A distributed FlexRay-based research platform

Author	Mattias Pettersson, Petter Uvesten

Document Id	012
Date	3 mars 2006
Availability	Public
Status	Final

CHALMERS

A distributed FlexRay-based research platform

Mattias Pettersson
Petter Uvesten

Master's Thesis

Computer Science and Engineering Program

Chalmers university of technology
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2005

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

© Mattias Pettersson and Petter Uvesten, Göteborg 2006.

Abstract

This paper describes the implementation of a FlexRay-based microcontroller cluster for research and demonstration purposes within the CEDES project. The cluster has been built using prototype hardware from the GAST project, more specifically the GAST G2 MPC565-based controller board and the MFR4200-based real-time communication controller board. Initially, a development environment based on GNU compilers and the Eclipse IDE was compiled, this environment was then used for the production of well-documented FlexRay drivers for the MPC565 and MFR4200. Finally, the completed nodes were used to develop an example cluster which will serve as a basis for further work on the CEDES demonstrator system. The example cluster uses a simple interrupt-driven non-preemptive RTOS providing membership functionality for the running applications in the distributed system. A number of membership protocols were proposed and their possible implementation in a FlexRay environment is discussed. A short background is presented and a description of hardware problems and solutions to those is included.

Keywords:

CEDES, GAST, FlexRay, membership algorithms, real-time communications, MFR4200, embedded development, distributed embedded systems

Sammanfattning

Denna rapport beskriver utvecklingen av ett FlexRay-baserat kluster av processornoder. Detta kluster skall användas i forsknings- och demonstrationssyfte inom CEDES-projektet. Klustet byggdes med protoyphårdvara från GAST-projektet, specifikt GAST G2, ett MPC565-baserat processorkort, och GASTs MFR4200-baserade kommunikationskort. Först sammanställdes en utvecklingsmiljö baserad på GNU-kompilatorer och Eclipse. Denna miljö användes sedan för utveckling av väldokumenterade FlexRay-drivrutiner för MPC565 och MFR4200. Slutligen användes de färdiga noderna för att utveckla ett exempelkluster som skall fungera som grund för det fortsatta arbetet med CEDES demonstratorsystem. Exempelklustret använder ett grundläggande realtidsoperativssystem baserat på en avbrottsstyrd processkörningsmodell och som innehåller medlemskapsfunktioner för applikationerna i systemet. Ett antal medlemskapsprotokoll föreslås, och deras eventuella användning i en FlexRay-miljö diskuteras. En kort bakgrund till projektet presenteras och beskrivningar av hårdvaruproblem och lösningar till dessa finns inkluderade i rapporten.

Nyckelord:

CEDES, GAST, FlexRay, medlemskapsalgoritmer, realtidskommunikation, MFR4200, utveckling för inbyggda system, distribuerade inbyggda system

Preface

This master thesis project was done as a part of the CEDES [2] project. The aim of the CEDES project is to perform research in the area of Cost Efficient Dependable Electronics Systems, with a focus on their use the automotive industry.

To demonstrate the work done in CEDES to a wider audience, a demonstrator of the technology is to be built. This demonstrator will show the concept of dependable electronics by incorporating fault-tolerant architechture and system design.

SP - the Swedish National Testing and Research Institute, a partner in the CEDES project, posted a job advertisement for two undergraduate students to perform the first step in building the demonstrator as a master thesis project.

Since we felt the project would be both challenging and interesting we applied for it, and got accepted.

The work done in this project has been performed at the Open Arena at Lindholmen Science Park, Göteborg, Sweden.

The authors would like to thank the following people for their help and support during the project.

- Roger Johansson - Our examiner at Chalmers, for his invaluable tips and extensive knowledge of the hardware used and of embedded systems development
- Carl Bergenhem - Our project supervisor at SP, for giving us the opportunity to do this thesis project and for interesting discussions regarding membership algorithms.
- Jimmy Myhrman and Nikola Vorkapic - for excellent cooperation during the development of the FlexRay drivers.

Contents

1 Introduction 1

2 Purpose and thesis objectives 5

3 Project overview 7

3.1 Development environment 7

3.2 Node development 8

3.3 Cluster development 10

4 Background 11

4.1 Project Hardware 11

4.2 Communication framework 13

4.3 Dependable electronic systems 16

4.4 Membership agreement 20

5 Implementation 25

5.1 Hardware issues 25

5.2 Node Development 26

5.3 Cluster Development 30

5.4 Utilities and example applications 32

6 Results and discussion **35**

6.1 Development tools 35

6.2 The FlexRay node 39

6.3 Membership agreement in a FlexRay network 41

6.4 Thesis demonstrator system 44

References **49**

List of figures **51**

List of tables **53**

Index **55**

Glossary **57**

A The GAST Eclipse Environment (GEE) for Windows **59**

A.1 Introduction 59

A.2 Installation instructions 60

A.3 Getting started 61

A.4 The GAST connector plugin 62

B FlexRay API documentation **65**

B.1 FlexRay driver package 65

B.2 src/chi_prot_cfg.c File Reference 65

B.3 src/mfr_mb.c File Reference 98

B.4 src/mfr_mb_fifo.c File Reference 106

API Index 115

Chapter 1

Introduction

Since the introduction of electronic fuel injection in automobiles, there has been a trend towards more and more electronic systems in cars. Today there can be 70 ECUs - Electronic Control Units in a typical modern series car. Most of these ECUs are interconnected using some sort of communication interface. Presently the main communication interface used in cars is CAN [23] , an event-triggered communication protocol.

In the last years the ultimate goal of the electronics development in the automotive industry has been x-by-wire, taking the name from the aircraft industry's fly-by-wire [22] concept.

X-by-wire includes technologies such as brake-by-wire and steer-by-wire. Replacing the hydraulic, pneumatic and mechanical systems in use today would lead to dramatic savings in production cost and savings in weight, but also to other advantages, such as easier replacing of faulty parts and easier diagnostics.

However, the event-triggered communications protocols in use today are poorly suited to x-by-wire systems. For an x-by-wire system to be considered for real use in series cars all the basic infrastructure (communication protocols, ECUs etc.) must be constructed with a high degree of dependability, reliability (see Storey [20])and predictability. When more and more functions move from the mechanical/hydraulical systems to electrical systems there is also a requirement for more bandwidth in the communication interfaces.

In response to these demands, the FlexRay consortium [5] was formed in 2000 by

BMW, Bosch, Philips, Freescale and other actors in the microprocessor and automotive industries.

FlexRay is a time-triggered communications protocol with 2x10 Mbit/s bandwidth. This addresses two of the problems with the protocols in use today, namely their event-triggered behavior and low bandwidth. The FlexRay protocol will be further discussed in section 4.2.2 of this report.

While the potential savings are enormous when x-by-wire systems are in use, the technology used to build today's existing fly-by-wire systems in aeroplanes is too costly to be used by the automotive industry in series production. Therefore new and considerably less expensive technologies have to be developed. These technologies, however, must still attain a very high level of reliability for x-by-wire to be a viable alternative to today's systems.

The CEDES (Cost Efficient Dependable Systems) project [2] at Chalmers aims to research technologies that will provide dependable and reliable electronics systems that are affordable and possible to use in series production in the automotive industry.

GAST [8], another project led by Chalmers, has developed hardware in the form of processor boards and communication controller boards which can be used to simulate real ECUs and networks of ECUs in vehicles today. These boards are built using standard components and open specifications, thereby providing an excellent development and research platform.

The goal of the thesis project described in this report was to use the hardware produced by the GAST project to build a cluster of ECUs which communicates via FlexRay. This cluster will be used as base platform for coming projects within CEDES, and be the basis for the future CEDES demonstrator.

This task was non-trivial, since the processor board and communications board were untested together at the start of the project, a driver package had to be developed to be able to use the FlexRay communications controller board. In addition to this we wrote a small RTOS - Real Time Operating System to run applications on the processor board. As a small step towards real reliability in this cluster, we studied a number of different membership algorithms (see section 4.4 for a definition of membership in our system). Following this, we designed a rudimentary membership protocol which integrates well with FlexRay and implemented this as a service in our RTOS.

Since this is a rather large project, we had to set up some delimitations. These are:

- The hardware used is only the G2 board and the FlexRay board from the GAST projects. Other hardware or protocols have not been considered. Specifically, we have only worked with the main MPC565 processor on the G2 board, not the help processor HCS12.
- We have worked with a simple FlexRay cycle and not tested long cycles with large number of slots.
- The RTOS we have developed is rudimentary. A fully preemptive RTOS was out of the scope for this project, and thus was not considered.
- The membership algorithm we proposed has not been proved to have all necessary properties for a real-life error situation.

Chapter 2

Purpose and thesis objectives

This thesis was done for SP (Swedish Testing and Research Institute) as part of their participation in the CEDES[2] research project. This is one of the first in a series of thesis projects within CEDES that uses the GAST[8] hardware to build an experiment system. At the initial stage three main objectives were set for the thesis. These were:

1. Set up an integrated development environment for GAST hardware based upon the GNU-tools and an open source environment like Eclipse. Collect available code for the G2 platform and adapt it to this environment.
2. Connect the available prototype boards for G2 and FlexRay and create a working node. Provide solutions to any hardware issues found on any of these prototypes.
3. Write a software driver package to configure and operate a FlexRay cluster with at least four nodes. Provide an API and documentation to make it possible to quickly write applications that run on this cluster using FlexRay for communication.

The purpose of the platform will be to build a demonstrator system for CEDES. This demonstrator will in turn implement a brake by wire system using a braking algorithm provided by Volvo and connect sensors, actuators and a simulator to the cluster. The system will be used to study aspects of dependable systems and cost efficient implementations of these. Especially the area of implementing membership agreement in a real system will be studied. Agreement is needed by the brake control loop to be able to decide on the correct braking algorithm.

Considering the future use of the project two additional objectives were added. They were to be implemented if there was time and available hardware.

- Write a few sample applications using the FlexRay software drivers to demonstrate the use of the system. A simple producer/consumer pair using a sensor and an actuator was deemed suitable.
- Implement a simple membership protocol to get a first impression of practical constraints and possibilities when working with a FlexRay system.

Chapter 3

Project overview

This chapter gives a quick overview of the project and a quick summary to the different phases. The main purpose was to establish a distributed platform for further simulation and research. At the start of the project we had access to a prototype host application board from the GAST project with a MPC565 microprocessor and a few prototype FlexRay communication boards based on the Freescale MFR4200 chip. These had not been tested together and a very limited amount of software and drivers was available. To structure the project we identfied three main phases, which are described in the following sections.

3.1 Development environment

Initially we inherited some simple development tools from the GAST-project along with source code and documentation. These tools were not consistent and while well suited for some of the tasks we needed to perform they had limited support for others. A general development environment with integrated functionality was desired. As a first task on the project we decided to evaluate what tools were available and try to build an environment from them. The specifications which were to be met by the development platform were:

- Strong support for C-development with indexing, project management and built in functions.
- Ability to integrate cross-compilers for both MPC565 and HCS12 processors.

- Support for version control.
- Preferably direct access to serial communication to be able to download and run programs on the fly.
- Ability to run on win32-platform.
- Relatively simple installation.
- Standardized, open and configurable for other users to make sure it can be used by future projects.

The open source environment Eclipse was the prime candidate and after some testing of its C/C++ mode and advanced plug-in system we decided to build our environment on this platform. We also needed to have support for the cross-compilers on the win32 platform and for this purpose we decided on mingw, minimalist GNU for Windows. All compiler tools were installed under mingw. Using the mingw system libraries with eclipse we had a full featured C-environment with cross-compilation capability. For version control we decided upon subversion, a system with an available plugin for eclipse. Other version control systems are available and eclipse has support for a wide variety. This met most of the requirements and only the question about a serial interface remained.
We were unable to find a plug-in for eclipse that easily integrated serial communication so we decided to write our own. The advanced support for plug-in development helped us a lot and we could also include basic menus and mouse functionality. The rudimentary plug-in is capable of establishing serial communication at different baud rates and ports, it has a basic command line interface with the serial connection and it is also possible to download s-records through a simple menu system. Further documentation and installation instructions for the development environment can be found in appendix A.

3.2 Node development

Having established the set of tools needed for our project we were ready to proceed with the actual development. Our primary goal was to create a working network node consisting of a micro-controller board and a FlexRay communication controller board. The communication between these cards is handled over a standard backplane bus and the micro-controller addresses the communication controller register as external

memory. After extensive study of the documentation and data sheets of the involved chips we decided on the following working order:

- Establish initial communication between the two controllers, i.e. read and write operations
- Write driver software to write FlexRay configuration to communication controller registers
- Download configuration to communication controller and check if controller can transmit wakeup
- Write driver software to handle communication buffers and send/transmit operations
- Connect two communication controllers over a FlexRay bus and check if they sync and exchange information

This workflow served us well and it proved very useful to take this step by step approach. Our prototype cards had several defects that we were able to identify relatively early. For all problems workarounds were found for the prototypes and changes in the specification were made for the production card. The most significant problems we encountered with the communication controller card was that the crystal oscillator was not stable at 40 MHz and a bus driver enable signal that was not inverted although it should be. The first problem was solved by replacing the oscillator and the second by replacing a logic gate on the prototype board. Some other minor issues were also found and have been rectified in the final design.

The software drivers were written in close cooperation with other thesis projects using the GAST hardware. The focus when writing the software was to create an API that was well documented, easy to use and utilized the FlexRay protocol parameters in its implementation rather that the chip-specific parameters used by our controller. The drivers are given in two versions. The first one includes all configuration management in the API and requires the programmer to manually input the global configuration parameters. The second version is more lightweight and uses an external configuration tool written by two thesis project workers at QRtech. This is the version we have used in our cluster as it provides better configuration management and less memory usage. The first version is better for future development of the drivers since the code is more readable and there are no external tools that need to be modified as well.

3.3 Cluster development

The final micro-controller network was developed with a lot of help from the knowledge gained in the previous phases of the thesis project. The aim of the demonstrator is to demonstrate a working FlexRay-based system with some kind of fault tolerance. The platform should also provide example applications that highlights the use of the FlexRay driver software and can serve as a base for future projects using our cluster.

The design decisions taken here influence the kind of research that can be done on the system and we worked in close cooperation with our supervisor Carl Bergenhem during this phase.

Chapter 4

Background

4.1 Project Hardware

4.1.1 The GAST project

The hardware that have been used in this thesis project originates from a project called GAST - General Application development boards for Safety critical Time triggered systems [8]. The GAST project develops open source platforms of hardware and software for distributed embedded real-time systems. There are five types of development boards originating from the project. Three real-time communication boards implementing TTCAN, TTP/C and FlexRay. The two processor boards are called G1 and G2 where G1 is based on the HCS12 processor while the G2 is a dual processor board in a main/monitor configuration. The G2 processors are the Motorola MPC565 as main processor and the HCS12 as the monitor.

4.1.2 The MPC565 processor

Our main processor platform is the GAST G2 board. Most of our work has been done on a prototype release version 2 of the G2. Since the purpose was to develop a working FlexRay cluster we have focused on the main processor and developed our software set for this target. The MPC565 [6] is a PowerPC/RISC microcontroller of automotive class C. The main features of the processor are summarized in table 4.1.

The Motorola MPC 565	
Clock frequency	40 MHz
memory RAM	36 kB
memory Flash (UC3F)	1 MB
Time processor units	3
Interfaces	3xCAN, Serial, J1850,NEXUS, JTAG, modular IO and A/D-channels

Table 4.1: Some of the key features of the MPC565 microcontroller

4.1.3 The MFR4200 FlexRay communications controller

The processor boards in our network cluster are connected using the FlexRay communications protocol. The real-time communication is handled by a communications board using the Freescale MFR4200[7] communications controller. This is the first commercially available FlexRay controller and the main features are summarized in table 4.2. All configuration and handling of the MFR-chip is done from the host processor, the MPC565. For more information about the specifics of the FlexRay protocol see section 4.2.2

The Freescale MFR4200	
Clock frequency	40 MHz
max bit rate	10 Mbps/channel
message buffers	59
message buffer size	32 bytes
max frame size	254 bytes (uses padding)
Host interfaces	HCS12 and asynchronous
Message filtering, FIFO, error handling and interrupt source	

Table 4.2: Some of the key features of the MFR4200 communications controller

4.2 Communication framework

4.2.1 Time triggered systems

Time triggered communication is one way to solve the problem of network access in a distributed system. A time triggered system uses designated time slots of predefined duration to control each nodes access to the channel. This is very efficient in high load situations since every node is guaranteed to send in their allotted time slots. The other popular approach is to use an event triggered architecture which allows a node to try to send as soon as there is a message ready. This is then often combined with some kind of collision avoidance or collision detection and a random back-off time to handle attempts of simultaneous transmission. The event triggered approach is effective in low to medium load situations and allows fast message transport when there are few collisions. In a real time system though the event triggered approach has several drawbacks even in low load situations since there is no guaranteed delivery time for a message. In a distributed system it is often of vital importance to be able to tell whether a message was correctly received, or not received, after a certain time. The main challenge when building a time triggered system is to solve the problem of a global time. Since each node has allotted time slots it is vital that all nodes agree upon the start time of these slots. Some kind of synchronizing must be done on the network level to prevent individual clock drift. For high speed networks the physical size of the network plays an important role since signal propagation delays have to be taken into account.

