

## **Implementation of Membership algorithms in GAST-cluster with FlexRay**

Author	Mattias Bergström Johan Högberg
Document Id	029
Date	March 2007
Availability Status	Public Final



# CHALMERS



## Implementation of Membership algorithms in GAST-cluster with FlexRay

*Master of Science Thesis in the Computer Science and Engineering program*

MATTIAS BERGSTRÖM  
JOHAN HÖGBERG

Department of Computer Science and Engineering  
*Division of Computer Engineering*  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Göteborg, Sweden, 2007

Implementation of Membership algorithms in GAST-cluster with FlexRay

MATTIAS BERGSTRÖM

JOHAN HÖGBERG

© MATTIAS BERGSTRÖM, JOHAN HÖGBERG, March 2007.

Department of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Göteborg

Sweden

Telephone + 46 (0)31-772 1000

Göteborg, Sweden March 2007

## ***Abstract***

In recent years there has been a shift in the automotive industry away from the complex mechanical and hydraulic systems previously used for safety-critical functions such as steering and brakeage in automobiles. It is beyond any doubt that future generations of automobiles instead will implement distributed solutions where safety-critical functionality is performed by a number of small, light, and inexpensive ECUs (electronic control units) equipped with electromechanical actuators and sensors. The ECUs are in turn interconnected by a bus network. Since it is absolutely vital that distributed safety-critical systems such as steering or brakeage be as fault-tolerant as the systems they are replacing it is necessary to implement some sort of redundancy within the systems. Membership algorithms, which are the subject of this thesis project, may be described briefly as a mechanism for ECUs in the network to maintain a consistent view of what other ECUs in the network are malfunctioning, in order that the functionality performed by the malfunctioning nodes be rerouted instead to redundant operational ECUs. The purpose of this thesis project was to implement, test and evaluate the membership algorithms recently developed by Carl Bergenhem on a FlexRay-based cluster consisting of GAST G2 MPC565-based microcontroller boards paired with Freescale 4200 FlexRay communication controllers and interconnected by a FlexRay bus. A basic simulation application was also developed during the course of the thesis in order to more easily evaluate the membership algorithms on a limited scale. The thesis project was conducted within the scope of the CEDES research project.

### **Keywords**

CEDES, GAST, FlexRay, Time-Triggered Architecture, Membership Algorithms, MFR4200, Distributed Embedded Systems



## **Sammanfattning**

Under senare år har det i bilindustrin förekommit en stor förändring då många biltillverkare har valt att byta ut komplexa mekaniska och hydrauliska säkerhetskritiska system så som bromssystem och styrning, och istället implementera dessa funktioner med distribuerade system bestående av elektroniska styrsystem försedda med elektromekaniska aktuatorer och sensorer. Dessa styrsystem är i sin tur sammankopplade av ett bussnätverk. Eftersom det är en nödvändig förutsättning att distribuerade säkerhetskritiska system är lika feltoleranta eller mer feltoleranta än systemen de ersätter så är det nödvändigt att bygga in någon form av redundans i systemen. Medlemskapsalgoritmer, som är temat för examensarbetet, kan kortfattat beskrivas som en mekanism för styrsystemen i nätverket att kunna bilda sig en gemensam uppfattning om vilka andra styrsystem i nätverket som är felaktiga. På så sätt kan de funktioner som skulle utföras av den felaktiga noden istället kopplas om till ett fungerande redundant styrsystem. Syftet för detta examensarbete är att implementera, testa samt utvärdera de medlemskapsalgoritmer som nyligen utvecklats av Carl Bergenhem i ett FlexRay-baserat kluster bestående av GAST G2 MPC565-baserade mikrokontrollerkort hopparade med Freescale 4200 FlexRay kommunikationskontrollerkort. Dessa par är sedan sammanlänkade av en FlexRay-buss. En simuleringsapplikation av enkel modell utvecklades dessutom under examensarbetets gång för att på ett behändigt sätt kunna testa och utvärdera medlemskapsalgoritmerna i begränsad skala. Detta examensarbete har utförts inom ramen för CEDES-projektet.

### **Nyckelord**

CEDES, GAST, FlexRay, tids-triggad arkitektur, medlemskapsalgoritmer, MFR4200, distribuerade inbyggda system



## **Preface**

This masters thesis forms part of the CEDES (Cost-Efficient Dependable Electronic Systems) project; a cooperative project between Volvo Cars, Volvo, Autoliv, Chalmers University of Technology[1] and SP ( Swedish Testing and research institute) whose purpose is to develop low-cost electronic equipment for use especially in the automotive industry [2]. Our tutor Carl Bergenhem, as well as equipment needed for the project, was provided by SP.

We would like to thank the authors of the thesis immediately preceding ours (also part of the CEDES projects): Andreas Sjöblom and Christian Archer, “Membership implementation on time triggered architectures” for their support.

Furthermore, we would like to extend our appreciation to Mattias Pettersson and Petter Uvesten for their excellent thesis “A Distributed FlexRay-based Research Platform”, from which we have drawn much inspiration.

We would also like to acknowledge the technical report “Survey of Membership Agreement Protocols” by Carl Bergenhem, as well as the articles that we have based our membership implementation on. That is, the technical report “A Configurable Membership Service for Active Safety Systems” and the

Lastly, and most importantly, we would like to thank:

- **Roger Johansson** - Our thesis project examiner, who provided us much assistance with hardware- related issues as well as advice and inspiration due to his extensive knowledge about embedded systems.
- **Carl Bergenhem** - Our thesis project tutor from SP, whose ideas underlie the entire thesis project, and who contributed greatly to the development of the thesis with advice and ideas. Thank you for giving us the opportunity to conduct this masters thesis.



## Table of contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Purpose .....	2
1.2	Scope .....	2
1.3	Related work .....	3
1.4	Objectives .....	3
<b>2</b>	<b>Background .....</b>	<b>5</b>
2.1	Fault tolerance .....	5
2.1.1	Reliability .....	6
2.2	Dependability in the automotive industry .....	6
2.2.1	X-by-wire technology .....	6
2.2.2	Membership .....	7
2.3	G1 and G2 .....	8
2.4	G2 Debugger .....	9
2.5	FlexRay Controller .....	9
2.6	Experimental setup .....	10
2.7	Node development environment .....	11
2.8	Host development environment .....	11
2.9	Time-Triggered communication .....	12
2.10	CAN protocol .....	12
2.11	TTCAN protocol .....	12
2.12	FlexRay protocol .....	13
2.12.1	Overview .....	13
2.12.2	Scheduling .....	14
2.12.3	Time representation .....	15
2.12.4	Time Synchronizing .....	16
2.12.5	Wakeup and Startup .....	16
<b>3</b>	<b>Membership Protocol .....</b>	<b>19</b>
3.1.1	Overview .....	19
3.1.2	Definitions .....	19
3.1.3	Protocol Overview .....	20
3.1.4	Fault-model .....	21
3.1.5	Failure cases .....	22
3.1.6	Decision function .....	23
3.1.7	Basic Membership algorithm .....	24
3.1.8	Reintegration algorithm .....	24
3.1.9	Dynamic message scheduling algorithm .....	25
<b>4</b>	<b>Work Performed .....</b>	<b>27</b>
4.1	Host Application .....	27
4.1.1	Overview .....	27
4.1.2	Communication with the G2 debugger .....	28
4.1.3	Uploading files .....	29
4.1.4	Performing tests .....	30
4.2	Node Application .....	31
4.2.1	Overview .....	31

4.3	Simulated cluster environment .....	31
4.4	Membership algorithm implementation .....	32
4.4.1	Overview .....	32
4.4.2	Membership start-up protocol .....	32
4.4.3	Fault injection .....	33
4.5	Membership Testing Parameters .....	34
4.6	Test cases .....	34
4.7	Simulated Environment .....	35
4.8	Problems encountered .....	36
4.8.1	Hardware problems .....	36
4.8.2	Software problems .....	36
4.7.3	FlexRay related problems .....	38
<b>5</b>	<b>Conclusion .....</b>	<b>41</b>
5.1	Correctness test results .....	41
5.2	Overhead estimation .....	44
5.3	Extending the existing membership algorithm .....	44
5.4	Extending the host application .....	45
5.5	Conducting and analyzing the exhaustive membership tests .....	45
5.6	Conclusion .....	45
	<b>References .....</b>	<b>47</b>
	<b>Figure References .....</b>	<b>49</b>
	<b>Figure List .....</b>	<b>51</b>
	<b>Table List .....</b>	<b>53</b>
	<b>Index .....</b>	<b>55</b>
	<b>Appendix A .....</b>	<b>2</b>
	<b>Basic membership protocol with reintegration .....</b>	<b>2</b>
A.1	Introduction .....	2
A.2	Pseudo code .....	2
	<b>Appendix B .....</b>	<b>4</b>
	<b>The Host – Node Communication Protocol .....</b>	<b>4</b>
B.1	Introduction .....	4
B.2	Pre membership communication .....	4
B.3	Post membership communication .....	4
	<b>Appendix C .....</b>	<b>6</b>
	<b>The Test Specification File Format .....</b>	<b>6</b>
C.1	Introduction .....	6
C.2	File Format .....	6
	<b>Appendix D .....</b>	<b>8</b>
	<b>Node Application API documentation .....</b>	<b>8</b>
D.1	Introduction .....	8
D.2	src/constraints.c File Reference .....	8
D.3	src/faultinjection.c File Reference .....	8
D.4	src/funcs.c File Reference .....	9
D.5	src/hostcommunication.c File Reference .....	10
D.6	src/math.c File Reference .....	11
D.7	src/membership.c File Reference .....	11
D.8	src/node1-4Application.c File Reference .....	15
D.9	src/output.c File Reference .....	16
D.10	src/set.c File Reference .....	17
D.11	src/timer.c File Reference .....	19

<b>Appendix E .....</b>	<b>21</b>
<b>Host Application API documentation .....</b>	<b>21</b>
E.1 Introduction.....	21
E.2 src/CommSettings.c File Reference .....	21
E.3 src/CommSettingsUI.c File Reference .....	22
E.4 src/CommWindowUI.c File Reference .....	23
E.5 src/CommWorker.c File Reference.....	23
E.6 src/Converter.c File Reference.....	24
E.7 src/HostApplicationUI.c File Reference.....	25
E.8 src/HostMessage.c File Reference .....	26
E.9 src/MainTestSpecification.c File Reference.....	27
E.10 src/Message.c File Reference.....	27
E.11 src/MessageQueue.c File Reference.....	28
E.12 src/NodeCommunication.c File Reference .....	28
E.13 src/NodeCommunicationHeader.c File Reference .....	29
E.14 src/NodeMessage.c File Reference .....	30
E.15 src/TerminalMessage.c File Reference.....	30
E.16 src/TestCommunicator.c File Reference .....	31
E.17 src/TestData.c File Reference .....	32
E.18 src/TestErrorSpecification.c File Reference .....	33
E.19 src/TestFileParser.c File Reference .....	34
E.20 src/TestFileSettings.c File Reference .....	35
E.21 src/TestFileUI.c File Reference .....	35
E.22 src/TestFileWorker.c File Reference.....	36
E.23 src/TestSpecification.c File Reference .....	36
E.24 src/UploadFileUI.c File Reference.....	37
<b>Appendix F .....</b>	<b>40</b>
<b>Membership Simulator API documentation .....</b>	<b>40</b>
F.1 Introduction.....	40
F.2 src/errorSimulation.c File Reference .....	40
F.3 src/main.c File Reference.....	41
F.4 src/membership.c File Reference .....	41



## Glossary

<b>ABS</b>	Anti-Lock Braking System
<b>BG</b>	Bus Guardian, used by the FlexRay CC to control access to the FlexRay bus
<b>CAN</b>	Controller Area Network
<b>CEDES</b>	Cost Effective Dependable Computer Systems, a Swedish research project with colorations in both government and industry
<b>CLI</b>	Command Line Interface
<b>Cluster</b>	a group of loosely coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer.
<b>CS</b>	Chip Select
<b>CSP</b>	Clock Synchronization Value
<b>ECU</b>	Electronic control unit
<b>FD</b>	Failure Detection, the first membership protocol phase
<b>FlexRay</b>	a automotive network communications protocol under development by the FlexRay Consortium
<b>FPU</b>	Floating-Point Unit
<b>FTM</b>	Fault tolerant midpoint algorithm, used by the FlexRay CC to calculate each node's derivation from the global time base
<b>G2DBG</b>	G2 Debugger
<b>GAST</b>	General Application development boards for Safety critical Time triggered systems
<b>GEE</b>	GAST Eclipse Environment, development environment for GAST components
<b>ILF</b>	Incoming Link Failure
<b>IO</b>	Input/Output
<b>JDK</b>	Java SE Developing Kit
<b>LCU</b>	Local Time Unit
<b>MC</b>	Membership Communication, the second membership protocol phase

<b>MCU</b>	Micro Controller Unit, is a computer-on-a-chip, a type of microprocessor emphasizing self-sufficiency and cost-effectiveness
<b>MH</b>	Membership Handling, the third and last membership protocol phase
<b>mingw</b>	Minimalist GNU for Windows
<b>MTG</b>	Macrotick Generation Process
<b>NF</b>	Node Failure
<b>NIT</b>	Network Idle Time, used by the FlexRay CC for time synchronization
<b>ODEEP</b>	Open Dependable Electrical and Electronics Platform
<b>OLF</b>	Outgoing Link Failure
<b>PPC</b>	Power Pc, a RISC microprocessor architecture
<b>RTCOM</b>	Real-Time Communication
<b>SOH</b>	Start of Header
<b>SP</b>	Swedish national Testing and research institute
<b>SPI</b>	Serial Peripheral Interface
<b>TMR</b>	Triple Modular Redundancy, redundancy using three systems and voting to determine the result
<b>TTCAN</b>	Time Triggered CAN
<b>TTP</b>	Timed Token Protocol
<b>X-by-wire</b>	replacing mechanical and hydraulic systems with electrical systems

# 1 Introduction

The use of electronic and computerized systems has in last few decades virtually revolutionized the automotive industry across the world. Since the advent of inexpensive electronic circuits in the 1970s the use of electronics in order to simplify a large number of tasks has increased exponentially until the present day. Where until the early 1970s automobiles consisted almost exclusively of mechanical and hydraulic systems there are today a host of diverse tasks ranging from electronic ignition to ABS (Anti-Lock Braking System) to seat heating systems being performed instead by ECUs (Electronic Control Units). It is not uncommon for an automobile of a recent model to contain 70 ECUs, of which most are interconnected by a bus network.

The bus networks that are relaying information between the ECUs within the vehicles are also evolving at a fast pace. One of first bus networks to be used in automobiles was the CAN (Controller Area Network) network, which is an event-triggered system (much like Ethernet). The term “event-triggered” denotes that information packets are not guaranteed to be delivered, and that there is no system-wide synchronization system involved in the transmission of packets on this network. If collisions occur on such a bus network the packets will have to be retransmitted at a later time. Since more and more of the tasks that ECUs in automobiles are used for today are safety-critical, there has been a shift in the automotive industry towards so-called time-triggered bus networks.

On a time-triggered bus networks such as TTCAN (Time-Triggered CAN) or FlexRay (see section 5.4) data is sent on a periodic time slot schedule called a “network cycle”. Each ECU connected to the time-triggered bus may have one or more statically defined time-slots in which they exclusively are granted permission to transmit data. This property allows the network to be used successfully for safety-critical tasks, since packet delivery is always guaranteed given that the bus network operates correctly. Some time-triggered bus networks, such as FlexRay, also have support for a dynamically defined segment. This allows for a more flexible usage of the bus, since in the dynamic segment packets of varying size may be sent by any of the connected ECUs (although they may or may not be transmitted due to the packets’ priority ratings)

For some time now there has been a trend in the automotive industry to move towards what is called the x-by-wire architecture. The name x-by-wire originally derives from the aircraft industry’s fly-by-wire architecture; the concept consists of replacing various functions previously performed by hydraulic or mechanical systems by simple electromechanical actuators or sensors controlled by ECUs. The ECUs are then interconnected by a time-triggered bus network. The x-by-wire architecture makes it possible in future generations of automobiles to eliminate the whole steering column, the hydraulic braking system as well as all intermediate shafts. The benefits of using an x-by-wire architecture are potentially huge, and include lower production costs, easier

diagnostics, less cumbersome replacement of faulty parts as well as savings in weight (as shafts and hydraulic lines tend to be heavy).

## **1.1 Purpose**

When x-by-wire is used for systems that are safety-critical in nature (such as braking systems) it is vital that the system be as fault-tolerant as the original system or moreso. The fault-tolerance may be achieved through redundancies in sensors, bus signals or ECU hardware as well as in software. Additionally the concept of membership algorithms, which are the subject of this thesis, is introduced. Membership algorithms are used to monitor and possibly exclude malfunctioning ECUs from the bus network.

Membership algorithms provide a mechanism by which each node (ECU) keeps track of which of all other nodes are functioning correctly as well as which other nodes are not functioning correctly. The membership algorithm may be implemented either at the node level, or, in case of nodes capable of running more than one application at a time, the membership algorithms may be implemented on the application level. There are many variants of membership algorithms, but they all have in common the idea that all functioning nodes in a network must agree on an identical membership set, as well as the feature of membership voting. Each node needs to transmit some sort of message periodically to all other nodes in the network in order to indicate that the node itself is functioning correctly. Based on these messages each node makes up what is called an “opinion”. The nodes then exchange opinions and use majority voting to decide upon whether a given node in the network should be excluded or not.

Membership algorithms have been researched since the late 1970s, and the algorithms have been continually improved to make use of various technological innovations. Some of the currently most recent membership algorithms are presented in the article “A Configurable Membership Service for Active Safety Systems” by Carl Bergenhem and Johan Karlsson. These algorithms have never been tested in a real bus network, however, and it is the purpose of this thesis to implement and evaluate these algorithms in a modern time-triggered FlexRay bus network.

## **1.2 Scope**

In order to be able to develop useful applications of dependable electronic systems the CEDES project has developed hardware to be used solely for this purpose [1]. The hardware which was used in this thesis is a GAST-cluster consisting of passive backplanes onto which are attached a microcontroller board and a FlexRay communication controller board. GAST stands for General Application development boards for Safety critical Time-Triggered systems [3]. For this thesis project the Motorola MPC565-based G2 microcontroller board will be used. Communications is handled by a Freescale MFR4200-based FlexRay communications controller board. All these pairs of G2 cards and FlexRay controllers are then finally interconnected using a FlexRay bus cable.

