaking this migration procedure.

- **Section IV** : Testing & Results
  Section IV, Testing & Results, presents the system model and the actual migration framework. The section proceeds with the abstract and experimental implementations of the generic model and the ACC (Adaptive Cruise Control) reference model and the Verification of Time-Triggered properties. The section concludes with a presentation of findings and a discussion of results.

- **Section V** : Conclusion
  Section V contains the conclusion, and any potential future work that would improve this research.

- **Section VI** : Appendices
  Section VI concludes the thesis with the appendices.

1.4.1 References

THOMAS NOLTEY, H. H., LUCIA LO BELLO (2005) Automotive Communications - Past, Current and Future

Schedl D A,2007, Goals and Architecture for FlexRay at BMW, 1st Vector FlexRay Symposium, Stuttgart, Germany.

Section II - Literature Review

2 Automotive Networks Review:

2.1 Introduction

This chapter contains a general assessment of what a computer network is, and reviews common network topologies. The TCP/IP and OSI reference model layers are presented due to their use in the transportation of data packets. This leads onto a history of automotive networks discussing various classes of automotive protocols. The Electronic Control Unit (ECU) is presented due to its core use in the development and implementation of networks in the automotive industry. Finally automotive protocols are discussed at event-triggered and time-triggered level and a direct comparison is made between the natures of both protocols.

2.2 Computer Networks

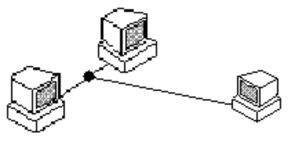
"A network is a series of points or nodes interconnected by communication paths" (Harbeck, 2006)

The Internet is the most common data network that people come into contact with on a daily basis. The Internet evolved from a small academic research project involving a few dozen sites to become a vast worldwide system of interconnected networks providing various functions and services. The first computer networks were timesharing networks that used mainframes and attached terminals (Teare, 1999). Timesharing is the concept of multiple users getting access to the processor during the processors 'idle' time.

By networking remote locations to these 'powerful' computers (time-sharing) the required tasks could be completed more economically. Currently an entire industry provides networking technologies and services.

2.3 Network Function

A network consists of two or more devices interconnected as illustrated in Figure 2-1. One of the earliest network types was built to allow several computers to share a single printer.



A network's function is the transportation of data (e.g. data from a computer to a printer). Networking can be complex because there are so many different technologies available that can be used to connect two or more networks together (Comer, 2001).
As data networks were developing during the 1970s and the 1980s the automotive industry was also starting to realise the advantages of implementing networks in automobiles. Originally, automobiles had relatively small quantities of electric and electronic components, usually in the form of closed loop circuits. For example, one of the earliest applications was the control and operation of lights. Additional features such as windscreen wipers and engine management systems increased the complexity and volume of wiring in the wiring harness. This led to the development of the bus structure in the 1980s. The de-facto bus structure in the automotive industry became the Controller Area Network (CAN). The structures of conventional data networks and automotive networks have some similarities. In the following section the features of both network types are discussed at network level in relation to the following; topology, architecture and protocols.

2.4 Network Topologies

The term ‘Topology’ refers to the layout of the interconnecting devices on a network. There are two ‘types’ of topologies; the physical topology which describes the cable, node, and connector arrangements, and there is the logical topology which describes the arrangement of devices on a network (Steinke, 2000). The following description deals with physical topologies.

The main physical topologies are Bus, Star and Ring. The names are given in relation to the general shape of each network. The type of topology used depends on the configuration requirements of the system.

2.4.1 Bus Topology

The Bus Topology (shown in Figure 2-2) can consist of a common medium, such as co-axial cable (10 base-2 (thin net) or 10 base-5 (thick net)) or un/shielded twisted pair (Institute, 2002).

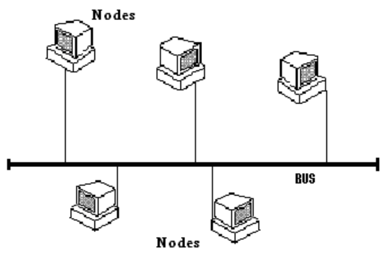
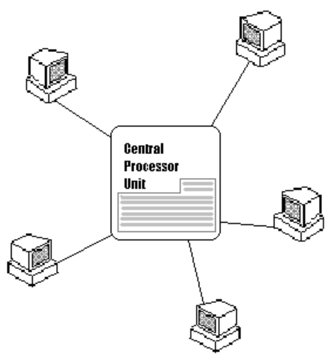


Figure 2-2: Bus topology

All the other nodes on the network can be directly connected off this. Because all nodes share a common bus they can also share communication. The beginning and end of the bus are terminated to prevent signal reflecting back down the cable. If the central bus fails (e.g. the cable is cut) nodes at either side of the break can cease to function correctly, if they are required to communicate with each other. Using the CAN protocol information travels along the central bus and whichever node requests the information matches the message ID and then it can extract the information from the bus. If the message ID does not match the node ID the node does not gain access to that message. If a single node is removed from the network it does not affect the rest of the system.

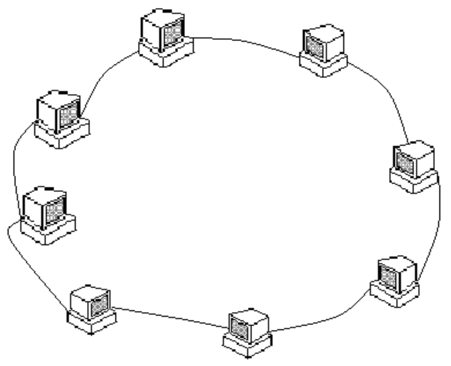
2.4.2 Star Topology

The Star Topology is based on the principle of a centralised host through which all other nodes communicate (Figure 2-3). If one node wants to communicate with another node it is required to send the data through the central processor and then distribute it to the desired node. If the central processor node fails, subsequently communication is halted.



2.4.3 Ring Topology

The Ring Topology or Ring Network consists of each node being connected to-two other nodes to form a ring as illustrated in Figure 2-4. If one node sends data to a node that is not directly connected, the data has to go through all the other nodes in its path. The data can travel two paths (left or right) to get to its destination node.



2.4.4 Mesh Topology

In a fully connected Mesh Topology (as illustrated in Figure 2-5) every node is directly connected to all other nodes on the network. This makes it possible for all nodes to send data at the same time. This also allows for redundancy to be incorporated into any system using a mesh topology configuration. A message can take an alternative route if one route is corrupted or blocked. Due to the amount of wiring required this is considered a costly approach.

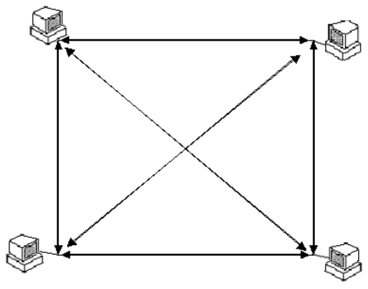


Figure 2-5: Mesh Topology

2.4.5 Hybrid Topology

These three topology types can be combined to make Hybrid Topologies such as a Tree Topology where there is a central bus and there is a ring and/or star topologies branching off.

2.5 LAN (Local Area Network) & WAN (Wide Area Network)

Networks can generally be categorised in one of two fundamental groups; LAN (Local Area Network) and WAN (Wide Area Network). These incorporate some features from the OSI (Open Systems Interconnect) model which is discussed in detail in section 2.9.

2.5.1 LAN

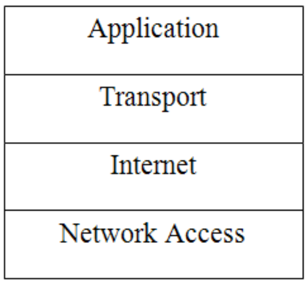
LANs connect as few as two devices together or as many as a few thousand. The interconnection between the devices can be via cable or wireless means. LANs usually connect devices that are in relative close proximity to each other such as a workstation and a printer in the same building. In a LAN, a server can contain data applications and services that other devices are allowed access. Ethernet is a technology that is widely associated with LANs. Ethernet LAN primarily deals with the physical and data-link layers (Teare, 1999). The MAC layer on a LAN uses a method called Carrier Senses Multiple Access/Collision Detection (CSMA/CD) for dealing with contention on the network. A device using CSMA/CD, listens on the network when it has data to be transmitted and, if no other device is using the network it transmits. After transmission it listens to see if a collision has occurred. If a collision has occurred each message is re-transmitted after a random length of time. If a number of other devices are using the network, performance degrades due to the increased number of collisions. Introducing switches on a network sub-divides the network into smaller collision domains resulting in less contention and improved performance.

2.5.2 WAN

WANs are used to connect multiple LANs over a larger area. A WAN uses dedicated or switched connections to link computers in geographically remote locations that are too widely dispersed to be directly linked to the LAN (Parnell, 1997). They can be connected via public telecommunications system or private communications. The Internet is an example of a public communications system.

2.6 Communication Protocol

A communication protocol is a set of rules or standards that enables communications or data transfer between two end-points (SearchNetworking.com, 2007). A protocol enables communication between a host and a remote host as long as the communication takes place on the same level. If the rules are not kept then communication cannot occur. The TCP/IP reference model is illustrated in figure 2-6 but is not presented in detail as it is not covered in the scope of this research.



2.7 Automotive Networks History

As data networks began evolving, the automotive industry was beginning to add more electronic components and features to automobiles, such as air conditioning and electric windows. While data networks were developed to improve quality of service and increase data transfer rates, the automotive industries initial motivation would have been fundamentally financial. This has changed in recent years as the number of safety critical applications increases and consumers demand more comfort applications such as climate control. The introduction of a networked bus structure in automobiles has lead to a reduction in the size of wiring harnesses. Reducing the size of a wiring harness also yielded a reduction in the weight of the automobile and in turn this would improve fuel efficiency.

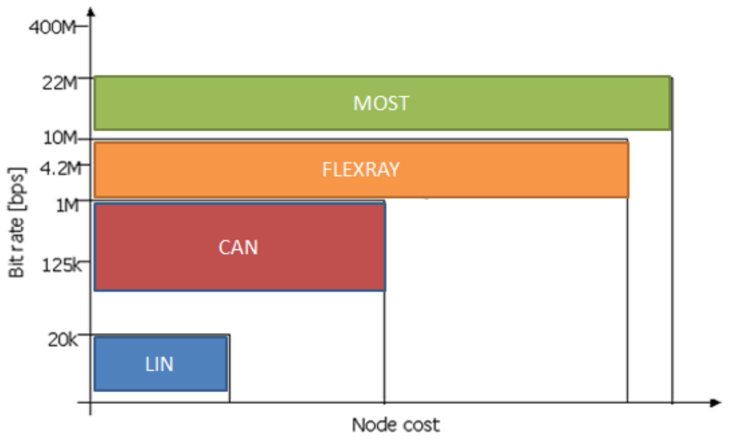
While initially introducing networks in cars reduced weight, this also lead to other safety/non-safety applications being developed, this increased weight and power consumption. The availability of integrated circuits in the 1960s and the 1970s allowed further development of automotive electronics and the development of the electronic control unit (ECU). The use of the ECU allowed engine management parameters to be varied depending on external conditions like engine load and air temperature. At the time, the main factor pushing automotive electronics development was exhaust emissions regulations. Mechanical methods alone could not provide the means to meet these new requirements while maintaining performance and efficiency (Fischerkeller, 2007). It was then realised that the microcomputer adapted for automotive use could address the requirements for emission controls while maintaining performance for the driving enthusiast. The microcomputer could handle data from the spark timing with variations in the load speed, inputs from sensors providing data on crankshaft position, coolant temperature etc (Jurgen, 1999). Increased development of integrated circuits allowed car manufacturers to eliminate passive components. Then with the development of microprocessor, electronic engine controls, anti-lock braking systems and trip computer revolutionised automobile electronics (Buchholz, 2006).

Originally, as automobile manufactures started to develop more electronic based applications for cars, there was no link between the electronics industry and the automotive industry. The automotive industry wanted the technology to be as economical as possible, but within the electronics industry all new technologies are initially expensive (due to research and development costs) and then become more cost effective due to improvements in production methods, product life cycles and widespread adaption.

It is no coincidence that during the 1980s the seminal breakthrough in automotive network development was facilitated by the production of more powerful ICs (Integrated Circuits). The CAN protocol was also developed during this period and by the early 1990s had become the de-facto standard in the automotive industry. There have been other variants of this protocol such as TTCAN, CANOpen, DeviceNet, CAN Kingdom and J1939. Proprietary variants such as J1850 PWM, J1850 VPM and ISO 9141-2 were developed individually by manufacturers but these are not included in the scope of this research. There are also three other primary automotive protocols. Local Interconnect Network (LIN) was developed as a low cost, lower speed alternative to CAN for use primarily in non-critical body control unit (BCU) functions. The Media Oriented System Transport (MOST) protocol was developed with high bandwidth infotainment systems specifically in mind. The FlexRay protocol was developed for use in safety critical applications.

2.8 Automotive Networks Description

Automotive networks can be classified by their protocols. There are four main protocols CAN, LIN, FlexRay and MOST. Each protocol is selected for its ability to meet the system requirements while keeping the implementation and build costs as low as possible. The relative cost per data rate is illustrated in Figure 2-8.



A general rule of thumb is that the fastest networks tend to be the most expensive as is illustrated by Figure 2-8. LIN operates at speeds with a maximum data rate of 20 Kbit/s and is typically used in applications that require the driver to initiate operation. In critical applications that occur rapidly or do not require the driver to initiate operation the faster networks such as CAN and FlexRay with speeds of up to 10Mbit/s (1Mbit/s for CAN) are used even though they are more expensive to implement than LIN. In 1994 the Society of Automotive Engineers (SAE) defined the classification of four classes of networks; Class A, Class B, Class C and D.

- Class A: Data rates 10 kbits/ or lower
- Class B: Data rates from 10 to 125 kbits/s
- Class C: Data rates from 125 kbits/s to 1 Mbit/s
- Class D: Data rates over 1 Mbit/s

Class A is used in the body electronics of the car and an example is LIN as mentioned above. Class B is used to share information between ECUs to reduce the number of sensors required. A low speed CAN network would fall into this category. A class C network would be utilised by a real-time high-speed communications system i.e.

powertrain or chassis applications using high speed CAN. And finally the class D network is used by MOST and FlexRay in applications that require predictability and fault tolerance (NICOLAS NAVET, 2005).

While the protocol parameters change from protocol to protocol a basic network structure can be summarised in Figure 2-9. CAN (Controller Area Network) is the de-facto network standard in the automotive industry. CAN operates on a peer-to-peer basis i.e. no master or slave, a message is transmitted and the node that requires the data gains access to the data and processes it.

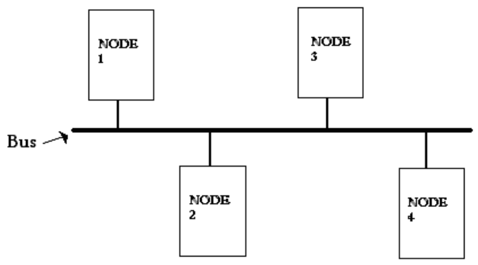
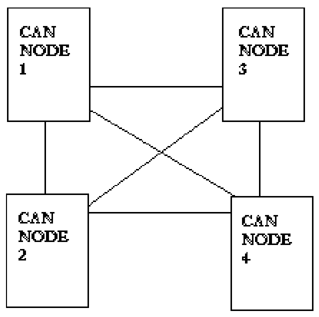


Figure 2-8: Basic Network Configuration

In Figure 2-10 we see an illustration of a network configuration where each node is directly connected to every other node on the network (point-to-point). Figure 2-5 (mesh topology) is also an example of this as is previously explained. For a simple four-node system there is not much system complexity, but modern high to medium end automobiles have over 70 interconnected ECUs therefore dramatically increasing system complexity (A. Albert1, 2005). Networking eliminates redundant wiring because a sensor only has to be wired to the nearest controller and the controller will transmit the information over the network (Jared Busen, 2006).



Networking improves many aspects within the automotive industry such as design flexibility, diagnostics and reduced wiring/weigh/cost (Philip Koopman, 1998). Design flexibility is improved because designers just have to design as far as the controller and the network takes the information to the required controller in a different part of the automobile. Having one central access point on the network and connecting this into the diagnostics tool enables error messages on the bus to be read and analysed. This enables problems to be located quicker than having to test each functional area individually.

Networking allows the provision for scalability within a system. This allows a system to grow as more applications are added. A system can only grow if in the initial design stage provision is made for possible expansion or increased data through put at a later stage.

2.8.1 Generic Node Composition

Examining the components of a typical automotive network node reveals a general outline similar to that in Figure 2-11. Here a sensor provides input data to an Electronic Control Unit (ECU). The sensors data is processed in the ECU and transmitted across the network so that any required actions will be taken if necessary. For output data we could have an actuator in place of the sensor.

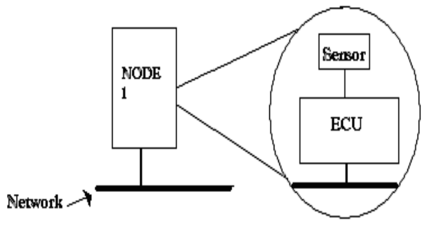


Figure 2-10: A Basic Node Comprising of a Sensor and ECU

Before the introduction of ECUs into engine management systems the amount of fuel in a cylinder, quantity of air mixed in and the ignition time of the spark plugs were all parameters set by the designer at design time. By introducing ECUs this enabled these parameters to be mapped depending on varying engine load values, quality of air mixture etc. By having optimum operating conditions the car can achieve its best performance as efficiently as possible. With greater emphasis on reducing carbon emissions, it is with the aid of an ECU that these regulations can be met. The ECU receives data from the sensor and uses data tables and calculations to make adjustments to the actuating devices (Jurgen, 1999). The ECU can also help to perform system diagnosis whenever an error occurs.

2.9 ECU

The electronic control unit (ECU) in an automobile has three main functions;

- Perform decision-making capabilities (e.g. adjust intake values)
- Perform arithmetic calculations
- Store data
- Read inputs from sensors
- Perform Actuations based on decisions

The ECU is based around the microprocessor because it can perform the three basic requirements as mentioned above. Modern automobiles need processing capabilities of computers and computers are based around microprocessors. The microprocessor acts as a CPU (Central Processing Unit) for a computer. Early microprocessors processed up to eight bits at a time but nowadays microprocessors are available in 16-bit and 32-bit format, which enables higher processing speeds.

The CPU works in a sequential manner and a clock, usually a crystal clock, controls the timing. This enables frequencies of several megahertz giving higher processor speeds and greater accuracy for clock samples. When an instruction is received and needs temporary storage in the CPU it is stored in the memory registers. Within the registers each word received is stored in memory.

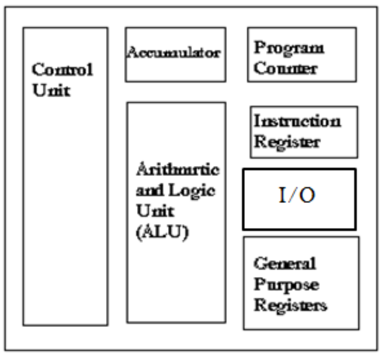


Figure 2-12 illustrates a microprocessor with associated registers. It also illustrates the program counter that keeps track of where any instructions are stored and what instructions the CPU requires running. The ALU (Arithmetic Logic Unit) performs the arithmetic calculations in binary. Any data that is being processed by the ALU is stored in the accumulator until the calculation is finished. The control unit directs movement of data in the computer. What enables a microprocessor to carry out instructions in the desired manner are the instructions it receives, these instructions would usually be conveyed via software through a program.

A basic computer can consist of the following elements; a microprocessor, extra memory, inputs and outputs. Data is stored at a specific address is memory and, when the processor requires data, fetches it from that address.
When storing data in memory the type of memory used is influenced by the tasks function. Source code does not change and therefore would be stored in ROM (Read Only Memory). If the same task was performed repeatedly without change in the process then ROM type of memory can be selected, an example of this would be on a fuel management system that uses fixed data. If the data is only required for a temporary period then RAM (Random Access Memory) can be used. Once the power is disconnected from this type of memory the data is lost. An example of this would be a trip computer in a automobile (also called run-time data) (Hillier, 1996). Non-volatile RAM (NVRAM) is also used in instances where stored data is not to be erased. The primary area where NVRAM is used is in data recording such as in the odometer.
A bus interconnects the components of the microprocessor; these buses are named in relation to the data they carry such as the address bus, data bus and control bus. For an automotive based computer the inputs and outputs would typically be sensors, transducers and actuators.

Current practice in the automotive industry is that the physical network bus interconnecting between nodes is a cable. Connectors attach the cable to the hardware. The protocol is a contributing factor when cable specifications are set.

2.10 Automotive Network Protocols

As discussed in the previous section one reason for introducing networks into modern automobiles was due to the increased levels of new features and applications in automobiles. Design engineers decide what protocol is run on the network. Some considerations that have to be taken into account when deciding what protocol shall be used are;

- Is the application critical?
- What speeds are required for the application?
- What costs are associated with each protocol?
- Is the technology readily available to develop the application with the chosen protocol?

The answer to these questions will often decide the protocol chosen, whether it is an event-triggered (ET) or time-triggered (TT) system.

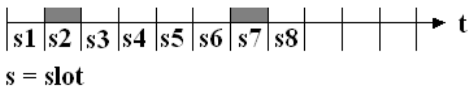
2.10.1 Event-Triggered Protocols

In an event-triggered (ET) system, traffic gets put onto the network after an event has occurred such as the driver pressing the brakes. This is asynchronous transfer because there is no predetermined time at which these events will occur. Because any event can occur at any time in any order, the network has to have a developed system that will avoid collisions if two messages on two separate nodes try to gain access to the network at the same time. This is achieved by tagging each message with a priority level. The message with the highest priority will be granted access to the bus once it is free. This is an efficient use of bandwidth due to the fact that only messages that need to be transmitted will be looking for access to the network (NICOLAS NAVET, 2005). An event-triggered communication controller does not need a dispatching table because the transmission of a message is triggered by a send command from the host (Kopetz, 2000).

2.10.2 Time-Triggered Protocols

With the time-triggered (TT) protocol, transmission occurs at predefined points in time (Andrei Hagiescu, 2007). Activities can only occur with the progression of time and the activity is predefined. This requires the network to have a pre-defined schedule that assigns activities/tasks a section of the time (time slot) to perform the required action. Each task is made up of messages. If a message is not transmitted in its defined time slot it waits until its next time slot.

In Figure 2-13, a message is assigned slot two (S2) in which to transmit. If the message does not obtain access to the bus in slot (S2) its next assigned slot where it will get a chance to gain access to the bus is in slot seven (S7).



A time-triggered communication controller contains a dispatching table in its local memory that determines what point in time a particular message is transmitted or when that message is expected to be received (Kopetz, 2000). If a new node is added to the network all other nodes need to be updated due to a change in the schedule (if the new node was an unplanned addition). This can result in inefficient bandwidth usage as some messages might not need to transmit the whole time, but will need to be allocated slots by the system designer (Andrei Hagiescu, 2007).

2.10.3 Comparison of Event-Triggered and Time-Triggered Networks

ET systems have an advantage over TT systems when it comes to adding new nodes onto the network. With ET systems the new node can be added on where as in TT systems the new node has to be scheduled into the system at design time. This involves the system designer re-organising the system schedule to accommodate the new IDs on the new node. As stated previously bandwidth efficiency is greater on ET systems because a message can be transmitted if bandwidth is available where as on a TT system a message can only transmit when it has been scheduled to do so even if there are no messages scheduled in the slots before it. TT systems can be fault tested to a higher degree because the designer has the schedule prior to execution. In ET systems run time testing is critical for catching potential faults due to the aperiodic nature of the protocol. Also in TT networks, since the application does not control the timings there is a common time base for all nodes which allows the communication controller to synchronise to the message schedule (Hartwich, 2007).

After considering these features the properties of each protocol can be examined. Only CAN and FlexRay are discussed as LIN and MOST were not encountered during this research.

2.11 Conclusion

In this chapter the concept of computer networks was introduced starting with the general principle and moving onto automotive systems. Differing network topologies and architectures were discussed before an example of the TCP/IP reference model is presented, whose protocol is used extensively in the Internet. The seven layers of the OSI reference model are explained as this reference model is used to compare layers of other protocols. A brief history about the reasons for using of networks within the automotive industry is presented along with an overview of a nodes hardware composition. Finally the chapter concludes with a discussion on TT and ET protocols and a comparison of strengths and weakness of the two.

This chapter presents the necessary background to understanding the purposes and requirements of networks within the automotive industry. By presenting data network models such as the OSI model this allows comparisons to be made when discussing automotive protocols in later chapters such as the comparison made in Chapter 3 between CAN and the OSI model. By presenting an overview of the various network types (TT and ET) the limitations and advantages of each architecture type are revealed.

2.12 References

A. Albert1 B P, F. Voetz1,2005, Simulation Environment for Investigating the Impacts of Time-Triggered Communication on a Distributed Vehicle Dynamics Control System, 1st International ECRTS Workshop on Real-Time and Control,

Andrei Hagiescu U D B, Samarjit Chakraborty, Prahladavaradan Sampath, P.Vignesh, V. Ganesan, S. Ramesh, 2007 Performance Analysis of FlexRay-based ECU Networks

Buchholz K, 2006, Electronics history lesson. http://www.sae.org/automag/electronics/09-2002/

Casad J 2001 *Teach Yourself TCP/IP in 24 Hours*: Sams)

Comer D E 2001 *Computer Networks and Internets*: Prentice Hall)

Fischerkeller S R a R, 2007, Latest Trends In Automotive Electronic Systems - Highway Meets Off-Highway?

Harbeck R, 2006, Network, http://searchnetworking.techtarget.com/sDefinition/0,sid7_gci212644,00.html [15/03/2008]

Hartwich F,2007, Implementation Concept for Bosch FlexRay Communication controller, 1st Vector FlexRay symposium, Stuttgart.

Hillier V A W 1996 *Hilliers Fundamentals of Automotive Electronics*: Stanley Thornes Ltd)

Institute N B R, 2002, INFORMATION COMMUNICATION TECHNOLOGY (ICT) DEVELOPMENTS – ESTABLISHMENT OF THE FIRST SWITCH BASED LOCAL AREA NETWORK AT NBRI (NBRI-LAN), [26/02/2008]

Jared Busen L E J a B K, 2006, How Automotive Networks are Configured, http://www.asashop.org/autoinc/jan2006/mech2.htm [10-Feburary-2008]

Jurgen R K 1999 *Automotive Electronics Handbook*: McGraw-Hill)

Kopetz H, 2000, A Comparison of CAN and TTP, Editor, Conference Name, Conference Location.

NICOLAS NAVET Y S, FRANÇOISE SIMONOT-LION, AND CÉDRIC WILWERT, 2005, Trends in Automotive Communication Systems, Editor, PROCEEDINGS OF THE IEEE,

Parnell T 1997 *LAN Times a Guide to Wide Area Networks*: Osborne McGraw-Hill)

Philip Koopman E T, Geoff Hendrey, 1998, Toward Middleware Fault Injection for Automotive Networks, Editor, Conference Name, Conference Location.

SearchNetworking.com, 2007 Protocol Definition http://searchnetworking.techtarget.com/sDefinition/0,sid7_gci212839,00.html [19th March 2007],

Steinke S 2000 *Network Tutorial*: CMP Books)

Stevens W R 1994 *TCP/IP Illustrated* vol Volume1: Addison Wesley)

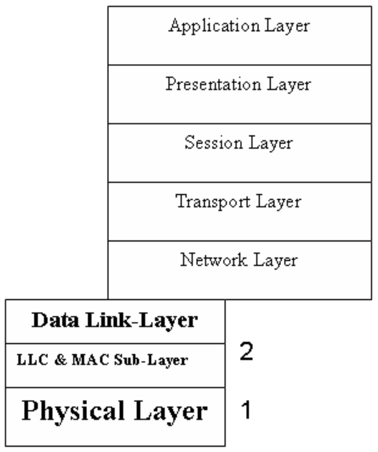
Teare D, 1999, Internetworking History. (Internet: Cisco Press)http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/introint.htm#wp1020552 [12th October 2006]

3 CAN (Controller Area Network):

3.1 Introduction

Robert Bosch GmbH originally developed CAN in 1986. It was designed as an asynchronous serial bus network for connecting devices, sensors and actuators in a system or sub-system, and was developed to be robust in electromechanically noisy environments. It was initially developed for automotive applications but also has been used in Industrial applications that are subject to noise (Corrigan, 2002).

The CAN protocol is an ISO standard (ISO 11519 for applications up to 152Kbps, ISO 11898 for applications up to 1Mbps) and includes a physical layer and Data Link Layer, layers 1 and 2 respectively, of the OSI model as illustrated in Figure 3-1. Only layers 1 and 2 of the OSI model have been defined for CAN; the Data Link Layer is composed of the Logical Link Control (LLC) sub-layer and the Media Access Control (MAC) sub-layer ((CiA), 2008, cia.org, 2001-2007). In layer two the data for transfer is encapsulated within the network level information packets and a unique identifier is assigned to each of these "communication objects" (Rufino, 1997, (CiA), 2008).

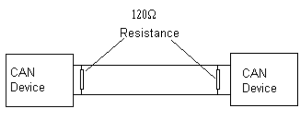


The CAN standard version is a Carrier-Sense Multiple-Access protocol with Collision Detection and Arbitration on Message Priority (CSMA/CD+AMP)(Corrigan, 2002). CSMA requires each node on a bus to wait a prescribed period of inactivity before attempting to send a message. Carrier-Sense entails that the devices attached to the network listens for other signals on the line before transmitting. If there are other devices transmitting on that line the listening device waits until the line is clear before transmitting. Multiple-Access allows many devices to connect and share the same network.

Transmission occurs if the CAN node requests the transmission of a message and the message is the highest priority message that wins arbitration over other messages trying to gain access to the network. The Arbitration process is explained in detail in section 3.2.2 and Figure 3-5.

3.2 CAN Physical Structure

CAN prioritises messages by configuring their IDs. Payload lengths of eight bytes or less are sent on a serial bus. A CAN bus is a balanced two-wire interface running over a shielded twisted pair, unshielded twisted pair or ribbon cable. In some applications different kinds of links are used e.g. optical or radio links. It is not uncommon to use different transmission mediums for applications with specific requirements. Electrical signals on the bus will be reflected back at the end of the line unless the lines are correctly terminated as illustrated in Figure 3-2. The node may receive a reflected signal instead of the intended signal and this will cause inaccuracies if the signals are different. Placing a resistor at both terminating ends of the bus can fix this and avoid unnecessarily long stub-ends of the bus.



The bit encoding used is Non-Return to Zero (NRZ) encoding with bit stuffing for data communicating on a differential two-wire bus. A sequence of more than five identical bits is a violation of the bit-stuffing rule. The CAN bus is a broadcast type bus. This necessitates that if a message is broadcast all other connecting nodes receive the message, but only the node requiring the message will react to it and process it accordingly.

3.3 CAN Frame Format

There are two types of CAN frame format;

- Standard Frame
- Extended Frame

Formatted: Bullets and Numbering

The standard frame is presented in Figure 3-3 and the CAN extended frame is presented in Figure 3-4. Standard CAN (2.0A) uses an 11bit identifier while extended CAN (2.0B) uses a 29 bit identifier (Corrigan, 2002).

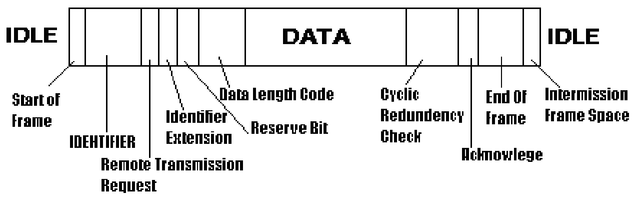
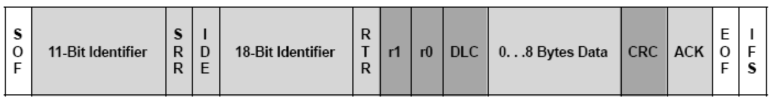


Figure 3-3: CAN Standard Frame Format

The arbitration field in standard CAN comprises of the 11bit identifier and the RTR (Remote Transmission Request) frame.



In extended CAN the arbitration field comprises of a 29bit identifier, RTR frame, IDE (IDentifier Extension) frame and SRR (Substitute Remote Request) field. Standard CAN has one reserve bit r0; Extended CAN has two reserve bits r0 and r1. The reserved bits and the DLC (Data Length Code) of both frame types are contained in the control field. CAN is comprised of four different frame types that control message transfer and they are;

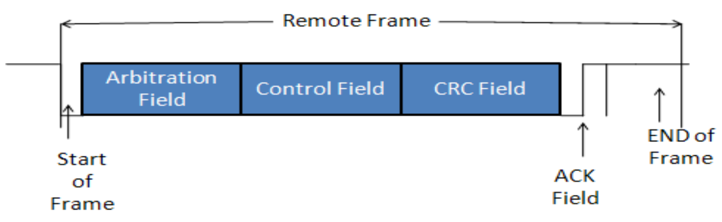
- Data Frame
- RTR Frame
- Error Frame
- Overload Frame

Data frame

The data frame is the most common message type and is made up of the data field, arbitration field, the CRC (Cyclic Redundancy Check) field and the acknowledgment field as is illustrated in Figure 3-3. This frame interacts with the MAC layer (layer 2) on the OSI model. The arbitration field determines the priority of the message when two or more nodes are contending for the bus. The data field contains zero to eight bytes of data and the CRC field contains the 16-bit check sum used for error detection. The acknowledgement field is used when a message is received and the ACK bit overwrites the recessive bit at the end of correct message transmission. The transmitter checks for the presence of an ACK bit and retransmits the message if no acknowledge bit is detected.

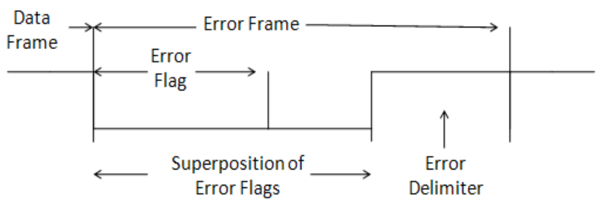
Remote transmission request frame (RTR)

The remote frame is intended to solicit the transmission of data from another node. There are two main differences between this and the data frame; first this type of message is explicitly marked as a remote frame in the arbitration field and secondly it contains no data. The RTR bit is dominant is a data frame and recessive in a remote frame. The remote frame is presented in Figure 3-5 (BOSCH, 1991).



Error frame

The error frame is transmitted when a node detects an error in a message and causes all other nodes in the network to send an error frame to confirm they also have detected the error. The error frame contains two fields; Error Flag and Error Delimiter. The error flag is followed by the superposition of error flags as illustrated in Figure 3-6. To terminate an error frame correctly the bus may need to be idle for three bit times (BOSCH, 1991).



Overload frame

The overload frame is transmitted when a node becomes too busy and it gives an extra delay between messages. It is similar to the error frame in format. The overload frame contains two fields; the Overload Flag and the Overload Delimiter. At most two overload frames can be transmitted to delay the next data or remote frame. The overload frame is illustrated in Figure 3-7 (BOSCH, 1991).

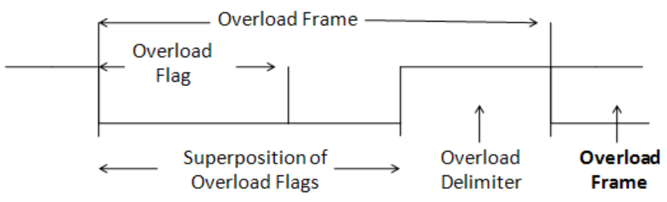


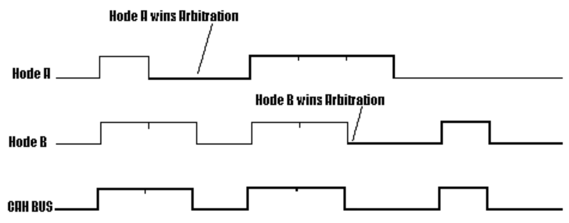
Figure 3-7: Overload Frame Format

3.4 Bus Arbitration

Bus arbitration is the method by which messages gain access to the network when two or more messages request access at the same time.
Each message is tagged with a priority (lowest value is the highest priority). Each message identifier must be unique and assigned at design time. The message ID defines the priority level. The CAN protocol uses non-destructive bit wise arbitration to control access to the bus. It is non destructive as the node winning arbitration continues on with the message without ‘the losing’ message being destroyed or corrupted by another node. The node that loses arbitration joins the queue again. This guarantees access to the bus for the controller with the highest priority message. There can be some latency if a message is already being transmitted or a higher priority message wants access to the bus. If a low priority message and a high priority message are vying for bus access, there will be greater latency for a lower priority message.

For this non-destructive bit-wise arbitration to take place some preconditions have to be established. First the logic states need to be defined as dominant or recessive. Second the transmitting node must monitor the state of the bus to see if the logic state it is trying to send actually appears on the bus (Pazul, 1999). Normally logic high is associated with a one and a logic low is associated with a zero however the CAN bus defines a logic bit zero (0) as the dominant bit and logic bit one (1) as the recessive bit. The dominant bit state will always win arbitration over the recessive bit state therefore the lower the value of the bit identifier (value of all zeros) the higher the priority of the message. Messages that need to be transmitted more often on the network should be assigned a higher priority ID e.g. Data coming from engine management is of higher criticality than data for air conditioning if on the same bus.
If two nodes are trying to get access the bus say nodes ‘A’ and ‘B’ and node ‘A’ wins arbitration. Because the nodes are continuously monitoring transmission node ‘B’ sees that the bus state does not match what it transmitted so it stops transmitting. This allows node ‘A’ to continue with transmission. Both nodes vie for the bus again once

node 'A' releases control and both nodes want to re-transmit at the same instant. Figure 3-8 shows this process.



3.5 CAN Error Handling

Error handling prevents a single node from locking the network (Corrigan, 2002). The CAN protocol in total incorporates five error checking methods (BOSCH, 1991) ;

- Bit Error
- Stuff Error
- CRC Error
- Form Error
- Acknowledgment Error

When the CAN controller detects an error it transmits an error frame. Every CAN controller on the bus will try to detect errors in its messages. The node that discovers the error will transmit an error flag. By transmitting an error flag the bus traffic is destroyed and any other node that detects the error flag will also discard its current message. Each node maintains two error counters, a transmit error counter and a receive error counter. The error counters are modified accordingly.

1. When a RECEIVER receives an error the RECEIVE ERROR COUNT is increased by one except when the received error was a bit error sent during an ACTIVE ERROR FLAG or an OVERLOADING FLAG.

2. When the receiver detects a ‘dominant’ bit as the first bit after sending an error flag, the receive error count will be increased by 8.
3. When the transmitter sends an error flag the transmitter increases the error count by 8. Two exceptions are:

   If the TRANSMITTER is ’error passive’ and detects an ACKNOWLEDGMENT ERROR because of not detecting a ’dominant’ ACK and does not detect a ’dominant’ bit while sending its PASSIVE ERROR FLAG

If the TRANSMITTER sends an ERROR FLAG because a STUFF ERROR occurred during ARBITRATION whereby the STUFF BIT is located before the RTR bit, and should have been ‘recessive’, and has been sent as ‘recessive’ but monitored as ‘dominant’.

4. If a TRANSMITTER detects a BIT ERROR while sending an ACTIVE ERROR FLAG or an OVERLOAD FLAG the TRANSMIT ERROR COUNT is increased by 8.
5. If a RECEIVER detects a BIT ERROR while sending an ACTIVE ERROR FLAG or an OVERLOAD FLAG the RECEIVE ERROR COUNT is increased by 8.
6. Any node tolerates up to 7 consecutive ’dominant’ bits after sending an ACTIVE ERROR FLAG, PASSIVE ERROR FLAG or OVERLOAD FLAG. After detecting the 14th consecutive ’dominant’ bit (in case of an ACTIVE ERROR FLAG or an OVERLOAD FLAG) or after detecting the 8th consecutive ’dominant’ bit following a PASSIVE ERROR FLAG, and after each sequence of additional eight consecutive ’dominant’ bits every TRANSMITTER increases its TRANSMIT ERROR COUNT by 8 and every RECEIVER increases its RECEIVE ERROR COUNT by 8.
7. After the successful transmission of a message (getting ACK and no error until END OF FRAME is finished) the TRANSMIT ERROR COUNT is decreased by 1 unless it was already 0.
8. After the successful reception of a message (reception without error up to the ACK SLOT and the successful sending of the ACK bit), the RECEIVE

ERROR COUNT is decreased by 1, if it was between 1 and 127. If the RECEIVE ERROR COUNT was 0, it stays 0, and if it was greater than 127, then it will be set to a value between 119 and 127.

9. A node is ’error passive’ when the TRANSMIT ERROR COUNT equals or exceeds 128, or when the RECEIVE ERROR COUNT equals or exceeds 128. An error condition letting a node become ’error passive’ causes the node to send an ACTIVE ERROR FLAG.
10. A node is ’bus off’ when the TRANSMIT ERROR COUNT is greater than or equal to 256.
11. An ’error passive’ node becomes ’error active’ again when both the TRANSMIT ERROR COUNT and the RECEIVE ERROR COUNT are less than or equal to 127.
12. A node which is ’bus off’ is permitted to become ’error active’ (no longer ’bus off’) with its error counters both set to 0 after 128 occurrence of 11 consecutive ’recessive’ bits have been monitored on the bus.

Using the error counters the CAN node not only detects errors but also contains them by detecting the constant transmission of error frames. The CAN bus changes between Error Active, Error Passive and Bus Off depending on the error counter value (Daniel Mannisto, 2003). The parameters are illustrated in Table 3-1. They also ensure that a node cannot tie up the bus by repeatedly re-transmitting error frames.

Table 3-1: CAN Error States

	0 – 127 Errors	128-255 Errors	>255 Errors	Comment
Error Active	X			Error counter operates as normal, if error occurs error flag is sent and message is destroyed
Error Passive		X		Can only send out passive error flag once an error is detected, can still send and receive messages. Cannot become error active until counter falls below 128
Bus Off			X	Can only enter this state due to transmit error counter exceeding 255. Node that is bus off cannot influence the bus in this state. Controller and counter reset and start sequences are sent

The maximum and minimum data rate for a CAN network is 1Mbps and 10Kbps respectively. Cable length depends on the data rate being used. Normally all the devices in the system transfer at a uniform and fixed bit rate. The maximum line length is 1km; 40metres at 1Mbps. Termination resistors are used at each end of the cable.

3.6 CAN Protocol Features

The CAN protocol provides four primary benefits;

- A standardised communication protocol simplifies and economises interfacing subsystems from various OEMs onto a common network
- The communication load is shifted from the host CPU to the intelligent peripheral that gives the host CPU more time to run its system tasks
- Using a multiplexed network vastly reduces the size of the wiring harness and eliminates point-to-point wiring. This can increase the efficiency of a vehicle due to the fact it is carrying less weight
- The broad market appeal of CAN is that it can be employed in multiple industries (aerospace, manufacture of automobiles (robotics)) and motivates the semiconductor industry to manufacture and develop competitively priced CAN devices

Because CAN is a mature technology and has been developed over a number of years, the cost of developing and improving CAN based systems has fallen. The CAN communication network is an event-triggered architecture. This means that the occurrence of an event is recognised by the system as a change at an input or sensor etc. By operating a message priority system CAN is deployable in real-time distributed systems. For real-time deployment to work there has to be a guaranteed minimum message delivery time, where the worst case delays should not be allowed exceed the maximum delay. Also an 8bit microcontroller with as little as 4K of memory and 256 bytes of RAM is able to support a CAN application. One of the main drawbacks of CAN is its non-deterministic nature. This makes it impossible to test completely a CAN based system for every possible error therefore making live testing much more critical to enable delivery of an error free product to the customer. One of the major disadvantages of CAN is the restricted room for further development of applications as the bandwidth limitations are being reached (i.e. lack of scalability). Traditionally CAN systems were designed to have busloads of 30-40%. This increased further to busloads of 80% with the development of scheduling tools such as Volcano Network Architecture (Robert I. Davis, 2007). To guarantee message delivery time's a certain percentage of the bandwidth needs to be unused to prevent the blocking of low priority messages at high busloads. Future scalability cannot be improved upon further in these type CAN systems without adding additional CAN sub-buses.

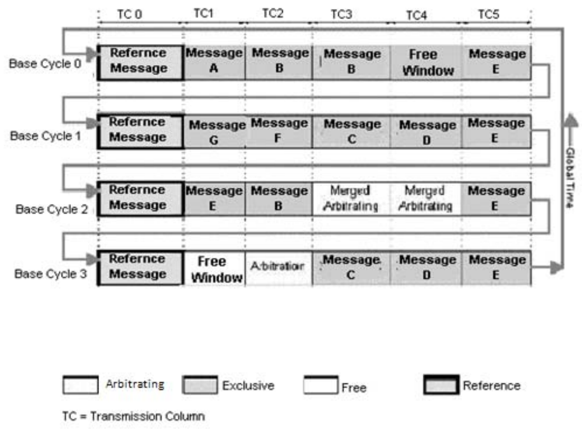
3.7 TTCAN (Time-Triggered Controller Area Network) Introduction

TTCAN addresses some of the shortcoming of CAN such as non-determinism through the use of TT architecture. TTCAN offers a predefined time-triggered method of scheduling CAN messages for synchronous message transfer. For asynchronous message transfer, TTCAN instigates message transfer with occurrence of an event (Centre, 1998 - 2005) .

3.7.1 TTCAN Cycle Structure

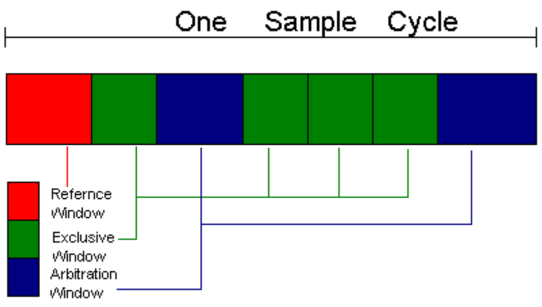
To maximise the use of TTCAN over CAN the network requires a deterministic element. This allows a reduction in latency (jitter) values of high priority messages getting access to the bus, therefore providing a deterministic aspect to the network. A time master bases communication on the periodic transmission of a reference message. Once this reference message is sent by a node all other nodes on the network synchronise themselves to this. This provides CAN with quasi event-triggered protocol features.
An advantage of using the TTCAN based system is the possibility of transmitting an event-triggered window during a time-triggered slot. This is achieved by transmitting the event-triggered message in the arbitrating time window. During this window ET messages vie for access to the network. This is achieved through the arbitration method CAN uses, if there is more than one message looking for access to the network.

The TTCAN communication cycle is called a matrix cycle as seen in Figure 3-9 (G. Leen, 2001).



The start of each cycle is denoted by its reference message. The time masters reference messages are used to guarantee the operation of CAN. This operates on extension level 1. Extension level 1 guarantees the TT operation of CAN based on the reference message of a time master. The reference message on this extension level only requires one byte for control information the rest of the bytes (7bytes) can be used for data. In extension level 2 a globally synchronized time base is established and a continuous drift correction among the CAN controllers is realised. On extension level 2, a global synchronisation time base is used to counter any drift that occurs. This extension level requires 4 bytes for control information and additional bytes can contain data (Thomas Führer, 2000).

As can be seen from Figure 3-9 the matrix cycle is made up of a base cycle. Each base cycle is the length from one reference message to the start of the following reference message. In between the reference messages are time windows. Each window per base cycle can be different but all base cycles have to contain the same properties relating to time windows. Time windows can be use to transmit periodic or aperiodic messages depending on the configuration by the design engineer. This means that the cycle is fixed and is not changed while online. The time window that transmits periodic messages is called an exclusive window, where as a time window for aperiodic messages are called arbitration windows as illustrated in Figure 3.10. After the second arbitration window the cycle repeats to build up into a matrix.



TTCAN can be implemented where CAN has already been implemented, and is covered under the standard ISO11898-4 (A. Albert, 2003). TTCAN is compatible with the existing CAN network as they have the same physical layer.

TTCAN has advantages over CAN such as it contains time-triggered and event-triggered properties. This means TTCAN is deterministic yet still has some of the flexibility of CAN that allows for more efficient bandwidth usage. TTCAN can also be used in companies that have invested in knowledge of CAN because the tools and equipment are cross compatible. Even though TTCAN offers improved bandwidth utilisation ~~was developed after One area where~~it does not offer increased bandwidth when compared to the maximum bandwidth available in CAN.~~falls down is it has the same bandwidth limitations as CAN.~~

3.8 Conclusion

In this chapter the specifics of the CAN protocol are discussed in depth. The chapter starts off with an overview of the CAN layers in relation to the OSI reference model. Then the CAN frame format is explained in relation to the standard 11-bit frame format and the 32-bit format. Arbitration and error checking are extensively discussed as these parameters play a significant role in the successful implementation of CAN. TTCAN is also discussed in this chapter due to it being derived from CAN. The differences between CAN and TTCAN are discussed as is the cycle structure of TTCAN.

Because CAN is an established protocol it continues to be widely used within the automotive industry. Factors such as integration into a company's development model and availability of expertise will ensure that the CAN protocol will not face a sudden discontinuation in use. It has also been recognised by the automotive industry that CAN is incapable of providing for the future requirements of all applications. TTCAN is further proof that TT features in a protocol are required in the automotive sector. This has lead to alternative protocols such as FlexRay and TTTech (TTTech joined the FlexRay consortium (Technology, 2005)). It appears that through the volume of research (the FlexRay consortium have extended their agreement until 31st December 2009) and through product releases, FlexRay will be the de-facto automotive protocol (Thomas Noltey, 2005) where CAN is not suitable.

3.9 References

(CIA), C. I. A. (2008) CAN Open,06/11/2008
A. ALBERT, R. S., A. TR¨ACHTLER (2003) 'Migration from CAN to TTCAN for a Distributed Control System' 9th international CAN Conference, Munich,
BOSCH (1991) CAN Specification Version 2.0.
CENTRE, C. A. S. R. (1998 - 2005) TTCAN,03/03/2008
CIA.ORG, C. (2001-2007) CAN Physical Layer,
CORRIGAN, S. (2002) Introduction to Controller Area Network (CAN)
DANIEL MANNISTO, M. D. (2003) An Overview of Controller Area Network (CAN) Technology. mBus.
G. LEEN, D. H. (2001) TTCAN: a new time-triggered controller area networkTTCAN: a new time-triggered controller area network,12/02/2008
PAZUL, K. (1999) Controller Area Network (CAN) Basics. Microchip Technology Inc.
ROBERT I. DAVIS, A. B., REINDER J. BRIL AND JOHAN J. LUKKIEN (2007) Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. Springer Science + Business Media.
RUFINO, J. E. (1997) An Overview of the Controller Area Network. Instituto Superior T´ecnico, in the Technical University of Lisbon.
TECHNOLOGY, P. E. (2005) TTAutomotive joins FlexRay Consortium,05/11/2008
THOMAS FÜHRER, B. M., WERNER DIETERLE, FLORIAN HARTWICH, ROBERT HUGEL, MICHAEL WALTHER, ROBERT BOSCH GMBH (2000) 'Time Triggered Communication on CAN' Proceedings 7th International CAN Conference, Amsterdam, Robert Bosch GmbH
THOMAS NOLTEY, H. H., LUCIA LO BELLO (2005) Automotive Communications - Past, Current and Future

4 FlexRay:

4.1 Introduction

This chapter describes the FlexRay protocol in detail. Specific areas of focus are on FlexRay's physical structure, frame structure and communication cycle operation. The following sections are critical to the development of the migration framework as they form the core of the FlexRay protocol.