4.2.2 The FlexRay protocol

FlexRay [4] is a time triggered communications protocol primarily designed for automotive applications. It allows for networks with a maximum of 64 nodes communicating over an electrical or optical physical layer. The network topology can be either a bus, an active star or combinations thereof. Addressing is done using frame id:s and a communication cycle consists of a maximum of 2048 frames divided into two segments. The static segment holds a maximum of 1024 frames with guaranteed transmission times in a standard TDMA schedule. The slot times in the static segment are reserved for the frames even if there is no transmission. The dynamic segment allows for transmission of event type messages and has both guaranteed and best-effort service. There is also a symbol window and a network idle time with the latter used

for clock synchronization calculations. A detailed figure of a communication cycle in FlexRay can be seen in fig 4.1.

A node is assigned certain frame id:s during the design phase of a system and is allowed to transmit when the slot counter for the communication cycle reaches the slot number corresponding to the frame id. The media access scheme is somewhat different in the dynamic segment compared to the static segment. In the static segment the slot counters are incremented simultaneously on both channels at the end of each slot. The slot duration is the same regardless of a frame being transmitted in the slot or not. In the dynamic segment a mini-slot scheme is used and the channel is available to the next frame id as soon as the corresponding mini-slot for the previous id has ended if there is no transmission. If a frame is transmitted in a mini-slot then the slot counter will not be incremented until the whole frame is transmitted even though this takes several mini-slots. Depending on what is transmitted on each channel the slot counters are allowed to be incremented independently while in the dynamic segment.

Communication on the physical layer in FlexRay is handled by a communication controller that also takes care of clock synchronization. FlexRay uses synchronization frames to calculate the global time but no global time is transmitted on the network. All nodes register the arrival times of the sync frames and then performs an algorithm to adjust their local clocks accordingly. The precision of the global time state varies with the actual configuration but has a worst case of about 10 μs mostly due to propagation delay. The communication controller is able to provide this time to the host processor and it is therefore possible to have an agreement of the global time in all nodes on the network.

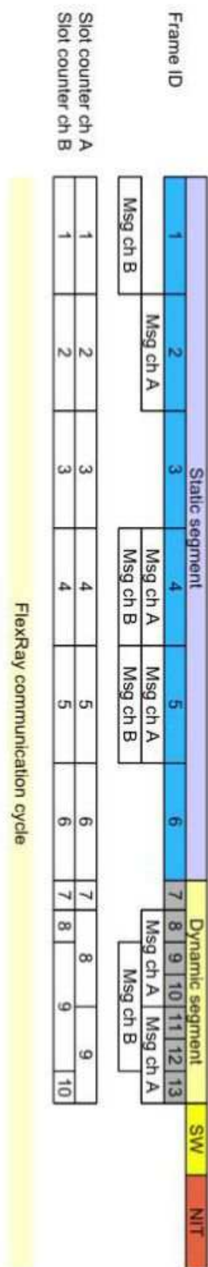

Figure 4.1: Example of a FlexRay communication cycle using both static and dynamic segment

4.3 Dependable electronic systems

4.3.1 Fault tolerance

By dependability we refer to ”a property of a system that justifies placing one’s reliance on it” This is a term quantifying a collection of different factors such as reliability and availability. One important aspect of a dependable system is its fault tolerance, i.e. its ability to perform its specified function in the presence of faults. For a thorough discussion of the definition of faults and fault types see Storey [20]. In the scope of this thesis we will concentrate on the ways of achieving fault tolerance and more specifically different ways to implement redundancy. A distinction of different forms of redundancy is made by Storey [20] as

Hardware redundancy The use of hardware in addition to that which would be required to implement the system in the absence of faults, with the aim of detecting and tolerating faults.

Software redundancy The use of software in addition to that which would be required to implement the system in the absence of faults, with the aim of detecting and tolerating faults.

Information redundancy The use of information in addition to that which would be required to implement the system in the absence of faults, with the aim of detecting and tolerating faults. Examples include the use of parity bits, error detecting codes and checksums.

Temporal redundancy The use of time in addition to that which would be required to implement the system in the absence of faults, with the aim of detecting and tolerating faults. Examples include performing calculations twice and comparing the results.

There are several ways to utilize the different types of redundancy to handle faults. A common way of implementing hardware redundancy is to have a backup that can be used in case any of the main units should fail, e.g. the spare tyre of a car. This type of redundancy is called dynamic redundancy and requires some kind of fault detection that will switch operation to the backup. In electronic systems and especially real-time systems there is often a need for the system to be operating continuously. This means faults must be detected and handled immediately and this often requires redundancy configurations other than a simple backup.

One method is to use a voting system consisting of several units working in parallel and then performing a compare operation on their output and take the majority as the result. Using n units a voting system can tolerate $(n-1)/2$ faults. This is a kind of static redundancy where faults will be handled without any active fault detection mechanism. The drawback is the rapidly increasing amount of hardware needed to handle faults.

4.3.2 Reliability

The reliability of a system is the probability that it is functioning correctly for a given period of time. One common definition is to denote reliability over time by $R(t)$ and define it for a given number N of identical components that start operating at the same time as

$$R(t) = \frac{n(t)}{N} \tag{4.1}$$

where $n(t)$ is the number of components still operating correctly at time t. For a system consisting of different sets of redundant components one may use reliability models such as reliability diagrams to find the reliability of the entire system. The configuration of the system components, their individual reliability and the number of redundant units will all greatly influence the reliability, cost and performance of the system. When designing a dependable system this adds considerable complexity to finding the optimum configuration.

4.3.3 Dependable systems in the automotive industry

Overview

In today's cars most safety-critical functions are handled by mechanical, hydraulic and electro- mechanical systems. Fault tolerance is achieved through hardware redundancy, e.g., two separate braking fluid lines. These systems are proven safe and have a long record of operation. However these systems are static and cannot adapt to things like erratic driver behavior. The advance in electronics and the flexibility of electronic systems have seen the introduction of more and more electronic components into vehicles such as anti-lock braking, active suspension, collision avoidance etc. Until now they have mostly been a complement to the mechanical systems, and not a replacement for them, adding to rapidly increased cost of development and production. The vision is however to be able to replace mechanical systems by electronics

and electro-mechanical. The purpose of this is to achieve safer, lighter, cheaper and better vehicles. For this to happen the electronic systems must prove to be at least as reliable as their mechanical predecessors.

As mentioned in the previous section one way to achieve reliability is to use hardware redundancy to reach appropriate fault tolerance and overall system reliability. This approach is used successfully in the aviation industry for fly-by-wire [22] systems. The drawback is that more hardware is costly and the automotive industry is very cost sensitive so to employ these solutions the cost issue must be addressed in some way. One method studied within the CEDES-project [2], of which this thesis is a part, is the ability to tolerate hardware faults through the use of software. This becomes especially interesting when several separate hardware/software systems can be integrated on one hardware platform cutting the number of electronic controllers needed drastically while still maintaining some hardware redundancy. The challenge is in making sure that the software systems do not interfere with each other in non-permitted ways and that the overall dependability of the systems can be guaranteed.

By wire systems

X-by-wire is a term used for replacing mechanical systems with electronics, e.g., brake by wire means electronic braking without direct coupling between brake pedal and brakes. A braking system is safety- critical and it must be 100% reliable under normal operation. A typical system would have some kind of central controller unit and a sensor/actuator unit at each wheel. The units would be interconnected by a bus. To achieve reliability the central unit could use a backup system and a failure of a wheel unit could be handled by applying a different braking algorithm using three wheels instead of four. This kind of system works well in a standalone configuration and similar systems are implemented in avionics. To add functionality of steer by wire another similar system may be introduced with backup-configured central controller and bus access to the wheel units. An example of what this would look like can be seen in figure 4.2.

This kind of system only includes a couple of functions but the fault tolerance requirements makes the system use a lot of hardware. We have five advanced micro-controllers for implementing controller functions and four basic micro-controllers handling the wheel units. Incorporating systems in this way may be called a federated design. The systems are clearly separated and loosely coupled and the safety of each of the systems is ensured through the use of separate backups. The drawbacks are

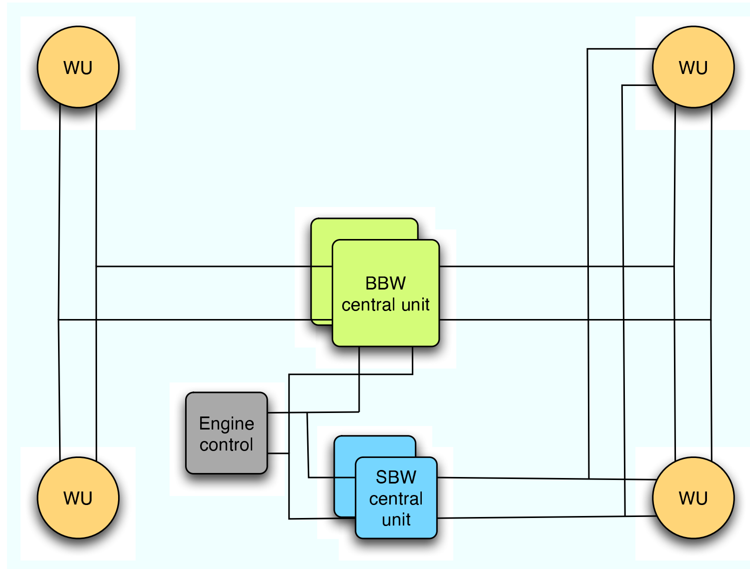

Figure 4.2: Example of a federated design for x-by-wire. All control functions have their own dedicated hardware and are loosely coupled through bus systems. Fault tolerance is achieved by redundant central controllers and bus lines. Note the amount of hardware needed for each function.

clearly that the amount of hardware needed grows rapidly when more and more systems are needed. This is tolerable to a certain degree but there is a physical limit to how much hardware that can be squeezed together in a vehicle for reasons of electromagnetic interference, power consumption, cabling weight and cost.

An alternative is to use the kind of system design that this thesis aims to develop; an integrated design where software is responsible for a lot of the fault tolerance capabilities and redundancy. A system that performs the same functions as the one described above may be implemented as shown in figure 4.3. Here we have three advanced micro-controllers and two basic micro-controllers, a significant reduction in hardware. Fault-tolerance and redundancy is implemented through the ability to run all control functions on each of the three advanced micro-controllers and simply move functions to another controller should the current one fail. This kind of integrated approach adds a lot of complexity on the software level though and is the object of intense interest from the automotive industry in projects like Autosar [1]. The challenge for integrated systems in safety-critical applications is the issue of reliability and how to prove that a complex system with several control functions achieves a certain reliability. This thesis is meant only to provide the tools for some of this research but a few points can be made even at this level.

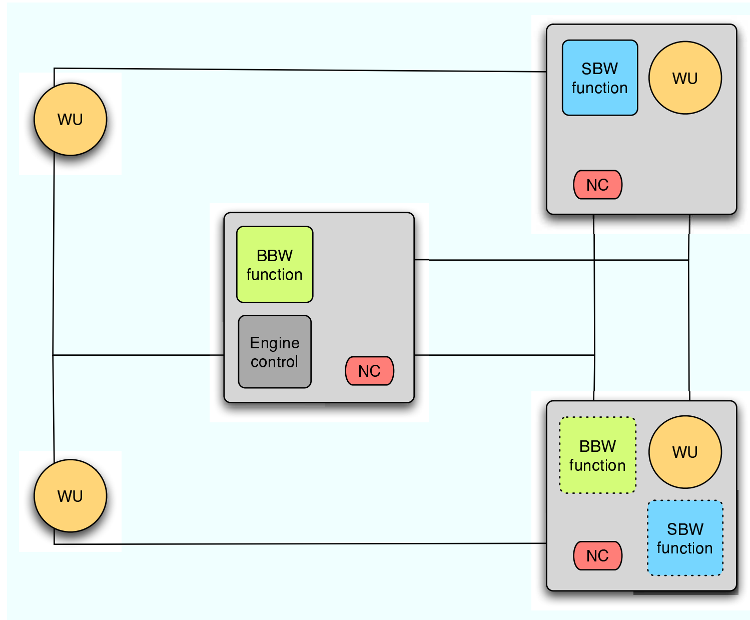

Figure 4.3: Example of an integrated design for x-by-wire. A common hardware platform is used by all the control functions. Fault tolerance is achieved by having backup processes on different nodes. The system is tightly coupled and must be able to assure that control functions cannot interfere in non-permitted ways.

One reliability property we can show using this system is how fast a fault can be detected and handled. Another is the level of confidence with which an entity in the system can know which of the other entities functions correctly by using some kind of membership algorithm. The use of a deterministic network protocol like FlexRay makes this possible even in a real-time system.

4.4 Membership agreement

4.4.1 Definitions

Before discussing the concept of membership there is a need to make some definitions. In the context of this thesis we refer to a number of different properties of our system such as nodes, entities and host processor. We have tried to follow conventions when introducing these terms and our aim is that it should be clear from the context what we refer to. Even so a more strict set of definitions is needed for the theoretical part.

Beginning at the system level we refer to our demonstrator *system* as the combina-

tion of all hardware and software that makes up our distributed platform. The system consists of four *nodes*, each one consisting of a communication controller card, a host processor card and all software running on the host processor. The *communication controller* is an integrated circuit in charge of handling the FlexRay communications system and in our system we use the Freescale MFR4200 chip. The *host processor* is the main processing unit of each node and consists of a MPC565 chip, a power-pc processor from Freescale. The *communication channel* is the inter-node connections through which signals are conveyed for the purpose of communication. The communication controller is connected to the communication channel via a *bus driver* that is an electronic component consisting of a transmitter and a receiver.

An *application* is a piece of software running on the host processor with a specific purpose, for example reading and transmitting a sensor value. Applications are considered independent of each other and interact only by sending messages on the communication channels. A *service* is a piece of software running on the host processor of each node providing specific functionality for the applications running on the same node. An *entity* is a part of the system considered for membership. There are three types of entities in our system: applications, hardware units and entire nodes.

By *membership* we refer to a view of the state of the entire system regarding entities functioning correctly or not. A correctly functioning entity is said to be a member and included in the membership. Node-level membership *agreement* implies that all the nodes in the system have the same consistent view of the membership status of all entities in the system.

4.4.2 Protocol comparison

The existing and proposed protocols for vehicle communications handle membership differently. CAN provides a partial solution by its use of multicast acknowledgment messages but it will not detect a dead node by itself, some kind of activity from the application side is needed. TTP [13] on the other hand provides a solution based upon checksums and group membership that will guarantee that all nodes in your group have seen the same messages you have seen. If messages diverge in TTP it will result in groups splitting for a short period of time which means a faulty node is automatically excluded from the group membership. In FlexRay there is no built-in membership service and it is entirely up to the application to handle membership functionality. One method would be to let each distributed application handle its own

	A	B	C	D
A	5	1	5	5
B	1	5	1	1
C	5	1	5	5
D	5	1	5	5

Table 4.3: Data matrix collected at the nodes in membership protocol 1

membership but it may also be desirable to have a common service for all applications.

4.4.3 Some existing algorithms

Membership agreement is a property of concern for many distributed systems and has been well studied for communication protocols like TCP. The main issue for these kinds of protocols is the lack of a global time and no guaranteed message arrival. Algorithms for membership agreement therefore tend to be complicated, have timing properties unsuited for real time applications and require a lot of bandwidth overhead. With a time triggered communications protocol like FlexRay many of these problems can be solved since there is a global time base and known message arrival times. Several simple protocols were discussed as candidates in this project and brief descriptions of two of them follows.

Membership protocol 1

Table 4.3 depicts a data matrix that is collected in all nodes. Each row corresponds to a nodes view of the rest of the cluster. The data in the matrix can be in its simplest form a bit value that indicates that a node is member or not. A more advanced matrix may contain the (time-) slot in which a node was last heard from. In the example in Table 4.3, all nodes have the common view that B has not sent anything (valid) since slot 1. Since a majority of nodes in the cluster shares the same view, B can be excluded from membership and allowed to attempt to reintegrate. Exclusion from the membership does not necessarily imply that B cannot participate in the communication. The results from B can still be valid but should have lower trust than results from a node within the group. An approach to this could be to let the application decide weather to trust a non-member node.

In this simple case each node simply sends a message with its view of the cluster, corresponding to a row in the table, at regular intervals. This causes some bandwidth overhead, especially if the view needs to be updated often.

Membership protocol 2

This protocol is described and implemented by Lönn in [15] and is based on the periodic group creation protocol suggested by Cristian [3]. This protocol is adapted to suit a time-triggered environment with inherently atomic broadcast capabilities. The protocol works by requesting that node j+1 through j+1+f acknowledge the transmission of node j and thus that node j is alive. This is done by adding f+1 bits to data from each node to handle f faults. A node is considered to be member as long as at least one other node correctly received the message. The advantage is the minimal overhead and fast agreement that the protocol adds.

Chapter 5

Implementation

5.1 Hardware issues

The first step in our work was to make sure that the hardware used (see section 4.1 for an overview) worked together in a reliable way. When we started the project the FlexRay card and the G2 card were not tested together.

The communication from the G2 to the FlexRay card is done via a backplane bus with 16 bits addressing. The MPC565 controls the MFR4200 by referencing it as external memory, writing to and reading from its registers.

Using the default settings in our debugger software we couldn't get this to work. After some hardware debugging using an oscilloscope we found that the main problem was that a WE signal didn't get transmitted onto the backplane bus. This problem took some time for us to solve, partly since we did not have any experience testing prototype hardware, partly since there was ambiguity in the schematics of the G2 board, which indicated that the signal was transmitted.

