



Cost Efficient Dependable Electronic Systems

# **TTCAN Reference Application**

## **An investigation of time-triggered network performance**

Author	Mikael Fernström and Daniel Unger Dahl
Document Id	015
Date	15 June 2006
Availability	Public
Status	Final



# CHALMERS



## TTCAN Reference Application An investigation of time-triggered network performance

MIKAEL FERNSTRÖM  
DANIEL UNGERDAHL

*Master's Thesis*

*Computer Science and Engineering Program*

CHALMERS UNIVERSITY OF TECHNOLOGY  
Department of Computer Science and Engineering  
Division of Computer Engineering  
Göteborg 2006

All rights reserved. This publication is protected by law in accordance with "Lagen om Upphovsrätt, 1960:729". No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the authors.

© Mikael Fernström, Daniel Unger Dahl, Göteborg 2006.

## **ABSTRACT**

The mechanical and hydraulic systems found in vehicles today will in the future be replaced by electronic drive-by-wire systems. These safety-critical, distributed systems will put higher demands on the network communication. Providing deterministic behaviour and guaranteed response times, time-triggered communication systems are predicted to take the place of the currently used event-triggered architectures. There are recently developed protocols and hardware for time-triggered communication, but no studies of actual network behaviour and performance have been conducted.

The objective of this project was to develop a reference application for the time-triggered protocol TTCAN, and then use the application to gather information on the behaviour and performance of a simple network. To meet the objective, a basic driver package for the hardware used had to be developed. Also, a graphical tool to aid configuration and debugging was constructed. It was named Comtest, short for Compact TTCAN Environment Setup Tool. Based on this software, the reference application could then be developed and provided with a battery of test configurations. The test cases were executed on the hardware and the results were compiled and analysed.

The tested time-triggered network consistently provided good predictability and also acceptable performance when run with well-designed communication schedules. The effective bandwidth was then comparable with the bandwidth of event-triggered CAN communication. Tests with schedules that were badly suited for the lengths of the sent messages demonstrated risks of long gaps in the communication and large performance deficiencies. This result points out the importance of analysing the network traffic and adapting the schedules accordingly.

## **SAMMANFATTNING**

De mekaniska och hydrauliska system som sitter i dagens fordon kommer i framtiden att ersättas av elektroniska drive-by-wiresystem. Dessa säkerhetskritiska, distribuerade system kommer att ställa högre krav på nätverkskommunikationen. Med sitt deterministiska beteende och sina garanterade svarstider väntas tidsstyrda kommunikationssystem ersätta de händelsestyrda system som används idag. Det finns nyligen utvecklade protokoll och hårdvara för tidsstyrd kommunikation, men inga studier av faktiskt beteende och prestanda har genomförts.

Målsättningen med detta examensarbete var att utveckla en referensapplikation för det tidsstyrda protokollet TTCAN och sedan använda denna för att samla in information om beteende och prestanda hos ett enkelt nätverk. För att kunna uppnå målet utvecklades ett drivrutinspaket för den aktuella hårdvaran. Dessutom skapades ett grafiskt verktyg för att underlätta konfiguration och debugging. Verktöget fick namnet Comtest, förkortning för Compact TTCAN Environment Setup Tool. Med denna mjukvara som grund kunde sedan referensapplikationen utvecklas och förses med en uppsättning av testkonfigurationer. Testerna kördes på hårdvaran och resultaten samlades in och analyserades.

Det testade nätverket uppvisade genomgående god förutsägbarhet och även godtagbar prestanda när det kördes med välanpassade kommunikationsscheman. Den effektiva bandbredden var då i paritet med bandbredden för händelsestyrd CAN-kommunikation. Tester med scheman som var dåligt anpassade till längden på de meddelanden som skickades visade på risker för långa luckor i kommunikationen och nedsatt prestanda. Detta resultat visar hur viktigt det är att analysera nätverkstrafiken och anpassa schemana därefter.

## **PREFACE AND ACKNOWLEDGEMENTS**

This master's thesis concludes our Master of Science in Computer Engineering education, with focus on embedded systems, at Chalmers University of Technology. The project is a part of the CEDES project and was carried out at Open Arena Lindholmen in Göteborg. This thesis can, together with other related material, be found in electronic form at [www.cedes.se](http://www.cedes.se). Information on the GAST project is available at [www.chl.chalmers.se/gast](http://www.chl.chalmers.se/gast).

We would especially like to thank our supervisor at Chalmers and CEDES, Roger Johansson, for his constructive ideas and for sharing his expertise. Roger has provided great support throughout our education, including several courses and two theses.

More thanks go to our brothers in arms, Christian Archer and Andreas Sjöblom, who bravely covered our backs during The TTCAN Wars and also let us make fun of their opinion of a working day.

### **Thanks also to:**

Jonas Eriksson and Magnus Källvik for the nice company and for lending us some hardware when we had shortages. These two electrical engineers also truly deserve credit for taming the elusive G2 processor board.

Håkan Edler and all other CEDES people for welcoming us into the project and arranging interesting seminars.

The staff at Lindholmen Science Park for their friendliness and for providing us with alarm codes when the project stretched into the weekends.

The restaurant Tres for the wonderful ciabatta sandwiches at the CEDES seminars.

Danska bageriet for their tasty pastries.

The weather of late May 2006 for keeping us concentrated and not longing to get on the golf course.



## **GLOSSARY**

<b>application watchdog</b>	A timer that the application periodically has to reset to signal that it is still alive.
<b>arbitrating time window</b>	Time window where several nodes has access to the bus. Collisions are resolved by CAN arbitration.
<b>arbitration</b>	Procedure to determine which of the nodes that are attempting to transmit on the bus that has the highest priority.
<b>arbitration field</b>	The part of a CAN frame that holds the identifier.
<b>base size</b>	The time window size used by all reference application schedules. At 500 kbps the base size was 320 NTUs.
<b>basic cycle</b>	A part of the matrix cycle that consists of consecutive time windows. A reference message is sent at the start of each basic cycle.
<b>bit flip</b>	The undesired change of a bit from a logical ‘zero’ to a ‘one’, or vice versa.
<b>bit stuffing</b>	Technique used in the CAN protocol to prevent nodes from going out of synchronisation. After five consecutive identical bits, a bit of opposite level is inserted to produce a flank on the bus.
<b>bus</b>	Communication medium. In distributed automotive systems, the bus often consists of two copper wires.
<b>bus guardian</b>	A unit that prevents failing nodes in a time-triggered network from transmitting on the bus at the wrong time.
<b>bus load</b>	The percentage of the time that the bus is occupied.
<b>calculated minimal bus load</b>	The least possible bus load given a time period, a number of messages and the shortest possible length of each message.
<b>CAN</b>	Controller Area Network. An event-triggered communication protocol widely used in automotive applications. See section 2.2.1.1.
<b>CAN data frame</b>	A sequence of between 64 and 128 bits that is transferred on the bus and organised as specified in the CAN protocol specification. All CAN frames are followed by 3 bits of interframe space.
<b>CAN message</b>	A CAN data frame or remote frame. Remote frames, which are requests for data frames, are not further described in this thesis.
<b>CEDES</b>	Cost Efficient Dependable Electronic Systems. A project aiming at developing techniques and methods to design and build cost efficient dependable automotive electronics. See section 2.4.

<b>Comtest</b>	Compact TTCAN Environment Setup Tool. A graphical configuration tool for TTCAN developed in this project. See chapter 4.
<b>control-by-wire</b>	The class of applications where physical connections are replaced by electronic signals.
<b>CPU</b>	Central Processing Unit. The processor of a computer, micro-controller etc.
<b>CSMA/CA</b>	Carrier Sense Multiple Access with Collision Avoidance. One principle of event-triggered communication. See section 2.2.1.
<b>CSMA/CD</b>	Carrier Sense Multiple Access with Collision Detection. One principle of event-triggered communication. See section 2.2.1.
<b>dense schedules</b>	The class of TT/ET schedules where the exclusive time windows are concentrated to the start of each basic cycle.
<b>distributed system</b>	A system where applications are spread over several independent and communicating nodes.
<b>dominant bit</b>	A bit that, when sent on the CAN bus, causes the whole bus to adapt to its level.
<b>drive-by-wire</b>	The class of applications in road vehicles where physical connections are replaced by electronic signals.
<b>effective bandwidth</b>	The percentage of the bandwidth that is used to transfer payload data.
<b>effective bitrate</b>	The mean transfer rate of payload data, given a time period and the data amount transferred that same period.
<b>ECU</b>	Electronic Control Unit. A microcontroller unit in, for instance, a control-by-wire system.
<b>ET</b>	Event-triggered.
<b>ET message</b>	A message sent in an arbitrating time window.
<b>ET schedule</b>	A schedule only consisting of arbitrating time windows.
<b>event-triggered</b>	Triggered by the occurrence of an event, rather than of a certain point in time.
<b>exclusive time window</b>	Time window where one node has exclusive access to the bus.
<b>FlexRay</b>	A time-triggered communication protocol, primarily designed for automotive applications. See section 2.2.2.2.
<b>G1</b>	GAST processor board based on a HCS12 with 8 MHz clock frequency.
<b>G2</b>	GAST processor board based on a MPC565 PowerPC with 40 MHz clock frequency.

<b>GAST</b>	General Application Development Boards for Safety Critical Time-Triggered Systems. A project that has developed hardware for distributed control-by-wire applications. See section 2.3.
<b>identifier</b>	A sequence of dominant and recessive bits used for identification and prioritisation of a CAN message.
<b>kbps</b>	Kilobits per second. A transfer rate of 1 kbps equals 1,000 bits per second.
<b>KiB</b>	Kibibyte. 1 KiB equals 1024 bytes.
<b>matrix cycle</b>	The TTCAN communication cycle. Consists of one or more basic cycles.
<b>Mbps</b>	Megabits per second. A transfer rate of 1 Mbps equals 1,000,000 bits per second.
<b>membership</b>	A field of study concerning how to achieve a common view of the state of all tasks in a distributed system.
<b>message object</b>	The collected information of one TTCAN message. Resides in the Message RAM of the TTCAN chip.
<b>Message RAM</b>	Memory on the TTCAN chip that contains the message objects.
<b>mini-slots</b>	Time windows in the dynamic segment of the FlexRay communication cycle. See section 2.2.2.2.
<b>NTU</b>	Network Time Unit. The TTCAN time unit. Has to have the same duration at all nodes in the network. In this project, the NTU was set to 1 microsecond.
<b>node</b>	A physical unit connected to the network. In this thesis, a node is assumed to have only one communication controller and to be running one single task.
<b>overhead</b>	Any combination of excess or indirect computation time, memory, bandwidth or other resources consumed to enable a particular goal.
<b>recessive bit</b>	A bit that, when sent on the CAN bus, does not cause the whole bus to adapt to its level.
<b>reference message</b>	Message sent by the current time master at the start of every basic cycle in a TTCAN schedule.
<b>response time</b>	The time a system or a functional unit takes to react to a given input.
<b>RAM</b>	Random Access Memory. See RWM.
<b>RWM</b>	Read/Write Memory. See RAM.
<b>SAE</b>	Society of Automotive Engineers. A professional organization and standards body for the engineering of powered vehicles.

<b>sparse schedules</b>	The class of TT/ET schedules where every exclusive time window is followed by an arbitrating time window.
<b>starvation</b>	A situation in which a task never is completed because a needed resource is constantly occupied by higher prioritised tasks.
<b>stuff bit</b>	A bit inserted in a CAN frame to prevent nodes from going out of synchronisation. After five consecutive identical bits, a stuff bit of opposite level is inserted to produce a flank on the bus.
<b>TDMA</b>	Time Division Multiple Access. The principle of time-triggered communication with exclusive time windows. See section 2.2.2.
<b>time-triggered</b>	Triggered by a certain point in time, rather than of the occurrence of an event.
<b>time window</b>	A limited period of time connected to certain activities on the bus.
<b>trigger</b>	Property of a TTCAN schedule, specifying an action to be taken after a certain elapsed number of NTUs in a basic cycle.
<b>TT</b>	Time-triggered.
<b>TT message</b>	A message sent in an exclusive time window.
<b>TT schedule</b>	A schedule only consisting of exclusive time windows.
<b>TT/ET schedule</b>	A schedule evenly divided into exclusive time windows and arbitrating time windows.
<b>TTCAN</b>	Time-Triggered CAN. A higher layer, time-triggered protocol placed on top of the CAN protocol. See section 2.2.2.1.
<b>TTP/C</b>	Time-Triggered Protocol for SAE class C. A time-triggered protocol that provides some fault-tolerance services. See section 2.2.2.3.
<b>TUR</b>	Time Unit Ratio. The ratio between the length of an NTU and the length of the system clock period. In this project, the TUR was set to 8.

# TABLE OF CONTENTS

<b>1. INTRODUCTION</b> .....	<b>1</b>
1.1. BACKGROUND.....	1
1.2. PROBLEM STATEMENT .....	1
1.3. PURPOSE AND SCOPE.....	1
1.4. DELIMITATIONS .....	1
1.5. METHOD .....	1
1.6. THESIS OUTLINE.....	2
<b>2. BACKGROUND</b> .....	<b>3</b>
2.1. DRIVE-BY-WIRE.....	3
2.2. COMMUNICATION IN DISTRIBUTED SYSTEMS.....	3
2.2.1. <i>EVENT-TRIGGERED ARCHITECTURES</i> .....	4
2.2.2. <i>TIME-TRIGGERED ARCHITECTURES</i> .....	4
2.3. THE GAST PROJECT.....	6
2.4. THE CEDES PROJECT .....	6
<b>3. THE TTCAN DRIVERS</b> .....	<b>8</b>
3.1. REQUIREMENTS .....	8
3.2. DESIGN CONCEPTS .....	8
3.2.1. <i>BASE ADDRESS POINTERS</i> .....	8
3.2.2. <i>PARAMETER AND CODE COMPONENT SEPARATION</i> .....	9
3.2.3. <i>NAMES AND ORDER</i> .....	9
3.2.4. <i>LAYERED CODE APPROACH</i> .....	9
3.2.5. <i>NON-BLOCKING MESSAGE FUNCTIONS</i> .....	10
3.3. LIMITING DECISIONS .....	10
3.3.1. <i>FIXED PARAMETERS</i> .....	10
3.3.2. <i>NO REGISTER CONTENT VERIFICATION</i> .....	10
3.3.3. <i>FORCED MESSAGE WRITES</i> .....	11
3.3.4. <i>ONLY IF1 USAGE</i> .....	11
3.3.5. <i>NO PARALLEL MESSAGE RECEPTION CHECKS</i> .....	11
3.4. CENTRAL FUNCTIONS .....	11
<b>4. THE TTCAN CONFIGURATION TOOL</b> .....	<b>13</b>
4.1. REQUIREMENTS .....	13
4.2. LIMITING DECISIONS .....	14
4.3. DESIGN AND FEATURES .....	14
4.3.1. <i>THE GENERAL TAB</i> .....	14
4.3.2. <i>THE MESSAGE OBJECTS TAB</i> .....	16
4.3.3. <i>THE TRIGGERS TAB</i> .....	17
4.3.4. <i>SAVING AND LOADING CONFIGURATIONS</i> .....	18
4.4. CODE GENERATION .....	19
4.4.1. <i>REGISTER VALUE CALCULATIONS</i> .....	19
<b>5. THE REFERENCE APPLICATION</b> .....	<b>20</b>
5.1. MAIN CONCEPT .....	20
5.2. MESSAGE NOTATION .....	20

5.3. SCHEDULE PROPERTIES .....	20
5.3.1. <i>TIME WINDOW SIZE</i> .....	20
5.3.2. <i>SCHEDULE CLASSES</i> .....	22
5.3.3. <i>SCHEDULE VARIABLES</i> .....	22
5.4. THE NODES .....	24
5.4.1. <i>THE SENDER NODES</i> .....	24
5.4.2. <i>THE RECEIVER NODE</i> .....	25
<b>6. RESULTS AND ANALYSIS .....</b>	<b>27</b>
6.1. EXCLUDED TEST CASES .....	27
6.2. THE PRE-TEST ANALYSIS.....	27
6.3. TESTS WITH DIFFERENT BITRATES .....	27
6.4. THE NUMBER OF TRANSFERRED MESSAGES.....	28
6.4.1. <i>COMPARISON OF THE SCHEDULE CLASSES</i> .....	28
6.4.2. <i>COMPARISON OF THE NUMBER OF BASIC CYCLES</i> .....	28
6.4.3. <i>COMPARISON OF THE NUMBER OF DATA BYTES</i> .....	28
6.5. EFFECTIVE BANDWIDTH .....	30
6.5.1. <i>COMPARISON OF THE SCHEDULE CLASSES</i> .....	30
6.5.2. <i>COMPARISON OF THE NUMBER OF BASIC CYCLES</i> .....	30
6.5.3. <i>COMPARISON OF THE NUMBER OF DATA BYTES</i> .....	30
6.6. CALCULATED MINIMAL BUS LOAD .....	32
6.7. RESULTS FOR EACH SENDER NODE .....	34
<b>7. PROBLEMS ENCOUNTERED.....</b>	<b>36</b>
7.1. NETWORK CONFIGURATION AND START-UP .....	36
7.2. THE BACKPLANE BUS.....	36
7.2.1. <i>DOUBLE MEMORY ACCESS</i> .....	37
7.2.2. <i>SHORTER POINT-TO-POINT BUS</i> .....	37
7.3. SENSITIVE MESSAGE OBJECTS .....	37
7.4. FUNCTION CALL ORDER .....	38
<b>8. CONCLUSION .....</b>	<b>39</b>
<b>REFERENCES.....</b>	<b>40</b>
<b>APPENDICES</b>	
A. RESULTS: NUMBER OF TRANSFERRED MESSAGES	
B. RESULTS: EFFECTIVE BANDWIDTH	
C. RESULTS: CALCULATED MINIMAL BUS LOAD	
D. REFERENCE APPLICATION SCHEDULES	
E. COMTEST TUTORIAL	
F. RUNNING THE TUTORIAL APPLICATION	
G. CALCULATION OF BIT TIMING PARAMETERS	

## **1. INTRODUCTION**

### **1.1. Background**

Within a few years, the automotive industry will stand before a major shift in technology. The mechanical and hydraulic systems found in vehicles today will in the future be replaced by electronic drive-by-wire systems. These safety-critical, distributed systems will put new and higher demands on the communication between nodes. Drive-by-wire applications will require deterministic behaviour with guaranteed response times. This introduces the need for new ways to communicate, since the event-triggered communication systems found in today's vehicles does not provide the desired level of service. Efforts are therefore made to develop systems for time-triggered communication between synchronised nodes. With such systems, a bus can periodically be exclusively dedicated to nodes that critically need to transfer their messages at precise times. Then, message transfer times can be guaranteed and safety-critical applications will function as intended. Three protocols for time-triggered communication that are available are TTCAN, FlexRay and TTP/C. The recently concluded GAST project has developed processor and communication boards for implementations of these protocols. This hardware forms the base for a test bed in the CEDES project, which aims to develop cost efficient dependable electronic systems for the automotive industry. All hardware and software developed for time-triggered protocols are still in their prototype stages and no studies of actual network behaviour and performance have been conducted.

### **1.2. Problem statement**

Within the CEDES project, an investigation of the performance of a time-triggered network would be regarded as interesting. Such an investigation requires a reference application, collecting experimental data while testing a battery of different network settings.

### **1.3. Purpose and scope**

The objective of this project was to develop a reference application for the time-triggered protocol TTCAN, and then use the application to gather information on the behaviour and performance of a simple network. The implementation should be developed using GAST hardware.

### **1.4. Delimitations**

The project is limited to developing, running and analysing the results of a TTCAN network reference application. Any other software developed will primarily be considered as tools to meet the project purpose. No comparisons of system performance with other time-triggered protocols, such as FlexRay and TTP/C, will be made.

### **1.5. Method**

From the hardware developed by the GAST project, only the G1 processor board and the TTCAN and TTP/C communication boards were available. Due to its capabilities of mixed time-triggered and event-triggered communication, TTCAN was chosen for the realisation of the reference application. Also, there was some software that earlier projects had developed for the G1 and TTCAN boards available. To be able to start developing the reference application, a driver package for the G1/TTCAN configuration, supplying some basic functionality, would be necessary. The drivers should provide means to configure and start a TTCAN network. Some basic message handling functionality was also required. The already available code had been developed specifically for the projects in question and to cleanly extract the parts needed for this project was expected to be rather difficult. Furthermore; simple, appropriate and well-documented drivers would greatly aid future projects. Therefore, it was deci-

ded that a simple driver package for the G1 and TTCAN boards should be developed from scratch. Partly in parallel with the driver package development, a graphical configuration tool was developed. The purpose of this was to facilitate organisation of the large number of parameters connected to a TTCAN configuration. Also, it would make debugging significantly more straightforward. The tool was named Comtest, short for Compact TTCAN Environment Setup Tool. When both the drivers and Comtest had been successfully finished, the development of the reference application was initiated. Interesting TTCAN system configuration properties were identified and incorporated as variables in the application. When the application was complete, a battery of test configurations was constructed and executed on the hardware. The results were compiled and analysed, which resulted in a number of conclusions regarding the network behaviour and performance. The project was then documented, resulting in this thesis. While documenting, the intentions were to introduce the areas concerned sufficiently enough to understand the project context, and facilitate future projects intending to extend or repeat the results.