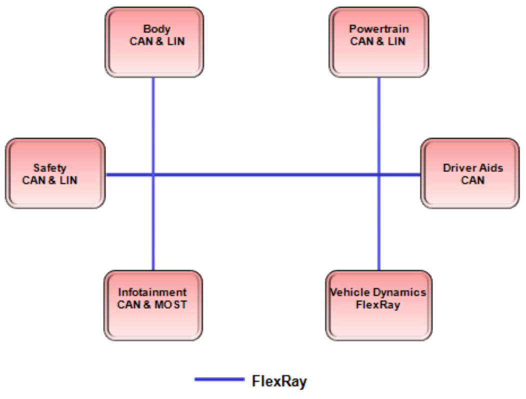
FlexRay offers reliable, real-time capable, high-speed data transmission between electrical and mechatronic components. The FlexRay consortium was founded by;

- Automobile manufacturers
- Semiconductor manufacturers
- Experts in communication technology
- And system suppliers to benefit the automotive industry ~~founded the FlexRay consortium~~

All companies involved jointly developed a new standard that is openly available to members and is intended to supplement the established data busses of CAN, LIN and MOST. A communication network is the backbone of an X-by-wire system. Initial development of X-by-wire will involve having the mechanical and hydraulic mechanism as a back up to the critical devices such as steering, breaking etc (Thomas Führer, 2000).

FlexRay is an alternative protocol to complement existing protocols as illustrated in~~can be seen from~~ Figure 4-1.

Comment [FW1]: Complement existing protocols and furthermore, address implementations that are not suitable....



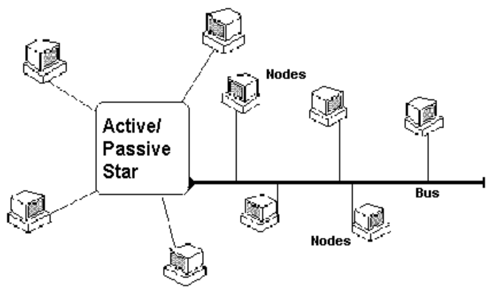
The objectives of the FlexRay consortium are the joint development of an innovative and high-quality communication system and a comprehensive infrastructure for future distributed applications. This involves developing the specifications for the communication protocol, the physical layer, the bus guardian, hardware and the software interfaces. The consortium was founded by BMW, Daimler Chrysler, Motorola semiconductors products sector (now Freescale semiconductors) and Philips semiconductors in 2000. In total 103 companies are part of the FlexRay consortium; 7 of which~~them~~ are~~s~~ core members, 25~~ as~~ premium associate members, and 71~~ as~~ associate members (Consortium, 2008).

4.2 FlexRay Features

The FlexRay consortium was set up because it became obvious that existing data transfer rates used within the chassis, body and power trains of today's vehicles will reach their limits in the next generation systems. The main features of FlexRay are (Consortium, 2008);

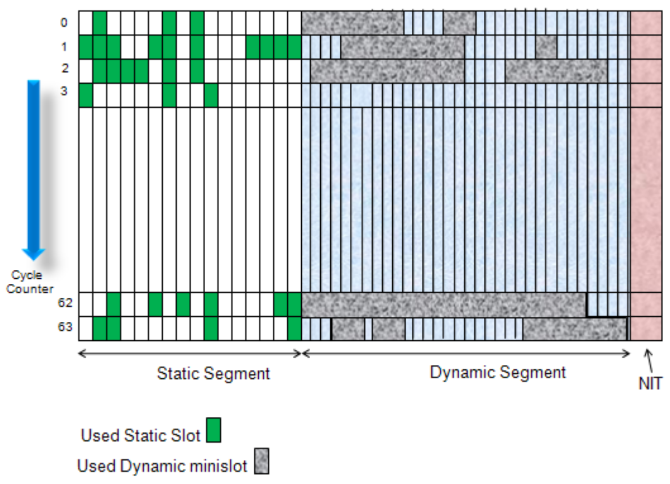
- Synchronous and asynchronous data transmission
- Support of a fault tolerant scalable time-base
- Scalable electrical/electronic architectures supporting a multiple of platforms
- Single channel gross data rate of 10Mbits/s
- Deterministic data transmission with guaranteed message latency and message jitter
- Support for redundant transmission channels
- Fault tolerant and time triggered services available in hardware
- Arbitration free transmission
- Support for bus and star topologies
- Fast error detection and signalling
- Support of wake-up and sleep functionality via the bus

FlexRay can attain a~~has~~ data transfer rate of 20Mbit/s over dual channels or a data rate of 10Mbit/s on a single channel. FlexRay supports deterministic data transfer and~~.~~ ~~FlexRay supports~~ numerous network topologies such as point-to-point, passive star, linear passive bus, active star network, cascaded active stars and hybrid topologies. For interconnection two primary topologies are proposed; star based interconnection topology and a bus based interconnection topology. Both of these interconnection methods can use dual channel systems. The star configuration can be deployed in distributed configuration such that two star-based subsystems are connected by star-to-star links. A distributed system can also be designed by combining the star and bus approach allowing several nodes to be connected to a branch of the star as shown in Figure 4-2.



4.3 FlexRay Cycle Structure

FlexRay communication is based on 64 ~~occurs in~~ reoccurring communication cycles, composed of a Static Segment (ST), Dynamic Segment (DYN), Network Idle Time (NIT) and an optional Symbol Window, as illustrated in Figure 4-3. The communication cycle is the fundamental element of the media access scheme within FlexRay. FlexRay offers two media access schemes; TDMA (Time Division Multiple Access) and a dynamic mini-slotting based access scheme. The duration of the cycle is fixed when the network is~~becomes~~ configured. A time window, which is defined by the communication cycle, can be divided into two parts, a static segment and a dynamic segment. In addition each cycle can contain a symbol window that is used for run time testing, and a network idle time that allocates a communications free period upon the conclusion of each communication cycle.

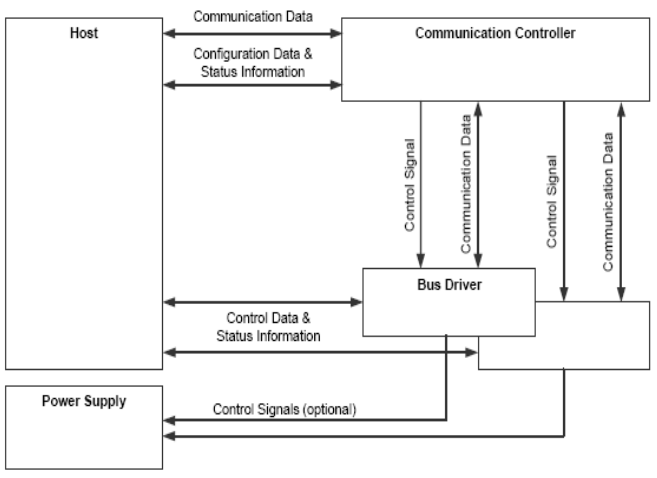


The purpose of the static segment is to provide a time window for scheduling ~~a number of~~ time-triggered messages. This part of the cycle is reserved for synchronous communication, which guarantees a certain frame latency and jitter through fault tolerant clock synchronization. The messages which are to be transferred through the static segment must be configured before starting communication (offline). For this to be achieved the static segment uses a TDMA based communication scheme. In the dynamic part of the communication cycle each device may transfer event-triggered messages, which are prioritised by frame ID. Temporal characteristics of the communication cycle are defined at design time and stored statically in each node. Nodes that require greater bandwidth are assigned more slots than those that require less bandwidth.

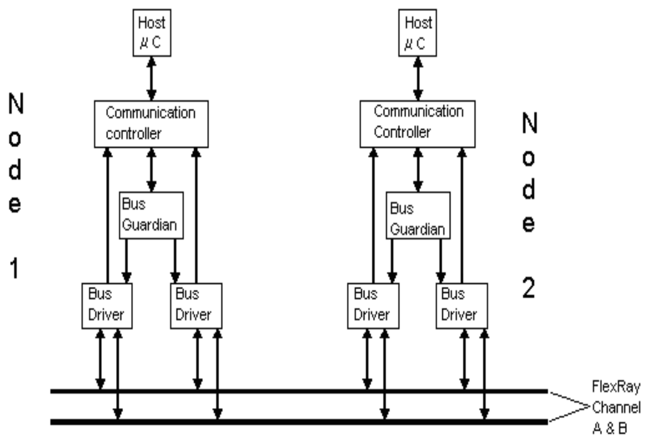
Currently most ~~H~~high-speed vehicle control systems are ~~today~~ networked ~~by~~using CAN. If one of the wires in two-wire CAN becomes cut or shorts, the timing behaviour of the CAN bus becomes unpredictable. When one wire is cut or shorted interference can increase on the bus because the inverted voltages on each wire minimise the interference cancelling effect. With excess interference CAN bus data can be rendered useless for its intended purpose. One of the major aims when developing FlexRay was scalable fault tolerance. Using this approach FlexRay could be used in non-fault tolerant systems as well as fault tolerant systems. FlexRay provides scalable fault tolerance by a means of a dual channel system with mixed connectivity (some nodes connected to both channels other nodes connected to only one).

4.4 FlexRay Node Structure

Each FlexRay node consists of a controller component and a driver component as can be seen in Figure 4-4.

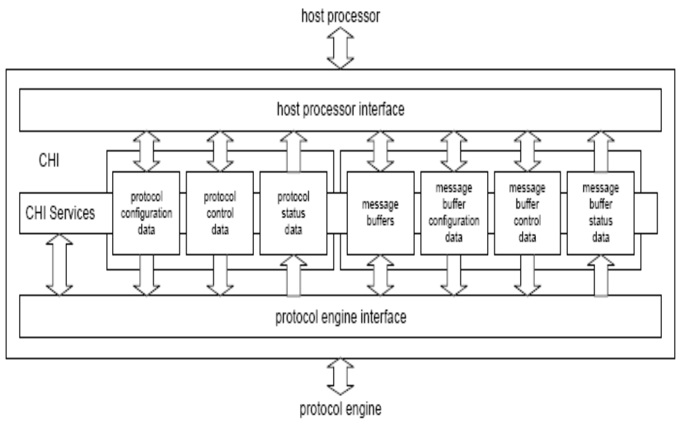


The controller component includes a host processor and a communication controller. The driver component includes the bus drivers and optional bus guardians. The bus guardian protects the communication channels from transmission faults that violate the TDMA scheme. The bus guardian also prevents malfunctions by only granting bus access for sending a message at predefined times for each node. The bus driver connects both the communication controller ~~to the bus~~ and the bus guardian ~~access~~ to the bus. The host informs the bus guardian which time slots the communication controller has allocated. The bus guardian then allows the communication controller to transmit data only in these time slots. If the bus guardian detects a gap in the timing it disconnects the communication channel. A summary of how each component of a FlexRay node interacts with another component to enable data to be transmitted onto the network is illustrated in Figure 4-5.



4.4.1 CHI (Controller Host Interface)

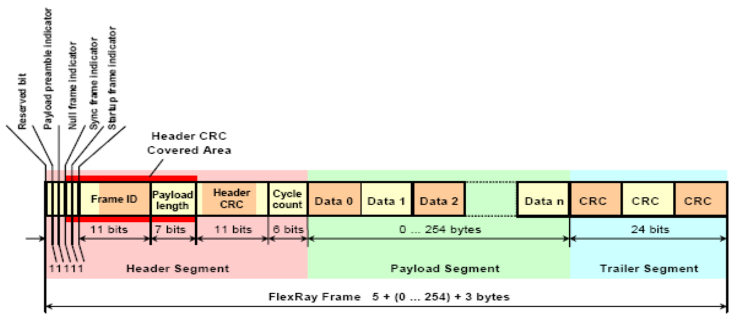
The FlexRay core (between the host processor and the protocol engine) is partitioned such that the Protocol Engine (PE) is responsible for all FlexRay specific protocol handling and the Controller Host Interface (CHI) handles all tasks of integrating FlexRay functionality into the rest of the system. Figure 4-6 shows how the protocol engine interfaces with the host processor via the CHI.


host processor
host processor interface
CHI
CHI Services
protocol configuration data
protocol control data
protocol status data
message buffers
message buffer configuration data
message buffer control data
message buffer status data
protocol engine interface
protocol engine

The CHI provides host access to the FlexRay protocol's core~~'s~~ configuration~~;~~ (control and status register). ~~as~~Also the CHI provides access to ~~well as to t~~the message buffer configuration~~,~~ (control and status register). The FlexRay buffers hold the FlexRay frames (Receive and Transmit frames) including the frame header, payload data and frame status information. The message buffer data is stored in the FlexRay memory and the message buffer control structures are implemented in the CHI. Different end user applications have different requirements therefore the core should be configurable to optimise application performance.

4.5 FlexRay Frame Structure

The FlexRay frame format is illustrated in Figure 4-7. The frame consists of three segments; the header segment, the payload segment and the trailer segment. When a node configuration~~ appears on the network it~~ is transmitted~~s~~ on the network, transmission occurs in ~~-~~this order (header, payload and trailer).



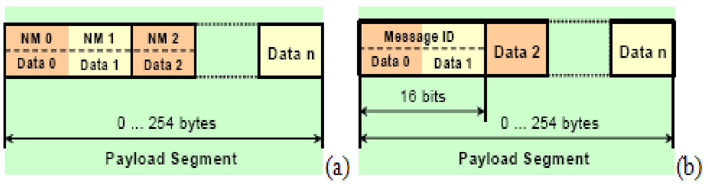
4.5.1 Header Segment (5 bytes)

Reserve Bit

The 1 bit reserve bit is reserved for further development so it shall be ignored and set to zero.

Payload Preamble Indicator

The payload preamble indicator (1 bit) indicates if the payload section contains an optional network management vector as illustrated in Figure 4-8a. If the message is transmitted in the static segment it indicates the presence of a network management vector (see Figure 4-8a). If it is transmitted in the dynamic segment it indicates if there is a message ID (see Figure 4-8b). If it is set to zero neither are contained in the payload preamble indicator.



Null Frame Indicator

The null frame indicator (1 bit) indicates whether there is a null frame. If the null frame is set to zero the frame contains no valid data and all bytes in the payload section are set to zero.

Sync Frame Indicator

The Sync frame indicator (1 bit) indicates whether or not the frame is to be used for system synchronisation. When set to a zero the frame is not considered for synchronisation, otherwise it is considered for synchronisation.

Start-Up Frame Indicator

The start-up frame indicator (1 bit) indicates if the frame is a start-up frame otherwise is given a value of one otherwise it is given a value of zero. Only coldstart nodes can transmit start-up frames.

Frame ID

The frame ID (11 bits) defines the slot in which the frame should be transmitted. The frame ID ranges from 1 to 2047, an ID value of zero is invalid. The node transmits the frame ID with the most significant bit (MSB) being transmitted first and decreasing in order (to the LSB).

Payload Length

The payload length (7 bits) indicates the size of the payload segment. It is set by getting the number of payload data bytes and dividing it by two (e.g. a frame that contained a payload segment of 72 bytes would be transmitted with a length 36). The payload length is fixed when sent in the static segment but may vary for frames sent in the dynamic segment.

Header CRC

The header CRC (cyclic redundancy check) (11 bits) uses a CRC code calculated over the sync frame indicator, the start up frame indicator, the frame ID and the payload length. The communication controller (CC) calculates the header CRC of a received frame in order to check that the CRC is correct. The header CRC of transmitted frames is calculated offline and is provided to the communication controller during configuration. Computation of the header CRC is done by first shifting in the sync frame indicator then the MSB of the frame ID followed by the rest of the frame ID, then the MSB of the payload length and the subsequent bits of the payload length. The header CRC is transmitted with the MSB transmitted first.

Cycle Count

The cycle count (6 bits) keeps track of the value of the cycle counter at the time of frame transmission.

4.5.2 Payload Segment (0 – 254 bytes)

Because the payload segment contains two-byte-words it must contain an even number of bytes. The bytes are numbered starting at zero and increasing by one byte with every subsequent byte~~ after~~. In Figure 4-8~~10~~ Data0 is referred to as the first byte and Data1 as the second byte as so forth. For frames in the dynamic segment the first two bytes may contain the message ID field. For frames in the static segment up to twelve bytes may be used to indicate a network management vector. The MSB of each byte will be transmitted first with the subsequent bytes following.

4.5.3 Trailer Segment (3 bytes)

The trailer segment contains a 24bit CRC for the frame. The CRC frame contains a CRC code computed over the header segment and the payload segment of the frame (all fields within these segments are included). The CRC is computed using the same generator polynomial on both channels but a different initialisation vector is used for each of the two channels. The frame fields are fed into the generator starting with the reserved bit and ending with the least significant bit (LSB) of the last byte of the payload segment. The CRC frame is transmitted starting with the MSB descending in sequence to the LSB (Consortium, 2005).

4.6 FlexRay Timing Hierarchy

Having discussed the FlexRay frame format the next critical component to understanding FlexRay is its timing hierarchy. The timing hierarchy can be broken into four distinct levels for analysis:

Level 1: MicroTick level (µT)
Level 2: MacroTick level (MT)
Level 3: Arbitration Grid Level
Level 4: Communication Cycle Level

Formatted: Bullets and Numbering

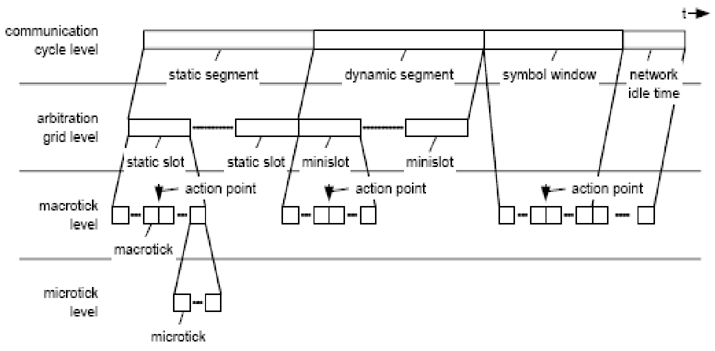
Formatted: Bullets and Numbering

Level 1 (Microtick): The µT is a time unit that is extracted from the communication controller's oscillator clock. A pre-scalar value can be used if necessary. The microtick value is given in units of microseconds. The smallest temporal granularity of a node is in µT.

Level 2 (macrotick): The MT value is an integer multiple of µTs. The number of µTs per MT can fluctuate between nodes and also fluctuate between MTs on the same node (Consortium, 2005). Action points designate when transmissions on the MT level start. The MT figure is also presented in microseconds.

Level 3 (arbitration grid level) is where the static slots are defined for the static segment and mini slots are defined for the dynamic segment.

Level 4 (communication cycle) is composed of the static segment, dynamic segment, network idle time and optional symbol window. The symbol window is where media access takes place. Network idle time is where no transmission takes place except to apply clock correction values and it is required to conclude every communication cycle. The number of MTs per cycle shall be an integer value and this is constant through all nodes in a specific cluster(Consortium, 2005). All four levels are illustrated in Figure 4-9.



As mentioned previously there are two media access schemes, TDMA and a dynamic mini slotting access scheme (or flexible time division multiple access (FTDMA) as it is sometimes referred). The static segment operates~~runs on the~~a TDMA scheme and the dynamic segment operates a~~runs on the~~ dynamic mini slotting scheme. Each static slot for the TDMA scheme is obtained from the arbitration level. The mini slot scheme also obtains its base mini slot from this level as can be seen in Figure 4-9.

4.7 Communication Cycle

The FlexRay communication cycle frame is composed of the ST segment the DYN segment the optional symbol window and the NIT (Network idle Time). It is during the NIT that calculations are carried out for the offset and phase as discussed in the section 4.7.5. Figure 4-10 illustrates an example of the composition of a FlexRay frame for transmission in a communication cycle. The communication cycle is as illustrated in Figure 4-3 previously.

The communication cycle is composed of an integer number of macroticks. The number of macroticks per cycle is the same for all nodes in a cluster. Also all nodes should have the same cycle value at any given instance of time. The cycle counter value ranges from 0-63. Once the cycle counter reaches the maximum value it resets and starts counting again from zero in the next communication cycle. The cycle counter value is incremented at the start of each communication cycle.

4.7.1 Wake-Up

For a cluster wake-up all bus drivers are required to be supplied with power. Start up occurs on all channels synchronously. At least one node in the cluster requires an external wake-up source. The CC (Communication Controller) provides the host with the ability to transmit the wake-up pattern and it prevents a disturbance on the communication channel once communication occurs. The host sets the configuration in the CC as to what channel is to be woken up. Each channel available is woken separately. Both channels must not be woken at the same time to prevent a faulty node disturbing communication simultaneously on both channels with its transmission.

The wake-up pattern causes any fault free node to change from sleep mode to wake-up if it is still in sleep mode. The bus driver of the receiving node recognises the wake-up patter and triggers wake-up. During the wake-up procedure any number of nodes can try and wake-up a channel. This issue is resolved during the wake-up process. The wake-up pattern is collision resilient. If a fault causes two nodes to transmit wake-up patterns the signal resulting from the collision can wake-up other nodes.

4.7.2 Communication Start Up

As FlexRay is based on a TDMA scheme, synchronicity has to be present for successful communication. A node has to be in the wake up state before start up can commence. Start up is initiated by a coldstart node. Only a node assigned as a coldstart node may initiate this process. The node that starts the cluster is called the leading coldstart node and the nodes that follow are called the follow coldstart nodes.

Start up is initiated by the coldstart node transmitting a CAS (Collision Avoidance Symbol). Only this coldstart node can transmit in the four cycles that follow this transmission. Next other coldstart nodes are allowed transmit then all other nodes follow. The coldstart node contains the *pKeySlotUsedForStartUp* parameter set to true and the header contains the start-up frame indicator set to one. The *pKeySlotUsedForStartUp* parameter set a bit that signifies the node as being a coldstart node. If the cluster contains three nodes or less each node shall be considered a coldstart node. Each node is also configured to be a sync frame, enabling each node to also be a sync node. Only start-up frames can be transmitted during the start-up process. For a follow-coldstart node first it listens on the FlexRay channel for FlexRay frames. On reception of a valid pair of start-up frames, the node tries to derive its clock correction and schedule from the coldstart node. If this is unsuccessful it collects all sync frames and performs clock correction in the following double cycle. If clock correction does not signal any errors and the node continues to receive sufficient frames from the node it had integrated upon, it transmits its start-up frame. If this is unsuccessful it returns to listen-mode.

For non-coldstart nodes, the node listens to the FlexRay channel to receive FlexRay frames. It searches for two coldstart nodes that transmit start-up frames that fit its own schedule. If this is unsuccessful or clock correction throws an error the node aborts integration and tries again from the start. Once two valid start-up frames are received the node can leave the start-up phase and move into the operation phase.

The coldstart inhibit mode is available to prevent the node initialising the communication cycle. This mode can be used to prohibit active start-up attempts of a node or delay start-up attempts.

4.7.3 Clock Start-Up

Before synchronisation can occur, clock start-up has to take place. This is dependent on the start and initialisation of the MTG (Macro Tick Generation) process and the initialisation and start of the CSP (Clock Synchronisation Process). The clock within the node can be started through the coldstart node process if it is the leading coldstart node or it adopts the initialisation values of a coldstart node.

4.7.4 Cold-Start

If no communication is detected on the channels, a leading coldstart node needs to be assigned to trigger communication. This leading coldstart-node initiates the start-up procedure by sending start up frames. There has to be a minimum of two fault-free coldstart nodes for successful start-up. There is a limit of between 2-31 coldstart attempts that can be made as specified by (Consortium, 2005).

4.7.5 Node Synchronisation

As FlexRay is run on a distributed communications system every node requires its own individual clock. This leads to the challenge of synchronising every node because operating a time-triggered protocol requires a global time base. This is achieved in FlexRay through the ~~use~~transmission of sync frames~~ being transmitted~~. Synchronisation can be divided into two concurrent processes; the macrotick generation process (MTG) and the clock synchronisation process (CSP).

The MTG controls the cycle and MT counter and applies rate and offset correction values. This is achieved by adjusting the number of µT per MT.

The CSP is achieved by performing initialisation at the start of a cycle and then calculating and storing the new deviation, offset and rate correction values. There are a set of precision ~~differing~~ values ~~allowed~~to be selected from. There are two types of precision~~differing~~ values to chose from; offset (phase) and rate (frequency) differences. FlexRay combines both of these to calculate offset and rate correction values that in turn synchronises the local time base of each node.

~~Synchronisation on the FlexRay network is carried out in a number of stages. A FlexRay driver implements these steps~~The driver operation examined in detail in this thesis is the DECOMSYS COMMSTACK and is discussed in detail in Chapter 10~~ the COMMSTACK<FLEXRAY> that is a controller software driver package~~. Other software driver packages are available from companies such as Fujitsu (emea, 2008) and Vector (Informatik, 2008).

~~The version used is 1.8.2 from DECOMSYS. The FlexRay controller is reset therefore no access to the controller is permitted.~~

~~Reset is performed again~~

~~1 causes transition to the Config state reset has been completed~~

~~Wakeup: The FlexRay controller transmits a wakeup pattern~~

~~Abort causes the transmission of the wakeup pattern to be aborted immediately and returns to the off state.~~

4.8 Conclusion

Over the past few years the development of the communication hardware has been the main priority, but as this reaches a more advanced stage ~~we can start looking at~~ developing the required applications can now be investigated. Due to FlexRay's characteristics (Large bandwidth, deterministic behaviour and fault tolerance) it is ideally suited to the role of a central backbone of future ECU architectures. With the introduction of a new bus system, there is a~~we~~ need to be able to incorporate the older systems and products as well as being able to integrate support for the new FlexRay features. When testing and developing these systems it is vital that they can be tested under real-time conditions.

Judging by the strength and backing of the core and associate members of the FlexRay consortium it appears that it will becomes a dominant standard in automotive distributed applications. The most probable~~likely~~ use ~~for it~~ in the near future will be as a backbone network used in conjunction with other networks such as CAN and LIN. These other networks are not expected to become redundant; instead these will be used for other (less critical) control systems in the automobile. By the fact that the FlexRay consortium agreed to extend their agreement from the 1st of January 2009 until 31st December 2009 demonstrates that they are intent on taking the technology forward. The FlexRay consortium was formed out of necessity due to present protocols not being able to meet predicted demands of future vehicles. TTCAN went some way towards meeting these requirements but it was not able to fully cope the way FlexRay could with extra bandwidth requirements. One potential major advantage for FlexRay over other protocols is the reserve for future functional extensions (Schedl, 2007). FlexRay will very likely become the de-facto standard for in-vehicle communications (Traian Pop, 2007).

4.9 References

Consortium F, 2005, FlexRay protocol specification Version 2.1 Revision A, [15th December 2005]

Consortium F, 2008, The FlexRay Consortium Website, http://www.flexray.com

emea F, 2008, FlexRay Product Overview. http://mcu.emea.fujitsu.com/mcu_product/overview_FlexRay.htm

Informatik V, 2008, XL-Driver Library. http://www.vector-worldwide.com/vi_xl_driver_library_en.html

Schedl D A,2007, Goals and Architecture for FlexRay at BMW, 1st Vector FlexRay Symposium, Stuttgart, Germany.

Thomas Führer B M, Werner Dieterle, Florian Hartwich, Robert Hugel, Michael Walther, Robert Bosch GmbH,2000, Time Triggered Communication on CAN, Proceedings 7th International CAN Conference, Amsterdam.

Traian Pop P P, Petru Eles, Zebo Peng,2007, Bus Access Optimisation for FlexRay-based Distributed Embedded Systems, EDAA,

5 Automotive Embedded Systems Design:

5.1 Automotive Embedded System Design

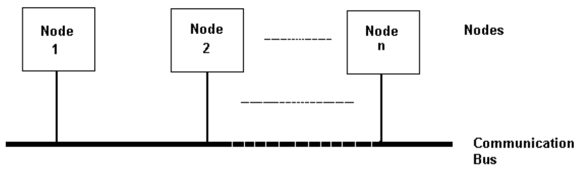
In this chapter, methods of automotive embedded design are discussed. First there is a review of the concept of distributed architectures and real-time operating systems in the context to the automotive industry. Secondly, design approaches for automotive embedded systems are discussed. Thirdly, scheduling techniques are reviewed and finally analysis and conclusions are presented.

5.2 Distributed Architectures - Introduction

Computer architecture is concerned with the design and performance of the system as a whole (Jordan, 2004). In distributed systems processing power is distributed among computers in a cluster or network (Englander, 2000). Communication protocols and standards used in data networks allow data to be transferred between different systems. This allows each system to do part of the processing giving higher overall throughput and fault tolerance. This allows tasks that involve mass amounts of computation to carry out this computation across the network of connected computers.

The complexity and physical distribution of modern active-safety automotive applications requires the use of distributed architectures (Abhijit Davare, 2007).
In distributed computing the fact there are multiple autonomous computers should be transparent to the end user (Lobur, 2003). When a command is executed in a distributed operating system it selects the processors, manages transport to the

processors and stores results (Tanenbaum, 1996). In computer networking all file management has to be called explicitly (Tanenbaum, 1996). This is where a 'true' distributed system varies from a networked system. A basic distributed architecture consists of a more than one node connected by a communication bus as illustrated in Figure 5-1.



5.2.1 Basic Node Design

Each node (as illustrated in Figure 5-2) contains a CPU, communications controller, memory and an interface for an IO (Input/output). Typical IO interfaces are the reception of data from a sensor or a signal to actuate an actuator.

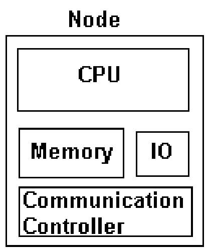


Figure 5-2: General Node Properties

The node architecture for a FlexRay node is to some extent different to the general example in Figure 5-2. This is due to a CHI (Controller Host Interface) being required to interact between the CPU and the communication controller. The CHI explicitly implements the FlexRay protocol and is independent of the CPU (Traian Pop, 2007). The FlexRay node architecture is presented in detail in Figure 4-4.

The communication bus is configured according to the protocol being implemented. Tasks can be assigned to processors through software and communicate with each other using messages. Each task is capable of communicating with another task on a separate processor by transmitting a message on the communication bus. The tasks can be periodic or aperiodic with or without precedence constraints. The deadlines can be soft (non-critical) or hard (critical) depending on the application requirements.

5.3 Real-Time Operating System (RTOS)

The primary difference between real-time systems and other computer systems is that the response time is viewed as the crucial part of the application software in a real-time system (E. Douglas Jensen, 1985).
Real-time and embedded systems operate in environments that offer restricted memory and processing power. For this reason an operating system capable of responding in real-time is required. In these environments an RTOS can be designed to extract fast and predictable real-time responses (Schaffer, 2006). A hard real-time system is one that has fatal errors if deadlines are missed. Soft real-time systems deadlines are not as critical as hard real-time deadlines, if deadlines are missed.

Distributed architectures supporting the execution of hard real-time applications are not only common in automotive systems, but also in avionics and industrial control systems (Abhijit Davare, 2007). A RTOS has a set of functions that are required to be carried out when implementing an application on the host system. Some of these functions are pre-emption, time-sharing, interrupt handling and memory allocation.

5.3.1 RTOS Functions

The operating system controls tasks that carry out specific functions. The task to be executed is defined by a schedule (TT) or the occurrence of an external event (ET). Through the use of interrupts a task with a high priority can get executed if a lower priority task is currently being executed. An interrupt declares that a task with a higher priority than the task currently being executed is waiting for execution. Once the lower priority task finishes execution the higher priority task is then executed. If pre-emption is implemented the higher priority task will be executed before a lower priority task is finished executing. The lower priority task is then executed if it is the next highest priority task is scheduled for execution. Without pre-emption the high priority task will have to wait until the lower priority task is finished execution before it gets executed. An interrupt handler is required to enable pre-emption. When a message is available for execution it notifies the interrupt handler and it is assigned an interrupt priority and placed in the interrupt queue, the position in the queue depends on its priority level. The interrupt request handler (IRQ) notifies the system that an interrupt is pending at a certain location, so the system pauses momentarily while it decides what action is required to deal with the pending interrupt. The interrupt can be a hardware interrupt (external interrupt) or software interrupts (from an instruction set).

5.4 OSEK/VDX

OSEK/VDX is a set of standards for distributed real-time architectures developed by a consortium of European automobile manufacturers and suppliers in conjunction with the University of Karlsruhe, Germany (Lemieux, 2001). The OSEK/VDX RTOS is widely used within the automotive industry. The primary parameters standardised are;

- The Operating System (OS)
- Communication (COM)
- Network Management (NM)
- The OSEK Implementation Language (OIL)

5.4.1 OS (Operating System)

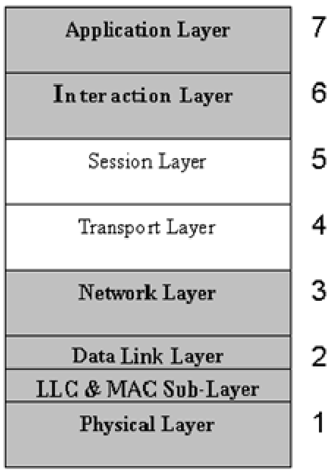
The OSEK OS is a single processor OS designed for distributed embedded controls. The OS specification is intended to give an efficient utilisation of the environment resources for automotive control applications (OSEK, 2005). This system can be applied in applications that require a compact, real-time distributed system that is not automotive based such as consumer electronics items.

5.4.2 COM (Communications)

The OSEK standard supports both inter-task communication and inter-process communication. The OSEK/VDX model covers five layers that are defined by the OSI reference model as illustrated in Figure 5-3;

- Application
- Interaction
- Network
- Data link
- Physical

In Figure 5-3 the layers of the OSEK model that are also defined in the OSI reference model are shaded in (Layers 1, 2, 3, 6 & 7). The Interaction layer is similar to the presentation layer in the OSI model. This is where the exchange of data between tasks is specified.



5.4.3 NM (Network Management)

The NM specification was designed primarily for the automotive industry. The basic function of NM is to monitor the status of the nodes on the network, report status information to the application and to handle APIs (Application Programming Interfaces) for control of NM components.

5.4.4 OIL (Operating system Implementation Language)

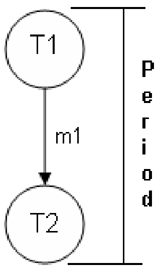
Since OSEK is a static operating system the definition of the system objects used is required before compile time (Informatik, 2008). The OIL file is composed of two parts; the implementation specific part and the application specific part. The implementation specific part is supplied with the OSEK/VDX implementation (Lemieux, 2001) and should not be changed. The application specific part can be changed as deemed necessary by the designer during application development.

5.5 Process Models

A process model shows the relationship between processes that make up a system. This helps identify changes that maybe required in helping the system function more efficiently. When implementing a process model the first basic parameter is to know what system you are modelling. You can break down the model to the following configurations

- ET system
- TT system
- Mixed system

A system can be broken down into set of basic components. As illustrated in Figure 5-4 a simple task graph contains Tasks *T1* and*T2*, the message *m1* and the task graph period. Each task can run on the same ECU, or in the other extreme every task in a task graph can run on a separate ECU.



5.6 Task Scheduling Policies

There are a number of scheduling policies (procedural rules) to consider when formalising a scheduling solution in a real-time system. Some of the methods for consideration are earliest deadline (ED), deadline monotonic (DM), rate monotonic

(RM), least slack scheduling (LSC), priority promotion (PP) and mixed traffic scheduling (MTS). The primary objective of any scheduling policy is to meet the message and task deadlines. Scheduling methods are classed as fixed priority scheduling, static priority scheduling or dynamic priority scheduling.

5.6.1 Dynamic Priority Scheduling

Dynamic priority scheduling allows a tasks priority to be altered during run time. This increases the overhead required to implement this scheduling method. This method is non-deterministic which is not ideal for real-time systems (Oshana, 2007b).

5.6.2 Least Slack Scheduling

In this scheduling method the slack is calculated from the absolute deadline minus the execution time for the task to complete execution. The task with the least slack is then given the highest priority to best guarantee it will meet its deadline.

5.6.3 Earliest Deadline (ED)

The earliest deadline method assigns the highest priority values to tasks that have the shortest time in which to complete. This method works in a system that has plenty of capacity and uses pre-emption, but once the system starts to overload performance degrades rapidly (Carey, 1991). This is due to tasks that have not already missed their deadline, but are closest to missing them, being assigned the highest priority value. This then results in tasks that can still make their deadline not being assigned the highest priority values. The task that is closest to its deadline might not actually be able to complete in due time. If the processor utilisation bound is greater than 100% deadlines will start to be missed. The utilisation bound is calculated by each process execution time divided by its period as presented in Equation 5-1.

$$\frac{\text{Process ExecutionTime}}{\text{Process Period}} = \text{Utilisation Bound}$$

Equation 5-1: Utilisation Bound

5.6.4 Fixed Priority Scheduling

Fixed priority scheduling requires that the priority of a task is not changed in real-time. The message or task schedule has to be known prior and scheduled off-line in a scheduling table. This approach does not allow new messages that are created during the run-time to be processed.

Two of the most common protocols that use this method are rate monotonic (RM) and deadline monotonic (DM). The most significant advantages of these methods are simple implementation and good exploitation of available bandwidth (Natale, 2000).

5.6.4.1 Rate Monotonic (RM)

Using the RM scheduling method the higher a tasks frequency the higher its priority, therefore the task with the shortest period has the highest priority (Buttazzo, 2004). When using RM the following can be assumed (Sha, 1991)

- Task switching is instantaneous.
- Tasks account for all execution time.
- Task interactions are not allowed.
- Tasks become ready to execute precisely at the beginning of their periods and relinquish the CPU only when execution is complete.
- Task deadlines are identical with task periods.
- Tasks with shorter periods are assigned higher priorities; the criticality of tasks is not considered.
- Task execution is always consistent with its rate monotonic priority: a lower priority task never executes when a higher priority task is ready to execute.

Using these factors, worst-case latencies can be calculated then compared to the timing requirements to determine if deadlines shall be met.

5.6.4.2 Deadline Monotonic (DM)

Deadline monotonic was derived from the rate monotonic scheduling; it is a generalised version. The task with the shortest deadline has the highest priority; the priority is derived from its message ID. No task can have a task deadline longer than the task period because the task needs to be finished before it can run again.

Both (RM and DM) systems require the highest priority task running, but in practice this is not always possible. It is impossible to get immediate transition between tasks due to a transition period existing; this transition period can be as small as a faction of a millisecond.

5.6.5 Mixed Traffic Schedule

Mixed Traffic Scheduling (MTS) is employed through a combination of; fixed priority scheduling and dynamic scheduling.
One example of this approach was by (Shin, 1995) where a MTS consisting of ED and DM but without the need for long IDs was implemented. The non-pre-emptive version of DM is used so that once a message starts transmission it always completes. Simulated results from Shin show that in terms of timing, MTS has superior performance when compared to ED.

5.7 Task Graphs

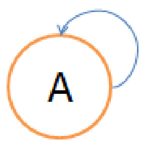
Task graphs visually represent the parameters associated with each task that comprises an application. The task graph also shows the sequence of tasks that require execution to successfully execute the application. Task graphs are used to analyse and adjust constraints when developing an algorithm. They can also be used in ECU task allocation, network/process design and performance modelling (Vikram Adve, 2006). Thus a graph is a way of representing connections or relationships between pairs of objects (Michael T. Goodrich, 2002). Task graph analysis involves similar techniques used in critical path analysis. This allows for an abstract representation of the application or system.

5.8 Critical Path Analysis (CPA)

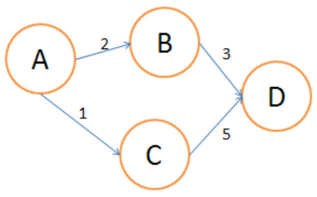
Critical Path Analysis (CPA) techniques were developed separately in Great Britain and the USA around the 1950s and 1960s. In Great Britain in the 1950s the Operational Research (OR) section of the Central Electricity Generating Board, investigated methods concerned with the overhaul of a generating plant. This led to a technique identifying the longest irreducible sequence of event. Using this technique the overhaul time of a plant was reduced by 42%, in the 1960s this was further reduced by 32% (Lockyer, 1974). Also in the USA in early 1958 the US navy special projects office set up a team to deal with the planning and control of complex works. This was known as Performance Evaluation Review Task (PERT). This work resulted in early arrow diagram drawings. Furthermore, in 1958 the US air force implemented a technique called Critical Path Method (CPM) to control large projects (Lockyer, 1974).

CPA techniques are used across a wide variety of industries from construction, sales, marketing, production or any sector where project planning is required. Path analysis was developed because the previous methods such as Gantt charts did not sufficiently demonstrate the inter-relationship between various tasks (Lockyer, 1974). An example of this would be the designer not being able to deduce by examining a Gantt that task *i* is or is not permissible for execution with or without precedence constraints. A precedence constraint is if task *i* is unable to start execution until task *i-1* has been received.

Using CPA the network undergoing planning can be abstracted and thought of as a model. The model undergoing CPA is composed of activities/tasks connected by arrows. Arrow diagrams themselves are graphical models scientifically drawn that represent the logical sequence of the network (Reynaud, 1970). The arrow as illustrated in Figure 5-5 shows the activities progression. The activity progression takes place in the direction of the arrow. The most basic configuration is one node and one arrow as illustrated in Figure 5-5. The head of the arrow indicates the completion of an activity.



The expected duration of the activity is indicated by the placing of a value as a subscript on the arrow indicating the direction of process flow as illustrated in Figure 5-6 (for example the activity duration A-B will take 2 units).



Dummy activities are sometimes used. These are activities that do not require resources, but can take time (Gordon, 1991). This is illustrated with a broken arrow and is used in situations where two or more parallel activities have the same head and tail. The dummy activity is used to demonstrate different process paths. A dummy path is also used to indicate logic conditions (e.g. precedence constraints).

Further developing the activity graph representation, constraints such as earliest time to commence activity and latest time to finish an activity can be illustrated. Two widely used configurations are seen in Figure 5-7. Here *A* represents the nodal label, *E* is the earliest deadline and *L* is the latest deadline.

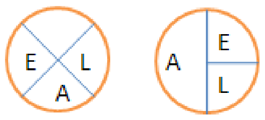
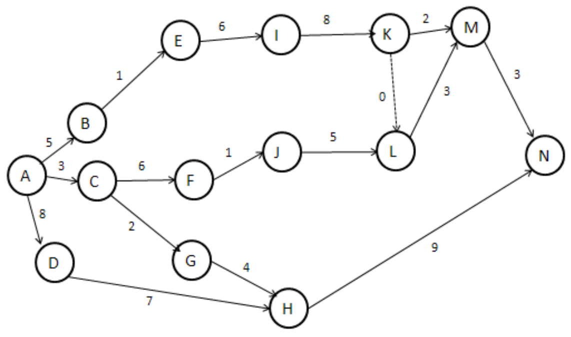


Figure: 5-7: Alternative Activity Task Configurations

Process models using CPA techniques can range from the extremely simple as per Figure 5-5, to the complicate as per Figure 5-8, and with higher complexity if required.



5.9 Task Graph Analysis

Each task on a task graph can be represented as illustrated in Figure 5-9, it contains the task number and the process time for that task. Task graphs can contain as little as one task and there is no upper limit. Task graphs can be segmented to show different process running on different systems and each task is connected by a line showing the direction of data transfer. The number on this line is the message number (Figure 5-10) for communication between two tasks (Lockyer, 1991).

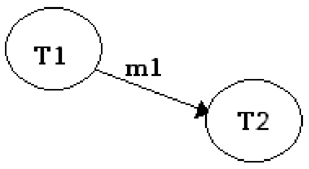


Figure: 5-10: Two Tasks, *T1* & *T2*, connected by a message *m1*

From the task graph, each task's processing time and the task graph period can be calculated. The release time (time at which the task is available for execution) r_i and the deadline time (time at which the task has completed execution of designated task) d_i for each task can also be individually calculated. In Figure 5-11, *T1* has been assigned a release time of 15*ms* and a deadline time of 18*ms*. Any unrequited task graph process time that was previously designated for the execution of tasks can now be considered slack. This slack in the system can be evenly assigned to each task on the path as per (Aloul, 2005). This will increase the amount of time that each task has to process its message.

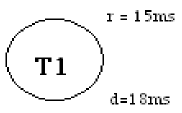
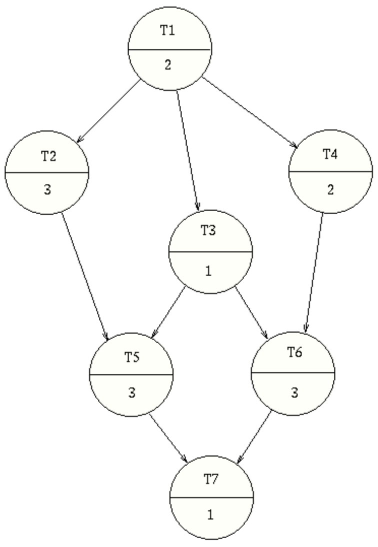


Figure 5-11: Task T1s release and deadline parameters



The assignment of tasks to processors can be presented in Gantt charts. If the diagram in Figure 5-12 is used, combined with the task parameters shown, it can be proven that different path choices are available.

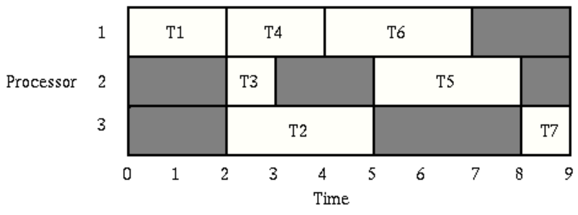


Figure 5-13: Gantt Chart for Tasks in Figure 5-12 (Hurley, 1994)

Figure 5-13 demonstrates the advantage of having multiple processors (distributed architecture) because the total execution time of all the tasks (if they were all processed on one processor) would be 15units but the execution time is down to 9units using this configuration. There are restrictions, one of which is that some tasks cannot execute until the previous task is finished. For example *T4* cannot be processed until *T1* has been executed but *T6* can be processed before *T4* if *T6* does not have restrictions such as requiring input from *T4* and *T3*.

5.9.1 Worst Case Execution Time (WCET) & Best Case Execution Time (BCET)

WCET and *BCET* analysis is used to determine if performance goals are met and if interrupts have sufficiently short time to react (Jakob Engblom, 2000). The BCET is the *"shortest execution time",* while the WCET is the *"longest execution time"* (Reinhard Wilhelm J E, 2006)

The Once the release times and deadline times are calculated for each task in a system (including the slack if appropriate), there is enough information available to calculate the *WCET* for each task. Equation 5-2 can be used to test if the obtained WCET is feasible. Where the *WCET* should be less than or equal to the deadline minus the release time, otherwise it cannot be guaranteed that the task will meet its deadlines. The message delay is the time from the signal being generated, passing through the ECU/s and communication bus/es and arriving at its destination completing execution as per Equation 5-3 (e.g. actuating an actuator) (Andrei Hagiescu, 2007).

$$w \leq d - r$$

Equation 5-2: WCET

The *WCET*, deadline and release time can be used to calculate the message delay as shown in Equation 5-3.

$$delay(m_i) = d - r - w$$

Equation 5-3: Message Delay

The *BCET* is the quickest time from when a message leaves one task and the time taken until it completes execution. This includes time spent propagating through the network. Factors such as bandwidth limitations, latency in the system and jitter affect the *BCET*.

When developing automotive applications the state space is too large to completely determine all possible combinations therefore giving absolute *WCETs*. One method around this is to get end-to-end execution times for a set of scenarios or test cases. The problem with this approach is that the minimum and maximum times obtained for that particular test case will lead to an overestimation of the *BCET* and underestimation of the *WCET*, which is not safe for hard real-time systems (Reinhard Wilhelm J E, 2006).

5.9.2 Response Time Analysis (RTA)

Response Time Analysis determines if deadline times are met. Before the final system or application parameters are defined there is a method for examining if the chosen system will meet its required deadlines. This method requires the *WCRT* (Worst Case Response Time) to be calculated and then compared to the required deadline. If a pre-emptive model is used the execution time and period of each task are required. Equation 5-4, (Burns, 1994b), calculates the execution time based on the maximum interference from a higher priority task.

$$w_i^{n+1} = c_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \quad \text{Where} \quad w_i^0 = c_i$$

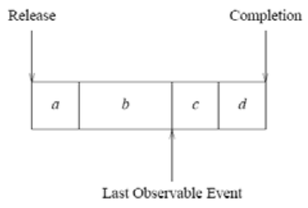
Equation 5-4: Recurrence Response Time

Where *hp(i)* is the set of all higher priority tasks than task *i*. Task *i* is delayed by execution time C_j where *j* is a periodic task. The periodic interval time is T_j.
To get an indication of whether the tasks meet the system constraints firstly sum the response times and then subtract the results from the deadline constraint (Oshana, 2007a).

In this framework the response time of a message is crucial in determining if the tasks and applications meet their required deadlines. While tools are available that determines the *WCRT* of a message this research did not have the appropriated budget or access to these tools so the equations used in developing these tools were used. In 1994 schedulability analysis was developed for CAN showing *WCRTs* (Robert I. Davis, 2007). In 1995 work by Tindell and Burns was adopted by automotive manufacturer Volvo Car Corporation. Using these works, configuration and schedulability analysis was successfully carried out and implemented on the CAN bus in the Volvo S80. This lead to the release of Volcano Network Architect by Volcano Communications Technologies AB (Robert I. Davis, 2007). This allowed improvements to scheduling as message timings could now be guaranteed. The methods previously used to calculate *WCRTs* were underutilising bus bandwidths. Using a systematic approach, bandwidth utilisation to increased from the 30-40% mark to 80% (Robert I. Davis, 2007).

Works by(Burns, 1994a),(Burns, 1994b) ,(Robert I. Davis, 2007) and (Tindell, 1994) are used primarily in this framework in abstracting the necessary equations before obtaining *WCRTs*.

In (Burns, 1994b) the author presents four phases of a tasks execution as illustrated in Figure 5-14. Here *a* is the initial context switch, *b* is the Tasks Actual Execution, *c* is the internal actions after the last observable event and *d* is the context switch away form the task. The author explains that utilisation analysis is more appropriate in cases where the task set conforms exactly to the rate monotonic model. If the deadline is less than or equal to the period Equation 5-5 can be used.



$$R_i = C_i + B_i + I_i$$

Equation 5-5: Response Time

Where *B* is the blocking time, *C* is the computational time, *I* is the interference. The author provides equations to calculate computational deadlines, periods and multiple iterations. The author specifically assumes no jitter in the test system and therefore does not provide for it in any of the calculations. However in (Burns, 1994b) the author discusses message queuing jitter as also discussed by (HerKert, 1996). This led to a proposed WCRT as in Equation 5-6. Here J_m is the queuing jitter of a message, w_m is the worst case delay of a message (due to higher priority messages pre-empting and lower priority message already on the bus) and C_m is the time taken to physically send the message onto the bus.

$$Rt_m = J_m + w_m + C_m$$

Equation 5-6: Response Time (Including Jitter)

C_m is presented in Equation 5-7. Here s_m denotes the bounded size of messages in bytes and τ_{bit} is the bus bit time. The constants and coefficients in equation 5-7 are presented by (Burns, 1994b).

$$C_m = \left(\left\lfloor \frac{34 + 8s_m}{5} \right\rfloor + 47 + 8s_m \right) \tau_{bit}$$

Equation 5-7: Communication Time

B_m is given by Equation 5-8 where *lp(m)* is the set of lower priority messages.

$$B_m = \max_{\forall j \in hp(m)} (c_k)$$

Equation 5-8: Blocking Delay

The term w_m is presented as in Equation 5-9 but is rewritten as in Equation 5-10 to allow for a recurrence relation due to there being no simple method of presenting in terms of w_m.

$$w_m = B_m + \sum_{\forall j \in hp(m)} \left\lceil \frac{w_m + j_j + \tau_{bit}}{T_j} \right\rceil C_j$$

Equation 5-9: Queuing Delay

$$w_m^{n+1} = B_m + \sum_{\forall j \in hp(m)} \left\lceil \frac{w_m^n + j_j + \tau_{bit}}{T_j} \right\rceil C_j$$

Equation 5-10: Reoccurrence Queuing Delay

Where *hp(m)* is the set of messages with a higher priority than *m*, T_j is the period of message j. B_m is the longest time a messages can be delayed by a lower priority message and is defined by Equation 5-8. A value of zero can be used for w_m^0. Iteration proceeds until Equation 5-11 is satisfied

$$w_m^{n+1} = w_m^n$$

Equation 5-11: Convergence

In (Tindell, 1994) the author presents the worst case response time as per Equation 5-12. Here C_i is the worst case computation time of a task, *hp(i)* is the set of tasks with a higher priority than task *i*. T_j is the minimum time between successive arrivals of task *j*. B_i is the longest time that task *i* could spend blocked by a lower priority task.

$$r_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{T_j} \right\rceil C_j$$

Equation 5-12: Response Time

For the recurrence relation Equation 5-13 illustrates this. Again, as in previous works, zero is a suitable value for r_i.

$$r_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j$$

Equation 5-13: Recurrence Response

The author includes jitter values in these equations and the assumption that priorities are unique is made here also. Equation 5-13 will guarantee convergence if processor utilisation is less than 100% and the task deadline is less than the period for situations where the task is not released immediately and is placed in a priority order run queue. Equation 5-14 can be used where w_i is given by Equation 5-15.

$$r_i = J_i + w_i$$

Equation 5-14: Updated Response Time

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i}{T_j} \right\rceil C_j$$

Equation 5-15: Worse Case Delay

In Equation 5-15 the term J_i is the worst case delay between a task arriving and being released and is termed release jitter. This can be an issue if the worst case time between successive releases is shorter than the worst case time between arrivals. To account for a later release of a task being delayed by non-completion of an earlier release, the time spent in the run queue must be less than the task period. Applying this to Equations 5-14 and 5-15 results in Equations 5-16 and 5-17 respectively.

$$r_i = \max_{q=0,1,2\ldots} (J_i + w_i(q) - qT_i)$$

Equation 5-16: Updated *WCRT*

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil C_j$$

Equation 5-17: Updated Delay

Previous Equations, 5-16 and 5-17 are guaranteed to produce a result if utilisation is less than or equal to 100%.