Having solved this problem, we faced timing issues and voltage differences between the MFR and MPC, luckily these could be solved in a fairly short order by using the MPC:s built in configuration registers, which provides options for it to adjust timings and protect itself from high voltage when accessing an external memory.

As we now had reliable communications to the controller, and could configure it using direct memory writes to it, we decided to program a simple configuration to

the controller to find out if it really worked the way it was supposed to. We did this by setting up an extremely simple FlexRay cycle and sending wakeup symbols to the bus. The wakeup symbol doesn't have much practical value in the MFR4200, since it can only send and not receive the symbol. As a startup test, however, it is perfect.

When we had verified that the hardware worked, we moved on to node development and writing the drivers.

5.2 Node Development

5.2.1 Programming environment and tools

While we chose to do our main development and debugging using the firmware debugger produced by the GAST project (see [8]), we also used the built-in graphical debugger in eclipse in combination with a Macraigor wiggler and the OCDRemote program. There is an excellent tutorial by Jim Lynch freely available on the web, see [16]. This tutorial is aimed at development on a ARM-based board, but the ARM-specific settings can easily be changed to work with the MPC-based G2 card instead.

Note that using the Wiggler cable will put the MPC565 in serialized mode and makes all writes to development registers, such as ICTRL, non-functional. Also, the wiggler/remote debugger is also not suitable when using interrupts in your code. However, the debugger works very well for debugging ordinary programs.

5.2.2 FlexRay driver package

Considerations

When we started to think about how to implement our FlexRay drivers and API, we took a design decision to make them conform as closely as practically possible to the FlexRay Protocol Specification v. 2.1 [4].

Secondly, we wanted hardware-independence. Many of the users of the driver package will only want to understand the FlexRay protocol and not the intricasies of the MFR4200 controller [7]. There is also the possibility that the project would want

to change controller chip, and our driver package and API should be easily portable without changing any of the functions visible to the application programmer.

Thirdly, the drivers should be easy to use and the functions self-explanatory. Since many of the users will be students and people not very familiar with low-level C-programming, we decided to favour readability and clarity instead of terseness and (possibly) speed and memory footprint.

Initial draft

Since we wanted to achieve a high level of hardware independence and readability, we took a layered approach to developing the code, see figure 5.1.

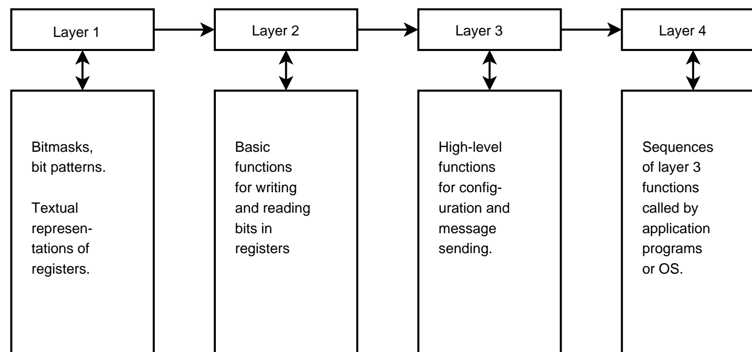

Figure 5.1: The layered design of the FlexRay driver package

Layer 1 is where we define structures and bitfields mapping the hardware addresses of the MFR4200 into textual names according to the MFR data sheet [7]. This made the work writing our functions in later stages much more efficient and less error prone.

Layer 2 contains simple C macros for writing and reading either entire registers or specified bits in the registers of the MFR4200.

Layer 3 is where the configuration and message sending/recieving functions are located. This is the layer which is planned to be hardware independent. The functions here are named in a way consistent with the FlexRay Specification v.2.1 [4].

This is the layer with wich the application programmer interacts. With this layer finished, a programmer need not have more than a basic understanding

of the message buffers on the MFR4200 to use the drivers to sucessfully run communication on a FlexRay network.

Note! Documentation of the layer 3 functions is available in appendix B.

Layer 4 is the domain of the application/operating system programmer. Though this is not a part of the FlexRay driver package per se, some example layer 4 code is included in the software distribution of the FlexRay driver package [18].

Development methodology

During development of these drivers, we worked in close cooperation with three other thesis workers working with similar projects using the GAST hardware. Since we all needed a FlexRay driver package, we felt that it would be redundant if each projects wrote their own drivers. By coordinating our efforts we could produce a common driver package, and we could all help developing and testing. This made the development of the drivers very efficient.

Early on, after the design phase, we decided on how to split the workload and decided on a commmon version control system to use. Our choice in this case was to set up a Subversion [21] server which all of the members had access to. We also set up a mailing list to easily be able to report on the progress of different parts of the package.

Pitfalls

Memory access and bitfields

Our first attempt at defining the memory registers on the MFR4200 was done using bitfields to address individual bits in the hardware registers, but we later found out that this method doesn't work using our current compiler and microprocessor. The root cause is that the MFR4200 uses 16 bit words internally and on the adress bus, while the MPC565 uses 32-bit words.

Thus, we had to rewrite the hardware mappings using bitmasks.

Hardware issues

At one point in development, when we had verified that our relevant set-up functions worked as specified, we tried to connect two FlexRay controllers to the bus to get the Flexray cycle running. However, this did not work, despite the fact that our efforts to check all constraints in the protocol against our configuration indicated that everything was correct. To solve this, we had to go back to the hardware level again. After some more probing on the card we found that what should have been an inverter between the MFR4200 and the MAX3845 [17] bus driver had been replaced by an AND gate on the protoype card. We removed the erroneous gate from the board and soldered on an XOR-gate to invert the signal, and our problem was solved.

Protocol constraints

When configuring the FlexRay protocol (see section 4.2.2) there is a large number of parameters to be configured. These have to fulfill 43 interdependent constraints. This work is tedious and error prone. The thesis worker Nikola Vorkapic whom we developed the FlexRay driver package together with, decided to write a configurator for our driver package. This is a GUI program for Microsoft Windows, written in .NET, which lets a user configure the flexray cycle parameters and message buffers. The program checks all constraints and produces a valid configuration. This greatly reduces the time needed to configure a correct cycle.

Memory constraints

The original design of the drivers, which favour readability and clarity, is memory-consuming and not suited to memory-constrained systems. Since another project using the drivers on a G1 card had problems fitting them into the system memory, a “light” version see section (5.2.2) of the FlexRay driver package was derived from the original package. This work was mainly done by Jimmy Myhrman.

The light version of the drivers is more hardware-dependent than the ordinary version. The light version uses code generated from the configurator to initialize the protocol parameters on the MFR4200 by writing directly to its memory registers. To be usable, the light drivers requires that the user has access to the configurator. However, the functions to send and recceive messages in the light package are well-documented and easy to use directly by a programmer.

Flexray Driver Light

Lightweight FlexRay driver for the MFR4200 controller. Doesn't provide any functionality for controller configuration since this is done by code generated by external tools like QRtech FlexRay configurator. This driver was derived from the GAST FlexRay driver package. Documentation [19] may be accessible to memebers of the GAST and CEDES projects.

5.3 Cluster Development

Having functional nodes, we moved on to development of the demonstrator system (see section 3.3).

5.3.1 Operating system

After reviewing a number of freely available operating systems for embedded systems, and evaluating our needs for the demonstrator system, we decided not to invest time in porting a whole operating system to our platform, and instead to write our own. The operating system is a minimal non-preemptive operating system of the type "multi-rate executive with interrupts". (See [12] for a survey of different types of operating systems.)

Since we have a very precise and distributed time source in our cluster, namely the FlexRay network, we decided to use this as time-base for our operating system. This principle is visualized in figure 5.2

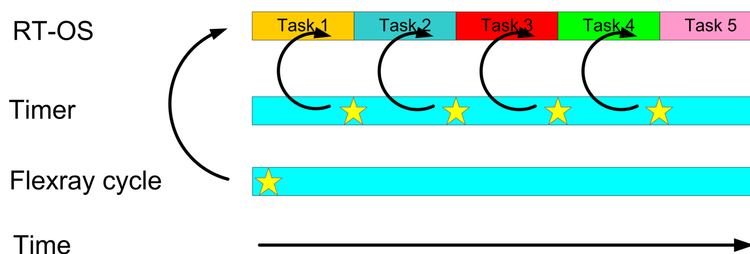

Figure 5.2: Principle of our interrupt-driven OS

On the machine level, the system works like this: Each time the FlexRay controller card detects the start of a new FlexRay cycle, it generates an interrupt to the G2 card. The G2 card catches the interrupt and starts a new OS cycle. At the same time it starts the internal PIT timer on the MPC565, and uses this timer to generate internal interrupts. The PIT interrupts control the stop and start of subsequent tasks in the OS cycle. This gives us very precise control over when in the FlexRay cycle the different tasks and the membership service (see section 6.3.3) is run.

5.3.2 Membership algorithm implementation

The membership algorithm (described in section 6.3.3) works as decribed in the results section 6.4.3.

The actual implementation is divided in four parts.

1. A service that runs once for every configured slot in the flexray cycle. This service notes all messages sent on the network from all nodes, and updates a bit in its status word for every correctly received message.

2. A process, run one time at the end of the static segment of every cycle. This process sends the nodes current status word in it's assigned slot in the dynamic segment.

3. A calculation and voting process, run at the end of the dynamic segment and in the Network Idle Time. This compares the status words of all membership-responsible nodes. If there is a difference in the statuses, this process is responsible for changin the membership status of the node, or to initiate a new round of status updates to ensure that the difference wasn't the result of a one-time error.

4. System calls used by the application processes. The membership library exports the functions `ms_write` and `ms_read` to the application programs. These functions works as intermediaries for the Flexray functions. `ms_write` updates the nodes own status word for the correct slot, since a node's communication controller can't see the messages it sends to the bus, and passes the message on to the communications controller. `ms_read` returns the data received in the correct slot for the application and appends the clusters current membership.

5.4 Utilities and example applications

5.4.1 Simple debugger/loader

We were provided with two simple applications for loading and running programs on the MPC565. It was the bootloader from the miniODEEP[14] project and a debugger/ram-loader written by Roger Johansson. The latter had basic functionality for displaying memory and starting programs from ram. We quickly found that we needed more functionality from these applications and modifications have been made to both of them throughout this project.

Bootloader modifications

We noticed when working with Eclipse that the bootloader had problems reading s-records from our serial plugin. It turned out this was due to the way the java comm package for serial communication handled new lines by sending both a carriage return and a line feed. The existing bootloader was reading s-records on a line basis into a buffer. It reads until it finds a line feed or carriage return and then works with the buffer to burn to flash. It then proceeds to the following line and if the next character is not 'S' it will abort and report an error. In the case of both a LF and a CR it will find the first character of the next line to be illegal and report an error. We needed to handle this situation and modified the bootloader to become more lean and robust.

The first step was to change the line-based reading of s-records and apply a character based approach. The modified bootloader will drop characters until it reads an 'S' on the serial line and then read and save record size and address accordingly. Another improvement is that only the actual data is read into a buffer and this eliminates the need of a separate output buffer for burning that was used initially. The end result is a more robust bootloader that handles all types of s-records. The rest of the routines such as flashburning have only seen minor modifications and the bulk of the bootloader is still made up of the original application.

Debugger/Loader modifications

The on-board debugger/loader was provided with functionality to display memory and download and run small programs from ram. Initially this was a good platform

to get familiar with- , and to be able to work with, the MPC565. As our work progressed we found that we needed more functionality in this program to provide better debug functionality. A lot of stubs for further functionality were provided by Roger Johansson with the debugger. We wrote code for a couple of these such as ability to display registers and modify memory.

The main limitation of the debugger/loader came to light when our programs had become too large to fit in ram and we needed ability to download and burn into flash instead. There were no routines for flash handling present in the debugger and we also needed to address how to handle this in the linker scripts. The debugger is loaded to the processor using the bootloader and resides in flash. This raises an issue since the flash burning routines need to be ran from ram since it is not allowed to work from-, and burn to, flash at the same time. After some study of the GNU[9] linker we found a way to solve this by separating the physical and logical addresses for the flash burning functions and perform a copy operation at initialization.

The debugger/loader was extended with two functions called fload and go flash for loading and running program from flash. Most of these routines are modified versions taken from the improved bootloader. The major changes needed to implement this functionality also prompted a general restructuring of the code and better documentation. Up until this stage the debugger/loader had been a work in progress incorporating code from different sources and without much effort going into structure. Even after the rework it remains a utility to help in development but it is easier to overview and read the code.

5.4.2 Example applications

In order to develop our demonstrator system as stated in our secondary objectives we needed to get familiar with some of the I/O-modules included on the MPC565. This work was conducted in the form of a test program for primarily the MIOS-module of the processor. The functionality we needed for our applications was determined to be PWM output and input. We also wanted to test simple parallell I/O and a few other features.

The MIOS implements PWM-functionality in two of its submodules, the PWM-modules and the dual action modules. There are 12 PWM submodules and 10 dual action submodules. The test program uses one of the PWM modules to drive a pulse where the duty cycle and period can be modified by simple commands from the ter-

minal interface. In a similar way a dual action module is configured to measure signal input, both period and pulse width.

I/O pitfalls

One thing found when writing the test program for input and output is the care needed when handling the MIOS timers. The MIOS14 have several prescalers, counter buses and other timing functions that need to be initialized correctly in order for some of the MIOS14 submodules to work as expected. Especially the dual action module is sensitive to be being connected to the correct counter bus as set in its control register.

Another thing to note when driving the HDO-circuits on the G2-board is that they float in the low state and that means the PWM-signal fall time will appear very long when measuring unloaded output. Our prototype board had a problem with the power supply to the HDO-circuits causing a processor reset when we connected a load to the output. This issue is addressed on the production cards.

Chapter 6

Results and discussion

6.1 Development tools

6.1.1 The Gast Eclipse Environment

Overview

The Gast-Eclipse-Environment (GEE) is a set of tools that were put together by this project for the purpose of having an effective development environment on win32 systems. It consists of the open source platform Eclipse that is a development environment with support for development in a number of programming languages. The C/C++ mode is capable of project management, indexing, syntax support, integrated debugging, version control and many more features. Some of these features are implemented through an extensive plug-in system. Eclipse is in wide use in the industry providing a large user base and an extensive support community.

Eclipse is capable of integrating almost any compiler and can be set to manage make files and other project settings. However we found it more suitable to use the standard make option in Eclipse and work with our own make files. The compiler toolset we decided on are the GNU-tools[9]. This is a set of open source compilers, linkers and utilities available for a wide range of platforms. In GEE we have included gcc-elf for the HCS12 target and gcc-eabi for MPC565. Since the GNU-tools are originally written for UNIX-like systems a runtime environment is needed to run them on a windows system. The one included in GEE is MinGW - Minimalist GNU for Windows.

Eclipse provide most of what is needed to develop efficiently for the MPC565 and HCS12 but one thing we found to be useful was to have a way of downloading the compiled targets (s-records) directly to the hardware from within Eclipse. Our primary method of loading is to use a loader on the target platform and download from the development machine using the serial interface. This meant we needed a terminal facility in Eclipse. Since there was no plugin that we could find that solved this issue in a way we liked we decided to write our own. The plugin was written in Java and provides a simple command line interface, a preference pane for serial communication options, menu items for downloading s-records and functionality for open and closing the serial port. Initially this plugin was based on the java.comm package from SUN but this was discontinued in November 2005. A new version was written instead using the open source rxtx [11] package from a third party developer.

Distribution

The resulting distribution of GEE from this project include detailed installation instructions as seen in appendix A. Since ease of use was one of the implicit goals for the development environment, the main distribution is in the form of a zip-archive that is easy to install. This archive includes all that is needed to get started and the only user interaction required except for unzipping the file is to add one item to the system path. The serial plug-in is included in the distribution as a separate file and it is optional to install in case future developers prefer an external terminal with more advanced functionality.

An alternative install option that uses the sources directly is described in the installation instructions. This option is for users who have more specific requirements on their environment. An example that we have used is when more advanced support for GNU tools is needed. In this case MinGW may not be sufficient and the more full featured CygWin environment can be an alternative. The serial plugin can still be used with Eclipse even if the installation was done by using the sources instead of the zip- archive.

Experience and future improvements

GEE was compiled as the first part of this thesis project and we have used it for all further development. Our experience have been very positive and Eclipse compares well to other platforms we have used before, not to mention the classic Emacs, gcc

approach. We quickly found the need to expand our system with version control and a documentation system. After some evaluation we decided on doxygen [10] for documentation and subversion[21] for version control. Both are commonly used and supported by existing plug-ins for Eclipse. We chose not to include them in our distribution since there are several similar systems available, e.g. CVS, that can be used with equal success. We strongly recommend adding this kind of functionality to GEE when working with anything more advanced than testing and evaluation. An example of the powerful use of a documentation system can be found in appendix B that has been generated with doxygen using comments from the source code.

The C/C++ perspective in Eclipse is powerful and features that we have used extensively is the indexing tools that keeps an up-to-date view of the project status including all declarations, definitions and data types. It is very easy to find the definition of a function by simply right-clicking it in the editor. Another result of the indexing is the use of auto-complete that comes in handy when working with large amounts of register definitions like on the MPC565. Eclipse is a live and evolving project so further improvements is to be expected. Eclipse also support advanced project management. These features where not put to much use in our project. This was mainly due to the low level nature of our projects and the amount of configuration that would have been required to get Eclipse working properly with our embedded targets.

