REAL-TIME OPTIMISATION
OF TTCAN NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTEMENT OF ENGINEERING TECHNOLOGY
OF WATERFORD INSTITUTE OF TECHNOLOGY
IN COMPLETE FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ENGINEERING

By:
Henry L. Acheson.

Supervised By:
John Manning.

June 2007

Dedicated to:

My Wife: Patricia

And

My Children: Sinéad, Paul, Susan, and Gail.

Declaration

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

Signed: _____

Date: _____

Acknowledgements

I hereby acknowledge the contributions to this thesis and offer my thanks to the people who have helped and supported me during my work over the past three years.

My Family: I would like to formally acknowledge the tremendous patience, understanding an encouragement shown to me by my wife Patricia and my children Sinéad, Paul, Susan, and Gail.

My Supervisor: Mr. John Manning, I would like to express my gratitude to John for his continuous supervision, support, assistance, and invaluable guidance in all aspects of the research.

Additional Support: I would like to thank Mr Denis O'Shea for his help and cooperation over the past three years and the financial support of Waterford Institute of Technology .

There are also many other people who have contributed in both direct and indirect ways, and they deserve my thanks – Thank you all.

Abstract

Controller Area Network is widely used as a communications network in automotive applications, typically motor cars, commercial vehicles, and utility vehicles. CAN is operated either by spontaneous messaging or by time triggered messaging. Time triggered messaging is the preferred option on modern systems as it allows all messages access to the bus at some defined period in time. Using time triggered messaging alone does not allow real-time access to the network, therefore spontaneous messaging is used in conjunction with the time triggered messaging to ensure this.

Presently there are two types of schedulers available for the development of TTCAN message sets. They are the stochastic and heuristic scheduler, which are both useful, but they do not provide the capability of ensuring real-time messages arrive within their deadline.

Stochastic schedulers generate message sets by a probability distribution and heuristic schedulers develop a message set solution by trial and error. They both define the optimum message set as the one with the least jitter by use of a cost function analysis.

Neither of the two methods take into account the effect the schedules may have on spontaneous real-time messaging.

Real-time messages have the best opportunity of meeting their deadline, if the TTCAN messages are not sent sequentially, in fact the larger the arbitration window size between TTCAN messages, the more successful will be the real-time performance of the network.

Schedulers are essentially designed to load balance system resources and in the case of a TTCAN network, the ideal situation is that all messages are separated by the same size arbitration window. This provides the optimum real-time performance, however with messages of different time periods being broadcast, it should be realised that arbitration windows will nearly always be of different sizes. A statistical message scheduler has been devised and demonstrated to produced an optimum message set for real-time operation on a TTCAN network and hence improve the results produced by stochastic or heuristic techniques.

Table of Contents

Figures List

List of Tables

List of Abbreviations

ABS	Anti-lock Braking System
ACC	Adaptive Cruise Control
ACK	Acknowledge
ANSI	American National Standards Institute
BRP	Baud Rate Prescaler
CAN	Controller Area Network
CAN_H	CAN High
CAN_L	CAN Low
CiA	CAN in Automation
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
csv	comma-separated values
DLC	Data Length Code
DoR	Distinctness of Reaction
ESP	Electronic Stability Control
f	Frequency
GUI	Graphical User Interface
HIL	Hardware-in-the-Loop
ID	Identifier
IPT	Information Processing Time
ISO	International Standards Organisation
kbits/s	Kilobits per second

LAN	Local Area Network
LCM	Lowest Common Multiple
LED	Light Emitting Diode
MCU	Microcontroller
MDI	Medium Dependant Interface
ms	Millisecond
NBR	Nominal Bit Rate
NBT	Nominal Bit Time
ns/m	nanosecond/metre
NTU	Network Time Unit
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PMA	Physical Medium Attachment
PS	Physical Signalling
PSB1	Phase Segment Buffer 1
PSB2	Phase Segment Buffer 2
RAM	Random Access Memory
RJW	Re-Synchronisation Jump Width
SAE	Society of Automotive Engineers
SJW	Synchronization Jump Width
SM	System Matrix
SOF	Start of Frame
TCS	Traction Control System
TFT	Thin-Film Transistor

TQ	Time Quantum
TT	Time Triggered
TTCAN	Time Triggered Controller Area Network
TUR	Time Unit Ratio

Chapter 1: Introduction

1.1 Introduction

This chapter gives a brief outline of the complete research. It starts with a historical look at how automotive networks evolved within the automobile industry. It states how the material uncovered during the study will be presented and describes the sequence that will be followed in the design and testing of the solution.

1.2 History

The past four decades have witnessed an exponential increase in the number and sophistication of electronic systems in vehicles. In developed countries, the average cost of electronic devices per vehicle accounts for 20-25% of the total, or even as high as 50% for limousines[1]. Analysts estimate that more than 80 percent of all automotive innovation now stems from electronics. To gain an appreciation of the change in the average Euro amount of electronic systems and silicon components such as transistors, microprocessors, and diodes in motor vehicles, we need only note that in 1977 the average amount was €90, while in 2001 it had increased to €1500. Meanwhile China's total automotive output sales value for automotive electronic products in 2005 reached €6.1 billion[1].

The growth of electronic systems has had implications for vehicle engineering. For example, high-end vehicles in 1995 may have had more than 4 kilometers of wiring compared to 45 meters in vehicles manufactured in 1955. In the past, wiring was the standard means of connecting one element to another. As electronic content increased, however, the use of more and more discrete wiring hit a technological wall.

Added wiring increases vehicle weight, weakens performance, and makes adhering to reliability standards difficult. For an average vehicle, every extra 50 kilograms of wiring increases fuel consumption by 0.2 liters for each 100 kilometers travelled. Also, complex wiring harnesses take up large amounts of vehicle volume, limiting expanded functionality. Eventually, the wiring harness had become the single most expensive and complicated component in vehicle electrical systems [2].

Today's control and communications networks are based on serial protocols that counter the problems of large amounts of discrete wiring. For example, in a 1998 press release, Motorola reported that by replacing the wiring harnesses with a LAN in the four doors of a BMW, which is just one sub-system, it had reduced the weight by

15 kilograms while enhancing functionality[3]. Figure 1.1 shows the sheer number of systems and applications contained in a modern automobile's network architecture.

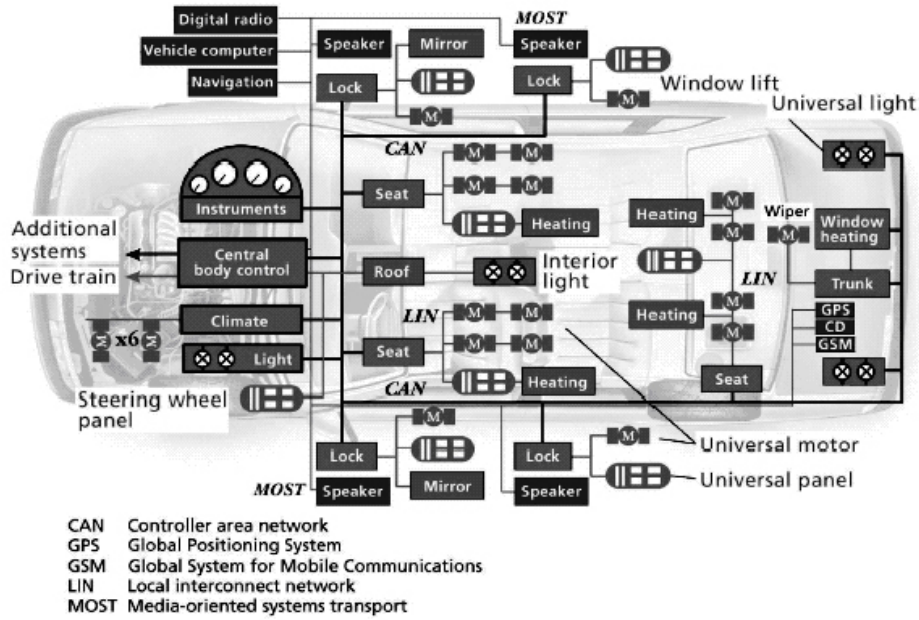


Figure 1.1 : Network Communication System

Controller Area Network is widely used as a communications network in automotive applications, typically motor cars, commercial vehicles, and utility vehicles. CAN is also used in trains, medical equipment, building automation, household appliances and office automation [4].

CAN is operated either by spontaneous messaging or by time triggered messaging. Time triggered messaging is the preferred option on modern systems as it allows all messages access to the bus at some defined period in time. Using time triggered messaging alone does not allow real-time access to the network, therefore spontaneous messaging is used in conjunction with the time triggered messaging to ensure this.

This research investigates the scheduling algorithms presently used with TTCAN and will reveal the flaws with regard to real-time messaging. It will then describe a technique that ensures all TTCAN messages are broadcast, but also allows sufficient bandwidth for real-time messaging.

1.3 Thesis Contributions

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

(i) Literature review, which includes CAN, TTCAN and message scheduling.

(ii) Designing the Optimum TTCAN Message Scheduler.

(iii) Implementation and Testing.

Chapter Two gives an overview of the most relevant information from all literature reviewed during research for this thesis. It outlines the operation of the CAN protocol for both spontaneous messaging and time triggered messaging. It takes an in depth look at scheduling algorithms used with networks and in particular the TTCAN network.

Chapter Three provides an overview of the methods presently used for scheduling TTCAN messages and uncovers the disadvantages they possess. The chapter shows the design process of a new type of message scheduler which negates the problems of the present schedulers. It also discusses how the designed scheduler is implemented in software.

Chapter Four focuses on the design, construction and testing of a four node TTCAN network. It explains how three different message sets, the schedules of which were developed and documented in Chapter Three were tested and how the results from the tests were analysed.

Chapter Five outlines the conclusions made by the author based on the research and testing. A discussion on the further possibilities of research, based on the findings from this study, are also provided.

Chapter 2: Literature Review and Technical Background

2.1 Introduction

This chapter outlines the areas of review relevant to the research/study from all literature assessed. It summarises the possible choices available for CAN messaging strategies and optimisation.

The literature review chapter is presented as follows:

- Section 2.2 discusses the OSI Data Link Layer as defined by the CAN specification. It looks at the MAC and LLC within this layer, both of which manage data encapsulation/de-capsulation, error detection and control, bit stuffing/destuffing, serialisation of data, overload notification, and recovery management. It also considers part of the Physical Layer, namely the Physical Signalling, which includes bit encoding/decoding together with bit timing and synchronisation. All of these functions are carried out by the CAN Controller.
- Section 2.3 looks at another part of the Physical Layer, the Physical Medium Attachment, which is not part of the CAN Specification, but is defined by ISO-11898.
- Section 2.4 provides an insight into the functionality of the CAN controller and CAN transceiver.
- Section 2.5 details why message scheduling is required, and investigates some of the message scheduling strategies that are available at present.
- Section 2.6 looks at message scheduling and its relevance to the automotive industry.

2.2 CAN Data Link Layer

Robert Bosch GmbH began to create a robust asynchronous serial communication system for automotive applications which they called CAN[5] in 1983. The CAN protocol was first officially released in 1986, with Intel and Philips releasing the first CAN controllers and CAN transceivers in 1987. CAN was designed so that cars trucks and buses would be more reliable, safe, and fuel-efficient while at the same time reducing wiring harness weight and complexity. The CAN protocol has gained widespread popularity in industrial automation, medical equipment and mobile machines[4].

2.2.1 Introduction

With the widespread use of electronic open and closed loop control systems fitted to cars such as:

- Electronic engine management
- Electronic transmission shift control
- Anti-lock braking system(ABS)
- Traction control system (TCS)
- Electronic stability programme (ESP)
- Adaptive cruise control (ACC)

and the consequent sharing of information between these systems, it became essential to interconnect all ECUs by means of a network or networks. The conventional point-to-point exchange of data through individual wires has reached its practical limits in the size of the wiring harness and all the associated plugs and sockets (Figure 2.1)[5]. There is also a limit to the number of pins that can be fitted to an ECU and this has slowed the development of ECUs and their software[6].

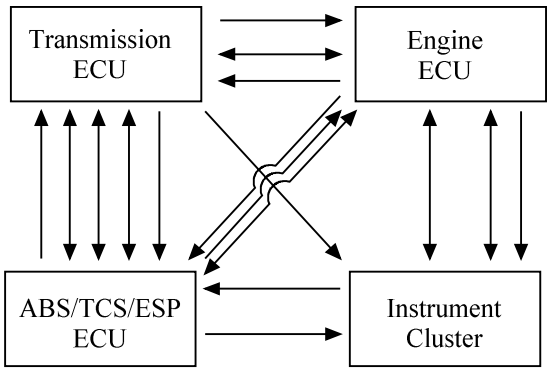


Figure 2.1: Conventional Wiring of ECUs

If we use the method of data transfer as shown in Figure 2.1 in the car, the wiring harness would be made up of approximately one mile of wiring for a medium-size car, plus approximately 300 plugs and sockets and approximately 2000 ECU plug pins [4, 7].

CAN has a linear network topology (Figure 2.2) and was specifically designed for automotive applications, but it is used by several other industries including the medical and buildings installation industry [7].

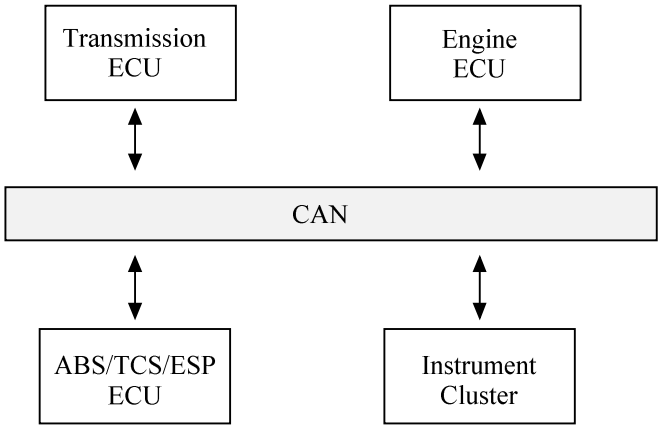


Figure 2.2: Linear Bus Topology

Data is sent in serial format and all CAN nodes (ECUs) have access to the network and can transmit and receive data from the network. This is a multi-master system where the transmitter is the master and all other nodes are slaves. Once the transmitter has control of the network all other nodes become slaves [6]. Since all ECUs can be attached too a single network, this results in far fewer wires being required in the wiring harness.

2.2.2 Vehicle Applications

There are four areas of application for CAN, each of which has a different requirement [7]:

- Multiplex applications
- Mobile communications applications
- Diagnostic applications
- Real-time applications

2.2.2.1 Multiplex Applications

Multiplex applications include the open and closed loop control of components in the body electronics, comfort, and convenience systems. These include such items as climate control, central locking, and seat adjustments. Transfer rates for the data are typically between 10 kbaud and 125 kbaud (low speed CAN) [4].

2.2.2.2 Mobile Communications Applications

CAN is used for such components as navigation systems, telephone and audio installations with the vehicle's central display normally the instrument cluster or a TFT screen centrally fitted in the vehicle. With these applications large quantities of data are required and transfer rates are in the order of 100 and 250 kbaud [4].

2.2.2.3 Diagnostic Applications

Using the CAN network, it is possible to integrate several ECU's on the one network. Presently there are several protocols used for diagnostics. They include ISO 9141-2, J1850 VPWM, J1850 PWM, and ISO 14230-4 which are now becoming invalid. Large quantities of data are also transferred in diagnostic applications and data transfer rates of 250 kbaud and 500 kbaud are being used presently [4].

2.2.2.4 Real-time Applications

Real-time applications include the open and closed loop control of the vehicle's movements. ECUs, such as engine management, transmission control, and electronic stability programme, are networked together to exchange real-time information. Data transfer rates are normally between 125 kbaud and 1Mbaud (high-speed CAN). These bus speeds are required to give real-time response to all situations [4].

2.2.3 Network Configuration

CAN uses a linear bus topology as shown in Figure 2.2. In comparison to other logical structures, for example the ring bus or star bus, it features a lower bus failure probability (Figure 2.3). If one node fails, the bus remains fully accessible to all the other nodes. These nodes can be ECUs, display devices, sensors or actuators, all of which have equal priority regarding access to the bus [7].

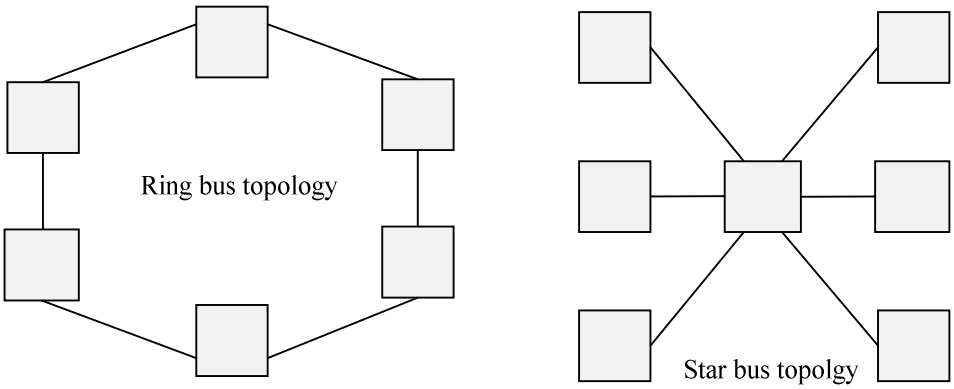


Figure 2.3 : Ring Bus and Star Bus Topology

2.2.4 OSI Model

Almost all network applications follow a layered approach, which allows the interconnection of different devices from different manufacturers. A standard created by the ISO to allow manufacturers to follow this layered approach is called the ISO OSI network layering reference model [8].

CAN is standardised by the ISO and SAE but it only implements the lower two layers of the ISO reference model (Figure 2.4).

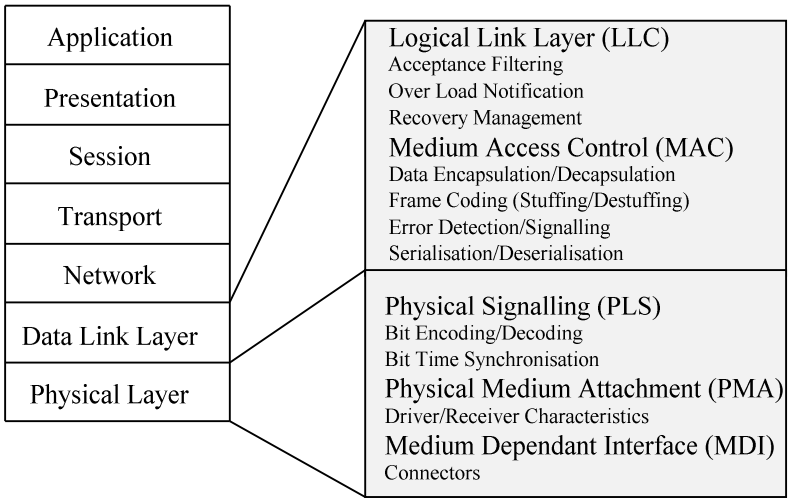


Figure 2.4: Two Lower Layers Implemented from ISO Model

Almost all of these two layers are contained within the CAN controller, such as Microchip's MCP2515. The two components that are not contained within the CAN controller are the PMA which is implemented within the CAN transceiver

(Microchip's MCP2551) and MDI which are the external connectors and wires [9]. The communication medium, which is the upper five layers, was left out of the Bosch CAN specification to allow designers to adapt the communication protocol on multiple media for maximum flexibility i.e. twisted pair, single wire, optically isolated, etc. [5].

The ISO and the SAE have defined protocols based on CAN that include the Media Dependent Interface definition such that all of the lower two layers are specified. ISO 11898 is a standard for high-speed CAN applications; ISO 11519 is the standard for low speed CAN applications. The J1939 protocol is used for truck and bus applications. All the above protocols are specified at a 5V differential electrical bus as the physical interface [10]. The system software designer implements the five other layers of the ISO/OSI protocol stack.

2.2.5 Content-based addressing

The CAN bus system does not address nodes directly but rather according to the message contents. It gives each message a fixed identifier, that identifies the contents of the message in question (could be engine rpm). This identifier can be either 11 bits long (standard format) or 29 bits long (extended format) [6].

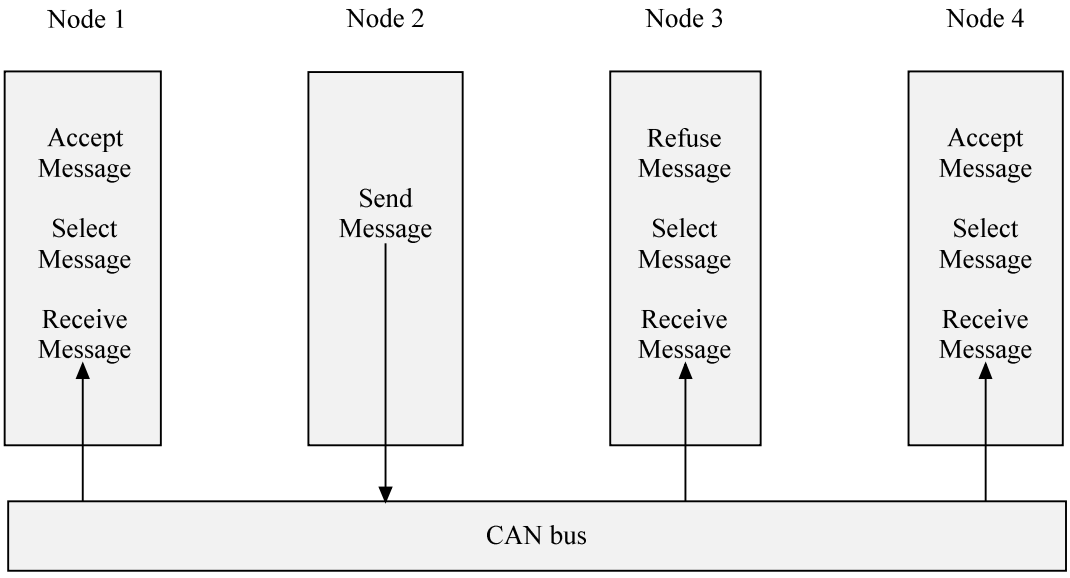


Figure 2.5: Addressing & Message Filtering

With content based addressing each node will have to decide if it is interested in the message or not. If an ECU requires new data which is already on the bus, all it needs to do is extract the message from the bus, see Figure 2.5 [7].

If the node is not interested in the message, it is filtered out by hardware (Full CAN), and therefore saves processing time for the ECU's microprocessor. However, if using Basic CAN the processor must read all messages. Using content based addressing, as opposed to allocating node addresses, allows for greater flexibility in that new equipment is easier to install and operate.

2.2.6 Bus Arbitration:

The identifier used in CAN not only identifies the data content but also defines the message priority. If the identifier is a low number, then it has a high priority in the system. Message priorities are used to gain access to the bus rapidly, but there cannot be two messages allocated the same identifier on the same network [11]. Every node can attempt to send a message as soon as the bus is unoccupied. The message that gains access to the bus is determined by applying a bit by bit identifier arbitration, where the message with the highest priority (lowest identifier) has access to the bus first, and without loss of data.

The CAN protocol is based on two states, the dominant state "logic zero" and the recessive state "logic one". Bus access is handled via the advanced serial communications protocol called Carrier Sense Multiple Access/Collision Detection with Non-Destructive Arbitration. This arbitration concept avoids collisions of messages where transmission of messages are started by more than one node simultaneously and ensures the most important message is sent first without any time loss [5].

The arbitration system used allows the dominant bits transmitted by a node to overwrite the recessive bits written by any node. It can be seen using the example in Figure 2.6 that all four nodes are at the start of transmitting. The bus initially departs from the recessive state and switches to a dominant state of logic zero. All four nodes send a recessive bit logic one next, followed by nodes 2, 3 and 4 sending a dominant bit and node 1 sending a recessive bit logic one. Node 1 has now lost access to the bus. Nodes 2 and 4 send a dominant bit next, while node 3 sends a recessive bit, and therefore, node 3 has lost arbitration on the bus. Nodes 2 and 4 now send recessive bits each and neither loses arbitration, but then node 2 sends a dominant bit and node

4 sends a recessive bit, and therefore, node 4 loses access to the bus. Node 2 now continues to send the rest of its message (Figure 2.6).

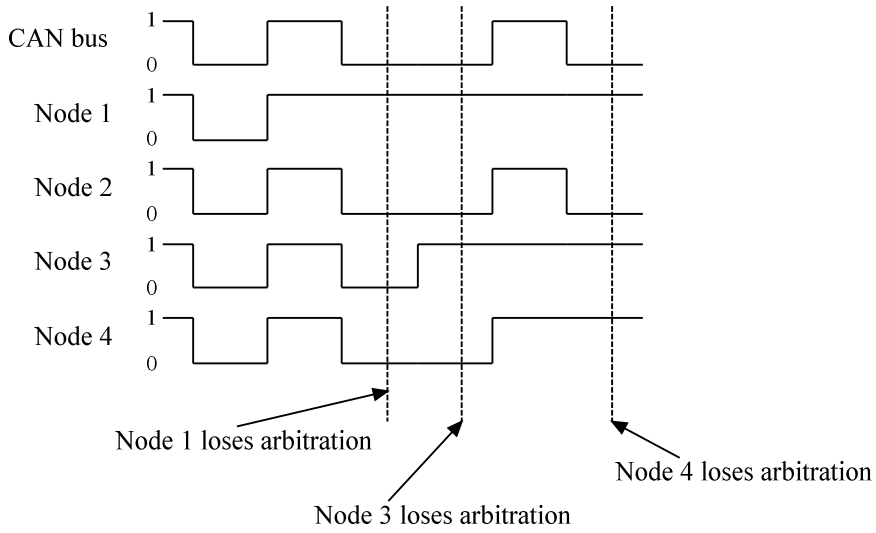


Figure 2.6: CAN Bus Bit Arbitration

The transmitting nodes with lower priority messages now automatically become receivers and then attempt to retransmit their messages when the bus becomes vacant again.

2.2.7 CAN Bus "Wired – AND"

As stated earlier, CAN uses two logic states called "dominant", which is logic zero, and "recessive", logic one. Once a dominant state is issued by any one node regardless of the state issued by any other node, the bus state will be dominant [5]. This is shown in the truth table (Table 2.1).

Node 1	Node 2	Node 3	Bus
D	D	D	D
D	D	R	D
D	R	D	D
D	R	R	D
R	D	D	D
R	D	R	D
R	R	D	D
R	R	R	R

Table 2.1: Truth Table for "Wired AND"

The physical Wired-AND hardware is shown in Figure 2.7, where all nodes are transmitting recessively.

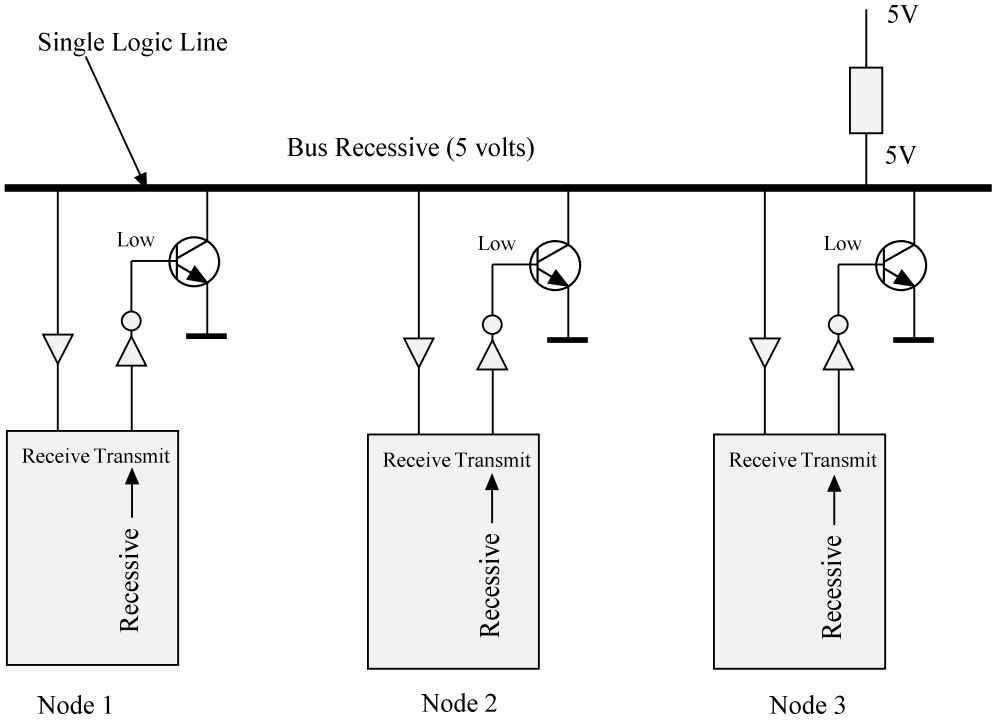


Figure 2.7: Wired-AND (recessive state)

The recessive signal entering the inverter is a logic one, 5 volts, but the inverter outputs a logic zero 0 volts. The transistor is non-conducting due to no base current; therefore, the bus remains in a recessive state.

In Figure 2.8 node 1 is transmitting a dominant bit, which is logic zero, 0 volts, but the inverter will place logic 1 to the base of the transistor and make it conductive. The effect on the circuit is that the supply of 5 volts to the resistor attached to the single logic line remains at 5 volts, but the side of the resistor nearest the bus drops to 0 volts. This is due to the bus being grounded through the transistor attached to node 1. The bus is now in a dominant state. If more than one node transmits a dominant bit, the bus will always be dominant [5].

In both Figure 2.7 and Figure 2.8, the bus consists of a single logic line of 5 volts. This is not the normal bus configuration for CAN, as high speed CAN requires two logic lines CAN_High and CAN_Low [11]. The actual connections to the bus are discussed later in this chapter.

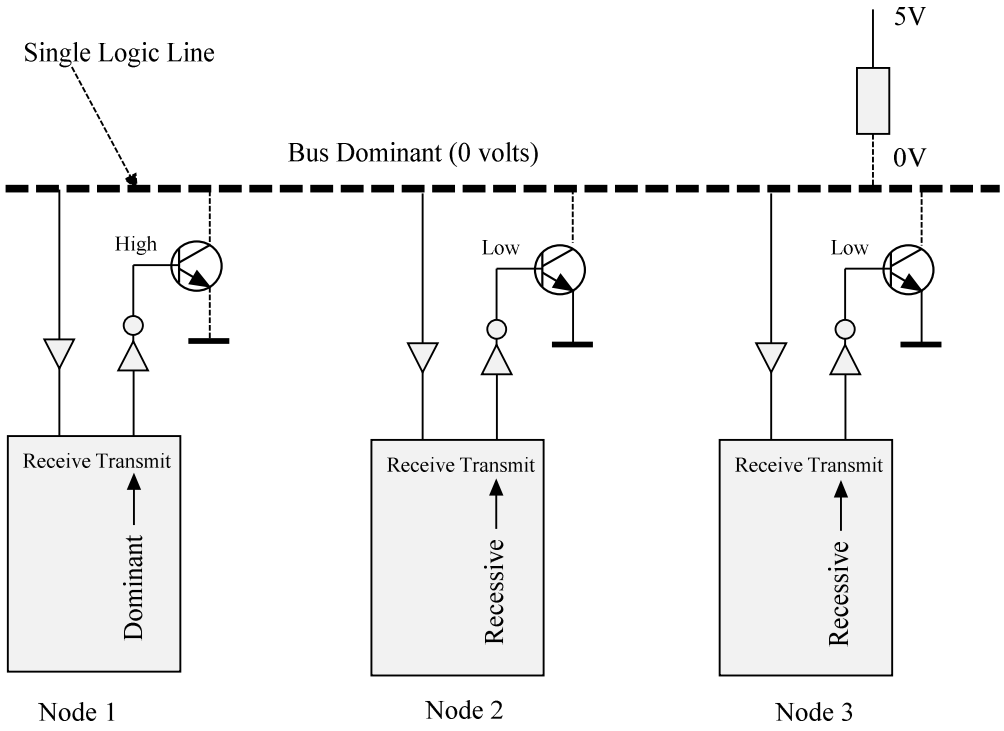


Figure 2.8: Wired-AND (dominant state)

2.2.8 CAN Frames

CAN has five different types of frames:

- Data Frame
- Remote Frame
- Error Frame
- Overload Frame
- Inter-frame space

2.2.8.1 Data Frame or Message Frame

CAN has two different message formats for the identifier, the standard-format identifier, which is 11 bits long, and the extended-format identifier of 29 bits. Both formats will operate on the same CAN network providing it meets CAN Specification 2.0 [11]. A Data Frame consists of seven different fields and may be up to 127 bits long for the standard-format, as seen in Figure 2.9 and 154 bits for the extended-format.

The bus is always in a recessive state when idle logic one, with a dominant bit signifying the Start of Frame. This indicates the beginning of the message and it will synchronise all nodes connected to the network.

The Arbitration Field follows the Start of Frame bit. It is often called the ID, or identifier, and has an additional control bit within it. While the ID is being transmitted the transmitter will check to ensure that it is still authorised to transmit the message, or if another node with a higher priority message has control of the bus. The control bit following the identifier is the RTR bit (Remote Transmission Request) and identifies whether the message is a Data Frame for a receiving node or a Remote Frame (request for some data) from a transmitting node.

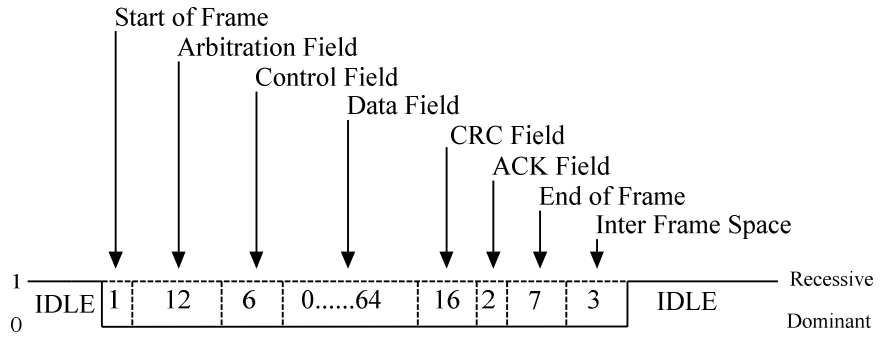


Figure 2.9: CAN Data Frame Standard Format

The *Control Field* has the IDE bit (Identifier Extension Bit), which is used to determine whether the message is of standard format (IDE = 0) or of the extended format (IDE = 1), followed by another bit which is reserved for future use. The last four bits in this field determine the number of data bytes in the data field. This allows the receiving nodes to determine if all data was received.

The *Data Field* contains the actual information contained within the message frame and can consist of between zero and eight data bytes of information.

The *CRC Field* (Cyclic Redundancy Check) contains the frame check word that is used for error checking.

The *ACK Field* is the acknowledgement field of the message and is used by the receiving nodes to acknowledge receipt of the message in a non-corrupted form.

The *End of Frame* marks the end of the message and comprises of seven recessive bits.

The *Inter-frame Space* has three bits, which are used to separate successive messages on the bus. This will allow the bus to remain in an idle mode until a node starts another transmission.

Generally a node starts the data transmission by sending a Data Frame, but it is also possible to request data by sending a Remote Fame asking for data to be supplied (Figure 2.10) [7].

2.2.8.2 Remote Frame

Normally data transmission is performed on an autonomous basis with the data source node (example a sensor) sending the Data Frame, and any another node that requires the data accepting this data through their filtering system. However, it is possible to request data from a source node by using a Remote Frame.

There are two differences between a standard Data Frame and a Remote Frame (Figure 2.10). Firstly the RTR-bit is transmitted as a dominant bit in the Data Frame, whereas it is transmitted as a recessive bit in the Remote Frame. Also there is no Data field in the Remote Frame.

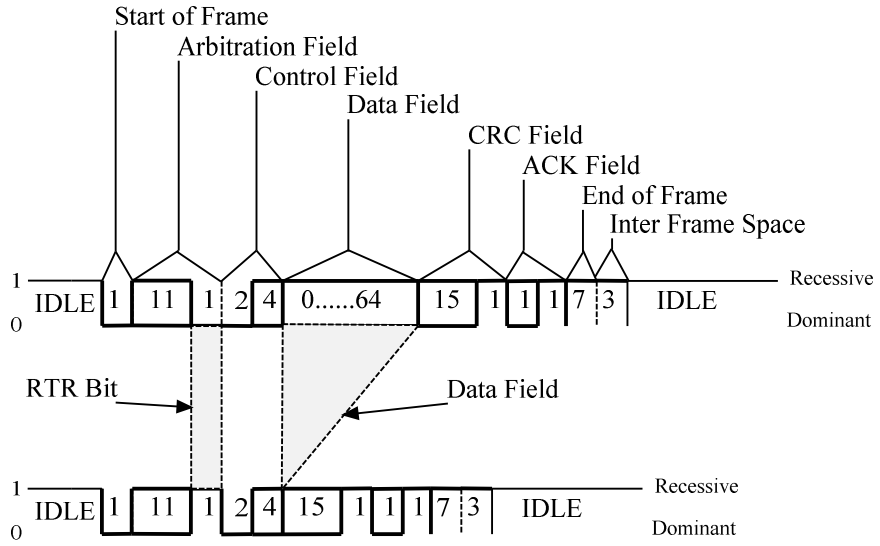


Figure 2.10: Remote Frame

In the improbable case of a Data Frame and a Remote Frame with the same identifier being transmitted simultaneously, the Data Frame will be transmitted due to the dominant RTR bit following the identifier. In this way, the node that transmitted the Remote Frame receives the desired data immediately.

2.2.8.3 Error Frame

An Error Frame will be generated by any node that detects a bus error. The Error Frame has two fields as seen in Figure 2.11, the Error Flag Field and Error Delimiter Field.

There are two forms of Error Flag fields. The type of Error Flag field depends on the "error status" of the node that detects the actual error. If an "error-active" node detects a bus error then that node will interrupt transmission of the current message by generating an "active error flag".

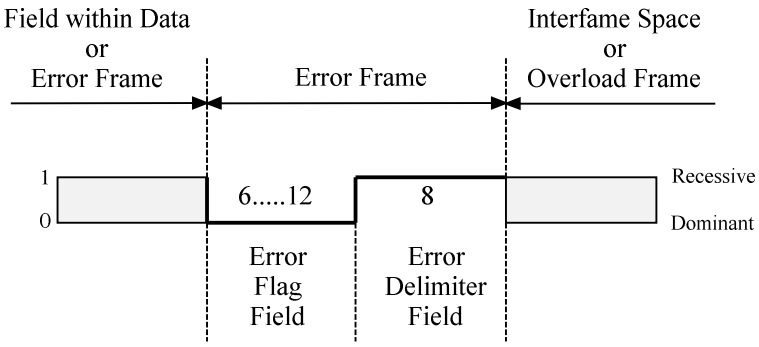


Figure 2.11: Error Frame

The "active error flag" is composed of six consecutive dominant bits. This violates the bit-stuffing rule, which is discussed later. All other nodes recognise the bit stuffing error, and in turn, generate Error Frames themselves. The Error Flag field will be between six and twelve dominant bits. The Error Delimiter consists of eight recessive bits. This permits all nodes to restart bus communications cleanly after such an error. After completion of the Error Frame the bus returns to normal and the node that caused the bus error attempts to retransmit the message.

If an "error passive" node detects a bus error then it will transmit a "passive Error Flag", followed again by the Error Delimiter field. The "passive Error Flag" consists of six consecutive recessive bits, and therefore, the Error Frame for an "error passive" node consists of fourteen recessive bits (Passive Error Flag six recessive bits followed by the Error Delimiter Field of eight recessive bits). If the node that is transmitting identifies the bus error, the transmission of an Error Frame by an "error passive" node will not affect any other node on the bus. If the bus master node generates an "error

passive flag" then this may cause other nodes to generate error frames due to the resulting bit stuffing violation.

2.2.8.4 Overload Frame

An Overload Frame has the same format as an "Active Error Frame". However, it can only be produced during Interframe Space not during the transmission of a message as seen in Figure 2.12.

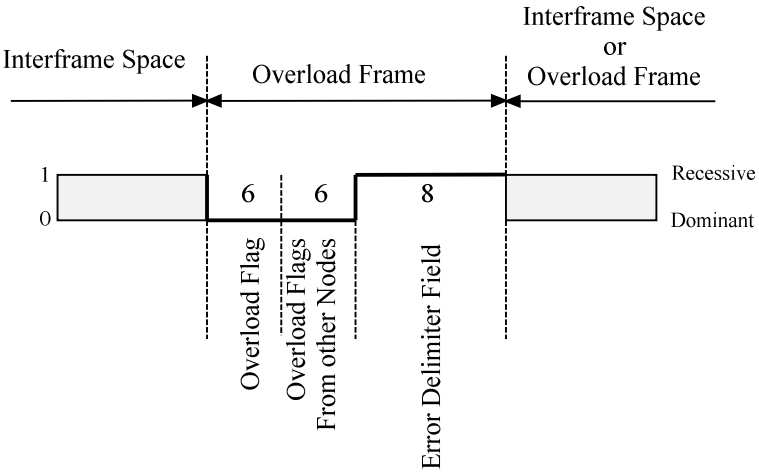


Figure 2.12: Overload Frame

The Overload Flag consists of six dominant bits followed by Overload Flags generated by other nodes as the bit-stuffing rule has been violated. The Overload Delimiter consists of eight recessive bits. The Overload Frame has two fields, an Overload Flag followed by an Overload Delimiter.

A node, if due to internal conditions, can generate an Overload Frame; the node is not yet able to start reception of the next message. A node may only generate a maximum of two sequential Overload Frames to delay the start of the next message.

2.2.8.5 Interframe Space

The Interframe Space separates a preceding frame from the next Data or Remote Frame. An Interframe space is made up of at least three recessive bits; these bits are also known as the "Intermission". This Interframe Space permits nodes to process internal data before the next message frame (Figure 2.13). After the Intermission, for

error active nodes the bus remains in the recessive state until the next transmission starts.

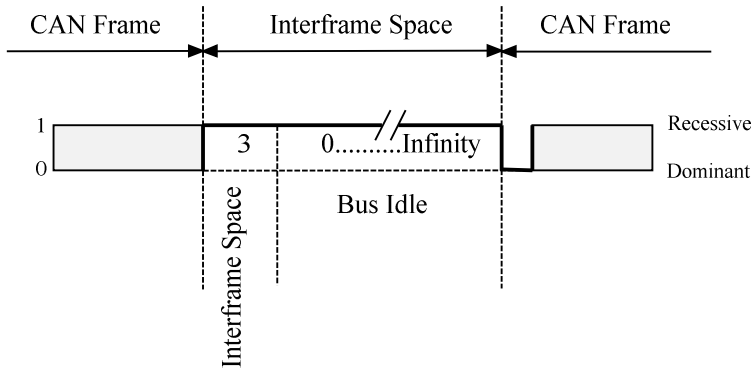


Figure 2.13: Interframe Space

The Interframe Space has a slightly different format for error passive CAN nodes, which was the transmitter of the previous message. These nodes have to wait an additional eight recessive bits, often called "Suspended Transmission" before the bus turns into bus idle for them. After this time, they are allowed to transmit messages again. This arrangement allows error active nodes to broadcast their messages before an error passive node is allowed to start the retransmission of messages.

2.2.9 Error Detection

The CAN protocol has several mechanisms for error detection as listed:

- CRC Error
- ACK Error
- Form Error
- Bit Error
- Stuff Error

2.2.9.1 CRC Error

Using the Cyclic Redundancy Check, the transmitter calculates a checksum for the bit sequence from the start of frame bit to the end of the Data Field. This CRC checksum is then transmitted in the CRC Field of the message [10]. The receiving node calculates the CRC checksum of the message using the same formula and compares it

to the CRC of the received message [11]. Figure 2.14 shows a CRC error occurring in node 2.

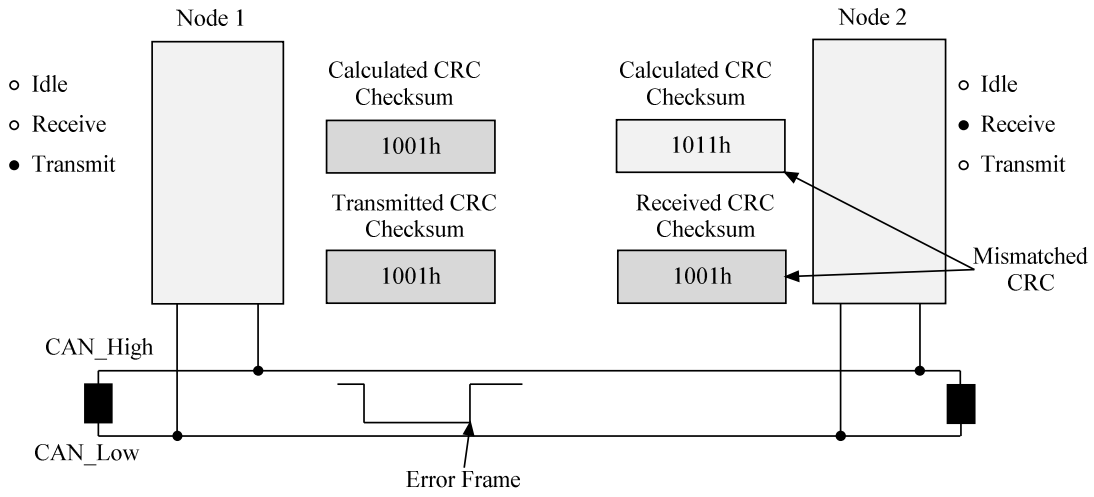


Figure 2.14: CRC Error

2.2.9.2 Acknowledge Error (ACK)

The ACK check in the receiving nodes will confirm that the message frame has been received (Figure 2.15). If the transmitting node does not receive the acknowledgement then the transmitting node will know an error has been detected and will retransmit the message [10, 11].

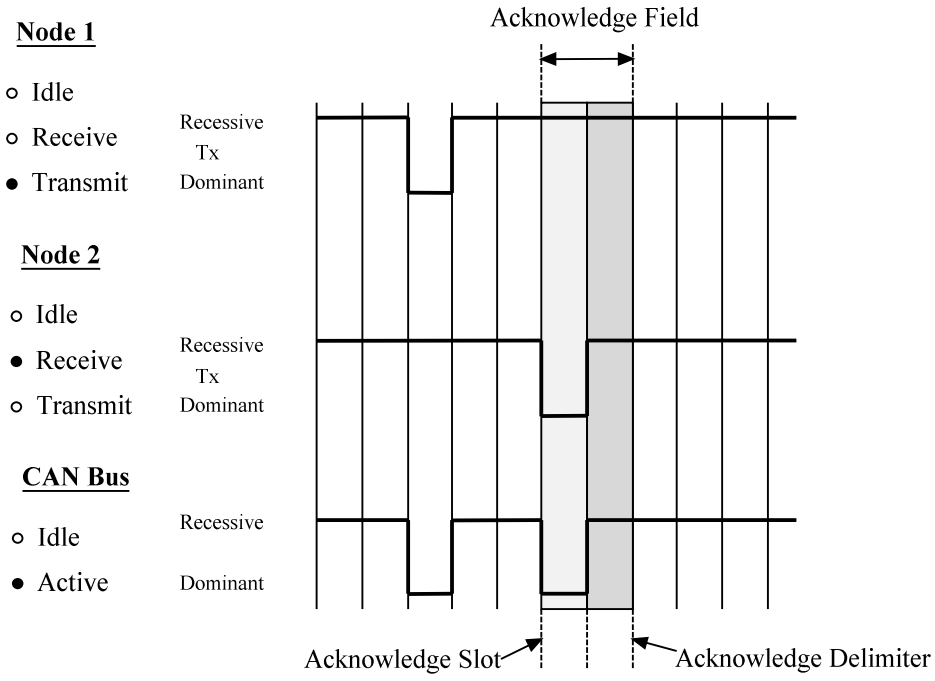


Figure 2.15: Acknowledge Field

2.2.9.3 Frame Check

The Frame Check will check the frame for errors based upon the frame structure. The CAN protocol has a number of fixed format bit fields, which are checked by all nodes.

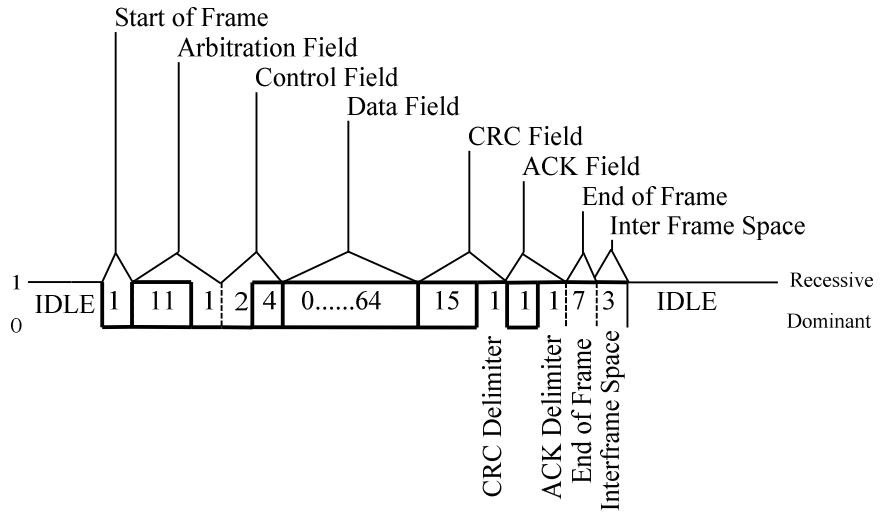


Figure 2.16: Frame Check

If a transmitter detects a dominant bit in one of the following four segments, the CRC Delimiter, the Acknowledge Delimiter, the End of Frame or the Interframe Space [11, 12] then a Form Error has occurred and an Error Frame will be placed on the bus (Figure 2.16) and the original message will be retransmitted.

2.2.9.4 Bit Error

A Bit Error occurs if the transmitter places a dominant bit on the bus but then detects a recessive or sends a recessive bit, but detects a dominant bit. An Error Frame is generated and the message is retransmitted [12].

There are two exceptions to the above rule. When a dominant bit is detected instead of a recessive bit, no error will occur during the Arbitration Field or the Acknowledge Slot. Also these fields may be overwritten by a dominant bit in order to accomplish arbitration and acknowledge functionality [11].

2.2.9.5 Bit-Stuffing Check

The Bit-stuffing rule stipulates that in every Data Frame or Remote Frame a maximum of five successive equal priority bits can be sent between the Start of Frame and the end of the CRC field. As soon as the five identical bits have been transmitted in succession, the transmitter inserts an opposite priority to those already been sent as seen in Figure 2.17. The receiving node checks the message, and ignores the opposite priority bit, after receiving the message [11].

