Fault Tolerance in Real-time Distributed System Using the CT Library

Hany Ferdinando

Master's Thesis
Fault-tolerance in real time distributed system using The CT Library
Fault tolerance is more and more important nowadays. The role of computer in industries cannot be neglected. The computers that control machines are often inter-connected via fieldbuses. This can be CAN (Control Area Network), I²C (Inter Integrated Circuit), Spacewire, etc.

The goal of this project is to build a fault tolerant distributed system using the CT (Communicating Thread) Library. The CT Library offers an Application Programming Interface (API) for CSP (Communicating Sequential Process) programming in popular programming languages.

Fault tolerance itself is too general. Therefore, this project implemented the fault tolerance on communication links in a distributed system. In case of a network failure, the system will switch to the alternative one.

The system was implemented on the ADSP-21992 EZ-KIT LITE board from Analog Devices, Inc. This chip has a CAN bus interface for Control Area Network application.

This project used the CTC (the CT Library for C). In the application on a specific processor, the CT Library needs addition of hardware dependent part, encapsulated in link driver objects and the context switch mechanism. The link driver part will encapsulate all hardware related operation into specific place in order to maintain portability of the CT Library. Therefore, a programmer can use the same program on different processors by changing its link drivers and context switch. This enables single processor to do parallel processing. This is a kernel function to handle parallel programming in a general-purpose processor.

Here, the link driver concept of Hilderink was extended by dividing the link driver object into a Remote Link Driver and a Network Device Driver. The Remote Link Driver is a link driver that provides general purposes methods for communication, while the Network Device Driver is abstract class of all board and fieldbus protocol dependent part of communication.

This project used two communication links, i.e. CAN (Control Area Network) and SPORT (Serial PORT). Both need their own link driver.

The experiments shows that fault tolerance can be built in the CT Library framework easily by exploiting the hardware dependent part, i.e. the link driver. In a normal operation, the ADSP-21992 EZ-KIT LITE board is driven by 16MHz clock. This makes the CAN module cannot be used in a hard-real time control loop due to its latency. But with a software manipulation, that clock can be raised up to 160MHz and the latency will be decreased.
Fault-tolerance in real time distributed system using The CT Library
Preface

With this project, I finished my study to get a Master degree at the University of Twente, the Netherlands. I am interested in embedded control systems since more and more devices are built using this approach. Using a processor in embedded application is a challenging task, because one has to deal with limited resources.

There are lots of people whom I cannot forget. Prof. Job van Amerongen was the first person I met in this group. He is the program director of my study. I discussed all my subjects with him. Through his guidance, I could choose my subjects fit to my specialization.

I would like to thank to Jan Broenink as my supervisor. He has taught me how to manage my project well. Sometimes he visited me while I am working in my project and we had a talk for a while. Kruidnoten! I could not forget these cookies. He made me to taste it.

Bojan, my daily supervisor has guided me to finish my thesis. He always came to visit me when he went out from his room. From our discussion, I got many inspirations to solve my problem in programming.

Gerald was also helpful. He taught me how to write a link driver for my thesis. When I was programming this link driver, he always asked me how my link driver is. This gave me such a motivation to make it works. Thank you, Gerald!

Also to my friends in Embedded System group: Dusko, Peter, Thiemo and Marcel. We had a great time every Tuesday morning for discussion. You guys have been my supporter for my thesis.

I also got lots of support from my International Bible Study group. Thank you for your prayer. I could not make this without God’s help and you all have done great things in my life.

My friends in Petra Christian University, Surabaya, Indonesia never stopped to encourage me to finish my study. They have done their best effort to support me.

Last but not least, I know that lots of support has come from my own family. They supported me with their own way and I like it.

I know that I can do all things through Christ who strengthens me (Philippians 4:13)

Enschede, January 2004

Hany Ferdinando
Fault-tolerance in real time distributed system using The CT Library
1 Introduction
1.1 Context 1
1.2 Goal and Constraint 3
1.3 Outline of the Report 4

2 Background
2.1 Fault tolerance – General Overview 5
  2.1.1 Fault – Error – Failure 5
  2.1.2 Role of Fault Tolerance 6
  2.1.3 Fault Tolerance Method 6
2.2 Distributed System 7
  2.2.1 Physical Model 7
  2.2.2 Logical Model 8
2.3 ROPES Development Cycle 8
2.4 The CT Library 9
  2.4.1 Context Switch 9
  2.4.2 Link Driver 9
2.5 The CAN bus 10
2.6 The SPORT (Serial Port) 13
2.7 The ADSP-21992 14
  2.7.1 CAN in the ADSP-21992 15
  2.7.2 SPORT in the ADSP-21992 16
2.8 Hardware of the Plant 17

3 Software and Hardware Design 19
3.1 Context Switch 19
Fault-tolerance in real time distributed system using The CT Library

3.1.1 setjmp() function 20
3.1.2 longjmp() function 20
3.2 Embedded System Requirement 21
3.3 Extension of Link Driver Concept 21
3.4 Description of Transmit and Receive Process 24
3.5 Fault Tolerance in the CT Library 25
3.6 Hardware of the Plant 26
3.6.1 Potentiometer 27
3.6.2 H-bridge 27
3.6.3 LED indicator of link availability 27

4 Implementation 29
4.1 Object-Oriented Programming in C 29
4.2 Context Switch Implementation 29
4.3 Peripheral LinkDriver Implementation 30
4.3.1 ADCLinkDriver 30
4.3.2 AuxPWMLinkDriver 31
4.3.3 EIULinkDriver 31
4.4 Communication LinkDriver Implementation 32
4.4.1 RemoteLinkDriver 33
4.4.2 NetworkDeviceDriver 33
4.5 Altering Code to Achieve Fault Tolerance 35

5 Experimental Results 37
5.1 Experiment with simple producer-consumer 37
5.2 Experiment with LINIX – one-way configuration 40
5.3 Fault tolerance experiment using one-way configuration 41

6 Conclusions and Recommendations 45
6.1 Conclusions 45
6.2 Recommendations 45
<table>
<thead>
<tr>
<th>Appendix</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A</td>
<td>Description and Specific Properties of CAN and SPORT in the ADSP-21992</td>
<td>47</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Experiments performed</td>
<td>55</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Source Code of ADSP-21992 Specific Parts</td>
<td>63</td>
</tr>
<tr>
<td>Reference</td>
<td></td>
<td>69</td>
</tr>
</tbody>
</table>
Fault-tolerance in real time distributed system using The CT Library
1.1 Context

In industries, the use of computers cannot be ignored. It relates to quality control, data acquisition for various management levels, etc. Computers control almost each machine and most computers are usually also connected via a certain network. Therefore, the computers can communicate to each other. We call it a distributed system. Figure 1.1 shows the configuration of a distributed system.

Due to increasing complexity of requirements and applied control algorithms, more and more processing power is needed. There is a trend in control industry to implement control systems as distributed, by delegating part of the work from central computer to intelligent controllers located near the drives. In this way system architecture becomes more modular and resilient to node and/or network failures. Networks provide means to download code into the boards, to change parameters online, identify devices on network, and transfer sensor, actuator and calculation data between nodes. To reduce wiring costs and associated problems, fieldbus architectures are often used.

Figure 1.1 Distributed systems with several nodes connected to a computer and a robot.
In a distributed system, several computers work in parallel and they can share their data via a fieldbus. Now, the problem becomes more and more complex, because the system consists of more computers instead of only one. Concurrency is the main idea. Here, this kind of system should be able to handle conditions such as deadlock, livelock, starvation, alarm shower, etc. These should be solved because their effect can be disastrous to the whole system.

In order to make a system robust to those kinds of condition, fault tolerance is needed. A system with fault tolerance can be considered as a robust system. It means the system is built less sensitive to some error. In general, there are two types of fault tolerance, i.e. hardware fault tolerance and software fault tolerance. This project will use software fault tolerance. For software fault tolerance, many techniques have been developed nowadays, but the goal is one, i.e. to build a reliable system. Unfortunately, there is no general fault tolerance technique for all problems; a problem is specific and its solution as well. Fault tolerance implemented in this project is concerned with switching from broken network to the available one. Therefore, when system detects, that the current network is broken; system replaces it with the available one.

The CSP (Communicating Sequential Process) is a notation for describing concurrent systems whose component processes interact with each other by communication (Hilderink et al., 2000). The CSP programming concept has been successfully implemented in Occam programming language, used so far mostly in a specific kind of processors known as transputers (Welch et al., 1993).

This project will use the Communicating Thread (CT) library made by Hilderink (Hilderink et al., 2000). The CT library implements occam-like processes, constructs and channels offering the API (Application Programmer Interface) of the CSP in popular programming languages. This library was developed after the transputer disappeared from the market. There are three libraries for CT, i.e. CT for Java (CTJ), CT for C (CTC) and CT for C++ (CTCPP). This thesis will use CTC.

To use the CT library on a specific processor, one should write hardware dependent parts, i.e. a context switch and a link driver. The context switch deals with how to handle transition from one thread to the next one. Every thread has its own context or state. When there is transition from one thread to another, this state should be saved to stack. This state will be restored from stack when the system jumps back to the thread.

In the CSP, processes communicate via channel. Channels between processes on one processor use built-in memory driver, while channels between processes on different processor use peripheral driver. These drivers are called link drivers (Hilderink et al., 2000). Essentially, link driver is object in which all hardware dependent code needed for proper operation of channel is encapsulated. This concept of separating hardware dependent and independent part came from (Hilderink et al., 2000). With this concept, if one wants to run the same program on different processor or to use different peripherals, only changes are needed in the hardware dependent part of context switch and/or link drivers, leaving the rest of the code intact. This makes the link driver concept very suitable for implementing the CT library in a distributed system. Figure 1.2 shows this diagram.

This project uses a CAN (Control Area Network) bus and SPORT (Serial PORT) to connect processes on different processors. The CAN bus is one of field buses used in industries nowadays. The maximum data transfer rate for the CAN bus is 1 Mbps. The CAN bus uses
specific ID for each message, in other words the addressing is message based instead of location based.

![Diagram of Process A and Process B](image)

Figure 1.2 Separation between hardware dependent part and hardware independent part (Hilderink et al., 2000)

The SPORT is a point-to-point communication link. Various serial communication protocols, like RS-232, can be based on SPORT links.

The fault tolerance implemented in this project deals with handling communication link failure. Therefore, when the system detects that the current communication link failed, that link will be replaced with the available one. The system always uses the available communication link with the highest priority. If the higher priority link is recovered, the system will switch back to this link.

### 1.2 Goal and Constraint

The goal of this project is to implement fault tolerance in a distributed system using the CT library. The link driver concept will be implemented for the CAN bus and the SPORT.

For there are many methods in the software fault tolerance, this project chose to implement fault tolerance to handle the broken link. In this idea, every node will have two communication links. If the primary link fails, the system communication will be transferred automatically to the alternative one. The CAN bus will be the primary link, while the SPORT will be the alternative one. The system will use two ADSP-21992 EZ-KIT LITE boards from Analog Devices. Other important constraint is that this thesis will use a simple transmitted data instead of data in structure.

For this project emphasizes on the implementation of fault tolerance using the CT Library, the design of additional circuit, controller design and detailed information about the plant are not explained.
1.3 Outline of the report

Chapter 2 carries out the background to implement this project. Fault tolerance part explains the difference between fault, error and failure. It includes the role of fault tolerance and method to implement it. The overview of a distributed system is also explained here. This project uses the ROPES (Rapid Object-oriented Process of Embedded System) software development approach. A brief discussion about ROPES is found here. This chapter discusses the CT Library, the heart of this project. The discussion emphasizes on the link driver and context switch concept. The hardware part such as the ADSP-21992, its communication link (CAN and SPORT) and the plant close this chapter.

Chapter 3 explains about the software and hardware design in this project. Especially, the design of the hardware dependent part, i.e. the link driver and the context switch for this project, are discussed here. A brief overview of the hardware part finishes this chapter.