Our opinion is that GEE provides a professional level development environment that is very well suited for coming projects within CEDES and other developers using the GAST hardware. The main thing that could use improvements is the serial plug-in. The terminal interface of the plug-in consists of a separate command line for input of commands to the target processor and a text area where the output is displayed. This could be integrated into a more effective interface with extended editing and command functionality.

To conclude we have fulfilled the first objective of the thesis project and the development environment has worked even better than we expected. It has all the features that we specified initially and the plugin system has helped us expand it when needed.

Alternative method of development using Eclipse

An alternative to working with the GEE environment is to use Eclipse with CygWin. This is a more complicated install but provides ability to work with the built-in

graphical debugger in eclipse in combination with a Macraigor wiggler and the OC-DRemote program. A short description of this and references to other sources can be found in the implementation section.

6.1.2 Example code and utilities

Simple loader/on-board debugger

To be able to work with the MPC565 over the serial interface some kind of loader application and basic firmware is needed. The basic code for the application we used was written by Roger Johansson. It also included some routines, mainly for serial communication, taken from the bootloader written by Carl Leskinen and Xiajoe Shen in their miniODEEP[14] project. Modifications and improvements have been made to the loader as described in the implementation chapter but it is still very much a work in progress.

The features of the current loader program include:

- Ability to download and run programs from ram or flash via the serial interface.
- Display or modify memory on the MPC565.
- Display basic register contents.

There are stubs included in the loader program in the form of dummy functions for future improvements. The loader can be used with success on an as is basis or modified to fit the needs of coming projects.

Driver examples

To understand some of the functionality of the MPC565 and to test submodules a few testing programs were written. These include a program demonstrating the use of the modular input/output submodule (MIOS) and some code demonstrating interrupt handling that we used for developing our simple OS. The MIOS program provides examples on pwm-output, input measurement, timing and counting. This program is documented using doxygen[10]. The other examples are less well documented but still provides insight for future projects. For more extensive code examples refer to the applications in our implemented demonstrator system described later in this report.

6.2 The FlexRay node

6.2.1 Hardware

During the course of the thesis project we noted several hardware issues on our prototype hardware that needed resolution in order to get a fully working FlexRay node. The areas of concern where several and we had to carefully review all aspects of the communications controller board design. There are several signal interfaces that must be correctly wired and timed: the controller host interface, the communications controller and bus driver interface, and the bus driver connection to the FlexRay bus. See the chapter on implementation for a detailed discussion of the hardware issues. The main issues solved are presented in table 6.1.

6.2.2 Software drivers

Overview

The software drivers for the FlexRay controller are a result of the cooperation between three different thesis projects as described in the implementation section. They are delivered in two versions. The main version provides functions for configuration and message handling corresponding to the parameters defined in the FlexRay protocol specification. The light version uses a configuration tool written by Jimmy Myhrman and Nikola Vorkapic and modified versions of the message handling routines. Both sets of drivers are designed for a general approach and are able to handle many different configuration choices.

A layered approach have been applied to the driver development and the end result provide a relatively easy to use application programming interface (API). The lower layers mask the specifics of the MFR4200 communication controller and provides a mapping of the protocol specification parameters to the physical registers of the chip. The API is only concerned with parameters for the protocol itself and this facilitates migration to another controller. It may be done by replacing the lower layers with only minor changes needed in the application.

A detailed description of the API is provided in appendix B. The main features of the driver package include:

Problem description	Fault	Solution
Unable to read the registers on the communication controller	Chip select and read enable signals did not follow the specification in the MFR42000 data sheet[7].	Adjusted bus timing in the MPC565 clock control registers.
Unable to write to registers on the communication controller.	Missing jumper for the WE signal on the host processor card.	Installed jumper
Bus driver not driving bus even when TX-signal is present on the communication controller.	TX- enable signal was not inverted	Hot fix by replacing a logic gate used for bus driver disable to become an inverter instead. Fix for production board by adding inverter logic.
Incorrect frame and cycle times on the FlexRay bus	Communication controller not running at 40 MHz	Hot fix by replacing the oscillator crystal with a more advanced oscillator. Fix for production boards by using more advanced oscillator.
No interrupt present on bus IRQ-channels when communication controller generates interrupt	Missing jumper on FlexRay controller board.	Installed jumper.

Table 6.1: Overview of encountered hardware issues

- Configuration functions for all FlexRay parameters to use with application code or automatic configuration with configuration tool (light version).
- Buffer configuration functions for message buffer configuration, manual or automated.
- Functions for sending and receiving FlexRay frames using dedicated buffers.
- Functions for receiving FlexRay frames using FIFO buffers.
- Support for any design of communication cycle allowed by the protocol. The drivers handle both static and dynamic segment transmissions of all kinds.

Discussion and improvements

The drivers produced by the project correspond very well to what was specified in the thesis objectives. The ability to work with the other thesis projects have greatly sped up development and have resulted in more effective testing of the drivers than would otherwise have been possible. The use of Doxygen for documentation and consistent commenting of the source code provides a lot of material for developers who need to get familiar with the API.

The focus has been to create a set of drivers for general testing and evaluation purposes. While supporting all possible configurations of the FlexRay network they are not optimized for any specific case. The software should be used as a tool to evaluate and test different system configurations. It may then serve as a base when writing optimized drivers for final target systems.

6.3 Membership agreement in a FlexRay network

6.3.1 Implementation of an existing TDMA membership protocol

The issue of implementing a membership service in FlexRay have been studied quite extensively in the second half of this project. Our prime candidate for test implementation was the membership protocol 2 discussed in section 4.4.3 and this was implemented and tested in our one processor/two node prototype setup. This is an ack by frame approach where acknowledgments for previous frames are piggy-backed onto messages in the following frames. In our limited setup it was not possible to do a full simulation of agreement times and evaluate properties. However, the practical

implications of using a frame by frame approach were well studied and some issues were found. The main considerations when trying to reach membership agreement using protocol 2 were:

Processor overhead FlexRay is a high speed protocol which means that frame duration is relatively short. An application that needs to check each frame as soon as they arrive will need to run at very short intervals on the host processor. This means a lot of CPU overhead for task switching.

Addressing Since FlexRay does not address on physical nodes but instead uses frame ids that are in turn assigned to specific nodes it is possible to create cycles where one node will have a number of slots in sequence assigned to it. In a frame by frame acknowledgment scheme this node would be acking itself for a number of frames.

Controller limitations The communications controller (MFR4200) used in this project requires that a message buffer is committed for transmission at least $1\mu s$ before the start of slot $n-1$ if it is to be transmitted in slot n. This causes an inherent lag that means an acknowledgment for the message in slot $n-2$ can not be sent until slot $n+1$.

The three items described above were the major reasons this protocol was abandoned for the demonstrator system. The inherent lag combined with the addressing issue means that the advantage of the frame by frame approach, fast agreement, is lost. This could probably be solved with a careful choice of schedule but it does not remove the main obstacle of processor overhead. The standard way to work with the communication controller is to let it handle the task of sending and receiving messages on the physical bus. The host processor can then read and commit messages via the controller host interface whenever it finds it appropriate. If the host processor needs to poll the communication controller after each slot then there is no gain in queuing messages on the communication controller. We also found that the process time to read a message frame, handle the acknowledgment vector, update it and then piggy back the new vector to the next message was quite long. In order to run other applications in between the membership service we needed to extend the duration of the slots in the cycle to well beyond what was needed to send the data.

6.3.2 General considerations for FlexRay

Based on the experience from the implementation of membership protocol 2 described in 4.4.3 a more general discussion about membership in FlexRay is needed. The speed of the network makes a frame by frame approach hard to implement since the suitable place for that kind of algorithm would be to run it on the communication controller like in TTP. Otherwise physical limitations of the controller host interface will have an impact on how fast a received message can be handled and acknowledged by the host.

The natural time scale to work with for a membership service in FlexRay instead seems to be the communication cycle. This is because received messages are often processed by the applications on a per cycle basis and the results produced are committed for transmission in the next cycle. If agreement is reached at a determined time in the cycle then this fact can be used when writing the applications and the impact of the delay reduced to a minimum.

Based upon a cycle approach a new protocol for membership agreement is proposed in this thesis. It uses a voting to ensure agreement.

6.3.3 Membership protocol 3

This protocol is an evolved version of protocol 1 that uses the properties of the dynamic segment in FlexRay to provide a low overhead membership protocol. Membership is calculated on a per cycle basis which has some impact on the time to agreement compared to protocol 2 but has several advantages. The proposed protocol acts on the node level and is implemented as follows:

- A membership service running on the nodes is assigned the n first slots in the dynamic segment.
- Time slots in the static segment are regarded as the information bearer for a corresponding membership entity, i.e. application.
- The entities (time slots) that should be included in the membership are decided in the system design phase.
- The membership service on each node monitors the time slots in the static segment to see if there is correct transmission/reception.
- An optional function can be called by the membership to check whether the information sent in the slot is valid.

- If no change in the membership is detected by a node during a cycle the membership service will not transmit on the bus.
- If a node detects a change in the membership it will signal this in its assigned slot in the dynamic segment.
- If a majority of the nodes in a cycle signal a change then the membership can be updated.

6.3.4 Properties of protocol 3

Assume that nodes and processes are fail silent and that a maximum of $(n/2 - 1)$ nodes or membership service routine may fail during a cycle. Also assume that all correct membership services monitor messages as they arrive and commit their view for transmission at their assigned slots in the dynamic segment. If all correct nodes have a consistent local state then it is possible to reach agreement on group membership before the start of the next communication cycle in a network consisting of n nodes when using the algorithm of membership protocol 3.

If we design a system to be able to tolerate one, fail silent, node failure then we need three nodes and three slots in the dynamic segment to implement the protocol. The overhead generated by the membership service will be three minislots per cycle when the system is working correctly. Three minslots correspond to roughly one half or one third of a regular frame duration depending on the configuration of the cycle. This is a significant improvement compared to using assigned frames in the static segment.

6.4 Thesis demonstrator system

6.4.1 System design

The demonstrator system's aim is to serve both as an example for the FlexRay software drivers and a first benchmark of membership implementation. It also incorporates some of the software based fault tolerance discussed in the background section. The demonstrator includes three basic applications, a membership service and a simple non-premptive real time operating system. All these five programs were developed within the thesis project. There is also some additional hardware in the form of two simple electric DC-motors and a pedal with integrated electronic sensor from Scania. Figure 6.1 provides a simple representation of the system.

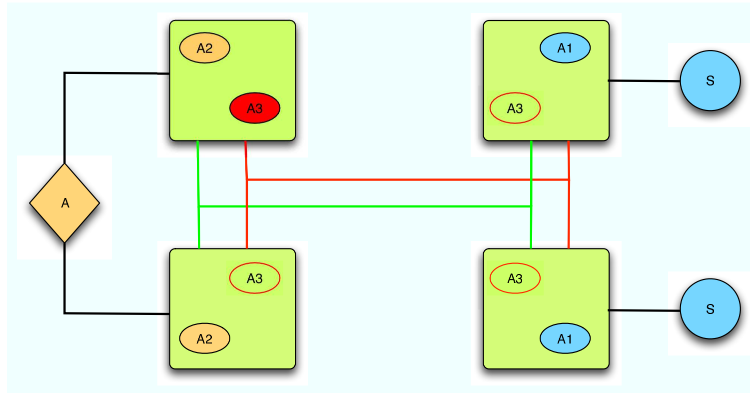

Figure 6.1: The system representation of the thesis demonstrator. The four nodes are connected through a redundant (using channel A and B) FlexRay bus. Two nodes are connected to the sensors in the pedal, each running an instance of application A1 that reads sensor values. The other two nodes are connected to the actuator in the form of a DC-motor. The actuator control application is A2. The control algorithm is run by application A3 and this is run at one of the nodes but implemented at all four. In the case of failure of the running control application it will be started on one of the other nodes. Each node also run a membership service and a simple RTOS

The four nodes are connected by a dual channel FlexRay network in a bus configuration. Each node run the real time operating system that is responsible for scheduling of the applications and services running on the node. The membership service is implemented as a middleware between the applications and the FlexRay driver layer. The membership service provide agreement on communication membership and may also implement correctness checks for received data to assure application membership. The three applications are presumed independent and communicate only through FlexRay.

6.4.2 Interrupt driven RTOS

The RTOS is written as a simple non-preemptive multi-rate scheduler. It is responsible for starting applications at given times and to remove applications that fail or does not terminate in time. The OS uses two kinds of timer interrupts for scheduling, an external interrupt from the FlexRay controller and an MPC565 internal from the PIT-timer. The schedule is synchronized with the FlexRay communications cycle.

At cycle start an interrupt is generated to the RTOS interrupt handler. On this first interrupt the RTOS starts the PIT-timer and launches the first application to run in the cycle. When the application is finished it signals this to the RTOS and enters an idle loop. The next interrupt comes from the PIT-timer and the RTOS resets the timer, checks that the previous application terminated correctly, and launches the next one in the schedule. This is repeated until all applications in the schedule has run. Then the RTOS enters an idle phase and waits for the next FlexRay interrupt.

6.4.3 Membership service

The membership service is implemented as a layer between the applications and the FlexRay driver package. The service monitors the slots in the static segment corresponding to the entities that are included in membership. The current state of the membership is kept as a local state vector. Since the membership service is responsible for reading messages from FlexRay, and then deliver them to the applications, it may also incorporate constraint checks on the payload data to avoid delivery of bad data. To ensure that the state of the entities is consistent throughout the cluster the membership also keeps a global state vector. It is the global state that is reported to the applications running on the node.

The global state is updated at the end of each cycle based upon data sent in the dynamic segment of the cycle. If there is no change detected during the static segment of the communications cycle the service will simply keep the same local and global states. If the membership service detects a change it will update its local state and then transmit the new local state in the dynamic segment. All nodes monitor the dynamic segment and if there is a majority of nodes transmitting a new (consistent) state it will be accepted as the new global state and the global state is updated in all nodes. By applying this algorithm all applications on all nodes are provided with the same state view from the membership service and we have agreement all over the cluster.

6.4.4 Sensor application

The sensor application runs a routine for reading sensor values. It is a simple producer application that will read a voltage level using the queued analog to digital converter (QADC) on the MPC565 corresponding to the state of the sensor(pedal). The voltage value is sent in the assigned slot in the static segment of the communication cycle.

The application is part of the membership but only uses membership for performing a restart if it is voted out. The application has two functions. The read/send function that is normally executed when the application is called by the OS and an initialization and startup function that is called after system reset or at restart.

6.4.5 Actuator application

The actuator application runs a routine for driving the small electric motors that make up the actuators. It is part of the membership and uses the global state for two things. First it checks whether it is part of the membership and allowed to drive the actuator and secondly it checks what frame id holds the information from the controller application. It then reads a value from the correct frame and outputs a corresponding driving pulse (pwm) to the motors. Finally it sends an acknowledgment in its assigned static slot for the membership to see that it is live and driving the correct value. The application has two functions. The main read/drive routine that is normally executed when the application is called by the OS and the startup function that is called after system reset or at restart.

6.4.6 Control application

The control application uses a value from the sensor to calculate the correct output drive for the actuator. It is configured in a standby/active configuration according to a priority scheme. The control application with the highest priority will be active, perform the calculation, and output a result in its assigned slot in the static segment. The standby applications will send an empty frame to indicate that they are live and standby. The control application uses the global state of the membership to see which sensor to use. If one sensor fails it will read the value from the other one instead. It also uses the global state to go from standby to active if it detects that the active control application has failed. There is no conflict if two control applications would happen to run at the same time since the actuator will use the value from the highest priority application that is part of membership.

The calculation algorithm is a conversion of the voltage level of the sensor to a value for the actuator drive. It uses a linear model for the sensor. This function could have been implemented in the actuator application, but the purpose of including the control application is the simplicity of replacing it with a more sophisticated algorithm. It also demonstrates the use of active standby functionality implemented in software.

6.4.7 System status and future work

Due to delays in production we were not able to test the demonstrator system on a full scale. We had only access to one G2 processor card and used this to control two FlexRay controllers to get a simple FlexRay network. All applications have been written and tested to the extent possible in this limited setup. The system should be able to perform in the complete setup described in figure 6.1 with only minor adjustments.

The system design is modular and can easily be modified to perform more advanced functions by replacing one or more of the applications. Since there are well defined interfaces between the membership service and applications it is a relatively uncomplicated task to add or remove entities from membership control. The FlexRay driver package can be used in parallel with the membership service to enable applications that are not part of membership to still function in a distributed manner. A primary future function of the system is to test and evaluate different algorithms for membership agreement in a FlexRay network. This has been a primary concern in designing the system and the membership service is separated from the applications in a manner that the underlying algorithm could be changed. According to the plans made when we created the demonstrator we suggest the following development approach:

- Test the complete system when hardware becomes available to make sure all applications and services function as intended on the system-wide level.
- Run simulations on the membership protocol implemented to determine benchmarks. Properties of interest, among others, are agreement time, bandwidth usage, processor usage and consistency.
- Inject faults into the system and study how the membership service and fault tolerant applications perform in the presence of faults.
- Implement alternative protocols for membership agreement and do comparison studies.