## 1.6. Thesis outline

The thesis is divided into the following sections:

- **Background**  
Background information on topics related to the project. The drive-by-wire concept, distributed system communications, a few time-triggered protocols and related research projects are introduced.
- **The TTCAN drivers**  
Requirements, design concepts and delimitations of the developed driver package.
- **The TTCAN configuration tool**  
Requirements and functionality of the developed configuration tool Comtest.
- **The reference application**  
Information about the design and properties of the reference application.
- **Results and analysis**  
The data collected from the test sessions is here accounted for and analysed.
- **Problems encountered**  
An account of the problems encountered during the course of the project.
- **Conclusion**  
The conclusions drawn from the analysis of the reference application results.
- **Appendices**  
Result data tables, reference application schedules, tutorials and some code.

## 2. BACKGROUND

This chapter is intended to give an overview of the background of the thesis. It includes introductions to the areas of drive-by-wire technology and distributed system communications. It also briefly describes some communication protocols designed for usage in vehicles today and in the future, and is concluded with information on two research projects related to this thesis.

### 2.1. Drive-by-wire

In a *drive-by-wire* system, direct mechanical control of a vehicle is replaced by electronic control. Drive-by-wire systems are forecast to replace many of the traditional hydraulic and mechanical systems found in vehicles today. Thereby the automotive industry is following the path of the aerospace industry, which successfully has been using the fly-by-wire concept for many years in both military and commercial applications [1]. The class of applications where physical connections are replaced by electronic signals is in this thesis referred to as *control-by-wire*.

A drive-by-wire system is composed of several subsystems, each electronically controlled and responsible for a specific function. Examples are steer-, throttle-, brake- and shift-by-wire. They all have in common that a traditional, physical link is replaced by sensors, wires, electrical motors and Electronic Control Units (*ECUs*). In the case of steer-by-wire, the mechanical transmission of steering wheel movements through the steering column and steering rack are replaced by electronic signals sent from a position sensor by the steering wheel (or joystick). The signals are received by the ECU, which controls an electrical motor that steers the front wheels to the correct position. The same principle can be applied to the throttle, brake and gearbox systems [2].

There are many benefits of using drive-by-wire instead of traditional systems. A mechanical or hydraulic system takes up a lot of space and weighs significantly more. Less weight means less fuel consumption and lower emissions. Heavy and bulky components, like a steering column, can also mean worse consequences in case of an accident. With smaller, lighter equipment safety can be improved. Also, by-wire systems enable extended functionality and the important cost factor is considered to eventually advocate the exchange of technologies, as the systems becomes mass produced and thereby cheaper. Costs will also be affected positively by increased adaptability – for instance, with steer-by-wire only the steering wheel (or joystick) has to be moved to turn a left-steered car into a right-steered one.

Although being advantageous in many ways, the introduction of drive-by-wire systems in cars still faces some obstacles. A difficulty is the absence of mechanical feedback from the wheels to the driver, a phenomenon that has to be emulated to create the correct driver experience. The crucial safety issue of what happens in case of system failure has to be resolved – a drive-by-wire system must, through redundancy or other measures, guarantee stable operation for billions of hours. Also, true steer-by-wire systems are in most countries prohibited by law – there still has to be a solid connection between elements of the steering system [3].

### 2.2. Communication in distributed systems

Control-by-wire systems typically involve a number of ECUs, which are physically separated. Such *distributed systems* make some form of communication system necessary. The ECUs are therefore commonly linked together as *nodes* in a network, with some form of communication medium connecting them. This medium, most often consisting of some copper wiring, is called a *bus*. To avoid collisions when different nodes want access to the bus, the network traffic

has to be regulated. This can be done in different ways, and there are two main communication concepts known as the *event-triggered (ET)* and *time-triggered (TT)* architectures [4].

### **2.2.1. Event-triggered architectures**

In an event-triggered system, nodes wanting to send information on the bus may do so at any time, as long as the bus is not already used by another node. This approach may lead to collisions when two or more nodes simultaneously try to send messages on the bus [4]. One method to resolve this is called *CSMA/CD* (Carrier Sense Multiple Access with Collision Detection), in which a node that is trying to send a message also listens on the bus to detect collisions. If a collision is detected, the node waits for a predetermined amount of time before trying again to send its message. Should there be multiple nodes that want to use the bus, waiting times may be extensive and much bus capacity will be wasted on colliding messages. The method *CSMA/CA* (CSMA with Collision Avoidance) addresses these collision consequences. *CSMA/CA* involves either a certain bus access ordering or a distinct prioritisation of the nodes. All nodes listen to the bus and the node that is next in line or has the highest priority is allowed to transmit its message. By using this principle, collisions are avoided. Priority-based systems require some kind of procedure to determine which of the nodes that are attempting to transmit that has the highest priority. This procedure is called the *arbitration* [4,5].

Event-triggered architectures are relatively simple to implement and expand. When new nodes are connected to the bus, the existing nodes do not need to be reconfigured. The major drawback of event-triggered architectures is that they are non-deterministic. That is, there is no way to guarantee when a message will be successfully transmitted [4,6]. An event-triggered protocol commonly used in vehicle systems today is *CAN* (Controller Area Network) [6,7].

#### **2.2.1.1. The CAN protocol**

*CAN* is an event-triggered, serial communication, priority-based *CSMA/CA* protocol widely used in automotive applications. The priority arbitration in *CAN* is based on the concept of *dominant* and *recessive bits*. A node that sends a dominant bit on the bus causes the whole bus to adapt to this level. This is not the case when a recessive bit is sent. Each message is given a unique *identifier*, which consists of a number of dominant and recessive bits. Nodes that want to send a message start sending the identifier in question, while at the same time listening on the bus. A node that sends a recessive bit but reads a dominant bit on the bus has lost the arbitration and stops sending its message. The message with the largest number of consecutive dominant bits, starting with the most significant bit, will then win the arbitration and will be granted exclusive access to the bus [4]. A *CAN* message consists of an identifier of either 11 or 29 bits, a 16-bit checksum, 19 bits of other *overhead* and it can carry up to 8 bytes of payload data. The *CAN* protocol supports bitrates of up to 1 *Mbps* [8].

### **2.2.2. Time-triggered architectures**

The non-deterministic behaviour of event-triggered systems is unacceptable in safety-critical real-time applications, whose correct operation depends on guaranteed *response times*. These requirements can be met by a time-triggered system. There, the nodes are assigned *exclusive time windows* where they are allowed to send their messages. Then, at any given time, only one node is authorised to send its data and the risk of collisions on the bus is eliminated. In this way, the time at which a message is transferred can be guaranteed and the predictable behaviour required by, for instance, control-by-wire applications can be achieved. Also, it becomes possible to implement the strictly periodical behaviour needed by some control algo-

rithms. This time-triggered approach, which requires all nodes in the network to have a common view of the system time, is called *TDMA* (Time Division Multiple Access) [4].

One disadvantage of time-triggered systems is that if a node does not need to send anything in its time window, the window will be left unused. Time-triggered systems also need to be synchronised and are complicated to expand; all future nodes need to be included in the time schedule from the start, or extensive reconfigurations of the network may be required [6]. Two recently developed protocols for time-triggered communication are *TTCAN* and *FlexRay*. There is also an older time-triggered protocol called *TTP/C* [9].

#### 2.2.2.1. The *TTCAN* protocol

*TTCAN* (Time-Triggered CAN) is a higher-layer protocol placed on top of the unchanged CAN protocol [9]. It synchronises the communication, which then can be organised into exclusive time windows and *arbitrating time windows* where ordinary, arbitrating, CAN communication can take place. *TTCAN* can thus be used to combine time-triggered and event-triggered communication. Time is in a *TTCAN* network counted in Network Time Units (*NTUs*), which have the same duration at all nodes, regardless of each node's hardware configuration. *TTCAN* communication is organised into cycles, where the *matrix cycle* consists of one or more *basic cycles* in which the time windows are placed [10]. A *TTCAN* message is identical to a CAN message and includes the identifier even when sent in an exclusive time window, where no arbitration is necessary. *TTCAN*, like CAN, supports bitrates of up to 1 Mbps and an advantage of it is that it with only minor modifications can be used on already existing CAN systems. However, it does not provide certain important dependability services and it is considered that vehicle manufacturers might only use *TTCAN* during a transition period until the *FlexRay* technology is fully mature [6]. Still, *TTCAN* is a standard specified by ISO, which *FlexRay* is not.

#### 2.2.2.2. The *FlexRay* protocol

*FlexRay* is a protocol recently developed by a consortium consisting of BMW, Bosch, DaimlerChrysler, Freescale, General Motors, Philips and Volkswagen. Like *TTCAN*, it provides both time-triggered and event-triggered messaging. The communication cycle is divided into one static and one dynamic segment, which, in turn, are divided into frames where messages are sent. The static segment holds frames with guaranteed transmission times and the dynamic segment can be used for event-triggered messages. The dynamic segment is divided into a number of *mini-slots*, which are assigned to nodes in priority order. If the highest prioritised node chooses to use its mini-slot for a message transmission, the other nodes have to wait to transmit their messages. However, when a mini-slot is left unused the next node can start sending its message much sooner. *FlexRay* messages can hold up to 246 bytes of payload data and the overhead is 8 bytes per message [11]. The protocol supports bitrates of up to 10 Mbps on two channels. These two channels can be used separately to achieve a combined bitrate of 20 Mbps, or as redundant channels to implement fault-tolerance [9].



#### 2.2.2.3. The *TTP/C* protocol

The Time-Triggered Protocol for SAE class C (*TTP/C*) provides time-triggered communication, but not event-triggered. The development of it started in 1979 at the Vienna University of Technology and it has evolved ever since. Beside guaranteed transmission times and node synchronisation, *TTP/C* also provides fault-tolerance services like *bus guardians* and a *membership* service [4,9]. *TTP/C* puts no limit on the bitrate. There are *TTP/C* controllers that

support maximum bitrates of between 2 and 25 Mbps, depending on the transmission medium. A TTP/C message consists of about 4 bytes of overhead and can hold 240 bytes of payload data [12].

### 2.3. The GAST project

*GAST*, short for General Application Development Boards for Safety Critical Time Triggered Systems, was a recently concluded non-commercial project funded by the Swedish Agency for Innovation Systems. The project's starting point was the challenges of finding suitable and cost-efficient hardware for development of embedded applications in the automotive industry [13]. One focus was on future communication protocols in the control-by-wire sector. The project's aim was to gather stakeholders to jointly document industry specific requirements for distributed control-by-wire applications. These requirements were used in the design of two general purpose development boards, which implement several different communication concepts. The boards can be used to simulate the real ECUs and networks of ECUs found in current and future vehicles. The development boards will facilitate joint, low-cost research and development activities in academia and industry.



The hardware developed by GAST more specifically consists of two processor boards and three real-time communication boards. The first processor board, the *G1*, is based on the Motorola HCS12 processor while the second, the *G2*, is a dual processor board with a Motorola PowerPC MPC565 main processor and a HCS12 monitor processor. The three communication boards can be used with either processor board and provide communication controllers for the three time triggered protocols TTCAN, FlexRay and TTP/C.

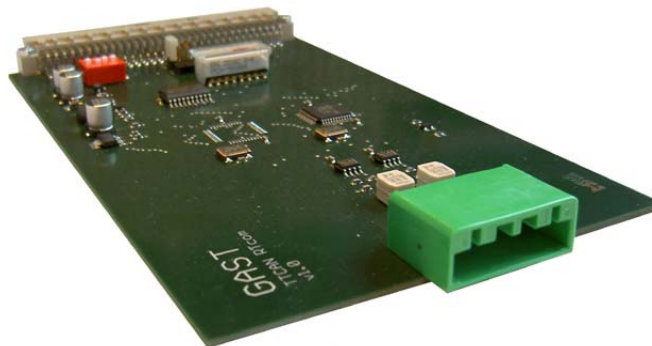


Figure 2.1. The GAST TTCAN communication board.

### 2.4. The CEDES project

*CEDES*, short for Cost Efficient Dependable Electronic Systems, is a research project on dependable electronic systems in road vehicles with industry and academia in cooperation. Volvo Car Corporation, Volvo AB, Autoliv Electronics AB, Chalmers University of Technology and SP Swedish Testing and Research Institute are partners in the project, which started in 2004 and will be concluded at the end of 2008. CEDES is financed by IVSS (Intelligent Vehicle Safety Systems), a Swedish research program with industry and government in collaboration [14].



The main goal of the project is to develop techniques and methods to design and build cost efficient dependable automotive electronics. Future electronic automotive systems must meet very high demands on dependability and fault tolerance, as well as on low costs. As reproduction costs for software are negligible compared to costs of physical devices, the main track of CEDES is the study of software-based mechanisms that require a minimum of hardware redundancy to achieve a desired level of system dependability.

The CEDES test bed is an experimental system, based on the electronic components for distributed vehicle systems developed by the GAST project. By using real control system applications and running those in the test bed against realistic environment simulators, the developed techniques and methods can be demonstrated under realistic circumstances.

### 3. THE TTCAN DRIVERS

In this chapter, the requirements, design concepts and limitations of the developed TTCAN driver package are described. Figure 3.1 shows the hardware used, with a G1 processor board on top of a TTCAN communication board. The two boards are connected by a backplane bus. The G1 is also connected to a power source and the PC, while the TTCAN board is connected to the bus.

#### 3.1. Requirements

The following requirements were identified for the TTCAN driver package:

- The drivers should comply with the CAN specification [8] and follow the guidelines given in the TTCAN IP Module User's manual [10].
- The drivers should be as hardware independent as possible, and should in particular be easy to port to the G2 processor board. Further, hardware aspects like register addressing should be transparent to the application programmer.
- The drivers should, with minimal alterations, be prepared to accept and incorporate automatically generated code.
- The driver code should be easy to read and understand.
- The compiled code should not occupy unnecessarily much memory space, as the G1 target system has limited read/write memory resources (12 KiB).

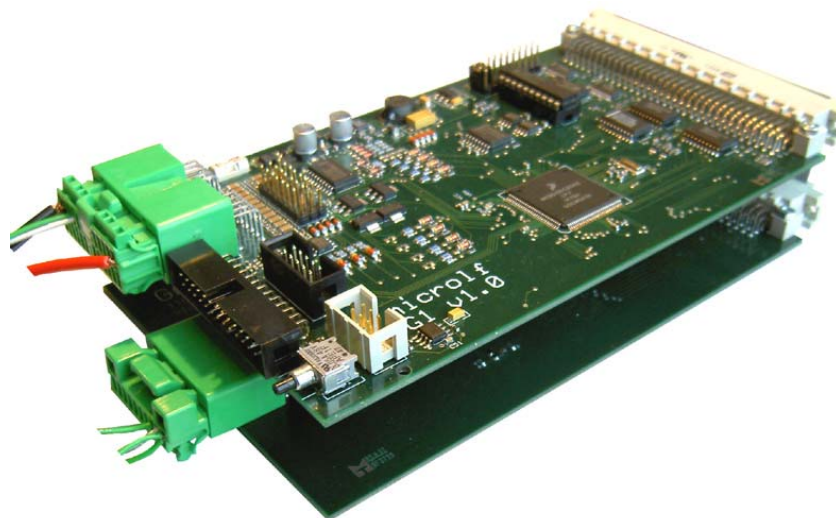


Figure 3.1. A node consisting of a G1 processor board and a TTCAN communication board.

#### 3.2. Design concepts

Some concepts were formed to guide the driver development and make the final product as compliant with the requirements as possible. These concepts are described below.

##### 3.2.1. Base address pointers

In the target system, the TTCAN module registers are accessed as external memory with 16-bit addressing. The address of each register is therefore heavily dependant on the properties of the system to which the TTCAN module is connected. Additionally, as the TTCAN board is (or rather, will be) equipped with two TTCAN controller chips, explicit addressing of

the registers would either require two separate, practically disjoint, driver packages or at least two separate address definition files. This issue was resolved by the specification of one comprehensive data type, covering the whole TTCAN register set. Then, throughout the driver package code, the TTCAN module registers can be accessed by the use of a pointer to the base address of the module being configured. All register addresses are derived from this base address and an offset, defined in the TTCAN IP Module User's manual [10]. The module base address is easily configured, and a second module only requires a second pointer. By using this concept, satisfying portability and hardware independency can be achieved.

### **3.2.2. Parameter and code component separation**

In order to facilitate integration with automatically generated code, it is advantageous to identify and separate all configurable parameters from fixed ones. The configurable properties can then be set by the code generator, while the fixed ones can be statically defined. Some sort of combination of the two sets into a complete setting for the TTCAN module must then be performed by the drivers. Some examples of configurable parameters are the bitrate, bit timing, time master priority, which interrupts to enable and the *application watchdog* limit. It is also vital that the configuration code that is to be automatically generated is completely separated from the driver and application code. In the developed driver package, all configurable parameters and register settings for a node were placed in a separate header file, while all the node's *message object* properties and its send/receive schedule resides in a separated C source file.

### **3.2.3. Names and order**

To be in accordance with official documentation for the TTCAN IP Module, the driver package was developed as closely as possible to what is specified and recommended in the TTCAN IP Module User's manual [10]. This includes agreement in bit definition names, variable names, message object configuration and send/receive functionality. In fact, for maximum conformity, the order in which registers are set in the different functions is the same as in the Configuration example supplied by the User's manual.

### **3.2.4. Layered code approach**

The driver code was divided into the following theoretical layers:

- **Bit definitions**

Textual representation of bit positions and parameters used to set registers. These include individual bits as well as bit combinations. Hardware addresses were also replaced by text. The definitions are used by driver functions on higher levels to enhance readability and provide hardware independence. Examples:

```
#define BIT0          0x0001
#define TX_MERGED    0x6000
#define VALID        0x8000
```

- **Fundamental functions**

These functions provide access to, and manipulation of, bits and registers in the memory. The functions are used throughout the driver code to access the memory. They are called by configuration functions as well as higher level functions. They are also available to application programmers needing direct access to memory contents. Examples:

```
getRegister
setBit
IFIsBusy
```

- **Application functions**

These functions perform high-level tasks and are called by the application programmer. The functions typically consist of a sequence of calls to lower-level functions. Examples:

```
ini tTTCAN
writeEventDrivenMessage
readMessageData
serveApplicationWatchdog
```

### 3.2.5. Non-blocking message functions

The read and write message functions in the driver package is non-blocking. That is, they do not wait for a successful transmission or the reception of a message before returning. This enables the *CPU* to handle a message and then continue working, without risking to be locked in a waiting stage. Yet, there is some busy-wait behaviour implemented. This since it takes a short period of time to transfer data between the read/write memory (*RWM*) of the G1 and the *Message RAM* of the TTCAN module. When reading, the drivers have to wait for the data to arrive from the Message RAM. When writing, the drivers wait for a successful transmission to the Message RAM, which frees the interface registers for further usage. Without this check, separate instruction sequences using the same registers could interfere with each other.

## 3.3. Limiting decisions

Some decisions regarding limitations in the driver functionality had to be made. Prior to each decision, an assessment of the usefulness, added complexity and estimated work to implement the functionality was made. As the main purpose of the driver package was to form a base for development of the reference application, a central question was if the functionality was necessary for this application. In the following cases, it was decided that so was not the case and/or the required amount of work was expected to be too extensive.

### 3.3.1. Fixed parameters

To simplify the development process, the following parameters of the TTCAN configuration were selected as fixed:

- **Normal operation**

Test mode is disabled, which makes the loopback and silent modes unattainable.

- **Extended identifiers**

Only extended, 29-bit, identifiers are supported. Extended identifiers are widely used by the automotive industry, especially in heavy trucks [7]. Therefore, they were selected for this project as well. The use of Standard, 11-bit, identifiers is disabled.

- **TTCAN level 2**

Global time with clock drift compensation enabled on all nodes. TTCAN level 1, lacking these features, is disabled.

### 3.3.2. No register content verification

As there were considerable problems with *bit flips* on the backplane bus (see section 7.2), some sort of verification of values written to registers should be in order. However, a decision was made that to perform this check every time a register value was edited would be too time-consuming. Instead, there is functionality supplied to catch seriously erroneous behaviour and restart node configuration and message handling. As this approach does not detect the fault when it occurs, only the manifestation of it, the fault causing the problem will not be identi-

fied. However, since the faults are transient, complete information about one fault still does not help in predicting the next occurring fault.

### **3.3.3. Forced message writes**

New messages and new message data are always written to the given message object, regardless of whether the object already contains data waiting to be sent. With this approach, there is a risk of lost data. An alternative that would allow both forced and non-forced writes is to introduce an extra parameter, supplied by the application programmer when calling the write function, giving the desired behaviour. For event-triggered messages this would be quite feasible, but for time-triggered ones a solution demands some tracking of prior values and a rather complex analysis. The decision to only support forced writes was made based on prioritisation of simplicity and space economy.

### **3.3.4. Only IF1 usage**

There are two sets of registers controlling access to the Message RAM: the IF1 and IF2 register sets. The two sets' functionalities are identical and the duplicity can for instance be used to assign different tasks to different interface register sets. Then tasks may interrupt tasks assigned to the other register set, but not tasks in its own group. A simple example of this is the wide-known producer/consumer case. Throughout the driver package, the only interface register set used is the IF1.