Chapter 4 brings the design into implementation. The motivation behind several implementations will be explained here. Detailed information about dedicated link drivers for the ADSP-21992 is found here. It includes the implementation of fault tolerance in the Remote Link Driver class.

All experiments are presented in chapter 5.

Chapter 6 closes this discussion with conclusions from this project. Several recommendations for the next phase of this project are also presented here.

For the completeness, appendices are added.
Chapter 2

Background

2.1 Fault tolerance – general overview

Fault tolerance is an approach by which the reliability of a computer system can be increased (Jalote, 1994). Fault tolerance itself is not a new area. There are many algorithms developed in order to get a better approach. Generally, fault tolerance consists of two main areas, i.e. the hardware and the software fault-tolerance.

Fault tolerance is a part of a larger set called dependability of a computer system. The development of a dependable computing system calls for the combined utilization of a set of four techniques: fault prevention, fault tolerance, fault removal and fault forecasting (Avizienis et al.).

2.1.1 Fault – error – failure

To deal with fault tolerance, one needs to understand concepts of fault, error and failure. Sometimes, it is difficult to distinguish them. According to (Jalote, 1994), fault will generate error and an error is that part of the system that is liable to lead to subsequent failure. A failure of the system occurs when the behavior of the system deviates from that required by its specifications. In (Avizienis et al.), fault is the adjudged or hypothesized cause of an error (a fault is active when it produces an error, otherwise it is dormant), an error is that part of the system state that may cause a subsequent failure (a failure occurs when an error reaches the service interface and alters the service). A system failure is an event that occurs when the delivered service deviates from the correct service. Figure 2.1 shows this relation in a diagram.

![Figure 2.1 the fundamental chain of threats to dependability (Avizienis et al.)](image)

Though a fault has the potential for generating errors, it may not generate any error during the period of observation. In other words, the presence of fault does not ensure that an error will occur. The reverse; however is not true (Jalote, 1994). When there is an error, fault(s) must have occurred prior to error. For the source of the failure comes from fault, one should take care this fault.
2.1.2 Role of Fault Tolerance

Fault tolerance comes from the fact that fault prevention is not enough. The fault prevention approach is achieved by eliminating as many faults as possible before the system is put in regular use (Jalote, 1994). However, it is very difficult to guarantee that fault prevention has eliminated all faults in a system. This is the reason why fault tolerance is important. Therefore, fault prevention will do its part during design phase, while fault tolerance will do its part in the operational phase.

Fault tolerance will not eliminate faults in a system, but will handle it in order to avoid generating error. Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults (Avizienis et al.) or to avoid system failure, even if faults are present (Jalote, 1994).

Systems with a fault tolerant facility will have redundant components inside the system, because the redundancy is the key to supporting fault tolerance. The redundant component will not play its role when there is no fault. Unfortunately, a system, as a whole, cannot be made fault tolerant against its own failure (Jalote, 1994). A system can be made fault tolerant to other system’s failure. This is the reason why in a fault tolerant system, there will be redundant components. Therefore, these redundant components are such a small sub-system against other sub-system’s failure. There are two types of redundancy, i.e. hardware redundancy and software redundancy. All are used in the implementation of fault tolerance in a distributed system (Jalote, 1994).

2.1.3 Fault tolerance Method

Nowadays, no general technique can be proposed to add fault tolerance in a system. It depends on the requirements of the application (Torres-Pomales, 2000). Fault tolerance can be implemented using two approaches, i.e. hardware and software fault tolerance.

The **Hardware fault tolerance** relates to an electrical characteristic of the system, e.g. voltage level, sink-source current. The component factories, especially for semiconductor components, implement this kind of fault tolerance. Once a chip is on the market, there is no possibility to repair it if there is an error. That is why, the hardware fault tolerance is also important. However, the hardware fault tolerance is not the focus of this project.

The **Software fault tolerance** deals with software design. A programmer will add some redundant parts inside the software to handle faults and prevent failures. The development of object-oriented programming is very helpful in this technique.

In providing a fault tolerance, four phases can be identified: error detection, damage confinement, error recovery, and fault treatment and continued system service (Jalote, 1994). The error detection is important since this indicates the presence of fault (and failure). Any damage caused by that error will be identified by in damage confinement phase, and then error recovery is done. After recovering from the error, fault treatment will localize the erroneous component from the operation and continue system service.

Generally, software fault tolerance can be divided into two groups: single version and multi-version software technique (Torres-Pomales, 2000). Single version technique achieves that goal
with a single piece of software by adding a mechanism into the design to detect and handle the errors. For multi-version technique, fault tolerance uses several different versions (or variants) of the software with an assumption that error from different version of software will be different as well. For detail discussion about software fault tolerance, see (Torres-Pomales, 2000).

2.2 Distributed Systems

(Jalote, 1994) categorized two views on a distributed system namely as a physical model or a logical model. The former is defined by the physical component of the system, but the later is defined from processing or computation point of view. Both are important.

2.2.1 Physical Model

In the physical model (or application model), a system consists of several computers (called nodes) in different places. They are connected through some communication network.

The way in which the different links are connected to different nodes is called a network topology. There are many network topologies, e.g. fully connected, star, tree, bus. The most popular one is the bus topology. For the bus topology, there are many protocols available now, e.g. the CAN bus, the I²C, etc.

![Figure 2.2 Physical model of the system](image)
2.2.2 Logical Model

From the application point of view, distributed processes can be implemented in a single computer, thus without a communication network. This model consists of concurrently executing processes that co-operate with each other to perform some task.

At a logical model (or an architecture model), a distributed system considers as consisting of finite set of processes and channels between the processes. Channels represent the logical connection between the processes. Two processes will communicate to each other through this channel. There are two kinds of channels, i.e. synchronous and asynchronous channels. In the synchronous channel, if one process sends something to other process, it must wait until the receiver acknowledges receipt of that data. In addition, a receiver must wait until the transmitter sends something. The Asynchronous channel omits this kind of synchronization, because the channel is buffered.

![Logical model of the simple user application based on process/channel paradigm](image)

Figure 2.3 Logical model of the simple user application based on process/channel paradigm

2.3 ROPES development cycle

To develop a system, one has to do some iterative process. In case of an Embedded Systems, this can be ROPES (Rapid Object-oriented Process of Embedded System) (Douglass, 2003). This approach is used in many real-time and embedded development environments. This iterative process is from simple to complex, from general to specific.

![ROPES microcycle](image)

Figure 2.4 ROPES microcycle (Douglass, 2003)
Figure 2.4 shows this iterative approach. Each cycle starts with analysis followed subsequently by design, implementation and testing. The result of one cycle is a prototype.

2.4 The CT Library

The CT library consists of hardware dependent and hardware independent parts. A user needs to write his own hardware dependent parts when these parts are not available yet, usually when a user uses a new processor. The hardware dependent parts include context switch (it depends on the processor architecture) and link driver (it depends on the peripherals on the processor).

2.4.1 Context Switch

Running parallel programs in a single ordinary processor is tricky. It must use some mechanism to share the CPU execution time among parallel software units of execution. Indeed, the execution inside the processor is sequential, but from outside point of view, all processes look like they run in parallel. The problem is how to move from one process to another. The answer lies in the context switch. That is why this part is very important in the CT library.

All parallel processes will be considered as threads. All threads will wait in the ready queue before running. Each thread has a priority and its own stack. It is the dispatcher’s responsibility to manage those threads: to decide which thread from ready queue will run next, and to perform a context switch from currently executing thread to the next one.

The dispatcher will compare the priority of the current thread to that of the thread from the top of the sorted ready queue. If the priority of current thread is higher than or the same as that of in the ready queue, there is no context switching. Otherwise, context switching will occur.

If there is a context switch, all necessary information on the current thread is saved to a stack and all necessary information of the next thread is loaded. Part of the code that deals with a stack and a memory is most often hardware specific. This is the reason why the user needs to know how to implement it in a specific processor.

2.4.2 Link Driver

There are two types of channels in the CT library: channels with and channels without a link driver. A channel without a link driver is a simple entity that connects processes internally in a single processor. This type of channel can be viewed as a channel with built-in local memory link driver handling either a rendezvous or buffered communication. On the other hand, a channel with a link driver is a special channel, because processes use hardware dependent part of the processor, e.g. ADC, RS-232, etc. Therefore, channels in a single processor share memory and execution time; while in distributed system, channels share a peripheral driver (Hilderink et al., 2000).

Rendezvous synchronization implies that two processes are allowed to engage in communication only after both of them become ready. Thus, the first process that reach channel communication point will have to wait until the second process reaches the point of communication. In the implementation of distributed rendezvous channel, a process, which sends data to another process will be blocked until it receives acknowledge from the receiver. After receiving this
acknowledge, that process is released. For buffered channel, a process does not wait for acknowledge because the data is received in buffers. When the receiver is ready, the data will be read.

Link drivers deal with hardware dependent parts of a program code. This hardware can be a communication link (such as RS-232, SPORT, SPI, CAN bus, USB, Firewire, etc.) or a peripheral device (such as Timer, PWM, ADC, DAC, etc.).

A process connected to a channel does not need to know whether that channel has a link driver or not. If that channel uses a link driver, then the read and write methods of that channel will be delegated to those of that link driver.

2.5 The CAN bus

The CAN bus was developed by Robert Bosch GmBH in 1980s. Originally, the CAN bus was developed for automobile industry, but now the application become wider and wider in industries.

The CAN bus is classified as CDMA/CD (Carrier Detect Multiple Access with Collision Detection). It means, whenever one node wants to use the bus, it needs to detect first if the bus is occupied or not. The detection procedure uses an arbitration algorithm. This will be explained later. More details are presented in (Robert Bosch GmBH, 1991).

The CAN bus uses content based addressing instead of location based addressing. Thus in CAN, message ID (identifier) does not contain address of receiver node, but every node knows IDs of messages it wants to receive. Therefore, several nodes can receive data with the same ID. This will make the system simpler since if several nodes need data from the same type of sensor, the system does not need to install one sensor for each node. Beside, this makes system more flexible. To add a receiver-type node, the system does not need to be re-programmed anymore. The node transmitting the lowest number of ID (this means that node has highest priority) will get the bus. The other nodes wait until the bus is free again (bus idle state).

There are two CAN bus frame formats, namely 2.0 A and 2.0 B. The difference between them is in length of their ID. The former uses an 11-bit ID, while the later uses a 29-bit ID. The CAN bus version 2.0B is extended mode of the 2.0A. Therefore, a node with 2.0B, can detect IDs from 2.0A, but not vice versa. For this reason, there are two version of the 2.0 B format, i.e. a passive and an active one. The passive means the node only acts as a receiver, while the active acts as a transceiver. Figure 2.5a shows the CAN frame format (this is CAN 2.0A). CAN 2.0B is different from 2.0A, figure 2.5b and 2.5c show this difference.

![Figure 2.5a the standard CAN frame format (CAN in Automation, 2003)](image)

In the CAN bus the digital levels are represented using ‘dominant’ and ‘recessive’, usually implemented as ‘0’ and ‘1’ respectively. To control if the bus is free or not, nodes have to do an
arbitration process. The arbitration begins with the start of a frame. A node can send *start of frame* signal (SOF on Figures 2.5a-c) only when the bus is idle, otherwise, that node must wait. Table 2.1 provides explanation of several abbreviation used in figure 2.5.

Transmitting of SOF is followed by a message ID. While transmitting their message ID, nodes have to monitor the bus level also. If the bus level and their transmitted level are different, then those nodes have lost arbitration and should stop transmitting their ID. This is the arbitration process (see figure 2.6). This operation enables the CAN bus to act as a multi-master bus that resolves bus access conflicts in a deterministic way.