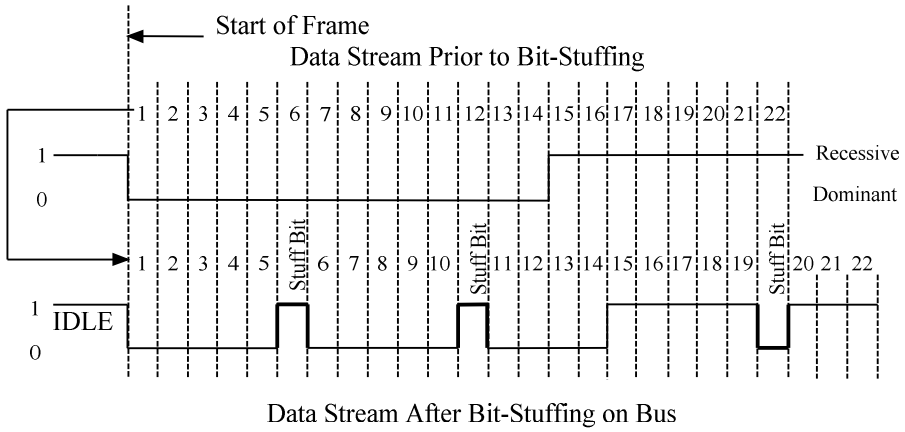


Figure 2.17 Bit-Stuffing

If a Stuff Error occurs, an Error Frame is transmitted, and the message is resent. Code check is a method to check that Bitstuffing has been carried out correctly. If one of the nodes detects an error on the bus, it interrupts the actual transmission by sending an Error Frame comprising of six successive dominant bits [4]. Broadcasting this Error Frame violates the Bitstuffing rule and this prevents all nodes from receiving the message [12].

2.2.9.6 Error Handling

All known errors are made public, to all other nodes on the bus via Error Frames. The transmission of the damaged message is aborted, and the frame is retransmitted as soon as possible. Each node is in one of three error states, either Error Active, Error Passive, or Bus Off, depending on the count in the error counter registers (Figure 2.18).

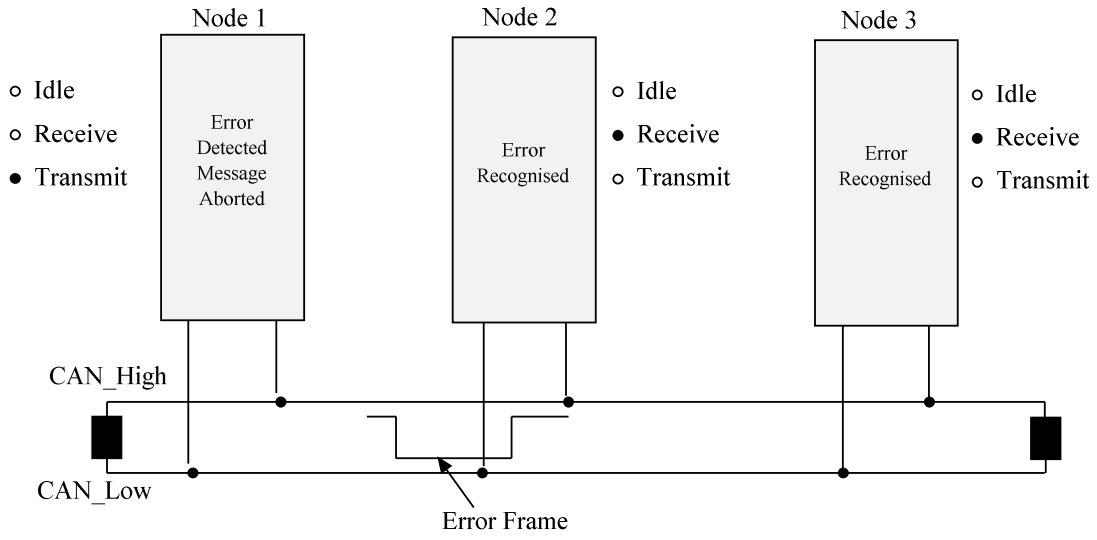


Figure 2.18: Error Frame Transmitted

.

The error-active state is the normal state after a reset for any node. It can actively receive and transmit messages and transmit active Error Frames without any limitations. In normal CAN communication, the error counters are updated according to complex rules [11]. For each error on receipt or transmission of a message, the relevant error counters are incremented. For each successful transaction, the error counters are decremented. The error active state is valid as long as both error counters are less than or equal to 127 [12].

If either receive or transmit error counters exceed a value of 127, the node switches to the error-passive state. In the error-passive state, messages can still be received and transmitted, although, after transmission of a message the node must suspend transmission. It must wait 8-bit times longer than error-active nodes before it may transmit another message. Error Passive nodes can signal other nodes with only passive Error Frames.

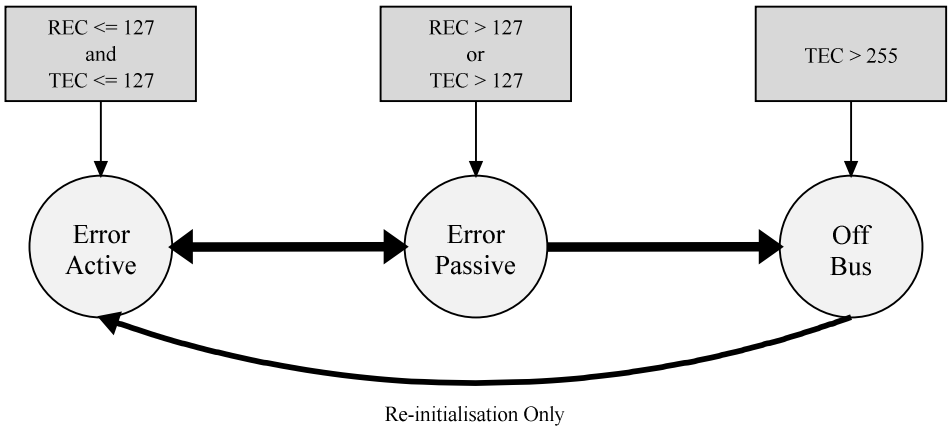


Figure 2.19: Node Error Counters

If both error counters decrement below 128 due to successful message transmission, the node switches back to the error-active state (Figure 2.19).

The CAN protocol allows faulty nodes to remove themselves from the bus automatically. The bus-off error state is entered if the transmit error counter exceeds the value of 255. All bus activities are stopped for that node (both transmit and receive). For the error node to reconnect to the bus the node has to be reinitialised [10].

The error detection capabilities of CAN are such that a vehicle equipped with this network, running for 2000 hours per year, at a bus speed of 500 kbps with 25% bus load should only generate one undetected error every 1000 years [12].

2.2.10 Protocol Versions

CAN specifications versions 1.0, 1.2, and 2.0A define an 11-bit message identifier. They are known as Standard CAN. With an 11-bit identifier, it is only possible to define 2048 different messages.

There is also a further limit to the messages due to lowest 16 priority messages also being reserved.

Version 2.0A (Standard) - 11 bit ID's - 2048 ID's Available

Start of Frame	Identifier 11 bit	Control Field	Data Field 0....8 Bytes	CRC Field	ACK Field	End of Frame

Version 2.0B (Extended) - 29 bit ID's - More then 536 million ID's Available

Start of Frame	Identifier 29 bit	Control Field	Data Field 0....8 Bytes	CRC Field	ACK Field	End of Frame

Figure 2.20; CAN Standard and Extended Data Frames

Specification version 2.0A has been updated to version 2.0B to remove this possible message number limitation and also meet the SAE J1939 standard for the use of CAN in trucks [13]. Version 2.0B is known as Extended CAN due to its 29-bit identifier. With a 29-bit identifier, it can now have over 536 million different message identifiers (Figure 2.20).

The 29-bit identifier consists of the original 11-bit identifier and an 18-bit Extended Identifier. Version 2.0B allows a message identifier length of 11 bits to be used.

There are three different types of CAN modules available. CAN modules specified version 2.0 part “A” [11] are only able to transmit and receive Standard Frames according to the Standard CAN protocol. Messages using the 29-bit identifier sent to a Standard CAN module will cause errors. If a device is specified CAN V2.0 part “B” [11], there is one more distinction (Figure 2.21):

	Frame With 11 bit ID	Frame With 29 bit ID
V2.0B Active CAN Module	TX/RX OK	TX/RX OK
V2.0B Passive CAN Module	TX/RX OK	Tolerated
V2.0A CAN Module	TX/RX OK	**Bus Error**

Figure 2.21: CAN Version Modules

Modules called version 2.0B Passive can only transmit and receive Standard Frames but accept Extended Frames without generating Error Frames. Version 2.0B Active devices are able to transmit and receive both Standard and Extended Frames.

2.2.11 Message Coding

The CAN protocol uses Non-Return-to-Zero or NRZ bit coding. This permits the signal on the network to remain at the same voltage for one-bit time and only one time segment is required to represent the one bit (Figure 2.22). A zero corresponds to a dominant bit, which causes the bus to be placed in a dominant state, and a one corresponds to a recessive bit, placing the bus in the recessive state.

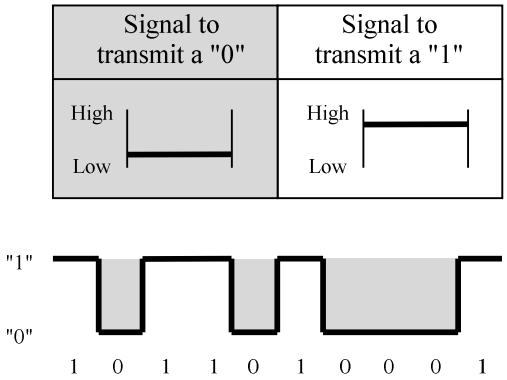


Figure 2.22: Message Coding

One problem of using NRZ code is that the signal provides no edges for use in re-synchronisation when transmitting a large number of consecutive bits with the same priority (Dominant or Recessive bits). To overcome this, bit stuffing is used to guarantee synchronisation of all bus nodes. As discussed earlier, a maximum of five consecutive bits may have the same priority, and then the transmitter will insert one additional bit of the opposite polarity into the bit stream before transmitting further bits. The receiver also checks the number of bits with the same priority and removes the stuff bits again from the bit stream. This technique is called “destuffing”.

2.2.12 Bus Synchronisation

CAN uses two types of synchronisation, Hard Synchronisation and Re-Synchronisation. In contrast to many other field buses, CAN handles message transfers synchronously. All nodes are synchronised at the beginning of each message

with the first falling edge of a frame, which belongs to the Start of Frame bit. This is called Hard Synchronization (Figure 2.23).

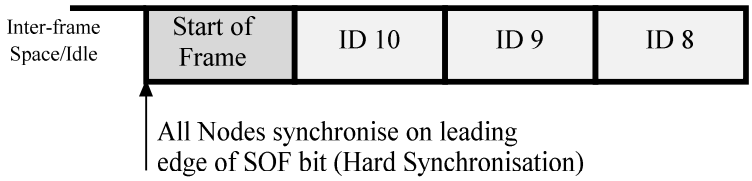


Figure 2.23: Hard Synchronisation

To ensure correct sampling up to the last bit of the CAN Frame, the CAN nodes need to re-synchronise throughout the entire frame. This is achieved on each recessive to dominant edge (Figure 2.24).

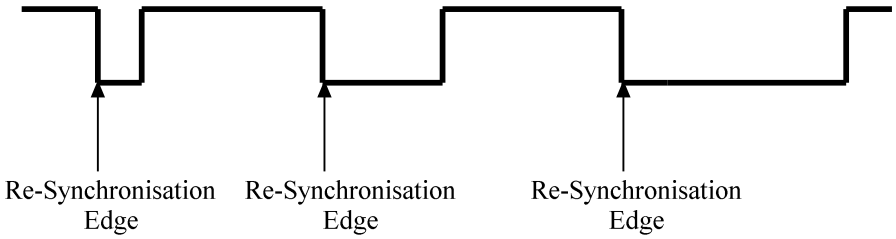


Figure 2.24: Re-synchronisation

2.2.13 Bit Construction

One bit time of either a high or a low pulse of the NRZ code is specified as four non-overlapping time segments (Figure 2.25) [11]. Each segment within the bit time is made up of an integer multiple of the Time Quantum.

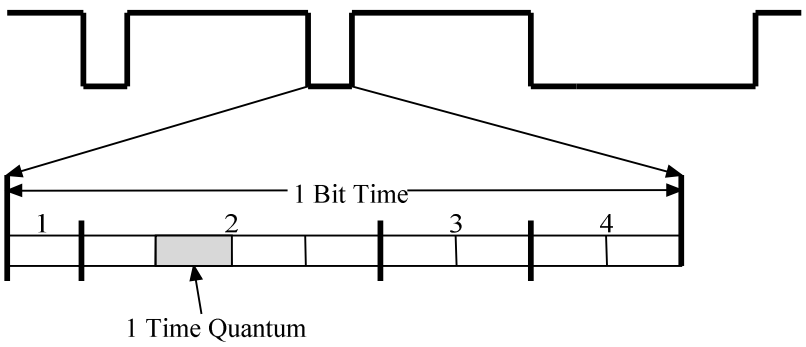


Figure 2.25: Bit Construction

The Time Quantum, or TQ, is the smallest discrete timing resolution used by a CAN node. Its length of the Time Quantum is generated by a programmable division of the CAN node's oscillator frequency. A bit time has a minimum of 8 Time Quanta and a maximum of 25 Time Quanta [14]. The bit time, and therefore the bit rate, is selected by programming through software the width of the Time Quantum and also the number of Time Quanta in the various segments [9]. The CAN baud rate can be determined by dividing 1 by the bit time.

Therefore:

$$NBR = f_{bit} = \frac{1}{t_{bit}} \tag{2.1}$$

2.2.13.1 Baud-rate Prescaler

The length of the TQ, which is the basic time unit of the bit time is defined by the CAN Controller's system clock f_{sys} and the BRP [15].

$$TQ = BRP/f_{sys} \tag{2.2}$$

The relationship between the oscillator and the MCU system clock can be seen in Figure 2.26.

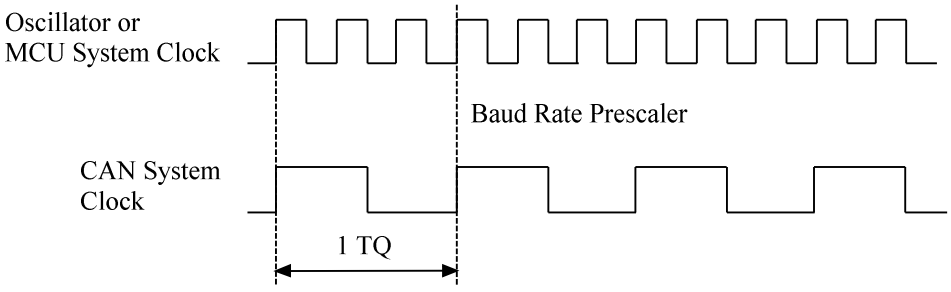


Figure 2.26: Baud Rate Prescaler

2.2.13.2 Synchronisation Segment

The first segment in a CAN bit is called the Synchronisation Segment and is used to synchronise all bus nodes. On transmission, at the start of this segment, the current bit level is output. If a bit state alters between the previous bit and the current bit, the bus state change should happen within this segment [16]. The length of this segment is always one Time Quantum (Figure 2.27).

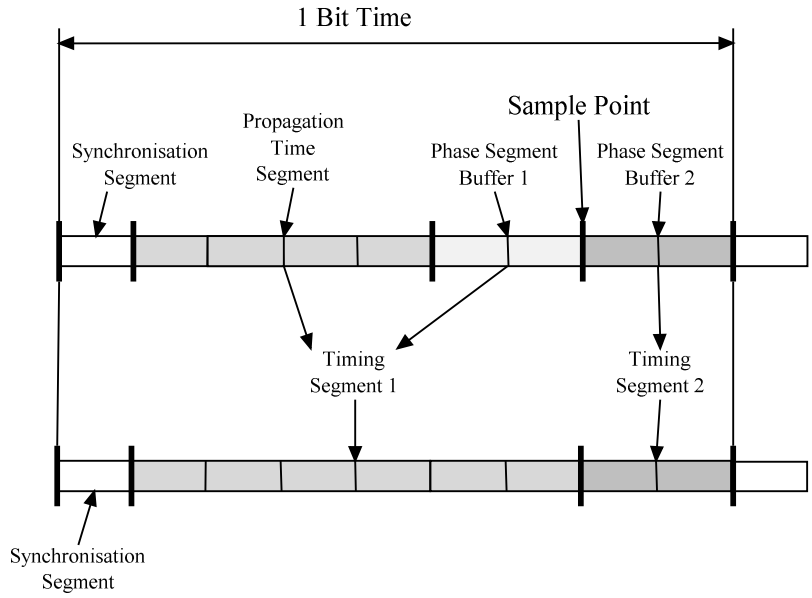


Figure 2.27: The Four Segments of 1 Bit Time

2.2.13.3 Propagation Time Segment

The Propagation Segment is next and is used to compensate for the physical delays in signal propagation between nodes. The propagation delay is defined as twice the sum of the signal's propagation time on the bus line, including the delays associated with the bus driver [16]. The Propagation Segment is programmable with values between 1TQ and 8TQ. Figure 2.27 shows a Propagation Segment of 4TQ.

2.2.13.4 Phase Segment Buffer 1

Phase Segment Buffer 1 is used to compensate for edge phase errors on the bus. Phase Segment Buffer 1 can be lengthened for re-synchronisation and will be discussed later in the chapter. It is programmable from 1TQ to 8TQ. Figure 2.27 shows 2TQ for

Phase Segment Buffer 1 [12, 14]. Some manufacturers describe the Propagation Segment and Phase segment Buffer 1 as Timing Segment 1.

2.2.13.5 Phase Segment Buffer 2

Phase Buffer Segment 2 is also used to compensate for edge phase errors. This segment may be shortened only during resynchronisation. Phase Buffer Segment 2 may be between 2TQ to 8 TQ long, and has to be at least as long as the information processing time, but may not be more than the length of Phase Buffer Segment 1. Figure 2.27 shows Phase Segment Buffer 2 to equal 2 TQ [11, 14]. Phase Segment buffer 2 is sometimes described as Timing Segment 2.
Therefore:

$$PSB2_{min} = IPT = 2TQ \tag{2.3}$$

2.2.14 Information Processing Time

The Information Processing Time is necessary for the logic to determine the bit level of a sampled bit. The IPT begins at the sample point and is measured in TQ. For the Microchip CAN module, it is fixed at 2TQ. Since phase segment 2 also begins at the sample point and is the last segment in the bit time [14, 16], it is a prerequisite that Phase Segment Buffer 2 be a minimum of 2 TQ as shown in Figure 2.27.

2.2.15 Re-Synchronisation

All oscillators do no run exactly at the specified frequency. Therefore, each node being independently operated, using separate oscillators, runs at a slightly different frequency. This could cause problems for CAN message receiving nodes as they could be running at a slightly different frequency to the transmitting node. To overcome this problem the transition from recessive to dominant provides a re-synchronisation edge, as discussed above, but extra TQ will have to be added or removed in order to achieve re-synchronisation.

2.2.15.1 Bit Lengthening

Either lengthening Phase Buffer Segment 1or reducing Phase Buffer Segment 2 by a given TQ carries out the resynchronisation of a Bit Time.

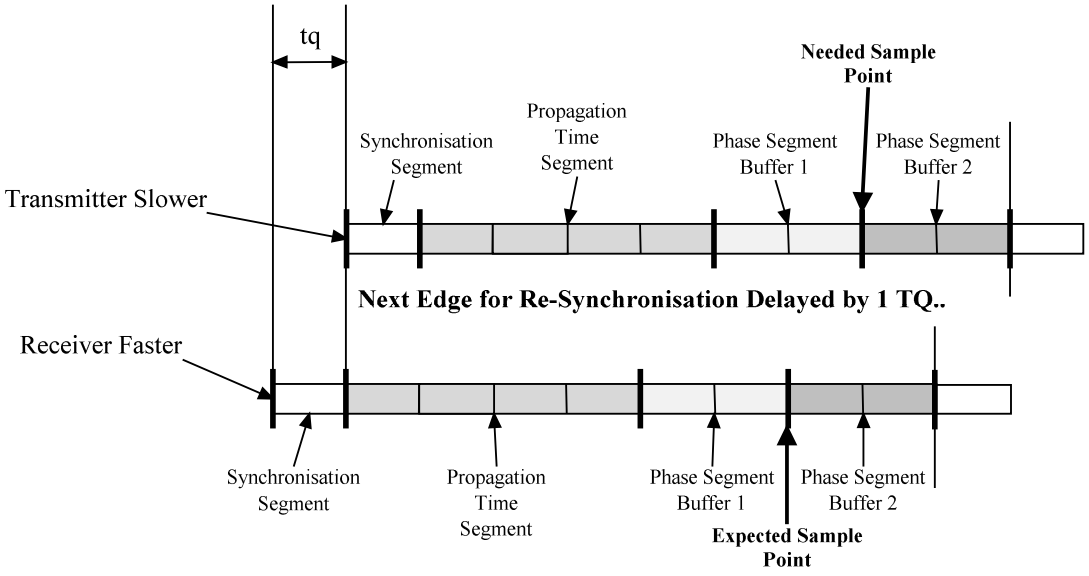


Figure 2.28: Re-Synchronisation Edge Delayed

Figure 2.28 above shows that the transmitter oscillator is slower than the receiver oscillator. The next falling edge used for resynchronization will have to be delayed for the receiving node, so Phase Buffer Segment 1 is lengthened for the receiver in order to align the sample points of the message as depicted in Figure 2.29.

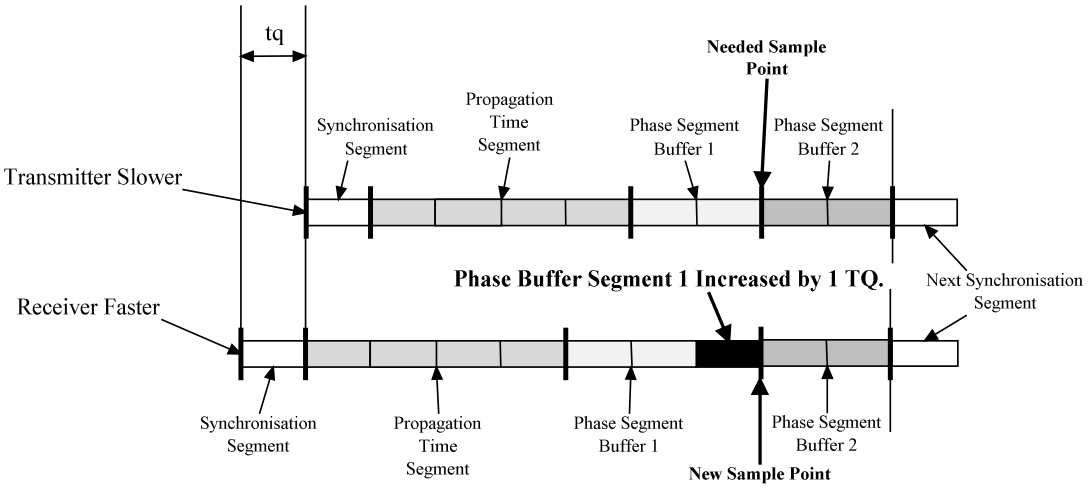


Figure 2.29: Re-Synchronisation by Increasing Phase Segment Buffer 1

2.2.15.2 Bit Shortening

If the transmitter node oscillator is faster than the receiver node oscillator then the next falling edge used for resynchronisation could be too early as shown in Figure 2.30.

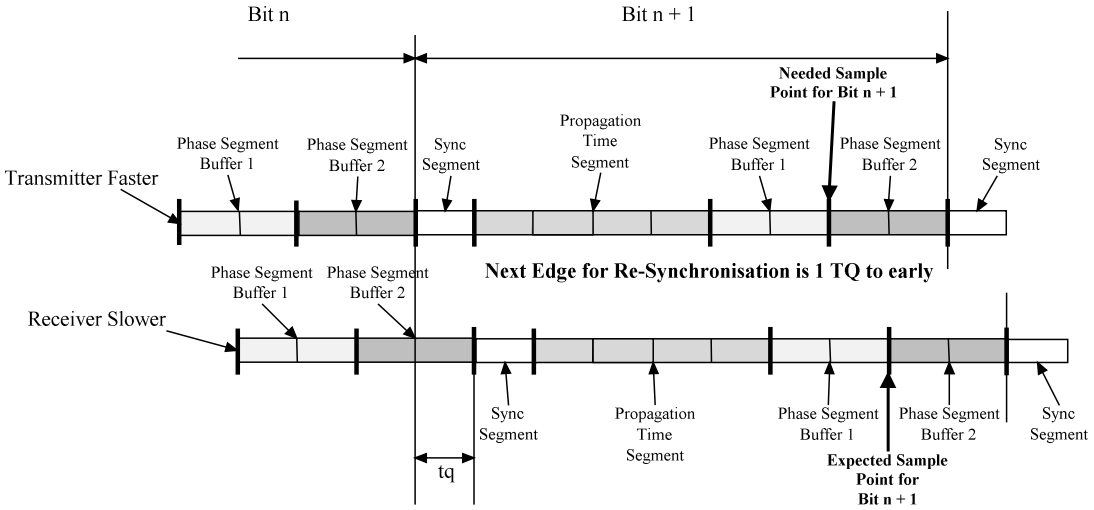


Figure 2.30: Re-Synchronisation Edge Increased

Figure 2.31 shows Phase Buffer Segment 2 in bit N has been shortened so the sample points are realigned.

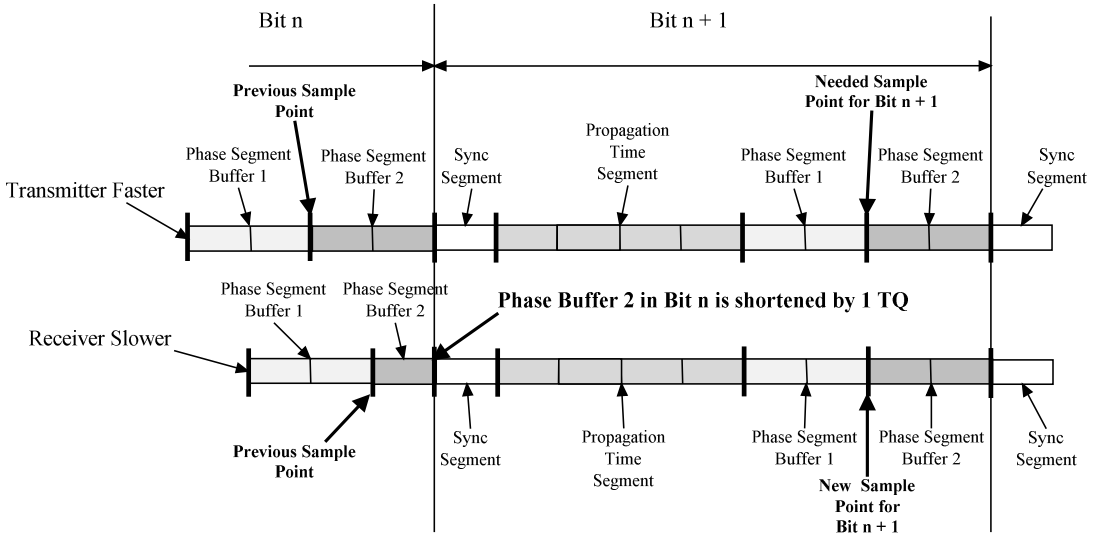


Figure 2.31: Re-Synchronisation by Decreasing Phase Segment Buffer 2

2.2.15.3 Re-Synchronisation Jump Width

The RJW or SJW is the amount by which a bit length can be readjusted during a re-synchronisation. It is the TQ by which Phase Segment Buffer 1 can be lengthened or Phase Segment Buffer 2 can be shortened. The SJW is programmable in software and can have a value of between 1 TQ and 4 TQ, but it may not be longer than Phase Segment Buffer 2 [14].

2.2.16 Bit Timing

For ease of programming many CAN Modules often combine the Propagation Time Segment and Phase Buffer Segment 1, as shown below in Figure 2.32, and is known as Timing Segment 1.

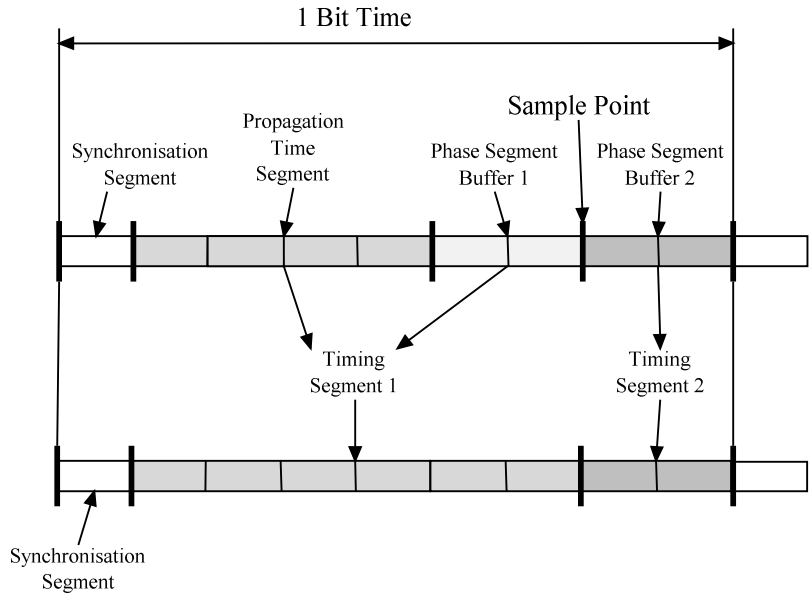


Figure 2.32: Two Timing Segments

2.2.17 Programming the Sample Point

Programming of the sample point allows for some of the bus characteristics to be taken into account. Early sampling allows greater TQ in Phase Segment Buffer 2 so the SJW can be programmed to its maximum of 4 TQ [14]. Using this maximum TQ to shorten or lengthen the bit time decreases the effect of node oscillator tolerances, therefore lower cost oscillators may be used with these nodes (Figure 2.33).

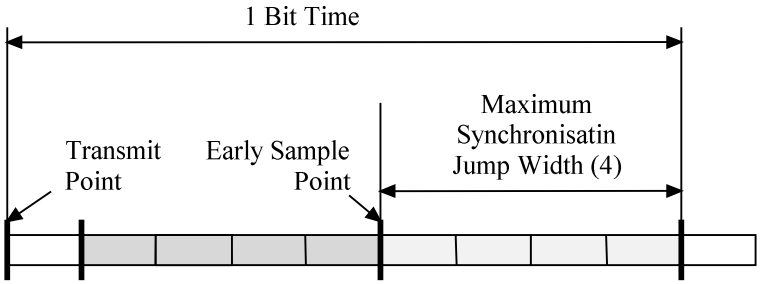


Figure 2.33: Early Sampling Point

Figure 2.34 shows a late sampling point, allowing for the maximum signal propagation time, and therefore, long bus lengths with poor bus topologies can be handled with ease [12].

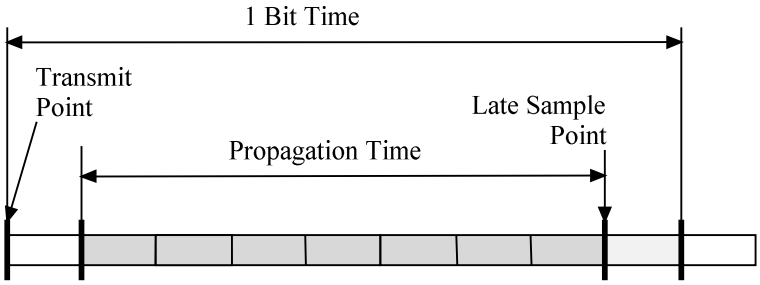


Figure 2.34: Late Sampling Point

2.3 CAN Physical Layer

The Physical Layer as defined under the OSI Model is defined in three parts:

- Physical Signalling
- Physical Medium Attachment
- Medium Dependant Interface

Physical Signalling is implemented within the CAN controller and has been discussed in Section 2.2. The PMA is not part of the CAN Specification, but is defined by ISO-11898. ISO 11898-2 specifies high-speed CAN, with transmission rates of up to 1 Mbit/s, with the PMA and some MDI features defined by ISO 8802-3, which comprises of the physical layer of the controller area network [17].

ISO 11898-3 defines the exchange of digital information between electronic control units using CAN at transmission rates of between 40kbit/s and 125kbit/s [18].

2.3.1 Bus Construction

The CAN bus line has two logic states: a recessive and a dominant state. The ISO-11898 defines a differential voltage to represent both states (Figure 2.35) [9].

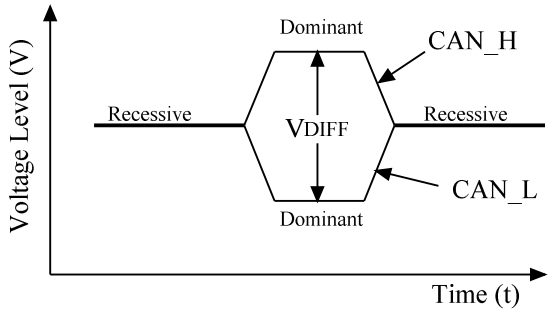


Figure 2.35: The Differential CAN bus

These differential voltages help to reduce electrical interference on the bus, but the actual voltage levels depend on the standard being used and are shown in Table 2.2 and Table 2.3 [19].

Signal	Recessive State			Dominant State		
	Min	Nominal	Max	Min	Nominal	Max
CAN_H	2.0V	2.5V	3.0V	2.75V	3.5V	4.5V
CAN_L	2.0V	2.5V	3.0V	0.5V	1.5V	2.25V

Table 2.2: ISO 11898 (CAN High Speed)

Signal	Recessive State			Dominant State		
	Min	Nominal	Max	Min	Nominal	Max
CAN_H	1.6V	1.75V	1.9V	3.85V	4.0 V	5.0V
CAN_L	3.1V	3.25V	3.4V	0.0V	1.0V	1.15V

Table 2.3: ISO 11519 (CAN Low Speed)

In the case of ISO 11898, the recessive state, nominal voltage for the two wires is always the same voltage at 2.5 volts. This decreases the power consumption of the network when the nodes are not transmitting as seen in Figure 2.36.

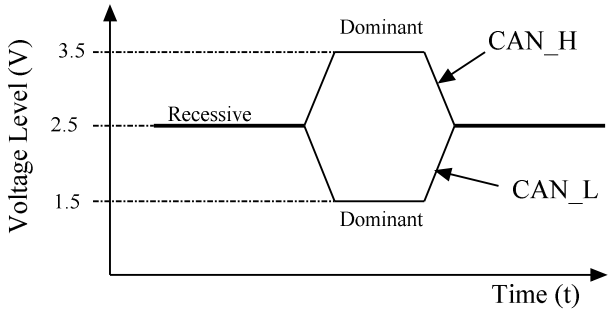


Figure 2.36: ISO 11898 Nominal Bus Voltage Levels

2.3.2 Wires and Connectors

For the CAN bus lines, a physical medium must be chosen that is able to transmit the two possible bit states. One of the most common and cheapest ways is to use a twisted pair of wires. The two bus lines CAN_H and CAN_L are then driven by the transceivers attached to the CAN controllers with a differential voltage signal. These twisted pair of wires, are terminated in accordance with ISO 11898 by 120 ohm resistors at each end of the bus line (Figure 2.18). An optical medium for the CAN bus may also be employed under CAN specification. In this case, the recessive state would be represented by the signal LED being off, the dominant state by the signal LED being switched on.

As discussed earlier the differential nature of the bus makes it virtually insensitive to electromagnetic interference. In order to reduce sensitivity even further the wires are twisted and are often shielded when fitted in a very harsh electrical environment. This also reduces the electromagnetic emission of the bus itself, especially when high baud-rates are being used [12].

2.3.2.1 Bus Lengths

ISO 11898 states that a transceiver must be able to drive a 40m bus at 1 Mbit/s. Bus lines of longer length may be used by decreasing the baud rate, as can be seen in Figure 2.37 [17].

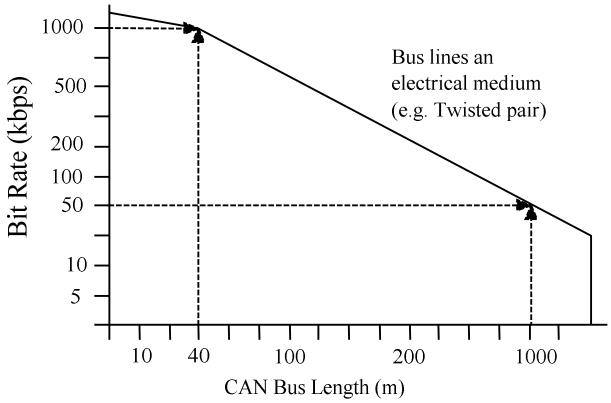


Figure 2.37: "Bus Length" v "Baud-rate"

2.3.2.2 Propagation Delay

The CAN protocol defined a recessive and dominant state for the implementation of a non-destructive bit-wise arbitration scheme. This arbitration methodology is affected most by propagation delays. Each node involved in arbitration has to be able to sample each bit level within the same bit time. For example, if two nodes at opposite ends of the bus network start transmitting their messages at the same time, they must arbitrate for control of the bus. Arbitration will only be effective if both nodes are able to sample the bus during the same bit time. Figure 2.38 shows a possible one-way propagation delay between two nodes. Any propagation delays outside the sample point will result in invalid arbitration.

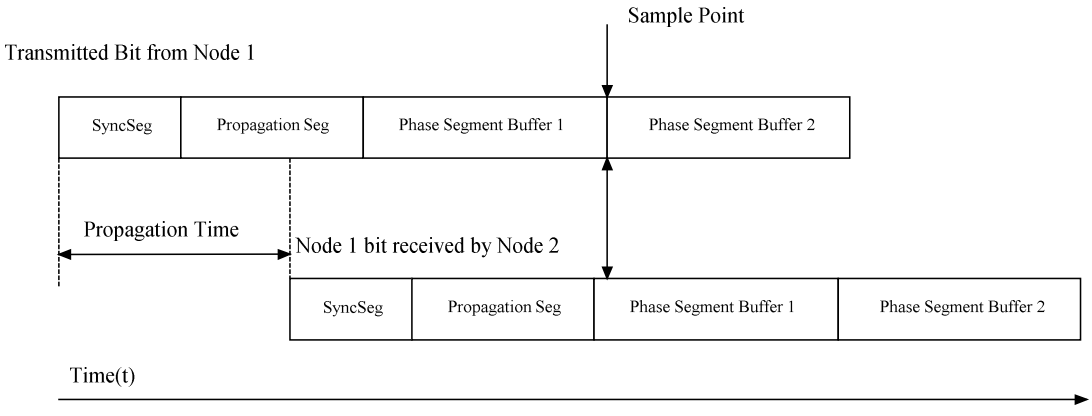


Figure 2.38: Propagation Delay

A CAN system's propagation delay can be calculated as being a signal's round-trip time on the physical bus (t_{bus}), the output driver delay (t_{driver}) and the input comparator delay (t_{cmp}) [9]. Assuming all nodes in the system have similar component delays, the propagation delay mathematically is:

$$Propagation\ Time\ (t) = 2 * (t_{bus} + t_{driver} + t_{cmp}) \tag{2.4}$$

2.3.3.3 Connections

In order to use CAN as an industrial field bus, the CiA created a standard called CiA DS 102-1, which is based on ISO 11898. Of importance in this standard is the use of a 9 pole SUB-D connector for the connection of nodes to the CAN bus lines, as shown in Figure 2.39 [20].

The bus signals CAN_H and CAN_L are available on pins 7 and 2 of the 9 pin connector, while the other pins serve as power or ground wires, or are reserved for future extensions of the standard (Table 2.4).

Pin	Function
1	Reserved
2	CAN_L
3	0V Ground
4	Reserved
5	Reserved
6	0V Ground
7	CAN_H
8	Reserved
9	V+ Power Supply

Table 2.4: CiA DS 102-1 Nine Pole SUB-D Pin-outs

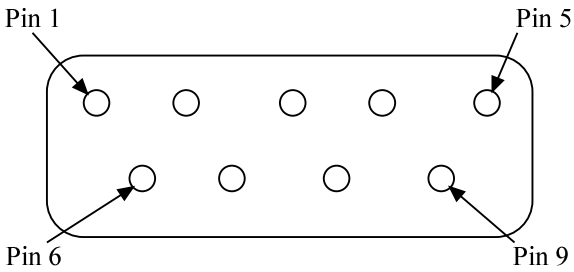


Figure 2.39: Nine Pole SUB-D Connector

2.3.3 Oscillator Tolerance

The CAN system clock for each CAN node will be based upon the individual oscillators of the node. Therefore, the actual CAN system clock frequency for each node will be slightly different, and hence, the actual bit time will be subject to a tolerance. The initial tolerance of the oscillators will also differ due to operating temperature, age, voltage supply, etc. All of these factors will have an effect on the operating frequency. The CAN system clock tolerance is defined as a relative tolerance, where f is the actual frequency and f_n is the nominal frequency [15].

$$\Delta f = \frac{(f - f_n)}{f_n} \tag{2.5}$$

To guarantee error free communication, the minimum requirement for a CAN network is that two nodes, each at opposite ends of the network with the largest propagation delay between them, and also each of them having a CAN system clock frequency at the opposite limits of the specified frequency tolerance of the oscillators, must be able to correctly receive and decode every message transmitted on the network. If this is adhered to, all nodes should be able to sample the correct bit of any message [16].

2.3.4 Cable

According to ISO-11898-2, cables chosen for use in a CAN network as bus lines should have a nominal impedance of 120Ω, and a specific line delay of nominal 5 ns/m. Bus line termination has to be provided through termination resistors of 120Ω located at each end of the bus line. The length related resistance should be 70 MΩ/m.

2.4 CAN Controllers

2.4.1 Introduction

This section looks at the differences between the stand alone CAN controller and an integrated CAN controller. It will consider this device by the load placed on the CPU.

2.4.2 CPU Loading

Figure 2.40 shows a Stand Alone CAN controller layout, which requires three devices: a microcontroller, a standalone CAN controller, and a CAN bus transceiver. The interface between the microcontroller and the CAN controller is an address/data bus or a serial link such as the SPI protocol. The CAN controller is driven by a low-tolerance input clock supplied by a crystal oscillator. The microcontroller also uses a crystal oscillator. The system uses an interrupt line from the stand-alone CAN controller to the microcontroller to signal the reception of a message or the occurrence other CAN events.

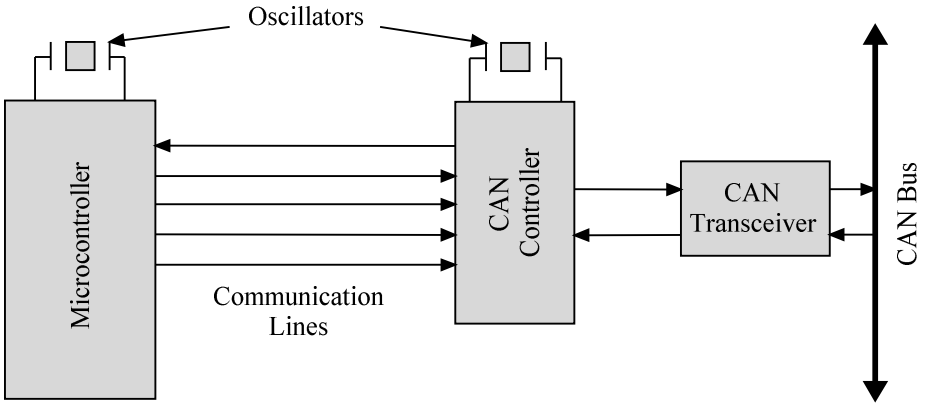


Figure 2.40: Stand Alone CAN Controller Layout

Figure 2.41 implements a microcontroller with an on-chip CAN controller, which clearly simplifies hardware design. In addition, this system uses less printed circuit board area and generates less board noise by eliminating board traces used to interface the microcontroller to the CAN controller. Software development costs are nearly the same for integrated or stand-alone CAN peripherals. In both cases, software must be developed for the microcontroller to read and to write messages to the CAN controller [21].

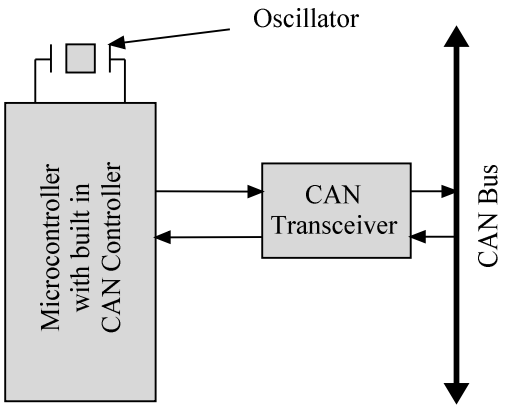


Figure 2.41: Integrated CAN Controller

Table 2.5 shows the communications duties carried out by each CAN node with respect to the protocol, messaging, and system/error response. The CAN protocol involves the controller transmitting and receiving bits according to arbitration rules defined by the CAN protocol. It must also calculate a 15-bit CRC code, which is transmitted with each message and is verified by each receiving CAN node. The CAN Controller must implement all the protocol tasks without CPU intervention.

Protocol	Bitwise reception/transmission Bus arbitration Error code generation/checking
Messaging	Write data to be transmitted Read received data Manage control/status registers
System/Error Response	Node configuration System commands Local Bus off

Table 2.5: CAN Node Communication Tasks

The CPU must service all messaging tasks. It requires the CPU to write the data to be transmitted, to read received data back from the controller and manage the status/control registers in the CAN peripheral. Since the CPU uses the CAN peripheral as a smart RAM, messaging tasks are fundamentally CPU read/write operations. A CPU with an inbuilt CAN controller will read/write to register locations using its own internal bus. For a CPU with an interface to a stand-alone CAN chip,

these read/write operations typically use the external address/data bus or a serial link using the SPI protocol [21].

In addition to these read/write operations, the CPU may be required to manipulate message identifier bits and data fields. For example, a data byte may contain two or more parameters such as engine airflow and engine temperature within the one byte of information. In this case, the CPU must execute bit shifting and masking operations to extract the correct data bit/bits.

The CPU burden required to manipulate this data is the same for on-chip and stand-alone CAN peripherals. The CPU demand differs for on-chip and stand-alone CAN, due to the access time of CAN registers for the different controllers [21].

	CPU Load		
Stand Alone CAN	**250kbits/s**	**500kbits/s**	**1Mbit/s**
8 bit A/D Bus	5.5%	11%	21.9%
16 bit A/D Bus	4.2%	8.4%	16.7%
Integrated CAN			
Registered RAM	2.0%	4.0%	8.0%

Table 2.6: CPU Loading

System/error response is a category of infrequent use and is initiated by the system or by an unusual number of bus errors. The CPU executes error recovery routines when the CAN peripheral is in bus off state. Recovery from bus-off requires a hardware or software reset of the CAN peripheral. The CPU burden to communicate with the CAN peripheral is dependent on a few factors. The most critical factor is the amount of time required to read/write to the CAN peripheral. In the case of an on-chip CAN peripheral, the CAN registers are addressed using the internal address/data bus designed for high-speed access. In the case of a stand-alone CAN chip, the CPU uses an external address/data bus or a serial communications link. Table 2.6 shows the level of CPU burden while receiving CAN messages for three CAN bus transmission

rates. This analysis compared the CPU burden of an Intel 82527 stand-alone CAN chip to an Intel 87C196CA 16-bit microcontroller with on-chip CAN.

2.5 Message Sending

2.5.1 Introduction

This section will look at the various types of message scheduling that are available to the system designer in order to leverage the optimum benefits from the network. There are broadly two types of scheduling available to the designer, namely:

- Event Triggered
- Time Triggered

There is a trend towards an increased number of interconnected devices on a network with the use of smart sensors giving increased data throughput, which results in an increased functionality of the system. This increased data reduces available bandwidth on the bus, thus message scheduling systems that maximise the utilisation factor, while supporting message deadlines together with optimising microcontroller loads are very important in reducing costs.

2.5.2 Event Triggered CAN

CAN is a serial bus system with multi-master capabilities. All CAN nodes are able to transmit data and several CAN nodes can request the bus simultaneously. The serial bus system with real-time capabilities is the subject of the ISO 11898. In a CAN network there is no addressing of subscribers or stations in a conventional sense, but a prioritised message is transmitted instead whenever an event occurs, e.g. coolant temperature changes. The transmitter sends a message to all CAN nodes and each node decides on the basis of the identifier received whether it should process the message or not. The identifier of the message determines its priority, and is defined by the network designer. The message priority is critical in that it will dictate the message's success in arbitration for bus access, although a high priority identifier does not always ensure immediate access to the bus. Also low priority messages may never gain access to the bus under certain circumstances. Some of the problems associated with real time event triggered systems will be discussed in the next section.

2.5.2.1 Event Triggered Problems

In order to show some of the problems that can affect an event triggered network some definitions regarding message transfer will be given.

The response time, R_m, of a CAN message, is the time interval from when the message is ready for transmission until the time it is acknowledged, and successfully received by any other node or nodes. The message does not have to be repeated in any sense and will thus not demand any further bus resource.

The delivery time D_m of a message can be defined as the time interval from when the message is delivered by an application in a node, until it becomes available at other nodes (Figure 2.42).

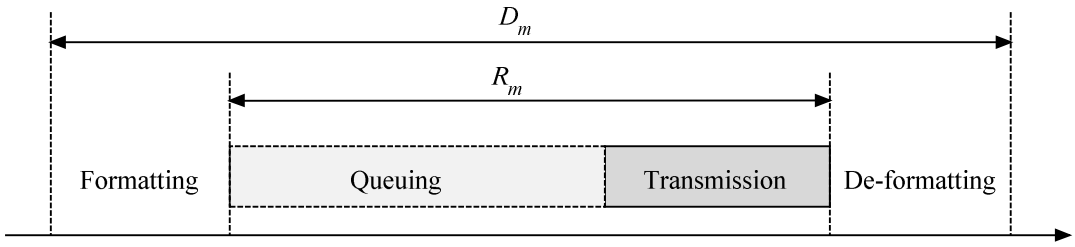


Figure 2.42: Message Delivery Time

The CAN network is a single user resource; once allocated it cannot be shared, and once its message is started, it is guaranteed to complete its transmission unless it loses arbitration or an error occurs. The message schedule is determined by CAN message identifiers since CAN uses a Fixed Priority Scheduling system.

The transmission time of a message is constant, since once the message length is known as well as the baud rate, then the transmission time can be calculated [22].

