

ISSN 0280-5316
ISRN LUTFD2/TFRT--5688--SE

Distributed Brake-By-Wire Based on TTP/C

Maria Bruce

Department of Automatic Control
Lund Institute of Technology
June 2002

Department of Automatic Control Lund Institute of Technology Box 118 SE-221 00 Lund Sweden		<i>Document name</i> MASTER THESIS	
		<i>Date of issue</i> June 2002	
		<i>Document Number</i> ISRN LUTFD2/TFRT---5688--SE	
<i>Author(s)</i> Maria Bruce		<i>Supervisor</i> Olof Bridal, Volvo Karle-Erik Årzén and Magnus Gäfvert, LTH	
		<i>Sponsoring organization</i>	
<i>Title and subtitle</i> Distributed Brake-By-Wire based on TTP/C. (Distribuerat Brake-By-Wire system baserat på TTP/C).			
<i>Abstract</i> <p>Brake-By-Wire means that the hydraulic-mechanical brake system in a car is partly or completely replaced with an electronic/electromechanical brake system. Brake-By-Wire systems are safety critical. A Brake-By-Wire system must also be fault tolerant and have a dependable real-time performance. One communication protocol that is developed to meet these requirements is called TTP/C (Time Triggered Protocol, class C). It is a protocol where all operations are initiated and scheduled in advance.</p> <p>In this Master's thesis, I have put a TTP/C system into operation and developed a simple, fault tolerant Brake-By-Wire system. The work is concentrating on the fault tolerant and the fault detection aspects. Also the TTP/C concept with its software development environment has been evaluated.</p> <p>Using TTP/C as effectively as possible requires that the developer has planned the implementation of the system carefully. The work with the TTP/C software development tools will be made easier the more that is known in advance about the system. A more detailed documentation than presently available of how to use the TTP/C tools would also facilitate the work.</p>			
<i>Keywords</i>			
<i>Classification system and/or index terms (if any)</i>			
<i>Supplementary bibliographical information</i>			
<i>ISSN and key title</i> 0280-5316			<i>ISBN</i>
<i>Language</i> English	<i>Number of pages</i> 105	<i>Recipient's notes</i>	
<i>Security classification</i>			

The report may be ordered from the Department of Automatic Control or borrowed through:
University Library 2, Box 3, SE-221 00 Lund, Sweden
Fax +46 46 222 44 22 E-mail ub2@ub2.se

Table of contents

1	INTRODUCTION	5
1.1	PROBLEM FORMULATION	5
1.2	ORGANISATION OF THE REPORT	6
1.3	ACKNOWLEDGEMENTS.....	7
2	BRAKE-BY-WIRE	8
2.1	REQUIREMENTS FOR AUTOMOTIVE BRAKE-BY-WIRE SYSTEMS	8
2.2	REASONS TO USE BRAKE-BY-WIRE.....	9
3	TTP/C	10
3.1	BACKGROUND	10
3.2	FUNCTIONAL DESCRIPTION.....	11
3.3	SOFTWARE TOOLS.....	13
3.4	TTP/C HARDWARE.....	15
3.5	POTENTIAL ALTERNATIVES TO TTP/C.....	16
3.5.1	<i>FlexRay</i>	16
3.5.2	<i>CAN and TTCAN</i>	17
4	REALISATION OF A BRAKE-BY-WIRE SYSTEM	19
4.1	BRAKE-BY-WIRE DESIGN	19
4.2	BRAKE-BY-WIRE SCHEDULING.....	20
4.3	IMPLEMENTATION AND FAULT TOLERANCE ASPECTS OF THE SUBSYSTEMS.....	23
4.3.1	<i>Valid pedal sensor values</i>	24
4.3.2	<i>Calculation of brake force</i>	31
4.3.3	<i>Status messages of the system</i>	32
4.3.4	<i>Applying of the brake force at the wheels</i>	33
4.4	IMPLEMENTATION OF THE I/O CPU.....	34
4.5	COMPILATION, LINKING AND DOWNLOADING.....	35
4.6	SIGNAL GENERATOR	37
5	CONCLUSION AND EVALUATION	38
5.1	TTP/C CONCEPT.....	38
5.2	TTP/C SOFTWARE DEVELOPMENT ENVIRONMENT.....	39
6	DISCUSSION	41
7	FUTURE WORK	43
	APPENDIX A	47
	APPENDIX B	51
	APPENDIX C	56
	APPENDIX D	98
	REFERENCES	105

List of figures

<i>Figure 3.1 Computational cluster. Figure taken from [4]</i>	11
<i>Figure 3.2 Architecture of computational cluster. Figure taken from [2]</i>	12
<i>Figure 3.3 Overview of how and in what order the software tools are used. Figure taken from [5]</i>	14
<i>Figure 3.4 The MEDL scheme is transferred via an Ethernet link to the TTPmonitoring node, which then informs all other nodes in the system. Figure taken from [5]</i>	15
<i>Figure 3.5 TTPnode architecture. Figure taken from [6]</i>	16
<i>Figure 4.1 The Brake-By-Wire system design of this Master's thesis.</i>	20
<i>Figure 4.2 The TTP/C communication schedule for the Brake-By-Wire system created in TTPplan.</i>	22
<i>Figure 4.3 Scheme over what sensor values the pedal nodes sample and how they exchange the values with each other.</i>	25
<i>Figure 4.4 Diagram over the ratio between the pedal position and the pedal force. Values < 100 or > 923 are considered as incorrect values and indicates that there is something wrong with the sensors.</i>	27
<i>Figure 4.5 The sensor values are unified to the values PF and FF.</i>	27
<i>Figure 4.6 Summary of the main steps from the point where the pedal node has four sensor values till the point where the final sensor value is calculated.</i>	30
<i>Figure 4.7 The status messages shown in TTPview.</i>	32
<i>Figure 4.8 The backside of the signal generator. The output signals are numbered according to the figure.</i>	37
<i>Figure 7.1 Example of pedal position and pedal force diagram.</i>	45

List of equations

<i>Equation 4.1 The equation used to calculate the final sensor value from the pedal force value and the pedal position/force value.</i>	28
--	----

1 Introduction

The number of safety related electronic systems in vehicles are increasing within the automotive industry. The systems will assist the driver in critical situations and also liberate the driver from routine tasks. To facilitate the integration of these electronic systems, it is in the interest of the automotive industry to replace conventional mechanic or hydraulic systems in vehicles with distributed fault tolerant mechatronic systems. These new systems are called X-By-Wire systems, where X represents the basis of any safety related application like, for example, braking or steering.

By-Wire systems have been used in aircraft construction for years, but their current systems are too expensive to be used outside the aerospace market. The automotive industry needs dependable and cost effective X-By-Wire systems, which are suitable for mass production and easy to maintain. The development of new X-By-Wire systems represents a huge expenditure in advance for a single company. In order to lower the costs, the European vehicle industry share some of their research efforts by setting standards and establishing frameworks for X-By-Wire systems. This resulted in an EU-funded project that took place in 1996 to 1998 and that was called X-By-Wire with a consortium consisting of Volvo, Daimler-Benz (nowadays Daimler-Chrysler), Centro Ricerche Fiat, Ford Europe, Bosch, Mecel and Magneti Marelli, Chalmers University of Technology and Vienna University of Technology. [1]

Some fundamental requirements are that the X-By-Wire systems must be fault tolerant and that the real-time performance must be dependable since most X-By-Wire systems will be safety critical. To meet these requirements a time-triggered communication protocol called TTP/C has been developed. Despite the new communication protocol it will be a great challenge for the automotive industry to make the different X-By-Wire systems reliable and safe. In the last years a new protocol called FlexRay has been developed and this new protocol is now in competition with the TTP/C protocol. It is in this very moment not known which of the protocols that will be used in the future. The development of By-Wire systems in vehicles is still going on.

1.1 Problem formulation

The main goal of this Master's thesis work is to develop a simple Brake-By-Wire system based on TTP/C. This is partly done with some TTP/C development tools. The Brake-By-Wire system will be simulated with a signal generator supplied by Volvo Technological Development. The TTP/C concept will also be evaluated, especially concerning fault tolerance and how the development tools work. Suitable fault tolerance mechanisms on the node and system level will be studied and implemented.

The tasks of this Master's thesis work:

- Study the TTP/C concept
- Study the Brake-By-Wire application
- Put the TTP/C system into operation
- Develop and implement a fault tolerant Brake-By-Wire system
- Connect the Brake-By-Wire system to a signal generator in order to enable experimentation
- Evaluate the TTP/C concept and the TTP/C software development environment
- Write the Master's thesis report

Two earlier Master's thesis works have been done within the area "Distributed systems based on TTP/C" [9], [10]. The work called "Implementation of a distributed Control Application Based on the TTP/C Architecture" was a Master's thesis work which was concentrated on the theory behind a Steer-By-Wire system developed with TTP/C. The work "Distribuerad Brake-By-Wire-demonstrator baserad på TTP/C kommunikation" was an attempt to work with the same tasks as I have tried to do in this Master's thesis work. The problem then was that there was so many bugs in the TTP/C development tools that the work just became a Master's thesis work on how to put a TTP/C system into operation. The TTP/C software development tools have now been further developed for two years and this has made it possible to continue the earlier Master's thesis works with the goal to develop a Brake-By-Wire system.

1.2 Organisation of the report

Chapter 2 describes the distinguishing characteristics for a Brake-By-Wire system. It also lists some of the advantages of a Brake-By-Wire system compared to a hydraulic brake system used in today's cars.

Chapter 3 treats the TTP/C concept. A description of the TTP/C software tools and the TTP/C hardware is also given. Also some potential alternatives to TTP/C are briefly presented.

Chapter 4 describes the Brake-By-Wire system that has been developed during this Master's thesis work. Certain system functions that have been implemented are described more in detail. The chapter also gives a description of practical things like how to compile and link the code, how to flash down the code to the hardware nodes, and how the system is started.

Chapter 5 is about the evaluation of the TTP/C concept and the TTP/C software tools.

Chapter 6 contains some personal thoughts about the Brake-By-Wire concept and TTP/C.

Chapter 7 gives some examples of future work on the Brake-By-Wire system developed during this Master's thesis.

Appendix A lists all subsystems, tasks and messages that are used in the different nodes in the Brake-By-Wire system developed in this Master's thesis work.

Appendix B contains the files that were executed in the program Trace 32. This program was used to flash down the .s19 files into the flash memories of the nodes. The appendix also contains the make files for the compilation and linking of the .c files used in the I/O CPU, Motorola MC68EN376.

Appendix C contains the C code for one of the pedal nodes and one of the wheel brake nodes that were used in the Brake-By-Wire system developed in this Master's thesis work. It also contains the code for the input and output handling in the nodes.

Appendix D shows the schedules for the different tasks in the different nodes.

1.3 Acknowledgements

First of all I would like to thank my advisor Olof Bridal at Volvo Technological Development. His support has made it a pleasure to complete this Master's thesis. I also thank my advisors Karl-Erik Årzén and Magnus Gäfvert both at the Department of Automatic Control at Lund Institute of Technology. They have given me valuable comments on my Master's thesis report. I would also like to express my gratitude to all the people at Volvo Technological Development who have helped me, especially Fredrik Bernin and Henrik Hallberg. I am also grateful to Per Johannessen at Volvo Car Corporation for sharing his experience of development of TTP/C-based systems.

2 Brake-By-Wire

Brake-By-Wire means that the hydraulic brake system in a vehicle is partly or completely replaced by an electromechanical braking system. With this new system the brake pedal generates an electrical signal to act on the electromechanical brake actuators placed at every wheel.

There are two ways to implement a Brake-By-Wire system:

Electro-Hydraulic Brake, EHB - The Brake-By-Wire function is realised through hydraulic pumps and additional electrically controlled valves. There is no mechanical connection between the brake pedal and the hydraulic brake system. Instead the brake pedal sends electrical signals to an actuator that acts on the hydraulic brake system. It is possible to make a hydraulic backup, which delivers an emergency function with reduced brake force. If the system detects a fault the complete electro-hydraulic system will be shut down and a direct hydraulic brake circuit will be closed with the help of some valves. [2]

Electro-Mechanical Brake, EMB – The system is based on electromechanical actuators. The brake force and brake control is realised by electric components. There is no possibility to make a mechanical backup so the system has to be operational even if a fault occurs. [2]

2.1 Requirements for automotive Brake-By-Wire systems

There are high safety requirements on Brake-By-Wire systems. After a fault the system has to be operational until a safe state is reached. The driver must be able to rely as much on the new brake system as he or she does on present brake systems. To get a safe Brake-By-Wire system the communication system must be able to manage hard real-time requirements. The software has to be verified with respect to those requirements. Since the Brake-By-Wire system depends on constant power supply two independent power supplies are also needed. The measurement values from the sensors at the brake pedal and at the wheels also have to be reliable. To meet this request the sensors have to be replicated and compared. Another requirement is that the Brake-By-Wire system must be as good as (or better than) present braking systems in fields like lifetime, availability, maintainability and costs. [2]

2.2 Reasons to use Brake-By-Wire

If the safety requirements can be met, there are several advantages for introducing Brake-By-Wire instead of using hydraulic brakes in vehicles:

- Assistance functions like Anti Blocking Systems, Brake Assistants and Electronic Stability Programs can be realised by using only software and sensors. Additional mechanical or hydraulic components will be superfluous. [2]
- Electrical interfaces instead of hydraulic interfaces allow easier adapting of assistance systems. [2]
- Reduction of packaging problems. There will be fewer parts inside the vehicle if you do not have any mechanical links between the brake components as you have in a hydraulic based system. It will probably also result in lighter weight. [2]
- No brake fluid will be needed. The advantages are that it is ecological and the maintenance is simple. [2]
- No mechanical links between the brake components and the engine compartment, which improves the passive safety. This means that there are no mechanical pieces, which can hurt the driver in a collision. [2]
- Reduced costs for assembly during line production. [2]
- It is easier to adjust the brake system to right steered cars.
- The proportion between the pedal force and the pedal position can be set to a desirable value.
- The pedal vibrations and brake noise during ABS braking will disappear.
- The interface between the driver and the brake can easily be done more flexible. Perhaps a disabled driver could use a joystick instead of a pedal.
- It is easier to maintain and install an electrical system than a hydraulic system.
- Brake-By-Wire will be cheaper than today's system if many assistance functions are required.
- It will be possible to test every single brake system at the supplier.
- When a trailer is involved, brakes will respond simultaneously on both carriage and trailer. [3]
- Minimised brake wear, (spreads load across wheel more evenly). [3]

3 TTP/C

This chapter treats the TTP/C concept. First, the background to TTP/C is presented. Then the functional description follows and the TTP/C software tools and the TTP/C hardware are explained. Finally the chapter ends with some examples of potential alternatives to TTP/C.

3.1 Background

Since Brake-By-Wire is a system with high safety requirements, it is very important to have a fail operational and predictable communication system with high reliability. The Society of Automotive Engineers, SAE, has classified the communication systems into three categories. Class A is for low-speed networks, class B is for high-speed networks with no safety critical requirements and class C has certain stringent safety critical requirements. [3] For a system like Brake-By-Wire a communication system of class C is required.

In a communication system of class C the communication over the network has to be strongly deterministic. It must be possible to develop subsystems independently and integrate them in the system by just plugging them together. The system has to support the connection and distribution of replicas to avoid common mode failures. The communication over the network has to be guarded from for example babbling idiots, which means that one node permanently uses the data bus without letting any other node use the data bus. All nodes have to have clocks, which are synchronised with each other. The system also has to support some kind of membership service, which knows the current status of all connected components in the system. [2]

Some common communication systems used in vehicles are for example CAN, A-BUS, VAN, J1850-DLC and J1850-HBCC. They do not support the class C requirements since they fail in being deterministic, fault-tolerant and fully synchronised. In order to meet these new requirements the University of Vienna and Daimler-Benz Research developed a communication protocol called TTP/C (Time Triggered Protocol, class C). This was done within the framework for two EU funded research projects (X-By-Wire and TTA). [2] In 1998 the company TTTech was founded. This is a high-tech spin-off from the Vienna University of Technology. TTTech develops TTP/C software and hardware.

3.2 Functional description

A system, computational cluster, built with TTP/C consists of a set of nodes (self-contained computers) which communicate via a broadcast data bus using the TTP/C protocol, Figure 3.1.

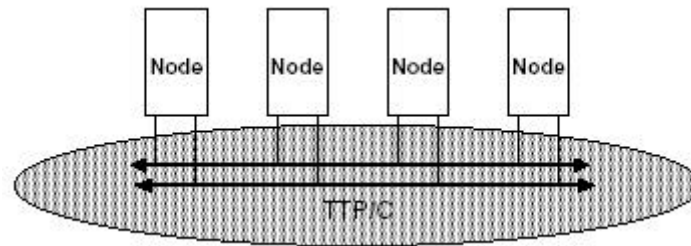


Figure 3.1 Computational cluster. Figure taken from [4]

The TTP/C protocol is time-triggered, which means that all protocol operations are statically scheduled and initiated in advance. As a result all nodes know beforehand when send and receive operations are carried out. Also all tasks are scheduled in advance so that they can deliver and receive the messages on time. It is important that all nodes have the same time reference, a global time. That is why all clocks inside the nodes are synchronised. The synchronisation is possible since all nodes know when a certain node has to send a message. By comparing this time and the time when the node got the message, the receiving node knows the difference between the sender's clock and its own clock.

The computational cluster comprises communication subsystems and host subsystems. The communication subsystem provides a reliable real-time message transmission by executing the TTP/C protocol, while the host subsystems execute the local real-time applications (implemented by the programmer of the system) in every node. The interface between the two subsystems is called Communication Network Interface, CNI, see Figure 3.2.

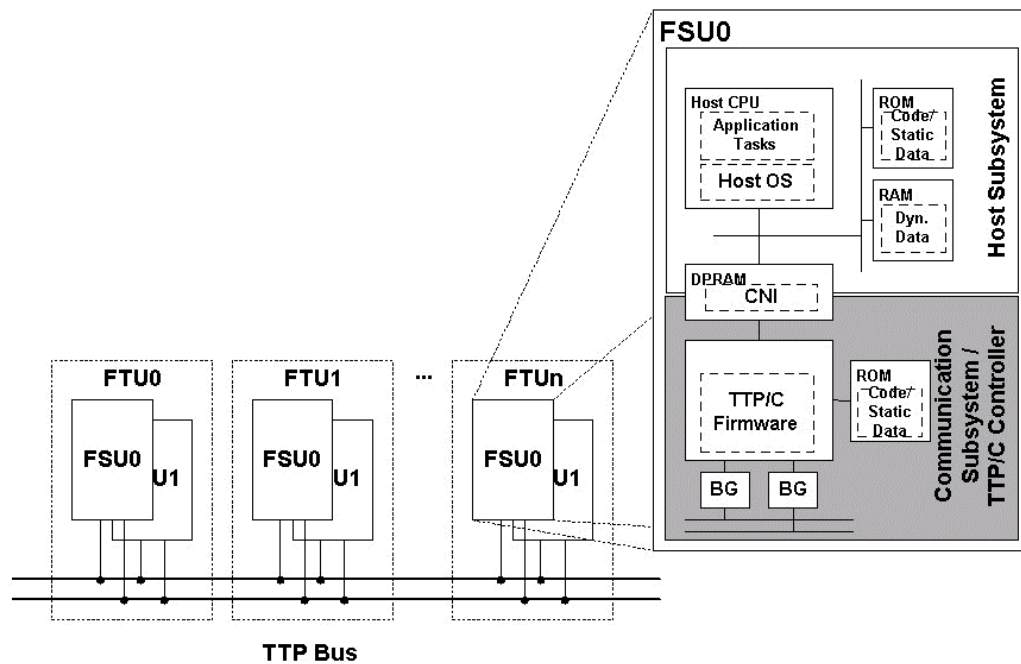


Figure 3.2 Architecture of computational cluster. Figure taken from [2].

The host subsystem and the communication subsystem represent a so-called Fail Silent Unit, FSU. All nodes are assumed to be fail silent, which means that the output of a node has to be correct or otherwise the node has to be silent. A solution to the problem with node failures and communication failures is to replicate every FSU. This new constellation is called a Fault Tolerant Unit, FTU. For an FTU to be operational it is sufficient that one of the FSUs is operational. Since each node is fail-silent, a node failure cannot disturb the communication between the remaining nodes in the system. The message transmission can be replicated by using two data bus channels and by sending every message on both channels.

The nodes follow a TDMA (Time Division Multiple Access) bus access strategy. This means that every node is scheduled to send messages at a predefined time span called a TDMA slot. Only one node is allowed to send messages in every TDMA slot and all nodes send their messages in turn. It is called a TDMA cycle when all nodes have had the possibility to send their messages once. The contents and length of the messages can differ from one TDMA cycle to another but sooner or later the first TDMA cycle will recur again and the set with TDMA cycles start all over again. A set of periodically recurring TDMA cycles is called a cluster cycle.

The Message Descriptor List, MEDL, contains the address and length of each message. It specifies at which instant a message has to be transmitted by the node. The MEDL message schedule is stored locally within each node. Every node is supplied with an autonomous device called a Bus Guardian, BG. The Bus Guardian gives the node permission to transmit only during the node's TDMA slot. This protects the data buses from a timing failure of a node, for example the babbling idiot failure (see Chapter 3.1). One advantage with defining TDMA slots is that it is much easier to integrate different subsystems or nodes, since there is no variance or uncertainty in the communication timing. The pre-determined point in time when the

message is sent also gives a minimal latency jitter, which means that the difference between the longest and the shortest time for a message to be sent and received is minimal.

The TTP/C protocol provides a consistent membership service. All correct nodes learn about node failures or new nodes that are joining the network at the same point in time. It is easy for the system to recognise a node failure since it is known to the system when a node has to send a message.

Messages can be of different frame types. I-frames are initialisation frames and contain the internal state of the TTP node. They are sent by the communication subsystem during start-up and at predefined intervals to allow failed nodes to re-integrate. N-frames (normal frames) contain application data and are sent during normal operation.

3.3 Software tools

In most cases the nodes execute some tasks, which often means that they have to deliver some messages to the other nodes. The tasks are grouped into something called a subsystem. A subsystem is by other words a set of tasks that take some input and produce some output. Several subsystems may be executed on the same node. It is also possible for several nodes to execute the same subsystem at the same time. In that case the subsystem is replicated.

TTTech has developed some software tools, which are used during the system design, Figure 3.3. By using the tools it is possible to allocate subsystems to nodes, and to define which tasks they contain and what kind of messages they are sending at certain points in time. The software tools are also used when it comes to downloading the scheduling information and the contents of the subsystems to the hardware. It is also possible to see what messages that are sent on the data bus between the nodes.

Explanation of the software development tools:

TTPplan

Creates a TTP/C communication schedule, which is the basis for the generation of MEDL files. At this stage the user defines the nodes in the system and what subsystems they use. The user also has to tell TTPplan which messages every subsystem can produce and send. It is possible to graphically show the communication schedule and also edit it graphically.

TTPbuild

The node design tool, which can be used when TTPplan has created the communication schedule. This tool links a node's application software to the TTP/C network and generates a communication driver that handles the interface to the communication system. In TTPbuild the user defines the tasks that every subsystem can execute in order to produce some results or messages. The user also defines what

inputs and outputs the tasks have. TTPbuild generates the configuration for the embedded operating system and schedules the task executions.

TTPos

The operating system that has been developed for fault tolerant real-time applications based on the time-triggered approach. It supports error detection and synchronises an application task's local timer to the global time base.

TTPload

Downloads the cluster and node configurations, which have been generated with TTPplan and TTPbuild. The MEDL scheme gets transferred via Ethernet from the PC you are working on via a TTPmonitoring node to the nodes of the system.

TTPview

A visualisation instrument with a graphical user interface, which shows relevant information about, for example, messages and nodes. This tool supports on-line monitoring of data, which can be shown in diagrams and tables etc. TTPview can record all real-time data and then search through the material off-line to find interesting points. This tool is helpful when you will analyse and debug the TTP/C system.