**Standard Format**

![Standard Format Diagram](image)

Figure 2.5b CAN frame format for 2.0A (Robert Bosch GmBH, 1991)

**Extended Format**

![Extended Format Diagram](image)

Figure 2.5c CAN frame format for 2.0B (Robert Bosch GmBH, 1991)

**Table 2.1 List of terminologies in figure 2.5**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOF</td>
<td>Start of Frame</td>
</tr>
<tr>
<td>RTR</td>
<td>Remote Transmit Request</td>
</tr>
<tr>
<td>IDE</td>
<td>IDentifier Extension</td>
</tr>
<tr>
<td>SRR</td>
<td>Substitute Remote Request (extended ID only)</td>
</tr>
<tr>
<td>DLC</td>
<td>Data Length Code, from 0 to 8</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundant Check</td>
</tr>
<tr>
<td>ACK</td>
<td>ACKnowledge</td>
</tr>
<tr>
<td>EOF</td>
<td>End of Frame</td>
</tr>
<tr>
<td>IFS</td>
<td>Inter Frame Space</td>
</tr>
<tr>
<td>r0 and r1</td>
<td>Reserved bit, always sent as dominant</td>
</tr>
</tbody>
</table>

To achieve design transparency and implementation flexibility, the CAN bus protocol has been subdivided into different layers:

- Physical layer
This layer deals with an electrical characteristic of signal on the transmission media, and the transmission media itself. The transmission media can be a single wire, a twisted pair (shielded or unshielded), a fiber optic, etc.

- **Transfer layer**
  The transfer layer consists of kernel of the CAN, a transfer protocol, bit timing procedure, control over CAN.

- **Object layer**
  Tasks of this layer are message filtering, message and status handling. The message filtering is important to detect if the ID is important to that node or not, while message and status handling interpret a message and a status of the bus.

![Figure 2.6 Arbitration process (Kvaser)](image)

The CAN controller handles all CAN operation such as arbitration, error checking, framing, acknowledge signaling, error handling and fault confinement. The following text will explain each of these operations.

The five frames are data frame, remote frame, error frame, overload frame and interframe space. Since there are five types of frames in the CAN bus, a controller should choose which frame is appropriate for a given task. 1 bit (RTR) is used to distinguish data from a remote frame. If one node is busy and cannot receive more data, then an overload frame is sent. Inter Frame Space (IFS) is space between two frames.

There is an acknowledgement in the bus if a receiver receives messages. This acknowledgement will tell the transmitter that at least one node received its message. If other nodes encounter error, those nodes will send an error frame and the transmitter will send that message again.
There are several error types in CAN, they are bit error, stuff error, CRC (Cyclic Redundancy Check) error and acknowledge error. The bit error actually happens on the arbitration process. The CAN uses bit stuffing in sending data. It means for five consecutive bits with the same value, one opposite bit will be inserted (see figure 2.7). Stuff error takes care of an error when this rule is violated. The CRC does the error checking. In this operation, a receiver node will calculate the CRC of all bits then compare it to CRC calculation from the transmitter. If they are different, that receiver will send an error frame to ask for retransmission.

To guarantee the CAN bus operation, a fault confinement algorithm is made. These algorithms deal with a transmitter and a receiver error counter. The value of this counter influences the behavior of that node. If the value is between 0 and 127, a node is in active error mode. For 128 to 255, a node is in passive error mode. When the value exceeds 255, a node is in bus off mode. (Robert Bosch GmBH, 1991) explain this in detail.

![Figure 2.7 Bit stuffing in CAN bus (Kvaser)](image)

### 2.6 The SPORT (Serial Port)

The SPORT stands for Serial Port. It is a general-purpose serial point-to-point link to transfer data from one processor to another. The heart of SPORT is the clock. The clock will control the data transfer between two nodes. It means the speed can be varied during sending the data. (Analog Device Inc., 2002a).

The SPORT is a full duplex communication link. It can transfer data simultaneously in both directions. It means two nodes can exchange data at the same time. For the operation, it consists of two sets of terminals, one for transmitting and the other for receiving. There are data, clock and frame synchronization. To use the SPORT, all terminals from the transmitter are connected to the corresponding terminals of the receiver.
The SPORT operation enables data framing or not. This means that user can use a frame to indicate a group of data. SPORT can be used to emulate the serial RS-232 protocol. The ADSP-21992 EZ-KIT LITE boards even have RS-232 connector on board.

2.7 ADSP-21992

The ADSP-21992 (Analog Device Inc., 2002a) is a 16 bits fixed-point DSP processor from Analog Devices. This chip is developed from the ADSP-21990. The main difference is the addition of the CAN controller. The ADSP-21992 is a complete chip not only for DSP operation but also for a general-purpose controller. This chip is also equipped with DMA (direct memory access) controller. There are three buses inside this chip; they are PM (Program Memory) bus, DM (Data Memory) bus and DMA bus. Each bus consists of separate address and data bus.

Several peripherals are on-chip, namely AD Converter, DA Converter, Auxiliary PWM output, Timer, SPI port, SPORT, Digital I/O flags, Encoder Interface Unit and CAN interface. During this project, link drivers have been built for most of those peripherals.
The ADSP-21992 comes with a development kit board called the ADSP-21992 EZ-KIT LITE. Figure 2.8 shows this development board. In normal operation, this evaluation board is driven by 16MHz for its operation clock (CCLK – Core CLK), derived from 32MHz clock generator. The peripheral clock (HCLK – Peripheral CLK) is always half of the operation clock. This information is very important since all timing for peripherals will be derived from HCLK. With software manipulation, the user can drive the ADSP-21992 chip on this board up to 160MHz as CCLK. Appendix A, section A.1.1 explains how to do this. There are two connectors for the CAN bus and a jumper to enable 120Ω resistor to terminate the network. With these two connectors, a user can daisy chain the board. The SPORT connectors will be used to connect this board to another board with SPORT facility.

2.7.1 CAN in the ADSP-21992 EZ-KIT LITE

The CAN controller in the ADSP-21992 organizes CAN buffer as 16 different mailboxes. A mailbox contains data buffer, the number of transmitted or received byte, message IDs and message filters. A user can configure each mailbox either as a transmitter or as a receiver.

Configuring the CAN controller means to set up the CAN speed, to configure direction of the mailboxes to transmit or to receive data and to set a value of filter mask for the receiver mailboxes. The maximum speed of the CAN bus is 1Mbps, but Analog Devices recommends using speed lower than 1Mbps (Analog Device Inc., 2002a).

Appendix A, section A.2 gives more detail on the CAN module in the ADSP-21992 and its initialization. After all initializations, one needs to know the performance of CAN module in the ADSP-21992, and appendix B showed the result of experiments designed to benchmark its performance.

The ADSP-21992 provides three interrupts for CAN, i.e. a transmitter mailbox interrupt, a receiver mailbox interrupt and a CAN global interrupt. They are peripheral interrupts mapped to user’s interrupt. Every successful transmitting message from the transmitter mailbox is followed by invoking a mailbox transmitter interrupt. When a receiver mailbox receives message well, the CAN controller will generate a mailbox receiver interrupt. The Global interrupt handles other events important for CAN operation. Appendix A, section A.2 gives some errata for chapter 13 of (Analog Device Inc., 2002a).

For the transmitter mailbox, a user should write all necessary data first before enabling it. Data here refer to message ID for CAN, the size of message and the data itself. When all data is ready, the transmitter can send it after enabling that mailbox. For every successful transmission, there will be a CAN transmitter interrupt (if this interrupt is enabled). This interrupt is only information that the message is transmitted well. This is not an indicator that a receiver mailbox has received that message. That signal means that the data arrived safely on the receiver’s side. This paradigm is very important. Therefore, a user cannot use this interrupt as an indicator that the target mailbox received that data well. A user can choose to overwrite the unread data or to protect those unread data. If choice is made to protect unread data, the user will lose the incoming data although the receiver’s side already received that data. The experiment presented in the Appendix B, on section B.2 illustrates this phenomenon.

The best way in using receiver mailboxes is to use a receiver mailbox interrupt request to notify the user that there is a new unread message in the receiver mailbox. In that mailbox, a user will
find data length code, the data bytes and the message ID. Inside this \textit{Interrupt Service Routine} (ISR), the interrupt signal should be cleared.

\textbf{2.7.2 SPORT in the ADSP-21992 EZ-KIT LITE}

The SPORT in the ADSP-21992 is clocked by either HCLK or an external source. The user can select this source. These are the features of SPORT in ADSP-21992 (Analog Device Inc., 2002a):

- Provides independent transmit and receive functions
- Transfers serial data words from three to sixteen bits in length, either MSB-first or LSB-first
- Double-buffers data (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT
- Internally generates serial clock and frame sync signals in a wide range of frequencies or accepts clock and frame sync input from an external source
- Performs interrupt-driven, single-word transfers to and from on-chip memory under DSP core control
- Provides Direct Memory Access transfer to and from memory under I/O processor control. DMA can be autobuffer-based (a repeated, identical range of transfers) or descriptor-based (individual or repeated ranges of transfers with differing DMA parameters).
- Executes DMA transfers to and from on-chip memory—the SPORT can automatically receive and transmit an entire block of data
- Permits chaining of DMA operations for multiple data blocks
- Has a multichannel mode for TDM interfaces—the SPORT can receive and transmit data selectively from channels of a time-division-multiplexed serial bitstream multiplexed into up to 128 channels—this mode can be useful as a network communication scheme for multiple processors
- Can operate with or without frame synchronization signals for each data word; with internally-generated or externally-generated frame signals; with active high or active low frame signals; and with either of two configurable pulse widths and frame signal timing

The most important thing using SPORT is the interrupt behavior. The transmitter interrupt is generated whenever a transmit buffer is empty. It means SPORT is ready for the next data. To fill this buffer means clear the interrupt signal. The receiver interrupt will be generated if there is new unread data in its buffer. Reading the data will automatically clear the interrupt. If the user does not need a transmitter interrupt, this function should be disabled. The Interrupt Mask register of the ADSP-21992 control this functionality (see chapter 13 of (Analog Device Inc., 2002a)). If transmit interrupt is not masked this will disturb the whole SPORT operation, because the transmit interrupt is generated whenever the transmitter buffer is empty. The user should fill this buffer with data – this means to send that data – in order to clear the interrupt. If there is no data to be sent, this interrupt will not be cleared and the program will stuck on that interrupt.
To do some basic configuration for the SPORT means to set the speed of the clock (derived from HCLK), to select between using external or internal clock source, to select using frame synchronization or not, and to set the number of bit sent via the SPORT. Chapter 8 of (Analog Device Inc., 2002a) gives detailed explanation about the SPORT in the ADSP-21992. (Analog Device Inc., 2002a) also writes that the maximum data transfer rate of the SPORT is half of the HCLK. For initialization procedure, see Appendix A, section A.3.

### 2.8 Hardware of the plant

A plant is used to demonstrate whether fault tolerance is working or not. The plant used is the LINIX. The LINIX is a DC motor with an inertia as load, a flexible transmission belt and a rotary encoder. The motor and the inertia load are connected with a belt. Since the focus of this project is on implementing fault tolerance mechanism in scope of the CT library communication subsystem, and not on the control design, a simple PID control law was used. The control loop is implemented as distributed using the CAN bus or alternatively, the SPORT link. After unplugging the CAN connection, the system should continue to work using the alternative SPORT connection. After the CAN is plugged-in again, this should be detected by software and the CAN should be automatically put back in a service. If now we disconnect the SPORT, the plant should again continue to work as though nothing is going on.

The system uses one-way configuration. On this configuration, there is only one data communication, from board 1 to board 2. Board 1 reads the reference signal and actual speed of the motor, calculates the difference between these signals and sends that to board 2. Board 2 calculates the PID control action and steers the motor.

![Figure 2.9 Photograph of Linix plant](image)
Fault-tolerance in real time distributed system using The CT Library
3.1 Context Switch

There are two approaches to implement a context switch, i.e. using an assembly language or using setjmp() and longjmp() C functions. Both have their own advantages and disadvantages.