References

[1] The AUTOSAR project. Automotive open systems architecture [online]. 2006. Available from: http://www.autosar.org.

[2] CEDES project. The cedes project homepage [online]. 2006. Available from: http://www.cedes.se.

[3] F. Christian. Agreeing on who is present and who is absent in a synchronous distributed system. 1998.

[4] FlexRay Consortium. *FlexRay Communications System - Protocol Specification - Version 2.1*, May 2005.

[5] FlexRay Consortium. Flexray consortium homepage [online]. Sep 2005 [cited 2006-02-22]. Available from: http://www.flexray.com/.

[6] Freescale semiconductor. *MPC565/566 User's manual*, Sep 2002.

[7] Freescale semiconductor. *MFR4200 Data Sheet Rev 0*, Aug 2005.

[8] GAST project. The gast project homepage [online]. april 2004. Available from: http://www.chl.chalmers.se/gast.

[9] GNU project. The gnu operating system homepage [online]. 2006. Available from: http://www.gnu.org.

[10] Dimitri van Heesch. Doxygen home page [online]. Sep 2002 [cited 2006-02-02]. Available from: http://www.stack.nl/ dimitri/doxygen/.

[11] Trent Jarvi. Rxtx homepage [online]. Sep 2006 [cited 2006-02-02]. Available from: http://www.rxtx.org/.

[12] D. Kalinsky Associates. A survey of task schedulers [online]. Sep 2006 [cited 2006-02-02]. Available from: http://www.kalinskyassociates.com/Wpaper3.html.

[13] H. Kopetz and G. Grünsteidl. Ttp-a protocol for fault-tolerant real-time systems. *IEEE Computer (Vol 27, No 1 pp. 14-23)*, January 1994.

[14] Carl Leskinen and Xiajoe Shen. miniodeep - a foundation for the open dependable electrical and electronics platform. Technical report, Computer Science and Engineering, Chalmers, Göteborg, Sweden, 2005.

[15] H. Lonn and R. Snedsbol. Efficient synchronisation, atomic broadcast and membership agreement in a tdma protocol. Technical report, Dijon, France, 1996.

[16] James P. Lynch. Arm cross development with eclipse version 3 [online]. Dec 2005 [cited 2006-02-02]. Available from: http://www.olimex.com/dev/pdf/ARM Cross Development with Eclipse version 3.pdf.

[17] Maxim Integrated Products. *3.3V-Powered, 10 Mbps and Slew-Rate-Limited True RS-485/RS-422 Transcievers*, Dec 1994.

[18] J. Myhrman, M. Pettersson, P. Uvesten, and N. Vorkapic. *Flexraydriver Documentation*, Feb 2006.

[19] J. Myhrman, M. Pettersson, P. Uvesten, and N. Vorkapic. *Flexraydriver Light Documentation*, Feb 2006.

[20] N. Storey. *Safety-Critical Computer Systems*. Addison Wesley Longman, 1996.

[21] Tigris.org. Subversion home page [online]. Sep 2002 [cited 2006-02-02]. Available from: http://subversion.tigris.org/.

[22] Wikipedia. Aircraft flight control systems article [online]. Feb 2006 [cited 2006-02-21]. Available from: http://en.wikipedia.org/wiki/Fly-by-wire.

[23] Wikipedia. Controller area network article [online]. Feb 2006 [cited 2006-02-21]. Available from: http://en.wikipedia.org/wiki/Controller_area_network.

List of Figures

4.1 Example of a FlexRay communication cycle using both static and dynamic segment 15

4.2 Example of a federated design for x-by-wire. All control functions have their own dedicated hardware and are loosely coupled through bus systems. Fault tolerance is achieved by redundant central controllers and bus lines. Note the amount of hardware needed for each function. 19

4.3 Example of an integrated design for x-by-wire. A common hardware platform is used by all the control functions. Fault tolerance is achieved by having backup processes on different nodes. The system is tightly coupled and must be able to assure that control functions cannot interfere in non-permitted ways. 20

5.1 The layered design of the FlexRay driver package 27

5.2 Principle of our interrupt-driven OS 30

6.1 The system representation of the thesis demonstrator. The four nodes are connected through a redundant (using channel A and B) FlexRay bus. Two nodes are connected to the sensors in the pedal, each running an instance of application A1 that reads sensor values. The other two nodes are connected to the actuator in the form of a DC-motor. The actuator control application is A2. The control algorithm is run by application A3 and this is run at one of the nodes but implemented at all four. In the case of failure of the running control application it will be started on one of the other nodes. Each node also run a membership service and a simple RTOS 45

List of Tables

4.1 Some of the key features of the MPC565 microcontroller 12

4.2 Some of the key features of the MFR4200 communications controller . 12

4.3 Data matrix collected at the nodes in membership protocol 1 22

6.1 Overview of encountered hardware issues 40

Index

Actuator, 47

Brake by wire, 18

cross-compilers, 7

Debugger, 38

Demonstrator system, 44

doxygen, 62

Eclipse, 35

Fault tolerance, 16, 19

    Example of a federated design for x-by-wire, 19

firmware, 38

FlexRay, 13

    example of cycle, 15

    deterministic network protocol, 20

    driver package, 26

    physical layer, 14

GAST, 26

    communications board, 12

    G2 board, 11

    hardware, 11

    hardware issues, 25

Gast Eclipse Environment, 35

GEE, 37

h, 61

hardware independence, 26

Loader, 38

Membership, 20, 46

MIOS, 38

redundancy, 19

Reliability, 17

RTOS, 2, 45

Sensor, 46

subversion, 62

TTP, 21

x-by-wire

    introduction to, 1

Glossary

API Application Programming Interface - a system of tools and resources in a computer system enabling developers to create software applications

Assembler low-level non-portable programming language

C high-level programming language with good support for directly manipulating the hardware. Used in this project and in most embedded systems development

CAN Controller Area Network

CEDES Cost Efficient Dependable Electronics Systems, a research project led by Chalmers

cluster a group of loosely coupled computers that work together closely so that in many respects it can be viewed as though it were a single computer

cross-compiler A compiler which compiles code for a different architecture than that on which the compiler itself resides

dependability a property of a system that justifies placing one's reliance on it

ECU Electronic Control Unit

FlexRay a time triggered communications protocol primarily designed for automotive applications

GAST General Application Development Boards for Safety Critical Time-Triggered Systems

hardware independence Designing software so that it runs the same way regardless of underlying hardware

membership field of study in computer science concerning how to achieve a common view of the state of all tasks in a distributed system

MFR4200 FlexRay controller chip used in this project

MPC565 a PowerPC-processor. Used as the main processor on the G2 board

RTOS Real-Time Operating System

SP Swedish National Testing and Research Institute

TTCAN Time-Triggered version of CAN

TTP/C Time Triggered Protocol class C

x-by-wire replacing mechanical and hydraulic systems with electrical systems

Appendix A

The GAST Eclipse Environment (GEE) for Windows

A.1 Introduction

Version 1.10 2006-02-20

The open source development tools for the GAST-hardware is based on a collection of tools, mainly eclipse and the gcc-compilers. These tools are provided on an "as-is" basis to facilitate development on the provided hardware. The aim is also to provide a common and tested platform for developers and decrease the startup time for developers unfamiliar with the project. This version was put together by Mattias Pettersson and Petter Uvesten as a first attempt for an integrated environment.

GEE is based on the following tools:

1. Included in zip installation:
    - EclipseSDK Version: 3.1.0 `http://www.eclipse.org/platform`
    - Eclipse CDT Version: 3.0 `http://www.eclipse.org/cdt`
    - Mingw runtime Version: 3.8 `http://www.mingw.org/download.shtml`
    - Mingw32-make Version: 3.8 `http://www.mingw.org/download.shtml`

- Msys Version: 1.0.10 `http://www.mingw.org/download.shtml`
- Gcc* Version: 3.4.2 `http://www.mingw.org/download.shtml`
- Gdb* Version: 5.2.1 `http://www.mingw.org/download.shtml`
- g1-elf-gcc `http://www.chl.chalmers.se/gast/FIRMWARE/`
- g2-eabi-gcc `http://www.chl.chalmers.se/gast/FIRMWARE/`

(* means components that may not be strictly necessary but are included for convenience.)

2. Optional add-ons

- GAST-connector basic serial connection plugin for eclipse requires the rxtx package for win32
`ftp://ftp.qbang.org/pub/rxtx/rxtx-2.-7`-bins.zip1

A.2 Installation instructions

There are two methods for installing the GAST Eclipse Environment. Either use the provided zip-file or download and install the different components from their respective source. A Brief guide to both methods is provided. You need to have JDK 1.5 or better installed on you machine for Eclipse and GastConnector to work properly.

1. Installing from the zip file

(a) Unzip gee.zip directly to `C:\`. This will create a directory called `C:\gee` containing the files of the environment. Note that the amount of included files will cause this operation to take some time.

(b) Add the following to your path `;C:\gee\mingw\bin;C:\gee\msys\bin` by right clicking on my computer, choose properties, advanced tab and then choose environment variables and append the above to the path string.

(c) Create shortcuts to `C:\gee\eclipse\eclipse.exe` from the desktop and start menu.

(d) Start eclipse.

(e) To install the optional plugin for serial communication within eclipse you first need to install the javax.comm package if it is not installed on the machine. This package is not part of the J2SE installation. Follow this procedure:

i. Download and unpack the rxtx-2.1-7-bins.zip file.
ii. Place the rxtxSerial.dll in <jdk>\jre\bin directory (or the \jre\bin subdirectory of your current J2SDK directory.)
iii. Place the RXTXcomm.jar in <jdk>\jre\lib\ext.
iv. Do not alter the CLASSPATH.

where <jdk> refers to the directory path for your JDK, e.g. C:\ProgramFiles\Java\jdk1.5.0. For help with installation problems, see the file Install that comes with the rxtx files or look at the documentation at the rxtx site [11].

When rxtx is configured correctly simply unzip the file GastConnector.zip to C:\gee\eclipse and restart eclipse. For more info about plugins and eclipse see http://www.eclipse.org

2. Downloading and installing from the sources

    (a) Download the different packages from the sources above except CDT.
    (b) Choose a directory for the environment, recommended is C:\gee. If another path is chosen make sure to adjust the following steps accordingly.
    (c) Install the mingw runtime to C:\gee\mingw
    (d) Install msys to C:\gee\msys. The installer will run a postinstall where you specify the path to where you installed mingw above.
    (e) Add the following to your path ;C:\gee\mingw\bin;C:\gee\msys\bin
    (f) Unzip g1-elf-gcc and g2-eabi-gcc to the mingw directory i.e. C:\gee\mingw. Skip or replace the .info files that will be the same. Check C:\gee\mingw\bin to make sure you now have files named g1-gcc and g2-eabi-gcc etc.
    (g) Install eclipse to C:\gee\eclipse
    (h) Start eclipse and download and install the C development tools using the built in installer system. Instructions can be found at http://www.eclipse.org/cdt
    (i) If you would like to use the optional GastConnector plugin follow step 5 above.

A.3 Getting started

To learn more about development using eclipse we recommend one of the many tutorials provided with the platform. Of particular interest may be some of the C/C++ tutorials. These can be accessed from the welcome screen when you start eclipse for the first time or by choosing from the help menu.

To get started on GAST development open the C/C++ perspective and start a new project. We recommend using a standard make C-project since configuration of the managed make to work with the cross compilers can be tricky. This means you have to manually specify where your compilers are in the make file. An example of this using the standard install and the g2-eabi-gcc cross compiler is:

```
# change following due to Your installation
TOOLDIR     = c:/gee/mingw/bin/
ARCH        = eabi
PREFIX      = g2-$(ARCH)

CC          = $(TOOLDIR)$(PREFIX)-gcc
LD          = $(TOOLDIR)$(PREFIX)-ld
AS          = $(TOOLDIR)$(PREFIX)-as
OC          = $(TOOLDIR)$(PREFIX)-objcopy
```

More configurations for the project can be found by right clicking on the project and choosing properties or through the project menu. Also make sure you have a correct configuration for the C/C++ perspective under window - preferences.

Eclipse powerful plugin system makes it easy to use other tools to help the development. Two plugins that we have used extensively is subclipse (`http://subclipse.tigris.org`), adds support for subversion (see [21]), and eclox (`http://home.gna.org/eclox/`),adds support for doxygen (see [10]) documentation.

A.4 The GAST connector plugin

The GAST-connector plugin is a serial communication interface written in java as a plugin for the eclipse environment. It was created by Petter Uvesten and Mattias Pettersson based on example code provided by Sun with the javax.comm package. It adds very basic functionally with a terminal-like interface and simple menu functions to download s-records to the development boards. The main purpose of the plugin is to avoid having to use an external terminal during development. This enables the developer to write, compile, download and run a program from within eclipse over the serial interface.

Instructions:

1. Options
   Choose window - preferences from the menu and open the GastConnector tab PORT - chose what serial port to use for board communication (COM1,...) SPEED - baud rate for the serial connection DELAY - Add a delay in milliseconds between s-records on the serial line can be useful when working with flash-burning.

2. Start the terminal view
   Choose window - show view - gastConnector - terminal view to start the plugin. Note that the port cannot be opened before this step is performed. This will add a tab called terminal view to the bottom right corner of eclipse containing the command line interface

3. Open the port
   Choose gastConnector - open port. This will open a serial port according to the settings made under options if the terminal view is running. If the port open fails an error message will be displayed.

4. Communicating with the board
   If the development board is executing a program that communicates over the serial interface it is now possible to send commands and receive output from the microcontroller using the command line in terminal view.

5. Downloading S-records
   If you compiled and s-record and would like to download it to the micro- controller this is possible using a simple menu item. First use terminal view to make sure the microcontroller is ready to accept an s-record. Then simply right click on the s-record of interest and chose gastConnector - download s-record.

The plugin has been tested with the existing versions of the on chip debugger and bootloader for the G1 and G2 boards.

Appendix B

FlexRay API documentation

B.1 FlexRay driver package

This is the layer 3 functions from the FlexRay driver package. These functions are meant to be used by application and OS programmers using the GAST hardware to develop systems. They should work on most embedded systems using the MFR4200 as FlexRay controller. With modifications to the underlaying layers they should be easily portable to another FlexRay controller chip.

Extensive documentation of the whole driver package is available in [18], which may be available to members of the GAST and CEDES project from the GAST project [8].

B.2 src/chi_prot_cfg.c File Reference

B.2.1 Detailed Description

Functions for setting and reading flexray protocol configuration data.

Author:

Petter Uvesten

Mattias Pettersson

Nikola Vorkapic (`nikola@vorkapic.com`)

Jimmy Myhrman (`jimmy@myhrman.org`)

Date:

2005-10-28

```
#include "chi_prot_cfg.h"
```

Functions

- INT32 **set_pMicroPerCycle** (UINT32 cyclelength, UINT16 macro_per_cycle)

  Set the number of microticks constituting the duration of the communication cycle.

- UINT32 **get_pMicroPerCycle** ()

  Get the number of microticks constituting the duration of the communication cycle.

- INT32 **set_pMicroPerMacroNom** (UINT16 micro_per_macro)

  Set number of microticks per nominal macrotick.

- UINT8 **get_pMicroPerMacroNom** ()

  Get number of microticks per nominal macrotick.

- INT32 **set_gMacroPerCycle** (UINT16 cyclelength)

  Set the number of macroticks constituting the duration of the communication cycle.

- UINT16 **get_gMacroPerCycle** ()

  Get the number of macroticks constituting the duration of the communication cycle.

- INT32 **set_gNumberOfStaticSlots** (UINT16 slots)

  Define the number of static slots in a cycle.

- UINT16 **get_gNumberOfStaticSlots** ()

  Get the number of static slots in a cycle.

- INT32 **set_gdStaticSlot** (UINT8 ticks)

  Set the length of a static slots in macroticks.

- UINT8 **get_gdStaticSlot** ()

  Get the length of a static slots in macroticks.

- INT32 **set_gdActionPointOffset** (UINT8 ticks)

  Set the action point offset in macroticks.

- UINT8 **get_gdActionPointOffset** ()

  Get the action point offset in macroticks.

- INT32 **set_gdMinislot** (UINT8 ticks)

  Set the length of a minislot in macroticks.

- UINT8 **get_gdMinislot** ()

  Get the length of a minislot in macroticks.

- INT32 **set_gdMinislotActionPointOffset** (UINT8 offset)

  Set the number of macroticks constituting the offset of the action point within a minislot of the dynamic slot.

- UINT8 **get_gdMinislotActionPointOffset** ()

  Get the number of macroticks constituting the offset of the action point within a minislot of the dynamic slot.

- INT32 **set_gdDynamicSlotIdlePhase** (UINT8 slots)

  Set the number of minislot constituting the duration of the idle phase within a dynamic slot.

- UINT8 **get_gdDynamicSlotIdlePhase** ()

  Get the number of minislot constituting the duration of the idle phase within a dynamic slot.

- INT32 **set_pLatestTx** (UINT16 ticks)

  Set the latest time in macroticks at which transmission can start in the dynamic segment.

- UINT16 **get_pLatestTx** ()

Get the latest time in macroticks at which transmission can start in the dynamic segment.

- INT32 **set_gdSymbolWindow** (UINT16 ticks)

  Set the start time in macroticks for the symbol window during the communication cycle.

- UINT16 **get_gdSymbolWindow** ()

  Get the start time in macroticks for the symbol window during the communication cycle.