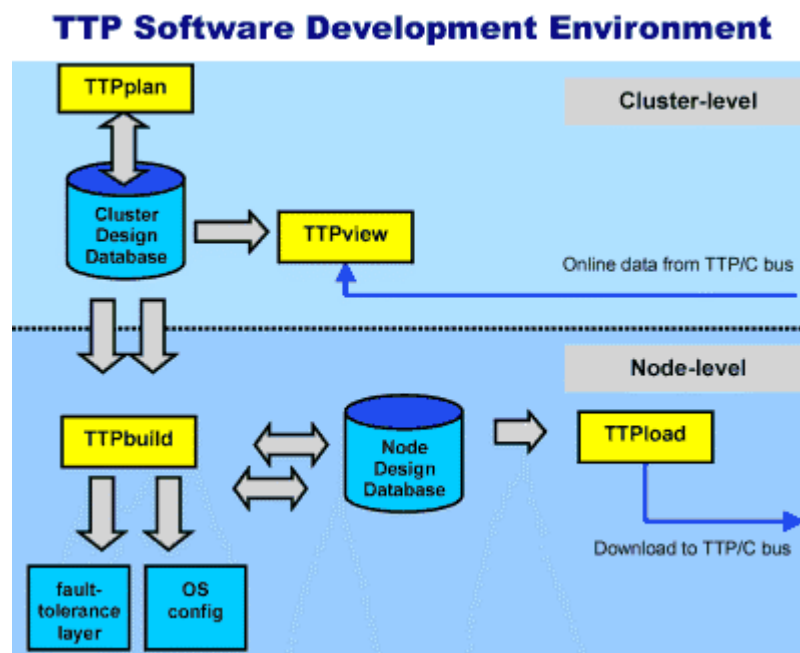


Figure 3.1 Overview of how and in what order the software tools are used. Figure taken from [5].

3.4 TTP/C Hardware

The hardware consists of a number of TTPnodes and a TTPmonitoring node. The TTPmonitoring node has a unique Ethernet MAC address and is provided with software for monitoring and downloading. The MEDL scheme is transferred via a 10 Mbit twisted pair Ethernet link from the PC to the TTPmonitoring node, which then informs all other TTPnodes in the system, see Figure 3.4. The TTPnodes are used for evaluation and prototyping purposes. All nodes are connected to each other via two data buses on which all messages are sent. The TTP/C protocol can only be used in the hardware developed by TTTech. The developer also has to use the TTP/C software development tools that is provided for the scheduling procedure.

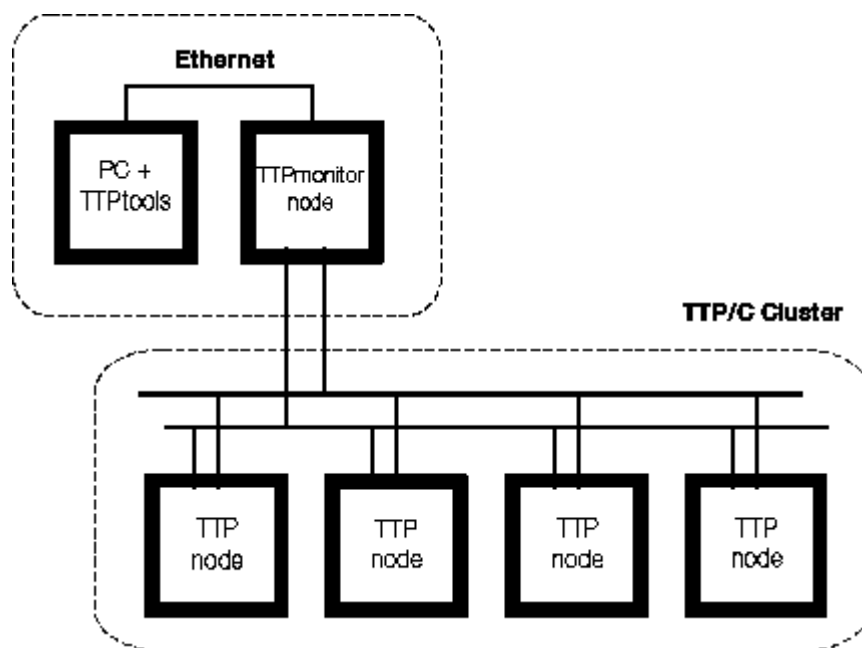


Figure 3.1 The MEDL scheme is transferred via an Ethernet link to the TTPmonitoring node, which then informs all other nodes in the system. Figure taken from [5].

A TTPnode contains a silicon implementation of a TTP communication controller (TTPChip), one host CPU and one I/O CPU. The host CPU is a Motorola MC68360, which runs at 33 MHz and has a static RAM of 128 Kbytes and a Flash Memory of 512 Kbytes. The I/O CPU is a Motorola MC68EN376, which runs at 21 MHz. It has a Flash Memory of 256 Kbytes and a static RAM of 64 Kbytes plus an internal RAM of 4 Kbytes. The I/O CPU also has an 8 Kbytes dual ported RAM to the Host CPU.

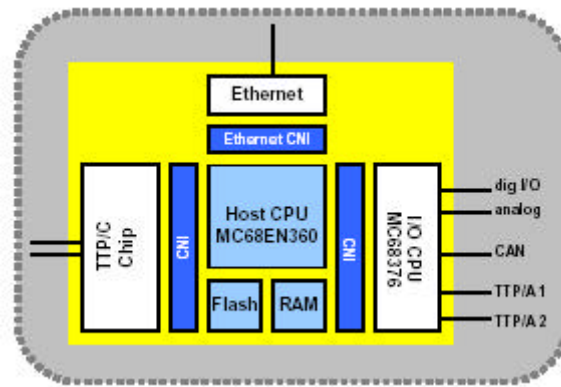


Figure 3.2 TTPnode architecture. Figure taken from [6].

Some of the interfaces that the TTPnode provides:

- 10 Mbit Ethernet Interface
- CAN physical layer Philips 82C250
- Serial communication interface
- 16 analogue inputs
- 4 PWM channels
- 42 digital I/O pins

3.5 Potential alternatives to TTP/C

TTP/C has been known in the academic world for over ten years. A lot of work and time has been put into the TTP/C project and several parts of TTP/C have been formally verified in the terms of mathematical proofs. TTP/C is however not the only protocol that has been developed to meet the demands from systems with hard real-time requirements. During the last years, the time-triggered aspect has also been developed in two competitive protocols named FlexRay and TTCAN.

3.5.1 FlexRay

The FlexRay Group started as a co-operation between BMW and DaimlerChrysler in 1998. Other companies that have joined the consortium since then are, for example, General Motors, Bosch, Motorola and Philips. The major difference between the TTP/C and FlexRay protocols is that the FlexRay protocol is time-triggered and event-triggered at the same time. The protocol has a static segment for static deterministic data transmission and a dynamic segment for dynamic event-triggered data transmission. There is no interference between the static and dynamic segments. The time slots in the static segments are always of the same length. The developer

chooses if only the static segment, only the dynamic segment or both the static and dynamic segments should be used when the system is created. It is also possible for different nodes to use the same time slot for sending messages. If different nodes use the same time slot they have to send on different channels. [7]

The behaviour of the communication will be different at different points in time in a FlexRay system with dynamic segments, since the content in the dynamic segment varies every cluster cycle. The behaviour of the dynamic segment can also change character when more nodes are included in the system. The new nodes might send high priority messages, which will be handled before the messages that were sent in the old system.

Both FlexRay and TTP/C can be used on a data bus of either bus topology or star topology. The central node in the star topology in the FlexRay case does not know when the other nodes are supposed to send their messages. In the TTP/C case the central node knows everything about the MEDL scheme and can detect if a message is sent on the wrong time. FlexRay does not provide functions like, for example, membership agreement handling. If the developer would like to have some membership handling he has to implement the functions for it himself.

Another large difference between FlexRay and TTP/C is that in the FlexRay case every node in the system only knows when the node itself is supposed to send or receive its messages, while all nodes are given a schedule for the whole cluster in the TTP/C case. At system start-up the node in the FlexRay case sees when the messages sent on the data bus arrives and then learns how the scheme for the other nodes look like.

3.5.2 CAN and TTCAN

CAN (Controller Area Network) was developed by Robert Bosch GmbH in 1986. It is a serial data bus system for broadcast messaging. The messages sent on the bus have different priorities and are handled according to these priorities. CAN is very common within today's industry. All messages on CAN are event-triggered. The data messages transmitted from a node do not contain addresses of either the transmitting node or of any receiving node. Instead every data message contains an identifier that is unique throughout the network. Every node checks the identifier to see if the received message is relevant to this particular node.

The standard ISO 11898-4 also called TTCAN (time-triggered CAN) is a further development of CAN. This new protocol allows messages to be sent in three different types of time slots. There are some static time slots, which are scheduled just as in TTP/C, there are time slots where all nodes can try to send their messages according to their priority as in CAN and finally some empty time slots, which are reserved for future use. A master node sends the start time for every cluster cycle to the rest of the nodes in the system.

The TTCAN protocol does not explicitly include a Bus Guardian concept. As a result there seems to be no protection against the babbling idiot problem described in Chapter 3.1. TTCAN can only be used in a data bus with a bus topology. The protocol does not support a data bus with a star topology. The TTCAN protocol (as in the case with the FlexRay protocol) does not provide any membership agreement handling. The present TTCAN protocol does not seem to cover all safety requirements that are necessary for hard real-time systems like, for example, Brake-By-Wire systems. The ISO 11898-4 standard is however still under development. [8]

4 Realisation of a Brake-By-Wire system

This chapter describes the design and implementation of the Brake-By-Wire system that has been developed during this Master's thesis. The different assignments of the nodes and their tasks are explained. Also the general work procedure is described together with some practical tips concerning settings and implementation. This is not a detailed description on how to get a TTP/C cluster to run; instead it explains some important work procedures which are not explained in a satisfying way in the TTP/C manuals.

4.1 Brake-By-Wire design

The Brake-By-Wire system in this Master's thesis consists of six nodes and is implemented as an Electro-Mechanical Brake system. There are four different wheel brake nodes and two pedal nodes, Figure 4.1. The wheel brake nodes should be placed at every wheel where they each should control an electromechanical brake actuator. The pedal nodes are the "intelligent" nodes and contain all important information about the system. All heavy calculations, as well as all decisions about how much brake force the wheel brake nodes should apply, is made by the pedal nodes. Since the pedal nodes are so important there are two of them. The two pedal nodes work in parallel, if one pedal node should go down there is still one working and the brake system can continue to work.

The pedal nodes sample sensor values from the brake pedal. These values will then be exchanged between the pedal nodes and a final sensor value will be calculated. The final sensor value will be used to calculate the brake force that should be applied at every wheel. This brake force will, apart from the brake pedal sensor values, depend on how many brakes and wheel brake nodes that are working. When the brake values have been calculated they are sent as messages to the different wheel brake nodes. The pedal nodes will also collect status information from the wheel brake nodes. This information will be possible to send to some display at the driver's instrumental panel.

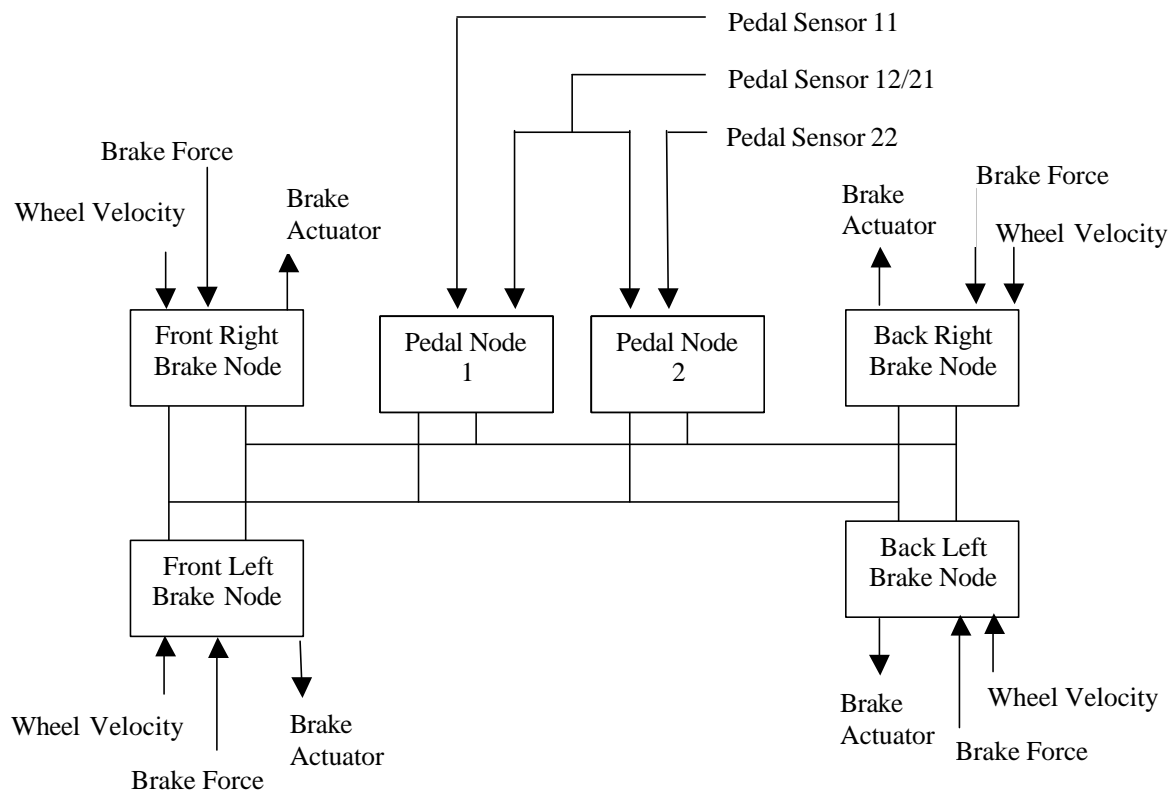


Figure 4.1 The Brake-By-Wire system design of this Master’s thesis.

Sensor values like wheel velocity and brake force will be sampled in the wheel brake nodes. The wheel brake nodes will also have the possibility to shut down the brake if they find that the real brake force is out of range. Besides this function the wheel brake nodes just get the brake instruction messages from the pedal nodes and then try to meet the brake force requirements. They also send some status messages to the pedal nodes.

4.2 Brake-By-Wire scheduling

TTPplan is the first tool to be used when the application is going to be scheduled. It is important to know how many nodes that will be used, what subsystems they will have and what messages they should send. To enable checking that the messages are updated and sent in a correct way it is also possible to define that the messages will have a sender status which must be set every time a message is sent. The more messages that are sent and the more the messages depend on each other, the more difficult it will get for TTPplan to schedule. The user is told to set the `tr_period` in the beginning of the scheduling. This parameter is the time one TDMA cycle will take to execute. Since the six nodes will send many messages the `tr_period` value $2000 \mu\text{s}$ was chosen. The cluster cycle consists of two TDMA cycles, so it will take at most

4000 μ s to execute from the point when the system gets a sensor value from the brake pedal and to the point when the wheel brake nodes give the commands to brake.

Every subsystem must, during the specified TDMA slot, be able to send all messages it is supposed to produce. It is therefore important to keep the subsystems uncomplicated and try not to have subsystems that depend on too many arriving messages, which can be hard to get within the specified time. Since the wheel brake nodes are simple they only have one subsystem each. The pedal nodes are a little bit more complicated. They have one subsystem each to handle sampling of the brake pedal values, one subsystem that handles the final sensor value and brake force computation, and finally, one common subsystem, which handles the status messages to the driver.

If two or more nodes have an identical subsystem, this is defined as a “replicated subsystem”. This means that these subsystems produce the same messages with the same message names. If several messages with the same name are sent on the data bus it can be hard for the other nodes to know which message they should use. In TTPbuild it is possible to set some predefined choices like “take the first one” or “take the mean value” and so on. The problem is that there might have happened something to the value of the message on the way from the sending node to the receiving node. It is therefore desirable to let the different nodes send messages with different names since it then will be easy to do your own comparison and select a valid value. The only way to get messages with different names is to create different subsystems. From a safety point of view it is important that an algorithm for the comparison of messages gets implemented in every node’s software application. It is easy to tell if one node sends strange messages and since it is known which node that sent the messages countermeasures can be taken. All important messages are sent with their own names in this Brake-By-Wire system. TTP/C also gives a possibility to create own algorithms, which automatically will be used by the protocol when it should decide one value from several replicated messages. It is however easier to write this sorting algorithm in the application code for the different nodes as I have done in this Master’s thesis work. It will then also be easier to overview how the algorithm works and which decisions it will make.

Every message must have a C-type. This is also defined in TTPplan. All the different status messages in this Brake-By-Wire system have the size of 1 bit and are defined as UINT in TTPplan and described as a bool in TTPbuild. All value messages have the size 16 bits and they are also defined as UINT in TTPplan but described as an unsigned short int in TTPbuild. The description in TTPbuild is important for the compiler and must agree with the predefined types in the programming language C.

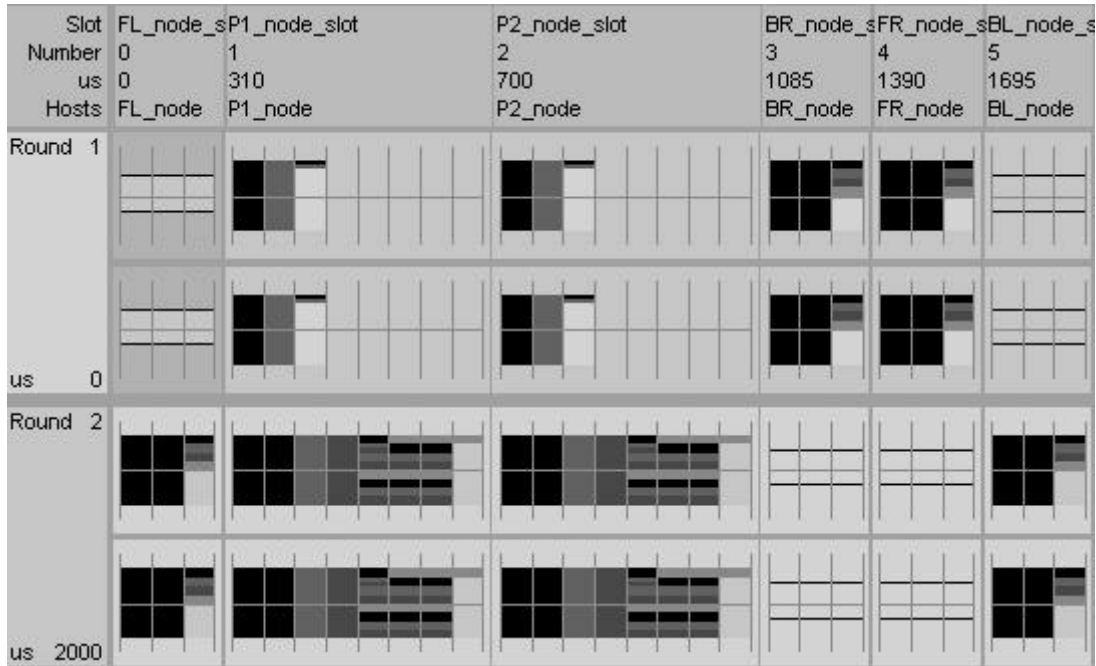


Figure 4.1 The TTP/C communication schedule for the Brake-By-Wire system created in TTPplan.

Figure 4.2 shows the communication schedule for the Brake-By-Wire system used in this Master's thesis. The two TDMA cycles are shown as Round 1 and Round 2 and they are both 2000 μ s long. The TDMA slots of the six nodes are shown in every round and you can also see what frames that are sent. The slot for the FL_node (Front Left node) in Round 1 shows that an I-frame is sent (I-frames are sent during start-up and when a node reintegrates into the system). I-frames are also sent in the BL_node (Back Left node) slot in Round 1 and in the BR_node (Back Right node) and FR_node (Front Right node) slots in Round 2. N-frames are sent in the rest of the slots where it also is possible (when TTPplan is used) to see what messages that are sent.

As seen in Figure 4.2 the first nodes to send messages in the cluster cycle are the two pedal nodes P1_node (Pedal node 1) and P2_node (Pedal node 2). They send their sampled sensor values to each other. Then it is time for the wheel brake nodes BR_node and FR_node to send their status messages to the pedal nodes. Round 2 starts with that the wheel brake node FL_node sends its status messages. Then it is time for the pedal nodes to send some messages again. Now they send the status messages, which can be viewed in TTPview, and the brake force messages to the wheel brake nodes. The last one to send its status messages in the cluster cycle is the wheel brake node BL_node.

When the MEDL schedule has been made in TTPplan it is time to schedule all tasks in every node, in TTPbuild. All tasks, which are executed within every subsystem, are defined and also which messages they send and receive. The user is also supposed to define a time budget for every task, so that the execution of the tasks can be scheduled. This is very hard to do since it is difficult to find out in advance how long

time it will take to execute the C code. TTPbuild is also limited as a consequence of the MEDL schedule. All nodes have already been given a TDMA slot and the tasks must fit into this tight schedule. It is of course more difficult to set long time budgets the more tasks you have and the more messages they are supposed to produce. If it turns out to be too difficult to do the scheduling of the tasks in TTPbuild there is always the possibility to go back to TTPplan and reschedule the TDMA slots for every node. Hopefully this will help to succeed with the task scheduling in TTPbuild.

All subsystems, tasks and messages that are used by every node in this Brake-By-Wire system are declared in Appendix A. The schedules made for every node and every task is shown in Appendix D.

4.3 Implementation and fault tolerance aspects of the subsystems

The implementation of the tasks is made in the programming language C. Every node has to have one main file and at least one .c file with code for the different tasks. Several subsystems can be programmed in the same .c file but it is also possible to write the code for the different subsystems in separate .c files. Every task that was defined in TTPbuild has to be written like:

```
tt_task(taskname)
{
    C code for task
}
```

It is also possible to write functions, which a task can call to do some calculations or other difficult things. The tasks in this project have been programmed so that they call many functions instead of writing all code inside the task. In this way the work of the tasks gets more easy-to-grasp.

The advantage of writing the code for several subsystems in the same .c file is that you can use global variables, which all tasks can reach. This is perhaps not a good idea from a programming safety aspect but it is good from the TTP/C scheduler's point of view. It is possible to let one task receive some messages and then let the same task set some global variables with the message values, which then can be reached by the other tasks. It is of course possible to let the other tasks receive these messages too, but it will be harder to schedule the messages and it might even be impossible for the scheduler to handle all these messages in a satisfying way. A lot of status messages has to be sent over the data bus to keep this Brake-By-Wire system as fault tolerant as possible. It arose big scheduling problems when I tried to let all the different tasks receive all messages that they would need for further calculations. Since the different tasks used the messages in a similar way I let one of the tasks

receive the messages and do the comparison before it set the global variables. In this way a lot of unnecessary calculations were avoided and the scheduling could be done relatively easy.

The TTP/C protocol creates a membership vector that keeps track of which nodes that are working correctly. This vector can also be viewed and used in the tasks. Every time a message is sent the status of the sent message is set to true, since it was declared in TTPplan that every message has a status field. If a status field is declared it is possible to use some predefined TTP/C functions to get information about the received message. The functions tell how many nodes that sent the message and how many replicas of the message that was sent.

A fault tolerant system must be able to handle all sorts of problems in the software and the hardware. The software has to be thoroughly tested with the help of different test programs and simulation models. In this way (hopefully) the major bugs can be found and corrected. Hardware problems are more problematic, since they can occur at any moment without notice once the system has been put into operation. There are several scenarios, which can occur:

- A sensor can be broken and give wrong values.
- The data bus can be broken.
- There might happen something with the messages when they are sent or received in the nodes.
- A node can go down and can stop to work.
- Problems can arise in the electromechanical brake actuator.

The following four chapters describe how the nodes work and how they will react in the hardware breakdown situations described above. For references to the different messages and tasks, see Appendix A.

4.3.1 Valid pedal sensor values

The tasks P1ReadSensorValue and P2ReadSensorValue that are executed by the pedal nodes sample sensor values from the brake pedal. It is very important that the sensor values are valid. It is a great risk to only sample one sensor value. There might be something wrong with the sensor and one could get a sensor value, which is totally out of range without being able to do something about it. Even if one samples two values it might be a problem to decide a valid sensor value if one value is high and one value is low. That is why I sample three values at my brake pedal. The third value can hopefully be decisive if the two other values differ too much.