Implementation using an assembly language is difficult, especially if a user is not familiar with the processor’s architecture. The main reason is that a user must decide on set of registers that need to be saved into a stack. This approach allows a context switch to happen during ISR execution.

On the other hand, the implementation using two functions setjmp() and longjmp() is easier than using assembly, because the compiler will translate those functions into assembly code and a user does not need to decide which registers should be saved into the stack. The compiler generates assembly code from C/C++ code in such a way that the set of the registers used in one C instruction is much larger than set of registers common to neighboring C instructions. Therefore, if the context switch can only happen between C instructions, only a small subset of registers is kept on stack. This idea is exploited by using setjmp()/longjmp() to implement a fast and absolutely portable context switches mechanism. However, this solution does not allow context switching to happen during the ISR, because interrupts can happen between any two-assembly instructions.

This project uses the second approach, i.e. using setjmp() and longjmp() functions. The reason is implementing a context switch as soon as possible is more needed than the way it handles the ISR. These two functions are already implemented in most of the C libraries (also in the C library for the ADSP-219x), therefore, a user will not really face incompatibilities if he changes the processor. Thus, the only instructions added in assembly are concerned with setting the initial stack pointer value for newly created process.

A user needs to know which register is used as a stack pointer register on that processor. For ADSP-21992, this register is the index register I4 (it will serve as stack pointer).

As the program is written in C, it uses inline assembly code, with asm() function. (Analog Device Inc., 2002c) explains how to use this function to write inline assembly code inside C code.
3.1.1 setjmp( ) function

Here is the description of the setjmp() function according to (Analog Device Inc., 2002c). The purpose of this function is saving all necessary parameters to a stack. This function saves the calling information in the jmp_buf argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to longjmp.

When setjmp is called, it immediately returns with a result of zero to indicate that the environment has been saved in the jmp_buf argument. If, at some later point, longjmp is called with the same jmp_buf argument, longjmp will restore the environment from the argument. The execution will then resume at the statement immediately following the corresponding call to setjmp. The effect is as if the call to setjmp has returned for a second time but this time the function will return a non-zero result.

The effect of calling longjmp will be undefined if the function that called setjmp has returned in the interim.

3.1.2 longjmp( ) function

Here is the description of the longjmp() function according to (Analog Device Inc., 2002c). The longjmp function causes the program to execute a second return from the place where setjmp(env) was called (with the same jmp_buf argument).

The longjmp function takes as its arguments a jump buffer that contains the context at the time of the original setjmp. It also takes an integer, return_val, which setjmp returns if return_val is non-zero. Otherwise, setjmp returns a one.

If env was not initialized through a previous call to setjmp or the function that called setjmp has since then returned, the behavior is undefined. In addition, automatic variables that are local to the original function calling setjmp, that do not have volatile-qualified type, and that have changed their value prior to the longjmp call, have indeterminate value.

Figure 3.1 Timing diagram of context switch mechanism

Figure 3.1 shows the timing diagram of context switch mechanism. The grey box is only done first time new thread is scheduled.
3.2 Embedded system requirements

Since resources in an embedded system are limited, the ISR should be able to direct itself efficiently to the appropriate location without wasting time. Several CAN mailboxes will be used. They will serve as a transmitter, a receiver, an acknowledge transmitter and an acknowledge receiver. It also needs two additional mailboxes to detect link failure. In case of the SPORT, there will be simple protocol to distinguish data, acknowledge and link test.

The ISR should last as minimal amount of time as possible. Calling any potentially blocking synchronization primitive (like rendezvous channels communication) is not permitted from inside of the ISR.

3.3 Extension of the Link Driver concept

Hilderink proposed a link driver as a hardware dependent part of a processor (Hilderink et al., 2000). The idea behind this is to encapsulate hardware related operations into certain parts of the software. This will facilitate the use of the same program in different processors or with different peripheral hardware. What needs to be done in such a case is only changing the link drivers. No changes are needed in user’s application. Therefore, it is easy to move an established program from one processor to other processor.

Figure 3.2 shows link driver concept. It starts from a concept of a channel. All processes communicate to each other via channel only, through read and write functions. Inside each channel, there is a link driver. The link driver encapsulates hardware dependent part of code used inside a channel. Processes are independent from the hardware. Therefore, in figure 3.2 there is a dashed line to separate hardware independent part from hardware dependent part.

![Figure 3.2 Link Driver framework in CT library (Hilderink et al., 2000)](image-url)
For peripherals handling communication (CAN, SPORT) this concept has been extended, as shown on Figure 3.3 (Orlic et al., 2003). The motivation to extend Hilderink’s idea is to implement a mechanism for addressing channels in a distributed system, and to divide the part of the LinkDriver implementing higher communication OSI layers from the part implementing lower level hardware and fieldbus protocol specific parts. The RemoteLinkDriver class will be derived from the LinkDriver class, thus inheriting the interface of the Link Driver and ability to be plugged-in the channel.

![Diagram showing channel and device drivers](image)

**Figure 3.3 Modified concept of implementing link and device drivers (Orlic et al., 2003)**

Specific for communication link driver, there will be the Network Device Driver (NDD). All communication link drivers (CAN Device Driver and SPORT Device Driver on Figure 3.3) should be derived from this abstract class.

Besides, this modification aims at obtaining flexibility and fault tolerance in using the available communication peripherals. This means the communication link can be changed (for example, in case of network failure or to implement some traffic congestion control) during the operation, without shutting down the system. Therefore, if there is an alternative route for a message, switching to that route is done transparent to processes using the channel.

The RemoteLinkDriver class saves the available communication link driver inside a lookup table. This class also holds a method to switch from a failed communication link to one that is available or to switch from the currently used to recovered higher priority link.

Classes for peripherals, such as ADC, DAC, PWM, Encoder Interface Unit and flag I/O., are derived from the LinkDriver class directly.

Shortly, it can be concluded that for a communication peripheral, instead of the LinkDriver class handling hardware peripheral directly, there is one object of the RemoteLinkDriver type, per channel, handling high-level aspects of communication and one Network Device Driver object handling hardware dependent parts of communication per each network interface on a given

22
board. In this project, two different types of Network Device Drivers are implemented: one dedicated to managing the CAN fieldbus and the other controlling the SPORT. Requests of Producer/Consumer process to write/read on a channel will be delegated to the Remote Link Driver object, which will divide/assembly messages into/from packets and use the services of one of available NDDs to transmit/receive packets. A channel will never know which NDD is used. For the other peripheral device, its link driver will be called from the LinkDriver class. It is strongly advised that users never accesses the hardware dependent part outside the LinkDriver class. Figure 3.4 shows the class structure implemented in this project.

![Diagram showing class structures of link drivers in this project]

Figure 3.4 Class structures of link drivers in this project
The address of each channel in the system is composed of two parts: a node identifier (nodeID) and channel identifier (channelID). A node identifier is used to route message to a destination node. Channel ID is a unique identifier of a channel in scope of its node. Together node ID and channel ID make a unique address. Such an address is independent of the communication link used. However, the way in which this address is actually used is specific to the fieldbus protocol used.

The SPORT is a point-to-point link and therefore address of the destination node is fixed in wires and need not be sent. Address of the channel has to be sent if multiple channels are intended to be used.

In the CAN bus, addressing is specific in a sense that it is content based. All mailboxes from all nodes in the system filter the identification bits (messageID) of every message on bus. A message will enter a mailbox if a messageID match the filter and there is free place in mailbox. Clearly, in case of the point-to-point channels, the most efficient solution is to incorporate the destination nodeID as part of messageID.

### 3.4 Description of complete transmit-receive process

This section explains the detail description of transmit-receive process between one producer (P) and one consumer (C) on a different processor. Figure 3.5 shows the sequence diagram of this process.

![Sequence diagram of transmit and receive process in the producer and consumer](image)

Figure 3.5 Sequence diagram of transmit and receive process in the producer and consumer
Other necessary information specific to this test case is that the default link is CAN bus and SPORT point-to-point link is used after detection of CAN bus failure. First, it is assumed that there is no failed link. P wants to write data to its channel. Because a link driver object is plugged-in to that channel, the execution of its operation is delegated to the Remote Link Driver’s `write()` operation. Inside this, the transmit method of default NDD (CAN Device Driver in this case) will be called to send this data packet by packet. Here, the data will be transmitted and the P is blocked (because P has to wait for acknowledge) from the C. The blocking mechanism is implemented using a semaphore.

At the other side, the C will read from its channel, but this process is blocked by a semaphore. This semaphore will be released in the ISR of receive mailbox interrupt. When data arrive at the CAN mailbox, an ISR is generated. The ISR will release the semaphore and the blocking process can continue. Here, the data is read and the C will send an acknowledge signal to the P to indicate that the data is already read.

At the other board, the P is waiting for the acknowledge signal. Upon receiving this signal, the P will be released and the next loop can run again. This is the normal operation without considering a network failure.

### 3.5 Fault Tolerance in the CT library

For there are many implementations of fault tolerance in a distributed system, one needs to define what kind fault tolerance will be implemented here. This fault tolerance will be built in the CT framework.

The fault tolerance built in this project relates to the communication link. Generally, several links might be available to transport message between two nodes. In this study case, the main link is the CAN bus (this is the active link by default with the highest priority), while the alternative one is the SPORT. If something happened which makes the default link fail, the system will automatically replace it with the alternative one, without bothering a user's application with exceptions and communication details.

To detect whether there is link failure or not, one board (master) will send regular message to the other board (slave). The slave will send back this signal to the master (echo). If within pre-defined time there is no echo received, the master knows that the communication link has failed. For the slave, if after sending an echo there is no more regular message, that node knows that the communications failed. The channel will not know whether there is changing from default to alternative link or not. In case the higher communication link is available again, the system will switch back to this link.

For demonstration purpose, there will be a switch on the cable. Instead of unplugging or unplugging the cable, this switch will simulate breaking the cable.

When there is a network failure, the broken link should be replaced as soon as possible by the RemoteLinkDriver object. After replacing the failed link, the write method of the RemoteLinkDriver object should check whether the previous data is already acknowledged or not. If not, then the P should be released from blocking by signaling associated semaphore and data will be sent again.
Figure 3.6 shows the sequence diagram used in the fault tolerance implementation. It also illustrates when there is a link failure.

![Sequence diagram of fault tolerance implementation](image)

**Figure 3.6 Sequence diagram of fault tolerance implementation**

### 3.6 Hardware of the plant

A plant is used to demonstrate whether the implemented fault tolerance is working or not. The plant used is the LINIX. Several additional hardware circuits are used to enable LINIX work with the ADSP boards:

- PWM motor driver
- Potentiometer (as a reference signal)
- Power supply circuit
• 2 LED indicators are used per each board to display if CAN and SPORT links are available.

![Diagram](image)

Figure 3.7 One-way configurations

Figure 3.7 shows the block diagram of the plant while figure 3.8 shows the complete additional hardware circuit used to enable LINIX works with the ADSP-21992 board. See figure 3.7, board 1 receives data from reference (set point) and encoder (present speed) then send synchronization pulse to board 2 to steer the DC motor via PWM circuit using value from previous cycle. Board 1 calculates the difference between set point and the present speed then send the result to board 2. Board 2 upon receiving this data calculates controller action for the next cycle using PID controller.

3.6.1 Potentiometer

On figure 3.4, a Potentiometer (R6) acts as a reference signal. It will deliver a voltage from a regulated power supply, LM317, to the ADC unit of the ADSP-21992. When R6 is in maximum value, R5 should be adjusted to get 1 volt at ADC channel 1 output.

3.6.2 H-bridge

To control speed of the DC motor, one can control the voltage applied to that motor. In addition, inverting the polarity will change the direction of the motor. In order to generate a variable DC voltage from a digital system, the PWM approach is used. Usually, the current is not enough. For this purpose, an H-bridge is used. The H-bridge here is taken from the ARTY robot project (Engelen, 1999).