The total message delivery time, D_m, for a message, m, is the time from which it has been disposed of by an application to the CAN controller in the sending node, until the message is available for another application in the receiving node. The message delivery time in total is the sum of the following:

- The time taken to format the message for transmission on the network.
- The queuing time (waiting time due to loss of bus arbitration).
- The transmission time depending on the message length and the bit rate.
- The time required to de-format the message and notify the receiver of safe message arrival.

While the time for formatting and de-formatting the message are normally constant, depending on the actual CAN controller and operating system, and while the transmission time can be calculated for each message, the queuing time depends on the actual schedule of priorities of message identifiers.

For a message m, in a set schedule of N periodic messages ($m = 1 \ldots N$) with a period of p_m, a queuing time of Q_m and transmission time of T_m the message response time R_m is calculated as follows [22].

$$R_m = Q_m + T_m \tag{2.6}$$

If three messages, M1, M2, and M3 are to be transmitted from different nodes, it can be shown that the worst case queuing time occurs if all three messages become available for transmission at the same time, as seen in Figure 2.43.

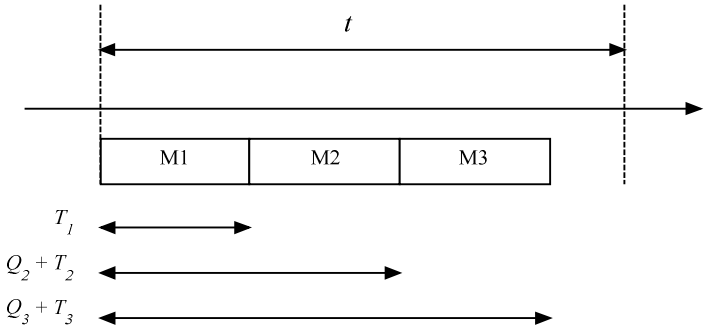


Figure 2.43: Queuing Time

In addition, Message 1 has the highest priority and Message 3 has the lowest priority. Message 1 will not have any delay in queuing since the CAN arbitration protocol will resolve the bus conflict in favour of Message 1.

T_j is the transmission time for a higher priority message j and P_j is the period time of the higher priority message j. Equation 2.7, provides the solution to the maximum queuing time [22].

$$Q_m = \sum_{\forall j:hp(m)}^{1} \frac{Q_m}{P_j} (T_j) \tag{2.7}$$

Due to the non pre-emptive property of a CAN message transmission, we also have to consider message blocking. This can cause additional queuing delays for a message of high priority, when a lower priority message has control of the bus. This event can occur when a low priority message becomes available for transmission just before a high priority message requires use of the bus. Since the transmission is non pre-emptive, the entire low priority message is transmitted and delays the high priority message from point P_2 to D_1, as shown in Figure 2.44.

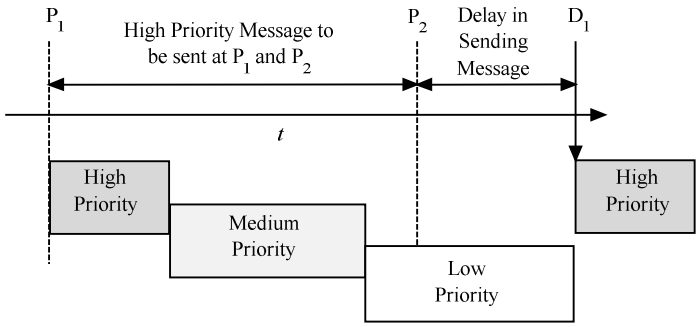


Figure 2.44: Queuing Delay Due to Blocking

The equation for blocking is:

$$B_m = max\ (T_k)\ \{\forall k\ lp\ (m)\} \tag{2.8}$$

Where B_m is the blocking term for message m which is obtained by getting the transmission time for the longest message of a lower priority [22].
To calculate the complete response time we use:

$$R_m = B_m + T_m + Q_m \tag{2.9}$$

Where:

- R_m is the total response time for message m.
- B_m is the blocking time for message m as a result of interference from lower priority messages.

- Q_m is the queuing time for message m as a result of higher priority messages being transmitted and thus delaying the message m.
- T_m is the transmission time for message m.

It can be seen that event message handling does not guarantee the arrival of any message on time, even the messages with the highest priority can be delayed by the lowest priority message under certain conditions. The lower the priority of the message, the higher the latency jitter is likely to be [23] and in some instances the message may never get access to the bus due to being blocked by higher priority messages. The goal of Time Triggered CAN is to avoid these latency jitters and to guarantee a deterministic communication pattern on the bus [24].

2.5.3 Time Triggered CAN

TTCAN allows the designer to use the physical bandwidth of the network more efficiently, under the constraint of determinism [24]. The TTCAN protocol is specified in ISO 11898-4.

TTCAN is based on the CAN data link layer protocol ISO 11898-1 and does not infringe any part of that protocol. Time-triggered communication means that activities are triggered by the elapsing of time segments. In a time-triggered communication system, all points of time of message transmission are defined during the development of a system. A time-triggered communication system is ideal for applications in which all or most data traffic is of a periodic nature [25].

TTCAN provides the possibility to schedule CAN messages in a time-triggered mode as well as in an event-triggered mode. This type of message strategy is very effective when a network is used for a closed-loop control system such as the powertrain in a vehicle. Also the real-time performance of a CAN network increases with the use of TTCAN.

Most vehicle networks dictate that data traffic must usually be both event-triggered e.g. temperature change in the transmission system and time-triggered e.g. gearbox torque output versus engine speed.

2.5.3.1 TTCAN Extension Level 1

ISO 11898-4 defines two levels of TTCAN. Extension level 1 guarantees the Time Triggered operation of CAN based on a reference message of a time master. Fault

tolerance of the functionally is determined by using redundant time masters. This type of TTCAN is capable of being fully implemented in software [24].

2.5.3.2 TTCAN Extension Level 2

Extension level 2 uses a globally synchronised time base, which is established on the network and any drift due to oscillator differences are corrected by the TTCAN controllers. This category of TTCAN is implemented in hardware [24].

2.5.3.3 The Reference Message

TTCAN Extension Level 1 is based on a periodic reference message, which all nodes can recognise by its identifier. This reference message only holds the control information of one byte and the rest of the CAN message can be used for data transfer if required. In Extension level 2, the reference message holds the actual global time of the current TTCAN time master and uses four bytes of data to execute this. The remaining 4 bytes of this message may be used for data communication [24, 26].

2.5.3.4 TTCAN Basic Cycle

The period between two reference messages is called the basic cycle (Figure 2.45). A basic cycle can involve the use of several time windows of different sizes and allows other necessary messages to be transmitted.

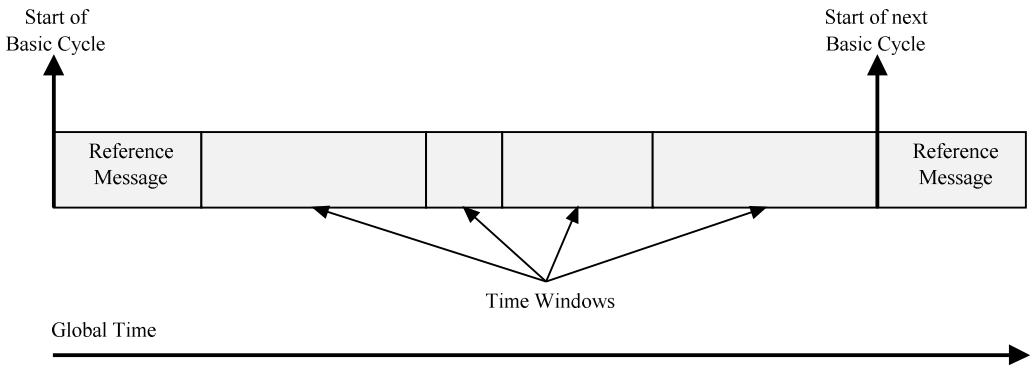


Figure 2.45: Reference Message – TTCAN Basic Cycle

The time windows of the basic cycle can be used for periodic messages and/or for unplanned messages, which will use arbitration to obtain control of the TTCAN network. A time window for periodic messages is known as an exclusive time window (Figure 2.46). Within exclusive time windows, the beginning of the time

window determines the sending point of a predefined message from a node. If the system was properly specified, the design tool used for TTCAN should analyse the communication time periods, and ensure no conflicts will occur. If a conflict occurs due to poor system design, the CAN protocol properties of bit arbitration are valid. The system designer has to determine which message will be sent in each exclusive time window, as the automatic retransmission of CAN messages due to errors or loss of arbitration problems is not allowed in the exclusive time window; therefore the use of "one shot mode" [27] must be used.

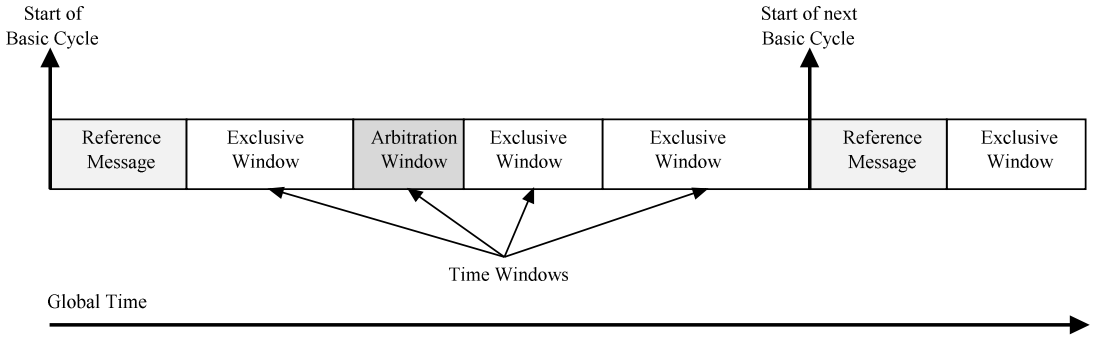


Figure 2.46: Exclusive and Arbitration Windows – TTCAN Basic Cycle

A time window for event messages is called an arbitrating time window and control of the arbitrating time window is by bitwise arbitration, as with event triggered messages. The designer can allow all messages to compete for the arbitrating time window. This permits the application to elect at runtime which messages should use the arbitrating time window and in which time period. The automatic retransmission of CAN messages is also not authorised within the arbitrating time windows.

During the design phase of the network message set, it is also possible to reserve free time windows for further extensions of the network. These reserved arbitrating, or exclusive, time windows can be reconfigured as required if additional nodes require bandwidth on an existing network [24].

2.5.3.5 Node Specific Knowledge

A TTCAN node does not need knowledge of all messages on the network; it only needs to be familiar with the time for sending and receiving of exclusive messages in particular to itself, and where the arbitration window time slots are set. An example of this strategy is seen in Figure 2.47.

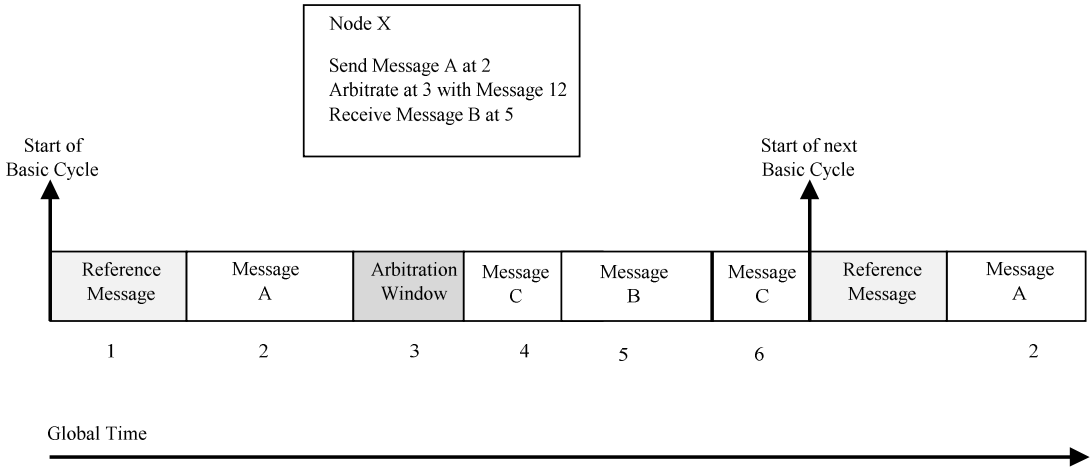


Figure 2.47: TTCAN Communication

This arrangement gives maximum memory optimisation with sufficient information for the node regarding the actual message scheduling. It also offers a high level of flexibility during the development stages as only the message schedule would require updating if there were changes to the network message communication structure [26].

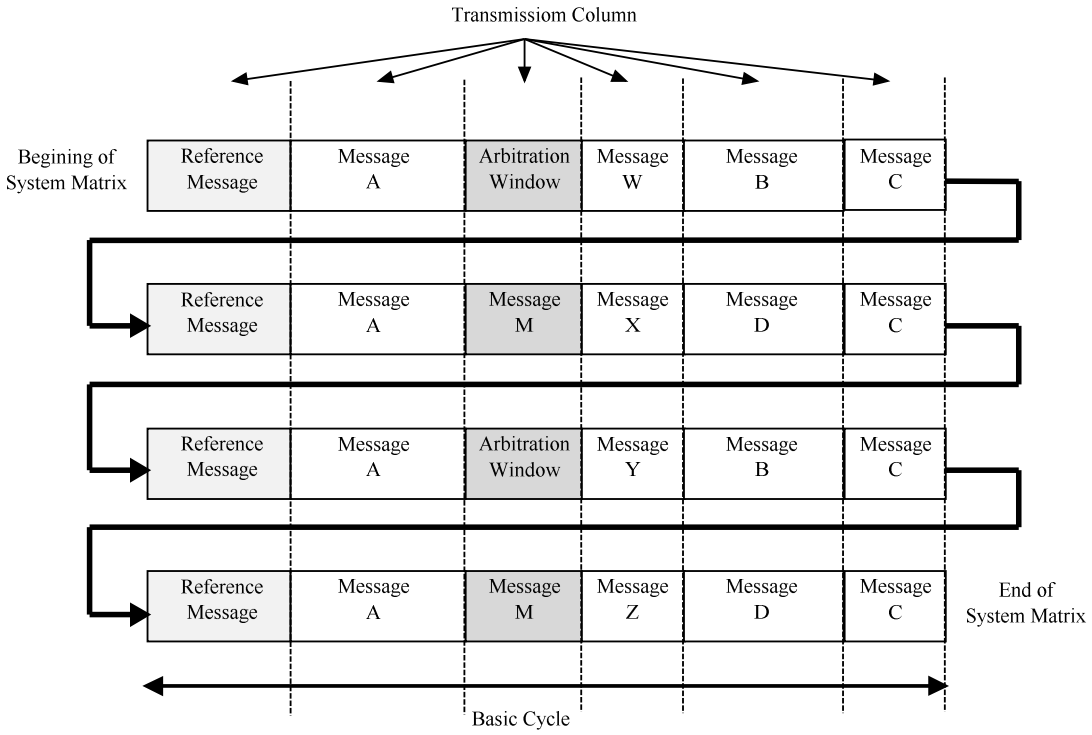


Figure 2.48: TTCAN System Matrix

2.5.3.6 System Matrix

Due to system complexity, a simple basic cycle would not suffice in a modern vehicle, which has many control functions and tasks operating on the one network. TTCAN allows the use of more than one basic cycle. By connecting several basic cycles together we can build what is termed a System Matrix (Figure 2.48) [28].

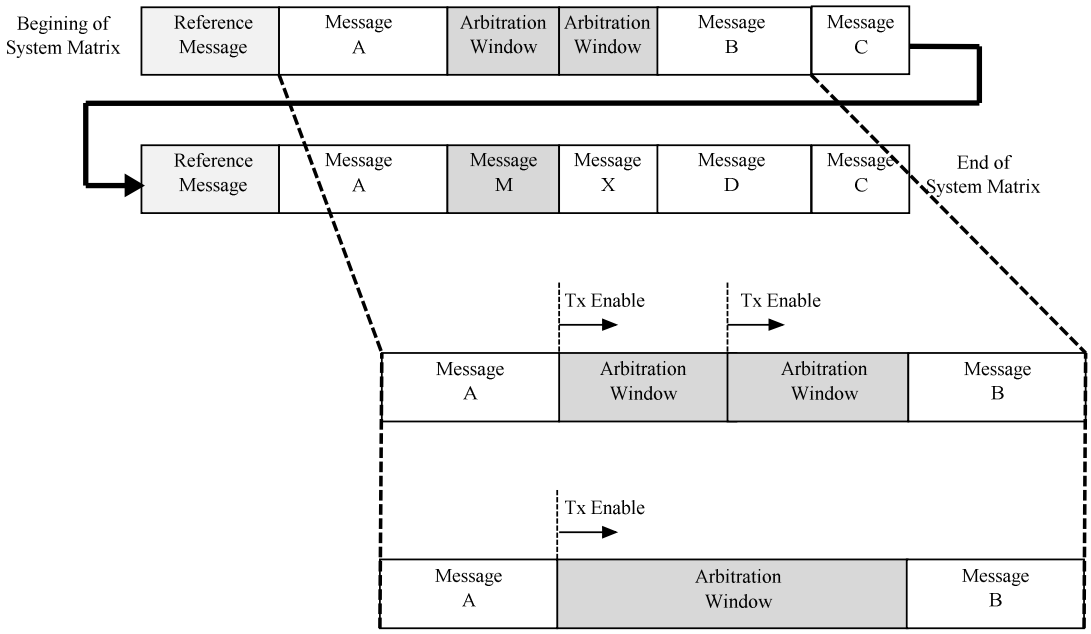


Figure 2.49: Merging Arbitration Windows

This arrangement gives great flexibility to the designer and even permits the use of different column widths by joining two or more time windows together within a System Matrix (Figure 2.49).

The network designer must formulate the column widths in such a way that a CAN message including stuff bits can be transmitted within the allotted time. Failure to enforce this rule will cause the failure of the next message within the System Matrix.

2.5.3.7 Time and Base Marks

TTCAN is enforced by the progression of time within the basic cycle. After every Reference Message, the basic cycle time, or time mark, is set to zero, with time marks dictating the beginning of the exclusive or arbitrating time windows. Base marks are used to track the number of basic cycles within a System Matrix and are set to zero at the beginning of a System Matrix (Figure 2.50) [28].

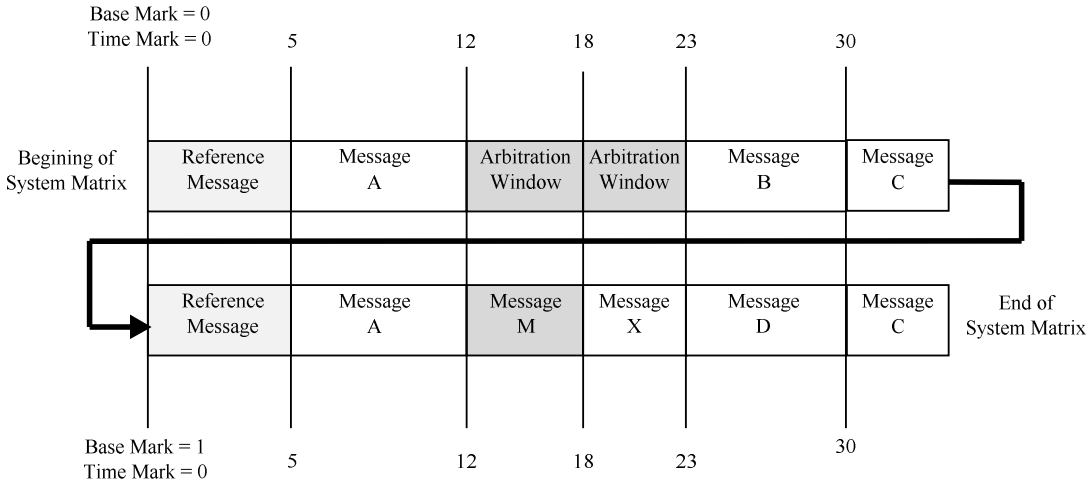


Figure 2.50: Time and Base Marks

2.5.3.8 TTCAN Network Time Units (NTU)

The basic cycle time is the prime time used by TTCAN, and all timing within the TTCAN network utilises the NTU. For Extension level 1 TTCAN, the NTU is based on the nominal CAN bit time and Extension level 2 employs the physical second as the time base. Extension level 2 establishes a system wide NTU by setting a relationship between the node's physical oscillator attached to the TTCAN controller and the data in the Reference Message [28].

2.5.3.9 Global Time Extension Level 2

All level 2 nodes sample their time value at the frame synchronisation (SOF) of the Reference Message sent by the TTCAN Master, and is known as the global time. Following the receipt of the reference message the local node can calculate the local offset as the difference between the SOF of the reference and its own value for global time.

$$Global\ Time = Local\ Time + Local\ Offset \tag{2.10}$$

It is of utmost importance that local time and global time are synchronised, so that all nodes have the same view of time in the network. Global time and local time differences occur due to slightly different clock drift within the CAN nodes. To solve

this problem a mechanism is built into level 2 CAN controllers, which continuously updates the local time offset, by use of a TUR. To achieve this, the local node measures in clock cycles, the time between two successive frame synchronisations and then calculates the TUR and re-establishes the correct global time [29].

2.5.3.10 Initialisation

The TTCAN, as previously stated, has a Reference Message broadcast by the Time Master. In the event of a problem with the Time Master on initialisation, there would be no Reference Message broadcast, so the protocol has allowed for up to 8 potential time masters on any level 1 or level 2 TTCAN network. If the time master level 1 node fails to start then another node will take over its role until the network is switched off. On re-initialisation, the time master will resume its normal function, if it can restart. With level 2 TTCAN, there are 8 potential time masters, which are distinguished by the three-bit time master priority in the Reference Message. The time master priority is given by the three least significant bits of the Reference Message that is transmitted by the respective potential time master. If the original time master gets reattached to the network it will take over the position of global time master [30].

2.6 Message Scheduling Algorithms

2.6.1 Introduction

In computer science, a scheduling algorithm is a method by which a process is given access to system resources, usually processor time, RAM, etc. This is usually done to effectively load balance a system. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking, or execute more than one process at a given time. Scheduling algorithms are generally only used in a time slice-multiplexing kernel (the core of the operating system). The reason is that in order to effectively load balance a system the kernel must be able to forcibly suspend execution of some processes in order to begin execution of the next process. In the case of some embedded systems, this can be achieved by the use of system interrupts. The algorithm used may be as simple as “round-robin” in which each process is given equal time, for instance 1 ms in a cycling list.

More advanced algorithms can take into account process priorities, or the importance of the process. This allows some processes to use more time than other processes. It

should be noted that the kernel always uses whatever resources it needs to ensure proper functioning of the system, and so can be said to have infinite priority [31].

2.6.2 Scheduling

Time-triggered CAN (TTCAN) combines the advantages of event and time triggered communication to fulfil the requirements of a distributed real-time system. Of crucial importance is the generation of the communication schedule, which should consider the demands of the time-triggered system on the one hand, while maintaining a good real-time performance for the event-triggered part of the system on the other. Scheduling is a key requirement for a real-time operating system and regulates the order in which processes are assigned priorities in a priority queue. This message assignment is usually carried out by a piece of software known as a scheduler. In real-time environments such as a braking system on a car, the scheduler ensures that processes can meet the deadlines set, therefore keeping the system stable. Scheduling concepts with particular emphasis on TTCAN networks have been reviewed from the following articles [22, 32, 33].

Long-term schedulers decide which processes can be admitted to the queue. It will decide when an attempt will be made to execute part of the process or program. Its admission to the set of currently executing processes is either authorised or delayed by the long-term scheduler. Thus the scheduler dictates what processes are to run on a system and the degree of concurrency to be supported at any one time - i.e. whether a large or small amount of processes are to be executed concurrently, and how the split between input/output intensive and CPU intensive processes is to be handled. Long-term scheduling is very important in real-time systems, as the ability to meet process deadlines may be compromised by the slowdowns and contention resulting from the admission of more processes than the system can safely handle.

The main purposes of scheduling algorithms, is to minimise resource starvation and to ensure fairness amongst all processes using the resources [31, 34].

The next few sub-sections will discuss some of the schedulers that are suitable for use within a real-time automotive network. An extensive list of scheduling algorithms is shown in Appendix 1.

2.6.2.1 Deadline-monotonic Scheduling

Deadline-monotonic priority assignment is a priority assignment policy used with fixed priority pre-emptive scheduling. Using deadline-monotonic priority assignment,

tasks are assigned priorities according to their deadlines. The task with the shortest deadline, being assigned the highest priority. This priority assignment policy is optimal for a set of periodic or sporadic tasks that comply with the following:

1. All tasks have deadlines less than or equal to their minimum inter-arrival times (or periods).
2. All tasks have worst-case execution times that are less than or equal to their deadlines.
3. All tasks are independent and so do not block each other's execution.
4. No task voluntarily suspends itself.
5. There is some point in time, referred to as a critical instant, where all of the tasks become ready to execute simultaneously.
6. Scheduling overheads (changing from one task to another) are zero.
7. All tasks have zero release jitter (the time from the task arriving to it becoming ready to execute).

If restriction seven is not adhered to, then "deadline minus jitter" monotonic priority assignment is the optimal solution. Deadline monotonic priority assignment is not an optimal solution for fixed priority non-pre-emptive scheduling [35].

2.6.2.2 Earliest Deadline First Scheduling

Earliest Deadline First (EDF) scheduling is a dynamic scheduling principle used in real-time operating systems. It places processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline. This process will then be scheduled for execution next. EDF is an optimal scheduling algorithm on pre-emptive single processors in the following sense: if a collection of independent tasks, each characterised by an arrival time, an execution requirement, and a deadline, can be scheduled, such that all the tasks are completed by their deadlines, the EDF will schedule this collection of tasks such that they all complete by their deadlines.
When scheduling periodic processes that have deadlines equal to their periods, EDF has an utilisation of 100 percent. That is, EDF can guarantee that all deadlines are met if the total CPU utilisation is not more than 100 percent. So, compared to fixed

priority scheduling techniques like rate-monotonic scheduling, EDF can guarantee all the deadlines in the system at higher loading.
However, when the system is overloaded and tasks miss their deadlines, this is largely unpredictable and is a considerable disadvantage to a real time systems designer [36].

2.6.2.3 Rate Monotonic Scheduling

Operating systems are generally pre-emptive and have deterministic guarantees with regard to response times. Rate monotonic analysis is used in conjunction with those systems to provide scheduling guarantees for a particular application.
A simple version of rate-monotonic analysis assumes that processes have the following properties:

1. No resource sharing (processes do not share resources, e.g. a hardware resource, a queue, or other blocking mechanism).
2. Deterministic deadlines are exactly equal to periods.
3. Static priorities with the task with the highest static priority that is available, immediately pre-empts all other tasks
4. Static priorities assigned according to the *rate monotonic* conventions (tasks with shorter periods/deadlines are given higher priorities)

It is a mathematical model that contains a calculated simulation of periods in a closed system, where round robin and time-share schedulers fail to meet the scheduling needs. Rate monotonic scheduling looks at a run modelling of all tasks in the system and determines how much time is needed to meet the guarantees for the set of tasks in question [37].

2.6.3 Stochastic Optimisation Algorithm

Stochastic is derived from the Greek word “stochos”, which translates to a meaning of conjecture and randomness and is a process that can be described best by a probability distribution when used in the scheduling of messages on a TTCAN network. This section is based on work carried out by three Higher Institutes of Education for the 2002 International CAN Conference [33].

2.6.3.1 TTCAN Scheduling Using Stochastic Optimisation

The process of building a message scheduling set for a TTCAN network using stochastic optimisation consists in building the SM. This SM typically includes the following elements:

- The determination of the number of columns required in the matrix
- Establishing the number of rows
- The definition of the duration of each column
- The message to be transmitted in each cell (row, column)

It is important with this approach to keep the average period of each message identical to the respective instantaneous period. This is done using the appropriate number of message instances in the system matrix. The average period is equal to the duration of SM divided by the number of message instances [33].

The message duration will determine the column width and despite the restriction imposed to the number of basic cycles as indicated in section 2.5.3.6, it is usually possible to build several distinct system matrices for the same message set. If several different messages sets are built, then we will have to assess the optimum schedule in the generated SM's according to some pre-defined criterion.

In most cases a cost function based on the sum of the message jitter values is used as the criteria, which is used extensively by the automotive industry [33]. Jitter is determined for every instance of every message, covering the complete SM. The cost function is weighted by the SM duration and can be tested by using the following expression:

$$Jitter = \frac{1}{M} \sum_{p} \sum_{i} \left| e_i^p \quad a_i^p \right| \tag{2.11}$$

Where e_i^p is the expected beginning time of transmission of instance $_i$ of message $_p$ and a_i^p is the actual beginning time and M is the duration of the system matrix.

In order to build the SM some type of software will have to be written and must be capable of scheduling all messages and optimise the message set.

2.6.3.2 Stochastic Scheduling

The scheduler must be capable of generating a series of feasible message sets of which all are distinctly different SMs. The optimisation part of the software must be capable of selecting the best SM based on the cost function in Equation 2.11. In order to be able to maintain the average period of every message, the SM's duration M, must be the least common multiple of the message periods, or an integer multiple of it. The average period is kept with M / P_i instances of every message $_i$ of period P_i during the SM [33].

In order to schedule a set of messages we must:

- Determine the maximum number of lines of the system matrix
- Set the message allocation.
- Calculate the free time redistribution.

If we disregard the restriction of the number of basic cycles, it is obvious that the maximum number of lines in the SM is bounded by:

$$L_{max} = \frac{M}{T_{max}} \tag{2.12}$$

Where L_{max} is the maximum number of lines and T_{max} is the maximum transmission time of all the messages in the set.

Before starting the allocation process, the software will have to generate ordered set I, which includes every instance of every message in the initial set. This set is organised in decreasing order of the message transmission time T:

$$I = \{I_1^1, i_1^2, ..., I_1^{K_1}, I_2^1, ...I_2^{K_2}, ..., I_n^1, ..., I_n^{K_n}\} \tag{2.13}$$

With n being the number of different messages in the initial set and:

$$K_i = \frac{M}{P_i} \tag{2.14}$$

$$T_{max} = T_{max} > T_2 > ..... > T_1 \tag{2.15}$$

The software would now need to define a random number of lines for the SM, where:

$$L \leq L_{max} \tag{2.16}$$

It is now necessary to remove the first L instances in I and allocate them to the first column of the matrix. We repeat this cycle until all instances of I are dealt with. We now have a SM with #C columns.

$$\#C = \frac{\#I}{L} \tag{2.17}$$

As the longer messages are taken into account first, $\#C$ is now at its minimum. It would now be possible to determine the minimum duration of the basic cycle for this particular matrix also by using:

$$D_{BC} = \sum_{i=1}^{\#C} D_i \tag{2.18}$$

In Equation 2.18, D_i is the duration in time for each column of the matrix. With this value, it is possible to ascertain if the set is schedulable by using Equation 2.19:

$$D_{bc} \leq \frac{M}{L} \tag{2.19}$$

If the set is schedulable, then we should check for any free time that is available within the basic cycle. This can be checked by using Equation 2.20.

$$t_{free} = \frac{M}{L} - D_{BC} \tag{2.20}$$

Free columns are placed between each two occupied columns and then there is a redistribution of the free time randomly between the first columns. This is the only random factor in the construction of the first set of system matrices.

2.6.3.3 Stochastic Optimisation

The optimisation process uses a set of system matrices built under the rules for the stochastic scheduler. These matrices are deemed as the initial population and will be subject to random alterations.

The alterations to the SM must assure that the matrices are still feasible after the random changes. The cost function defined above is now used to determine the jitter within the matrices [33].The steps in the optimisation process are shown in Table 2.7 below. Step 1 is the user defining the number of elements that constitute the initial population of an SM. The scheduler then generates a usable system matrix based on Section 2.6.2.2.

1	Generate an initial population
2	Diversify the population.
3	Select a SM.
4	Randomly transform the selected matrix.
5	Evaluate the cost function for this SM.
6	If the cost function (jitter) is lower for this SM, keep this matrix and eliminate the old SM.
7	Repeat steps 4 to 6 until the pre-defined maximum number of iterations is attained (normally 1000).

Table 2.7: Steps Required for Stochastic Optimisation of a TTCAN SM

In step 2, the elements of the population are randomly moved as each new SM is developed. The random movement of the elements provides diversity in the initial population and produces a number of different SMs made up from the one population (set of messages).

Steps 3 to 7 is the optimisation stage of the process. Step 3 selects a SM, with step 4 using an algorithm based on a modified steady state genetic algorithm [33] which eliminates the problem of crossover, therefore, keeping all transformed system matrices useable. Another issue regarding the completion of the algorithm is the number of iterations that should be used, as jitter depends on the transmission load

and on the relationship between message periods. The number of iterations must be determined by the user, but must be the same for all SMs.

The focus of this optimisation process is to use as many different transformations of the message schedule in order to reduce or eliminate message jitter. It should be remembered the lower the message jitter the greater optimisation of the system.

2.6.4 Heuristic Scheduling Concepts

Heuristic is derived from the Greek word "heurisko", which translates directly to "I find". The definition given in the Oxford English Dictionary is "proceeding to a solution by trial and error or by rules which are loosely defined" [38]. This section is based on work carried out by Robert Bosch GmbH for the 2005 International CAN Conference [32]. The use of heuristic scheduling will provide a resolution to the message-scheduling problem, but this solution may not be the optimum schedule.

2.6.4.1 TTCAN Scheduling Using Heuristic Methods

Designing a message schedule is comparatively simple using the heuristic method. It involves sorting the messages according to:

- Repetition rate (period)
- Message length (total bits per message)
- Are they periodic or spontaneous messages?
- Deciding the maximum response time to a spontaneous message
- Knowing the dependencies between messages

Once the messages have been sorted, a basic attempt at Rate Monotonic (Section 2.6.2.3) scheduling can be implemented. The length of the basic cycle is chosen according to the shortest period and the number of basic cycles is derived from the longest period [32].

Below in Figure 2.51 depicts an example of a Heuristic Schedule based around the following data:

- CAN baud rate is set at 62.5 kbits/s.
- Message 101 has a period of 10ms. It is a standard frame message with 7 data bytes, so the message can be up to 118 bits long.
- Message 201 has a period of 20ms. It is a standard frame message with 7 data bytes, so the message can be up to 118 bits long.

- Message 202 has a period of 20ms. It is a standard frame message with 7 data bytes, so the message can be up to 118 bits long.
- Message 301 has a period of 30ms. It is a standard frame message with 7 data bytes, so the message can be up to 118 bits long.
- Message 302 has a period of 30ms. It is a standard frame message with 7 data bytes, so the message can be up to 118 bits long.

The first schedule is generated in a rate monotonic fashion and any unused windows in Figure 2.51 can be used as arbitration windows for spontaneous messages or for further expansion of the network.

Each message has been calculated to take 1.888ms of bus time, therefore, if message 101 is sent at zero time, message 201 can be sent at the start of 2ms.

0ms	1ms	2ms	3ms	4ms	5ms	6ms	7ms	8ms	9ms
101		201		202		301		302	
101									
101		201		202					
101									
101		201		202					
101									

Figure 2.51: Basic Heuristic Message Schedule

This gives a time between message 101 and message 201 of 0.112 ms. This time period between these two messages is too small to allow for an arbitration window. Looking at the first 12 milliseconds of the schedule, we cannot have any arbitration windows, but from the beginning of the 12ms until the end of the 19ms, we can have an arbitration window.

The rest of the schedule has different sized arbitration windows, which can be seen in Figure 2.52. If we use this message schedule, there will be no transmission of spontaneous messages for the first 12 ms of the schedule, but after this, there is a large arbitration window. This will have an impact on a real-time application. The

microcontroller will also be tied up with the CAN message schedule for most of this time and have little time to look after other events.

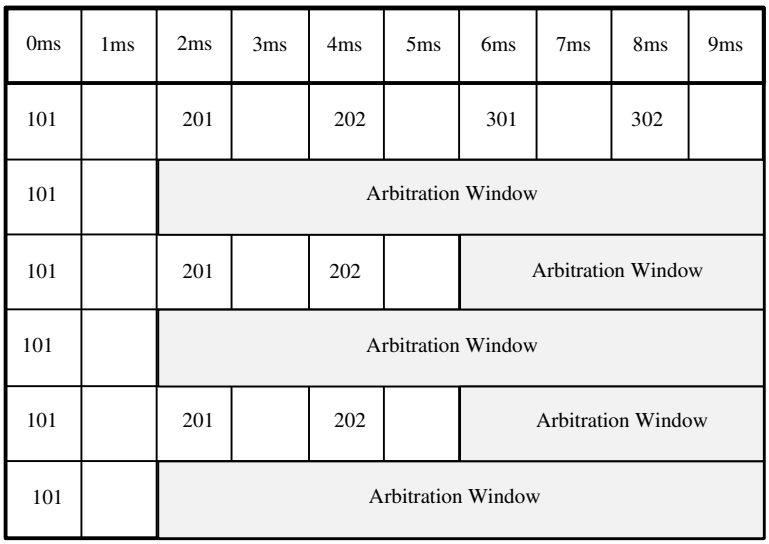


Figure 2.52: Heuristic Scheduling showing Arbitration Windows

If the arbitration windows can be distributed more evenly throughout the SM a significant improvement can be made in real-time performance [32]. Testing of real-time performance can be carried out by evaluating the ability of the CAN network to react to an asynchronous event by the "Distinctness of Reaction", based on the orthogonal Walsh correlation and gives a reliability measure, which will give the average latency response time and the jitter when reacting to asynchronous external events [32].

2.6.4.2 Heuristic Message Strategies

There are two distinct message strategies available with heuristic scheduling. The first strategy minimises the number of basic cycles in order to produce a useable schedule which will use the greatest amount of triggers in the SM. The second strategy is to minimise the length of the basic cycle in order to minimise the number of triggers required to operate the system. TTCAN level 2 relies on hardware triggers and has to be in the order of 2^n up to a maximum value of 2^6. Level 1 systems do not have this constraint.

Robert Bosch GmbH found from DoR testing that it was advantageous to use long basic cycles since they use less reference messages. Also, as additional basic cycles

are incorporated into a SM, the real-time performance will deteriorate and the jitter and average latency increase and this, in turn, will increase bandwidth usage [32].

2.6.4.3 Allocation of Message Slots

From a message point of view it doesn't matter where the actual time slots are within the SM, the only constraint being the Reference message that has to be sent by the system Master at time zero. There are two methods of allocating message time slots; one uses what is termed "dense" allocation and the other uses "sparse" allocation.
The dense allocation is suitable in some instances where, for example, sensor data is gathered from different nodes and broadcast on the bus.

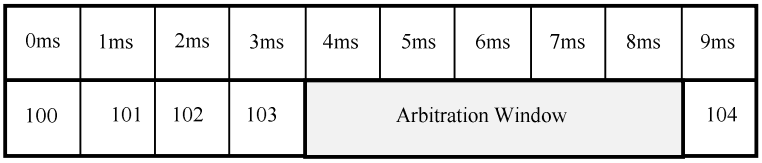


Figure 2.53: Heuristic Dense Message Allocation

These values should be available as soon as possible to minimise control system delays and therefore, a dense allocation is preferable (Figure 2.53). This method will create long intervals where spontaneous messages that use arbitration will be blocked and therefore, if the data has a deadline, it may be exceeded.

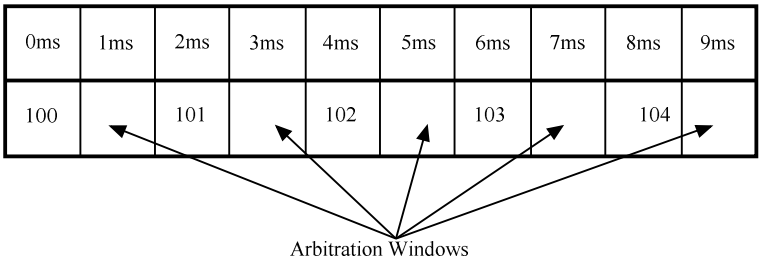


Figure 2.54: Heuristic Sparse Message Allocation

The second option is sparse allocation were the TTCAN messages are spread out as far as possible. This results in shorter blocking periods with more arbitration windows available for spontaneous messages (Figure 2.54) and is better suited to real-time applications.

Optimisation of a heuristic schedule is calculated by using the cost function based on the sum of the message jitter as with stochastic message scheduling. The schedule with the lowest cost function is deemed the optimum schedule.

2.7 Summary

This chapter examined reasons for the development of CAN for the automotive industry. It scrutinised the CAN specification, including the OSI Data Link Layer and the Physical Medium Attachment as defined by ISO-11898. The physical layer wasinvestigated in terms of functionality and operation, as was the bus differential voltage and physical attachment to the transceivers. It considered the effect of propagation delay and the oscillator tolerances on network stability, and how these can be compensated by use of the CAN bit timing. Microcontroller CPU loading was explored, with particular attention being focused on the effects of stand alone CAN and integrated CAN on the CPU.

CAN message sending was researched in respect of both event triggered messages (real-time) and time triggered messages. It found serious drawbacks in using event triggered messaging due to the type of arbitration used with CAN e.g. high priority messages will always control the bus. Time triggered messaging guaranteed all time triggered messages would have access to the bus at some time and would allow event triggered messaging at slack periods.

TTCAN was investigated and it was found that there are two different implementations available. A level 1 system, executed through software and a level 2 system, implemented through hardware.

Several scheduling algorithms were investigated, but only the stochastic and heuristic schedulers showed any promise with a TTCAN network. After detailed analysis of the schedulers, it was found that they can develop useable schedules, either by probability distribution or by trial and error, but are unable to generate every possible message set from any group of message periods. Neither can they elicit any information about the arbitration window size. Message scheduling algorithms were examined from a viewpoint of dense and sparse allocation. It was found that sparse allocation was the preferred option for real-time messaging.

The next chapter will illustrate the methods used to solve both problems endured by the stochastic and heuristic scheduler. It demonstrates a methodology, which ensures

all message sets can be developed from a group of periodic messages and also solves the problem of finding the optimum message set with regard to real-time messaging.

Chapter 3: Designing the Optimum TTCAN Message Scheduler

3.1 Introduction

This chapter investigates the reasons why optimisation of a TTCAN System Matrix is necessary and introduces a design process to produce the most effective solution. The chapter is set out in the following sections:

- Section 3.2 explores the reasons why an effective scheduler is required and illustrates the problems with the stochastic and heuristic schedulers presently used.
- Section 3.3 seeks to find a method to solve the shortcomings of the stochastic and heuristic schedulers.
- Section 3.4 demonstrates the mathematical solution to the above problems and implements it in software.

3.2. Stochastic and Heuristic Scheduler Problems

3.2.1 Introduction

A scheduling algorithm is a means by which a process is given access to system resources and is used to efficiently load balance the system at all times, as described in section 2.6.1. It was outlined in section 2.6.4.3 that when dense scheduling was employed in the System Matrix, the TTCAN network could be operating at maximum load for the duration of time that these messages were being sent. Whereas, in sparse message scheduling the load on the TTCAN network had the load spread evenly and arbitration windows inserted between each TTCAN message.

3.2.2 Stochastic Scheduling

The stochastic scheduler, as described in section 2.6.3, relies on devising usable message sets by randomly distributing TTCAN messages and arbitration windows [33]. Distributing message windows and free time in an indiscriminate way indicates that the optimum schedule may result only by chance.

The manner in which the arbitration windows are distributed can increase the time a spontaneous or real time message has to wait in order to access the TTCAN network in a worst-case scenario. In Figure 3.1, the spontaneous message in a worst-case situation will have to wait approximately 1.3 milliseconds for transmission on the network, if we use single columns in the SM. Figure 3.2 shows the waiting time for

the spontaneous message to be in the order of about 2.3 milliseconds if the TTCAN messages are distributed in pairs.

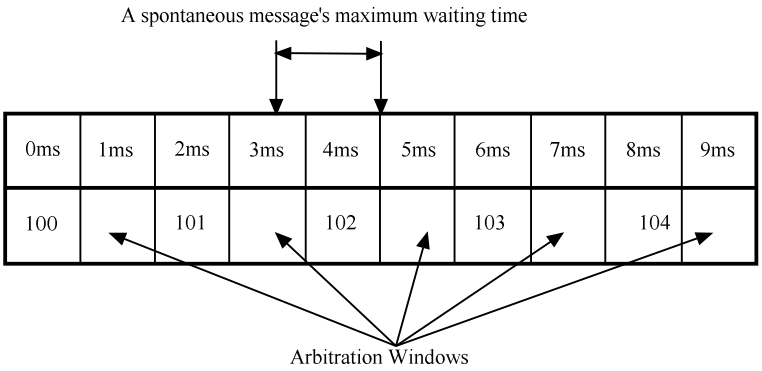


Figure 3.1: Spontaneous Message Waiting with Single Columns

It should be noted that the message waiting time for a spontaneous message is longer than the message time allocated for an actual TTCAN message. The spontaneous message must complete its message transmission prior to the TTCAN message starting to send its own message.

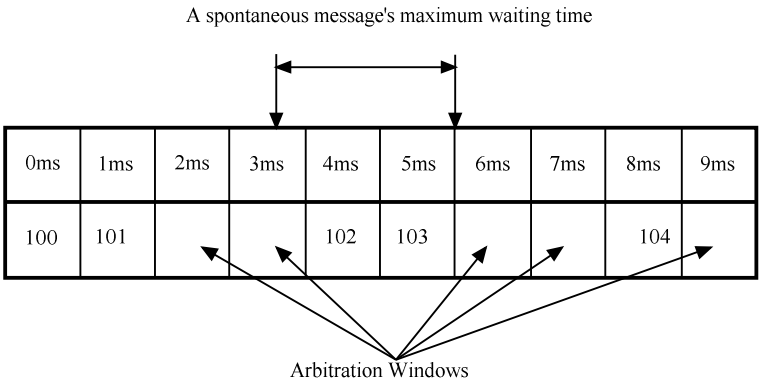


Figure 3.2: Spontaneous Message Waiting with Double Columns

Stochastic scheduling generates a large number of different message sets. Some of these message sets are not usable, and of the usable message sets, it applies the cost function analysis to find the optimum message set. Using this type of scheduling does not mean that the best message schedule has been developed; it means that it has found the best message set with the lowest cost function from the generated message sets. Other message sets may be available outside those that were generated.

3.2.2.1 Designing a Stochastic Message Set

A stochastic Message Set is developed by placing the message population onto the SM randomly, and then testing for usable Message Sets and finally testing their optimisation by using a cost function as described in section 2.6.3.1.

Example 1: Three CAN standard messages with periods of 20ms, 30ms and 40ms operating on a bus with a baud-rate of 62.5kbits/s and each message has 7 data bytes.

Longest message duration:

$$t_{sec} = \frac{Message\_Length_{bits} + Stuffbits_{bits}}{Baud \quad rate_{bits}}$$

$$\frac{100 + 18}{62500} = 1.888ms$$

Length of the SM:

$$LCM\ (20, 30, 40) = 120ms$$

Number of Triggers in SM:

$$Triggers = \sum_{i}^{N} \frac{LCM(M)}{M_i}$$

$$\frac{120}{20} + \frac{120}{30} + \frac{120}{40} = 13$$

Figure 3.3 and Figure 3.4 show two stochastic message sets using the data from Example 1. Both have valid message sets, consequently the cost function, as outlined in section 2.6.3.1, will be used to find the optimum message set.