- INT32 **set_gColdStartAttempts** (UINT16 attempts)

  Set the max number of coldstart attempts a node is allowed to make.

- UINT16 **get_gColdStartAttempts** ()

  Get the max number of coldstart attempts a node is allowed to make.

- INT32 **set_isStartupNode** (UINT8 isStartup)

  Set if this node is a startup node.

- UINT8 **get_isStartupNode** ()

  Get if this node is a startup node.

- INT32 **set_SyncFrameIdentifier** (UINT16 pattern)

  Set sync frame idetifier.

- UINT16 **get_SyncFrameIdentifier** ()

  Set sync frame idetifier.

- INT32 **set_SyncFrameHeaderCRC** (UINT16 crc)

  Set header CRC of the sync frame to be transmitted.

- UINT16 **get_SyncFrameHeaderCRC** ()

  Get header CRC of the sync frame to be transmitted.

- INT32 **set_gListenNoise** (UINT8 cycles)

  Set duration in cycles of listen timeout with noise.

- UINT8 **get_gListenNoise** ()

  Get duration in cycles of listen timeout with noise.

- INT32 **set_gMaxWithoutClockCorrectionFatal** (UINT16 odd)

  Set maximum number of odd cycles before node enters stop state due to missing sync frame pairs.

- UINT16 **get_gMaxWithoutClockCorrectionFatal** ()

  Get maximum number of odd cycles before node enters stop state due to missing sync frame pairs.

- INT32 **set_gMaxWithoutClockCorrectionPassive** (UINT16 odd)

  Set maximum number of odd cycles before node enters passive state due to missing sync frame pairs.

- UINT16 **get_gMaxWithoutClockCorrectionPassive** ()

  Get maximum number of odd cycles before node enters passive state due to missing sync frame pairs.

- INT32 **set_gdBit** (UINT16 ticks)

  Set the bit duration in microticks.

- UINT16 **get_gdBit** ()

  Get the bit duration in microticks.

- INT32 **set_gdMaxDrift** (UINT16 ticks)

  Set the maximum frequency deviation of the oscillator clock.

- UINT16 **get_gdMaxDrift** ()

  Get the maximum frequency deviation of the oscillator clock.

- INT32 **set_pDelayCompensationA** (UINT8 micros)

  Set compensation for reception delays on channel A in microticks.

- UINT8 **get_pDelayCompensationA** ()

  Get compensation for reception delays on channel A in microticks.

- INT32 **set_pDelayCompensationB** (UINT8 micros)

  Set compensation for reception delays on channel B in microticks.

- UINT8 **get_pDelayCompensationB** ()

  Get compensation for reception delays on channel B in microticks.

- INT32 **set_gdNIT** (UINT16 ticks)

  Set the start cycle time of the network idle time.

- UINT16 **get_gdNIT** ()

  Get the start cycle time of the network idle time.

- INT32 **set_pOffsetCorrectionOut** (UINT16 micros)

  Set maximum absolute offset correction value in microticks.

- UINT16 **get_pOffsetCorrectionOut** ()

  Get maximum absolute offset correction value in microticks.

- INT32 **set_pRateCorrectionOut** (UINT16 micros)

  Set maximum permitted absolute rate correction value in microticks.

- UINT16 **get_pRateCorrectionOut** ()

  Get maximum permitted absolute rate correction value in microticks.

- INT32 **set_pClusterDriftDamping** (UINT8 micros)

  Set value used to minimize the accumulation of rounding errors in clock synchronization.

- UINT8 **get_pClusterDriftDamping** ()

  Get value used to minimize the accumulation of rounding errors in clock synchronization.

- INT32 **set_gOffsetCorrectionStart** (UINT16 macros)

  Set delay in multiples of macroticks after which the offset correction will start.

- UINT16 **get_gOffsetCorrectionStart** ()

Get delay in multiples of macroticks after which the offset correction will start.

- INT32 **set_gPayloadLengthStatic** (UINT8 words)

  Set maximum data length for static frames in words.

- UINT8 **get_gPayloadLengthStatic** ()

  Get maximum data length for static frames in words.

- INT32 **set_gSyncNodeMax** (UINT16 frames)

  Set the maximum number of sync frames that can be transmitted on any channel including the nodes own sync frame.

- UINT16 **get_gSyncNodeMax** ()

  Get the maximum number of sync frames that can be transmitted on any channel including the nodes own sync frame.

- INT32 **set_pWakeupChannel** (UINT8 mode)

  Set the channel on which a wakeup symbol shall be sent.

- UINT8 **get_pWakeupChannel** ()

  Get the channel on which wakeup symbols are sent.

- INT32 **set_pWakeupPattern** (UINT8 symbols)

  Set the number of wakeup symbols to be transmitted.

- UINT8 **get_pWakeupPattern** ()

  Get the number of wakeup symbols to be transmitted.

- INT32 **set_gdTSSTransmitter** (UINT8 bits)

  Set the number of bits within the transmitter start sequence for transmission.

- UINT8 **get_gdTSSTransmitter** ()

  Get the number of bits within the transmitter start sequence for transmission.

- INT32 **set_pdTSSReceiver** (UINT8 bits)

  Set the number of bits within the transmitter start sequence for reception.

- UINT8 **get_pdTSSReceiver** ()

  Get the number of bits within the transmitter start sequence for reception.

- INT32 **set_gdWakeupSymbolTxIdle** (UINT8 bdurs)

  Set the duration of the idle period for wakeup symbols.

- UINT8 **get_gdWakeupSymbolTxIdle** ()

  Get the duration of the idle period for wakeup symbols expressed in bit durations for the network.

- INT32 **set_gdWakeupSymbolTxLow** (UINT8 bdurs)

  Set the duration of the low period of wakeup symbols in terms of bit durations for the network.

- UINT8 **get_gdWakeupSymbolTxLow** ()

  Get the duration of the low period for wakeup symbols expressed in bit durations for the network.

- INT32 **set_BusGuardianTick** (UINT8 ticks)

  Set the period length of the bus guardian tick to be provided by the CC to the bus guardian.

- UINT8 **get_BusGuardianTick** ()

  Get the period length of the bus guardian tick provided by the CC to the bus guardian.

B.2.2 Function Documentation

UINT8 get_BusGuardianTick ()

Get the period length of the bus guardian tick provided by the CC to the bus guardian.

Returns:

the number of microticks

See also:

BGTR_TAG

UINT16 get_gColdStartAttempts ()

Get the max number of coldstart attempts a node is allowed to make.

Returns:

the maximum number of attempts

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.2

CSMR_TAG

UINT8 get_gdActionPointOffset ()

Get the action point offset in macroticks.

Returns:

number of macroticks

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

UINT16 get_gdBit ()

Get the bit duration in microticks.

Returns:

number of microticks

See also:

FlexRay Protocol specification v2.1 section B.2.2

BDR_TAG

UINT8 get_gdDynamicSlotIdlePhase ()

Get the number of minislot constituting the duration of the idle phase within a dynamic slot.

Returns:

idlephase in minislots

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

IDLR_TAG

UINT16 get_gdMaxDrift ()

Get the maximum frequency deviation of the oscillator clock.

Returns:

number of microticks

See also:

FlexRay Protocol specification v2.1 section B

MCLDAR_TAG

UINT8 get_gdMinislot ()

Get the length of a minislot in macroticks.

Returns:

the number of macroticks

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

UINT8 get_gdMinislotActionPointOffset ()

Get the number of macroticks constituting the offset of the action point within a minislot of the dynamic slot.

Returns:

offset in macroticks

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

MSAPOR_TAG

UINT16 get_gdNIT ()

Get the start cycle time of the network idle time.

Returns:

number of macroticks

See also:

FlexRay Protocol specification v2.1 section B

NITCR_TAG

UINT8 get_gdStaticSlot ()

Get the length of a static slots in macroticks.

Returns:

the number of macroticks

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

SSLR_TAG

UINT16 get_gdSymbolWindow ()

Get the start time in macroticks for the symbol window during the communication cycle.

Warning:

The gdSymbolWindow protocol definition is the duration of the symbol window. MFR4200 uses the start time in macroticks for the symbol window within the communication cycle instead.

Returns:

number of macroticks

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

SWCR_TAG

UINT8 get_gdTSSTransmitter ()

Get the number of bits within the transmitter start sequence for transmission.

Returns:

the numbre of symbols

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.3

TSSLR_TAG

UINT8 get_gdWakeupSymbolTxIdle ()

Get the duration of the idle period for wakeup symbols expressed in bit durations for the network.

Returns:

the number of bit durations

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.4

WUSTXIR_TAG

UINT8 get_gdWakeupSymbolTxLow ()

Get the duration of the low period for wakeup symbols expressed in bit durations for the network.

Returns:

the number of bit durations

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.4

WUSTXLR_TAG

UINT8 get_gListenNoise ()

Get duration in cycles of listen timeout with noise.

Returns:

the duration in cycles

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.2

LNLR_TAG

UINT16 get_gMacroPerCycle ()

Get the number of macroticks constituting the duration of the communication cycle.

Returns:

the number of macroticks per cycle

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

UINT16 get_gMaxWithoutClockCorrectionFatal ()

Get maximum number of odd cycles before node enters stop state due to missing sync frame pairs.

Returns:

the number of odd cycles

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.2

MOCWCFR_TAG

UINT16 get_gMaxWithoutClockCorrectionPassive ()

Get maximum number of odd cycles before node enters passive state due to missing sync frame pairs.

Returns:

the number of odd cycles

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.2

MOCWCPR_TAG

UINT16 get_gNumberOfStaticSlots ()

Get the number of static slots in a cycle.

Returns:

number of slots

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

NSSR_TAG

UINT16 get_gOffsetCorrectionStart ()

Get delay in multiples of macroticks after which the offset correction will start.

Returns:

multiples of macroticks.

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.2

SOCCTR_TAG

Section 3.2.3.3.34 in MFR4200 Data Sheet Rev. 0 8/2005.

UINT8 get_gPayloadLengthStatic ()

Get maximum data length for static frames in words.

Returns:

maximum data length or error on failure.

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.3

SPLR_TAG

UINT16 get_gSyncNodeMax ()

Get the maximum number of sync frames that can be transmitted on any channel including the nodes own sync frame.

Returns:

number of frames

See also:

FlexRay Protocol specification v2.1 section B

MSFR_TAG

UINT8 get_isStartupNode ()

Get if this node is a startup node.

Returns:

1/0

See also:

SYNCFR_TAG

UINT8 get_pClusterDriftDamping ()

Get value used to minimize the accumulation of rounding errors in clock synchronization.

Returns:

numer of microticks

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.2

CDDR_TAG

UINT8 get_pDelayCompensationA ()

Get compensation for reception delays on channel A in microticks.

Returns:

the number of microticks

See also:

DCAR_TAG

UINT8 get_pDelayCompensationB ()

Get compensation for reception delays on channel B in microticks.

Returns:

the number of microticks

See also:

DCBR_TAG

UINT8 get_pdTSSReceiver ()

Get the number of bits within the transmitter start sequence for reception.

Returns:

the numbre of symbols

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.3

TSSLR_TAG

UINT16 get_pLatestTx ()

Get the latest time in macroticks at which transmission can start in the dynamic segment.

Returns:

number of macroticks

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

LDTSR_TAG

UINT32 get_pMicroPerCycle ()

Get the number of microticks constituting the duration of the communication cycle.

Returns:

the cycle length in microticks as a 24-bit value

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.1

UINT8 get_pMicroPerMacroNom ()

Get number of microticks per nominal macrotick.

Returns:

the number of microticks

See also:

NMLR_TAG

FlexRay Protocol specification v2.1 section B.3.2

UINT16 get_pOffsetCorrectionOut ()

Get maximum absolute offset correction value in microticks.

Returns:

offset correction value in microticks.

See also:

MOCR_TAG

UINT16 get_pRateCorrectionOut ()

Get maximum permitted absolute rate correction value in microticks.

Returns:

offset rate correction value in microticks.

See also:

MRCR_TAG

UINT8 get_pWakeupChannel ()

Get the channel on which wakeup symbols are sent.

1 means channel A, 2 means channel B and 0 means both channels are disabled.

Returns:

the wakeup channel enable mode

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.2

WMCTRLR_TAG

UINT8 get_pWakeupPattern ()

Get the number of wakeup symbols to be transmitted.

Returns:

the numbre of symbols

See also:

FlexRay Protocol specification v2.1 section 9.3.1.1.4

WMCTRLR_TAG

UINT16 get_SyncFrameHeaderCRC ()

Get header CRC of the sync frame to be transmitted.

Returns:

the crc pattern

See also:

SYNCHR_TAG

UINT16 get_SyncFrameIdentifier ()

Set sync frame idetifier.

Returns:

the sync frame identifier pattern

See also:

SYNCFR_TAG

INT32 set_BusGuardianTick (UINT8 *ticks*)

Set the period length of the bus guardian tick to be provided by the CC to the bus guardian.

This must be in the range [2:16]

Parameters:

ticks period length in microticks

Returns:

the period length. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

BGTR_TAG

INT32 set_gColdStartAttempts (UINT16 *attempts*)

Set the max number of coldstart attempts a node is allowed to make.

Parameters:

attempts number of attempts

Returns:

the set value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.2

CSMR_TAG

INT32 set_gdActionPointOffset (UINT8 *ticks*)

Set the action point offset in macroticks.

Parameters:

ticks number of macrotics

Returns:

number of macroticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

INT32 set_gdBit (UINT16 *ticks*)

Set the bit duration in microticks.

This controls the bus speed. Global cluster parameter.

Parameters:

ticks the number of microticks.

Returns:

number of microticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section B.2.2

BDR_TAG

INT32 set_gdDynamicSlotIdlePhase (UINT8 *slots*)

Set the number of minislot constituting the duration of the idle phase within a dynamic slot.

Slots must be [1:15].

Warning:

This function is linked to gdBit,gdMinislotLength, pMicroPerMacroNom

Parameters:

slots is a 4-bit value in the range [1:15].

Returns:

The idlephase in minislots. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

IDLR_TAG

INT32 set_gdMaxDrift (UINT16 *ticks*)

Set the maximum frequency deviation of the oscillator clock.

Parameters:

ticks the number of microticks.

Returns:

number of microticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section B

MCLDAR_TAG

INT32 set_gdMinislot (UINT8 *ticks*)

Set the length of a minislot in macroticks.

Parameters:

ticks the number of macroticks

Returns:

the number of macroticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

INT32 set_gdMinislotActionPointOffset (UINT8 *offset*)

Set the number of macroticks constituting the offset of the action point within a minislot of the dynamic slot.

Parameters:

offset is a 4-bit value in the range [1:15]

Returns:

offset in macroticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

MSAPOR_TAG

INT32 set_gdNIT (UINT16 *ticks*)

Set the start cycle time of the network idle time.

Time is expressed in macroticks

Parameters:

ticks the number of macroticks.

Returns:

number of macroticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section B

NITCR_TAG

INT32 set_gdStaticSlot (UINT8 *ticks*)

Set the length of a static slots in macroticks.

Parameters:

ticks number of macroticks

Returns:

the number of macroticks. On failure one of the following error codes is returned: ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

SSLR_TAG

INT32 set_gdSymbolWindow (UINT16 *ticks*)

Set the start time in macroticks for the symbol window during the communication cycle.

Warning:

The gdSymbolWindow protocol definition is the duration of the symbol window. MFR4200 uses the start time in macroticks for the symbol window within the communication cycle instead.

Parameters:

ticks the number of macroticks.

Returns:

number of macroticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

SWCR_TAG

INT32 set_gdTSSTransmitter (UINT8 *bits*)

Set the number of bits within the transmitter start sequence for transmission.

Parameters:

bits the number of bits

Returns:

the number of bits. On failure one of the following error codes is returned: ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.3

TSSLR_TAG

INT32 set_gdWakeupSymbolTxIdle (UINT8 *bdurs*)

Set the duration of the idle period for wakeup symbols.

Parameters:

bdurs nr of bit durations of the idle period

Returns:

the number of bit durations. On failure one of the following error codes is returned: ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.4

WUSTXIR_TAG

INT32 set_gdWakeupSymbolTxLow (UINT8 *bdurs*)

Set the duration of the low period of wakeup symbols in terms of bit durations for the network.

Parameters:

bdurs 5-bit value with nr of bit durations of the low period

Returns:

the number of bit durations. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.4

WUSTXLR_TAG

INT32 set_gListenNoise (UINT8 *cycles*)

Set duration in cycles of listen timeout with noise.

Parameters:

cycles 5 bit value, number of cycles.

Returns:

the set value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.2

LNLR_TAG

INT32 set_gMacroPerCycle (UINT16 *cyclelength*)

Set the number of macroticks constituting the duration of the communication cycle.

Parameters:

cyclelength 16-bit value number of macroticks

Returns:

the cycle length. On failure one of the following error codes is returned: ENOT-CONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

INT32 set_gMaxWithoutClockCorrectionFatal (UINT16 *odd*)

Set maximum number of odd cycles before node enters stop state due to missing sync frame pairs.

Parameters:

odd odd cycles, 1:32767

Returns:

the set value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.2

MOCWCFR_TAG

INT32 set_gMaxWithoutClockCorrectionPassive (UINT16 *odd*)

Set maximum number of odd cycles before node enters passive state due to missing sync frame pairs.