The H-bridge gets its name from the fact that the circuit forms the letter ‘H’. With this circuit, one can get a variable DC voltage by controlling one of the input pins with a PWM signal. The other pin is for the direction. The later pin is not used, since this project relates to speed control.

3.6.3 LED indicators of link availability

This additional circuit contains LEDs to indicate that the communication links are available. The green and yellow LED is for CAN and SPORT respectively. When the LED is on, the corresponding communication is available. Digital I/O flags of the ADSP-21992 control the LEDs.
Fault-tolerance in real time distributed system using The CT Library
Chapter 4

Implementation

4.1 Object-Oriented Programming in C

The C language does not support the Object-Oriented Programming. For this reason, Hilderink mimics the behavior of this kind of programming for C (Hilderink). It includes single inheritance, polymorphism, packages, interfacing, instance construction and instance destruction.

With this style, it is easy to manage the program. Another advantage is that a user can easily transfer such a C program to C++ and the other way around. But the program looks like more complex.

4.2 Context Switch Implementation

First, a simple context switching mechanism is made using setjmp/longjmp functionality but without the CT library. The code of this minimal kernel is in Appendix C section C.1.1. Then the context switching mechanism of the CT library is carefully examined and changes are made in several functions based on the minimal-kernel example.

In the CTC, atomic sections are implemented using two functions: Processor__enterAtomic() and Processor__exitAtomic(). Inside Processor__exitAtomic() function, context switch is performed if needed. The heart of the context switch is saving and restoring context of threads to and from the stack. The implementation of the context switch in the CT library means writing three methods:

- void Processor__startswitch(void)
- void Processor__stopswitch(void)
- void Processor__contextswitch(void)

Unfortunately, assembly language cannot be avoided because pointer to the stack of the new thread must be written into the I4 register (used as a stack pointer by the C compiler). There is no other way to access this register except using assembly language. Changes to those functions are described in Appendix C section C.1.2.
4.3 Peripheral LinkDriver Implementation

4.3.1. ADCLinkDriver

ADC stands for Analog to Digital Converter. It converts analog signal input and gives digital signal as its result. It means the only operation related to value of this part is to read it. Therefore, ADCLinkDriver will only have read operation. The write operation will throw an error exception if one calls it.

The ADSP-21992 has eight channels of 14-bit ADC. Actually, there are several modes of operation but for simplicity, only the mode (simultaneous sampling mode) related to this project will be implemented. This project also uses only one channel.

To initialize ADC in the ADSP-21992 means to choose the mode of operation and select the clock divider. These will be done by writing a value to ADCCTRL register. To know whether the conversion is finished or not, an interrupt can be used. The ADC interrupt is mapped on IPR3 register at bit 12 to 15, written in (Analog Device Inc., 2002a) as IPR3[15:12]. This interrupt can be used to synchronize ADC operations.

Here is the idea for the read() method in the ADCLinkDriver. First, user should give command to ADC to start to convert the analog signal then this process will be blocked until the ADC interrupt releases it. After releasing that signal, the read() method continues by reading the result. Listing 4.1 shows the read() method in the ADCLinkDriver.

Listing 4.1 ISR and read() method of the ADCLinkDriver

```c
void ADCLinkDriver__read(ADCLinkDriver me, Object obj,unsigned size){
    sysreg_write(sysreg_IOPG,ADC_Page);  // set to ADC page
    io_space_write(ADC_SOFTCONVST, 1);  // start conversion
    // wait for end of conversion interrupt
    Processor__enterAtomic();   // enter atomic section
    Semaphore__p(me->sem);    // wait here
    Processor__exitAtomic();   // exit atomic section
    sysreg_write(sysreg_IOPG,ADC_Page);  // set to ADC page
    *(me->buffer) = io_space_read(ADC_DATA1); // read data from ADC register buff.
    *(int*)obj = *(me->buffer);   // put data to the output
}

void ADC0_IRQ(){
    unsigned temp;
    sysreg_write(sysreg_IOPG, ADC_Page); // set to ADC page
    io_space_write(ADC_STAT,0x0100); // clear the interrupt
    Semaphore__v(semADC); // release the reader
}
```

Note that signaling a semaphore is in the CTC library implemented as non-blocking operation. Thus, it can be performed inside ISR.
4.3.2 AuxPWMLinkDriver

PWM stands for Pulse Width Modulation. The idea is to produce variable DC voltage by varying the duty cycle of the square wave signal and keep the frequency constant. Duty cycle is varied from 0-100%. Indeed, the value of this duty cycle can be read, but the main operation for PWM is to set this duty cycle. This means PWM only has write operation. Opposite to ADCLinkDriver, the read operation will throw an error exception if one calls it.

This project only uses one of two channels. Therefore, the AuxPWMLinkDriver will also deal with one channel only. There is no interrupt used for this PWM and the initialization only consists of setting the operation frequency. In this operation, there is no synchronization as well.

Here is the idea for write() method in the AuxPWMLinkDriver. After writing a value to Auxiliary PWM register, control should be given to the dispatcher to perform the context switch if needed. In the CT library this can be done by explicitly calling pair of functions used to enter and exit atomic section. Listing 4.2 shows this function.

Listing 4.2 write() method of the AuxPWMLinkDriver.

```c
void  AuxPWMLinkDriver__write(AuxPWMLinkDriver me, Object obj, unsigned size){
    unsigned temp;     // declare temp variable
    // only to change duty cycle, no lock and unlock mechanism needed
    me->buffer = obj;    // get the value
    temp = (unsigned)(*me->buffer);  // write the value to temp
    sysreg_write(sysreg_IOPG,Aux_PWM_Page); // set to AuxPWM page
    io_space_write(AUX_CHA0,temp);  // write the value to register
    Processor__enterAtomic();   // enter atomic section
    Processor__exitAtomic();   // exit atomic section, will allow
                               // context switch to happen here
}
```

4.3.3 EIULinkDriver

EIU stands for Encoder Interface Unit. This peripheral accepts pulses as input from an encoder and delivers the number of count from its counter. From this, it is clear that the EIU has a read operation only. Similar to ADCLinkDriver, the write operation will throw an error exception if one calls it.

The EIU in the ADSP-21992 is used to estimate the speed of the motor shaft. This means the number of pulses in certain time duration should be counted. To generate this constant time duration, the EIU Loop Timer will be helpful. Beside, this can also generate an interrupt. Therefore, there will be a constant interval for this interrupt and at every interrupt the user can read the number of pulses within that time duration and store it into global variable of EIULinkDriver. After reading that value, this interrupt will reset the value inside the quadrature counter, and the incremental counter will start from zero again.

The initialization for the EIU is setting the EIU Loop Timer for specific duration and configuring the operation mode. From the initial experiment, the time duration is 1ms. This is long enough for low speed rotation. For time duration less than 1ms, the encoder output gives unstable value at low speed. The EIU Loop Timer interrupt is mapped on IPR3[3:0].
EIULinkDriver uses no synchronization to read the quadrature counter because for every EIU Loop Timer, the value of quadrature counter has already been stored to a buffer and read() method can read it directly.

Here is the idea of read method in the EIULinkDriver. Since there is no synchronization, after reading quadrature counter from certain buffer, control should be given to the dispatcher to perform the context switch if needed. Listing 4.3 shows the write() function of the EIULinkDriver.

**Listing 4.3 ISR and write() method of the EIULinkDriver**

```c
void EIULinkDriver__read(EIULinkDriver me, Object obj, unsigned size){
    *(me->buffer) = encoder;  // read from buffer
    *(int*)obj = *(me->buffer);  // put the value to the output
    Processor__enterAtomic();  // enter atomic section
    Processor__exitAtomic();  // exit atomic section
}
void EIU0_Timer(){
    sysreg_write(sysreg_IOPG, EIU0_Page);  // set to EIU page
    encoder = (unsigned)io_space_read(EIU0_CNT_LO);  // store value to buffer
    io_space_write(EIU0_STAT,0x0021);   // clear the interrupt
    io_space_write(EIU0_CNT_HI,0x0000);   // reset the EIU counter high
    io_space_write(EIU0_CNT_LO,0x0000);   // reset the EIU counter low
}
```

4.4 Communication LinkDriver Implementation

The RemoteLinkDriver class will be derived from the LinkDriver class. Since in the parent class there are five undefined methods, a user should write down those methods. Unfortunately, only read, write and isExternal methods will be used.

![Class diagram of communication link driver](image)

Figure 4.1 Class diagram of communication link driver
Special for the communication peripheral, an additional class is derived from the abstract NetworkDeviceDriver class. This abstract class is used to define an interface common to all fieldbus protocols and board specific Network Device Driver objects. It has four methods inside, i.e. transmit, receive, init (for initialization) and checkLink (for fault tolerance purpose). Figure 4.1 shows class diagram for the implementation of communication link driver. Actually, this is the upper part of figure 3.3.

4.4.1 Remote Link Driver

The RemoteLinkDriver will encapsulate all general-purpose communication operations. It means if one process uses its channel to send or to receive data, write or read methods of the RemoteLinkDriver will do this.

The RemoteLinkDriver consists of three methods, i.e. read, write and isExternal method. The isExternal method only returns a TRUE value. The read and write methods will use the available link with the highest priority in the system. Those methods never access hardware or register directly but use the NDD's functionality instead.

With this approach, it is possible to switch from one communication link to the other without bothering its operation. The RemoteLinkDriver uses only communication links available inside an array of a structured data. This structure should be made such that handling link failures and handling link recovery can be done easily. The elements are sorted from the most to the least important. From this idea, one can build fault tolerance in a distributed system, i.e. the system will switch from one communication link to another in case there is a problem in that link, e.g. broken link, unplugged link, etc.

This class has a buffer to store the data and a flag, i.e. acknowledge flag. Before transmitting data, this flag will be cleared. After transmitting data, the producer will wait for the acknowledgement signal. When this signal is received, the acknowledge flag will be set.

The RemoteLinkDriver has two special methods, i.e. the handleLinkFailure and the handleLinkRecovery methods. When there is a link failure, the handleLinkFailure method will replace the broken link with the first active one. In case there is link recovery, handleLinkRecovery method will check if the recovered link has higher priority than the current. If this is the case, this method will replace the current link with the recovered one.

4.4.2 Network Device Driver

For the communication peripheral driver, the Network Device Driver (NDD) class will be made. The NDD class will be the parent class for both communication links in this project. The CAN and the SPORT specific NDDs are implemented. Inside the NDD class, there are four methods, i.e. transmit, receive, initialization and checkLink methods. The last method is invoked periodically to check if the corresponding communication is available or not.

Specific for checkLink method, it works together with timer. The implementation is based on figure 3.6.
CAN Device Driver

The ADSP-21992 EZ-KIT LITE board has 16 mailboxes for the CAN module. Each mailbox can be configured either as a transmitter or as a receiver. Mailboxes can be configured to receive a group of messages with corresponding message IDs determined by enabling the acceptance mask filter. For each mailbox, there are set of following registers:

- 2 registers for message ID
- 2 registers for acceptance mask filter
- 4 registers for data (each register holds 2 bytes of data)
- 1 register for total bytes in that mailbox
- 1 register for time stamp value

Before using the CAN module, one needs to initialize it first. The initialization process includes setting the CAN speed (maximum 1Mbps) and configuring the mailbox (either as a transmitter or as a receiver, setting the acceptance mask filter and enabling receiver mailboxes). Appendix A section A.2 gives information about how to initialize and set the CAN speed. The most important thing before doing this is to know the peripheral clock from which all timing will be derived. On the ADSP-21992 EZ-KIT LITE, this clock is HCLK and has a frequency of 8MHz by default. Usage of this frequency limits speed of CAN bus to maximum of 111.111 kbps. The code snippet of this experiment is provided in Appendix B section B.1.

In this project, one node will only have one possible fixed connection to other node. For each pair of nodes, there are six mailboxes for its operation, i.e. data transmitter, data receiver, acknowledge transmitter, acknowledge receiver, check link transmitter and check link receiver. The number of used mailboxes can be optimized and that is actually out of the scope of this project.