In (Robert I. Davis, 2007) the authors determine Cm as per Equation 5-18 where the maximum bit stuffing is assumed. Here g is a value of 34 which represents the 11-bit identifier format (it would be a value of 54 for the 29-bit identifier format). The term τ_{bit} is the time for 1 bit on the bus.

$$C_m = (g + 8s_m + 13 + \left\lfloor \frac{g + 8s_m - 1}{4} \right\rfloor)\tau_{bit}$$

Equation 5-18: Transmission Time

Equation 5.18 is simplified to Equation 5.19.

$$C_m = (55 + 10s_m)\tau_{bit}$$

Equation 5-19: Simplified Transmission Time

In (Robert I. Davis, 2007), the authors assume that communication is attained through a CAN bus and that the highest priority message queued at a node is entered into arbitration. The system is assumed to contain a static set of hard real-time messages, where each message has a fixed identifier and hence a unique priority. Each message has the maximum number of data bytes s_m and a maximum transmission time C_m. The authors define the response time of a message as

"The longest time from the initiating event occurring to the message being received by the nodes that require it".

A message is said to be schedulable if its response time is less than its deadline. The system is then said to be schedulable if all the messages in the system are schedulable. The authors use the same response times in their equations as stated previously in Equations 5-6 and 5-12.

5.10 Design Process

When designing an automotive system one of the first tasks undertaken is to deciding on the design process required. Two methods available are heuristic and algorithmic design. Both these methods can provide a suitable solution but there are different approaches in each method. Each process will take a certain amount of time before a valid answer is extracted. The designer has to decide which solution best optimises the time taken to develop a solution. Some problems do not have any possible utilisable outputs. Others are only fully solvable using optimised results if an appropriate method is used in the design approach. These problems are called NP problems and different classifications of them are explained in the following section. Heuristic and Genetic algorithms are also discussed in the following section.

5.10.1 NP Problem

The NP in a NP problem stands for Non-deterministic Polynomial. For a problem to be classified as NP it has to be solvable in polynomial time by a non-deterministic Turing Machine (Weisstein, 2008). A problem can also be considered to be NP if its solution can be guessed and it is deemed non-deterministic because there is no particular method to guessing. The non-deterministic Turing machine is a computational device and can be simulated in C++ or other languages. It takes in many paths and computes them to obtain a result. To study a problem for NP completeness the input n has to be the number of bits required to encode the input. It is also assumed that characters and numbers in the input are encoded using a reasonable binary encoding scheme so that each character uses a constant number of bits and each integer $M>0$ is represented with at most $c\log M$ bits, for a constant $c>0$ (Michael T. Goodrich, 2002).

A problem can be determined NP complete, NP hard and P. A problem is NP hard if the algorithm for solving it can be easily converted to solve any other NP problem. NP hard means *"at least as hard as any NP problem"* (Weisstein, 2009). A problem that is NP hard is considered to be NP complete. A problem might not have an efficient solving algorithm if it is NP complete. The general bin packing problem is known to be NP complete (Nossal, 1996). Each NP complete problem is as difficult as another. Some of the classical examples are satisfyability, vertex cover, knapsack and travelling salesperson.

5.10.2 Heuristics

Taking a heuristics design approach involves using experience gained when making decisions.

A heuristic approach can be taken during the design process that has multiple possible solutions. The designer then selects a solution that best suits and then recreates the process to find a new set of solutions. By continuing the process this converges to an optimal result. This approach does not have a formal method or step approach. This process is appropriate when searching for an optimal solution and is used for mixed systems and single structured systems. An example of a mixed system would be FlexRay or TTCAN where the parameters of either of the system can change (e.g. timing and segment sizes) which will give a different solution each time(Skiena, 1997). A single structured system is CAN, LIN and MOST to name but three examples.

5.10.3 Algorithms

An algorithm is a procedure for solving a problem. Using an algorithmic approach in the design process follows a set of predefined steps. In this approach there is a definitive result. The problem to be scheduled would have its characteristics and parameters defined which creates some restrictions in the design process.

Algorithm design process can be summarised in the following steps (Turner, 1996)

1. Clarify the problem; Note any assumptions and simplifications. Constantly restated the problem

Formatted: Bullets and Numbering

2. Define inputs; to set out what constitutes allowable inputs and what data types these are
3. Define outputs; State what expected data types are and their format
4. Write "first pass" algorithm solution; this will show up any issues overlooked and ambiguities of language. Write down the improved version and re-do process as necessary
5. Implement solution in code; here any programming technicalities will show up.
6. Upon successful error free compilation compare results with those expected
7. Comprehensive test process; devise a comprehensive test procedure that covers as many possible solution scenarios

The algorithm method can also be used as an optimisation approach depending on the rigidness of the parameter constraints and repeatedly re-inputting the solution set.

5.10.4 Genetic Algorithm (GA)

Genetic Algorithms use a stochastic and heuristic method to give a mutated solution. This mutated solution is then applied to a fitness function to give an optimal solution set.

GAs (Genetic Algorithms) are easiest explained using the concept of natural genetics whereby in the majority of cases only the 'strongest, fittest and smartest' survive. This produces offspring better than the previous generations. GAs have been successfully applied to optimisation problems like wire routing, scheduling adaptive control, game playing, cognitive modelling, transportation problems, travelling salesman problems, optimal control problems, database query optimisation etc (Michalewicz, 1996). The genetic algorithm itself does not contain any application specific parameters; these are not included until the fitness function is developed. The fitness function results can then be fed back into the GA for further optimisation of the results. GAs should not be purely considered optimisation algorithms. An example of this is in evolution; sometimes luck is involved in the survival process. Similar to genetic algorithms, an optimisation solution and a non-optimal solution could be accepted.

5.11 Conclusion

This chapter begins by discussing distributed architectures and basic node design. Next there is a brief discussion on RTOS and the OSEK/VDX which is a standard developed by European automobile manufacturers. Some scheduling policies used in real-time embedded systems are presented. The history of CPA and some associated parameters are introduced. This leads into task graphs which are introduced as a method of visually representing tasks and application data. From the task graphs *BCET*, *WCET* and *RTA* parameters can be derived. Next the type of NP problem can be diagnosed to help determine the design approach. The Heuristic method, algorithmic and GA approaches to design have been discussed. This chapter gives an overview into the types of approaches used in this research and similar work carried out that has been reviewed during the course of this research.
After review scheduling methods and design processes used by other authors, the strengths and weaknesses of each approach can be assessed. Using task graph analysis allows the application undergoing migration to be assigned into its component sets. This allows any slack in the system to be redistributed among other tasks in the application. Task graph analysis is a critical component in the abstraction of CAN and FlexRay parameters.
After examining design processes in section 5.9 it is apparent that none of these will fit directly into my application test cases. One factor for this is that these design process were designed for test cases carried out using simulation and not in real-time. Some aspects of the design processes can be incorporated into the application test case e.g. algorithm approach allowing the steps in the design process be defined.

The design approaches taken in the automotive industry have also been used in other industry sectors such data networks, telecommunications and industrial computing (real-time systems) (Vasilios Pasias, 2006).

5.12 References

ABHIJIT DAVARE, Q. Z., MARCO DI NATALE, CLAUDIO PINELLO, SRI KANAJAN, ALBERTO SANGIOVANNI-VINCENTELLI (2007) 'Period Optimisation for Hard Real-Time Distributed Automotive Systems' DAC 2007, San Diego, Calafornia, USA, ACM

ALOUL, N. K. A. F. (2005) 'The Synthesis of Dependable Communication Networks for Automotive Systems' SAE Internalional 2005,

ANDREI HAGIESCU, U. D. B., SAMARJIT CHAKRABORTY, PRAHLADAVARADAN SAMPATH, P.VIGNESH, V. GANESAN, S. RAMESH (2007) Performance Analysis of FlexRay-based ECU Networks. ACM.

BURNS, A. (1994a) Fixed priority scheduling with deadlines prior to completion

BURNS, K. T. A. A. (1994b) Guaranteeing Message Latencies on Control Area Network (CAN)

BUTTAZZO, E. B. A. G. C. (2004) Schedulability Analysis of Periodic Fixed Priority Systems. IEEE.

CAREY, J. R. H. M. L. M. J. (1991) Earliest Deadline Scheduling for Real-Time Database Systems. IEEE.

E. DOUGLAS JENSEN, C. D. L., HIDEYUKI TOKUDA (1985) A Time-Driven Scheduling Model for Real-Time Operating Systems. IEEE.

ENGLANDER, I. (2000) *The Architecture of Computer Hardware and Systems Software*, Wiley.

GORDON, K. L. A. J. (1991) *Critical Path Analysis-and other project network techniques*, The Bath Press.

HERKERT, K.-J. L. A. A. (1996) 'Jitter Control in Time-Triggered Systems' International Conference on System Sciences, Hawaii, IEEE

INFORMATIK, V. (2008) Solutions for OSEK/VDX,20/02/2008

JAKOB ENGBLOM, A. E., FRIEDHELM STAPPERT (2000) 'Structured Testing of Worst-Case Execution Time Analysis Methods' Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, USA, IAR Systems

JORDAN, V. P. H. A. H. F. (2004) *Computer Systems Design and Architecture*, Pearson Prentice Hall.

LEMIEUX, J. (2001) *Programming in the OSEK/VDX Environment*, CMP.

LOBUR, L. N. A. J. (2003) *Computer Organization and Architecture*, Jones and Bartlett Publishers.

LOCKYER, K. G. (1974) *An introduction to Critical Path Analysis*, Pitman Publishing.

LOCKYER, K. G. (1991) *Critical path analysis and other project network techniques.*, Pitman.

MICHAEL T. GOODRICH, R. T. (2002) *Algorithm Design*, Wiley.

MICHALEWICZ, Z. (1996) *Genetic Algorithms + data Structures = Evolution programs*, Springer.

NATALE, M. D. (2000) 'Scheduling the CAN bus with Earliest Deadline Techniques' Proceedings of the The 21st IEEE Real-Time Systems Symposium (RTSS’00),

NOSSAL, R. (1996) Pre-Runtime Planning of a Reliable Real-Time Communication System. Institut fur Technische Informatik, Technichal University of Vienna.
OSEK (2005) Operating System Specification 2.2.3.
OSHANA, R. (2007a) Real-Time Operating Systems for DSP.
OSHANA, R. (2007b) Real-Time Operating Systems for DSP part 6. DSP Design Line.
REINHARD WILHELM J E, A. E., NIKLAS HOLSTI, STEPHAN THESING, DAVID WHALLEY, GUILLEM BERNAT, CHRISTIAN FERDINAND, REINHOLD HECKMANN, TULIKA MITRA, FRANK MUELLER, ISABELLE PUAUT, PETER PUSCHNER, JAN STASCHULAT, PER STENSTR¨OM (2006) 'The Worst-Case Execution Time Problem— Overview of Methods and Survey of Tools' ACM Transactions on Embedded Computing Systems,
REYNAUD, C. B. (1970) *The Critical Path*, George Goodwin Limited.
ROBERT I. DAVIS, A. B., REINDER J. BRIL AND JOHAN J. LUKKIEN (2007) Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. Springer Science + Business Media.
SCHAFFER, P. N. L. A. J. (2006) Exactly When Do You Need Real Time?20/03/2006
SHA, L., KLEIN, MARK H., AND GOODENOUGH, JOHN B (1991) Rate Monotonic Analysis.
SHIN, K. M. Z. A. K. G. (1995) 'Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time control Applications' Proceedings of the Real-Time Technology and Applications Symposium (RTAS ’95),
SKIENA, S. (1997) How to Design Algorithms. Department of Computer Science
SUNY Stony Brook.
TANENBAUM, A. S. (1996) *Computer Networks*, prentice-Hall Inc.
TINDELL, K. (1994) Holistic Schedulability Analysis for Distributed Hard Real-Time Systems.
TRAIAN POP, P. P., PETRU ELES, ZEBO PENG (2007) 'Bus Access Optimisation for FlexRay-based Distributed Embedded Systems' EDAA,
TURNER, D. B. A. J. (1996) *Understanding Algorithms and Data Structures*, McGraw Hill.
VASILIOS PASIAS, D. A. K. A. R. C. P. (2006) 'Heuristic Algorithms for Efficient Wireless Multimedia Network Design' IEEE, Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06,
VIKRAM ADVE, R. S. (2006) Compiler Synthesis of Task Graphs for Parallel Program Performance Prediction,10/03/2008
WEISSTEIN, E. (2008) Mathworld.com,18/02/2008
WEISSTEIN, E. W. (2009) NP-Problem,12/01/2009

6 Automotive Network Migration:

6.1 Automotive Network Migration

Migration (at hardware level, software level or both) is a change from one system to another. Methods of analysing an embedded system at task and message level in conjunction with developing timing constraints were examined in the previous chapter. The implementation of these constraints on the chosen protocol is critical to achieving a successful migration. This chapter discusses two approaches to achieving successful migration; gateways and full conversion methods. Both methods are discussed with reference to previous works by different authors.

6.2 Introduction

Automotive network migration at the application level, involves changing from one protocol and successfully executing on a different protocol without compromising functionality or practicality.

When undertaking the migration process key parameters such as timing analysis, predictability analysis and optimisation procedures were examined in works carried out by (Shan Ding, 2005), (Aloul, 2005) and (Traian Pop, 2007). Some of the potential methods for undertaking these procedures were discussed in Chapter 5.

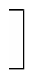
6.3 Protocol Migration Requirements

Migration to a multi protocol system is required if no single protocol is able meet the designer and manufacturer’s requirements such as equipment costs, bandwidth requirements and determinism. If a specific protocol is unable to meet the specified requirements but an alternative protocol allows these specified requirements to be met, then migration can also take place between these two protocols. In this second instance it is the network component that is changed.

In order to successfully convert from one protocol to another there has to be some common functionality in terms of the protocol (Lam, 1989) as illustrated in Figure 6-1. Here node *X1* communicates with node *X2* using protocol *X* and similarly node *Y1* communicates with node *Y2* using a different protocol *Y*. If *X1* needs to communicate with *Y2* it is unable due to no common functionality existing.

ɔnality

An example of this common functionality is some form of protocol converter as illustrated in Figure 6-2 where *C* is the converter.



/o

One reason for a lack of compatibility between protocols is the need to improve functionality as new protocols replace older ones. Older protocols are sacrificed for superior quality (Lam, 1989). An example of this is a LIN network beside a CAN network as discussed in Chapter 3. This placement coupled with different protocols being more

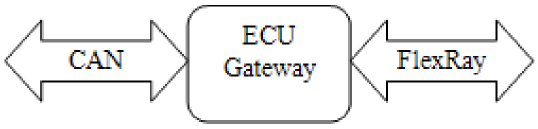
efficient at transferring different types of information leads to numerous protocols potentially operating in a single system.
When a migration takes place it is insufficient to map a single message from one protocol to the new protocol. A sequence of events such as tasks in one protocol should be mapped so as to attain proper flow, no undue termination, no deadlock, no unexpected reception or errors (ZP. Tao, 1992).

There are two methods for network migration, these are;

- Side-by-Side Migration
- Full Conversion

6.4 Side-by-Side Migration (Using a Gateway)

A gateway enables migration of necessary data and parameters from one protocol to another (Suk-Hyun Seol, 2006).
In the side-by-side method there are at least two different protocols in operation on either side of a gateway. The gateway is used to carry out the conversion process of the protocol's parameters. A gateway can carry out the conversion of more than one protocol.
Data transfer takes place through an ECU. This ECU is called a gateway as illustrated in Figure 6-3. Communication occurs in a single direction or in both directions as illustrated in Figure 6-3.



6.4.1 Gateways

A gateway allows separate protocols to engage with each other. This enables the transfer of data between different networks. Due to the variety of networks available when developing a gateway the system designer must consider the following questions (Smith, 2005)

- What applications will be used?
- Which networks need to be bridged?
- What is the bridge topology?
- Is DMA (Direct Memory Access) required?
- What size should the data buffers be?
- What bus should be used for internal transfer?
- How wide should this be?
- What arbitration mechanism should be employed?
- How much processing power is required?

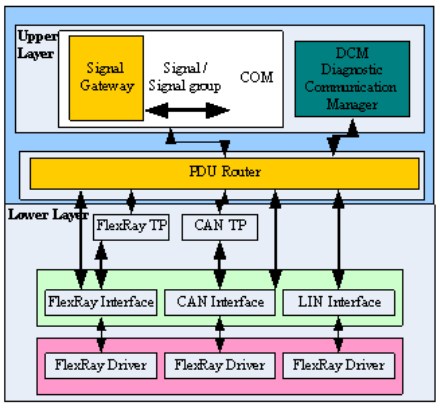
The question, “Which networks need to be bridged?” is a complex question in its own right. To take the example of CAN and FlexRay (as per Figure 6-3) they both have different payloads; CAN 8 bytes max, FlexRay 254 bytes max, thereby requiring a gateway able to handle both these payload ranges. The CAN data would need to be buffered up to a larger data rate but this would lead to jittering delays while the information is being buffered so there are trade offs to be considered.

Ideally the processors primary objective is to keep processing; the DMA feature can accordingly be included. DMA can deal with data transfers between the gateway and memory interfaces leaving the processor to deal with processing duties.

The AUTOSAR (Automotive Open Systems Architecture) partnership consisting of leading OEMs (Original Equipment Manufacturers) and tier one suppliers defines a gateway structure (AUTOSAR, 2007).

6.4.2 AUTOSAR Gateway Structure

AUTOSAR defines the basic gateway structure as illustrated in Figure 6-4. The upper layer contains the communication parameters for the exchange of data between ECUs. The lower layer contains the interfaces and the drivers for the protocols. A router connects the upper layer and lower layer, and communication (COM) is contained in the upper layer. COM is a method for exchanging data between different tasks on an ECU or on multiple ECUs (Jackman, 2007).

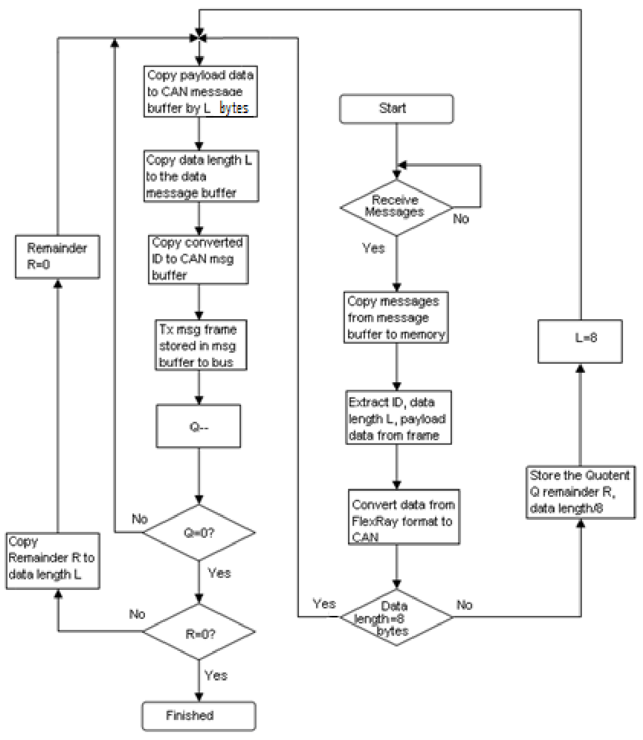


Using the side-by-side migration example, the characteristics of the first protocol are maintained until such time as the conversion takes place. The features of the second protocol cannot be utilised until after the conversion has taken place. For example using a gateway to convert from CAN to FlexRay (as illustrated in Figure 6-3), some features of CAN (e.g. being a pure event-triggered system) and some features of FlexRay (e.g. determinism and dual channels), are unable be used on both protocols.
Because CAN is a mature protocol in comparison to FlexRay automotive manufacturers might prefer to work with CAN as much as possible (if previous developments using the CAN protocol were undertaken), and switch to FlexRay when deemed completely necessary to take advantage of some of its properties as discussed in Chapter 4. In (A. Albert, 2003) a gateway is used to convert received CAN data to a TTCAN network for use in a vehicle dynamic management system which contained; an electronic stability

program, active front steering and an electronic active roll stabiliser. Off the shelf gateways can be purchased from companies such as NEC (GmbH, 2007) and TZM (Mikroelektronik, 2008) to name but two.
Examples of different functional gateways are; in-vehicle, inter-vehicle and vehicle-to-infrastructure communication. Examples of each of the above are infotainment systems, live traffic and travel information and remote diagnostics for more efficient breakdown assistance (Guido Gelen, 2006).

Specific migration issues involved in migration of the FlexRay protocol data and parameters to the CAN protocol are illustrated in Figure 6-5.

6.4.3 FlexRay to CAN Migration


Start
Receive Messages
No
Yes
Copy messages from message buffer to memory
Extract ID, data length L, payload data from frame
Convert data from FlexRay format to CAN
Data length=8 bytes
Yes
No
Store the Quotient Q remainder R, data length/8
L=8
Copy payload data to CAN message buffer by L bytes
Copy data length L to the data message buffer
Copy converted ID to CAN msg buffer
Tx msg frame stored in msg buffer to bus
Q--
Q=0?
No
Yes
R=0?
No
Yes
Finished
Copy Remainder R to data length L
Remainder R=0

In this section issues such as extracting necessary data for the FlexRay protocol and configuration for the correct transmission in the CAN protocol are examined.

From Figure 6-5 (Suk-Hyun Seol, 2006) message transfer starts when the first message buffer is full until the data is transferred to memory. Then the data length, ID and payload are extracted from the message frame. Subsequently the ID field is converted from a FlexRay 12 bit field to a CAN 29 bit ID field. The data length is then split in lengths of 8 byte segments. After that the ID, data length and payload data are copied to the host's message buffers from the queue. Each time a message is taken from the queue a counter is decremented and any time a message is put into the queue the counter is incremented. Finally the message frames are transmitted on the CAN bus (Suk-Hyun Seol, 2006).

The above format deals with converting from a TT protocol to an ET protocol. Other research has dealt with converting different TT protocols. (Shehryar Shaheen, 2006) developed a gateway for TT control networks. These included FlexRay, Byteflight, TTP/C and TTCAN. In this approach the packet routing architecture and message queue format were abstracted from conventional gateway design methods. Other issues were encountered that would not occur on general data communications networks such as timely reliable delivery of all message frames across the gateway (Shehryar Shaheen, 2006). This is due to the different network characteristics of a TT network from an ET network.

6.5 Full Migration

The full conversion method implies complete migration from one protocol to the new protocol. For a full conversion to be deemed successful, all necessary parameters that were fulfilled in the old system need to be at least met if not improved in the new system. For example, it would not make sense to have safety critical applications on the CAN network when they were previously on the FlexRay network, if the characteristics of the FlexRay protocol were critical to successful implementation of the application.

6.5.1 Migration to a TT (Time-Triggered) Protocol

Many authors have researched the area of migrating to a TT protocol including (Kanchi, 2007), (Wei Zheng, 2005), (Aloul, 2005),(Andrei Hagiescu, 2007), (Traian Pop, 2007).

Some protocols are easier to migrate to a TT system than others. One such case is where the initial protocol and the new protocol contain common properties. An example of this case is a migrating from TTCAN to FlexRay. Both of these protocols contain TT and ET properties. Migrating from an ET system to a TT system requires more planning (than from TT to ET) because a schedule table needs to be developed prior to run-time. This is achieved in (Suri, 2004) where the authors use the TTP/C protocol as a base and sporadic data transfer is included in the modified frame structure. Migration is accomplished through the development of a Sporadic Information Transfer (SIT) bit in the frame header file. This SIT bit can either send sporadic data or request additional slots to transmit sporadic data. Small amounts of sporadic data are transmitted in the SIT section of the frame but if large amounts of sporadic data are required they will be sent in the requested additional slots.

6.5.2 Migration to an ET (Event-Triggered) Protocol

One of the more difficult systems to migrate is a TT protocol system onto an ET protocol system. There is no guarantee that deadlines will be met due to the characteristics of both domains.

It is not enough to just make sure deadlines from one protocol are met in the new protocol. Maximum utilisation of the frame is another critical objective in achieving a successful migration. In the above example since the payloads of both protocols are known it is possible to determine that one TTCAN frame of 8 bytes (max size) will fit in a FlexRay frame of 127 two-word bytes (maximum size). If more than one TTCAN frame is accommodated in the FlexRay frame, bandwidth utilisation is increased. The major stumbling block to doing this is optimising the frame sizes. This is achieved using optimisation and best fit algorithms as discussed in (Abhijit Davare, 2007) and (Traian Pop, 2003) as presented in Chapter 5.

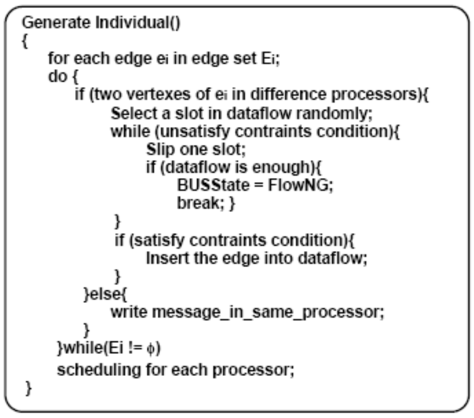
In (Andrei Hagiescu, 2007) the authors migrate to the DYN (dynamic) domain in the FlexRay network. The authors commence by declaring that the worst-case end-to-end delay values are more appropriate to calculate. This is due to the possibility that as a signal propagates through a system it will possibly get modified by scheduling policies it encounters in the system. An example of this is a message originating from a sensor passing over multiple ECUs and activating an actuator. The authors take a framework by (S. Chakraborty, 2003) and modify it to model the FlexRay protocol. Multiple ECUs are modelled on the FlexRay bus with application tasks mapped onto the ECUs. The worst-case length of specified tasks is a constant. The framework developed allows DYN messages be transmitted over two cycles allowing messages larger than the DYN domain to be transmitted.

6.5.3 Migration to a Mixed System

When migrating from an ET or TT system to a mixed system the designer determines where each segment domain transfers to in the new protocol. This could involve a direct transfer to its equivalent domain (static or dynamic) in the mixed system or to the alternative domain in the mixed system (static or dynamic). Earlier work has been carried out migrating to a mixed system but the designer does not fully utilise both TT and ET aspects of these systems. (Shan Ding, 2005) and (Cummings, 2008) are examples of this as explained in sections 6.5.4 and 6.5.5 respectively.

6.5.4 GA (Genetic Algorithm) Approach

In (Ding Shan, 2005) the author restricts migration to only the static segment of a mixed system protocol using a GA static scheduling method. (Shan Ding, 2005) presents an application representing a set of task graphs G_i containing tasks T_i and edges connecting the tasks E_i. Each node contains known worst case execution times C_i, periods T_i and a deadline D_i. The author uses the constraints; Response Time (RT), Freshness Time (FT), Maximum Response, Maximum Freshness Time, Response and Freshness Constraint, Input and Output Constraint and Slot Redundancy to develop "an individual" algorithm. This "individual" algorithm is presented in Figure 6-6.



```
Generate Individual()
{
    for each edge ei in edge set Ei;
    do {
        if (two vertexes of ei in difference processors){
            Select a slot in dataflow randomly;
            while (unsatisfy contraints condition){
                Slip one slot;
                if (dataflow is enough){
                    BUSState = FlowNG;
                    break; }
            }
            if (satisfy contraints condition){
                Insert the edge into dataflow;
            }
        }else{
            write message_in_same_processor;
        }
    }while(Ei != ϕ)
    scheduling for each processor;
}
```

Figure 6-6: Algorithm for generating an individual (Shan Ding, 2005)

The results obtained are optimised by selecting routes according to their fitness then applying crossover and mutation.

(Nossal, 1996) also presents a GA that generates a static schedule for bus access to the TTP (Time-Triggered Protocol). The GA includes a fitness function. The author presents the planning method (heuristic based GA) in developing an algorithm that is then applied to the chosen TDMA protocol. Below is a list of constraints from the communication system that are necessary for the planning algorithm to complete. Two requirements to be fulfilled are requirements by the application and requirements by the protocol. The message size is calculated in Equation 6-1 and the maximum message size is calculated from Equation 6-2. Where *M* is the bus message, *i* is the node, *j* is the TDMA cycle, s_d is the size in bits and *d* is the data element.

$$ms_{ij} = \left[\sum_{d \in M_{ij}} s_d \right] [bit]^2$$

Equation 6-1: Used message size of node *i*

$$MS_i = \max_{1 \le j \le C} \{ ms_{ij} \} [bit]$$

Equation 6-2: Maximum message size of node *i*

The optimum period is calculated from Equation 6-3 where o_d is the optimum period, *rn* is the receiving node and *l* is the maximum latency from the sending node to the receiving node. RN_d is the receiving node's communication latencies, rp_{rn} is the smallest period of any task receiving data element *d* at node *rn*.

$$o_d = \frac{\min}{(rn, l) \in RN_d} \left\{ l - \min\left\{ p_{d, rp_{rn}} \right\} \right\}$$

Equation 6-3: Data element optimum period

Next the fitness function is applied to check the validity of the schedule. An optimum ratio of 1 (actual period and maximum period) is desirable. A ratio greater than 1 should be avoided as it violates the applications temporal requirements such as scheduled periods larger than the maximum period. A ratio less than 1 means the data element is transmitted more often than required and therefore would constitute a waste of bandwidth.

The overall fitness function is defined by Equation 6-4. For a complete derivation see (Nossal, 1996), where *D* is the number of data elements.

$$F = \frac{\sum_{d=1}^{D} f\left(\frac{a_{d * t_{TDMA}}}{p_d}\right)}{D}$$

Equation 6-4: Overall Fitness

Both of these GA scheduling methods do not take account of scheduling aperiodic signals. This will lead to under utilisation of the bandwidth in a protocol that contains TT and ET domains.

6.5.5 Other Approaches

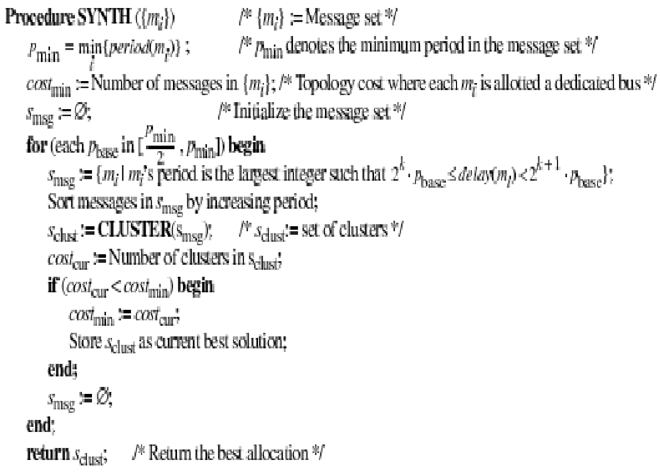
Other approaches previously taken are discussed in this section such as the Heuristic design method and other developer specific paradigms. There is a commonality between them in that all methods develop schedulability analysis and optimisation techniques. In (Traian Pop, 2002), (Traian Pop, 2003), (Traian Pop, 2006), (Traian Pop, 2007), (Jan Carlson, 2003) the authors take the heuristic approach.

(Aloul, 2005) synthesises each application task to a scheduled message level using only a TDMA protocol. Task graphs are used to develop message clustering once the parameters such as deadline d_i, release time r_i, execution time c_i and task T_i are known. Slack is calculated and distributed among the tasks to obtain a worst-case response time for each task w_i. A message delay *delay(m_i)* is calculated as presented in Equation 5-3 in Chapter 5. Once the bandwidth is known a slot size can be calculated from Equation 6-5.

$$\Delta_{slot} = \min_i \{size(m_i)\} / B_j^{speed} \ \mu s$$

Equation 6-5: Slot Duration

An algorithm is then developed to first synthesise (Figure 6-7) the network topology and then to cluster the network topology. A message period equal to a harmonic multiple is required to enable scheduling of multiple task graphs with different periods.



```
Procedure SYNTH ($\{m_i\}$)          /* $\{m_i\}$ := Message set */
  $p_{\min} = \min_i\{period(m_i)\}$ ;          /* $p_{\min}$ denotes the minimum period in the message set */
  $cost_{\min}$ := Number of messages in $\{m_i\}$; /* Topology cost where each $m_i$ is allotted a dedicated bus */
  $s_{msg} := \emptyset$;          /* Initialize the message set */
  for (each $p_{base}$ in $[\frac{p_{\min}}{2}, p_{\min}]$) begin
    $s_{msg} := \{m_i \mid m_i$'s period is the largest integer such that $2^k \cdot p_{base} \le delay(m_i) < 2^{k+1} \cdot p_{base}\}$;
    Sort messages in $s_{msg}$ by increasing period;
    $s_{clust}$ := CLUSTER($s_{msg}$);     /* $s_{clust}$ := set of clusters */
    $cost_{cur}$ := Number of clusters in $s_{clust}$;
    if ($cost_{cur} < cost_{\min}$) begin
      $cost_{\min} := cost_{cur}$;
      Store $s_{clust}$ as current best solution;
    end;
    $s_{msg} := \emptyset$;
  end;
  return $s_{clust}$;     /* Return the best allocation */
```

Figure 6-7: Network Topology Synthesis Algorithm (Aloul, 2005)

Transmission slot reuse is applied to increase bandwidth utilisation. This is achieved through Equations 6-6 and 6-7.

$$n_{reuse} = \sum_{m_i} n_i - \sum_{m_i} \left\lceil \frac{period(m_{new})}{arrival(m_i)} \right\rceil . n_i$$

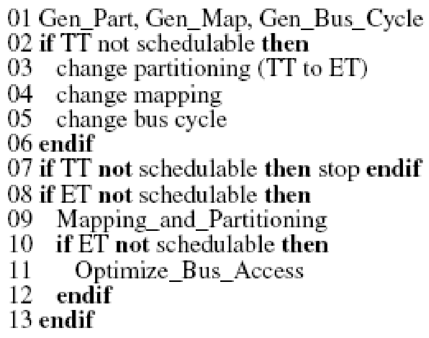
Equation 6-6: Slot Reuse

$$\left\lceil \frac{size(m_{new})}{B_j^{speed} . \Delta_{slot}} \right\rceil - n_{reuse}$$

Equation 6-7: m_{new} transmission slot portion within *period(m_{new})*

However in (Aloul, 2005) no mention is made of using an ET protocol or mixed system which increases bandwidth utilisation when compare to pure TT systems (as discussed in Chapter 2).

In (Traian Pop, 2002) the authors develop a design heuristic algorithm for mixed time/event triggered distributed embedded systems as presented in Figure 6-8.

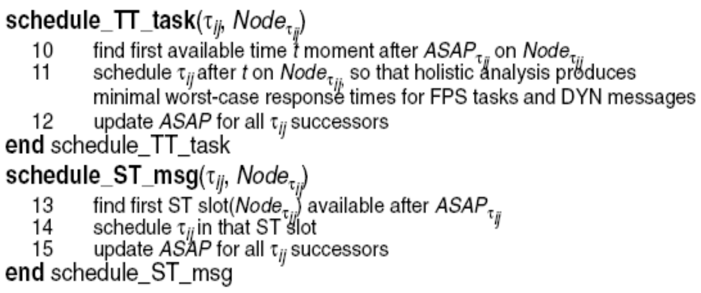


```
Gen_Part, Gen_Map, Gen_Bus_Cycle
if TT not schedulable then
   change partitioning (TT to ET)
   change mapping
   change bus cycle
endif
if TT not schedulable then stop endif
if ET not schedulable then
   Mapping_and_Partitioning
   if ET not schedulable then
      Optimize_Bus_Access
   endif
endif
```

This design heuristic involves three primary steps;

- Build initial configuration
- Mapping and partitioning
- Bus cycle optimisation

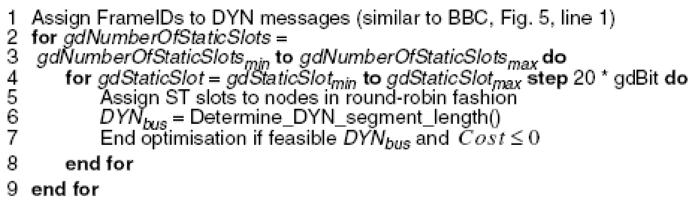
In (Traian Pop, 2006) the authors build on their previous work to specify implementation on the FlexRay protocol. The application model was developed using acyclic task graphs and timing analysis was performed on the ST (static) and DYN (dynamic) segments. The schedulability algorithms are presented in Figure 6-9.



```
schedule_TT_task($\tau_{ij}$, $Node_{\tau_{ij}}$)
   find first available time $t$ moment after $ASAP_{\tau_{ij}}$ on $Node_{\tau_{ij}}$
   schedule $\tau_{ij}$ after $t$ on $Node_{\tau_{ij}}$, so that holistic analysis produces
   minimal worst-case response times for FPS tasks and DYN messages
   update ASAP for all $\tau_{ij}$ successors
end schedule_TT_task
schedule_ST_msg($\tau_{ij}$, $Node_{\tau_{ij}}$)
   find first ST slot($Node_{\tau_{ij}}$) available after $ASAP_{\tau_{ij}}$
   schedule $\tau_{ij}$ in that ST slot
   update ASAP for all $\tau_{ij}$ successors
end schedule_ST_msg
```

Figure 6-9: TT and DYN Schedulability Algorithms (Traian Pop, 2006)

Having prior knowledge of the scheduled tasks is required to carry out schedulability analysis of the TT domain. This means a scheduling table can be set up. The DYN domain analysis is carried out by determining the worst case response time *(WCRT)*. This includes taking into account the transmission delay caused by the transmission of higher priority tasks. Blocking by the ST domain also has to be considered. This work is improved on in (Traian Pop, 2007) where the authors propose a FlexRay bus configuration for a specific application. An algorithm is developed to optimise bus access on the FlexRay network as is illustrated in Figure 6-10.



```
Assign FrameIDs to DYN messages (similar to BBC, Fig. 5, line 1)
for gdNumberOfStaticSlots =
 $gdNumberOfStaticSlots_{min}$ to $gdNumberOfStaticSlots_{max}$ do
    for $gdStaticSlot = gdStaticSlot_{min}$ to $gdStaticSlot_{max}$ step 20 * gdBit do
        Assign ST slots to nodes in round-robin fashion
        $DYN_{bus}$ = Determine_DYN_segment_length()
        End optimisation if feasible $DYN_{bus}$ and $Cost \leq 0$
    end for
end for
```

While there are proposed techniques for looking at ET (dynamic) protocols such as CAN, these techniques are not applicable when considering the dynamic segment of the FlexRay protocol. (Valenzano, 2004) examines the Byteflight protocol but implements a quasi-TDMA transmission scheme to guarantee message transmission. This is not fully compatible to the ET nature of the dynamic segments in FlexRay. Another technique as proposed in (Oshana, 2007) involves making the dynamic segment as big as the largest dynamic message to run. This will guarantee all dynamic messages get transmitted but it is an inefficient method if the DYN messages have different periods. For example if there are 10 messages to run and 9 have periods of 1*ms* and one message has a period of 4*ms*. This results in the dynamic segment of 4*ms* being required. The predominant DYN message size being transmitted (90% of the time) is a 1ms message. This results in an inefficient allocation of bandwidth.

In (Wei Zheng, 2005) the authors focus on TT scheduling. The authors use two metrics to obtain the design goal in scheduling hard real-time distributed embedded systems.

These are extensibility and scalability. The principle of orthogonalisation is used where a functional description is first applied and then functional components are mapped to architectural components. The system is represented by direct acyclic graphs. Task parameters are consistent with other methods examined previously in this chapter such as WCET, Release Time, Deadline, Period, Start Time, and Finish Time etc. These constraints were then used to develop feasibility constraints. The schedule is considered feasible if it satisfies constraints imposed by the architecture (Wei Zheng, 2005). The inclusion of scalability and extensibility metrics allows limited design changes without modifying the existing schedule. This is achieved through adding slack into the system. The system cost function is then compared to another optimisation technique of minimising the end-to-end delay, which is then proved successful under tested conditions.

In (Traian Pop, 2006) the authors propose methods for scheduling a FlexRay communication protocol. For the ST segment the SCS (Static Cyclic Scheduling) and the FPS (Fixed Priority Scheduling) approach is used, FPS is also used in the DYN segment. The author iteratively builds up a scheduled table for SCS tasks. This leads the author to develop the scheduling algorithm presented in Figure 6-11. The authors develop the DYN segment separately based on the RTA principle discussed in Chapter 5. A maximum worst case response time is calculated using Equation 6-8.

$$Rm(t) = \sigma_m + w_m(t) + C_m$$

Equation 6-8: DYN Worse Case Response Time

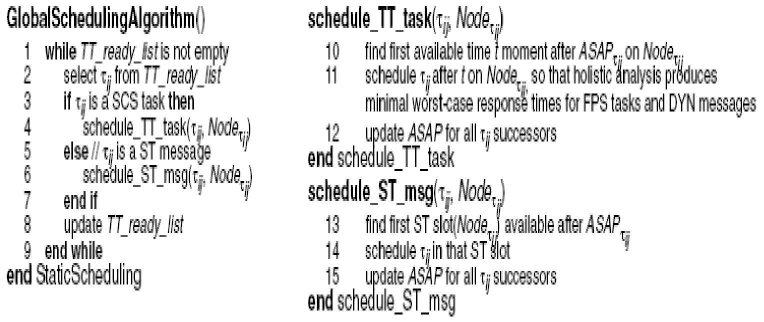
Where R_m is worst case response time, δ_m is the longest delay suffered during one bus cycle if a message is generated by sender task just after its slot has passed, w_m is the worst case delay caused by transmission of ST messages and higher priority DYN messages and C_m is the communication time on the particular bus.

Each equation segment (δ_m, w_m and C_m) is further analysed to accurately determine any issues that could arise to delay a messages response.

Equation 6-9 shows δ_m as a function of the ST segment the message's frame identifier and *gdMinislot* minus the total bus length.

$$\sigma_m = T_{bus} - (ST_{bus} + FramID_m.gdMinislot)$$

Equation 6-9: Worse case delay δ_m



```
GlobalSchedulingAlgorithm()
1  while TT_ready_list is not empty
2      select $\tau_{ij}$ from TT_ready_list
3      if $\tau_{ij}$ is a SCS task then
4          schedule_TT_task($\tau_{ij}$, $Node_{\tau_{ij}}$)
5      else // $\tau_{ij}$ is a ST message
6          schedule_ST_msg($\tau_{ij}$, $Node_{\tau_{ij}}$)
7      end if
8      update TT_ready_list
9  end while
end StaticScheduling

schedule_TT_task($\tau_{ij}$, $Node_{\tau_{ij}}$)
10     find first available time t moment after $ASAP_{\tau_{ij}}$ on $Node_{\tau_{ij}}$
11     schedule $\tau_{ij}$ after t on $Node_{\tau_{ij}}$, so that holistic analysis produces
       minimal worst-case response times for FPS tasks and DYN messages
12     update ASAP for all $\tau_{ij}$ successors
end schedule_TT_task

schedule_ST_msg($\tau_{ij}$, $Node_{\tau_{ij}}$)
13     find first ST slot($Node_{\tau_{ij}}$) available after $ASAP_{\tau_{ij}}$
14     schedule $\tau_{ij}$ in that ST slot
15     update ASAP for all $\tau_{ij}$ successors
end schedule_ST_msg
```

Figure 6-11: Global Scheduling Algorithm (Traian Pop, 2006)

Equation 6-10 illustrates the communication time C_m, where *Frame_size(m)* is the message frame size and bus_speed is the operating bandwidth speed

$$C_m = \frac{Frame\_size(m)}{bus\_speed}$$

Equation 6-10: Communication Time

w_m is the maximum amount of delay on the bus from ST messages, higher priority messages *hp(m)*, lower frame identifier messages *lf(m)* and unused DYN slots with frame identifiers lower than those currently sending, which equals one minislot *ms(m)*. *FrameID(m)* reuse is not part of the FlexRay specifications but is considered by the authors in analysing the DYN segment.

Developing these equations (6-8, 6-9 and 6-10) further an optimal solution for *BusCycles(m)* is obtained using an ILP. The authors determine the worst case delay for

BusCycles$_m$(t), produced by *lf(m,t)* and *ms(m,t)*, adding up to the delay produced by messages in *hp(m,t)*.

In (Cummings, 2008) the author presents a sample CAN network and describes how it is transitioned over to the FlexRay network. The primary criticisms of FlexRay by the author are in relation to its cost and complexity. In the sample CAN network presented the standard 11-bit identifier is chosen. The bus rate of 500Kbit/s is also applied. The author presents a table of data as illustrated in Table 6-1.

Table 6-1: Example CAN Network

Number of Frames	Frames Per Second	Average Payload	Bus Time Per Second
2	500	8.0	262ms
3	200	7.0	145.2ms
7	100	6.7	165.9ms
8	50	7.1	97.6ms
6	20	5.4	25.4ms

This example CAN network had a bus load calculated at 75%. Adding two more frames at 2*ms* periods increases bus load beyond what is available. The author deduces that it is more cost beneficial to implement FlexRay than operate two CAN networks with a gateway in between. The FlexRay network is set up to physically compare as close as possible to the CAN configuration. The FlexRay configuration parameters are;

- A single channel at 10 Mbit/s
- 2 Static Frames
- Payload of 2 Bytes
- 125 Dynamic Frames
- 1000 µs Cycle Time

Two static frames are included as FlexRay requires two synchronisation frames for start up. The designer provides more dynamic frames than is required. The 1*ms* cycle time is chosen because previous CAN rates were multiples of 1*ms*.
The author concludes that implementing this set up allows continued use of some CAN design practices but comes at a cost of deferring redundancy and determinism. FlexRay's performance is improved over CAN and there is further room for improvement through the possibility of increasing the frequency at which the DYN frames are transmitted.

6.6 Conclusion

This chapter starts by presenting an introduction to automotive network migration and some of the basic requirements. Following this section a discussion of side-by-side migration methods (gateways) and examples are given of other works carried out in this area are offered. The next part discusses full conversion methods. This is broken down into migrations to TT protocols, ET protocols and mixed systems. Then the genetic algorithm approach is discussed with examples of implementations by other authors. The chapter concludes with a discussion of other methods and the outcomes obtained. All previous works are backed up by methods and results to verify the findings.

This chapter goes some way towards answering research question number two. This is achieved by presenting cases where some authors have implemented gateways to enable the operation of CAN, where FlexRay was initially used. Other cases are presented where systems are completely specified to operate on a single protocol only. Due to their characteristics some protocols are more straightforward to convert than others. Numerous authors have carried out previous work on protocol migration. Each author's implementation techniques differ but some works contain similarities. All approaches delivered valid results.

6.7 References

A. ALBERT, R. S., A. TR¨ACHTLER (2003) 'Migration from CAN to TTCAN for a Distributed Control System' 9th international CAN Conference, Munich,

ABHIJIT DAVARE, Q. Z., MARCO DI NATALE, CLAUDIO PINELLO, SRI KANAJAN, ALBERTO SANGIOVANNI-VINCENTELLI (2007) 'Period Optimisation for Hard Real-Time Distributed Automotive Systems' DAC 2007, San Diego, Calafornia, USA, ACM

ALOUL, N. K. A. F. (2005) 'The Synthesis of Dependable Communication Networks for Automotive Systems' SAE Internalional 2005,

ANDREI HAGIESCU, U. D. B., SAMARJIT CHAKRABORTY, PRAHLADAVARADAN SAMPATH, P.VIGNESH, V. GANESAN, S. RAMESH (2007) Performance Analysis of FlexRay-based ECU Networks. ACM.

AUTOSAR (2007) AUTOSAR Specification. AUTOSAR.

CUMMINGS, R. (2008) 'Easing the Transition of System Designs from CAN to FlexRay' SAE International, Detroit,

DING SHAN, M. N., TOMIYAMA, HIROYUKI, TAKADA, HIROAKI (2005) A GA-Based Scheduling Method for FlexRay Systems. *5th ACM international conference on Embedded software EMSOFT '05.* Jersey City, NJ, USA.

GMBH, N. E. E. (2007) FlexRay Solutions by NEC Electronics,11/03/2008

GUIDO GELEN, E. W., SVEN LUKAS, CARL-HERBERT ROKITANSKY, BERNHARD WALKE. (2006) Architecture of a Vehicle Communication Gateway for Media Independent Handover.

JACKMAN, W. Z. A. B. (2007) 'Using Simulation for Designing In-Vehicle Network Gateways' SAE International, Detroit,

JAN CARLSON, T. L. A. G. F. (2003) 'Enhancing Time Triggered Scheduling with Value Based Overload Handling and Task Migration' Sixth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03), IEEE

KANCHI, J. R. P. A. S. (2007) An Optimized Real-Time Distributed Control System Scheduler for TDMA Based Protocols.

LAM, K. L. C. A. S. S. (1989) Deriving a Protocol Converter: a Top-Down Method. ACM.

MIKROELEKTRONIK, T. (2008) FlexRay Products,11/03/2008

NOSSAL, R. (1996) Pre-Runtime Planning of a Reliable Real-Time Communication System. Institut fur Technische Informatik, Technichal University of Vienna.

OSHANA, R. (2007) Real-Time Operating Systems for DSP, part 8,13/02/2008

S. CHAKRABORTY, S. K. A. L. T. (2003) A General Framework for Analysisng System Properties in Platform Based Embedded Systems.

SHAN DING, N. M., HIROYUKI TOMIYAMA AND HIROAKI TAKADA (2005) 'A GA-Based Scheduling Method for FlexRay Systems' Proceedings of the 5th ACM international conference on Embedded software, Jersey City, NJ, USA,

SHEHRYAR SHAHEEN, D. H. A. G. L. B. (2006) A gateway for time-triggered control networks. Science Direct.

SMITH, P. (2005) Automotive gateways spearhead in-car network integration. Automotive Design Line.

SUK-HYUN SEOL, S.-W. L., SUNG-HO HWANG AND JAE WOOK JEON (2006) 'Development of Network Gateway Between CAN and FlexRay Protocols for ECU Embedded Systems' SICE-ICASE International Joint Conference 2006, Bexco, Busan, Korea,

SURI, V. C. A. N. (2004) 'TTET: Event-Triggered Channels on a Time-Triggered Base' Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age,

TRAIAN POP, P. E., ZEBO PENG (2002) Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems.

TRAIAN POP, P. E., ZEBO PENG (2003) Design Optimization of Mixed Time/Event-Triggered Distributed Embedded Systems. ACM.

TRAIAN POP, P. P., PETRU ELES, ZEBO PENG (2007) 'Bus Access Optimisation for FlexRay-based Distributed Embedded Systems' EDAA,

TRAIAN POP, P. P., PETRU ELSES, ZEBO PENG, ALEXANDRU ANDREI (2006) Timinig Analysis of the FlexRay Communication Protocol

VALENZANO, G. C. A. A. (2004) Performance Analysis of Byteflight Networks. IEEE.

WEI ZHENG, J. C., CLAIDIO PINELLO, SRI KANAJAN, ALBERTO SANGIOVANNI-VINCENTELLI (2005) Extensible and Scalable Time-Triggered Scheduling. IEEE.