### **1.3 Related work**

In some aspects this thesis project is the direct extension of several earlier theses projects developed within the CEDES project. In their thesis “A distributed FlexRay-based research platform”[9], Mattias Pettersson and Petter Uvesten created a functioning FlexRay-based GAST cluster consisting of one G2 card and two FlexRay cards. They also implemented a relatively simple membership algorithm that would run on their GAST-cluster. Moreover Andreas Sjöblom and Christian Archer, in their thesis “Membership implementation on time triggered architectures”, implemented similar membership algorithms as those used in this thesis project, although they used a different type of GAST cluster consisting of an earlier G1 microcontroller board and a TTCAN bus. Additionally, in another earlier thesis project, Jimmy Myhrman and Nicola Vorkapic developed FlexRay configuration software which simplifies the setup of FlexRay communication dramatically.

### **1.4 Objectives**

The purpose of this thesis is to develop an implementation of the membership algorithms based on [4] on a working GAST cluster. The membership algorithms will then be extensively tested in order to assure that they are functioning correctly as well as efficiently. To be able to fully test the membership algorithms artificial errors will have to be introduced into the communication in order to simulate real situations where errors arise. Moreover a test will be set up in order to measure how often errors arise spontaneously in a FlexRay cluster. It has to be taken into account, however, that the FlexRay cluster used in this thesis is located in an office environment which is fairly different in terms of electromagnetic disturbances which may cause the communication to fail. Finally the results of all these tests will be analysed and evaluated.



## 2 Background

The section also contains a more detailed description of fault tolerance and how it can be achieved in automotive applications. Moreover the hardware environment and software development environments are also presented in this section.

### 2.1 Fault tolerance

The main objective in the field of fault tolerant systems design is to be able to design a system capable of functioning with full or degraded capacity under the influence of errors. It is impossible to design a system that is 100% fail safe in practice, since one cannot prepare for all possible contingencies. The aim when designing a fault tolerant system is instead to make the system “safe enough”. For example, in x-by-wire systems such as brake-by-wire systems, where braking is achieved by electromechanical brakes connected to a bus network (as opposed to hydraulic braking systems), the purpose is instead to make the replacement x-by-wire system at least as safe as the system it is replacing.

If, in a fault tolerant system, a fault occurs during normal operation the system should enter a state of *graceful degradation*. This signifies that as much of the system as possible should be operational, and that the fault (in a distributed system) cannot propagate to nodes outside of the one in which the fault originated). If *graceful degradation* is impossible to obtain the system should instead shut down in a fail-safe manner. In a distributed system this means that a faulty node should be removed in such a way that it does not bring about any hazardous events among the remaining nodes.

The base for fault tolerance is redundancy of some sort. Redundancy may be implemented on many levels such as software redundancy, hardware redundancy, time (temporal) redundancy or information redundancy [2].

- **Information Redundancy**

In a system containing information redundancy supplementary information in addition to that needed by the system in absence of faults is used. Examples of information redundancy include parity bits or CRC checksums.

- **Software Redundancy**

Using software in addition to that required to implement a system in the absence of faults. Examples of this are membership algorithms.

- **Hardware Redundancy**

Using hardware in addition to what is required to implement a system in the absence of faults. Examples of this are airplane onboard computers.

- **Time Redundancy**

Using time in addition to what is required to implement a system in the absence of faults. Example of this are recalculating a result multiple number of times and then using majority voting to obtain the final result.

Many early fault tolerant systems duplicated hardware in such a way that if one module failed the rest of the system would not fail. One example is a TMR (Triple Modular Redundancy) system where three identical modules produce the same output. A voting procedure using majority voting was used to detect faulty units.

### **2.1.1 Reliability**

When talking about safety related systems different terms such as reliability and availability are examples of topics. A safety critical system such as a brake-by-wire system in a car must be reliable. [5] defines reliability as “The probability of a component, or system, functioning correctly over a given period of time under a given set of operating conditions. This attribute is very important to a safety critical system in contrast to non safety critical like a banking system or a time sharing computer where availability is a more critical attribute.

## **2.2 Dependability in the automotive industry**

### **2.2.1 X-by-wire technology**

The concept of *x-by-wire*, as was explained in the introduction, consists of replacing mechanical and hydraulic systems in automobiles with nodes of electromechanical actuators and sensors interconnected by a bus network. *Drive-by-wire* is one example of this. In a *drive-by-wire* system the accelerator pedal is not mechanically connected to the throttle, as in older generations of automobiles. The amount of acceleration that should be applied by the node controlling the throttle is determined by another node controlling an accelerator pedal position sensor. This second node periodically sends the acceleration data over the bus network.

An *x-by-wire* system must be close to 100 % reliable under normal operation (as stated above it is not possible to obtain 100% reliability in these kinds of systems). To improve reliability backup ECU units can be added in such a way that two or more units may control for example the throttle in the case that errors arise. Other safety-critical systems such as brake-by-wire and steer-by-wire usually contain hardware redundancy in this way as well. It is however not possible to introduce any great amount of hardware redundancy in automobiles, since this adds weight and especially cost to the automobile.

Another way to achieve fault tolerance in an *x-by-wire* system is to design a system where fault tolerance is maintained by introducing redundancy in the software. Membership algorithms may be used to assure that a system is reliable. Software redundancy is the topic of this thesis, where several membership algorithms are going to be implemented and analyzed using a FlexRay bus network. The FlexRay bus does not, unlike TTP (Timed Token Protocol), have a built-in membership implementation at the network protocol level, so when setting up a FlexRay network it is necessary to implement membership at a software level instead.

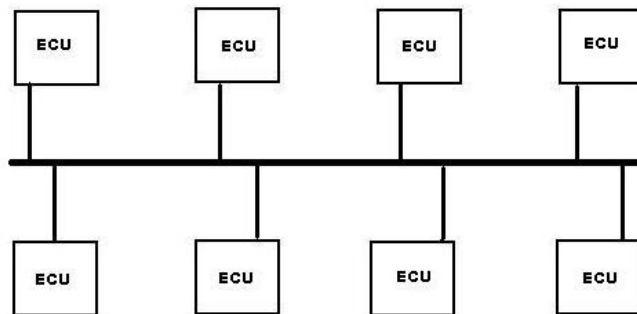


Figure 2.1 – Illustration of the topology of a bus network.

## 2.2.2 Membership

As was described in the introduction membership services is a necessary component in implementing fault tolerance in a distributed system of nodes. This thesis will see the implementation of the membership algorithms described in [17] in a GAST-cluster consisting of G2-cards and FlexRay controllers interconnected by a FlexRay bus cable. The hardware used in the cluster is described in the following section of the report.

### 2.3 G1 and G2

G1 and G2 are the two microcontroller boards produced by the GAST project. The G1 card is based on a Motorola HCS12 MCU (Micro Controller Unit). The G2 board is based on Motorola MPC 565 MCU PPC (Power PC) architecture. Both are equipped with versatile IO (Input/Output) that uses signal levels specific for automotive electronics. The cards are connected to the GAST RTCOM (Real-Time Communication) backplane via a 96-bit IO backplane. Each G2 card has 1 MB of flash memory as well as 32 kB of RAM. As has been stated in previous sections, for this thesis G2 cards are used exclusively.

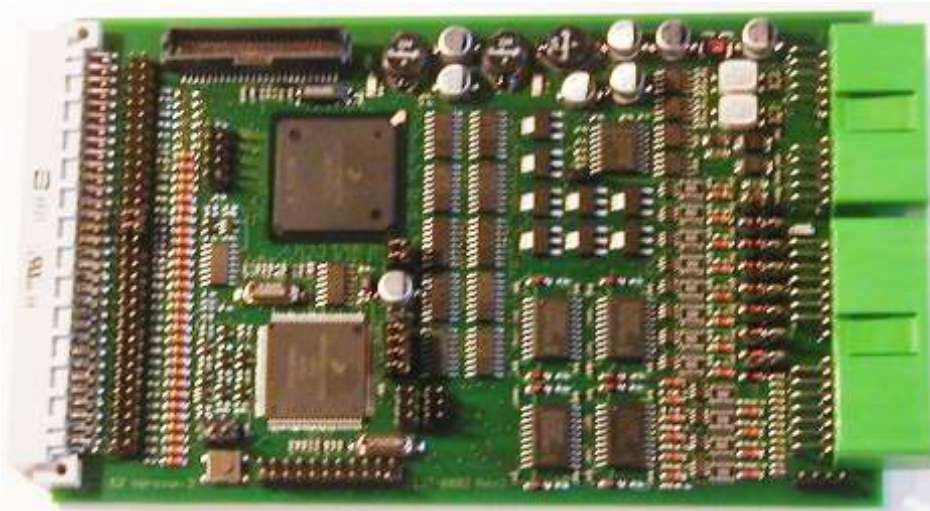


Figure 2.2 Overview of the G2 card (Revision 3)

The G2 card has two processors, the MCU565 and the HCS12. MPC565 is equipped with a FPU (floating-point unit). The MCU565 and the HCS12 can communicate over a SPI (Serial peripheral interface) or over a dedicated CAN bus. It is possible to set up the HCS12 processor as a bus guardian for the MCP565. The guardian can be used to detect errors and disable outputs upon error detection. For the purposes of this thesis, however, it was not necessary to utilize the HCS12 processor.

The address space in the G2 MPC is set up to facilitate space for the G2 debugger, user array as well as IO-area for FlexRay controller boards. This thesis uses an experimental setup where each G2 card is connected to one FlexRay card.

To be able to use the G2 card memory, both volatile and flash has been reserved for the user. It is possible to download and run programs in these sections. This thesis only uses the flash section for storing and running applications.

## 2.4 G2 Debugger

The G2 board used in this thesis is equipped with a debugger called G2DBG. The G2 Debugger has been developed by Roger Johansson at Chalmers Lindholmen, and it greatly simplifies the use of the G2 board in a FlexRay network since it greatly simplifies debugging of software as well as the transfer of software onto the G2 card. When the card is reset the debugger is loaded, and it then proceeds to initialize the hardware clock and set up a number of parameters such as ports. A debugger interface is then established through the serial port.

The debugger interface consists of a simple command prompt where the user may enter commands. During the course of the thesis the debugger proved to be especially useful in the debugging of certain difficult errors since it allowed viewing and modification of registers and memory addresses. For more information on the usage and features of the debugger, see [6].

## 2.5 FlexRay Controller

The FlexRay communication card is based on the Freescale MFR4200 communication controller (based on the Freescale MFR4200 FlexRay controller chip). The FlexRay card handles real-time communication between the GAST G2 microcontrollers. The MFR4200 card provides all the networking capabilities required by the FlexRay consortium [7], and its features include a maximum bit rate of 10 Mbits/s on each one of its two available communications channels.

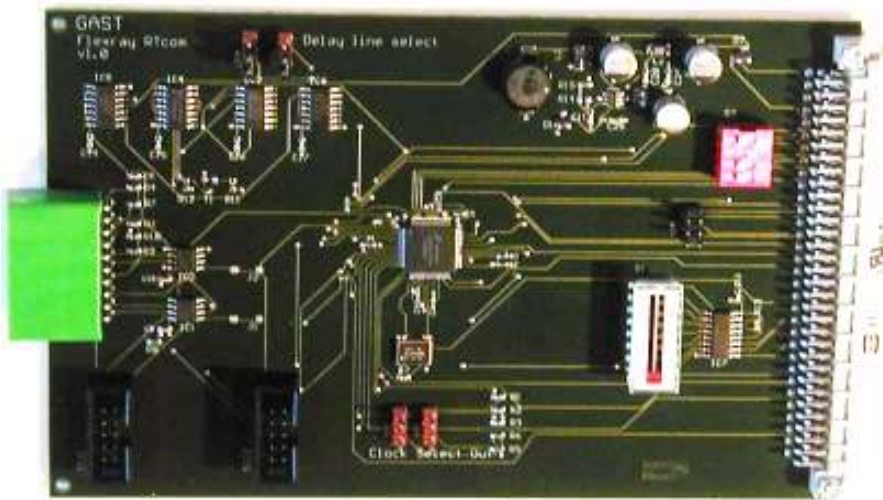


Figure 2.3. A picture of the FlexRay controller card v1.0

FlexRay communication controllers communicate via the FlexRay data buss attached to the green connector to the right on the picture. For more information on the MFR4200 controller, see [8].

## 2.6 Experimental setup

The experimental setup consists of a number of G2 processor cards as well as a number of MRF4200 FlexRay communication controllers. Each G2 card is connected to a FlexRay communication controller via a passive backplane bus. These pair wise combinations are called nodes. Each FlexRay card is connected to a bus cable, allowing all connected nodes to form a FlexRay network. This system is called a GAST cluster and is an experimental system of how a distributed communication network can look like i.e. a car.

For the purposes of our experiments each G2 card is also connected to a serial port on a PC (here called the host computer).

Registers on the FlexRay cards are mapped to memory addresses on the G2 cards so that by writing to specific memory addresses on the G2 card registers on the FlexRay cards can be modified.



Figure 2.4. *The experimental GAST cluster connected to the Infiniium Oscilloscope*

In the figure above one of the oscilloscope probes are connected to the bus connector on one of the FlexRay cards. This is done to demonstrate signal traffic on the FlexRay bus when FlexRay communication is in process.

## **2.7 Node development environment**

The open source development Eclipse was chosen for developing software for the GAST cluster nodes. This was due to a number of reasons.

In one of the theses preceding ours, “A distributed FlexRay-based research platform” [9] a substantial amount of work was put into finding an optimal development environment for developing GAST software. The authors of that thesis, Pettersson and Uvesten, decided to combine the open source environment Eclipse with a GNU GCC-based cross compiler, mingw (minimalist GNU for Windows), to obtain a full featured C development environment for Motorola MPC565 cross-compilation purposes.

This resulted in generation of a development package, GEE (GAST Eclipse Environment), which was put together in order to provide future GAST software developers with a powerful development environment.

Another strong reason to use Eclipse is the existence of a plug-in, developed by the aforesaid Pettersson and Uvesten, which provides the Eclipse environment with an integrated serial port terminal program. It is thus possible to both upload and execute code on the GAST target platform without having to use any external software. For more information regarding the use of Eclipse and the GEE package, see [9].

## **2.8 Host development environment**

For the development of host application software the Java IDE Netbeans was selected. This was decided upon based on the thesis authors’ previous positive experiences with Netbeans combined with the fact that Netbeans is freely distributed and readily available. Moreover since it was important for the host application to have a powerful GUI Netbeans proved an excellent choice as a development environment.

The Netbeans development package is freely available for download from Sun’s Java website (<http://java.sun.com>), and it can be downloaded as a bundle together with the latest JDK (Java SE Development Kit).

In the development of the host application the Java Communication package (javax.comm) was used for implementing communication over the serial port. Java Communication has been developed in order to enable programmers to easily write Java code for communication across serial ports. For more information on Java Communication, see [10].

## **2.9 Time-Triggered communication**

In a distributed network providing fair access to all entities participating in the network constitutes a problem. There exist two common solutions to this problem. Either what is called an *event-triggered* is used. In an *event-triggered* network a node wishing to transmit a message on the network will attempt to do it instantly. Usually event-triggered network architectures provide mechanisms for collision avoidance and collision detection, in order to prevent messages from being lost in case of two or more entities attempting to transmit at the same time. It has proven effective to use *event-triggered* network architectures in networks with light data loads, since they provide fast message transmission when the network suffers from a small number of collisions. Unlike time-triggered networks event-triggered networks do not guarantee that a given message will be sent within a given time interval. One example of an event-triggered network architecture is CAN.

In *time-triggered* networks the communication is performed in periods called cycles, which are further subdivided into timeslots. An entity's access to the network is strictly controlled by assigning it a timeslot with a predefined duration in which the entity itself exclusively is allowed to transmit information. The entity may not transmit information outside of its timeslots. This provides the advantage that it is possible to monitor whether a message has arrived in time or not at all, whereas in an event-driven architecture it is not possible to tell the difference between a lost message and a message being delayed by traffic congestion in the network. The main problem when building a time-triggered network is time synchronisation. All entities in the network must in some way agree on a global time base in order to avoid time slot discrepancies between the entities in the network.

## **2.10 CAN protocol**

The CAN was developed for the automotive industry in the early 1980s. It consists of a serial bus system that comprises the seven layer of the OSI/ISO reference model. CAN networks provide the user with four communication services; sending messages, data frame transmission, requesting messages and remote transmission request. All other services provided, such as error handling, are performed transparent to the user.

For more information about CAN, see [11].

## **2.11 TTCAN protocol**

The TTCAN protocol [12] is a higher level protocol compared to the CAN protocol. TTCAN is a protocol that provides both event-triggered communication, as in the CAN protocol, and time-triggered communication. In order to be able to send messages in a time-triggered network all nodes in the network must be synchronized. The synchronization in TTCAN is done by periodically sending a reference message with the time base that is seen by all nodes in the network.

## 2.12 FlexRay protocol

This section describes the *time-triggered* real time communication protocol FlexRay [13] which is used in this thesis for creating a communication network environment used for evaluating membership algorithms. The FlexRay protocol is an indispensable part of this thesis work, and it is therefore described in detail here.

### 2.12.1 Overview

FlexRay is a network protocol based on a time-triggered architecture. It can be used as a purely time-triggered network, a purely event-triggered network, or as a combination of both. FlexRay may be configured to make use of several different network topologies. The two basic topologies used in FlexRay are the bus topology and the star topology. These topologies can be combined to form hybrid topologies.

One example of a hybrid topology would be to connect a number of linear bus networks with an active star network to create a bigger hybrid-type network. Since FlexRay uses two communication channels the topologies can both be used in a single or dual channel configuration.

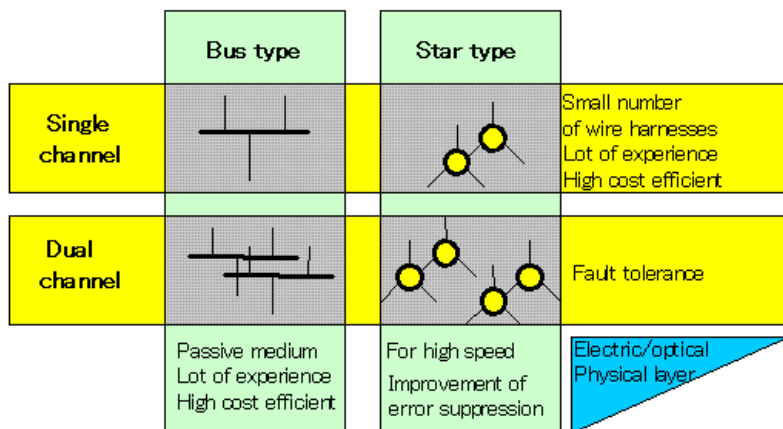


Figure 2.5 shows the two topologies utilised in the FlexRay protocol

Fault tolerance may be improved in the FlexRay protocol by using a dual channel configuration. Safety-critical systems may in that case use a second channel in order to transmit a redundant copy of the data on the first channel. FlexRay may be set up to use either only one channel or two channels simultaneously when sending data. This thesis uses a single channel FlexRay configuration.

The minimum number of nodes in a FlexRay network is 2, and the maximum number of connected nodes is 64.