The message ID will be used to indicate the address of the node. The CAN ISR only uses the receiver mailbox interrupt. There will be three receiver mailboxes monitored, i.e. data, acknowledge and check link. If the interrupt for the data mailbox has occurred, this ISR will release the reader (or consumer) to read data. The interrupt on the acknowledge mailbox will release the writer (or producer). Special care is taken for check link because the behavior between master and slave are different. For the master, when this interrupt is generated, it will only set a flag to indicate that the network is available. Unfortunately, in case of the slave, not only set a flag is set, but also it should send an echo to the master.

SPORT Device Driver

The SPORT in the ADSP-21992 EZ-KIT LITE board has only one register for the transmitter and one register for the receiver. There is no message ID, the number of data, time stamp value and acceptance mask filter like in the CAN module on this chip. This is a really simple communication link.

Like all peripheral devices, the SPORT needs initialization too. The SPORT initialization means to configure the SPORT operation such as

- to choose the source of the clock is internal or external
• to set the speed of data transfer
• to set whether it will use framing or not
• to choose the number of transmitted bit for every session (user can choose 3-16 bits)

Since the SPORT has only one internal 16-bit buffer for each transmitter and receiver, it needs a certain simple protocol to handle channel addressing, distinguish data from acknowledge and check link related frames.

Three messages need to be distinguished as in the CANDeviceDriver, i.e. data, acknowledge and check link message. Therefore, for every type, a special code will be transmitted before the real data and the other party will prepare itself the next message. For data message, this code will set the flag to indicate that the next message is data. After receiving this data, the system is ready again for the next session. In case of an acknowledge, this will automatically release the producer and be ready for the next session. When the code is for checking the link, the system will automatically do the procedure to check the possible link failure.

All read from the SPORT’s buffer will be done inside the ISR. Only a receiver interrupt will be used. Reading this data from the internal buffer also clears the interrupt flag. For the transmitter the ISR is not used, the user has to mask this interrupt by clearing bit 1 of the PIMASKL register. This should be done inside the initialization. To forget to clear this bit will disturb the SPORT operation since all the time the transmit buffer is empty an interrupt will be generated. To solve this, the user has to fill it with data. It means new data is sent, and when this buffer is empty, there will be an interrupt again. That is why this interrupt should be masked.

### 4.5 Altering code to achieve fault tolerance

To achieve fault tolerance, one needs to add some additional code inside the RemoteLinkDriver, the CANDeviceDriver and the SPORTDeviceDriver. An additional method for the CANDeviceDriver and the SPORTDeviceDriver is the checkLink method. Its role is to check the availability of the corresponding communication link.

Specific for the RemoteLinkDriver class, there are the handleLinkFailure and handleLinkRecovery methods. The first method replaces the current communication link when that link is broken. The second method always guarantees that the system uses the highest priority communication link. Therefore, when the recovered link has higher priority than the current link, this method replaces the current with the recovered. For this purpose, every communication link has an index number; the smaller the index number, the higher the priority of that communication link.

To check the availability of the communication link, one board as a master sends data (or a message) periodically to a slave. Therefore, it needs a timer to generate periodic interrupt for this. After receiving this message from the master, a slave must send an echo to the master. If in pre-defined time there is no echo from the slave, the master knows that there is a link failure. For this purpose, it also needs a timer. The slave also used pre-defined time in order to detect if there is a link failure. When there is no periodic data within that period, the slave knows that there is a link failure. Therefore, a timer in the master and the slave has different behavior. For the master, this timer generates interrupt twice in one cycle, one for the pre-defined time serve
as time out and the other for sending periodic data using checkLink method. While in the slave, it only generates interrupt once serve as time out.

The implementation of fault tolerance in the CT Library is based on figure 3.6. It includes the CANDeviceDriver, the SPORTDeviceDriver and the RemoteLinkDriver classes.
Chapter 5

Experimental Results

The experiments were conducted in several steps, i.e.:

- Experiment for the CANDeviceDriver and the SPORTDeviceDriver with a simple producer-consumer run on different processors. This experiment will show whether the CANDeviceDriver and the SPORTDeviceDriver work or not.
- Experiment for the CANDeviceDriver and the SPORTDeviceDriver with one-way configuration. This experiment is little bit more complex since it involves the plant (LINIX motor) and uses a timer for sampling frequency.
- Fault tolerance experiment using one-way configuration.

All communication channels are rendezvous. This means there is synchronization between processes, which communicate to each other. For every transmitted data, the transmitter will be blocked until it receives acknowledge signal.

To run the experiment with LINIX setup in one-way configuration means that the system should have an ADCLinkDriver (a link driver for Analog to Digital Converter), an EIULinkDriver (a link driver for Encoder Interface Unit) and an AuxPWMLinkDriver (a link driver for Auxiliary Pulse Width Modulation link driver).

All experiments use normal clock operation 16MHz instead of 160MHz. The reason lies on the fact that there is no such information on (Analog Device Inc., 2002a). The way user can use a software manipulation is given by the Analog Devices later.

5.1 Experiments with simple producer-consumer

The experiments here are measuring the time consumed by writing to channel with RemoteLinkDriver. RemoteLinkDriver can use either CANDeviceDriver or SPORTDeviceDriver. Table 5.1 shows the experimental results for CANDeviceDriver with various CAN speed. Table 5.2 shows the results for SPORTDeviceDriver.

Figure 5.1 shows the timing diagram of writing to a channel either with the CANDeviceDriver or with the SPORTDeviceDriver. Here, after sending data, the producer will be blocked until it receives acknowledge from the consumer. The consumer itself will be blocked until it receives data from the producer. If one of the processes is not ready yet, the other process will wait forever until it is ready. On this experiment, there is no sampling time, which controls the behavior of the system. The software implementation of producer and consumer are hardware
Fault-tolerance in real time distributed system using The CT Library

independent. All hardware dependent code is hidden in appropriate Link Driver objects. Therefore, to port the application to another hardware, the user only changes the link driver object plugged-in the channel object.

The experiment used two communication links separately. Here, the behavior of the program is not real-time because of `printf()` command. Therefore, it cannot be used for fault tolerance demonstration.

Because, in this case two communication links are to be elaborated, the program was prepared for both but it uses only one of them. Inside the RemoteLinkDriver constructor method, the user can choose between those two possible communication links.

Table 5.1 Average time consumed by writing to channel for different CAN speed in absence of other traffic on bus

<table>
<thead>
<tr>
<th>CAN speed (kbps)</th>
<th>Time consumed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.00</td>
<td>20.4413</td>
</tr>
<tr>
<td>12.50</td>
<td>13.0220</td>
</tr>
<tr>
<td>25.00</td>
<td>6.5775</td>
</tr>
<tr>
<td>50.00</td>
<td>3.3906</td>
</tr>
<tr>
<td>80.00</td>
<td>2.1974</td>
</tr>
<tr>
<td>100.00</td>
<td>1.7986</td>
</tr>
</tbody>
</table>

Plot of Table 5.1

Figure 5.1 Plot of Table 5.1

Table 5.2 Average time consumed by writing to channel for different SPORT speed

<table>
<thead>
<tr>
<th>SPORT speed (kbps)</th>
<th>Time consumed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.00</td>
<td>18.7963</td>
</tr>
<tr>
<td>80.00</td>
<td>2.0880</td>
</tr>
<tr>
<td>800.00</td>
<td>0.4728</td>
</tr>
</tbody>
</table>
If one compares table 5.1 and table 5.2 then it is found that the time consumed by the CAN bus and the SPORT are almost the same. But for the SPORT, one can have higher speed data transfer than the CAN bus, half of the HCLK. For this experiment uses normal clock operation, this data transfer rate is 4Mbps.

![Figure 5.2 Plot of Table 5.2](image)

Listing 5.1 and 5.2 shows the run method for both producer and consumer. When the communication link between the ADSP-21992 EZ-KIT LITE boards is changed, these run methods remain the same. This means the producer and consumer do not care with the communication link driver used by their channel. This experiment also clarifies that the separation of hardware dependent part from hardware independent part is useful.

**Listing 5.1 Producer's run method**

```c
void Producer__run(Producer me)
{
    printf("Producer is running...
");
    while(me->info < 20)
    {
        me->channelout->->write(me->channelout,&(me->info));
        printf("Producer write: %i\n",me->info);
        me->info++;
    }
}
```

![Figure 5.3 Timing diagram for producer-consumer experiment](image)
Listing 5.2 Consumer’s run method

```c
void Consumer__run(Consumer me){
    printf("Consumer is running...\n");
    while(me->info < 20){
        me->channelin->read(me->channelin, &(me->info));
        printf ("Integer read: %i\n", me->info);
    }
}
```

5.2 Experiment with LINIX – one-way configuration

The goal of this experiment is to show if the CANDeviceDriver and the SPORTDeviceDriver can be used in a real mechatronics application. Linix mechatronics setup described in section 2.8, additional hardware circuits described in section 3.6 and appropriate peripheral drivers described in section 4.3 were used in scope of this experiment. Although there is no real need to use two ADSP boards to control this particular setup, experiment is performed to show limitations in case distributed approach is necessary. One-way configuration (see Figure 3.7) is used. Thus, cycle starts when first board samples data from reference (set point) and encoder (present speed) then send synchronization pulse to second board to enable steering the DC motor via PWM circuit using value from previous cycle. This synchronization pulse is necessary to avoid interference of actuating performed by second board on sensor measurements done by first board. First board calculates the difference between set point and the present speed, then send the result to second board. Second board upon receiving this data calculates controller action for the next cycle using PID control. Detailed time diagram describing this sequence of actions on both boards is shown on Figure 5.4.

From the experiment, the user can control the speed of the LINIX motor by turning the potentiometer. It means the plant is ready for a fault tolerance experiment.
As illustrated in the timing diagram on figure 5.4, the communication via CAN consumes much more time than the control action calculation. Thus, bottleneck of such a distributed system seems to be in communication. This is consequence of the fact that the CAN bus communication is not as fast as processor instructions execution. This imposes potential limitations to maximal applicable sampling frequency, causing limited suitability of such a system for fast hard-real time control loops. This hardware and software configuration was used in Embedded Systems realization project to determine maximum possible sampling frequency. If we experiment with relatively low CAN speeds, bottleneck of the system is determined by CAN communication latency. However, if we gradually increase used CAN speed, at certain point bottleneck of the system performance will be determined by the highest sampling frequency obtainable by used encoder. As explained in section 4.3.3. the way in which encoder is used enables sampling frequency of up to the 1KHz. Result of this experiment for several lower values of CAN speed is summarized in Table B.2 In Appendix B. Results should be taken skeptically, since measurements were done in absence of any other traffic over CAN bus.

5.3 Fault Tolerance experiment using one-way configuration

For there is a mechanism to check the communication periodically, it needs a timing diagram to explain what happened in the system. Figure 5.5 shows the timing diagram for link checking.

![Timing Diagram for Link Checking](image)

The master sends regular data every cycle (full line arrow) and then waits for an echo until the timer is time out (dashed arrow). After receiving regular data from the master, the slave will set a certain flag to indicate that the corresponding link is available, and send an echo to the master. When the master receives an echo from the slave, a certain flag will be set and the master waits for the next cycle to send again regular data to the slave. This cycle is repeated on and on.
For this purpose, a master needs two timers, one to control the cycle and the other for the time out. But the number of timers available in a system is always limited. Therefore, only one timer is used for both purposes.

The first experiment was using simple producer-consumer processes. These processes were chosen since there is no influence from sampling time as in the one-way configuration. From the experiment, the system can replace the broken link with the available one. However, this application is not visible for user. Therefore, it needs to apply this experiment using the one-way configuration based on Linix setup.

The length of the cycle time and the time-out are different for different CAN speeds. The slower the CAN speed the larger the minimal possible and used values for cycle time and time out. The reason is that the CAN bus is the primary and default communication link while the SPORT is only secondary link. Therefore, in the SPORT, there is only one data transfer speed used and this is even higher than the CAN speed. Table 5.3 shows the values both for cycle time and for time out.