Cost Function (Message Set 1)

$$Jitter = \frac{1}{M} \sum_{p} \sum_{i} \left| e_i^p - a_i^p \right|$$

$$\frac{1}{120} * [(0_1 \quad 0_1) + (1_2 \quad 2_2) + (16_3 \quad 16_3) + (20_4 \quad 20_4) + (40_5 \quad 40_5)$$

$$+ (41_6 \quad 42_6) + (46_7 \quad 46_7) + (60_8 \quad 60_8) + (76_9 \quad 76_9) + (80_{10} \quad 80_{10})$$

$$+ (81_{11} \quad 82_{11}) + (100_{12} \quad 100_{12}) + (106_{13} \quad 106_{13})] = 0.025$$

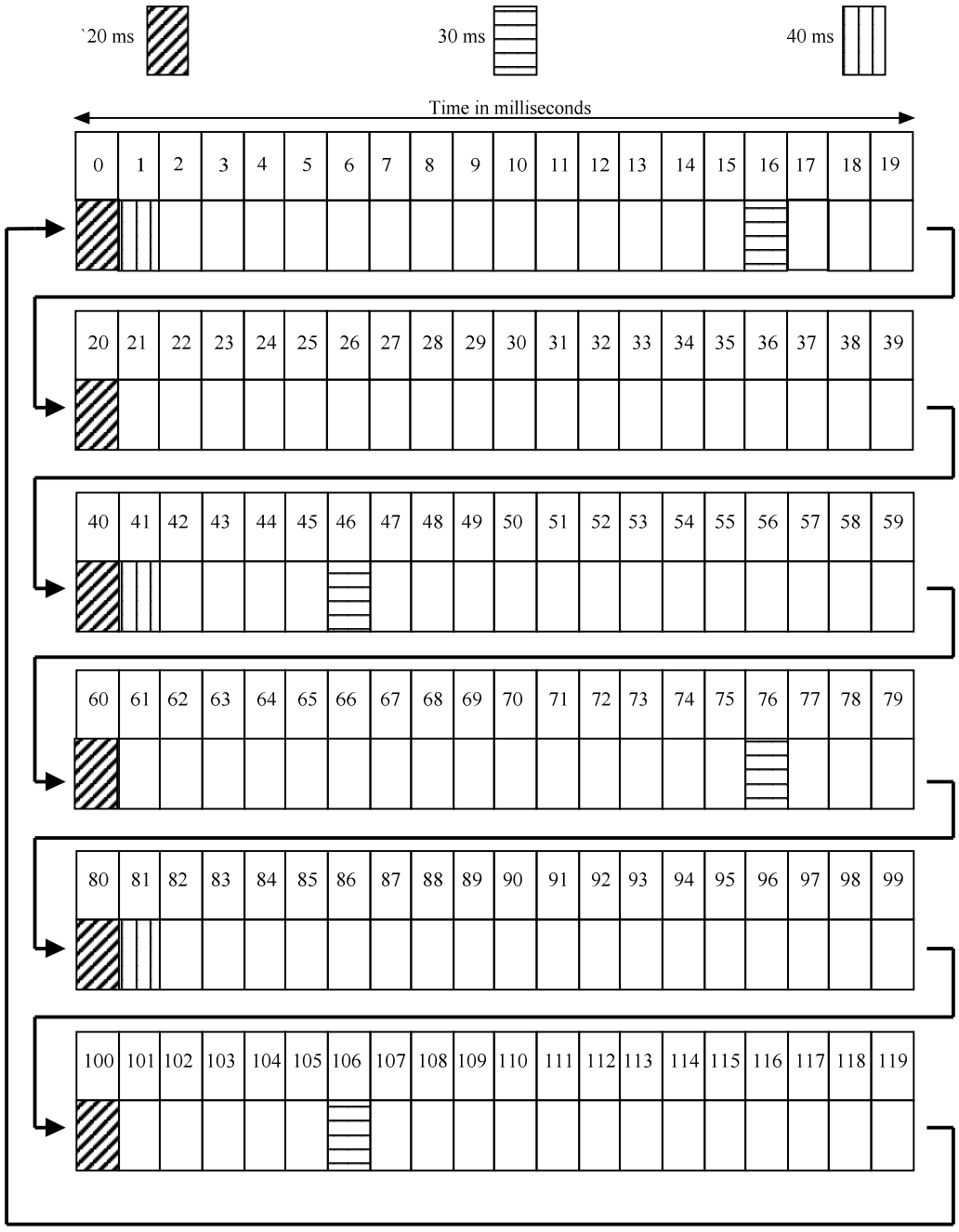


Figure 3.3: Stochastic Message Set 1

Cost Function (Message Set 2)

$$Jitter = \frac{1}{M} \sum_{p} \sum_{i} \left| e_i^p \quad a_i^p \right|$$

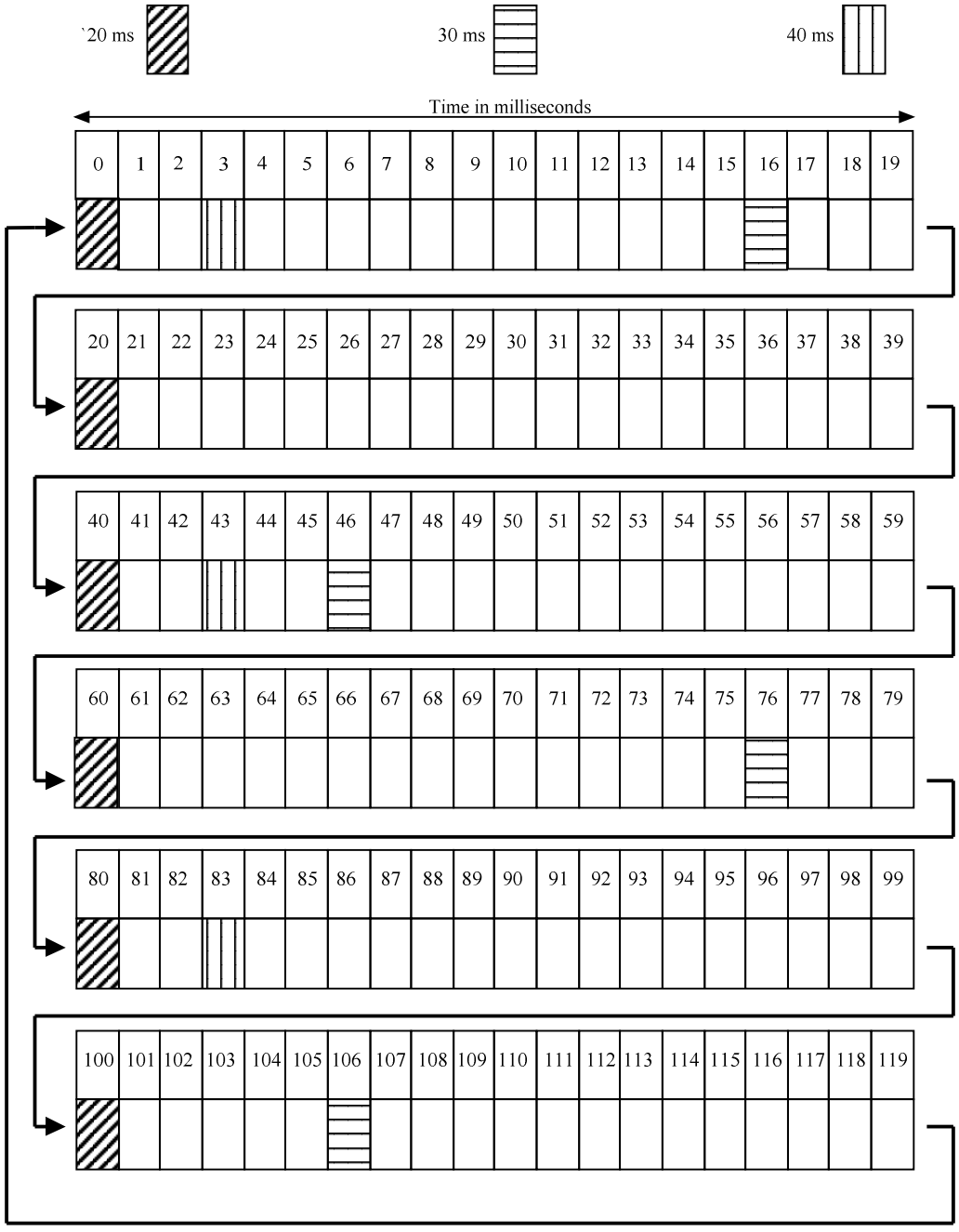


Figure 3.4: Stochastic Message Set 2

$$\frac{1}{120} * [(0_1 \quad 0_1) + (3_2 \quad 3_2) + (16_3 \quad 16_3) + (20_4 \quad 20_4) + (40_5 \quad 40_5)$$

$$+ (43_6 \quad 43_6) + (46_7 \quad 46_7) + (60_8 \quad 60_8) + (76_9 \quad 76_9) + (80_{10} \quad 80_{10})$$

$$+ (83_{11} \quad 83_{11}) + (100_{12} \quad 100_{12}) + (106_{13} \quad 106_{13})] = 0$$

The cost function for Message Set 1 is -0.025 whereas Message Set 2 shows a cost function of zero which denotes that Message Set 2 is the optimum stochastic Message Set.

3.2.3 Heuristic Scheduling

The heuristic scheduler as described in section 2.6.4 relies on developing a usable message set by placing the messages in columns starting with the messages having the shortest period in the first column and messages with longer time periods in new columns in ascending order. A message set will now be developed using heuristic scheduling and employing the same data that was used for the stochastic message sets.

Example 2: Three CAN standard messages with periods of 20ms, 30ms and 40ms operating on a bus with a baud-rate of 62.5kbits/s and each message has 7 data bytes.

Longest message duration:

$$t_{sec} = \frac{Message\_Length_{bits} + Stuffbits_{bits}}{Baud \quad rate_{bits}}$$

$$\frac{100 + 18}{62500} = 1.888ms$$

Length of the SM:

$$LCM\ (20, 30, 40) = 120ms$$

Number of Triggers in SM:

$$Triggers = \sum_{i}^{N} \frac{LCM(M)}{M_i}$$

$$\frac{120}{20} + \frac{120}{30} + \frac{120}{40} = 13$$

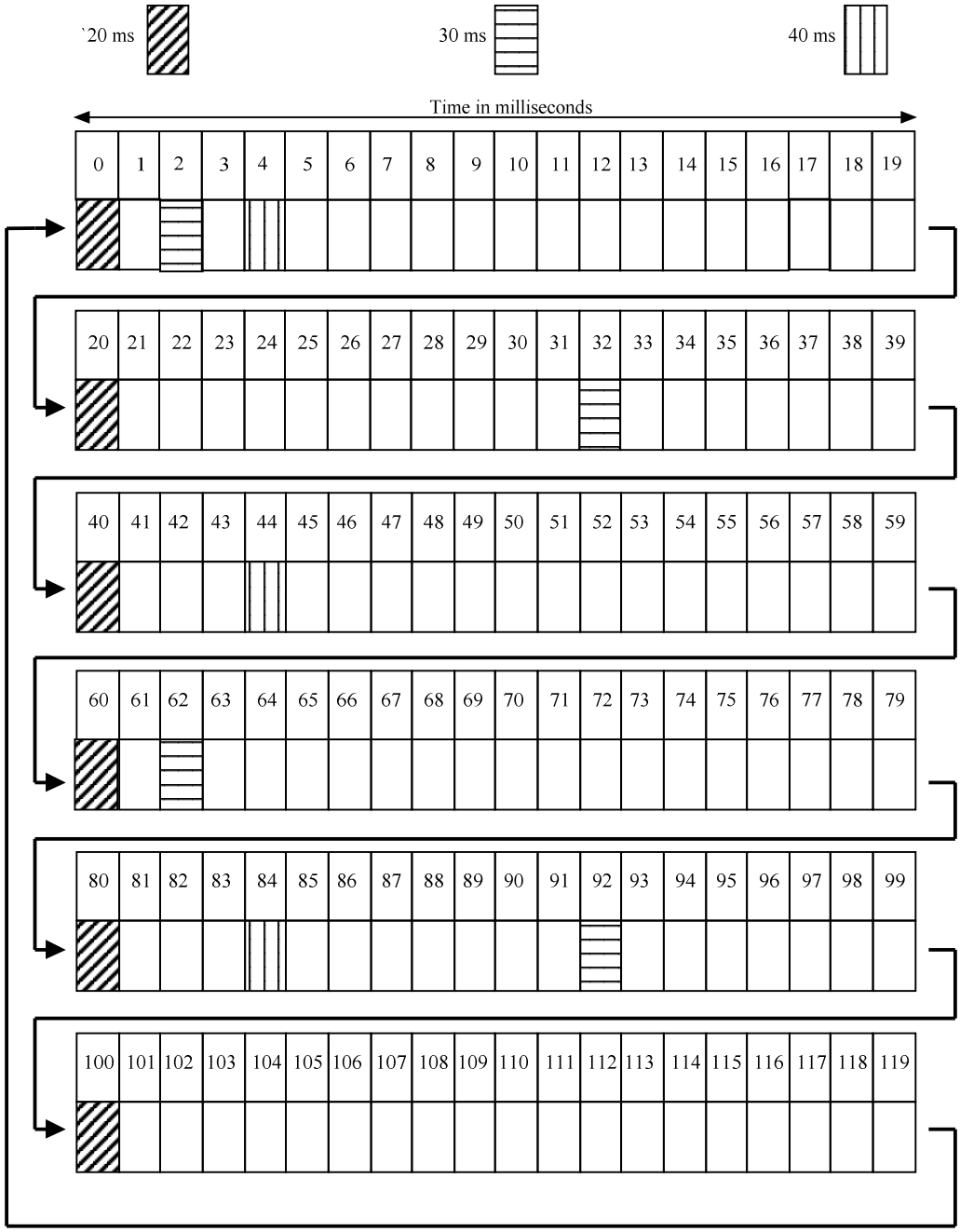


Figure 3.5: Initial Heuristic Message Set

Figure 3.5 shows the initial layout of the heuristic message set, where the Reference message of period 20ms is placed in the first column. The 30ms message is now placed in the next column, which starts at 2ms. The rationale for starting at 2ms is that the Reference message will take 1.888ms to transmit. All of the 30ms messages are inserted in the correct place on the SM. Next, the 40ms message is inserted into the SM again 2ms later, as the previous message will again take 1.888ms to transmit. All instances of the 40ms message are incorporated into the SM.

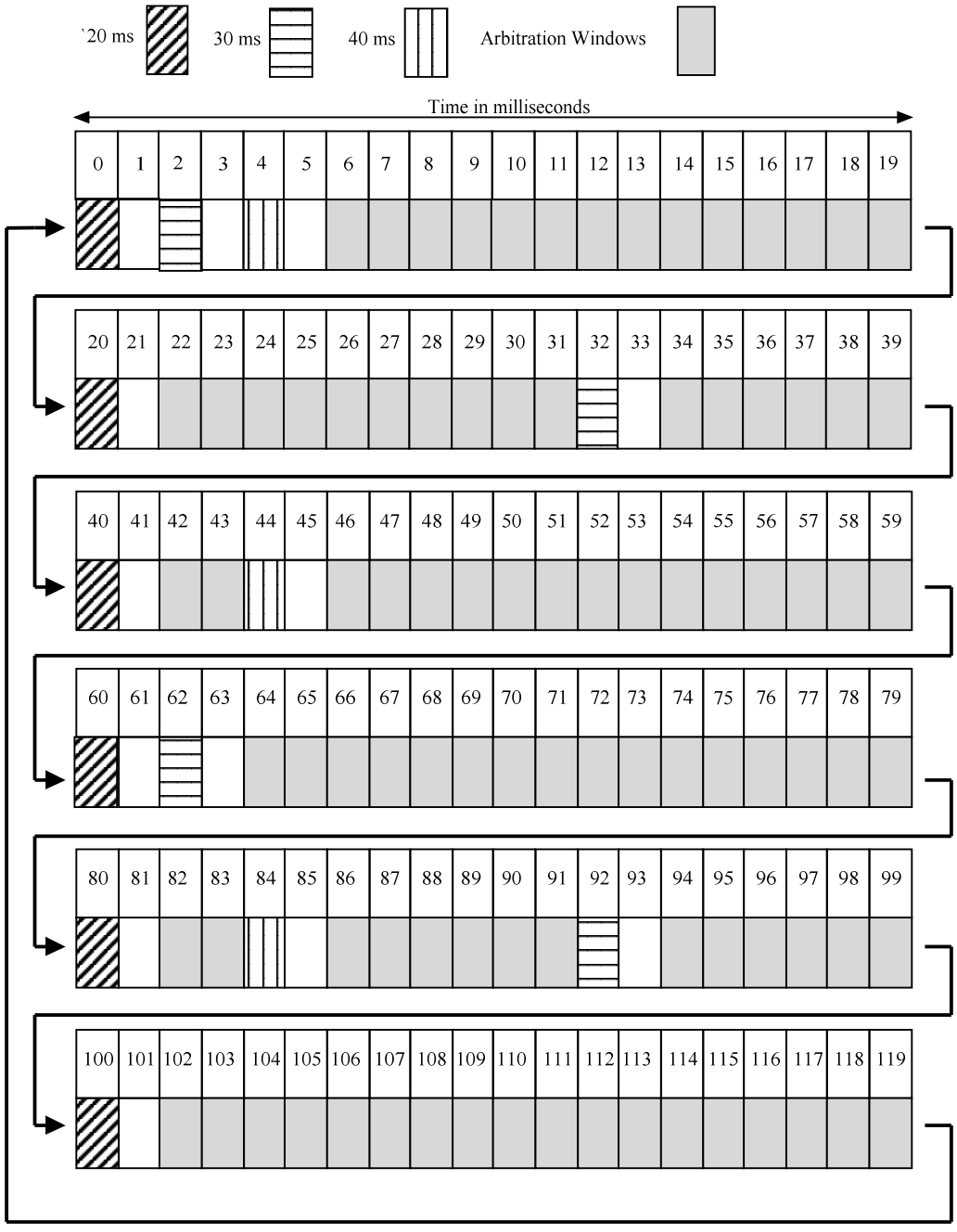


Figure 3.6: Initial Arbitration Windows with Heuristic SM

This message set is now a valid SM, but two problems are associated with it. Firstly, spontaneous real-time messages cannot be broadcast until the start of the 6ms slot, of the SM. Secondly, the spread of arbitration windows across the SM is not the optimum for spontaneous messages, as can be seen in Figure 3.6.

Figure 3.7 shows a valid message set but the arbitration windows have been adjusted by observation, so we can have real-time spontaneous messages while the first three TTCAN messages are being sent.

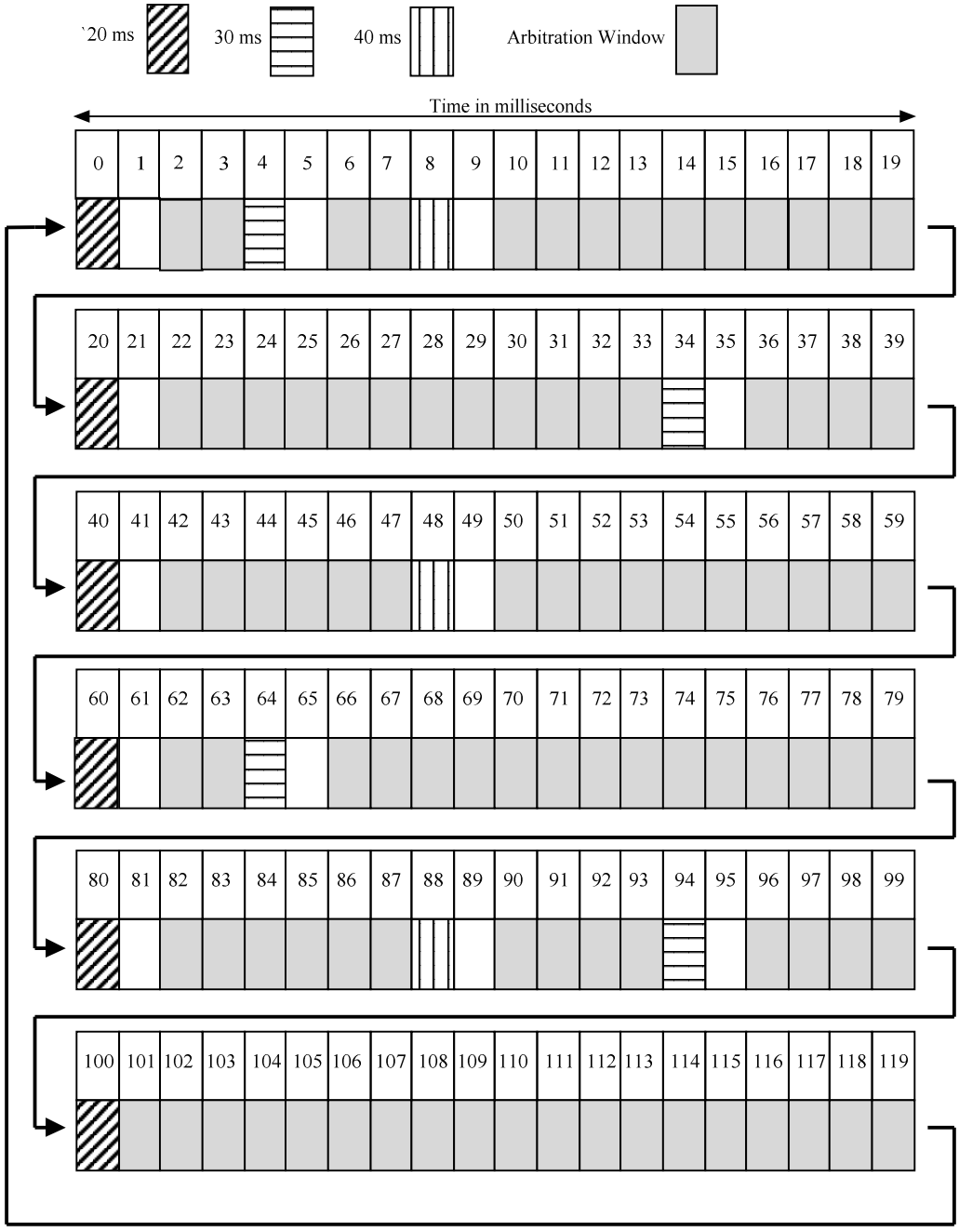


Figure 3.7: Arbitration Window Adjustment Heuristic SM

$$\frac{1}{120} * [(0_1 \quad 0_1) + (4_2 \quad 4_2) + (8_3 \quad 8_3) + (20_4 \quad 20_4) + (34_5 \quad 34_5)$$

$$+ (40_6 \quad 40_6) + (48_7 \quad 48_7) + (60_8 \quad 60_8) + (64_9 \quad 64_9) + (80_{10} \quad 80_{10})$$

$$+ (88_{11} \quad 88_{11}) + (94_{12} \quad 94_{12}) + (100_{13} \quad 100_{13})] = 0$$

The cost functions for the heuristic message set shown in Figure 3.6 and Figure 3.7 are both zero, which denotes that there are two optimum heuristic message sets. The message set shown in Figure 3.6 will have to wait during the first 6ms before a spontaneous message can be sent, whereas in Figure 3.7 a spontaneous message can be sent between each of the first three TTCAN messages. Although the message set in Figure 3.7 offers a more optimised message set, it still does not mean it is the best solution for real-time messaging within the TTCAN network for those message periods.

3.2.4 How Optimised are Stochastic and Heuristic Schedules

The word optimum is a derivative of optimal and has a meaning of "best or most favourable" [38]. This implies that using either stochastic or heuristic scheduler, we should be able to produce the most favourable message set in regards of spreading the load across the TTCAN network and microcontrollers.

It should be apparent that using the rules for stochastic and heuristic message scheduling, several optimised message sets could be developed, as can be seen in Figures 3.5, 3.6, and 3.7. As previously stated, the stochastic scheduler develops sufficient message sets to generate several usable message sets and then uses the cost function analysis to calculate the optimum message set. The cost function analysis will normally find several message sets, each with a cost function of zero. However, all message sets having a cost function of zero are not equally the most favourable, as will be proven later.

Heuristic scheduling relies on trial and error in order to obtain an optimised message set. Again as with stochastic message scheduling, there can be several heuristic schedules with a zero cost function, but without indication that all are optimised to the same extent, or if not, then an indication of the most optimised message set in relation to real-time messaging.

3.3 The Mathematical Approach to TTCAN Scheduling

3.3.1 Introduction

This section will look at mathematics as a possible way of solving the optimum message set. It will not rely on obtaining a solution, by "randomly" (Stochastic) generating a message set or by finding the solution by "trial and error" (Heuristic).

3.3.2 The Mathematical Design Process

This section will outline the design of a very simplified SM consisting of just two messages, using mathematics, and will investigate what rules were applied to the building of it. If the rules developed hold true, then software can be designed to accomplish the process.

The rules for an optimum TTCAN schedule are as follows:

- The messages should have no jitter
- Arbitration Windows to be such that spontaneous real-time messages can operate within their deadlines
- The system resources are to be used as efficiently as possible by load balancing the system at all times

3.3.2.1 Mathematical Two Message SM

Example 3: Two CAN standard messages with message periods of 20ms each operating on a bus with a baud-rate of 125kbits/s and each message has 7 data bytes. Message M1 is the Reference message with system data within the message and M2 is a normal data message in the TTCAN network.

Longest message duration:

$$t_{sec} = \frac{Message\_Length_{bits} + Stuffbits_{bits}}{Baud\quad rate_{bits}}$$

$$\frac{100 + 18}{125000} = 0.944ms$$

Length of the SM:

$$LCM\ (20, 20) = 20ms$$

Number of Triggers in SM:

$$Triggers = \sum_{i}^{N} \frac{LCM(M)}{M_i}$$

$$\frac{20}{20} + \frac{20}{20} = 2$$

Figure 3.8 shows a SM designed by observation and it is a possible solution to an optimum Message set. Does it comply with the rules of an optimum SM as stated in section 3.3.2?

It shows that no message jitter would occur and that the arbitration windows are constant at 9ms although the full 9ms would not be able to be used due to constraints shown in section 3.2.2. The load on system resources is spread evenly.

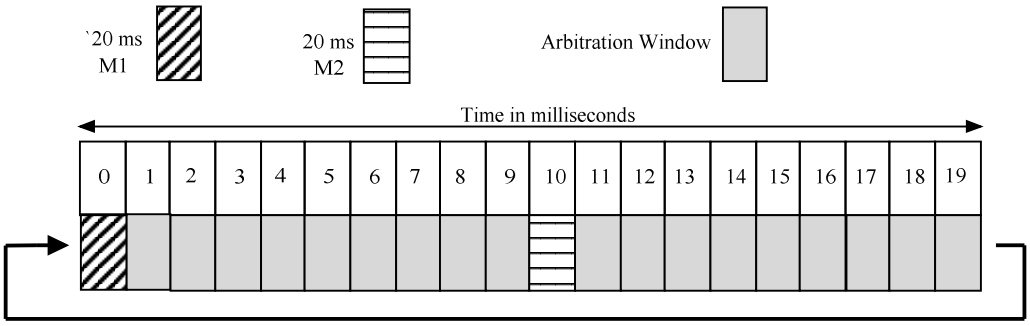


Figure 3.8: Mathematical Design "A" of Period 10ms

3.3.2.2 Modelling Results of Two Message SM

The SM has two messages, M1 is the Reference message and message M2 is broadcast at 10ms. Evaluating the message set above, shows a relationship between M1 and M2. Inspection of the message set in Figure 3.8, shows that the 20ms message

is at the midpoint between the Reference message. This association could be described by Equation 3.1, which gives the position of the messages within the SM:

$$Optmum\_Position\_in\_SM = \frac{System\_Matrix}{Number\_of\_Triggers}$$

$$10 = \frac{20}{2}$$

(3.1)

Alternatively, by using equation 3.2 we can find the optimum position from the previous message.

$$Optmum\_Position = \frac{\sum Arbitration\_Time}{Number\_of\_Triggers}$$

$$\frac{9+9}{2} = 9$$

(3.2)

Both equations 3.1 and 3.2 have given the correct position in the message set. Figure 3.9 uses the data from Example 3, but used a stochastic approach to designing the message set. The cost function analysis shows that the message set is valid and the cost factor is zero.

Inspecting the message set it can be seen that no jitter will occur, but the arbitration windows are of different sizes. Applying Equation 3.1 to the problem shows that the SM is not the optimum, as the message M2 should be located in the 10ms time slot.

Equation 3.1:

$$10 = \frac{20}{2}$$

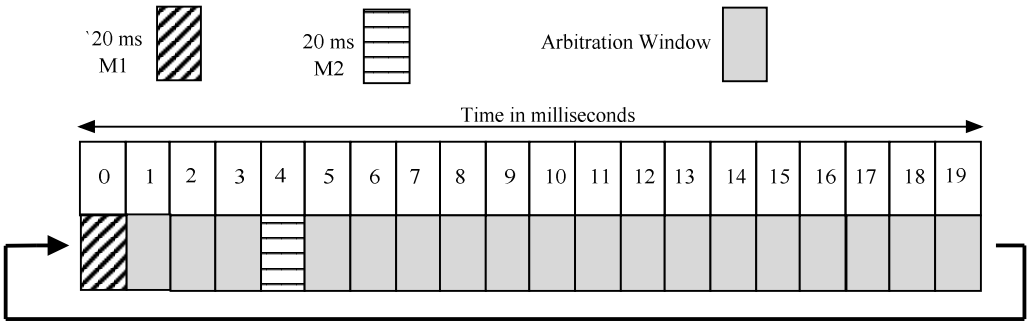


Figure 3.9: Mathematical Design "B" of Period 10ms

Applying equation 3.2 to the SM shows that the message is not the optimum. Message M2 should be located 9ms after the Reference message.

Equation 3.2:

$$9 = \frac{3 + 15}{2}$$

Example 4: Two CAN standard messages with periods of 20ms and 30ms respectively, operating on a bus with a baud-rate of 125kbits/s and each message has 7 data bytes. Message M1 is the Reference message with system data within the message and M2 is a normal data message in the TTCAN network.

Longest message duration:

$$t_{sec} = \frac{Message\_Length_{bits} + Stuffbits_{bits}}{Baud \quad rate_{bits}}$$

$$\frac{100 + 18}{125000} = 0.944ms$$

Length of the SM:

$$LCM\ (20, 30) = 60ms$$

Number of Triggers in SM:

$$Triggers = \sum_{i}^{N} \frac{LCM(M)}{M_i}$$

$$\frac{60}{20} + \frac{60}{30} = 5$$

Using Equation 3.1 and Equation 3.2:

$$Optmum\_Position\_in\_SM = \frac{System\_Matrix}{Number\_of\_Triggers}$$

$$\frac{60}{5} = 12$$

$$Optmum\_Position = \frac{\sum Arbitration\_Time}{Number\_of\_Triggers}$$

$$\frac{11+7+19+1+17}{5} = 11$$

Figure 3.10 shows that Equation 3.1 and Equation 3.2 give the same solution to the problem. If the message M2 of period 30ms could be moved further away from the Reference message M1 at the 40ms time slot, it would have an effect on the real-time messages within the system. It would increase the arbitration window in the message set, which may enable two or more real-time messages to be broadcast during this arbitration window. Figure 3.11 shows such an arrangement, which has the benefits of the largest arbitration windows possible, while providing the best load balance for the complete system.

It can be seen with this message set, that there are five arbitration windows in total, but the two smallest arbitration windows are at time slots 16ms to 19ms inclusive and 41ms to 44ms inclusive.

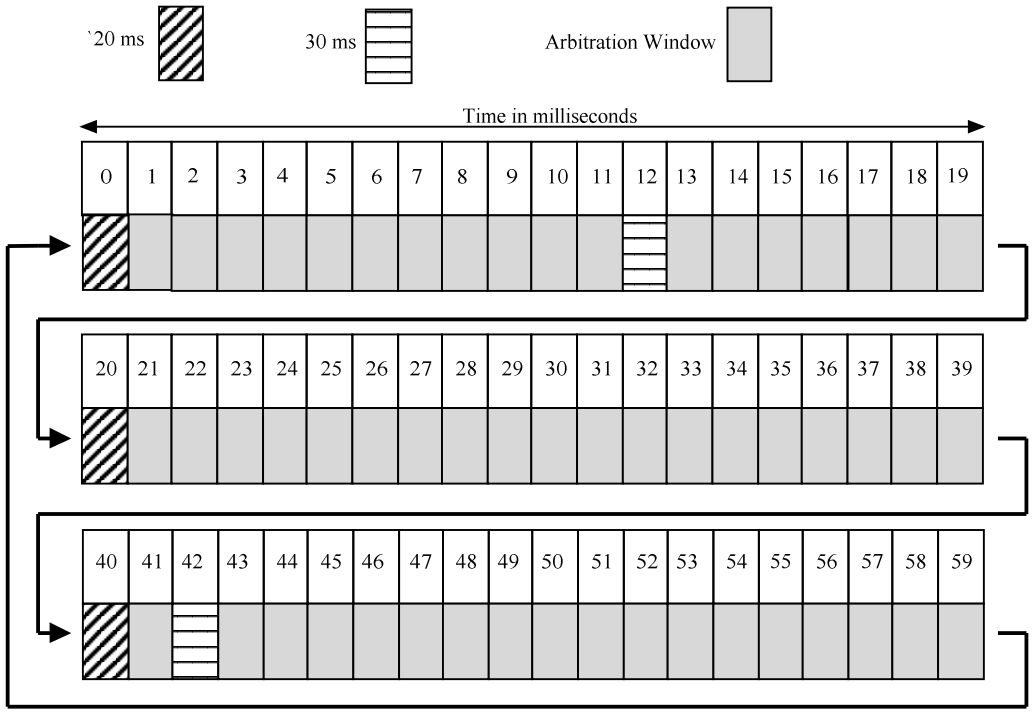


Figure 3.10: SM for Example 4 using Equation 3.1 and Equation 3.2

These two windows are 4ms in duration. By decreasing the arbitration window size, starting at time slot 13ms in Figure 3.10, and moving it towards the Reference message at time slot 20ms, it has given an increase in the arbitration window starting a time slot 45 as seen in Figure 3.11.

Using Equation 3.1 or Equation 3.2 by themselves does not provide the optimum message set. The optimum schedule for Example 3 was found by calculating the mean or average time between messages, but this method did not give the optimum result in Example 4. This can be attributed to the fact that in Example 3 both messages have the same periodic time, whereas for Example 4 the periodic time is different for both messages and therefore it could not achieve the same result. Although Example 4 was completed by inspection, it showed there is a relationship between the size of the arbitration windows and this may be the key to the solution.

In Example 4, it was calculated that the optimum message was to be sent at 12ms, but the actual optimum position was found by inspection to be 3ms from the calculated value. A statistical approach, will now be outline, which provides a message set that ensures the optimum real-time performance for the TTCAN network.

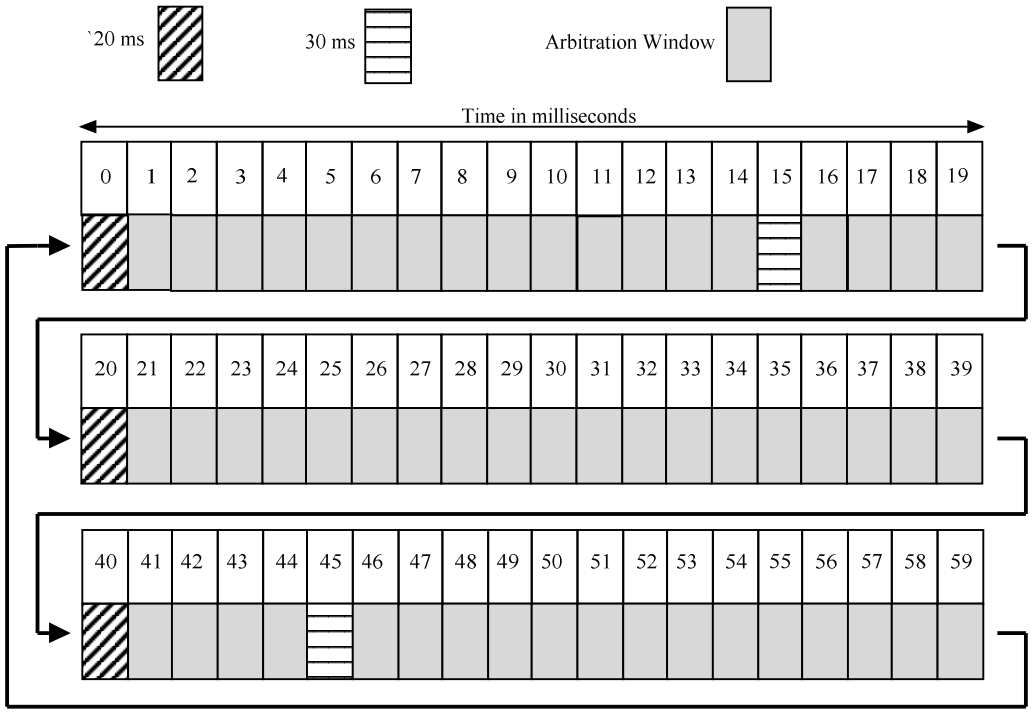


Figure 3.11: Optimum SM for Example 4, by Inspection

3.3.2.3 Statistical Approach to SM Design

A statistical approach to the scheduling problem in Example 3 was undertaken using Microsoft Excel. Care was taken to ensure that the correct analysis tools were used since there are several formulae for Standard Deviation.

The formula to calculate the Standard Deviation of a sample of a population is:

$$\sqrt{\frac{\sum (x-\bar{x})^2}{n-1}} \tag{3.3}$$

Whereas the formula required to determine the Standard Deviation of the complete population is:

$$\sqrt{\frac{\sum (x-\bar{x})^2}{n}} \tag{3.4}$$

Analysing a SM for standard deviation was completed using the "STDVEP" command within Excel. This calculated the standard deviation of a complete population, rather than a sample of the population [39] as all developed message sets were complete populations. The data was further evaluated using MATLAB, which is a mathematical computation, analysis and visualisation tool. It is used extensively for rapid design and testing of systems [40].

3.3.2.4 A Statistical Approach to the Scheduling Problem in Example 3

Two CAN standard messages with periods of 20ms each operating on a bus with a baud-rate of 125kbits/s and each message has 7 data bytes. Message M1 is the Reference message with system data within the message and M2 is a normal data message in the TTCAN network.

Table 3.1 demonstrates 21 possible message sets using the data in Example 3. Heading "Message Set Number" lists the SMs and is incremented in 1 millisecond intervals. The next column shows the start time of the Reference message and will always be zero time.

Column 3 contains the proposed start time of message M2 and, as shown in the list, it is incremented by 1ms throughout the column. Column 4 gives the SM length in milliseconds.

Columns 5 and 6 state the time in milliseconds between the start time of each message in the message set, for example, for message set 1, the Reference message is sent at time zero and finishes broadcasting at just 1ms. Message M2 is proposed to be broadcast at 1ms, therefore, there is zero time between messages M1 and M2. Message M2 will complete transmitting its message just before the 2ms time slot. The network will not broadcast any TTCAN messages for the next 18ms until message M1 is retransmitted to repeat the message set.

Column 8 holds the value of the calculated "Mean" time between messages and it should be noted that it does not remain constant across the data. Column 9 has calculated the standard deviation for that particular message set and shows that there is no standard deviation in message set 10, which is the optimum message set. The worst usable message sets in the set are message set 1 and message set 19.

Figure 3.12 displays the mean and standard deviation graphed against time. It should be noted that it is symmetrically built around the 10ms time slot. Message set 0 and message set 20 are not usable as messages M1 and M2 cannot be transmitted at the

same instance. All other message sets are usable with no jitter and any of these could have been developed by either stochastic or heuristic methods.

Message Set Number	Reference Message M1 Start Time	Data Message M2 Start Time	System Matrix Time (20ms)	Time Difference Between M1 and M2	Time Difference Between M2 and End of SM	Mean Time of Message Departure	Standard Deviation From the Mean Time
0	0	0	20	0	19	9.5	9.5
1	0	1	20	0	18	9	9
2	0	2	20	1	17	9	8
3	0	3	20	2	16	9	7
4	0	4	20	3	15	9	6
5	0	5	20	4	14	9	5
6	0	6	20	5	13	9	4
7	0	7	20	6	12	9	3
8	0	8	20	7	11	9	2
9	0	9	20	8	10	9	1
10	0	10	20	9	9	9	0
11	0	11	20	10	8	9	1
12	0	12	20	11	7	9	2
13	0	13	20	12	6	9	3
14	0	14	20	13	5	9	4
15	0	15	20	14	4	9	5
16	0	16	20	15	3	9	6
17	0	17	20	16	2	9	7
18	0	18	20	17	1	9	8
19	0	19	20	18	0	9	9
20	0	20	20	19	0	9.5	9.5

Table 3.1: The Mean and Standard Deviation of Message Times Example 3

It is evident that the message sets developed using statistical methods produced the optimum message set by sending message M2 at 10 milliseconds. This gives two arbitration windows of 9ms each and a minimum wait time for spontaneous messages (Figure 3.10). This is the point at which the standard deviation is at its minimum.

The graph in Figure 3.12 shows some peculiarities in the “mean”. It changes from 9 to 9.5 when two messages attempt to transmit at the same period of time (message set 0 and message set 20). This is where the Reference messages M1 and the data message M2 are attempting to broadcast at the same instance. The standard deviation is also at

its maximum value when Reference messages and data messages are attempting to be transmitted at the same instance.

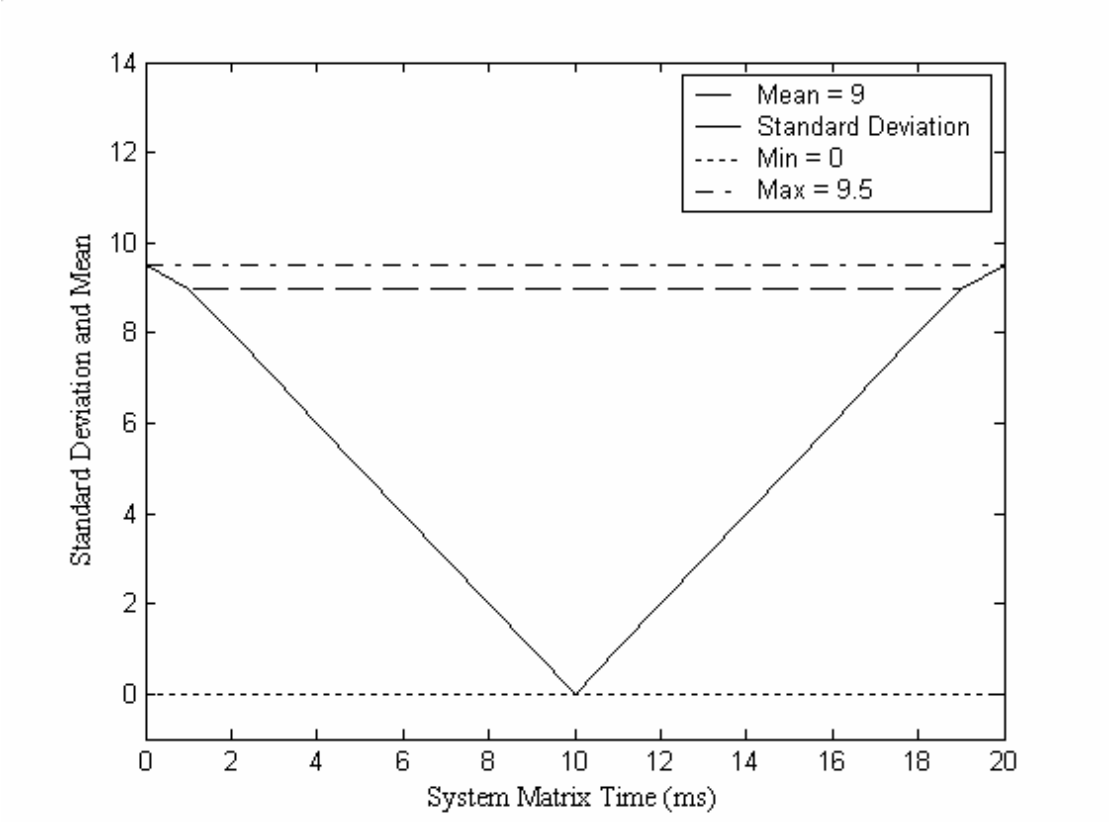


Figure 3.12: Graph Developed by use of MATLAB for Example 3

3.3.2.5 A Statistical Approach to the Scheduling Problem in Example 4

Two CAN standard messages with periods of 20ms and 30ms respectively each, operating on a bus with a baud-rate of 125kbits/s and each message has 7 data bytes. Message M1 is the Reference message with system data within the message and M2 is a normal data message in the TTCAN network.

If the statistical approach holds true from Example 3, the optimum point for message transfers is when the standard deviation is at its minimum. This point is the lowest part of the standard deviation curve.

Appendix 2 shows all message sets developed from Example 4. The data was analysed in MATLAB and a graph produced, which is shown in Figure 3.13. From the graph, it can be ascertained that the optimum point to send a message would be at the minimum standard deviation. It can also be seen that any of the following points could be the optimum; 5ms, 15ms, 25ms, 35ms, 45ms or 55ms. The scheduling problem in Example 4 had been optimised by inspection in Figure 3.11 and the time

slot for message M2 coincides with one of the values calculated by this statistical method at 15ms.

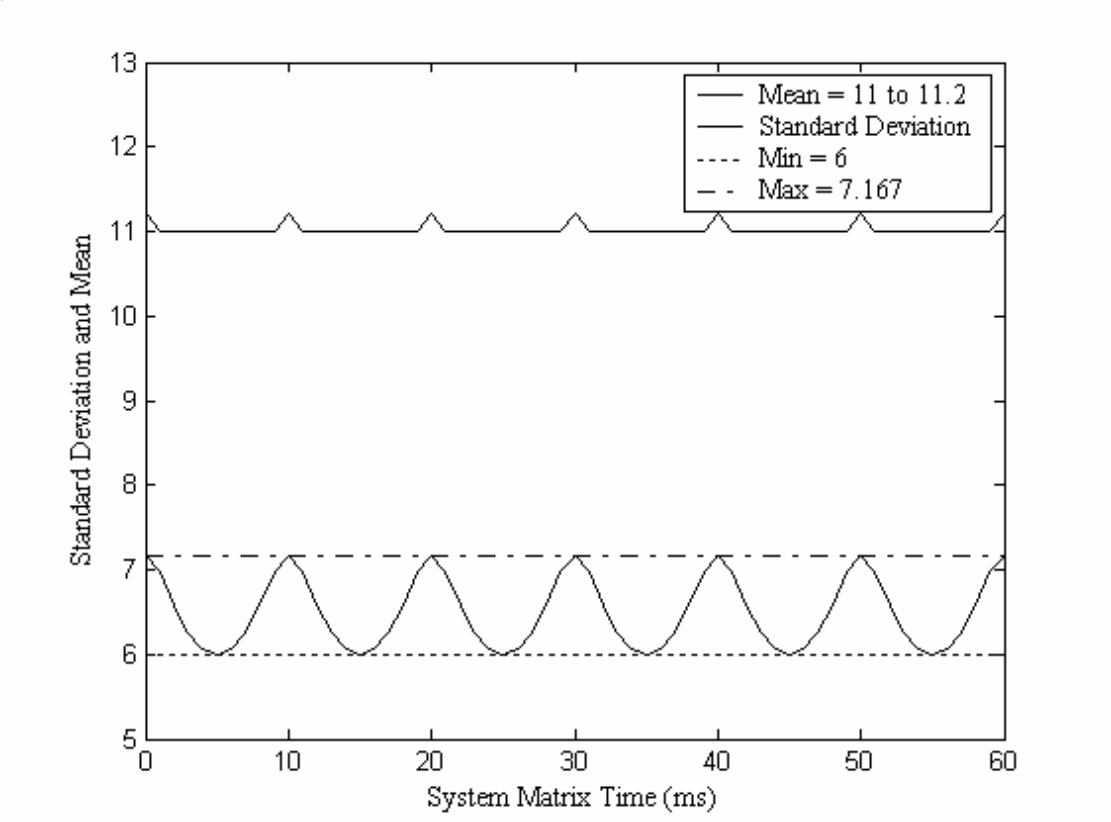


Figure 3.13: Graph Developed with MATLAB for Example 4

Again, the same peculiarities appear in the graph for Example 4 (Figure 3.13) as in the graph for Example 3 (Figure 3.12). The "mean" in this instance changes from 11 to 11.2 when two messages are transmitted in the same period of time, as in message set 0, 10, 20, 30, 40, 50 and message set 60. These points on the graph equate to the Reference Message. Again, the standard deviation is at its maximum when two messages are to be transmitted in the same instance and the standard deviation is at its minimum value when it is the optimum time to transmit a message.

3.3.3 Is There a Trend?

Examining the results of Example 3 and Example 4 it seems a trend or set of rules are starting to be formed, namely:

1. Develop all possible message sets from the data supplied.
2. Calculate the Mean for each message set.
3. Calculate the Standard Deviation for each message set.

4. If the Mean increases in value sharply to a maximum, then it is likely that two messages have been scheduled to be transmitted at once.
5. The point at which the Standard Deviation is at its maximum it is also likely to be the position where more than one message is scheduled to be transmitted.
6. When the Standard Deviation is at its minimum value, this is the optimum position for transmitting the data message or messages.
7. The Mean and Standard Deviation are cyclical throughout the message set.

These set of rules have only been developed by generating two SMs with two messages each. A third test set will be generated using the above rules for Example 5 in order to determine if the rules hold true.

Example 5: Two CAN standard messages with periods of 20ms and 25ms respectively each operating on a bus with a baud-rate of 125kbits/s and each message has 7 data bytes. Message M1 is the Reference message with system data within the message and M2 is a normal data message in the TTCAN network.

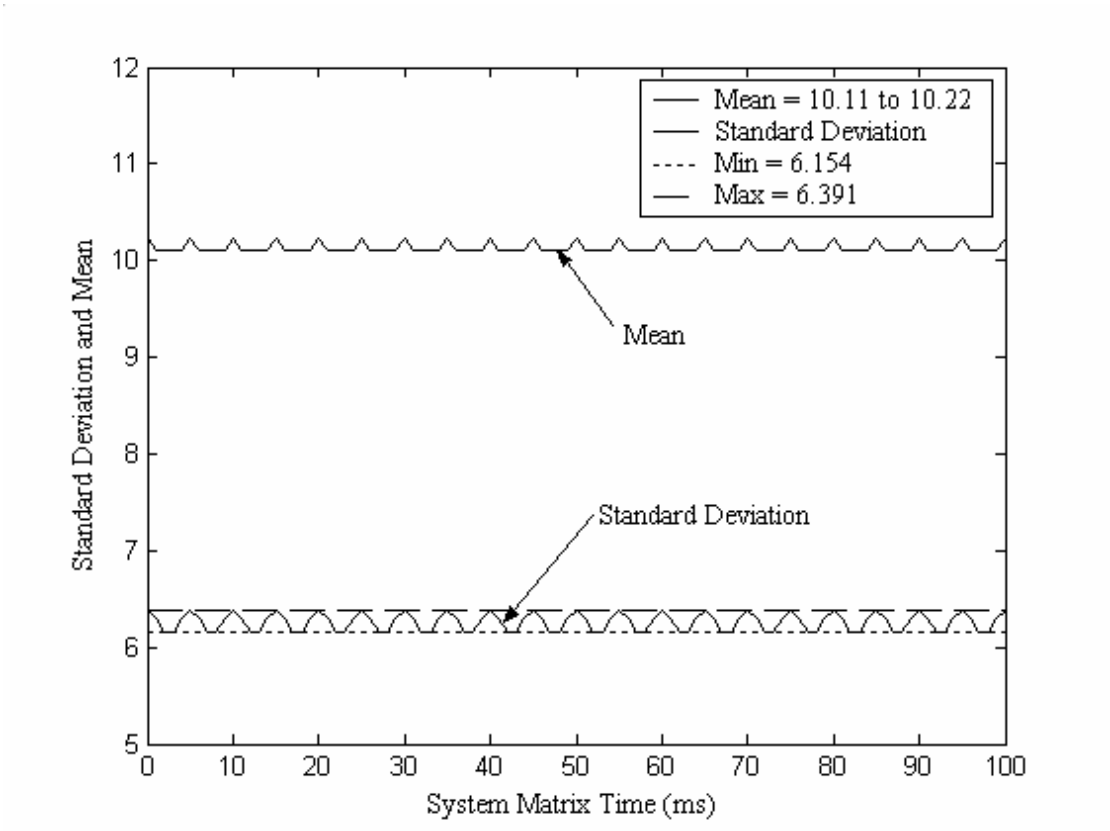


Figure 3.14: Graph Developed with MATLAB for Example 5

Appendix 3 shows all developed message sets from the data in Example 5. This data was developed in the same manner as for Examples 3 and 4. It appears to exhibit the same traits with the change in Mean from 10.11 to 10.22. The standard deviation's maximum value coincides with the maximum value of the mean. This should be the position where two messages could be transmitted at once if the message set was implemented. The minimum value of the standard deviation is flat, for example, between 2ms and 3ms. It was felt that this problem occurred due to the time slots been at 1ms intervals (Figure 3.14).

Part of the graph was redrawn with intervals of 0.5ms (Figure 3.15), but only covers the first 10ms of the message sets. This clearly shows that the optimum time for message M2 is to start at 2.5ms or 7.5ms and that maximum mean and maximum standard deviation occur at 5ms. All the rules that were stated at the beginning of this section hold true for Example 5.