## 2.12.2 Scheduling

A FlexRay communication cycle is divided into a static and a dynamic segment. In the static segment messages are sent using the time-triggered approach with a fixed number of timeslots with fixed duration. The dynamic segment, on the other hand, is used to send messages in a more flexible manner. Apart from the static and the dynamic segment a communication cycle also contains a *symbol window* segment as well as a *network idle time* segment (NIT).

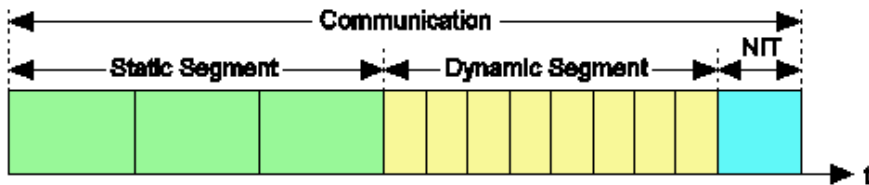


Figure 2.6 shows the different segments making up a communication cycle

A minimal communication cycle shall contain a static segment and NIT. Neither of these may be excluded from the cycle. The dynamic message and symbol window segments are optional parts of the communication cycle.

In the static segment time windows are provided for sending time-triggered messages. The static segment is further subdivided into a fixed number of timeslots of equal size. The static segment is used for sending communication as reliably as possible. Most of the data pertaining to safety-critical applications will be transmitted in this segment.

The dynamic segment is used to send event-triggered messages. The dynamic segment is divided into minislots that each has a unique frame id. A minislot counter is used to provide collision avoidance in the dynamic segment.

During a cycle the minislot counter value increases with a short time interval between ticks. A dynamic message is then sent when its frame id is equal to the current value of the minislot counter.

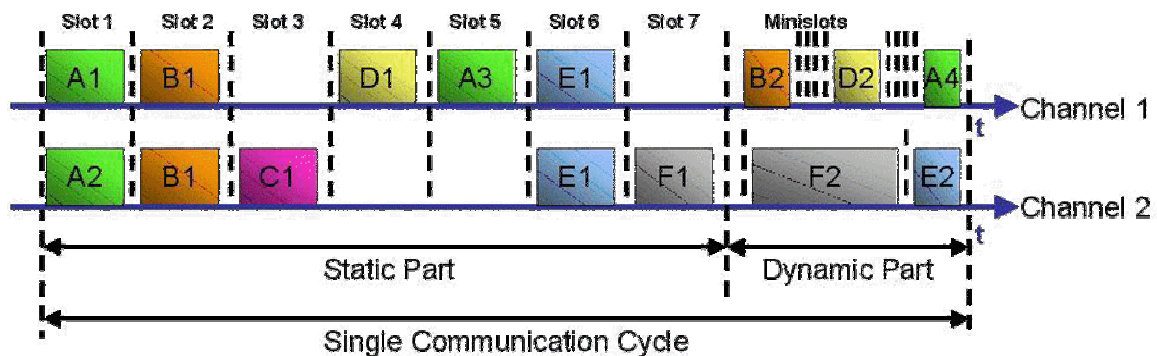


Figure 2.7 shows messages being sent in the static and the dynamic segment

If nothing is sent in the dynamic segment of a cycle the minislots of that cycle all have the same length. When a message is sent in the dynamic segment the duration of the minislot expands until the end of the message transmission. For example, in figure 2.7 the message F2 is being sent on channel two during most of the dynamic segment of that cycle. The message E2 can not be transmitted until the transmission of F2 has completed.

In this way collision avoidance is implemented in the dynamic segment. Two dynamic messages can never attempt to transmit at the same time on the same channel. This is the case because each message has a unique Frame ID. During the transmission of a message the value of the minislot counter is unchanged.

Message priority is maintained in both the static and the dynamic segment by using the Frame ID of the message. Messages with lower Frame ID have priority over messages with higher Frame IDs.

As was stated before, beside the static and the dynamic segments a FlexRay communication cycle consists of a NIT segment and a symbol window segment. The NIT is used for clock synchronization purposes while the symbol window is used for transmission of certain symbols used e.g. in the wakeup process.

### 2.12.3 Time representation

Time in a FlexRay node is represented by cycles, macroticks and microticks. A cycle is composed of an integer number of macroticks. A macrotick is composed of an integer number of microticks.

A microtick is a multiple of the clock tick of an oscillator in the communication controller. Since the tick length of oscillators in different communication controllers vary slightly, this has the signification that the length of a microtick is controller specific i.e. it may vary between different communication controllers.

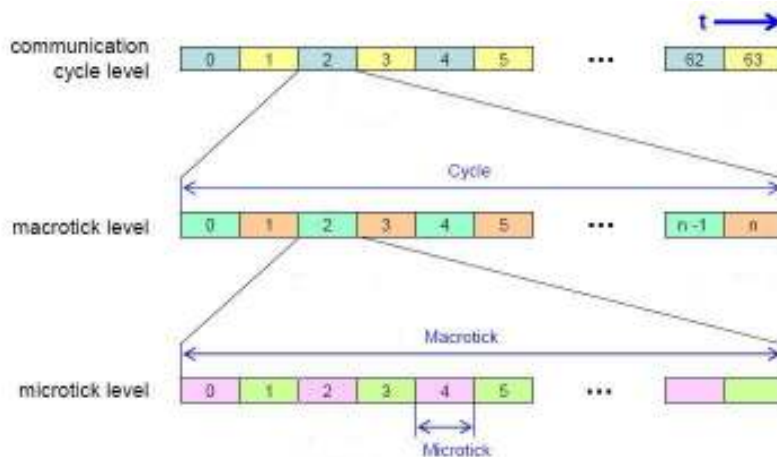


Figure 2.8 Time representation in the FlexRay communication controller

The FlexRay controller measures time on a local as well as global time basis. The local time in a node is defined as the time determined by using the node's internal clock. The global time of a cluster is defined as the commonly agreed upon view of the passage of time inside a cluster. A global time is periodically determined through the exchange of messages, and the local time in each node is subsequently adjusted to match the global time.

#### **2.12.4 Time Synchronizing**

It is essential that the local time in the various nodes in a FlexRay network remains synchronized. Therefore the difference between local time in a node and the global time must not differ by any large amount. Any node straying too much from the local time too long is disconnected from the network. The time synchronization is done by the sending of a reference message in the static segment. The message is sent by the preselected synchronization nodes (nodes that have the sync bit in the header set).

#### **2.12.5 Wakeup and Startup**

In the FlexRay wakeup procedure a communication controller initiates a mechanism that wakes up an entire channel of "sleeping" nodes by sending a specific signal pattern over the network indicating *channel wakeup*.

Each communication controller in the FlexRay network has the ability to wakeup the communicating channel by transmitting a wakeup pattern. A wakeup pattern is a predefined signal recognized by all communication controllers as an attempt to wakeup the specified channel. Upon reception of a wakeup pattern the controller enters the startup phase. For safety reasons a single communication controller can only wake up one channel at the time. This hinders a faulty node from disturbing the communication on both channels simultaneously.

FlexRay startup has to be executed before any normal communication can take place. The FlexRay startup procedure is called a coldstart. A coldstart is executed simultaneously by a number of the nodes in the network configured as coldstart nodes. In a network consisting of three nodes or more at least three nodes in the network have to be configured as coldstart nodes for the network to be able to startup. If the cluster consists of only two nodes it is necessary for both nodes to be coldstart nodes. The coldstart node that actively initiates the startup is called a *leading coldstart node*. The remaining coldstart nodes are called *integrating coldstart nodes*.

The startup procedure is done for cluster synchronization purposes. Each coldstart node enters a number of internal states sequentially while executing a coldstart attempt. These states differ for the leading coldstart node compared to the integrating coldstart nodes. During each coldstart attempt the coldstart nodes send a number of startup frames in order to synchronize themselves to the other nodes as well as to calculate a global time base. Each non-coldstart node has to integrate to the network by receiving at least two

startup frames from two distinct coldstart nodes. When a nodes has fully synchronized with the other starting nodes it enters the state of normal FlexRay communication.

For more information on wakeup and startup in the FlexRay protocol, and on the FlexRay protocol in general, see [13].



## 3 Membership Protocol

In this section a detailed description is given of the membership protocol that has been implemented and tested in this thesis. The protocol consists of the three algorithms that are described below.

### 3.1.1 Overview

The following membership protocol was proposed by Carl Bergenheim and Johan Karlsson in their article “A Configurable Membership Service for Active Safety Systems” [4]. In this thesis parts of the protocol described in the article are implemented and tested. The membership protocol is a membership service based on the idea that each entity in the network creates its own view of the status of all other entities in the network, i.e. which entities are operational and which are not. Based on all these individual views a common view is agreed upon.

A view is achieved by conducting what is called majority voting based on a variety of messages sent by the entities over the network. Since FlexRay is a time-triggered protocol each entity in the network always has the ability to decide whether any particular message is missing (because of a failure in the entity itself, in the network, or in the sending entity). This property makes it possible to perform a vote to decide whether a node is operational or not.

### 3.1.2 Definitions

A number of concepts are introduced in this section, and so for clarity these concepts will be defined here and then used without further explanation.

In order to understand the fault-injection mechanisms in this section it is necessary to define the concepts failure and fault. One definition given by Storey [5] defines a fault as a defect within a system, and that a failure (system failure) occurs when a system fails to perform its required function.

This means that when introducing a fault into a system, the system may fail to perform a certain required function, and therefore a failure occurs.

The cluster consists of  $N$  nodes that each hosts  $P$  processes. Each process may be part of the membership but does not have to be.

The membership is a set of processes in such a way that if a process falls out of the membership it is removed from the membership set, and when a process reintegrates to the membership it is added to the membership set.

A network consists of at least three nodes. When the membership starts all nodes and all processes are considered members.

This thesis implements the version of the protocol that does not support nodes hosting multiple processes. In the implemented version each node itself is considered a process so the membership is conducted on node level rather than process level. This version utilizes set theory where each member is part of a membership set, and exclusion and inclusion of the membership is a matter of set manipulation. This protocol version was described in [14] by Rosset, Souto and Vasques. The code generated in this thesis can easily be extended to support the algorithm purposed by Bergenhem and Karlsson in [4].

No message redundancy is used, e.g. each message is only sent on one channel.

### 3.1.3 Protocol Overview

The protocol operation is divided into three phases that take place during different parts of a FlexRay cycle. The protocol described, proposed by [4] support nodes with multiple processes.

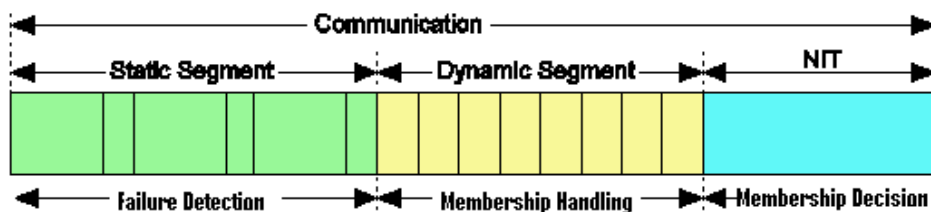


Figure 3.1. A FlexRay cycle with the membership protocol

The FD (Failure Detection) phase takes place during the static segment in the cycle. Each node in the network has a number of processes running. Each of these processes has a static slot assigned to them. They use their slot to send application data but also to send what is called a heartbeat message. The heartbeat message contains the current status of the process. Since incoming and outgoing links on the FlexRay cards are separated, each node will theoretically receive messages from their own processes.

The protocol assumes that each node sends in its allocated static slot. All nodes read the messages sent over the network and note the reception of heartbeat messages in an opinion vector. If a node has received a heartbeat message it adds this process as alive into the opinion vector, and if no heartbeat was received the process is added as dead.

The MC (Membership Communication) phase takes place during the dynamic segment of the FlexRay cycle. All nodes in the network broadcast their opinion vector and receive the opinion vectors sent by all nodes, including those sent by themselves. The dynamic segment has to be long enough to allow for all nodes to send their opinions. Each node assembles an opinion matrix consisting of the received opinions. Table 6.2 is one example of an opinion matrix.

During the NIT the MD (Membership Decision) phase takes place. The FlexRay cycle must be configured with a NIT long enough to accommodate the MD phase. The majority voting process that decides upon a global membership takes place in this phase. Each

column in the opinion matrix is sent to the decision function along with the process id and the total number of opinion vectors received from the other nodes in the network.

Each result from the decision function is added to a result vector. The result is then checked against the opinions to assure that the protocol constraints are fulfilled. A number of error cases can set a process to *not member*, halt it, or halt the node. The functionality of the three phases differs a bit depending on the version of the algorithm.

To be able to use the algorithm described in a GAST cluster, some sort of reintegration of nodes that has been excluded from the membership is needed. This is a part of the extended basic membership algorithm.

The basic algorithm requires each node to send its membership information, even when no change has occurred. This creates a lot of network overhead. In order to reduce this traffic an algorithm supporting dynamic message scheduling has been introduced. In the dynamic reintegration algorithm version the MC and the MD phases are only performed when a change has occurred in the membership, i.e. when a failure or reintegration has occurred.

### 3.1.4 Fault-model

This section describes the fault model proposed by Bergenhem and Karlsson in [4]. The membership protocol is designed to handle the faults described in the model.

Failed item:		Type of faults that the item may introduce:	
		Fault set A:	Fault set B:
1.	Node(consistent) (i.e. dead node)	<sup>1</sup> Fail-silence (of all hosted processes)	
2.	Node incoming link (inconsistent)	<sup>2</sup> Fail-XXX (of all incoming msgs.)	
3.	Node outgoing link (consistent)	<sup>3</sup> Fail-silence (of all outgoing msgs.)	<sup>7</sup> CRC-error (of all/some msgs.)
4.	Process (consistent)	<sup>4</sup> Fail-silence (of all msgs.)	<sup>8</sup> Fail-reporting e.g. cause of fault

Table 3.1. *The fault model containing two set of faults*

All faults that can occur in a system are assumed to originate from a node failure, a process failure or from a failure in a node's incoming or outgoing links.

The presented fault model shown in table 6.1 covers two fault sets, A and B. The algorithm implemented in this thesis covers the first fault set. Faults are divided further into consistent and inconsistent faults. An inconsistent fault like the incoming link failure is a fault that can introduce inconsistency into the system.

### 3.1.5 Failure cases

This section describes the four different failure cases from the fault model introduced in section 6.1.4. All four failure cases are explained, although in our implementation of the membership algorithm node and process failures are handled as one single failure case.

A **process failure** can be detected by the absence of a heartbeat message. If the node fails after sending its heartbeat the failure will not be detected until the MD phase in the next cycle. This is true because the receiving node will pick up the absence of a heartbeat message in the following FD phase but the process and the hosting node will not be excluded from the membership until the end of the cycle where the MD is performed.

A **node failure** can be detected as the absence of a heartbeat message from the process or processes hosted by a node. It can also be detected by the absence of sending opinion in the MC phase. If a node fails before the MC phase it will be handled in the same way as process failures. In a worst case scenario a node failure will not be detected until the end of the following communication cycle.

An **OLF (Outgoing Link Failure)** occurs when the outgoing link of a node fails. This means that the node may still receive messages, but is unable to send any data onto the network. If the OLF occur before the nodes start sending out heartbeat messages the node will halt in the same cycle, and all non-faulty nodes will reach a consensus in this cycle.

If an OLF occur after the nodes have finished sending out heartbeat messages the node will halt in the following cycle. However, the non-faulty nodes will still reach a consensus in the current cycle.

An **ILF (Incoming Link Failure)** is the most difficult failure to detect. In order to find an ILF the voted result has to be compared to the opinion vector received from each node. Two cases of IFL are defined. One case occurs when the affected node only receives messages from itself. Another case is when the node does not receive any messages at all.

A node experiencing an ILF during the FD phase will consider non-faulty nodes as faulty because it did not receive any heartbeat messages from them. The node suffering from ILF will broadcast an opinion vector that differs from the voted result in the decision function, and as a consequence all non-faulty nodes will exclude all processes hosted by the faulty node. This is the best case scenario of the ILF where consensus is achieved, and the affected node will be halted in the current cycle.

A node experiencing an ILF during the MC-phase will remove non-faulty processes from its membership set because it did not receive any opinion vector from the other nodes. This node will not receive any heartbeat messages during the following cycle, and it will therefore end up with an opinion vector that differs from the voted one. As a consequence it will halt. This is the worst case scenario; consensus is achieved in the following cycle.

### 3.1.6 Decision function

The decision function is an important part of a membership protocol. Each node uses the decision function to obtain its global view of the system. The idea is that all nodes should have the same view of the system, thus a global view of the system is achieved. The decision function used in this protocol is “strict majority”, i.e. more than 50% of the nodes need to consider a process a member in order for it to be part of the membership.

Each node in the network has an opinion matrix that constitutes all nodes’ local view of the network. Each node also carries what is called an u value. The u value indicates how many nodes that each node has received opinions from. In the basic algorithm without reintegration each node only stores its own u value. In the other two algorithm versions, however, a vector is used to store the u values of all nodes in the network.

The decision function takes a process id, a column from the opinion matrix, and the u value as input. In the protocol versions supporting reintegration the u value is the largest value in the vector. The decision function forms a global view for one process at the time. The decision function needs a strict majority to be able to decide upon a status for the process.

The status of a process is defined by three different states. A process can either have the status member, non-member or undefined. The status is undefined when no majority can be found, i.e. an equal number of nodes believe the process should be dead or alive. Strict majority is given by the u value. At least *ceiling function* of  $(u/2)$  votes are needed in order to decide a process’ status.

	<b>PROCESS 1</b>	<b>PROCESS 2</b>	<b>PROCESS 3</b>	<b>PROCESS 4</b>
<b>NODE 1</b>	ALIVE	ALIVE	ALIVE	ALIVE
<b>NODE 2</b>	ALIVE	ALIVE	ALIVE	ALIVE
<b>NODE 3</b>	DEAD	DEAD	DEAD	DEAD
<b>NODE 4</b>	ALIVE	ALIVE	ALIVE	ALIVE

Table 3.2. *An opinion matrix used for majority voting in the decision function*

Table 3.3 is one example how an opinion matrix may look like. In this case there are four nodes, each with one process. Imagine that the  $u$  vector looks like this:

$u\_vector = \{4, 4, 4, 4\};$

This means that the decision function needs at least 2 votes for either alive or dead for each process in order to decide the process status. In the event of a tie, i.e. the column consists of an equal number of alive and dead votes, and  $u$  is an even number, the outcome will be not member.

In the case of the table above the resulting opinion vector will be  $\{M, M, M, M\}$ .