To get these values is a trade off between how fast the system will response when there is a problem with the current link and overload of the network due to this mechanism. Here the values for the time out column are chosen three times of those from table 5.1. For the cycle time column, they are four times those from table 5.1. With this value, it is assumed that the mechanism to check the link is done not too often and mainly the processes use the communication link.

<table>
<thead>
<tr>
<th>CAN speed (kbps)</th>
<th>Time out (ms)</th>
<th>Cycle time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.00</td>
<td>61.3239</td>
<td>81.7652</td>
</tr>
<tr>
<td>12.50</td>
<td>39.0660</td>
<td>52.0880</td>
</tr>
<tr>
<td>25.00</td>
<td>19.7325</td>
<td>26.3100</td>
</tr>
<tr>
<td>50.00</td>
<td>10.1718</td>
<td>13.5624</td>
</tr>
<tr>
<td>80.00</td>
<td>6.5922</td>
<td>8.7896</td>
</tr>
<tr>
<td>100.00</td>
<td>5.3958</td>
<td>7.1944</td>
</tr>
</tbody>
</table>

It is not useful to measure the time between link failure and the time RemoteLinkDriver finishes replacing it with the available one. When the link is broken, the flag, which indicates that the link is available, is never set inside the receive ISR of the CANDeviceDriver or the SPORTDeviceDriver (see grey boxes in figure 5.5). This condition with broken link stays until the timer in the master is time out (or the timer in the slave generates the next cycle interrupt). Inside this timer ISR, there is an action in case of the current link is broken. The action is simply to replace it with the available one. Therefore, replacing the broken link is not done immediately, but it has to wait until the timer is time out. Thus, maximal time between link failure and its substitution is determined by value used as a period of timer ISR.

The order in which potential Network Device Drivers are kept in the table of Remote Link Driver implicitly defines the priority of links. The primary link is the one with the highest priority. It means when the current link has lower priority and link with higher priority is available, the system will replace the current link with higher priority link. In this project, there
are only two links. The CAN bus has higher priority than the SPORT. Therefore, when the RemoteLinkDriver is using the SPORTDeviceDriver and the CAN bus is available again, the RemoteLinkDriver will replace the SPORTDeviceDriver with the CANDeviceDriver. With this requirement, the system should always check the link even when that link is broken. The ISR of the CANDeviceDriver and the SPORTDeviceDriver will take care of this. Every time the instruction pointer reaches this area, program always tests if the current link has lower priority than the recovered link or not.

The experiments show that the system can replace the broken link with the available link. When higher priority link is recovered, the system will replace the current link.
Chapter 6

Conclusions and Recommendations

6.1 Conclusions

Conclusions drawn from this project are:

- The CT Library can be used to build a fault tolerant system that will automatically switch from the broken communication link to the available one. It can also replace the current link with the recovered link of higher priority.
- To build a fault tolerant system, the user needs to add an additional part inside the program. If the system runs well, that part will be redundant, but since there is no guarantee that there is no fault in the system, this part should stay.
- Encapsulating the hardware dependent part in link driver, as proposed by Hilderink is useful in dealing with different microprocessors. The user can use the exactly the same program and only change the hardware dependent part to fit to the new microprocessor.

6.2 Recommendations

This project also recommends several points for a next possible project:

- The experiments done in this project only deals with simple data. This is typical for the communication in hard-real time control loop. However, if more complex data structures are transferred over CAN, one needs special protocols to handle disassembly/assembly messages to/from packets for this kind of data. This protocol should include retransmission of lost data and should be implemented in RemoteLinkDriver or on top of it.
- Number of mailboxes used can be minimized
- If more than two nodes are involved, then the fault tolerance mechanism is a little bit different. The concept about master and slave remains the same. To avoid overloading the network, only one slave should send echo in each link checking cycle. This is enough to determine network failure. But if we aim to detect node failures as well, the order in which slaves will send send echo can be imposed, etc. by making virtual ring.
- Make link drivers for peripherals of the ADSP-21992 which are not used in scope of this project.
- If for some purpose the user needs higher clock, then a software manipulation can be done to raise the CCLK up to 160MHz.
Appendix A

Description and specific properties of CAN and SPORT in the ADSP-21992

A.1 General Information

Figure A.1 shows the lay out of the ADSP-21992 EZ-KIT LITE board. It shows board connectors for ADC, DAC, Encoder, PWM, Digital I/O flags and external memory. Table A.1 gives information how to prepare the ADSP-21992 EZ-KIT LITE for all experiments in this project. (Analog Device Inc., 2002b) gives description for each jumper.

Figure A.1 Lay out of the ADSP-21992 EZ-KIT LITE board (Analog Device Inc., 2002b)
Table A.1 Preparation for the ADSP-21992 EZ-KIT LITE board

<table>
<thead>
<tr>
<th>Jumper</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP1</td>
<td>Closed</td>
</tr>
<tr>
<td>JP2</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>JP3</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>JP4</td>
<td>Open</td>
</tr>
<tr>
<td>JP5, JP6, JP7, JP8</td>
<td>2-3 closed</td>
</tr>
<tr>
<td>JP9, JP10, JP11</td>
<td>Open</td>
</tr>
<tr>
<td>JP12</td>
<td>Closed</td>
</tr>
<tr>
<td>JP13, JP 14</td>
<td>Open</td>
</tr>
<tr>
<td>JP15, JP16</td>
<td>Closed</td>
</tr>
<tr>
<td>JP17, JP18, JP19, JP20</td>
<td>Open</td>
</tr>
<tr>
<td>JP21</td>
<td>Open</td>
</tr>
<tr>
<td>JP22</td>
<td>1-2 closed</td>
</tr>
<tr>
<td>JP23</td>
<td>Closed</td>
</tr>
<tr>
<td>JP24</td>
<td>Closed</td>
</tr>
<tr>
<td>JP25, JP26, JP27</td>
<td>Open</td>
</tr>
<tr>
<td>JP28, JP29, JP30</td>
<td>Open, they are used directly as input for Encoder Interface Unit</td>
</tr>
<tr>
<td>JP31</td>
<td>Open</td>
</tr>
</tbody>
</table>

A.1.1 Rules of thumb in using Visual DSP++ environment and EZ-Kit Light boards

By default, the ADSP-21992 EZ-KIT LITE board is driven with 16MHz clock, though (Analog Device Inc., 2002a) writes that this chip can accept clock up to 160MHz. For several applications this is enough. If the user wants to use higher clock, then there is extra code for this. It uses frequency multiplication. First, the user writes a routine, written on the separate file, as shown on Listing A.1, and then called that routine from the main program at the beginning. Listing A.2 shows how to call the routine.

Listing A.1 Routine for software manipulation to raise the CCLK up to 160MHz.

```c
#define Wait_for_Lock_Count 0x200;
.section/pm program;
.global SetPLL;
SetPLL:
    iopg = Clock_and_System_Control_Page;
    ar = io.PLLCTL;
    ar = setbit 8 of ar;
    io.PLLCTL = ar;   // temporarily set PLL in Bypass
    ar = setbit 8 of ax0;
    io.PLLCTL = ar;   // set PLL configuration (in Bypass)
    ay0=Wait_for_Lock_Count;
Wait_for_PLL_lock:
```

48
Listing A.2 Snippet of code to configure the CCLK manipulation from main program.

```c
... #include <adsp-21990.h>
... void main() {
...    asm("ax0 = 0x0A50;" "call SetPLL;" );
...}
```

Once the CCLK is changed, this configuration stays until the system is shut down. Therefore, in order to use the ADSP-21992 EZ-KIT LITE board with default CCLK, the user should reset this board.

One can run two ADSP-21992 EZ-KIT LITE boards using a single PC. Of course, the user should install USB drivers for two USB ports also. This evaluation board after being powered up will reset itself and will do some internal checking for a while. LED CR5 of this board (see figure A.1) will be on when this board is ready. This means, the user can open the VisualDSP++ environment.

When both are ready and the user opens the VisualDSP++ environment, only one board will be connected to this environment. LED CR5 will be blinking when the communication between the VisualDSP++ and the board is established. Later, this board will be referred as the primary board, while the other as the secondary one.

To use two boards in one PC, the user should press the reset button on both boards (S1) after both LED CR5s are on and wait until the LED CR5s are on again. When both LED CR5s are on, the primary board should be reset again and the VisualDSP++ can be opened. The VisualDSP++ will connect to secondary board since the primary is not ready yet. When the primary is ready, the user can open the VisualDSP++ again and the second VisualDSP++ will be connected to the primary board.

User cannot build the program on both the VisualDSP++ at the same time. This will generate a license error and one of the building processes will be terminated. Therefore, the programs should be build at different times.

The ADSP-21992 EZ-KIT LITE board is very sensitive. When someone powers up other device near to this board, this can influence the board and the communication via USB cable can be disturbed. It means before power up this board, the user should power up other circuit first.

For the experiments with the CAN bus, JP23 should be closed. This will give 120Ω for termination resistor on the CAN bus. The other jumpers are in default position.
Writing software using VisualDSP++:

Software in this project is written using the VisualDSP++ 3.0. This debugging environment is similar to the Microsoft Visual Studio. The programming language will be C. Special attention should be given in how to access the ADSP-21992's register and internal memory with the VisualDSP++.

To access the ADSP-21992's register, user must use inline assembly language. (Analog Device Inc., 2002c) gives detail explanation how to do this.

All peripherals in the ADSP-21992 have registers, which is mapped into the I/O space of the memory. To access this, user must choose the page where the peripheral register is located. In the adsp-2199x.h header file, there will be list of the page available in the ADSP-21992. Instruction: sysreg_write(sysreg_IOPG,<page>) will point to the page referred by <page>. After doing this, all registers on that page will be accessible. To read data, user uses io_space_read(<memory_location>). The <memory_location>s are listed on the adsp-2199x.h and adsp-219x.h header files. This must be a constant value. Using other than constant value will generate a compiler internal error. This instruction will return an integer value. io_space_write(<memory_location>,<value>) is for writing. The <value> can be constant value or from variables.

The peripheral interrupt is mapped to the user's interrupt. There are twelve user's interrupts. The user's interrupt #0 has highest priority over all users' interrupts and it is indicated by SIG_INT4. Therefore, the user’s interrupt #1 is on SIG_INT5 and so on. To connect the peripheral interrupt to users’ interrupt, a specific value is needed for the Interrupt Priority Register (IPR). SIG_INT4 has value 0, SIG_INT5 has 1 and so on.

A.2 Initialization and detail operation of the CAN module in the ADSP-21992

To set the CAN speed, user should configure value of CANBCR0 and CANBCR1. CANBCR0 holds 10-bit of BRP (Baud Rate Prescaler). BRP means the width of one time quantum and several time quanta form one bit of data. One bit of data is 4-25 time quanta and divided into four regions, i.e. synchronization, propagation segment, phase segment 1 and phase segment 2 (see figure A.2).

<table>
<thead>
<tr>
<th>TSEG1</th>
<th>TSEG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>Propagation Segment</td>
</tr>
</tbody>
</table>

Figure A.2 One-bit time representation

In the ADSP-21992, the propagation segment and phase segment 1 are combined into one, called TSEG1, while phase segment 2 is TSEG2 as shown in figure A.2.
Combination of BRP, TSEG1 and TSEG2 will form the desired CAN speed as written in equation (1) (Analog Device Inc., 2002a). HCLK is the peripheral clock, usually half of processor clock.  

\[
\text{desired\_speed} = \frac{HCLK}{[1+(1+\text{TSEG1})+(1+\text{TSEG2})](\text{BRP}+1)}
\]  

(1)

It is important to note that all combinations are only candidates. Only some combinations will really work in the system. They have to be tested in system. In Appendix B section B.1 gives a description and code of experiments used to determine working combinations. The rule of thumb for choosing value for BRP, TSEG1 and TSEG2 are

- TSEG1 \( \geq \) TSEG2
- SJW \( \leq \) TSEG2
- SAM = 1, only if BRP < 4
- If BRP = 0, minimum value of TSEG2 is 3
- If BRP = 1, minimum value of TSEG2 is 2
- If BRP > 1, minimum value of TSEG2 is 1

The sample point is the point in time at which the CAN bus level is read and interpreted as the value of that respective bit. Its location is at the end of TSEG1. If SAM is 0 the CAN bus is sampled only once and the value that is read in that moment is assigned to the bit. If SAM is 1 the CAN bus is sampled 3 times and the majority determines the resulting value. These three times sampling mode are not allowed when BRP is less than four.