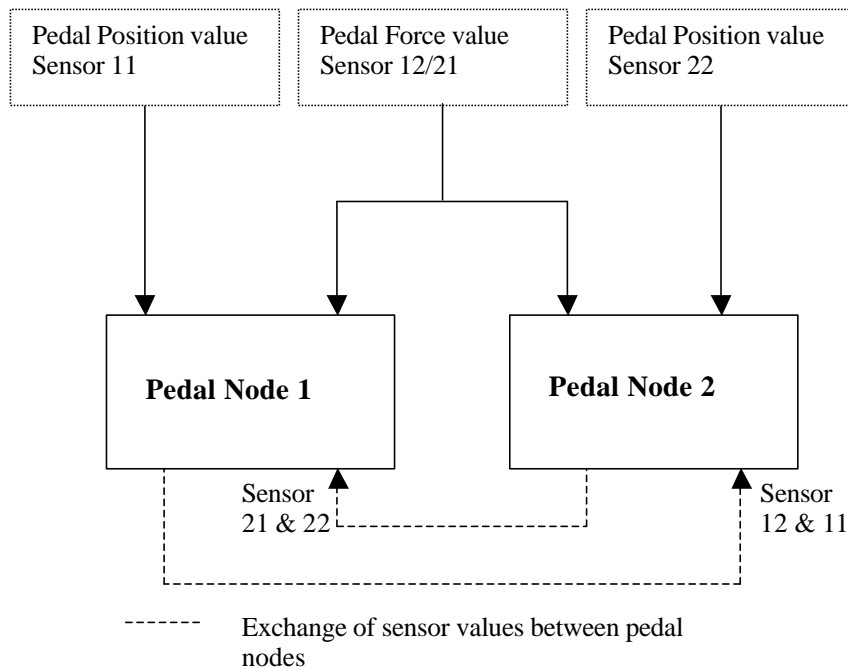


Figure 4.1 Scheme over what sensor values the pedal nodes sample and how they exchange the values with each other.

The values you are able to sample is one pedal force value and two pedal position values. The ReadSensorValue tasks used in the pedal nodes, each sample one of the pedal position values (either from Sensor 11 or Sensor 22 but not from the same sensor), see Figure 4.3. The tasks in the two pedal nodes also sample the same pedal force value from the same sensor but this value is regarded as two different values from two different sensors (called Sensor 12 and Sensor 21), see Figure 4.3. The different sensor values can then be compared to each other in order to find one final, valid sensor value.

The size of the sensor input is 10 bits. This means that the sensor values have a range between 0 and 1023. To sort out the most incorrect values only values between 100 and 923 are used in the calculation of the brake force. The idea is that the sensors and the A/D conversion will be designed so that they never give values outside these limits. If they do there is something wrong with the sensors.

When the first rough sorting is done the two pedal nodes exchange the values with each other and send the values to the CalculateSensorAndBrakeValue tasks (see Appendix A), which use the sensor values to decide a final value. This means that in the end (and in best case) each pedal node has four sensor values to compare with each other. The tasks check how many values that have passed the first rough sorting. If a sensor value has been wrong for more than 2 seconds or if it differs too much

from the other sensor values the system is told to never use a sampled value from that sensor again.

It is possible to check whether the received messages have arrived correctly or not. This is done with a function called `tt_Receiver_Status` provided by TTTech. I have implemented my system in the way that if received sensor value messages are correct they will be used in the further calculations, if not they will be regarded as invalid values (just like the ones that did not pass the first rough sorting) and will not be used.

The next step is to compare the two pedal force values with each other and decide one representative value. The pedal force values are sampled from the same sensor and should for that reason have identical or nearly identical values. One cannot rely on that the values have remained unchanged, so to be on the safe side these two pedal force values have to be regarded as if they were sampled from two different sensors. There are four cases, which must be taken care of:

1. There are two pedal force values, which have identical or nearly identical values.
2. There are two pedal force values with a pronounced difference.
3. Only one pedal force value is valid, the other one has not passed the first sorting.
4. None of the pedal force values have passed the first sorting.

This is the way I have solved the problems:

1. Take the mean value of the two pedal force values.
2. This case is the most complex case. It can be divided into several special cases. The two pedal force values are compared with the two pedal position values if it is possible. Then five possible cases arises:
 - If the position values are similar and if one of the force values corresponds closely to the position values, choose this force value.
 - If the force values are far from the similar position values, the force values will not be used and the mean value of the position values will be used instead.
 - If the position values differ and the force values also differ in the same way so that two groups are formed with one force value and one position value in each, the biggest force value is chosen.
 - If only one position value and one force are close to each other, then this force value is chosen.
 - If the force values differ but they both still lie near one of the separated position values, the force value closest to the position value is chosen.There might be only one position value to compare with. In this case take the closest force value. In the case where all sensor values differ the biggest force value is chosen.
3. The only valid force value is chosen.
4. None of the force values will be used for the final sensor value calculation.

If the pedal position values have passed the first sorting, which means that they are within the range 100 – 923, they have to be transformed into pedal force values since they will be compared with the force sensor value. To the help I was provided with a simple diagram with the ratio between the pedal position and the pedal force, see Figure 4.4. There are only four pairs of co-ordinates in the diagram so I have to

calculate the best corresponding value from these co-ordinates by using linear interpolation. In brake pedals of today there is a natural inertia. The driver has to push the brake pedal to a certain position before any brake force will be applied to the wheels. To resemble current vehicles the Brake-By-Wire system will react in the same way and therefore not brake until the brake pedal has been pushed to the force value 130.

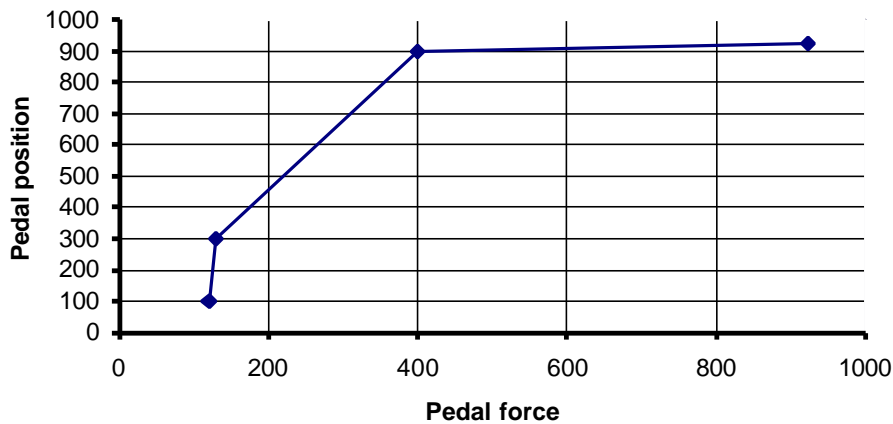


Figure 4.2 Diagram over the ratio between the pedal position and the pedal force. Values < 100 or > 923 are considered as incorrect values and indicates that there is something wrong with the sensors.

In the best-case scenario there are three sensor values left to work with at this stage. Now it is time to decide a final sensor value. This sensor value will be a function (see Equation 4.1) of the pedal force value FF and the transformed pedal position/force value PF, see Figure 4.5.

$$\left. \begin{array}{l} \text{Transformed Pedal Position value, Sensor 11} \\ \text{Transformed Pedal Position value, Sensor 22} \end{array} \right\} \Rightarrow PF$$

$$\left. \begin{array}{l} \text{Pedal Force value, Sensor 12} \\ \text{Pedal Force value, Sensor 21} \end{array} \right\} \Rightarrow FF$$

Figure 4.3 The sensor values are unified to the values PF and FF.

Which of the two values FF and PF that you pay most attention to when you calculate the final sensor value (Equation 4.1) is decided by x, which can take the values between 0 and 1. I have just set x to either 0.25 or 0.75 depending on the FF and PF values. If the mean value of FF and PF is in the first half of the diagram in Figure 4.4 (0 – 399 on the pedal force axis) x is set to 0.25 so that most weight is laid on the PF value. If instead the mean value is in the second half of the diagram in Figure 4.4 (400

– 923 on the pedal force axis) x is set to 0.75. It is probably not enough to let x only vary between two values. The Final Sensor Value will most certainly make a little jump every time x changes value. This could perhaps be prevented by letting x vary in a more continuously way by introducing more possible values for x to adopt.

$$\boxed{\text{Final Sensor Value} = x \cdot FF + (1 - x) \cdot PF}$$

Equation 4.1 The equation used to calculate the final sensor value from the pedal force value and the pedal position/force value.

There are several special cases that must be handled when FF and PF are set:

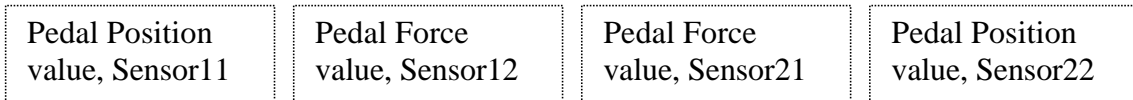
1. Three sensor values available
 1. All three values are close to each other
 2. Only one position/force value matches the force value
 3. The two pedal position/force values are within the same interval but the pedal force value is outside this interval
 4. All sensor values are different
2. Two sensor values available
 1. The pedal force value and the pedal position/force value are within the same interval
 2. The pedal force value and the pedal position/force value are **not** within the same interval
 3. The two pedal position/force values are within the same interval
 4. The two pedal position/force values are **not** within the same interval
3. One sensor value available
 1. One pedal position/force value available
 2. One pedal force value available
4. No sensor values available

The final sensor value will, if it is possible, be calculated according to Equation 4.1. This is how the Brake-By-Wire system will handle the different cases which can occur:

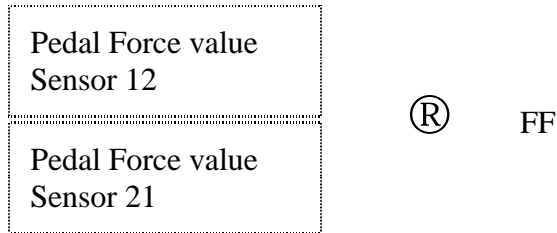
- 1.1 FF = pedal force value
PF = mean value of the pedal position/force values
Final sensor value = Equation 4.1
- 1.2 FF = pedal force value
PF = pedal position/force value
Final sensor value = Equation 4.1
- 1.3 Final sensor value = the pedal force value **or** the mean value of the pedal position/force values that is closest to the last used sensor value
- 1.4 Final sensor value = the pedal force value **or** the pedal position/force value that is closest to the last used sensor value
- 1.1 FF = pedal force value
PF = pedal position/force value
Final sensor value = Equation 4.1

- 1.2 Final sensor value = the pedal force value **or** the pedal position/force value that is closest to the last used sensor value
- 1.3 Final sensor value = the mean value of the pedal position/force values
- 1.4 Final sensor value = the pedal position/force value that is closest to the last used sensor value
- 3.1 Final sensor value = the pedal position/force value available
- 3.2 Final sensor value = the pedal force value available
- 4 Final sensor value = the last used sensor value

When the final sensor value has been calculated it is compared with the last used sensor value. If these two values differ too much the last used sensor value will be used again. For an overview of the calculation from four different sensor values into one final sensor value see Figure 4.6.



Decide if the four sensor values are within the valid range 100 – 923. Make sure that every sensor value comes from a working sensor otherwise do not use that value.

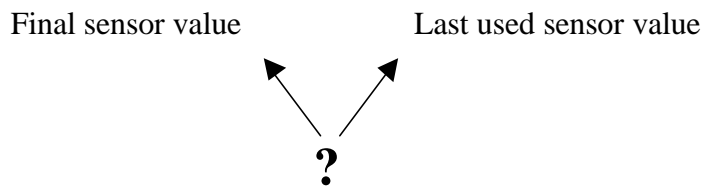


Compare the values from Sensor12 and Sensor 21 with each other and unify them into one value FF.



$$FF + PF + \text{Equation 4.1} \rightarrow \text{Final sensor value}$$

Compare the FF value with the transformed values from Sensor 11 and Sensor 22. Calculate PF and use Equation 4.1 if it is possible. Decide a final sensor value.



Compare the final sensor value with the last used sensor value and decide if the new final sensor value has a reasonable and valid value that should be used instead. If it does not have a reasonable value use the last used sensor value.

Figure 4.4 Summary of the main steps from the point where the pedal node has four sensor values till the point where the final sensor value is calculated.

Since the system will sample the pedal sensor values as often as every 4th ms, there will be no problems if old sample values will be used occasionally in the calculations. If the old sensor value has been used for more than five times in a row the old sensor value will not be used again and a relatively high brake force will be required from the wheel brake nodes. In the case when the brake pedal is pushed very hard, there still should be enough time for the new final sensor value and the old sensor value to be within the value range, which means that they agree. Most of the drive time however the brake pedal remains unused and the sensors will have values corresponding to approximately zero force and zero torque, respectively.

The system will inform the driver if a sensor has failed for more than two seconds, as mentioned earlier. The driver is then supposed to take the car to a servicing garage to get the sensor changed. The situation where more sensors than one is out of order is supposed to be very unusual, but if a system is supposed to be fault tolerant it must be able to handle even this problem in a satisfying way.

4.3.2 Calculation of brake force

The brake force is calculated in the pedal nodes by the P1- and P2CalculateSensorAndBrakeValue tasks. The two pedal nodes each send a brake value to each wheel brake node. It is up to the wheel brake nodes to compare the two brake messages and follow the instructions. If the pedal nodes have detected some internal problems they will send the maximum value ($923 * 64$) to warn the wheel brake nodes from using the brake force values. In this case the wheel brake nodes will not use the brake force value that is sent from the wrong pedal node. If everything works accordingly the wheel brake nodes send a status message, which says that everything runs as normal. This status message and the TTP/C provided membership vector is decisive for the calculations of the brake force.

The membership vector is momentarily changed if a node goes down or up. However the membership vector does not indicate if the tasks in a node work as they should or if there for example is some input problems from a sensor. This software and input check has to be implemented by the programmer. In this Brake-By-Wire system it requires that the membership vector says that the wheel brake node is alive and that the wheel brake node itself says it is running, if a brake value should be calculated and sent to that node. The different cases that can occur are the following:

1. All brakes work
2. One front brake is broken
3. One rear brake is broken
4. Only two diagonal brakes work
5. Only two left brakes or two right brakes work
6. Only two rear brakes work
7. Only two front brakes work
8. Only one front brake works
9. Only one rear brake works
10. None of the brakes works

Every case should have a smart algorithm that calculates how much the different brakes should brake to achieve the best and safest braking. It would also be possible to implement algorithms for the brake force distribution in functions like ABS, anti skid brakes etc. How these smart algorithms should be implemented is however beyond this Master's thesis and therefore not implemented. I have just divided the brake force with the number of working brakes (in the case where there are some broken nodes) to show that it is possible to set different values in the different cases.

4.3.3 Status messages of the system

It is possible to let the driver be informed about the status of the brake system via a display at the instrumental panel. All status messages in the system are produced by the StatusMessages task in the pedal nodes, but as it is now it is only possible to view them in TTPview, Figure 4.7. StatusMessages is the only task, which is common for the two pedal nodes. This also means that all message names are the same. For a detailed list of the different messages see Appendix A.

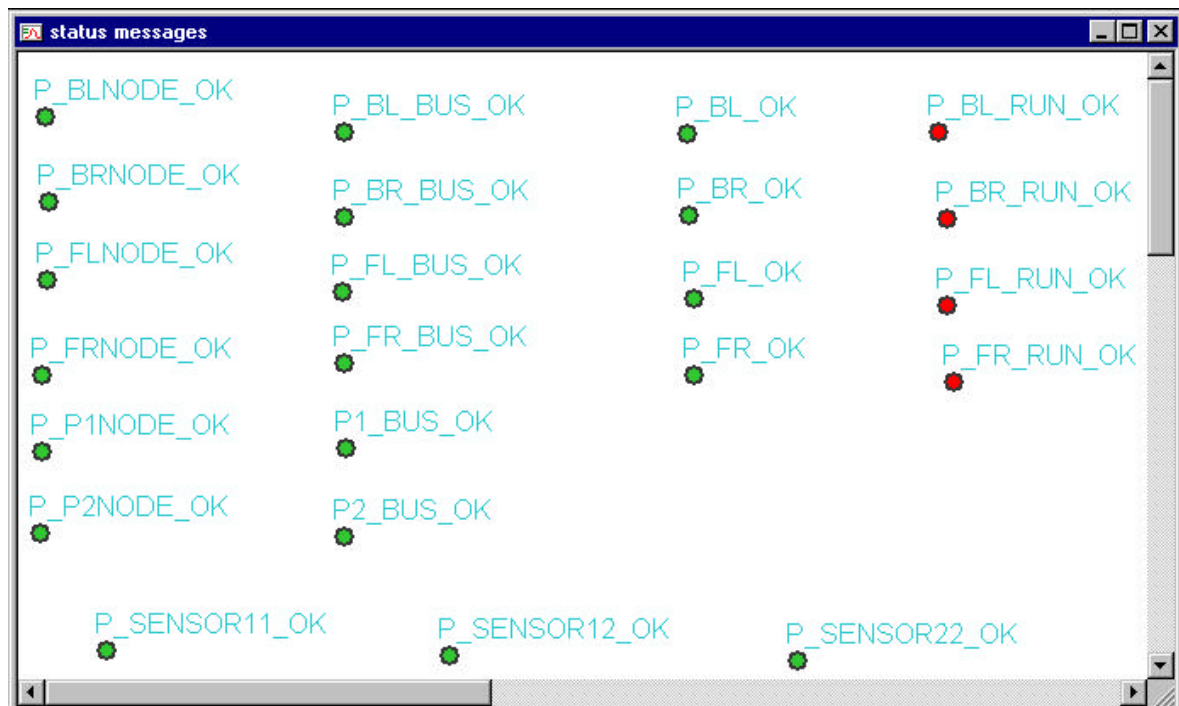


Figure 4.1 The status messages shown in TTPview.

The wheel brake nodes send information about if everything is OK, if they are running and if both of the data bus links work to the pedal nodes. To keep all information in one place it is only the pedal nodes that pass on this information to the

driver. All nodes are also checked in the membership vector and messages are sent to tell if a node is OK or not. The pedal nodes also inform how their own data bus links work.

To see if both of the data bus links work the predefined function `CNI_GET_MSGSTATE_EEIF` was used. This function takes as input the CNI address of an N-frame. The N-frame address given in `TTPplan` is only an offset address so to be able to use this function a CNI base address has to be added. This base address was found in the automatic generated file `ttpv4.h` where also all predefined functions are listed. The predefined function returns the value 8 if the status of the specified N-frame is OK. If the statuses of all N-frames sent on the data bus are OK the `BUS_OK` message is set to true. This means that the two data bus links work normally. If there is something wrong with one or both of the data bus links the `BUS_OK` message is set to false.

The task `StatusMessages` also sends information about the three brake pedal sensors. If a sensor has been incorrect for more than two seconds the sensor is regarded as broken and the sensor is not OK.

4.3.4 Applying of the brake force at the wheels

Every wheel brake node receives two brake force messages, one from each pedal node. The pedal nodes are the “intelligent” nodes in the system so the wheel brake nodes just have to obey the wishes of the pedal nodes. As a result there is no comparison between the last desired brake force and the new required one, to see if the new brake force has any reasonable value. The wheel brake nodes take the mean value of the two brake force values if they are close to each other, in other case the biggest brake force value will be used.

Since I have not been able to run my Brake-By-Wire system with a real actuator, there is no code written for how to translate the requested brake force value into control signals for an electromechanical actuator. Instead a variable has been implemented that contains the desired brake force value. The value of this variable is sent as a PWM (Pulse Width Modulation) signal on the output of the wheel brake nodes. To be able to watch the output as a voltage level instead of a squarewave, a low pass filter has been put on the output of the wheel brake nodes. The voltage level makes it easier to see how the demanded brake force changes in relation to how the input sensor values changes. To be able to steer a brake actuator you have to recalculate the brake force into currents which turn on and off some switches in the actuator and that tells the actuator in which direction it should work (brake harder or release the brake). The regulation of a brake actuator would typically be controlled using feedback from sensors in the brake actuator.

In a final version of the program every wheel brake node compares the desired brake force with the real brake force applied at the wheel. If the difference is too big during a significant time interval, the wheel brake node has the authority to shut down the brake actuator. When the actuator is turned off, the status message `*_RUN_OK` is set

to false and sent to the pedal nodes. The pedal nodes will not use this brake until the brake is fixed and the actuator is turned on again. There is probably something wrong with either the brake force sensor or the actuator if a difference between the real and the desired brake force occurs. There is no input with a real brake force to my wheel brake nodes since I do not have a real brake actuator to interact with. In my program version the *_RUN_OK variables always get the value true. I have marked out the place in the code where the comparison of the brake force values should be done and I have also implemented a comparison function, which is not used in the present program version.

The messages received by the wheel brake nodes are all checked with `tt_Receiver_Status`, which is a function provided by TTTech to see if the message has been sent and received in a proper way. Only brake force messages with an OK status will be used. In the case where there is no valid messages the wheel brake node tells the brake actuator to brake relatively firmly. This braking should be implemented with some smart brake algorithm but I have only indicated where it should be written in the code and then applied full brake instead.

If a pedal node sends brake force messages with the maximum value ($923 * 64$), then this is an indication that there is something wrong with the pedal node and that the wheel brake node should not trust the brake force value. The wheel brake node will continue to follow the brake force command messages as long as at least one brake force message is valid. If both of the pedal nodes are wrong the wheel brake node will apply the brake in a firm way, just as in the case when the received messages only had invalid statuses, as mentioned above. If the maximum value is sent from the pedal nodes the OK message sent from the wheel brake node will be set to false as an indication that there is something wrong with the system (even if there is no problem with the wheel brake node itself).

The system has been designed so that the wheel brake nodes send messages on the data bus about the wheel velocity and the real brake force. Sensors at the wheels should in a final version of the program provide these values but this has not yet been implemented in lack of sensors to simulate with. The values are for the moment only showed in TTPview. In the future however they might become useful in some calculations and I have for that reason already now scheduled the system and prepared the program so it can handle these messages.

4.4 Implementation of the I/O CPU

The major part of the implementation of the input for the brake pedal sensor values in the I/O CPU, Motorola MC68EN376, had already been done in an earlier Master's thesis work. [9] Only small changes had to be made to suit my system of nodes. There are four .c files written for the I/O handling in the pedal nodes (Appendix C).

The pedal nodes sample two sensor values each. The sensors are connected to the I/O CPU pins AN48 and AN3 on the port CON2 for analogue inputs. [6] The sensor

values are stored at the memory addresses 0x100000 and 0x100002 where the Host CPU, MC68360, also can read them. The two sensor values are sampled alternately, since there is only one A/D converter in the I/O CPU.

There were some problems with the I/O CPU pins in the pedal nodes during the development. The pins got short-circuited, as it seemed without any reason, so I had to reprogram the code so that I could use some new working pins. The pins that are broken in the pedal nodes are AN0, AN1 and AN2.

The sampling interval is not optimised to suit the Brake-By-Wire system in the best way. As it is now the values from the sensors are sampled as often as possible, without taking the time when the values should be used in the cluster cycle into consideration. It might even be possible that the I/O CPU samples the sensor values more often than the application for the pedal nodes has need for. I have just assumed that there always will be a new sampled sensor value for the pedal nodes to use in their calculations. The sampling interval could be changed to suit the times in the cluster cycle better, but then the time for the use of the values in the application has to be known.

The I/O CPU also handles the PWM signal on the output in the wheel brake nodes. The frequency of the PWM signal was chosen to be 10 kHz. The brake force value that will be sent as a PWM signal has to be transformed from the range 0 – 65535 (the brake force variable is a 16 bits integer) into the range 0 – 1048 (the range of the PWM signal) in a final program version. Since the brake force value in my program version very seldom reaches the upper range of a 16 bits integer I only transformed the variable from the range 0 – 35000 to 0 – 1048 to be able to see changes in the PWM signal more easily. This means that all brake force values above 35000 will result in a maximum PWM signal. The reason that the brake force does not reach high values is that the signal generator, which have been used to simulate the brake pedal, does not generate the input brake pedal position and force values with the proportions as the program is suited for, see Figure 4.4. The Host CPU stores the desired brake force value on the address 0x100004 where the I/O CPU can read it and translate it into a PWM signal. The .c files written for the PWM handling are found in Appendix C.