### 3.1.7 Basic Membership algorithm

The basic membership algorithm is the simplest algorithm in the membership protocol described in this thesis. This algorithm does not support any reintegration of nodes or processes. If a node or process halts or gets voted out of the membership it may not reintegrate again even though it may have become fully functioning. This is not feasible to use in a realistic environment like a GAST cluster. Nodes or processes can be fully functioning and still become excluded from the membership due to e.g. transient faults. The preferred way to handle this would be to reintegrate these nodes or processes into the membership again.

In the basic algorithm all three protocol phases are performed in every cycle. Because the membership information has to be sent in every cycle the algorithm contributes significantly to network overhead.

However, the general trend in communication networks is increasing communication speed. This is also true in distributed networks where FlexRay communication controllers have a transmission speed of 10 Mbits/s, which is a much higher transmission rate than any of the earlier time-triggered network systems. A network overhead comparison in distributed networks is done by Bergenheim and Karlsson in [4].

### 3.1.8 Reintegration algorithm

The basic algorithm described in the previous section does not support reintegration of excluded nodes or processes. In order to use a membership algorithm in a realistic way the algorithm must support reintegration of both nodes with transient failures as well as non-faulty nodes and processes.

A process or node may be excluded from the membership by being voted as not member or by being halted. If for example one node hosts four processes and one of these processes is halted it will not affect the other three. If one node is halted all processes hosted by this node are also halted.

If a node or process wants to reintegrate with the membership it sends a “membership join request” message instead of its heartbeat message in the FD phase. It then participates in the MC and MD phases as a member. All processes belonging to a joining node will also participate in the MC and MD phases as a member. Since the integrating node does not know anything about the current membership all nodes must send their group size  $u$  along with their opinion. This is done in order for the integrating node to be able to vote for a membership. When a node existing in the membership receives a membership join request it assumes that all processes belonging to this node are members.

### 3.1.9 Dynamic message scheduling algorithm

In the two previous membership algorithm versions the MC and MD phases were required in all communication cycles. This creates a lot of unnecessary network overhead. In the current version of the algorithm the MC and MD phases only take place when a node detects any change in the membership. A change in the membership is defined as a node or process failure or a node or process reintegration.

To utilize the new algorithm feature two new states are added. M-request indicates that a change has been detected and therefore the MD and MC phases should be performed. The genvector contains the generation id of the membership set at each node. Every time the MD phase is performed, i.e. a change has occurred, each node increments its generation counters in the genvector.

A node can detect a change in its local opinion vector during the MC phase. When a change is detected the m-request is set to true. All nodes will detect the m-request in the FD-phase and will consequently perform the other phases in the following two cycles. A node can locally detect any faults during the first cycle with all phases, but must get a global view to be able to determine if it experienced a receive fault or if the fault originated in a faulty node.

The generation vector is sent along with the opinion and upper bound,  $u$  in the MC phase. The upper bound is defined as the maximum value in the  $u$  vector. All non-faulty nodes in the network must have the same local generation id value as their membership vectors. To ensure this the local generation values are compared with the maximum value of the generation id received from the other nodes. In the case where any derivations is found the node will halt.

The scenario where a faulty node has the highest generation id value can occur. This will result in all non-faulty nodes with lower generation id value being halted. However this scenario required that the fault node computed a valid result from the decision function. Otherwise it can not increment its generation id value.

This should on the other hand never happen because one requirement for the protocol is that non-faulty nodes are in majority and therefore they will obtain the highest generation id value.

## **4 Work Performed**

In this section the applications developed during the course of the thesis are described. An application running on a PC connected to the GAST cluster using serial lines, from hereon called the “Host Application”, was developed in order to simplify testing and general usage of the GAST cluster. Furthermore the GAST cluster applications are described. Finally the testing procedures are presented together with a summary of the problems encountered.

### **4.1 Host Application**

In this section “Host Application”, implemented to aid testing procedures as well as the general development of applications for the GAST cluster. An overview of the host application is given together with a section on the automated testing in some more detail.

#### **4.1.1 Overview**

The Host Application has been implemented with the intention of assisting the thesis workers as well as future application programmers in using and testing a GAST cluster. The application has been built specifically for developing node applications on a GAST cluster consisting of MPC 565 G2 cards communicating over a FlexRay network.

The main purpose when developing the host application was to create a user-friendly environment for conducting automated testing on a membership algorithm running on a GAST cluster. The remaining functionality of the Host Application was developed in order to simplify other time-consuming tasks such as simultaneous upload of software into a GAST-cluster.

Since the host application was developed specifically to communicate with the G2DBG debugger running on a G2 microcontroller board the code would have to be rewritten if it is going to be used with another boot loader. The program also conducts automated fault injection testing on the implemented membership algorithms. The host application communicates with a node through a predefined communication protocol. For more information on the communication protocol, see Appendix B.

In the rest of the section the concept of a node will be used. A node is defined as a G2 microcontroller board connected to a FlexRay communication controller via a passive bus.

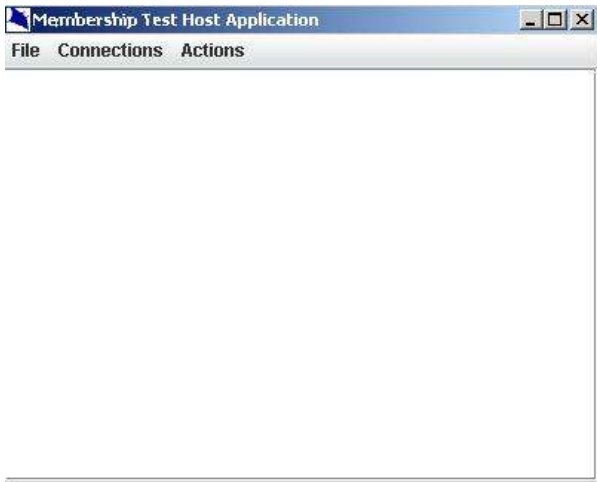


Figure 4.1 *The Host Application main window*

In the figure above the main window of the Host Application is displayed. The text area is used for displaying diagnostics and error messages regarding the application. Using the host application a user may connect to nine nodes simultaneously, upload node applications to them or load and perform fault injection tests on the nodes.

#### 4.1.2 Communication with the G2 debugger

The host application can be used to upload applications to all connected nodes. What makes this feature particularly appealing is because one of the most time-consuming tasks that were encountered while developing applications for the nodes was the process of uploading files to the nodes.

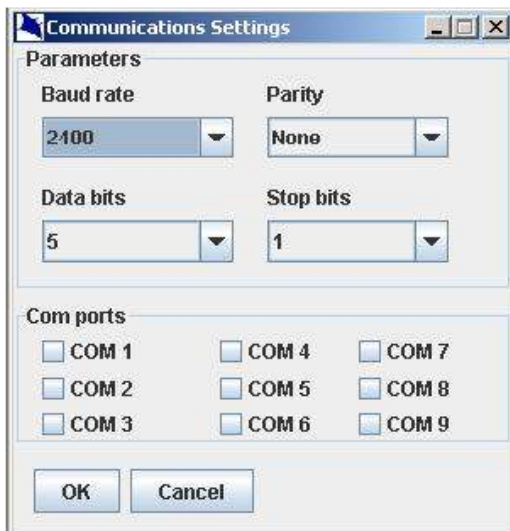


Figure 4.2 *Communications setting in the Host Application*

The host application may create simultaneous serial port connections to at most nine nodes. All communication ports are required to use the same communication parameters, e.g. baud rate, parity and data bits. This simplifies the use of the GAST cluster where manually handling connections to the individual G2 boards usually is a time-consuming and cumbersome task.

The user may at the same time communicate with each node via a terminal window.

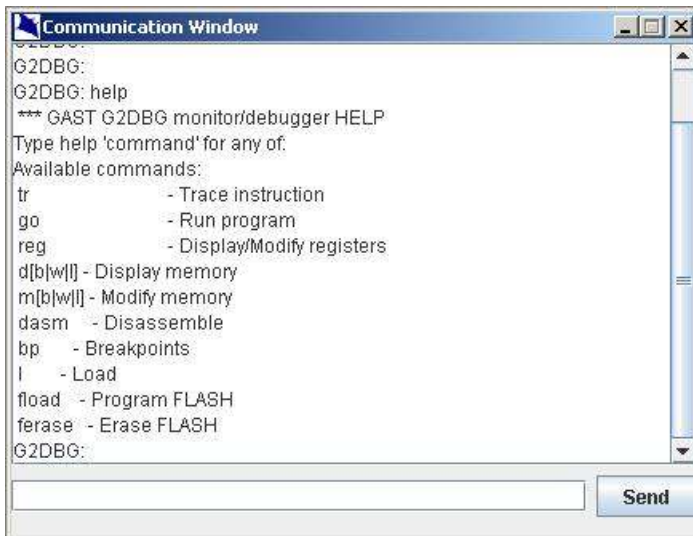


Figure 4.3 *Window used to communicate to a single G2 card*

The figure above shows the communication window that is used to communicate with a single node. Each node in the system is associated with a communication window during the course of the connection. The text area in this window is used for node-specific diagnostics and error messages. It is also used for communication with the G2 debugger. Commands can be sent to the debugger by using the text input area at the bottom of the window.

### 4.1.3 Uploading files

Uploading new applications to the nodes can be a time-consuming task. During the course of the node application development a large number of uploads have to be performed since small changes in the software usually need to be tested. One of the goals when developing the host application was to make the upload functionality as fast and convenient as possible.

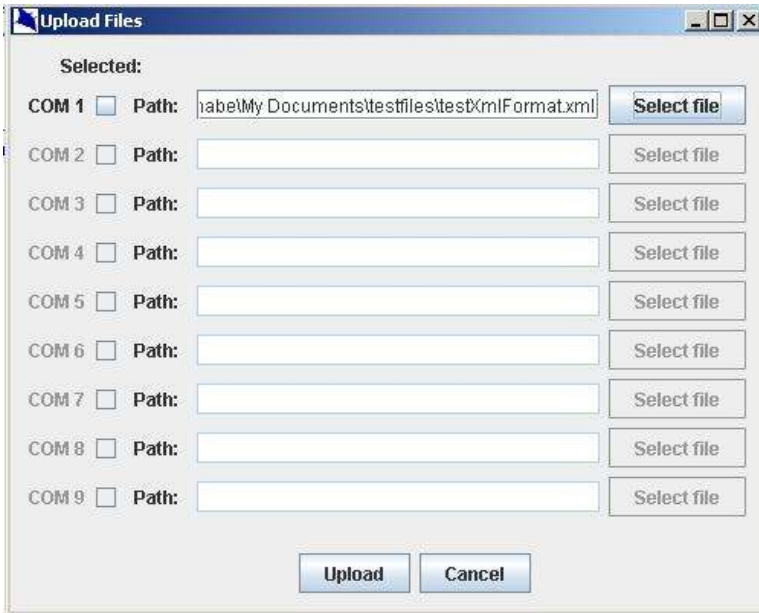


Figure 4.4 *The process of uploading software in the Host Application*

To make the upload functionality user-friendly files may only be uploaded to nodes with a live connection to the host computer. Each node may be assigned to receive a different file. The upload process is then conducted simultaneously to all nodes.

#### 4.1.4 Performing tests

To test the membership algorithm an automated testing procedure was developed where test cases are specified in an XML (Extensible Markup Language) file. The tests are then carried out on each of the nodes in the cluster. This provides us with a flexible way to specify and conduct membership tests on all nodes in the system as swiftly and easily as possible.

The test specification file format is specified in Appendix B.



Figure 4.5 *Selection of test files and output file directory*

The host application gives a user the possibility to choose which test specification file to load, and to specify in which directory the test results should be saved. No error check is performed on the XML parsing, so the test specification file has to be written exactly in

the format specified in Appendix C. When a test has been conducted a number of files with the format **Node<nr>output.csv** are created in the specified directory for output files.

Each test result file consists of a number of rows, where each row represents stored data from one cycle. The data stored in each file has the following format:

**<Nr of test data rows>**  
**<Current cycle>**, **<membership>**, **<joining nodes>**, **<opinion >**, **<halt status>**  
**<Current cycle>**, **<membership>**, **<joining nodes>**, **<opinion >**, **<halt status>**  
.  
.  
.

Data is stored in the test result files only when a change in any of the data members has occurred.

## **4.2 Node Application**

### **4.2.1 Overview**

In order to evaluate the membership algorithms in practice, an application written especially for the GAST cluster had to be implemented. The membership algorithm services form part of this application. Services that perform FlexRay communication also needs to be set up and implemented on each node.

Communication between the Node Application and the Host Application only takes place before and after the execution of the membership algorithm. The communication follows the serial communication protocol that is described in Appendix B.

For more extensive reading about the functionality of the node application, see the doxygen manual at Appendix D.

## **4.3 Simulated cluster environment**

To be able to easily test the membership algorithms a simulated cluster environment was developed. Since it is a difficult and cumbersome process to set up a functioning GAST cluster and coordinate the execution of the membership algorithm software on the cluster it was decided to test the membership algorithms under ideal conditions in a simulated cluster environment instead.

The simulated cluster environment is a relatively straightforward PC application written in C. The application runs each of the simulated nodes sequentially. The messages passed between nodes are thus stored as memory vectors by the application. Since the environment is simulating ideal cluster conditions all messages that are transmitted are received without errors in the same cycle. The simulated cluster environment proved

invaluable when developing the membership algorithm code, since it made testing and debugging a much easier procedure.

## **4.4 Membership algorithm implementation**

This section describes the implementation of the basic membership algorithm with reintegration, proposed in [4]. A short overview of the algorithm is presented followed by a description of the startup protocol along with a description of the fault injection process.

### **4.4.1 Overview**

The membership protocol, proposed by [4] and [14], formed the basis for implementing a membership algorithm. It was decided that the basic membership algorithm with reintegration was going to be implemented. There were two reasons for this. Firstly, the basic algorithm without reintegration is not complete for practical purposes, and it can not be used to run software or perform tests in a FlexRay network. The basic algorithm does not support reintegration of nodes, which is needed to run a membership algorithm on a FlexRay network where nodes by default integrate into the network during different points in time. This algorithm, which was proposed by [14], can be found in Appendix A.

### **4.4.2 Membership start-up protocol**

This section describes the start-up protocol added to the membership protocol in order to make the protocol fully functioning in practise.  $N$  is defined as the number of nodes in a cluster.

The membership protocol, proposed by [4], assumes that all nodes in the network are members in the beginning. Since each node in a FlexRay network enters normal operation mode (see section 5.4.2) and starts sending messages at different points in time this is not a plausible assumption to make. To be able to make the protocol a complete functioning protocol a start-up protocol was added to the membership protocol used. See section 9.3 for further details.

When using fault injection to test the membership algorithm all nodes in the network are required to start and stop executing the tests in the same FlexRay cycle. This fact assures that all nodes run the test in the same cycles. This proved to be a useful by-product of the start-up protocol.

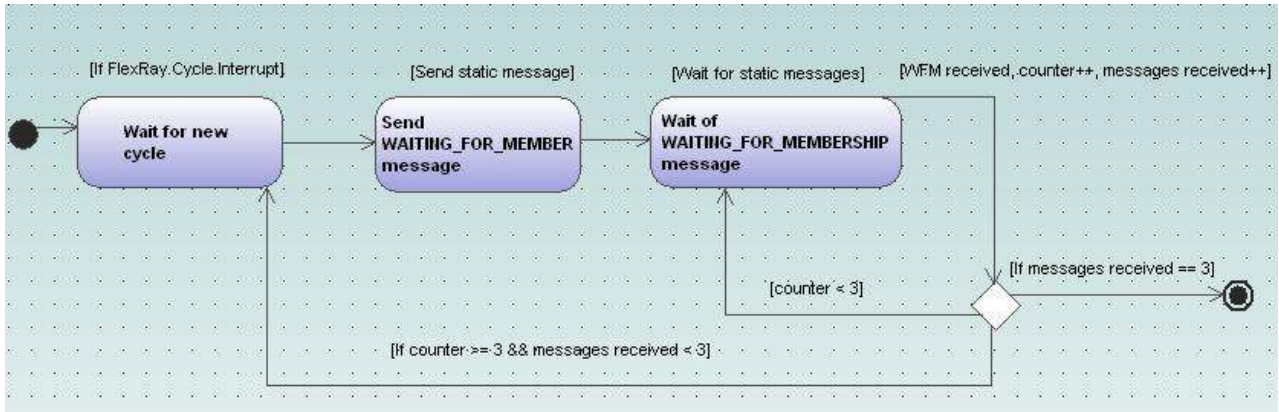


Figure 4.6 State transition diagram of the startup protocol

When a FlexRay node enters normal active operation state (see section 5.4.2) the membership start-up protocol is executed.

Figure 4.6 shows a state transition diagram for the start-up protocol. The start-up protocol sends out a message called a WAITING\_FOR\_MEMBERSHIP message, and then listens for other nodes sending WAITING\_FOR\_MEMBERSHIP messages. Each of the nodes listens until a cycle interrupt occurs, which is when the FlexRay protocol changes cycle. This procedure is repeated until messages have been received from all other nodes. If a node has received N-1 WAITING\_FOR\_MEMBERSHIP messages during one FlexRay cycle it assumes that all other nodes in the network ready and running. The node then leaves the start-up protocol and starts executing the membership code during the following FlexRay cycle. This assures that all nodes that are part of the network start executing their membership code in the same cycle.

#### 4.4.3 Fault injection

Fault injection is performed by simulating the fault set A discussed in section 6.1.4. Fault set B is easily simulated with only slight modifications to the membership algorithm. The membership algorithm implementation supports the failure cases presented in section 6.1.5.

A failure can be introduced at the beginning of the FD phase, the beginning of the MC phase or in the beginning of the MD phase. For more extensive reading about the fault injection, see section 8.2.

## **4.5 Membership Testing Parameters**

This section contains a discussion of the test parameters that were part of the original intended test cases for membership algorithm. As was described before it was not possible to perform tests on the actual GAST cluster, and instead a simulated test environment was used. The testing that took place in the simulated environment is also described in this section.

Since the membership algorithm in question is a recent development, and since it has never before (to the knowledge of the authors) been used or tested in a FlexRay network, the functioning of the algorithm has to be verified. It was intended that the testing performed on the GAST cluster be conducted by injecting various failure cases (described in section 6.1.5) into the software that is running the membership algorithms in the cluster. The testing was performed in this way to ensure that all functioning nodes exclude the faulty node and reach consensus according to the constraints defined in [4].

Performance testing on the membership algorithm was also originally in the scope of this thesis. This could be done by running the same test specification with different FlexRay configuration parameters. Examples of parameters that one could look at would be cycle length, slot length, NIT length and payload length. This testing would be conducted with a purpose to find an optimal configuration in terms of throughput.

It was possible to perform the correctness tests in the simulated environment. However, the simulated environment was not accurate enough for it to be meaningful to conduct performance tests as simulations. The results of the tests are presented in section 10.

## **4.6 Test cases**

In this section is shown how the failure cases described in section 6.1.5 were mapped to test cases used in the membership algorithm testing.