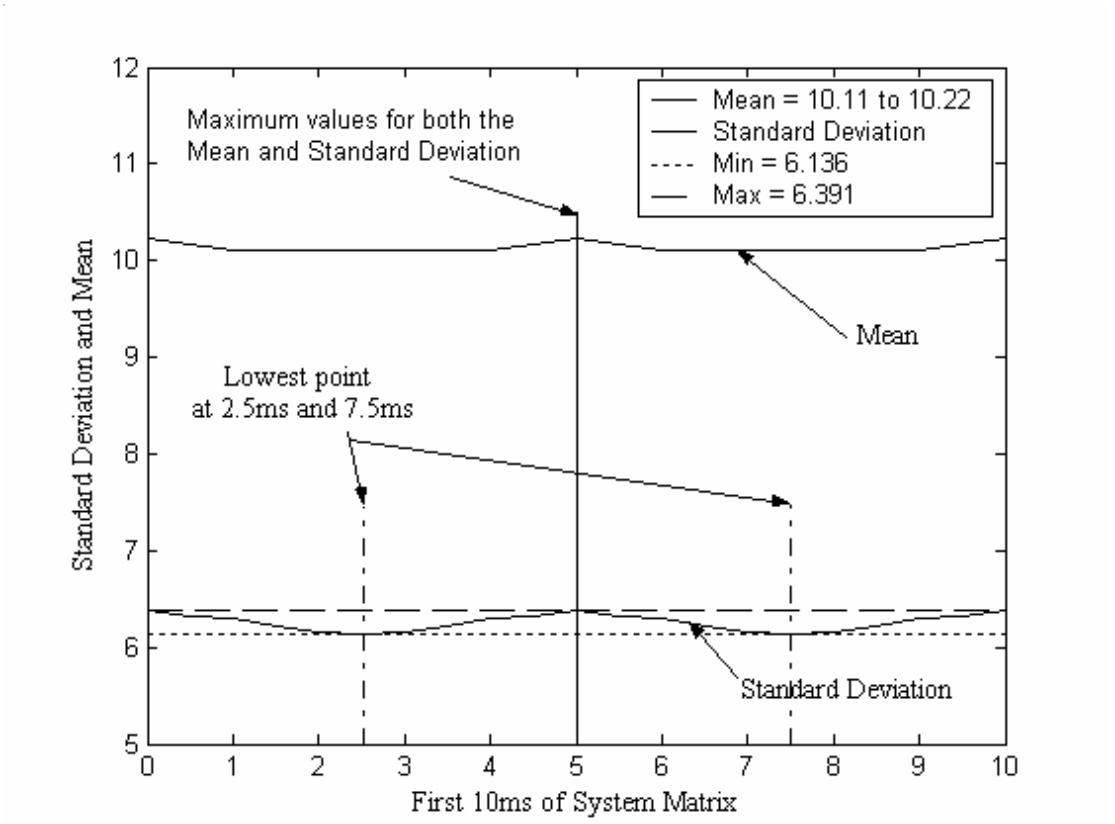


Figure 3.15: Graph for Example 5 over the first 5ms

3.4 Statistical Software Scheduler Development

3.4.1 Introduction

This section will seek to find a software solution to the Statistical Scheduler. As can be seen in Appendix 3, even with just two messages in the SM, there was a possible 100 different message sets available. The 100 message sets were developed by the use of Microsoft Excel and imported into MATLAB in order to graph these results. This was extremely time consuming. The number of possible message sets to be developed is dependant on two factors; the LCM of the message periods and the number of actual messages in the SM.

3.4.2 Software Design

For the software to be useable it must be able to complete the following steps:

1. Allow user input of data.
2. Develop all possible message sets from the data supplied.
3. Calculate the mean for each message set.
4. Calculate the standard deviation for each message set.
5. Find the maximum standard deviation of all developed message sets.
6. Display which message set exhibits the maximum standard deviation.
7. Find the minimum standard deviation of all developed message sets.
8. Display which message set exhibits the minimum standard deviation.

3.4.2.1 Programming Language

Several programming languages were investigated in order to develop the Statistical Scheduler. Amongst these were Microsoft VB 2005, Visual C# 2005, Visual C++ 2005, and Sun Micro Systems Java.

It was decided to use Microsoft VB 2005 Express Edition as it is a free tool which offers an easy to learn language [41-43] and enables the Rapid Application Development (RAD) of Graphical User Interface (GUI) applications.

3.4.2.2 Number of Message Sets to Be Developed

As shown in Example 4, 60 message sets were developed (Appendix 2). This was in spite of the Reference message having a time period of 20ms. As was evident in Examples 3, 4, and 5, there is symmetry to the data values and graphs. Taking the examples above, the same answers could have resulted, had the mean and standard

deviation only been found from one Reference message to the next in the SM. In Example 4, this would mean building only the first 20 message sets and therefore, a reduction of 67% in the workload.

Example A: If a TTCAN SM has to be developed with a Reference message M1 and two data messages M2 and M3 with periods of 20ms, 30ms, and 40ms respectively.

i) Calculate the number of message sets possible, using the full SM.

ii) Calculate the number of message sets possible, using the time frame from one Reference message to the next.

First, find the LCM.

$$SM_{size} = LCM$$

$$LCM\ \{20, 30, 40\} = 120$$

Since the Reference message is always set at zero time it has no effect on the combinations of possible message sets, but the data messages have, therefore:

a) Take the number of periodic messages, in this example 3 and subtract 1:

$$n = Messages_{total} - 1$$

$$3 - 1 = 2$$

b) Possible number of different message sets in a given SM, in this case 120, it can be found by:

$$Q_{Message\_sets} = SM^n$$

$$14400 = 120^2$$

c) If , as stated above, we use only span from one Reference message to the next we get:

$$Q_{Message\_sets} = M^n$$

$$400 = 20^2$$

d) Percentage saving on processing time:

$$97.2\% = (1 - \frac{400}{14400}) * 100$$

It can be seen form Example A that if the full SM is used, 14,400 message sets will be built and evaluated, whereas if the period between Reference messages is used, only 400 possible message sets are available. By using only the period times between Reference messages, the processing time can be reduced by over 97% for the example above.

3.4.2.3 Software Flow Chart

Figure 3.16 shows the flow chart that was developed prior to writing of any code. It shows the sequence of events that are required to produce the optimised schedule.

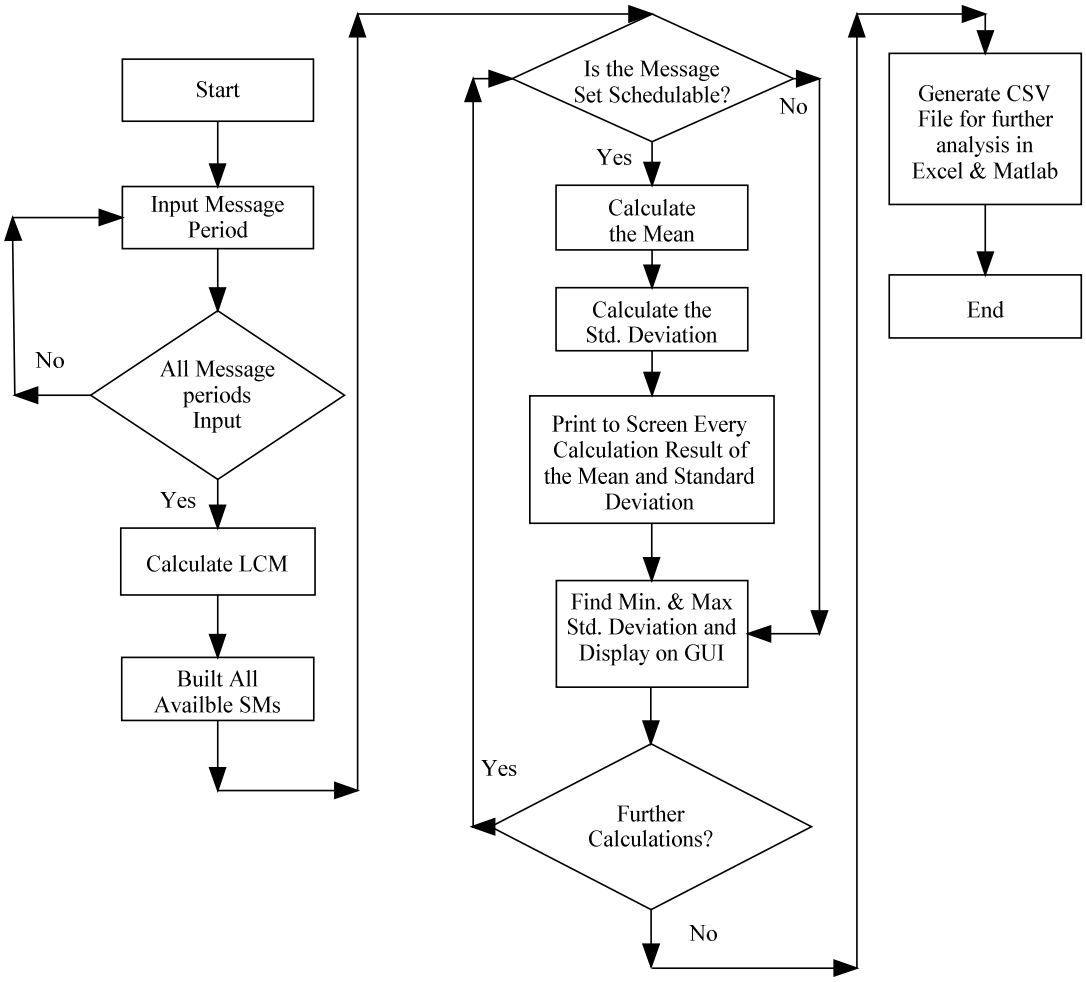


Figure 3.16: Flow Chart for Development and Evaluation of TTCAN Message Sets

3.4.2.4 Program Flow of the Statistical Scheduler

Once the user executes the application software (Figure 3.17), the message periods have to be input into the application.

When all message periods are entered including the Reference message (Appendix 4, lines 31 to 45), the user presses the button marked “Press to Built System Matrix” (Figure 3.17). The application now sorts the message periods in ascending order, displaying them in the window marked “Message Periods” on the GUI. It proceeds to calculate the LCM of all message periods, as this determines the size of the SM and displays this in the window marked “LCM”.

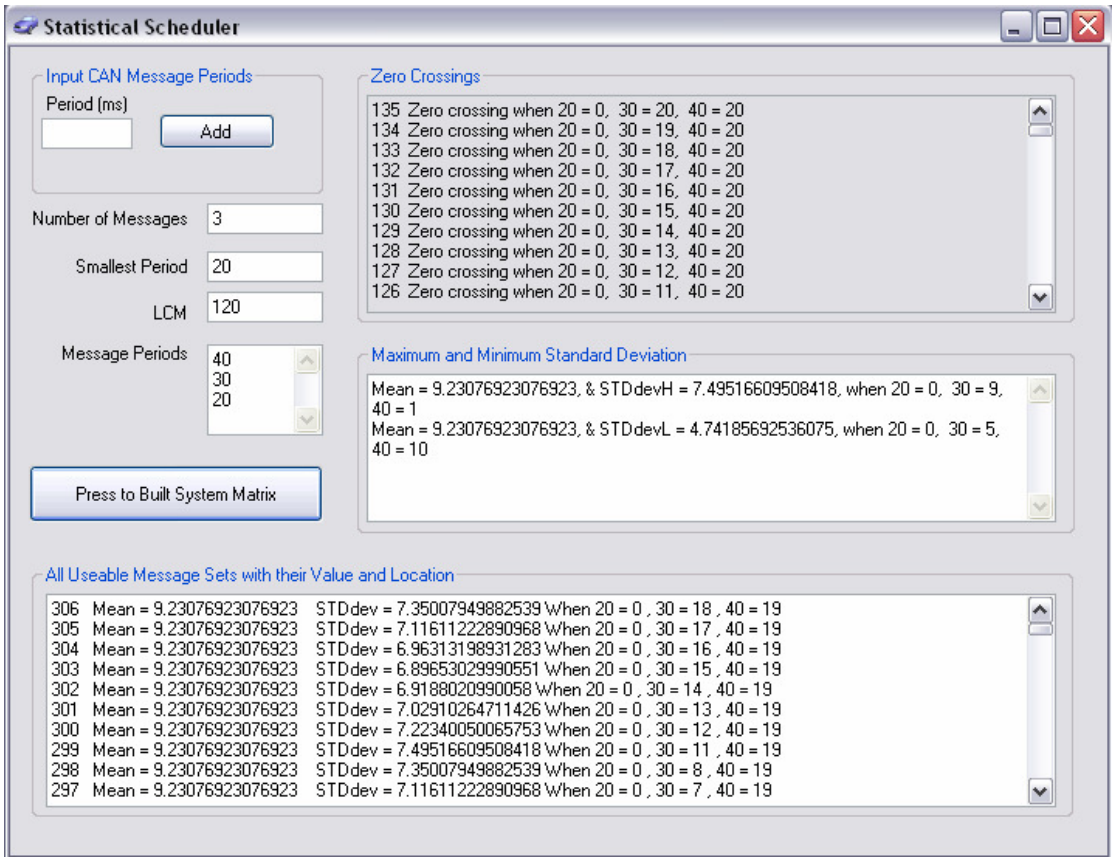


Figure 3.17: Statistical Scheduler

The application next calls a function for building of message sets, but this function is dependant on the number of message periods that the user originally input. This application is only suitable for 2, 3, or 4 different message periods, but could be expanded to accommodate additional message periods.

The scheduler starts the process of building all message sets for the required number of message periods and uses the range from one Reference message to the next Reference message as shown in Figure 3.18. Example B shows the process required to find all possible SMs and develop them.

Example B: Find the first 5 message sets that are schedulable without zero crossing if the Reference message period is 20ms and Data Message A and B have message periods of 30ms and 40ms respectively.

Figure 3.18 shows the layout of the periodic messages. The Reference message will always start at time zero and the next Reference message will be at 20ms. If message A and B both start at zero time, we will have "zero crossing" at time zero; in other words a collision of messages will take place on the TTCAN network.

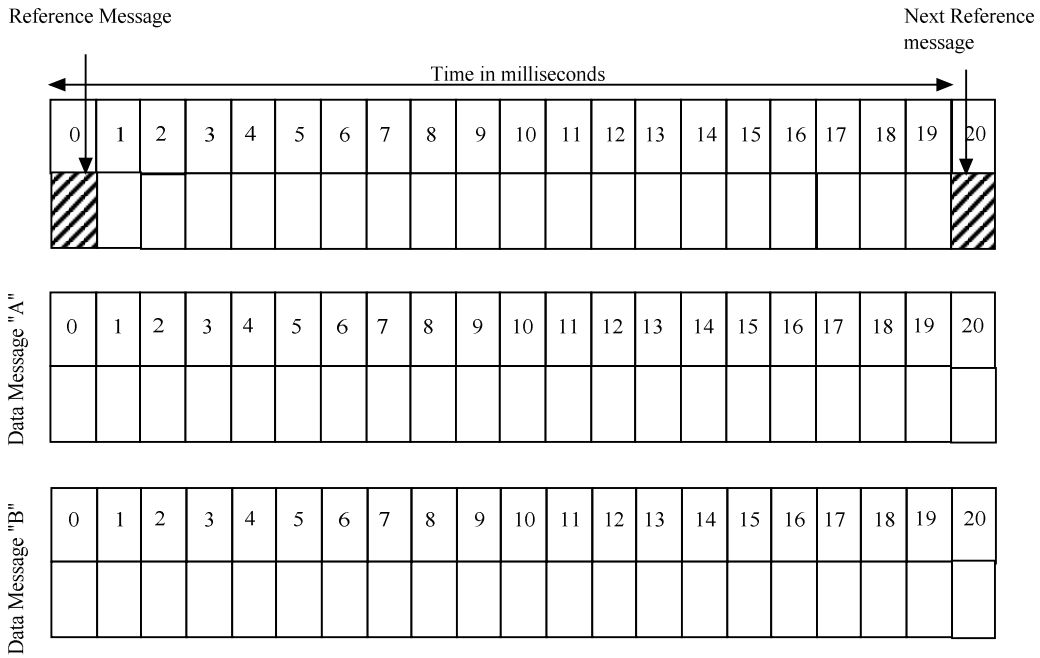


Figure 3.18: Finding All Possible SMs

It is clear that only the Reference message can be sent at time zero and that messages A and B will have to be sent at different times. Taking this in to account the first occasion that a useable message set arises is when the Reference message is sent at time zero, message A is sent at time 1ms and message B is transmitted at 2ms. This is the start position for the three messages and the message set can now be developed as:

Reference Message Transmitting time (ms) = 0, 20, 40, 60, 80, and 100
Message A Transmitting time (ms) = 1, 31, 61 and 91
Message B Transmitting time (ms) = 2, 42, and 82

The actual message schedule for message set 1 is:

$$Message\ set\ 1 = 0_R, 1_A, 2_B, 20_R, 31_A, 40_R, 42_B, 60_R, 61_A, 80_R, 82_B, 91_A, 100_R$$

Where: M_R = Reference Message, M_A = Message A and M_B = Message B.

The messages are structured in the above format within the software application to ensure that zero crossings are recognised (Appendix 4 lines 347 to 464). Message B is now incremented in 1ms intervals to obtain the other four SM. Therefore the other four message sets are:

$$Message\ set\ 2 = 0_R, 1_A, 3_B, 20_R, 31_A, 40_R, 43_B, 60_R, 61_A, 80_R, 83_B, 91_A, 100_R$$
$$Message\ set\ 3 = 0_R, 1_A, 4_B, 20_R, 31_A, 40_R, 44_B, 60_R, 61_A, 80_R, 84_B, 91_A, 100_R$$
$$Message\ set\ 4 = 0_R, 1_A, 5_B, 20_R, 31_A, 40_R, 45_B, 60_R, 61_A, 80_R, 85_B, 91_A, 100_R$$
$$Message\ set\ 5 = 0_R, 1_A, 6_B, 20_R, 31_A, 40_R, 46B, 60_R, 61_A, 80_R, 86_B, 91_A, 100_R$$

If the application finds a zero crossing, the calculating of the mean and standard deviation are not completed with that particular message set, but the data is displayed within the GUI (Appendix 4 lines 462 to 492).
If the message set is useable (no zero crossing) then the application will calculate the mean (Appendix 4 lines 497 to 502), followed by the standard deviation (Appendix 4 lines 503 to 510). The results of these calculations are written to the GUI and displayed in the window called “All usable message sets, etc” (Appendix 4 lines 511 to 515).
The software locates and displays the mean, the maximum and minimum standard deviation, and the start location of the messages in the “Maximum and Minimum Standard Deviation” window. The next stage of the process is to create a “.csv” file that can be used for further data analysis by applications such as Microsoft Excel or MATLAB.

Figure 3.19 shows the results from message periods 20ms and 30ms. The user can see clearly the following data when the application has completed its calculations:

- Number of messages input.
- Smallest message period (normally the Reference message).
- Calculated LCM, which is the length of the SM.
- The message periods for all messages input for calculation.
- It displays which message times will cause a zero crossing.
- Displays the Mean, the maximum and minimum Standard Deviation
- It displays all useable message sets together with their Mean, the maximum and minimum Standard Deviation together with the start times of the message.
- The user can further use the data that is made available on the hard drive of his/her computer in the form of a ".csv" file

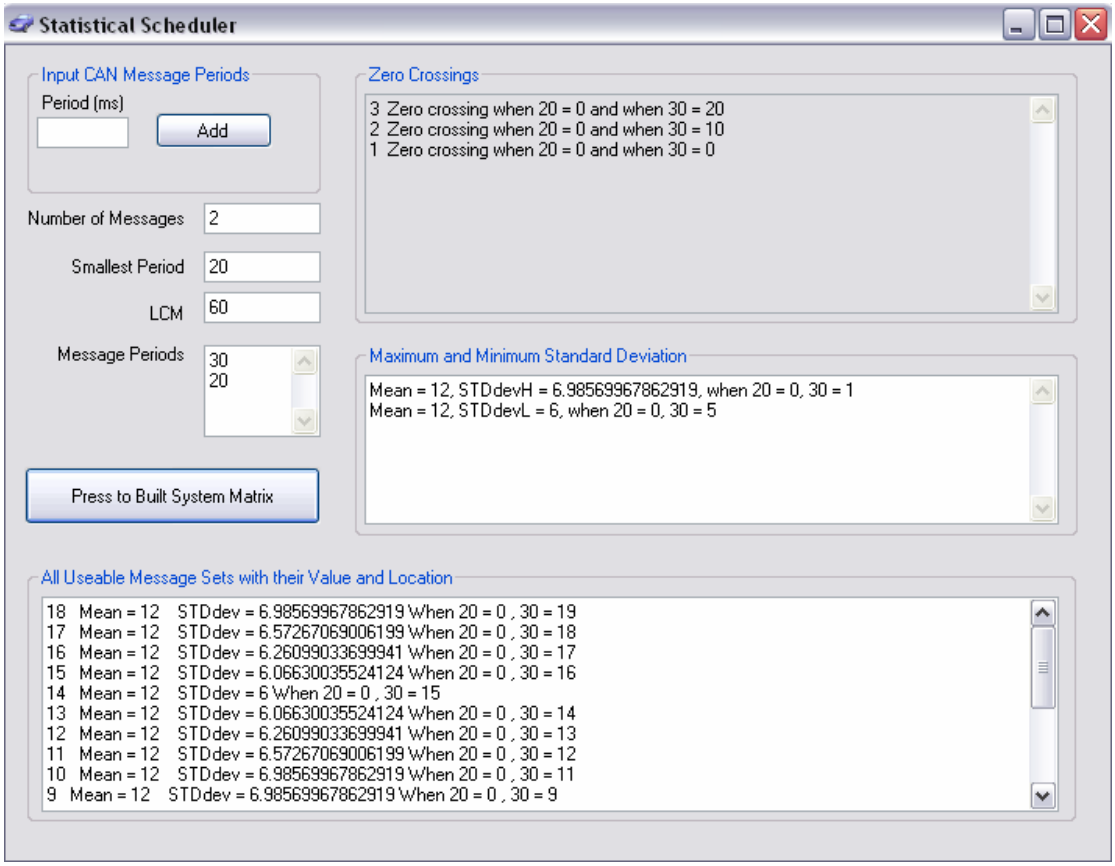


Figure 3.19: Message Schedule for 20ms and 30ms Messages

It can be seen in Figure 3.19, that the maximum standard deviation is 6.99 (correct to two decimal places) and the minimum standard deviation is 6. Using the minimum

standard deviation ensures the largest possible arbitration window size, which provides the optimum real-time performance. If we use the largest standard deviation, there will be two TTCAN messages broadcast consecutively and this will produce the worst real-time performance.

Appendix 5 shows the .csv file, which was generated by the message periods, used in Figure 3.19. This file was imported into MATLAB [40] and used to generate the graph in Figure 3.20. It shows the optimum transmitting point of 5ms for the message with a period of 30ms.

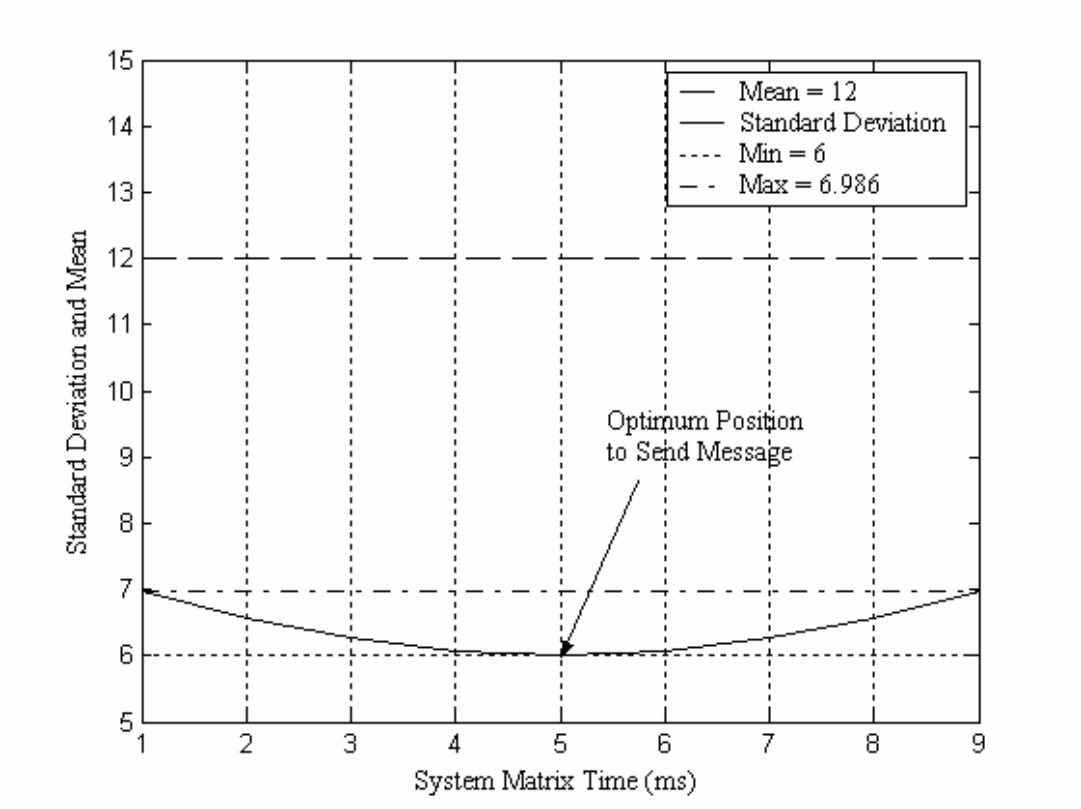


Figure 3.20: Optimum Position for Message Periods 20ms and 30ms

Implementation of the message set can be seen in Figure 3.21, and shows the smallest arbitration window to be 4ms.

If the 30ms message period is decrement from its present position by 1ms, the smallest arbitration window will now be 3ms down from 4ms. If the same message is incremented by 1ms from its present position, it will also leave the message set with the smallest arbitration window of 3ms. Therefore, the present position is optimum.

The Statistical Scheduler also found another start position with the same standard deviation at the 15ms slot. This second position offers the same optimisation and, as stated before, there can be more than one optimum position in a message set.

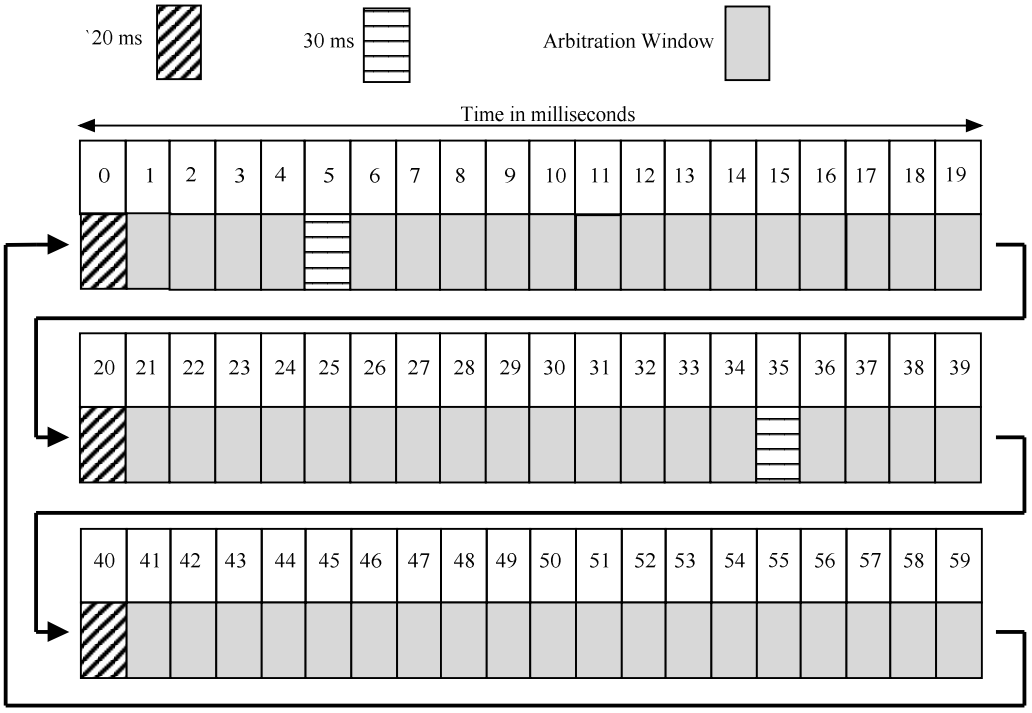


Figure 3.21: Implementation of Figure 3.19

3.5.1 Extended Testing with Three Periodic Messages

Example 6: Three CAN standard messages with periods of 20ms, 30ms and 40ms respectively, operating on a bus with a baud-rate of 125kbits/s and each message has 7 data bytes. Message M1 is the Reference message with system data within the message and M2 and M3 is a normal data messages in the TTCAN network.

Longest message duration:

$$t_{sec} = \frac{Message\_Length_{bits} + Stuffbits_{bits}}{Baud \quad rate_{bits}}$$

$$\frac{100 + 18}{125000} = 0.944ms$$

Length of the SM:

$$LCM\ (20, 30, 40) = 120ms$$

Number of Triggers in SM:

$$Triggers = \sum_{i}^{N} \frac{LCM(M)}{M_i}$$

$$\frac{120}{20} + \frac{120}{30} + \frac{120}{40} = 13$$

Using Equations 3.1 and 3.2 above will not give us a solution to this problem, as the message periods are different. In addition, the number of message sets to be developed will be:

$$Q_{Message\_sets} = M^n$$

$$21^2 = 441$$

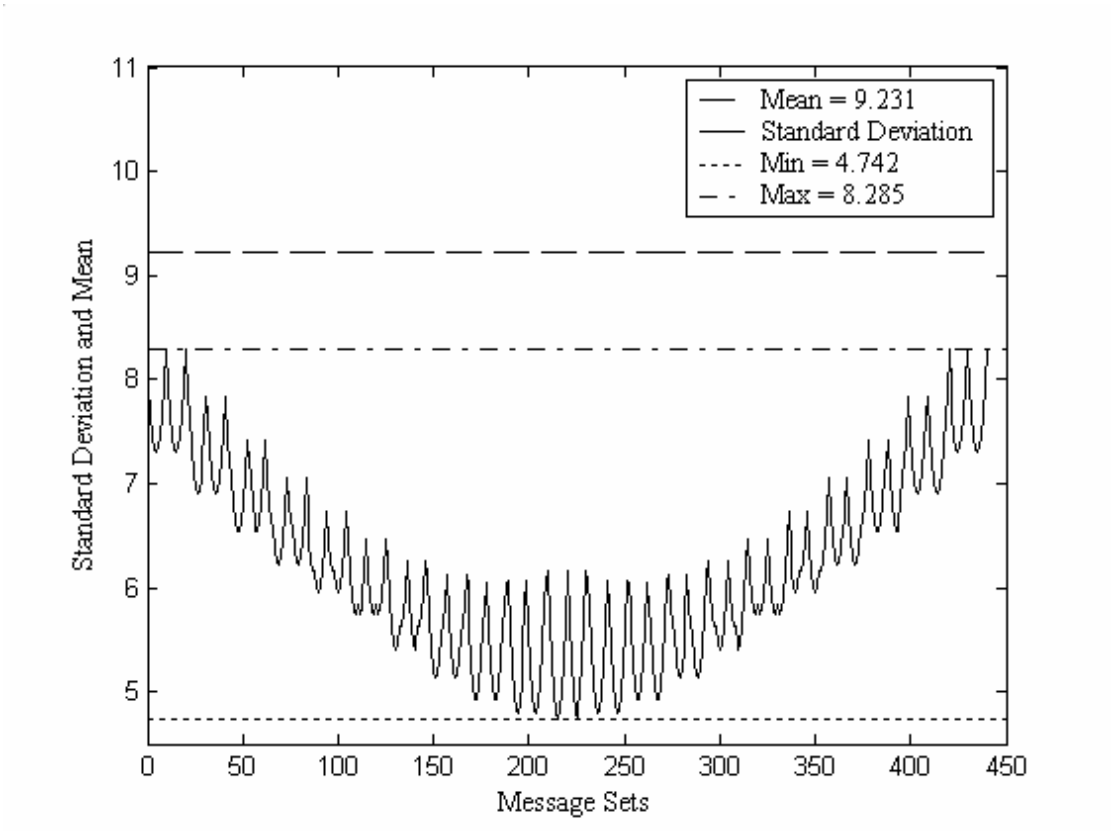


Figure 3.22: Graphed Data for Message Periods 20ms, 30ms, and 40ms

The Statistical Scheduler will be used and the answer analysed by MATLAB. Appendix 6 shows the complete output from the Statistical Scheduler for the message

periods in Example 6. It can be seen that there was 441 message sets developed along with their statistical data. The GUI also output a .csv file, which was manipulated in MATLAB to graph all data as before, and is shown in Figure 3.22. The graph looks somewhat different to the previous graphs, but in this instance, we are using three message periods instead of two.

The graph is symmetrical as before, but differs in that it forms a curve. Although there are 441 iterations, the actual duration in time is from one Reference message until the next which is 21ms including both Reference messages. This gives us a possible 21 different message combinations per millisecond.

Inspecting Figure 3.22, it shows the minimum standard deviation to be approximately the 215 message set to be developed. Figure 3.23, shows clearly that the minimum standard deviation occurred when the start time for the message of period 20ms is 0ms in the SM. The periodic message of 30ms should start at 5ms in the SM, and the periodic message of 40ms should start at 10ms in the SM. This coincides with message set 216 in the "All Usable Message sets……." of the GUI.

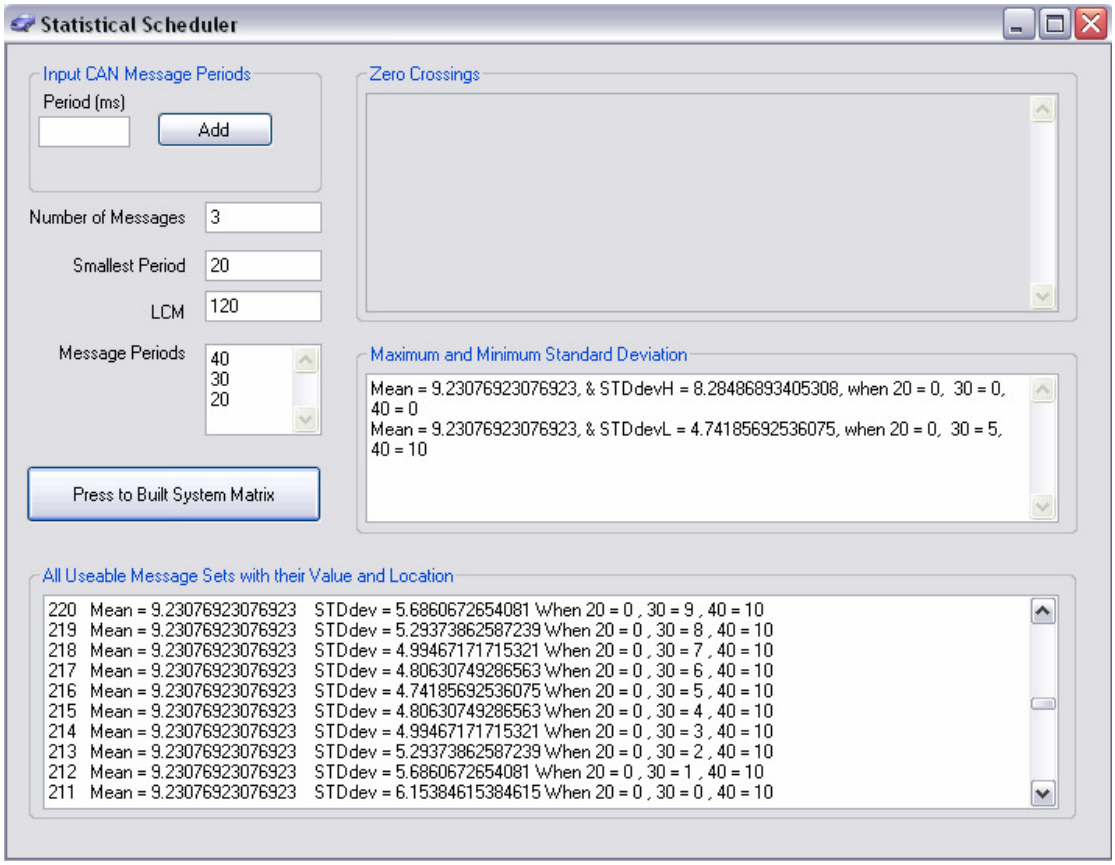


Figure 3.23: GUI Output for Message Periods 20ms, 30ms, and 40ms

Example 7: Four CAN standard messages with periods of 20ms, 30ms, 40ms, and 50ms respectively each, operating on a bus with a baud-rate of 125kbits/s and each message has 7 data bytes. Message M1 is the Reference message with system data within the message and M2, M3, and M4 are normal data messages in the TTCAN network.

Longest message duration:

$$t_{sec} = \frac{Message\_Length_{bits} + Stuffbits_{bits}}{Baud \quad rate_{bits}}$$

$$\frac{100 + 18}{125000} = 0.944ms$$

Length of the SM:

$$LCM\ (20,\ 30,\ 40,\ 50) = 600ms$$

Number of Triggers in SM:

$$Triggers = \sum_{i}^{N} \frac{LCM(M)}{M_i}$$

$$\frac{600}{20} + \frac{600}{30} + \frac{600}{40} = \frac{600}{50} = 77$$

Using Equations 3.1 and 3.2 above will not give us a solution to this problem, as the message periods are different. Also the number of message sets to be developed will be:

$$Q_{Message\_sets} = M^n$$

$$9261 = 21^3$$

The Statistical Scheduler was used to develop all message sets for the message periods stated in Example 7. The output data from the scheduler was again used in MATLAB to construct the graph in Figure 3.24.

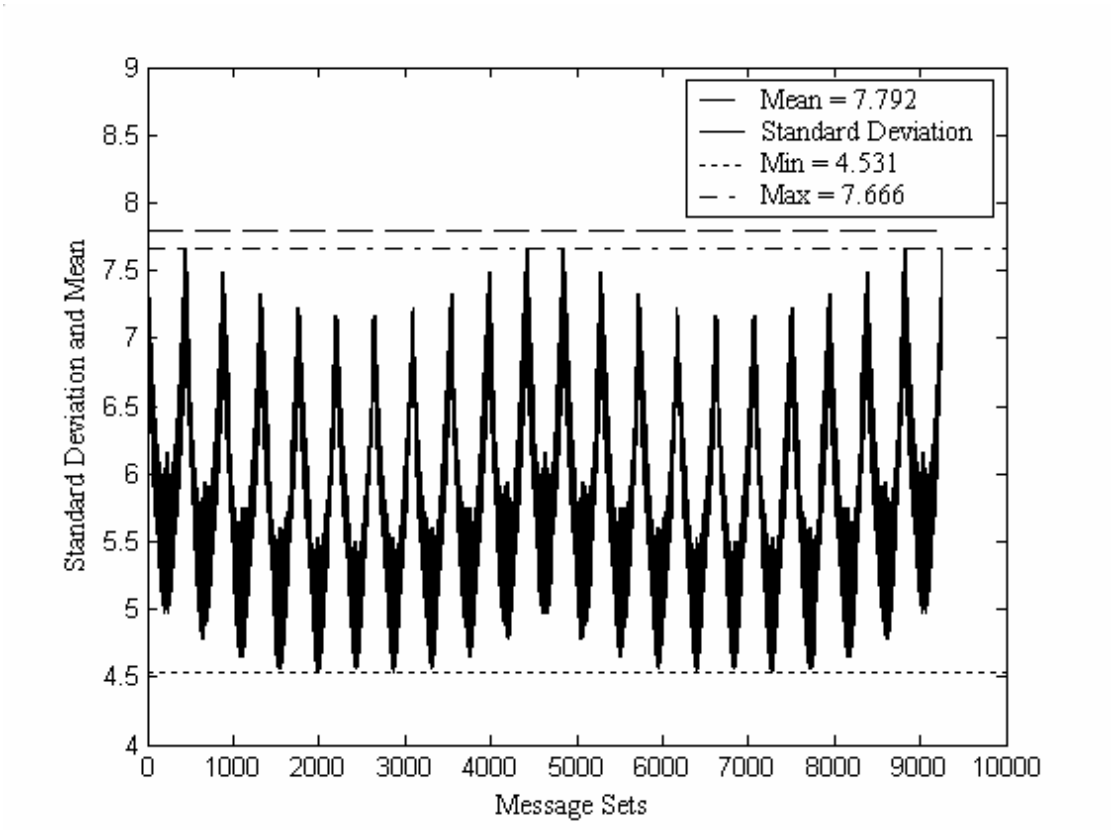


Figure 3.24: Graphed Data for Message Periods 20ms, 30ms, and 40ms

The graph is again symmetrical with minimum and maximum points of standard deviation. The minimum standard deviation occurs in two places on the graph. The first at next message set 2000 and again near message set 6400. The optimum message set was at calculated as message set 1981. The optimum start position in the SM for the for the periodic message of 20ms is 0ms, the start point of the periodic message of 30ms is at 6ms in the SM, the message of period 40ms should start at 10ms in the SM, and the periodic message of 50ms message duration should begin at 4ms in the SM (Figure 3.25).

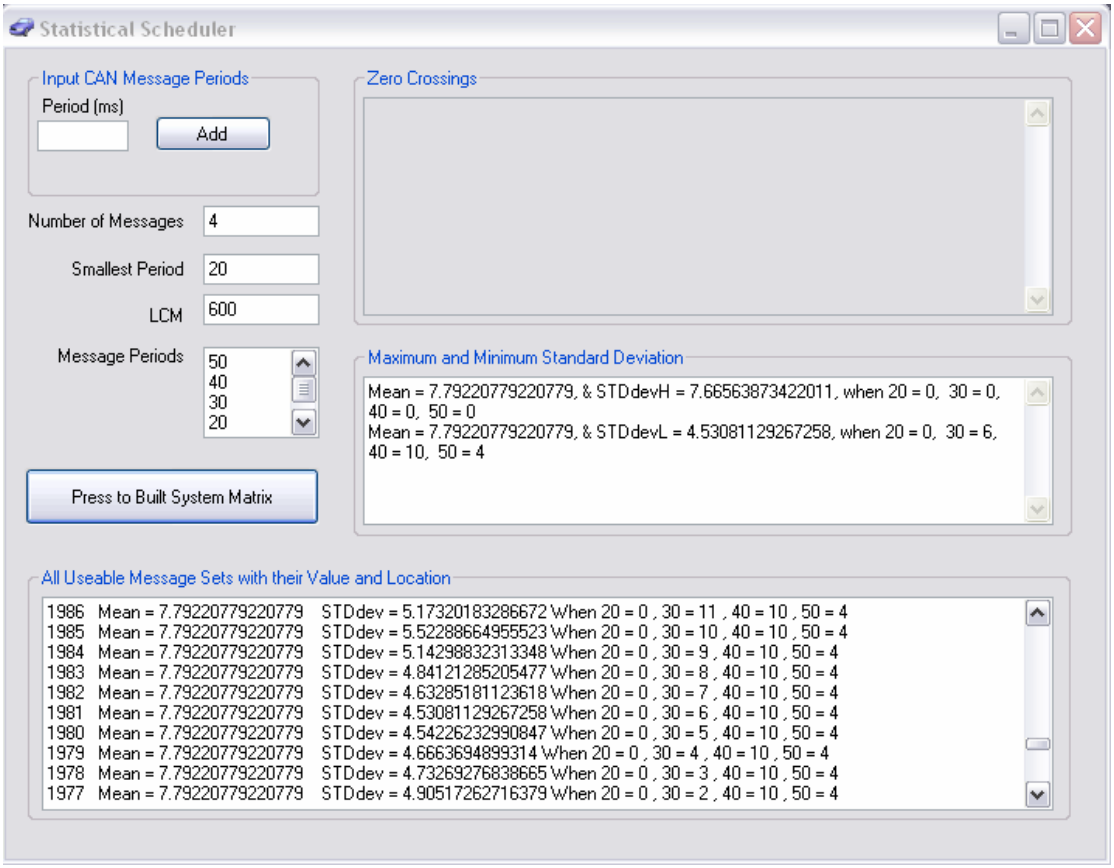


Figure 3.25: GUI Output for Message Periods 20ms, 30ms, 40ms, and 50ms

The output of message sets from the Statistical Scheduler for the message periods in Example 7 were not included in the Appendix, as they would fill in excess of 300 pages.

3.6. Summary

This chapter has examined the drawbacks associated with both stochastic and heuristic schedulers. These problems mainly stem from either trying to make the best schedule by virtue of using the best probability as with the stochastic scheduler or by endeavouring to find the best solution by trial and error in the case of heuristic scheduler. With both types of scheduling, there is no way of checking the optimisation of two different usable message sets.

Surely with today's computing power there must be a method of building all message sets and then calculating the optimised in relation to arbitration window size, which has a direct effect on real-time performance.

This thesis has shown that there is a solution available for the building of message sets from any group of periodic messages. The developed software is capable of excluding all message sets that have zero crossing. If these message sets were to be included in the final result, they would be a cause of jitter for the TTCAN network.
Finally, the Statistical Scheduler can find the message set with the largest arbitration windows by the use of standard deviation. It can therefore guarantee the optimum spontaneous response from any set of periodic messages used and therefore improve the real-time performance of the TTCAN network.

Chapter 4: Implementation and Testing

4.1 Introduction

This chapter seeks to confirm that the Statistical Scheduler introduced in the previous chapter can develop a TTCAN network that will return the optimum results when implemented in hardware. Testing was performed on a physical TTCAN network to prove that the actual optimisation levels predicted by the Statistical Scheduler can be actually attained once invoked in hardware.

The chapter is laid out in the following sections:

- Section 4.2 deals with the hardware implementation of the system, which is built around four CAN nodes
- Section 4.3 illustrates the procedure involved in writing the embedded C code for the TTCAN network.
- Section 4.4 details the testing procedure including the test results.

4.2.1 Hardware Implementation

Hardware implementation was divided into two areas:

- Physical Interface
- TTCAN nodes

4.2.1.1 Physical Interface

A twisted pair of cables is the normal medium used for the connection of nodes in a CAN network. Cable propagation delay and skew are factors that effect the operation of this physical medium.

Propagation delay is the time taken for a signal to travel the length of a cable. Large propagation delays lead to bus errors, which ultimately cause malfunctions within the network system.

Prior to building the network, it was decided to test the wire being used in the network to ensure that it conformed to the CAN standard ISO 11898-2. Figure 4.1 shows the inherent propagation delay in the oscilloscope and leads when tested, which was 10ns.

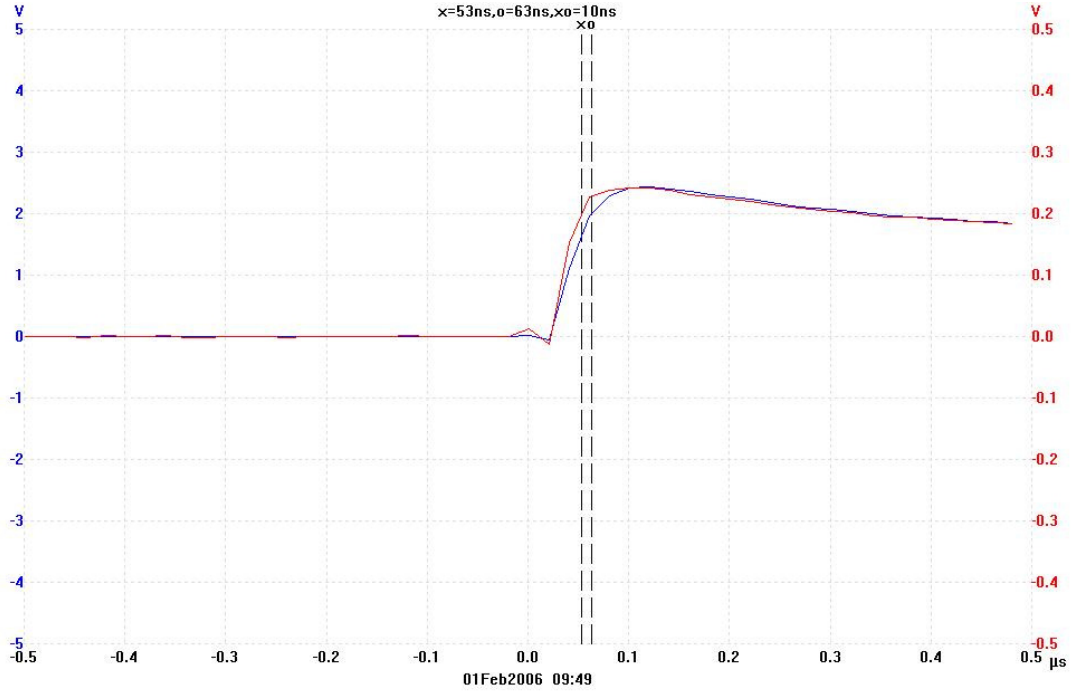


Figure 4.1: Propagation Delay of Oscilloscope Channel 1 and Channel 2 Leads

Figure 4.2 shows the result from a piece of CAN cable of length 1.5m being tested for propagation delay. The delay shown is 16 ns, of which 10 ns is due to the equipment leads, as shown in Figure 4.1.

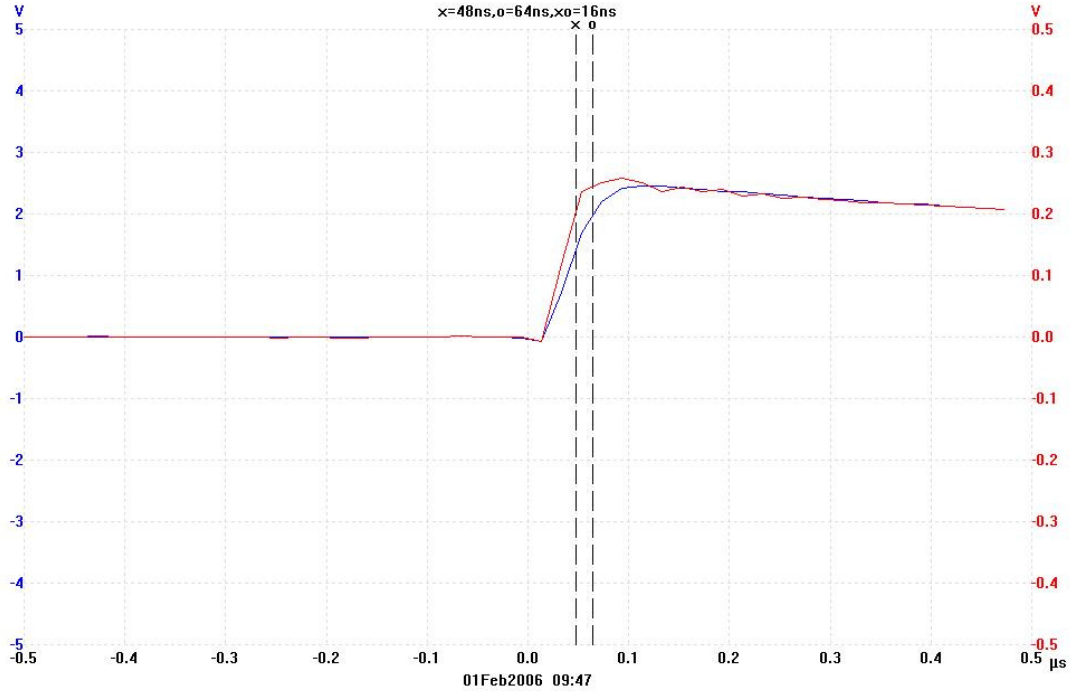


Figure 4.2: Propagation Delay for 1.5m of CAN Cable.