4.5 Compilation, linking and downloading

This chapter is mainly written as help for future work with the TTP/C system. It describes things which I had problems with during my Master's thesis work. It also describes the work procedure from the point where the application code for the nodes has been written till the point when it is executed in the nodes.

One important thing to set in TTPbuild is what compiler that is used (default diab). Another thing that can be defined is where the automatically generated files should be stored. The filenames of the automatic generated files (ttpc_ftl.c, ttpc_msg.h and ttpc_conf.c) can also be changed to something else than the predefined names. In this

case a Microtec compiler version 4.5G was used and this compiler could only work with filenames with eight characters and for that reason the filenames had to be shortened.

The demo application that came with the software tools contained three files (microtec.bat, make.bat and mysetup.bat), that were used during the compilation and linking of the application code for the host CPU, Motorola MC68360. Since it is hard to know what files you have to compile and link, and since the software tools creates automatic generated files that use files that have to be included from TTPos, I decided to use the make files from the demo application. Some modifications had to be made to the three files like, for example, that the names of the catalogues that contained my own .c files had to be set and that the attribute LIB was set to ttpos360_c1_mic.lib, which was a file that came with the new TTPos version. To be able to use the make files without too many changes I also had to save all my files in the same way with the same names as they had done in the demo application.

The compilation and linking of the .c files used for reading input signals to the node was done by using a make file (Appendix B) that had been implemented in an earlier Master's thesis work [9]. Also a make file for the compilation and linking of the .c files written for the PWM output signal has been written (Appendix B). All input and output handling is done in the I/O CPU, Motorola MC68EN376.

For each processor that should be used in each node one .s19 file and one .abs file are created during the compilation. The .s19 files are flashed into the flash memories of the nodes and the .abs files can be used to debug the application. I used Lauterbach's BDM-debugger and the program Trace 32 to flash and debug the nodes. First, the processors Motorola MC68360 and Motorola MC68EN376 have to be initialised with some initialisation files (Appendix B). Then the flash files (Appendix B) are used to flash down the .s19 files to the flash memories of the two processors in the node.

The last thing that has to be done to make the application run is to download the MEDL scheme. This is done with TTPload via the TTPmonitoring node. To be able to use the TTPmonitoring node it had to be upgraded with some Upgrade software that came with the software tools. A PC IP address also had to be defined. This IP address was chosen to 192.168.1.15. The TTPmonitoring node is connected via an Ethernet link to the PC. A second network card had to be installed to be able to establish a connection to the TTPmonitoring node and still be able to use the network for emails and Internet. This was a little bit tricky since the system has to be told that TTPload and TTPview are supposed to use the second network card. The setting for the new network card was done in the attribute interface_number in TTPload. To know which interface number that is used one just has to look in the Control Panel under Network/Adapters. My network card was the second network card so my interface number got the value two. TTPload also has to be informed about the MAC address (00-50-C2-00-E0-17) and the IP address (192.168.1.18) of the TTPmonitoring node.

In TTPload a download collection of the cluster nodes first has to be set up by loading the .ddb file of the cluster. To make sure that the contact with the TTPmonitoring node is established the Connect-to-Monitor-button is used. Then it is time to download the MEDL scheme with the download-button. TTPload will ask the user to reset the cluster by removing the terminating resistors at the ends of the data bus. The

terminating resistors are important since they prevent reflection of signals on the data bus. When the resistors are replaced again the user just have to finish the downloading by confirming that the cluster is in reset mode. Then (hopefully) all communication on the data bus works and the system is up and running.

4.6 Signal generator

The Brake-By-Wire system has been tested with a signal generator, which was developed during an earlier Master's thesis work. [10] It can generate a voltage between 0 V – 15 V. In order not to exceed the limit for allowed voltage in the nodes I have only let the signal generator generate voltages between 0 V – 5 V.

The signal generator has been used for simulation of the sensor values from the brake pedal. It is possible to let the signal generator generate three different signals. The three output values can all be set to one original signal value, but two of the output signals can also be loaded with one disturbance each. The output pins on the backside of the signal generator is shown in Figure 4.8. The original signal is generated on the pins 46 and 48, the signal with disturbance 1 on pin 23 and the signal with disturbance 2 on pin 22. The pins 24 and 45 are connected to ground.

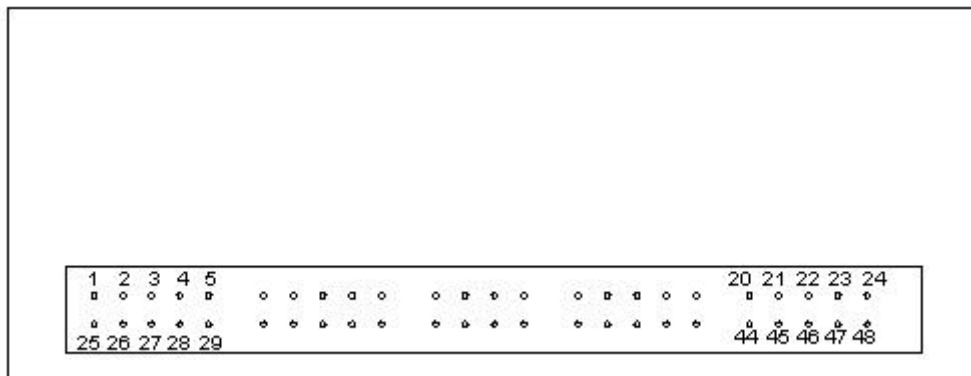


Figure 4.1 The backside of the signal generator. The output signals are numbered according to the figure.

As described in Chapter 4.3.1 the pedal nodes sample one pedal force value and one pedal position value each. These values relate to each other as seen in Figure 4.4. I have tried to simulate the pedal sensor values according to Figure 4.4 by trying to adjust the output values from the signal generator with the help of the disturbances. In this way I have got a very simple brake pedal which can simulate the most basic features in my pedal node application.

5 Conclusion and evaluation

TTP/C is a communication protocol with associated tools and hardware available from TTTech in Austria. The company TTTech was founded in 1998 and has worked with the TTP/C concept since then. I think that it is possible to make improvements on the documentation and usability of the TTP/C development environment despite these years of development.

5.1 TTP/C concept

The TTP/C protocol is designed to be a safe protocol, which promises that all messages sent on the data bus will arrive at the receiving nodes in exactly the same time and with the right content, assuming that there are no faults in the system. If a valid message cannot be sent it will not be sent at all and the other nodes will not fail to operate if one node goes down. Using the TTP/C protocol also means that if you have two data buses and send the same message on both of them it does not matter which of the two messages that is used by the receiving node. This assumes that there are no hardware problems with the data bus or with the different nodes. Since nobody can promise that this is not the case, the best way to make the system as fault tolerant as possible is to let the receiving nodes check the validity of every single message as I have tried to do in this Master's thesis work.

The major advantage of TTP/C is that it is a time-triggered protocol. This means that the communication is scheduled at design time, which simplifies the design and analysis of the distributed system. An analysis of the system would have been harder to do if an event-triggered protocol would have been used instead. It would in the latter case have been harder to know when messages are sent and which node that is in turn to send.

If the TTP/C protocol is going to be used it is important that the developer knows everything about how the system should be implemented in advance, if the scheduling of the system is going to be as effective as possible. This means that the user has to know what messages that will be sent and received by all tasks and subsystems in advance. Since the developer, during the scheduling procedure, is told to set the time budget for every task, one also has to know how many microseconds every task will take to execute. This is very hard to estimate in advance. Of course it is possible to change the settings during the work. This is however time consuming if many things have to be changed and it is likely that one forgets to change the settings in all places that has to be changed, especially in a big system with many messages, tasks and subsystems.

The system will stop to execute a task when it has executed the task as long time as it was set to do in the time budget attribute. This means that if a too short time budget is

set the task will not be executed properly. I happened to set a too short time budget to one of my tasks. As a result my program started to behave very strange. Sometimes it worked and sometimes not. Sometimes all messages were produced and sent and sometimes the membership vector said that the node was down. No messages at all were received or sent by the node in the latter case. To avoid this problem it is best to spend much time on trying to estimate a time budget that will be long enough for the task.

Practically all scheduling is done automatically in the scheduling tools. You just set all attributes like time budgets, deadlines etc and then TTPplan and TTPbuild schedules the messages. In TTP plan it is possible to visually show the schedule for when the different nodes send their messages. This gives the user a very good overview of the system. A visible schedule like this would have been very nice to have for the task scheduling in TTPbuild too. As it is now you just get a table with the times when every task is started. I think it would have been user-friendlier to show this information in a visible schedule instead. It would also have been nice if the user could change the executing order of the tasks in the same way as one is able to change the sending order of the messages in TTPplan.

One thing to say about the scheduling procedure is that very much responsibility is laid on the programmer who has to know everything in advance about the program implementation, times, number of messages etc. Predictions of the time and of what the program will do in the end of the implementation are very hard to do. The better the program is planned the better for the future work. Of course, the application code for the nodes should be developed before the communication schedule is made, so that the developer somehow could analyse and test it to get a rough estimation of what the different time budgets should be.

5.2 TTP/C software development environment

The software development environment is not satisfactory. There were many bugs in the software and during the beginning of my Master's thesis work TTPtech had to send me one new version of TTPos and one new version of TTPview. In the end of my Master's thesis work they also sent me one CD with new versions of all software tools. Since I already had a working cluster I did not want to change anything by installing new versions of the tools. The tool versions I have used come from the CD TTPtools 01/2002 and are:

- TTPplan 3.0
- TTPbuild 3.0
- TTPos 3.0
- TTPload 4.0
- TTPview 5.0

Something they also had to send was an installation program for an application called NDIS, which was necessary to have if the TTPmonitoring node should be able to be upgraded.

The documentation of TTPplan and TTPbuild starts with a basic example, which a beginner is supposed to work with. This is a very good way to learn how to use the basic settings in these tools. The problem is that when one has used these tools and want to go on with the C code, compiling, linking and downloading the documentation does not continue with the example and the user is left to conquer the problems on her/his own. This is not an easy task. The documentation of the different predefined functions is also very limited and this has lead to a lot of misunderstanding and unnecessary problems. It is possible to program the LEDs (Light-Emitting Diodes) on the TTPnodes as they have done in a demo application, which is enclosed with the software tools. The problem is that you cannot program the LEDs yourself since there is no documentation of the LED functions that they have used.

In both TTPbuild and TTPplan there is a guide, which step by step goes through all required settings. This is a very good help since every tool contains up to 8 different windows with settings that a user has to open to set all necessary variables. If one points with the mouse in the field where the developer is supposed to enter a value, a description of what this particularly attribute means pops up. This is a good help during the setting procedure.

When all required attributes in TTPplan and TTPbuild are set the user has to press a check button to see if there are any errors. If everything is all right the next step is to make the message schedule. If the scheduling goes wrong the system will tell what errors that have occurred. The error messages can be very cryptic and hard to understand. If you are told that something has gone wrong there is often very brief information about what you are supposed to do to get a valid system, if there is any information at all.

In TTPview there are some predefined variables called the status area, which can be monitored. These variables contain important information about the TTP/C protocol. The variables EIFA and EIFB were very important for me to watch, since they tell if both of the data buses are working or not. When I visualised them in TTPview they showed different values. EIFB showed the value eight, which meant that everything was all right. EIFA jumped between the values zero, one and fifteen and those values were not defined in the TTP/C protocol. When I asked TTTech why I got these values they answered me that they will remove EIFA and EIFB from the status area in the next version of TTPview, since they thought that nobody could be interested in watching these two variables. I cannot agree with them on this point. I think that it is very important to be able to show these values, since it from a safety aspect is very interesting to know if both of the data buses are working or not. If only one data bus should work the whole brake system still works but the system is no longer fault tolerant, and the user should be informed.

6 Discussion

It took me five weeks to get a simple system up and running on the TTPnodes. With better documentation of the software tools and with a continued example to work with I am sure I could have made it work much earlier. I had the possibility to talk to people who had worked a little with TTP/C programming and they told me the basic things, for example how the .c files should be written to be compatible with the code generated by the TTP/C tools. It would have taken me much longer time to figure out how to write the C code if I had been the first ever to work with these tools. I would recommend a beginner to take some courses in how to use TTP/C (courses are held by TTTech). The best way to get started would, of course, be to have a personal teacher that worked with you at your workplace in the beginning. TTTech's support office for detailed questions about TTP/C is only located in Vienna at present. As a result it has been a little problematic to ask questions since all support has to be done via emails. It is quite hard to learn everything about TTP/C just by reading the TTP/C manuals. Often different features are only mentioned briefly and the manuals have a tendency to give rise to more questions than answers. As it is now it feels like that you might have missed some valuable information about usable functions in TTP/C. Perhaps some of the things I have implemented or the settings in the software tools could have been done in a more efficient way?

A fully fault tolerant Brake-By-Wire system should possibly be built with double wheel brake nodes at all wheels. In the system I have built there is only one wheel brake node at every wheel. I have tried to implement a system that handles the situations when wheel brake nodes go down for any reason. The number of wheel brake nodes depends on costs and how likely it is that a wheel brake node goes down. As long as only one wheel brake node is not operational the car could still be braked in a satisfying way. It was also easier for me to work with as few nodes as possible since I just had 20 weeks to work on my Master's thesis. Another thing, which also is important, is to always have a working power supply. To be on the safe side a double power supply would be needed in a real Brake-By-Wire system.

The wheel brake nodes have the possibility to shut down the brake actuator if they think that something is wrong. Perhaps the best thing to do is to let the actuator be permanently shut down if it once has been shut down. The wheel brake node should perhaps not have the responsibility or permission to check if everything works again and then turn on the actuator. The wheel brake node can be wrong and the fault might still be there and wrong decisions can be devastating in such serious safe related systems like brake systems. I think the best thing would be to let authorised repair persons at a garage be the only ones who can restart a brake. In this case you know that competent people have looked at the problem and also fixed it if it is necessary.

One problem, as I see it, is that people can come to rely too much on this new braking system. Maybe they know that it can handle a single wheel brake node loss and they know that the system works even if only one data bus is working. I see a great risk that many drivers will not take the warnings displayed as seriously as they should do. Perhaps they continue to drive as nothing has happened and think that there must be something wrong with the error indication since the brakes still works. I do not know

how this problem best should be handled. One possibility is of course to let the car shut down itself when too many errors have arisen or some errors have been left without countermeasures for too long time. The car would then not be able to start until the brakes or the data buses have been repaired.

I think that the rules for how to write the code should be more restricted. The C programming language is too flexible and easy to manipulate. As it is now all responsibility that you for example get all types right or all allocations of pointer variables right is up to the programmer. Since the C language (and thereby the compiler) allows so much, it is very easy to make devastating programming mistakes if you are not completely sure of how to program in C. The safety problems with C is however known to the automotive industry and MISRA (The Motor Industry Software Reliability Association) have therefore written an instruction book (“Guidelines for the use of the C language in vehicle based software”) about how to write good C code. [11]

7 Future work

The development of a complete Brake-By-Wire system is time consuming and demands great work efforts. A single person cannot do a work like this in 20 weeks. That is why I have tried to concentrate on the fault tolerance aspects and tried to implement a foundation, which can underlie future work and development. There are still several things left to do of which I would like to give some examples.

As mentioned in section 4.3.2 there is a possibility to give unique brake force commands to every wheel brake node in order to compensate for lost wheel brake nodes. The ideal wheel brake forces needed to brake a car depends on different things like the speed of the car and if the car is turning or going straight ahead. If only three wheel brakes are working I think that it would be best to let the diagonal working brakes brake hardest. The third working brake, diagonal opposite to the broken brake, will introduce a torque if it is given some brake force and should for that reason be given as little brake force as possible. The possibility to give individual commands to the different wheel brake nodes also means that it is relatively easy to introduce helping brake functions like for example:

- ABS – Antilock Braking System
- ESP – Electronic Stability Program is a safety system that ensures the stability of a vehicle. It recognises when the vehicle understeers or oversteers. By braking individual wheels, it corrects understeer and oversteer and thus keeps the vehicle on the road, especially on slick roads and in curves.
- Brake Assistance – Investigations show that drivers are quick to react but do not exert enough pressure in dangerous situations. Brake Assistance is a braking system that recognises a panic brake situation and applies appropriate brake force, guaranteeing more efficient braking capacity.
- Adaptive speed control

The brake functions above will very likely have to use the information about wheel velocity and real brake force in the calculation of brake force to be applied. I have prepared the system for sending these messages as mentioned in section 4.3.4. In a “real” brake system the pedal nodes, which do all brake force calculations, would probably also need information about the movements of the car concerning, for example, yaw rate, steering wheel angle, side acceleration etc. The pedal nodes should also communicate with the different parts of the car (engine, transmission, spring system, instrumental panel etc.).

The wheel brake nodes should also be implemented so that they can turn off the brake actuator if there is something wrong with, for example, the brake force sensor, see section 4.3.4. The values sampled at the wheel brake nodes could for safety reasons also be sampled from more than one sensor, just like the brake pedal values have been sampled.

If there is something wrong with the pedal nodes, they have the possibility to send this information to the wheel brake nodes by sending a brake force message with the maximum value ($923 * 64$), see section 4.3.4. I have not implemented for what

reasons a pedal node should tell that there is something wrong with it. As it is now the pedal nodes always send the information that they are OK.

The complex procedure of deciding a valid pedal sensor value is based on what value the last used sensor value had. This is perhaps not the best way of comparing the sensor values. For instance there might be a short period of time when all sensors reads wrong values and then starts to work again. In this case there is a chance that the correct sensor value now has increased or decreased so much that it still lies too far from the last used sensor value to be regarded as a valid sensor value. It is hard to know how to handle this case. Perhaps it would be possible to write a function that somehow predicts how the brake pedal moves by using the information about how it has been moved earlier. A prediction function for the brake force could perhaps also be implemented to prepare and facilitate the calculation of the brake force.

When I tested the system with my signal generator a problem occurred. It seems like a node starts from the init function if its data bus has been broken and then starts to work again. This means that the node has forgotten everything about the status of the system before the node went down. As it turned out in my system the pedal nodes began to work with different number of sensor values if one of the pedal nodes had been down and had forgotten that one sensor was broken. If this broken sensor had started to work again the newly awakened pedal node would use this sensor value, while the pedal node that not had failed kept counting as if the sensor was broken. One way to solve this problem would be to introduce some more messages in the system. For example the pedal nodes can exchange their status messages with each other so that they are sure that they agree in all cases. If a pedal node forgets the present status the other working pedal node can remind it.

The brake force is calculated with the help of Equation 4.1. This equation contains the variable x , which is a factor for how much weight that is put on the variables FF and PF in the calculations. I have just chosen to set x to either 0.25 or 0.75 as described in 4.3.1. This is just a rough allotment and could probably be done in a more sophisticated way. For example the trend in the pedal force and pedal position diagram could be more decisive when x is set. One rule one could use is that the greater the span is on one of the axis (compared to the span on the other axis), the more weight should be put on the variables on that axis. As example take Figure 7.1. In the span where the pedal force takes the values 400 – 923 the pedal position only varies between 900 – 923. This means that the pedal force has a better resolution in this interval than the pedal position. More weight should be put on the values on the pedal force axis since they are more varied and thereby contains more information in this interval. I think you need to simulate the brake force calculated from Equation 4.1 to see what the optimal value of x should be in the different cases and how it should vary in accordance to the diagram.

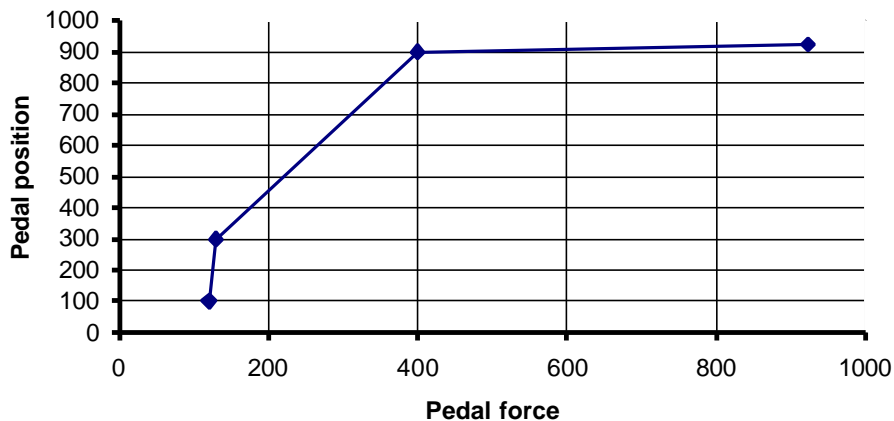


Figure 7.1 Example of pedal position and pedal force diagram.

During my Master's thesis I was provided with four pairs of co-ordinates with the ratio between the brake pedal position and the brake pedal force, see Figure 4.4. These co-ordinate values have been written directly into the application code for the pedal nodes, where they have been used in the calculation of a final sensor value. Only four co-ordinates is a very rough approximation of the ratio between the pedal position and the pedal force, and to improve the approximation more pairs of co-ordinates would certainly be needed in a real Brake-By-Wire system. This could be implemented as a big matrix where the system developer could enter the valid co-ordinate values. When all values have been entered the programmer could set some kind of flag in the end of the matrix as a sign that the matrix ends. This flag will be useful when it comes to while-loops etc. The co-ordinate values will affect the sensor value transformation calculations done with linear interpolation (Chapter 4.3.1), the calculation of x in Equation 4.1 and the limit for the brake pedal position and force where the car will start to brake.

The wheel brake nodes have the responsibility to check whether the real brake force value agrees with the desired brake force value. The brakes will probably not have enough time to meet the request if the brake force value is checked right after the request is made. It is hard for me to guess how long time it takes for the brakes to meet the demands on brake force. To get a reasonable time I guess you have to do some simulations. The time probably depends on the situation, how efficient the brakes are, how hard the brake pedal is pushed and how slippery the surface is. Perhaps the best thing is to save the desired value and wait some milliseconds before the real brake force is checked. Something that can be investigated immediately is what the trend looks like. If the brake force message wants the system to brake harder then the brakes should start to brake harder as soon as they have got the request. I think it will be too hard to demand that the real brake force should have exactly the same value as the desired brake force. It is better to say that the real brake force should be within some certain limits.

The program written for this Master's thesis has only partly been tested, since I had not got the time to test it with a real brake pedal and real brake actuators. To get the program fully tested a simulation with a car has to be done. A test like this will get cheaper in the long run if it first is done on a simulated car. A simulation can be done using Hardware-in-the-loop.

Since I do not know what hardware that this Brake-By-Wire application will be connected to, there is nothing done with the implementation concerning input through the I/O CPU in the wheel brake nodes. The output from the wheel brake nodes contains the desired brake force value and is implemented as a PWM signal. This brake force value has to be recalculated in a final program version if it will be able to control a brake actuator, as mentioned in Chapter 4.3.4.

Appendix A

- All subsystems, tasks and messages that are used in the different nodes.