These tests were, as was previously stated, conducted in a simulated test environment. In order to simplify the environment a fault is assumed to be able to occur either before the FD phase, or in between the FD phase and the MC phase. A fault is also assumed to be able to occur in a node just after it has sent its heartbeat message, or just after it has sent its opinion. These tests can therefore not be assumed to entirely reflect the functionality of a GAST cluster, but rather only verify the membership algorithm during the circumstances given above.

In the presence of a particular fault this simulation will provide both a best and a worst case scenario for the algorithm. Recall that one of the requirements for the algorithm was that it must be able to detect and exclude a faulty node from the global membership view within two communication cycles. This was verified by injecting worst case scenario test cases.

ILF (Incoming Link Failure) is simulated differently in the system depending on where in a communication cycle it occurs. If an ILF occurs before the FD phase, this is simulated by ignoring received heartbeat messages from the other nodes. If the fault occurs after the FD phase, but before the node receives opinion vectors from the other nodes the fault is simulated by ignoring the received opinion vectors from the other nodes.

ILF is the hardest error to detect and worst case scenario for reaching consensus is two cycles. If a node experiences ILF in the MC phase consensus will be reached in the following cycle because the affected nodes will remove opinions of function nodes due to reception failure.

OLF (Outgoing Link Failure) is also handled differently depending on where in the cycle it is specified to occur. If the fault is injected before the FD phase then the fault is simulated in each node by omitting the heartbeat message in the assigned slot. This failure is easily detected by the other nodes and the membership implementation will halt the affected node and all functioning nodes will reach consensus at the end of the current cycle.

If the fault occurs in the beginning of the MC phase, the fault is simulated in each node by omitting the opinion vector during the dynamic phase of the communication cycle. In this case, consensus will be reached in the current cycle and the affected node will halt in the following cycle.

Node failure and process failure are not distinguished in this implementation of the membership algorithm. This simplification is made since here each node only contains one process. A node failure may also occur in the beginning of the FD or MD phases. The fault is simulated by omitting any messages in the communication cycle after the fault was injected, and ignoring any messages received after the fault was injected.

One possibility of future work is to extend the simulator to function more realistically like a GAST cluster. In order to guarantee a good simulation a substantial amount of testing has to be performed on the simulated environment itself.

## **4.7 Simulated Environment**

During the course of the thesis a simulated environment was created in order to perform limited tests on the membership algorithms. This simulated environment consisted of an exact copy of the membership code running on the node applications, but instead all “nodes” were running simultaneously as an application on a Windows PC. Instead of simulating all properties of a FlexRay network (which would undoubtedly be very difficult and require a large study to be undertaken of the physical properties of the FlexRay network and FlexRay microcontrollers), it was instead assumed that the network functioned correctly, and thus all messages were simply copied in a manner that a

message sent by one node would always be available to another node, unless it was specified otherwise.

The simulated environment supports the introduction of the ILF, OLF and NS faults in any cycle, either before the FD phase in that cycle or before the MC phase in that cycle. Future versions of the simulated environment should have support for introducing faults in a specified slot during the cycle, which would allow the application to test and evaluate a further range of faults.

## **4.8 Problems encountered**

In this section the problems encountered during the thesis are described. The problems are divided into three sections. Hardware problems describe problems directly related to the hardware, e.g. a faulty backplane or a G2 card behaving erratically. The software problems section describes all software-related problems encountered during the thesis, e.g. errors in the G2 debugger. The FlexRay-related problems section describes problems related to the FlexRay-related complications as well as problems related to the prototype hardware.

### **4.8.1 Hardware problems**

In the early development stages when first setting up a GAST cluster, a number of problems were encountered. The process of reading and writing to the registers mapped from the FlexRay card onto the G2 card gave strange results. When attempting a write sometimes no information was written into the register, and sometimes the adjacent register was written to instead. This fault originated in the strapping of the G2 card. When the correct jumpers were strapped this problem disappeared.

Another hardware-related problem encountered involved the relative physical position of the cards in respect to the backplane. It seems that the position of each card in the backplane affects the result, i.e. if communication between the G2 and FlexRay cards work or not. This cause of this fault is probably inconsistencies relating to the length of the backplane. In order to avoid this problem the backplane should be kept as short as possible. A simple yet efficient solution was to position the FlexRay/G2 pair next to each other on the backplane.

### **4.8.2 Software problems**

To get started using FlexRay a small GAST cluster was created, consisting of one G2 card and two FlexRay controllers on the same backplane. This was done in order to replicate an experiment performed by the earlier thesis workers (Pettersson & Uvesten), so that their FlexRay light drivers could be tested to make sure that communication was taking place between the FlexRay cards. The first attempt to set up communication between the two FlexRay cards failed, and no communication took place over the FlexRay bus.

In order to solve this problem the FlexRay Protocol Specification [13] was consulted, and it was expected that the error could be located somewhere during the wakeup or startup stages of the FlexRay protocol. The procedure that each node in the FlexRay network undertakes during an attempted startup of the network was investigated in detail. In this case none of the nodes could hear the transmission of the other node, so each of the nodes would eventually stop trying to startup or integrate into the network. It took much time and effort to locate the origin of this error. This was due to the fact that the FlexRay protocol is very complex, as was the FlexRay Protocol Specification [13]. Also, the fact that the authors had had little previous experience in debugging hardware contributed to make the work time-consuming.

To find the origin of the communication problem a systematic investigation of the functionality of the wakeup and startup stages of the FlexRay protocol was set about.

The configuration settings in each node were also verified against the protocol constraints specified in [13]. After careful investigation of the FlexRay protocol, and after some additional debugging, the error was assumed not to be related to the network configuration.

An Infiniium Oscilloscope (provided by Open Arena Lindholmen) was then utilized in order to examine whether any information was being over the FlexRay bus or not. The oscilloscope did not discover any of the specified signal levels on the FlexRay protocol, and this in turn led to the assumption that the error was purely hardware-related.

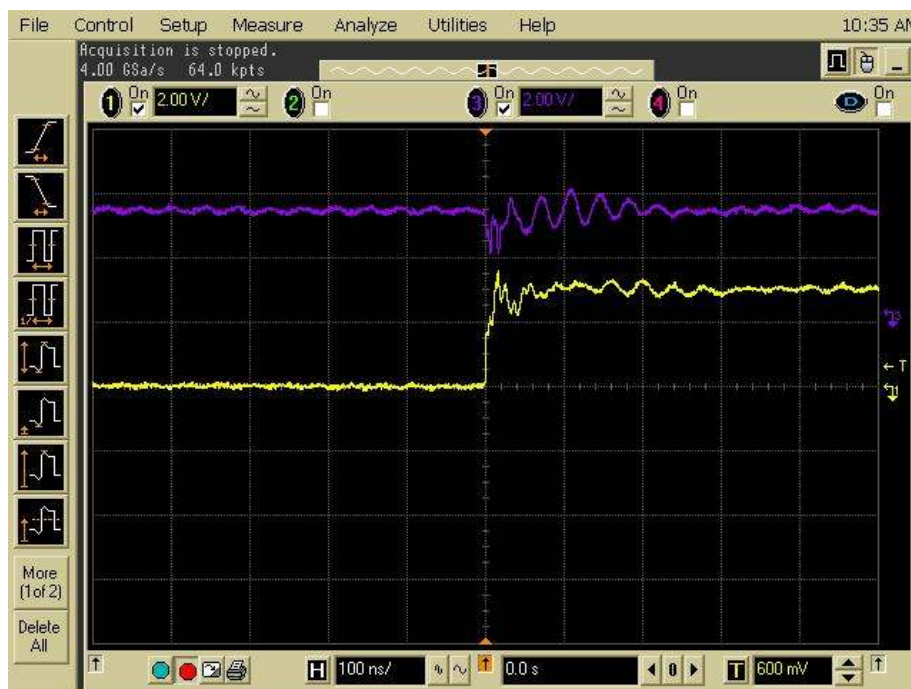


Figure 4.7 shows the disable signal coming from the HCS MCU

During this state the authors were given assistance by the supervisors, Roger Johansson and Carl Bergenhem.

After some hardware debugging using the oscilloscope it was discovered that a disable signal generated from the HCS12 MCU on the G2 board stopped the FlexRay signals from being transmitted onto the bus altogether.

The above figure from the oscilloscope shows two signals transmitted by two probes on the FlexRay controller. The mauve signal indicates that a disable signal is transmitted from the HCS12 MCU, and it is therefore disabling all communication being transmitted on the FlexRay bus. This effectively caused the whole FlexRay network to remain silent.

The reason why the HCS12 controller can disable signal transmission on the bus is so that it may act as a bus guardian. In our case no bus guardian was needed, and so Roger reprogrammed the G2 startup code so that signals were no longer being prevented.

### **4.7.3 FlexRay related problems**

In the sections below follows a description of the various problems that were encountered during the course of the thesis project. All of the problems except the last one were overcome.

During initial tests of the FlexRay network it was discovered that nodes could not receive messages from themselves. The outgoing and incoming links on each node are separated, which means that all nodes should be able to receive all messages transmitted on the FlexRay bus (including messages sent by themselves). After an exhaustive debugging procedure it was decided that problem should instead be ignored because it would only take a small alteration of the node application code to make sure that the problem would not affect the proposed membership algorithm. The changes meant that each node would use its internally stored heartbeat value and opinion vector instead of receiving a heartbeat and opinion vector from itself on the network.

Furthermore, during initial tests of the membership algorithm on the FlexRay network, a number of synchronisation and timing issues were discovered that severely affected the applications running the membership algorithm on the FlexRay network.

The membership protocol requires a network, with at least 3 nodes in order to run, see [4]. In the case of nodes integrating into a FlexRay network (see section 5.4.5) a FlexRay network can exist, consisting only of one node. This causes a problem with the membership, which would never start during the circumstances in the FlexRay startup sequence. This problem was overcome by implementing the simple startup protocol described in section 7.3.2.

While performing the initial tests of the implemented membership algorithm a problem was found that would greatly affect the outcome of the thesis project. The implemented membership algorithm showed problems with receiving static messages. It was shown

that for some of the messages sent in the static segment the messages would not be received until during the next cycle. Some of the messages, however, were correctly received in the cycle they were sent.

It was verified that the messages in question were actually deposited onto the send buffer of the FlexRay card in the correct cycle within the allowed time slot for sending messages onto the bus. According to the protocol specification the messages should thus have been received by all other nodes.

A FlexRay network consisting of four nodes was used for debugging this problem. All nodes contained identical node applications. Three of the nodes had problems receiving the static messages from the fourth node in the cluster. Each message sent could not be read by the receiving nodes until during the consecutive cycle. Therefore none of the three nodes could use the information as input to the membership algorithm, since the membership algorithm requires that the nodes be able to communicate with each other in every cycle. This was a major problem when dealing with the membership algorithms where the input of one cycle output completely depends on the output of the previous cycle.

According to the FlexRay protocol specification [13] this problem should not be able to arise. After the nodes in the FlexRay network reach their Normal Active internal state the messages deposited at the message buffers should be sent in the same FlexRay cycle with no delay whatsoever between the time of deposition and the time of transmission. The messages need however be deposited a certain amount of time before the start of the actual time slot available for the node in question. It was ascertained that the messages to be transmitted were in fact deposited well ahead of the deposition deadline.

If an error is detected in a FlexRay node then that node will change its internal state from Normal Active to one of the error handling states. It was determined that no such change of states occurred even though the error described above did occur (one cycle delay between time of transmission and time of reception).

In an attempt to solve this problem a slot counter interrupt was set up in the FlexRay communication controller and used to make sure that no node tried to read messages that had not yet been sent. The RX (Receive) buffers were also examined to make sure that all received data was read by the communication controller. It was necessary to reset the interrupt flag telling the communication controller that new data has been received in the buffer and preparing the buffer for the reception of new data. For more information about Message buffer handling in FlexRay, see [8] page 150.

None of these however proved to be a solution for the problem. Because of the present problem, the fact that this thesis was in its last phase and the indication from Roger Johansson that the problem could be hardware-oriented the decision was made to ignore the problem and continue to conduct tests on a simulated environment.

Using a simulated FlexRay network causes constraints on the test cases because the possibility to change parameters in the FlexRay configuration is removed. It does however provide us with the ability to verify the functionality of the protocol.

## 5 Conclusion

During the development of the cluster software, an inordinate number of problems arose (described in the previous section), of which some took a long time to solve. Due to lack of time it was not possible to overcome these problems within the time allotted to the thesis project and test the membership algorithms on a functioning GAST-cluster. The correctness tests of the algorithms were instead performed on the FlexRay simulated environment. Unfortunately it was not possible to perform the other planned tests.

### 5.1 Correctness test results

The correctness tests were performed using the GAST cluster simulator developed for these purposes. The GAST cluster simulator is described in section 4.3 of this report. In total 42 tests were conducted in order to verify that the protocol functioned correctly given that there were no externally introduced faults besides those introduced in the actual testing.

Since the GAST-cluster simulator may only introduce faults at two set points within the cycle (before the FD phase and between the FD and MC phases). It was initially believed that being able to introduce faults at other times during the cycle would not increase testing precision by any large amount. Further investigation, however, has revealed that this is not the case, and that several of the most difficult failure scenarios may not be modelled by the simulator in its current version. For example, in theory there should be some failure scenarios where the nodes don't come to a consensus until after two cycles. Those scenarios were not achieved here.

In the table below are listed the failure scenarios together with their correctness results as well as response time for the error. The response time of the protocol is defined as the number of cycles from the start of the cycle where the faults were introduced until the end of the cycle where all entities in the membership has a commonly agreed-upon view of which entities are members and which are not. Theoretically the maximum response time for the protocol should be two cycles for certain difficult cases. However, due to the limitations of the simulator as well as due to lack of time it was not possible to investigate the matter further. The tests were simulated on a cluster of five nodes.

As has been described before, ILF stands for Incoming Link Failure (the node in question fails to receive data), OLF stands for Outgoing Link Failure (the node in question fails to transmit data), and NS stands for node silent (there is no activity in the node). For all the faults tested in the simulator it was possible for the membership nodes to come to a consensus within one cycle.

<b><u>Type of test</u></b>	<b><u>Response Time (# cycles)</u></b>	<b><u>Correctness Achieved</u></b>
ILF before FD phase	1	Yes
OLF before FD phase	1	Yes
NS before FD phase	1	Yes
ILF before MC phase	1	Yes
OLF before MC phase	1	Yes
NS before MC phase	1	Yes
ILF before FD phase for node 1, ILF before FD phase for node 2	1	Yes
ILF before FD phase for node 1, ILF before MC phase for node 2	1	Yes
ILF before MC phase for node 1, ILF before FD phase for node 2	1	Yes
ILF before MC phase for node 1, ILF before MC phase for node 2	1	Yes
ILF before FD phase for node 1, OLF before FD phase for node 2	1	Yes
ILF before FD phase for node 1, OLF before MC phase for node 2	1	Yes
ILF before MC phase for node 1, OLF before FD phase for node 2	1	Yes
ILF before MC phase for node 1, OLF before MC phase for node 2	1	Yes
OLF before FD phase for node 1, OLF before FD phase for node 2	1	Yes
OLF before FD phase for node 1, OLF before MC phase for node 2	1	Yes
OLF before MC phase for node 1, OLF before FD phase for node 2	1	Yes
OLF before MC phase for node 1, OLF before MC phase for node 2	1	Yes
ILF before FD phase for node 1, NS before FD phase for node 2	1	Yes
ILF before FD phase for node 1, NS before MC phase for node 2	1	Yes
ILF before MC phase for node 1, NS before FD phase for node 2	1	Yes
ILF before MC phase for node 1, NS before MC phase for node 2	1	Yes

Table 5.1 *The results from the correctness tests of the simulator..*

<b><u>Type of test (continued)</u></b>	<b><u>Response Time (# cycles)</u></b>	<b><u>Correctness Achieved</u></b>
NS before FD phase for node 1, NS before FD phase for node 2	1	Yes
NS before FD phase for node 1, NS before MC phase for node 2	1	Yes
NS before MC phase for node 1, NS before FD phase for node 2	1	Yes
NS before MC phase for node 1, NS before MC phase for node 2	1	Yes
NS before FD phase for node 1, OLF before FD phase for node 2	1	Yes
NS before FD phase for node 1, OLF before MC phase for node 2	1	Yes
NS before MC phase for node 1, OLF before FD phase for node 2	1	Yes
NS before MC phase for node 1, OLF before MC phase for node 2	1	Yes
ILF before FD phase for node 1, ILF before FD phase for node 2 (in next cycle)	1	Yes
ILF before FD phase for node 1, ILF before MC phase for node 2 (in next cycle)	1	Yes
ILF before MC phase for node 1, ILF before FD phase for node 2 (in next cycle)	1	Yes
ILF before MC phase for node 1, ILF before MC phase for node 2 (in next cycle)	1	Yes
ILF before FD phase for node 1, OLF before FD phase for node 2 (in next cycle)	1	Yes
ILF before FD phase for node 1, OLF before MC phase for node 2 (in next cycle)	1	Yes
ILF before MC phase for node 1, OLF before FD phase for node 2 (in next cycle)	1	Yes
ILF before MC phase for node 1, OLF before MC phase for node 2 (in next cycle)	1	Yes
OLF before FD phase for node 1, OLF before FD phase for node 2 (in next cycle)	1	Yes
OLF before FD phase for node 1, OLF before MC phase for node 2 (in next cycle)	1	Yes
OLF before MC phase for node 1, OLF before FD phase for node 2 (in next cycle)	1	Yes
OLF before MC phase for node 1, OLF before MC phase for node 2 (in next cycle)	1	Yes

Table 5.2 *The results from the correctness tests of the simulator (continued).*

## **5.2 Overhead estimation**

In addition to the data payload an additional amount of overhead data in the form of FlexRay protocol data as well as membership information will be sent each cycle. Since the length of the FlexRay cycle may take values ranging from 154 microseconds to 16000 microseconds, and since the amount of overhead in a FlexRay cycle is constant for a fixed number of nodes irrespective of the cycle length, it is thus not possible to calculate the overhead in bytes/second. In other words, the amount of overhead data sent on the FlexRay bus per second decreases as the cycle length increases.

The total amount of membership overhead for the basic membership protocol with reintegration in a cluster with 5 nodes is ca 40 bytes. For a minimal FlexRay cycle of 342 microseconds the amount of overhead sent out per second would be 117kB/s. On the other extreme, with a FlexRay cycle of 16000 microseconds, the amount of overhead data per second would be 2.5kb/s. The amount of overhead in the dynamic algorithm is estimated to be much less (although this is difficult to analyze).

## **5.3 Extending the existing membership algorithm.**

The membership protocol, presented by Carl Bergenheim and Johan Karlsson in [4], consists of three different algorithms: the basic algorithm, the basic algorithm with reintegration support and the dynamic scheduling membership algorithm. In this thesis only the basic membership algorithm supporting reintegration was implemented.