### **3.3.5. No parallel message reception checks**

The TTCAN IP module allows checking for newly arrived messages in parallel, which means that all message objects can be checked in the same time that it takes to check one single object. For this to work smoothly, transmit and receive objects should be grouped together. Since there were some trouble with certain message objects (see section 7.3), which led to their exclusion, the number of available message objects significantly dropped. Therefore, it was considered to be more appropriate with a dynamic configuration of the remaining objects, allowing arbitrary assignment and usage. The consequence of this is that message reception checks have to be carried out in a per-object manner.

## **3.4. Central functions**

The resulting driver package contains 32 functions and occupies about 4.86 KiB of the RWM when loaded to the G1. Many of the functions implement basic functionality used by higher-level functions, while others provide functionality not so frequently used. Below is a list of the most important functions, accompanied by short descriptions of what they do. For documentation on the rest of the driver functions, see the software package.

- **i ni tTTCAN**  
Configures the controller, initiates all message objects, sets all triggers and starts TTCAN communication.
- **i nterruptHandl er**  
Takes appropriate actions when an interrupt has occurred. Includes application-specific functionality, as well as error handling mechanisms.
- **wri tePeri odi cMessage**  
Stores data in a message object configured for sending in exclusive time windows, and starts the periodic transmission of it.

- **wri tePeri odi cMessageData**  
Stores data in a message object configured for sending in exclusive time windows.
- **wri teEventDri venMessage**  
Stores data in a message object configured for sending in arbitrating time windows, and marks it for transmission.
- **readMessageData**  
Fetches the data portion from a received message object.

## 4. THE TTCAN CONFIGURATION TOOL

Configuring a node for operation in a TTCAN network involves setting a large number of parameters: the NTU, the bitrate, whether the node shall be a time master or a slave, its master priority, operation mode and so on. Additionally, the node has a number of message objects that need to be properly configured and of top of that the properties of all the *triggers*, defining the node's time-triggered behaviour. In C code, important parameters can be hard to distinguish from irrelevant code details and covering different elements of a configuration might require opening several files and quite some scrolling. Based on this, it was decided that a graphical configuration tool, collecting all the relevant parameters for a node, was desirable. The tool was named *Comtest*, short for Compact TTCAN Environment Setup Tool. An illustration of Comtest usage is shown in figure 4.1.

### 4.1. Requirements

Some main outlines for Comtest were formed:

- It should provide a graphical interface for editing all relevant parameters of a TTCAN node configuration, including message objects and triggers.
- It should primarily be an aid in testing and debugging code used in this specific project. Little time would be spent on other considerations or adaptations for future usage.
- It should allow manual setting of most registers, but also provide calculation of register values from more high-level specifications of wanted node behaviour.
- It should as far as possible be hardware independent. More specifically, it should work with both the G1 and G2 processor boards.
- It should be easy to use. It should also generate code that is easy to read and understand. The generated code should contain detailed comments to facilitate debugging.
- It should be able to open and save node configurations, and allow multiple instances for parallel editing of configurations.

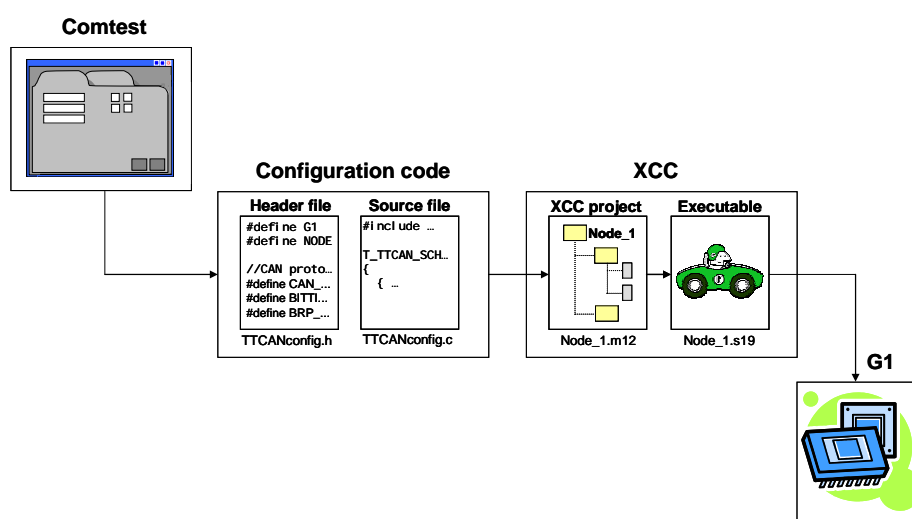


Figure 4.1. Schematic Comtest / XCC development process.

## 4.2. Limiting decisions

Guided by the outlines stated above, some design decisions were made. The result of these decisions was that the final version of Comtest comes with some limitations:

- Comtest is exclusively designed for the specific driver package also developed in this project. All limitations of the drivers also apply to Comtest. However, the generated code still may be included and used by any software and in any environment.
- The reference message contains exactly 4 bytes of data, which is the minimum for TTCAN level 2. The application is not allowed to put custom data in the reference message. This makes the message object configuration more straightforward.
- Only identifier bits 0-15 are used. A full, 29-bit, identifier with an, also 29-bit, mask for all 32 objects were hard to fit in the Comtest window in a good way. Instead, as 16-bit identifiers are more than enough for this project, only a part of the identifier field is used. This made room for the usage of masks, which was considered a more vital feature than full extended identifiers.

## 4.3. Design and features

The Comtest window was assigned a fixed size of 800 by 600 pixels, which should allow multiple instances on most screens and also be usable with lower resolutions. In order to fit all settings in the window and to make them easily overviewed, tabs were used. The three different types of settings were placed in tabs named General, Message Objects and Triggers. Images of the three different tabs are displayed by figure 4.2, 4.3 and 4.4 and their contents are described below.

### 4.3.1. The General tab

As displayed by figure 4.2, the General tab contains settings for the TTCAN configuration. The settings are described below and they control define statements in the generated header file `TTCANconfi.g.h`. Examples of these statements are given after the description of each setting.

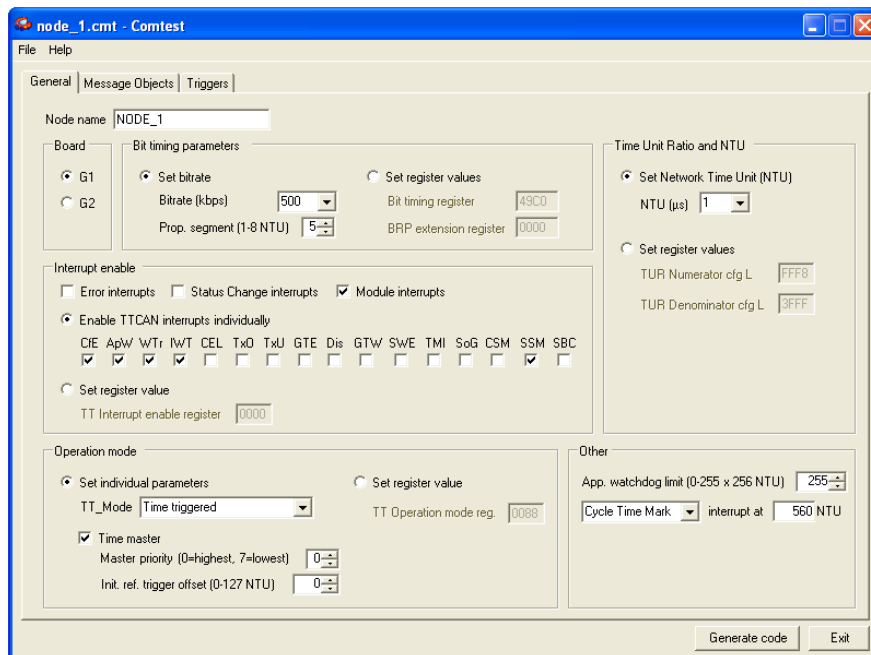


Figure 4.2. The Comtest General tab.

- **Node name**

A custom name can be specified. The name can then be used for implementation of node-specific behaviour in generic applications.

```
#define NODE_1
```

- **GAST processor board**

Code can be generated for the G1 or G2 processor board.

```
#define G1_BOARD
```

- **Bitrate, Propagation segment length**

For the bitrate, there are predefined values of 1, 10, 50, 125, 250, 500 and 1000 kbps. The propagation segment length may be set to values between 1 and 8 NTUs.

```
#define BITRATE 500
#define PROP_SEG 5
```

- **Bit timing and BRP extension registers**

Instead of setting the bitrate and the propagation segment length, the involved registers can be set manually.

```
#define BITTIMING 0x49C0
#define BRP_EXTENSION 0x0000
```

- **Interrupt enable**

The Error, Status change and Module interrupts can be individually enabled. Also, all TTCAN interrupts can be enabled individually or by setting the register value.

```
#define CAN_CONTROL_CFG 0x0002
#define TT_INTERRUPTENABLE 0xF002
```

- **Length of the NTU, TUR registers**

For the NTU length, there are predefined values of 1, 1.25, 1.5, 2, 3, 4 and 5 microseconds. The selected length is mapped to settings for the TUR registers, as will be described in section 4.4.1 and displayed in table 4.1. The TUR registers can also be set manually.

```
#define TUR_NUMERATORCONFIGURATIONLOW 0xFFF8
#define TUR_DENOMINATORCONFIGURATION 0x3FFF
```

- **TT mode, Time master, Time master priority, Initial reference trigger offset**

The TT mode can be set to Event driven, Time-triggered or Event synchronised time-triggered. For time masters, a priority between 0 and 7 can be set, as well as an initial reference trigger offset between 0 and 127 NTU. The configurable part of the TT operation mode register can also be set manually.

```
#define TT_OPERATIONMODE_CFG 0x0082
```

- **Application watchdog limit**

The application watchdog limit can be set to values between 0 and 255 times 256 NTUs, which gives an effective interval of 0 to 65280 NTUs.

```
#define TT_APPLICATIONWATCHDOGLIMIT 0x00FF
```

- **Time mark interrupt**

Three types of time mark interrupts are available: Cycle time, Local time and Global time interrupt. The Time mark can be set to values between 0 and 65535 NTUs.

```
#define TT_CLOCKCONTROL_CFG 0x0040
#define TT_TIMEMARK 0x0230
```

### 4.3.2. The Message Objects tab

The Message Objects tab is shown in figure 4.3. The settings in this tab control parts of the schedule contents in the generated file TTCANconfi.g.c. Message object 1 is dedicated to the reference message and is not open for configuration. The following settings are available for the remaining 31 message objects:

- **Enabled / Disabled**

Enabling an object will open the rest of the fields for configuration. It will also set the object as VALID in TTCANconfi.g.c.

- **Identifier**

The 16 least significant bits (ID 15-0) of the extended CAN identifier can be set.

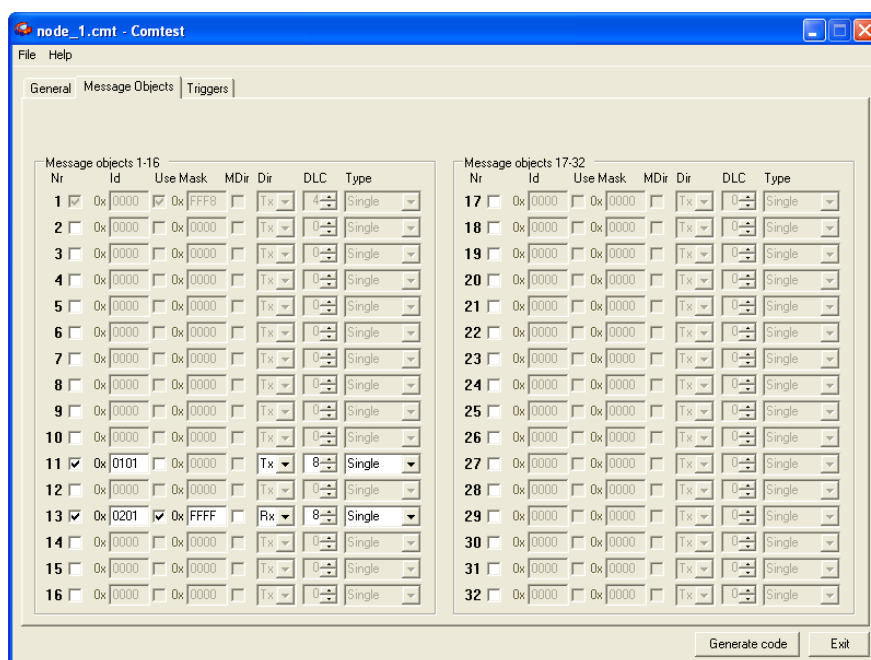


Figure 4.3. The Comtest Message Objects tab.

- **Use mask, Mask, MDir**

Enabling the mask fields will set the UMask bit for the message object. The 16 least significant bits (Msk 15-0) of the mask can be set. Checking the MDir box will set the corresponding bit for the message object.

- **Direction**

The message object can be configured as a Transmit (Tx) or Receive (Rx) object.

- **Data length code**

Specifies the number of data bytes that the message object shall contain. The CAN specification, as well as Comtest, allows between 0 and 8 data bytes.

- **Object type**

The message object can be configured as a single object or as part of a FIFO queue.

The settings in the Message Object tab will result in one configuration line for each object in TTCANconfi g. c. Example:

```
{ VALID, MESSAGE_13, 0x0201, USE_MASK, 0xFFFF, NOT_USED, RX_MESSAGE, 8, SINGLE }
```

### 4.3.3. The Triggers tab

The Triggers tab is displayed by figure 4.4. The tab controls both some define statements in TTCANconfi g. h and parts of the schedule contents in TTCANconfi g. c. When applicable, examples of the define statements in question are shown after the description of each setting.

- **Number of basic cycles in a matrix cycle, Tx enable window**

For the number of basic cycles in a matrix cycle, the possible choices are 1, 2, 4, 8, 16, 32 and 64. The Tx enable window can be set to values between 0 and 15 NTUs.

```
#define TT_MATRXLIMITS2_CFG 0x051F
```

- **Number of Tx triggers in a matrix cycle**

Specifies the total number of transmit triggers in a matrix cycle.

```
#define TT_MATRXLIMITS1 0x0010
```

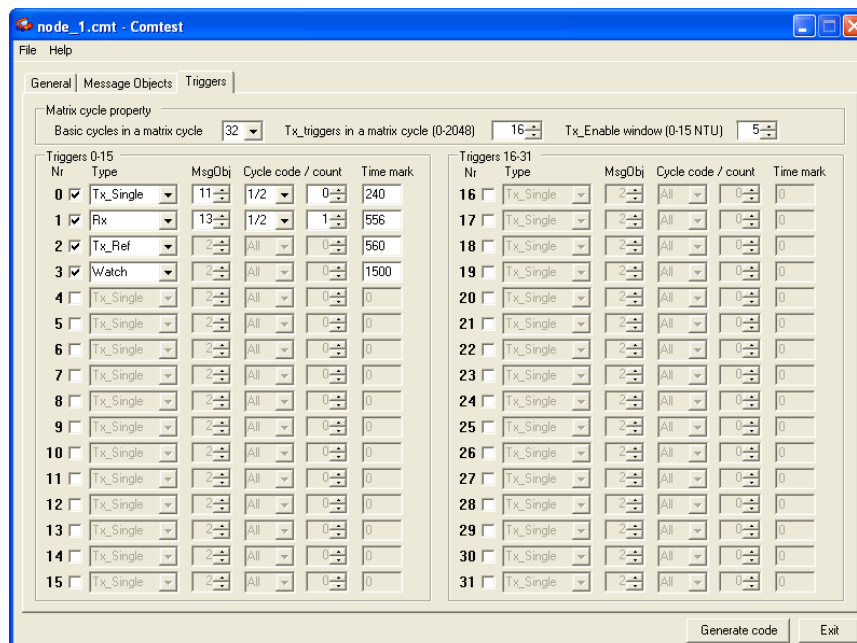


Figure 4.4. The Comtest Triggers tab.

The following settings are available for all 32 triggers:

- **Enabled / Disabled**

Enabling a trigger will open the rest of the fields for configuration. It will also enable Comtest to write the settings of the trigger to `TTCANconfig.c`. Note that all used triggers must be collected at subsequent numbers, starting with trigger 0. Any other configuration will result in an error condition when executed.

- **Trigger type**

There are seven trigger types: `Tx_Ref_Trigger`, `Tx_Ref_Trigger_Gap`, `Tx_Trigger_Single`, `Tx_Trigger_Merged`, `Watch_Trigger`, `Watch_Trigger_Gap` and `Rx_Trigger`.

- **Message object**

The number of the message object connected to the trigger.

- **Cycle code**

The cycle code specifies the periodicity of the trigger. A trigger can be activated at its time mark in all cycles or every 2nd, 4th, 8th, 16th, 32nd or 64th cycle.

- **Cycle count**

The cycle count specifies the cycle offset of the trigger activation. Possible choices depend on the choice of cycle code.

- **Time mark**

The time mark when the trigger shall be activated.

The trigger settings will result in one configuration line for each trigger in `TTCANconfig.c`. Example:

```
{ TRIGGER_0, TX_SINGLE, TRIGGERMESSAGE_11, EVERY_2ND_CYCLE, 0, 240 }
```

#### **4.3.4. Saving and loading configurations**

The configurations created in Comtest can be saved and loaded in a traditional manner. The configurations are saved as plain text files with the extension `.cmt` and are designed to be as readable as possible. There should be no trouble finding the desired information when opening a configuration file in an ordinary text editor.

## 4.4. Code generation

When the “Generate code” button in Comtest is pressed, the user is prompted to select the location for the generated code. Then, the required calculations and formatting of the settings are made and put into the two files `TTCANconfi g. h` and `TTCANconfi g. c`, which are created at the specified location. Note that any already existing files with the same names are overwritten without prompting. For details on the calculations of the register values, see below.

### 4.4.1. Register value calculations

The register values that is put in `TTCANconfi g. h` are in Comtest calculated in accordance with the TTCAN IP Module User’s manual [10]. It is often only a matter of shifting the specified value to a fixed position in the register of current interest. The only exceptions are the Time Unit Ratio (*TUR*) and bit timing registers. In the case of the *TUR* registers, the desired NTU duration time is mapped to specific values, displayed in table 4.1. These values have been calculated to give the desired *TUR* with highest possible `NumCfg`, which allows the highest possible accuracy in later calculations of `NumAct`. The bit timing parameters are not calculated by Comtest. Instead, this task is fulfilled at run-time by the driver package. The calculations are there performed in an iterative manner, arriving at appropriate prescaler values that, in turn, give parameter values within the boundaries stated in the TTCAN IP Module User’s manual. The code for this, in form of the function `getBi tTi mi ng`, is included in appendix G.

NTU	G1			G2		
	TUR	Numerator	Denominator	TUR	Numerator	Denominator
1	8	1FFF8	3FFF	40	1FFE0	0CCC
1.25	10	1FFFE	3333	50	1FFEA	0A3D
1.5	12	1FFF8	2AAA	60	1FFE0	0888
2	16	1FFF0	1FFF	80	1FFE0	0666
3	24	1FFF8	1555	120	1FFE0	0444
4	32	1FFE0	0FFF	160	1FFE0	0333
5	40	1FFE0	0CCC	200	1FFB8	028F

Table 4.1. Numerator and denominator values for achieving different NTU durations for G1 and G2.

## 5. THE REFERENCE APPLICATION

The reference application was developed to test network performance while running different schedules, with varying message lengths and contents. From the application output, factors such as *bus load* and *effective bandwidth* then could be calculated and analysed. This chapter describes the concept of the reference application and goes through the details of its design properties.

### 5.1. Main concept

The application was developed for a TTCAN network of three nodes, where two of the nodes act as senders and the third listens to the bus, counting the number of received messages from the sender nodes. After running a number of matrix cycles, the gathered data is output in form of frequencies of the (possibly different) message counts.

### 5.2. Message notation

There are two types of messages sent on the TTCAN bus, which are here denoted as *TT messages* and *ET messages*. TT messages are simply messages sent in exclusive time windows, while ET messages are sent in arbitrating time windows. There might be some confusion concerning this classification as ET messages are, in fact, also time-triggered. Nevertheless, they are as close one can get to true event-triggered behaviour in time-triggered surroundings.

### 5.3. Schedule properties

The schedules run by the reference application during the test sessions had some common properties, while other properties were varied in a number of ways. This section describes and motivates the different settings of these parameters.