This leaves an actual propagation delay per metre of:

$$\frac{16-10}{1.5}=4ns/m$$

ISO 11898 demands a propagation delay < 5 ns/m; therefore, the cable tested was within the specified value.

Skew is the difference in delay between the data signal travelling along a pair of wires. In the case of the CAN network this is the difference between the CAN bit arriving at the receiver on CAN_H and CAN_L. A large skew reading indicates a considerable delay between the data arriving on CAN_High and CAN_Low and this can lead to bus errors.

Skew delay should be zero, but as seen in Figure 4.3, the skew is 19ns. This is made up of the actual 19 ns shown on the oscilloscope but minus the inherent propagation delays in the equipment, which is 10 ns.

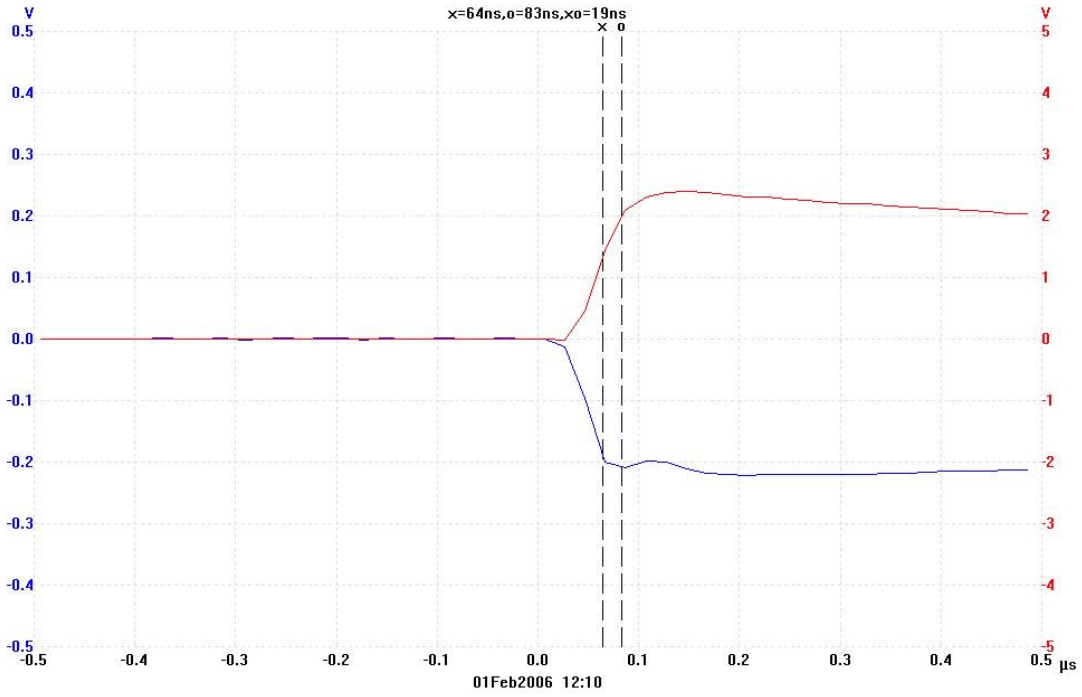


Figure 4.3: Testing Network Cable Skew

This leaves an actual skew:

$$19-10 = 9ns$$

The skew of 9ns was due to different lengths of cable for CAN_H and CAN_L. All cables used in the TTCAN network were adjusted so that the skew result was always zero.

4.2.1.2 Embedded Tool Chain

It was decided to develop CAN nodes using the minimum number of components possible so that interfacing problems would be minimised. The next stage of the process was to decide whether to use an 8 or 16-bit platform. Other than the fact that a 16-bit device could handle a Standard CAN identifier in one operation of the CPU there is no advantage to using such a device for testing in this research. It was therefore decided to use an 8-bit device. Atmel, Freescale, and Microchip all produce 8-bit devices. Further research was carried out in order to determine the most suitable device for the project. It was eventually decided to use Microchip as they had the Microchip PIC18F2480, which included a CAN v2.0B interface and could be interfaced to their MCP2551 CAN transceiver.

The development tool chain was investigated next and three manufacturers were identified: Microchip, Custom Computer Services and MikroElektronika. The decision was made to use the MikroElektronika development environment for the following reasons:

- They offered a fully featured development board.
- The development board included a USB on board programmer
- A CAN daughter board was available.
- The compiler provided was ANSI C compatible.
- Cost

4.2.1.3 Equations for Propagation Delay and Oscillator Tolerance

To ensure effective communication, the minimum requirement for a CAN network is that two nodes, each at opposite ends of the network with the largest propagation delay between them, and each having a CAN system clock frequency at the opposite limits of the specified frequency tolerance, must be able to correctly receive and decode every message transmitted on the network. This requires that all nodes sample the correct value for each bit [15, 44].

The minimum time for the propagation delay segment to ensure correct sampling of bit values is given by:

$$t_{PROP\_SEG} = t_{Prop\ (A,\ B)} + t_{Prop\ (B,\ A)} \tag{4.1}$$

Where nodes A and B are at opposite ends of the network, i.e. the propagation delay is a maximum between nodes A and B.

$$t_{PROP\_SEG} = 2(t_{Bus} + t_{Tx} + t_{Rx}) \tag{4.2}$$

Where t_{Bus} is the propagation delay of the signal along the longest length of the bus between two nodes and t_{Tx} is the propagation delay of the transmitter part of the physical interface and t_{Rx} is the propagation delay of the receiver part of the physical interface. If the propagation delay of the transmitters and receivers on a network is not uniform, the maximum delay values should be used in the equation.

The minimum number of Time Quanta that must be allocated to the PROP_SEG segment is therefore:

$$PROP\_SEG = Round\_UP\ (\frac{t_{PROP\_SEG}}{t_Q}) \tag{4.3}$$

Where the function ROUND_UP( ) returns the argument rounded up to the next integer value [45].

The oscillators in a CAN network will drift due to a change of temperature, a change in voltage, age, etc. This will cause the oscillators at each node to operate at slightly different frequencies. In the absence of bus errors due to, for example electrical disturbances, bit stuffing guarantees a maximum of 10-bit periods between re-synchronisation edges (5 dominant bits followed by 5 recessive bits will then be followed by a dominant bit). This represents the worst-case condition for the accumulation of phase error during normal communication. The accumulated phase error must be compensated for by re-synchronisation following the recessive to dominant edge and, therefore, the accumulated phase error must be less than the programmed Re-synchronisation Jump Width (t_{RJW}).

The accumulated phase error is due to the tolerance in the CAN system clock, and this requirement can be expressed as:

$$(2 * \Delta f) * 10 * t_{NBT} < t_{RJW} \tag{4.4}$$

However, real systems must operate in the presence of electrical noise which may induce errors on the CAN bus. In the event of an error being detected, an Error Flag is transmitted on the bus. In the case of a local error, only the node that detects the error will transmit the Error Flag and all other nodes receive the Error Flag and then transmit their own Error Flags as an echo. If the error is global, all nodes will detect it within the same bit time and will, therefore, transmit Error Flags simultaneously. A node can, therefore, differentiate between a local error and a global error by detecting whether there is an echo after its Error Flag. This requires that a node can correctly sample the first bit after transmitting its Error Flag.

An Error Flag from an Error Active node consists of 6 dominant bits, and there could be up to 6 dominant bits before the Error Flag, if, for example, the error was a stuff error. A node must, therefore, correctly sample the 13th bit after the last re-synchronisation [45]. This can be expressed as:

$$(2 * \Delta f) * (13 * t_{NBT} - t_{PHASE\_SEG2}) < MIN\,(t_{PHASE\_SEG1,\ tPHASE\_SEG2}) \tag{4.5}$$

where the function *MIN ( )* returns the smaller of the two arguments. Thus, there are two clock tolerance requirements, which must be satisfied. It should be noted that at high bit rates (small Nominal Bit Time), the CAN clock tolerance is specified over a relatively short time: 10^{th} t_{NBT} in the case of Equation 4.4, and 13^{th} t_{NBT} in the case of Equation 4.5. This is important for systems that derive the CAN clock from a Phase Locked Loop circuit for which the relative accuracy decreases over short time periods due to output jitter.

The selection of bit timing values involves consideration of various fundamental system parameters [46]. The requirement of the PROP_SEG value imposes a trade-off between the maximum achievable bit rate and the maximum propagation delay, due to the bus length and the characteristics of the bus driver circuit. The maximum achievable bit rate is also influenced by the tolerance of the CAN clock source. The highest bit rate can only be achieved with a short bus length, a fast bus driver circuit and a high-frequency high-tolerance CAN clock source. In many systems, the bus

length will be the least variable system parameter, which will impose the fundamental limit on bit rate. However, the actual bit rate chosen may involve a trade-off with other system constraints, such as cost [45].

4.2.1.4 Calculation of Bit Timing and Oscillator Tolerance

The following calculations relate to the bit segments required for the TTCAN network being developed in this thesis [44].

The following are the constraints of the proposed system:

Bit rate = 125k bit per second

Bus length = 4m

Bus propagation delay = 5 x 10^{-9} sm^{-1}

Physical Interface (MCP2551) transmitter plus receiver propagation delay = 150ns

MCU oscillator frequency = 8MHz

Step 1: Physical delay of bus = 4 x 5 x 10^{-9} = 20ns

$$t_{PROP\_SEG} = 2(20ns + 150ns) = 340ns$$

Step 2: A prescaler value of 8 gives a CAN system clock of 1MHz and a TQ of 1000ns.

$$8000 / 1000 = 8 \text{ time TQ bit.}$$

Step 3:

$$PROP\_SEG = Round\_UP\left(\frac{340ns}{1000ns}\right) \Rightarrow 1$$

Step 4: From 8 TQ per bit, subtract 1 for PROP_SEG and 1 for SYNC_SEG. This leaves 6 TQ, so PHASE_SEG1 = 3 and PHASE_SEG2 = 3.

Step 5: RJW is the smallest of 4 and PHASE_SEG1, so RJW will be 3.

Step 6:

$$\Delta f = \frac{RJW}{20 * NBT} = \frac{3}{20 * 8} = 0.01875$$

$$\Delta f < \frac{M(PHASE\_SEG1, PHASE\_SEG2)}{2(13 * NBT \quad PHASE\_SEG2)} \Rightarrow \frac{3}{2(13*8-3)} = 0.01485$$

The required oscillator tolerance is the smaller of these values, i.e. 0.01485 (1.485%).
In summary:

Prescaler = 8

Nominal Bit Time = 8

PROP_SEG = 1

PHASE_SEG1 = 3

PHASE_SEG2 = 3

RJW = 3

Oscillator tolerance = 1.485%

4.2.1.5 Node Implementation

Four nodes were constructed on veroboard, using the Microchip PIC18F2480 microcontroller with on-board CAN. The CAN transceivers were the Microchip MCP2551 and the oscillators conformed to the calculations made in section 4.2.1.5. Other items to complete the node included a voltage regulator and smoothing capacitors. All components were wire wrapped, as required. A diagram of the complete node can be viewed in Appendix 7.

4.3.1 Embedded Software Development

The embedded software for the testing was developed using a version of MikroElektronika 'C', call MikroC [47]. A software flow chart for the Reference Message node is shown in Figure 4.4.

The program starts by loading the timer registers with values that will cause an interrupt at 20ms (Appendix 8 lines 20 and 21). The prescaler for Timer0 and 16-bit mode are executed (Appendix 8 line 22). All CAN registers are set to the values as calculated in section 4.2.1.5 (Appendix 8 lines 34 to 43 inclusive).

Following this, the CAN is set to "normal mode" (Appendix 8 line 44). All data bytes of the CAN message are loaded with default values and the ID is given to the Reference message, as is its DLC (Appendix 8 lines 45 to 52). Timer0 interrupt is set to expire at 20ms (Appendix 8 line 53).

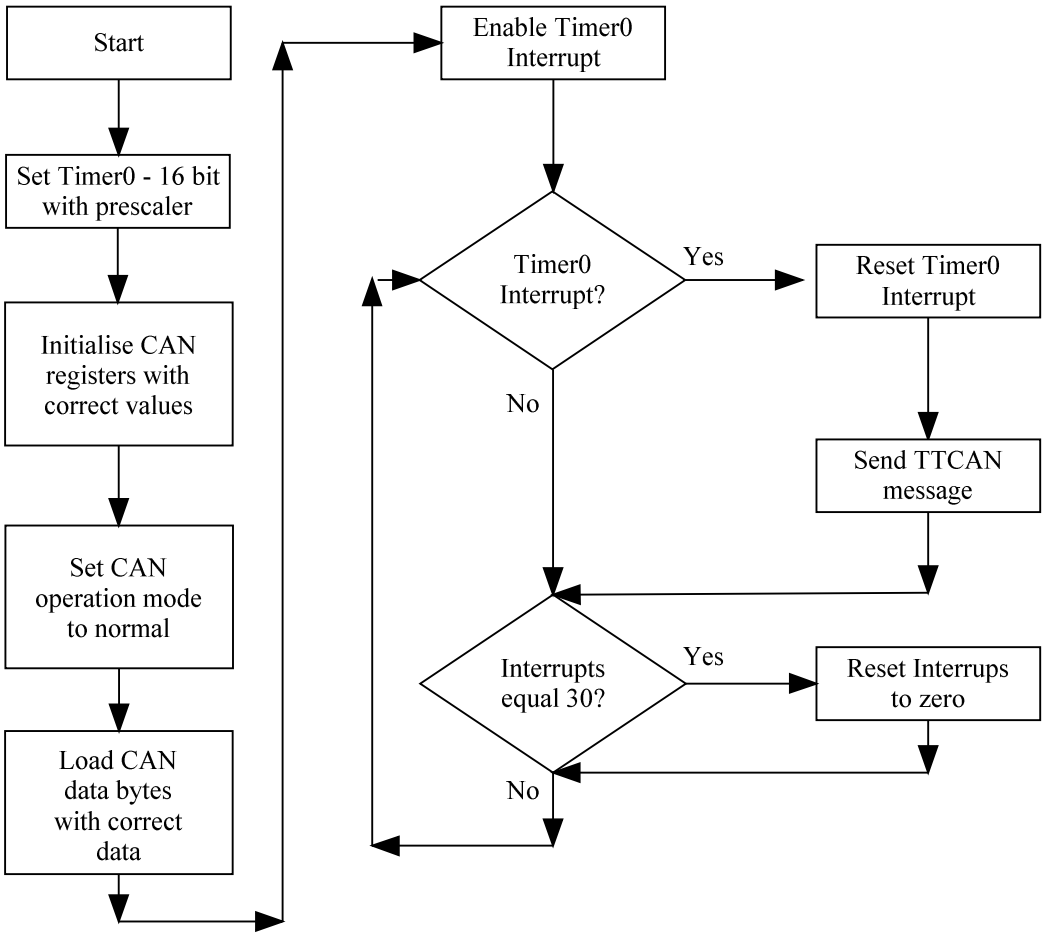


Figure 4.4: Embedded Software Flow Chart

In Appendix 8, lines 54 to 68 is the main loop where the system waits for the interrupt from Timer0 and checks to see if the number of interrupts exceeds 30, as this is the number of interrupts that occur in the SM of 600ms.

The program shown in Appendix 8 is one of nine such programs that were developed while testing the various message sets.

4.4.1 Testing Procedure

Embedded software was written so that three different message sets could be tested, namely:

- A TTCAN network with two nodes with the Reference message period was set at 20ms and another message was used with a period of 30ms. Both messages were capable of carrying 7 bytes of data.

- A TTCAN network with three nodes, the Reference message with period 20ms, and two other messages with periods of 30ms and 40ms. All messages were capable of carrying 7 bytes of data.
- A TTCAN network with four nodes with the Reference message period of 20ms and three other messages with periods of 30ms, 40ms and 50ms respectfully. All messages are capable of carrying 7 bytes of data.

4.4.1.1 Data Acquisition

CAN data acquisition was undertaken by use of CANalyzer. This tool is the automotive research and design preferred tool for analysis of any CAN network [48]. The tool was set to 'listen only' mode so it would not interfere with the TTCAN network during data collection.

Figure 4.5 shows the layout of the GUI CANalyzer. It has five windows which are used for interpreting all data on the CAN network.

The "Write" window displays statistical data at the end of a testing session that shows the average message periods, with their minimum and maximum transmission periods and the standard deviation from the mean. It also exhibits the number of messages processed during the test, together with the start and stop time. This data can be saved directly to a ".txt" file.

The "Statistics" window allows the user to check at a glance the message rates on the bus for each individual message in the form of a bar graph. Both transmitted and received messages are displayed in the same window.

The "Graphics" window displays individual messages rates against time. In Figure 4.5 it is used to collect data about error messages. It is set to give a cumulative response over time.

The "Bus Statistics" window provides a real-time view of the CAN network. It gives information relating to the actual load and peak load on the bus, total number of CAN frames on the network, number of error frames generated, etc.

The "Trace" window presents streamed information about message activity on the network. Individual components of the CAN message are displayed separately and all messages are time stamped in real-time. The output from this window cannot be directly saved to a file, it can only be "cut and pasted" into another document as a ".txt" file.

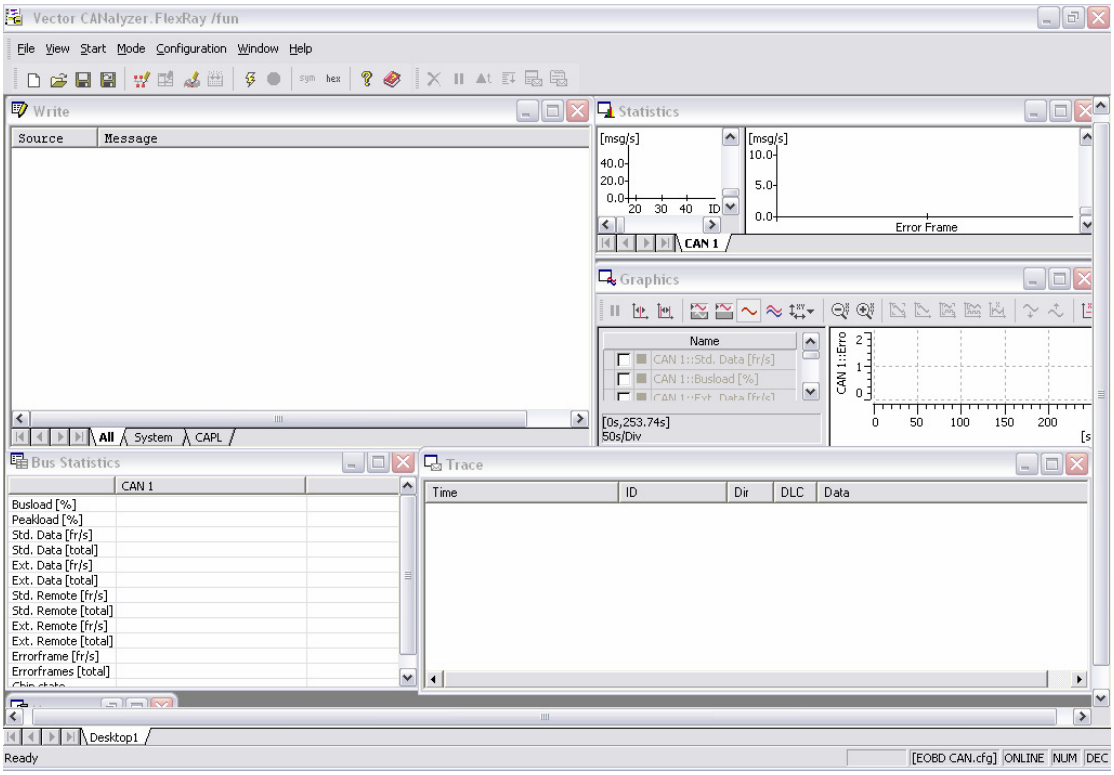


Figure 4.5: CANalyzer

The time stamped information for all messages was very useful for determining the size of the arbitration windows between messages. The data needed to be extracted from the ".txt" file and converted into a ".csv" file for statistical analysis in Microsoft Excel or MATLAB. CANalyzer does not provide a function to generate or export ".csv" files from this window. It was necessary to implement in software a parser, which would deal with this extraction process. This was accomplished by use of VB 2005 Express Edition (Figure 4.6).

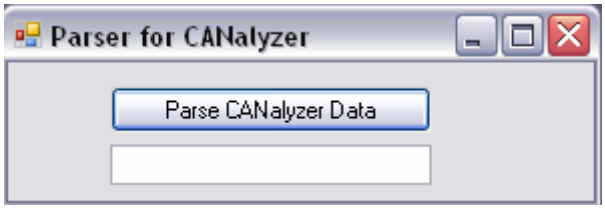


Figure 4.6: Parser for CANalyzer

4.4.1.2 Test 1

It was decided to initially test a network having nodes with two different messages. The message structure to be used was the same as in Figure 3.19, where the Reference message had a period of 20ms and the data message was of period 30ms. Figure 3.19 shows that the optimum message strategy is to send message of period 20ms at 0ms and message of period 30ms at 5ms. The calculated mean for the message set was 12 and a standard deviation 6.

These message periods were invoked in hardware by the use of software similar to that shown in Appendix 8.

The TTCAN network was tested using CANalyzer as seen in Figure 4.7. The Write window shows the duration of the test was 152 seconds, while the total messages processed were 12,623. The average message periods for the 20ms and 30ms messages were 20.005ms and 30.007ms respectively. The exported document from this Write window can be seen in Appendix 9.

It can be seen in the Bus Statistic's window Figure 4.7 that the total Bus load was 7.71%. The Trace window stores a maximum of 5000 messages regardless of how many messages are dealt with. The Graphics window shows that errors on the TTCAN network were zero.

The contents of the Trace window was processed through the parser and then evaluated in Microsoft Excel. The data from the Trace window is not documented in this thesis as it exceeds 100 pages.

Microsoft Excel derived the mean of the messages as 12.00131304 compared to 12 for the Statistical Scheduler and the standard deviation was 5.999744 compared to 6 for the Statistical Scheduler. The above results show a slight difference between the values that were determined by the Statistical Scheduler and those found under testing. The error for the mean is 0.0109% and for the standard deviation is 0.0042%.

The discrepancies occurred due to the TTCAN network being implemented at level 1 (software). This can be seen in the report in Appendix 9, where the mean time for each message is not exactly 20ms and 30ms, but 20.005ms and 30.007ms respectively.

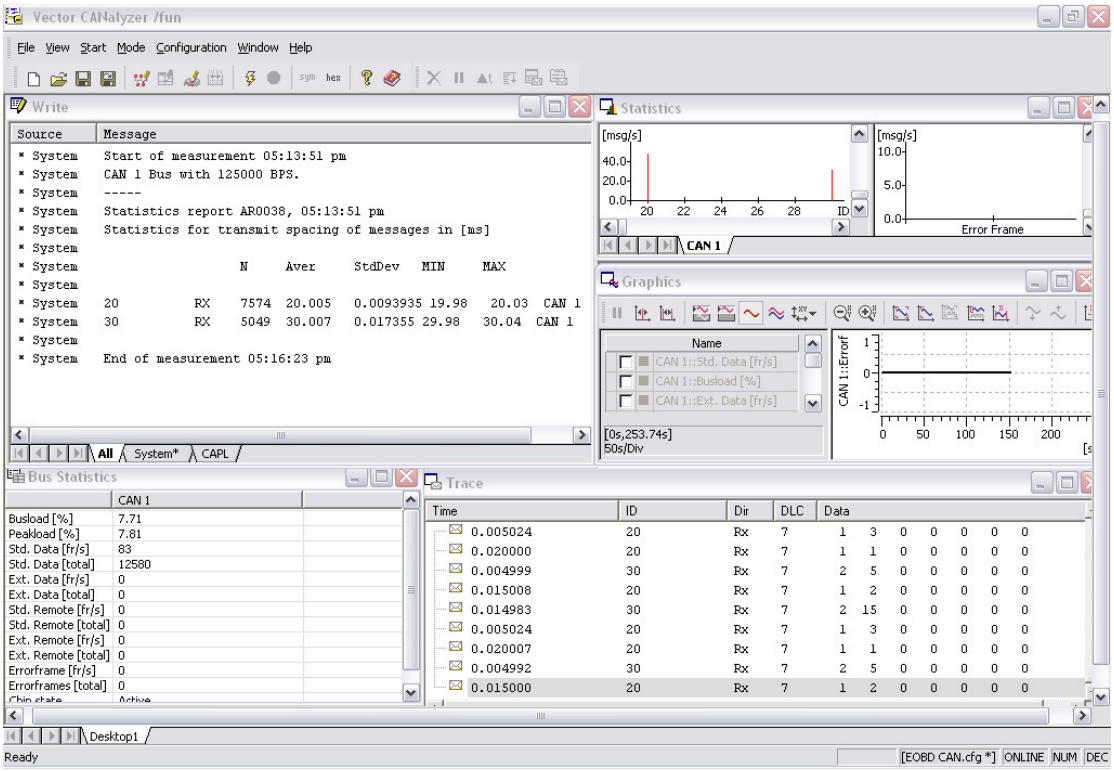


Figure 4.7: Data Acquisition 20ms and 30ms Message Periods

4.4.1.3 Test 2

The second test is based around Example 6 in section 3.5.1. The Statistical Scheduler gave an optimum message strategy, which sends message of period 20ms at 0ms and message 30ms at 5ms and message 40ms at 10ms. This gives a mean for the message set of 9.2307 and a standard deviation of 4.742.

Again, embedded software was written to implement this message scheme.

Appendix 10 shows the data derived from the Write window. 20,036 messages were sent across the network, with a busload of 9.98%. The test took 186 seconds to complete and the 5000 messages saved in the Trace window were again parsed and imported into Microsoft Excel for evaluation. The results from Microsoft Excel showed a mean of 9.2324 and a standard deviation of 4.7425. The Statistical Scheduler calculated the mean to be 9.2307 and the standard deviation to be 4.7419. The error between the Statistical Scheduler and Test 2 for the mean was 0.0184% and the standard deviation was 0.0126%.

Again, the discrepancies are due to the TTCAN network being implemented in level 1 (software). This can be seen in Appendix 10, where the mean time for each message

is not exactly 20ms, 30ms and 40ms, but 20.005ms, 30.007ms and 40.01ms respectively.

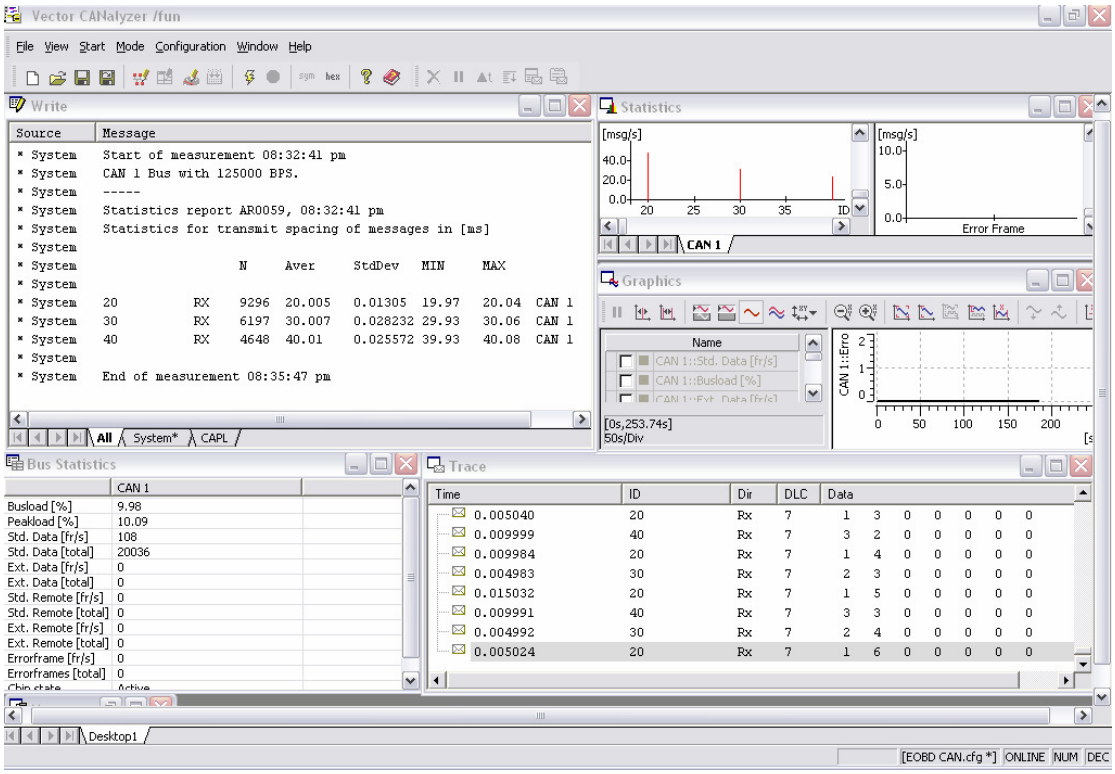


Figure 4.8: Data Acquisition 20ms, 30ms, and 40ms Message Periods

4.4.1.4 Test 3

The third test was based on Example 7 in section 3.5.1. The Statistical Scheduler gave an optimum message strategy, requiring the sending of message of period 20ms at 0ms, message of period 30ms at 6ms, message of period 40ms at 10ms and message of period 50ms at 4ms. The scheduler gave a mean for the message set of 7.7922 and a standard deviation of 4.5308. Again, embedded software was written to implement the message set.

The Write window data is available in Appendix 12. The test was conducted for 158 seconds, with a busload of 11.91%. This allowed in excess of 20,000 messages to use the bus. Again, the Graphics window shows zero cumulative error frames.

The 5000 messages stored in the Trace window were again analysed in Microsoft Excel, where the mean was found to be 7.7981 and the standard deviation was 4.5321.

The error between the Statistical Scheduler and Test 3 for the mean was 0.0012% and for the standard deviation was 0.0286%. Again, the discrepancies are due to the TTCAN network being implemented in level 1 (software). The mean time for each message is not exactly 20ms, 30ms, 40ms and 50ms, but 20.011ms, 30.016ms, 40.01ms and 50.027 respectively.

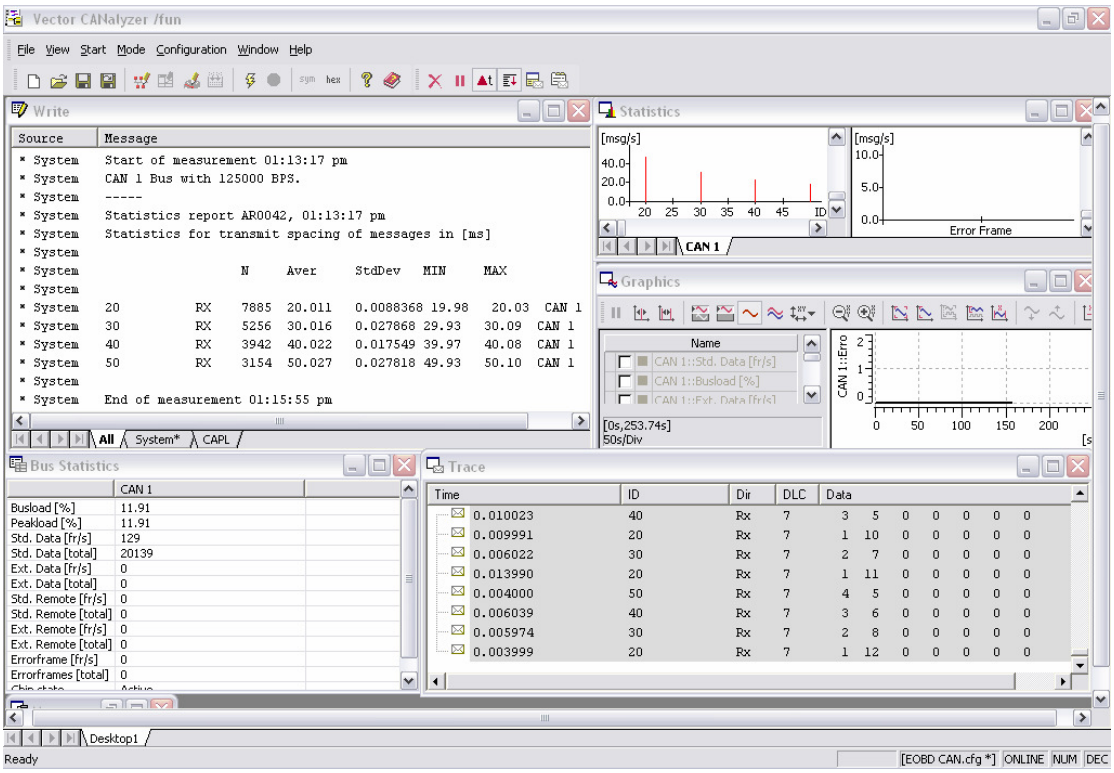


Figure 4.9: Data Acquisition 20ms, 30ms, 40ms, and 50ms Message Periods

4.4.1.5 Testing for Errors

Tests 1 to 3 were limited to approximately 20,000 message frames, and no error frames were generated during this period. Figure 4.10 shows the output windows of CANalyzer while extended testing for message errors was conducted.

The message set used was taken from Test 3, but as can be seen from the Bus Statistics window the system was monitored while in excess of 250,000 message frames were transmitted. The Graphics window shows the time to be at approximately 2000 seconds, but during this time no error messages occurred.

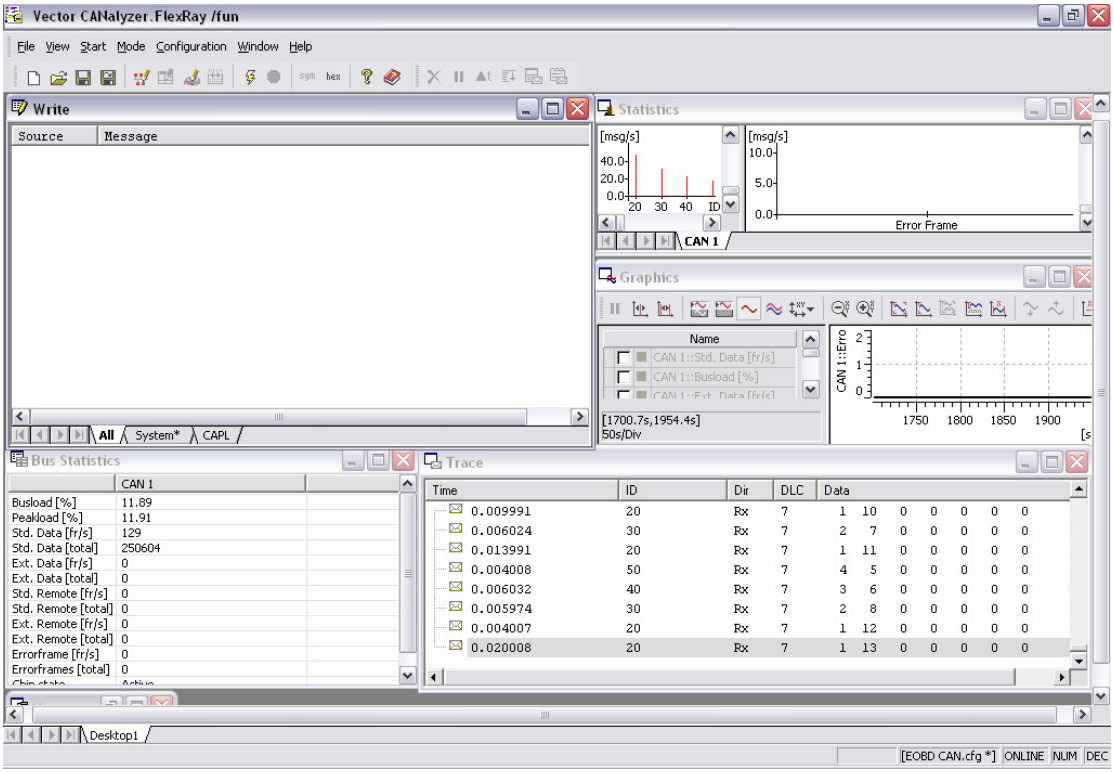


Figure 4.10: Extended Testing for Errors

4.5.1 Summary

The chapter explains the methodology required to successfully implement a TTCAN network in hardware. It began by verifying the propagation delay and skew of the physical medium used to connect all nodes. An illustration was given of the problems associated with propagation delay and how oscillator tolerances of different nodes can affect the CAN network. Included in the chapter is the calculation for CAN Bit timing and it explains the methods used for compensation of the oscillator tolerance. The design process for the hardware was considered together with the selection and integration of the software tool chain.

The embedded software was reviewed and example code for one node was given in the Appendix. The testing procedure was examined and the data acquisition tool discussed. The problems associated with the data acquisition tool were scrutinised and strategies were devised to surmount those difficulties, which included the development of a software parser.

The testing phase was documented and it showed that the physical test scenarios virtually attained the same results as the Statistical Schedule. Any inconsistencies in message timing were caused by the level 1 implementation of TTCAN network.

Chapter 5: Conclusions

5.1 Introduction

This chapter summarises the research project. Chapter One takes a historical look at the automotive electronics and networking systems. It sets out the reasons for undertaking this research and lists the benefits to be gained.

Chapter Two investigates the CAN data link layer in detail, the physical layer, methods of message sending, and message scheduling algorithms.

Chapter Three explores the problems associated with both stochastic and heuristic schedulers, which are presently used to implement automotive TTCAN networks. It examines the potential for using a mathematical model for the design and optimisation of a message schedule.

Chapter Four covers the hardware implementation of level 1 TTCAN, with sections covering the challenges posed by propagation delays and oscillator tolerances. It presents the CAN bit timing of the various segments within the NBT. The embedded software is explained and example code is shown in the appendix. It then details the four test scenarios used to verify the accuracy of the message sets developed by the statistical scheduler and confirmed all findings.

5.2 Conclusions

Modern motor vehicles are efficient and safe when they can operate in real-time. TTCAN is a network topology used in modern motor vehicles, which can offer the potential of real-time capability providing the messages have unhindered access to the network.

This research has highlighted some of the shortcomings of an event driven CAN system operating by arbitration only. This type of system cannot guarantee the delivery of low priority messages at any time and even reasonably high priority messages may have problems broadcasting a message if the highest priority message of the network uses all available bandwidth.

An obvious improvement is TTCAN, which can ensure the transmission of all valid messages within a message set. All other messages, which are not included within the message set, are not guaranteed a time of broadcast, but rely on arbitration. These messages can be of low priority, such as engine rpm or could be a very high priority message such as engine oil pressure failure.

TTCAN requires a message schedule to be generated from the message periods within the message set. If all message periods within the message set are the same e.g. 20ms, it is relatively simple to develop a message set, but message periods within a message set are normally all different, therefore, it is much more difficult to develop useable message sets.

TTCAN schedules are often generated by use of a stochastic or heuristic scheduler. Stochastic schedulers attempt to generate the best SM by a probability distribution. It then uses a 'cost function' to check all message sets for jitter and uses the message set with the least jitter. There may be several message sets with the same jitter, perhaps zero, but the stochastic scheduler cannot differentiate between these message sets and therefore, cannot tell which set is actually the most optimised, with regard to real-time messaging.

Heuristic schedulers initially place all message periods on the SM, using a predefined method and then adjust the arbitration window sizes by trial and error. They again check all message sets for jitter using the cost function and again cannot differentiate between message sets as to their level of optimisation, with regard to real-time messaging.

If the above schedulers devise a message set with three messages broadcast consecutively, then any spontaneous real-time message will have to wait in excess of the time interval of the three messages before making an attempt to broadcast its message. This implies that a TTCAN scheduler should attempt to place arbitration windows between each TTCAN message in order to achieve real-time performance. Also, these arbitration windows should be as large as possible to allow as many spontaneous messages as possible to be broadcast before the next TTCAN message is sent.

This research has shown two problems associated with the stochastic and heuristic schedulers:

- Neither scheduler can produce all available message sets from a group of periodic messages.
- Neither scheduler has a method to verify the real-time performance of the SM.

Both problems were examined in great detail in Chapter 3 of the thesis and a system was devised so all possible message sets within a SM were constructed. This was accomplished by use of software, but requires large computational processes.

It was shown that if two periodic messages of the same time period were used (e.g. 20ms) in a SM then the optimum position for messages are 10ms apart. This is the midpoint or mean of the two messages, allowing two arbitration windows of 5ms each and this will give an optimum SM, with respect to real-time performance. It was observed if two periodic messages were of different periods the optimum position in the SM was not the midpoint or mean, but at a place relative to the midpoint or mean. If three periodic messages of the same period (e.g. 30ms) were used it was found that the mean period was the optimum position in the SM. If the three messages were of different periods then the optimum position was unclear, but it appeared to relate to a position relative to the mean.

This gave the researcher a direction for further study and it was found that the solution lay in finding the mean and standard deviation of each message set. Once this was achieved the message set with the lowest standard deviation was the optimum message set for real-time operation.

Additional software was written to extract the required data from all SM message sets. If a developed message set had two different messages within the same time period, this was deemed a 'zero crossing'. A message set with zero crossing is not useable and was therefore excluded from statistical analysis. All other message sets were evaluated sequentially, calculating the mean and standard deviation. The message set with the lowest standard deviation is the message set with the largest average arbitration window size and will provide the best real-time performance.

It was decided to confirm the Statistical Schedulers results on a physical TTCAN network by devising three tests.

- Test 1: TTCAN network using two messages and two nodes.
- Test 2: TTCAN network using three messages and three nodes.
- Test 3: TTCAN network using four messages and four nodes.

In order to complete the test plan, the following would be required:

- A complete TTCAN network with up to four nodes.
- Implement the optimum message sets, using level 1, TTCAN.

- Collect all network data with a data acquisition tool.
- Analyse data extracted from the TTCAN network with the use of Microsoft Excel and MATLAB.

Four TTCAN nodes were constructed for testing. All CAN cables were validated to ISO11898 standards for propagation delay and skew and the oscillator tolerances were verified. CAN bit timing was calculated and implemented.

Embedded C software was written for all TTCAN nodes, using the optimum message sets developed by the Statistical Scheduler. The C code used interrupts for timing of all TTCAN messages; this minimised the CPU load for all nodes.

The automotive industry's standard tool for data acquisition for CAN networks is CANalyzer. This tool was used and has the ability to calculate the mean and standard deviation for each periodic message. CANalyzer cannot directly calculate the mean or standard deviation between two different periodic messages. In order to calculate the mean and standard deviation of a message set it was necessary to use the time stamp data from each message transmitted on the TTCAN network. This involved implementing a software parser, which could extract the time stamped data from CANalyzer. The parser then manipulated the data, and change the data file type to a ".csv". It could then be imported into either Microsoft Excel or MATLAB for statistical analysis.

The results from the hardware testing compared very favourably with that of the Statistical Scheduler as can be seen in Table 5.1.

	Statistical Scheduler		Hardware Testing	
	Mean	Standard Deviation	Mean	Standard Deviation
Test 1: 20ms, 30ms	12	6	12.0013	5.9997
Test 2: 20ms, 30ms, 40ms	9.2037	4.7419	9.2324	4.7425
Test 3: 20ms, 30ms, 40ms, 50ms	7.7922	4.5308	7.7981	4.5321

Table 5.1: Statistical Scheduler v Hardware Implementation

The research and subsequent testing has shown that message sets without jitter often do not enable a system to operate in real-time and that considerable improvements in the real-time performance of a TTCAN network can be achieved by using the correct message set.

The major drawback with both the heuristic and stochastic schedulers are that a cost function of zero can be attained, but the arbitration windows may not be set at the optimum number or size for real-time messaging. The devised statistical scheduler overcomes these major disadvantages by distributing the message sets in such a manner that allows the optimum number and size of arbitration windows for real-time messaging within a given message set.

Using the message data taken from "Example 1" page 71, three message schedules were devised employing a stochastic scheduler, a heuristic scheduler and a statistical scheduler. These schedules can be seen in Figures 5.1, 5.2 and 5.3.

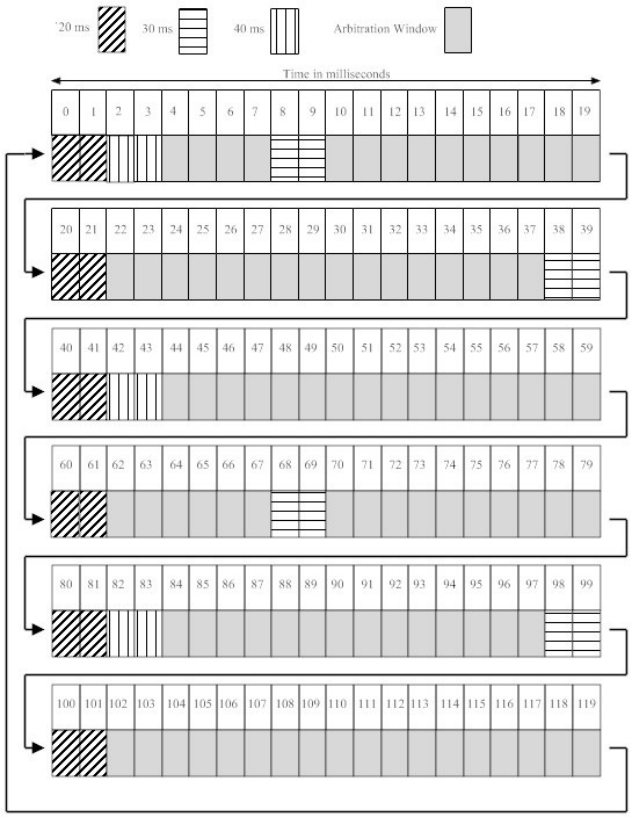


Figure 5.1: Stochastic Message set devised from Example 1, page 71

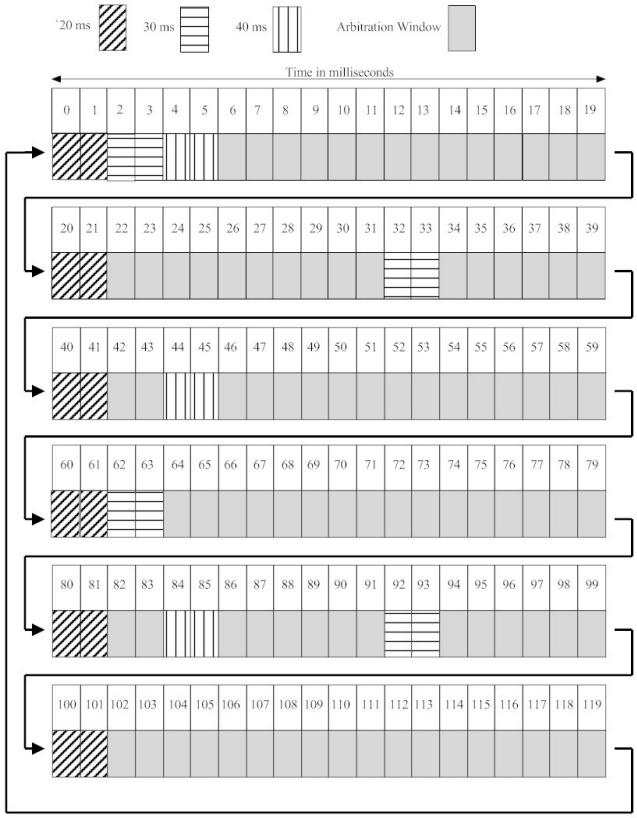


Figure 11: Heuristic Message set devised from Example 1, page 71

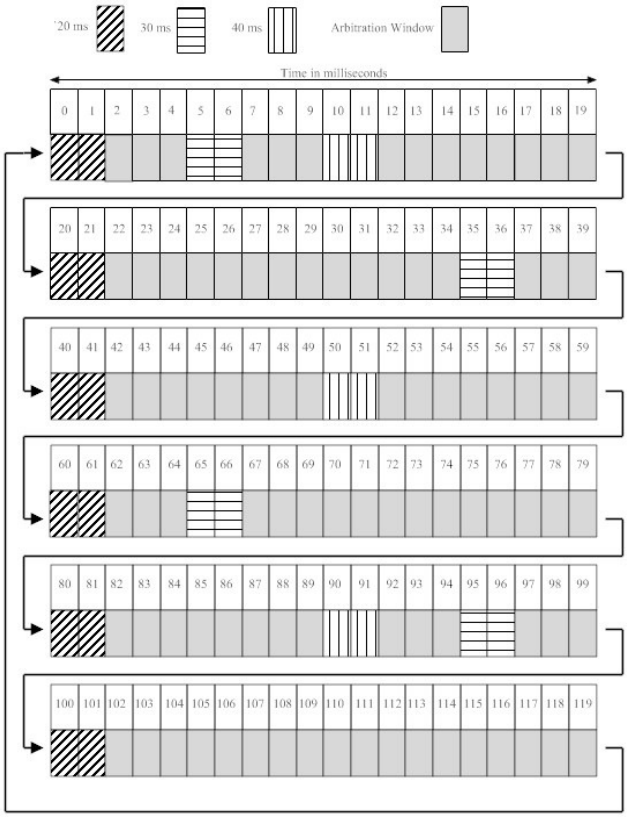


Figure 12: Statistical Message set devised from Example 1, page 71

Table 5.2 compares the effectiveness of real-time messaging within each schedule and to the number of arbitration windows available for real-time messaging.

Type of Scheduler	Maximum wait time to send Real-time Message	Minimum time between periodic messages	Maximum time between periodic messages	Number of arbitration slots in message set
Stochastic	*6ms*	*0ms*	*18ms*	*8*
Heuristic	*8ms*	*0ms*	*18ms*	*10*
Statistical	*4ms*	*3ms*	*18ms*	*13*

Table 5.2: Comparison of Real-time Messages with different message schedules.

The table shows that the maximum wait time for an arbitration slot can be as long as 8ms; this is for the heuristic scheduler. The stochastic scheduler has a maximum wait time of 6ms, which is an improvement, whereas the statistical scheduler has the least wait time of 4ms. The minimum time between periodic messages being broadcast on the network, for both the stochastic and heuristic schedulers, which is 0ms. The minimum time between periodic messages being broadcast on a network, for the statistical scheduler is 3ms. All schedulers gave the same maximum time between periodic messages, which were 18ms. The stochastic and heuristic schedulers had eight and ten arbitration slots respectively, but the statistical scheduler achieved thirteen arbitration slots.