Parameters:

odd odd cycles, 1:32767

Returns:

the set value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.2

MOCWCPR_TAG

INT32 set_gNumberOfStaticSlots (UINT16 *slots*)

Define the number of static slots in a cycle.

Parameters:

slots 11-bit value describing number of static slots in a cycle

Returns:

number of slots. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

NSSR_TAG

INT32 set_gOffsetCorrectionStart (UINT16 *macros*)

Set delay in multiples of macroticks after which the offset correction will start.

Parameters:

macros multiplas of macroticks (at most 14-bit value)

Returns:

multiples of macroticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.2

SOCCTR_TAG

Section 3.2.3.3.34 in MFR4200 Data Sheet Rev. 0 8/2005.

Warning:

The formula given in MFR manual is not implemented yet!

INT32 set_gPayloadLengthStatic (UINT8 *words*)

Set maximum data length for static frames in words.

Parameters:

words maximum data length (at most 7-bit value)

Returns:

maximum data length. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h
FlexRay Protocol specification v2.1 section 9.3.1.1.3
SPLR_TAG

INT32 set_gSyncNodeMax (UINT16 *frames*)

Set the maximum number of sync frames that can be transmitted on any channel including the nodes own sync frame.

This number must be in the range [2:15]

Parameters:

frames the number of frames.

Returns:

number of frames. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h
FlexRay Protocol specification v2.1 section B
MSFR_TAG

INT32 set_isStartupNode (UINT8 *isStartup*)

Set if this node is a startup node.

Parameters:

isStartup 1/0 to define if we are startup node.

Returns:

1/0. On failure one of the following error codes is returned: ERANGE ENOT-CONFIG

See also:

mfr_errno.h

SYNCFR_TAG

INT32 set_pClusterDriftDamping (UINT8 *micros*)

Set value used to minimize the accumulation of rounding errors in clock synchronization.

Parameters:

micros number of microticks used, [1:15]

Returns:

numer of microticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.2

CDDR_TAG

INT32 set_pDelayCompensationA (UINT8 *micros*)

Set compensation for reception delays on channel A in microticks.

Parameters:

micros number of microticks [0:127]

Returns:

the set value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

DCAR_TAG

INT32 set_pDelayCompensationB (UINT8 *micros*)

Set compensation for reception delays on channel B in microticks.

Parameters:

micros number of microticks [0:127]

Returns:

the set value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

DCBR_TAG

INT32 set_pdTSSReceiver (UINT8 *bits*)

Set the number of bits within the transmitter start sequence for reception.

Parameters:

bits the number of bits

Returns:

the number of bits. On failure one of the following error codes is returned: ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.3

TSSLR_TAG

INT32 set_pLatestTx (UINT16 *ticks*)

Set the latest time in macroticks at which transmission can start in the dynamic segment.

Warning:

The pLatestTx protocol definition is the number of the last minislot in which transmission can start in the dynamic segment while the MFR4200 implements this using macroticks.

Parameters:

ticks the number of macroticks.

Returns:

number of macroticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

LDTSR_TAG

INT32 set_pMicroPerCycle (UINT32 *cyclelength*, UINT16 *macro_per_cycle*)

Set the number of microticks constituting the duration of the communication cycle.

Parameters:

cyclelength the length of a communication cycle in microticks (24-bit value)

Returns:

cyclelength in microticks. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.1

INT32 set_pMicroPerMacroNom (UINT16 *micro_per_macro*)

Set number of microticks per nominal macrotick.

before calling this function.

Returns:

the number of microticks/macroticks. On failure one of the following error codes is returned: ENOTCONFIG

See also:

mfr_errno.h

NMLR_TAG

FlexRay Protocol specification v2.1 section B.3.2

INT32 set_pOffsetCorrectionOut (UINT16 *micros*)

Set maximum absolute offset correction value in microticks.

Parameters:

micros correction value in microticks

Returns:

the set value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

MOCR_TAG

INT32 set_pRateCorrectionOut (UINT16 *micros*)

Set maximum permitted absolute rate correction value in microticks.

Parameters:

micros rate correction value in microticks

Returns:

the set value. On failure one of the following error codes is returned: ENOT-CONFIG

See also:

mfr_errno.h

MRCR_TAG

INT32 set_pWakeupChannel (UINT8 *mode*)

Set the channel on which a wakeup symbol shall be sent.

1 will enable channel A, 2 will enable channel B and 0 will disable both channels.

Parameters:

mode the channel to be enabled.

Returns:

the correct mode. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.2

WMCTRLR_TAG

INT32 set_pWakeupPattern (UINT8 *symbols*)

Set the number of wakeup symbols to be transmitted.

Parameters:

symbols the number of symbols

Returns:

the number of symbols. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

FlexRay Protocol specification v2.1 section 9.3.1.1.4

WMCTRLR_TAG

INT32 set_SyncFrameHeaderCRC (UINT16 *crc*)

Set header CRC of the sync frame to be transmitted.

Note:

The Sync Frame Header CRC value overwrites the header CRC field of a transmit message buffer that: a) has the same frame ID as the SYNCF[0:10] field of the SYNCFR b) was selected by the CC for transmission as a sync message.

Parameters:

crc an 11-bit pattern

Returns:

the crc pattern. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

SYNCHR_TAG

3.2.3.3.29 Sync Frame Register (SYNCFR

3.2.3.3.30 Sync Frame Header Register (SYNCHR).

INT32 set_SyncFrameIdentifier (UINT16 *pattern*)

Set sync frame idetifier.

Parameters:

pattern an 11-bit pattern

Returns:

the pattern. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h

SYNCFR_TAG

B.3 src/mfr_mb.c File Reference

B.3.1 Detailed Description

Message buffer related functions.

This is specific to the MFR4200 CC, hence the separation from the FlexRay procotol functions. There is functionality for configuring the message buffer map layout in the CC, as well as for reception and transmission of FlexRay frames via the buffers. The FIFO- related functions have been separated and put into **mfr_mb.c**(p. 98) for clarity and structure.

Author:

Jimmy Myhrman (`jimmy@myhrman.org`)
Nikola Vorkapic (`nikola@vorkapic.com`)
Petter Uvesten
Erik Bengtsson

Date:

2005-12-02

```
#include "mfr_mb.h"
```

```
#include "mfr_mb_fifo.h"
```

Functions

- int **fr_config_rxbuf** (int buf_id, mb_ch_t ch_filter, UINT16 frame_id)

  Configures an RX buffer.

- int **fr_config_txbuf** (int buf_id, mb_ch_t ch_filter, mb_txtype_t TT, int payload_-preamble, UINT16 frame_id, int payload_length)

  Configures an TX buffer.

- int **mb_buf_config** (int buf_id, mb_buftype_t buf_type, mb_ch_t ch_filter, mb_-txtype_t TT, int payload_preamble, UINT16 frame_id, int payload_length)

  Configures a message buffer.

- int **fr_dynamic_reconfigure** (int buf_no, mb_frame_t *frame)

  Reconfigures a buffer in the dynamic segment.

- INT32 **mb_rxbuf_read** (mb_frame_t *frame)

  *Reads a frame from the the rxbuf specified in *frame.*

- void **mb_active_rxbuf_read** (mb_frame_t *frame)

  Reads a frame from the active Receive Buffer Window.

- INT32 **mb_buf_lock** (UINT8 buf_id)

  Locking mechanism for the message buffers.

- INT32 **mb_buf_unlock** (UINT8 buf_id)

  Unlocking mechanism for the message buffers.

- void **mb_buf_commit** (UINT8 buffer_id)

  Commits a TX buffer for transmission.

- int **mb_buf_has_new_data** (int buf_no)

  Checks if a RX buffer contains any new data.

- int **fr_read** (int buf_no, int size, UINT16 *data)

  Read the data in a receive buffer.

- int **fr_read_with_header** (int buf_no, mb_frame_t *frame, int size, UINT16 *data)

  Read the data in a receive buffer, including header data.

- int **fr_write** (int buf_no, int size, UINT16 *data)

  Send a frame over the flexray bus using the specified buffer.

- UINT16 **mb_calc_header_crc** (UINT8 SyFIndicator, UINT8 SuFIndicator, UINT16 FrameID, UINT8 PayloadLength)

  Calculates the Header CRC for a frame.

B.3.2 Function Documentation

int fr_config_rxbuf (int *buf_id*, mb_ch_t *ch_filter*, UINT16 *frame_id*)

Configures an RX buffer.

: doc

int fr_config_txbuf (int *buf_id*, mb_ch_t *ch_filter*, mb_txtype_t *TT*, int *payload_preamble*, UINT16 *frame_id*, int *payload_length*)

Configures an TX buffer.

: doc

int fr_dynamic_reconfigure (int *buf_no*, mb_frame_t ∗ *frame*)

Reconfigures a buffer in the dynamic segment.

Parameters:

buf_no The buffer to reconfigure

Returns:

1 on successful reconfiguration, <0 otherwise

Warning:

using this function for a transmit buffer for the static segment will result in an error.

Todo

Error handling.

int fr_read (int *buf_no*, int *size*, UINT16 ∗ *data*)

Read the data in a receive buffer.

Returns:

the number of words read, or 0 if there was no words to read. If the specified buffer doesn't contain any valid data, -1 is returned.

Parameters:

buf_no the rx-buffer

size the size of the data array in words

∗*data* the data array to copy the read data to.

Note:

this function locks the buffer, no need to put lock/unlock around it

See also:

MFR4200 data sheet rev.0 page 175

int fr_read_with_header (int *buf_no*, mb_frame_t * *frame*, int *size*, UINT16 * *data*)

Read the data in a receive buffer, including header data.

Returns:

the number of words read, or 0 if there was no words to read

Parameters:

buf_no the rx-buffer

****fram*** the header data struct.

size the size of the data array in words

****data*** the data array to copy the read data to.

Note:

this function locks the buffer, no need to put lock/unlock around it

See also:

MFR4200 data sheet rev.0 page 175

int fr_write (int *buf_no*, int *size*, UINT16 * *data*)

Send a frame over the flexray bus using the specified buffer.

Parameters:

buf_no the buffer

size the length of the data in words

****data*** the data to be sent

Returns:

status code, 1 on successful transmission <0 otherwise.

Note:

the function takes care of buffer locking and committing.

void mb_active_rxbuf_read (mb_frame_t * *frame*)

Reads a frame from the active Receive Buffer Window.

Todo

Documentation and error handling.

void mb_buf_commit (UINT8 *buffer_id*)

Commits a TX buffer for transmission.

Todo

Check if TX buf, check if valid and so on...

int mb_buf_config (int *buf_id*, mb_buftype_t *buf_type*, mb_ch_t *ch_filter*, mb_txtype_t *TT*, int *payload_preamble*, UINT16 *frame_id*, int *payload_length*)

Configures a message buffer.

All configuration data is stored in a message buffer config structure. The configuration procedure follows the steps described in the MFR4200 datasheet.

Note:

a TX-buf for the static segment _cannot_ be reconfigured (I.e. you cannot reset frame id, frame header crc etc.) during the course of normal operations.
you _must_ configure a RX-buffer with an active channel, otherwise it will not receive anything. you must also configure a frame-id for a rx-buffer, or it will be disabled.

Parameters:

buf_id ID of the message buffer to configure.

buf_type Type of buffer (MB_BUFTYPE_RX or MB_BUFTYPE_TX).

ch_filter Channel filter (MB_CH_A, MB_CH_B or MB_CH_AB).

TT Type of transmission for TX buffers (MB_TXTYPE_STATE or MB_TXTYPE_-EVENT).

PP Payload Preamble bit (NMVect/msgID in data0-1).

frame_id Frame ID (11 bits).

paylen Number of 16bit words in payload (7bit value).

See also:

MFR4200 datasheet rev.0 section 3.5.4.

Parameters:

cfg The buffer configuration data.

Returns:

1 if we successfully created a single buffer, 2 if we created a double buffer. On failure one of the following error codes is returned: ENOTCONFIG, ERANGE, EGENERAL, ECHI

See also:

mfr_errno.h

Todo

Functionality for double TX buffers.

int mb_buf_has_new_data (int *buf_no*)

Checks if a RX buffer contains any new data.

This is done by polling the corresponding interrupt flag.

Parameters:

buf_no The buffer ID. Valid range is [0:58].

Returns:

1 If the buffer has new data, or 0 if it doesn't. In case of errors, one of the following error codes can also be returned: ERANGE, EINVALIDPARAM.

INT32 mb_buf_lock (UINT8 *buf_id*)

Locking mechanism for the message buffers.

Tries to set the corresponding LOCK bit in the BUFCSnR register.

Parameters:

buf_id The buffer to lock.

Returns:

The value 1 if the message buffer was locked successfully (or if it was already locked) or ECHI if there was a locking error. In case of locking error, see the CHI Error Register (CHIER) for more detailed info on what caused the locking error.

Note:

If the LOCK bit is already set, it will not be altered.

See also:

MFR4200 data sheet rev.0 section 3.5.3.4.

BUFCSnR_TAG

CHIER_TAG if the function returns the ECHI error code.

INT32 mb_buf_unlock (UINT8 *buf_id*)

Unlocking mechanism for the message buffers.

Tries to clear the corresponding LOCK bit in the BUFCSnR register.

Parameters:

buf_id The buffer to unlock.

Returns:

The value 1 if the message buffer was unlocked successfully, or EGENERAL if the buffer was already unlocked (more of a warning than an error).

See also:

MFR4200 data sheet rev.0 section 3.5.3.4.

BUFCSnR_TAG

CHIER_TAG if the function returns the -4 error code.

UINT16 mb_calc_header_crc (UINT8 *SyFIndicator*, UINT8 *SuFIndicator*, UINT16 *FrameID*, UINT8 *PayloadLength*)

Calculates the Header CRC for a frame.

Parameters:

SyF Sync frame indicator. 1bit value. 0 or 1.

SuF Startup frame indicator. 1bit value. 0 or 1.

FrameID Frame ID. 11bit value.

PayloadLength Payload length. 7bit value denoting the nr of 16bit words in payload.

Returns:

The header CRC (11bit value).

See also:

FlexRay Protocol specification v2.1 section 4.5.2 "Header CRC calculation"

INT32 mb_rxbuf_read (mb_frame_t ∗ *frame*)

Reads a frame from the the rxbuf specified in ∗frame.

Parameters:

∗*frame* frame including the buffer id, and the data are to return value to.

Returns:

1 if the rxbuffer contained valid data, -1 if there was no valid data to read.

Note:

this function locks the buffer, no need to put lock/unlock around it

See also:

MFR4200 data sheet rev.0 page 175

Todo

Documentation and error handling.

B.4 src/mfr_mb_fifo.c File Reference

B.4.1 Detailed Description

Message buffer FIFO related functions.

These are closely related to the general message buffer functions, but were separated in order to keep things structured.

Author:

Jimmy Myhrman (jimmy@myhrman.org)
Petter Uvesten
Mattias Pettersson

Date:

2005-12-02

```
#include "mfr_mb_fifo.h"
```

Functions

- int **mb_FIFO_set_size** (UINT8 size)

  Sets size of the Receive FIFO.

- int **mb_FIFO_get_size** ()

  Gets the size of the FIFO RX buffer.

- int **mb_FIFO_is_not_empty** ()

  Checks whether the FIFO buffer is empty.

- int **mb_FIFO_overrun_detected** ()

  Checks whether the FIFO has been overrun.

- int **mb_FIFO_get_next** (mb_frame_t *frame)

  Gets the next available frame (if any) from the Receive FIFO.

- void **mb_FIFO_get_header** (mb_frame_t *frame)

  Reads from the Active FIFO Message Buffer.

- int **mb_FIFO_set_messageID_acceptance_filter** (UINT16 value)

  Sets the value of the FIFO message ID acceptance filter.

- UINT16 **mb_FIFO_get_messageID_acceptance_filter** ()

Gets the value of the FIFO message ID acceptance filter.

- INT32 **mb_FIFO_set_frameID_rejection_filter** (UINT16 value)

  Sets the value of the FIFO frame ID rejection filter.

- UINT16 **mb_FIFO_get_frameID_rejection_filter** ()

  Sets the value of the FIFO frame ID rejection filter.

- void **mb_FIFO_set_channel_acceptance_filter** (int ChA, int ChB)

  Sets the FIFO channel filter.

- INT32 **mb_FIFO_set_messageID_filter_mask** (UINT16 mask)

  Sets filter mask for the FIFO message ID filter.

- UINT16 **mb_FIFO_get_messageID_filter_mask** ()

  Gets filter mask for the FIFO message ID filter.

- INT32 **mb_FIFO_set_frameID_filter_mask** (UINT16 mask)

  Sets filter mask for the FIFO frame ID filter.

- UINT16 **mb_FIFO_get_frameID_filter_mask** ()

  Gets filter mask for the FIFO frame ID filter.

- int **fr_read_fifo** (int size, UINT16 *data)

  Read the data currently in the FIFO.

- int **fr_read_fifo_with_header** (mb_frame_t *frame, int size, UINT16 *data)

  Read the data and header information currently in the FIFO.

B.4.2 Function Documentation

int fr_read_fifo (int *size*, UINT16 * *data*)

Read the data currently in the FIFO.

Returns:

the number of words read, or 0 if there was no words to read

Parameters:

size the size of the data array in words

∗***data*** the data array to copy the read data to.