ZP. TAO, M. G. (1992) Synthesizing Communication Protocol Converter:A Model and Method. *ACM.* ACM.

7 Literature Review Conclusion

7.1 Conclusion

Modern automobiles contain an ever-increasing number of electronic components due an increasing number of critical and non-critical applications. The increased number of these applications is driven by consumer's desires for more comfort applications and also improvements in the development of safety applications. It is apparent that no single protocol will meet all the requirements of all applications. This is evident by the wide variety of protocols with differing features such as; varying bandwidth, different costs and levels of complexities at the protocol network layer.

Networks have proven their use and effectiveness either on a small scale or large scale, not only in the automotive industry but in any industry where communication is critical for operation.

By implementing a chosen protocol the characteristics of the network that operates that protocol are defined. There are four predominant automotive protocols CAN, LIN, MOST and FlexRay. Each of these has their own merits of use in their required setting. CAN is used for BCU (Body Control Unit) applications, LIN is used in non-critical BCU applications but has lower cost and speed than CAN. MOST is used for infotainment purposes in an automobile and FlexRay is planned for use in high speed safety critical applications.

FlexRay has an advantage over other existing protocols in that it utilises both ET and TT characteristics and has greater bandwidth. Primarily due to cost factors, FlexRay will not be the only network in an automobile; this leads to the existence of multiple networks in a structure. To enable applications currently executing on older protocols to take advantage of FlexRay's features, migration to FlexRay will be required.

Complete migration may be necessary if the older protocol does not have the features to successfully implement an application. If complete migration is not necessary then a gateway (as discussed in Chapter 6) could be a viable solution to working with more than one protocol.

There are many approaches that can be undertaken during the migration process. The two process discussed (Algorithmic and Heuristic) offer suitable solutions for most cases. The method chosen will depend on a number of factors such as system parameters, the application requirements, available hardware and software. To efficiently migrate from one network protocol to another the application is firstly analysed at task level and modelled in a task graph. Each task is then synthesised at message level. An optimal frame size can be found from a combination of; the messages release time, deadline time, WCET, slot size and slot delay as explained in Chapter 5.

As discussed previously in Chapter 6 most of the work previously available by other authors focuses on implementing the ET protocol in the static segment of the FlexRay frame. This results in scheduled messages meeting their deadlines but bandwidth utilisation is not optimised. To improve bandwidth utilisation in FlexRay, use of the DYN (ET) domain is required.

If each task schedule is configured in advance, it is this feature that makes time-triggered networks deterministic and predictable but if the networks schedule is not set up but activated on the occurrence of an event this is called an event-triggered network. When choosing the type of network in an automobile there are some factors that need to be taken into consideration such as what applications shall be executed. For example powertrain or air conditioning systems. If powertrain data is being transmitted on the network a FlexRay based protocol would be one possible option, but if it is the air conditioning system that runs on the network then maybe the LIN protocol may be more suitable. It is also possible to execute more than one protocol on the network; as per the above case we could have both networks running on the one vehicle. This has been achieved on modern vehicles using gateways.

With software becoming increasingly predominant in the development process within the automotive industry choosing the right network for the correct application is a major factor in cost. The worldwide value creation in automotive electric/electronics (including software) amounts to an estimated €127 billion in 2002 and an expected €316 billion in 2015 according to a Mercer study (Alexander Pretschner, 2007). Software will make up an estimated 40 percent of this value creation by 2010 (Alexander Pretschner, 2007).

7.2 References

ALEXANDER PRETSCHNER, M. B., INGOLF H. KRÜGER, THOMAS STAUNER (2007) 'Software Engineering for Automotive Systems: A Roadmap' Future of Software Engineering(FOSE'07),

Section III - Framework Development

8 Migration Framework Requirements:

8.1 Introduction

Chapter 6 describes methods other authors have used when scheduling various protocols on embedded systems. In this chapter the full migration method is chosen. This is due to it being deemed the most suitable alternative to the use of a gateway. The chosen migration method reduces the complexity of multiple protocols by employing a single protocol system. This chapter presents the process by which the migration framework was developed. In Chapter 9 the migration procedure that deals with the actual migration steps undertaken are presented. The framework was developed through logical procedural steps that allows for a variety of CAN configurations to be processed once a set of basic parameters are obtained and defined. The migration framework is started by defining the CAN application that will undergo the migration procedure. The migration procedure is summarised and graphically represented in Figure 8-1.

8.2 Migration Requirements

By undertaking this migration procedure the configuration of tasks is not affected. This is because this migration procedure is applied to the messages of each task cluster or application. Tasks are not required to be reassigned to different ECUs at any stage of the migration procedure, but can be done so to reduce the amount of data transferred on the network. The physical layer of the original system (CAN) is replaced with the physical layer of the migrated system (FlexRay). From the CAN system, the CAN controller is substituted with a FlexRay driver. The FlexRay driver requires a communications controller (CC). The CC can be integrated into the MCU. A FlexRay physical layer interface is required to meet the FlexRay standard physical layer interface specification. The communications stack or COMMSTACK links the application to the FlexRay communications controller.

Stages

8.3 Application Definition

Obtain the CAN application that will be used to undergo the migration procedure. This is the application from which the migration will proceed from and also the exact application that is to be implemented in FlexRay.

8.4 CAN Parameter Abstraction

Extracting the parameters from the CAN application is required so the critical properties of the CAN application are integrated into the FlexRay parameters. This is done by representing the application in task graph format and analysing its properties such as those presented in 8.4.2.

8.4.1 CAN Graph Abstraction

The migration framework can be undertaken on the premise that the CAN configuration is already setup and operational. Once this condition is met the CAN application needs to be represented in task graph format. This is done by abstracting each process, whether it is functional or computational, as an activity on the task graph. The activities are placed on the task graph in an order that meets the precedence constraint requirements. Activities are linked to each other by an arrow, with the arrow head indicating the progression flow through the task graph.

8.4.2 CAN Analysis

The CAN system is represented in task graph format, the next step is to abstract the initial CAN parameters. The set of CAN components are all directly derived from the task graph and are listed below;

- Task graph release (start) time r_i
- Task graph deadline (end) time D_i
- Worst Case Execution Time (*WCET*)
- Task period

The release time r_i will have an initial value of zero as this is the release time of the first task. The deadline time D_i will be a value at which all tasks in the task graph have completed execution. The WCET value can vary depending on hardware (CAN controller) and software constraints (Jitter). This value can be obtained by means of a specialist application that extracts the WCET of each message. This value can also be calculated (using Equation 5-6) if necessary parameters are known. These parameters are the jitter j_m, queuing delay/interference w_m and communication time C_m. The final parameter for definition before migration can commence is the CAN message set M_{CAN}, and its associated parameters CAN_{mi} and w_i. These parameters are explained in section 9.8.

8.5 Migration

The basic parameters are defined, the migration procedure (as per Chapter 9) can be undertaken. The resulting FlexRay parameters M_{FR} and $FR_{compset}$ will be either ideal for implementation or as close to ideal as is possible due to other constraint complexities mentioned. Such is the complexity and volume of parameters for configuration, there are some parameters whose values interconnect and are interdependent with respect to each other. The FlexRay parameters will be in the form of time units or slot number units.

8.6 FlexRay Frame Implementation

The FlexRay parameters extracted from the framework are used to configure the FlexRay frame. Each FlexRay parameter can be configured manually in XML file format. This approach has a greater chance of resulting in configuration constraint errors. If a FlexRay designer tool is used this will flag errors and violations. The designer tool for example will detect if by adjusting one parameter this affects and results in a constraint violation in a separate parameter. The designer tool can generate CHI (Controller Host interface) files that contain critical FlexRay frame parameters. The frame parameters can also be contained in a FIBEX (Field Bus Exchange) file. Then when implementing the FlexRay frame it takes on the parameters associated with the CHI or FIBEX file.

8.7 FlexRay Application Configuration

Configure and implement the FlexRay application based on the CAN application and the FlexRay frame parameters. The FlexRay applications structure does not change by undertaking the migration procedure. If there is a situation that requires the physical structure to change, return to section 8.1 and start over. Any difference in the FlexRay application structure, when compared to the CAN application structure, can result in different parameter values being extracted from the framework when compared to those that would have been obtained otherwise.

8.7.1 Validation

To validate migration the deadlines of both systems need to be compared. The framework is validated by examining execution times and busloads. This allows a direct comparison between both protocols. The migration benefits can be summarised in three scenarios.

Scenario I

Here both CAN and FlexRay meet their respective deadlines but due to low bus load volumes CAN messages get access to the bus faster than they do on the FlexRay protocol. In this situation under these conditions there is no benefit to implementing FlexRay unless other features of FlexRay are required by applications using the bus. The CAN bus is at the 30% busload mark in this scenario.

Scenario II

In this scenario both CAN and FlexRay meet their required deadlines but the CAN messages are suffering some delays getting access to the bus when compared to Scenario I. Implementation of FlexRay is more justifiable than in Scenario I but there is equally as strong a case to be made for the continued use of the CAN bus. The designer will have to make a decision based on a number of factors such as future loads on that bus and if FlexRay's features are required for the implementation of other applications, to give but two factors. CAN busload is at approximately 60% busload.

Scenario III

In this Scenario some CAN deadlines are being violated and/or busload has reached saturation. CAN busload exceeds the maximum attainable value and complete migration is justifiable.

8.8 Conclusion

Through the implementation of the steps discussed in this chapter a successful migration framework was developed. These steps are summarised in the flow chart in Figure 8-1. The chapter presents the steps necessary to process an application originally configured to operate on the CAN network and is now required to operate on the FlexRay network.

9 CAN to FlexRay Migration Methodology:

9.1 Introduction

This research is deemed a success upon achieving a successful implementation of the reference CAN application on the CAN protocol, and then migrating and implementing the same application using the FlexRay protocol, by applying the framework. The approach taken is similar to the approach used by Pop (2007). This involves exploring task deadline on the message level. Message level analysis techniques are designed to;

- Determine if deadlines are met in the ST segment
- Obtain the FlexRay frame length
- Find the WCRTs in the DYN segment

9.2 Framework Development

To develop a framework, initial parameters need to be determined, implemented and tested. Initial parameters are obtained from the reference CAN application such as initial release time, task graph deadline, task execution times etc, as specified in section 8.4.2.

9.2.1 System Definition

When migrating from CAN to FlexRay the first notable issue is that the two network protocols are different in their fundamental principle of operation. CAN operates on the principle of an ET protocol and FlexRay operates both an ET and a TT type protocols.

Before any migration can take place CAN data is required. This data can be generated by the user using parameters that provide a fair representation of a CAN system. The recorded CAN data is analysed to obtain deadline timings and bus parameters that FlexRay will need to meet upon successful migration.

9.2.2 Defining CAN Data in FlexRay Format

All CAN traffic is ET, thereby critical tasks can be given higher priority IDs to aid access to the bus. After the CAN parameters are obtained the designer is required to decide where in the FlexRay communication cycle this data can be placed. A possible solution is to place all ET CAN messages into the DYN segment of the FlexRay frame. This ensures the ET characteristics of CAN are met through the use of the ET characteristics of FlexRay's DYN segment as implemented by Cummings (2008). However it does not make efficient use of FlexRay's features, such as having both a ST and DYN segment; it only takes advantage of the high bandwidth available.
FlexRay's bus bandwidth was also necessary to consider. The options available were 2.5MBit/s, 5MBit/s and 10MBit/s. The value chosen would have a major bearing on other system parameters specified in appendix A and B of the FlexRay specifications v2.1 rev A.

If certain CAN messages are schedulable and occur at predefined moments in time a scheduling table can be constructed as to the exact moment in time when a message will require transmission and how often this occurs. Due to a message being triggered for transmission on a predefined moment in time, the network protocol responsible for transmission is considered a time-triggered protocol. FlexRay's ST segment fulfils this requirement. All critical schedulable messages can be placed in the ST segment of the FlexRay protocol to guarantee transmission.

A possible alternative to implementing all the critical data in the ST segment is to place some of the critical data in the DYN segment. The DYN segment has a defined segment in the FlexRay cycle; therefore if a single message was assigned the highest priority in this segment on its own it would be guaranteed to get access to the bus. As more messages are added, their priorities will be lower than the initial message thereby not guaranteeing timely access to the bus. This approach is not investigated further in this research as the only way of guaranteeing critical messages get access to the bus is by allocating them slots in the ST segment.

9.3 System Implementation

To achieve a successful system implementation the CAN application is implemented separately to the FlexRay implementation. First the reference CAN application is implemented and the consequential data is logged for analysis. Through the framework the FlexRay frame, cluster and message parameters are obtained. Using these parameters the reference application is implemented in FlexRay and its consequential data is logged for analysis.

9.3.1 Periodic Task Analysis

The aim of the analysis at this stage is to determine deadlines at the message level of the periodic tasks. The technique described in Aloul (2005) is used to synthesise the periodic tasks in the ST segment onto message level. The ST segment slot size is defined at this stage. A heuristic approach is taken when choosing the ST payload size. This then allows the FlexRay frame size to be defined and a FlexRay cycle period is also obtained.

9.3.2 Aperiodic Task Analysis

Due to the nature of the DYN segment a different approach has to be taken to that used for the periodic tasks. Aperiodic tasks are scrutinised down to their message level using a RTA procedure. Because each message is aperiodic a WCRT is calculated for each message so the designer knows under the worst circumstances what the maximum delay for each message is.

9.4 Verification

The application running on the FlexRay protocol is required to meet or improve on the timing and bus parameters obtained using the same application on the CAN protocol. Under very small bus loads CAN is able to marginally beat FlexRay timings due to there being a reduced chance of messages been blocked or delayed. At higher busloads FlexRay's additional bandwidth should enable it to handle higher volumes of data. This is verified in the Testing and Results section.

9.5 Implementation of Analysis Findings

The FlexRay frame can be configured as per results obtained in analysis of the ST and DYN segments. At this stage there could be discrepancies between values calculated and achievable values. This is a result of configuration values being different from the achievable minimum or maximum parameter ranges. A solution as close as possible to the calculated results is used in this scenario. This solution is attainable from the FlexRay frame designer program, or through the examination of FlexRay's constraints in appendix A and B of FlexRay's specifications.

9.6 Applying Framework to a Real-World Application

Using the framework steps in the abstract implementation (a Traction Control application); the framework is applied to a real-world application configuration. Bus statistics are recorded and compared to those achieved using the CAN protocol. The abstract implementation contains more nodes than the experimental implementation (an Adaptive Cruise Control Application) model and is also used to acquire associated FlexRay parameters.

9.7 Generic CAN to FlexRay Development

The experimental implementation of the application was processed in the same physical development environment where possible. This includes both the reference CAN implementation of the application and the reference FlexRay implementation of the application. This allowed a direct comparison between the two set-ups. The same can be said for the third test case "Verification of Time-Triggered Properties", both CAN and FlexRay implementations were implemented in CANalyzer. These implementation test cases (Traction Control, Adaptive Cruise Control and Verification of Time-Triggered Properties are presented in detail in Chapter 11). The initial model for obtaining the ST segment size and cycle period was abstracted from (Aloul, 2005) as described in section 6.5.5. The DYN segment size was developed from work carried out by (Traian Pop, 2006) as described in section 6.5.5. Key to obtaining the FlexRay frame and cluster configuration parameters was synthesising the application tasks to a message level. As FlexRay is composed of a static and dynamic segment, messages transmitted in the static segment are scheduled prior to execution while messages transmitted in the dynamic segment are event triggered and it is not known exactly when these events will occur. This lead to RTA approach being used in configuration of the DYN segment.

The CAN message set M_{CAN} (Equation 9-1) is composed of CAN messages, CAN_{mi}, which connect tasks on the task graph, where *i* is the number of messages on the task graph.

$$M_{CAN} = \{CAN_{m1} \dots CAN_{mi}\}$$

Equation 9-1: CAN Message Set

The CAN message parameter set is composed of a component set of message size (*size(m$_i$)*), message ID (ID$_i$)and the message period (*period(m$_i$)*). This is illustrated in Equation 9-2.

$$CAN_{mi} = (size(m_i), ID_i, period(m_i))$$

Equation 9-2: Message components

9.8 Static Segment Development

Initial requirements before a message can be scheduled are the tasks *worst case execution time (WCET) (w$_i$)*, process deadline time *D$_i$*, release time *r$_i$* and the task period. The final task graph deadline *D$_i$* is obtained by summing all the periods in the task graph. Each individual task can be compiled into a task graph which shows the execution time of each task and the task deadline. Any precedence constraints on each task should be taken into consideration at this stage. The task graph execution time is calculated by summing up each tasks execution time along the chosen path of the task graph. The longest path is the one that will finish at the latest time. To find the worst possible time a task will take to complete, select the longest path in the task graph (Equation 9-3).

$$w_i = \sum longest\, path \;\; through\; task\, graph$$

Equation 9-3: Task Graph Execution Time

Therefore the execution time of a path *i* is the sum of each tasks execution time along the chosen path. The *WCET* is dependent on a variety of factors such as the processor the task is being executed on and the task priority. *WCET* is calculated from the longest path a task takes from the moment it is called until it has successfully completed execution.

9.8.1 Task Scheduling

To schedule an intermediate task r_i and w_i of the intermediate task are required. The intermediate task deadline is calculated as illustrated in Equation 9-4.

$$d_i = w_i + r_i$$

Equation 9-4: Intermediate Task Scheduling

Any slack in the system can then be calculated and equally reallocated among each task. First the total slack *TotalSlack$_i$* is required. This is obtained by subtracting the task graph deadline from the sum of the execution times along a chosen task graph path as presented in Equation 9-5.

$$TotalSlack_i = D_i - \sum c_i$$

Equation 9-5: Total Available Slack

The total available slack is then reallocated equally among the number of tasks *(x)* along the chose path as illustrated in Equation 9-6.

$$slack_i = \frac{TotalSlack_i}{path_x}$$

Equation 9-6: Slack per Task

With the new slack reallocated, each tasks release time r_i and deadline time d_i is now re-calculated to include the slack.

After the final r_i, d_i and w_i are known we perform a check to determine if the values obtained thus far will potentially lead to a schedulable solution. Equation 9-7 is used in determining this result. Scheduling is considered successful if the *WCET* is less than or equal to d_i minus r_i.

$$w_i \le d_i - r_i$$

Equation 9-7: Parameter Validation Check

9.8.2 Message Deadline

Once the final r_i, d_i and w_i are known we use these parameters to calculate the deadline of each message as illustrated in Equation 9-8. This deadline is the transmission deadline time *td(m$_i$)*, by which time transmission of the message is required to have been completed.

$$td(m_i) = d_i - r_i - w_i$$

Equation 9-8: Transmission Deadline

Any factors affecting a delay in transmission need also be accounted for. Due to the ST segment being TT there is no network contention when a message tries to access the bus. Resulting from this the transmission delay is calculated as per Equation 9-9.

$$transmission\ delay = \frac{size(m_i)}{Bus_{speed}}$$

Equation 9-9: Transmission Delay

For multi-rate task graphs the LCM (Lowest Common Multiple) of all coupled task graph periods is used to guarantee the timely execution of all deadlines. An example of this is if there are two task graphs with periods of 2*ms* and 5*ms* respectively. A hyper-cycle of 10*ms* is required to guarantee timely transmission of all messages.

9.8.3 Payload Definition

To calculate the slot delay and the transmission time, the message size and frame payload size parameters need to be finalised. All the messages that require transmission are summed up and the header and trailer bytes are included to give a more realistic frame size in bytes. The header and trailer data are collectively termed the 'overhead', as this data accompanies the payload but is not required by the application. The number of message transmissions required to transmit all data is calculated. In determining the frame size the payload size also needs defining. This is due to the payload being part of the FlexRay frame.

Before obtaining the total number of FlexRay frames required to transmit the entire CAN payload data at the chosen payload it was required to know the FlexRay frame size. The FlexRay frame size is determined by Equation 9-10. Start at a payload size of one byte and increase by a value of one byte with each iteration. Payload sizes are required to be an even integer value.

$$FRsize(m_i) = (pd_j + O_h)$$

Equation 9-10: FlexRay Frame Size

Equation 9-11 presents the total number of frames required to transmit all ST data at the chosen payload size *pd*$_j$. Where *j* is an integer value in one application cycle. In Equation 9-12 *Cf*$_{FR}$ is the number of frames pre cycle. If the *size(m*$_i$*)* parameter includes bit stuffing this will lead to a data cycle total size (in terms of number of bytes) that is too large.

$$Cf_{FR} = \sum_{i=1}^{n} \left(\left\lceil \frac{size(m_i)}{pd_j} \right\rceil \right)$$

Equation 9-11: Frames per Application Cycle (ST Data)

At the set payload size the number of frames required to transmit the complete cycle data can be found. Start at a payload size of one byte and increase in single integer values until ten consecutive increases in the total number of bytes is obtained. Now that the total number of required frames per application cycle is found the total amount of data for transmission is also known using Equation 9-12. *D* is the total transmitted data at the chosen payload value.

$$D = Frame\,Size \times Cf_{FR}$$

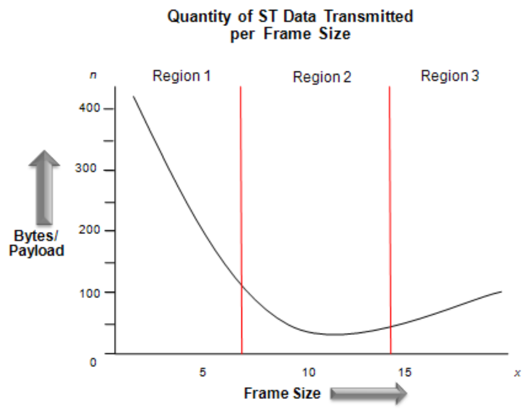
Equation 9-12: Total Required Bytes

A list is generated containing the total number of bytes required to transmit all the data at the chosen frame size. Graphing this data results in a graph with the general profile similar to that as illustrated in figure 9-1. Each region on the graph is explained in the following paragraphs.

Region 1: Payload sizes are initially very small in relation to the total amount of data for transmission therefore a large number of frames are required to transmit the data. As there are more frames there is also more overhead accompanying those frames. To give an example, if there are 40 Bytes for transmission in total and the payload size is 1 Byte then 40 frames are required. If the payload is increased to 2 Bytes then only 20 Bytes would be required. This means the overhead is reduced from 40 frames to 20 frames.

Region 2: This is the region from which the payload and frame size are chosen. The lower limit is reached in terms of the total number of bytes transmitted per chosen payload size; therefore the payload reaches its optimal size in relation to the total number of bytes transmitted. This continues the trend of less total data due to less overhead. The total data transmitted starts to increase due to the payload continuing to increase.

Region 3: The payload is continuing to increase. The increase is linear because the number of frames required has reached its optimal point and cannot be reduced any further. For example if there are 3 messages for transmission (regardless of how small the payload size) all the data cannot be transmitted in less than 3 frames.



The payload and frame value are chosen heuristically. The chosen value is not necessarily the value that results in the smallest total number of bytes for transmission. A smaller or larger payload value might better suit the applications requirements. A smaller payload size requires more frames to transmit the same quantity of data. Choosing a larger payload size can result in a larger frame size which allows reduced system granularity.

9.8.4 ST Slot Size

In the ST segment a task transmits a message and that message is transmitted in an assigned slot. In this framework this assignment is implemented as follows. *T1* transmits *m1* which is assigned to slot 1, *T2* transmits *m2* which is assigned to slot 2 and so forth until all messages have been assigned a unique slot. Each message requires an individual slot due to there being no slot reuse in the ST segment of the FlexRay protocol.

Each message has a period, *period(m$_i$)* equal to its deadline *td(m$_i$)*. Transmission slot duration *gdStaticSlot* is calculated by summing the message payload size and message overhead and then dividing by the bus bandwidth (9-13).

$$gdStaticSlot = \frac{size(payload_i) + size(overhead_i)}{Bus_{speed}} \mu s$$

Equation 9-13: Slot Duration

9.8.5 Discretisation

By discretising the slot duration the message period can be expressed as a function of slots instead of as a function of time. This allows greater ease in setting the FlexRay frame parameters. Message period is now expressed in terms of transmission slot intervals. This is achieved by dividing *td(m$_i$)* by *gdStaticSlot* (9-14).

$$td(M_i) = \left\lfloor \frac{td(m_i)}{gdStaticSlot} \right\rfloor$$

Equation 9-14: Discretised Slot Delay

The discretised slot duration *td(M$_i$)* is labelled so to differentiate it from the undiscretised slot duration *td(m$_i$)*.

9.8.6 Periodicity and Distance Constraints

To guarantee a message will meet its deadlines two constraints are invoked;

- Periodicity: The minimum message period, p_{min}, is required to be a multiple harmonic of a base period, p_{base}.
- Distance: The distance between two successive messages must be less than or equal to the message period, *period (m$_i$)*.

To obtain a FlexRay frame cycle that meets both these constraints we use the minimum message period, p_{min}, as our starting point (the smallest *td(M$_i$)* value). We obtain a p_{base} value that is a multiple harmonic period of p_{min}. The other message periods can be adjusted downwards (if message periods are increased there is the possibility of missing deadlines). This allows the periodicity constraint to be met. The distance constraint is met because the distance between slot 1 in cycle 1 and slot 1 in cycle 2 is constant and so on. Also as the cycle period is modified to p_{min} or less (as defined by the periodicity constraint) and the distance between slot 1 in successive cycles is less than or equal to the message period. Where slot 1 is mentioned it is also applicable for the range of slots *1, 2, 3……..n*, where *n* is the number of ST slots.

Examining Figure 9-2, if P_{min} is 6 and p_{base} contains a value of 1.5, this meets the distance and periodicity requirements. This is backed up by Equation 9-15.

When scheduling the FlexRay frame it is suggested to reduce the cycle size as much as possible without exceeding any timing constraints. This is to aid the DYN segment in accessing the FlexRay bus as frequently as possible, thereby enabling DYN messages to be transmitted more frequently on the bus, if required by the application.

Equation 9-15 is used in the validation of the p_{base} value as a multiple harmonic, which allows the periodicity and distance constraints to be finalised.

$$2^k \cdot p_{base} \leq p_{\min} < 2^{k+1} \cdot p_{base}$$

Equation 9-15: Periodicity and Distance Constraint Validation

The number of iterations required to validate what *k* is equal to in *2^k* part of Equation 9-15, can be determined from the number of (integer) iterations taken to get from p_{min} to p_{base} minus one. For example if p_{min} was 48*ms* and p_{base} was 3*ms*, thereby k is equal to 4 (5 iterations minus 1).

$3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 48$

$$2^4 \cdot 3 \leq 48 < 2^{4+1} \cdot 3$$

$$48 \leq 48 < 96$$

Any modifications still have to meet the periodicity and distance constraint requirements.

9.9 Dynamic Segment Development

As the DYN segment is used to transmit aperiodic messages these cannot be scheduled prior to transmission. This leads to the use of RTA in determining the worst case delay of each DYN message. Some parameters that can affect the transmission of the DYN messages are;

- The earliest a message can be transmitted in a slot in the DYN segment is after the ST segment has finished transmission
- DYN messages are assigned on a priority basis; therefore higher priority messages can potentially block or delay lower priority messages
- Transmission of a message cannot occur if the minislot counter value is less than the *pLatestTx* value. The frame and payload sizes are calculated as per DYN message sizes.
- Only one node can transmit on the bus at any one time

- Only one slot is assigned to a node at any one time
- The node determines when the slot counter value is equal to the frame identifier value.

The DYN segment size is already defined. This is because the ST segments size, FlexRay cycle size and the NIT (obtained through a FlexRay designer tool) (Dependable Computer Systems, 2007) parameters are predefined through our ST segment analysis. This leaves the remaining time allocated for the DYN segment.

First a check is required to assess if it is possible for all DYN messages to transmit in the DYN segment. For this to be possible the number of minislots should be greater than or equal the number of DYN tasks. If this is not the case some tasks will not be allocated a slot for transmission. Reducing the minislot size as per (Consortium, 2005) allows more slots to be allocated without increasing the DYN segment size.

To verify if the DYN segment is large enough for the transmission of all DYN tasks we need to view the DYN segment size as a function of the number of minislots. This is done using Equation 9-17. Here the total communication time of all DYN messages (Equation 9-16) is divided by the number of minislot. If the number of provided slots is greater than or equal to the number of required slots, transmission is possible under correct scheduling. This is shown in Equation 9-18.

$$TotalDYN\ C_m = \sum (C_{m1} \ldots C_{mn})$$

Equation 9-16: Total DYN Communication Time

$$Required\ Slots = \frac{TotalDYN\ C_m}{gdminislot}$$

Equation 9-17: Minimum Number of Required DYN Slots

$$No.\,of\ \ allocated\ slots \geq No.\,of\ \ required\ slots$$

Equation 9-18: Slot Quantity Verification

The maximum worst case delay, calculated using Equation 9-19 takes into account delays caused by other DYN messages, the ST segment and associated FlexRay parameters.

$$R_m(t) = C_m + \sigma_m + w_m(t)$$

Equation 9-19: Dynamic Worst Case Response Time

Where;

R_m is worst case response time

C_m as required for ST segment, is the message communication time, Equation 9-20

δ_m is the longest delay suffered during one bus cycle if a message is generated by sender task just after its slot has passed, Equation 9-21

w_m is the worst case delay caused by transmission of ST messages and higher priority DYN messages, Equation 9-23

The communication time, as illustrated in Equation 9-20 is the DYN frame size multiplied by the bus bit time. The DYN frame size is composed of the DYN message and associated overhead. The same overhead value as calculated per the ST segment is used.

$$C_m = DYNFrame_i \times Bit\ Time$$

Equation 9-20: Communication Time

$$\sigma_m = FR(t) - (ST_{bus} + C_m + NIT)$$

Equation 9-21: δ_m Delay

Where *FR(t)* is the length of the FlexRay bus and ST_{bus} is the length of the ST segment, and C_m is the time the DYN message itself takes on the bus.

w_m can be caused by higher priority messages *hp(m)*, and unused DYN slots with lower priority identifiers. Each unused minislot gives a delay of one minislot *ms(m)*. The single minislot enables the minislot counter to increment to the next minislot value. In the worst case to obtain the minislot number it is assumed that all messages of a higher priority (than the current message) have transmitted in the previous cycle. This results in equation 9-22.

$$ms(m) = Total\ DYN\ \text{minislots} - 1$$

Equation 9-22: Number of Unused Minislots

For this calculation the worst case delay occurs if the message requires transmission at the moment the ***pLatestTx*** value is the same as the minislot counter. Therefore all minislots after this value cannot be used for transmission. This is illustrated in Equation 9-23.

$$w_m(t) = ST_{bus} + hp(m) + ms(m) + (pLatestTx.gdMinislot) + NIT$$

Equation 9-23: *$w_m(t)$* Delay

Using Equation 9-19 presents the *WCRT* delay of a DYN message. This enables the network designer to know how many FlexRay cycles it will take to guarantee the transmission of each DYN message.

9.10 Conclusion

Initial CAN parameters are processed to deliver the CAN message parameters M_{CAN}. The component set of M_{CAN} parameters are the message size (*size(m_i)*), message ID (*ID_i*)and the message period (*period(m_i)*). These parameters allow the development of the FlexRay cluster configuration parameters. The cluster configuration parameters concerned are;

- Cycle Length
  - gMacroPerCycle
- ST Segment
  - gNumberofStaticSlots

  - gdStaticSlot
  - gPayloadLengthStatic
- DYN Segment
  - gdMinislot
- Network Idle Time
  - gdNIT

The cluster parameters are necessary when configuring the FlexRay frame. From the framework the FlexRay message set is M_{FR} = {FR_{m1}.....FR_{mn}}. The message set is defined as FR_{mn} = (FR_{period}, FR_{size} and w_i), where FR_{size} and w_i are the same values as for M_{CAN} because the application task set does not change. The FlexRay frame component set can now be defined as $FR_{compset}$ = *(ST, DYN, and NIT)*. With further sub division the ST segment is composed of the parameter set FR_{ST} = ($slot_{size}$, num_{slot}, ST$payload_{size}$). The DYN segment is composed of the parameter set FR_{DYN} = ($mslot_{size}$ *and* $DYNpayload_{size}$). The network idle time is the final parameter set FR_{NIT} = (NIT_{size}).

Once complete migration has taken place from the CAN protocol to the FlexRay protocol the system designer should have all necessary parameters to configure the FlexRay frame. The framework defines the frame size, slot sizes and payload sizes in the static segment and DYN segment minislot and payload sizes.

To summarise, the framework can be analysed in algorithmic format. Figure 9-2 presents the algorithm for the ST segment, Figure 9-3 presents the DYN segment algorithm and Figure 9-4 presents the FlexRay cluster and frame parameter message sets.

ST Segment

```
Graphically represent app in TG format
CAN parameters ($r_i$, $D_i$,)
if WCRT not available
  Determine using $j_m$, $w_m$ and $C_m$
for $M_{CAN}$ ={$CAN_{m1....}CAN_{mi}$} CAN message set definition
{
  $CAN_{mi}$=(size($m_i$), $ID_i$, period($m_i$))
}
Extract Intermediate Deadlines
Reallocate Slack
Obtain recalculated $r_i$ and $d_i$ deadline
for validation check
{
  $w_i \leq d_i - r_i$
}
define transmission deadline td($m_i$)
determine transmission delay

if multi rate graphs
  use LCM to obtain cycle size
else
 Define $pd_j$ payload size
 using FRsize($m_i$), $O_h$, $Cf_{FR}$
 for total bytes required at chosen frame size D
{
  determine gdStaticSlot size of static slot
}
 discretise td($M_i$)
 Verify via periodicity and distance constraint
          $p_{min}$ and $p_{base}$
          $2^k.p_{base} \leq p_{min} \leq 2^{k+1}.p_{base}$
End;
Return ST Parameters /*ST slot size, payload, frame size*/
```

Figure 9-3: ST Scheduling Algorithm

DYN Segment

```
Determine total DYN communication time $TotalDYNC_m$
verify {
        slot quantity
        WCRT
        $C_m$ communication time
        $\delta_m$ longest delay during one bus cycle if
        slot has just passed
        }
for
 {
  $w_m(t)$ delay caused by ST msgs and hp ($m_i$)
 }
end;
Return max delay /*Max possible delay for DYN msgs*/
```

Figure 9-4: DYN Scheduling Algorithm

Frame & Cluster Parameters

```
flexRay message set is defined as $M_{FR}$ = {$FR_{m1}$.....$FR_{mn}$}
        for
        {
         $FR_{mn}$ = ($FR_{period}$, $FR_{size}$ and $wi$)
        }
flexRay component set  $FR_{compset}$ = (ST, DYN, and NIT)
        for
        {
         $FR_{ST}$ = ($slot_{size}$, $num_{slot}$, $STpayload_{size}$)
         $FR_{DYN}$ = ($mslot_{size}$ and $DYNpayload_{size}$)
         $FR_{NIT}$ = ($NIT_{size}$)
        }
end;
return Flexray data /*ST and DYN config data*/
```

Figure 9-5: FlexRay Extraction Parameters

9.11 References

Aloul N K a F,2005, The Synthesis of Dependable Communication Networks for Automotive Systems, SAE Internalional 2005,

Consortium F, 2005, FlexRay Communications System Protocol Specification Version 2.1 Revision A. p 245,

Cummings R,2008, Easing the Transition of System Designs from CAN to FlexRay, SAE International, Detroit.

Dependable Computer Systems D, 2007, Designer PRO 4.3.0. In: *Designer pro, Designer Pro<Light> and Designer Pro<System>,*

Pop T, 2007,Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, Department of Computer and Information Science, Linköping University Institute of Technology

Traian Pop P P, Petru Elses, Zebo Peng, Alexandru Andrei, 2006, Timinig Analysis of the FlexRay Communication Protocol, Editor, Conference Name, Conference Location.

10 Development Tools and Applications:

10.1 Introduction

This section describes all the physical hardware and the software applications used in the development and implementation of the chosen applications. For the abstract implementations the only hardware used was the pc upon which the FlexRay frame parameters were developed. From the software perspective, for the abstract implementation the FlexRay designer tool was used to verify parameters extracted from the framework. The experimental implementation involved the use of all hardware and software mentioned in this chapter. During the initial development of the generic application the designer can set the parameters as per the application. This includes the number of nodes, tasks, channels. For implementation of the ACC configuration, restrictions such as the minimum number of tasks shall be defined by the application. For the implementation of the third test case CANalyzer was used to configure and process both the CAN and Flexray configurations.

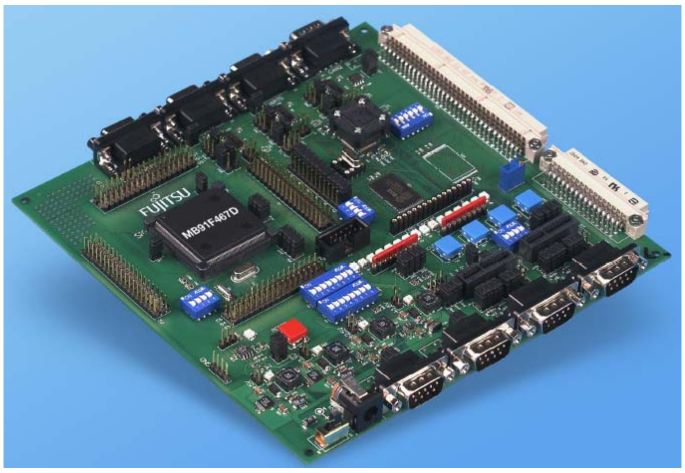
10.2 Hardware

10.2.1 FlexRay Development Kit

The initial stage of implementing the CAN and FlexRay application was carried out on the SK-91F467-FlexRay which is illustrated in Figure 10-1. Each starter kit can be considered a CAN node or a FlexRay node depending on the protocol on which the application is running. The CAN and FlexRay ACC applications were built tested and debugged using this starter kit. The starter kit allows the developer to use CAN and FlexRay and is part of the Fujitsu MB91460 series. At the core of the starter kit is a 32-bit flash MCU (microcontroller unit); this MCU is the MB91F467D. The Fujitsu FlexRay communications controller (CC) MB88121B enables FlexRay communication. The starter kit can be used in standalone mode or in monitor debugger mode. Using the *FlexTiny FR*, the physical layer, as specified in the FlexRay specifications is achieved. This device is plugged in to the ports provided on the boards.

The main features of the starter kit are listed below (Europe, 2007);

- Supports 32-bit Flash microcontroller MB91F467D
- Supports FlexRay CC MB88121
- On-board Memory: 32Mbit (4MByte) SRAM
- It is possible to connect the FlexRay CC in different ways to the MCU
- All microcontroller resources available for evaluation
- In-circuit serial flash programming
- Three selectable RS-232 or LIN UART-interfaces
- Three High-Speed CAN interfaces
- Two FlexRay channels (Ch-A, Ch-B)
- FlexRay physical layer RS-485 available
- FlexRay physical layer driver module from TZM (FT1080) connectable
- 16 User LEDs



10.2.2 Key Parameters

10.2.2.1 FlexRay Physical Layer

The physical layer connection is achieved through a RS485 transceiver or a physical layer driver module from TZM, called FlexTiny (FT1080). It can be plugged in and by-passes this transceiver. The FlexTiny FR (FlexRay) is illustrated in Figure 10-2. This device offers a physical layer connection as specified in the FlexRay physical layer specifications.

10.2.3 CAN Channels

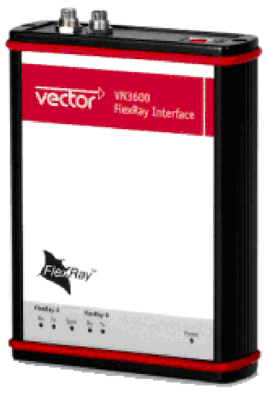
There are three high speed CAN channels available (CAN0, CAN1 and CAN2) for connection to the CAN controller through 9 pin D-sub connectors.

10.2.4 Interfaces

There are 4 push button switches that are connected to the MCU. These can act as external inputs, reload timer triggers, and input capture. There is also a separate system reset button. There are 16 output LEDs connected to two ports (port 16 and 25).

10.2.5 VN3600 FlexRay interface

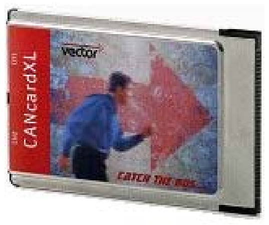
The Vector VN3600 as illustrated in Figure 10-3 is a USB FlexRay interface.



The VN3600 carries out analysis with the use of the Bosch E-Ray CC and start up is enabled through the use of the MB88121B CC (GmbH, 2008). Each interface allows access for one FlexRay cluster (Channel A & B) therefore if the designer wishes to integrate more clusters, more interface units are required. It supports the maximum payload of 254bytes, it can coldstart a cluster without an additional node and performs start up and synchronous monitoring.

10.2.6 CANcardXL

The CANcardXL is a CAN PCMCIA interface suitable for notebook or desktop (illustrated in Figure 10-4).



It contains two (independent) CAN channels that are configured for 11 or 29 bit identifiers. It can be used for error detection and remote frame generation.

10.2.7 Passive Star

The TZM FlexPS as illustrated in Figure 10-5 (FlexRay Passive Star) allows the user to configure up to six passive branches. It was developed with the FlexRay physical layer test working group. It also contains a car body ground connection and several test points. Use of the FlexPS enables a more realistic network topology over direct connection between channels. Channel A and Channel B busses require a FlexPS each otherwise conflict will occur on the bus.

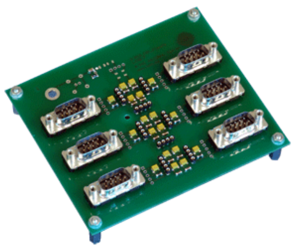
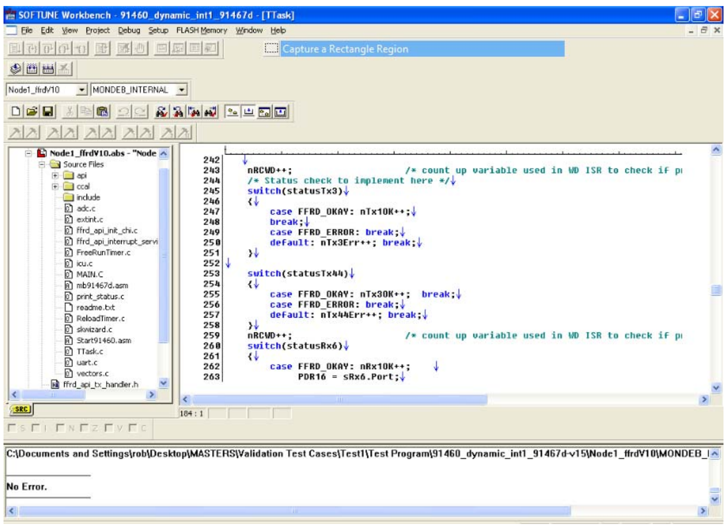


Figure 10-5: FlexRay Passive Star

10.3 Software

10.3.1 FR Family Softune Workbench

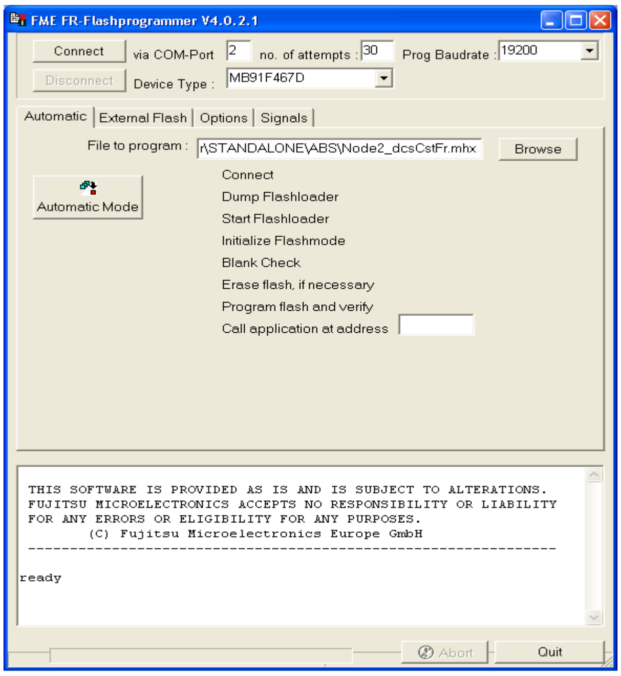
The FR family Softune workbench that was supplied with the starter kit was version 6 as can be seen in Figure 10-6. Softune workbench enables programs to be built and compiled.



```
    nRCWD++;                    /* count up variable used in WD ISR to check if p
    /* Status check to implement here */
    switch(statusTx3)
    {
        case FFRD_OKAY: nTx1OK++;
        break;
        case FFRD_ERROR: break;
        default: nTx3Err++; break;
    }

    switch(statusTx44)
    {
        case FFRD_OKAY: nTx3OK++;  break;
        case FFRD_ERROR: break;
        default: nTx44Err++; break;
    }
    nRCWD++;                    /* count up variable used in WD ISR to check if p
    switch(statusRx6)
    {
        case FFRD_OKAY: nRx1OK++;
            PDR16 = sRx6.Port;
```

```
C:\Documents and Settings\rob\Desktop\MASTERS\Validation Test Cases\Test1\Test Program\91460_dynamic_int1_91467d-v15\Node1_ffrdV10\MONDEB_I

No Error.
```

Once the program is successfully built and compiled a .mhx (Motorola hex) (Manual, 2006) output file is produced. If two nodes are being built for example a .mhx file will be required for each node. Next application is the FME (Fujitsu Microelectronics Europe) FR Flash programmer whose interface is illustrated in Figure 10-7.



via COM-Port: 2
no. of attempts: 30
Prog Baudrate: 19200
Device Type: MB91F467D
File to program: r\STANDALONE\ABS\Node2_dcsCstFr.mhx

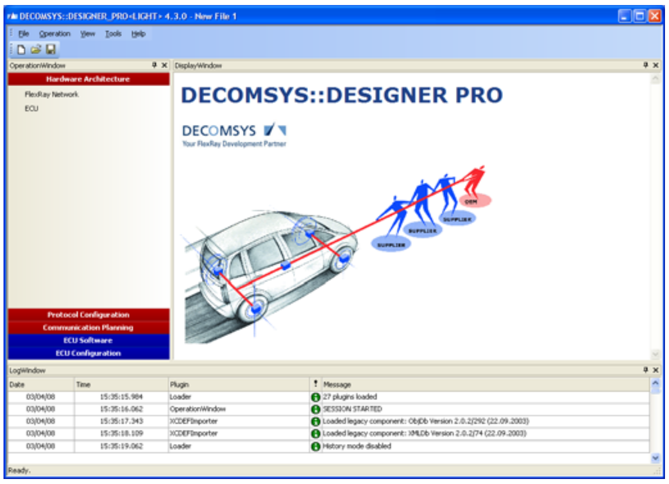
```
 THIS SOFTWARE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS.
 FUJITSU MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY
 FOR ANY ERRORS OR ELIGIBILITY FOR ANY PURPOSES.
        (C) Fujitsu Microelectronics Europe GmbH
 ----------------------------------------------------------------

ready
```

The appropriate MCU device is selected along with the baud rate and port number. Once these parameters have been selected the .mhx file is selected to be flashed into the memory of the starter kit. Each node has to be flashed separately.

If monitor debugger mode of the workbench is required then once the program is successfully built and compiled the debugger is then started (the monitor debug kernel is required to be pre-flash loaded to each board before monitor debug mode can be entered). When the debugger is running the program can be stepped through as required. Break points and watch windows can be set.

10.3.2 DECOMSYS DESIGNER PRO

DECOMSYS Designer Pro is used to set the FlexRay cycle parameters. The actual version used was DECOMSYS::DESIGNER_PRO <LIGHT> 4.3.0 as illustrated in Figure 10-8.



```
Date       Time           Plugin            Message
03/04/08   15:35:15.984   Loader            27 plugins loaded
03/04/08   15:35:16.062   OperationWindow   SESSION STARTED
03/04/08   15:35:17.343   XCDEFImporter     Loaded legacy component: ObjDb Version 2.0.2/292 (22.09.2003)
03/04/08   15:35:18.109   XCDEFImporter     Loaded legacy component: XMLDb Version 2.0.2/74 (22.09.2003)
03/04/08   15:35:19.062   Loader            History mode disabled
Ready.
```

DECOMSYS designer pro has five sub headings with which the FlexRay frame details are defined be defined in:

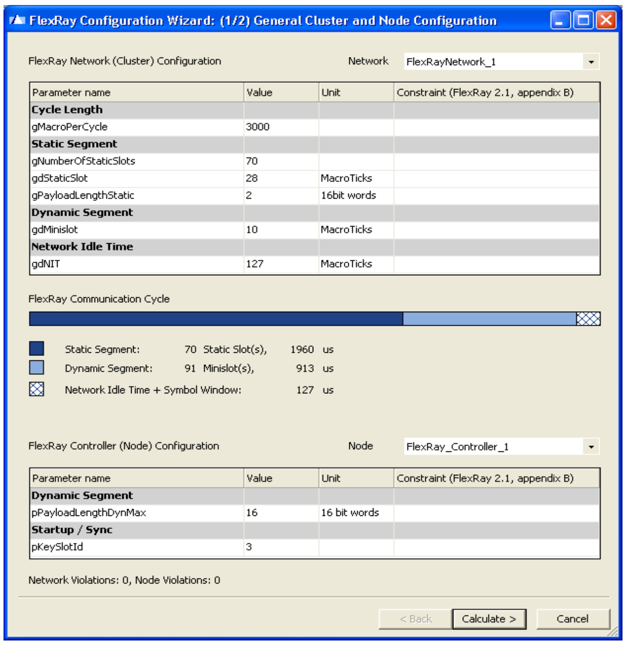
- Hardware Architecture
- FlexRay Frame Configuration
- Communication Planning
- ECU Software
- ECU Configuration

Hardware Architecture

The FlexRay network name is created and the Channel usage (A, B, A & B or none) is defined. Each node ECU is named and the communication controller (CC) is also selected.

FlexRay Frame Configuration

Here the main parameters of the FlexRay frame are defined such as cycle length, static segment size, dynamic segment size, slot size and mini slot size. A graphical representation of the frame is shown as illustrated in Figure 10-9.


FlexRay Configuration Wizard: (1/2) General Cluster and Node Configuration

FlexRay Network (Cluster) Configuration — Network: FlexRayNetwork_1

Parameter name	Value	Unit	Constraint (FlexRay 2.1, appendix B)
Cycle Length			
gMacroPerCycle	3000		
Static Segment			
gNumberOfStaticSlots	70		
gdStaticSlot	28	MacroTicks	
gPayloadLengthStatic	2	16bit words	
Dynamic Segment			
gdMinislot	10	MacroTicks	
Network Idle Time			
gdNIT	127	MacroTicks	

FlexRay Communication Cycle

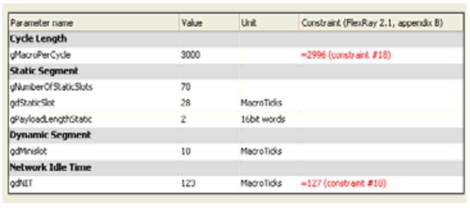
■	Static Segment:	70	Static Slot(s),	1960	us
■	Dynamic Segment:	91	Minislot(s),	913	us
▨	Network Idle Time + Symbol Window:			127	us

FlexRay Controller (Node) Configuration — Node: FlexRay_Controller_1

Parameter name	Value	Unit	Constraint (FlexRay 2.1, appendix B)
Dynamic Segment			
pPayloadLengthDynMax	16	16 bit words	
Startup / Sync			
pKeySlotId	3		

Network Violations: 0, Node Violations: 0

These parameters are then used to calculate other required parameters. If a constraint violation has occurred it is highlighted in red with the constraint number. In Figure 10-10 constraint #18 is violated for example.



Parameter name	Value	Unit	Constraint (FlexRay 2.1, appendix B)
Cycle Length			
gMacroPerCycle	3000		=2996 (constraint #18)
Static Segment			
gNumberOfStaticSlots	70		
gdStaticSlot	28	MacroTicks	
gPayloadLengthStatic	2	16bit words	
Dynamic Segment			
gdMinislot	10	MacroTicks	
Network Idle Time			
gdNIT	123	MacroTicks	=127 (constraint #18)

Figure 10-10: Constraint Violation

This constraint number can be checked in the FlexRay specifications appendix B. Once these parameters are met, select the finish button to keep the configuration.