One possible way of continuing the work done in this thesis is to implement the dynamic scheduling membership algorithm and test its performance in a running GAST-cluster. This can be done with only slight modifications to the existing membership code. The rest of the node application code and host application code can be used to run and test the dynamic scheduling membership algorithms without any other further modification.

The membership algorithm versions implemented are based on a node architecture where each node hosts only one process. This is a simplification of the proposed protocol in [4], and it can be compared to the protocol proposed in [14] where each entity in the algorithm hosts no processes other than itself.

One other possible extension is to implement an algorithm where each node supports multiple processes. This can also be achieved with only small modifications to the existing membership code. Each process will have one static slot assigned to it where the heartbeat message is sent. Each node will send its opinion vector in the dynamic vector as is done in the basic protocol, because the voting process is handled on the node level instead of on the process level. To achieve this, the code needs to be modified to support storing and handling multiple processes along with a reference to whichever the node they belong.

Further extension of the thesis work could be to implement functionality in the nodes for taking over processes running in faulty nodes. Imagine a network of nodes where a

number of processes containing different functionality is running in each node. If a node or a process within a node is excluded from the membership, another node could take over the responsibilities that the faulty node or process had. Implementing this type of functionality could be one way of showing the strengths of using membership algorithms in a safety-critical system.

For more information of extending the algorithm implementation to the algorithm proposed by Bergenhem and Karlsson, see [4] or section 6.

#### ***5.4 Extending the host application***

The host application supports all functionality needed to run and test a membership protocol in a GAST cluster except for analysing the received test data. The test data from a coordinated membership test is however saved as comma-separated text files, which are easily read, manipulated and analyzed using a numerical computing package such as Matlab. It would thus be necessary to write Matlab programs to analyze the output text files.

#### ***5.5 Conducting and analyzing the exhaustive membership tests***

Exhaustive testing of the implemented membership algorithm with respect to performance and functionality was omitted from this thesis, see section 9.3. In the work conducted for this thesis tests was conducted with a simulated test environment in order to verify the correctness of the membership algorithm. More exhaustive testing is necessary to accurately verify the correctness of the membership algorithm implementation in an actual running GAST cluster. It would moreover be necessary to perform some sort of performance testing looking at parameters such as network overhead and response time.

#### ***5.6 Conclusion***

The work conducted in this thesis was both related to hardware debugging and software implementation. The most serious problem encountered was to set up and run a FlexRay network, took a considerable amount of time and effort to solve. Much time was spent looking for the answer by studying the FlexRay protocol in detail, since it was assumed that the error was related to the configuration of the FlexRay network. However, as has been described before, the cause of the error was that a disable signal was generated from the HCS12 processor on the G2 microcontroller board. When the cause was located the problem was easily removed.

The software implemented in this thesis includes a Host Application running on a host computer and a Node Application running on each node in the GAST cluster. The Host Application and Node Application communicate with each other by serial line communication. The Node Application contains code for setting up and running a FlexRay network, along with an implemented membership algorithm. Both the Host

Application and the Node Application were developed to run on an experimental GAST-cluster, consisting of FlexRay communication controllers and G2 microprocessor boards.

A simulated test environment was set up to verify the correctness of the implemented membership algorithm. According to the correctness tests performed the algorithm functioned correctly in the simulated test environment. It was not possible to carry out performance tests since the simulated environment was very limited.

A demonstrator environment was not developed due to the problems encountered (explained in section 4.7.3).

The result of this thesis are going to be delivered to members of the GAST project and the CEDES project. The package available for delivery is the following:

- A functioning Host Application used to run and perform automated testing in a GAST cluster.
- A GAST cluster Node Application, containing a membership algorithm implementation. The Node Application utilizes the FlexRay Light drivers and the ODEEP FlexRay Configurator developed by earlier thesis workers. Many aspects of the Node Application are fully functioning. As described in the “Problems” section, however, one major problem related to the timing of FlexRay messages has not been resolved.

## References

- [1] CEDES Homepage, **The CEDES project homepage**, <http://www.cedes.se/>, 2007-01-29
- [2] Swedish National Testing and Research Institute, <http://www.sp.se/eng/default.htm>, 2007-01-29
- [3] GAST Project, **The GAST project homepage**, <http://www.chl.chalmers.se/gast/>, 2007-01-29
- [4] Carl Bergenhem, Johan Karlsson, **A Configurable Membership Service for Active Safety Technical report**, 2006-12-14
- [5] Neil Storey, **Safety-critical computer systems**, Addison Wesley Longman, 1996, ISBN 0-201-42787-7
- [6] Roger Johansson, Lars-Ake Johansson, **G2 board, Manual**, <http://www.chl.chalmers.se/gast/downloads/TR-WP-3-5-manual.pdf>, 2007-01-29
- [7] FlexRay Consortium, <http://www.flexray.com/index.php>, 2007-01-29
- [8] Freescale, **MFR4200 data sheet** [http://www.freescale.com/files/peripherals\\_coprocessors/doc/data\\_sheet/MFR4200.pdf](http://www.freescale.com/files/peripherals_coprocessors/doc/data_sheet/MFR4200.pdf), 2007-01-29
- [9] Mattias Pettersson, Petter Uvesten, **A distributed FlexRay-based research platform**, <http://www.chl.chalmers.se/gast/downloads/Pettersson-Uvesten-2006.pdf>, 2007-01-29
- [10] Sun Developer Network, **Java Communications API** <http://java.sun.com/products/javacomm/reference/faqs/index.html>, 2007-01-29
- [11] CAN in Automation, **CAN Protocol Overview**, <http://www.can-cia.org/can/>, 2007-01-29
- [12] CAN in Automation, **TTCAN Protocol Overview**, <http://www.can-cia.org/can/ttcan/>, 2007-01-29
- [13] FlexRay Consortium, **FlexRay Communication System Protocol Specification V 2.1, Revision A**, Order From: [http://www.flexray.com/specification\\_request\\_v21.php](http://www.flexray.com/specification_request_v21.php)
- [14] Valério Rosset, Pedro F. Souto and Francisco Vasques, **A Group Membership Protocol for Communication Systems with both Static and Dynamic Scheduling**, Faculdade de Engenharia da Universidade do Porto
- [15] Mattias Bergström, Johan Högberg, **Node Application Doxygen API**, 2007-01-31

[16] Mattias Bergström, Johan Högberg, **Host Application Doxygen API**, 2007-01-31

## Figure References

- [1] Figure 2.5, **Bus Topologies in FlexRay**,  
[http://www.fujitsu.com/global/services/microelectronics/technical/flexray/index\\_p11.html/](http://www.fujitsu.com/global/services/microelectronics/technical/flexray/index_p11.html/), 2007-01-31
- [2] Figure 2.6, **Communication cycle in FlexRay**,  
<http://www.evaluationengineering.com/archive/articles/0305/0305flexray.asp>, 2007-01-31
- [3] Figure 2.7, **Examples of message transmission in FlexRay**,  
[http://www.tzm.de/FlexRay/FlexRay\\_Introduction.html/](http://www.tzm.de/FlexRay/FlexRay_Introduction.html/), 2007-01-31
- [4] Figure 2.8, **Clock Synchronization in FlexRay**, FlexRay Consortium,  
**FlexRay Communication System Protocol Specification V 2.1, Revision A**,  
Order From:[http://www.flexray.com/specification\\_request\\_v21.php](http://www.flexray.com/specification_request_v21.php)



## Figure List

2.1	Illustration of the topology of a bus network.....	11
2.2	Picture of the Freescale MPC 565 MCU G2 Card. The card on the picture is from the third revision.....	12
2.3	Picture of version one of the FlexRay MRF4200 communication controller.....	13
2.4	Shows the experimental GAST cluster connected to the Infiniium Oscilloscope.....	14
2.5	Shows the two basic topology types used in FlexRay. A FlexRay network can be organized as a bus type or a star type topology. Both above can be combined to create a hybrid type topology.....	17
2.6	Shows the communication cycle in the FlexRay protocol. It also shows the different segments that make up the cycle.....	18
2.7	Shows a example of messages being sent during a single communication cycle In the FlexRay protocol.....	18
2.8	Shows the timing relationship between the media access schedule and the clock synchronization algorithms.....	19
3.1	Shows how the membership protocol communicates in a FlexRay cycle.....	24
4.1	Shows the main GUI window in the host application.....	32
4.2	Shows the communication settings GUI window in the host application.....	32
4.3	Shows the GUI window used to communicate to a single G2 card used.....	33
4.4	Shows the upload functionality GUI in the Host Application.....	34
4.5	Shows the GUI that handles Selection of test files and CSV output directory.....	34
4.6	Shows a state transition diagram of startup protocol.....	37

4.7 Shows the disable signal coming from the HCS MCU.....41

## Table List

3.1	The fault model containing two set of faults.....	25
3.2	An opinion matrix used for majority voting in the decision function.....	27
5.1	The results from the correctness tests of the simulator.....	46
5.2	2 The results from the correctness tests of the simulator (continued).....	47



# Index

## A

ABS, 5, 63

## C

CAN, 1, 5, 13, 19, 20, 55, 63, 64

CEDES, 6, 7, 53, 55, 63, 73, 87

## D

decision function, 29, 30, 31, 33, 61

*Drive-by-wire*, 10

## E

Eclipse, 17, 63

ECU, 11

*event-triggered*, 5, 19, 20, 21

## F

failure cases, 30, 41, 43

Fault injection, 2, 41

fault model, 29, 30, 61

fault tolerance, 9

fault-tolerance, 6

FlexRay, 1, 2, 5, 6, 7, 11, 13, 14, 15, 16, 17, 19, 20,  
21, 22, 23, 24, 25, 27, 28, 32, 35, 36, 39, 40, 41,  
43, 45, 46, 47, 48, 49, 53, 55, 57, 59, 61, 63, 64,  
73, 87, 109

## G

G1, 7

G2, 6

GAST, 6, 7, 11, 13, 14, 15, 16, 17, 27, 29, 32, 35, 36,  
39, 43, 44, 45, 49, 52, 53, 55, 59, 63, 71, 73, 87

GAST cluster, 7, 35, 39, 53

GEE, 17

*graceful degradation*, 9

## H

host application, 2, 17, 35, 36, 37, 38, 39, 51, 52, 53,  
59, 69, 71, 87

## M

MCU565, 13

membership, 2, 3, 6, 7, 9, 11, 20, 27, 28, 29, 30, 31,  
32, 33, 35, 38, 39, 40, 41, 43, 44, 47, 48, 49, 51,  
52, 53, 59, 63, 64, 67, 69, 71, 73, 75, 76, 77, 78,  
79, 80, 83, 93, 99, 100, 108, 109, 110, 111, 112

Membership algorithm, 1, 2, 6, 32, 39

MFR4200, 6, 14, 15, 55

mingw, 17, 64

MPC565, 6

## N

Netbeans, 17

## R

redundancy, 9, 11, 28, 64

reliability, 9, 10

## S

startup protocol, 2, 39, 40, 41, 47, 59

## T

*time-triggered*, 5, 6, 19, 20, 21, 27

TTCAN, 1, 5, 7, 13, 19, 20, 55, 64

## X

x-by-wire, 5, 6, 9, 10, 11





## Appendix A

### Basic membership protocol with reintegration

#### A.1 Introduction

The paper written by Rosset, Souto and Vasques [14] along with the paper by Bergenhem and Karlsson was the bases for the membership algorithm implemented in this thesis. To give the reader a clearer view of how the algorithm works, the basic membership protocol with reintegration as defined by [14] is presented here.

#### A.2 Pseudo code

##### *State*

**P** the set of all processors

**M-SET** the set of group members, initially set to P

**M-set** the set of candidate group members, initially set to P

**u** upper bound of the group's size, initially set to  $|P|$

##### *FD-phase*

###### **Communication step:**

**If** processor is group member

**Then** broadcast heartbeat message

**Else if** wishing to join group

**Then** set the M-set and the M-SET to P, u to the size of P, and broadcast join-req message.

###### **Processing step:**

1. Remove from the M-set every processor from which no heartbeat message was received.

2. For every join-req message received add its sender to the M-set.

##### *GM-phase*

###### **Communication step:**

Broadcast message with the M-set and the group's size upper bound, u.

###### **Processing step:**

1. Let Maj-set be the result of applying the majSet function to P, the set of all the M-set's received from processors in the M-SET and the minimum of all u's received in the same messages.

2. If the Maj-set

(a) is undefined, or

- (b) is different from the M-set the processor broadcasted and the processor is a member of the group, or
- (c) is not a subset of the M-set the processor broadcasted and the processor is joining the group, or
- (d) does not contain the processor then halt.

**3.** Remove from the M-set

- (a) every group member from which an M-set different from the Maj-set was received;
- (b) every joining processor whose M-set is not a superset of the Maj-set;

**4.** Set  $u$  to the size of the M-set.

**5.** Remove from the M-set every processor from which no message was received in this phase.

**6.** Set the M-SET to the M-set.

## **Appendix B**

### **The Host – Node Communication Protocol**

#### ***B.1 Introduction***

Communication between host and node is done by sending a message header specifying what kind of message the recipient can accept. The receiver then takes appropriate measures to receive and handle the following data. Each header is preceded by a SOH (Start of Header) symbol. This is a one byte ASCII symbol (0x01) representing the start of a communication header, and in this case start of a communication phase between the host and a node.

#### ***B.2 Pre membership communication***

The pre host communication protocol handles the transmission of test specifications from the host application to each node in the system. The received test specifications are stored locally on each node and are the bases for the fault injection in this node. If no test was specified for a particular node this is indicated to the node in the communication through sending a special message indicator. If this is the case the membership algorithm running on the node will exclude the fault injection parts.

#### ***B.3 Post membership communication***

When each node stops to execute the membership algorithm it will send a number of stored test data to the host. The test data are stored locally on each node and reflects changes made locally in the node; see section 7.1.4 for more details. Each node can store a predefined maximum of structures, each containing test data reflecting the view of the particular node in the cycle when that data is stored. The real upper bound on the number of data structures that can be stored, are related to memory size of the MPC 565 MCU.

Each MPC 565 has around 1 MB user defined flash memory. The node applications files each contain 95 KB of data so the rest can be used to store test data structures. The node application is only responsible for sending the test data to the host application. No work goes into analysing or formatting the data more then storing it into structures.



## Appendix C

### The Test Specification File Format

#### C.1 Introduction

To perform automated tests on the GAST cluster the tests has to be specified in some way. In this thesis the fault injection test are specified through a XML file. A test specification file contains a number of test cases that each will be injected into the membership algorithm in a predefined cycle.

#### C.2 File Format

```
<?xml version="1.0"?>

<testcases>

  <recording>
    <startcycle>10</startcycle>
    <stopcycle>30</stopcycle>
  </recording>

  <testcase>
    <node>0</node>
    <testtype>ILF</testtype>
    <startcycle>10</startcycle>
    <startphase>FD</startphase>
    <stopcycle>12</stopcycle>
    <stopphase>MD</stopphase>
  </testcase>

</testcases>
```

The code snippet above is an example of a test specification file.

A test specification file consists of a number of tags. The recording tag tells the node when to start injecting faults in the system and when to stop injecting faults and start returning test data to the host. Each testcase section is a test specified with a number of tags. Each testcase tells the host which node that shall receive the specified test, what kind of test that the node shall conduct and when the fault shall be injected and removed from the system. All this information is sent to the nodes by the host application before the membership protocol is run on the nodes.



## Appendix D

### Node Application API documentation

#### ***D.1 Introduction***

This is the Node Application written to run on a G2 Processor card. These function are meant to be used by a application programmer to run and test membership algorithms in a GAST cluster.

Extensive documentation can be found in [15], which are available to any member of the GAST and CEDES project.

#### ***D.2 src/constraints.c File Reference***

##### **D.2.1 Details Description**

Reads the different configuration values in the FlexRay communication controller and checks if they comply with the constraints specified in PS v2.1.

**Date:**

2006-12-01 14:16

**Author:**

Mattias Bergström

**See also:**

For more information, see PS v2.1, page 214

##### **D.2.2 Functions**

- **void check\_constraints ()**

Includes the .h file defining function prototypes for this .c file Check the constraints specified in the FlexRay Protocol Specification.

#### ***D.3 src/faultinjection.c File Reference***

##### **D.3.1 Details Description**

Handles the fault injection stuff.

**Date:**

2006-12-27 18:09

**Author:**

### D.3.2 Functions

- **int getNextErrorSpecification (errorSpecification errorSpec[], errorSpecification currentError)**  
Gets the next errorSpecification struct from an array of errorSpecifications.
- **int ilfDuringCurrentPhase (errorSpecification currentError, int cycleNr, int phaseNr)**  
Checks if we should have an ILF in the specified phase and cycle.
- **int nsDuringCurrentPhase (errorSpecification currentError, int cycleNr, int phaseNr)**  
Checks if we should have an ILF in the specified phase and cycle.

### D.3.3 Variables

- **UINT8 receive\_buff [20]**  
Used to store received bytes.

## D.4 src/funcs.c File Reference

### D.4.1 Details Description

**Author:**

Mattias Bergström

### D.4.2 Functions

- **void memcpy (UINT8 \*destinationPointer, UINT8 \*sourcePointer, int nBytes)**  
Copies the memory of one pointer to another.
- **void delay (UINT32 length)**  
Perform a number of dummy operations.
- **void putc (UINT8 ch)**  
Writes a character to the SCIDR1 register.
- **UINT8 hasSerialLineAvailableBytes ()**  
Returns TRUE if there are bytes to read on the serial line
- **UINT8 getc (void)**  
Reads one byte from the serialport.
- **void getBytes (int length, UINT8 valArray[])**  
Get n number of bytes from the serialport.

- **void puts (UINT8 \*s)**  
Writes n number of characters to the serialport.
- **UINT8 \* gets ()**  
Get a number of characters from the serialport.

#### D.4.3 Variables

- **UINT8 receive\_buff [20]**  
Used to store received bytes.

### D.5 *src/hostcommunication.c* File Reference

#### D.5.1 Details Description

Communication between host and node.

**Date:**

2007-01-18 14:03

**Author:**

Mattias Bergström

#### D.5.2 Functions

- **int preHostCommunication (errorSpecification errorSpec[])**  
Constitutes the communication between host and node before the membership is running.
- **void postHostCommunication (testData tData[], UINT16 nrOfTestCases)**  
Constitutes the communication between the host and the node taking place after the membership code is run.
- **void sendMessageHeader (UINT16 msgType)**  
Sends a message header.
- **hostCommunicationMessageHeader getMessageHeader ()**  
Reads the message header data from the host.
- **void getMainTestParameters (mainTestParameters mainParam)**  
Gets the main test parameters from the host.
- **void getNrOfTestErrorSpecifiactions ()**  
Gets the number of test error specifications we can expect to receive.
- **void getErrorSpecification (errorSpecification errorSpec[])**  
Collects N number of error specifications from the host.