#### 5.3.1. Time window size

A *CAN data frame* in extended format consists of between 0 and 8 data bytes plus 67 bits overhead, of which 54 bits are subject to *bit stuffing*. See table 5.1 and figure 5.1 on the following page for details. The method of bit stuffing is used to prevent nodes from going out of synchronisation, which might be the case if they for a long period do not get a flank on the bus to synchronise to. With bit stuffing, a sequence of more than five equal bits is broken by insertion of a *stuff bit* of opposite level. In addition to most of the overhead bits, the data bytes are also bit stuffed and if we assume a message with  $i$  data bytes, the maximum theoretical number of stuff bits is given by:

$$\left\lceil \frac{54 + 8i}{4} \right\rceil$$

Using this formula, an 8 data byte message is predicted to have at most 29 stuff bits [15]. However, in all CAN frames certain bits have fixed values. This fact actually lowers the upper limit, but is not taken into consideration in the calculation above. In an extended data frame, the bits SOF, RTR, r1 and r0 are always recessive while the SRR and IDE bits are dominant. In figure 5.2 on the next page these bits are shown as shaded, and stuff bits are indicated by arrows. By setting the remaining bits as shown in the figure, a practical stuff bit maximum is obtained. The maximum number of stuff bits in a frame with  $i$  data bytes is then given by:

$$12 + 8i/4 = 12 + 2i$$

A worst-case CAN frame with 8 data bytes then contains 28 stuff bits, which gives a total frame size of  $64 + 67 + 28 = 159$  bits. At the bitrate  $500 \text{ kbps}$ , such a frame takes 318 microseconds to transfer, which in the reference application is equal to 318 NTUs. This was rounded up to 320 NTUs and used as the *base size* for all time windows in the reference application schedules. When the application was run at other bitrates, the window size was adapted accordingly. For instance, at  $250 \text{ kbps}$  it was set to 640 NTUs while a bitrate of  $1 \text{ Mbps}$  led to a window size of 160 NTUs.

	Nr of bits	Start of frame	Arbitration field	Control field	Data field	CRC field	ACK field	End of Frame	Interframe Space
	1	32	6	0-64	16	2	7	3	
Subject to bit stuffing	X	X	X	X	X*				

\* The CRC delimiter bit is not subject to bit stuffing, as shown in figure 5.1.

Table 5.1. CAN frame fields that are subject to bit stuffing.

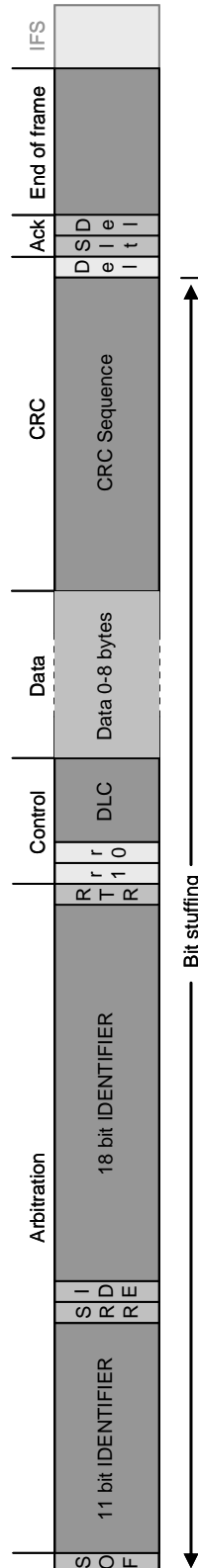


Figure 5.1. CAN data frame.

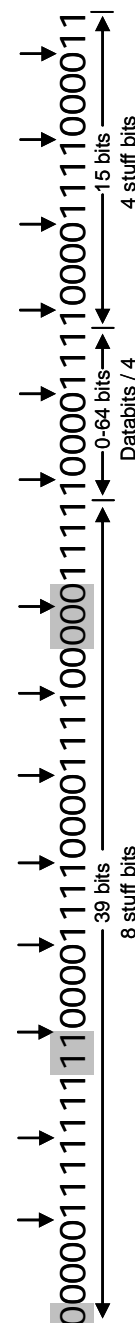


Figure 5.2. The specific CAN data frame that gives the maximum number of stuff bits (28).

### 5.3.2. Schedule classes

In the reference application, the nodes were configured with three different classes of schedules, each designed to occupy the bus with a certain percentage of exclusive time windows. The remaining time (if any) was allocated as arbitrating time windows, where both sender nodes may compete for bus access. In the three classes of schedules, the exclusive time windows occupied 100, 50 or 0 percent of the matrix cycle time and these three classes are here denoted as *TT*, *TT/ET* and *ET* schedules. The three classes are illustrated by example schedules in the figures 5.3, 5.4, 5.5 and 5.6 below. The grey (dark) portions represent exclusive time windows and the green (light) portions are arbitrating time windows.

### 5.3.3. Schedule variables

A number of variables that, in varying degree, were expected to affect the result of the tests were identified. These variables, together with the different schedule classes, were then used to comprise different settings for the evaluation of the target system. Explanations and motivations regarding the variables and their settings are found below and illustrations of the resulting schedules are shown in appendix D. The schedule variables were:

- Type of message distribution (only TT/ET schedules)
- Number of basic cycles in a matrix cycle
- Number of data bytes in an ET message (only TT/ET and ET schedules)
- Number of stuff bits in an ET message (only TT/ET and ET schedules)
- Bitrate

#### 5.3.3.1. Type of message distribution

While TT and ET schedules, as can be seen in figures 5.3 and 5.6, only have one possible way to distribute message windows over the matrix cycle, TT/ET schedules certainly offer more flexibility. Naturally, different applications may have different needs when it comes to this scheduling aspect. A control application, for instance, might periodically need to transfer a large chunk of information to other nodes that require the information for their calculations as soon as possible. Such an application would benefit from a schedule where the TT messages are gathered as early as possible in the cycle, as can be seen in figure 5.5. Certain real-time applications, on the other hand, might quickly need to send data that is to be treated and then transferred further, or send data about an unforeseen event. Then it would be unacceptable to be unable to access the bus for a long time and a schedule more like the one in figure 5.4 is

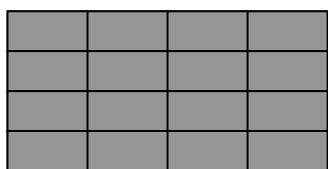


Figure 5.3. Example TT Schedule.

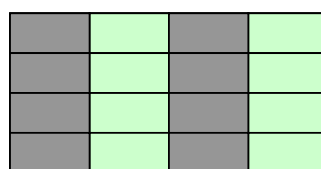


Figure 5.4. Example sparse TT/ET schedule.

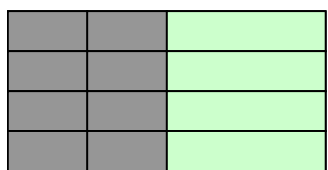


Figure 5.5. Example dense TT/ET schedule.

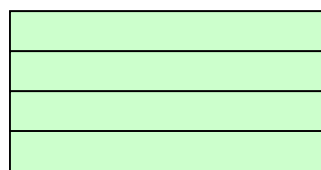


Figure 5.6. Example ET schedule.

preferable. These two classes of TT/ET schedules, here taken to their extremes, are called *dense* and *sparse* and are both included in the reference application tests.

### 5.3.3.2. Number of basic cycles in a matrix cycle

A matrix cycle of a certain length is divided into one or more basic cycles. Few (and therefore longer) basic cycles lead to less overhead in form of reference messages and enables larger coherent message windows. More, but shorter, basic cycles require less triggers, facilitate the usage of different periodicities and allows better synchronisation, as reference messages are sent more often [7]. The TTCAN protocol allows a matrix cycle to consist of 1, 2, 4, 8, 16, 32 or 64 basic cycles. However, two desired properties of the reference application were scalability and the ability to keep certain variables fixed while varying others. That is, a matrix cycle with twice as many basic cycles should still have the same duration and total number of message windows, and therefore only have half as many message windows per basic cycle. The identification of this essential property led to the conclusion that it was impossible to span the entire interval from 1 to 64 basic cycles. A complete coverage would either have demanded 96 triggers (there are 32 available) or a quite substantial compromise in the scalability of the basic cycles. It was decided that only testing schedules with 2, 4, 8, 16 and 32 basic cycles per matrix cycle would be an acceptable limitation. Even this would require some alterations in the sparse TT/ET / 2 basic cycles cases to manage scheduling it with the available number of triggers. This deviation is illustrated in figure 5.7, where the upper schedule shows the originally desired version and the lower one shows the compromise.

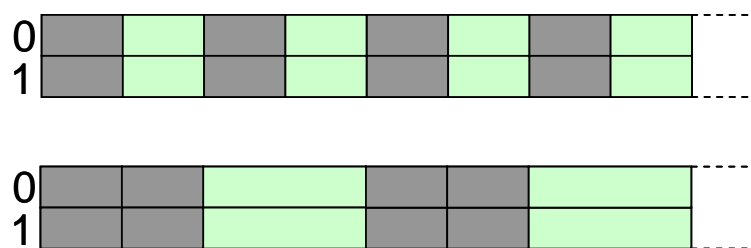


Figure 5.7. Two variants of a sparse TT/ET schedule.

### 5.3.3.3. Number of data bytes in an ET message

The time to transfer an ET message depends on the length of the CAN frame in question, which, in turn, depends on the number of data bytes in the message. Shorter messages should lead to a higher number of message transfers and less empty space at the end of each arbitrating time window. However, if the window is not large enough to fit these additional messages, bus usage and effective bandwidth should drop since less data is transferred. A CAN frame can hold from 0 to 8 data bytes. The reference application was decided to test sending ET messages with 0, 4 and 8 data bytes. An alteration of the TT message data amount was considered to be unnecessary as well as infeasible, as it would require adjusting the schedules accordingly to obtain correct results. Further, an alteration would also have complicated comparisons between schedules, since the window proportions would have been affected. TT messages were therefore decided to always contain 8 data bytes.

### 5.3.3.4. Number of stuff bits in an ET message

The length of a CAN frame is not only decided by the amount of data it holds, but also on the information actually being transmitted. As described in section 5.3.1, certain stuff bits are added to prevent nodes from going out of synchronisation. These bits are not considered for anything else than synchronisation by the receiving node(s), but still lengthens the CAN fra-

me and therefore its transfer time. In the reference application, the number of stuff bits is controlled by the message data. Bit stuffing also applies to other parts of the CAN frame, such as the identifier, but these parts have not been adjusted to produce more or fewer stuff bits. To produce the minimum number of stuff bits for a frame, the data was simply set to altering ‘1’ and ‘0’ bits – the (hexadecimal) data byte AA. To produce the maximum number of stuff bits, one has to consider the surrounding fields in the CAN frame. This leads to different data for the cases of 4 and 8 data bytes (the 0 data byte case is not applicable here). For 4 data bytes the data was set to 1E, and for 8 data bytes it was set to 3C. See figures 5.8 and 5.9 below for details. Disappointingly enough, this showed not to give the desired result and was concluded to be too error prone, as one flipped bit in the data would disturb the whole sequence of stuff bits. The data was therefore instead set to 00, reducing the resulting number of data stuff bits from 8 to 6 for the 4 byte case and from 16 to 13 for the 8 byte case.

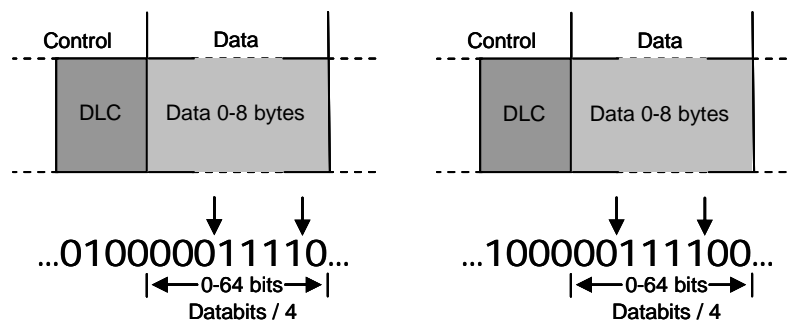


Figure 5.8. Data causing maximum number of stuff bits for 4 data bytes.

Figure 5.9. Data causing maximum number of stuff bits for 8 data bytes.

### 5.3.3.5. Bitrate

An interesting property of any computer network is the bitrate, since it so drastically affects the effective bandwidth. Both the CAN and the TTCAN protocol support bitrates of up to 1 Mbps. As a full set of test cases for several different bitrates were considered to be unreasonable to manage, a base bitrate of 500 kbps was chosen and only some isolated tests were carried out with the bitrates 250 kbps and 1 Mbps. The purpose of these additional tests was to confirm the theory that the results from one bitrate should also be valid and fully apply to another one. That is, a doubled bitrate should result in a doubled amount of transferred data. Further, a schedule with the same proportions, which means half the duration time for a doubled bitrate, should always result in the same amount of transferred data.

## 5.4. The nodes

The three nodes running in the application have, as mentioned, different tasks. Two of the nodes were designed to be senders, which repeatedly attempts to send messages on the bus. The third node’s task is to listen on the bus and count the number of messages received from each sender node. The nodes will from now on be denoted as *Sender 1*, *Sender 2* and *Receiver*.

### 5.4.1. The Sender nodes

The sender nodes send both TT and ET messages. The sending of TT messages (if any) that are to occupy the schedule are distributed evenly between the two nodes. Generally, this is implemented by providing them with the same trigger setup, only differing in the cycle count of the TT triggers. This makes them take turns in sending TT messages, so that no node at any time has to send two TT messages in a row.

When it comes to the sending of ET messages, it is important that the application puts as heavy load as possible on the TTCAN bus. This is necessary to obtain useful results concerning maximum performance. A simple approach to achieve maximum bus load would be to have one sender node, constantly sending messages to a receiver node. As soon as the sender node has successfully transferred a message, it simply initiates the sending of a new one. The re-initiation can, however, not be done instantly and therefore the bus will be idle for a short period of time before the new message is ready to be transferred. This imperfection can be corrected by using two sender nodes, each constantly trying to send messages. Then one node will always be ready to transfer its message as soon as the bus becomes idle. This assumption will hold as long as a re-initiation of a message transfer does not take longer than the shortest time to transfer a message on the bus. A setup with two sender nodes that compete for bus access can also provide additional information, addressing questions about node prioritisation and possible *starvation*.

To determine which node that is allowed to send its message in case of a potential collision, each message must have a unique priority. In this application, the messages from Sender 1 are prioritised over the messages from Sender 2. This is indicated by a lower identifier in the *arbitration field* of Sender 1's TTCAN message frames.

#### 5.4.2. The Receiver node

While running the application, the central task for Receiver is to record all incoming messages and the node of their origin. Messages sent in exclusive time windows are, by definition, always guaranteed access to the bus and are therefore always transferred at the expected time. This fact was also confirmed by several tests. Hence, Receiver does only need to keep track of messages sent in arbitrating time windows. To do this, the node keeps message counters which is cleared at the start of every matrix cycle and then incremented as messages arrive. At the end of a matrix cycle, the counters' values are used to update frequency tables. After a certain number of cycles (during the test sessions the number was 100), Receiver loops through its tables and outputs the results. Both all occurring frequencies in the tables and the majority value of these are displayed. The example text below shows the output results when running a dense TT/ET schedule with 4 basic cycles, 8 data bytes per ET message and minimal bit stuffing in 500 kbps.

```
G1: go
Synchronised to schedule
Press any key to start test...
Test started.

Statistics (100 matrix cycles, 04 basic cycles, DLC 08)
24 64
Messages received: 24
14 64
Sender 1: 14
10 64
Sender 2: 10
```

The unanimous output says that during all 100 matrix cycles (64 hexadecimal), 36 ET messages were received (24 hexadecimal). Of these, 20 came from Sender 1 and 16 from Sender 2. Had there been variations between matrix cycles, additional rows would have appeared in the application output. An example of this is shown on the next page, where a sparse TT/ET schedule with 16 basic cycles, 8 data bytes per ET message and minimal bit stuffing has been executed in 1 Mbps. Receiver records 32 ET messages in all cycles, but in one case Sender 1 only

manages to send 31 of these, and in two cases it only sends 30 messages. The remaining messages, as can be seen, are supplied by Sender 2. Though, in the vast majority of matrix cycles, Sender 1 is responsible for all the 32 ET messages transferred on the bus.

Statistics (100 matrix cycles, 10 basic cycles, DLC 08)

20 64

Messages received: 20

1E 02

1F 01

20 61

Sender 1: 20

00 61

01 01

02 02

Sender 2: 00

## **6. RESULTS AND ANALYSIS**

The results collected from the reference application test sessions are here accounted for and analysed. First, the exclusion of some test cases is reported and motivated. After this the results and their accordance with the expected outcome are described, followed by a report from the tests with different bitrates. Then the result analysis starts with the number of message transfers per matrix cycle, continues with the effective bandwidth and ends with the calculated minimal bus load. The chapter is concluded by some observations regarding the number of messages each sender node transmitted in the different schedules.

### **6.1. Excluded test cases**

During the test sessions, there were extensive problems in getting the schedules with 2 basic cycles per matrix cycle to deliver reliable results. No thorough investigation was conducted, but the problems seemed to be related to the hardware and the synchronisation between nodes. The results from the tests showed that a constant number of messages were transferred every matrix cycle. This number was however only between 50 and 75 percent of the expected amount and it also varied when different physical units were set to execute different parts of the application. This fact, combined with the knowledge that another thesis group [16] had successfully configured a TTCAN network with basic cycles of 400 milliseconds, led to the conclusion that the problem must be hardware-related. A qualified guess was that some, or all, of the nodes fell out of synchronisation after about 50 and 75 percent of the basic cycle and re-synchronised at every reference message. The length of the basic cycle in these test cases was approximately 16 milliseconds, which means that the troubles seemed to arise after between 8 and 12 milliseconds. In lack of reliable results, and with no time to investigate the matter further, the 2 basic cycles test cases were excluded from the investigation.

### **6.2. The pre-test analysis**

Prior to the test sessions, an analysis of all the schedules that were to be run by the application had been conducted. It consisted of considerations of window sizes and message lengths, where possible variations in the number of stuff bits were included in the calculations. For each schedule, it had resulted in an interval where the number of transferred messages should be found. In many cases, the interval was as narrow as just one number. The intervals and the actual outcome from the tests are displayed in the tables A.1, A.2, A.3 and A.4 in appendix A. All obtained results were inside the specified intervals. The results are also shown in a more concise way in the tables A.5, A.6, A.7 and A.8. The figures found in appendix A will be further visualised and discussed below.

### **6.3. Tests with different bitrates**

All results presented and analysed in this chapter were collected from test runs with a bitrate of 500 kbps. Some tests with the bitrates 250 kbps and 1 Mbps were conducted to investigate if results obtained with one bitrate were directly transferable to another. It was found that so indeed was the case. When a tested schedule's time windows were doubled in length and run at half the bitrate, it gave identical results. Also, when the windows of the original schedule were halved in length and run at the double bitrate the same results were obtained. That is, a doubled bitrate actually resulted in a doubled amount of transferred messages, since two matrix cycles then run in the time of one at the lower rate. Likewise, a halved bitrate resulted in half the amount of transferred messages in any given time period. As a summary, the behaviour at different bitrates showed to be exactly as was predicted. Therefore, the decision to run the bulk of the tests at 500 kbps did not have to be altered.

## 6.4. The number of transferred messages

On the next page, there are four bar charts showing total amounts of transferred messages. There is one chart for each schedule class (TT, sparse TT/ET, dense TT/ET and ET). The charts are denoted as figure 6.1, 6.2, 6.3 and 6.4 and all show the number of transferred messages for 32, 16, 8 and 4 basic cycles per matrix cycle and for differing numbers of ET message data bytes. The ET message data also differ in the concentration of stuff bits. The exact figures these diagrams were based on can, as was previously mentioned, be found in appendix A.

### 6.4.1. Comparison of the schedule classes

An expected result displayed by figure 6.1 is that the number of messages for the TT schedules is always the same, namely 64. This is exactly the number of base size message windows per matrix cycle in all schedules run by the application. This is also the lowest number of message transfers that was recorded for any schedule throughout the test sessions. Moving from the TT schedule diagram, through the sparse TT/ET, dense TT/ET and ET schedule diagrams, the number of transferred messages gradually increases for most categories. The largest differences between the schedules appear for messages with data amounts less than 8 bytes, where it becomes obvious that the large arbitrating time windows provide superior flexibility. This as the dense TT/ET and ET schedules allows all ET messages, regardless of their lengths, to be instantly followed by new ones as long as there are room left in the basic cycle. In general, most successful in getting many messages transferred show to be the ET schedules followed by the dense TT/ET, sparse TT/ET and TT schedules.

### 6.4.2. Comparison of the number of basic cycles

As was noted, the TT schedule shows no variations in the number of transferred messages, regardless of the number of basic cycles per matrix cycle. The sparse TT/ET schedules show the same behaviour for both the 8 and 4 data byte cases. Only when the messages are transferred without any data at all, two messages fit into the single-sized arbitrating time windows. Then, as expected, the number of ET messages doubles and the result is a total message count of 96. The same pattern appears for 32 and 16 basic cycles with the dense TT/ET schedules, but there is a difference emerging when the basic cycles get fewer and, as a consequence, the arbitrating time window at the end of each basic cycle gets longer. Then, more 4 byte than 8 byte ET messages fit into the window. The number of messages increases the most when there are few stuff bits and less when there are more stuff bits, as might have been expected. The ET schedules show a slight increase in the number of transferred messages as the number of basic cycles gets smaller. This is most likely caused by longer coherent time windows and less overhead in form of reference messages.