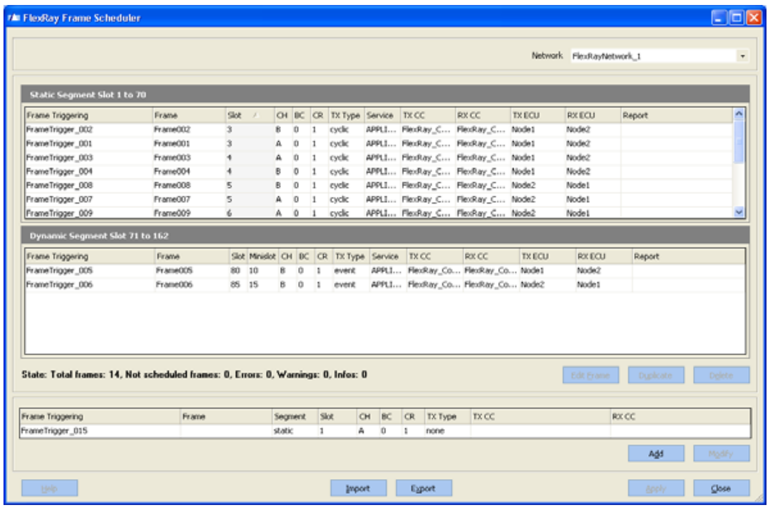
Communication Planning

FlexRay frames are created and named. Each frame is assigned an associated signal value. The frame schedule is built here. Each frame has the following parameters:

- Frame Triggering
- Frame
- Channel
- Slot
- Base Cycle
- Cycle Repetition
- Tx (transmit) CC
- Rx (receive) CC
- Tx ECU
- Rx ECU
- Report
- Service

The dynamic cycle has two parameters specific to its cycle: Minislot and Tx type.

A sample Frame schedule is shown in Figure 10-11.



Network FlexRayNetwork_1

Static Segment Slot 1 to 70

Frame Triggering	Frame	Slot	CH	BC	CR	TX Type	Service	TX CC	RX CC	TX ECU	RX ECU	Report
FrameTrigger_002	Frame002	3	B	0	1	cyclic	APPLI...	FlexRay_C...	FlexRay_C...	Node1	Node2	
FrameTrigger_001	Frame001	3	A	0	1	cyclic	APPLI...	FlexRay_C...	FlexRay_C...	Node1	Node2	
FrameTrigger_003	Frame003	4	A	0	1	cyclic	APPLI...	FlexRay_C...	FlexRay_C...	Node1	Node2	
FrameTrigger_004	Frame004	4	B	0	1	cyclic	APPLI...	FlexRay_C...	FlexRay_C...	Node1	Node2	
FrameTrigger_008	Frame008	5	B	0	1	cyclic	APPLI...	FlexRay_C...	FlexRay_C...	Node2	Node1	
FrameTrigger_007	Frame007	5	A	0	1	cyclic	APPLI...	FlexRay_C...	FlexRay_C...	Node2	Node1	
FrameTrigger_009	Frame009	6	A	0	1	cyclic	APPLI...	FlexRay_C...	FlexRay_C...	Node2	Node1	

Dynamic Segment Slot 71 to 162

Frame Triggering	Frame	Slot	Minislot	CH	BC	CR	TX Type	Service	TX CC	RX CC	TX ECU	RX ECU	Report
FrameTrigger_005	Frame005	80	10	B	0	1	event	APPLI...	FlexRay_Co...	FlexRay_Co...	Node1	Node2	
FrameTrigger_006	Frame006	85	15	B	0	1	event	APPLI...	FlexRay_Co...	FlexRay_Co...	Node2	Node1	

State: Total frames: 14, Not scheduled frames: 0, Errors: 0, Warnings: 0, Infos: 0

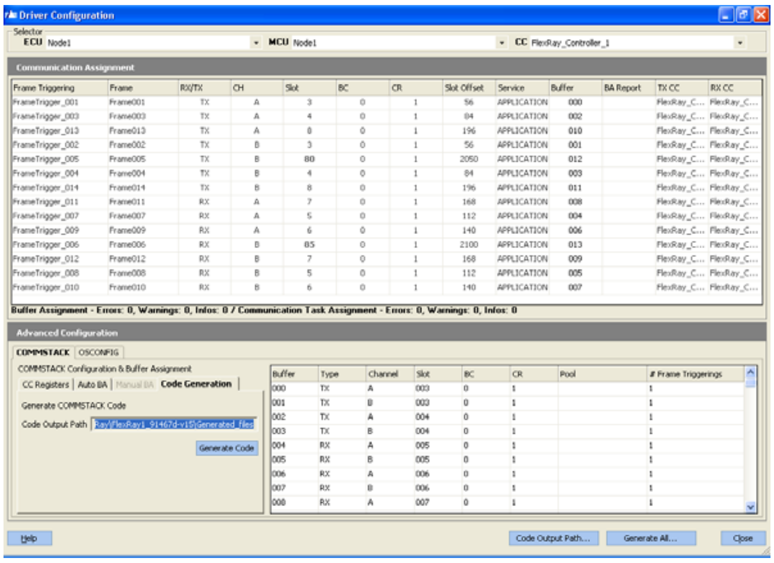
Frame Triggering	Frame	Segment	Slot	CH	BC	CR	TX Type	TX CC	RX CC
FrameTrigger_015		static	1	A	0	1	none		

ECU Software

This area is used to configure the periodic tasks for any application and interrupt service routines. The files generated here are used by OSCONFIG (Operating System CONFIGuration) in the ECU driver configuration (Dependable Computer Systems, 2007). These files were not required for implementation in this research and as a result they will not be discussed any further.

ECU Configuration

The ECU driver configuration allows buffers to be assigned and generate the code required for the COMMSTACK configuration.



ECU Node1 MCU Node1 CC FlexRay_Controller_1

Frame Triggering	Frame	RX/TX	CH	Slot	BC	CR	Slot Offset	Service	Buffer	BA Report	TX CC	RX CC
FrameTrigger_001	Frame001	TX	A	3	0	1	56	APPLICATION	000		FlexRay_C...	FlexRay_C...
FrameTrigger_003	Frame003	TX	A	4	0	1	84	APPLICATION	002		FlexRay_C...	FlexRay_C...
FrameTrigger_013	Frame013	TX	A	8	0	1	196	APPLICATION	010		FlexRay_C...	FlexRay_C...
FrameTrigger_002	Frame002	TX	B	3	0	1	56	APPLICATION	001		FlexRay_C...	FlexRay_C...
FrameTrigger_005	Frame005	TX	B	80	0	1	2050	APPLICATION	012		FlexRay_C...	FlexRay_C...
FrameTrigger_004	Frame004	TX	B	4	0	1	84	APPLICATION	003		FlexRay_C...	FlexRay_C...
FrameTrigger_014	Frame014	TX	B	8	0	1	196	APPLICATION	011		FlexRay_C...	FlexRay_C...
FrameTrigger_011	Frame011	RX	A	7	0	1	168	APPLICATION	008		FlexRay_C...	FlexRay_C...
FrameTrigger_007	Frame007	RX	A	5	0	1	112	APPLICATION	004		FlexRay_C...	FlexRay_C...
FrameTrigger_009	Frame009	RX	A	6	0	1	140	APPLICATION	006		FlexRay_C...	FlexRay_C...
FrameTrigger_006	Frame006	RX	B	85	0	1	2100	APPLICATION	013		FlexRay_C...	FlexRay_C...
FrameTrigger_012	Frame012	RX	B	7	0	1	168	APPLICATION	009		FlexRay_C...	FlexRay_C...
FrameTrigger_008	Frame008	RX	B	5	0	1	112	APPLICATION	005		FlexRay_C...	FlexRay_C...
FrameTrigger_010	Frame010	RX	B	6	0	1	140	APPLICATION	007		FlexRay_C...	FlexRay_C...

Buffer Assignment - Errors: 0, Warnings: 0, Infos: 0 / Communication Task Assignment - Errors: 0, Warnings: 0, Infos: 0

COMMSTACK Configuration & Buffer Assignment

Generate COMMSTACK Code

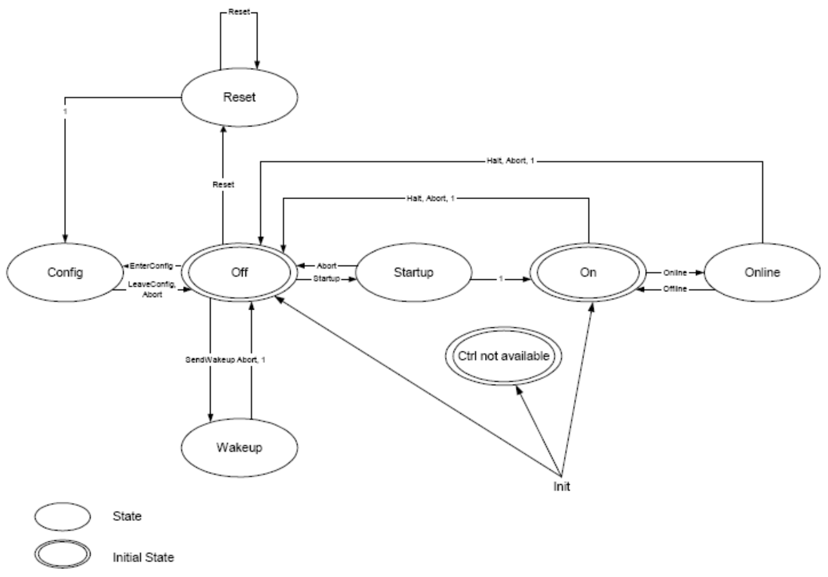
Code Output Path Ray\FlexRay1_91467d-v15\Generated_files

Buffer	Type	Channel	Slot	BC	CR	Pool	# Frame Triggerings
000	TX	A	003	0	1		1
001	TX	B	003	0	1		1
002	TX	A	004	0	1		1
003	TX	B	004	0	1		1
004	RX	A	005	0	1		1
005	RX	B	005	0	1		1
006	RX	A	006	0	1		1
007	RX	B	006	0	1		1
008	RX	A	007	0	1		1

Figure 10-12 shows the ECU configuration screen with the frame values. The Auto BA (Buffer Assignment) tab automatically assigns the buffer configuration for each slot. This has to be completed for each node. The code generation tab presents the output path of the generated code. Code has to be generated for each node.

Three files are generated; node.c cfg (configuration), node.h cfg and node memory cfg. These files are then included in the Softune workbench when creating and compiling a program. This ensures the frame parameters for the application will be the same as those configured in the designer.

10.3.3 COMMSTACK



Before synchronisation can occur a node has to be "online" and~~. This~~ is achieved through the COMMSTACK ~~COMMSTACK~~. Figure 10-13 shows the possible states from the *off state* to the *online state*. The states are presented in detail later in the chapter.

DECOMSYS COMMSTACK structure is made up of four main sections:

- The application (Configuration)
- COMMSTACK (Library)
- Hardware (FlexRay)
- Hardware (Configuration)

The application consists of all the specifications required before the application build takes place.

The COMMSTACK contains all of the library files ~~information data~~ that the host CPU requires to carry out its functions while being as modular as possible.~~.~~ This includes the connection schema for the CC (communications controller) hardware.

A FlexRay hardware node contains a minimum of one communication controller but can contain more if required.

The hardware configuration contains the device mapping and the reset configuration of the CC device(s). The static segment configuration data is required at pre-compile time.

The state model controls how the COMMSTACK behaves in any situation. In Figure 10-13, each state is contained in an oval while the arrows show the action taking place. Tables 10-1 – 10-7 show the different states of the COMMSTACK FlexRay driver.

10.3.3.1 Off State

Table 10-1: Off State

Off state	The FlexRay CC is not able to access the network nor is it configurable. This state is entered after the COMMSTACK is initialised if the CC is detected successfully and the cluster is not yet synchronised.	
	Reset	Reset performs a hard reset of the FlexRay CC
	EnterConfig	ENTERCONFIG ENABLES THE CC TO BE CONFIGURED
	SendWakeUpChA	SendWakeUpChA enables the transmission of the wake up pattern on channel A of the dedicated CC.

1. ~~ ~~
2. ~~Off state: The FlexRay CC is not able to access the network nor is it configurable. This state is entered after the COMMSTACK is initialised if the CC is detected successfully and the cluster is not yet synchronised.~~
3. ~~Reset performs a hard reset of the FlexRay CC~~
4. ~~EnterConfig enables the CC to be configured~~
5. ~~SendWakeUpChA enables the transmission of the wake up pattern on channel A of the dedicated CC.~~
6. ~~SendWakeUpChB enables the transmission of the wake up pattern on channel B of the dedicated CC.~~

10.3.3.2 Start-up State

Table 10-2: Start-up State

Start-up state	The FlexRay CC enters the Startup state. Either active or passive startup is initiated depending on configuration.	
	Abort	Returns back to the off state exiting the startup process.
	1	Action takes place once startup is

10.3.3.3 On State

Table 10-3: On State

On state	This state is entered once the CC and the cluster are successfully synchronised. Buffer access for COMMSTACK API functions are turned off	
	Online	Online state is entered once the on state has been successful.
	Halt	Halt immediately returns to the off state at the end of the current communication cycle.

10.3.3.4 Online State

Table 10-4: Online State

Online	The FlexRay controller is synchronised to the network and the software driver is able to access transmission and reception buffers.	
	Offline	Offline state is entered.
	Halt	Halt immediately returns to the off state at the end of the current communication cycle.
	Abort	Abort causes the state to be

10.3.3.5 Configuration State

Table 10-5: Configuration State

Config	THE FLEXRAY CONTROLLER CAN BE CONFIGURED.	
	LeaveConfig	LeaveConfig transition occurs when exiting the Config state. Transition takes us into the off state.
	Abort	Leaves the config state, enters the off state

10.3.3.6 Reset State

10.3.3.7 Wakeup State

10.3.3.8 Vector CANAlyzer.FlexRay

Table 10-7: Wakeup State

Wakeup	FlexRay controller transmits a wakeup pattern	
	Abort	Abort causes the transmission of the wakeup pattern to be aborted immediately and returns to the off state.
	1	Transition occurs once the wakeup pattern has been successfully completed

CANalyzer is a tool for analysing CAN and FlexRay networks. The version used in this research is 6.1.33 (SP3). The primary function of CANalyzer is to monitor traffic on the network (GmbH, 2007), other functions include;

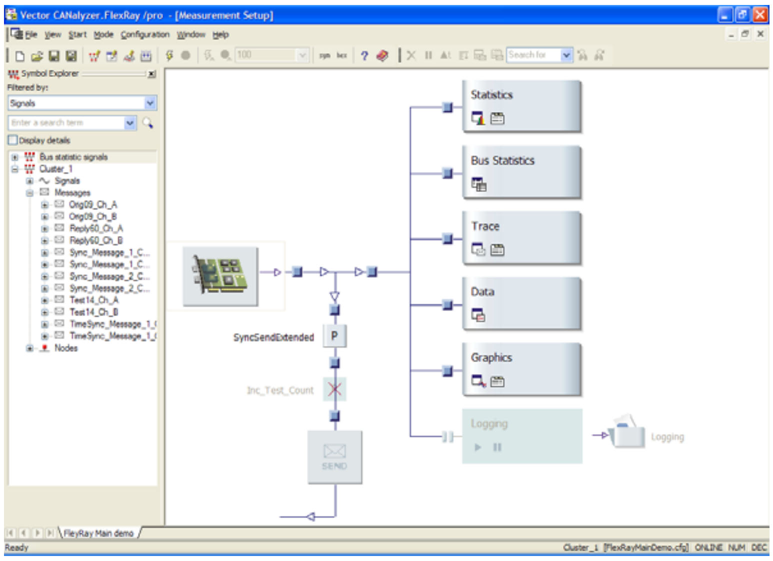
- Listen to bus data traffic
- Display data message segments
- Statistics on message frequency
- Insert prepared functional blocks such as filters generator blocks
- Record information for offline evaluations

CANalyzer contains increased functionality that enables the user to programmatically implement application specific parameters. This can be achieved through the use of function blocks (e.g. test case three) or by using CAPL (CANalyzer Programming Language). CANalyzer contains the necessary browser for creation, building and compilation of CAPL programs (GmbH, 2007).

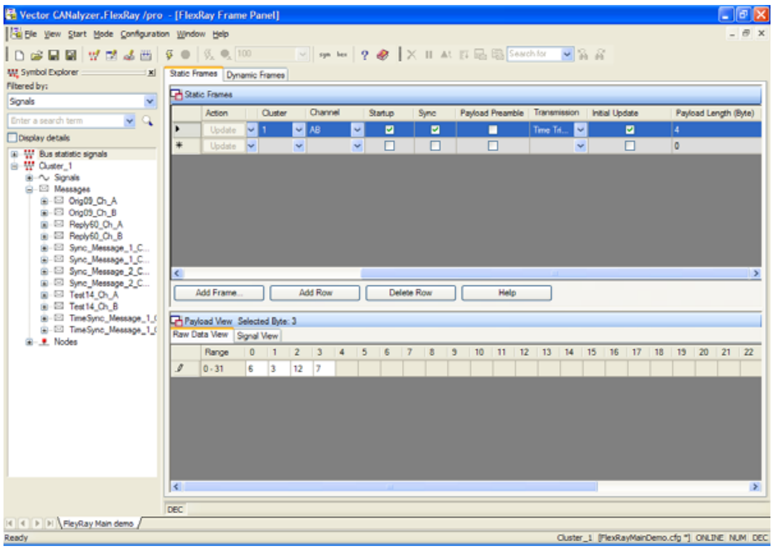
A Flow diagram (FlexRay example is illustrated in Figure 10-14) is used to show the functions of the bus monitoring function blocks such as; Statistics, Bus statistics, Trace, Data, Graphics and Logging. Table 10-8 gives an overview of each function block.

Table 10-8: Analysis Function Block Overview

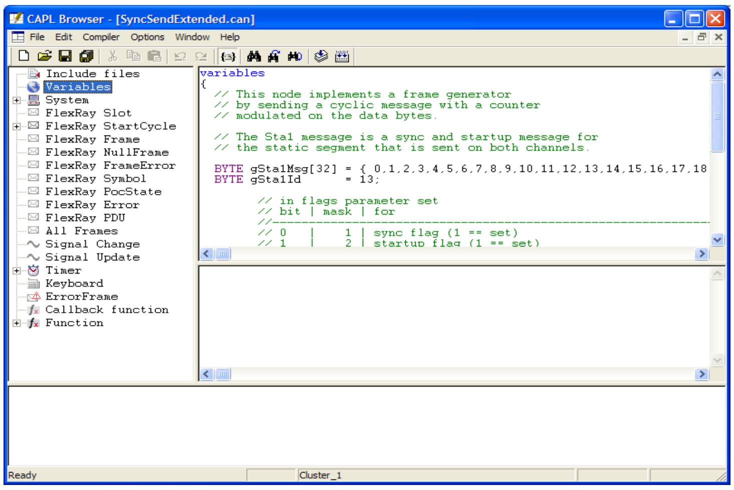
Function Block	**Comment**
Statistics	Shows the mean frequency or spacing of message/s.
Bus Statistics	Displays the statistics such as frames per second, frame errors and null frames to name a few parameters. CAN data is acquired from the CAN controller
Trace	Displays the trace data on input lines



Function blocks can be included to add data onto the network. This can be done two ways; inserting a FlexRay frame panel as seen in Figure 10-15. This allows the user to configure the frame parameters including the payload data and frame ID. The second ways is by inserting a CAPL function block and writing the system parameters in the CAPL browser (see illustration 10-16).



	Action	Cluster	Channel	Startup	Sync	Payload Preamble	Transmission	Initial Update	Payload Length (Byte)
▸	Update	1	AB	✓	✓	☐	Time Tri...	✓	4
*	Update			☐	☐	☐		☐	0



```
variables
{
  // This node implements a frame generator
  // by sending a cyclic message with a counter
  // modulated on the data bytes.

  // The Sta1 message is a sync and startup message for
  // the static segment that is sent on both channels.

  BYTE gSta1Msg[32] = { 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18
  BYTE gSta1Id      = 13;

        // in flags parameter set
        // bit | mask | for
        //----------------------------------------------------------
        // 0   |    1 | sync flag (1 == set)
        // 1   |    2 | startup flag (1 == set)
```

Figure 10-16: CAPL Browser Panel

One feature of CANAlyzer.FlexRay is that CAN and FlexRay can be analysed at the same time. This is achieved through the use of a CANcardXL for the CAN bus and the VN3600 FlexRay interface for the FlexRay bus. Both are produced by Vector.

10.4 Conclusion

This chapter presents the tools required to progress from the implementation of the CAN application to the verification of FlexRay properties to the final implementation of the FlexRay application. The features of the hardware tools required are presented, such as the Fujitsu development environment, CANcardXL and the VN3600 along with the any software tools used such as, DECOMSYS designer and Softune monitor debugger.

10.5 References

Dependable Computer Systems D, 2007, Designer PRO 4.3.0. In: *Designer pro, Designer Pro<Light> and Designer Pro<System>,*

Eggenbauer M, 2006, COMMSTACK<FLEXRAY> 1.8. Dependable Computer Systems)

Europe F M 2007 *SK-91F467-FLEXRAY User Guide FMEMCU-UG-910017-16*: Fujitsu Microelectronics Europe)

GmbH V I, 2007, CANalyzer Help.

GmbH V I, 2008, VN3600 Data sheet. In: *Vector Informatik GmbH,*

Manual F S C, 2006, FR FAMILY SOFTUNETM WORKBENCH OPERATION MANUAL for V6 CM71-00328-3E. Fujitsu)

Section IV - Testing & Results

11 System Model

11.1 Introduction

In this chapter the implementation models are presented. All preconditions to migration have been presented together with any configuration techniques required. The test environment and the test cases are described.

The theoretical implementation (test case 1) uses a TC (Traction Control) model. The physical implementation model (test case 2) used was an ACC (Adaptive Cruise Control) system. The final test case, test case 3, “The Verification of Time-Triggered Properties” was implemented using a straight forward task graph configuration. The hardware and software used in the test case implementations are presented along with the specified configurations.

11.2 Traction Control & Adaptive Cruise Control Summary

Traction control involves reducing the amount of slip between the tyre and the surface (Yoichi Hori, 1997) for example, road or ice. This is done by limiting the power being delivered to the wheel that is slipping. The same sensor that is used for the Anti-lock Brake Systems (ABS) can be used in determining slip.
Adaptive cruise control allows the vehicle to maintain a preset speed without the danger of hitting the vehicle in front if it stops. Once the driver sets the desired speed the vehicle will remain at that speed as long as there is no obstacle in front. If a vehicle in front is encountered travelling at a slower speed than your vehicle then the brakes are applied accordingly. A sensor at the front of the vehicle detects any obstacles. If the vehicle in front accelerates away your vehicle will accelerate back to its desired preset speed.

11.3 Application Models

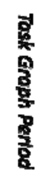
Test Case 1: Traction Control

The first test case will be strictly an abstract implementation. This will involve obtaining a CAN application and processing it through the migration framework to obtain the FlexRay frame parameters. This test case will demonstrate the framework's ability to produce viable FlexRay frame and cluster parameters. The abstract implementation only requires five stages of the process flow diagram to be undertaken. The five stages are highlighted in orange in Figure 11-1.

Stages

The TC model used was presented in (Aloul, 2005) and is illustrated in Figure 11-2. Each tasks function is described as in Table 11-1.

The TC model has ten tasks, with each task performing a ‘calculation’ or ‘action’ function. Communication between the tasks is demonstrated with connecting arrows showing the direction communication takes place in.



Due to the TC model not being implemented on the development environment there was no requirement to assign tasks to certain nodes.

Test Case 2: Adaptive Cruise Control

The second test case will be an experimental implementation of an ACC CAN application. Upon successful migration, precisely the same application will be implemented in FlexRay using the frame and cluster parameters obtained from this framework. This second test case, together with demonstrating the framework's ability to produce viable FlexRay parameters, is a different task graph configuration scenario compared to the first test case. This second test case also demonstrates that the parameters obtained can be implemented in a real FlexRay cluster. All seven stages of the flow diagram in Figure 11-1 are required in this test case.

The ACC model used in the experimental implementation case is a modified version of a model presented by (Riis, 2007) and (Madsen, 2007). For this thesis, the application was modelled using a two node system. The system has six tasks, and each task directly interacts with another task on the application. This is illustrated in Figure 11-3. The direction of message transfer is indicated by the direction of an arrow. Each task is assigned to a node with the action tasks assigned on node 1 (N1) and computational tasks assigned on node 2 (N2). The task functions are presented in Table 11-2.

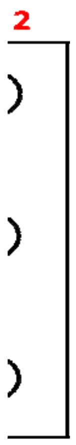
Table 11-2: Adaptive Cruise Control Task Functions

Function	**Task**
Obtain Vehicle Velocity	*T1*
Obtain Distance to Vehicle in Front	*T2*
Calculate Relative Speed of Vehicle in Front	*T3*
Calculate Desired Velocity	*T4*

Figure 11-3 illustrates the ACC model in a task graph format.

ntation

Communication occurs between tasks via messages. Messages are transmitted on the bus. The ACC application data is configured to transmit in the ST segment. Precedence constraints were defined in the application. An example of a precedence constraint is where m3 could not be transmitted unless m1 and m2 have been received in Figure 11-3. The tasks are assigned to the same nodes in the CAN and FlexRay configurations. T1 transmits m1 in slot 1; T2 transmits m2 in slot 2 and so forth. Figure 11-4 illustrates the block diagram representation of the ACC with tasks assigned to each node.



In FlexRay, when allocating the ACC application data to either the ST or DYN segment, the following approach is taken in this test case. All messages that interact in a critical application are placed in the deterministic ST segment. All data that is not used in a critical application is placed in the non-deterministic DYN segment.

Test Case 3: Verification of Time-Triggered Properties

Tests case three, the verification of time-triggered properties, demonstrates the effectiveness of the FlexRay protocol at guaranteeing constant application execution times. This test case will involve undergoing all stages of the model flow diagram in Figure 11-1. The CAN and FlexRay data shall be generated using CANalyzer.FlexRay. This test case demonstrates that if prior consideration is not taken when deciding message IDs, this can degrade CAN performance. As long as the messages in FlexRay are comparable in term of priority and IDs, it is demonstrated that the time-triggered properties of FlexRay result in no performance degradation when compared to CAN.

For the final test case "Verification of Time-Triggered Properties" the task graph was composed as per Figure 11-5. Because the simplistic task graph is not from a real world example each task does not have a predefine function. All associated task graph parameters are within the ranges of the real world parameters, as used in the two previous test cases (TC and ACC).

For this test case all application data is allocated to the ST segment. Any additional data required to increase the busload is also allocated to the ST segment. This is done so as to demonstrate that as long as each message has an allocated slot in the ST segment, transmission shall not be delayed. There is no DYN data transmitted in this test case.

11.4 Hardware

Each node is implemented on the SK 91F467D development board (two nodes in total hence two boards were required). The application is flash loaded onto each node individually and the system is configured as per the reference application.
For the CAN implementation the CAN channels are directly linked together using CAN cables in the traditional CAN bus structure format. A splice from the CAN cable

connects to the CANcardXL which is inserted into the laptop containing CANalyzer.FlexRay, for monitoring and bus loading purposes.

The physical layer is as per the FlexRay specifications through the use of the TZM FlexTiny as described in Chapter 10. Node1 (N1) CHA is connected to passive star A and N2 (Node2) CHA is connecter to passive star A. N1 CHB is connected to passive star B and N2 CHB is connecter to passive star B. Passive star A and B are then connected to the VN3600 FlexRay interface via FlexRay cabling. The cabling from the node to the passive star is done using CAN cables as this does not affect performance in this test case. If wake up symbols are a key feature of the FlexRay frame is it suggested to use FlexRay cabling instead. The VN3600 FlexRay interface connects to the laptop containing CANalyzer via a USB connection. Both the CAN and FlexRay cables are 9 pin d-sub connections.

The physical test environment used only applies to the ACC experimental implementation, test case two. For test case three, CANalyzer was used to generate the CAN data while the FlexRay data was generated using CANalyzer and the VN3600 FlexRay interface. The interface was required to coldstart the node.

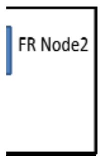
All tools and applications used were presented in Chapter 10.

11.4.1 CAN Hardware Configuration

The CAN configuration was set up by connecting two CAN nodes together with CAN specified cables and D9 connectors. A T-bus was used to splice off the CAN signal which was fed into the CANcardXL adaptor card. This configuration is illustrated in Figure 11-6.

11.4.2 FlexRay Hardware Configuration

The FlexRay configuration was somewhat different from the CAN configuration. This was to present a more realistic bus structure configuration. The FlexRay passive star was introduced to demonstrate the extra configuration options of FlexRay over CAN. The traditional CAN configuration is the single bus structure. Two FlexRay nodes were implemented using the FlexRay development kits. Both channels of FlexRay were utilised. Channel A on node 1 was connected with channel A on node 2 via the FlexRay passive star. Channel B on node 1 was connected with channel B on node 2 via a separate passive star. The two channels cannot be crossed over because this would cause the CRC to flag an error. Channel A and channel B on the passive star were connected to the VN3600 FlexRay interface. The physical layer interfaces are inserted into the designated slots (on the development boards) to meet the specification standards for the physical layer. The FlexRay VN3600 interface connects to the laptop via a USB connector and data is extracted via CANalyzer.FlexRay. The cables used are the same as for the CAN test case except one. This is the cable that connects from each passive star into the FlexRay interface. CAN cables were sufficient for this test case but in a case where wake up symbols are transmitted it is recommended to use FlexRay specified cabling. Without this there is a probability that the FlexRay signal would not get decoded correctly. This cable was supplied with the FlexRay interface. The FlexRay test configuration is illustrated in Figure 11-7.



11.5 Software

The software required for implementing the respective applications are presented in this section. One key software component is the application code. This defines the applications features such as precedence constraints. The protocols that the applications operate on are central to the research. These protocols are CAN and FlexRay and are discussed in depth in Chapters 3 and 4 respectively. The features and merits of each have previously been explained in depth. Each protocol's individual parameters are provided through the CAN controller for CAN applications, and the communication controller for FlexRay applications. The ACC application was implemented in C code. In situations where additional data is loaded onto the bus this data has been generated by CANalyzer.FlexRay. This allowed for the configuration of the message ID, period and payload. Other options include having a transmission cyclic or transmission activated upon transmission of a defined message. The additional frames (CAN or FlexRay) are not incorporated into the initial system design. This is because this data is used to demonstrate how FlexRay is able to handle additional data quantities without adversely affecting the performance of the preconfigured ACC application data.

11.6 Extraction Method

The CAN application is initially defined and represented as a task graph. The next step is to extract the parameters through the framework. In these three test cases the initial CAN data figures were input to Microsoft Excel. This application was used because it was easily accessible and simple to use. Calculations could be easily processed and through the modification of any equations elements, this allowed for any changes to propagate through all other equations that were associated with the parameter. The graphing of data (such as payload data) was also carried out using Microsoft Excel.

11.7 Verification of Parameter Consistency

Verification is required before parameters can be accepted as valid. This is accomplished using the DECOMSYS designer application whose features are explained in Chapter 10. If the ideal parameters obtained from the framework are not implementable these will be exposed in the designer tool and the implementable value is displayed. As can be the case, the value obtained from the framework might need slight tweaking due to another constraints being violated. These constraints can be calculated manually since each constraint and all the sub elements are presented in appendix B of the FlexRay specifications v2.1.

11.8 Validation Check

To confirm that migration is successful the deadlines of each CAN task are compared with those obtained through the FlexRay implementation. This alone is not a sufficient check as the application deadline has to be met also. The CAN and FlexRay application times are compared in addition. Busloads are examined to determine if there is a link between deadlines being missed and increasing busloads. CAN and FlexRay bus loads are increased to determine if FlexRay has the extra performance capacity it has been claimed to possess.
For test case three “Verification of Time-Triggered Properties”, verification is carried out solely through a comparison of application execution times

11.9 Conclusion

This chapter presents the structure, implementation and purpose of the three test cases. The applications used in obtaining the results are presented along with the configuration of the CAN and FlexRay test environment. The chapter concludes by presenting the methods used in determining if migration was successful and necessary.

11.10 References

ALOUL, N. K. A. F. (2005) 'The Synthesis of Dependable Communication Networks for Automotive Systems' SAE Internalional 2005,

MADSEN, J. P. K. A. S. H. (2007) Design Optimisation of Fault-Tolerant Event-Triggered Embedded Systems. IN POP, P. (Ed.) Kongens Lyngb.

RIIS, P. (2007) Simulation of a Distributed Implementation of an Adaptive Cruise Controller. *Department of Computer and Information Science.* Link¨oping University.

YOICHI HORI, Y. T. A. Y. T. (1997) Traction Control of Electric Vehicle Based on the Estimation of Road Surface Condition

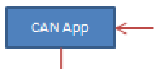
12 Framework Implementation Procedure:

12.1 Introduction

This chapter examines how the process of migration was defined and presents the implementation procedure that was undertaken for the three test cases. The abstract implementation is presented initially, followed by the experimental implementation and finally, concluding with the verification of time-triggered properties implementation. The abstract implementation is not implemented in the physical development environment. It provides an example of how FlexRay's frame and cluster parameters can be extracted from the migration framework. The results determining if migration is successful are presented in Chapter 13 only for the experimental implementation that was applied to the development environment and test case three, verifying FlexRay's time-triggered properties.

12.2 Abstract (TC) Implementation (Test Case 1)

The model chosen for the abstract application (Figure 12-1) presented a high level of complexity in relation to the number of branches and nodes it contained. While a simpler task graph model (containing less branches and nodes) would also have sufficed (see test case three), by using this more complex model the framework's strength in dealing with increased complexity is demonstrated. The associated application task data was extracted from vehicles made available by the industrial partner (SEWS-E Ystrad). Choosing the CAN application is step one of the migration framework development stage as illustrated by Figure 12-1.



12.2.1 Task Graph Model

Step two is task graph abstraction (Figure 12-2). The task graph model was adapted from (Aloul, 2005) and presented in detail in Chapter 11. By using the traction control model it presents sufficient detail in relation to the number of tasks and branches, to provide a realistic structure. Figure 11-2 illustrates the traction control task graph structure. The CAN bus speed of 125kbit/s is used in calculations. Calculations are based on FlexRay data bus rate of 10Mbit/s.

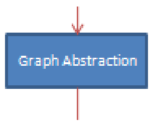
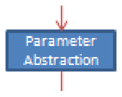


Figure 12-2 : Migration Step Two

12.2.2 Parameter Analysis

The CAN data used in this abstract case was obtained from an electric Smart Car (ST data) and a Peugeot 207 (DYN data). The data was logged using CANalyzer. The timings presented are actual task periods used in mass produced vehicles. This method of obtaining initial CAN parameters demonstrates how the framework can operate once the task's parameters are presented in the message domain. This proves that the framework's success is not application configuration dependant. This relates to stage three of the framework development (Figure 12-3), extracting the CAN parameters.



12.2.3 Execution Time

One final parameter, jitter, was required before RTA could be employed to obtain the CAN task's *WCETs*. The jitter value varies depending on the CAN controller and also varies for each individual message. Specialists equipment such as that by Agilent and Tecktronix can be bought that extract jitter times. This specialist equipment was not readily available during this research. Jitter values used were taken from (Burns, 1994). The jitter values were obtained from the Intel 82527 stand-alone CAN controller. It was considered more appropriate to use real jitter values even if they had come from a different CAN controller than just randomly selecting jitter times. Having the message periods and the message sizes (in bytes) allowed RTA to be use in determining the *WCET* of the messages. Using Equation 12-1 as presented in (Robert I. Davis, 2007) the response time is calculated. This response time is taken as the WCET of each task.

$$Rt_m = J_m + w_m + C_m$$

Equation 12-1: Task Response Time

Each element of Equation 12-1 is explained section 5.9.2. The initial data required to obtain the response time is presented in Table 12-1. A CAN bus of 125kbit/s was chosen. This results in a bit time of 8µs. The communication time of each message at the selected size in bits is calculated from Equation 12-2, as presented in Table 12-1.

$$C_m = (55 + 10s_m)\tau_{bit}$$

Equation 12-2: Message Transmission/Communication Time

Table 12-1: Parameter C_m & J_m Identification

Task	**Period (ms)**	**Message Size (Bytes)**	**Stuffed Message Size (Bits)**	C_m *(ms)*	J_m *(μs)*
T1	1	8	135	1.08	600
T2	2	5	105	0.84	700
T3	1	8	135	1.08	400
T4	1	8	135	1.08	800
T5	5	3	85	0.68	1100
T6	1	7	125	1.00	900

Jitter values are taken directly from (Robert I. Davis, 2007) and are shown in Table 12-1. The blocking delay B_m is calculated as per Equation 5-8 and the queuing delay w_m is calculated from Equation 5-10. The initial blocking delay is zero. Due to the possibility of multiple iterations the queuing delay will not be the same in every cycle therefore multiple iterations are required. The number of iterations will depend on the number of cycles required to determine that the WCRT value is not continually increasing. The number of decimal places in this value will also affect when a value is determined to have not changed. For example if one decimal place is used and a value 2.7*ms* does not change after the defined number of iterations, it can be determined that the WCRT is not continually increasing. If six decimal places are used instead it is observed that between the first two iterations the value changes from 2.766566*ms* to 2.778301*ms*, thereby taking more iterations before it can be determined to have remained constant. In this test case five iterations to six decimal places (values in *ms*) were required until two consecutive values did not change. Table 12-2 displays w_m queuing delays that are used in determining the response times.

Table 12-2: w_m Five Iterations of the Queuing Delay

Queue Delay	***W_1 (ms)***	***W_2 (ms)***	***W_3 (ms)***	***W_4 (ms)***	***W_5 (ms)***
w1	1.086566	1.09830	1.098428	1.098429	1.098429
w2	1.089540	1.10127	1.101402	1.101403	1.101403
w3	1.093946	1.10568	1.105808	1.105809	1.105809
w4	1.102673	1.11440	1.114534	1.114536	1.114536
w5	1.104180	1.11591	1.116041	1.116043	1.116043
w6	1.113260	1.12499	1.125121	1.125123	1.125123
w7	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Once the three parameters have been acquired for Equation 12-1 this allows the WCRT for each message to be calculated. The figures required are presented in Table 12-3. There are five iterations provided as this is the point at which the response time stabilises.

Table 12-3: Worst Case Response Time

Msg	**Rt_{m1} *(ms)***	**Rt_{m2} *(ms)***	**Rt_{m3} *(ms)***	**Rt_{m4} *(ms)***	**Rt_{m5} *(ms)***
m_1	2.766566	2.778301	2.778428	2.778429	2.778429
m_2	2.629540	2.641275	2.641402	2.641403	2.641403
m_3	2.573946	2.585681	2.585808	2.585809	2.585809
m_4	2.982673	2.994408	2.994534	2.994536	2.994536
m_5	2.884180	2.895915	2.896041	2.896043	2.896043
m_6	3.013260	3.024995	3.025121	3.025123	3.025123
m_7	2.293493	2.305228	2.305355	2.305356	2.305356

12.2.4 Calculating Slack

The slack requires calculating and reallocating to obtain the final intermediate release (r_i) and deadline (d_i) times. The initial r_i time is zero *ms* and the final deadline D_i is 2.6seconds. D_i is obtained by summing all task periods. The slack on each path is illustrated in Table 12-4. Once the slack per path is known, the r_i and d_i times of the intermediate tasks can be obtained. Path P4 is the longest path through the task graph (Figure 12-4).

The application deadline is 2.6 seconds which is obtained by summing the task periods.

12.2.5 Final Task Graph Parameters

The recalculated r_i and d_i values are presented in Table 12-5. These figures include slack reallocation.

Table 12-5: Task Graph Parameters

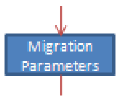
Task	WCET (ms)	r_i (ms)	d_i (ms)	Slack per task (ms)
T1	2.778429	0.000000	2.778429	647.4030
T2	2.641403	0.000000	2.641403	647.4373
T3	2.585809	0.000000	2.585809	647.4512
T4	2.994536	0.000000	2.994536	647.3490
T5	2.896043	2.994536E	656.2341	647.3490
T6	3.025123	0.000000	3.025123	647.5071
T7	[illegible]	[illegible]	[illegible]	[illegible]

The first validation check performed as per Equation 9-7 can be carried out. These results are displayed in Table 12-6. A green box represents a valid configuration and red box represents an invalid configuration.

Table 12-6: Final Task Graph Parameters

Task	Re-cal r_i (ms)	Re-calc d_i (ms)	Validation Check (ms)	
T1	0.000000	650.1814	2.778429<650.1814	🟩
T2	0.000000	650.0787	2.641403<350.0787	🟩
T3	0.000000	650.0370	2.585809<650.0370	🟩
T4	0.000000	650.3435	2.994536<650.3434	🟩
T5	650.3435	1300.589	2.896043<(1300.589-650.3435)	🟩
T6	0.000000	650.5322	3.025123<650.5322	🟩
T7	[illegible]	[illegible]	[illegible]	🟩

The figures in Table 12-6 are used in migrating to FlexRay. This is stage four (Figure 12-5) in the migration development flow chart.

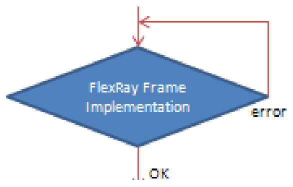


12.2.6 Message Deadlines

Each messages transmission deadline is represented by Equation 9-8 and the transmission delay is given in Equation 9-9. The transmission delay is the delay on the CAN bus. These values are presented in Table 12-7.

12.2.7 Payload Configuration

Sections 12.2.7 to 12.3.2 are required for stage five of the migration as per the flowchart (Figure 12-6).



The CAN message sizes were presented in Table 12-1. For this Framework an overhead size of 14 Bytes is configured. This comprises of;

- 5 Bytes Header
- 3 Bytes CRC
- 4 Bytes Clock and Security (there is a minimum variance required between messages from different nodes so there is no overlap. Includes a safety margin of $4\mu s$)
- 2 Bytes TSS

The total FlexRay frame size is composed of the overhead and the payload. The total number of frames required to transmit the entire payload data at a chosen payload size can be found. This figure is rounded up to the nearest integer value. This value can be used to determine the total data for transmission including payload and overhead data. These figures are presented in Table 12-8. The number of messages required column is presents how many FlexRay messages would be required to transmit the CAN data at the chosen payload. This includes all overheads. It is calculated by dividing the CAN payload by the FlexRay payload.

Table 12-8: Payload Calculations

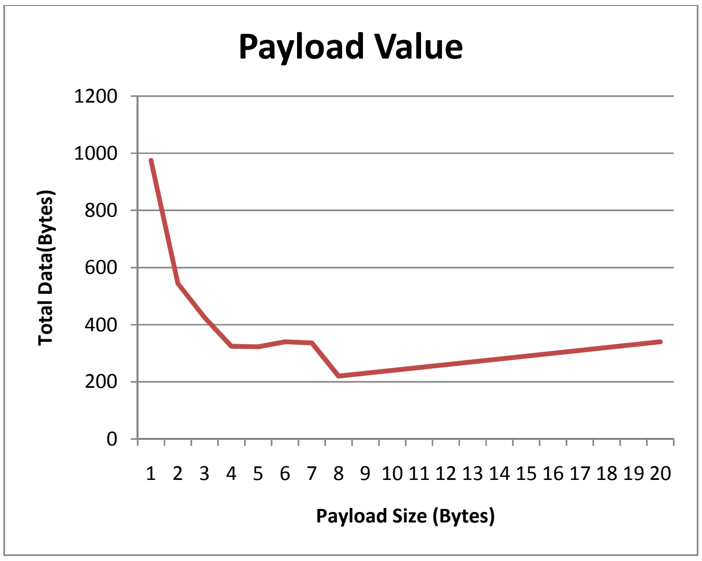
Payload Size	FlexRay Frame Size	Number of Messages Required	Total Bytes
1	15	65	975
2	16	34	544
3	17	25	425
4	18	18	324
5	19	17	323
6	20	17	340
7	21	16	336
8	22	10	220
9	23	10	230
10	24	10	240
11	25	10	250
12	26	10	260
13	27	10	270

By focusing specifically on the FlexRay frame in the abstract implementation the payload is configured as a two-byte-word. This allows only even payload values be considered for configuration. This results in Equation 12-3. The condition max occurs once there have been ten consecutive instances of *D* increasing.

$$D = Frame\,Size \times Cf_{FR} \qquad \text{For even } pd_j \text{ values of } \; pd_j = \{j < j+1......j+10\}$$

Equation 12-3: Revised Total FlexRay Data

By graphing the Cycle Data versus Payload Size results in the graph in Figure 12-7.



A payload size of 12 Bytes results in a frame size of 26 Bytes being chosen. The FlexRay bus speed is specified at 10Mbit/s.

12.2.8 Static Slot Size

The size of the static slot is calculated as per Equation 9-13 and is verified below.

$$gdStaticSlot = \left\lceil \frac{12Bytes + 14Bytes}{10Mbit/s} \right\rceil$$

$$gdStaticSlot = \left\lceil \frac{26Bytes}{10Mbit/s} \right\rceil = 21\mu s$$

At a payload size of 12 Bytes the smallest configurable ST slot size is 35μs due to configuration restrictions of the DECOMSYS designer program.

12.2.9 Discretising

The smallest message period whose value is obtained from the minimum *td(m$_i$)* value is 647*ms*, rounded down. With each ST slot size of 35µs this results in a discretised *td(M$_i$)* value of 18485 slots.

12.2.10 Periodicity and Distance Requirement

To configure the FlexRay frame and the ST segment so as to assist the DYN segment data in gaining access to the bus, the message periods are modified as explained in section 9.8.6.

To meet these requirements the p_{min} value is modified (down from 647*ms*) to 640*ms*. This results in a minimum FlexRay frame of 1.25*µs*. With 10 ST slots of 35µs each, the outcome is a ST segment size of 350*µs*. This coupled with an NIT of 21MT and each MT being 1*µs*, results in a DYN segment size of 879*µs*.

Proving that and frame size meets the periodicity and distance requirement:
1.25ms→2.5ms→5ms→10ms→20ms→40ms→80ms→160ms→320ms→640ms

Equation 9-15 provides a formal validation as also proven below.

$$2^9 \cdot 1.25ms \le 640ms < 2^{10} \cdot 1.25ms$$
$$640ms \le 640ms < 1280ms$$

12.3 Dynamic Segment Verification

The DYN segment calculations are based on (Traian Pop, 2006) and explained in detail in section 9.9. With a DYN segment size of 879μs and a minislot size of 6μs this results in 146 minislots. This number of 146 minislots allocates enough slots to transmit each DYN message. There is enough capacity to increase the minislot size if required.

12.3.1 Initial Dynamic Data

The initial CAN data for transmission in the DYN segment is presented in Table 12-9.

Table 12-9: Initial Dynamic Data

Task	CAN Message Size (Bytes)	Stuffed Message Size (Bits)	Task Period (ms)
T1	8	135	20
T2	8	135	20
T3	8	135	10
T4	8	135	20
T5	8	135	20
T6	8	135	20
T7	8	135	20
T8	8	135	20

12.3.2 Dynamic Segment Analysis

The Stuffed Message Size column was calculated as per stuffed message sizes in Table 12-1. The dynamic segment RTA is carried out using a modified version of Equation 12-1 as represented by Equation 12-4.

$$R_m(t) = C_m + \sigma_m + w_m(t)$$

Equation 12-4: Response Time Analysis Equation

The value C_m (Table 12-10) can be calculated as per the ST segment using the bit time. First the total FlexRay data including the overhead is calculated. The overhead is calculated using the same method as that used for ST segment data. Once the total communication time of all FlexRay data is know, it can then be assessed whether there is enough time allocated in the DYN segment for the transmission of all DYN data in the best case scenario, with no delays. Column two, FlexRay frame size, includes any overhead. The bit time is 0.1*μs*.

Table 12-10: Dynamic Communication Time

Message	**FlexRay frame Size (Bytes)**	**C_m (μs)**	**Total C_m (*μs*)**
m1	22	17.6	
m2	22	17.6	
m3	22	17.6	
m4	22	17.6	
m5	22	17.6	
m6	22	17.6	
m7	22	17.6	204.8
m8	22	17.6	
m9	20	16.0	

The total communication time of 204.8μs is obtained from equation 12-5 where *n* is the number of DYN messages.

$$TotalDYN\ C_m = \sum (C_{m1} \ldots C_{mn})$$

Equation 12-5: Total DYN Communication Time

This provides enough time to transmit the DYN data if no blocking occurs. There are 879μs available per cycle.

The next value for defining is *δ_m*, the delay caused by a message being generated just after its slot has passed. This results in Equation 12-6.

$$\sigma_m = FR(t) - (ST_{bus} + C_m + NIT)$$

Equation 12-6: δ_m Delay

The parameters for each message delay δm and associated parameters are shown in Table 12-11.

Table 12-11: δ_m Parameters

Message	***FR(t)(μs)***	***ST_{bus} (μs)***	***C_m (μs)***	***NIT (μs)***	***δ_m (μs)***
m1			17.6		861.40
m2			17.6		861.40
m3			17.6		861.40
m4			17.6		861.40
m5			17.6		861.40
m6			17.6		861.40
m7	1250	350	17.6	21	861.40
m8			17.6		861.40
[illegible]			[illegible]		[illegible]

The final delay variable for calculation is *w_m*, the delay caused by the transmission of ST messages and higher priority DYN messages. This is illustrated in Equation 12-7.

$$w_m(t) = ST_{bus} + hp(m) + ms(m) + (pLatestTx.gdMinislot) + NIT$$

Equation 12-7: w_m Delay

The $w_m(t)$ parameters are presented in Table 12-12.

Table 12-12: $w_m(t)$ Parameter

Message	*hp(m) (μs)*	*pLatestTx.gdminislot (μs)*	*Ms(m) (μs)*	*$w_m(t)$ (μs)*
m1	0.0000	48	0.0000	419
m2	17.600	48	6.0000	443
m3	35.200	48	12.000	466
m4	52.800	48	18.000	490
m5	70.400	48	24.000	513
m6	88.000	48	30.000	537
m7	105.60	48	36.000	561
m8	123.20	48	42.000	584

The *pLatestTx* value was 138 minislots, which was obtained from the designer tool. Subtracting *pLatestTx* this from the total number of minislots (146) results in a value of 8 minislots. These 8 minislots are unusable for transmission. Using Equation 9-23 allows the *ms(m)* value to be obtained.

To obtain the final worst case response time, sum the three parameters (C_m, δ_m and w_m) for each message. This gives the resulting delays as displayed in Table 12-13.

Table 12-13: Dynamic Response Times

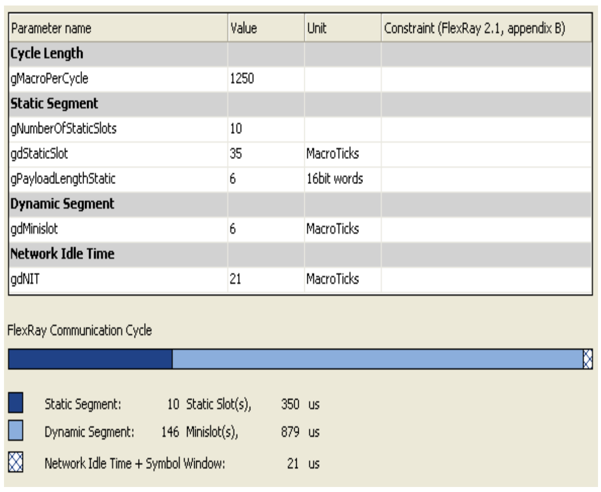
Message	*C_m (μs)*	*δ_m (μs)*	*$w_m(t)$ (μs)*	**R_m(t) (ms)**	**Frame Cycles**
m1	17.6	861.40	419	1.2980	1.0384
m2	17.6	861.40	443	1.3216	1.05728
m3	17.6	861.40	466	1.3452	1.07616
m4	17.6	861.40	490	1.3688	1.09504
m5	17.6	861.40	513	1.3924	1.11392
m6	17.6	861.40	537	1.4160	1.1328
m7	17.6	861.40	561	1.4396	1.15168
m8	17.6	861.40	584	1.4632	1.17056

12.4 Final Abstract Case Parameters

Using the framework, migration is undertaken commencing with the CAN application as illustrated in Figure 11-2. The FlexRay parameters are obtained by following the procedural steps. The FlexRay parameters obtained are:

- Frame Length ($1250\mu s$)
- Static Segment Size ($350\mu s$)
- Payload Size (6 2-byte-words)
- Static Slot Size ($35\mu s$)
- NIT ($21\mu s$)
- DYN Segment Size ($879\mu s$)

The Parameters as they are configurable in the designer tool are illustrated in Figure 12-8.