- **void sendTestData (testData tData[], UINT16 nrOfTestCases)**  
Sends test data we collected through the membership testing.
- **void sendData (UINT8 \*data, UINT32 msgLength)**  
Send raw data over the serial bus
- **UINT16 waitForData ()**  
Loops until data is available on the serial bus.

### D.5.3 Variables

- **UINT32 nrOfTestErrorSpec**  
Variable declared in the host communication file.

### D.5.4 Defines

- **#define MAX\_TEST\_ERROR\_NR**  
Defines the maximum of test error specifications we want to save

## D.6 *src/math.c* File Reference

### D.6.1 Details Description

All mathematic related functions used in the nodes.

**Date:**

2006-12-01 14:16

**Author:**

Mattias Bergström

### D.6.2 Functions

- **UINT16 ceil (float value)**  
Calculates a ceiling value for a given float.

## D.7 *src/membership.c* File Reference

### D.7.1 Details Description

Contains the membership voting process done in each node in the cluster.

**Date:**

2006-12-01 14:16

**Author:**

Mattias Bergström

Johan Högberg

## D.7.2 Functions

- **void initializeMembership (int nodeNr)**  
Initializes the membership functions
- **void waitUntilMembershipReady ()**  
Startup function
- **void setStaticMessageNodeStatus (UINT16 status)**  
Set the status (heartbeat/joining-node byte) for the current message
- **void setStaticMessageCycleNumber (UINT16 currentCycleCounter)**  
Set the cycle counter for the current message
- **void sendStaticMessage ()**  
Send the static message
- **int hasNodeSentStaticMessage (int nodeNr)**  
Return TRUE/FALSE whether the specified node has sent a static message during the current cycle
- **int receiveStaticMessage (int nodeNr)**  
Receive a static message from a specified node
- **int getReceivedStaticMessageStatusByte ()**  
Get the status (heartbeat/joining-node byte) for the received static message
- **void sendDynamicMessage ()**  
Send dynamic message
- **int receiveDynamicMessage (int nodeNr)**  
Receive dynamic message from a specified node
- **int hasNodeSentDynamicMessage (int nodeNr)**  
Return TRUE/FALSE whether the specified node has sent a dynamic message or not
- **void membershipVoting (set \*votedMemberNodesSet, set \*undefinedNodesSet, set \*opinions, int umin)**  
Controls the voting process for each process
- **void failureDetection ()**  
Send the raw data along with the heartbeat for the process.
- **void membershipCommunication ()**  
Broadcasts its opinion based on the heartbeats received in the failure detection phase.
- **void membershipDecision ()**

Uses the collection of opinion sets to perform a majority voting and obtain a "global" membership agreement.

- **void run\_membership ()**  
Calls the different parts of the membership protocol
- **int getTestResult (testData newTestData[])**  
Copies the stored test data into newTestData[]
- **void checkValues ()**  
Check if any of our trigger values changes from the once we stored before
- **void storeCurrentValues ()**  
Store the current test values each cycle
- **void start\_error (errorSpecification cError)**  
Starts executing an error injection
- **void stop\_error ()**  
Stops executing an error injection
- **void setToByte (set s, UINT8 byte)**  
Convert a set to a byte for test storing

### D.7.3 Variables

- **UINT16 cycle\_counter**  
keep track of the current cycle
- **int nMaximumNodesMembers**  
Maximum amount of membership nodes
- **int thisNodeIsMember**  
Indicate if the current node is a member of the cluster or not.
- **UINT8 halted**  
Indicate if the current node is halted or not.
- **int haltedCycleCounter**  
Contains the number of cycles since we got booted out of the membership
- **set membershipSet**  
To store the local membership
- **set joiningNodesSet**  
To store the nodes that is attempting to join the membership in this cycle.
- **staticSlotCommunicationStruct staticSendStruct**  
Structure used when the node is sending data in the static slot.

- **staticSlotCommunicationStruct staticReceiveStruct**  
Structure used when the node is receiving data in the static slot.
- **dynamicSlotCommunicationStruct dynamicSendStruct**  
Structure used when the node is sending data in the dynamic slot.
- **dynamicSlotCommunicationStruct dynamicReceiveStruct**  
Structure used when the node is receiving data in the dynamic slot.
- **set opinionSet**  
To store the local opinion for this cycle.
- **set opinionSets [N\_NODES]**  
Array of sets containing the opinions of the various nodes.
- **set allNodesSet**  
Set that contains all the nodes in the cluster.
- **multiset nodesAliveSet**  
To store the number of nodes that the various nodes have found to be alive in the current cycle.
- **int nNodesAlive**  
To store the number nodes that the current node has found to be alive in the current cycle.
- **int joinRequest**  
indicates whether we want to join the membership or not
- **int currentNodeNr**  
The index of the current node.
- **UINT16 staticBufferIndices [N\_NODES]**  
Used to store the indices of the buffers for the static transmissions.
- **UINT16 dynamicBufferIndices [N\_NODES]**  
Used to store the indices of the buffers for the dynamic transmissions.
- **int hasCurrentError**  
Indicate if a error is introduced this cycle or not.
- **errorSpecification currentError**  
Stores the current error we are injected.
- **UINT32 nrOfTestErrorSpec**  
Variable declared in the host communication file.

- **testData tData [MAX\_NR\_TEST\_DATA]**  
We can store a maximum of 100 test data struct.
- **tempData currentStatus**  
Stores the current status of all data members in the struct.
- **UINT32 nrOfSavedTestData**  
Stores the number of save test cases.

## **D.8 src/node1-4Application.c File Reference**

### **D.8.1 Details Description**

This code contains an implementation of the basic membership algorithm with reintegration. The algorithm is described in "A Configurable Membership Service for Active Safety Systems" in section 3.3.1

**Date:**

2006-12-01 14:16

**Attention:**

- 1) We assume one process per node
- 2) We assume that initially all processes are members
- 3) We assume that a node only tried to reintegrate if all its processes are functioning
- 4) We assume that if a node becomes halted it tries to reintegrate after 4 cycles
- 5) We assume that a halted node stops sending heartbeats but can still listen and receive messages. The node also skips the MC and MD phase

**Author:**

Mattias Bergström

**See also:**

"A Configurable Membership Service for Active Safety" in section 3.3.1

### **D.8.2 Functions**

- **void print\_state ()**  
Calls the flexray\_get\_state() function to obtain the current state for the CC and pretty prints this.
- **void init\_flexray ()**  
Initiates the Flexray communication by initiating the node in the network.
- **void wait\_for\_stateTwo ()**  
Sleeps for a predetermined time and then checks if the network has reached the normal active operation state.
- **void get\_startup\_states ()**  
Used to store the states the node goes through during startup. Good for debugging.

- **void run ()**  
Error handling code for the node.

### **D.8.3 Variables**

- **int cardOffset**  
Cardoffset used by the G2 card to indicate where in memory the code is stored.
- **UINT16 cycle\_counter**  
keeps track of the current cycle
- **int runTest**  
indicates if we should run any tests this cycle or not
- **errorSpecification errorSpec [MAX\_TEST\_ERROR\_SPEC]**  
To store the errorSpecifications from the host.
- **int hasCurrentTest**  
If we have a current test running or not.

## ***D.9 src/output.c File Reference***

### **D.9.1 Details Description**

Output functions.

**Date:**

2006-12-04

**Author:**

Carl Leskinen

Mattias Bergström

### **D.9.2 Functions**

- **void putBytes (int length, UINT8 \*valPtr)**  
Prints out n number of bytes as characters.
- **void putSingleDigit (UINT8 value)**  
Prints out a single digit.
- **void putUINT8 (UINT8 value)**  
Prints a character.
- **void putHexUINT8 (UINT8 value)**  
Prints a character in Hexadecimal form.

- **void putUINT16 (UINT16 value)**  
Takes a UINT16 as input and prints out its decimal representation on the form: "89".
- **void putHexUINT16 (UINT16 value)**  
Prints a short in Hexadecimal form.
- **void putUINT32 (UINT32 value)**  
Prints a int.
- **void putHexUINT32 (UINT32 value)**  
Prints a int in Hexadecimal form.
- **void xto2a (UINT8 reg)**  
Takes a UINT8 as input and prints out its hexadecimal representation on the form: "3F".
- **void ito2a (UINT16 reg)**  
Takes a UINT16 as input and prints out its decimal representation on the form: "89". '##' if reg is greater than 9999.
- **void itoax (UINT32 reg)**  
Takes a UINT32 as input and prints out its hexadecimal representation on the form: "0xDEADBEEF".
- **void nl ()**  
Prints a new line.

## ***D.10 src/set.c File Reference***

### **D.10.1 Details Description**

All set functions used for membership set manipulation.

**Date:**

2006-12-11 10:55

**Author:**

Mattias Bergström

Johan Högberg

### **D.10.2 Functions**

- **void setClear (set \*a)**  
Clear a set by setting the amount of members in the set to zero.
- **void multisetClear (multiset \*a)**

Clear a multiset by setting the amount of members in the set to zero.

- **int setSize (set \*a)**  
Set the size of a set.
- **int multisetSize (multiset \*a)**  
Set the size of a multiset.
- **int setIsEmpty (set \*a)**  
Check if a set is empty.
- **int multisetIsEmpty (multiset \*a)**  
Check if a multiset is empty.
- **set setCreate ()**  
Create and return an empty set
- **multiset multisetCreate ()**  
Create and return an empty multiset
- **int setIsSuperset (set \*a, set \*b)**  
Check if set a is a superset of set b.
- **int setIsSubset (set \*a, set \*b)**  
Check if set a is a subset of set b.
- **int setAreEqual (set \*a, set \*b)**  
Check if the set a is equal to the set b.
- **UINT16 setMin (set \*u\_set)**  
Returns the minimum value of a set
- **UINT16 multisetMin (multiset \*u\_set)**  
Returns the minimum value of a multiset
- **UINT16 setMax (set \*u\_set)**  
Returns the maximum value of a set
- **UINT16 multisetMax (multiset \*u\_set)**  
Returns the maximum value of a set
- **int setIsMember (UINT16 element, set \*a)**  
Check if an element is a member of the set.
- **int multisetIsMember (UINT16 element, multiset \*a)**  
Check if an element is a member of the multiset.
- **void setAddElement (UINT16 element, set \*a)**

Add an element to the set.

- **void multisetAddElement (UINT16 element, multiset \*a)**  
Add an element to the set.
- **int setRemoveElement (UINT16 element, set \*a)**  
Remove an element from the set.
- **int multisetRemoveElement (UINT16 element, multiset \*a)**  
Remove an element from the multiset.
- **void setPrint (set \*a)**  
Print the members of a set.
- **void multisetPrint (multiset \*a)**  
Print the members of a set.

## ***D.11 src/timer.c File Reference***

### **D.11.1 Details Description**

Functions implementing the timer used in the nodes.

**Date:**

2006-12-01 14:16

**Author:**

Johan Högberg

### **D.11.2 Functions**

- **void initTBSCR ()**  
This code enables the timer, and sets the timer period to TICK\_LENGTH.
- **UINT16 getTBSCR ()**  
This function returns the contents of the TBSCR register.
- **UINT32 getTBU ()**  
This function returns the upper 32 bits of the 64-bit counter.
- **UINT32 getTBL ()**  
This function returns the lower 32 bits of the 64-bit counter.
- **void getTB (UINT32 \*tbuVal, UINT32 \*tblVal)**  
This function returns the 64-bit counter, divided into two 32-bit variables.
- **double getTimeDifference (UINT32 time1tbu, UINT32 time1tbl, UINT32 time2tbu, UINT32 time2tbl)**  
This function returns the difference between the lower 32-bits of the timer (to be implemented: use of a 64-bit data type).
- **UINT32 getTimeTickDifference (UINT32 time1tbu, UINT32 time1tbl, UINT32 time2tbu, UINT32 time2tbl)**

This function returns the difference between the lower 32-bits of the timer (to be implemented: use of a 64-bit data type).

## Appendix E

### Host Application API documentation

#### ***E.1 Introduction***

This is the Host Application functions. These functions are meant to be used by application programmers to modify the host application made to run a GAST cluster consisting of FlexRay communication controllers and G2 processor cards and do automated fault injection testing on the cluster.

Extensive documentation can be found in [16], which are available to any member of the GAST and CEDES project.

#### ***E.2 src/CommSettings.c File Reference***

##### **E.2.1 Details Description**

Contains settings for serial port communication.

**Date:**

15 October 2006, 11:27

**Author:**

Mattias Bergström

##### **E.2.2 Public Member Functions**

- **CommSettings ()**  
Default constructor for the.
- **void setBR (int newBR)**  
sets the baud rate value to the value of the parameter
- **void setBRSelectedIndex (int index)**  
Set baud rate combo box index (used in the GUI).
- **void setP (int newP)**  
Set parity.
- **void setPSelectedIndex (int index)**  
Set parity combo box index.
- **void setSB (double newSB)**

Set stop bit settings.

- **void setSBSelectedIndex (int index)**  
Set stop bit combo box index.
- **void setDB (int newDB)**  
Set data bits setting.
- **void setDBSelectedIndex (int index)**  
Set stop bit combo box index.
- **int getBR ()**  
Returns the baud rate.
- **int getBRSelectedIndex ()**  
Returns selected baud rate index in the BDComboBox.
- **int getP ()**  
Returns the parity value.
- **int getPSelectedIndex ()**  
Returns selected parity combo box index in the PComboBox.
- **int getSB ()**  
Returns the stop bit.
- **int getSBSelectedIndex ()**  
Returns selected stop bits combo box index in the PComboBox.
- **int getDB ()**  
Returns the data bit.
- **int getDBSelectedIndex ()**  
Returns selected databits combo box index in the PComboBox.
- **boolean isComSelected (int index)**  
Returns selected databits combo box index in the PComboBox.
- **boolean isCom[1-9]Selected ()**  
Return com port status for a specified com port.

### **E.2.3 Package Functions**

- **void setCom[1-9] (boolean comPortSelected)**  
Set com port status for the specified com port.

### ***E.3 src/CommSettingsUI.c File Reference***

### **E.3.1 Details Description**

User interface for the communication settings.

**Date:**

23 November 2006, 16:01

**Author:**

Johan Högberg

### **E.3.2 Public Member Functions**

- **CommSettingsUI (HostApplicationUI newHostApplicationUI)**  
Constructor for the ComSettingsUI object.

### **E.3.3 Static Public Attributes**

- **static CommSettings commSettings**  
stores the COM settings done by the user

## ***E.4 src/CommWindowUI.c File Reference***

### **E.4.1 Details Description**

Provide a GUI for the terminal for each com port.

**Date:**

23 November 2006, 16:41

**Author:**

Johan Högberg

### **E.4.2 Public Member Functions**

- **CommWindowUI(HostApplicationUI hostApplicationUI, String title, String commPort, MessageQueue hostMessageQueue, MessageQueue commMessageQueue, int xpos, int ypos)**  
Constructor for the ComWindowUI object

## ***E.5 src/CommWorker.c File Reference***

### **E.5.1 Details Description**

There is one CommWorker thread underlying each of the terminals. The CommWorker threads handle the com port communication as well as communication with the master thread that is coordinating the threads.

**Date:**

27 November 2006, 12:44

**Author:**

Johan Högberg

Mattias Bergström

**E.5.2 Public Member Functions**

- **public CommWorker(HostApplicationUI hostApplicationUI, String commPort, CommSettings serSet, JTextArea newTerminalTextArea, MessageQueue hostMessageQueue, MessageQueue commMessageQueue)**  
Initiates a communication with the communication port
- **public void run()**  
Starting method of the thread
- **public void getMessage()**  
Gets the message header from the node and acts according to the message type

**E.6 src/Converter.c File Reference****E.6.1 Details Description**

Contains methods used to convert a data type into another.

**Date:**

11 January 2007, 09:18

**Author:**

Mattias Bergström

**E.6.2 Public Member Functions**

- **Converter ()**  
Default constructor for the Converter class.
- **byte[] shortToByteArray (short value)**  
Converts a short into a array of bytes.
- **byte[] intToByteArray (int value)**  
Converts a int into a array of bytes.
- **int byteArrayToInt (byte[] b)**  
Converts a nyte array into a integer value.

- **int byteArrayToInt (byte[] b, int offset)**  
Converts a byte array into a integer.
- **short byteArrayToShort (byte[] b)**  
Converts a byte array into a short value.
- **short byteArrayToShort (byte[] b, int offset)**  
Converts a byte array into a short.
- **String byteArrayToString (byte[] b)**  
Converts a byte array into a string

## ***E.7 src/HostApplicationUI.c File Reference***

### **E.7.1 Details Description**

Main GUI class.

**Date:**

23 November 2006, 11:14

**Author:**

Johan Högberg

### **E.7.2 Public Member Functions**

- **HostApplicationUI ()**  
Creates new form HostApplicationUI.
- **CommSettings getCommSettings ()**  
Return communication settings object.

### **E.7.3 Static Public Member Functions**

- **static void main (String args[])**  
Starting method.

### **E.7.4 Static Public Attributes**

- **static final String IMAGE\_FILE = "images/small\_flexray\_icon.gif"**  
Stores path to the image file.

### **E.7.5 Package Attributes**

- **MessageQueue hostMessageQueue\_**  
The message queue used when communicatiting with the host thread.

- **MessageQueue commMessageQueue\_ []**  
The message queue used when communicating with all the com worker threads.
- **boolean commPorts []**  
To send the connected comports to the upload files UI.

### E.7.6 Static Package Attributes

- **static TestFileSettings testFileSettings = new TestFileSettings()**  
Stores the testfile settings.
- **static CommSettings commSettings = new CommSettings()**  
Stores the communication settings.

## E.8 *src/HostMessage.c* File Reference

### E.8.1 Details Description

Message object for communication with the main host thread.

**Date:**

6 December 2006, 11:36

**Author:**

Johan Högberg

### E.8.2 Public Member Functions

- **HostMessage (Object sender, String newText)**  
Creates a new instance of HostMessage.
- **HostMessage (Object sender, TestSpecification testSpecification, File file)**  
Creates a new instance of HostMessage.
- **HostMessageType getMessageType ()**  
Return message type.
- **String getString ()**  
Return String (if message is of type TEXT\_MESSAGE).
- **TestSpecification getTestSpecification ()**  
Return TestSpecification object (if message is of type TEST\_SPECIFICATION\_MESSAGE).

## ***E.9 src/MainTestSpecification.c File Reference***

### **E.9.1 Details Description**

Stores the main test parameters. These are used to specify start and stopcycle for the membership testing

**Date:**

30 December 2006, 15:53

**Author:**

Mattias Bergström

### **E.9.2 Public Member Functions**

- **MainTestSpecification (int fromCycle, int toCycle)**  
Creates a new instance of MainTestSpecification.
- **int getFromCycle ()**  
Returns the start cycle.
- **int getToCycle ()**  
Returns the end cycle.
- **void setFromCycle (int fromCycle)**  
sets the value of the fromCycle
- **void setToCycle (int toCycle)**  
sets the value of the toCycle

## ***E.10 src/Message.c File Reference***

### **D.10.1 Details Description**

Abstract class used as a template for the various types of messages to be sent within the application.