### 6.4.3. Comparison of the number of data bytes

The total number of transferred messages can also be studied with respect to the number of data bytes in an ET message. With this shifted focus, it becomes obvious for which data amount the schedules actually were developed. The sparse TT/ET schedules, with single-sized arbitrating time windows, allow the same number of messages regardless of if it is 8 or 4 byte messages being transferred. The 0 byte messages, however, fit two in the space of one 8 byte message and therefore twice as many of these are transferred, as was noted above. When looking at the dense TT/ET schedule results, a trend where less data bytes lead to more message transfers can be distinguished. It is perhaps not that apparent in the 4 data byte cases, but should be more noticeable when looking at the 0 data byte bars. The same is also true, in a more obvious way, for the ET schedules where shorter message frames directly lead to the ability to transfer more messages.

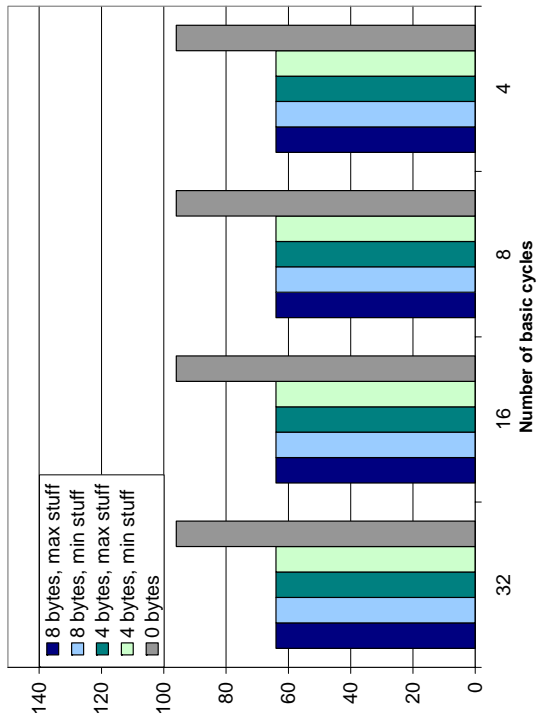


Figure 6.2. Number of transferred messages for the sparse TT/ET schedules.

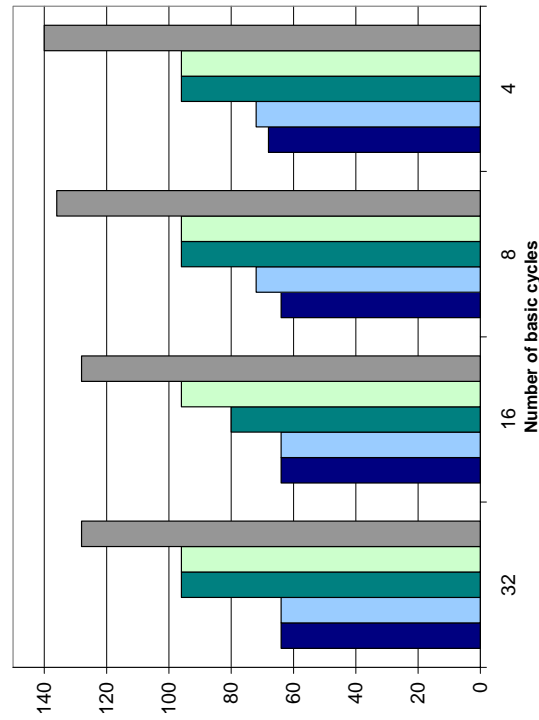


Figure 6.4. Number of transferred messages for the ET schedules.

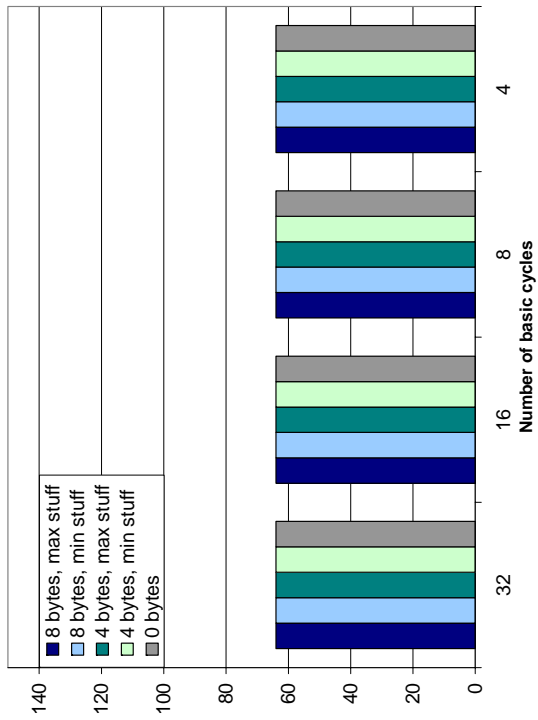


Figure 6.1. Number of transferred messages for the TT schedules.

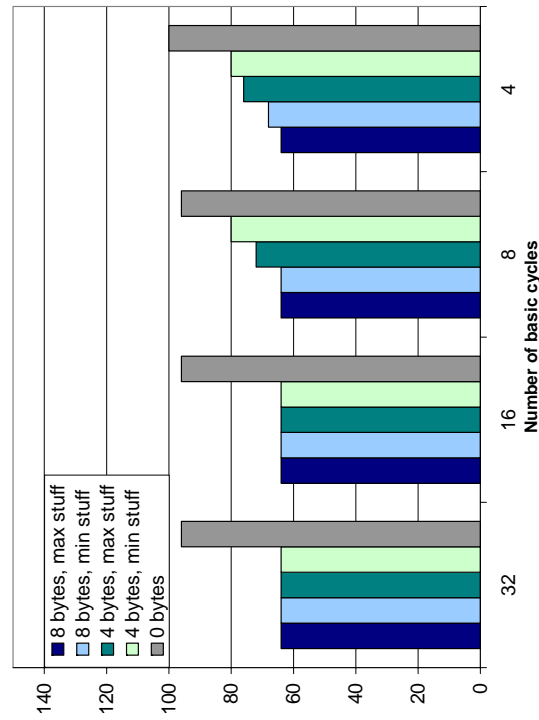


Figure 6.3. Number of transferred messages for the dense TT/ET schedules.

## 6.5. Effective bandwidth

Following this page, you will find four bar charts showing calculated effective bandwidth from the reference application test runs. There is one chart for each schedule class (TT, sparse TT/ET, dense TT/ET and ET). The charts are denoted figure 6.5, 6.6, 6.7 and 6.8 and all show the calculated effective bandwidth for 32, 16, 8 and 4 basic cycles per matrix cycle and for differing numbers of ET message data bytes. The ET message data also differ in the concentration of stuff bits. Naturally, the 0 data byte case has been excluded in these charts, as no data then are transferred. The exact figures the diagrams were based on can be found in appendix B.

### 6.5.1. Comparison of the schedule classes

When comparing the effective bandwidth for the 8 data byte cases, there are some differences between the schedule classes. These mainly occur in the lower numbers of basic cycles. The largest variation is the maximum marks for the TT and ET schedules in the 4 basic cycles case. They are 38.2 and 43.0 percent, respectively, which is a difference of about 13 percent. Note that 43 percent also is the maximum effective bandwidth reached during the tests, giving an *effective bitrate* of 215 kbps. This can be compared with pure event-triggered CAN communication, which has an upper limit on the effective bandwidth of 49.2 percent but typically only manages between 25 and 35 percent [12].

Comparison of the 4 data byte schedules shows that the TT schedules have the highest bandwidth, while the TT/ET and ET schedules have lower values. This, perhaps unexpected, result is caused by the fact that no 4 byte ET messages are actually sent in the TT schedules. In the TT/ET schedules the ET message share is 50 percent, while it is 100 percent in the ET schedules. Therefore, low data quantity has the largest impact on the ET effective bandwidth, where there are no 8 data byte TT messages included in the calculations. That is why the ET schedules show the most diverse results, with bandwidth values spanning from 21 up to 43 percent. Further noticeable about the 4 data byte schedules is the differences regarding the two TT/ET schedule classes, in the 8 and 4 basic cycles cases. The dense schedules there show a between 4 and 5 percentage points higher bandwidth than the sparse ones. This ought to be due to the increased number of messages transferred, as established in the previous section.

### 6.5.2. Comparison of the number of basic cycles

Looking at figure 6.5, the bandwidth increases as the number of basic cycles gets smaller. This is a direct result of the reduced overhead in form of reference messages. Remember, the same number of messages is always transferred in the TT schedules. In the case of the sparse TT/ET schedules, they are identical to the TT schedules in the 8 data byte cases while the 4 data byte cases show significantly lower bandwidth. Still, the pattern with higher bandwidth for smaller numbers of basic cycles remains. The trend in the dense TT/ET and ET schedule charts looks a lot like in the sparse one, except for some drops in bandwidth when more stuff bits are inserted. As before, this shows to be more significant as the arbitrating time windows get longer.

### 6.5.3. Comparison of the number of data bytes

The TT schedules all show the same bandwidth for every individual number of basic cycles, regardless of the ET message data quantity. This is natural as the amount of data in the TT messages is fixed at 8 bytes, and no ET messages are sent. Continuing on, the charts in figure 6.6, 6.7 and 6.8 show some striking similarities, where 8 data byte ET messages consistently

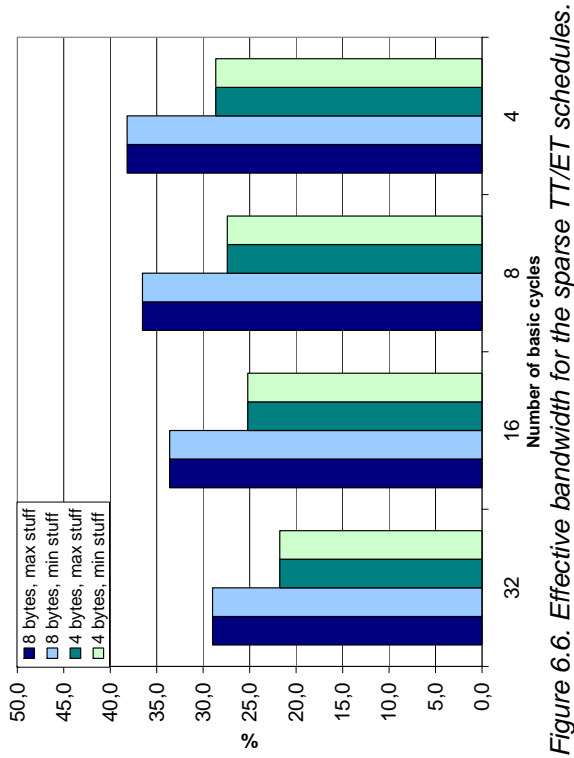


Figure 6.6. Effective bandwidth for the sparse TT/ET schedules.

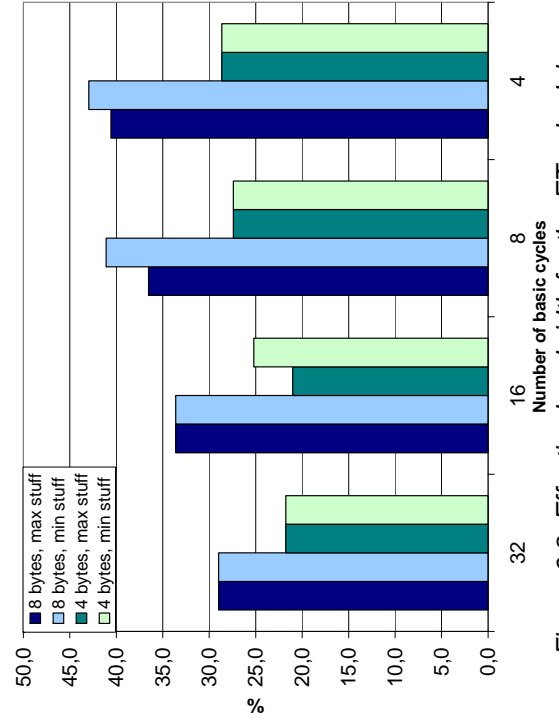


Figure 6.8. Effective bandwidth for the ET schedules.

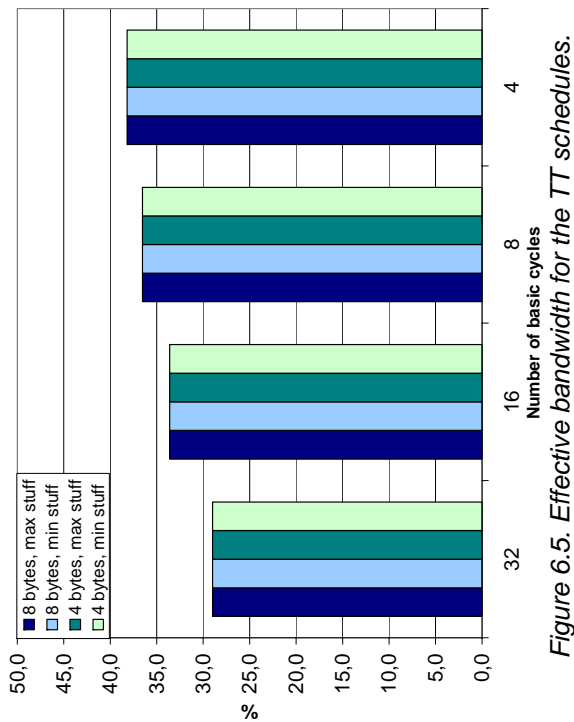


Figure 6.5. Effective bandwidth for the TT schedules.

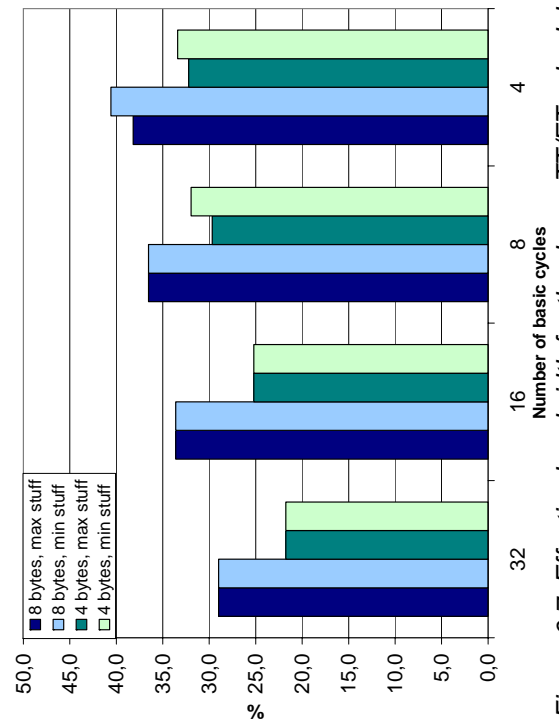


Figure 6.7. Effective bandwidth for the dense TT/ET schedules.

results in higher bandwidth than 4 byte messages. This is also quite natural: As in the case of number of message transfers, the effective bandwidth is very dependant on the data amount for which the schedule was designed. If the data quantity per message is halved while the number of messages stays the same, the effective bandwidth too is halved. Furthermore, when an ET messages only contains 4 data bytes, the overhead percentage of that message increases significantly. This is due to the fact that the overhead is the same for all CAN frames, regardless of the data quantity. As a consequence of these facts, a big drop in effective bandwidth should be expected when comparing schedules with 8 data byte ET messages with 4 data byte ones.

## 6.6. Calculated minimal bus load

This section includes an attempt to estimate and analyse the bus load for the different schedules. As it showed to be difficult to predict the exact number of stuff bits in a message, it is the *calculated minimal bus loads* that are displayed in figure 6.9, 6.10, 6.11 and 6.12 on the next page. The exact figures these charts were based on can be found in appendix C. Note that these are the absolute minimal bus loads; the actual bus load for a schedule might in some cases be significantly higher. This gap most certainly is quite large for test cases with maximum stuff bits. A bus load of 100 percent should however be unreachable, as there are sections of the matrix cycle that always are incapable of data transfer. Two examples are the small gap between the reference message and the rest of the schedule, and the 1 bit time plus 1 NTU gap that follow every basic cycle.

The ET schedules, which also brought the highest number of messages and effective bandwidth, show the highest bus load figures. While a high bus load is not a purpose of its own, in this case it shows how well the bus capacity is used. The maximum bus load reached is 92.4 percent for the ET schedule / 4 basic cycles / 4 data byte test cases. As mentioned above, the actual bus load probably is some additional percentage points higher than this for the maximum stuff bits case. Again, it is clear that a schedule with few basic cycles and long arbitrating time windows gives the best performance and makes best use of the bus capacity. The lowest bus load achieved during the test sessions was 72.4 percent for the sparse TT/ET schedule / 4 basic cycles / 4 data byte cases. This rather low figure stresses the importance of adjusting the schedule to the messages being sent. It is obvious that a poorly designed schedule leaves big gaps in the communication, which may result in quite low utilisation even when there are nodes constantly trying to access the bus.

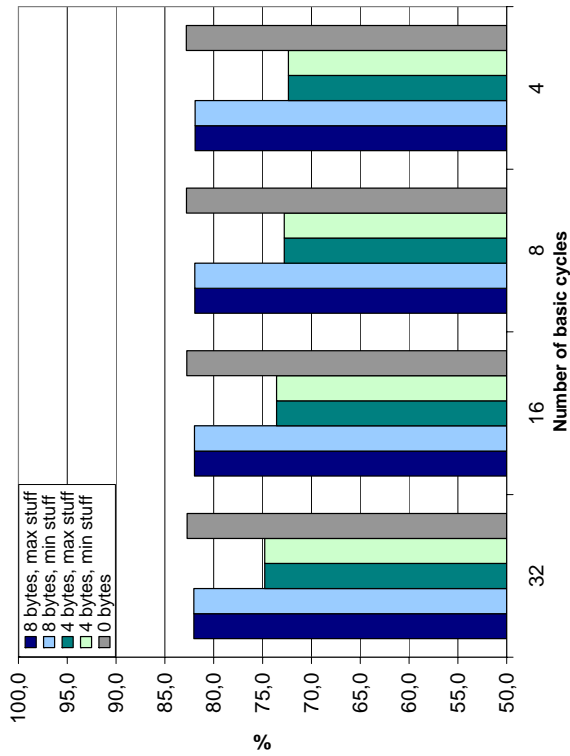


Figure 6.10. Minimal bus loads for the sparse TT/ET schedules.

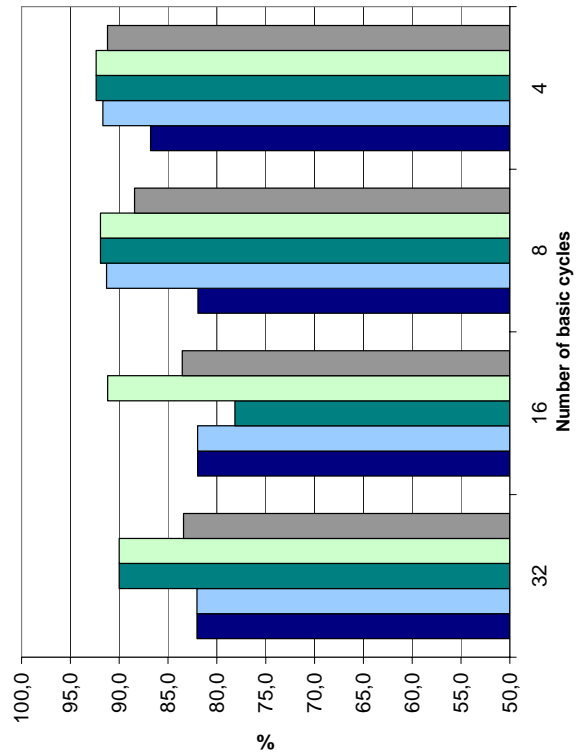


Figure 6.12. Minimal bus loads for the ET schedules.

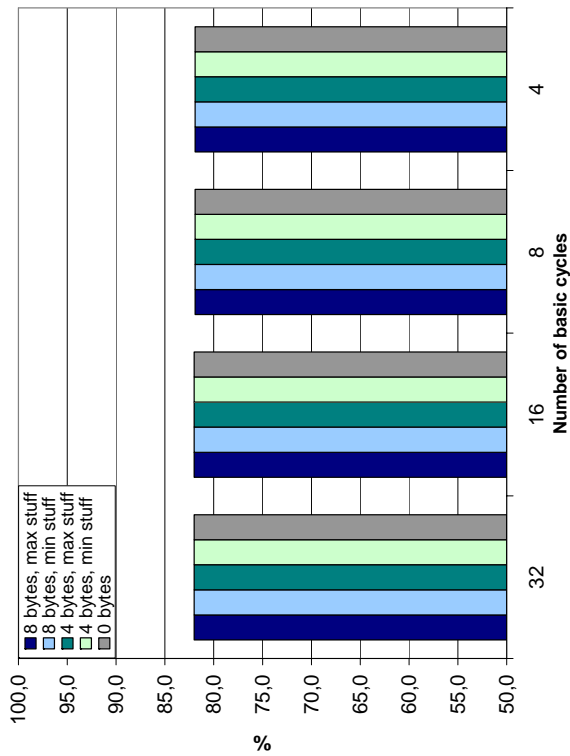


Figure 6.9. Minimal bus loads for the TT schedules.