Parameter name	Value	Unit	Constraint (FlexRay 2.1, appendix B)
Cycle Length			
gMacroPerCycle	1250		
Static Segment			
gNumberOfStaticSlots	10		
gdStaticSlot	35	MacroTicks	
gPayloadLengthStatic	6	16bit words	
Dynamic Segment			
gdMinislot	6	MacroTicks	
Network Idle Time			
gdNIT	21	MacroTicks	

FlexRay Communication Cycle

■	Static Segment:	10 Static Slot(s),	350	us
□	Dynamic Segment:	146 Minislot(s),	879	us
▨	Network Idle Time + Symbol Window:		21	us

From the framework a FlexRay frame size of 1.25*ms* is acquired. This is composed of a ST segment of 350μs (10 slots of size 35μs), a DYN segment size of 879μs (146 slot of 6μs in size) and a NIT of 21μs.

These parameters would be used to configure the FlexRay frame but this step is not implemented in this test case (it is used in test cases two and three).

12.5 Experimental Implementation (ACC) (Test Case 2)

While the above traction control model proved that it is possible to successfully abstract the parameters necessary to configure a FlexRay frame, these results were not implemented in hardware. This is required to back up the claim that the parameters obtained allow a FlexRay application to operate successfully meeting deadline, timing and busload constraints. This is proven using an Adaptive Cruise Control (ACC) system. A completely new application, that resulted in a different (to the Traction Control application previously used) task graph configuration, was used to demonstrate that the previous migration was not a once off chance occurrence. The results validating deadlines and busloads are presented in Chapter 13.

The ACC model used is also found in (Madsen, 2007) and (Riis, 2007). The model was modified slightly to increase the complexity. By adding a precedence constraint that the messages of Task 1 and Task 2 (in any order) have to be received by Task 3 before Task 3 can proceed. This results in the ACC task graph as illustrated in Figure 11-3. The CAN bus bit rate is 125kbit/s and the FlexRay bus used is specified at a bit rate of 10Mbit/s

12.5.1 Parameter Analysis

In (Madsen, 2007) the author provides the WCET values. The initial CAN parameters used in this test case are presented in Table 12-14. The maximum message sizes are used in all messages. The initial release time r_i is 0*ms* and the final task graph deadline D_i is 120*ms*.

Table 12-14: CAN Initial Parameters

Task	WCET (ms)	Deadline (ms)	Period (ms)	Size (Bytes)	Node
T1	0	20	20	8	1
T2	0	20	20	8	1
T3	6	20	20	8	2
T4	2	20	20	8	2

The tasks were assigned to a node on the basis of the function performed. The 'action' tasks were placed on node 1 and the 'computational' tasks were placed on node 2.

12.5.2 Intermediate Task Values

To be able to find the intermediate task deadlines the slack requires reallocation equally among all tasks. There are only two possible paths in this task graph configuration. Table 12-15 presents the slack per task on each path.

Table 12-15: Obtaining Slack Parameter

Path	Tasks on Path	Total Execution Time (ms)	Final Slack per Path per Task (ms)
P1	T1,T3,T4,T5, T6	16	17.3333
[illegible]	[illegible]	[illegible]	[illegible]

As can be seen in Table 12-16 a slack of 17.33*ms* is to be reallocated equally among all tasks.

Table 12-16: Task Graph Parameters

Task	WCET (ms)	Ri (ms)	Di (ms)	Slack (ms)
T1	0	0	0	17.3333
T2	0	0	0	17.3333
T3	6	0	6	17.3333
T4	2	6	8	17.3333
T5	6	8	14	17.3333

Equation 9-7 verifies the task graph parameters as illustrated in Table 12-17.

Table 12-17: Final Task Graph Parameters

Task	Re-cal r_i (ms)	Re-calc d_i (ms)	Validation Check (ms)	
T1	0	17.3333	0<17.33333	🟩
T2	0	17.3333	0<17.3333	🟩
T3	34.6666	57.6666	6<(57.6666-34.6666)	🟩
T4	57.6666	76.9999	2<(57.9999-76.9999)	🟩
[illegible]	[illegible]	[illegible]	[illegible]	🟩

12.5.3 Message Deadlines

Each messages transmission deadline is represented by Equation 9-8 and the transmission delay is given in Equation 9-9. These values are presented in Table 12-18.

Table 12-18: Message Deadlines

	Transmission Deadline (ms)	Transmission Delay (μs)
m1	17.333	512
m2	17.333	512
m3	17.000	512
m4	17.333	512

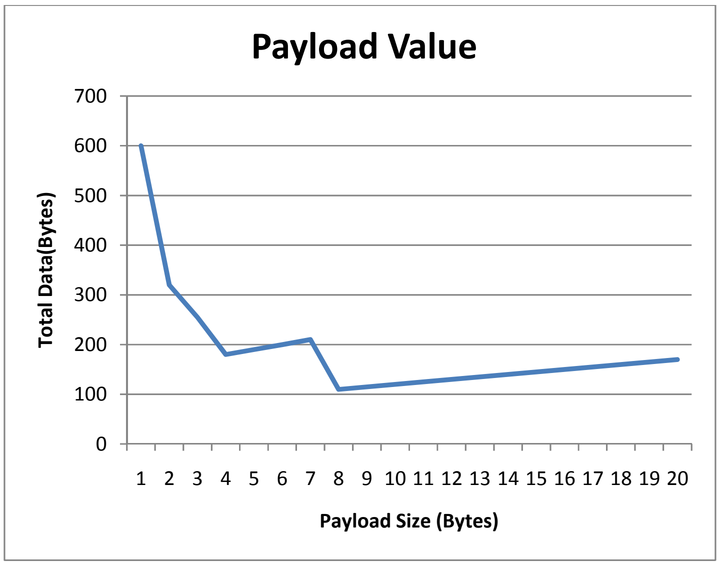
12.5.4 Payload Configuration

The overhead is defined as 14 Bytes as specified in section 12.2.7. As previous, the payload is chosen heuristically depending on the total number of bytes transmitted per payload and frame size. Table 12-19 presents the payload calculations.

Table 12-19: Payload Calculations

Payload Size	FlexRay Frame Size	Number of Messages Required	Total Bytes
1	15	40	600
2	16	20	320
3	17	15	255
4	18	10	180
5	19	10	190
6	20	10	200
7	21	10	210
8	22	5	110
9	23	5	115
10	24	5	120
11	25	5	125
12	26	5	130
13	27	5	135

Graphing the Cycle Data versus Payload Size results in that graph in Figure 12-9.



The payload size of 10 Bytes is chosen which results in a frame size of 24 Bytes including the calculated overhead.

12.5.5 Static Slot Size

The size of the static slot is calculated as per Equation 9-13 and verification is presented below. A slot size of 19.2μs is obtained but this is rounded up to 20μs.

$$gdStaticSlot = \left\lceil \frac{10Bytes + 14Bytes}{10Mbit/s} \right\rceil$$

$$gdStaticSlot = \left\lceil \frac{24Bytes}{10Mbit/s} \right\rceil = 20\mu s$$

At a payload size of 10 Bytes the smallest configurable ST slot size obtained using DECOMSYS designer is 33μs. For the purpose of implementation a static slot size of

40μs is chosen as it allowed a uniform fit for slot delays, otherwise a ST slot size of 33μs is chosen.

12.5.6 Discretising

The smallest message period whose value is obtained from the minimum $td(m_i)$ value is 14ms. With each ST slot size of 35μs this results in a discretised $td(M_i)$ value of 400 slots.

12.5.7 Periodicity and Distance Requirement

To configure the FlexRay frame and the ST segment, so as to assist the DYN segment data gaining access to the bus, the message periods are modified as explained in section 9.8.6.

To meet these requirements the p_{min} value of 14ms is chosen. This results in a minimum FlexRay frame of 1.75ms. With 6 ST slots of 40μs each this results in a ST segment size of 240μs. This coupled with an NIT of 25MT with each MT being 1μs, results in a DYN segment size of 1485μs.

Proving that and frame size meets the periodicity and distance requirement:
1.75ms→3.5ms→7ms→14ms

Equation 9.15 provides a formal validation as proved below.

$$2^3 \cdot 1.75ms \leq 14ms < 2^4 \cdot 1.75ms$$
$$14ms \leq 14ms < 28ms$$

12.5.8 Dynamic Segment Verification

The dynamic segment size is 1485μs as worked out in the previous section. This works out at 247 minislots with each minislot size 6MTs. Each MT is 1μs. There are only two tasks transmitted in the DYN segment. There are enough slots allocated for the number of messages. Each message is configured to be 4 Bytes in size.

12.5.9 Initial Dynamic CAN Data

CAN data was assigned on the basis that at a minimum of one task would be designated to each node. This format was chosen to guarantee one dynamic message would be transmitted on the bus from each node. The dynamic tasks were configured to transmit at semi-times within a defined range. This range was between 0-20*ms*. This aspect was implemented to demonstrate that dynamic messages could get access to the bus regularly on either node at random times.

12.6 Dynamic Segment Analysis

RTA is carried out using Equation 12-8.

$$R_m(t) = C_m + \sigma_m + w_m(t)$$

Equation 12-8: Response Time Analysis Equation

The value C_m can be calculated as per the ST segment previously. First the total FlexRay data including the overhead is required. The overhead is calculated using the same method applied in the ST segment. This results in an overhead of 14 Bytes. Once the total communication time of all FlexRay data is know it can then be assessed whether there is enough time allocated in the DYN segment for the transmission of all DYN data in the best case scenario with no delays. Column three, FlexRay frame size, includes any overhead. This is shown in column three in Table 12-20.

Table 12-20: Aperiodic CAN Data

Message	FlexRay Bit Time	FlexRay Size (Bytes)	C_m (µs)	Total C_m (per Node)(*µs*)
m1	0.1µs	18	17.6	14.4

The total communication time of 14.4*µs* per node allows enough time to transmit the DYN data if no blocking occurs. There are 1485*µs* available (in the DYN segment) per cycle.

The next value for defining is *δ_m*, the delay caused by a message being generated just after its slot has passed. This results in Equation 12-9.

$$\sigma_m = FR(t) - (ST_{bus} + C_m + NIT)$$

Equation 12-9: δ_m Delay

The parameters for each message delay δm and associated parameters are shown in table 12-21.

Table 12-21: δ_m Parameters

Message	*FR(t)(µs)*	ST_{bus} (µs)	C_m (µs)	*NIT* (µs)	δ_m (ms)
m1	1750	240	14.4	35	1.4806

The final delay variable for calculating is w_m the delay caused by the transmission of ST messages and higher priority DYN messages. This is shown in Equation 12-10

$$w_m(t) = ST_{bus} + hp(m) + ms(m) + (pLatestTx.gdMinislot) + NIT$$

Equation 12-10: w_m Delay

Table 12-22: $w_m(t)$ Parameter

Message	*hp(m)* *(µs)*	*ST_{bus}* *(µs)*	*pLatestTx. gdminislot* *(µs)*	*NIT (µs)*	*ms(m)* *(µs)*	*$w_m(t)$* *(µs)*
m1	0		48		0.0000	303

The *pLatestTx* value was 239 minislots which was obtained from the designer tool. Subtracting *pLatestTx* this from the total number of minislots (247) results in a value of 8 minislots. These 8 minislots are unavailable for transmission. Equation 9-22 produces the *ms(m)* value.

To obtain the final worst case response time, sum the three parameters (C_m, δ_m and w_m) for each message. This presents the resulting delays as displayed in Table 12-23.

Table 12-23: Dynamic Response Times

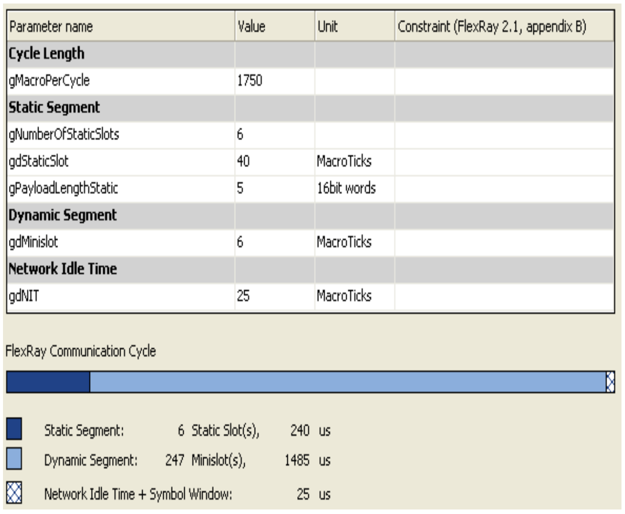
Message	*C_m* *(µs)*	*δ_m* *(µs)*	*$w_m(t)$* *(µs)*	**$R_m(t)$** **(ms)**	**Frame Cycles**
m1	14.4	1480.6	303	1.7980	1.027429

12.7 Final Experimental Case Parameters

The FlexRay parameters obtained for the experimental implementation case are:

- Frame Length (1750*µs*)
- Static Segment Size (240*µs*)
- Payload Size (5 2-word-bytes)
- Static Slot Size (40*µs*)
- NIT (25*µs*)
- DYN Segment Size (1485µs)

Figure 12-10 illustrates the parameters as they appear using the FlexRay configuration designer tool (DECOMSYS designer).



Parameter name	Value	Unit	Constraint (FlexRay 2.1, appendix B)
Cycle Length			
gMacroPerCycle	1750		
Static Segment			
gNumberOfStaticSlots	6		
gdStaticSlot	40	MacroTicks	
gPayloadLengthStatic	5	16bit words	
Dynamic Segment			
gdMinislot	6	MacroTicks	
Network Idle Time			
gdNIT	25	MacroTicks	

FlexRay Communication Cycle

■	Static Segment:	6 Static Slot(s),	240 us
▢	Dynamic Segment:	247 Minislot(s),	1485 us
▨	Network Idle Time + Symbol Window:		25 us

The final FlexRay frame size is 1.75*ms*. This is composed of a ST segment of six slots 40*μs* in size, a DYN segment of 247slots 6*μs* in size and finally a NIT of 25*μs*.

These final case parameters are used in stage six (Figure 12-11) of the migration framework development. These parameters are used to configure the FlexRay applications parameters that were implemented in the development environment. The results of this implementation are presented in Chapter 13.

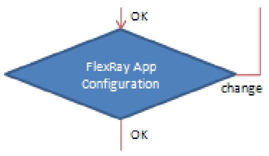


Figure 12-11: Framework Development Stage Six

12.8 Verification of Time-Triggered Properties (Test Case 3)

This test case was configured to demonstrate that bus loading would not have an adverse effect on the execution time of an application configured to operate on the ST segment of the FlexRay protocol. For this verification it is required to assign all application data to the ST segment. This is due to the ST segment containing time-triggered properties. The task graph model is presented in Figure 11-5. The results validating deadlines and busloads are presented in Chapter 13. The CAN bus bit rate used is 125kbit/s and the FlexRay bus bit rate used is specified at 10Mbit/s

12.8.1 Parameter Analysis

The WCET parameters are in keeping with those used in the previous two test cases. The initial CAN parameters used in this test case are presented in Table 12-24. The maximum message payload size is used in all messages. The initial release time r_i is 0*ms* and the final task graph deadline D_i is 40*ms*.

Table 12-24: CAN Initial Parameters Test Case Three

Task	**WCET (ms)**	**Deadline (ms)**	**Period (ms)**	**Size (Bytes)**
T1	4	4	4	8
T2	4	4	4	8
T3	4	4	4	8
T4	4	4	4	8
T5	4	4	4	8

12.8.2 Intermediate Task Values

To be able to find the intermediate task deadlines the slack needs to be reallocated equally among all tasks. There are is only one possible path through this task graph configuration (Figure 12-12).

e 12-12:
Through
Graph

Table 12-25 presents the slack per task on each path.

As illustrated in Table 12-26 a slack of 1.714ms is to be reallocated equally among all tasks.

Table 12-26: Task Graph Parameters

Task	WCET (ms)	Ri (ms)	Di (ms)	Slack (ms)
T1	4	0.000	5.714	1.714
T2	4	5.714	11.429	1.714
T3	4	11.429	17.143	1.714
T4	4	17.143	22.857	1.714
T5	4	22.857	28.517	1.714

Equation 9-7 verifies the task graph parameters as illustrated in Table 12-27.

Table 12-27: Final Task Graph Parameters

Task	Re-cal r_i (ms)	Re-calc d_i (ms)	Validation Check	
T1	0.000	5.714	0<5.714	🟩
T2	5.714	11.429	5.714<11.429	🟩
T3	11.429	17.143	11.429<17.143	🟩
T4	17.143	22.857	17.143<22.857	🟩
T5	22.857	28.517	22.857<28.517	🟩

12.8.3 Message Deadlines

Each message's transmission deadline is represented by Equation 9-8 and the transmission delay is given in Equation 9-9. These values are presented in Table 12-28.

Table 12-28: Message Deadlines

	Transmission Deadline (ms)	Transmission Delay (µs)
m1	1.714	512
m2	1.714	512
m3	1.714	512
m4	1.714	512
m5	1.714	512

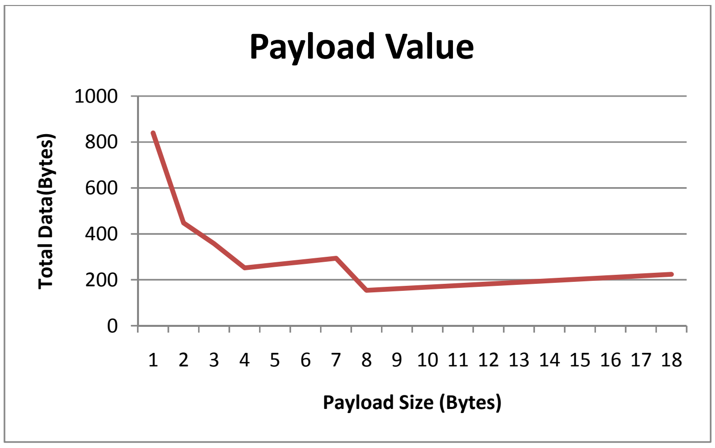
12.8.4 Payload Configuration

The overhead is defined as 14 Bytes as specified in section 12.2.7. As in previous test cases the payload is chosen heuristically depending on the total number of bytes transmitted per payload and frame size. Table 12-29 presents the payload calculations.

Table 12-29: Payload Calculations

Payload Size	FlexRay Frame Size	Number of Messages Required	Total Bytes
1	15	56	840
2	16	28	448
3	17	21	357
4	18	14	252
5	19	14	266
6	20	14	280
7	21	14	294
8	22	7	154
9	23	7	161
10	24	7	168
11	25	7	175
12	26	7	182

Graphing the Total Data versus Payload Size results in the graph in Figure 12-13.



The payload size of 14 Bytes is chosen which results in a frame size of 28 Bytes including the predetermined overhead Static Slot Size

The size is the static slot is calculated as per Equation 9-13 and is verified below. A slot size of 22.4μs is obtained but this is rounded up to 23μs.

$$gdStaticSlot = \left\lceil \frac{14Bytes + 14Bytes}{10Mbit/s} \right\rceil$$

$$gdStaticSlot = \left\lceil \frac{28Bytes}{10Mbit/s} \right\rceil = 23\mu s$$

At a payload size of 14 Bytes the smallest configurable ST slot size using DECOMSYS designer is 37μs. Being unable to set the static slot size to the figure extracted from the framework, this introduces redundancy into each static slot.

12.8.6 Discretising

A message deadline delay *td(m_i)* value of 1.7*ms* is extracted from the framework. Each ST slot is of size 37*μs*, this results in a discretised *td(M_i)* value of 459 slots as per equation 9-14.

12.8.7 Periodicity and Distance Requirement

To configure the FlexRay frame and the ST segment, so as to assist the DYN segment data in gaining access to the bus the message periods are modified as explained in section 9.8.6. For this test case all data used for loading the bus is placed in the ST segment. This resulted in four extra messages being added to the bus. This configuration demonstrates that by adding extra data in the ST segment the timing deadlines of the application are not affected. The four extra messages were assigned the highest identifiers possible to further demonstrate that this does not aid access to the bus. DYN slots were configured but unused for the purpose of demonstrating their potential use.

To meet these requirements the p_{min} value of 1.024*ms* is chosen (refer to section 9.8.6 for further details). This results in a minimum FlexRay frame of 512*μs*. The ST segment comprised of 11 ST slots of 37*μs* each, resulting in a ST segment size of 407*μs*. This coupled with an NIT of 18MT with each MT being 1*μs*. This results in a DYN segment size of 87*μs*.

Proving the frame size meets the periodicity and distance requirement:
0.512ms→1.024ms

Equation 9.15 provides a formal validation as proven below.

$$2^1 \cdot 0.512ms \leq 1024ms < 2^2 \cdot 0.512ms$$
$$1.024ms \leq 1.024ms < 2.048ms$$

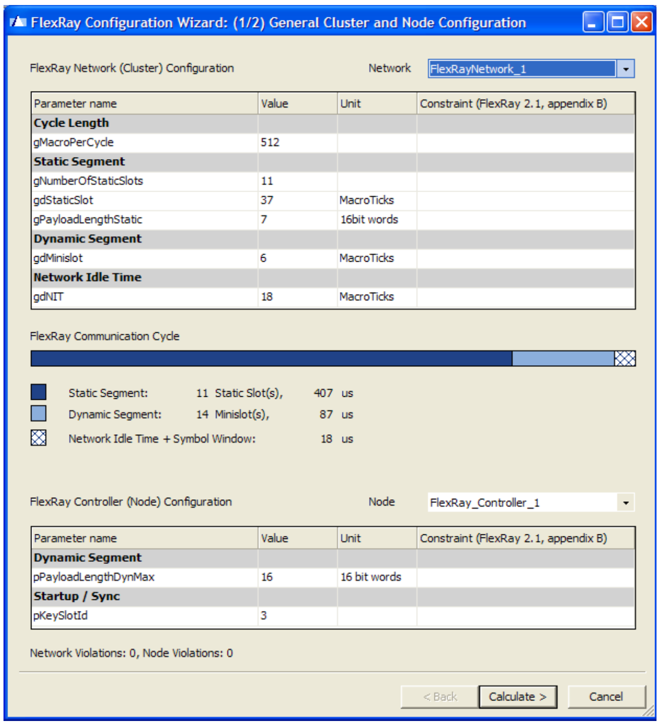
In this test case there is no DYN segment verification. While DYN slots were included in the configuration this was done to demonstrate that they could be included if required. Implementing the DYN segment would not add to this test case due to the DYN segment being event-triggered and the purpose of this test case was to demonstrate FlexRay's time-triggered properties.

12.9 Final Practical Case Parameters

The FlexRay parameters obtained for the practical implementation case are:

- Frame Length ($512\mu s$)
- Static Segment Size ($407\mu s$)
- Payload Size (7 2-word-bytes)
- Static Slot Size ($37\mu s$)
- NIT ($18\mu s$)
- DYN Segment Size (87µs)

Figure 12-14 illustrates the parameters as they appear using the FlexRay configuration designer tool (DECOMSYS designer).



FlexRay Configuration Wizard: (1/2) General Cluster and Node Configuration

FlexRay Network (Cluster) Configuration   Network: FlexRayNetwork_1

Parameter name	Value	Unit	Constraint (FlexRay 2.1, appendix B)
Cycle Length			
gMacroPerCycle	512		
Static Segment			
gNumberOfStaticSlots	11		
gdStaticSlot	37	MacroTicks	
gPayloadLengthStatic	7	16bit words	
Dynamic Segment			
gdMinislot	6	MacroTicks	
Network Idle Time			
gdNIT	18	MacroTicks	

FlexRay Communication Cycle

■	Static Segment:	11 Static Slot(s),	407	us
□	Dynamic Segment:	14 Minislot(s),	87	us
▨	Network Idle Time + Symbol Window:		18	us

FlexRay Controller (Node) Configuration   Node: FlexRay_Controller_1

Parameter name	Value	Unit	Constraint (FlexRay 2.1, appendix B)
Dynamic Segment			
pPayloadLengthDynMax	16	16 bit words	
Startup / Sync			
pKeySlotId	3		

Network Violations: 0, Node Violations: 0

The final FlexRay frame size is 512μs. This is composed of a ST segment of eleven slots 37μs in size, a DYN segment of 14slots 6μs in size and finally a NIT of 18μs. These parameters are used to configure the FlexRay application. The results of this implementation are presented in Chapter 13.

12.10 Conclusion

This chapter presents the actual migration in the context of applying facts and figures to the abstract case. There are three test cases presented; The Abstract Implementation, The Experimental Implementation and The Verification of Time-Triggered Properties.

The abstract implementation (TC) demonstrated the ability of the framework to handle a complex structured application with complex constraints. The Experimental implementation (ACC) demonstrated the frameworks ability to handle real constraints and also the frameworks flexibility in handling a different scenario application. The third test case, Verification of Time-Triggered Properties, simply assigns data (both from the application and loaded data) to the ST segment. A different task graph is also processed through the framework compared to the previous two test cases. This chapter presents the parameters necessary to configure a FlexRay frame for all test cases.

To assess the frameworks success in delivering parameters that are capable of producing a successful FlexRay configuration; task, application and deadlines are required. These are presented under varying busloads in Chapter 13.

12.11 References

Aloul N K a F,2005, The Synthesis of Dependable Communication Networks for Automotive Systems, SAE Internalional 2005,

Burns K T a A, 1994, Guaranteeing Message Latencies on Control Area Network (CAN).

Madsen J P K a S H, 2007,Design Optimisation of Fault-Tolerant Event-Triggered Embedded Systems,

Riis P, 2007,Simulation of a Distributed Implementation of an Adaptive Cruise Controller, Department of Computer and Information Science, Link¨oping University

Robert I. Davis A B, Reinder J. Bril and Johan J. Lukkien, 2007 Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised

Traian Pop P P, Petru Elses, Zebo Peng, Alexandru Andrei, 2006, Timinig Analysis of the FlexRay Communication Protocol, Editor, Conference Name, Conference Location.

13 Test Results & Verification:

13.1 Introduction

Chapter 12 presented the parameters as extracted from the migration framework after undergoing the migration procedure. The Abstract Implementation (TC) (Test Case 1) (Section 12.2) demonstrated the frameworks success at extracting results from a complex application scenario. The Experimental Implementation (ACC) (Test Case 2), while having reduced complex task graph structure, the ACC model demonstrated the generality of the framework under diverse conditions. The Verification of Time-Triggered properties (Test Case 3) presented another diverse task graph structure. To fully verify the migration procedure the Experimental Implementation was implemented in testing environment.

The ACC application was implemented in both CAN and FlexRay separately on the Fujitsu SK-91F467D development boards as per the system design specified in Chapter 11. The framework is considered successful if the CAN application is configured on the FlexRay protocol. Verification was carried out by obtaining message and application deadlines under various busload conditions and comparing the results obtained from the CAN and FlexRay implementations. The FlexRay application was tested without redundancy configured (only CH A transmits the ACC data) and with redundancy (ACC data is transmitted on CH A and CH B) to verify that it does not adversely affect timings.

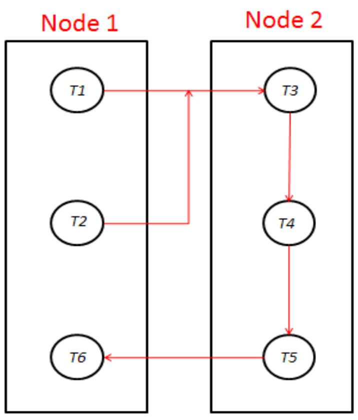
The third test case results were obtained by configuring the CAN and FlexRay configurations using CANalyzer's block generator function. The test case is considered successful if the FlexRay's applications timings are not affected by increases to the busload.

13.2 Test Case 2: ACC Configuration

The node configuration is illustrated in Figure 13-1 and 13-2. The same tasks are assigned to the same nodes for the CAN and FlexRay implementations. All busload data is recorded over a 30 second period and the CAN bus operates at 125kbit/s while FlexRay operates at 10Mbit/s. Where a 30 second sample period is not required a 30 cycle period suffices in such instances.

The messages ID1-ID6 (ACC application messages) are assigned to the ST segment in FlexRay because they are considered critical and ID7 and ID8 are assigned to the DYN segment due to them being perceived less critical than the application data.

The Standard Configuration is illustrated in Figure 13-1: The red lines indicate the messages that are transmitted across the bus.



The Minimal Configuration is illustrated in Figure 13-2: The red lines represent communication across the bus. The blue lines are only for communication within the node.

gnment

This chapter completes stage seven of the framework development as illustrated in Figure 13-3.

3-3: Stage
n of the
nework
lopment

13.3 CAN Results

The CAN test cases were divided into two sub sections to provide more comprehensive testing;

- CAN Standard – In this case all data is transmitted across the bus. It is possible another node or application could potentially require this data. See Figure 13-1.

- CAN Minimal – In this minimal configuration only the data required by inter communicating nodes is transmitted. All intra communicating data is not transmitted. See Figure 13-2.

CANalyzer detects a message when it appears on the bus; this is not necessarily the start of the application cycle. The application cycle starts when the first message in the application is generated. In the CAN test cases it is assumed that the application cycle starts 919*μs* before the first message appears on the bus. This value of 919μs is chosen because this is the time worked out for a message to appear on the bus after it has been generated. Because Test case 1 was completely theoretical no physical testing is required. All results that follow are for Tests Cases 2 and 3.

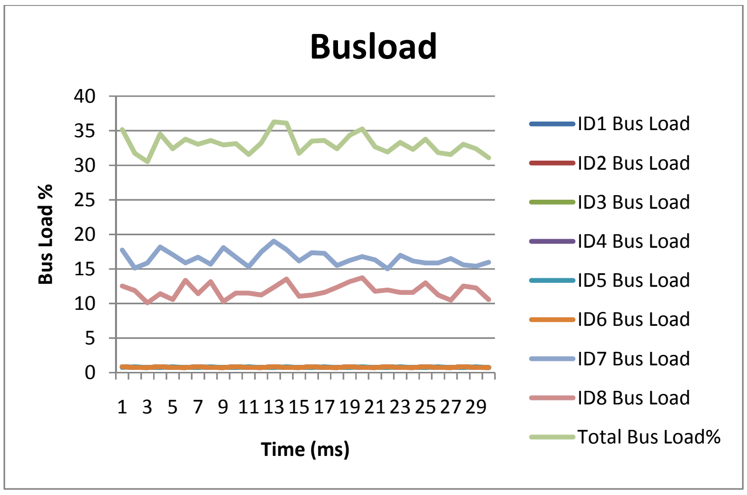
13.3.1 Test Case 2: CAN Standard Cases

Test Case2 results are split into CAN and FlexRay results. A further subdivision is created where results are divided into categories Case1,2 for CAN and 3,4 for FlexRay and the final division is into A, B, C for CAN and A, B, C and D for FlexRay depending on load levels. The FlexRay sub categories are explain in a paragraph prior to the results being presented.
Test Case 3 is divided into two categories, CAN and FlexRay.

13.3.1.1 CASE 1A: Normal Load

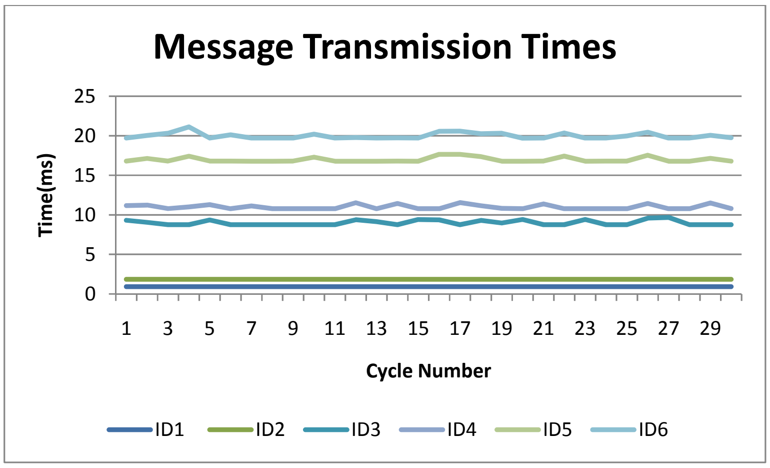
In this test case the total bus load is averaging at 33.09% for all CAN messages. The individual breakdown is presented in Table 13-1

Viewing Table 13-1 it is obvious that the application data from the ACC system (IDs 1-6) is not loading the CAN bus to any critical levels. This is graphically represented in Figure 13-4.



Message deadlines and application deadlines will require examination. With 30.09% busload overall message and application deadlines should be readily attainable. Each CAN message has a deadline as shown in Table 13-2. Only application messages of ID1-

ID6 are shown here as messages ID7 and ID8 are pseudo randomly sent. This results in higher bus loads for ID7 and 8. From Figure 13-5 the times that the application messages were transmitted on the bus are illustrated.

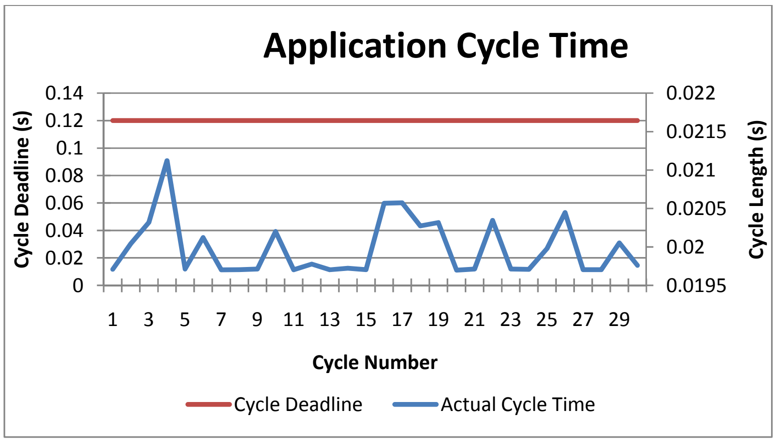


The deadlines for each message are presented in Table 13-2. It is immediately apparent that none of the deadlines are violated.

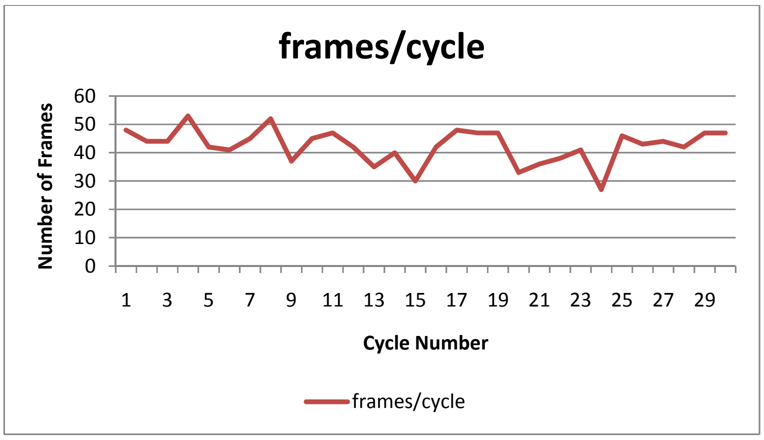
Table 13-2: Message Deadlines

Message ID	Deadline (ms)
1	20
2	40
3	60
4	80

To examine the message deadlines in relation to the application deadline it shows that the application deadline is easily attainable. The maximum cycle time was 21.12*ms*, with a minimum of 19.69*ms*. The average cycle time was 19.10*ms*, which easily allows the 120*ms* deadline time to be met. This is illustrated in Figure 13-6, where the Y-axes are scaled. The standard deviation in the application cycle time is 369*μs*.



Finally it can be shown how many frames are sent per cycle. This value can be compared with FlexRay's ST and DYN segments separately. Segregating the data it is shown that six messages between ID 1 to ID 6 are transmitted per application cycle as predicted. The total number of frames per cycle value fluctuates in CAN due to the messages with set deadlines and the messages with random transmit times both transmitting in the same cycle. The standard deviation is 5.99 frames per cycle. Figure 13-7 shows the number of frames fluctuating per cycle due to both the application data and random data being displayed. The maximum number of frames per cycle is 53, with the average being 42.43.



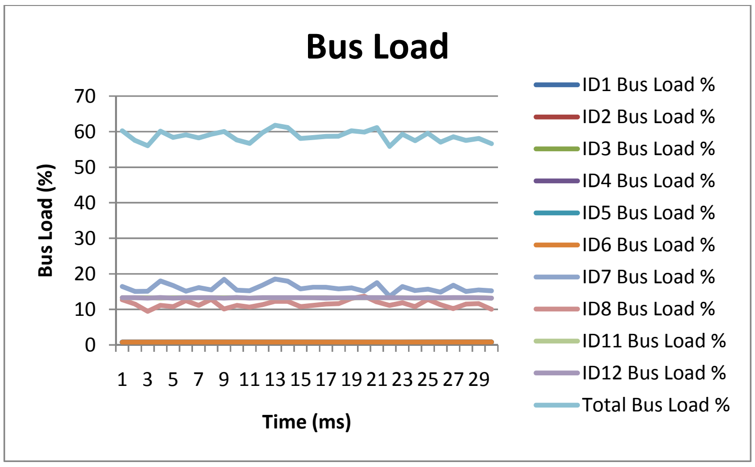
13.3.1.2 Case 1B: 60% Load

Increasing the busload to 60% is accomplished through the addition of two extra messages generated by CANalyzer. These messages we assigned IDs close to the highest priorities available (ID 11 and ID 12). The messages were set to transmit at periods of 7ms. The busload values are presented in Table 13-3 below.

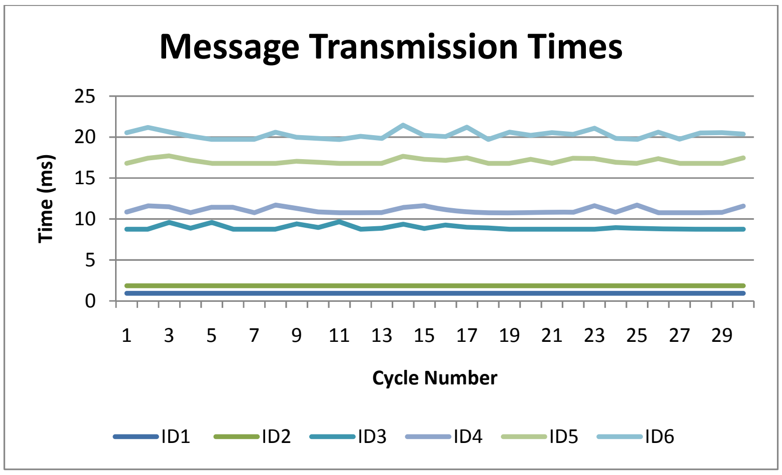
Table 13-3: 60% Busload

Msg ID	Min Load%	Max Load%	Average %
ID1	0.74	0.84	0.78
ID2	0.74	0.84	0.77
ID3	0.74	0.84	0.77
ID4	0.74	0.84	0.77
ID5	0.74	0.84	0.77
ID6	0.74	0.84	0.78
ID7	13.64	18.56	16.05

The figures presented in Table 13-3 are illustrated in graph format in Figure 13-8.



With an average busload of 58.71% and the maximum not exceeding 61.80% it is expected that deadlines are not violated. The CAN message deadlines are the same as for the previous case 1A. Only messages of ID1-ID6 are shown here as messages ID7 and ID8 are randomly sent and messages ID11 and ID12 are transmitting periodically every 7ms. Figure 13-9 illustrates the transmit times of the application messages. The deadlines are as presented in Table 13-2. The final message transmit time as seen from the graph in Figure 13-9 is at the 20ms mark where as the deadline is 120ms. The application is therefore still meeting its deadlines.



Examining the deadlines in relation to the application deadline Figure 13-10 illustrates that the application deadline is not violated (the Y-axes are scaled). The maximum cycle time is 21.45*ms* with a minimum of 19.70*ms*. The average cycle time is 20.28*ms* which is well below the 120*ms* deadline time. There is a standard deviation in the application cycle of 503*μs*.

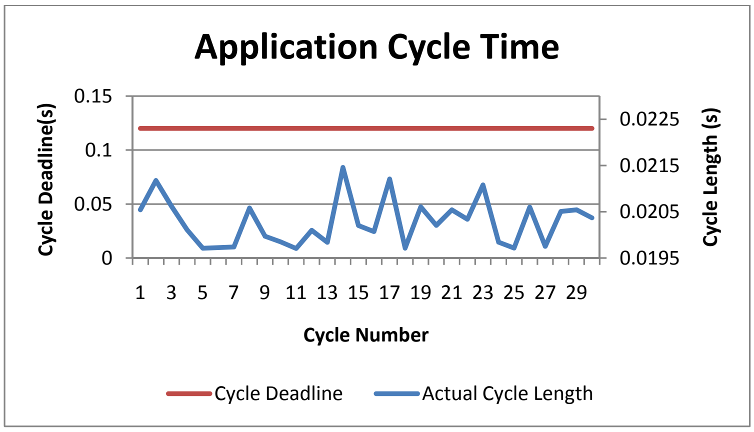
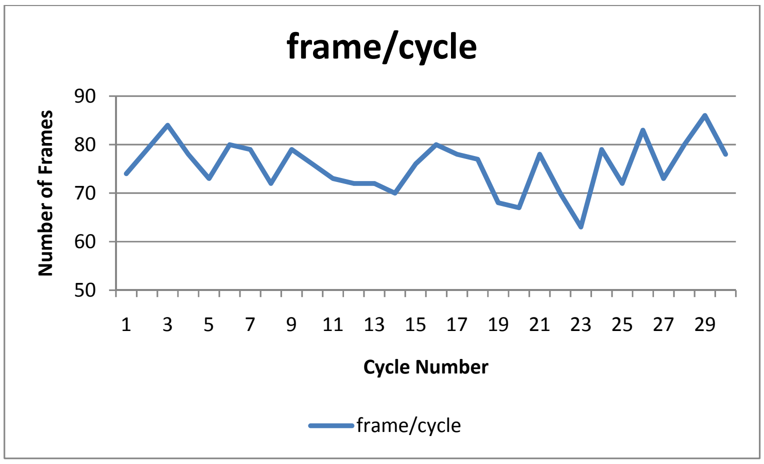


Figure 13-10: Application Cycle Times at 60% load

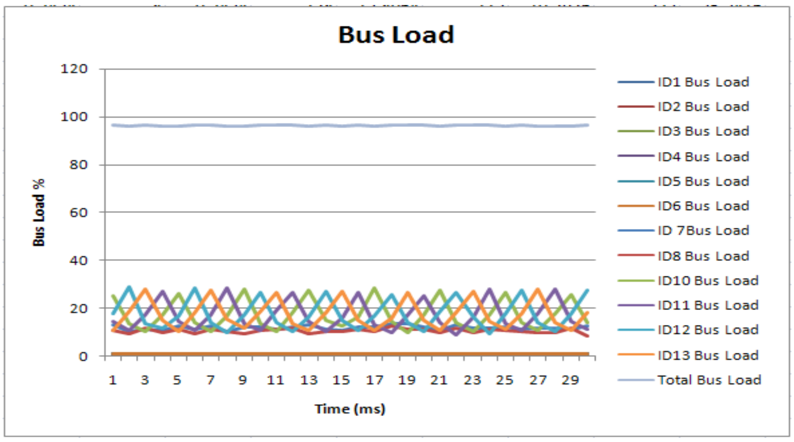
Figure 13-11 illustrates the number of transmitted frames fluctuating per cycle. The maximum number of frames per cycle is 86 at 60% load; with an average of 75.63 frames per cycle and a standard deviation of 5.22 frames per cycles. As in the previous case there are six application messages transmitted per application cycle.



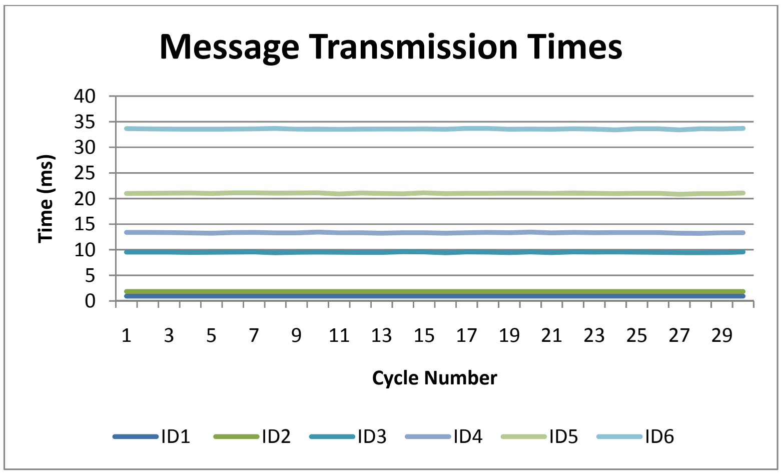
13.3.1.3 Case 1C: Maximum Load

To obtain a maximum busload, another two extra messages were generated on the bus with IDs of 10 and 13. These were again generated by CANalyzer. The messages ID10-13 were set to transmit at 1*ms* periods to put as much data on the bus as possible. The busload values are presented in Table 13-4 below.

The figures presented in Table 13-4 are illustrated in graphically in Figure 13-12.



With an average busload of 96.29%, it is expected that deadlines will be missed. This should be more pronounced for the lower priority messages. Only the applications messages (ID1-ID6) are shown here. Messages ID7 and ID8 are randomly sent and not considered critical and messages ID10-13 are set to transmit periodically every 1*ms* and are present with the purpose of loading the CAN bus. The application messages are still meeting their deadlines as illustrated below in Figure 13-13.



In Figure 13-13 all message deadlines are being met. The deadlines are provided in Table 13-2.

The messages generated by CANalyzer with the lower priorities are struggling to meet their deadlines. This results in messages with ID 10-13 being delayed gaining access to the bus by approx 4ms later than what was set. This is illustrated in Table 13-5.

Table 13-5: Message Delays

Msg ID	Set Period (ms)	Min (ms)	Max (ms)	Average (ms)
ID10	1	0.98	496.10	5.48
ID11	1	0.97	402.39	5.40

Examining the deadlines in relation to the application deadline, it shows that the application deadline is still not being violated even though the lower priority messages experience contention when accessing the bus. The reason for the application deadline not being violated is due to the message periods being significantly larger than those not associated with the application. This results in the messages getting access to the bus due to higher priority and occurring less frequently. The maximum application cycle time was 33.77ms with a minimum of 33.32ms. The average cycle time was 33.57ms which is vastly below the 120ms deadline time. The standard deviation is 91.34µs. Figure 13-14 illustrates this with the Y-axes scaled.

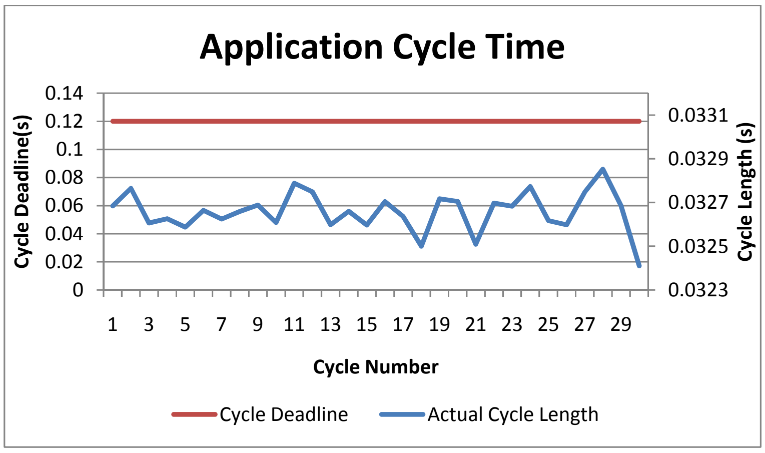
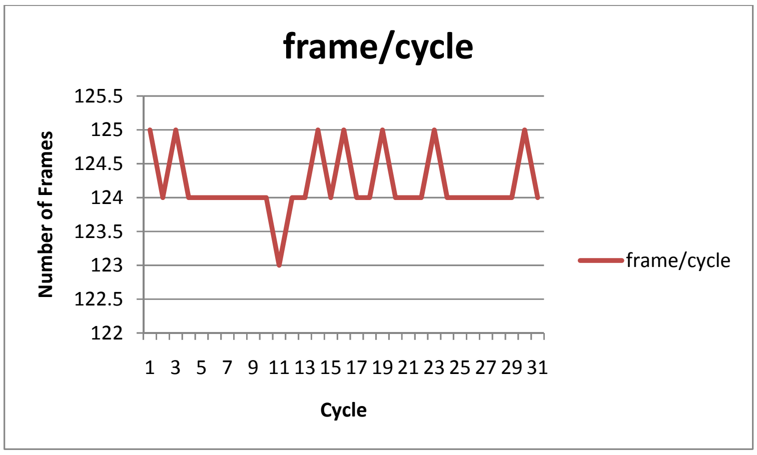


Figure 13-15 illustrates the number of frames fluctuating per cycle. The maximum number of frames per cycle is 125, at the maximum load, with an average of 124.20 frames per cycle. The standard deviation figure of 0.48 frames per cycles is small due to the upper limit being reached in terms of the number of frames that the bus can physically handle. There are six application messages (ID1 - ID6) transmitted per application cycle in keeping with previous case results.



13.3.2 CAN Minimal Cases

The minimal configuration test cases are configured as per Figure 13-2. It is expected that the results obtained in the minimal test case would be similar to those obtained through the normal test case.

13.3.2.1 CASE 2A: Normal Load

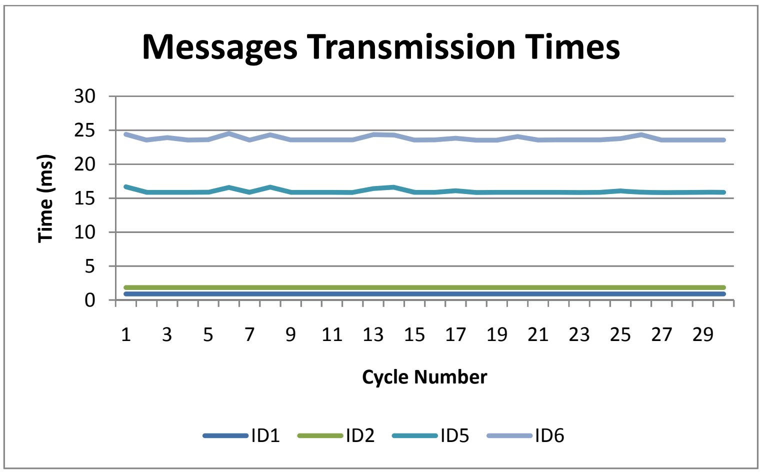
In this test case the total bus load is averaging at 33.40% for all CAN messages. The individual breakdown is presented in Table 13-6

Table 13-6: Normal Busload

Msg ID	Min Load%	Max Load%	Average %
ID1	0.74	0.84	0.77
ID2	0.74	0.84	0.77
ID5	0.74	0.84	0.77
ID6	0.74	0.84	0.77

Viewing Table 13-6 it is expected that this level of bus loading (Max 32.40%) should not affect any messages timing based on observations in the previous test case.

Message deadlines and application deadlines will be examined. Each CAN message has a deadline of 20*ms* as it was in the “Normal Case”. Only messages of ID1, 2, 5 and 6 are presented here as messages ID 3 and 4 are not transmitted across the bus. Messages ID7 and ID8 are randomly transmitted. Each message deadline is as presented in Table 13-2. Figure 13-16 illustrates the message transmit times. The results are similar to those obtained in the previous test case



The application deadline is not corrupted as illustrated in Figure 13-17. The maximum cycle time was 24.51*ms* and a minimum of 23.54*ms*. The average cycle time was 23.77*ms* which is below the 120*ms* application deadline time. The standard deviation in the application cycle is 328.43*μs*. In the minimal configuration there are four messages per application cycle. This is two less than in the normal configuration. The Y-axes in Figure 13-17 are scaled.