Pedal Node 1

Subsystem	P1_SENS_SYS	P1_BRAKE_SYS	P_SYS
Task	P1ReadSensorValue	P1CalculateSensorAndBrakeValue	StatusMessages
Sends Messages	SENSOR11	FL_BRAKE1	P1_BUS_OK
	SENSOR12	FR_BRAKE1	P2_BUS_OK
		BL_BRAKE1	P_SENSOR11_OK
		BR_BRAKE1	P_SENSOR12_OK
			P_SENSOR22_OK
			P_FL_BUS_OK
			P_FL_RUN_OK
			P_FL_OK
			P_FR_BUS_OK
			P_FR_RUN_OK
			P_FR_OK
			P_BL_BUS_OK
			P_BL_RUN_OK
			P_BL_OK
			P_BR_BUS_OK
			P_BR_RUN_OK
			P_BR_OK
			P_FLNODE_OK
			P_FRNODE_OK
			P_BLNODE_OK
			P_BRNODE_OK
			P_P1NODE_OK
			P_P2NODE_OK
Receives Messages		SENSOR11	FL_BUS_OK
		SENSOR12	FL_RUN_OK
		SENSOR21	FL_OK
		SENSOR22	FR_BUS_OK
			FR_RUN_OK
			FR_OK
			BL_BUS_OK
			BL_RUN_OK
			BL_OK
			BR_BUS_OK
			BR_RUN_OK
			BR_OK

Pedal Node 2

Subsystem	P2_SENS_SYS	P2_BRAKE_SYS	P_SYS
Task	P2ReadSensorValue	P2CalculateSensorAndBrakeValue	StatusMessages
Sends Messages	SENSOR21	FL_BRAKE2	P1_BUS_OK
	SENSOR22	FR_BRAKE2	P2_BUS_OK
		BL_BRAKE2	P_SENSOR11_OK
		BR_BRAKE2	P_SENSOR12_OK
			P_SENSOR22_OK
			P_FL_BUS_OK
			P_FL_RUN_OK
			P_FL_OK
			P_FR_BUS_OK
			P_FR_RUN_OK
			P_FR_OK
			P_BL_BUS_OK
			P_BL_RUN_OK
			P_BL_OK
			P_BR_BUS_OK
			P_BR_RUN_OK
			P_BR_OK
			P_FLNODE_OK
			P_FRNODE_OK
			P_BLNODE_OK
			P_BRNODE_OK
			P_P1NODE_OK
			P_P2NODE_OK
Receives Messages		SENSOR11	FL_BUS_OK
		SENSOR12	FL_RUN_OK
		SENSOR21	FL_OK
		SENSOR22	FR_BUS_OK
			FR_RUN_OK
			FR_OK
			BL_BUS_OK
			BL_RUN_OK
			BL_OK
			BR_BUS_OK
			BR_RUN_OK
			BR_OK

Front Left Wheel Brake Node

Subsystem	FLsys
Task	FLSetBrakeValue
Sends Messages	FL_BUS_OK
	FL_RUN_OK
	FL_OK
	FL_WHEELVEL
	FL_BRAKEFORCE
Receives Messages	FL_BRAKE1
	FL_BRAKE2

Front Right Wheel Brake Node

Subsystem	FRsys
Task	FRSetBrakeValue
Sends Messages	FR_BUS_OK
	FR_RUN_OK
	FR_OK
	FR_WHEELVEL
	FR_BRAKEFORCE
Receives Messages	FR_BRAKE1
	FR_BRAKE2

Back Left Wheel Brake Node

Subsystem	BLsys
Task	BLSetBrakeValue
Sends Messages	BL_BUS_OK
	BL_RUN_OK
	BL_OK
	BL_WHEELVEL
	BL_BRAKEFORCE
Receives Messages	BL_BRAKE1
	BL_BRAKE2

Back Right Wheel Brake Node

Subsystem	BRsys
Task	BRSetBrakeValue
Sends Messages	BR_BUS_OK
	BR_RUN_OK
	BR_OK
	BR_WHEELVEL
	BR_BRAKEFORCE
Receives Messages	BR_BRAKE1
	BR_BRAKE2

Appendix B

- The files used to flash down the .s19 files into the flash memories of the nodes. The program Trace 32 was used together with Lauterbach's BDM-debugger.
- The make files for the compilation and linking of the .c files used in the I/O CPU, Motorola MC68EN376.

init360.cmm – file contents for initialisation file for Motorola MC68360

```
SYStem.down  
sys.cpu M68360  
sys.option base 80000  
sys.option brkvector 400000  
sys.bdmclock 0F4240  
system.mode up
```

```
d.s CPU:3ff00 %1 80001  
d.s 00081054 %1 3ff80000  
d.s 00081050 %1 00000001  
d.s 00081064 %1 0FFFFE006  
d.s 00081060 %1 00100001  
d.s 00081084 %1 1ffe0000  
d.s 00081080 %1 00200001  
d.s 00081094 %1 -0FFA  
d.s 00081090 %1 00300001
```

```
MAP.RESet  
MAP.ROM  
MAP.bus16
```

```
REGISTER.SET PP 20040C
```

```
ENDDO
```

flash.cmm – file contents for flash file for MotorolaMC68360

```
flash.res  
flash.create 00--7fff AM29F100 Long  
flash.erase 00--7fff  
flash.program all
```

```
d.load.s2record c:\tttech\bbw-exjobb\build\*.s19 /1
```

```
flash.program
```

init376.cmm – file contents for initialisation file for Motorola MC68EN376

```
screen.on  
winpage.reset  
winclear
```

```
SYStem.down  
sys.cpu M68376  
sys.option base 0FFF000  
sys.option brkvector 0  
sys.bdmclock 1000000.  
system.mode up
```

```
map.reset  
;Memory attributes for flash memory  
map.bus16 0--03FFFF  
;Memory attributes for DPRAM  
map.bus16 0100000--0101FFF  
;Memory attributes for SRAM  
map.bus16 0200000--020FFFF
```

```
D.S SD:0FFFA00 %WORD 0404F ;SIM Configuration Register (40CF)  
D.S SD:0FFFA21 %BYTE 0C ;System Protection Control register (0)
```

```

D.S SD:0FFFA44 % WORD 000FF ;Chip Select Pin Assignment register 0 (3FFF)
D.S SD:0FFFA46 % WORD 00000 ;Chip Select Pin Assignment register 1 (03A5)
D.S SD:0FFFA48 % WORD 00005 ;Enable 256kB (0007)
D.S SD:0FFFA4A % WORD 07831 ;No waitstates for BOOT ROM (78F0)
D.S SD:0FFFA4C % WORD 01001 ;CS0 Chip Select Base Address Register
D.S SD:0FFFA4E % WORD 07BF1 ;CS0 Chip Select Option Registers
D.S SD:0FFFA54 % WORD 02003 ;CS2 Chip Select Base Address Register
D.S SD:0FFFA56 % WORD 07BB1 ;CS2 Chip Select Option Registers
D.S SD:0FFFA04 % WORD 0D300 ;(7f00)

D.S SD:0FFFB44 % WORD 00040
D.S SD:0FFFB46 % WORD 00000
D.S SD:0FFFB40 % WORD 00800

```

ENDDO

flash376.cmm – file contents for flash file for Motorola MC68EN376

```

flash.res
flash.create 00--3fff AM29F100 WORD
flash.erase 00--3fff
flash.program all

d.load.AUTO c:\*.s19 /l

flash.program

;d.load.ieee c:\*.abs

ENDDO

```

make file contents for compilation and linking of .c files for input sensor values to Motorola MC68EN376

```

# Makefile for TTP

# Filestructure
BaseDIR=c:\qadc

OutputFileName=pwm

# Compiler
CC=C:\MRI\MCC68K\Mcc68k.exe

# Assembler
ASM=C:\MRI\ASM68K\Asm68k.exe

# Linker
LNK=C:\MRI\ASM68K\Lnk68k.exe

# Compiler options
# '-H -Fsm' to generate an asm list file
CFLAGS=-c -g -pCPU32 -D OS_READY_STATE -v -H -zc

# Linker options
LFLAGSF=-c $(BaseDIR)\Linkf.cmd -M -Znptmx

# Assembler options
ASMFLAGS=-pCPU32

# Output code dependencies
$(OutputFileName): \
# Application code #

```

```

main.obj      \
qadc.obj      \
startup.obj   \
start.obj

# Link command
$(LNK) $(LFLAGSF) -Fiee -o $(OutputFileName)_Flash.abs
$(LNK) $(LFLAGSF) -Fs2 -o $(OutputFileName)_Flash.s19

# Application

main.obj: $(BaseDIR)\main.c Makefile $(BaseDIR)\qadc.h
$(CC) $(CFLAGS) $(BaseDIR)\main.c

qadc.obj: $(BaseDIR)\qadc.c Makefile $(BaseDIR)\qadc.h
$(CC) $(CFLAGS) $(BaseDIR)\qadc.c

startup.obj: $(BaseDIR)\startup.c Makefile
$(CC) $(CFLAGS) $(BaseDIR)\startup.c

start.obj: $(BaseDIR)\start.src Makefile
$(ASM) $(ASMFLAGS) $(BaseDIR)\start.src

# Delete all *.obj files
clean:
del *.obj
del *.s19
del *.abs

```

make file contents for compilation and linking of .c files for output PWM signal from Motorola MC68EN376

```

# Makefile for TTP
# Filestructure
BaseDIR=c:\pwm

OutputFileName=pwm

# Compiler
CC=C:\MRI\MCC68K\Mcc68k.exe

# Assembler
ASM=C:\MRI\ASM68K\Asm68k.exe

# Linker
LNK=C:\MRI\ASM68K\Lnk68k.exe

# Compiler options
# '-H -Fsm' to generate an asm list file
CFLAGS=-c -g -pCPU32 -D OS_READY_STATE -v -H -zc

# Linker options
LFLAGSF= -c $(BaseDIR)\Linkf.cmd -M -Znptmx

# Assembler options
ASMFLAGS=-pCPU32

# Output code dependencies
$(OutputFileName):
# Application code
main.obj      \
pwm.obj       \
startup.obj   \

```



```

start.obj

# Link command
$(LNK) $(LFLAGS) -Fiee -o $(OutputFileName)_Flash.abs
$(LNK) $(LFLAGS) -Fs2 -o $(OutputFileName)_Flash.s19

# Application

main.obj: $(BaseDIR)\main.c Makefile $(BaseDIR)\pwm.h
$(CC) $(CFLAGS) $(BaseDIR)\main.c

pwm.obj: $(BaseDIR)\pwm.c Makefile $(BaseDIR)\PWM.h
$(CC) $(CFLAGS) $(BaseDIR)\PWM.c

startup.obj: $(BaseDIR)\startup.c Makefile
$(CC) $(CFLAGS) $(BaseDIR)\startup.c

start.obj: $(BaseDIR)\start.src Makefile
$(ASM) $(ASMFLAGS) $(BaseDIR)\start.src

# Delete all *.obj files
clean:
del *.obj
del *.s19
del *.abs

```

Appendix C

- The C code for one of the pedal nodes and one of the wheel brake nodes.
- The C code in the files that are written for the I/O CPU.
 - * Input sensor values for the pedal nodes
 - * Output PWM signal from the wheel brake nodes

C code for one of the pedal nodes (P1_node)

```
#include "TTPos.h"
#include "led.h"

#include "TTPCftl.h"

#include "ttpc_msg.h"

#define NODEMASK 0x0001
#define CLUSTERCYCLESDOWN 500 // It takes approximately 2 seconds to execute 500 clustercycles
#define MAX_OLD_VALUES 5 //How many times in row an old sensor value can be used before emergency
braking
#define DIFF10BIT 105 // Max diff. if two sensor values will be regarded as similar, 10 bits
#define DIFF16BIT 7000 // Max diff. if two sensor values will be regarded as similar, 16 bits
#define SENSORDIFF 15000 // Max allowed difference between a new final sensor value and an old

#define SENSOR_VALUE_1      * (unsigned short int *) (0x100000)
#define SENSOR_VALUE_2      * (unsigned short int *) (0x100002)

// Brakevalues to be set by CalculateBrakeValue
brakeval BLbrake, BRbrake, FLbrake, FRbrake;

// These variables contain this pedal node's sampled sensor values
sensorval sens12, sens11;

// The last used brakevalues. Can be used in cases where its os hard to decide a
// valid sensorvalue. The sensorvalues can differ so much that you cannot unify them.
unsigned long int OldFinalSensorValue;

// NodeVector contains node-alive info in the order FLnode, P1node, P2node, BRnode
// FRnode and BLnode
bool NodeVector[6];

// BrakeRunVector contains Run-info in the order BLnode, BRnode, FLnode and FRnode
bool BrakeRunVector[4];

// Tells if the sensors are valid or not. Once set to false means forever false.
bool OK11, OK12, OK21, OK22;

// Variable that can be set to false if something is wrong with the pedal node.
// This setting is not yet implemented. The variable will always be true in this version.
bool PedalOK;

// Variable which counts how many times in row an old sensor value has been used.
int OldCount;

// Variable which contain total info about if the brakenodes can execute any brake
// commands or not. The order of the nodes are BLnode, BRnode, FLnode and FRnode
int BrakeStatus;

// These variables keep track on how many clustercycles a sensor is out of range
int Sens11Cycles, Sens12Cycles, Sens21Cycles, Sens22Cycles;

// These variables keep track on how many clustercycles a sensor's value
// is far from the other sensors' values
int Sens11FarAway, Sens1221FarAway, Sens22FarAway;

void init (void)
{
    int k, l;

    /* Set all LEDs */
    LED_ON (LED_GREEN_1);
    LED_ON (LED_GREEN_2);
}
```

```

LED_OFF (LED_RED_1);
LED_OFF (LED_RED_2);

// Set all brake values to zero
BLbrake = 0;
BRbrake = 0;
FLbrake = 0;
FRbrake = 0;

// Say that all sensors work
OK11 = 1;
OK12 = 1;
OK21 = 1;
OK22 = 1;

sens12 = 0;
sens11 = 0;

// Old final sensor value must be transformed into a 16bits by multiplying with 64
// Compensate for the fact that the brakes only starts to work from at minimum value
// 130 from the brake pedal sensor

OldFinalSensorValue = (SENSOR_VALUE_1 - 130) * 64;

Sens11Cycles = 0;
Sens12Cycles = 0;
Sens21Cycles = 0;
Sens22Cycles = 0;

Sens11FarAway = 0;
Sens121FarAway = 0;
Sens22FarAway = 0;

OldCount = 0;

// Say that the pedal node works
PedalOK = 1;

for (k = 0; k <= 5; k++)
{
    NodeVector[k] = 0;
}

for (l = 0; l <= 3; l++)
{
    BrakeRunVector[l] = 0;
}

BrakeStatus = 0x0000;
} /* init */

unsigned long int TranslatePositionToForce(sensorval p)
{
    unsigned long int x, y;

    // Translates the position value to a force value which can be used to compare with
    // the FF1 value. This function also translates the 10 bits input value to a 16 bits value

    y = p * 64 * 10000;    /* 10000 to be able to divide by 22222 (2.2222) resp 440 (0.0440) */

    if (p < 300)
    {
        // The brake force should be zero within this interval
        x = 0;
    }
    else if ((p >= 300) && (p < 900))
    {

```

```

// In this interval the brakeforce is within the steep curve in the diagram
// x is calculated from the equation of a straight line
x = ((y - 192000000) / 22222) + 8320;

// Compensate for the fact that the brakeforce should be zero for x < 130 * 64
x = x - (130 * 64);
}
else
{
// The brakeforce is in the end of the diagram
// x is calculated from the equation of a straight line
x = ((y - 576000000) / 440) + 25600;

// Compensate for the fact that the brakeforce should be zero for x < 130 * 64
x = x - (130 * 64);
}
return x;
} /* TranslatePositionToForce */

unsigned long int ReturnOneFFSensorValue(sensorval ab, sensorval ba, sensorval aa, sensorval bb)
{
// Compare sensor12 and sensor21 and decide one value to represent these two values
int abdiff, badiff, a_bdiff, a_bmean, a_abdiff, a_badiff, b_abdiff, b_badiff, a, b;
unsigned long int val, y;

if (abs(ab - ba) <= DIFF10BIT)
{
// The two sensor values ab and ba are similar. Take the mean value of them.
val = (ab + ba) / 2;
}
else
{
// The sensor values ab and ba are different. A careful comparison is needed.

// Translate the positionvalues into force values.
if (aa < 300)
{
a = 0;
}
else if ((aa >= 300) && (aa < 900))
{
y = aa * 10000; /* 10000 to be able to divide by 22222 (2.2222) resp 440 (0.0440) */
a = ((y - 3000000) / 22222) + 130;
}
else if ((aa >= 900) && (aa <= 923))
{
y = aa * 10000; /* 10000 to be able to divide by 22222 (2.2222) resp 440 (0.0440) */
a = ((y - 9000000) / 440) + 400;
}
else
{
a = ((9230000 - 9000000) / 440) + 400;
}

if (bb < 300)
{
b = 0;
}
else if ((bb >= 300) && (bb < 900))
{
y = bb * 10000; /* 10000 to be able to divide by 22222 (2.2222) resp 440 (0.0440) */
b = ((y - 3000000) / 22222) + 130;
}
else if ((bb >= 900) && (bb <= 923))
{
y = bb * 10000; /* 10000 to be able to divide by 22222 (2.2222) resp 440 (0.0440) */
b = ((y - 9000000) / 440) + 400;
}
}
}

```

```

}
else
{
    b = ((9230000 - 9000000) / 440) + 400;
}

// Check if possible with sensor11 and sensor22
if (OK11 && OK22 && (aa != 0) && (bb != 0))
{
    // Both sensor11 and sensor22 can be used in the comparison
    a_bdiff = a - b;

    if (abs(a_bdiff) <= DIFF10BIT)
    {
        // The two sensor values a and b are similar
        a_bmean = (a + b) / 2;
        abdiff = abs(a_bmean - ab);
        badiff = abs(a_bmean - ba);

        if (abdiff < badiff)
        {
            // The difference between ab, a and b is smallest
            if (abdiff <= DIFF10BIT)
            {
                // The difference between ab, a and b is small enough to be regarded
                // as the same value
                val = ab;
            }
            else
            {
                // The difference between ab, a and b is NOT small enough to be regarded
                // as the same value. Do not use ab at all. Set val to the mean
                // value of a and b.
                val = a_bmean;
            }
        }
        else
        {
            // The difference between ba, a and b is smallest
            if (badiff <= DIFF10BIT)
            {
                // The difference between ba, a and b is small enough to be regarded
                // as the same value
                val = ba;
            }
            else
            {
                // The difference between ba, a and b is NOT small enough to be regarded
                // as the same value. Do not use ba at all. Set val to the mean
                // value of a and b.
                val = a_bmean;
            }
        }
    }
}
else
{
    // The two sensor values a and b are different
    a_abdiff = abs(a - ab);
    a_badiff = abs(a - ba);
    b_abdiff = abs(b - ab);
    b_badiff = abs(b - ba);

    if ((a_abdiff <= DIFF10BIT) && (a_badiff <= DIFF10BIT))
    {
        // Both ab and ba are close to a. Choose the value which is closest.
        if (a_abdiff < a_badiff)

```

```

    {
        val = ab;
    }
    else
    {
        val = ba;
    }
}
else if ((a_abdiff <= DIFF10BIT) && (b_badiff <= DIFF10BIT))
{
    // ab is close to a and ba is close to b. Choose the biggest value of ab and ba.
    if (ab > ba)
    {
        val = ab;
    }
    else
    {
        val = ba;
    }
}
else if ((a_badiff <= DIFF10BIT) && (b_abdiff <= DIFF10BIT))
{
    // ba is close to a and ab is close to b. Choose the biggest value of ab and ba.
    if (ab > ba)
    {
        val = ab;
    }
    else
    {
        val = ba;
    }
}
else if ((b_abdiff <= DIFF10BIT) && (b_badiff <= DIFF10BIT))
{
    // Both ab and ba are close to b. Choose the value which is closest.
    if (b_abdiff < b_badiff)
    {
        val = ab;
    }
    else
    {
        val = ba;
    }
}
else if (a_abdiff <= DIFF10BIT)
{
    // Just ab is close to another value. In this case a.
    val = ab;
}
else if (a_badiff <= DIFF10BIT)
{
    // Just ba is close to another value. In this case a.
    val = ba;
}
else if (b_abdiff <= DIFF10BIT)
{
    // Just ab is close to another value. In this case b.
    val = ab;
}
else if (b_badiff <= DIFF10BIT)
{
    // Just ba is close to another value. In this case b.
    val = ba;
}
else
{
    // None of the values ab, ba, a and b are close to each other.

```

```

        // Choose the highest value of ab and ba.
        if (ab > ba)
        {
            val = ab;
        }
        else
        {
            val = ba;
        }
    }
}
else if (OK11 && !OK22 && (aa != 0))
{
    // Only sensor11 can be used in the comparison
    if (abs(a - ab) < abs(a - ba))
    {
        // The difference between a and ab is smallest. Choose ab.
        val = ab;
    }
    else
    {
        // The difference between a and ba is smallest. Choose ba.
        val = ba;
    }
}
else if (OK22 && !OK11 && (bb != 0))
{
    // Only sensor22 can be used in the comparison
    if (abs(b - ab) < abs(b - ba))
    {
        // The difference between b and ab is smallest. Choose ab.
        val = ab;
    }
    else
    {
        // The difference between b and ba is smallest. Choose ba.
        val = ba;
    }
}
else
{
    // None of sensor11 and sensor22 can be used in the comparison
    // Take the greatest value of sensor12 and sensor21
    if (ab > ba)
    {
        val = ab;
    }
    else
    {
        val = ba;
    }
}
}
return val;
} /* ReturnOneFFSensorValue */

unsigned long int CompareAllSensorValues(unsigned long int pf1, unsigned long int pf2, unsigned long int ff1)
{
    unsigned long int pfDiff, ff1Diff, pf_ff, pf_ffmean, pf, ff, pf1_sens, pf2_sens, ff1_sens;
    bool NewValueOK;

    // Use all three sensor values to decide one final value
    if (labs(pf1 - pf2) <= DIFF16BIT)
    {
        // pf1 and pf2 are close to each other. Calculate their mean value.
        pf = (pf1 + pf2) / 2;
    }
}

```



```

Sens11FarAway = 0;
Sens22FarAway = 0;

if (labs(pf - ff1) <= DIFF16BIT)
{
    // ff1 is close to pf. All three values are close to each other.
    // Calculate a final value. This final value depends on where on
    // the position/force curve that pf and ff1 are.

    pf_ffmean = (pf + ff1) / 2;

    Sens1221FarAway = 0;

    if ((pf_ffmean >= 0) && (pf_ffmean < (25600 - (130 * 64))))
    {
        // The first, steep bit of the curve. The pf value is the most important value.
        // I have set that pf is three times as important as ff1.
        pf_ff = ((3 * pf) + ff1) / 4;

        if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
        {
            // pf_ff is similare to the old sensor value. Use pf_ff as new value.
            NewValueOK = 1;
        }
        else
        {
            //pf_ff is too far from the old sensor value. Use the old value.
            NewValueOK = 0;
        }
    }
    else if ((pf_ffmean >= (25600 - (130 * 64))) && (pf_ffmean < (59072 - (130 * 64))))
    {
        // The last, flat bit of the curve. The ff1 value is the most important value.
        // I have set that ff1 is three times as important as pf.
        pf_ff = (pf + (3 * ff1)) / 4;

        if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
        {
            // pf_ff is similare to the old sensor value. Use pf_ff as new value.
            NewValueOK = 1;
        }
        else
        {
            //pf_ff is too far from the old sensor value. Use the old value.
            NewValueOK = 0;
        }
    }
    else
    {
        // The meanvalue is out of the diagram range and cannot be used.
        NewValueOK = 0;
    }
}
else
{
    // ff1 is NOT close to pf.
    pfDiff = labs(OldFinalSensorValue - pf);
    ff1Diff = labs(OldFinalSensorValue - ff1);

    // Choose the value closest to the old sensor value
    if ( ff1Diff < pfDiff)
    {
        // ff1 is closest to the old sensor value

        Sens1221FarAway = 0;
    }
}

```

```

    if (ff1Diff <= SENSORDIFF)
    {
        // ff1 is similare to the old sensor value. Use ff1 as new value.
        pf_ff = ff1;
        NewValueOK = 1;
    }
    else
    {
        // ff1 is too far from the old sensor value. Use the old sensor value.
        NewValueOK = 0;
    }
}
else
{
    // pf is closest to the old sensor value

    // Sensor12/21 is far away from a valid value. Note that and if it has been
    //wrong for more than 2 seconds do not use it any more
    Sens1221FarAway++;

    if (Sens1221FarAway > CLUSTERCYCLESDOWN)
    {
        // Sensor 12/21 has been wrong for more than 2 seconds.
        // Do not ever use a value from that sensor again.
        OK12 = 0;
        OK21 = 0;
    }

    if (pfDiff <= SENSORDIFF)
    {
        // pf is similare to the old sensor value. Use pf as new value.
        pf_ff = pf;
        NewValueOK = 1;
    }
    else
    {
        // pf is too far from the old sensor value. Use the old value.
        NewValueOK = 0;
    }
}
}
}
else if (labs(pf1 - ff1) <= DIFF16BIT)
{
    // pf1 and pf2 are NOT close to each other, but pf1 and ff1 are

    Sens1221FarAway = 0;

    // Make sure that pf2 and ff1 not are closer than pf1 and ff1.
    if (labs(pf2 - ff1) < labs(pf1 - ff1))
    {
        pf = pf2;

        Sens11FarAway++;
        Sens22FarAway = 0;
        if (Sens11FarAway > CLUSTERCYCLESDOWN)
        {
            OK11 = 0;
        }
    }
    else
    {
        if (labs(pf2 - ff1) > DIFF16BIT)
        {
            // Pf2 is far from ff1 and pf1.
            // Note this and do not use this sensor value