Note:

this function locks the buffer, no need to put lock/unlock around it

See also:

MFR4200 data sheet rev.0 page 175

int fr_read_fifo_with_header (mb_frame_t ∗ *frame*, int *size*, UINT16 ∗ *data*)

Read the data and header information currently in the FIFO.

Returns:

the number of words read, or 0 if there was no words to read

Parameters:

∗***frame*** the struct to hold the header information

size the size of the data array in words

∗***data*** the data array to copy the read data to.

Note:

this function locks the buffer, no need to put lock/unlock around it

See also:

MFR4200 data sheet rev.0 page 175

UINT16 mb_FIFO_get_frameID_filter_mask ()

Gets filter mask for the FIFO frame ID filter.

This mask is applied on the frame IDs before they are checked against the FIFO message ID rejection filter.

Returns:

the filter value

See also:

FRFFIDMR_TAG

UINT16 mb_FIFO_get_frameID_rejection_filter ()

Sets the value of the FIFO frame ID rejection filter.

This represents the pattern value of the frame ID to be rejected by FIFO.

Note:

The frame ID is masked by the frame ID mask before checked against this filter.

Returns:

the filter value

See also:

FRFFIDVR_TAG

void mb_FIFO_get_header (mb_frame_t * *frame*)

Reads from the Active FIFO Message Buffer.

Todo

Documentation and error handling.

UINT16 mb_FIFO_get_messageID_acceptance_filter ()

Gets the value of the FIFO message ID acceptance filter.

This represents the acceptable value of the message ID to be accepted by FIFO. message id to be received

Returns:

the filter value

See also:

FAFMIDVR_TAG

UINT16 mb_FIFO_get_messageID_filter_mask ()

Gets filter mask for the FIFO message ID filter.

This mask is applied on the message IDs before they are checked against the FIFO message ID acceptance filter.

Returns:

The bitmask in use.

See also:

FAFMIDVR_TAG

int mb_FIFO_get_next (mb_frame_t * *frame*)

Gets the next available frame (if any) from the Receive FIFO.

The function will try lock the FIFO and then fetch the frame. The buffer locking might fail though, in which case ECHI is returned. In case of locking error, see the CHI Error Register (CHIER) for more detailed info on what caused the error.

Parameters:

frame Pointer to a frame structure into which the new frame data should be put.

Returns:

1 on success, or ECHI if a locking error occurs. Most probably, this means the FIFO is empty (indicated by EFLE-bit in ECHI).

See also:

CHIER_TAG if the function returns the ECHI error code.

Note:

You should call mb_FIFO_is_not_empty prior to this function to make sure one or more frames are available in the FIFO.

See also:

mb_FIFO_is_not_empty(p. 112)

int mb_FIFO_get_size ()

Gets the size of the FIFO RX buffer.

Returns:

The FIFO size. Size 0 means FIFO disabled.

See also:

mb_FIFO_set_size(p. 115)

int mb_FIFO_is_not_empty ()

Checks whether the FIFO buffer is empty.

Returns:

1 if one or more frames are available in the FIFO. 0 otherwise.

int mb_FIFO_overrun_detected ()

Checks whether the FIFO has been overrun.

An overrun occurs if a new frame is accepted by the FIFO, but the FIFO is already full, resulting in the new frame being dropped.

Note:

This functions returns true only once per detected overrun. In other words, the internal overrun intterupt-flag will be cleared by this function.

void mb_FIFO_set_channel_acceptance_filter (int *ChA*, int *ChB*)

Sets the FIFO channel filter.

This determines on which channels the FIFO will listen for incoming frames (i.e. the "accepted" channels).

Parameters:

ChA channel A enable (0/1)

ChB channel A enable (0/1)

INT32 mb_FIFO_set_frameID_filter_mask (UINT16 *mask*)

Sets filter mask for the FIFO frame ID filter.

This mask is applied on the frame IDs before they are checked against the FIFO message ID rejection filter.

Parameters:

mask 11-bit mask for message id rejection.

Returns:

the filter mask. On failure one of the following error codes is returned: ERANGE ENOTCONFIG.

See also:

mfr_errno.h
FRFFIDMR_TAG

INT32 mb_FIFO_set_frameID_rejection_filter (UINT16 *value*)

Sets the value of the FIFO frame ID rejection filter.

This represents the pattern value of the frame ID to be rejected by FIFO.

Note:

The frame ID is masked by the frame ID mask before checked against this filter.

Parameters:

value The frame ID to reject.

Returns:

The filter value. On failure one of the following error codes is returned: ERANGE ENOTCONFIG

See also:

mfr_errno.h
FRFFIDVR_TAG

int mb_FIFO_set_messageID_acceptance_filter (UINT16 *value*)

Sets the value of the FIFO message ID acceptance filter.

This represents the acceptable pattern value of the message ID to be accepted by FIFO.

Note:

The message ID is checked against this filter after the FIFO message ID acceptance mask has been applied.

Parameters:

filter value for message id filtering

Returns:

the filter value. On failure one of the following error codes is returned: ENOTCONFIG

See also:

mfr_errno.h
FAFMIDVR_TAG

INT32 mb_FIFO_set_messageID_filter_mask (UINT16 *mask*)

Sets filter mask for the FIFO message ID filter.

This mask is applied on the message IDs before they are checked against the FIFO message ID acceptance filter.

Note:

Frame ID rejection filter is applied prior to the message ID filter.

Parameters:

mask The bitmask to be used.

Returns:

The bitmask. On failure one of the following error codes is returned: ENOTCONFIG

See also:

mfr_errno.h
FAFMIDMR_TAG

int mb_FIFO_set_size (UINT8 *size*)

Sets size of the Receive FIFO.

The FIFO spans multiple single buffer locations in the MFR4200 and builds up a frame queue with user-definable length. The buffer locations used for FIFO are always msgbuf(0)-msgbuf(size-1). The CC has to be in configuration state before calling this function, or else it will fail updating the FIFO size.

Parameters:

size The FIFO size. Range is [0:59] where 0 means disable FIFO.

Returns:

1 on success, or ENOTCONFIG if CC is not in configuration state.

See also:

mfr_errno.h for detailed info on the error codes.

API Index

chi_prot_cfg.c
- get_BusGuardianTick, 72
- get_gColdStartAttempts, 72
- get_gdActionPointOffset, 73
- get_gdBit, 73
- get_gdDynamicSlotIdlePhase, 73
- get_gdMaxDrift, 74
- get_gdMinislot, 74
- get_gdMinislotActionPointOffset, 74
- get_gdNIT, 74
- get_gdStaticSlot, 75
- get_gdSymbolWindow, 75
- get_gdTSSTransmitter, 75
- get_gdWakeupSymbolTxIdle, 76
- get_gdWakeupSymbolTxLow, 76
- get_gListenNoise, 76
- get_gMacroPerCycle, 77
- get_gMaxWithoutClockCorrectionFatal, 77
- get_gMaxWithoutClockCorrectionPassive, 77
- get_gNumberOfStaticSlots, 78
- get_gOffsetCorrectionStart, 78
- get_gPayloadLengthStatic, 78
- get_gSyncNodeMax, 78
- get_isStartupNode, 79
- get_pClusterDriftDamping, 79
- get_pDelayCompensationA, 79
- get_pDelayCompensationB, 80
- get_pdTSSReceiver, 80
- get_pLatestTx, 80
- get_pMicroPerCycle, 80
- get_pMicroPerMacroNom, 81
- get_pOffsetCorrectionOut, 81
- get_pRateCorrectionOut, 81
- get_pWakeupChannel, 81
- get_pWakeupPattern, 82
- get_SyncFrameHeaderCRC, 82
- get_SyncFrameIdentifier, 82
- set_BusGuardianTick, 83
- set_gColdStartAttempts, 83
- set_gdActionPointOffset, 83
- set_gdBit, 84
- set_gdDynamicSlotIdlePhase, 84
- set_gdMaxDrift, 85
- set_gdMinislot, 85
- set_gdMinislotActionPointOffset, 86
- set_gdNIT, 86
- set_gdStaticSlot, 87
- set_gdSymbolWindow, 87
- set_gdTSSTransmitter, 88
- set_gdWakeupSymbolTxIdle, 88
- set_gdWakeupSymbolTxLow, 88
- set_gListenNoise, 89
- set_gMacroPerCycle, 89
- set_gMaxWithoutClockCorrectionFatal, 90
- set_gMaxWithoutClockCorrectionPassive, 90
- set_gNumberOfStaticSlots, 91
- set_gOffsetCorrectionStart, 91
- set_gPayloadLengthStatic, 91

set_gSyncNodeMax, 92
set_isStartupNode, 92
set_pClusterDriftDamping, 93
set_pDelayCompensationA, 93
set_pDelayCompensationB, 94
set_pdTSSReceiver, 94
set_pLatestTx, 94
set_pMicroPerCycle, 95
set_pMicroPerMacroNom, 95
set_pOffsetCorrectionOut, 96
set_pRateCorrectionOut, 96
set_pWakeupChannel, 96
set_pWakeupPattern, 97
set_SyncFrameHeaderCRC, 97
set_SyncFrameIdentifier, 98

fr_config_rxbuf
mfr_mb.c, 100
fr_config_txbuf
mfr_mb.c, 100
fr_dynamic_reconfigure
mfr_mb.c, 101
fr_read
mfr_mb.c, 101
fr_read_fifo
mfr_mb_fifo.c, 108
fr_read_fifo_with_header
mfr_mb_fifo.c, 109
fr_read_with_header
mfr_mb.c, 102
fr_write
mfr_mb.c, 102

get_BusGuardianTick
chi_prot_cfg.c, 72
get_gColdStartAttempts
chi_prot_cfg.c, 72
get_gdActionPointOffset
chi_prot_cfg.c, 73
get_gdBit
chi_prot_cfg.c, 73
get_gdDynamicSlotIdlePhase
chi_prot_cfg.c, 73
get_gdMaxDrift
chi_prot_cfg.c, 74
get_gdMinislot
chi_prot_cfg.c, 74
get_gdMinislotActionPointOffset
chi_prot_cfg.c, 74
get_gdNIT
chi_prot_cfg.c, 74
get_gdStaticSlot
chi_prot_cfg.c, 75
get_gdSymbolWindow
chi_prot_cfg.c, 75
get_gdTSSTransmitter
chi_prot_cfg.c, 75
get_gdWakeupSymbolTxIdle
chi_prot_cfg.c, 76
get_gdWakeupSymbolTxLow
chi_prot_cfg.c, 76
get_gListenNoise
chi_prot_cfg.c, 76
get_gMacroPerCycle
chi_prot_cfg.c, 77
get_gMaxWithoutClockCorrectionFatal
chi_prot_cfg.c, 77
get_gMaxWithoutClockCorrectionPassive
chi_prot_cfg.c, 77
get_gNumberOfStaticSlots
chi_prot_cfg.c, 78
get_gOffsetCorrectionStart
chi_prot_cfg.c, 78
get_gPayloadLengthStatic
chi_prot_cfg.c, 78
get_gSyncNodeMax

chi_prot_cfg.c, 78
get_isStartupNode
chi_prot_cfg.c, 79
get_pClusterDriftDamping
chi_prot_cfg.c, 79
get_pDelayCompensationA
chi_prot_cfg.c, 79
get_pDelayCompensationB
chi_prot_cfg.c, 80
get_pdTSSReceiver
chi_prot_cfg.c, 80
get_pLatestTx
chi_prot_cfg.c, 80
get_pMicroPerCycle
chi_prot_cfg.c, 80
get_pMicroPerMacroNom
chi_prot_cfg.c, 81
get_pOffsetCorrectionOut
chi_prot_cfg.c, 81
get_pRateCorrectionOut
chi_prot_cfg.c, 81
get_pWakeupChannel
chi_prot_cfg.c, 81
get_pWakeupPattern
chi_prot_cfg.c, 82
get_SyncFrameHeaderCRC
chi_prot_cfg.c, 82
get_SyncFrameIdentifier
chi_prot_cfg.c, 82

mb_active_rxbuf_read
mfr_mb.c, 102
mb_buf_commit
mfr_mb.c, 103
mb_buf_config
mfr_mb.c, 103
mb_buf_has_new_data
mfr_mb.c, 104
mb_buf_lock
mfr_mb.c, 104
mb_buf_unlock
mfr_mb.c, 105
mb_calc_header_crc
mfr_mb.c, 105
mb_FIFO_get_frameID_filter_mask
mfr_mb_fifo.c, 109
mb_FIFO_get_frameID_rejection_filter
mfr_mb_fifo.c, 110
mb_FIFO_get_header
mfr_mb_fifo.c, 110
mb_FIFO_get_messageID_acceptance_filter
mfr_mb_fifo.c, 110
mb_FIFO_get_messageID_filter_mask
mfr_mb_fifo.c, 110
mb_FIFO_get_next
mfr_mb_fifo.c, 111
mb_FIFO_get_size
mfr_mb_fifo.c, 111
mb_FIFO_is_not_empty
mfr_mb_fifo.c, 112
mb_FIFO_overrun_detected
mfr_mb_fifo.c, 112
mb_FIFO_set_channel_acceptance_filter
mfr_mb_fifo.c, 112
mb_FIFO_set_frameID_filter_mask
mfr_mb_fifo.c, 112
mb_FIFO_set_frameID_rejection_filter
mfr_mb_fifo.c, 113
mb_FIFO_set_messageID_acceptance_filter
mfr_mb_fifo.c, 113
mb_FIFO_set_messageID_filter_mask
mfr_mb_fifo.c, 114
mb_FIFO_set_size
mfr_mb_fifo.c, 114
mb_rxbuf_read

mfr_mb.c, 106
mfr_mb.c
fr_config_rxbuf, 100
fr_config_txbuf, 100
fr_dynamic_reconfigure, 101
fr_read, 101
fr_read_with_header, 102
fr_write, 102
mb_active_rxbuf_read, 102
mb_buf_commit, 103
mb_buf_config, 103
mb_buf_has_new_data, 104
mb_buf_lock, 104
mb_buf_unlock, 105
mb_calc_header_crc, 105
mb_rxbuf_read, 106
mfr_mb_fifo.c
fr_read_fifo, 108
fr_read_fifo_with_header, 109
mb_FIFO_get_frameID_filter_mask, 109
mb_FIFO_get_frameID_rejection_filter, 110
mb_FIFO_get_header, 110
mb_FIFO_get_messageID_acceptance_filter, 110
mb_FIFO_get_messageID_filter_mask, 110
mb_FIFO_get_next, 111
mb_FIFO_get_size, 111
mb_FIFO_is_not_empty, 112
mb_FIFO_overrun_detected, 112
mb_FIFO_set_channel_acceptance_filter, 112
mb_FIFO_set_frameID_filter_mask, 112
mb_FIFO_set_frameID_rejection_filter, 113
mb_FIFO_set_messageID_acceptance_filter, 113
mb_FIFO_set_messageID_filter_mask, 114
mb_FIFO_set_size, 114

set_BusGuardianTick
chi_prot_cfg.c, 83
set_gColdStartAttempts
chi_prot_cfg.c, 83
set_gdActionPointOffset
chi_prot_cfg.c, 83
set_gdBit
chi_prot_cfg.c, 84
set_gdDynamicSlotIdlePhase
chi_prot_cfg.c, 84
set_gdMaxDrift
chi_prot_cfg.c, 85
set_gdMinislot
chi_prot_cfg.c, 85
set_gdMinislotActionPointOffset
chi_prot_cfg.c, 86
set_gdNIT
chi_prot_cfg.c, 86
set_gdStaticSlot
chi_prot_cfg.c, 87
set_gdSymbolWindow
chi_prot_cfg.c, 87
set_gdTSSTransmitter
chi_prot_cfg.c, 88
set_gdWakeupSymbolTxIdle
chi_prot_cfg.c, 88
set_gdWakeupSymbolTxLow
chi_prot_cfg.c, 88
set_gListenNoise
chi_prot_cfg.c, 89
set_gMacroPerCycle

chi_prot_cfg.c, 89
set_gMaxWithoutClockCorrectionFatal
chi_prot_cfg.c, 90
set_gMaxWithoutClockCorrectionPassive
chi_prot_cfg.c, 90
set_gNumberOfStaticSlots
chi_prot_cfg.c, 91
set_gOffsetCorrectionStart
chi_prot_cfg.c, 91
set_gPayloadLengthStatic
chi_prot_cfg.c, 91
set_gSyncNodeMax
chi_prot_cfg.c, 92
set_isStartupNode
chi_prot_cfg.c, 92
set_pClusterDriftDamping
chi_prot_cfg.c, 93
set_pDelayCompensationA
chi_prot_cfg.c, 93
set_pDelayCompensationB
chi_prot_cfg.c, 94
set_pdTSSReceiver
chi_prot_cfg.c, 94
set_pLatestTx
chi_prot_cfg.c, 94
set_pMicroPerCycle
chi_prot_cfg.c, 95
set_pMicroPerMacroNom
chi_prot_cfg.c, 95
set_pOffsetCorrectionOut
chi_prot_cfg.c, 96
set_pRateCorrectionOut
chi_prot_cfg.c, 96
set_pWakeupChannel
chi_prot_cfg.c, 96
set_pWakeupPattern
chi_prot_cfg.c, 97
set_SyncFrameHeaderCRC
chi_prot_cfg.c, 97
set_SyncFrameIdentifier
chi_prot_cfg.c, 98
src/chi_prot_cfg.c, 65
src/mfr_mb.c, 98
src/mfr_mb_fifo.c, 106