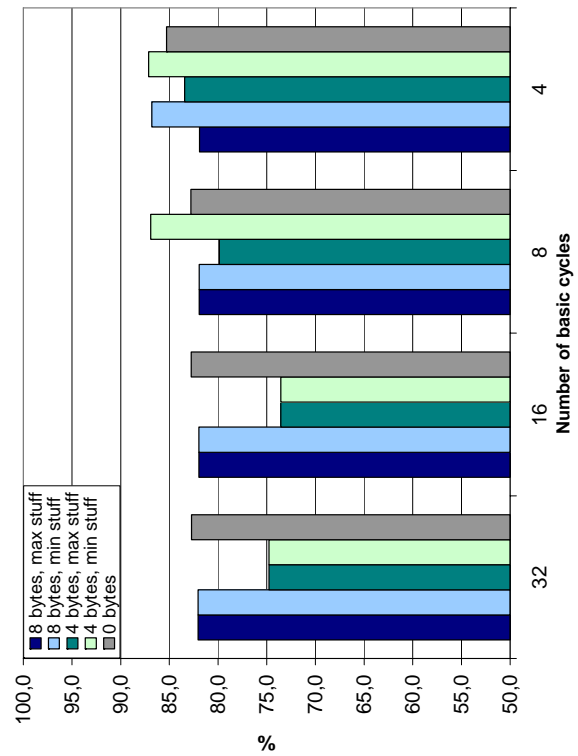


Figure 6.11. Minimal bus loads for the dense TT/ET schedules.

## 6.7. Results for each sender node

As was explained in section 5.4.1, Sender 1 and Sender 2 both constantly try to send ET messages on the bus. Since it takes a short period of time to re-initiate the sending of a message, the result should be that the nodes alternately get to transfer their messages. The higher prioritised Sender 1 should always be the first to send in an arbitrating time window, followed by Sender 2 if there is still time before the end of the window. Two conclusions can be drawn from this reasoning: First, Sender 2 should never be able to send more ET messages than Sender 1 in any schedule. Second, Sender 1 should at no time be able to transfer two more messages than Sender 2 in any arbitrating time window.

The tables 6.1, 6.2 and 6.3 on this and the following page show the total number of ET message transfers for all schedules, together with the number of messages each sender node transferred. Further, the right-most columns in the tables show the supposed number of messages each node transferred during one basic cycle. Since all basic cycles are identical, the values in these columns are easily calculated by dividing the number of messages with the number of basic cycles. Note that the results agree well with the above argument, and both the predictions regarding expected behaviour seem to hold. The only deviation is the results for the sparse TT/ET schedule / 4 basic cycles / 8 and 4 data byte cases. The overall results for these cases are the expected 32 ET messages, but a surprising fact is that Sender 2 somehow manages to transfer one of these messages. This should be impossible, as there are more than enough time for Sender 1 to re-initiate its sending between the arbitrating time windows in the sparse schedules. The result becomes even more puzzling when considering that the matrix cycle consists of 4 exactly identical basic cycles. Any erroneous configuration or other source of influence should affect all basic cycles in the same way, with a deviation that is a multiple of 4 as a result. The cause of this unexpected result remains unknown. Still, the total number of messages met the expectations and the result is included in the analyses.

Basic cycles	Data bytes	Stuff bits	Total	Sender 1 Total	Sender 2 Total	Sender 1 per BC	Sender 2 per BC
32	8	max	32	32	0	1	0
		min	32	32	0	1	0
	4	max	32	32	0	1	0
		min	32	32	0	1	0
	0		64	32	32	1	1
16	8	max	32	32	0	2	0
		min	32	32	0	2	0
	4	max	32	32	0	2	0
		min	32	32	0	2	0
	0		64	32	32	2	2
8	8	max	32	32	0	4	0
		min	32	32	0	4	0
	4	max	32	32	0	4	0
		min	32	32	0	4	0
	0		64	32	32	4	4
4	8	max	32	31	1	-	-
		min	32	31	1	-	-
	4	max	32	31	1	-	-
		min	32	31	1	-	-
	0		64	32	32	8	8

Table 6.1. Results for each sender node for the sparse TT/ET schedules.

Basic cycles	Data bytes	Stuff bits	Total	Sender 1 Total	Sender 2 Total	Sender 1 per BC	Sender 2 per BC
32	8	max	32	32	0	1	0
		min	32	32	0	1	0
	4	max	32	32	0	1	0
		min	32	32	0	1	0
	0		64	32	32	1	1
16	8	max	32	16	16	1	1
		min	32	16	16	1	1
	4	max	32	16	16	1	1
		min	32	16	16	1	1
	0		64	32	32	2	2
8	8	max	32	16	16	2	2
		min	32	16	16	2	2
	4	max	40	24	16	3	2
		min	48	24	24	3	3
	0		64	32	32	4	4
4	8	max	32	16	16	4	4
		min	36	20	16	5	4
	4	max	44	24	20	6	5
		min	48	24	24	6	6
	0		68	36	32	9	8

Table 6.2. Results for each sender node for the dense TT/ET schedules.

Basic cycles	Data bytes	Stuff bits	Total	Sender 1 Total	Sender 2 Total	Sender 1 per BC	Sender 2 per BC
32	8	max	64	32	32	1	1
		min	64	32	32	1	1
	4	max	96	64	32	2	1
		min	96	64	32	2	1
	0		128	64	64	2	2
16	8	max	64	32	32	2	2
		min	64	32	32	2	2
	4	max	80	48	32	3	2
		min	96	48	48	3	3
	0		128	64	64	4	4
8	8	max	64	32	32	4	4
		min	72	40	32	5	4
	4	max	96	48	48	6	6
		min	96	48	48	6	6
	0		136	72	64	9	8
4	8	max	68	36	32	9	8
		min	72	36	36	9	9
	4	max	96	48	48	12	12
		min	96	48	48	12	12
	0		140	72	68	18	17

Table 6.3. Results for each sender node for the ET schedules.

## 7. PROBLEMS ENCOUNTERED

This chapter will describe the problems encountered during the course of the project. Most of the problems concerned the hardware, more specifically the backplane bus used to connect the processor and communication boards.

### 7.1. Network configuration and start-up

In the early stages of the project, there was extensive trouble in attaining reliable and predictable behaviour from the TTCAN communication boards. Initially, the network did not start at all; no activity was detected on the bus, which meant that no synchronisation attempts were made. After some time, when several check routines ensuring correct configuration results had been implemented, the network occasionally started up as intended. Still, there were major problems related to the start-up sequence and there was trouble with transferred message data showing wrong values. Typical cases were received hexadecimal values of 01, 23 or 41 where they should have been 00, 20 and 40, respectively. In fact, in all these cases the received value was higher than expected. This led to suspicions that in some way, logical ‘zeros’ were flipped into ‘ones’. If this suspected bit flipping also applied to the configuration sequence, it could explain the very unpredictable behaviour that had been observed. The configuration of the TTCAN controller involves setting a large number of registers, where a faulty value of one single bit can have devastating effects. The whole configuration could be inconsistent, leading to error conditions and/or severe start-up problems. The project therefore embarked on a quest for the cause of the unholy bit flips.

### 7.2. The backplane bus

The suspicions of what was causing the bit flips were soon concentrated to the backplane bus, connecting the processor board to the real-time communication board. At all nodes, the bus consisted of a segment from one large original bus that at the start of the project was, quite brutally, cut in pieces. One such segment is shown in figure 7.2 on the next page. The length of the bus segments was approximately 6.5 centimetres and they were non-terminated. That is, the latches that are used to multiplex the address and data lines had no pull-down capabilities. There were some worries that this could make the bus lines act like capacitors, which take an amount of time to discharge, when changing the logical level from ‘one’ to ‘zero’. An investigation was conducted by another thesis group [16] and their result from probing the bus with a logic analyser is shown in figure 7.1. It clearly shows the well-known characteristic of a capacitor, with a fall time of between 2 and 3 microseconds. With this result, it is clear that

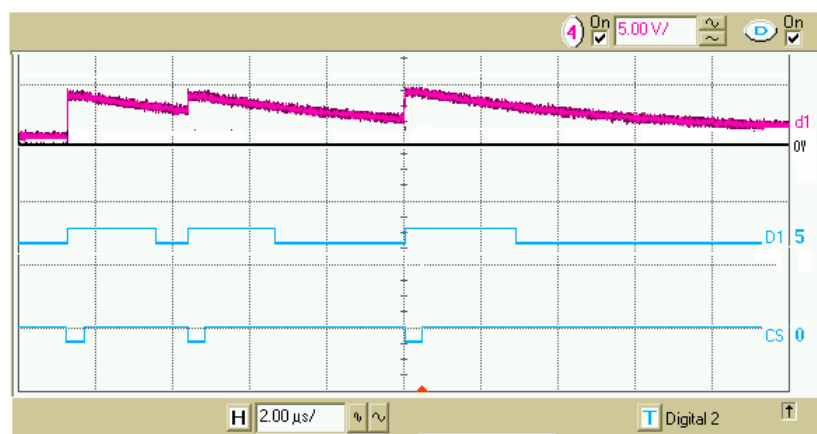


Figure 7.1. Signals on the backplane bus.

a ‘zero’ after a ‘one’ could be mistaken for a ‘one’ if the bit is sampled before the bus level has fallen below the threshold value. In the following sub-sections, the actions taken to reduce the impact of this problem are described.

### 7.2.1. Double memory access

The problems were related to the time it took for each line to adopt the correct level. When reading from or writing to the TTCAN registers, it would therefore be helpful if the bus lines could hold the data for a longer period of time. Then, they would eventually adopt the correct level. This realisation led to new implementations of the read and write functions, where the memory was accessed twice each operation. A call to the read function then resulted in two consecutive memory accesses at the same address, where the second value read was the one returned by the function. The write operation was constituted by a read/write combination at the same address. It was considered that this “double memory access” strategy should increase the probability that the bus lines really do hold the correct address the second access. The measure was taken quite early in the project and showed to eliminate a large part of the backplane bus problems.

### 7.2.2. Shorter point-to-point bus

The problems were assumed to be associated with undesired capacitances in the bus lines. Therefore, attempts to minimise these capacitances by constructing shorter buses with point-to-point connections were made. First, a hand-wired bus was constructed and then one with the flat-cable connectors shown in figure 7.3. The result was striking: The point-to-point connections and reduction of the bus length to 3 centimetres resulted in significantly higher reliability and much less data errors. All nodes were supplied with the flat-cable construction. When combined with interrupt routines that detected and eliminated configuration errors, the higher level of reliability reached was considered to be sufficient. No further investigations in the matter were therefore conducted.

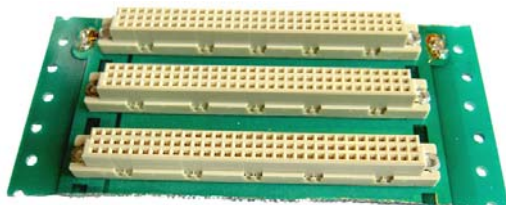


Figure 7.2. The first backplane bus.

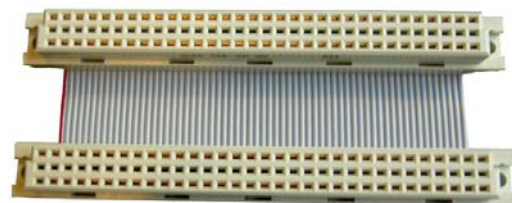


Figure 7.3. The second backplane bus.

## 7.3. Sensitive message objects

In addition to the trouble with system start-up and reliability, the experience from working with the system was that some of the 32 message objects seemed more error-prone than others. An extensive investigation of this phenomenon was conducted using the first version of the backplane bus, which was described above and is displayed in figure 7.2. The investigation consisted of a simple TTCAN message schedule, executed on two nodes, that each basic cycle sent three and received three messages to/from the other node. Network behaviour was observed and the received data was checked. When compiling the results, a striking picture appeared: almost every even-numbered object was concluded to be unreliable, while most of the odd-numbered ones had shown correct performance. Also, low-numbered objects seemed to be more frequently faulty than high-numbered ones. All conclusions can be seen in table 7.1. Note that message object 32 in hardware is represented with the hexadecimal number 00000. A connection with the bit flip behaviour described above seemed natural. The

fact that some message objects showed higher vulnerability to faults than others could be the result of bit flips from ‘zero’ to ‘one’ on the backplane bus. An investigation of the message objects while using the second backplane bus described above and shown in figure 7.3 would probably have showed less erroneous behaviour. Such an investigation has however not been conducted.

Object nr	Binary	Receive	Transmit	Conclusion
2	00010	✗	✗	
3	00011	✗	✗	
4	00100	✓	✗	
5	00101	✓	✓	✓
6	00110	✓	✗	
7	00111	✓	✓	✓
8	01000	✗	✗	
9	01001	✓	✗	
10	01010	✓	✗	
11	01011	✓	✓	✓
12	01100	✗	✗	
13	01101	✓	✓	✓
14	01110	✗	✗	
15	01111	✓	✓	✓
16	10000	✗	✗	
17	10001	✓	✓	✓
18	10010	✓	✗	
19	10011	✓	✓	✓
20	10100	✗	✗	
21	10101	✓	✓	✓
22	10110	✓	✗	
23	10111	✓	✓	✓
24	11000	✗	✗	
25	11001	✓	✓	✓
26	11010	✗	✗	
27	11011	✓	✓	✓
28	11100	✓	✓	✓
29	11101	✓	✓	✓
30	11110	✓	✗	
31	11111	✓	✓	✓
32	00000	✗	✗	

Table 7.1. Results from the message object investigation.

#### 7.4. Function call order

During the configuration sequence of the TTCAN controller, all message objects are initialised with their proper values. When the reference message initialisation was placed before the initialisation of the other objects, the system did not start at all. Switching the order of these two function calls, which by all logical reasoning should be independent, enabled the system to start as desired. The matter has not been investigated and the reason for this strange behaviour has therefore not been revealed. This issue remains to be one of the great mysteries that puzzle mankind, right up there with crop circles and Stonehenge.

## **8. CONCLUSION**

In this thesis, a reference application for the time-triggered network protocol TTCAN has been developed and used for a network performance investigation. Time-triggered protocols provide guaranteed response times, which is essential for certain real-time applications to work properly. As such applications and by-wire systems are forecast to be dominating in future vehicles, replacing many of the traditional mechanical and hydraulic systems, the performance of time-triggered networks is an interesting subject to investigate. In order to accomplish such an investigation, a basic driver package for the TTCAN protocol and certain GAST hardware had to be developed. The hardware used was the G1 processor board and the TTCAN communication board. Also, to facilitate configuration and debugging, a graphical configuration tool for TTCAN was developed. The tool was named Comtest and provides a graphical environment for setting up nodes with the main configuration parameters, as well as with message objects and triggers.

The results from the reference application showed, in all essentiality, total agreement with the predicted behaviour. All messages transferred in exclusive time windows were transferred when they were supposed to. The messages in arbitrating time windows were transferred as soon as there was room for them on the bus, and in priority order. When looking at the system performance, one conclusion is that an appropriate and well-designed schedule is of major importance. Message sizes and data quantities must be carefully analysed and fit into suitable time windows. Then, long gaps in the communication can be avoided. When using such well-designed schedules, there were little differences in performance between the different schedule classes. Nevertheless, studying all the tests, the ET schedules showed the best performance. They were followed by the dense TT/ET, the sparse TT/ET and the TT schedules, in that order. However, the generally good performance of ET schedules comes at the price of no guaranteed response times. The different TT/ET schedules proved to be a good compromise, with both guaranteed response times and the flexibility to transfer event-triggered messages.

When using well-designed schedules with time windows adapted for the message sizes that actually are sent, the number of basic cycles had minor impact on the system performance. When sending messages of different sizes, fewer basic cycles and longer arbitrating time windows were beneficial. Using the whole data capacity of 8 bytes per CAN frame showed superior performance in terms of effective bandwidth. When sending less data, the schedule needs to be adapted to avoid larger performance deficiencies than necessary. The number of stuff bits became a factor when sending messages in long arbitrating time windows. By keeping this number at a minimum, system performance can be enhanced.

Summing up, the conclusion is that the tested time-triggered network provided good predictability and acceptable performance when run with well-designed schedules. The effective bandwidth was then also comparable with the bandwidth of event-triggered CAN communication. However, the protocols FlexRay and TTP/C introduce much less overhead on the bus and would probably show better results in this area. Future projects are encouraged to investigate this.

## REFERENCES

- [1] Oberto, G. (2001) Drive-by-wire show potential in future car design.  
*Evolution*. 2001-11-15.  
<<http://evolution.skf.com>> (2006-05-23)
- [2] Dittmer, J. (2001) Are you ready for drive-by-wire?  
*Frost & Sullivan*. 2001-11-14.  
<<http://www.frost.com>> (2006-05-23)
- [3] Jackson, J. (2003) Mechatronics - from fuel injection to robotic soccer stars.  
*Evolution*. 2003-05-28.  
<<http://evolution.skf.com>> (2006-05-23)
- [4] Kvist Aronsen, A. (2005) Communication controllers in safety-critical applications.  
<<http://www.aronsen.de/university.html>> (2006-05-23)
- [5] Bell, J. (2002) Network protocols used in the automotive industry.  
*The University of Wales, Aberystwyth*. 2002-07-24.  
<<http://www.aber.ac.uk>> (2006-05-23)
- [6] Navet, N. Song, Y. et al. (2005) Trends in Automotive Communication Systems.  
*Proceedings of the IEEE*, vol 93, no. 6. 2005-06-06.  
<<http://www.loria.fr/~nnavet>> (2006-05-23)
- [7] Johansson, R. (2004) Time and event triggered communication scheduling for the CAN bus.  
*Göteborg: Chalmers Lindholmen University College*.
- [8] CAN Specification, Version 2.0  
*Bosch*. Sep 1991.  
<<http://www.semiconductors.bosch.de>> (2006-05-23)
- [9] Johansson, R. et al. (2003) On Communication Requirements for Control-by-Wire Applications.  
*Göteborg: Chalmers Lindholmen University College*.
- [10] TTCAN IP Module User's Manual, Revision 1.6  
*Bosch*. 2002-11-11.  
<<http://www.semiconductors.bosch.de>> (2006-05-23)
- [11] FlexRay Communications System, Protocol Specification, Version 2.1  
*FlexRay Consortium*. May 2005.  
<<http://www.flexray.com>> (2006-05-23)
- [12] *TTTech – Time-Triggered Technology*.  
<<http://www.tttech.com>> (2006-05-23)
- [13] *GAST Project*.  
<<http://www.chl.chalmers.se/gast>> (2006-05-23)

- [14] *CEDES*.  
<<http://www.cedes.com>> (2006-05-23)
  
- [15] Johansson, R. (2002) On calculating guaranteed message response times on the SAE J1939 bus.  
*Göteborg: Chalmers Lindholmen University College.*
  
- [16] Archer, C. Sjöblom, A. (2006) Membership implementations on time triggered architectures.  
*Göteborg: Chalmers University of Technology.*

## **APPENDICES**

*A. Results: Number of transferred messages*

*B. Results: Effective bandwidth*

*C. Results: Calculated minimal bus load*

*D. Reference application schedules*

*E. Comtest Tutorial*

*F. Running the Tutorial application*

*G. Calculation of bit timing parameters*

## A. Results: Number of transferred messages

Basic cycles	Data bytes	Stuff bits	Expected	Msgs
32	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		64	<b>64</b>
16	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		64	<b>64</b>
8	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		64	<b>64</b>
4	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		64	<b>64</b>

Table A.1. Number of expected and transferred messages for the TT schedules.

Basic cycles	Data bytes	Stuff bits	Expected	Msgs
32	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		96	<b>96</b>
16	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		96	<b>96</b>
8	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		96	<b>96</b>
4	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		96	<b>96</b>

Table A.2. Number of expected and transferred messages for the sparse TT/ET schedules.

Basic cycles	Data bytes	Stuff bits	Expected	Msgs
32	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64	<b>64</b>
		min	64	<b>64</b>
	0		96	<b>96</b>
16	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64-80	<b>64</b>
		min	64	<b>64</b>
	0		96	<b>96</b>
8	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	72	<b>72</b>
		min	72-80	<b>80</b>
	0		96-104	<b>96</b>
4	8	max	64	<b>64</b>
		min	64-68	<b>68</b>
	4	max	72-76	<b>76</b>
		min	72-80	<b>80</b>
	0		96-108	<b>100</b>

Table A.3. Number of expected and transferred messages for the dense TT/ET schedules.

Basic cycles	Data bytes	Stuff bits	Expected	Msgs
32	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	64-96	<b>96</b>
		min	64-96	<b>96</b>
	0		128-160	<b>128</b>
16	8	max	64	<b>64</b>
		min	64	<b>64</b>
	4	max	80-96	<b>80</b>
		min	80-96	<b>96</b>
	0		128-144	<b>128</b>
8	8	max	64	<b>64</b>
		min	64-72	<b>72</b>
	4	max	80-96	<b>96</b>
		min	80-96	<b>96</b>
	0		128-152	<b>136</b>
4	8	max	64-68	<b>68</b>
		min	64-76	<b>72</b>
	4	max	84-96	<b>96</b>
		min	84-100	<b>96</b>
	0		128-152	<b>140</b>

Table A.4. Number of expected and transferred messages for the ET schedules.

	Number of basic cycles			
	32	16	8	4
8 bytes, max stuff bits	64	64	64	64
8 bytes, min stuff bits	64	64	64	64
4 bytes, max stuff bits	64	64	64	64
4 bytes, min stuff bits	64	64	64	64
0 bytes	64	64	64	64

Table A.5. Number of transferred messages for the TT schedules.