**Date:**

4 December 2006, 15:22

**Author:**

Johan Högberg

### **E.10.2 Public Member Functions**

- **Message (Object sender)**  
Creates a new instance of Message.

- **Object getSender ()**  
Return sender object.

## ***E.11 src/MessageQueue.c File Reference***

### **E.11.1 Details Description**

Message queue object onto which all messages are posted. In the whole application there is one message queue for each thread that needs to receive communication from other threads.

**Date:**

4 December 2006, 14:24

**Author:**

Johan Högberg

### **E.11.2 Public Member Functions**

- **MessageQueue ()**  
Creates a new instance of MessageQueue.
- **synchronized Message dequeue ()**  
Get the first message in the message queue.
- **synchronized void enqueue (Message newMessage)**  
Deposit a new message in the message queue.
- **boolean isEmpty ()**  
Return message queue status.

## ***E.12 src/NodeCommunication.c File Reference***

### **E.12.1 Details Description**

Controls the communication with the nodes.

**Author:**

Mattias Bergström

**Date:**

11 januari 2007, 10:28

### **E.12.2 Public Member Functions**

- **NodeCommunication (InputStream inputStream, OutputStream outputStream, String commPort)**  
Creates a new instance of NodeCommunication.
- **void sendData (byte array[])**  
Writes a byte array to the serialport.
- **void sendData (short value)**  
Writes a short value to the serialport.
- **void sendData (int value)**  
Writes a int value to the serialport.
- **void sendSOH ()**  
Writes the Start Of Header byte to the serialport
  - **Exceptions:**
    - *IOException* is thrown if there is any problem when writing to the serialport.
- **NodeCommunicationHeader getHeader ()**  
Reads the message type data from the serial port.
- **byte[] getData (int msgLength)**  
Reads msgLength of bytes from the serialport.
- **int getData ()**  
Reads one byte from the serialport
  - **Exceptions:**
    - *IOException* is thrown if there is any problem when reading from the serialport.
- **void sendMessage (NodeMessageType type)**  
Sends the members of a NodeCommunicationHeader object to the serialport.

## ***E.13 src/NodeCommunicationHeader.c File Reference***

### **E.13.1 Details Description**

Stores message node headers. Used to store received message node headers and to store message node header scheduled for transmission

**Date:**

3 January 2007, 15:38

**Author:**

Mattias Bergström

### **E.13.2 Public Member Functions**

- **NodeCommunicationHeader (NodeMessageType type)**  
Creates a new NodeCommunicationHeader object.
- **NodeMessageType getMsgType ()**  
Returns the message type of this object.

## ***E.14 src/NodeMessage.c File Reference***

### **E.14.1 Details Description**

Used when communicating with the node.

**Date:**

3 January 2007, 13:44

**Author:**

Mattias Bergström

### **E.14.2 Public Member Functions**

- **NodeMessage (NodeCommunicationHeader header)**  
Creates a new instance of NodeMessage

## ***E.15 src/TerminalMessage.c File Reference***

### **E.15.1 Details Description**

Message object used when communicating with the CommWorker threads.

**Date:**

5 December 2006, 17:21

**Author:**

Johan Högberg

### **E.15.2 Public Member Functions**

- **TerminalMessage (Object sender, TerminalMessageType msgType, String newText)**  
Creates a new instance of TerminalMessage of type TEXT\_MESSAGE.
- **TerminalMessage (Object sender, TerminalMessageType msgType, TestSpecification testSpec)**  
Create a new instance of TerminalMessage of type TEST\_MESSAGE.
- **TerminalMessage (Object sender, TerminalMessageType msgType)**  
Create a new instance of TerminalMessage of type KILL\_MESSAGE.

- **TerminalMessageType getMessageType ()**  
Return message type.
- **String getString ()**  
Return String (for text message).
- **TestSpecification getTestSpecification ()**  
Return the testspecification object.

## ***E.16 src/TestCommunicator.c File Reference***

### **E.16.1 Details Description**

Handles the test communication with the node, i.e sending of test specifications and reception of test data

**Date:**

12 January 2007, 12:06

**Author:**

Mattias Bergström

### **E.16.2 Public Member Functions**

- **TestCommunicator (NodeCommunication nodeComm, String commPort)**  
Creates a new instance of TestCommunicator.
- **void runTest ()**  
Tells the node to start the test.
- **void sendTestSpecification (TestSpecification testSpec)**  
Send the testspecification to the node.
- **void sendErrorSpecification ()**  
Sends the testspecification to the node, if no testspecifications was specified for this node we send a NO\_TEST\_SPECIFIED message to the node to let it know that it can stop waiting
- **void sendNrOfTestErrorSpecifications (int nrOfTES)**  
Lets the node know how many test error specifications we are about to send.
- **void sendMessage (TestErrorSpecification testSpec[])**  
Send a array of testerror specifications to the node.
- **void sendMessage (MainTestSpecification mainTest)**  
Send a array of main test specifications to the node.
- **TestData[] getTestData ()**

Receives and saves the test data from the node.

- **void setTestSpecification (TestSpecification testSpec)**  
Sets the testspecification.

## ***E.17 src/TestData.c File Reference***

### **E.17.1 Details Description**

Stores the test data received from a node.

**Date:**

11 January 2007, 10:12

**Author:**

Mattias Bergström

### **E.17.2 Public Member Functions**

- **TestData ()**  
Creates a new instance of TestData.
- **TestData (int cycleNr, byte membership, byte jNodes, byte opinion, boolean halted)**  
Creates a new instance of TestData.
- **int getCycleNr ()**  
Returns the cycle stored in the object.
- **byte getMembership ()**  
Returns the byte representing the membership vector.
- **byte getjNodes ()**  
Returns the byte representing witch nodes are joining nodes in this cycle.
- **byte getOpinion ()**  
returns the current nodes
- **boolean getHalted ()**  
returns a boolean indicating if this node was halted or not in the particular cycle
- **void setCycleNr (int cycleNr)**  
sets the cycleNr\_ to the value of cycleNr
- **void setMembership (byte membership)**  
sets the membership vector

- **void setjNodes (byte jNodes)**  
Sets the value of jNodes\_
- **void setOpinion (byte opinion)**  
Sets the value of opinion\_
- **void setHalted (boolean halted)**  
Sets the value of halted\_

## ***E.18 src/TestErrorSpecification.c File Reference***

### **E.18.1 Details Description**

Stores the test error specifications that we are going to send to the node.

**Date:**

30 December 2006, 15:54

**Author:**

Mattias Bergström

### **E.18.2 Public Member Functions**

- **TestErrorSpecification (int nodeNr, ErrorType errorType, int startCycle, CommPhase startPhase, int stopCycle, CommPhase stopPhase)**  
Creates a new instance of TestErrorSpecification.
- **ErrorType getErrorType ()**  
Returns the errorType
  - **Returns:**
    - ErrorType\_ - The Error Type.
- **int getStartCyle ()**  
Returns the start cycle
  - **Returns:**
    - startCycle\_ - The start cycle.
- **CommPhase getStartPhase ()**  
Returns the startphase
  - **Returns:**
    - startPhase\_ - The start phase.
- **int getStopCycle ()**  
Returns the stop cycle
  - **Returns:**
    - stopCycle\_ - The stop cycle.
- **CommPhase getStopPhase ()**

- Returns the stop phase
  - **Returns:**
    - StopPhase\_ - The stop phase.
    -
- **int getNodeNr ()**
  - Returns the node number
  - **Returns:**
    - nodeNr\_ - The node number.
- **void setNodeNr (int nodeNr)**
  - Sets the node number.
- **void setErrorType (ErrorType errorType)**
  - Sets the error type.
- **void setStartCycle (int startCycle)**
  - Sets the start cycle.
- **void setStartPhase (CommPhase startPhase)**
  - Sets the start phase.
- **void setstopCycle (int stopCycle)**
  - Sets the stop cycle.
- **void setstopPhase (CommPhase stopPhase)**
  - Sets the stop phase.

## ***E.19 src/TestFileParser.c File Reference***

### **E.19.1 Details Description**

Parser class used for parsing the XML test specification files.

**Date:**

7 december 2006, 15:40

**Author:**

Mattias Bergström  
Johan Högberg

### **E.19.2 Public Member Functions**

- **TestFileParser ()**
  - Creates a new instance of TestFileParser.

### **E.19.2 Static Public Member Functions**

- **static TestSpecification parseFile (String fileName)**
  - Parses a XML File.

## ***E.20 src/TestFileSettings.c File Reference***

### **E.20.1 Details Description**

Class containing settings such as the last selected directory for storing output files as well as the last test specification file loaded.

**Date:**

11  
December 2006, 15:46

**Author:**

Johan Högberg

### **E.20.2 Public Member Functions**

- **void setTestSpecificationFile (File file)**  
Sets the test specification file.
- **File getTestSpecificationFile ()**  
Returns the test specification file descriptor.
- **void setOutputFileDirectory (File file)**  
Sets the Output file.
- **File getOutputFileDirectory ()**  
Returns the output file.

## ***E.21 src/TestFileUI.c File Reference***

### **E.21.1 Details Description**

User interface for the loading of test specifications files in XML format.

**Date:**

7 december 2006, 16:09

**Author:**

Mattias Bergström  
Johan Högberg

### **E.21.2 Public Member Functions**

- **TestFileUI (HostApplicationUI hostApplicationUI, MessageQueue hostMessageQueue)**  
Creates new form TestFileUI.

## ***E.22 src/TestFileWorker.c File Reference***

### **E.22.1 Details Description**

Gets the test data and stores it into a text file.

**Date:**

12 January 2007, 09:45

**Author:**

Mattias Bergström

### **E.22.2 Public Member Functions**

- **TestFileWorker ()**  
Creates a new instance of TestFileWorker.
- **TestFileWorker (TestData testData[])**  
Creates a new instance of TestFileWorker.
- **TestData[] getTestData ()**  
Returns the testdata.
- **void setTestData (TestData testData[])**  
Sets the testdata.
- **void storeTestData (String path, TestData testData[], String fileName)**  
Creates a new file and stores the test data in this file.

## ***E.23 src/TestSpecification.c File Reference***

### **E.23.1 Details Description**

Test specification object containing information about how the tests are to be performed. This class contains the information needed to send the main test parameters to the nodes

**Date:**

7 December 2006, 15:41

**Author:**

Mattias Bergström  
Johan Högberg

### **E.23.2 Public Member Functions**

- **TestSpecification (Document newDoc)**  
Creates a new instance of TestSpecification.

- **void setDocument (Document newDoc)**  
Sets the DOM object to be used.
- **Document getDocument ()**  
Returns the DOM object used.
- **MainTestSpecification getMainTestSpecification ()**  
Returns the main test specification.
- **TestErrorSpecification[] getTestErrorSpecifications ()**  
Returns the test error specification.
- **TestErrorSpecification[] getTestErrorSpecifications (int nodeNr)**  
Returns test error specifications for a specific node.
- **void createMainTestParameters ()**  
Parses and creates main test parameters from the source XML file.
- **void createTestErrorSpecification ()**  
Parses and creates test error spec parameters from the source XML file.

## ***E.24 src/UploadFileUI.c File Reference***

### **E.24.1 Details Description**

GUI for simultaneous upload files to the nodes. This functionality is not yet finished. The actual sending of an s19 file to the nodes needs to be implemented

**Date:**

30 November 2006, 17:45

**Author:**

Johan Högberg  
Mattias Bergström

### **E.24.2 Public Member Functions**

- **UploadFileUI (String title, JTextArea hostWindowTextArea, boolean commPorts[], MessageQueue commMessageQueue[])**  
Creates new form UploadFileUI.
- **void actionPerformed (ActionEvent e)**  
Catches action events.

### **E.24.2 Package Attributes**

- **MessageQueue commMessageQueue\_ []**

To be able to send messages to the comm workers.

- **boolean commPorts\_ []**  
Indicate which com port are in use.



## Appendix F

### Membership Simulator API documentation

#### *F.1 Introduction*

#### *F.2 src/errorSimulation.c File Reference*

##### **F.2.1 Details Description**

All functions relating to the error simulation. Simulates Incoming Link Failure, Outgoing Link Failure as well as Node Silent failure.

**Date:**

2007-01-23

**Author:**

Johan Högberg

##### **F.2.2 Functions**

- **void clearErrorSimulationStructs ()**  
Clear all the error structs.
- **int ilfDuringCurrentPhase (int nodeNr, int cycleNr, int phaseNr)**  
Function that checks whether ILF occurs.
- **int olfDuringCurrentPhase (int nodeNr, int cycleNr, int phaseNr)**  
Function that checks whether OLF occurs.
- **int nsDuringCurrentPhase (int nodeNr, int cycleNr, int phaseNr)**  
Function that checks whether NS occurs.

##### **F.2.3 Variables**

- **errorSimulationStruct ilfStruct [N\_NODES][MAX\_ERRORS]**  
Contains all the ILF (incoming link failure) errors to be simulated.
- **errorSimulationStruct olfStruct [N\_NODES][MAX\_ERRORS]**  
Contains all the OLF (outgoing link failure) errors to be simulated.
- **errorSimulationStruct nsStruct [N\_NODES][MAX\_ERRORS]**  
Contains all the NS (node silent) errors to be simulated.

## ***F.3 src/main.c File Reference***

### **F.3.1 Details Description**

This is a simulated "network" environment on which the membership protocol is run. It is possible to introduce errors, which are then simulated, into the environment.

**Date:**

2007-01-24

**Author:**

Johan Högberg

### **F.3.2 Functions**

- **int main (int argc, char \*\*argv)**  
This is the main function performs all the tests.

### **F.3.3 Variables**

- **short cycle\_counter**  
keeps track of the current cycle
- **FILE \* filePtr [5]**  
Pointers to output files.
- **errorSimulationStruct ilfStruct [N\_NODES][MAX\_ERRORS]**  
Contains all the ILF (incoming link failure) errors to be simulated.
- **errorSimulationStruct olfStruct [N\_NODES][MAX\_ERRORS]**  
Contains all the OLF (outgoing link failure) errors to be simulated.
- **errorSimulationStruct nsStruct [N\_NODES][MAX\_ERRORS]**  
Contains all the NS (node silent) errors to be simulated.

### **F.3.4 Defines**

- **#define OUTPUT\_FILENAME "output/test11\_node"**  
This is the name of the output file minus the suffix.

## ***F.4 src/membership.c File Reference***

### **F.4.1 Details Description**

Contains the membership voting process done in each node in the cluster.

**Date:**

2006-12-01 14:16

**Author:**

Mattias Bergström Johan Högberg

### F.4.2 Functions

- **void clearSimulatedBus ()**  
Clear the simulated FlexRay bus of data.
- **void initializeMembership ()**  
Initializes the membership functions.
- **void setStaticMessageNodeStatus (short status)**  
Set the status (heartbeat/joining-node byte) for the current message.
- **void setStaticMessageCycleNumber (short currentCycleCounter)**  
Set the cycle counter for the current message.
- **void sendStaticMessage (int currentNodeNr)**  
Send the static message.
- **int receiveStaticMessage (int nodeNr)**  
Receive a static message from a specified node.
- **int getReceivedStaticMessageStatusByte ()**  
Get the status (heartbeat/joining-node byte) for the received static message.
- **void sendDynamicMessage (int currentNodeNr)**  
Send dynamic message.
- **int receiveDynamicMessage (int nodeNr)**  
Receive dynamic message from a specified node.
- **void membershipVoting (set \*votedMemberNodesSet, set \*undefinedNodesSet, set \*opinions, int umin)**  
  
Controls the voting process for each process the function receives an array of opinion sets and decides for each node if it is a member or not.
- **void fdSendMessage (int currentNodeNr)**  
Send the heartbeat/joining node data.
- **void fdDetection (int currentNodeNr)**  
Receive heartbeat messages and create opinion vector.

- **void mcSendMessage (int currentNodeNr)**  
Send opinion message in the Membership Communication phase.
- **void mcReceptionPhase (int currentNodeNr)**  
Broadcasts its opinion based on the heartbeats received in the failure detection phase. Upon reception of opinion sets from the other nodes the node updates its collection of opinion sets if the sending node is considered a member.
- **void membershipDecision (int currentNodeNr)**  
Uses the collection of opinion sets to perform a majority voting and obtain a "global" membership agreement. The function then checks a number of special cases and halts the node if one or more of these cases are true.
- **void membership ()**  
Calls the different parts of the membership protocol.
- **void outputStatsToFile (int currentNodeNr)**  
Write statistics for the current cycle to the output files.

#### F.4.3 Variables

- **short cycle\_counter**  
keeps track of the current cycle
- **FILE \* filePtr [N\_NODES]**  
Pointers to output files.
- **short halted [N\_NODES]**  
Indicate if the current node is halted or not.
- **int haltedCycleCounter [N\_NODES]**  
Contains the number of cycles since we got booted out of the membership. If the value of haltedCycleCounter reaches the value REINTEGRATION\_THRESHOLD, it then attempts to reintegrate again.
- **set membershipSet [N\_NODES]**  
To store the local membership.
- **set joiningNodesSet [N\_NODES]**  
To store the nodes that are attempting to join the membership in this cycle.
- **staticSlotCommunicationStruct staticBusStruct [N\_NODES]**  
Structure used for the simulation runs. Contains all the data "sent" in the static slots.

- **staticSlotCommunicationStruct staticSendStruct**  
Structure used when the node is sending data in the static slot.
- **staticSlotCommunicationStruct staticReceiveStruct**  
Structure used when the node is receiving data in the static slot.
- **dynamicSlotCommunicationStruct dynamicBusStruct [N\_NODES]**  
Structure used in the simulation runs. Contains all the data "sent" in the dynamic part of the cycle.
- **dynamicSlotCommunicationStruct dynamicSendStruct**  
Structure used when the node is sending data in the dynamic slot.
- **dynamicSlotCommunicationStruct dynamicReceiveStruct**  
Structure used when the node is receiving data in the dynamic slot.
- **set opinionSet [N\_NODES]**  
To store the local opinion for this cycle.
- **set opinionSets [N\_NODES][N\_NODES]**  
Array of sets containing the opinions of the various nodes.
- **set allNodesSet**  
Set that contains all the nodes in the cluster.
- **multiset nodesAliveSet [N\_NODES]**  
To store the number of nodes that the various nodes have found to be alive in the current cycle.
- **int nNodesAlive [N\_NODES]**  
To store the number nodes that the current node has found to be alive in the current cycle.
- **int joinRequest [N\_NODES]**  
indicates whether we want to join the membership or not