The statistical message scheduler has been devised and demonstrated to produced an optimum message set for real-time operation on a TTCAN network and hence improve the results produced by stochastic or heuristic techniques.

5.3 Further Research

There are opportunities for further development of this system. It was evaluated using four periodic messages on four different nodes. Time constraints prevented further testing and the building of message sets was extremely computational, taking initially up to 24 hours to develop and evaluate some large SMs.

It was noted during the appraisal of the Statistical Scheduler that there appeared to be a linear correlation between message periods. This needs further investigation and if correct, means that the system is scaleable.

The greatest possibility for further research lies in the investigation of a mathematical formula for the generation and statistical analysis of all message sets using any number of periodic messages forming a SM.

Reference List

1. Research and Markets, *China Car Electronics Configuration Report 2007*, available at:http://www.electronics.ca/reports/automotive/car_electronics.html (accessed 3rd September 2007).

2. Gabriel Leen Donal Heffernan, *Expanding Automotive Electronic Systems,* I.E.E.E, Computer and Control Engineering Journal, Volume 35, Issue 1, Jan 2002 Page(s): 88 - 93.

3. G. Leen D. Heffernan and A. Dunne, *Digital Networks in the Automotive Vehicle.* I.E.E.E, Computer and Control Engineering Journal, Volume: 10, Issue: 6, Dec 1999, Page(s): 257 - 266.

4. Robert Bosch, 2004. *Gasoline Engine Management*. Second edition, H. Bauer, ed. Bosch, Page(s) 324 - 329.

5. CAN in Automation, *Controller Area Network.* 1998. available at: http:// www.can-cia.org/can/ (accessed 3rd September 2007).

6. Lembke., M., 1996. *Automotive Electric/Electronic Systems.* Second edition, Bosch. Page(s) 256-257.

7. Robert Bosch, 2004. *Diesel Engine Management*. Third edition, H. Bauer. ed. Page(s) 410 - 415.

8. Hubert Zimmermann, *OSI Reference Model.* IEEE Communications, Volume 28, Issue 4, Apr 1980, Page(s): 425 - 432.

9. Pat Richards. Microchip Technology Inc., *A CAN Physical Layer Discussion,* 2002, available at: http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en012057 (accessed 3rd September 2007).

10. Keith Pazul, M.I., *Controller Area Network (CAN) Basics.* 2002, available at: http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en011694 (accessed 3rd September 2007).

11. Robert Bosch, *CAN Specification Version 2.* 1995, available at: http://esd.cs.ucr.edu/webres/can20.pdf (accessed 3rd September 2007).

12. Siemens, *Controller Area Network.* 1998, available at: staticweb.rasip.fer.hr/rip/seminari/can_atlas/can_doc/canpres.pdf (accessed 3rd September 2007).

13. SAE, *Recommended Practice for a Serial Control and Communications Vehicle Network.* 2005, available at: http://www.sae.org/servlets/productDetail?PROD_TYP=STD&PROD_CD=J1939&HIER_CD=TETES7&WIP_SW=YES (accessed 3rd September 2007).

14. Florian Hartwich. Armin Bassemir, *The Configuration of the CAN Bit Timing.* 2001, available at: http://www.semiconductors.bosch.de/pdf/CiA99Paper.pdf (accessed 3rd September 2007).

15. Freescale, *CAN Bit Timing Requirements.* 1999, available at; http://www.freescale.com/files/microcontrollers/doc/app_note/AN1798.pdf (accessed 3rd September 2007)

16. Pat Richards. Microchip Technology Inc., *CAN Bit Timing.* 2001, available at: http://ww1.microchip.com/downloads/en/AppNotes/00754.pdf (accessed 3rd September 2007).

17. ISO, *Controller area network (CAN) -- Part 2: High-speed medium access unit*. 2003, available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33423 (accessed 3rd September 2007).

18. ISO, *Controller area network (CAN) -- Part 3: Low-speed, fault-tolerant, medium-dependent interface*. 2006, available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=36055. (accessed 3rd September 2007).

19. Tom Denton, *Advanced Automotive Fault Diagnosis*. Second ed. 2006: Butterworth-Heinemann. Page(s) 59-60, 202-203. ISBN-13: 978-0750669917

20. CAN in Automation, *The CAN Physical Layer.* 2005, available at: http://www.can-cia.org/can/physical-layer/index.html. (accessed 3rd September 2007).

21. Craig Szydlowski, Intel Corporation, *Tradeoffs Between Stand-alone and integrated CAN peripherals.* 1994. available at: http://www.sae.org/technical /papers/941655. (accessed 3rd September 2007).

22. Roger Johansson, *Time and event triggered communication scheduling for automotive applications.* 2004. available at: http://www.cedes.se/ Registrerade%20dokument/15%20MF%20DU%20A%20TTCAN%20referenc e%20application.pdf (accessed 3rd September 2007).

23. K. Tindell & A. Burns, *Guaranteed Message Latencies for Distributed Safety-Critical Hard Real-Time Control Networks.* 1994. available at: http://www.cs.york.ac.uk/ftpdir/reports/YCS-94-229.pdf (accessed 3rd September 2007).

24. Thomas Fuhrer. Bernd Muller. Werner Dieterle. Florian Hartwich. Robert Hugel. Michael Walther, *Time Triggered Communication on CAN (Time Triggered CAN - TTCAN).* 2000. available at: http://www.canopen.org/can /ttcan/fuehrer.pdf (accessed 3rd September 2007).

25. ISO, *Road vehicles -- Controller area network (CAN) -- Part 4: Time-triggered communication.* 2004. available at: http://www.iso.org/iso/search. htm?qt=11898-4&searchSubmit=Search&sort=rel&type=simple&published= true (accessed 3rd September 2007).

26. Thilo Schumann. Holger Zeltwanger, *TTCAN an improvement of CAN.* 2002. available at: http://cl-web1.techonline.com/community/member_company/non _member/tech_paper/2091/content_40033 (accessed 3rd September 2007).

27. Microchip, *MCP2515 Stand alone CAN Controller with SPI Interface.* 2005. available at: http://ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf (accessed 3rd September 2007).

28. Bernd Müller. Thomas Führer. Robert Hugel. Robert Bosch GmbH, *Timing in the TTCAN Network.* 2004.

29. Florian Hartwich. Bernd Müller. Thomas Führer. Robert Hugel. Robert Bosch GmbH, *CAN Network with Time Triggered Communication.* 2002. available

at: http://www.semiconductors.bosch.de/pdf/CiA2000Paper_2.pdf (accessed 3rd September 2007).

30. B. Moller, T.F., F. Hartwich, R. Hugel, H. Weiler, Robert Bosch GmbH, *Fault tolerant TTCAN networks.* 2003. available at: http://www.semiconductors. bosch.de/pdf/Fault_Tolerant_TTCAN.pdf (accessed 3rd September 2007).

31. K M Zuberi. K G Shin. Microsoft Corp. WA Redmond, *Design and Implementation of Efficient Message Scheduling for Controller Area Network.* I.E.E.E, Computer Journal, Volume: 49, Issue2, Feb 2000, Page(s): 182 - 188.

32. A. Albert, R. Hugel, *Heuristic scheduling concepts for TTCAN networks.* 2005. available at: http://www.semiconductors.bosch.de/pdf/Heuristic_ Scheduling_Concept.pdf (accessed 3rd September 2007).

33. Jose Fonseca*, Fernanda Coutinho, Jorge Barreiros**, *Scheduling for a TTCAN network with a stochastic optimization algorithm.* 2002. available at: http://www.canopen.org/can/ttcan/fonseca.pdf (accessed 3rd September 2007).

34. Pau Marti. Richard Villa. Josep Fuertes. Gerhard Fohler, *Real Time Scheduling Methods Requirements in Distributed Control Systems.* 2000. available at: http://www.upcnet.es/~pmc16/WRTP2000.pdf (accessed 3rd September 2007).

35. Ken Tindell, *Deadline Monotonic Analysis.* 2003. available at: http://www.embedded.com/2000/0006/0006feat1.htm (accessed 3rd September 2007).

36. Paulo Pedreiras, *EDF Message Scheduling on Controller Area Network.* I.E.E.E, Computer & Control Engineering Journal, Volume: 13, Issue 4, Aug 2002 Page(s): 163 - 170.

37. C. L. Liu and J. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment.* IEICE Transactions on Fundamentals of Electronics, Volume: 20, Issue 1, 1973 Page(s) 46–61

38. Oxford University Press, *The Concise Oxford English Dictionary*. 10th edition, editor. J. Pearsall. 2002: Oxford University Press.

39. P. McFedries, *Formulas and Functions with Microsoft Excel 2003*. 2004 Page(s) 248-282. ISBN-13: 9780789731531.

40. B. Hunt. Ronald Lipsman. J Rosenberg. Kevin R. Coombes. John E. Osborn. Garrett J. Stuck, *A Guide to MATLAB: For Beginners and Experienced Users*. 2 ed. Jun 2006: Cambridge University Press,.

41. Parsons, A., *Visual Basic 2005 Express Edition Starter Kit*. 2006: Wiley Press. ISBN-13: 978-0764595738

42. SAMS, *Visual Basic .Net 2003*, ed. C. Hall. 2003: SAMS Publishing. ISBN-13: 978-0672325311

43. Evangelos Petroutsos; Acey Bunch, *Mastering Microsoft Visual Basic 2005 Express Edition*. 2006: Sybex Inc,. ISBN-13: 978-0782143980

44. Christopher Albert Lupini, *Vehicle Multiplex Communication*. 2004: SAE International. ISBN-13: 978-0-7680-1218-7

45. Stuart Robb. Freescale Semiconductor Inc, *CAN Bit Timing Requirements.* 2004. available at: http://www.freescale.com/files/microcontrollers/doc/app_note/AN1798.pdf (accessed 3rd September 2007).

46. Peter Steffan and Kevin Lavery, *Frequency-Modulated PLL Impact on Controller Area Network (CAN) Communication.* 2004. available at: http:// focus.ti.com/lit/an/spna090/spna090.pdf (accessed 3rd September 2007).

47. MickroElektronika Team, *MikroC Language Reference*. 2006: available at: http://www.mikroe.com/pdf/mikroc/mikroc_manual.pdf (accessed 3rd September 2007).

48. Vector Informatik GmbH, *CANalyser*. 2006: Vector Informatik GmbH. available at: http://www.vector-worldwide.com/portal/medien/cmc/application_notes/AN-AND-1-110_Quick_Introduction_to_CANalyzer.pdf (accessed 3rd September 2007).

Appendix 1: Scheduling Algorithms

1. Borrowed-Virtual-Time Scheduling (BVT)
2. Critical Path Method of Scheduling
3. Deadline-monotonic scheduling (DMS)
4. Deficit round robin (DRR)
5. Earliest deadline first scheduling (EDF)
6. Elastic Round Robin
7. Fair-share scheduling
8. First In, First Out (FIFO), also known as First Come First Served (FCFS)
9. Gang scheduling
10. Genetic Anticipatory
11. Highest response ratio next (HRRN)
12. Interval scheduling
13. Last In, First Out (LIFO)
14. Job Shop Scheduling (see Job shops)
15. Least-connection scheduling
16. Least slack time scheduling (LST)
17. List scheduling
18. Lottery Scheduling
19. Multilevel queue
20. Multilevel Feedback Queue
21. Never queue scheduling
22. O(1) scheduler
23. Proportional Share Scheduling
24. Rate-monotonic scheduling (RMS)
25. Round-robin scheduling (RR)
26. Shortest expected delay scheduling
27. Shortest job next (SJN)
28. Shortest remaining time (SRT)
29. "Take" scheduling
30. Two-level scheduling
31. Weighted fair queuing (WFQ)

Appendix 2: Example 4, System Matrix Data

	M1		M2					
Matrix Number	Message Start Times (ms)						Mean	Standard Deviation
0	0	0	20	30	40	60	11.2	7.17
1	0	1	20	31	40	60	11	6.99
2	0	2	20	32	40	60	11	6.57
3	0	3	20	33	40	60	11	6.26
4	0	4	20	34	40	60	11	6.07
5	0	5	20	35	40	60	11	6
6	0	6	20	36	40	60	11	6.07
7	0	7	20	37	40	60	11	6.26
8	0	8	20	38	40	60	11	6.57
9	0	9	20	39	40	60	11	6.99
10	0	10	20	40	40	60	11.2	7.17
11	0	11	20	40	41	60	11	6.99
12	0	12	20	40	42	60	11	6.57
13	0	13	20	40	43	60	11	6.26
14	0	14	20	40	44	60	11	6.07
15	0	15	20	40	45	60	11	6
16	0	16	20	40	46	60	11	6.07
17	0	17	20	40	47	60	11	6.26
18	0	18	20	40	48	60	11	6.57
19	0	19	20	40	49	60	11	6.99
20	0	20	20	40	50	60	11.2	7.17
21	0	20	21	40	51	60	11	6.99
22	0	20	22	40	52	60	11	6.57
23	0	20	23	40	53	60	11	6.26
24	0	20	24	40	54	60	11	6.07
25	0	20	25	40	55	60	11	6
26	0	20	26	40	56	60	11	6.07
27	0	20	27	40	57	60	11	6.26
28	0	20	28	40	58	60	11	6.57
29	0	20	29	40	59	60	11	6.99
30	0	20	30	40	60	60	11.2	7.17
31	0	1	20	31	40	60	11	6.99
32	0	2	20	32	40	60	11	6.57
33	0	3	20	33	40	60	11	6.26
34	0	4	20	34	40	60	11	6.07
35	0	5	20	35	40	60	11	6
36	0	6	20	36	40	60	11	6.07
37	0	7	20	37	40	60	11	6.26
38	0	8	20	38	40	60	11	6.57
39	0	9	20	39	40	60	11	6.99

40	0	10	20	40	40	60	11.2	7.17
41	0	11	20	40	41	60	11	6.99
42	0	12	20	40	42	60	11	6.57
43	0	13	20	40	43	60	11	6.26
44	0	14	20	40	44	60	11	6.07
45	0	15	20	40	45	60	11	6
46	0	16	20	40	46	60	11	6.07
47	0	17	20	40	47	60	11	6.26
48	0	18	20	40	48	60	11	6.57
49	0	19	20	40	49	60	11	6.99
50	0	20	20	40	50	60	11.2	7.17
51	0	20	21	40	51	60	11	6.99
52	0	20	22	40	52	60	11	6.57
53	0	20	23	40	53	60	11	6.26
54	0	20	24	40	54	60	11	6.07
55	0	20	25	40	55	60	11	6
56	0	20	26	40	56	60	11	6.07
57	0	20	27	40	57	60	11	6.26
58	0	20	28	40	58	60	11	6.57
59	0	20	29	40	59	60	11	6.99
60	0	20	30	40	60	60	11.2	7.17

Appendix 3: Example 5, System Matrix Data

	M1		M2									
Matrix Number	Message Start Times (ms)										Mean	Standard Deviation
0	0	0	20	25	40	50	60	75	80	100	10.2	6.39
1	0	1	20	26	40	51	60	76	80	100	10.1	6.3
2	0	2	20	27	40	52	60	77	80	100	10.1	6.15
3	0	3	20	28	40	53	60	78	80	100	10.1	6.15
4	0	4	20	29	40	54	60	79	80	100	10.1	6.3
5	0	5	20	30	40	55	60	80	80	100	10.2	6.39
6	0	6	20	31	40	56	60	80	81	100	10.1	6.3
7	0	7	20	32	40	57	60	80	82	100	10.1	6.15
8	0	8	20	33	40	58	60	80	83	100	10.1	6.15
9	0	9	20	34	40	59	60	80	84	100	10.1	6.3
10	0	10	20	35	40	60	60	80	85	100	10.2	6.39
11	0	11	20	36	40	60	61	80	86	100	10.1	6.3
12	0	12	20	37	40	60	62	80	87	100	10.1	6.15
13	0	13	20	38	40	60	63	80	88	100	10.1	6.15
14	0	14	20	39	40	60	64	80	89	100	10.1	6.3
15	0	15	20	40	40	60	65	80	90	100	10.2	6.39
16	0	16	20	40	41	60	66	80	91	100	10.1	6.3
17	0	17	20	40	42	60	67	80	92	100	10.1	6.15
18	0	18	20	40	43	60	68	80	93	100	10.1	6.15
19	0	19	20	40	44	60	69	80	94	100	10.1	6.3
20	0	20	20	40	45	60	70	80	95	100	10.2	6.39
21	0	20	21	40	46	60	71	80	96	100	10.1	6.3
22	0	20	22	40	47	60	72	80	97	100	10.1	6.15
23	0	20	23	40	48	60	73	80	98	100	10.1	6.15
24	0	20	24	40	49	60	74	80	99	100	10.1	6.3
25	0	20	25	40	50	60	75	80	100	100	10.2	6.39
26	0	1	20	26	40	51	60	76	80	100	10.1	6.3
27	0	2	20	27	40	52	60	77	80	100	10.1	6.15
28	0	3	20	28	40	53	60	78	80	100	10.1	6.15
29	0	4	20	29	40	54	60	79	80	100	10.1	6.3
30	0	5	20	30	40	55	60	80	80	100	10.2	6.39
31	0	6	20	31	40	56	60	80	81	100	10.1	6.3
32	0	7	20	32	40	57	60	80	82	100	10.1	6.15
33	0	8	20	33	40	58	60	80	83	100	10.1	6.15
34	0	9	20	34	40	59	60	80	84	100	10.1	6.3
35	0	10	20	35	40	60	60	80	85	100	10.2	6.39
36	0	11	20	36	40	60	61	80	86	100	10.1	6.3
37	0	12	20	37	40	60	62	80	87	100	10.1	6.15
38	0	13	20	38	40	60	63	80	88	100	10.1	6.15
39	0	14	20	39	40	60	64	80	89	100	10.1	6.3
40	0	15	20	40	40	60	65	80	90	100	10.2	6.39

41	0	16	20	40	41	60	66	80	91	100	10.1	6.3
42	0	17	20	40	42	60	67	80	92	100	10.1	6.15
43	0	18	20	40	43	60	68	80	93	100	10.1	6.15
44	0	19	20	40	44	60	69	80	94	100	10.1	6.3
45	0	20	20	40	45	60	70	80	95	100	10.2	6.39
46	0	20	21	40	46	60	71	80	96	100	10.1	6.3
47	0	20	22	40	47	60	72	80	97	100	10.1	6.15
48	0	20	23	40	48	60	73	80	98	100	10.1	6.15
49	0	20	24	40	49	60	74	80	99	100	10.1	6.3
50	0	20	25	40	50	60	75	80	100	100	10.2	6.39
51	0	1	20	26	40	51	60	76	80	100	10.1	6.3
52	0	2	20	27	40	52	60	77	80	100	10.1	6.15
53	0	3	20	28	40	53	60	78	80	100	10.1	6.15
54	0	4	20	29	40	54	60	79	80	100	10.1	6.3
55	0	5	20	30	40	55	60	80	80	100	10.2	6.39
56	0	6	20	31	40	56	60	80	81	100	10.1	6.3
57	0	7	20	32	40	57	60	80	82	100	10.1	6.15
58	0	8	20	33	40	58	60	80	83	100	10.1	6.15
59	0	9	20	34	40	59	60	80	84	100	10.1	6.3
60	0	10	20	35	40	60	60	80	85	100	10.2	6.39
61	0	11	20	36	40	60	61	80	86	100	10.1	6.3
62	0	12	20	37	40	60	62	80	87	100	10.1	6.15
63	0	13	20	38	40	60	63	80	88	100	10.1	6.15
64	0	14	20	39	40	60	64	80	89	100	10.1	6.3
65	0	15	20	40	40	60	65	80	90	100	10.2	6.39
66	0	16	20	40	41	60	66	80	91	100	10.1	6.3
67	0	17	20	40	42	60	67	80	92	100	10.1	6.15
68	0	18	20	40	43	60	68	80	93	100	10.1	6.15
69	0	19	20	40	44	60	69	80	94	100	10.1	6.3
70	0	20	20	40	45	60	70	80	95	100	10.2	6.39
71	0	20	21	40	46	60	71	80	96	100	10.1	6.3
72	0	20	22	40	47	60	72	80	97	100	10.1	6.15
73	0	20	23	40	48	60	73	80	98	100	10.1	6.15
74	0	20	24	40	49	60	74	80	99	100	10.1	6.3
75	0	20	25	40	50	60	75	80	100	100	10.2	6.39
76	0	1	20	26	40	51	60	76	80	100	10.1	6.3
77	0	2	20	27	40	52	60	77	80	100	10.1	6.15
78	0	3	20	28	40	53	60	78	80	100	10.1	6.15
79	0	4	20	29	40	54	60	79	80	100	10.1	6.3
80	0	5	20	30	40	55	60	80	80	100	10.2	6.39
81	0	6	20	31	40	56	60	80	81	100	10.1	6.3
82	0	7	20	32	40	57	60	80	82	100	10.1	6.15
83	0	8	20	33	40	58	60	80	83	100	10.1	6.15
84	0	9	20	34	40	59	60	80	84	100	10.1	6.3
85	0	10	20	35	40	60	60	80	85	100	10.2	6.39
86	0	11	20	36	40	60	61	80	86	100	10.1	6.3
87	0	12	20	37	40	60	62	80	87	100	10.1	6.15
88	0	13	20	38	40	60	63	80	88	100	10.1	6.15
89	0	14	20	39	40	60	64	80	89	100	10.1	6.3
90	0	15	20	40	40	60	65	80	90	100	10.2	6.39
91	0	16	20	40	41	60	66	80	91	100	10.1	6.3
92	0	17	20	40	42	60	67	80	92	100	10.1	6.15

93	0	18	20	40	43	60	68	80	93	100	10.1	6.15
94	0	19	20	40	44	60	69	80	94	100	10.1	6.3
95	0	20	20	40	45	60	70	80	95	100	10.2	6.39
96	0	20	21	40	46	60	71	80	96	100	10.1	6.3
97	0	20	22	40	47	60	72	80	97	100	10.1	6.15
98	0	20	23	40	48	60	73	80	98	100	10.1	6.15
99	0	20	24	40	49	60	74	80	99	100	10.1	6.3
100	0	20	25	40	50	60	75	80	100	100	10.2	6.39

Appendix 4: VB Code to Develop System Matrix

```
1   Imports System
2   Imports System.IO
3
4   Public Class Form1
5
6       ' Public Declarations
7
8       Public a, b, q, r, s, t, v, w, x, y, l As Integer
9       Public Array1(9) As Integer ' Time between messages (ms)
10      Public ArrayTime0(200) As Integer
11      Public ArrayTime1(w, v) As Integer
12      Public ArrayTime2(w, v) As Integer
13      Public ArrayTime3(w, v) As Integer
14      Public ArrayMessage0(t, v) As Integer
15      Public ArrayFinalSort(s) As Integer
16      Public ArraySTD(r) As Double
17      Public ArrayOut(q) As Double
18      Public ArrayOut1(q) As Double
19      Public ArrayOut2(q) As Integer
20      Public ArrayOut30(q, r) As Double
21      Public ArrayOut31(q, r) As Double
22      Public ArrayOut32(q, r) As Integer
23      Public ArrayOut33(q, r) As Integer
24      Public LCM As Integer   ' Find Basic Matrix Size
25      Public Message_Time(a, b) As Integer ' Messages time(ms)
26      Public Message_List(50, 50) As Integer 'Complete message list
27      Public count As Integer ' Find end of array of messages
28
29      ' Enter Message durations
30
31      Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles Button1.Click
32          Try
33              Array1(x) = TextBox1.Text
34              x = x + 1
35              TextBox13.Text = x
36
37              ' Error check for data input
38
39          Catch
40              If TextBox1.Text = "" Then
41                  MsgBox("You Haven't Entered a Value")
42              End If
43          End Try
44          TextBox1.Text = ""
45      End Sub
46
47      ' Sort Message data and calculate System Matrix size (LCM)
48
49      Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
```

```
    Handles Button2.Click
50      Dim swap, index, index1, ItemCount, ItemCount1, temp1 As Integer
51      index = 0
52      ItemCount = x
53      ItemCount1 = x
54
55      ' Sort data to get smallest time into position in Array1(0)"Reference Message"
56      ' Check the other data remains intact in the other arrays
57
58      Do
59        swap = False
60        For index = 1 To ItemCount - 1
61          If Array1(index - 1) > Array1(index) Then
62            temp1 = Array1(index - 1)
63            Array1(index - 1) = Array1(index)
64            Array1(index) = temp1
65            swap = True
66          End If
67        Next index
68      Loop Until swap = False
69      Do
70        For index1 = 0 To x - 1
71          TextBox6.Text = Array1(index1) & vbNewLine & TextBox6.Text
72        Next index1
73      Loop Until ItemCount1 = x
74
75      ' Calculate system matrix size "LCM"
76
77      Dim SystemMatrix, Counter, Increment As Integer
78      TextBox11.Text = Array1(0)
79      LCM = Array1(0)
80      Do
81        For Counter = 1 To (x - 1)
82          SystemMatrix = LCM Mod Array1(Counter)
83          If SystemMatrix > 0 Then
84            Counter = 0
85            Increment = (Increment + 1)
86            LCM = Array1(0) * (Increment + 1)
87          End If
88
89        Next
90
91      Loop Until Counter = x
92
93      ' Print to Screen LCM value
94
95      TextBox12.Text = LCM
96
97      'Use this section of code if System Matrix has 2 Messages
98
99      If x = 2 Then
100       Call Message_Timing2()
101       Call Message_List2()
102       Call Sort2()
```

```
103             Call CsvFile2()
104         End If
105
106         'Use this section of code if System Matrix has 3 Messages
107
108         If x = 3 Then
109             Call Message_Timing3()
110             Call Message_List3()
111             Call Sort3()
112             Call CsvFile3()
113         End If
114
115         'Use this section of code if System Matrix has 2 Messages
116
117         If x = 4 Then
118             Call Message_Timing4()
119             Call Message_List4()
120             Call Sort4()
121             Call CsvFile4()
122         End If
123
124     End Sub
125
126     'Calculate initial timings between each message ID in Array(0) positions
127
128     Private Sub Message_Timing2()
129         Dim i, k, a, b, d, e, f, count As Integer
130         y = LCM / Array1(0)
131         ReDim ArrayTime0(y)
132         w = LCM / Array1(1)
133         ReDim ArrayTime1(Array1(0), (w - 1))
134         d = 0
135         k = 0
136         f = 0
137         count = 0
138         k = 0
139         b = Array1(count)
140         a = LCM / b
141         e = b
142         For i = 0 To a
143             d = d + Array1(count)
144             e = d - b
145             f = 0
146             ArrayTime0(i) = e
147         Next i
148         count = count + 1
149         d = 0
150         k = 0
151         b = Array1(1)
152         a = LCM / b
153         f = 0
154         For k = 0 To Array1(0)
155             For i = 0 To a - 1
156                 d = d + Array1(count)
```

```
157             e = d - b
158             ArrayTime1(k, i) = e
159         Next i
160         i = 0
161         f = f + 1
162         d = f
163     Next k
164 End Sub
165 
166 'Generate Message timing list for each ID and put into an array
167 
168 Private Sub Message_List2()
169     Dim u, i, k As Integer
170     ReDim ArrayMessage0((Array1(0)), (LCM / Array1(0)) + (LCM / Array1(1)))
171     t = 0
172     u = 0
173     i = 0
174     k = 0
175     For k = 0 To (Array1(0))
176         For i = 0 To (LCM / Array1(0))
177             ArrayMessage0(t, u) = ArrayTime0(i)
178             u = u + 1
179         Next i
180         For i = 0 To ((LCM / Array1(1)) - 1)
181             ArrayMessage0(t, u) = ArrayTime1(k, i)
182             u = u + 1
183         Next i
184         t = t + 1
185         u = 0
186     Next k
187 
188 End Sub
189 
190 'Sort the TTCAN Messages in to a useable message sequence and carry out statistical calculations
191 
192 Private Sub Sort2()
193     Dim b, d, i, j, k, l, m, n, swap, index, ItemCount, temp1, s0, s1, s2, s3, r0, r1 As Integer
194     Dim sum, mean, mean1, st, std, stddev, stddev1, stddev2 As Double
195     ReDim ArrayFinalSort(LCM / Array1(0) + (LCM / (Array1(1))))
196     ReDim ArraySTD((LCM / Array1(0) + ((LCM / Array1(1)))))
197     ReDim ArrayOut(Array1(0))
198     ReDim ArrayOut1(Array1(0))
199     ReDim ArrayOut2(Array1(0))
200     t = 0
201     n = 0
202     b = 0
203     stddev1 = 100
204     stddev2 = 0
205     index = 0
206     ItemCount = ((LCM / Array1(0)) + (LCM / Array1(1)))
207     For m = 0 To Array1(0)
208         For i = 0 To ((LCM / Array1(0)) + (LCM / Array1(1)))
209             ArrayFinalSort(i) = ArrayMessage0(t, i)
210         Next i
```

```
211         r0 = ArrayFinalSort(0)
212         r1 = ArrayFinalSort((LCM / Array1(0) + 1))
213
214
215         ' Sort data to get smallest time into position in Array1(0)
216         ' and make sure all other data remains intact in other arrays
217
218         Do
219            swap = False
220            For index = 1 To ItemCount
221               If ArrayFinalSort(index - 1) > ArrayFinalSort(index) Then
222                  temp1 = ArrayFinalSort(index - 1)
223                  ArrayFinalSort(index - 1) = ArrayFinalSort(index)
224                  ArrayFinalSort(index) = temp1
225                  swap = True
226               End If
227            Next index
228         Loop Until swap = False
229
230         For j = 0 To ItemCount - 1
231            ArraySTD(j) = ArrayFinalSort(j + 1) - ArrayFinalSort(j)
232
233            ' If two messages in matrix are in the same position
234            ' don't do the statistical maths
235
236            If ArraySTD(j) = 0 Then
237               b = b + 1
238               TextBox14.Text = b & "  Zero crossing when " & Array1(0) & " = " & r0 & " and when " & Array1(1) & " = " & r1 & vbNewLine & TextBox14.Text
239               GoTo Zero_Crossing2
240            End If
241         Next j
242
243         'Start the statistical maths
244         'Find the Mean
245
246         For k = 0 To ItemCount - 1
247            sum = sum + ArraySTD(k)
248         Next k
249         mean = sum / k
250         For l = 0 To ItemCount - 1
251            st = ((ArraySTD(l) - mean) ^ 2)
252            std = std + st
253         Next l
254
255         ' Calculate the Standard Deviation
256
257         stddev = Math.Sqrt(std / k)
258         ArrayOut(n) = stddev
259         ArrayOut1(n) = mean
260         ArrayOut2(n) = t
261         d = d + 1
262         TextBox2.Text = d & "   Mean = " & mean & "    STDdev = " & stddev & " When " & Array1(0) & " = " & r0 & " , " & Array1(1) & " = " & r1 & vbNewLine & TextBox2.Text
263
```

```
264             ' Find the highest Standard Deviation in the Matrix
265
266             If stddev1 > stddev Then
267                 stddev1 = stddev
268                 mean1 = mean
269                 s0 = r0
270                 s1 = r1
271             End If
272
273             ' Find the highest Standard Deviation in the Matrix
274
275             If stddev2 < stddev Then
276                 stddev2 = stddev
277                 mean1 = mean
278                 s2 = r0
279                 s3 = r1
280             End If
281
282             'Set Constant Values back to zero and increment arrays
283
284     Zero_Crossing2:
285             mean = 0
286             sum = 0
287             st = 0
288             std = 0
289             stddev = 0
290             n = n + 1
291             t = t + 1
292             index = 0
293         Next m
294
295         ' Print highest and lowest Mean, Standard Deviation and positions in Matrix to the screen
296
297         TextBox10.Text = "Mean = " & mean1 & ", STDdevL = " & stddev1 & ", when " & Array1(0)
    & " = " & s0 & ", " & Array1(1) & " = " & s1 & vbNewLine & TextBox10.Text
298         TextBox10.Text = "Mean = " & mean1 & ", STDdevH = " & stddev2 & ", when " & Array1(0)
    & " = " & s2 & ", " & Array1(1) & " = " & s3 & vbNewLine & TextBox10.Text
299
300     End Sub
301
    ---------------------------------------------------------------------------
302     'Output all data to a CSV file for further analysis in Excel and MATLAB
303
304     Private Sub CsvFile2()
305         Dim t As Double
306         Dim u, v, n As Integer
307         t = 0
308         n = 0
309
310         ' Create an instance of StreamWriter to write text to a file.
311
312         Using sw As StreamWriter = New StreamWriter("C:\TestFile.csv")
313             ' Add some text to the file.
314             For u = 0 To Array1(0)
315                 t = ArrayOut2(n)
316                 sw.Write(t)
```

```
317             sw.Write(",")
318             n = n + 1
319         Next u
320         sw.WriteLine("")
321         n = 0
322         For v = 0 To Array1(0)
323             t = ArrayOut1(n)
324             sw.Write(t)
325             sw.Write(",")
326             n = n + 1
327         Next v
328         sw.WriteLine("")
329         n = 0
330         For w = 0 To Array1(0)
331             t = ArrayOut(n)
332             sw.Write(t)
333             sw.Write(",")
334             n = n + 1
335         Next w
336         n = 0
337         sw.WriteLine("")
338         sw.WriteLine(TextBox14.Text)
339         sw.Close()
340
341     End Using
342
343 End Sub
344 ---------------------------------------------------------------------------
345 'Calculate initial timings between each message ID in Array(0) positions
346
347 Private Sub Message_Timing3()
348     Dim i, k, a, b, d, e, f, z, count As Integer
349     y = LCM / Array1(0)
350     ReDim ArrayTime0(y)
351     w = LCM / Array1(1)
352     ReDim ArrayTime1((Array1(0)), (w - 1))
353     z = LCM / Array1(2)
354     ReDim ArrayTime2((Array1(0)), (z - 1))
355     d = 0
356     k = 0
357     f = 0
358     count = 0
359     k = 0
360     b = Array1(count)
361     a = LCM / b
362     e = b
363     For i = 0 To a
364         d = d + Array1(count)
365         e = d - b
366         f = 0
367         ArrayTime0(i) = e
368     Next i
369     count = count + 1
370     d = 0
```

```
371         k = 0
372         b = Array1(1)
373
374         f = 0
375         a = LCM / Array1(count)
376         For k = 0 To Array1(0)
377            For i = 0 To a - 1
378               d = d + Array1(count)
379               e = d - b
380               ArrayTime1(k, i) = e
381            Next i
382            i = 0
383            f = f + 1
384            d = f
385         Next k
386         count = count + 1
387         k = 0
388         f = 0
389         b = Array1(count)
390         d = 0
391         a = LCM / Array1(count)
392         For k = 0 To Array1(0)
393            For i = 0 To a - 1
394               d = d + Array1(count)
395               e = d - b
396               ArrayTime2(k, i) = e
397            Next i
398            i = 0
399            f = f + 1
400            d = f
401         Next k
402      End Sub
403
404      'Generate Message timing list for each ID and put into an array
405
406      Private Sub Message_List3()
407         Dim i, k, l, h, u As Integer
408         ReDim ArrayMessage0((Array1(0) * (Array1(0) + 2)), ((LCM / Array1(0))) + ((LCM / Array1(1))) + ((LCM / Array1(2))))
409         t = 0
410         u = 0
411         h = 0
412         i = 0
413         k = 0
414         l = 0
415         For h = 0 To (Array1(0))
416            For k = 0 To (Array1(0))
417               For i = 0 To (LCM / Array1(0))
418                  ArrayMessage0(t, u) = ArrayTime0(i)
419                  u = u + 1
420               Next i
421               For i = 0 To ((LCM / Array1(1)) - 1)
422                  ArrayMessage0(t, u) = ArrayTime1(l, i)
423                  u = u + 1
```

```
424             Next i
425
426             For i = 0 To ((LCM / Array1(2)) - 1)
427                 ArrayMessage0(t, u) = ArrayTime2(h, i)
428                 u = u + 1
429             Next i
430             t = t + 1
431             u = 0
432             l = l + 1
433         Next k
434
435         l = 0
436
437     Next h
438
439 End Sub
440
441 'Sort the TTCAN Messages in to a useable message sequence
442
443 Private Sub Sort3()
444     Dim a, e, d, g, h, i, j, k, l, m, n, c, s0, s1, s2, s3, s4, s5, r0, r1, r2, swap, index, ItemCount, temp1 As Integer
445     Dim sum, mean, mean1, st, std, stddev, stddev1, stddev2 As Double
446     ReDim ArrayFinalSort((LCM / Array1(0) + LCM / Array1(1) + LCM / Array1(2)))
447     ReDim ArraySTD(LCM / Array1(0) + ((LCM / Array1(1)) - 1) + LCM / Array1(2))
448     ReDim ArrayOut30(Array1(0), Array1(0))
449     ReDim ArrayOut31(Array1(0), Array1(0))
450     ReDim ArrayOut32(Array1(0), Array1(0))
451     t = 0
452     n = 0
453     c = 0
454     d = 0
455     stddev1 = 100
456     index = 0
457     ItemCount = (LCM / Array1(0) + LCM / Array1(1) + LCM / Array1(2))
458     For h = 0 To Array1(0)
459         For m = 0 To Array1(0)
460             For i = 0 To (LCM / Array1(0) + LCM / Array1(1) + LCM / Array1(2))
461                 ArrayFinalSort(i) = ArrayMessage0(t, i)
462
463             Next i
464             r0 = ArrayFinalSort(0)
465             r1 = ArrayFinalSort((LCM / Array1(0) + 1))
466             r2 = ArrayFinalSort((LCM / Array1(0) + LCM / Array1(1) + 1))
467
468             ' Sort data to get smallest time into position in Array1(0)
469             ' and make sure all other data remains intact in other arrays
470
471             Do
472                 swap = False
473                 For index = 1 To ItemCount
474                     If ArrayFinalSort(index - 1) > ArrayFinalSort(index) Then
475                         temp1 = ArrayFinalSort(index - 1)
476                         ArrayFinalSort(index - 1) = ArrayFinalSort(index)
```

```
                    ArrayFinalSort(index) = temp1
                    swap = True
                End If
            Next index
        Loop Until swap = False

        For j = 0 To ItemCount - 1
            ArraySTD(j) = ArrayFinalSort(j + 1) - ArrayFinalSort(j)

            ' If two messages in matrix are in the same value
            ' don't do the statistical maths

            If ArraySTD(j) = 0 Then
                e = e + 1
                TextBox14.Text = e & "  Zero crossing when " & Array1(0) & " = " & r0 & ",  " & Array1(1) & " = " & r1 & ",  " & Array1(2) & " = " & r2 & vbNewLine & TextBox14.Text
                GoTo Zero_Crossing1
            End If
        Next j

        'Start the statistical maths
        'Find the Mean

        For k = 0 To ItemCount - 1
            sum = sum + ArraySTD(k)
        Next k
        mean = sum / k
        For l = 0 To ItemCount - 1
            st = ((ArraySTD(l) - mean) ^ 2)
            std = std + st
        Next l

        ' Calculate the Standard Deviation

        stddev = Math.Sqrt(std / k)
        ArrayOut30(a, g) = stddev
        ArrayOut31(a, g) = mean
        ArrayOut32(a, g) = t
        d = d + 1
        TextBox2.Text = d & "   Mean = " & mean & "    STDdev = " & stddev & " When " & Array1(0) & " = " & r0 & " , " & Array1(1) & " = " & r1 & " , " & Array1(2) & " = " & r2 & vbNewLine & TextBox2.Text

        ' Find the highest Standard Deviation in the Matrix

        If stddev1 > stddev Then
            stddev1 = stddev
            mean1 = mean
            s0 = r0
            s1 = r1
            s2 = r2
            c = t
        End If

        ' Find the Lowest Standard Deviation in the Matrix
```

```
529
530             If stddev2 < stddev Then
531                 stddev2 = stddev
532                 mean1 = mean
533                 s3 = r0
534                 s4 = r1
535                 s5 = r2
536             End If
537 Zero_Crossing1:
538             mean = 0
539             sum = 0
540             st = 0
541             std = 0
542             stddev = 0
543             g = g + 1
544             t = t + 1
545         Next m
546         a = a + 1
547         g = 0
548     Next h
549
550     ' Print highest and lowest Mean, Standard Deviation and positions in Matrix to the screen
551
    TextBox10.Text = "Mean = " & mean1 & ", & STDdevL = " & stddev1 & ", when " &
552 Array1(0) & " = " & s0 & ",  " & Array1(1) & " = " & s1 & ",  " & Array1(2) & " = " & s2 &
    vbNewLine & TextBox10.Text
    TextBox10.Text = "Mean = " & mean1 & ", & STDdevH = " & stddev2 & ", when " &
553 Array1(0) & " = " & s3 & ",  " & Array1(1) & " = " & s4 & ",  " & Array1(2) & " = " & s5 &
    vbNewLine & TextBox10.Text
554
555 End Sub
556
```

```
557 'Output all data to a CSV file for further analysis in Excel and MATLAB
558
559 Private Sub CsvFile3()
560     Dim t As Double
561     Dim a, b, c, d, g, u, n As Integer
562     t = 0
563     n = 0
564
565     ' Create an instance of StreamWriter to write text to a file.
566
567     Using sw As StreamWriter = New StreamWriter("C:\TestFile.csv")
568         ' Add some text to the file.
569         For d = 0 To (Array1(0))
570             For u = 0 To (Array1(0))
571                 t = ArrayOut32(a, g)
572                 sw.Write(t)
573                 sw.Write(",")
574                 g = g + 1
575             Next u
576             sw.WriteLine("")
577             g = 0
578             For b = 0 To (Array1(0))
579                 t = ArrayOut31(a, g)
```

```
                sw.Write(t)
                sw.Write(",")
                g = g + 1
            Next b
            sw.WriteLine("")
            g = 0
            For c = 0 To Array1(0)
                t = ArrayOut30(a, g)
                sw.Write(t)
                sw.Write(",")
                g = g + 1
            Next c
            g = 0
            a = a + 1
        Next d
        sw.WriteLine("")
        sw.WriteLine(TextBox14.Text)
        sw.Close()

    End Using

End Sub

'Calculate initial timings between each message ID in Array(0) positions

Private Sub Message_Timing4()
    Dim i, k, a, b, d, e, f, z, z1, g, count As Integer
    y = LCM / Array1(0)
    ReDim ArrayTime0(y)
    w = LCM / Array1(1)
    ReDim ArrayTime1(Array1(0), (w - 1))
    z = LCM / Array1(2)
    ReDim ArrayTime2(Array1(0), (z - 1))
    z1 = LCM / Array1(3)
    ReDim ArrayTime3(Array1(0), (z1 - 1))
    g = 0
    d = 0
    k = 0
    f = 0
    count = 0
    k = 0
    b = Array1(count)
    a = LCM / b
    e = b
    For i = 0 To a
        d = d + Array1(count)
        e = d - b
        f = 0
        ArrayTime0(i) = e
    Next i
    count = count + 1
    d = 0
    k = 0
    b = Array1(1)
```

```
634
635         f = 0
636         a = LCM / Array1(count)
637         For k = 0 To Array1(0)
638             For i = 0 To a - 1
639                 d = d + Array1(count)
640                 e = d - b
641                 ArrayTime1(k, i) = e
642             Next i
643             i = 0
644             f = f + 1
645             d = f
646         Next k
647         count = count + 1
648         k = 0
649         f = 0
650         b = Array1(count)
651         d = 0
652         a = LCM / Array1(count)
653         For k = 0 To Array1(0)
654             For i = 0 To a - 1
655                 d = d + Array1(count)
656                 e = d - b
657                 ArrayTime2(k, i) = e
658             Next i
659             i = 0
660             f = f + 1
661             d = f
662         Next k
663         count = count + 1
664         k = 0
665         f = 0
666         b = Array1(count)
667         d = 0
668         a = LCM / Array1(count)
669         For k = 0 To Array1(0)
670             For i = 0 To a - 1
671                 d = d + Array1(count)
672                 e = d - b
673                 ArrayTime3(k, i) = e
674             Next i
675             i = 0
676             f = f + 1
677             d = f
678         Next k
679     End Sub
680
681     "Generate Message timing list for each ID and put into an array
682
683     Private Sub Message_List4()
684         Dim i, k, l, h, g, m, n, u As Integer
685         ReDim ArrayMessage0(((Array1(0) * (Array1(0) + 2) * (Array1(0) + 2))), (LCM / Array1(0)) +
    (LCM / Array1(1)) + (LCM / Array1(2)) + (LCM / Array1(3)))
686         t = 0
```

```
687         u = 0
688         g = 0
689         h = 0
690         i = 0
691         k = 0
692         l = 0
693         n = 0
694         For g = 0 To (Array1(0))
695            For h = 0 To (Array1(0))
696               For k = 0 To (Array1(0))
697                  For i = 0 To (LCM / Array1(0))
698                     ArrayMessage0(t, u) = ArrayTime0(i)
699                     u = u + 1
700                  Next i
701                  For i = 0 To ((LCM / Array1(1)) - 1)
702                     ArrayMessage0(t, u) = ArrayTime1(l, i)
703                     u = u + 1
704                  Next i
705
706                  For i = 0 To ((LCM / Array1(2)) - 1)
707                     ArrayMessage0(t, u) = ArrayTime2(m, i)
708                     u = u + 1
709                  Next i
710
711                  For i = 0 To ((LCM / Array1(3)) - 1)
712                     ArrayMessage0(t, u) = ArrayTime3(n, i)
713                     u = u + 1
714                  Next i
715
716                  t = t + 1
717                  u = 0
718                  l = l + 1
719               Next k
720               m = m + 1
721               l = 0
722
723            Next h
724            l = 0
725            m = 0
726            n = n + 1
727         Next g
728      End Sub
729
730   'Sort Data in correct order and carry out statistical Maths
731      Private Sub Sort4()
732         Dim a, aa, aaa, b, g, h, i, j, k, l, m, n, c, d, s0, s1, s2, s3, s4, s5, s6, s7, r0, r1, r2, r3, swap, index,
    ItemCount, temp1 As Integer
733         Dim sum, mean, mean1, st, std, stddev, stddev1, stddev2 As Double
734         ReDim ArrayFinalSort((LCM / Array1(0) + LCM / Array1(1) + LCM / Array1(2) + LCM /
    Array1(3)))
735         ReDim ArraySTD(LCM / Array1(0) + ((LCM / Array1(1)) - 1) + LCM / Array1(2) + LCM /
    Array1(3))
736         ReDim ArrayOut30(Array1(0) * (Array1(0) + 2), Array1(0))
737         ReDim ArrayOut31(Array1(0) * (Array1(0) + 2), Array1(0))
738         ReDim ArrayOut32(Array1(0) * (Array1(0) + 2), Array1(0))
```

```
739         ReDim ArrayOut33(Array1(0) * (Array1(0) + 2), Array1(0))
740         t = 0
741         n = 0
742         c = 0
743         d = 0
744         stddev1 = 100
745         index = 0
746         ItemCount = (LCM / Array1(0) + LCM / Array1(1) + LCM / Array1(2) + LCM / Array1(3))
747         For b = 0 To Array1(0)
748            For h = 0 To Array1(0)
749               For m = 0 To Array1(0)
750                  For i = 0 To (LCM / Array1(0) + LCM / Array1(1) + LCM / Array1(2) + LCM /
    Array1(3))
751                     ArrayFinalSort(i) = ArrayMessage0(t, i)
752
753                  Next i
754                  r0 = ArrayFinalSort(0)
755                  r1 = ArrayFinalSort((LCM / Array1(0) + 1))
756                  r2 = ArrayFinalSort((LCM / Array1(0) + LCM / Array1(1) + 1))
757                  r3 = ArrayFinalSort((LCM / Array1(0) + LCM / Array1(1) + LCM / Array1(2) + 1))
758
759                  ' Sort data to get smallest time into position in Array1(0)
760                  ' and make sure all other data remains intact in other arrays
761
762                  Do
763                     swap = False
764                     For index = 1 To ItemCount
765                        If ArrayFinalSort(index - 1) > ArrayFinalSort(index) Then
766                           temp1 = ArrayFinalSort(index - 1)
767                           ArrayFinalSort(index - 1) = ArrayFinalSort(index)
768                           ArrayFinalSort(index) = temp1
769                           swap = True
770                        End If
771                     Next index
772                  Loop Until swap = False
773
774                  ' If two messages in matrix are in the same position
775                  ' don't do the statistical maths
776
777                  For j = 0 To ItemCount - 1
778                     ArraySTD(j) = ArrayFinalSort(j + 1) - ArrayFinalSort(j)
779
780                     If ArraySTD(j) = 0 Then
781                        c = c + 1
                           TextBox14.Text = c & "  Zero crossing when " & Array1(0) & " = " & r0 & ",  " &
782 Array1(1) & " = " & r1 & ",  " & Array1(2) & " = " & r2 & ",  " & Array1(3) & " = " & r3 & ",  " &
    vbNewLine & TextBox14.Text
783                        aaa = 1
784                        GoTo Zero_Crossing
785                     End If
786                  Next j
787
788                  'Start the statistical maths
789                  'Find the Mean
790
```

```
791                 For k = 0 To ItemCount - 1
792                     sum = sum + ArraySTD(k)
793                 Next k
794                 mean = sum / k
795                 For l = 0 To ItemCount - 1
796                     st = ((ArraySTD(l) - mean) ^ 2)
797                     std = std + st
798                 Next l
799
800                 ' Calculate the Standard Deviation
801
802                 stddev = Math.Sqrt(std / k)
803                 ArrayOut30(a, g) = stddev
804                 ArrayOut31(a, g) = mean
805                 ArrayOut32(a, g) = t
806                 d = d + 1
                    TextBox2.Text = d & "   Mean = " & mean & "    STDdev = " & stddev & " When " &
807 Array1(0) & " = " & r0 & " , " & Array1(1) & " = " & r1 & " , " & Array1(2) & " = " & r2 & " , " &
    Array1(3) & " = " & r3 & vbNewLine & TextBox2.Text
808                 aa = 1
809 Zero_Crossing:
810                 ' Find the highest Standard Deviation in the Matrix
811
812                 If ((aa = 1) And (aaa = 0)) Then
813                     If stddev1 > stddev Then
814                         stddev1 = stddev
815                         mean1 = mean
816                         s0 = r0
817                         s1 = r1
818                         s2 = r2
819                         s3 = r3
820                         c = t
821                     End If
822
823                     ' Find the Lowest Standard Deviation in the Matrix
824
825                     If stddev2 < stddev Then
826                         stddev2 = stddev
827                         mean1 = mean
828                         s4 = r0
829                         s5 = r1
830                         s6 = r2
831                         s7 = r3
832                     End If
833                     aa = 0
834                 End If
835                 aaa = 0
836                 mean = 0
837                 sum = 0
838                 st = 0
839                 std = 0
840                 stddev = 0
841                 g = g + 1
842                 t = t + 1
843             Next m
```

```
844             a = a + 1
845             g = 0
846           Next h
847         Next b
848
849         ' Print highest and lowest Mean, Standard Deviation and positions in Matrix to the screen
850
851         TextBox10.Text = "Mean = " & mean1 & ", & STDdevL = " & stddev1 & ", when " &
    Array1(0) & " = " & s0 & ",  " & Array1(1) & " = " & s1 & ",  " & Array1(2) & " = " & s2 & ",  " &
    Array1(3) & " = " & s3 & vbNewLine & TextBox10.Text
852         TextBox10.Text = "Mean = " & mean1 & ", & STDdevH = " & stddev2 & ", when " &
    Array1(0) & " = " & s4 & ",  " & Array1(1) & " = " & s5 & ",  " & Array1(2) & " = " & s6 & ",  " &
    Array1(3) & " = " & s7 & vbNewLine & TextBox10.Text
853       End Sub
854
855       'Output all data to a CSV file for further analysis in Excel and MATLAB
856
857       Private Sub CsvFile4()
858         Dim t As Double
859         Dim a, b, c, d, g, u, n As Integer
860         t = 0
861         n = 0
862
863         ' Create an instance of StreamWriter to write text to a file.
864
865         Using sw As StreamWriter = New StreamWriter("C:\TestFile1.csv")
866           ' Add some text to the file.
867           For d = 0 To (Array1(0))
868             For u = 0 To (Array1(0))
869               t = ArrayOut32(a, g)
870               sw.Write(t)
871               sw.Write(",")
872               g = g + 1
873             Next u
874             sw.WriteLine("")
875             g = 0
876             For b = 0 To (Array1(0))
877               t = ArrayOut31(a, g)
878               sw.Write(t)
879               sw.Write(",")
880               g = g + 1
881             Next b
882             sw.WriteLine("")
883             g = 0
884             For c = 0 To Array1(0)
885               t = ArrayOut30(a, g)
886               sw.Write(t)
887               sw.Write(",")
888               g = g + 1
889             Next c
890             g = 0
891             a = a + 1
892           Next d
893           sw.WriteLine("")
894           sw.WriteLine(TextBox14.Text)
```

```
895                 sw.Close()
896
897             End Using
898
899         End Sub
900     End Class
```

Appendix 5: CSV File

Generated from Messages with periods of 20ms and 30ms.