```

```

        // if this fault has remained for more than 2 seconds.
        Sens11FarAway = 0;
        Sens22FarAway++;
        if (Sens22FarAway > CLUSTERCYCLESDOWN)
        {
            OK22 = 0;
        }
    }
    else
    {
        Sens22FarAway = 0;
    }
    pf = pf1;
}

// ff1 is close to pf.
// Calculate a final value. This final value depends on where on
// the position/force curve that pf and ff1 are.

pf_ffmean = (pf + ff1) / 2;

if ((pf_ffmean >= 0) && (pf_ffmean < (25600 - (130 * 64))))
{
    // The first, steep bit of the curve. The pf value is the most important value.
    // I have set that pf is three times as important as ff1.
    pf_ff = ((3 * pf) + ff1) / 4;

    if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
    {
        // pf_ff is similare to the old sensor value. Use pf_ff as new value.
        NewValueOK = 1;
    }
    else
    {
        //pf_ff is too far from the old sensor value. Use the old value.
        NewValueOK = 0;
    }
}
else if ((pf_ffmean >= (25600 - (130 * 64))) && (pf_ffmean < (59072 - (130 * 64))))
{
    // The last, flat bit of the curve. The ff1 value is the most important value.
    // I have set that ff1 is three times as important as pf.
    pf_ff = (pf + (3 * ff1)) / 4;

    if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
    {
        // pf_ff is similare to the old sensor value. Use pf_ff as new value.
        NewValueOK = 1;
    }
    else
    {
        //pf_ff is too far from the old sensor value. Use the old value.
        NewValueOK = 0;
    }
}
else
{
    // The meanvalue is out of the diagram range and cannot be used.
    NewValueOK = 0;
}

}
else if (labs(pf2 - ff1) <= DIFF16BIT)
{
    // pf1 and pf2 are NOT close to each other and pf1 and ff1 are NOT
    // close to each other, but pf2 and ff1 are
    pf = pf2;
}

```

```

Sens1221FarAway = 0;
Sens22FarAway = 0;

// pf1 is far from the other sensor values.
Sens11FarAway++;

if (Sens11FarAway > CLUSTERCYCLESDOWN)
{
    OK11 = 0;
}
// ff1 is close to pf.
// Calculate a final value. This final value depends on where on
// the position/force curve that pf and ff1 are.

pf_ffmean = (pf + ff1) / 2;

if ((pf_ffmean >= 0) && (pf_ffmean < (25600 - (130 * 64))))
{
    // The first, steep bit of the curve. The pf value is the most important value.
    // I have set that pf is three times as important as ff1.
    pf_ff = ((3 * pf) + ff1) / 4;

    if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
    {
        // pf_ff is similare to the old sensor value. Use pf_ff as new value.
        NewValueOK = 1;
    }
    else
    {
        //pf_ff is too far from the old sensor value. Use the old value.
        NewValueOK = 0;
    }
}
else if ((pf_ffmean >= (25600 - (130 * 64))) && (pf_ffmean < (59072 - (130 * 64))))
{
    // The last, flat bit of the curve. The ff1 value is the most important value.
    // I have set that ff1 is three times as important as pf.
    pf_ff = (pf + (3 * ff1)) / 4;

    if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
    {
        // pf_ff is similare to the old sensor value. Use pf_ff as new value.
        NewValueOK = 1;
    }
    else
    {
        //pf_ff is too far from the old sensor value. Use the old value.
        NewValueOK = 0;
    }
}
else
{
    // The meanvalue is out of the diagram range and cannot be used.
    NewValueOK = 0;
}
}
else
{
    // Neither ff1, pf1 or pf2 are close to each other.

    // Check if any value is close to the old sensor value. If it is, use this value.
    pf1_sens = labs(OldFinalSensorValue - pf1);
    pf2_sens = labs(OldFinalSensorValue - pf2);
    ff1_sens = labs(OldFinalSensorValue - ff1);
}

```

```

// Find the smallest difference
if (pf2_sens < pf1_sens)
{
    // pf2 is closer to old sensor value than pf1
    pf_ff = pf2;
    if (ff1_sens < pf2_sens)
    {
        // ff1 is closest to old sensor value
        pf_ff = ff1;
    }
}
else
{
    // pf1 is closer to old sensor value than pf2
    pf_ff = pf1;
    if (ff1_sens < pf1_sens)
    {
        // ff1 is closest to old sensor value
        pf_ff = ff1;
    }
}

if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
{
    // The closest value is close enough to the old sensor value
    NewValueOK = 1;
}
else
{
    // The closest value is NOT close enough to the old sensor value
    NewValueOK = 0;
}
}

// Set the sensor value
if (NewValueOK)
{
    // Set the new value.
    OldFinalSensorValue = pf_ff;
    OldCount = 0;
    return pf_ff;
}
else
{
    // Use the old sensor value.
    if (OldCount <= MAX_OLD_VALUES)
    {
        // Use old sensor value
        OldCount++;
        return OldFinalSensorValue;
    }
    else
    {
        // The old sensor value has been used too many times. Apply full brake force.
        return (923 - 130) * 64;
    }
}
} /* CompareAllSensorValues */

unsigned long int CompareTwoPFValues(unsigned long int pf1, unsigned long int pf2)
{
    unsigned long int pf1Diff, pf2Diff, pf;
    bool NewValueOK;

    // Use only the position sensor values to decide one final sensor value

```

```

if (labs(pf1-pf2) <= DIFF16BIT)
{
    // pf1 and pf2 are similare. Take the mean value of them.
    pf = (pf1 + pf2) / 2;

    Sens11FarAway = 0;
    Sens22FarAway = 0;

    if (labs(OldFinalSensorValue - pf) <= SENSORDIFF)
    {
        // The mean value is close to the old sensor value. Use the mean value.
        NewValueOK = 1;
    }
    else
    {
        // The mean value is far from the old sensor value. Use the old value.
        NewValueOK = 0;
    }
}
else
{
    // pf1 and pf2 differ. Compare them with the old sensor value and decide which one to use
    pf1Diff = labs(OldFinalSensorValue - pf1);
    pf2Diff = labs(OldFinalSensorValue - pf2);

    if (pf1Diff < pf2Diff)
    {
        // pf1 is closest to the old sensor value
        if (pf1Diff <= SENSORDIFF)
        {
            // pf1 is similare to the old sensor value. Use pf1 as new value.
            pf = pf1;
            NewValueOK = 1;
            Sens11FarAway = 0;
            Sens22FarAway++;
        }
        else
        {
            //pf1 is too far from the old sensor value. Use the old value.
            NewValueOK = 0;
            Sens11FarAway++;
            Sens22FarAway++;
        }
    }
    else
    {
        // pf2 is closest to the old sensor value
        if (pf2Diff <= SENSORDIFF)
        {
            // pf2 is similare to the old sensor value. Use pf2 as new value.
            pf = pf2;
            NewValueOK = 1;
            Sens11FarAway++;
            Sens22FarAway = 0;
        }
        else
        {
            //pf2 is too far from the old sensor value. Use the old value.
            NewValueOK = 0;
            Sens11FarAway++;
            Sens22FarAway++;
        }
    }
}

if (Sens11FarAway > CLUSTERCYCLESDOWN)
{
    OK11 = 0;
}

```

```

    }

    if (Sens22FarAway > CLUSTERCYCLESDOWN)
    {
        OK22 = 0;
    }

}

// Set the sensor value
if (NewValueOK)
{
    // Set the new value.
    OldFinalSensorValue = pf;
    OldCount = 0;
    return pf;
}
else
{
    // Use the old sensor value.
    if (OldCount <= MAX_OLD_VALUES)
    {
        // Use old sensor value
        OldCount++;
        return OldFinalSensorValue;
    }
    else
    {
        // The old sensor value has been used too many times. Apply full brake force.
        return (923 - 130) * 64;
    }
}
} /* CompareTwoPFValues */

unsigned long int ComparePFAndFFValues(unsigned long int pf, unsigned long int ff, int pf_type)
{
    unsigned long int pf_ffmean, pf_ff, pfDiff, ffDiff;
    bool NewValueOK;

    // Use one position sensor value and one force sensor value to decide one final value

    if (labs(pf - ff) <= DIFF16BIT)
    {
        // pf and ff are similare. Calculate a final value. This final value depends on
        // where on the position/force curve that pf and ff are.

        pf_ffmean = (pf + ff) / 2;

        Sens11FarAway = 0;
        Sens1221FarAway = 0;
        Sens22FarAway = 0;

        if ((pf_ffmean >= 0) && (pf_ffmean < (25600 - (130 * 64))))
        {
            // The first, steep bit of the curve. The pf value is the most important value.
            // I have set that pf is three times as important as ff.
            pf_ff = ((3 * pf) + ff) / 4;

            if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
            {
                // pf_ff is similare to the old sensor value. Use pf_ff as new value.
                NewValueOK = 1;
            }
            else
            {
                //pf_ff is too far from the old sensor value. Use the old value.

```

```

        NewValueOK = 0;
    }
}
else if ((pf_ffmean >= (25600 - (130 * 64))) && (pf_ffmean < (59072 - (130 * 64))))
{
    // The last, flat bit of the curve. The ff value is the most important value.
    // I have set that ff is three times as important as pf.
    pf_ff = (pf + (3 * ff)) / 4;

    if (labs(OldFinalSensorValue - pf_ff) <= SENSORDIFF)
    {
        // pf_ff is similare to the old sensor value. Use pf_ff as new value.
        NewValueOK = 1;
    }
    else
    {
        //pf_ff is too far from the old sensor value. Use the old value.
        NewValueOK = 0;
    }
}
else
{
    // The meanvalue is out of the diagram range and cannot be used.
    NewValueOK = 0;
}
}
else
{
    // pf and ff differ. Compare them with the old sensor value and decide which one to use
    pfDiff = labs(OldFinalSensorValue - pf);
    ffDiff = labs(OldFinalSensorValue - ff);

    if (pfDiff < ffDiff)
    {
        // pf is closest to the old sensor value
        if (pfDiff <= SENSORDIFF)
        {
            // pf is similare to the old sensor value. Use pf as new value.
            pf_ff = pf;
            NewValueOK = 1;

            if (pf_type == 1)
            {
                // pf1 is similar to the old sensor value
                Sens11FarAway = 0;
            }
            else if (pf_type == 2)
            {
                // pf2 is similar to the old sensor value
                Sens22FarAway = 0;
            }
        }
        else
        {
            //pf is too far from the old sensor value. Use the old value.
            NewValueOK = 0;
            Sens11FarAway++;
            Sens22FarAway++;
        }
        Sens1221FarAway++;
    }
    else
    {
        // ff is closest to the old sensor value
        if (ffDiff <= SENSORDIFF)
        {
            // ff is similare to the old sensor value. Use ff as new value.

```



```

        pf_ff = ff;
        NewValueOK = 1;
        Sens1221FarAway = 0;
    }
    else
    {
        // ff is too far from the old sensor value. Use the old value.
        NewValueOK = 0;
        Sens1221FarAway++;
    }
    Sens11FarAway++;
    Sens22FarAway++;
}
}
if (Sens11FarAway > CLUSTERCYCLESDOWN)
{
    OK11 = 0;
}

if (Sens1221FarAway > CLUSTERCYCLESDOWN)
{
    OK12 = 0;
    OK21 = 0;
}

if (Sens22FarAway > CLUSTERCYCLESDOWN)
{
    OK22 = 0;
}

// Set the sensor value
if (NewValueOK)
{
    // Set the new value.
    OldFinalSensorValue = pf_ff;
    OldCount = 0;
    return pf_ff;
}
else
{
    // Use the old sensor value.
    if (OldCount <= MAX_OLD_VALUES)
    {
        // Use old sensor value
        OldCount++;
        return OldFinalSensorValue;
    }
    else
    {
        // The old sensor value has been used too many times. Apply full brake force.
        return (923 - 130) * 64;
    }
}
} /* ComparePFAndFFValues */

unsigned long int DecideFromOneSensorValue(unsigned long f)
{
    // Decide the final value by using only one sensor value

    // Compare the new sensor value with the last used sensor value
    if (labs(OldFinalSensorValue - f) <= SENSORDIFF)
    {
        // The difference between the new and old value is small. Use the new value.
        OldFinalSensorValue = f;
        OldCount = 0;
        return f;
    }
}

```

```

else
{
    // The difference between the new and old value is big. Use the old value.
    if (OldCount <= MAX_OLD_VALUES)
    {
        // Use old sensor value
        OldCount++;
        return OldFinalSensorValue;
    }
    else
    {
        // The old sensor value has been used too many times. Apply full brake force.
        return (923 - 130) * 64;
    }
}
} /* DecideFromOneSensorValue */

```

```

unsigned long int FinalSensorValue(sensorval a, sensorval ab, sensorval ba, sensorval b)
{
    bool PF1OK, abOK, baOK, PF2OK, FF1OK;

    unsigned long int PF1, PF2, FF1, ff;

    // Initialise all bools to true. Tells if the sensorvalues should be used or not
    // after getting rid of the worst values in P2ReadSensorValue. PF means PositionForce
    // FF means ForceForce

    PF1OK = 1;
    abOK = 1;
    baOK = 1;
    PF2OK = 1;
    FF1OK = 1;

    if ( ( a == 0) || (!OK11) || (a == 1))
    {
        // Sensor11 is not valid
        PF1OK = 0;

        if (a == 0)
        {
            // The sensor value is out of range
            if (Sens11Cycles <= CLUSTERCYCLESDOWN)
            {
                Sens11Cycles++;
            }
            else
            {
                // Do not ever use sensor11 in the future
                OK11 = 0;
            }
        }
    }
}

if ( ( ab == 0) || (!OK12) || (ab == 1))
{
    // Sensor12 is not valid
    abOK = 0;

    if (ab == 0)
    {
        // The sensor value is out of range
        if (Sens12Cycles <= CLUSTERCYCLESDOWN)
        {
            Sens12Cycles++;
        }
        else
        {

```

```

        // Do not ever use sensor12 in the future
        OK12 = 0;
    }
}

if ( (ba == 0) || (!OK21) || (ba == 1))
{
    // Sensor21 is not valid
    baOK = 0;

    if (ba == 0)
    {
        // The sensor value is out of range
        if (Sens21Cycles <= CLUSTERCYCLESDOWN)
        {
            Sens21Cycles++;
        }
        else
        {
            // Do not ever use sensor21 in the future
            OK21 = 0;
        }
    }
}

if ( (b == 0) || (!OK22) || (b == 1))
{
    // Sensor22 is not valid
    PF2OK = 0;

    if (b == 0)
    {
        // The sensor value is out of range
        if (Sens22Cycles <= CLUSTERCYCLESDOWN)
        {
            Sens22Cycles++;
        }
        else
        {
            // Do not ever use sensor22 in the future
            OK22 = 0;
        }
    }
}

// Now we know which values that passed the the first valid value test
// We have also prepared for the P2ReadSensorValue to determine if a sensor is OK or not

// Decide one common value (if there is any) for sensor12 and sensor21, translate all values
// from 10 bits to 16 by multiplying by 64
if (abOK && baOK)
{
    // Sensor12 and sensor21 have to be reduced to one value
    // Transform to 16 bits value and compensate for the fact that the brake force
    // is zero when the force i less than 130 * 64

    ff = ReturnOneFFSensorValue(ab, ba, a, b);

    // The brakeforce is zero when ff is less than 130
    if (ff < 130)
    {
        FF1 = 0;
    }
    else
    {
        FF1 = (ff - 130) * 64;
    }
}

```

```

    }
}
else if (abOK || baOK)
{
    // There is just one valid sensor value. Use this one.
    if (abOK)
    {
        // Transform to 16 bits value and compensate for the fact that the brake force
        // is zero when the force is less than 130 * 64
        if (ab < 130)
        {
            FF1 = 0;
        }
        else
        {
            FF1 = (ab - 130) * 64;
        }
    }
    else
    {
        // Transform to 16 bits value and compensate for the fact that the brake force
        // is zero when the force is less than 130 * 64
        if (ba < 130)
        {
            FF1 = 0;
        }
        else
        {
            FF1 = (ba - 130) * 64;
        }
    }
}
else
{
    // There is no valid sensor12 or sensor21 value.
    FF1OK = 0;
}

// Depending on how many sensor values there is left try to decide one brake force value
if (PF1OK && PF2OK && FF1OK)
{
    // All three sensor values have values which can be compared
    PF1 = TranslatePositionToForce(a);
    PF2 = TranslatePositionToForce(b);
    return CompareAllSensorValues(PF1, PF2, FF1);
}
else if (PF1OK && PF2OK && !FF1OK)
{
    // Just the two position sensors have values which can be compared
    PF1 = TranslatePositionToForce(a);
    PF2 = TranslatePositionToForce(b);
    return CompareTwoPFValues(PF1, PF2);
}
else if ((PF1OK && FF1OK && !PF2OK) ||
        (PF2OK && FF1OK && !PF1OK))
{
    // One position sensor has a value and the force sensor has a value
    if (PF1OK)
    {
        PF1 = TranslatePositionToForce(a);
        return ComparePFAndFFValues(PF1, FF1, 1);
    }
    else
    {
        PF2 = TranslatePositionToForce(b);
    }
}

```

```

        return ComparePFAndFFValues(PF2, FF1, 2);
    }
}
else if (PF1OK || PF2OK || FF1OK)
{
    // There is only one sensor with a value that can be used
    if (PF1OK)
    {
        PF1 = TranslatePositionToForce(a);
        return DecideFromOneSensorValue(PF1);
    }
    else if (PF2OK)
    {
        PF2 = TranslatePositionToForce(b);
        return DecideFromOneSensorValue(PF2);
    }
    else
    {
        return DecideFromOneSensorValue(FF1);
    }
}
else
{
    // There is no sensor value to use
    if (OldCount <= MAX_OLD_VALUES)
    {
        // Use old sensor value
        OldCount++;
        return OldFinalSensorValue;
    }
    else
    {
        // The old sensor value has been used too many times. Apply full brake force.
        return (923 - 130) * 64;
    }
}
} /* FinalSensorValue */

```

```

void CalculateBrakeValue(unsigned long int sensorvalue)
{
    // Decide what brakevalue to send to the different brakenodes
    // depending on how many brakenodes that are alive

    // Initialize BrakeStatus to zero
    BrakeStatus = 0x0000;

    // Set BLnode BrakeStatus
    if (BrakeRunVector[0] == 1)
    {
        if (NodeVector[5] == 1)
        {
            BrakeStatus = BrakeStatus | 0x1000;
        }
    }

    // Set BRnode BrakeStatus
    if (BrakeRunVector[1] == 1)
    {
        if (NodeVector[3] == 1)
        {
            BrakeStatus = BrakeStatus | 0x0100;
        }
    }

    // Set FLnode BrakeStatus
    if (BrakeRunVector[2] == 1)
    {

```

```

    if (NodeVector[0] == 1)
    {
        BrakeStatus = BrakeStatus | 0x0010;
    }
}

// Set FRnode BrakeStatus
if (BrakeRunVector[3] == 1)
{
    if (NodeVector[4] == 1)
    {
        BrakeStatus = BrakeStatus | 0x0001;
    }
}

// Interpret the BrakeStatus, all cases
// Make sure the calculated values are within a 16 bits range before they are set.
if (BrakeStatus == 0x1111)
{
    // All brakenodes work
    BLbrake = sensorvalue;
    BRbrake = sensorvalue;
    FLbrake = sensorvalue;
    FRbrake = sensorvalue;
}
else if (BrakeStatus == 0x1110)
{
    // FRbrakenode unable to operate
    BLbrake = sensorvalue/3;
    BRbrake = sensorvalue/3;
    FLbrake = sensorvalue/3;
    FRbrake = 0;
}
else if (BrakeStatus == 0x1101)
{
    // FLbrakenode unable to operate
    BLbrake = sensorvalue/3;
    BRbrake = sensorvalue/3;
    FLbrake = 0;
    FRbrake = sensorvalue/3;
}
else if (BrakeStatus == 0x1011)
{
    // BRbrakenode unable to operate
    BLbrake = sensorvalue/3;
    BRbrake = 0;
    FLbrake = sensorvalue/3;
    FRbrake = sensorvalue/3;
}
else if (BrakeStatus == 0x0111)
{
    // BLbrakenode unable to operate
    BLbrake = 0;
    BRbrake = sensorvalue/3;
    FLbrake = sensorvalue/3;
    FRbrake = sensorvalue/3;
}
else if (BrakeStatus == 0x0110)
{
    // Only two diagonal brakenodes BR and FL are able to operate
    BLbrake = 0;
    BRbrake = sensorvalue/2;
    FLbrake = sensorvalue/2;
    FRbrake = 0;
}
else if (BrakeStatus == 0x1001)

```

```

{
    // Only two diagonal brakenodes BL and FR are able to operate
    BLbrake = sensorvalue/2;
    BRbrake = 0;
    FLbrake = 0;
    FRbrake = sensorvalue/2;
}
else if (BrakeStatus == 0x1100)
{
    // Only the back brakenodes are working
    BLbrake = sensorvalue/2;
    BRbrake = sensorvalue/2;
    FLbrake = 0;
    FRbrake = 0;
}
else if (BrakeStatus == 0x0011)
{
    // Only the front brakenodes are working
    BLbrake = 0;
    BRbrake = 0;
    FLbrake = sensorvalue/2;
    FRbrake = sensorvalue/2;
}
else if (BrakeStatus == 0x1010)
{
    // Only the left brakenodes are working
    BLbrake = sensorvalue/2;
    BRbrake = 0;
    FLbrake = sensorvalue/2;
    FRbrake = 0;
}
else if (BrakeStatus == 0x0101)
{
    // Only the right brakenodes are working
    BLbrake = 0;
    BRbrake = sensorvalue/2;
    FLbrake = 0;
    FRbrake = sensorvalue/2;
}
else if (BrakeStatus == 0x0001)
{
    // Just FRbrakenode is working
    BLbrake = 0;
    BRbrake = 0;
    FLbrake = 0;
    FRbrake = sensorvalue;
}
else if (BrakeStatus == 0x0010)
{
    // Just FLbrakenode is working
    BLbrake = 0;
    BRbrake = 0;
    FLbrake = sensorvalue;
    FRbrake = 0;
}
else if (BrakeStatus == 0x0100)
{
    // Just BRbrakenode is working
    BLbrake = 0;
    BRbrake = sensorvalue;
    FLbrake = 0;
    FRbrake = 0;
}
else if (BrakeStatus == 0x1000)
{
    // Just BLbrakenode is working
    BLbrake = sensorvalue;
}

```

```

        BRbrake = 0;
        FLbrake = 0;
        FRbrake = 0;
    }
    else
    {
        // None of the brakenodes are working
        BLbrake = sensorvalue/4;
        BRbrake = sensorvalue/4;
        FLbrake = sensorvalue/4;
        FRbrake = sensorvalue/4;
    }
} /* CalculateBrakeValue */

tt_task (P1ReadSensorValue)
{
    sens12 = SENSOR_VALUE_1; // Values from pedal are read from input here
    sens11 = SENSOR_VALUE_2;

    if ((100 <= sens12) && (sens12 <= 923) && (OK12))
    {
        // Valid pedal sensor value
        tt_Raw_Value(SENSOR12) = sens12;
        tt_Set_Sender_Status (SENSOR12, 1);
    }
    else
    {
        // Not valid pedal sensor value, set the value to zero
        tt_Raw_Value(SENSOR12) = 0;
        tt_Set_Sender_Status (SENSOR12, 1);
        sens12 = 0;
    }

    if ((100 <= sens11) && (sens11 <= 923) && (OK11))
    {
        // Valid pedal sensor value
        tt_Raw_Value(SENSOR11) = sens11;
        tt_Set_Sender_Status (SENSOR11, 1);
    }
    else
    {
        // Not valid pedal sensor value, set the value to zero
        tt_Raw_Value(SENSOR11) = 0;
        tt_Set_Sender_Status (SENSOR11, 1);
        sens11 = 0;
    }
} /* tt_task (P1ReadSensorValue) */

tt_task (P1CalculateSensorAndBrakeValue)
{
    int SensorVal, Sens22ReceiveStatus, Sens21ReceiveStatus;

    Sens22ReceiveStatus = tt_Receiver_Status(SENSOR22);
    Sens21ReceiveStatus = tt_Receiver_Status(SENSOR21);

    // Check if the received sensor messages are correct
    if ((Sens22ReceiveStatus > 0) && (Sens21ReceiveStatus > 0))
    {
        // Decide a final sensor value from the four, correct sensor values
        SensorVal = FinalSensorValue(sens11, sens12, SENSOR21, SENSOR22);
    }
    else if ((Sens22ReceiveStatus > 0) && !(Sens21ReceiveStatus > 0))
    {
        // Only sensor22 is correct of the received sensor messages
        // Handle the incorrect message as a not valid message, set the value to zero
        SensorVal = FinalSensorValue(sens11, sens12, 1, SENSOR22);
    }
}