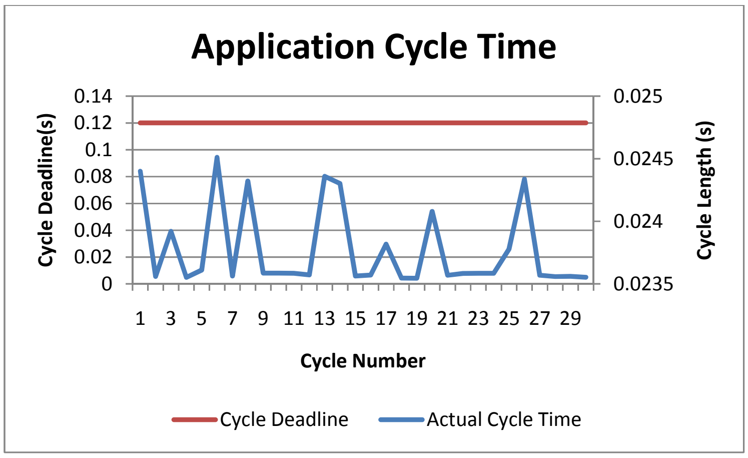


Figure 13-18 displays the number of frames fluctuating per cycle. The standard deviation is 5.05 frames per cycle. The maximum number of frames pre cycle is 49, with the average being 40.06 frames per cycle.

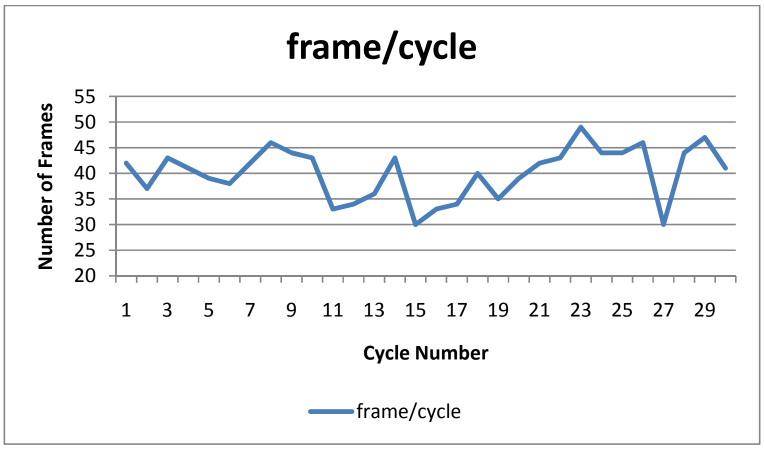
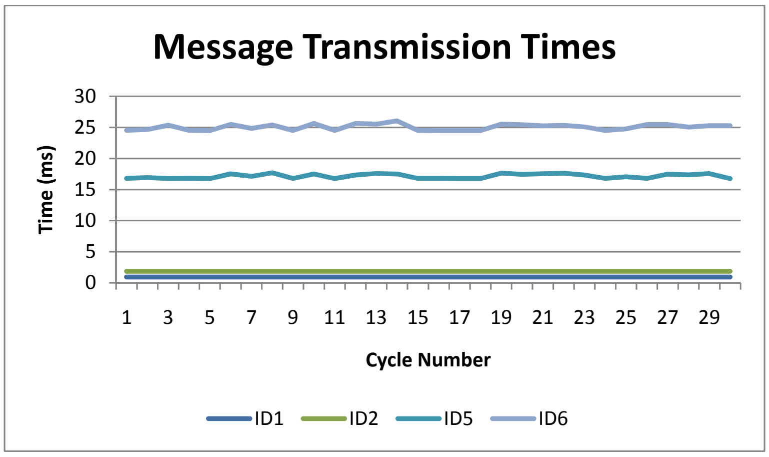


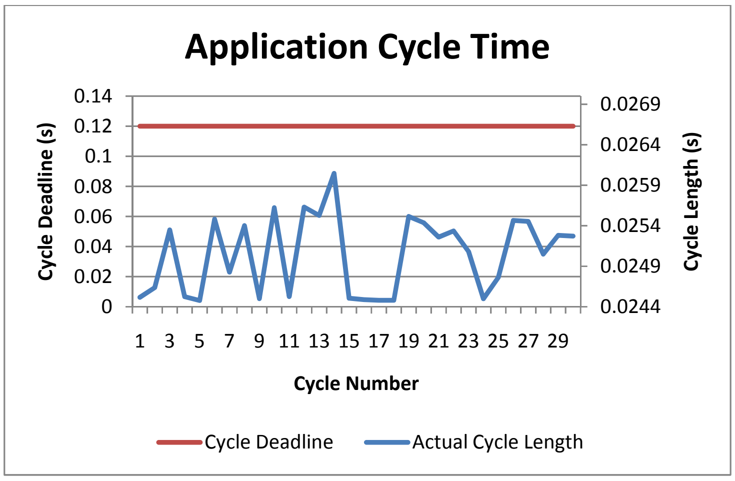
Figure 13-18: Number of Frames per Cycle at Normal Load

13.3.2.2 Case 2B: 60% Load

Increasing the busload to 60% required the addition of two extra messages generated by CANalyzer. These additional messages we assigned IDs close to the highest priorities available (ID 11 and ID 12). The messages were set to transmit at periods of 7*ms*. The busload values are presented in Table 13-7 below.

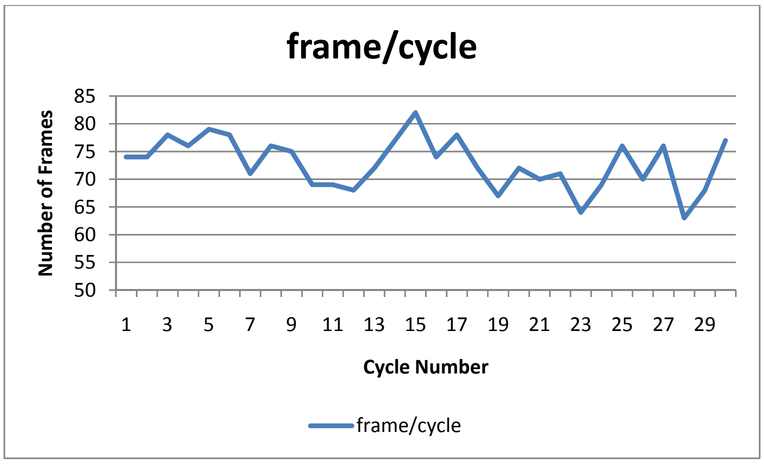


With an average busload of 56.78% and the maximum not exceeding 59.30% it would be expected that messages completed in advance of their deadlines. The CAN message deadlines are as presented in Table 13-2. Only the application messages (ID1, 2, 5, and 6) are presented here as messages ID7 and ID8 are randomly sent and messages ID11 and ID12 are transmitting periodically every 7*ms*. All data generated by CANalyzer gets access to the bus without any delays therefore meeting the associated message deadlines. The message transmit times are illustrated in Figure 13-19.



Examining the deadlines in relation to the application deadline it is proven that the application completes prior to its deadline in Figure 13-20. The maximum cycle time is 26.04*ms* with a minimum of 24.47*ms*. The average cycle time is 25.04*ms* which is below the 120*ms* deadline time. There is a standard deviation of 475.50*μs*.

Figure 13-21 shows the number of frames fluctuating per cycle. The standard Deviation is 4.58 frames per cycle. The max number of frames per cycle is 82 at 60% load, with an average of 72.83 frames per cycle.



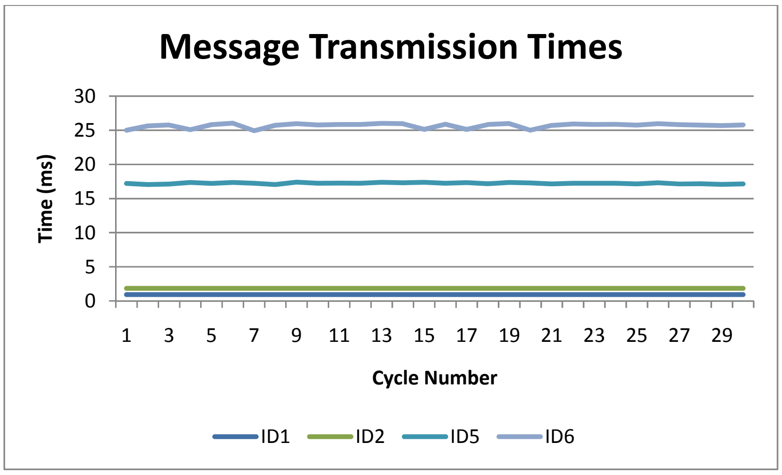
13.3.2.3 Case 2C: Maximum Load

To obtain the maximum busload two extra messages were generated on the bus with IDs of 10 and 13 as was done in Case 1C. The messages ID10-13 were set to transmit at 1*ms* periods to place as much data on the bus as possible. The busload values are presented in Table 13-8 below.

Table 13-8: Maximum Busload

Msg ID	**Min Load%**	**Max Load%**	**Average %**	**Total Min %**	**Total Max %**	**Average %**
ID1	0.74	0.84	0.77	96.23	96.60	96.39
ID2	0.74	0.84	0.77			
ID5	0.74	0.84	0.77			
ID6	0.74	0.84	0.77			
ID7	13.18	16.89	14.79			
ID8	9.65	12.71	11.05			

With an average busload of 96.39% it would be expected that deadlines are missed. This would be more pronounced for lower priority messages. Only application messages (ID1, 2, 5 and 6) are shown here due to messages ID7 and ID8 being randomly sent and messages ID10-13 are set to transmit periodically every 1*ms*. The application messages with ID1, 2, 5 and 6 are still meeting their deadlines as illustrated below in Figure 13-22. The deadlines are presented in Table 13-2.



Because of the large amount of slack in each message (e.g. ID5 transmission time of 17.235*ms* and a deadline of 100*ms*) and the low transmission frequency when compared to the other messages in the system, it is unlikely that the application will fail in its current configuration. The messages generated by CANalyzer with the lower priorities are struggling to meet their deadlines. This results in messages with ID 10-13 being delayed gaining access to the bus by approx 4*ms* after what was set. This is illustrated in Table 13-9.

Table 13-9: Message Delays

Msg ID	Set Period (ms)	Min (ms)	Max (ms)	Average (ms)
ID10	1	0.98	2012.66	5.39
ID11	1	0.97	2001.95	5.83

Examining application execution and deadline times (Figure 13-23) it shows that the application completes execution prior to its deadline as in the previous CAN test cases. The maximum cycle time was 26.03*ms* and a minimum of 24.76*ms*. The average cycle time was 25.68*ms* which is below the 120*ms* deadline time. The standard deviation is 338.72*μs*. The Y-axes are scaled in Figure 13-23.

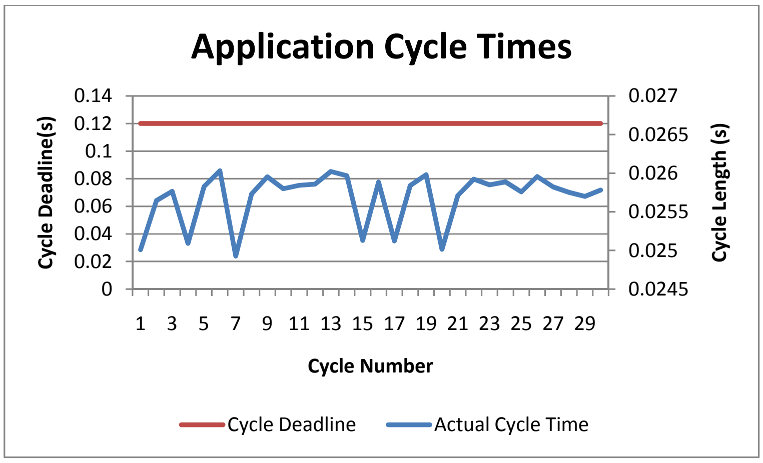
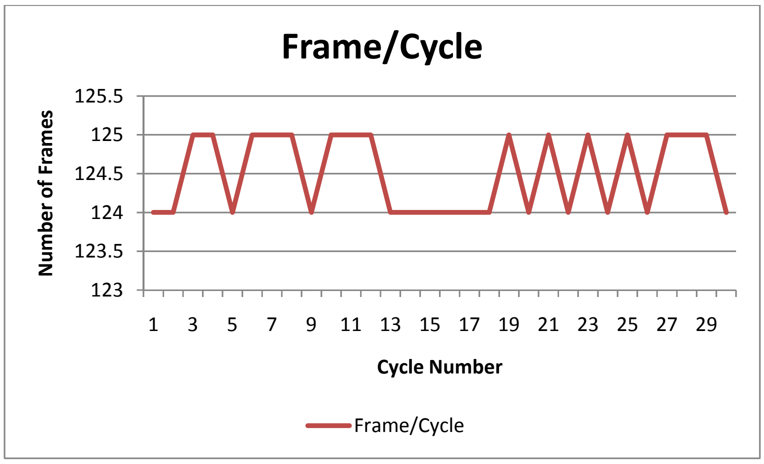


Figure 13-24 presents the number of transmitted frames fluctuating per cycle with a standard deviation of 0.50. The maximum number of frames per cycle is 125 at the maximum load, with an average of 124.50 frames per cycle.



13.4 FlexRay Results

The FlexRay results are divided into two sections;

- Without Redundancy
- With Redundancy

Within each of these sections there are further subdivisions:

- Standard
- Standard High Data Rate
- Minimal
- Minimal High Data Rate

The "Standard" configuration refers to Figure 13-1 and the "Minimal" configuration refers to Figure 13-2.

The CAN messages with ID1-ID6 are assigned to the ST segment and messages ID7 and ID8 are assigned to the DYN segment. Where loading of the bus is required this data was placed in the DYN segment because this data is not guaranteed to meet its deadlines because of the ET nature of the DYN segment. If the loaded data was placed in the ST segment it would be guaranteed to get access to the bus if scheduled correctly due to the TT nature of the ST segment.
The ST messages are transmitted on channel A (CH A) and the DYN data is transmitted on channel B (CH B). Where data is loaded onto the bus both CH A and CH B are used.

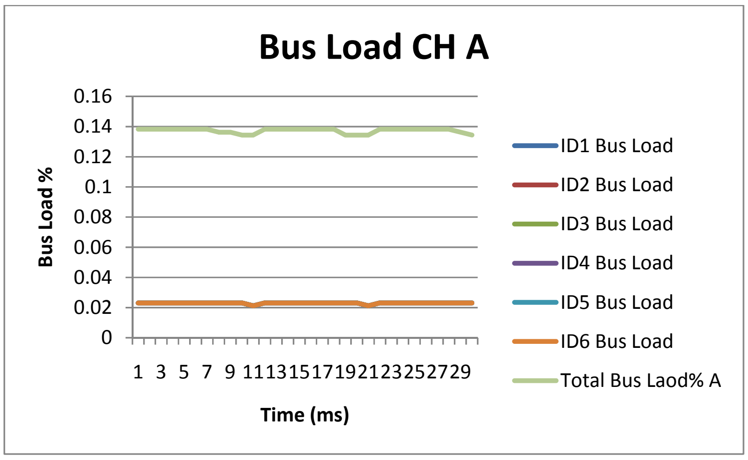
The application message cycle is assumed to begin once the latest dynamic message has transmitted in the previous cycle. This assumption is necessary because CANalyzer only records when messages appear on the bus not when the messages are signalled for transmission by the MCU through the communications stack. In the FlexRay test case the application cycle is assumed to start 1.391ms before the message of ID 1 appears on the bus. This is the time between the last DYN message of the previous cycle and the current ST message in the current cycle.

13.4.1 Without Redundancy

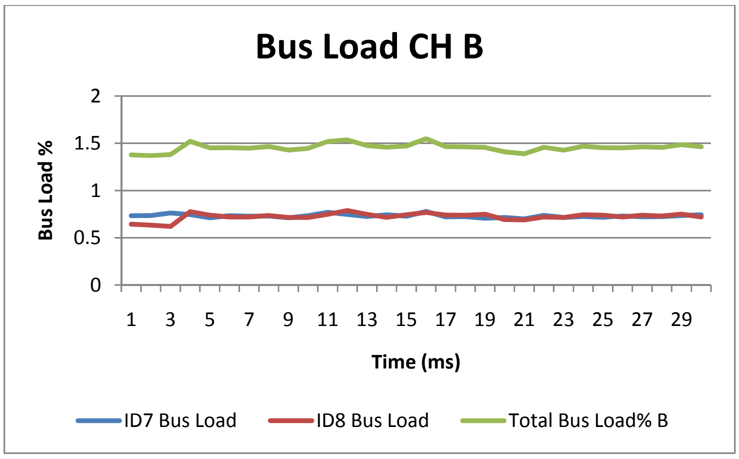
13.4.1.1 Case 3A: Standard Normal Load

Due to the FlexRay bus operating at 10 Mbit/s it would be expected that at normal busloads FlexRay would be using less of its percentage total capacity than the CAN equivalent. This is demonstrated in Table 13-10. Even by combining these two loads onto a single channel, the busload would still be considerably less than the CAN equivalent.

These busloads are illustrated graphically in Figures 13-25 and 13-26. The IDs 1-6 have the same loads so they appear as one line on the graph behind each other.



The busloads are higher on CH B due to the messages allocated there having smaller period than the messages on CH A. Therefore CH B data is on the bus more frequently.

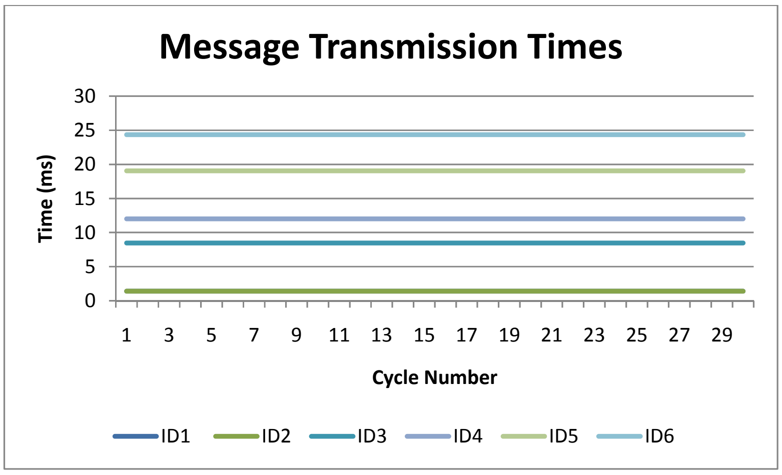


From the framework in section 12.5.7 the messages periods are modified to 14ms as illustrated in Table 13-11.

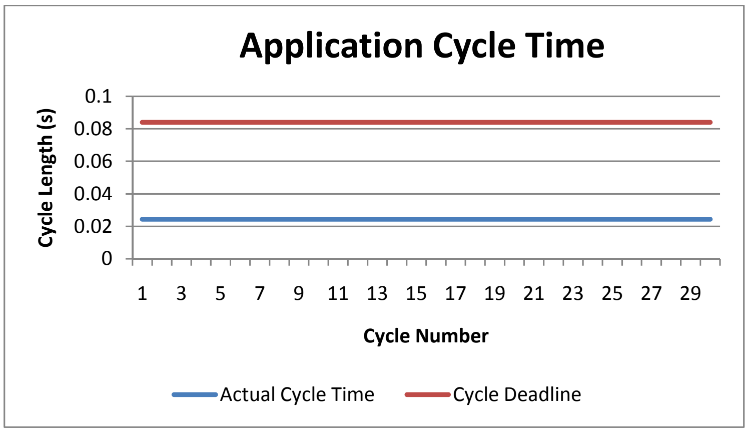
Table 13-11: Revised FlexRay Message Deadlines

Message ID	Deadline (ms)
1	14
2	28
3	42
4	56

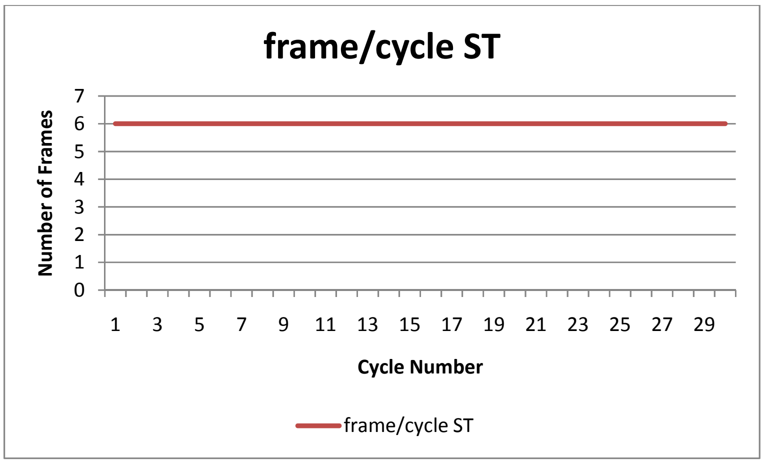
Figure 13-27 illustrates that the messages easily meet their deadlines. Due to some messages having different *WCETs* there will be slight variations between the messages actual cycle times.



To examine if the each message deadline can combine to complete the application before the application deadline of 84*ms*, Figure 13-28 is examined. The maximum cycle time is 24.33*ms* and a minimum of 24.33*ms*. The average cycle time was 24.33*ms* which is significantly below the 84*ms* deadline time. There was slight deviation which resulted in a standard deviation of 183*ns*.



In the ST segment the number of frames per cycle is static (6 frames per cycle) as the application data is scheduled at fixed points in time as illustrated in Figure 13-29. This corresponds to the six frames transmitted by the application data in CAN. In the DYN segment the number of frames per cycle is random due to the ET nature of the DYN segment. This is illustrated in Figure 13-30.



There are a maximum of 68 frames per cycle in the DYN segment with an average of 63.03 frames per cycle. The standard deviation is 3.96 frames per cycle.

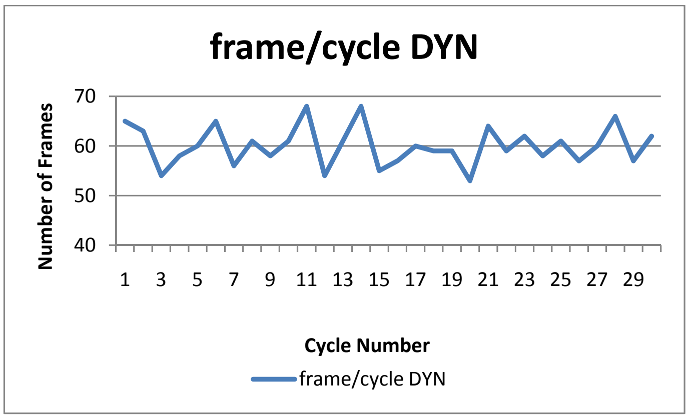


Figure 13-30: Number of DYN Frames per Cycle Normal Load

13.4.1.2 Case 3B: Standard at High Data Rate

At a High Data Rate (HDR) loads in FlexRay there is twice the amount of data on the bus when compared to the maximum amount of data on the CAN. Even at this HDR the FlexRay bus would be expected to have capacity to handle a higher volume of traffic if it was required to do so.

In addition to the two messages transmitted from the development boards (ID7 and ID8) the bus is loaded with messages (hex) ID 9, a, b, c, d, e, f and 10. These messages are set for transmission once in every FlexRay communication cycle at a payload of 8 Bytes.

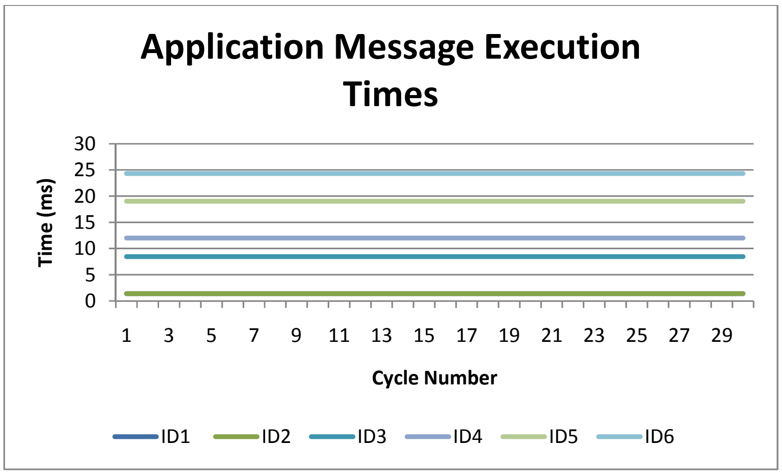
The busload data is presented in Table 13-12.

Table 13-12: High Data Rate Busload

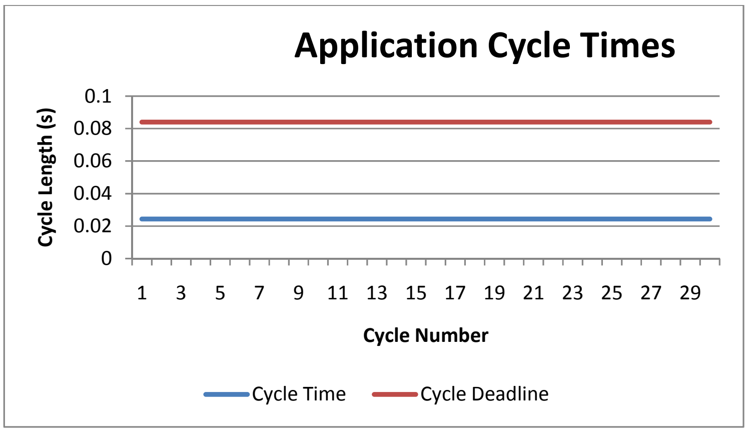
Msg ID	Minimum Load%	Maximum Load%	Average Load %
ST			
ID1	0.02	0.02	0.02
ID2	0.02	0.02	0.02
ID3	0.02	0.02	0.02
ID4	0.02	0.02	0.02
ID5	0.02	0.02	0.02
ID6	0.02	0.02	0.02
Total	0.13	0.14	0.14
DYN			
ID7	0.69	0.77	0.73
ID8	0.62	0.79	0.72
ID9	1.10	1.10	1.10
IDa	1.10	1.10	1.10
IDb	[illegible]	[illegible]	[illegible]

The DYN data was split for transmission between CH A and CH B. This was done to demonstrate the affect of additional data on the reference applications deadlines. The data loaded on CH A is assigned identifiers ID d, e, f and 10 while the data loaded on CH B is assigned identifiers ID a, b, c and 9.

Figure 13-31 illustrates that the application messages meet the required application deadlines.



To examine if the each message deadline can combine to complete the application before the application deadline of 84*ms*, Figure 13-32 is examined. The maximum cycle time was 24.33*ms* and a minimum of 24.33*ms*. The average cycle time was 24.33*ms* which is considerably less than the 84*ms* deadline time. The standard deviation in the cycle times is 183*ns*.



In the ST segment the number of frames per cycle is static at six frames per cycle. In the DYN segment the number of frames per cycle is aperiodic due to the ET nature of the DYN segment. This is illustrated in Figure 13.33. The DYN data contains a maximum of 455 frames per cycle with an average of 444.70 frames per cycle. There is a standard deviation of 5.08 frames per cycle.

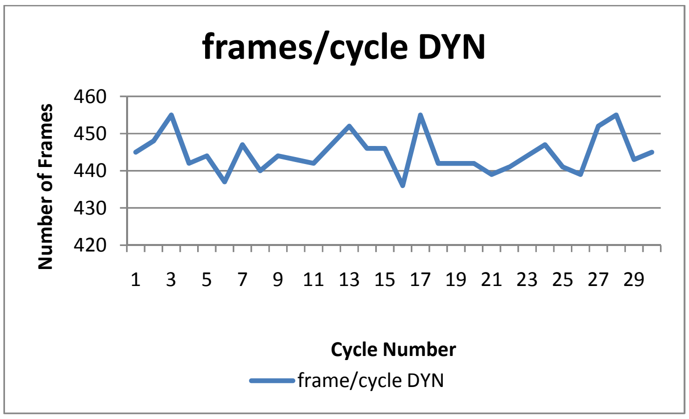


Figure 13-33: Number of DYN Frames per Cycle High Data Rate

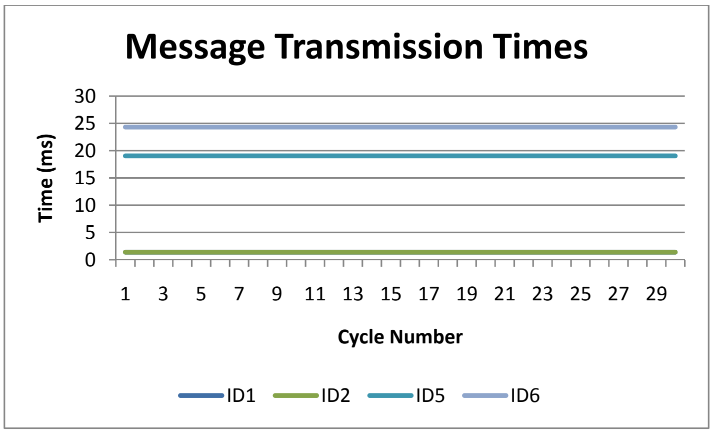
13.4.1.3 Case 3C: Minimal Normal Load

The minimal configuration at normal loads would be expected to yield slightly reduced busloads (compared to the standard configuration) in the ST segment as two less frames per application cycle are transmitted on the FlexRay bus. Table 13-13 presents the busloads.

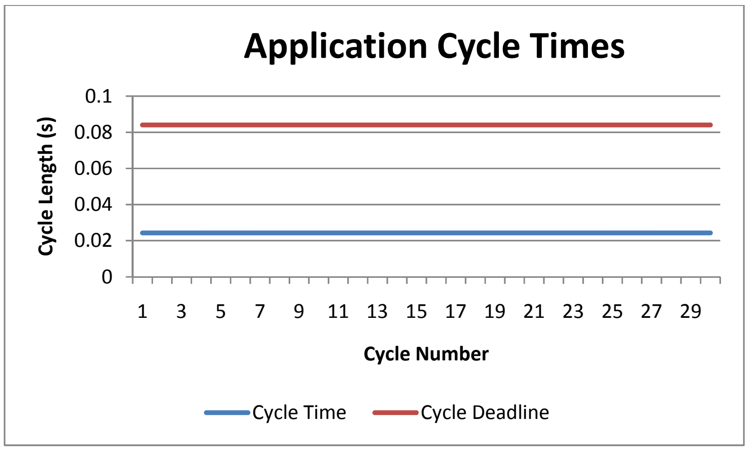
Table 13-13: Normal Busload

Msg ID	Minimum Load%	Maximum Load%	Average Load %
ST			
ID1	0.02	0.02	0.02
ID2	0.02	0.02	0.02
ID5	0.02	0.02	0.02
ID6	0.02	0.02	0.02
Total	0.09	0.09	0.09

If application messages are able to meet their deadlines needs determining. Figure 13-34 is used in determining this. It is illustrated that all messages meet their deadlines as defined in Table 13-11.



Determining if the each message deadline can combine to complete the ACC application before the application deadline of 84*ms*, Figure 13-35 is examined. The maximum, minimum and average cycle times are all 24.33*ms*. This consistency in the applications timing is in keeping with the TT nature of the ST segment in which the application data was assigned. The maximum ACC application cycle time of 24.33*ms* allows the application to complete greater than three times quicker than what is required (84*ms* deadline time). There is a standard deviation of 254*ns*.



There are constantly four frames per ACC application cycle; this is in keeping with the expected results. In the DYN segment the number of frames per cycle is aperiodic due to the ET nature of the DYN segment. This results in a standard deviation of 4.46 frames per cycle. There are a maximum of 69 frames per cycle in the DYN segment with an average of 61.6 frames per cycle. Figure 13-36 illustrates this.

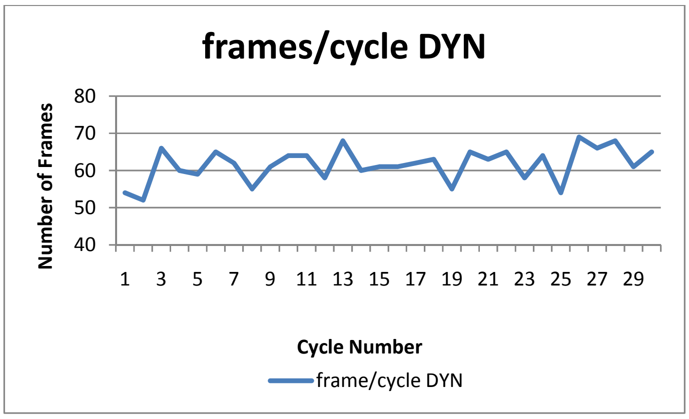


Figure 13-36: Number of DYN Frames per Cycle at Normal Load

13.4.1.4 Case 3D: Minimal at High Data Rate

At High Data Rates (HDR) there is twice the amount of data on the bus when compared to the maximum amount of data on the CAN bus. Even at this HDR the FlexRay bus would be expected to have capacity to handle more messages if it was required to do so.

In addition to the two messages sent from the development boards (ID7 and ID8) the bus is loaded with messages (hex) ID 9, a, b, c, d, e, f and 10. These messages are set for transmission once in every communication cycle with a payload of 8 Bytes.

The busload data is presented in Table 13-14.

Table 13-14: High Data Rate Busload

Msg ID	Minimum Load%	Maximum Load%	Average Load %
ST			
ID1	0.02	0.02	0.02
ID2	0.02	0.02	0.02
ID5	0.02	0.02	0.02
ID6	0.02	0.02	0.02
Total	0.09	0.09	0.09
DYN			
ID7	0.64	0.78	0.72
ID8	0.58	0.77	0.71
ID9	1.10	1.10	1.10
IDa	1.10	1.10	1.10
IDb	1.10	1.10	1.10

The loaded data generated from CANalyzer is assigned to each channel as per test case 3B. Figure 13-37 illustrates that message deadlines are not violated. The message closest to missing its deadline is message ID 1. It has 12.61ms slack which allows it too comfortably to meet its deadline of 14ms. The deadlines are presented in Table 13-11.

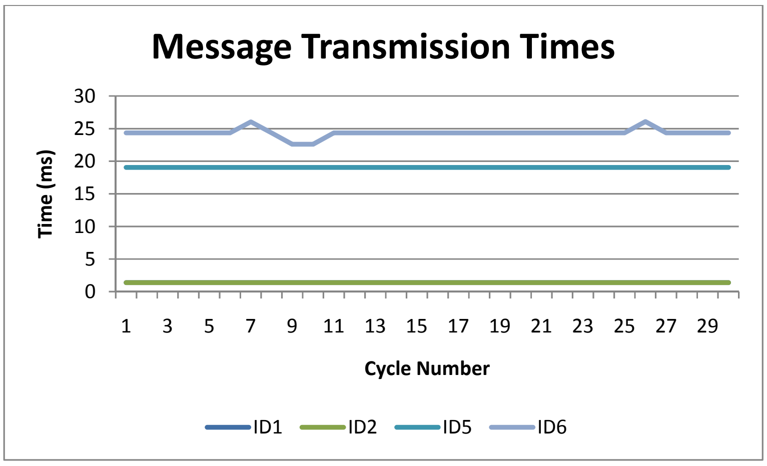


Figure 13-38 presents the ACC application cycle times. The maximum cycle time was 26.08*ms* and the minimum was 22.58*ms*. The average cycle time was 24.33*ms*. As with all previous test cases the ACC application cycle time is within the ACC application deadline. The standard deviation is 559.50*μs*.

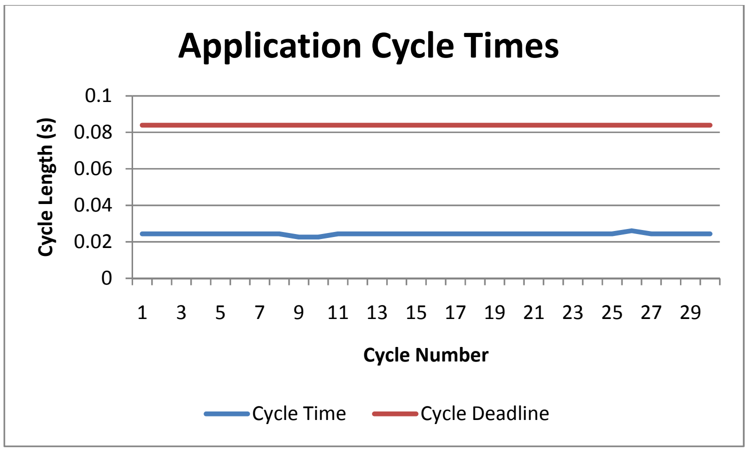
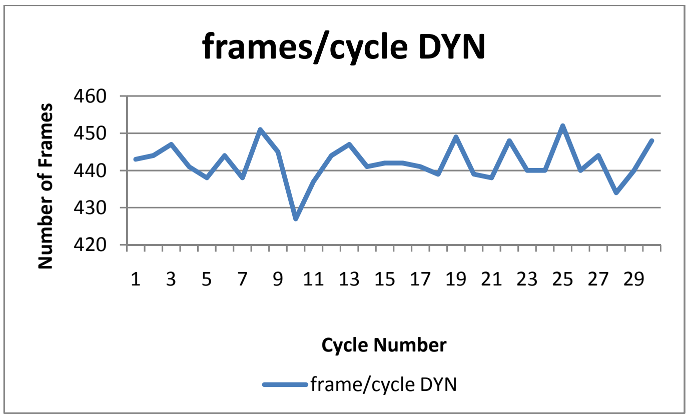


Figure 13-38: Application Cycle Times High Data Rate

As in the previous minimal configuration results (Case 3C) there are only four application frames per ACC application cycle. In the DYN segment the number of frames per cycle is aperiodic due to the ET nature of the DYN segment. This is illustrated in Figure 13-39. The standard deviation is 5.17 frames per cycle. There are a maximum of 452 frames per cycle with an average of 442.1 frames per cycle.



13.4.2 With Redundancy

In the following test cases 4A, B, C and D, redundancy was implemented for the data transmitted in the ST segment. This involved duplicating the data on CH A. CH B data was considered the redundant data in these test cases.

13.4.2.1 Case 4A: Standard Normal Load (Including Redundancy)

Due to the FlexRay bus operating at 10 Mbit/s it would be expected that 'normal' ACC application busloads with the two DYN messages would be less than the CAN equivalent. This is verified in Table 13-15. The ST data combines the loads of CH A and CH B.

From the framework in section 1.5.6 the messages periods are modified to 14ms (as was the case for the non-redundancy test cases). Figure 13-40 illustrates that the messages continue to easily meet their deadlines.

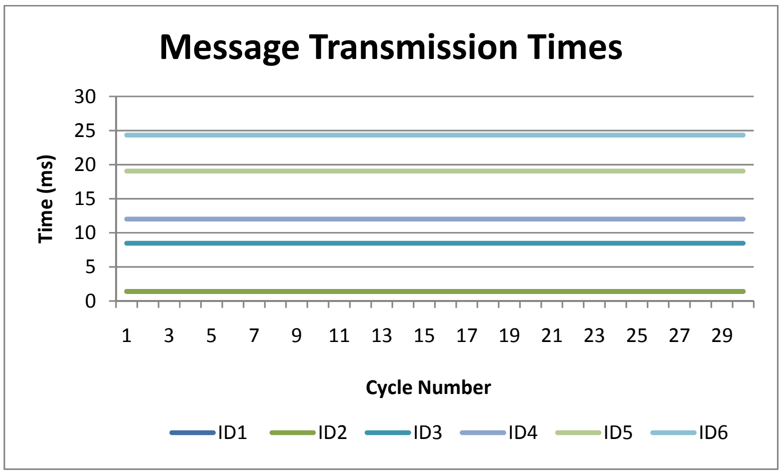
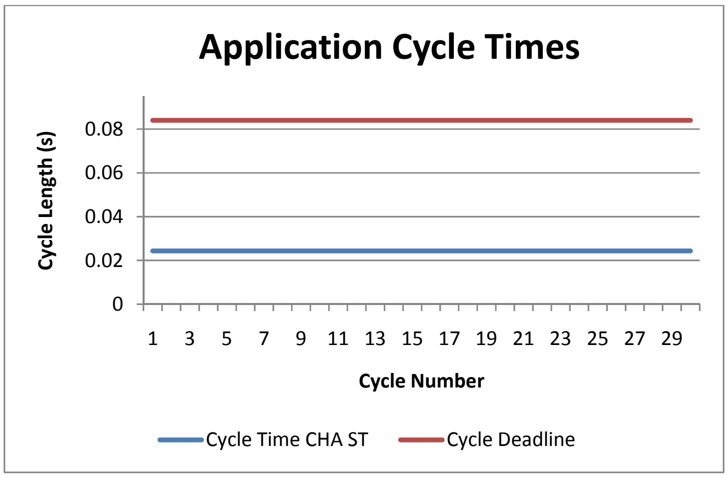
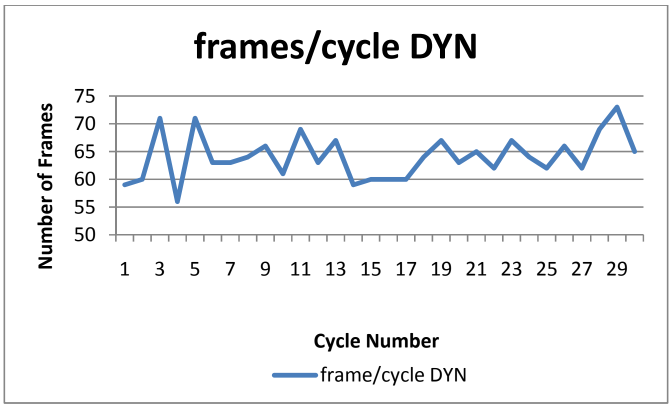


Figure 13-41 presents the ACC application cycles time. Examining if the each message deadline can be combined to complete the application cycle, before the application deadline of 84*ms*, see Figure 13-41. The maximum cycle time is 24.33*ms*, the minimum and average times are also 24.33*ms*. This consistency of application cycle times is a prime example of the deterministic properties of FlexRay. There is a standard deviation of 183*ns*.



The number of frames per ACC application cycle is static at twelve (6 frames each on channel A and B). This is in keeping with test cases 3A and 3B where redundancy is not implemented (therefore, 6 frames in total), but the application configurations have not changed. In the DYN segment the number of frames per cycle continues to be aperiodic due to the ET nature of the DYN segment. This is illustrated in Figure 13-42. There is a maximum of 73 frames per cycle in the DYN segment with an average of 64.03 frames per cycle. The standard deviation is 4.02 frames per cycle.



13.4.2.2 Case 4B: Standard at High Data Rate (Including Redundancy)

This test case is configured the same as Case 3B except redundancy is implemented. At High Data Rates (HDR) there is twice the amount of data on the bus when compared to the maximum amount of data on the CAN. Even at this HDR the FlexRay bus would be expected to have the capacity to handle more messages if it was required to do so.

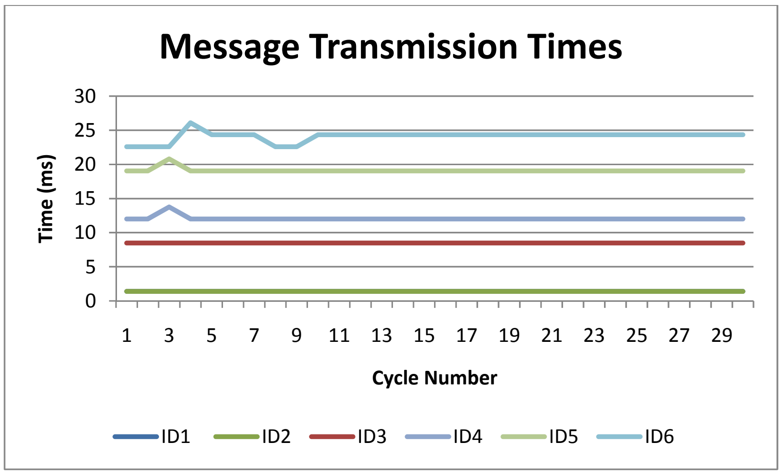
In addition to the two DYN messages transmitted from the development boards (ID7 and ID8) the bus is loaded with messages (hex) ID 9, a, b, c, d, e, f and 10. These messages are set to transmit once in every communication cycle in the DYN segment with a payload of 8 Bytes.

The busload data is presented in Table 13-16 where the redundant data is included.

Table 13-16: High Data Rate Busload

Msg ID	Min Load%	Max Load%	Average %
ST			
ID1	0.04	0.05	0.05
ID2	0.04	0.05	0.05
ID3	0.04	0.05	0.05
ID4	0.04	0.05	0.05
ID5	0.04	0.05	0.05
ID6	0.04	0.05	0.05
Total	0.27	0.28	0.27
DYN			
ID7	0.65	0.77	0.72
ID8	0.67	0.76	0.72
ID9	1.10	1.10	1.10
IDa	1.10	1.10	1.10

Figure 13-43 illustrates the message complete transmission prior to their deadlines. Fluctuations occur at plus/minus one frame cycle. This is caused by a message either being ready one cycle earlier or one cycle later than the ‘standard’ message execution time.



The timely execution of the ACC application deadlines of 84*ms* is verified in Figure 13-43.Table 13-11 contains the message deadlines. The maximum cycle time was 26.08*ms* and a minimum of 22.58*ms*. The average cycle time was 24.10*ms* which means the application has completed execution 59.9*ms* before the deadline time. The standard deviation is 759.66*μs*.

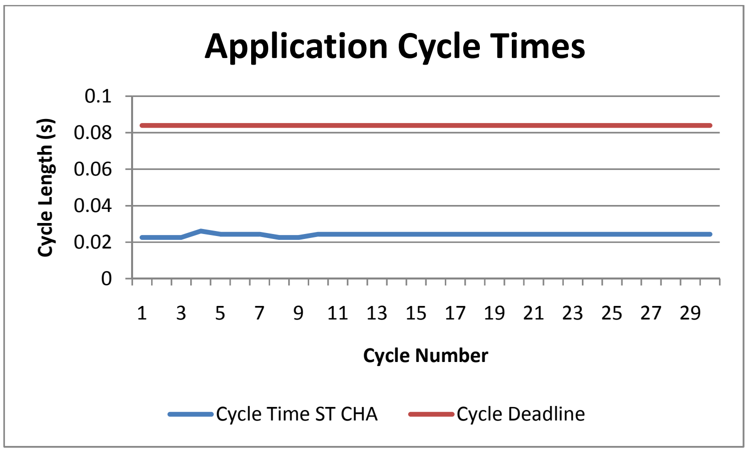
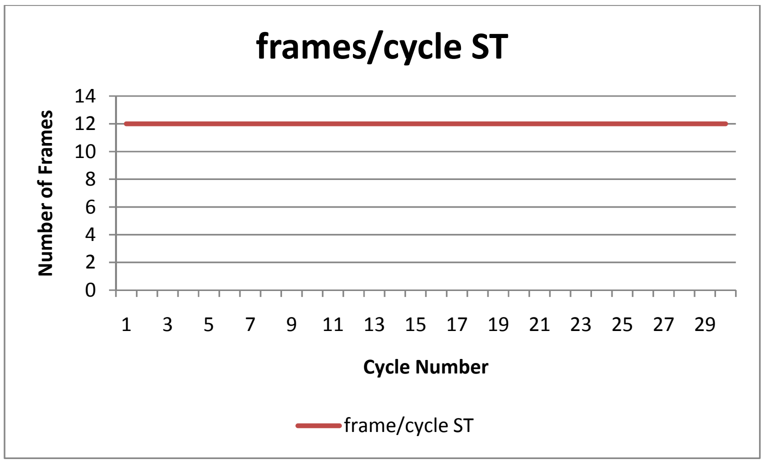


Figure 13-44: Application Cycle Times High Data Rate

There are twelve application frames in the ST segment as explained in the previous test case. This is illustrated in Figure 13-45. In the DY segment the number of frames per cycle is less cyclic due to the ET nature of the DYN segment. This is illustrated in Figure 13-46.



The DYN data contains a maximum of 458 frames per cycle with an average of 446.60 frames per cycle. The standard deviation is 6.34 frames per cycle.

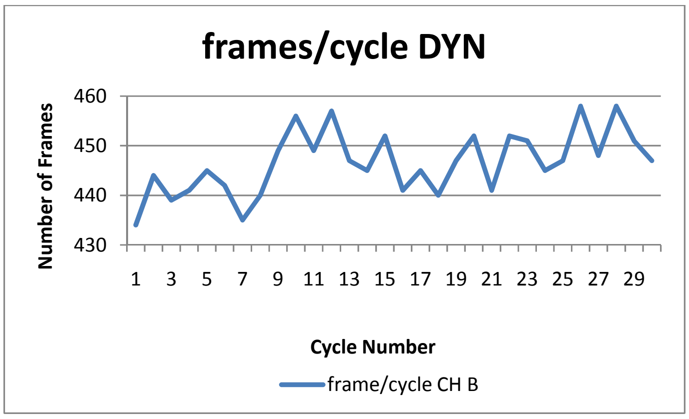


Figure 13-46: Number of DYN Frames per Cycle High Data Rate

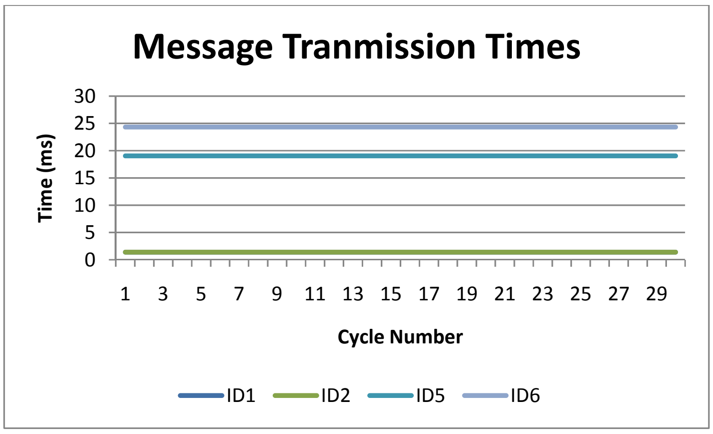
13.4.2.3 Case 4C: Minimal Normal Load (Including Redundancy)

The minimal configuration at normal busloads (no data loading) would be expected to yield slightly reduced busloads (compared to Standard configuration) in the ST segment, due to two less frames per application cycles being transmitted on the FlexRay bus per channel. Table 13-17 presents the busloads.

Table 13-17: Normal Busload

Msg ID	Minimum Load%	Maximum Load%	Average Load%
ST			
ID1	0.04	0.05	0.05
ID2	0.04	0.05	0.05
ID5	0.04	0.05	0.05
ID6	0.04	0.05	0.05
Total	0.09	0.09	0.09
DYN			

Figure 13-47 determines if the individual messages execution times meet their deadlines. Each message easily meets its set deadlines. The associated deadlines are as presented in Table 13-11.



Combining message times to build the application execution time, this enables the success of the ACC application to be determined. If the final message completes execution before the application deadline then the ACC application can successfully execute. Figure 13-48 presents this data. The maximum, minimum and average cycle times are 24.33*ms*. There is a standard deviation of 305*ns*.