	Number of basic cycles			
	32	16	8	4
8 bytes, max stuff bits	64	64	64	64
8 bytes, min stuff bits	64	64	64	64
4 bytes, max stuff bits	64	64	64	64
4 bytes, min stuff bits	64	64	64	64
0 bytes	96	96	96	96

Table A.6. Number of transferred messages for the sparse TT/ET schedules.

	Number of basic cycles			
	32	16	8	4
8 bytes, max stuff bits	64	64	64	64
8 bytes, min stuff bits	64	64	64	68
4 bytes, max stuff bits	64	64	72	76
4 bytes, min stuff bits	64	64	80	80
0 bytes	96	96	96	100

Table A.7. Number of transferred messages for the dense TT/ET schedules.

	Number of basic cycles			
	32	16	8	4
8 bytes, max stuff bits	64	64	64	68
8 bytes, min stuff bits	64	64	72	72
4 bytes, max stuff bits	96	80	96	96
4 bytes, min stuff bits	96	96	96	96
0 bytes	128	128	136	140

Table A.8. Number of transferred messages for the ET schedules.

## B. Results: Effective bandwidth

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	29.0	33.6	36.5	38.2
<b>8 bytes, min stuff bits</b>	29.0	33.6	36.5	38.2
<b>4 bytes, max stuff bits</b>	29.0	33.6	36.5	38.2
<b>4 bytes, min stuff bits</b>	29.0	33.6	36.5	38.2

Table B.1. Effective bandwidth for the TT schedules.

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	29.0	33.6	36.5	38.2
<b>8 bytes, min stuff bits</b>	29.0	33.6	36.5	38.2
<b>4 bytes, max stuff bits</b>	21.7	25.2	27.4	28.6
<b>4 bytes, min stuff bits</b>	21.7	25.2	27.4	28.6

Table B.2. Effective bandwidth for the sparse TT/ET schedules.

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	29.0	33.6	36.5	40.6
<b>8 bytes, min stuff bits</b>	29.0	33.6	36.5	38.2
<b>4 bytes, max stuff bits</b>	21.7	25.2	29.7	32.2
<b>4 bytes, min stuff bits</b>	21.7	25.2	32.0	33.4

Table B.3. Effective bandwidth for the dense TT/ET schedules.

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	29.0	33.6	36.5	40.6
<b>8 bytes, min stuff bits</b>	29.0	33.6	41.1	43.0
<b>4 bytes, max stuff bits</b>	21.7	21.0	27.4	28.6
<b>4 bytes, min stuff bits</b>	21.7	25.2	27.4	28.6

Table B.4. Effective bandwidth for the ET schedules.

### C. Results: Calculated minimal bus load

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	82.0	82.0	81.9	81.9
<b>8 bytes, min stuff bits</b>	82.0	82.0	81.9	81.9
<b>4 bytes, max stuff bits</b>	82.0	82.0	81.9	81.9
<b>4 bytes, min stuff bits</b>	82.0	82.0	81.9	81.9
<b>0 bytes</b>	82.0	82.0	81.9	81.9

Table C.1. Minimal bus loads for the TT schedules.

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	82.0	82.0	81.9	81.9
<b>8 bytes, min stuff bits</b>	82.0	82.0	81.9	81.9
<b>4 bytes, max stuff bits</b>	74.8	73.6	72.8	72.4
<b>4 bytes, min stuff bits</b>	74.8	73.6	72.8	72.4
<b>0 bytes</b>	82.7	82.8	82.8	82.8

Table C.2. Minimal bus loads for the sparse TT/ET schedules.

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	82.0	82.0	81.9	81.9
<b>8 bytes, min stuff bits</b>	82.0	82.0	81.9	86.8
<b>4 bytes, max stuff bits</b>	74.8	73.6	79.9	83.4
<b>4 bytes, min stuff bits</b>	74.8	73.6	86.9	87.1
<b>0 bytes</b>	82.7	82.8	82.8	85.3

Table C.3. Minimal bus loads for the dense TT/ET schedules.

	Number of basic cycles			
	32	16	8	4
<b>8 bytes, max stuff bits</b>	82.0	82.0	81.9	86.8
<b>8 bytes, min stuff bits</b>	82.0	82.0	91.3	91.7
<b>4 bytes, max stuff bits</b>	90.0	78.2	91.9	92.4
<b>4 bytes, min stuff bits</b>	90.0	91.2	91.9	92.4
<b>0 bytes</b>	83.4	83.6	88.4	91.2

Table C.4. Minimal bus loads for the ET schedules.

### D. Reference application schedules

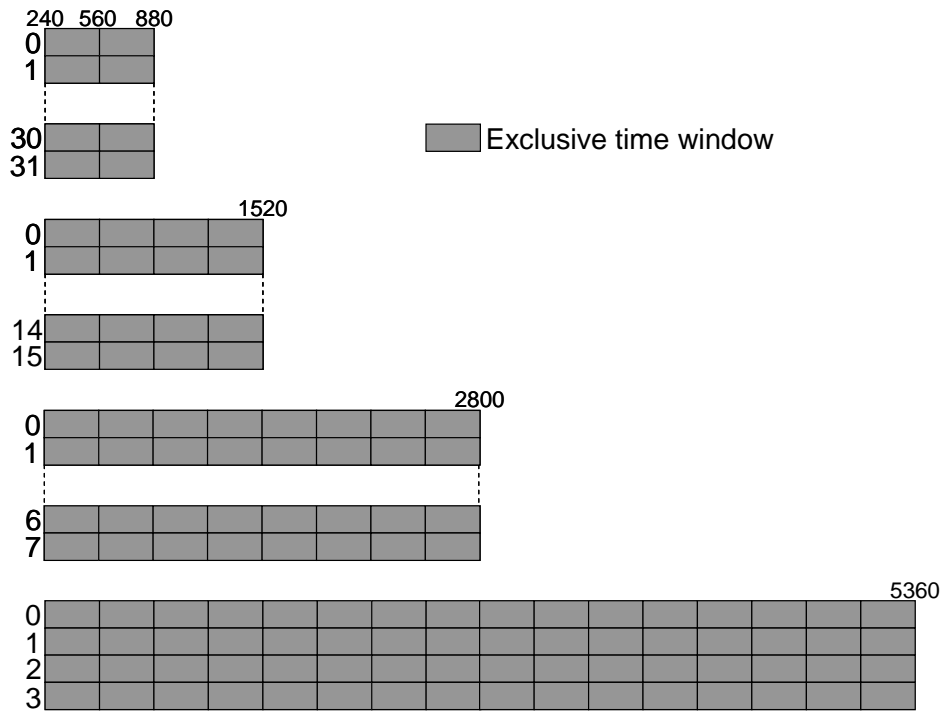


Figure D.1. TT schedules with 32, 16, 8 and 4 basic cycles.

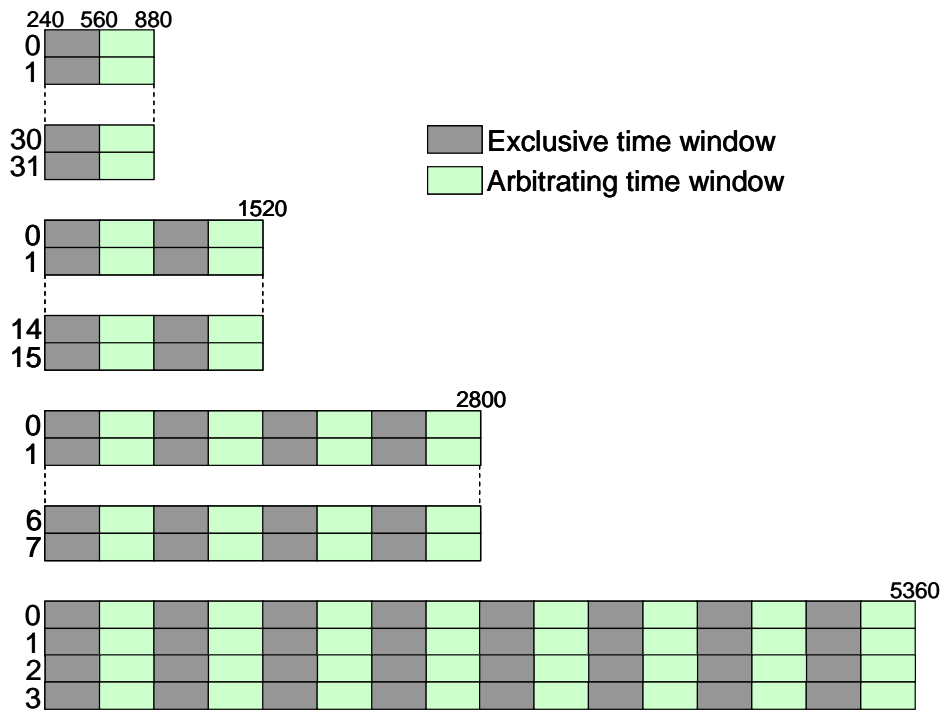


Figure D.2. Sparse TT/ET schedules with 32, 16, 8 and 4 basic cycles.

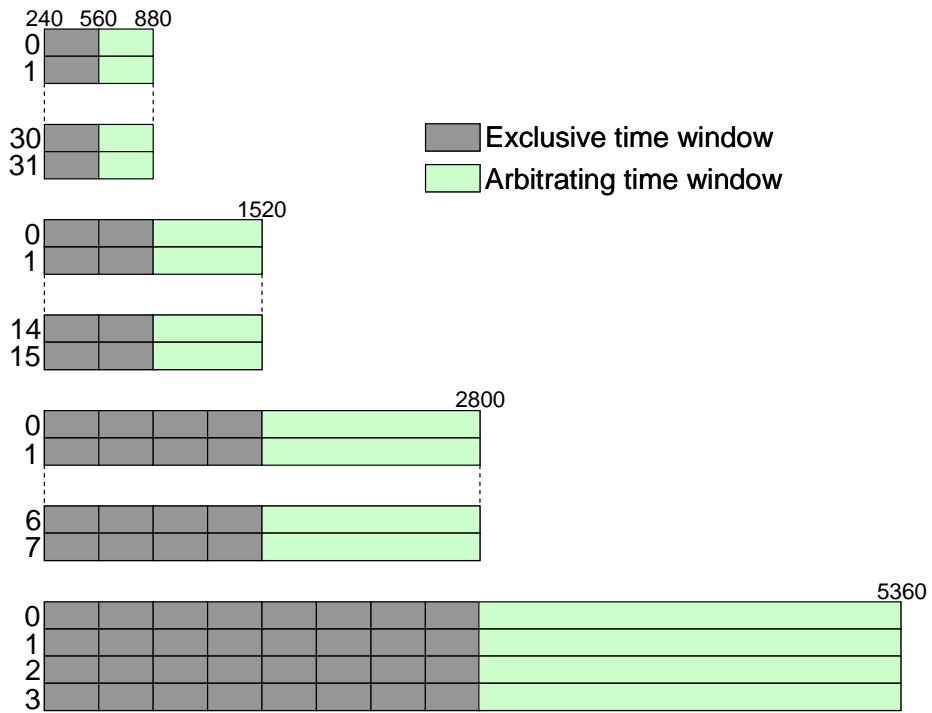


Figure D.3. Dense TT/ET schedules with 32, 16, 8 and 4 basic cycles.

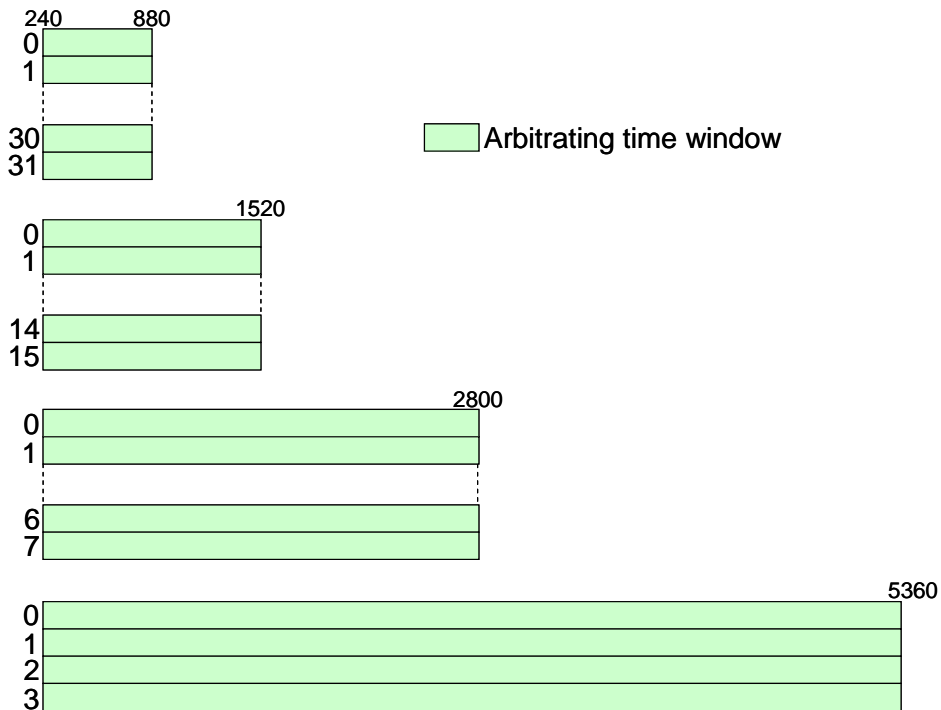


Figure D.4. ET schedules with 32, 16, 8 and 4 basic cycles.

## E. Comtest Tutorial

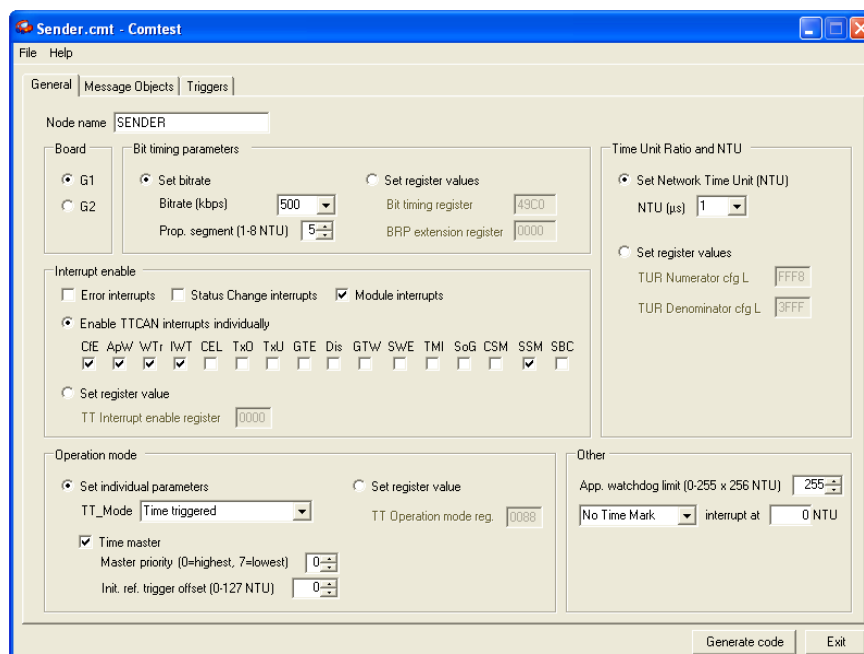
This tutorial features step-by-step instructions on how to set up a simple TTCAN network and message schedule using Comtest. The network to be configured is comprised by two G1 nodes with one TTCAN module each, connected to the G1s through backplane buses. The two nodes will communicate time-triggered as specified in the table below, where time is measured in NTUs. One of the nodes will act as a sender of both time-triggered and event-triggered messages, which the other node will receive. Table E.1 shows the very simple time schedule. The matrix cycle consists of just one basic cycle. **Sender** sends a periodic, time-triggered message in an exclusive time window at time mark 500. Additionally, **Sender** has the possibility to send a message in an arbitration time window at time mark 1000. **Sender** will also be the time master and therefore send a reference message every basic cycle, at time mark 1500. **Receiver** will receive this message and adjust its synchronisation to match the time master.

	500	1000	1500
<b>Basic cycle 0</b>	<b>Sender Message 1</b>	<b>Sender Message 2</b>	<b>Sender Reference msg</b>

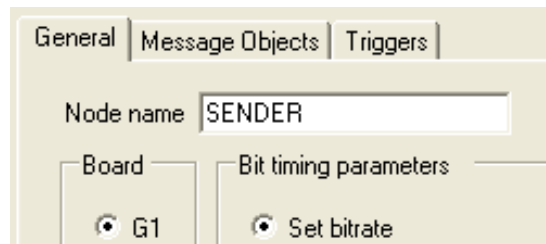
Table E.1. The matrix cycle.

### General settings

The General tab is where all fundamental parameters of the TTCAN network are configured. Some of these, such as the bitrate and the NTU, have to be the same on all nodes in the network. The General tab is also where node-specific settings like the name and time master priority is specified.



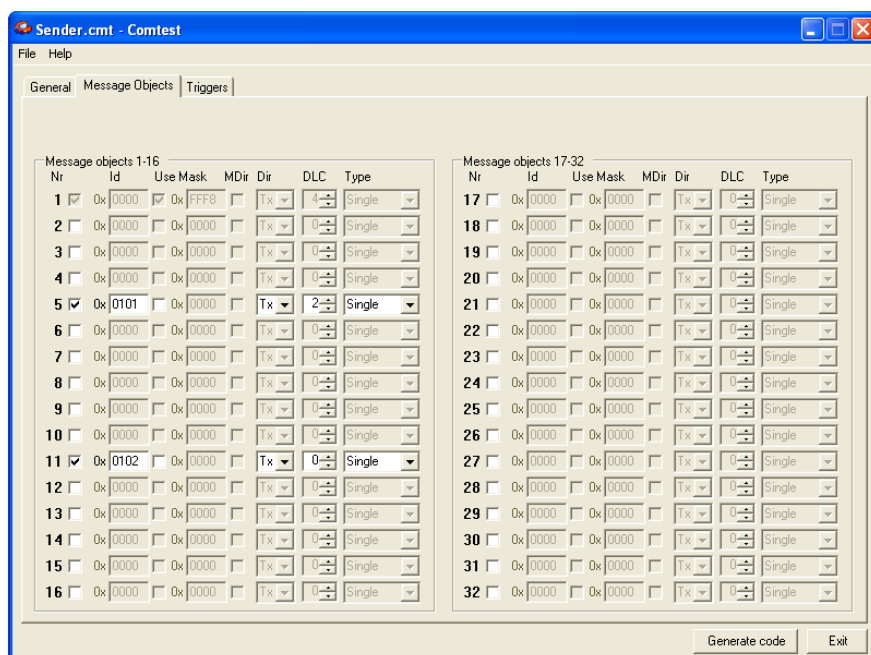
1. Open Comtest and select the General tab. Write the name of the node being configured in the **Node name** box. In this case, the name is “SENDER”. The node name can be used as an aid to implement node-specific behaviour in generic applications.



2. Check that the settings in the **Board**, **Bit timing parameters** and **Time unit ratio and NTU** areas are the same as in the figure on the previous page. These are settings concerning system clock frequency and length of the bits on the TTCAN bus. By tuning these parameters, differently constructed units can be configured to communicate over a common bus.
3. In the **Interrupt enable** box, check **Module interrupts**, **CfE**, **ApW**, **WTr**, **IWT** and **SSM**. This enables interrupts from the TTCAN module at fatal errors, initialisation watch trigger errors and at the start of each matrix cycle. By the use of the matrix cycle interrupt, message handling can be synchronised with the schedule. This synchronisation is an important ingredient in time-triggered communication.
4. In the **Operation mode** box, select **Time triggered** in the **TT\_Mode** menu. Check the **Time master** box and set both the priority and offset to **0**.
5. In the **Other** box, check that the settings are the same as in the figure on the previous page.

### Message objects

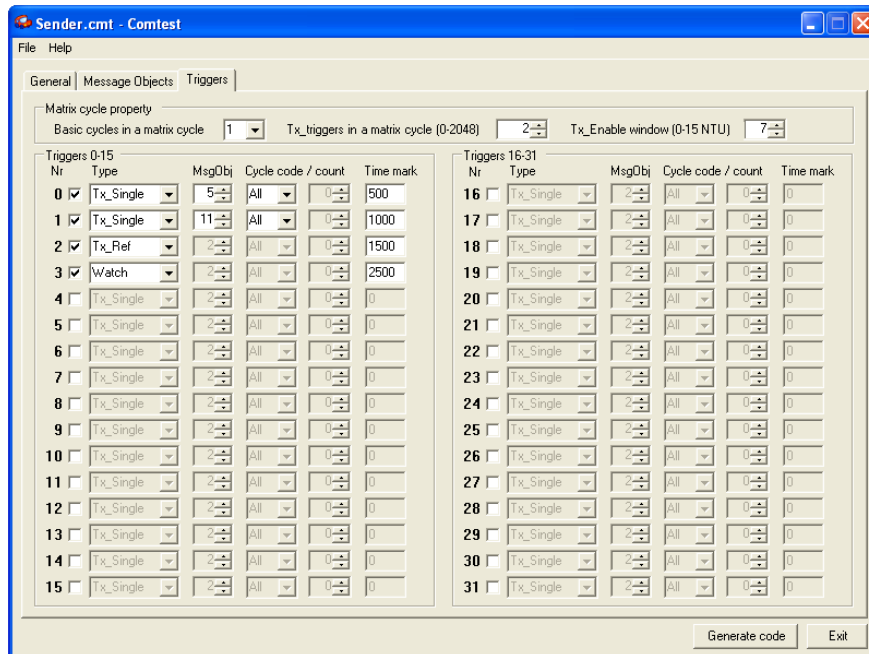
In this section, the message objects are configured. The settings of a message object include message identifier, acceptance mask, data direction and data length. Objects can also be configured to be part of a queue of incoming messages.