```



```

else if (!(Sens22ReceiveStatus > 0) && (Sens21ReceiveStatus > 0))
{
    // Only sensor21 is correct of the received sensor messages
    // Handle the incorrect message as a not valid message, set the value to zero
    SensorVal = FinalSensorValue(sens11, sens12, SENSOR21, 1);
}
else
{
    // None of the received sensor messages are correct
    // Handle the incorrect messages as not valid messages, set the values to zero
    SensorVal = FinalSensorValue(sens11, sens12, 1, 1);
}

// Calculate the brake values for the four brakenodes. Use the final sensor value as input

CalculateBrakeValue(SensorVal);

// Make sure that the brake values are within 16 bits before they are sent.
// Send maximum brake value if something is wrong with the pedal node.

// Here it is possible to set PedalOK.
// If this is not written here the variable will always be true.
// PedalOK = ?

if (PedalOK)
{
    // The Pedal node is OK. Make sure that the brake values are within 16 bits.

    if (BLbrake < (923 * 64))
    {
        // The brake value is within the valid range (100 * 64) - (922 * 64)
        tt_Raw_Value(BL_BRAKE1) = BLbrake;
        tt_Set_Sender_Status (BL_BRAKE1, 1);
    }
    else
    {
        // The brake value is too big.
        // Set BL_BRAKE1 to the biggest value within the valid range.
        tt_Raw_Value(BL_BRAKE1) = 922 * 64;
        tt_Set_Sender_Status (BL_BRAKE1, 1);
    }

    if (BRbrake < (923 * 64))
    {
        // The brake value is within the valid range (100 * 64) - (922 * 64)
        tt_Raw_Value(BR_BRAKE1) = BRbrake;
        tt_Set_Sender_Status (BR_BRAKE1, 1);
    }
    else
    {
        // The brake value is too big.
        // Set BR_BRAKE1 to the biggest value within the valid range.
        tt_Raw_Value(BR_BRAKE1) = 922 * 64;
        tt_Set_Sender_Status (BR_BRAKE1, 1);
    }

    if (FLbrake < (923 * 64))
    {
        // The brake value is within the valid range (100 * 64) - (922 * 64)
        tt_Raw_Value(FL_BRAKE1) = FLbrake;
        tt_Set_Sender_Status (FL_BRAKE1, 1);
    }
    else
    {
        // The brake value is too big.
        // Set FL_BRAKE1 to the biggest value within the valid range.
        tt_Raw_Value(FL_BRAKE1) = 922 * 64;
    }
}

```

```

        tt_Set_Sender_Status (FL_BRAKE1, 1);
    }

    if (FRbrake < (923 * 64))
    {
        // The brake value is within the valid range (100 * 64) - (922 * 64)
        tt_Raw_Value(FR_BRAKE1) = FRbrake;
        tt_Set_Sender_Status (FR_BRAKE1, 1);
    }
    else
    {
        // The brake value is too big.
        // Set FR_BRAKE1 to the biggest value within the valid range.
        tt_Raw_Value(FR_BRAKE1) = 922 * 64;
        tt_Set_Sender_Status (FR_BRAKE1, 1);
    }
}
else
{
    // The pedal node is not OK. Set brake value to maximum.
    tt_Raw_Value(BL_BRAKE1) = 923 * 64;
    tt_Set_Sender_Status (BL_BRAKE1, 1);

    tt_Raw_Value(BR_BRAKE1) = 923 * 64;
    tt_Set_Sender_Status (BR_BRAKE1, 1);

    tt_Raw_Value(FL_BRAKE1) = 923 * 64;
    tt_Set_Sender_Status (FL_BRAKE1, 1);

    tt_Raw_Value(FR_BRAKE1) = 923 * 64;
    tt_Set_Sender_Status (FR_BRAKE1, 1);
}
} /* tt_task (PICalculateSensorAndBrakeValue) */

tt_task (StatusMessages)
{
    int CState, i;
    int BLa, BLb, BRa, BRb, FLa, FLb, FRa, FRb, P2a, P2b;

    tt_keep_alive();

    // Tell if both busses to the pedalsensors work or not

    // Get the status for the different N-frames sent on the bus from the different nodes
    BLa = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000098);
    BLb = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x000000A0);
    BRa = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000020);
    BRb = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000028);
    FLa = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000040);
    FLb = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000048);
    FRa = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000030);
    FRb = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000038);
    P2a = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000010);
    P2b = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000018);

    // All status values for the N-frames have to be equal to 8 if the bus is OK
    if ((BLa == 8) &&
        (BLb == 8) &&
        (BRa == 8) &&
        (BRb == 8) &&
        (FLa == 8) &&
        (FLb == 8) &&
        (FRa == 8) &&
        (FRb == 8) &&
        (P2a == 8) &&
        (P2b == 8))

```

```

{
    // Both of the two busses works
    tt_Raw_Value(P1_BUS_OK) = 1;
    tt_Raw_Value(P2_BUS_OK) = 1;
}
else
{
    // There is something wrong with both or one of the busses
    tt_Raw_Value(P1_BUS_OK) = 0;
    tt_Raw_Value(P2_BUS_OK) = 0;
}
tt_Set_Sender_Status (P1_BUS_OK, 1);
tt_Set_Sender_Status (P2_BUS_OK, 1);

// Set the SENSOR**_OK message to false if a sensor has been down for more than 2 seconds
tt_Raw_Value(P_SENSOR11_OK) = OK11;
tt_Set_Sender_Status (P_SENSOR11_OK, 1);

if (OK12 || OK21)
{
    // At least one of sensor12 and sensor 21 works
    tt_Raw_Value(P_SENSOR12_OK) = 1;
    tt_Set_Sender_Status (P_SENSOR12_OK, 1);
}
else
{
    // Both sensor12 and sensor21 are wrong
    tt_Raw_Value(P_SENSOR12_OK) = 0;
    tt_Set_Sender_Status (P_SENSOR12_OK, 1);
}
tt_Raw_Value(P_SENSOR22_OK) = OK22;
tt_Set_Sender_Status (P_SENSOR22_OK, 1);

// Messages from brakenodes telling if the communication on both busses works or not
tt_Raw_Value(P_FL_BUS_OK) = FL_BUS_OK;
tt_Set_Sender_Status (P_FL_BUS_OK, 1);

tt_Raw_Value(P_FR_BUS_OK) = FR_BUS_OK;
tt_Set_Sender_Status (P_FR_BUS_OK, 1);

tt_Raw_Value(P_BL_BUS_OK) = BL_BUS_OK;
tt_Set_Sender_Status (P_BL_BUS_OK, 1);

tt_Raw_Value(P_BR_BUS_OK) = BR_BUS_OK;
tt_Set_Sender_Status (P_BR_BUS_OK, 1);

// Messages telling if a brakenode has shut down itself or not
// Set the BrakeRunVector

if (tt_Receiver_Status(FL_RUN_OK) > 0)
{
    // The received message is correct. Update the BrakeRunVector
    BrakeRunVector[2] = FL_RUN_OK;
}
// If the received message is incorrect the old BrakeRunVector will remain unchanged
// and the last used FL_RUN_OK message will be used again
tt_Raw_Value(P_FL_RUN_OK) = FL_RUN_OK;
tt_Set_Sender_Status (P_FL_RUN_OK, 1);

if (tt_Receiver_Status(FR_RUN_OK) > 0)
{
    // The received message is correct. Update the BrakeRunVector
    BrakeRunVector[3] = FR_RUN_OK;
}
// If the received message is incorrect the old BrakeRunVector will remain unchanged
// and the last used FR_RUN_OK message will be used again

```

```

tt_Raw_Value(P_FR_RUN_OK) = FR_RUN_OK;
tt_Set_Sender_Status (P_FR_RUN_OK, 1);

if (tt_Receiver_Status(BL_RUN_OK) > 0)
{
    // The received message is correct. Update the BrakeRunVector
    BrakeRunVector[0] = BL_RUN_OK;
}
// If the received message is incorrect the old BrakeRunVector will remain unchanged
// and the last used BL_RUN_OK message will be used again
tt_Raw_Value(P_BL_RUN_OK) = BL_RUN_OK;
tt_Set_Sender_Status (P_BL_RUN_OK, 1);

if (tt_Receiver_Status(BR_RUN_OK) > 0)
{
    // The received message is correct. Update the BrakeRunVector
    BrakeRunVector[1] = BR_RUN_OK;
}
// If the received message is incorrect the old BrakeRunVector will remain unchanged
// and the last used BR_RUN_OK message will be used again
tt_Raw_Value(P_BR_RUN_OK) = BR_RUN_OK;
tt_Set_Sender_Status (P_BR_RUN_OK, 1);

// Messages telling if a brakenode has got problems or not
tt_Raw_Value(P_FL_OK) = FL_OK;
tt_Set_Sender_Status (P_FL_OK, 1);

tt_Raw_Value(P_FR_OK) = FR_OK;
tt_Set_Sender_Status (P_FR_OK, 1);

tt_Raw_Value(P_BL_OK) = BL_OK;
tt_Set_Sender_Status (P_BL_OK, 1);

tt_Raw_Value(P_BR_OK) = BR_OK;
tt_Set_Sender_Status (P_BR_OK, 1);

// Decide if a node is alive or not by doing an AND-operation
// on the CState membership vector. Put the answer in the NodeVector of type bool.
for (i = 0; i <= 5; i++)
{
    CState = (CNI_GET_CSTATE_MEMBER(0) & (NODEMASK << (i)));
    if (CState != 0)
    {
        NodeVector[i] = 1;
    }
    else
    {
        NodeVector[i] = 0;
    }
}

// Send node alive/not alive messages
tt_Raw_Value(P_FLNODE_OK) = NodeVector[0];
tt_Set_Sender_Status (P_FLNODE_OK, 1);

tt_Raw_Value(P_FRNODE_OK) = NodeVector[4];
tt_Set_Sender_Status (P_FRNODE_OK, 1);

tt_Raw_Value(P_BLNODE_OK) = NodeVector[5];
tt_Set_Sender_Status (P_BLNODE_OK, 1);

tt_Raw_Value(P_BRNODE_OK) = NodeVector[3];
tt_Set_Sender_Status (P_BRNODE_OK, 1);

```

```

    tt_Raw_Value(P_P1NODE_OK) = NodeVector[1];
    tt_Set_Sender_Status (P_P1NODE_OK, 1);

    tt_Raw_Value(P_P2NODE_OK) = NodeVector[2];
    tt_Set_Sender_Status (P_P2NODE_OK, 1);
} /* tt_task (StatusMessages) */

```

C code for one of the wheel brake nodes (BL_node)

```

#include "TTPos.h"
#include "led.h"
#include "TTPCftl.h"

#include "ttpc_msg.h"

#define DIFF16BIT 640 // Max diff. if two brake values will be regarded as similar, 16 bits
#define BRAKEDIFF 100 // Max diff. between the real brake force and the desired brake force

#define PULS_WIDTH      * (unsigned short int *) (0x100004) /* Variable for pulse width to PWM */

// Real brake force value from brake sensor. A/D input
unsigned short int RealBrake;

// Real wheel velocity from wheel velocity sensor. A/D input
unsigned short int WheelVelocity;

// Desired brake value from the pedal nodes. This value shall be sent as D/A output in a fully
// working program but is not implemented in this version.
unsigned short int DesiredBrake;

// This variable shall be set to false if there is a small problem with the brake node
// that does not affect the efficiency of it, or when both of the pedal nodes are not working
// and the brake has to brake firm and to wheel stop.
bool OK;

void init (void)
{
    /* turn on/off LEDs */
    LED_ON (LED_GREEN_1);
    LED_ON (LED_GREEN_2);
    LED_OFF (LED_RED_1);
    LED_OFF (LED_RED_2);

    RealBrake = 0;

    WheelVelocity = 0;

    DesiredBrake = 0;

    OK = 1;
}

unsigned short int DecideBrakeForceFromOnePedal(brakeval p)
{
    unsigned short int brake;

    if (p != (923 * 64))
    {
        // The brake value comes from a pedal node that works. Use this value.
        // Set OK to true.

        OK = 1;
        brake = p;
    }
}

```

```

    }
    else
    {
        // There is something wrong with the pedal node. Do not use this value.
        // Set OK to false and brake firmly.

        OK = 0;
        brake = 922 * 64; // This should be a smart brake algorithm for firm braking
    }

    return brake;
} /* DecideBrakeForceFromOnePedal */

unsigned short int DecideBrakeForceFromTwoPedals(brakeval p1, brakeval p2)
{
    unsigned short int brake;

    if ((p1 != (923 * 64)) && (p2 != (923 * 64)))
    {
        // There is no problems with any of the pedal nodes
        // Set OK to true

        OK = 1;
        if (abs(p1 - p2) <= DIFF16BIT)
        {
            // The two brake values are close to each other. Take the mean value of them.
            brake = (p1 + p2) / 2;
        }
        else
        {
            // There is a big difference between the two brake values. Take the biggest of them
            if (p1 > p2)
            {
                // p1 is biggest take this one
                brake = p1;
            }
            else
            {
                // p2 is biggest take this one
                brake = p2;
            }
        }
    }
    else if ((p1 == (923 * 64)) && (p2 != (923 * 64)))
    {
        // There is a problem with pedal node 1. Do not use this value.
        // Set OK to true

        OK = 1;
        brake = p2;
    }
    else if ((p1 != (923 * 64)) && (p2 == (923 * 64)))
    {
        // There is a problem with pedal node 2. Do not use this value.
        // Set OK to true

        OK = 1;
        brake = p1;
    }
    else
    {
        // There is something wrong with both of the pedal nodes.
        // Do not use any of the values. Brake as firmly as possible.
        // Set OK to false

        OK = 0;
    }
}

```

```

        brake = 922 * 64; // This should be a smart brake algorithm for firm braking
    }

    return brake;
} /* DecideBrakeForceFromTwoPedals */

bool DesiredBrakeEqualsRealBrake(unsigned short int desired, unsigned short int real)
{
    // Decide if the brake manage to brake as it is told to do.
    if (abs(desired - real) <= BRAKEDIFF)
    {
        // The real brake force is close to the desired brake force
        return 1;
    }
    else
    {
        // The real brake force is not close to the desired brake force
        return 0;
    }
} /* DesiredBrakeEqualsRealBrake */

tt_task (BLSetBrakeValue)
{
    int P2ReceiveStatus, P1ReceiveStatus;
    int BRa, BRb, FLa, FLb, FRa, FRb, P1a, P1b, P2a, P2b;

    tt_keep_alive();

    P1ReceiveStatus = tt_Receiver_Status(BL_BRAKE1);
    P2ReceiveStatus = tt_Receiver_Status(BL_BRAKE2);

    if ((P1ReceiveStatus > 0) && (P2ReceiveStatus > 0))
    {
        // Both of the brake messages from the two pedal nodes have arrived properly
        DesiredBrake = DecideBrakeForceFromTwoPedals(BL_BRAKE1, BL_BRAKE2);
    }
    else if ((P1ReceiveStatus > 0) && !(P2ReceiveStatus > 0))
    {
        // Only the brake message from pedal 1 has arrived properly
        DesiredBrake = DecideBrakeForceFromOnePedal(BL_BRAKE1);
    }
    else if (!(P1ReceiveStatus > 0) && (P2ReceiveStatus > 0))
    {
        // Only the brake message from pedal 2 has arrived properly
        DesiredBrake = DecideBrakeForceFromOnePedal(BL_BRAKE2);
    }
    else
    {
        // None of the brake messages from the two pedalnodes have arrived properly
        // Set OK to false and brake the car in a firm way.
        OK = 0;
        DesiredBrake = 922 * 64; // This should be a smart brake algorithm for firm braking
    }

    // Here you send DesiredBrake to the brake actuator in the final version of the program
    // You also read the input values to WheelVelocity and RealBrake
    PULS_WIDTH = DesiredBrake;

    // Set the status of the different boolean messages
    // Get the status for the different N-frames sent on the bus from the different nodes
    BRa = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000020);
    BRb = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000028);
    FLa = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000040);
    FLb = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000048);
    FRa = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000030);
    FRb = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000038);
}

```

```

P1a = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000050);
P1b = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000062);
P2a = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000074);
P2b = CNI_GET_MSGSTATE_EEIF( CNIMSGBASE + 0x00000086);

// All status values for the N-frames have to be equal to 8 if the bus is OK
if ((BRa == 8) &&
    (BRb == 8) &&
    (FLa == 8) &&
    (FLb == 8) &&
    (FRa == 8) &&
    (FRb == 8) &&
    (P1a == 8) &&
    (P1b == 8) &&
    (P2a == 8) &&
    (P2b == 8))
{
    // Both of the two busses works
    tt_Raw_Value(BL_BUS_OK) = 1;
}
else
{
    // There is something wrong with both or one of the busses
    tt_Raw_Value(BL_BUS_OK) = 0;
}
tt_Set_Sender_Status (BL_BUS_OK, 1);

tt_Raw_Value(BL_RUN_OK) = 1; /* DesiredBrakeEqualsRealBrake(DesiredBrake, RealBrake); */
tt_Set_Sender_Status (BL_RUN_OK, 1);

tt_Raw_Value(BL_OK) = OK;
tt_Set_Sender_Status (BL_OK, 1);

// Send the input values from the brake on the bus
tt_Raw_Value(BL_WHEELVEL) = WheelVelocity;
tt_Set_Sender_Status (BL_WHEELVEL, 1);

tt_Raw_Value(BL_BRAKEFORCE) = RealBrake;
tt_Set_Sender_Status (BL_BRAKEFORCE, 1);
}

```

C code for the sensor inputs to the pedal nodes (7 files - glob_def.h, linkf.cmd, main.c, QADC.c, QADC.h, start.src, Startup.c)

glob_def.h (input sensor values)

```

#ifndef GLOBAL_DEF_H
#define GLOBAL_DEF_H

/* typedefs according to misra */
typedef unsigned short int uint16;

#endif

```

linkf.cmd (input sensor values)

```
; linker command file for ttp!
```



```

chip CPU32

debug_symbols
listabs publics, internals
listmap publics/by_addr, internals

; load in ttp's ram

sect vectab      = $000000
sect code        = $001000
sect zerovars    = $20a000
sect lstack      = $20effe
common stack     = $20efff

sectsize stack   = $1000

INITDATAvars, tags

BASE $000000

; rom
order vectab
order code,??INITDATA,literals,strings,const
; ram
order zerovars, vars, tags,ioports,superstack
order lstack
order stack

; startup code
; alla obj-filer som ska vara med.
load start.obj
load startup.obj
load main.obj
load qadc.obj

;linker

load c:\mri\mcc68k\cpu32f\mcc68kab.lib

start $0

```

main.c (input sensor values)

```

#include "qadc.h"
#include "glob_def.h"

#define SENSOR_VALUE_1      * (unsigned short int *) (0x100000)
#define SENSOR_VALUE_2      * (unsigned short int *) (0x100002)

void main(void)
{
    InitQADC();
    while (1)
    {
        SENSOR_VALUE_1 = read_sensor_1();
        SENSOR_VALUE_2 = read_sensor_2();
    }
}

```

QADC.c (input sensor values)

```

#include "QADC.h"

```

```

void InitQADC(void)
{
    /*INPUT SAMPLE TIME QCLK*16, PQA0*/
    /* QADC_CCW0 = 0x00F4; */
    /* CCW1 = 0x003F; */

    /* STOP = 0 (Disable stop mode) */
    /* FRZ = 1 (Finish any current conversion, */
    /* then freeze on IMB internal */
    /* freeze signal) */
    /* SUPV = 0 */
    /* IARB[3:0] = 0 (no interrupts from QADC) */
    QADC_MCR = 0x4000;

    /*NO INTERRUPTS*/
    QADC_INT = 0x0000;

    /*PORT DDQA0 IS DEFINED AS INPUTS*/
    QADC_DDRQA = 0x0000;

    /* MUX = 0 (Internally multiplexed, 16 channels) */
    /* PSH[8:4] = %01011 (12 clock cycles prescaler clock high time) */
    /* PSA[3:2] = 0 (QCLK high and low times are not modified) */
    /* PSL[1:0] = %011 (4 clock cycles prescaler clock low time) */
    QADC_QACR0 = 0x00B3; /* 0x00A4; */

    /* Clear status register */
    QADC_QASR = 0x0000;

    /* CEI1 = 0 (Q1: Disable completion interrupt) */
    /* PIE1 = 0 (Q1: Disable pause interrupt) */
    /* SSE1 = 0 (Q1: No trigger events accepted) */
    /* MQ1 = 0 (Q1: disabled) */
    QADC_QACR1 = 0x0000;

    /*QUEUE 2, Mode 10101. Compl int off. Begin of Queue 2 = 0 */
    /* CIE2 = 0 (Q2: Disable completion interrupt) */
    /* PIE2 = 0 (Q2: Disable pause interrupt) */
    /* SSE2 = 0 (Q2: No trigger events accepted) */
    /* MQ2 = %10001 (Q2: Continuous-scan software triggered mode) */
    /* MQ2 = %11001 (Q2: Periodic timer Continuous-scan mode:Period = QCLK*2^12) */
    /* RES = 1 (Q2: After suspension, resume with aborted CCW) */
    /* BQ2 = 0 (Q2 starts at CCW0) */
    QADC_QACR2 = 0x1980; /* 0x1540; */

    /* Queue 2 */
    QADC_CCW0 = 0x00f0; //PQB4 Pinne 6
    QADC_CCW1 = 0x00c3; //PQB3 Pinne 7
    QADC_CCW2 = 0x003f; // End of Queue
}

uint16 read_sensor_1(void)
{
    return QADC_RJURR0;
}

uint16 read_sensor_2(void)
{
    return QADC_RJURR1;
}

```

QADC.h (input sensor values)

```
#ifndef QADC_H
```

```

#define QADC_H

typedef unsigned short int    uint16;

/* SETUP FOR QADC MODULE */
#define      QADCbase      0xfff200L
/* MODULE CONFIGURATION REGISTER */
#define      QADC_MCR      (*(uint16 *) (QADCbase))

/* QADC INTERRUPT REGISTER */
#define      QADC_INT      (*(uint16 *) (QADCbase + 0x04))

/* (PORT QA DATA REGISTER), (PORT QB DATA REGISTER) */
#define      QADC_PORTQAQB      (*(uint16 *) (QADCbase + 0x06))

/* PORT QA DATA DIRECTION REGISTER */
#define      QADC_DDRQA      (*(uint16 *) (QADCbase + 0x08))

/* QADC CONTROL REGISTER 0 */
#define      QADC_QACR0      (*(uint16 *) (QADCbase + 0x0A))

/* QADC CONTROL REGISTER 1 */
#define      QADC_QACR1      (*(uint16 *) (QADCbase + 0x0C))

/* QADC CONTROL REGISTER 2 */
#define      QADC_QACR2      (*(uint16 *) (QADCbase + 0x0E))

/* QADC STATUS REGISTER */
#define      QADC_QASR      (*(uint16 *) (QADCbase + 0x10))

/* Conversion Command Word Table */
/* #define      QADC_CCW      (*(unsigned short int *) (QADCbase + 0x30)) */
#define      QADC_CCW0      (*(uint16 *) (QADCbase + 0x30)) //0x30
#define      QADC_CCW1      (*(uint16 *) (QADCbase + 0x32)) //0x32
#define      QADC_CCW2      (*(uint16 *) (QADCbase + 0x34)) //0x34
/* RESULT WORD TABLE */
#define      QADC_RJURR0      (*(uint16 *) (QADCbase + 0xB0))
#define      QADC_RJURR1      (*(uint16 *) (QADCbase + 0xB2))

extern void InitQADC(void);
extern uint16 read_sensor_1(void);
extern uint16 read_sensor_2(void);

#endif /* #ifndef QADC_H */

```

start.src (input sensor values)

```

; Startup file
; VTD 1998

; Some needed sections
    CHIP CPU32
    COMMON stack
    SECT code

; Symbols to import
    XREF __initcopy      ;copy inital values to variables
    XREF _startup        ;vector table base register setup

; Symbols to export
    XDEF Start           ;label in this file

; Here is where all the fun begins! :)
; (or starts all over again if there has been a software reset)
    SECT code

```

```

Start:
    lea.l  .startof.(stack)+.sizeof.(stack),sp    ; set up stackpointer
    move.w  #$2700,sr

```

```

;Memory initializations from crt376.src

```

```

.....
    move.w  #$404F,$FFFA00
    lea    $FFF000,a1
    move.w  #$D300,$A04(a1)
    move.b  #$0C,$A21(a1)
    move.w  #$0005,$A48(a1)
    move.w  #$7831,$A4A(a1)
    move.w  #$1001,$A4C(a1)
    move.w  #$7BF1,$A4E(a1)
    move.w  #$2003,$A54(a1)
    move.w  #$7BB1,$A56(a1)
    move.w  #$00FF,$A44(a1)
    move.w  #$0000,$A46(a1)
    move.w  #$0040,$B44(a1)
    move.w  #$0000,$B46(a1)
    move.w  #$0800,$B40(a1)
    .....
;    finished with register initialisation
    jsr    __initcopy        ; Some variables have initial values...
    jmp    _startup          ; From startup we go to main()
    .....