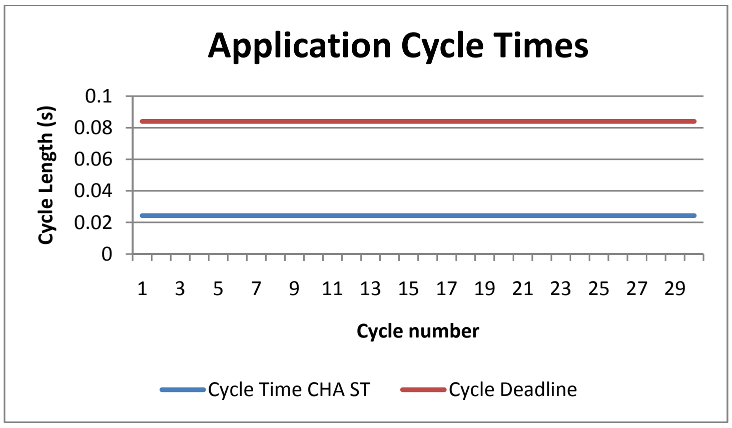
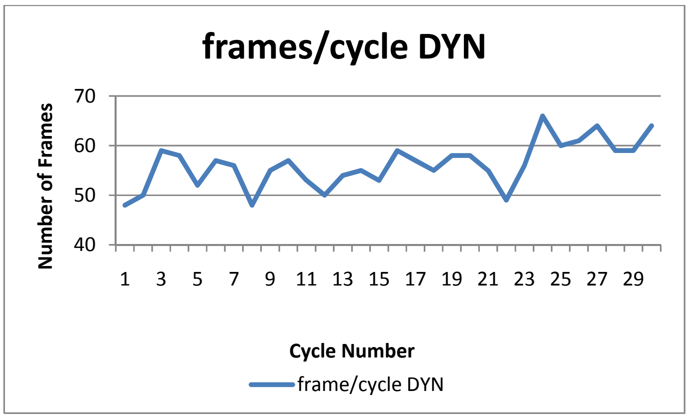


Figure 13-48: Application Cycle Times at Normal Loads

The ST segment application transmits eight application frames in total (4 per channel) as expected. This produces a horizontal straight line when graphed. In the DYN segment the number of frames per cycle is aperiodic due to the ET nature of the DYN segment. There are a maximum of 66 frames per application cycle in the DYN segment with an average of 56.16 frames per cycle and a standard deviation of 4.58 frames per cycle. Figure 13.49 illustrates the number of DYN data frames from CH B (as there is no loading in this test case all the DYN data is transmitted on CH B).



13.4.2.4 Case 4D: Minimal at High Data Rate (Including Redundancy)

This test case contains the same configuration as Case 3D with the addition of redundancy being implemented for the ACC application data in the ST segments. At High Data Rates (HDR) there is twice the amount of FlexRay data (in terms of the total number of frames) on the bus when compared to the maximum amount of data on the CAN bus. Even at this HDR the FlexRay bus would be expected to have capacity to handle additional messages.

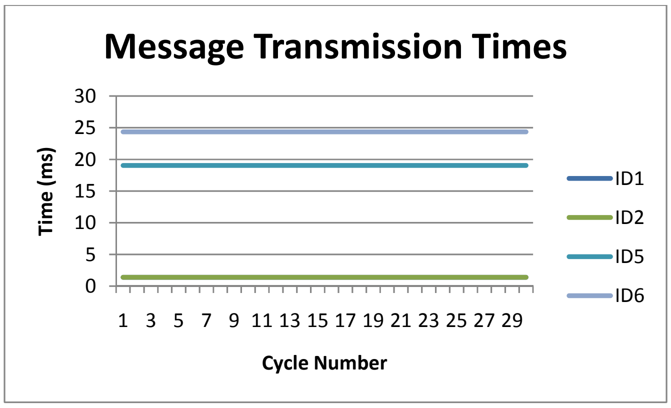
In addition to the two messages transmitted from the development boards (ID7 and ID8) the bus is loaded with additional messages of (hex) IDs 9, a, b, c, d, e, f and 10. These messages are set for transmission once in every communication cycle configured with a payload of 8 Bytes.

The busload data is presented in Table 13-18.

Table 13-18: High Data Rate Busload

Msg ID	Minimum Load%	Maximum Load%	Average Load%
ST			
ID1	0.04	0.05	0.05
ID2	0.04	0.05	0.05
ID5	0.04	0.05	0.05
ID6	0.04	0.05	0.05
Total	0.18	0.18	0.18
DYN			
ID7	0.61	0.79	0.71
ID8	0.62	0.78	0.72
ID9	1.10	1.10	1.10
IDa	1.10	1.10	1.10
IDb	1.10	1.10	1.10

Figure 15-50 illustrates that the message deadlines are not exceeded. The actual deadlines are presented in Table 13-11.



As in all previous test cases the ACC application cycle completes execution prior to the deadline; 59.67*ms* prior in this test case at a worst case scenario. Figure 13-51 illustrates the application's cycle times over a 30 application cycle period. The maximum ACC application cycle time was 24.33*ms* with the minimum and average time also 24.33*ms*. The standard deviation is 253.7*ns*.

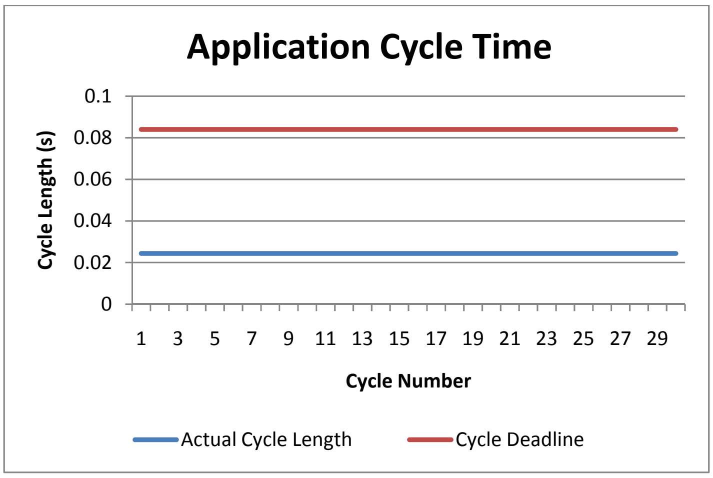
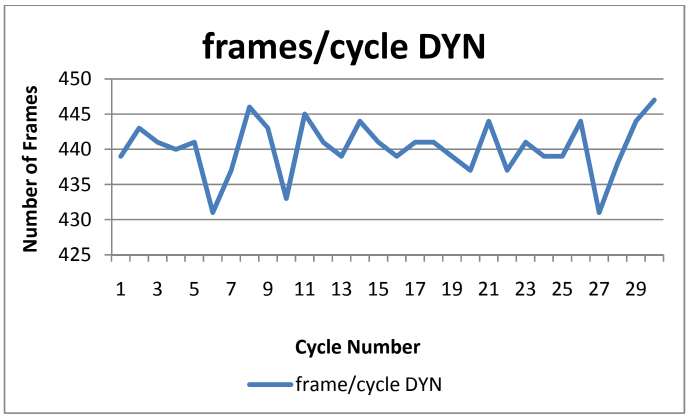


Figure 13-51: Application Cycle Times High Data Rate

The number of ACC application frames remains constant at eight for the minimal configuration. In Figure 13-52 the DYN data contains a maximum of 447 frames per cycle with an average of 440.16 frames per cycle. The standard deviation is 3.92 frames per cycle.



13.5 Discussion of Results

Using both protocols the ACC application was successfully implemented and executed in all scenarios. There is a higher through-put in the number of frames in one second in FlexRay when compared with CAN. The number of frames per second in each scenario and test case are presented in Table 13-19 and 13-20.

13.5.1 CAN Data

Table 13-19: Number of CAN Frames per Second

Standard	Busload	Min	Max	Average
Case 1A	Normal (30%)	329	391	357.31
Case 1B	60%	603	667	634.38
Case 1C	Max (100%)	1034	1040	1037.52
Minimal	**Busload**	**Min**	**Max**	**Average**
Case 2A	Normal	297	349	327.76

13.5.2 FlexRay Data

Table 13-20: Number of FlexRay Frames per Second

No Redundancy	Busload	Min	Max	Average
Case 3A	Normal	781	878	828.18
Case 3B	Normal HDR	4689	5442	5374.07
Case 3C	Minimal	741	844	799.89
Case 3D	Minimal HDR	5288	5426	5366.07
Redundancy	**Busload**	**Min**	**Max**	**Average**
Case 4A	Normal	865	934	899.29

At maximum busloads in case 1C or 2C the messages with the lowest priorities missed their deadlines. These messages ID 10, 11, 12 and 13 were gaining access to the bus later than 5*ms* on average as opposed to their scheduled time of every 1*ms*. These two test cases resulted in a maximum of 1040 and 1041 frames per second respectively.

Compare this to the FlexRay test cases of 3D and 4D. Here eight extra messages are added for transmission in the frame cycle (1.750ms). This was successfully achieved while attaining a maximum through put of 5426 and 5488 frames per second in total over both channels. In these test cases the through put of FlexRay versus CAN was five times larger for FlexRay. At 1041 frames per second the CAN bus has reached its maximum capacity busload of 96.60% where as at 5488 frames per second the FlexRay bus is at 10.43% busload.

It is recorded that at 60% loads the application cycle time was starting to increase when compared to the application time at normal busload. In case 1A the maximum application cycle is 21.12*ms*, this then increases slightly to 21.45*ms* in case 1B. In case 2A the maximum application time is 24.51*ms* but this increases to 26.04*ms* in case 2B.
In case 1A the minimum application cycle time cycle is 19.69*ms* and this is almost unchanged at 19.70*ms* in case 1B. In case 2A the minimum application cycle time is 23.54*ms* but this increase to 24.47*ms* in case 2B.
Finally in case 1A the average application execution time is 19.10*ms* compared with a time of 20.28*ms* in case 1B. In case 2A the average application cycle time is 23.77*ms* compared to the average application cycle time of 25.04*ms* in case 2B.
These individual message transmit times can be viewed individually and it is observed that as busloads increase so does the transmit time of each message. Figure 13-53 illustrates that the message transmit times are increasing as the busloads increase. Messages ID5 and ID6 transmit times increase the most as they are further into the application cycle.

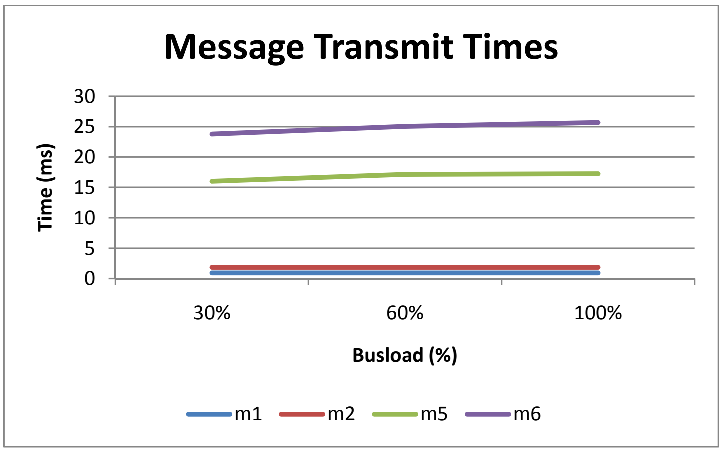


Figure 13-53 illustrates the message transmit times increasing as the busloads increase. The messages that transmit later in the application cycle are adversely affected to a greater degree than the messages transmitted earlier in the application cycle. In Figure 13-54, the transmit times of message six are delayed greater than the transmit times of message one for example.

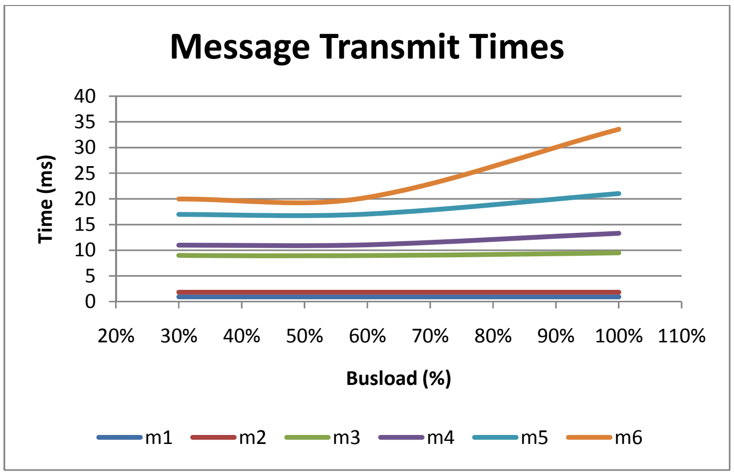


Figure 13-54: Message Transmit Times (Normal Configuration)

The deterministic nature of FlexRay is illustrated by the ACC application completion times. Seven of the eight FlexRay test cases completed transmission at 24.33*ms*, the exception being test case 3D where transmission was completed at 24.10ms.This is regardless of busloads which as illustrated in Figures 13-53 and 13-54 busloads affect the applications execution times in CAN.

Comparing these CAN values to those obtained in FlexRay produces the following results. The smallest application cycle time in FlexRay is 22.58*ms* compared with 19.69 *ms* at 357.31 frames per second in CAN. It could be stated than CAN is faster than FlexRay but to put it in context CANs quickest application cycle time was achieved at normal (30%) busload levels while FlexRay's quickest cycles time was achieved under HRD conditions with a through put of 5488 frames per second. This is where the deterministic feature of FlexRay is advantageous. Messages can be scheduled for transmission with a greater degree of certainty that those scheduled on the CAN bus. Test case three investigates this further.

Examining the standard deviations in the ACC application cycle times allows comparisons be made between the deterministic ST segment in FlexRay and the ET nature of CAN. Table 13-21.

Table 13-21: ACC CAN Application Cycle Standard Deviation

Test Case Number	**Standard Deviation (µs)**
1A	396
1B	503
1C	91.34
2A	328.43

Table 13-22 presents the standard deviations in the ACC application cycle.

Table 13-22: ACC FlexRay Application Cycle Standard Deviation

Test Case Number	Standard Deviation
3A	183ns
3B	183ns
3C	254ns
3D	559.50µs
4A	183ns

A general summarisation is that the FlexRay ACC application cycle times have a higher degree of consistency when compared to the times obtained in CAN. This results in all the CAN standard deviation timings being in units of microseconds while the FlexRay timings have six of the eight recordings in units of nanoseconds. Because of the nature of FlexRay's ST segments, unless a message transmits at the same instance in every application cycle, the actual transmission will be a multiple of the Flexray cycle earlier or later each time. In this ACC example if a message gets delayed by one frame cycle (1.750ms) then the message cannot transmit for one frame cycle. If the same scenario is examined in CAN the message is able to transmit as soon as it is ready therefore resulting in a smaller standard deviation time by not having to wait a pre-defined period.

13.6 Test Case 3: Verification of Time-Triggered Properties

First the CAN data is presented and then the FlexRay data is compared to the CAN results.

To verify the time-triggered properties of Flexray, the application execution times in CAN under various busloads are compared to the application execution times in FlexRay. Multiple busloads are not required in the FlexRay test because configuration will be implemented in such a way so the maximum amount of data (for this configuration) gets access to the bus without restrictions.

13.6.1 CAN Testing

Figure 13-55 contains the CAN configuration set up as viewed in CANalyzer. The message ID's are presented in Table 13-23. The final block generator at the end of the diagram has an X through it because the additional data was not transmitted at that stage. This function block contained IDs 4, 30, 175 and 350. These values were chosen because they gave a spread of ranges across all application IDs.

All CAN data is configured for the maximum payload of 8 bytes and the bus rate is set to 125kbit/s.

Table 13-23: Application IDs

Message ID	**CAN Block Label (Figure 12-55)**
1	Timer 40ms
20	Msg 1
50	Msg 20
150	Msg 50

Busloads of 17%, 30%, 48%, 63% and 100% are logged. This provides a gradual progression for increased busloads. At 17% busload only the application data was transmitted. At a 30%busload the loaded messages are transmitted periodically every 7ms. This is increased to every 3ms at 48% busload. At 63% busload the messages are transmitted every 2ms and finally the loaded messages are transmitted every 1ms at maximum busload.

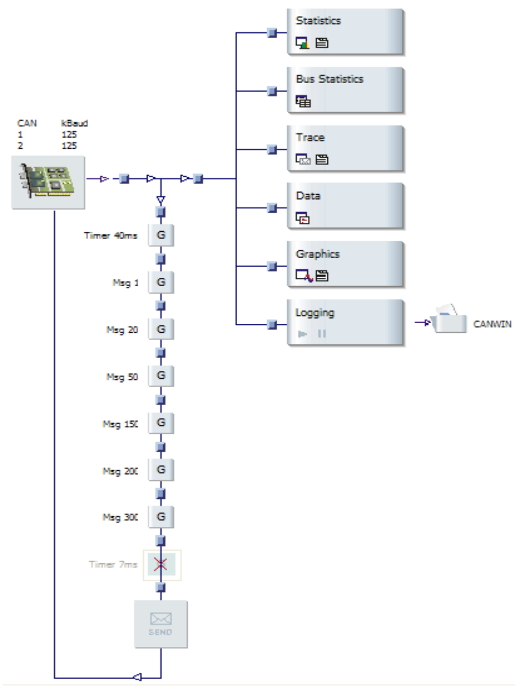
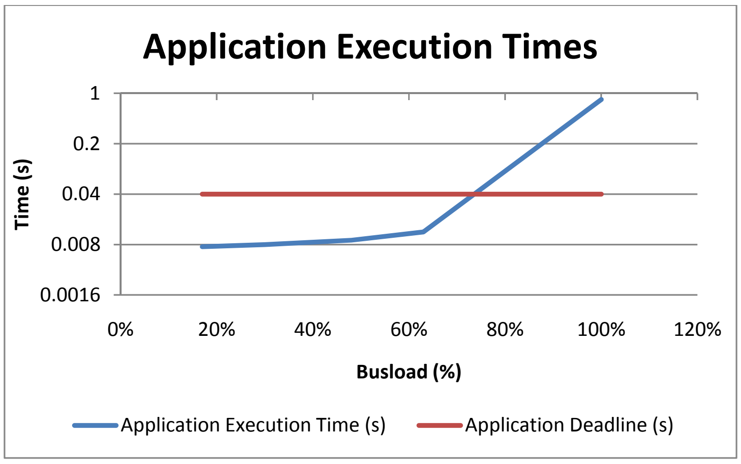


Table 13-24 presents the busload values and the associated application execution times.

Table 13-24: Application Execution Times

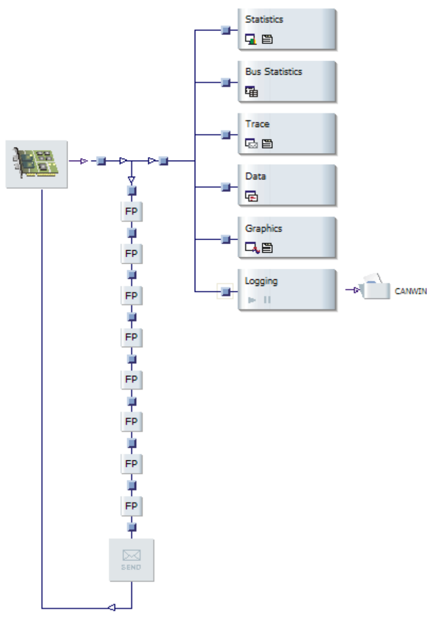
Busload (%)	App Execution Time (s)	App Deadline (s)
17	0.007472	0.04
30	0.007962	0.04
48	0.009164	0.04

Presenting this data graphically (Figure 13-56) demonstrates the extent of the CANs inability meet timing deadlines once capacity has been reached.



13.6.2 FlexRay Testing

The FlexRay data was generated using the CANalyzer configuration in Figure 13-57. The FlexRay bus was set for a data rate of 10Mbit/s. The Vector VN3600 FlexRay interface was used as a coldstart node.



Due to the frame size being set to 512μs (see section 12.7) it was decided to configure all FlexRay data (including messages not associated with the application data) to transmit in every cycle (this maximises busload). This included the application data and the loaded data. From Figure 13-57 it is observed that there are eight FlexRay generation blocks. This is due to the four extra frames not assigned to the application being transmitted from one single function block generator. This is the 'FP' block at the bottom of the stack. The IDs are assigned to each slot in the static segment. The loaded data (comprised of four messages from the CAN results, section 13.6.1) are assigned the first four slots thereby having IDs 1 to 4. IDs 5 to 11 are assigned to the remaining static slots. This configuration is chosen to demonstrate that assigning messages in any order will not have any effect on results.

By transmitting all eleven messages ID1 to 11 in every FlexRay frame cycle (512μs in length), this results in a busload of 48.75%

13.7 Conclusion

At low busload levels the CAN ACC application completes execution before the FlexRay application. As CAN busloads increase the application execution times started to increase also. Due to the configuration of the CAN application, its deadlines were likely to be missed as busloads increase. This was due to the priorities assigned to the messages not allowing the application data access to the bus as more messages are transmitted with increased frequency. The deterministic nature of FlexRay is proven through the consistency at which the ACC application completes execution.
By FlexRay having a greater than five times higher through-put (at 10% busload) than CAN (at maximum busload), this demonstrates FlexRay’s extra bandwidth feature.

The ST data obtained from CH A was the exact same as the ST data from CH B where redundancy was implemented. This further aids the “chance” of successful reception and transmission of these FlexRay frames.

FlexRay’s time-triggered properties were verified in test case three. This is proven by examining the application’s execution times. Using the CAN protocol, CAN was unable to meet its application deadline of 40*ms* once the bus was loaded with four extra messages transmitting periodically every 1*ms*. Comparing this to FlexRay where the configuration allowed all eleven messages to transmit every 512*μs*. The FlexRay application data therefore was able to complete execution before the revised deadline of 7.168*ms* deadline.

<u>Section V - Conclusion</u>

14 Conclusion:

14.1 Research Summary

This chapter provides a summary of the research carried out, the research questions are re-addressed and areas of potential in future research are suggested.
The research began with a broad but in-depth examination of automotive protocols such as CAN, LIN and FlexRay. This gave an understanding of the underlying principle of each protocol. A focus was placed on time-triggered and event-triggered protocols as FlexRay contained both elements within its frame structure. Previous works were reviewed primarily on time-triggered networks, since there was only a minimal amount of work carried out on the FlexRay protocol at the time. Further investigation into possible methods of scheduling event-triggered and time-triggered protocols was undertaken.

Investigative research was carried out into methods used by other authors in the scheduling of Time-Triggered and Event-Triggered protocols. All possible trade-offs and features of previous works that could help or hinder my work using the FlexRay protocol were defined. The features of CAN that were required on the FlexRay application were defined at this stage by deciding what would constitute a successful migration. From this a primary framework was developed for the ST and DYN segments of FlexRay. This evolved into the finished migration procedure.

Further in-depth research was required into the features of the development boards. Research was also carried out in parallel into the use and features of CANalyzer.FlexRay and DECOMSYS designer pro <light>; these programs would required in depth knowledge for the development of the FlexRay frame and cluster and also for the analysis of data. The possibilities were to generate either the CAN or

FlexRay data from CANalyzer or to generate it from the Fujitsu development boards. As the development boards provided greater flexibility for configuration purposes (in terms of the reference application) this method was used where possible.

14.2 Summary of Testing and Results

To verify the framework three test cases were devised. Each test case provided the framework to process a different application task graph configuration.

Test Case 1:

The abstract Traction Control Application was processed through the framework to extract FlexRay frame and cluster parameters. The task periods used were taken from data logged from production vehicles (Peugeot 207 and an electric Smart Car). Upon extraction of FlexRay parameters, these parameters were processed using the DECOMSYS designer tool to reveal any parameter violations. The aim of this test case was to extract the FlexRay parameters; therefore it was not implemented in hardware.

Test Case 2

Following this a reference ACC application was processed through the framework. This application was configured on the development boards and associated data logged. Using the ACC application allowed the framework to process a different task graph structure. This allowed the ACC application to be processed through the framework to extract the associated FlexRay frame and cluster parameters. Using these extracted parameters the ACC application was implemented in FlexRay. The associated data was logged using CANalyzer. Using the parameters extracted from the framework the FlexRay configuration of the ACC was setup. This was implemented on the development boards and data was logged in CANalyzer.

Busloads and deadlines were examined from the CAN results and compared to those obtained from FlexRay, to determine if the FlexRay application performed comparable to the CAN case.

The end results demonstrated that at low data rates CAN performed marginally superior than FlexRay. This was due to the ET nature of the CAN protocol allowing the messages transmit on the bus when required, where as in FlexRay data is only granted access to the bus according to a schedule. Once data rates increased FlexRay provided deterministic and constant timings. The extra capacity of FlexRay was also demonstrated without compromising the ACC application deadlines.

Test Case 3

The aim of third and final test case was to verify FlexRay's time-triggered properties by demonstrating its constant execution of the test application when compared to CAN. The test application was processed through the framework to obtain necessary FlexRay parameters. The test application for both CAN and FlexRay were implemented using CANalyzer's function block generators. These parameters were then implemented in CANalyzer on a FlexRay network. The application timings for CAN were logged over five busloads starting at 17% and finishing at maximum load. The FlexRay data was logged at a busload level of 48%. At this level FlexRay was able to transmit the same number of messages at a higher frequency than CAN was able to cope with. In addition to this the FlexRay application execution timings we transmitted every FlexRay frame cycle. Thereby the consistency of FlexRay's execution of the application was proven, where as CAN failed to meet deadlines under the maximum busload.

14.3 Research Questions

Question 1:

What are the benefits of using the migration framework versus the use of a gateway?

The full migration procedure reduces the complexity associated with having multiple protocols in operation side by side. This is achieved by employing a single protocol that allows the applications features continued use, while at the same time utilising other feature of the newer protocol. A gateway might be more suitable if a partial migration meets system requirements, where as complete migration reduces system complexity.

Question 2:

What migration techniques used in other or similar protocols are applicable in this research?

FlexRay is a unique protocol and there is no other protocol exactly like but it there are similar protocols like TTCAN for example. Also using some features of protocols that have commonalities to segments within FlexRay can aid in the development of a migration procedure. The migration techniques applicable to this research include task graph analysis, WCET analysis and RTA analysis.

Question 3:

What parameters are required in relation to the application and network protocol for migration to be undertaken?

The application parameters extracted from the framework are;

- Task graph release (start) time r_i
- Task graph deadline (end) time D_i
- Worst Case Execution Time (*WCET*)
- Task period

These parameters require the CAN application to be presented in task graph format before application parameters are extracted.

The frame parameters that the framework provides are;

- Frame Length
- Static Segment Size
- Payload Size
- Static Slot Size
- NIT
- DYN Segment Size

These parameters are extracted from the framework and can be input to a FlexRay designer tool to determine the associated parameters, if they are not already known. These frame and cluster parameters allow the configuration and implementation of the FlexRay application.

14.4 Areas for Future Research

The migration framework was a success in extracting the FlexRay parameters necessary for migration, but there is room for further research in some areas;

- In the reference application, the only method of recording an event was when it appeared on the bus. To obtain more accurate timings it is suggested to try and record when a message is generated and received by the MCU. In this thesis this was attempted through the UART but this was not practical due to the delay this caused on the message timings.

- While every effort was made to ensure the test cases were as close to real world a possible, to truly determine the effectiveness of the framework and parameters they should applied to a live applications on automobiles.

- Automate the migration process thereby providing initial configuration parameters allowing the final FlexRay parameters to be derived.

- Perform parameter validation checks in the framework instead of using a separate design tool. This would speed up the migration procedure.

- A cost benefit analysis could be carried out to define a breakeven point in the migration process.

Section VI- Appendices

Appendix A

14.5 Published Material

An oral presentation and a paper were presented at the AAE conference 2008. This took place at the RBS Williams F1 centre, Oxfordshire, on the 23rd of September 2008. The presentation and paper were based on the research presented in this thesis. The paper and presentation were titled “CAN to FlexRay Migration Framework”

Appendix B

14.6 Bibliography

Ajay K. Verma P B, Paolo Ienne, 2007 Progressive Decomposition:
A Heuristic to Structure Arithmetic Circuits
and R R a A H and Erns R, 2007 Automotive Software Integration
Bing Wu D L, Jesus Bisba, Ray Richardson, and Jane Grimson V W, Donie O'Sullivan,1997, The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems, Proceedings of the 3rd International Conference on Engineering of Complex Computer Systems (ICECCS'97),
Broy M,2006, Challenges in Automotive Software Engineering, ICSE'06,
Christoph A. Herrmann A B, Kevin Hammond, and Steffen Jost H-W L a R P, 2007, Automatic Amortised Worst-Case Execution Time Analysis.
Chuan Ku Lin H-W Y, Mu-Song Chen, and Chi-Pan Hwang 2007 A Neural Network Approach for Controller Area
Network Message Scheduling Control *IAENG International Journal of Computer Science*
Fischerkeller S R a R 2007 Latest Trends In Automotive Electronic Systems -
Highway Meets Off-Highway? *Agricultural Engineering International: the CIGR Ejournal*
GmbH R B 2004 *BOSCH Automotive Handbook 6th edition*
Guido Gehlen E W, Sven Lukas, Carl-Herbert Rokitansky and Bernhard Walke, 2006, Architecture of a Vehicle Communication Gateway for Media Independent Handover.
Harald Heinecke J B, BMW Group, Klaus-Peter Schnelle N M, Bosch, Helmut Fennel O W, Continental, Thomas Weber J R, DaimlerChrysler, Lennart Lundh T S, Ford Motor Company, Peter Heitkämper R R, General Motors, Jean Leflour A G, PSA Peugeot Citroën, Ulrich Virnich S V, Siemens VDO, Kenji Nishikawa K K, Toyota Motor Corporation and Thomas Scharnhorst B K, Volkswagen,2006, AUTOSAR – Current results and preparations for exploitation, 7th EUROFORUM conference 'Software in the vehicle, Stuttgart, Germany.

Helmut Fennel S B, Continental, Harald Heinecke J B, Simon Fürst, BMW Group, Klaus-Peter Schnelle W G, Nico Maldener, Bosch, Thomas Weber F W, Jens Ruh, DaimlerChrysler, Lennart Lundh T S, Ford Motor Company, Peter Heitkämper R R, General Motors, Jean Leflour A G, PSA Peugeot Citroën, Ulrich Virnich S V, Siemens VDO, Kenji Nishikawa K K, Toyota Motor Corporation and Klaus Lange T S, Bernd Kunkel, Volkswagen 2006 Achievements and exploitation of the AUTOSAR development Partnership

Lei JU A R a S C, 2007, Schedulability Analysis of MSC-based System Models. national university of Singapore)

Obermaisser R, 2006, Reuse of CAN-based Legacy Applications in
Time-Triggered Architectures.

Paskvan J R P a J, 2007 Experimental Jitter Analysis in a FlexCAN Based Drive-by-
Wire Automotive Application

R.C.Papademetriou V P D A K,2006, Heuristic Algorithms for Efficient Wireless Multimedia Network Design, Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06),

Rausch M, 2006, FlexRay Speeds automotive safety applications. Automotive DEsign Line.com)www.automotivedesignline.com

Reinhard Wilhelm J E, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley G B, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, and Frank Mueller I P, Peter Puschner, Jan Staschulat, Per Stenstr¨om, 2001 The Worst-Case Execution Time Problem
— Overview of Methods and Survey of Tools

Schoeberl R K a M, 2007 Modeling the Function Cache for Worst-Case Execution
Time Analysis¤

Schwefel J J M a H-P, 2005, Markov Chain-based Performance Analysis of FlexRay Dynamic Segment. ed A Hadan

Spuri M, 1996, Holistic Analysis for Deadline Scheduled
Real-Time Distributed Systems.

Steininger E A a A, 2006, Automatic Parameter Identification in FlexRay based on Automotive Communication Networks. ed M Horauer

Suk-Hyun Seol S-w L, Sung-Ho Hwang and Jae Wook Jeon,2006, Development of Network Gateway Between CAN and FlexRay Protocols
For ECU Embedded Systems, SICE-ICASE International Joint Conference 2006, Bexco, Busan, Korea.
Suri V C a N,2004, Event-Triggered Channels on a Time-Triggered Base, Ninth IEEE international cinference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age,
Thomas Noltey H H, Lucia Lo Belloz, 2005, Automotive Communications - Past, Current and Future.
ZP. Tao M G, 1992 Synthesizing Communication Protocol Converter:
A Model and Method

Appendix C

14.7 Calculations

Abstract Implementation Calculations

Parameter C_m & J_m Identification

Message Size (Bytes)	Bit Size Equation	Message Size (Bits)	C_m Equation	C_m *(ms)*	J_m *(µs)*
8	55+(10*8)	135	135*8µs	1.08	600
5	55+(10*5)	105	105*8µs	0.84	700
8	55+(10*8)	135	135*8µs	1.08	400
8	55+(10*8)	135	135*8µs	1.08	800
3	55+(10*3)	85	85*8µs	0.68	1100
7	55+(10*7)	125	125*8µs	1.00	900
8	55+(10*8)	135	135*8µs	1.08	100
8	55+(10*8)	135	135*8µs	1.08	100
8	55+(10*8)	135	135*8µs	1.08	200
2	55+(10*2)	75	75*8µs	0.60	400

Appendix D

w_m First Iteration

Queue Delay	**w_m Equation**	***w_m (ms)***
w1	1.08ms+(((0ms+0.6ms+8µs)/100ms)*1.08ms)	1.086566
w2	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms))	1.089540
w3	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms), (((0ms+0.4ms+8µs)/100ms)*1.08ms))	1.093946
w4	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms),((0ms+0.4ms+8µs)/100ms)*1.08ms),(((0+0.8ms+8µ)/100ms)*1.08ms))	1.102673
w5	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms),(((0ms+0.4ms+8µs)/100ms)*1.08ms),(((0+0.8ms+8µ)/100ms)*1.08ms),(((0+1.1ms+8µs)/500ms)*0.68ms))	1.104180
w6	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms),(((0ms+0.4ms+8µs)/100ms)*1.08ms),(((0+0.8ms+8µ)/100ms)*1.08ms),(((0+1.1ms+8µs)/500ms)*0.68ms),(((0+0.9ms+8µs)/100ms)*1ms))	1.113260
w7	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms),(((0ms+0.4ms+8µs)/100ms)*1.08ms),(((0+0.8ms+8µ)/100ms)*1.08ms),(((0+1.1ms+8µs)/500ms)*0.68ms),(((0+0.9ms+8µs)/100ms)*1ms),(((0+0.1ms+8µs)/500ms)*1.08ms))	1.113493
w8	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms),(((0ms+0.4ms+8µs)/100ms)*1.08ms),(((0+0.8ms+8µ)/100ms)*1.08ms),(((0+1.1ms+8µs)/500ms)*0.68ms),(((0+0.9ms+8µs)/100ms)*1ms),(((0+0.1ms+8µs)/500ms)*1.08ms),(((0+0.1ms+8µs)/100ms)*1.08ms))	1.114659
w9	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms),(((0ms+0.4ms+8µs)/100ms)*1.08ms),(((0+0.8ms+8µ)/100ms)*1.08ms),(((0+1.1ms+8µs)/500ms)*0.68ms),(((0+0.9ms+8µs)/100ms)*1ms),(((0+0.1ms+8µs)/500ms)*1.08ms),(((0+0.1ms+8µs)/100ms)*1.08ms),(((0+0.2ms+8µs)/500ms)*1.08ms))	1.115109
w10	1.08ms+Sum((((0ms+0.6ms+8µs)/100ms)*1.08ms),(((0ms+0.7ms+8µs)/200ms)*0.84ms),(((0ms+0.4ms+8µs)/100ms)*1.08ms),(((0+0.8ms+8µ)/100ms)*1.08ms),(((0+1.1ms+8µs)/500ms)*0.68ms),(((0+0.9ms+8µs)/100ms)*1ms),(((0+0.1ms+8µs)/500ms)*1.08ms),(((0+0.1ms+8µs)/100ms)*1.08ms),(((0+0.2ms+8µs)/500ms)*1.08ms),(((0+0.4ms+8µs)/400ms)*0.60ms)	1.115721

w_m Second Iteration

Queue Delay	***w_m* Equation**	***w_m (ms)***
w1	1.08ms+(((1.086566ms+0.6ms+8µs)/100ms)*1.08ms)	1.09830
w2	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms))	1.10127
w3	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms))	1.10568
w4	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms),(((1.102673ms+0.8ms+8µ)/100ms)*1.08ms))	1.11440
w5	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms),(((1.102673ms+0.8ms+8µ)/100ms)*1.08ms),(((1.104180ms+1.1ms+8µs)/500ms)*0.68ms))	1.11591
w6	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms),(((1.102673ms+0.8ms+8µ)/100ms)*1.08ms),(((1.104180ms+1.1ms+8µs)/500ms*0.68ms),(((1.113260ms+0.9ms+8µs)/100ms)*1ms))	1.12499
w7	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+07ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms),(((1.102673ms+0.8ms+8µ)/100ms)*1.08ms),(((1.104180ms+1.1ms+8µs)/500ms)*0.68ms),(((1.113260ms+0.9ms+8µs)/100ms)*1ms),(((1.113493ms+0.1ms+8µs)/500ms)*1.08ms))	1.12522
w8	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms),(((1.102673ms+0.8ms+8µ)/100ms)*1.08ms),(((1.104180ms+1.1ms+8µs)/500ms)*0.68ms),(((1.113260ms+0.9ms+8µs)/100ms)*1ms),(((1.113493ms+0.1ms+µs)/500ms)*1.08ms),(((1.114659ms+0.1ms+8µs)/100ms)*1.08ms))	1.12639
w9	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms),(((1.102673ms+0.8ms+8µ)/100ms)*1.08ms),(((1.104180ms+1.1ms+8µs)/500ms)*0.68ms),(((1.113260ms+0.9ms+8µs)/100ms)*1ms),(((1.113493ms+0.1ms+8µs)/500ms)*1.08ms),(((1.114659ms+0.1ms+8µs)/100ms)*1.08ms),(((1.115109ms+0.2ms+8µs)/500ms)*1.08ms))	1.12684
w10	1.08ms+Sum((((1.086566ms+0.6ms+8µs)/100ms)*1.08ms),(((1.089540ms+0.7ms+8µs)/200ms)*0.84ms),(((1.093946ms+0.4ms+8µs)/100ms)*1.08ms),(((1.102673ms+0.8ms+8µ)/100ms)*1.08ms),(((1.104180ms+1.1ms+8µs)/500ms)*0.68ms),(((1.113260ms+0.9ms+8µs)/100ms)*1ms),(((1.113493ms+0.1ms+8µs)/500ms)*1.08ms),(((1.114659ms+0.1ms+8µs)/100ms)*1.08ms),(((1.115109ms+0.2ms+8µs)/500ms)*1.08ms),(((1.115721ms+0.4ms+8µs)/400ms)*0.60ms)	1.12745

Appendix D

w_m third Iteration

Queue Delay	*w_m* **Equation**	*w_m* ***(ms)***
w1	1.08ms+(((1.098301ms +0.6ms+8µs)/100ms)*1.08ms)	1.098428
w2	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms))	1.101402
w3	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms))	1.105808
w4	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114408 ms +0.8ms+8µ)/100ms)*1.08ms))	1.114534
w5	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114408 ms +0.8ms+8µ)/100ms)*1.08ms),(((1.115915ms+1.1ms+8µs)/500ms)*0.68ms))	1.116041
w6	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114408 ms +0.8ms+8µ)/100ms)*1.08ms),(((1.115915ms+1.1ms+8µs)/500ms*0.68ms),(((1.124995ms+0.9ms+8µs)/100ms)*ms))	1.125121
w7	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+07ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114408ms+0.8ms+8µ)/100ms)*1.08ms),(((1.115915ms+1.1ms+8µs)/500ms)*0.68ms),(((1.124995ms+0.9ms+8µs)/100ms)*1ms),(((1.125228 ms +0.1ms+8µs)/500ms)*1.08ms))	1.125355
w8	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114408 ms +0.8ms+8µ)/100ms)*1.08ms),(((1.115915ms+1.1ms+8µs)/500ms)*0.68ms),(((1.124995ms+0.9ms+8µs)/100ms)*1ms),(((1.125228ms+0.1ms+µs)/500ms)*1.08ms),(((1.126394ms+0.1ms+8µs)/100ms)*1.08ms))	1.126521
w9	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114408s0.8ms+8µ)/100ms)*1.08ms),(((1.115915ms1.1ms+8µs)/500ms)*0.68ms),(((1.124995ms+0.9ms+8µs)/100ms)*1ms),(((1.125228ms+0.1ms+8µs)/500ms)*1.08ms),(((1.126394ms+0.1ms+8µs)/100ms)*1.08ms),(((1.126844ms+0.2ms+8µs)/500ms)*1.08ms))	1.126970
w10	1.08ms+Sum((((1.098301ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101275ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105681ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114408 ms +0.8ms+8µ)/100ms)*1.08ms),(((1.115915ms+1.1ms+8µs)/500ms)*0.68ms),(((1.124995ms+0.9ms+8µs)/100ms)*1ms),(((1.125228ms+0.1ms+8µs)/500ms)*1.08ms),(((1.126394ms+0.1ms+8µs)/100ms)*1.08ms),(((1.126844ms+0.2ms+8µs)/500ms)*1.08ms),(((1.127456 ms +0.4ms+8µs)/400ms)*0.60ms)	1.127582

w_m Fourth Iteration

Queue Delay	w_m Equation	w_m *(ms)*
w1	1.08ms+(((1.098428ms +0.6ms+8µs)/100ms)*1.08ms)	1.098429
w2	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+0.7ms+8µs)/200ms)*0.84ms))	1.101403
w3	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402s+0.7ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms))	1.105809
w4	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114534ms+0.8ms+8µ)/100ms)*1.08ms))	1.114536
w5	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114534 ms +0.8ms+8µ)/100ms)*1.08ms),(((1.116041ms +1.1ms+8µs)/500ms)*0.68ms))	1.116043
w6	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114534 ms +0.8ms+8µ)/100ms)*1.08ms),(((1.116041ms+1.1ms+8µs)/500ms*0.68ms),(((1.125121ms+0.9ms+8µs)/100ms)*ms))	1.125123
w7	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+07ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114534ms+0.8ms+8µ)/100ms)*1.08ms),(((1.116041ms+1.1ms+8µs)/500ms)*0.68ms),(((1.125121ms+0.9ms+8µs)/100ms)*1ms),(((1.125355 ms +0.1ms+8µs)/500ms)*1.08ms))	1.125356
w8	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114534ms+0.8ms+8µ)/100ms)*1.08ms),(((1.116041ms+1.1ms+8µs)/500ms)*0.68ms),(((1.125121ms+0.9ms+8µs)/100ms)*1ms),(((1.125355ms+0.1ms+µs)/500ms)*1.08ms),(((1.126521ms+0.1ms+8µs)/100ms)*1.08ms))	1.126522
w9	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114534ms+0.8ms+8µ)/100ms)*1.08ms),(((1.116041ms+1.1ms+8µs)/500ms)*0.68ms),(((1.125121ms+0.9ms+8µs)/100ms)*1ms),(((1.125355ms+0.1ms+8µs)/500ms)*1.08ms),(((1.126521ms+0.1ms+8µs)/100ms)*1.08ms),(((1.126970ms+0.2ms+8µs)/500ms)*1.08ms))	1.126972
w10	1.08ms+Sum((((1.098428ms+0.6ms+8µs)/100ms)*1.08ms),(((1.101402ms+0.7ms+8µs)/200ms)*0.84ms),(((1.105808ms+0.4ms+8µs)/100ms)*1.08ms),(((1.114534ms+0.8ms+8µ)/100ms)*1.08ms),(((1.116041ms+1.1ms+8µs)/500ms)*0.68ms),(((1.125121ms+0.9ms+8µs)/100ms)*1ms),(((1.125355ms+0.1ms+8µs)/500ms)*1.08ms),(((1.126521ms+0.1ms+8µs)/100ms)*1.08ms),(((1.126970ms+0.2ms+8µs)/500ms)*1.08ms),(((1.127582 ms +0.4ms+8µs)/400ms)*0.60ms)	1.127584

w_m Fifth Iteration

Queue Delay	***w_m* Equation**	***w_m (ms)***
w1	1.08ms+(((1.098429ms +0.6ms+8μs)/100ms)*1.08ms)	1.098429
w2	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms))	1.101403
w3	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms))	1.105809
w4	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms),(((1.114536 ms +0.8ms+8μ)/100ms)*1.08ms))	1.114536
w5	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms),(((1.114536 ms +0.8ms+8μ)/100ms)*1.08ms),(((1.116043ms +1.1ms+8μs)/500ms)*0.68ms))	1.116043
w6	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms),(((1.114536ms+0.8ms+8μ)/100ms)*1.08ms),(((1.116043ms+1.1ms+8μs)/500ms*0.68ms),(((1.125123ms+0.9ms+8μs)/100ms)*ms))	1.125123
w7	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+07ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms),(((1.114536ms+0.8ms+8μ)/100ms)*1.08ms),(((1.116043ms1.1ms+8μs)/500ms)*0.68ms),(((1.125123ms+0.9ms+8μs)/100ms)*1ms),(((1.125356 ms +0.1ms+8μs)/500ms)*1.08ms))	1.125356
w8	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms),(((1.114536ms+0.8ms+8μ)/100ms)*1.08ms),(((1.116043ms+1.1ms+8μs)/500ms)*0.68ms),(((1.125123ms+0.9ms+8μs)/100ms)*1ms),(((1.125356ms+0.1ms+μs)/500ms)*1.08ms),(((1.126522ms+0.1ms+8μs)/100ms)*1.08ms))	1.126522
w9	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms),(((1.114536ms+0.8ms+8μ)/100ms)*1.08ms),(((1.116043ms+1.1ms+8μs)/500ms)*0.68ms),(((1.125123ms+0.9ms+8μs)/100ms)*1ms),(((1.125356ms+0.1ms+8μs)/500ms)*1.08ms),(((1.126522ms+0.1ms+8μs)/100ms)*1.08ms),(((1.126972ms+0.2ms+8μs)/500ms)*1.08ms))	1.126972
w10	1.08ms+Sum((((1.098429ms+0.6ms+8μs)/100ms)*1.08ms),(((1.101403ms+0.7ms+8μs)/200ms)*0.84ms),(((1.105809ms+0.4ms+8μs)/100ms)*1.08ms),(((1.114536ms+0.8ms+8μ)/100ms)*1.08ms),(((1.116043ms+1.1ms+8μs)/500ms)*0.68ms),(((1.125123ms+0.9ms+8μs)/100ms)*1ms),(((1.125356ms+0.1ms+8μs)/500ms)*1.08ms),(((1.126522ms+0.1ms+8μs)/100ms)*1.08ms),(((1.126972ms+0.2ms+8μs)/500ms)*1.08ms),(((1.127584 ms +0.4ms+8μs)/400ms)*0.60ms)	1.127584

Obtaining Slack

Path	Tasks on Path	Execution Times of Tasks on Chosen Path (ms)	Total Execution Time (ms)	Calculate Slack per Path per Task (ms)	Final Slack per Path per Task (ms)
P1	T1,T5,T8,T9	2.778429+2.896043+ 2.306522+2.406972	10.38797	(2600-0.01038797)/4	647.4030
P2	T2,T5,T8,T9	2.641403+2.896043+ 2.306522+2.406972	10.25094	(2600-0.01025094)/4	647.4373
P3	T3,T5,T8,T9	2.585809+2.896043+ 2.306522+2.406972	10.19535	(2600-0.01019535)/4	6474512
P4	T4,T5,T8,T9	2.585809+2.896043+ 2.306522+2.406972	10.60407	(2600-0.01060407)/4	647.3490
P5	T1,T5,T8,T10	2.778429+2.896043+ 2.306522+2.127584	10.10858	(2600-0.01010858)/4	647.4729
P6	T2,T5,T8,T10	2.641403+2.896043+ 2.306522+2.127584	9.971552	(2600-0.009971552)/4	647.5071
P7	T3,T5,T8,T10	2.585809+2.896043+ 2.306522+2.127584	9.915958	(2600-0.009915958)/4	647.5210
P8	T4,T5,T8,T10	2.585809+2.896043+ 2.306522+2.127584	10.32468	(2600-0.009915958)/4	647.4188
P9	T6,T8,T9	3.025123+2.306522+ 2.406972	7.738617	(2600-0.007738617)/3	864.0871
P10	T6,T8,T10	3.025123+2.306522+ 2.127584	7.459229	(2600-0.007459229)/3	864.1803
P11	T7,T8,T9	2.305356+2.306522+ 2.406972	7.018850	(2600-0.007018850)3	864.3270
P12	T7,T8,T10	2.305356+2.306522+ 2.127584	6.739462	(2600-0.006739462)/3	864.4202

Payload Calculations

Payload Size	FlexRay Frame Size	Calculation of the Number of Messages Required	Number of Messages Required	Calculation for Total Bytes	Total Bytes
1	15	(8/1)+(5/1)+(8/1)+(8/1)+(3/1)+(7/1)+(8/1)+(8/1)+(8/1)+(2/1)	65	65*15	975
2	16	(8/2)+(5/2)+(8/1)+(8/1)+(3/1)+(7/2)+(8/1)+(8/1)+(8/1)+(2/1)	34	34*16	544
3	17	(8/3)+(5/1)+(8/1)+(8/1)+(3/1)+(7/3)+(8/1)+(8/1)+(8/1)+(2/1)	25	25*17	425
4	18	(8/4)+(5/1)+(8/1)+(8/1)+(3/1)+(7/4)+(8/1)+(8/1)+(8/1)+(2/1)	18	18*18	324
5	19	(8/5)+(5/1)+(8/1)+(8/1)+(3/1)+(7/5)+(8/1)+(8/1)+(8/1)+(2/1)	17	17*19	323
6	20	(8/6)+(5/1)+(8/1)+(8/1)+(3/1)+(7/6)+(8/1)+(8/1)+(8/1)+(2/1)	17	17*20	340
7	21	(8/7)+(5/1)+(8/1)+(8/1)+(3/1)+(7/7)+(8/1)+(8/1)+(8/1)+(2/1)	16	16/21	336
8	22	(8/8)+(5/1)+(8/1)+(8/1)+(3/1)+(7/8)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*22	220
9	23	(8/9)+(5/1)+(8/1)+(8/1)+(3/1)+(7/9)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*23	230
10	24	(8/10)+(5/1)+(8/1)+(8/1)+(3/1)+(7/10)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*24	240
11	25	(8/11)+(5/1)+(8/1)+(8/1)+(3/1)+(7/11)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*25	250
12	26	(8/12)+(5/1)+(8/1)+(8/1)+(3/1)+(7/12)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*26	260
13	27	(8/13)+(5/1)+(8/1)+(8/1)+(3/1)+(7/13)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*27	270
14	28	(8/14)+(5/1)+(8/1)+(8/1)+(3/1)+(7/14)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*28	280
15	29	(8/15)+(5/1)+(8/1)+(8/1)+(3/1)+(7/15)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*29	290
16	30	(8/16)+(5/1)+(8/1)+(8/1)+(3/1)+(7/16)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*30	300
17	31	(8/17)+(5/1)+(8/1)+(8/1)+(3/1)+(7/17)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*31	310
18	32	(8/18)+(5/1)+(8/1)+(8/1)+(3/1)+(7/18)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*32	320
19	33	(8/19)+(5/1)+(8/1)+(8/1)+(3/1)+(7/19)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*33	330
20	34	(8/20)+(5/1)+(8/1)+(8/1)+(3/1)+(7/20)+(8/1)+(8/1)+(8/1)+(2/1)	10	10*34	340

Experimental Implementation Calculations

Obtaining Slack

Path	**Tasks on Path**	**Execution Times of Tasks on Chosen Path (ms)**	**Total Execution Time (ms)**	**Calculate Slack per Path per Task (ms)**	**Final Slack per Path per Task (ms)**
P1	T1,T3,T4,T5, T6	0+0+6+2+6+2	16	(120-16)/6	17.3333
P2	T2,T3,T4,T5, T6	0+0+6+2+6+2	16	(120-16)/6	17.3333

Payload Calculations

Payload Size	**FlexRay Frame Size**	**Calculation of the Number of Messages Required**	**Number of Messages Required**	**Cacl for Total Bytes**	**Total Bytes**
1	15	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	40	40*15	600
2	16	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	20	20*16	320
3	17	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	15	15*17	255
4	18	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	10	10*18	180
5	19	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	10	10*19	190
6	20	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	10	10*20	200
7	21	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	10	10/21	210
8	22	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*22	110
9	23	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*23	115
10	24	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*24	120
11	25	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*25	125
12	26	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*26	130
13	27	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*27	135
14	28	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*28	140
15	29	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*29	145
16	30	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*30	150
17	31	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*31	155
18	32	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*32	160
19	33	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*33	165
20	34	(8/1)+(8/1)+(8/1)+(8/1)+(8/1)+(8/1)	5	5*34	170