6. Select the **Message Objects** tab. Configure objects **5** and **11** to be **Tx** (transmit) objects, as shown in the figure. Also, set the **Ids**, the **DLCs** and the **Types** to the same values as in the figure. With this setting, **Sender** will send messages **0101** and **0102** from objects **5** and **11**, respectively.

### Triggers

The triggers define time marks at which actions will be taken by the TTCAN module. The most commonly used triggers initiate sending and receiving of messages, specifying which message object to use and the time mark and periodicity for the trigger in question.

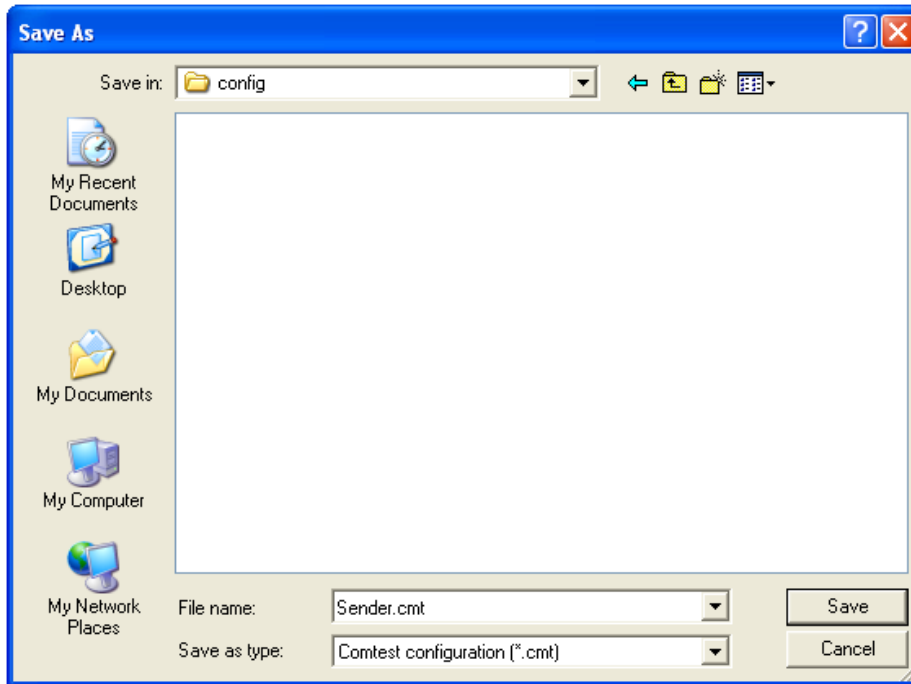


7. Select the **Triggers** tab. Select **1** in the **Basic cycles in a matrix cycle** menu. The number of **Tx\_triggers in a matrix cycle** is **2** (one for the exclusive window, one for the arbitration windows). Leave the **Tx\_Enable window** at **7** NTU.
8. Input the trigger data as shown in the figure. The triggers are specified according to the schedule in table E-1. The Tx\_Ref trigger causes a time master to send its reference message, which time slaves then may synchronise to. The Watch trigger is a guard, causing an interrupt if no reference message has been received at its time mark.

### Save and generate code

The configuration of **Sender** is now complete. The only thing left is to save the configuration and then generate code from it.

9. Save the configuration by selecting **Save As** from the **File** menu and specifying a file name.



10. Press **Generate code**. After browsing to the desired target folder, click **OK**. The files **TTCANconfig.h** and **TTCANconfig.c** can now be found in the specified folder. They should look like the samples on the next page (some rows omitted).

## TTCANconfig.h

```
/* TTCANconfig.h for SENDER
   Generated from 'Sender.cmt' by Comtest
   2006-05-25 08:42:59 */

#ifndef TTCANCONFIG_H
#define TTCANCONFIG_H

#define G1_BOARD // Processor board
#define SENDER // Node name

#define BTRATE 500 // Bitrate (kbps)
#define PROP_SEG 5 // Propagation time segment (NTU)

// CAN protocol related registers
#define CAN_CONTROL_CFG 0x0002 // Enabled interrupts

// Time triggered communication registers
#define TT_OPERATIONMODE_CFG 0x0082 // Time master, MPR=0, TTMode_2
#define TT_MATRIXLIMITS1 0x0002 // 2 Tx Triggers in a matrix cycle
#define TT_MATRIXLIMITS2_CFG 0x0700 // TEW=7, 1 basic cycles
#define TT_APPLICATIONWATCHDOGLIMIT 0x00FF // Application watchdog limit 65280 NTUs
#define TT_INTERRUPTENABLE 0xF002 // Enabled interrupts
#define TUR_NUMERATORCONFIGURATION 0xFFFF // Numerator=0x1FFF8
#define TUR_DENOMINATORCONFIGURATION 0x3FFF // Denominator=0x3FFF (TUR=8)
#define TT_CLOCKCONTROL_CFG 0x0000 // TMC=0
#define TT_TIMERMARK 0x0000 // Interrupt at time 0

#endif
```

## TTCANconfig.c

```
/* TTCANconfig.c for SENDER
   Generated from 'Sender.cmt' by Comtest
   2006-05-25 08:42:59 */

#include "TTCANdefine.h"
#include "TTCANtype.h"

T_TTCAN_SCHEDULE schedule =
{
    {
        // Message Objects
        // valid, messageNr, messageID, uMask, idMask, dirMask, direction, dataLength,
        // messageType
        { VALID, MESSAGE_1, 0x0000, USE_MASK, 0xFFFF8, NOT_USED, TX_MESSAGE, 4, SINGLE },
        { INVALID, MESSAGE_2, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_3, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_4, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { VALID, MESSAGE_5, 0x0101, NOT_USED, NOT_USED, NOT_USED, TX_MESSAGE, 2, SINGLE },
        { INVALID, MESSAGE_6, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_7, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_8, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_9, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_10, NULL, NULL, NULL, NULL, NULL, NULL, NULL },

        { VALID, MESSAGE_11, 0x0102, NOT_USED, NOT_USED, NOT_USED, TX_MESSAGE, 0, SINGLE },
        { INVALID, MESSAGE_12, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_13, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
        { INVALID, MESSAGE_14, NULL, NULL, NULL, NULL, NULL, NULL, NULL },

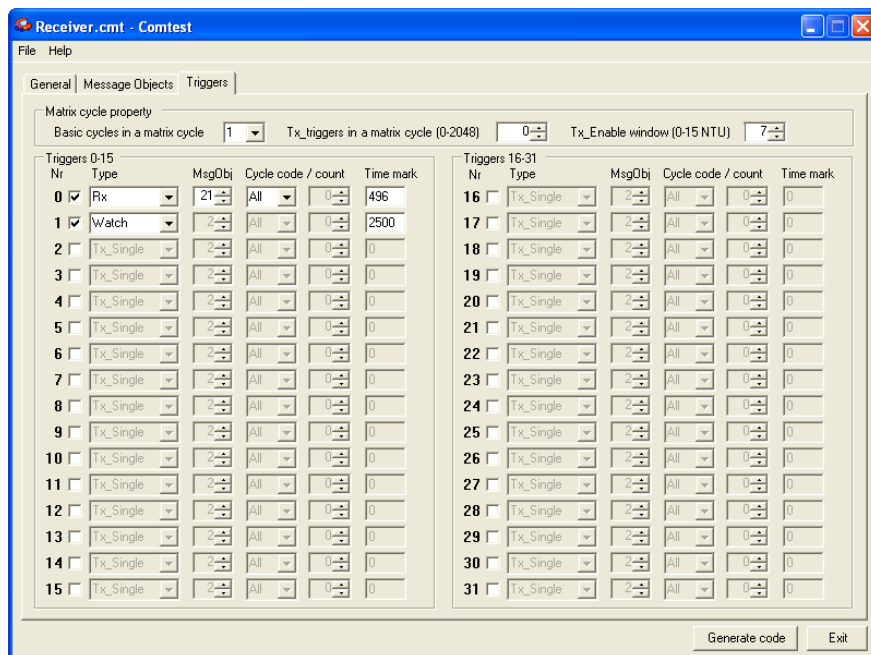
        ...
        { INVALID, MESSAGE_32, NULL, NULL, NULL, NULL, NULL, NULL, NULL }
    },
    {
        // Triggers
        // triggerNr, triggerType, messageNr, cycleCode, cycleCount, timeMark
        { TRIGGER_0, TX_SINGLE, TRIGGERMESSAGE_5, ALL_CYCLES, NULL, 500 },
        { TRIGGER_1, TX_SINGLE, TRIGGERMESSAGE_11, ALL_CYCLES, NULL, 1000 },
        { TRIGGER_2, TX_REF, TRIGGERMESSAGE_1, ALL_CYCLES, NULL, 1500 },
        { TRIGGER_3, WATCH, TRIGGERMESSAGE_32, ALL_CYCLES, NULL, 2500 },
        { TRIGGER_4, ENDOFLIST, TRIGGERMESSAGE_32, ALL_CYCLES, NULL, NULL_TIMERMARK },
        { TRIGGER_5, ENDOFLIST, TRIGGERMESSAGE_32, ALL_CYCLES, NULL, NULL_TIMERMARK },

        ...
        { TRIGGER_31, ENDOFLIST, TRIGGERMESSAGE_32, ALL_CYCLES, NULL, NULL_TIMERMARK }
    }
};
```

## Configuring Receiver

The steps for configuring **Receiver** are the same as for **Sender**. Follow the numbered list above, but with the following alterations:

1. The node name should be “**RECEIVER**”.
2. Same as above.
3. Same as above.
4. In the **Operation mode** box, uncheck the **Time master** box. This configures the node to be a time slave.
5. Same as above.
6. Select the **Message Objects** tab. Disable objects **5** and **11**. Configure objects **21** and **25** to be **Rx** (receive) objects with the identifiers **0101** and **0102**. Also, set the **DLCs** to **2** and **0**, respectively, and the **Types** to **Single**. With this setting, **Receiver** will receive messages **0101** and **0102** from **Sender** in objects **21** and **25**.
7. Select the **Triggers** tab. Select **1** in the **Basic cycles in a matrix cycle** menu. The number of **Tx\_triggers in a matrix cycle** is **0**. Leave the **Tx\_Enable window** at **7** NTU.
8. Input the trigger data as shown in the figure below. Note that **Receiver**, being a time slave, does not send a reference message.



9. Same as above.
10. Same as above. Though, a different target folder is recommended to avoid overwriting the files generated for **Sender**.

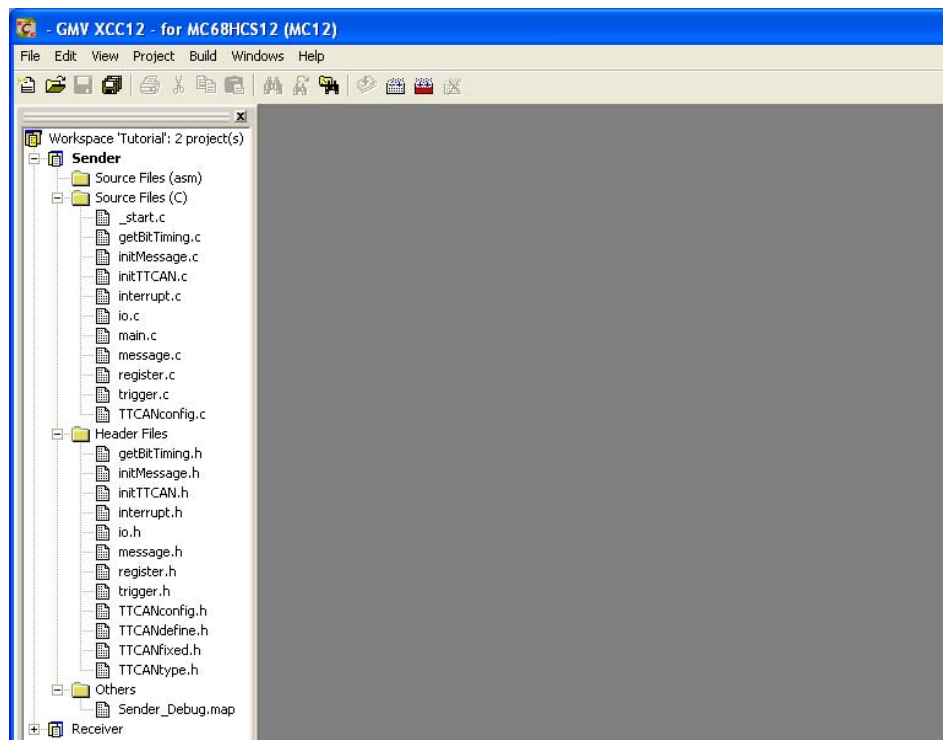
## F. Running the Tutorial application

The Tutorial application consists of two nodes, **Sender** and **Receiver**. **Sender** sends both time-triggered and event-triggered messages. The time-triggered messages, which contain a counter value that **Sender** increments at the start of every matrix cycle, are continuously received by **Receiver**. The counter, being a 16-bit value, wraps every 98.5 seconds. When a key is pressed on the keyboard of the PC monitoring **Receiver**, the latest received value will be printed to the screen.

In addition to the time-triggered messages, **Sender** sends an event-triggered message every time a key is pressed on the keyboard of the monitoring PC. This message is sent in the next arbitrating time window and its reception is indicated by a printed line on **Receiver**'s screen.

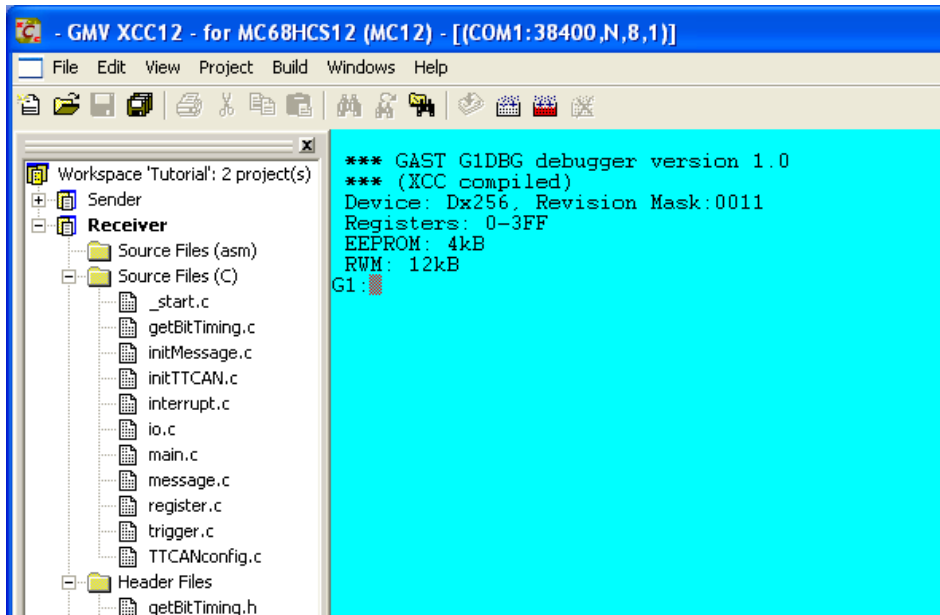
### Running the application

1. Start **XCC** and choose **Open Workspace** from the **File** menu. Browse to the **Tutorial** project folder and select **Tutorial.w12**.
2. If **Sender** is not already the active project, indicated by a **bold-letter** title, right-click the title and select **Set as Active Project**.

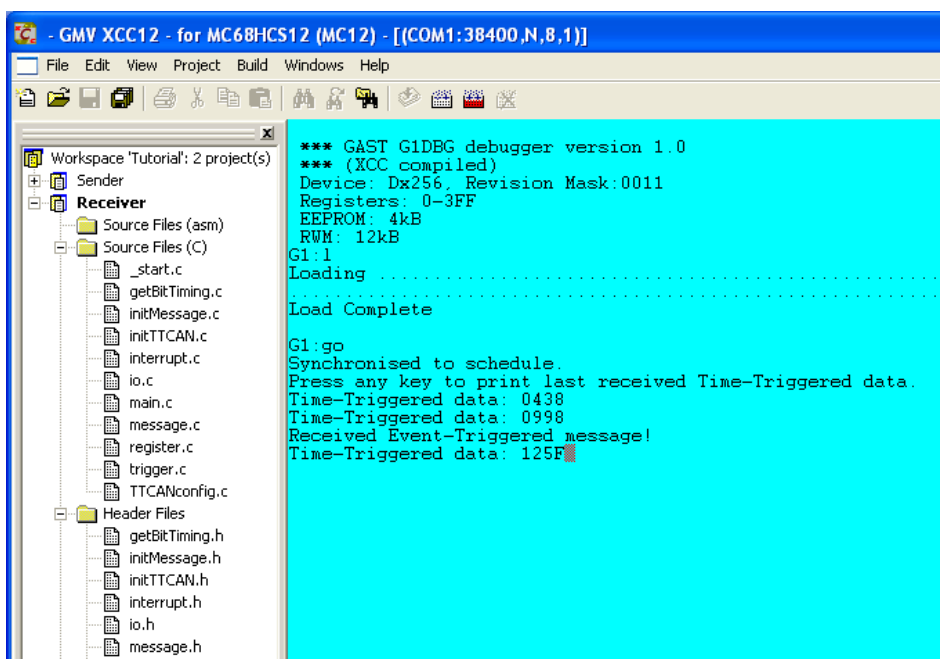


3. If you want to alter the schedule, which can be studied in the project file **TTCANconfig.c**, continue with instruction **4** and **5**. Otherwise, jump to instruction **6**.
4. Generate code for the desired schedule from **Comtest**. The **Comtest tutorial** in **Appendix E** describes how this is done. The configuration files for the already generated schedule can be found in the **config** folder in the **Tutorial** project folder.
5. Check that the project files **TTCANconfig.h** and **TTCANconfig.c** has been updated with the new configuration.
6. Click the **Build all** button to compile the code. A file named **Sender\_Debug.s19** will be generated to the Debug folder.

- Repeat steps 2-6 for the **Receiver** project.
- Open a **Sender** terminal window by selecting **Build > Start Debug > Open Terminal** and selecting the **COM port** to which one of the **G1s** is connected to.
- Reset the G1** and check that the output looks something like in the figure below.



- Right-click** in the terminal window and choose **Load new**. After browsing to the file **Sender\_Debug.s19**, click **Open** and wait for the file to be loaded to the G1.
- Repeat steps 8-10 for the **Receiver** project.
- Write **go** in both terminal windows to start the applications.
- The output on the **Receiver** side should now look like in the figure below.



## G. Calculation of bit timing parameters

```
/* *****  
Function:      getBi tTiming  
Description:   Calculates the values for the Bi tTiming and  
              BRP_Extensi on registers.  
  
Parameters:   bi tTimingPointer to the value for Bi tTiming  
              brpExtensi on Pointer to the value for BRP_Extensi on  
***** */  
  
void getBi tTiming( unsigned int *bi tTiming, unsigned int *brpExtensi on )  
{  
    unsigned int tqPerBi t, phaseSeg1, phaseSeg2, sjw;  
    unsigned int brp = 0;  
    unsigned int maxTq = SYNC_SEG + PROP_SEG + 2*MAX_PHASE_SEG;  
                                                // Time quanta per bit limit  
  
    tqPerBi t = maxTq + 1;  
  
    while( tqPerBi t > maxTq )  
    {  
        brp++;                                // Increase prescaler  
  
        if( SYSTEM_FREQ % ( brp * BITRATE ) == 0 )  
            tqPerBi t = SYSTEM_FREQ / ( brp * BITRATE ); // Time quanta per bit  
    }  
  
    if( SYSTEM_FREQ % BITRATE != 0 || brp > 1024 )  
        // Bit time must be multiple of clock period  
    {  
        puts( "ILLEGAL BITRATE" );  
        hal tExecution();  
    }  
  
    phaseSeg1 = ( tqPerBi t - SYNC_SEG - PROP_SEG ) / 2;  
    phaseSeg2 = tqPerBi t - SYNC_SEG - PROP_SEG - phaseSeg1;  
                                                // Phase2 possibly 1 tq longer than Phase1  
  
    if( phaseSeg1 > 4 )  
        sjw = 4;                                // Maximum sync jump width  
    else  
        sjw = phaseSeg1;  
  
    brp--;                                       // Register value one less than actual value  
  
    *brpExtensi on = brp >> 6;  
    brp = brp & 0x003F;  
    *bi tTiming = (phaseSeg2 - 1) << 12 |  
                  (phaseSeg1 + PROP_SEG - 1) << 8 | (sjw - 1) << 6 | brp;  
}
```