```

```

; Restart when exception
_afException:
    jmp Start

```

```

;Just to have a startup stackpointer
sect lstack
_lstack:

```

```

; Vectortable from crt376.src

```

```

sect vectab
_vectab:
    dc.l  _lstack
    dc.l  Start
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException
    dc.l  _afException

```


dcbl 65,_afException

Startup.c (input sensor values)

```
extern void main(void);

void startup(void);

/* make the vector table section available to the startup code */
#pragma asm
    SECT    vectbl,2,R
    SECTION code,,C
#pragma endasm
/*=====
    Function    : startup
    Input       : None
    Output      : None
    Side effects: None
    Description :Initializes the vector base register and then calls the main function.
=====*/
void startup(void) {
    /* Assume we're called with interrupts off, set things up */

    /* Set up vector base register to point to the section
       containing the interrupt vectors.
    */
    asm(" move.l    #.STARTOF.(vectbl),d0");
    /* We need to subtract 8 from this value, because uTPK
       constructs a vector table going from vector 2 and
       forward. This means that the vector table actually
       starts two long words earlier than this value shows.
    */
    asm(" sub.l     #8,d0");
    asm(" movec    d0,vbr");

    /* Now it's time to get the application going! */
    main();
}
```

C code for the output PWM signal from the wheel brake nodes (6 files – linkf.cmd, main.c, Pwm.c, Pwm.h, start.src, Startup.c)

linkf.cmd (output PWM signal)

```
; linker command file for ttp!

chip CPU32

debug_symbols
listabs publics, internals
listmap publics/by_addr, internals

; load in ttp's ram

sect vectab      = $000000
sect code        = $001000
sect zerovars    = $20a000
sect lstack      = $20effe
```

```

common stack      = $20efff

sectsize stack    = $1000

INITDATAvars, tags

BASE $000000

; rom
order vectab
order code,??INITDATA,literals,strings,const
; ram
order zerovars, vars, tags,ioports,superstack
order lstack
order stack

; startup code
; alla obj-filer som ska vara med.
load start.obj
load startup.obj
load main.obj
load pwm.obj

;linker

load c:\mri\mcc68k\cpu32f\mcc68kab.lib

start $0

```

main.c (output PWM signal)

```

#include "PWM.h"

#define PULS_WIDTH          *(unsigned short int *) (0x100004)

void main(void)
{
    InitPWM();
    while (1)
    {
        SetPWM(PULS_WIDTH);
    }
}

```

Pwm.h (output PWM signal)

```

#ifndef PWM_H
#define PWM_H

typedef unsigned short int    uint16;

/* SETUP FOR PWM MODULE */
#define          PWMbase      0xFFFF400

/* PWM5SIC - PWM5 STATUS/INTERRUPT/CONTROL REGISTER */
#define          PWM5_SIC     (*(uint16 *) (PWMbase + 0x28))

/* PWM5A1 - PWM5A PERIOD REGISTER */
#define          PWM5A_PERIOD          (*(uint16 *) (PWMbase + 0x2A))

/* PWM5B1 - PWM5 PULSE WIDTH REGISTER*/
#define          PWM5_WIDTH           (*(uint16 *) (PWMbase + 0x2C))

```

```

/* CPCR - CPSM CONTROL REGISTER */
#define      CPCR_CPSM      (*(uint16 *) (PWMbase + 0x08))

extern void InitPWM(void);

extern void SetPWM(unsigned short int x);

#endif /* #ifndef PWM_H */

```

Pwm.c (output PWM signal)

```

#include "PWM.h"

void InitPWM(void)
{
    /* Initialise teh CPSM Control Register */
    /* PRUN = 1 */
    /* DIV23 = 0 */
    /* PSEL = 0 */
    CPCR_CPSM = 0x0008;

    /* Initialise the PWM5A Period Register */
    /* The period is set to 1048 */
    PWM5A_PERIOD = 0x0418;

    /* Initialise the PWM5 Pulse Width Register */
    /* A value chosen at random */
    PWM5_WIDTH = 0x0100;

    /* Initialise the PWM5 Status/Interrupt/Control Register */
    /* FLAG = 0 */
    /* IL = 0 */
    /* LOAD = 1 */
    /* POL = 0 */
    /* EN = 1 */
    /* CLK = 000 */
    PWM5_SIC = 0x0028;
}

void SetPWM(unsigned short int x)
{
    static short int counter = 1;
    unsigned short int width;
    unsigned long int LongBrakePWM;

    counter = (counter - 1) % 100;

    if (counter == 0)
    {
        /* The period is set to 1048 */
        /* PWM5A_PERIOD = 0x0418; */

        counter = 100;

        if (x < 35000u)
        {
            /* Transform 16 bit brake value to range between 0 - 1048 */
            LongBrakePWM = x * 100000ul;
            LongBrakePWM = LongBrakePWM / 35000u;
            LongBrakePWM = LongBrakePWM * 1048u;
            width = (short int) (LongBrakePWM / 100000ul);
        }
        else
        {
            width = 1048;
        }
    }
}

```



```

    }

    /* Set the pulse width to x */
    PWM5_WIDTH = width;
}
}

```

start.src (output PWM signal)

```

; Startup file
; VTD 1998

; Some needed sections
    CHIP CPU32
    COMMON stack
    SECT code

; Symbols to import
    XREF __initcopy      ;copy initial values to variables
    XREF _startup        ;vector table base register setup

; Symbols to export
    XDEF Start           ;label in this file

; Here is where all the fun begins! :)
; (or starts all over again if there has been a software reset)
    SECT code
Start:
    lea.l .startof.(stack)+.sizeof.(stack),sp ; set up stackpointer
    move.w #$2700,sr

;Memory initializations from crt376.src
;.....
    move.w #$404F,$FFFA00
    lea    $FFF000,a1
    move.w #$D300,$A04(a1)
    move.b #$0C,$A21(a1)
    move.w #$0005,$A48(a1)
    move.w #$7831,$A4A(a1)
    move.w #$1001,$A4C(a1)
    move.w #$7BF1,$A4E(a1)
    move.w #$2003,$A54(a1)
    move.w #$7BB1,$A56(a1)
    move.w #$00FF,$A44(a1)
    move.w #$0000,$A46(a1)
    move.w #$0040,$B44(a1)
    move.w #$0000,$B46(a1)
    move.w #$0800,$B40(a1)
;.....
;    finished with register initialisation
    jsr    __initcopy      ; Some variables have initial values...
    jmp    _startup        ; From startup we go to main()
;.....
; Restart when exception
_afException:
    jmp    Start

;Just to have a startup stackpointer
sect lstack
_lstack:

; Vectortable from crt376.src
sect vectab
_vectab:
    dc.l _lstack

```



```

dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dc.l _afException
dcb.l 37,_afException
dcb.l 32,_afException
dcb.l 16,_afException
dcb.l 7,_afException
dc.l _afException
dcb.l 8,_afException
dcb.l 65,_afException

```

Startup.c (output PWM signal)

```

extern void main(void);

void startup(void);

/* make the vector table section available to the startup code */
#pragma asm
    SECT    vectbl,2,R
    SECTION code,,C
#pragma endasm
/*=====
    Function    : startup
    Input       : None
    Output      : None
    Side effects: None
    Description :Initializes the vector base register and then calls the main function.
=====*/
void startup(void) {
    /* Assume we're called with interrupts off, set things up */

    /* Set up vector base register to point to the section
       containing the interrupt vectors.
    */
    asm(" move.l    #.STARTOF.(vectbl),d0");
    /* We need to subtract 8 from this value, because uTPK
       constructs a vector table going from vector 2 and
       forward. This means that the vector table actually
       starts two long words earlier than this value shows.
    */
    asm(" sub.l     #8,d0");
    asm(" movec    d0,vbr");

    /* Now it's time to get the application going! */
    main();
}

```

Appendix D

- The task schedules for the six nodes in the Brake-By-Wire system

Task schedule for wheel brake node BL

Task Schedule for 'Normal Mode' -- period `4000`

- 0 : 'App_Task_local_time_Chain_0001'
 - ▶ 'BLSetBrakeValue'
Deadline `1000`, time_budget `350`, bcet `10`
- 350 : 'FT_Task_reference_time_Chain_0001'
 - 'FT_S_0'
Deadline `439`, time_budget `89`, bcet `1`
- 1695 : 'FT_Task_reference_time_Chain_0002'
 - ▶ 'FT_S_1'
Deadline `1862`, time_budget `167`, bcet `1`
- 3085 : 'FT_Task_reference_time_Chain_0003'
 - ▶ 'FT_R_2'
Deadline `3265`, time_budget `180`, bcet `1`
- 3770 : '_GroundTaskChain_ref'
 - 'tt_ground_state_task_ref'
Deadline `3895`, time_budget `125`, bcet `10`
- 3895 : '_GroundTaskChain_loc'
 - 'tt_ground_state_task_loc'
Deadline `4000`, time_budget `105`, bcet `10`

Expand all Search Save Close

Task schedule for wheel brake node BR

Task Schedule for 'Normal Mode' -- period `4000`

- 0 : 'App_Task_local_time_Chain_0001'
 - ▶ 'BRSetBrakeValue'
Deadline `1000`, time_budget `350`, bcet `10`
- 350 : 'FT_Task_reference_time_Chain_0001'
 - ▶ 'FT_S_1'
Deadline `517`, time_budget `167`, bcet `1`
- 1085 : 'FT_Task_reference_time_Chain_0002'
 - 'FT_S_0'
Deadline `1174`, time_budget `89`, bcet `1`
- 3085 : 'FT_Task_reference_time_Chain_0003'
 - ▶ 'FT_R_2'
Deadline `3265`, time_budget `180`, bcet `1`
- 3770 : '_GroundTaskChain_ref'
 - 'tt_ground_state_task_ref'
Deadline `3895`, time_budget `125`, bcet `10`
- 3895 : '_GroundTaskChain_loc'
 - 'tt_ground_state_task_loc'
Deadline `4000`, time_budget `105`, bcet `10`

Expand all Search Save Close

Task schedule for wheel brake node FL



Task Schedule for 'Normal Mode' -- period `4000`

- 0 : 'App_Task_local_time_Chain_0001'
 - ▶ 'FLSetBrakeValue'
Deadline `1000`, time_budget `350`, bcet `10`
- 350 : 'FT_Task_reference_time_Chain_0001'
 - ▶ 'FT_S_0'
Deadline `517`, time_budget `167`, bcet `1`
- 2000 : 'FT_Task_reference_time_Chain_0002'
 - 'FT_S_1'
Deadline `2089`, time_budget `89`, bcet `1`
- 3085 : 'FT_Task_reference_time_Chain_0003'
 - ▶ 'FT_R_2'
Deadline `3265`, time_budget `180`, bcet `1`
- 3770 : '_GroundTaskChain_ref'
 - 'tt_ground_state_task_ref'
Deadline `3895`, time_budget `125`, bcet `10`
- 3895 : '_GroundTaskChain_loc'
 - 'tt_ground_state_task_loc'
Deadline `4000`, time_budget `105`, bcet `10`

Expand all Search Save Close

Task schedule for wheel brake node FR

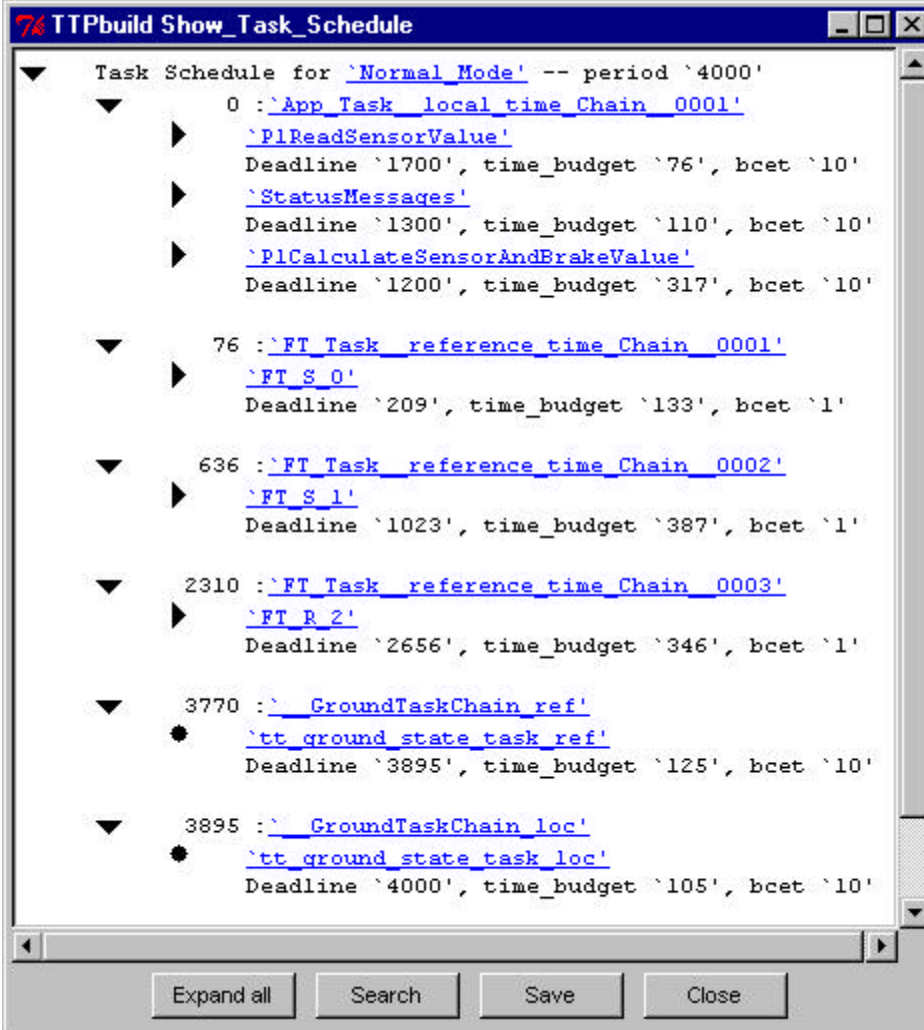
74 TTPbuild Show_Task_Schedule

Task Schedule for 'Normal Mode' -- period `4000`

- ▼ 0 : 'App_Task_local_time_Chain_0001'
 - ▶ 'FRSetBrakeValue'
Deadline `1000`, time_budget `350`, bcet `10`
- ▼ 350 : 'FT_Task_reference_time_Chain_0001'
 - ▶ 'FT S 0'
Deadline `517`, time_budget `167`, bcet `1`
- ▼ 1390 : 'FT_Task_reference_time_Chain_0002'
 - 'FT S 1'
Deadline `1479`, time_budget `89`, bcet `1`
- ▼ 3085 : 'FT_Task_reference_time_Chain_0003'
 - ▶ 'FT R 2'
Deadline `3265`, time_budget `180`, bcet `1`
- ▼ 3770 : '_GroundTaskChain_ref'
 - 'tt_ground_state_task_ref'
Deadline `3895`, time_budget `125`, bcet `10`
- ▼ 3895 : '_GroundTaskChain_loc'
 - 'tt_ground_state_task_loc'
Deadline `4000`, time_budget `105`, bcet `10`

Expand all Search Save Close

Task schedule for pedal node P1



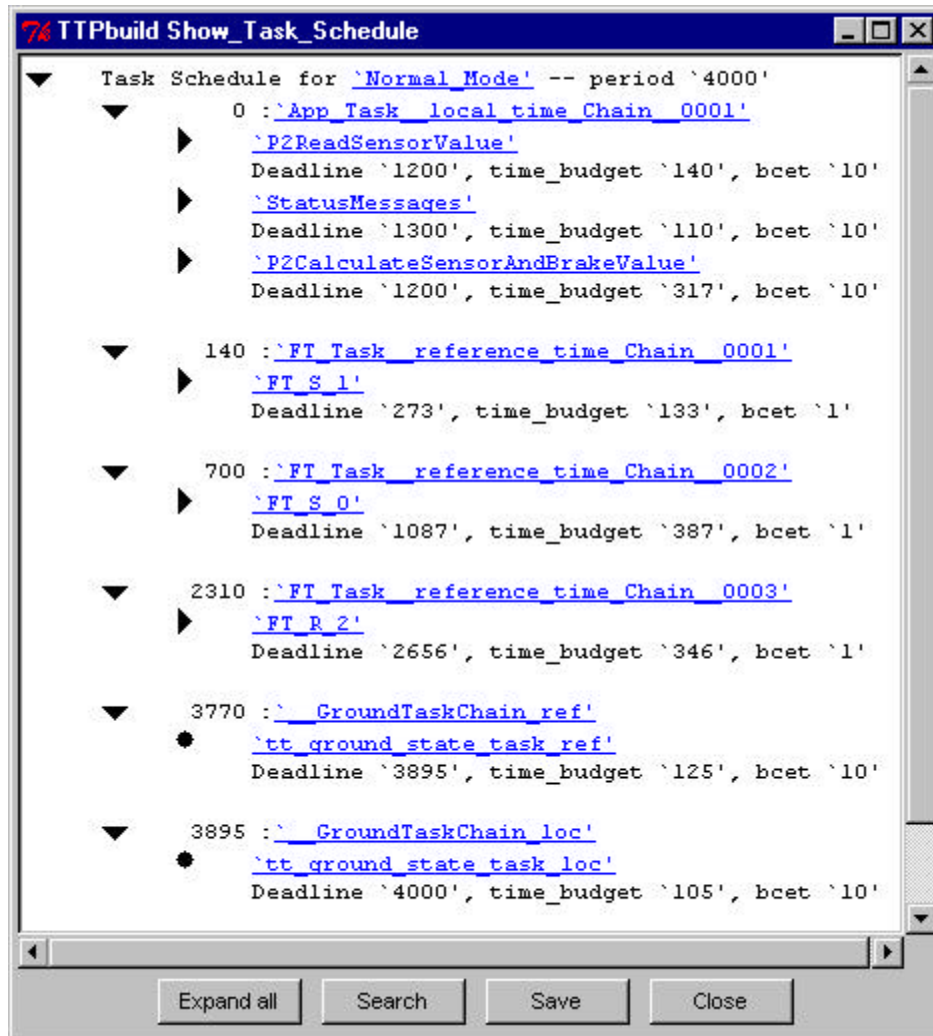
74 TTPbuild Show_Task_Schedule

Task Schedule for 'Normal Mode' -- period `4000`

- 0 : 'App_Task_local_time_Chain_0001'
 - ▶ 'PIReadSensorValue'
Deadline `1700`, time_budget `76`, bcet `10`
 - ▶ 'StatusMessages'
Deadline `1300`, time_budget `110`, bcet `10`
 - ▶ 'PICalculateSensorAndBrakeValue'
Deadline `1200`, time_budget `317`, bcet `10`
- 76 : 'FT_Task_reference_time_Chain_0001'
 - ▶ 'FT_S_0'
Deadline `209`, time_budget `133`, bcet `1`
- 636 : 'FT_Task_reference_time_Chain_0002'
 - ▶ 'FT_S_1'
Deadline `1023`, time_budget `387`, bcet `1`
- 2310 : 'FT_Task_reference_time_Chain_0003'
 - ▶ 'FT_R_2'
Deadline `2656`, time_budget `346`, bcet `1`
- 3770 : '_GroundTaskChain_ref'
 - 'tt_ground_state_task_ref'
Deadline `3895`, time_budget `125`, bcet `10`
- 3895 : '_GroundTaskChain_loc'
 - 'tt_ground_state_task_loc'
Deadline `4000`, time_budget `105`, bcet `10`

Expand all Search Save Close

Task schedule for pedal node P2



The screenshot shows a window titled "TTPbuild Show_Task_Schedule" with a blue title bar. The main content area displays a task schedule for "Normal Mode" with a period of 4000. The schedule is organized into several task chains, each with a start time and a list of tasks. The tasks are listed with their respective deadlines, time budgets, and bcet values.

```
Task Schedule for 'Normal Mode' -- period '4000'  
▼ 0 : 'App Task local time Chain 0001'  
▶ 'P2ReadSensorValue'  
Deadline '1200', time_budget '140', bcet '10'  
▶ 'StatusMessages'  
Deadline '1300', time_budget '110', bcet '10'  
▶ 'P2CalculateSensorAndBrakeValue'  
Deadline '1200', time_budget '317', bcet '10'  
  
▼ 140 : 'FT Task reference time Chain 0001'  
▶ 'FT_S_1'  
Deadline '273', time_budget '133', bcet '1'  
  
▼ 700 : 'FT Task reference time Chain 0002'  
▶ 'FT_S_0'  
Deadline '1087', time_budget '387', bcet '1'  
  
▼ 2310 : 'FT Task reference time Chain 0003'  
▶ 'FT_R_2'  
Deadline '2656', time_budget '346', bcet '1'  
  
▼ 3770 : 'GroundTaskChain_ref'  
● 'tt_ground_state_task_ref'  
Deadline '3895', time_budget '125', bcet '10'  
  
▼ 3895 : 'GroundTaskChain_loc'  
● 'tt_ground_state_task_loc'  
Deadline '4000', time_budget '105', bcet '10'
```

At the bottom of the window, there are four buttons: "Expand all", "Search", "Save", and "Close".

References

- [1] Günter Heiner and Thomas Thurner. Time-Triggered Architecture for Safety-Related Distributed Real-Time Systems in Transportation Systems. Fault-Tolerant Computing Symposium, Munich, 1998.
- [2] B. Hedenetz and R. Belschner. Brake-By-Wire without Mechanical Backup by Using a TTP-Communication Network. Society of Automotive Engineers, SAE Paper 981109, 1998.
- [3] Ross T. Bannatyne. Electronic braking control developments. Automotive Engineering International, February 1999.
- [4] TTPbuild – The Node Design Tool for the Time-Triggered Protocol TTP/C, TTTech Computertechnik AG, Vienna, Austria, 2001.
- [5] TTTech homepage: <http://www.tttech.com/>
- [6] TTPnode - A TTP Development Board for the Time-Triggered Architecture, document number D-NODE-M-HW-001. TTTech, Vienna, Austria, 2001.
- [7] FlexRay. Slides from the FlexRay International Workshop. Munich, Gemany, 2002.
- [8] Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, Michael Walther. Time Triggered Communication on CAN (Time Triggered CAN - TTCAN). CAN in Automation, International CAN Conference, Erlangen, Germany, 2002.
<http://212.114.78.132/can/ttcan/fuehrer.pdf>
- [9] Pär Björklund and Per Drougge. Distribuerad Brake-By-Wire-demonstrator baserad på TTP/C kommunikation. Volvo Technological Development, Chalmers University of Technology, Göteborg, 2000.
- [10] J. Bolin and J. Hedberg. Implementation of a Distributed Control Application Based on the TTP/C Architecture, Volvo Technological Development, 1999.
- [11] The Motor Industry Research Association. Guidelines For The Use Of The C Language In Vehicle Based Software. ISBN 0 9524156 9 0, 1999.