0,1,2,3,4,5,6,7,8,9,0,11,12,13,14,15,16,17,18,19,0,

0,12,12,12,12,12,12,12,12,12,0,12,12,12,12,12,12,12,12,12,0,

0,6.98569967862919,6.57267069006199,6.26099033699941,6.06630035524124,6,6.06630035524124,6.26099033699941,6.57267069006199,6.98569967862919,0,6.98569967862919,6.57267069006199,6.26099033699941,6.06630035524124,6,6.06630035524124,6.26099033699941,6.57267069006199,6.98569967862919,0,

3 Zero crossing when 20 = 0 and when 30 = 20

2 Zero crossing when 20 = 0 and when 30 = 10

1 Zero crossing when 20 = 0 and when 30 = 0

Appendix 6: Output from Statistical Scheduler, Periods 20, 30 and 40ms

441 Mean = 9.23076923076923 STDdev = 8.28486893405308 When 20 = 0 , 30 = 20 , 40 = 20

440 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 19 , 40 = 20

439 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 18 , 40 = 20

438 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 17 , 40 = 20

437 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 16 , 40 = 20

436 Mean = 9.23076923076923 STDdev = 7.2975638311578 When 20 = 0 , 30 = 15 , 40 = 20

435 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 14 , 40 = 20

434 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 13 , 40 = 20

433 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 12 , 40 = 20

432 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 11 , 40 = 20

431 Mean = 9.23076923076923 STDdev = 8.28486893405308 When 20 = 0 , 30 = 10 , 40 = 20

430 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 9 , 40 = 20

429 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 8 , 40 = 20

428 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 7 , 40 = 20

427 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 6 , 40 = 20

426 Mean = 9.23076923076923 STDdev = 7.2975638311578 When 20 = 0 , 30 = 5 , 40 = 20

425 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 4 , 40 = 20

424 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 3 , 40 = 20

423 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 2 , 40 = 20

422 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 1 , 40 = 20

421 Mean = 9.23076923076923 STDdev = 8.28486893405308 When 20 = 0 , 30 = 0 , 40 = 20

420 Mean = 9.23076923076923 STDdev = 7.83634379182464 When 20 = 0 , 30 = 20 , 40 = 19

419 Mean = 9.23076923076923 STDdev = 7.65761407061219 When 20 = 0 , 30 = 19 , 40 = 19

418 Mean = 9.23076923076923 STDdev = 7.35007949882539 When 20 = 0 , 30 = 18 , 40 = 19

417 Mean = 9.23076923076923 STDdev = 7.11611222890968 When 20 = 0 , 30 = 17 , 40 = 19

416 Mean = 9.23076923076923 STDdev = 6.96313198931283 When 20 = 0 , 30 = 16 , 40 = 19

415 Mean = 9.23076923076923 STDdev = 6.89653029990551 When 20 = 0 , 30 = 15 , 40 = 19

414 Mean = 9.23076923076923 STDdev = 6.9188020990058 When 20 = 0 , 30 = 14 , 40 = 19

413 Mean = 9.23076923076923 STDdev = 7.02910264711426 When 20 = 0 , 30 = 13 , 40 = 19

412 Mean = 9.23076923076923 STDdev = 7.22340050065753 When 20 = 0 , 30 = 12 , 40 = 19

411 Mean = 9.23076923076923 STDdev = 7.49516609508418 When 20 = 0 , 30 = 11 , 40 = 19

410 Mean = 9.23076923076923 STDdev = 7.83634379182465 When 20 = 0 , 30 = 10 , 40 = 19

409 Mean = 9.23076923076923 STDdev = 7.65761407061219 When 20 = 0 , 30 = 9 , 40 = 19

408 Mean = 9.23076923076923 STDdev = 7.35007949882539 When 20 = 0 , 30 = 8 , 40 = 19

407 Mean = 9.23076923076923 STDdev = 7.11611222890968 When 20 = 0 , 30 = 7 , 40 = 19

406 Mean = 9.23076923076923 STDdev = 6.96313198931283 When 20 = 0 , 30 = 6 , 40 = 19

405 Mean = 9.23076923076923 STDdev = 6.89653029990551 When 20 = 0 , 30 = 5 , 40 = 19

404 Mean = 9.23076923076923 STDdev = 6.9188020990058 When 20 = 0 , 30 = 4 , 40 = 19

403 Mean = 9.23076923076923 STDdev = 7.02910264711426 When 20 = 0 , 30 = 3 , 40 = 19

402 Mean = 9.23076923076923 STDdev = 7.22340050065753 When 20 = 0 , 30 = 2 , 40 = 19

401 Mean = 9.23076923076923 STDdev = 7.49516609508418 When 20 = 0 , 30 = 1 , 40 = 19

400 Mean = 9.23076923076923 STDdev = 7.83634379182464 When 20 = 0 , 30 = 0 , 40 = 19

399 Mean = 9.23076923076923 STDdev = 7.42297728111681 When 20 = 0 , 30 = 20 , 40 = 18

398 Mean = 9.23076923076923 STDdev = 7.21274348906526 When 20 = 0 , 30 = 19 , 40 = 18

397 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 18 , 40 = 18

396 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 17 , 40 = 18

395 Mean = 9.23076923076923 STDdev = 6.63503343164549 When 20 = 0 , 30 = 16 , 40 = 18

394 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 15 , 40 = 18

393 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 14 , 40 = 18

392 Mean = 9.23076923076923 STDdev = 6.63503343164549 When 20 = 0 , 30 = 13 , 40 = 18

391 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 12 , 40 = 18

390 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 11 , 40 = 18

389 Mean = 9.23076923076923 STDdev = 7.42297728111681 When 20 = 0 , 30 = 10 , 40 = 18

388 Mean = 9.23076923076923 STDdev = 7.21274348906526 When 20 = 0 , 30 = 9 , 40 = 18

387 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 8 , 40 = 18

386 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 7 , 40 = 18

385 Mean = 9.23076923076923 STDdev = 6.63503343164549 When 20 = 0 , 30 = 6 , 40 = 18

384 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 5 , 40 = 18

383 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 4 , 40 = 18

382 Mean = 9.23076923076923 STDdev = 6.63503343164548 When 20 = 0 , 30 = 3 , 40 = 18

381 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 2 , 40 = 18

380 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 1 , 40 = 18

379 Mean = 9.23076923076923 STDdev = 7.42297728111681 When 20 = 0 , 30 = 0 , 40 = 18

378 Mean = 9.23076923076923 STDdev = 7.05095570340368 When 20 = 0 , 30 = 20 , 40 = 17

377 Mean = 9.23076923076923 STDdev = 6.80671440173198 When 20 = 0 , 30 = 19 , 40 = 17

376 Mean = 9.23076923076923 STDdev = 6.64661679299322 When 20 = 0 , 30 = 18 , 40 = 17

375 Mean = 9.23076923076923 STDdev = 6.57681061532279 When 20 = 0 , 30 = 17 , 40 = 17

374 Mean = 9.23076923076923 STDdev = 6.36279868458776 When 20 = 0 , 30 = 16 , 40 = 17

373 Mean = 9.23076923076923 STDdev = 6.24073277445287 When 20 = 0 , 30 = 15 , 40 = 17

372 Mean = 9.23076923076923 STDdev = 6.21603195410367 When 20 = 0 , 30 = 14 , 40 = 17

371 Mean = 9.23076923076923 STDdev = 6.28984341438758 When 20 = 0 , 30 = 13 , 40 = 17

370 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 12 , 40 = 17

369 Mean = 9.23076923076923 STDdev = 6.71569741098989 When 20 = 0 , 30 = 11 , 40 = 17

368 Mean = 9.23076923076923 STDdev = 7.05095570340369 When 20 = 0 , 30 = 10 , 40 = 17

367 Mean = 9.23076923076923 STDdev = 6.80671440173198 When 20 = 0 , 30 = 9 , 40 = 17

366 Mean = 9.23076923076923 STDdev = 6.64661679299322 When 20 = 0 , 30 = 8 , 40 = 17

365 Mean = 9.23076923076923 STDdev = 6.57681061532279 When 20 = 0 , 30 = 7 , 40 = 17

364 Mean = 9.23076923076923 STDdev = 6.36279868458776 When 20 = 0 , 30 = 6 , 40 = 17

363 Mean = 9.23076923076923 STDdev = 6.24073277445287 When 20 = 0 , 30 = 5 , 40 = 17

362 Mean = 9.23076923076923 STDdev = 6.21603195410367 When 20 = 0 , 30 = 4 , 40 = 17

361 Mean = 9.23076923076923 STDdev = 6.28984341438758 When 20 = 0 , 30 = 3 , 40 = 17

360 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 2 , 40 = 17

359 Mean = 9.23076923076923 STDdev = 6.71569741098989 When 20 = 0 , 30 = 1 , 40 = 17

358 Mean = 9.23076923076923 STDdev = 7.05095570340368 When 20 = 0 , 30 = 0 , 40 = 17

357 Mean = 9.23076923076923 STDdev = 6.72714187971552 When 20 = 0 , 30 = 20 , 40 = 16

356 Mean = 9.23076923076923 STDdev = 6.4468697968483 When 20 = 0 , 30 = 19 , 40 = 16

355 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 18 , 40 = 16

354 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 17 , 40 = 16

353 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 16 , 40 = 16

352 Mean = 9.23076923076923 STDdev = 6.00197206250189 When 20 = 0 , 30 = 15 , 40 = 16

351 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 14 , 40 = 16

350 Mean = 9.23076923076923 STDdev = 6.00197206250189 When 20 = 0 , 30 = 13 , 40 = 16

349 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 12 , 40 = 16

348 Mean = 9.23076923076923 STDdev = 6.39896441325705 When 20 = 0 , 30 = 11 , 40 = 16

347 Mean = 9.23076923076923 STDdev = 6.72714187971552 When 20 = 0 , 30 = 10 , 40 = 16

346 Mean = 9.23076923076923 STDdev = 6.4468697968483 When 20 = 0 , 30 = 9 , 40 = 16

345 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 8 , 40 = 16

344 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 7 , 40 = 16

343 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 6 , 40 = 16

342 Mean = 9.23076923076923 STDdev = 6.00197206250189 When 20 = 0 , 30 = 5 , 40 = 16

341 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 4 , 40 = 16

340 Mean = 9.23076923076923 STDdev = 6.00197206250189 When 20 = 0 , 30 = 3 , 40 = 16

339 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 2 , 40 = 16

338 Mean = 9.23076923076923 STDdev = 6.39896441325705 When 20 = 0 , 30 = 1 , 40 = 16

337 Mean = 9.23076923076923 STDdev = 6.72714187971552 When 20 = 0 , 30 = 0 , 40 = 16

336 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 20 , 40 = 15

335 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 19 , 40 = 15

334 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 18 , 40 = 15

333 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 17 , 40 = 15

332 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 16 , 40 = 15

331 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 15 , 40 = 15

330 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 14 , 40 = 15

329 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 13 , 40 = 15

328 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 12 , 40 = 15

327 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 11 , 40 = 15

326 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 10 , 40 = 15

325 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 9 , 40 = 15

324 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 8 , 40 = 15

323 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 7 , 40 = 15

322 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 6 , 40 = 15

321 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 5 , 40 = 15

320 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 4 , 40 = 15

319 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 3 , 40 = 15

318 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 2 , 40 = 15

317 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 1 , 40 = 15

316 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 0 , 40 = 15

315 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 20 , 40 = 14

314 Mean = 9.23076923076923 STDdev = 5.8985506192864 When 20 = 0 , 30 = 19 , 40 = 14

313 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 18 , 40 = 14

312 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 17 , 40 = 14

311 Mean = 9.23076923076923 STDdev = 5.40873717884521 When 20 = 0 , 30 = 16 , 40 = 14

310 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 15 , 40 = 14

309 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 14 , 40 = 14

308 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 13 , 40 = 14

307 Mean = 9.23076923076923 STDdev = 5.73992557113158 When 20 = 0 , 30 = 12 , 40 = 14

306 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 11 , 40 = 14

305 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 10 , 40 = 14

304 Mean = 9.23076923076923 STDdev = 5.8985506192864 When 20 = 0 , 30 = 9 , 40 = 14

303 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 8 , 40 = 14

302 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 7 , 40 = 14

301 Mean = 9.23076923076923 STDdev = 5.40873717884521 When 20 = 0 , 30 = 6 , 40 = 14

300 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 5 , 40 = 14

299 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 4 , 40 = 14

298 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 3 , 40 = 14

297 Mean = 9.23076923076923 STDdev = 5.73992557113158 When 20 = 0 , 30 = 2 , 40 = 14

296 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 1 , 40 = 14

295 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 0 , 40 = 14

294 Mean = 9.23076923076923 STDdev = 6.11623119442591 When 20 = 0 , 30 = 20 , 40 = 13

293 Mean = 9.23076923076923 STDdev = 5.72650848321069 When 20 = 0 , 30 = 19 , 40 = 13

292 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 18 , 40 = 13

291 Mean = 9.23076923076923 STDdev = 5.22057830798682 When 20 = 0 , 30 = 17 , 40 = 13

290 Mean = 9.23076923076923 STDdev = 5.1314092554332 When 20 = 0 , 30 = 16 , 40 = 13

289 Mean = 9.23076923076923 STDdev = 5.16130344529731 When 20 = 0 , 30 = 15 , 40 = 13

288 Mean = 9.23076923076923 STDdev = 5.3082496920265 When 20 = 0 , 30 = 14 , 40 = 13

287 Mean = 9.23076923076923 STDdev = 5.56297991746601 When 20 = 0 , 30 = 13 , 40 = 13

286 Mean = 9.23076923076923 STDdev = 5.64533634827962 When 20 = 0 , 30 = 12 , 40 = 13

285 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 11 , 40 = 13

284 Mean = 9.23076923076923 STDdev = 6.11623119442591 When 20 = 0 , 30 = 10 , 40 = 13

283 Mean = 9.23076923076923 STDdev = 5.72650848321069 When 20 = 0 , 30 = 9 , 40 = 13

282 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 8 , 40 = 13

281 Mean = 9.23076923076923 STDdev = 5.22057830798682 When 20 = 0 , 30 = 7 , 40 = 13

280 Mean = 9.23076923076923 STDdev = 5.1314092554332 When 20 = 0 , 30 = 6 , 40 = 13

279 Mean = 9.23076923076923 STDdev = 5.16130344529731 When 20 = 0 , 30 = 5 , 40 = 13

278 Mean = 9.23076923076923 STDdev = 5.3082496920265 When 20 = 0 , 30 = 4 , 40 = 13

277 Mean = 9.23076923076923 STDdev = 5.56297991746601 When 20 = 0 , 30 = 3 , 40 = 13

276 Mean = 9.23076923076923 STDdev = 5.64533634827962 When 20 = 0 , 30 = 2 , 40 = 13

275 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 1 , 40 = 13

274 Mean = 9.23076923076923 STDdev = 6.11623119442591 When 20 = 0 , 30 = 0 , 40 = 13

273 Mean = 9.23076923076923 STDdev = 6.05302017627877 When 20 = 0 , 30 = 20 , 40 = 12

272 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 19 , 40 = 12

271 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 18 , 40 = 12

270 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 17 , 40 = 12

269 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 16 , 40 = 12

268 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 15 , 40 = 12

267 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 14 , 40 = 12

266 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 13 , 40 = 12

265 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 12 , 40 = 12

264 Mean = 9.23076923076923 STDdev = 5.79328319500392 When 20 = 0 , 30 = 11 , 40 = 12

263 Mean = 9.23076923076923 STDdev = 6.05302017627877 When 20 = 0 , 30 = 10 , 40 = 12

262 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 9 , 40 = 12

261 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 8 , 40 = 12

260 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 7 , 40 = 12

259 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 6 , 40 = 12

258 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 5 , 40 = 12

257 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 4 , 40 = 12

256 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 3 , 40 = 12

255 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 2 , 40 = 12

254 Mean = 9.23076923076923 STDdev = 5.79328319500392 When 20 = 0 , 30 = 1 , 40 = 12

253 Mean = 9.23076923076923 STDdev = 6.05302017627877 When 20 = 0 , 30 = 0 , 40 = 12

252 Mean = 9.23076923076923 STDdev = 6.06571507806656 When 20 = 0 , 30 = 20 , 40 = 11

251 Mean = 9.23076923076923 STDdev = 5.61801834969545 When 20 = 0 , 30 = 19 , 40 = 11

250 Mean = 9.23076923076923 STDdev = 5.24996477869279 When 20 = 0 , 30 = 18 , 40 = 11

249 Mean = 9.23076923076923 STDdev = 4.97924687159454 When 20 = 0 , 30 = 17 , 40 = 11

248 Mean = 9.23076923076923 STDdev = 4.8222855442023 When 20 = 0 , 30 = 16 , 40 = 11

247 Mean = 9.23076923076923 STDdev = 4.79027614675085 When 20 = 0 , 30 = 15 , 40 = 11

246 Mean = 9.23076923076923 STDdev = 4.88567523329244 When 20 = 0 , 30 = 14 , 40 = 11

245 Mean = 9.23076923076923 STDdev = 5.10133988664285 When 20 = 0 , 30 = 13 , 40 = 11

244 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 12 , 40 = 11

243 Mean = 9.23076923076923 STDdev = 5.83298111080889 When 20 = 0 , 30 = 11 , 40 = 11

242 Mean = 9.23076923076923 STDdev = 6.06571507806656 When 20 = 0 , 30 = 10 , 40 = 11

241 Mean = 9.23076923076923 STDdev = 5.61801834969545 When 20 = 0 , 30 = 9 , 40 = 11

240 Mean = 9.23076923076923 STDdev = 5.24996477869279 When 20 = 0 , 30 = 8 , 40 = 11

239 Mean = 9.23076923076923 STDdev = 4.97924687159454 When 20 = 0 , 30 = 7 , 40 = 11

238 Mean = 9.23076923076923 STDdev = 4.8222855442023 When 20 = 0 , 30 = 6 , 40 = 11

237 Mean = 9.23076923076923 STDdev = 4.79027614675085 When 20 = 0 , 30 = 5 , 40 = 11

236 Mean = 9.23076923076923 STDdev = 4.88567523329244 When 20 = 0 , 30 = 4 , 40 = 11

235 Mean = 9.23076923076923 STDdev = 5.10133988664285 When 20 = 0 , 30 = 3 , 40 = 11

234 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 2 , 40 = 11

233 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 1 , 40 = 11

232 Mean = 9.23076923076923 STDdev = 6.06571507806656 When 20 = 0 , 30 = 0 , 40 = 11

231 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 20 , 40 = 10

230 Mean = 9.23076923076923 STDdev = 5.6860672654081 When 20 = 0 , 30 = 19 , 40 = 10

229 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 18 , 40 = 10

228 Mean = 9.23076923076923 STDdev = 4.99467171715321 When 20 = 0 , 30 = 17 , 40 = 10

227 Mean = 9.23076923076923 STDdev = 4.80630749286563 When 20 = 0 , 30 = 16 , 40 = 10

226 Mean = 9.23076923076923 STDdev = 4.74185692536075 When 20 = 0 , 30 = 15 , 40 = 10

225 Mean = 9.23076923076923 STDdev = 4.80630749286563 When 20 = 0 , 30 = 14 , 40 = 10

224 Mean = 9.23076923076923 STDdev = 4.99467171715321 When 20 = 0 , 30 = 13 , 40 = 10

223 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 12 , 40 = 10

222 Mean = 9.23076923076923 STDdev = 5.6860672654081 When 20 = 0 , 30 = 11 , 40 = 10

221 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 10 , 40 = 10

220 Mean = 9.23076923076923 STDdev = 5.6860672654081 When 20 = 0 , 30 = 9 , 40 = 10

219 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 8 , 40 = 10

218 Mean = 9.23076923076923 STDdev = 4.99467171715321 When 20 = 0 , 30 = 7 , 40 = 10

217 Mean = 9.23076923076923 STDdev = 4.80630749286563 When 20 = 0 , 30 = 6 , 40 = 10

216 Mean = 9.23076923076923 STDdev = 4.74185692536075 When 20 = 0 , 30 = 5 , 40 = 10

215 Mean = 9.23076923076923 STDdev = 4.80630749286563 When 20 = 0 , 30 = 4 , 40 = 10

214 Mean = 9.23076923076923 STDdev = 4.99467171715321 When 20 = 0 , 30 = 3 , 40 = 10

213 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 2 , 40 = 10

212 Mean = 9.23076923076923 STDdev = 5.6860672654081 When 20 = 0 , 30 = 1 , 40 = 10

211 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 0 , 40 = 10

210 Mean = 9.23076923076923 STDdev = 6.06571507806656 When 20 = 0 , 30 = 20 , 40 = 9

209 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 19 , 40 = 9

208 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 18 , 40 = 9

207 Mean = 9.23076923076923 STDdev = 5.10133988664285 When 20 = 0 , 30 = 17 , 40 = 9

206 Mean = 9.23076923076923 STDdev = 4.88567523329244 When 20 = 0 , 30 = 16 , 40 = 9

205 Mean = 9.23076923076923 STDdev = 4.79027614675085 When 20 = 0 , 30 = 15 , 40 = 9

204 Mean = 9.23076923076923 STDdev = 4.8222855442023 When 20 = 0 , 30 = 14 , 40 = 9

203 Mean = 9.23076923076923 STDdev = 4.97924687159454 When 20 = 0 , 30 = 13 , 40 = 9

202 Mean = 9.23076923076923 STDdev = 5.24996477869279 When 20 = 0 , 30 = 12 , 40 = 9

201 Mean = 9.23076923076923 STDdev = 5.61801834969545 When 20 = 0 , 30 = 11 , 40 = 9

200 Mean = 9.23076923076923 STDdev = 6.06571507806656 When 20 = 0 , 30 = 10 , 40 = 9

199 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 9 , 40 = 9

198 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 8 , 40 = 9

197 Mean = 9.23076923076923 STDdev = 5.10133988664285 When 20 = 0 , 30 = 7 , 40 = 9

196 Mean = 9.23076923076923 STDdev = 4.88567523329244 When 20 = 0 , 30 = 6 , 40 = 9

195 Mean = 9.23076923076923 STDdev = 4.79027614675085 When 20 = 0 , 30 = 5 , 40 = 9

194 Mean = 9.23076923076923 STDdev = 4.8222855442023 When 20 = 0 , 30 = 4 , 40 = 9

193 Mean = 9.23076923076923 STDdev = 4.97924687159454 When 20 = 0 , 30 = 3 , 40 = 9

192 Mean = 9.23076923076923 STDdev = 5.24996477869279 When 20 = 0 , 30 = 2 , 40 = 9

191 Mean = 9.23076923076923 STDdev = 5.61801834969545 When 20 = 0 , 30 = 1 , 40 = 9

190 Mean = 9.23076923076923 STDdev = 6.06571507806656 When 20 = 0 , 30 = 0 , 40 = 9

189 Mean = 9.23076923076923 STDdev = 6.05302017627877 When 20 = 0 , 30 = 20 , 40 = 8

188 Mean = 9.23076923076923 STDdev = 5.79328319500392 When 20 = 0 , 30 = 19 , 40 = 8

187 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 18 , 40 = 8

186 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 17 , 40 = 8

185 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 16 , 40 = 8

184 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 15 , 40 = 8

183 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 14 , 40 = 8

182 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 13 , 40 = 8

181 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 12 , 40 = 8

180 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 11 , 40 = 8

179 Mean = 9.23076923076923 STDdev = 6.05302017627877 When 20 = 0 , 30 = 10 , 40 = 8

178 Mean = 9.23076923076923 STDdev = 5.79328319500392 When 20 = 0 , 30 = 9 , 40 = 8

177 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 8 , 40 = 8

176 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 7 , 40 = 8

175 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 6 , 40 = 8

174 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 5 , 40 = 8

173 Mean = 9.23076923076923 STDdev = 4.93268293596351 When 20 = 0 , 30 = 4 , 40 = 8

172 Mean = 9.23076923076923 STDdev = 5.05590053081692 When 20 = 0 , 30 = 3 , 40 = 8

171 Mean = 9.23076923076923 STDdev = 5.29373862587239 When 20 = 0 , 30 = 2 , 40 = 8

170 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 1 , 40 = 8

169 Mean = 9.23076923076923 STDdev = 6.05302017627877 When 20 = 0 , 30 = 0 , 40 = 8

168 Mean = 9.23076923076923 STDdev = 6.11623119442591 When 20 = 0 , 30 = 20 , 40 = 7

167 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 19 , 40 = 7

166 Mean = 9.23076923076923 STDdev = 5.64533634827962 When 20 = 0 , 30 = 18 , 40 = 7

165 Mean = 9.23076923076923 STDdev = 5.56297991746601 When 20 = 0 , 30 = 17 , 40 = 7

164 Mean = 9.23076923076923 STDdev = 5.3082496920265 When 20 = 0 , 30 = 16 , 40 = 7

163 Mean = 9.23076923076923 STDdev = 5.16130344529731 When 20 = 0 , 30 = 15 , 40 = 7

162 Mean = 9.23076923076923 STDdev = 5.1314092554332 When 20 = 0 , 30 = 14 , 40 = 7

161 Mean = 9.23076923076923 STDdev = 5.22057830798682 When 20 = 0 , 30 = 13 , 40 = 7

160 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 12 , 40 = 7

159 Mean = 9.23076923076923 STDdev = 5.72650848321069 When 20 = 0 , 30 = 11 , 40 = 7

158 Mean = 9.23076923076923 STDdev = 6.11623119442591 When 20 = 0 , 30 = 10 , 40 = 7

157 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 9 , 40 = 7

156 Mean = 9.23076923076923 STDdev = 5.64533634827962 When 20 = 0 , 30 = 8 , 40 = 7

155 Mean = 9.23076923076923 STDdev = 5.56297991746601 When 20 = 0 , 30 = 7 , 40 = 7

154 Mean = 9.23076923076923 STDdev = 5.3082496920265 When 20 = 0 , 30 = 6 , 40 = 7

153 Mean = 9.23076923076923 STDdev = 5.16130344529731 When 20 = 0 , 30 = 5 , 40 = 7

152 Mean = 9.23076923076923 STDdev = 5.1314092554332 When 20 = 0 , 30 = 4 , 40 = 7

151 Mean = 9.23076923076923 STDdev = 5.22057830798682 When 20 = 0 , 30 = 3 , 40 = 7

150 Mean = 9.23076923076923 STDdev = 5.42294053292756 When 20 = 0 , 30 = 2 , 40 = 7

149 Mean = 9.23076923076923 STDdev = 5.72650848321069 When 20 = 0 , 30 = 1 , 40 = 7

148 Mean = 9.23076923076923 STDdev = 6.11623119442591 When 20 = 0 , 30 = 0 , 40 = 7

147 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 20 , 40 = 6

146 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 19 , 40 = 6

145 Mean = 9.23076923076923 STDdev = 5.73992557113158 When 20 = 0 , 30 = 18 , 40 = 6

144 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 17 , 40 = 6

143 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 16 , 40 = 6

142 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 15 , 40 = 6

141 Mean = 9.23076923076923 STDdev = 5.40873717884521 When 20 = 0 , 30 = 14 , 40 = 6

140 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 13 , 40 = 6

139 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 12 , 40 = 6

138 Mean = 9.23076923076923 STDdev = 5.8985506192864 When 20 = 0 , 30 = 11 , 40 = 6

137 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 10 , 40 = 6

136 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 9 , 40 = 6

135 Mean = 9.23076923076923 STDdev = 5.73992557113158 When 20 = 0 , 30 = 8 , 40 = 6

134 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 7 , 40 = 6

133 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 6 , 40 = 6

132 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 5 , 40 = 6

131 Mean = 9.23076923076923 STDdev = 5.40873717884521 When 20 = 0 , 30 = 4 , 40 = 6

130 Mean = 9.23076923076923 STDdev = 5.46532912871742 When 20 = 0 , 30 = 3 , 40 = 6

129 Mean = 9.23076923076923 STDdev = 5.63169391314558 When 20 = 0 , 30 = 2 , 40 = 6

128 Mean = 9.23076923076923 STDdev = 5.8985506192864 When 20 = 0 , 30 = 1 , 40 = 6

127 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 0 , 40 = 6

126 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 20 , 40 = 5

125 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 19 , 40 = 5

124 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 18 , 40 = 5

123 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 17 , 40 = 5

122 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 16 , 40 = 5

121 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 15 , 40 = 5

120 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 14 , 40 = 5

119 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 13 , 40 = 5

118 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 12 , 40 = 5

117 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 11 , 40 = 5

116 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 10 , 40 = 5

115 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 9 , 40 = 5

114 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 8 , 40 = 5

113 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 7 , 40 = 5

112 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 6 , 40 = 5

111 Mean = 9.23076923076923 STDdev = 5.83298111080888 When 20 = 0 , 30 = 5 , 40 = 5

110 Mean = 9.23076923076923 STDdev = 5.75331136963543 When 20 = 0 , 30 = 4 , 40 = 5

109 Mean = 9.23076923076923 STDdev = 5.77998996743668 When 20 = 0 , 30 = 3 , 40 = 5

108 Mean = 9.23076923076923 STDdev = 5.91157724825872 When 20 = 0 , 30 = 2 , 40 = 5

107 Mean = 9.23076923076923 STDdev = 6.14133343268064 When 20 = 0 , 30 = 1 , 40 = 5

106 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 0 , 40 = 5

105 Mean = 9.23076923076923 STDdev = 6.72714187971552 When 20 = 0 , 30 = 20 , 40 = 4

104 Mean = 9.23076923076923 STDdev = 6.39896441325705 When 20 = 0 , 30 = 19 , 40 = 4

103 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 18 , 40 = 4

102 Mean = 9.23076923076923 STDdev = 6.00197206250189 When 20 = 0 , 30 = 17 , 40 = 4

101 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 16 , 40 = 4

100 Mean = 9.23076923076923 STDdev = 6.00197206250189 When 20 = 0 , 30 = 15 , 40 = 4

99 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 14 , 40 = 4

98 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 13 , 40 = 4

97 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 12 , 40 = 4

96 Mean = 9.23076923076923 STDdev = 6.4468697968483 When 20 = 0 , 30 = 11 , 40 = 4

95 Mean = 9.23076923076923 STDdev = 6.72714187971552 When 20 = 0 , 30 = 10 , 40 = 4

94 Mean = 9.23076923076923 STDdev = 6.39896441325705 When 20 = 0 , 30 = 9 , 40 = 4

93 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 8 , 40 = 4

92 Mean = 9.23076923076923 STDdev = 6.0019720625019 When 20 = 0 , 30 = 7 , 40 = 4

91 Mean = 9.23076923076923 STDdev = 5.95048603255807 When 20 = 0 , 30 = 6 , 40 = 4

90 Mean = 9.23076923076923 STDdev = 6.00197206250189 When 20 = 0 , 30 = 5 , 40 = 4

89 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 4 , 40 = 4

88 Mean = 9.23076923076923 STDdev = 6.15384615384615 When 20 = 0 , 30 = 3 , 40 = 4

87 Mean = 9.23076923076923 STDdev = 6.25304659473895 When 20 = 0 , 30 = 2 , 40 = 4

86 Mean = 9.23076923076923 STDdev = 6.4468697968483 When 20 = 0 , 30 = 1 , 40 = 4

85 Mean = 9.23076923076923 STDdev = 6.72714187971552 When 20 = 0 , 30 = 0 , 40 = 4

84 Mean = 9.23076923076923 STDdev = 7.05095570340368 When 20 = 0 , 30 = 20 , 40 = 3

83 Mean = 9.23076923076923 STDdev = 6.71569741098989 When 20 = 0 , 30 = 19 , 40 = 3

82 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 18 , 40 = 3

81 Mean = 9.23076923076923 STDdev = 6.28984341438758 When 20 = 0 , 30 = 17 , 40 = 3

80 Mean = 9.23076923076923 STDdev = 6.21603195410367 When 20 = 0 , 30 = 16 , 40 = 3

79 Mean = 9.23076923076923 STDdev = 6.24073277445287 When 20 = 0 , 30 = 15 , 40 = 3

78 Mean = 9.23076923076923 STDdev = 6.36279868458776 When 20 = 0 , 30 = 14 , 40 = 3

77 Mean = 9.23076923076923 STDdev = 6.57681061532279 When 20 = 0 , 30 = 13 , 40 = 3

76 Mean = 9.23076923076923 STDdev = 6.64661679299322 When 20 = 0 , 30 = 12 , 40 = 3

75 Mean = 9.23076923076923 STDdev = 6.80671440173198 When 20 = 0 , 30 = 11 , 40 = 3

74 Mean = 9.23076923076923 STDdev = 7.05095570340369 When 20 = 0 , 30 = 10 , 40 = 3

73 Mean = 9.23076923076923 STDdev = 6.71569741098989 When 20 = 0 , 30 = 9 , 40 = 3

72 Mean = 9.23076923076923 STDdev = 6.45879062451795 When 20 = 0 , 30 = 8 , 40 = 3

71 Mean = 9.23076923076923 STDdev = 6.28984341438758 When 20 = 0 , 30 = 7 , 40 = 3

70 Mean = 9.23076923076923 STDdev = 6.21603195410367 When 20 = 0 , 30 = 6 , 40 = 3

69 Mean = 9.23076923076923 STDdev = 6.24073277445287 When 20 = 0 , 30 = 5 , 40 = 3

68 Mean = 9.23076923076923 STDdev = 6.36279868458776 When 20 = 0 , 30 = 4 , 40 = 3

67 Mean = 9.23076923076923 STDdev = 6.57681061532279 When 20 = 0 , 30 = 3 , 40 = 3

66 Mean = 9.23076923076923 STDdev = 6.64661679299322 When 20 = 0 , 30 = 2 , 40 = 3

65 Mean = 9.23076923076923 STDdev = 6.80671440173198 When 20 = 0 , 30 = 1 , 40 = 3

64 Mean = 9.23076923076923 STDdev = 7.05095570340368 When 20 = 0 , 30 = 0 , 40 = 3

63 Mean = 9.23076923076923 STDdev = 7.42297728111681 When 20 = 0 , 30 = 20 , 40 = 2

62 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 19 , 40 = 2

61 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 18 , 40 = 2

60 Mean = 9.23076923076923 STDdev = 6.63503343164548 When 20 = 0 , 30 = 17 , 40 = 2

59 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 16 , 40 = 2

58 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 15 , 40 = 2

57 Mean = 9.23076923076923 STDdev = 6.63503343164549 When 20 = 0 , 30 = 14 , 40 = 2

56 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 13 , 40 = 2

55 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 12 , 40 = 2

54 Mean = 9.23076923076923 STDdev = 7.21274348906526 When 20 = 0 , 30 = 11 , 40 = 2

53 Mean = 9.23076923076923 STDdev = 7.42297728111681 When 20 = 0 , 30 = 10 , 40 = 2

52 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 9 , 40 = 2

51 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 8 , 40 = 2

50 Mean = 9.23076923076923 STDdev = 6.63503343164549 When 20 = 0 , 30 = 7 , 40 = 2

49 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 6 , 40 = 2

48 Mean = 9.23076923076923 STDdev = 6.54162819245209 When 20 = 0 , 30 = 5 , 40 = 2

47 Mean = 9.23076923076923 STDdev = 6.63503343164548 When 20 = 0 , 30 = 4 , 40 = 2

46 Mean = 9.23076923076923 STDdev = 6.81800609420318 When 20 = 0 , 30 = 3 , 40 = 2

45 Mean = 9.23076923076923 STDdev = 7.08360888198236 When 20 = 0 , 30 = 2 , 40 = 2

44 Mean = 9.23076923076923 STDdev = 7.21274348906526 When 20 = 0 , 30 = 1 , 40 = 2

43 Mean = 9.23076923076923 STDdev = 7.42297728111681 When 20 = 0 , 30 = 0 , 40 = 2

42 Mean = 9.23076923076923 STDdev = 7.83634379182464 When 20 = 0 , 30 = 20 , 40 = 1

41 Mean = 9.23076923076923 STDdev = 7.49516609508418 When 20 = 0 , 30 = 19 , 40 = 1

40 Mean = 9.23076923076923 STDdev = 7.22340050065753 When 20 = 0 , 30 = 18 , 40 = 1

39 Mean = 9.23076923076923 STDdev = 7.02910264711426 When 20 = 0 , 30 = 17 , 40 = 1

38 Mean = 9.23076923076923 STDdev = 6.9188020990058 When 20 = 0 , 30 = 16 , 40 = 1

37 Mean = 9.23076923076923 STDdev = 6.89653029990551 When 20 = 0 , 30 = 15 , 40 = 1

36 Mean = 9.23076923076923 STDdev = 6.96313198931283 When 20 = 0 , 30 = 14 , 40 = 1

35 Mean = 9.23076923076923 STDdev = 7.11611222890968 When 20 = 0 , 30 = 13 , 40 = 1

34 Mean = 9.23076923076923 STDdev = 7.35007949882539 When 20 = 0 , 30 = 12 , 40 = 1

33 Mean = 9.23076923076923 STDdev = 7.65761407061219 When 20 = 0 , 30 = 11 , 40 = 1

32 Mean = 9.23076923076923 STDdev = 7.83634379182465 When 20 = 0 , 30 = 10 , 40 = 1

31 Mean = 9.23076923076923 STDdev = 7.49516609508418 When 20 = 0 , 30 = 9 , 40 = 1

30 Mean = 9.23076923076923 STDdev = 7.22340050065753 When 20 = 0 , 30 = 8 , 40 = 1

29 Mean = 9.23076923076923 STDdev = 7.02910264711426 When 20 = 0 , 30 = 7 , 40 = 1

28 Mean = 9.23076923076923 STDdev = 6.9188020990058 When 20 = 0 , 30 = 6 , 40 = 1

27 Mean = 9.23076923076923 STDdev = 6.89653029990551 When 20 = 0 , 30 = 5 , 40 = 1

26 Mean = 9.23076923076923 STDdev = 6.96313198931283 When 20 = 0 , 30 = 4 , 40 = 1

25 Mean = 9.23076923076923 STDdev = 7.11611222890968 When 20 = 0 , 30 = 3 , 40 = 1

24 Mean = 9.23076923076923 STDdev = 7.35007949882539 When 20 = 0 , 30 = 2 , 40 = 1

23 Mean = 9.23076923076923 STDdev = 7.65761407061219 When 20 = 0 , 30 = 1 , 40 = 1

22 Mean = 9.23076923076923 STDdev = 7.83634379182464 When 20 = 0 , 30 = 0 , 40 = 1

21 Mean = 9.23076923076923 STDdev = 8.28486893405308 When 20 = 0 , 30 = 20 , 40 = 0

20 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 19 , 40 = 0

19 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 18 , 40 = 0

18 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 17 , 40 = 0

17 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 16 , 40 = 0

16 Mean = 9.23076923076923 STDdev = 7.2975638311578 When 20 = 0 , 30 = 15 , 40 = 0

15 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 14 , 40 = 0

14 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 13 , 40 = 0

13 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 12 , 40 = 0

12 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 11 , 40 = 0

11 Mean = 9.23076923076923 STDdev = 8.28486893405308 When 20 = 0 , 30 = 10 , 40 = 0

10 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 9 , 40 = 0

9 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 8 , 40 = 0

8 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 7 , 40 = 0

7 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 6 , 40 = 0

6 Mean = 9.23076923076923 STDdev = 7.2975638311578 When 20 = 0 , 30 = 5 , 40 = 0

5 Mean = 9.23076923076923 STDdev = 7.33960642577019 When 20 = 0 , 30 = 4 , 40 = 0

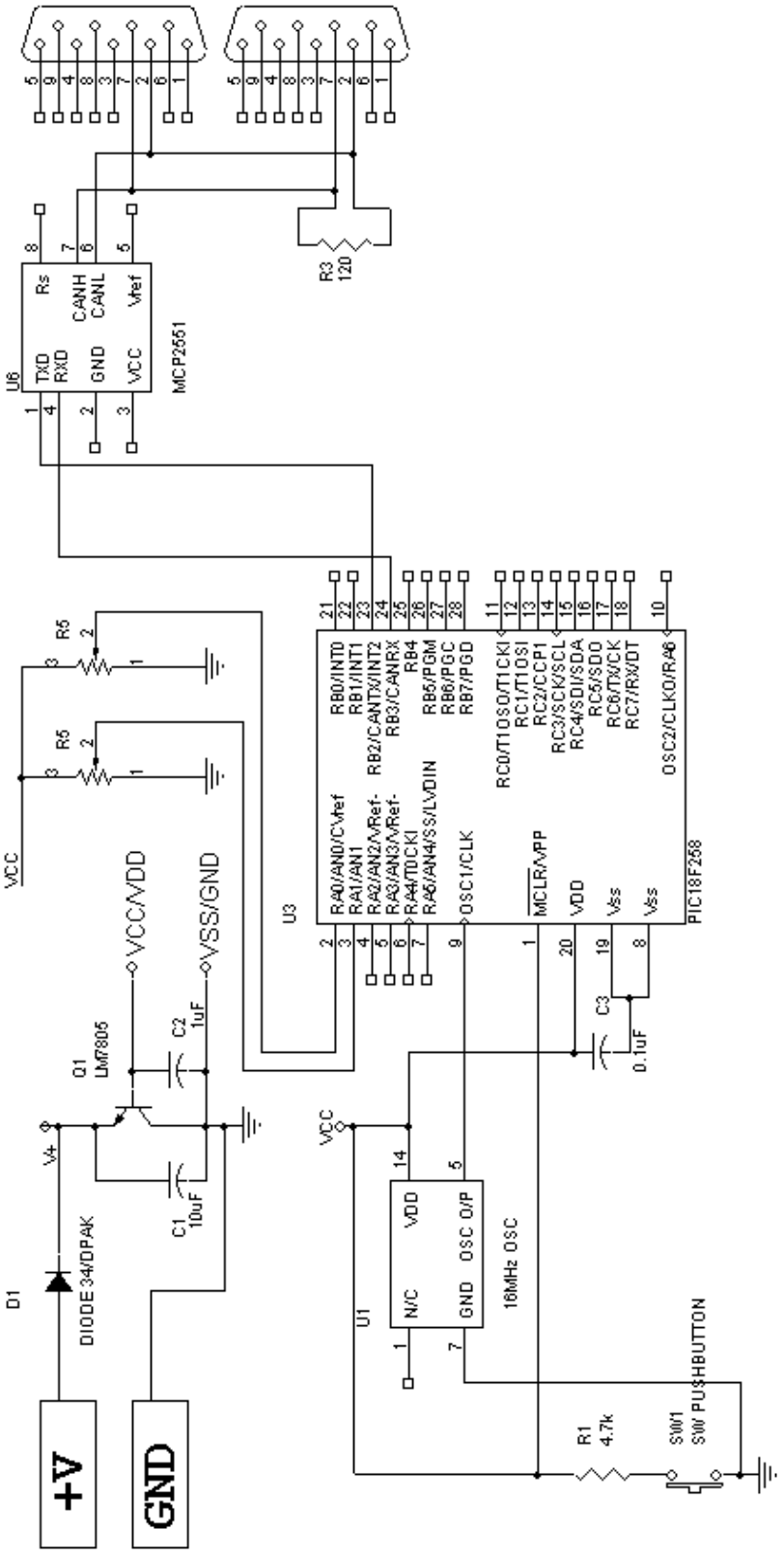
4 Mean = 9.23076923076923 STDdev = 7.46431352043582 When 20 = 0 , 30 = 3 , 40 = 0

3 Mean = 9.23076923076923 STDdev = 7.66765279653976 When 20 = 0 , 30 = 2 , 40 = 0

2 Mean = 9.23076923076923 STDdev = 7.9435880882619 When 20 = 0 , 30 = 1 , 40 = 0

1 Mean = 9.23076923076923 STDdev = 8.28486893405308 When 20 = 0 , 30 = 0 , 40 = 0

Appendix 7: TTCAN Node Schematic



Appendix 8: Embedded 'C' Software

```
1   //*********************************************************************
2   //PIC microcontroller P18F2480.
3   //
4   // Master in Engineering
5   //
6   //      "Master Node sending Message every 20ms for a matrix 600ms long"
7   //
8   // Written by Henry Acheson of Advanced Automotive Electronic Control Group,
9   //                  WIT, Cork Road, Waterford, Ireland.
10  //
11  //Date: 06-02-2007
12  //Version 1.00
13  //*********************************************************************
14  unsigned count;
15  char aa, aa1, lenn, i;
16  char data[8];
17  long id;
18   void interrupt()
19  {
20  TMR0L  = 0xdb;
21  TMR0H = 0xB1;
22   INTCON = 0x20;     // Set T0IE, clear T0IF
23   i++;              // Increment value of "i" on every interrupt
24  count = i;
25   }
26  void main()
27  {
28   i = 0;
29  count = 1;
30  T0CON = 0x80;       // Assign prescaler to TMR0
31  TMR0L = 0xd3;
32  TMR0H = 0x9e;
33
34  aa = 0;
35  aa1 = 0;
36  aa = CAN_CONFIG_SAMPLE_THRICE & CAN_CONFIG_PHSEG2_PRG_ON &
37  CAN_CONFIG_ALL_MSG & CAN_CONFIG_DBL_BUFFER_ON &
38  CAN_CONFIG_LINE_FILTER_OFF;              //Used with CANInitialize
```

```
39      aa1 = CAN_TX_PRIORITY_0 & CAN_TX_STD_FRAME &
40      CAN_TX_NO_RTR_FRAME;                    //Used with CANSendMessage
41
42      //CAN Baudrate = 4 = 125kbits/sec 8MHz clock
43      CANInitialize(2,4,3,3,1,aa);            //Initialize CAN Controller
44      CANSetOperationMode(CAN_MODE_NORMAL,0);    //Configure Normal Mode
45      data [0] = 1;
46      data [2] = 0;
47      data [3] = 0;
48      data [4] = 0;
49      data [5] = 0;
50      data [6] = 0;
51      id = 20;
52      lenn = 7;
53      INTCON = 0xA0;      // Enable TMRO interrupt
54              do
55              {
56              if (count == i)
57                      {
58                      data [1] = i;
59                      CANWRITE(id,data,lenn,aa1);
60                      count = count + 1;
61                              if (count == 31)
62                              {
63                              count = 1;
64                              i = 0;
65                              }
66                      }
67              }
68      while(1);       // loop
69      }//~!
```

Appendix 9: Write Data for Message Periods 20ms and 30ms.

SystemStart of measurement 05:13:51 pm

SystemCAN 1 Bus with 125000 BPS.

System-----

SystemStatistics report AR0038, 05:13:51 pm

SystemStatistics for transmit spacing of messages in [ms]

System

System		N	Aver	StdDev	MIN	MAX	
System20	RX	7574	20.005	0.0093935	19.98	20.03	CAN 1
System30	RX	5049	30.007	0.017355	29.98	30.04	CAN 1

System

SystemEnd of measurement 05:16:23 pm

Appendix 10: Write Data for Message Periods 20ms, 30ms and 40ms.

SystemStart of measurement 08:32:41 pm

SystemCAN 1 Bus with 125000 BPS.

System-----

SystemStatistics report AR0059, 08:32:41 pm

SystemStatistics for transmit spacing of messages in [ms]

System

System		N	Aver	StdDev	MIN	MAX	
System20	RX	9296	20.005	0.01305	19.97	20.04	CAN 1
System30	RX	6197	30.007	0.028232	29.93	30.06	CAN 1
System40	RX	4648	40.01	0.025572	39.93	40.08	CAN 1

System

SystemEnd of measurement 08:35:47 pm

Appendix 11: Write Data for Message Periods 20ms, 30ms, 40ms and 50ms.

SystemStart of measurement 01:13:17 pm

SystemCAN 1 Bus with 125000 BPS.

System-----

SystemStatistics report AR0042, 01:13:17 pm

SystemStatistics for transmit spacing of messages in [ms]

System

System		N	Aver	StdDev	MIN	MAX	
System20	RX	7885	20.011	0.0088368	19.98	20.03	CAN 1
System30	RX	5256	30.016	0.027868	29.93	30.09	CAN 1
System40	RX	3942	40.022	0.017549	39.97	40.08	CAN 1
System50	RX	3154	50.027	0.027818	49.93	50.10	CAN 1

System

SystemEnd of measurement 01:15:55 pm