This procedure must be followed to configure the CAN module:

1. Enable configuration mode by setting bit CCR (7) in CANMCR (Master Control Register) register: CANMCR=0x0080.
2. Wait until the CAN module is in configuration mode: test bit CCA (7) of CANGSR (Global Status Register) register until it becomes 1.
3. Initialize CANBCR0, CANBCR1 (Bit Configuration Register) and eventually the acceptance mask registers (CANMBAMxH, CANMBAMxL (MailBox Acceptance Mask Filter), x=0,..,F)
4. Cancel the configuration mode and enter in the normal mode by clearing the bit CCR (CAN Configuration Request) (7) in CANMCR: CANMCR=0x0.
5. Wait until the CAN module is in normal mode: test bit CCA (CAN Configuration Acknowledge) (7) of CANGSR register until it becomes 0.
6. Wait for CAN module to power up, i.e. 11 consecutive recessive bits must be detected on the CAN bus line.

ADSP-21992 has 16 CAN mailboxes. Those mailboxes can be configured either as transmitter or receiver. Before using these mailboxes, they must be set up first either for transmitter or receiver. This set up should be done while those mailboxes are disabled. CANMD (Mailbox

---

1 By default, the ADSP-21992 EZ-KIT LITE is clocked by 16MHz for processor clock, therefore the HCLK is 8MHz.
Direction) will set a mailbox as transmitter or receiver, while CANMC (Mailbox Control) will enable or disable them.

In a receiver mailbox, all enabled mailboxes will check message ID from CAN bus and compare it with its own. If it is the same as its message ID, data from that message will be saved to its buffer, i.e. CANMBxxDATA3-0. The first byte goes to high byte of CANMBxxDATA3. After all data are saved, receiver mailbox interrupt will be generated (if the interrupt is enabled) and CANRMP (Receive Message Pending) for that mailbox is set. User should reset this bit manually after reading the data. For receive mailboxes, user can also choose between overwritten unread data or not. CANOPSS (Over Protection Single Shot) (1-bit for each mailbox) will take care of this.

To set the interrupt for CAN, user needs to connect this peripheral interrupt to user’s interrupt. User’s interrupt is from INT4 to INT15. The next step is to fill the appropriate value into the IPR7 register as defined on chapter 13 of (Analog Device Inc., 2002a). For INT4, the value should be zero; INT5 will get one, and so on.

On page 13-3 of (Analog Device Inc., 2002a), there is no information about CAN interrupt in Table 13-1. According to Analog Device support system for mixed signal DSP, the following information should be added to the bottom of Table 13-1.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>IPR7[3:0]</td>
<td>CAN_RX_IRQ</td>
<td>CAN mailbox receive interrupt</td>
</tr>
<tr>
<td>29</td>
<td>IPR7[7:4]</td>
<td>CAN_TX_IRQ</td>
<td>CAN mailbox transmit interrupt</td>
</tr>
<tr>
<td>30</td>
<td>IPR7[11:8]</td>
<td>CAN_ERR_IRQ</td>
<td>CAN Error Condition Interrupt</td>
</tr>
<tr>
<td>31</td>
<td>IPR7[15:12]</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

A.3 Initialization and detail operation of the SPORT in the ADSP-21992

Three pairs of line interface support SPORT. They are data, clock and frame synchronization. To use SPORT, user must connect the cable appropriately, i.e.:

- connect data pin of transmitter to that of receiver (DT to DR)
- connect clock pin of transmitter to that of receiver (TCLK to RCLK)
- connect frame synchronization of transmitter to that of receiver (TFS to RFS)

These are the hardware part. Figure A.3 shows the cable connection between two ADSP-21992 boards.
For the software part, user needs to initialize the configuration of the SPORT, especially the speed or baudrate. The basic configuration of SPORT is that the transmitter will determine the clock rate. It means that the transmitter will generate the clock and receiver depends on that clock. This also applies to the frame synchronization. For this purpose, the transmitter will choose internal clock and frame synchronization, while the receiver will choose the external ones. For the transmitter determines the clock, SP_TSCLKDIV will divide HCLK to get the baudrate.

The user should disable data independent transmit frame synchronization bit (in the SP_TCR), otherwise SPORT will continually send data though there is no new data in the transmit buffer.

According to page 8-8 of (Analog Device Inc., 2002a), the SPORT should be configured before enabling it. Both the configuration and the enable bit share the same register. Therefore, a user should write values to SP_RCR and SP_TCR except for bit 0 first. The next step is writing the same data with bit-0 is set.

To set the interrupt for CAN, user needs to connect this peripheral interrupt to user's interrupt. User's interrupt is from INT4 to INT15. The next step is to fill the appropriate value into the IPR0 register as defined on chapter 13 of (Analog Device Inc., 2002a). For INT4, the value should be zero; INT5 will get one, and so on.

Writing a value to SP_TX (transmit buffer) will automatically send that value. While this buffer is empty, transmit interrupt is generated no matter whether user connects this interrupt to user's interrupt or not. If not, the program will jump to unknown place. This is the reason why user should disable all interrupt in the beginning of the initialization as written in chapter 4.

After receiving data into SP_RX, the receive interrupt is generated. To read that data will also clear the interrupt signal. That is why reading the data into buffer is done here.
 Fault-tolerance in real time distributed system using The CT Library
Appendix B

Experiments performed

B.1 CAN in ADSP-21992 - pre-experiment

The purpose of this experiment is to see the performance of CAN controller in ADSP-21992. For this experiment, two boards of ADSP-21992 EZ-KIT LITE were needed. Each board has its own PC to download the program.

Figure C.1 shows the set up for this experiment. Board 1 will send a message to board 2 via CAN bus, and then as soon as board 2 receives that message, board 2 will send a message back to board 1. Board 1 will measure the time delay of this one time loop and send it to PC via RS-232. Therefore, before sending a message, board 1 will start its timer and stop it as soon as it receives a message from the other board. This experiment uses several CAN speed with various payload.

![Set up system for performance experiment](image)

The first thing to do is set up the CAN speed. According to the discussion on chapter 2, combination of BRP, TSEG1 and TSEG2 will form desired CAN speed, but their combinations are only candidates, they should be tested in experiment. To choose the best candidate, user can use these rules of thumb:

1. BRP should be as small as possible. This will enable user to tune where the sampling process is occurred.
2. The comparison between 1+TSEG1 and 1+TSEG1+TSEG2 is 60% to 80%, because the sampling process is happened between TSEG1 and TSEG2.

A user has to calculate the combination between BRP, TSEG1 and TSEG2 in order to get the desired speed. The problem is those combinations are only candidates. Therefore, the user should try them one by one. The higher the CAN speed the more difficult it is to find a good combination. For this reason, the code below is written in order to find a valid combinations. The following code is part of the code for this experiment.

One board acts as a master, which will determine the combination. Once the combination is ready, the master initializes the CAN controller. When the combination is unacceptable, program will be trapped inside this initialization. Timer will help the program jump out of this trap. If the combination is acceptable, that combination is sent via SPORT to the slave. After the slave is ready (after initialization), the master starts to send one data and wait for an echo. If in predefined time there is no echo, it means that combination is not working although it is acceptable in the initialization process.

**Board 1:**

```c
void main() {
    // do initialization for CAN, SPORT and timer
    // set the interrupt
    // from this nested 'for' the combination of BRP, TSEG1 and TSEG2 are generated
    for(BRP=3;BRP<20;BRP++){
        for(TSEG1=15;TSEG1>=0;TSEG1--){
            for(TSEG2=0;TSEG2<=TSEG1;TSEG2++){
                if(TSEG2 <= 7){
                    printf("BRP = %i, TSEG1 = %i, TSEG2 = %i
",BRP,TSEG1,TSEG2);
                    bOk = false;
                    bChange = false;
                    bOk = true;
                    BCR0 = BRP;
                    BCR1 = TSEG2 << 4;
                    BCR1 |= TSEG1;
                    BCR1 |= 0x200;
                    // start timer
                    sysreg_write(sysreg_IOPG, Timer_Page);
                    io_space_write(T_GSR0,0x0100);
                    can_init();
                    // stop timer
                    sysreg_write(sysreg_IOPG, Timer_Page);
                    io_space_write(T_GSR0, 0x0200);
                    if(!bInitFailed){
                        // send data via SPORT here
                        buffer = BRP << 7;
                        buffer |= (TSEG2 << 4);
                        buffer |= TSEG1;
                        sysreg_write(sysreg_IOPG, SPORT0_Controller_Page);
                        while((io_space_read(SP0_STATR) & 0x0004)==0x0004);
                        io_space_write(SP0_TX,buffer);
                    }
                }
            }
        }
    }
}
```

56
while(!bOk);

// start timer
sysreg_write(sysreg_IOPG, Timer_Page);
io_space_write(T_GSR0, 0x0100);
printf("sending message...\n");
CANTx();
while(!bChange){
    if(bAck){
        printf("Success!!\n");
        printf("BRP=%i, TSEG1=%i,
            TSEG2=%i\n", BRP, TSEG1, TSEG2);
        bAck = false;
        bChange = true;
    }
}
}

On the part of code above, there are nested loops. With these loops, combinations of BRP, TSEG1 and TSEG2 are made. The max value for TSEG2 is equal to TSEG1 but it cannot exceed 7. The other nested loops are used to send the combination and to examine whether that combination is acceptable or not.

Software in board 2 is passive. It only receives combination from board 1 and initializes itself. Then board 2 waits for data from board 1. As soon as it receives data, this data will be transmitted back to board 1.

According to the experiments, not all combinations for highest speed worked, but the lower the speed, the more combination worked. The higher the speed, the more difficult the configuration will take place. The SJW and SAM gave no influences in this experiment. When the combination between BRP, TSEG1 and TSEG2 worked, all SJW and SAM in this combination also worked.

Table B.1 gives information about one time loop in CAN. Time loop means measurement of once transmit and once receive, while figure B.2 shows its plot. In this experiment, two ADSP-21992 EZ-KIT LITE boards are connected via CAN bus as seen in figure B.1. The first ADSP will start its timer and send message to the second ADSP. After receiving that message, the second ADSP send message to the first one. The first board will stop its timer as soon as message from the other board is received. The experiments were done with several payloads, in this case one to eight bytes.
Table B.1 One time loop in CAN experiment

<table>
<thead>
<tr>
<th>Payload (bytes)</th>
<th>8 kbps (ms)</th>
<th>12.5 kbps (ms)</th>
<th>25 kbps (ms)</th>
<th>50 kbps (ms)</th>
<th>80 kbps (ms)</th>
<th>100 kbps (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.44</td>
<td>9.25</td>
<td>4.63</td>
<td>2.31</td>
<td>1.46</td>
<td>1.05</td>
</tr>
<tr>
<td>2</td>
<td>16.45</td>
<td>10.53</td>
<td>5.27</td>
<td>2.63</td>
<td>1.66</td>
<td>1.20</td>
</tr>
<tr>
<td>3</td>
<td>18.57</td>
<td>11.89</td>
<td>5.95</td>
<td>2.97</td>
<td>1.87</td>
<td>1.35</td>
</tr>
<tr>
<td>4</td>
<td>20.44</td>
<td>13.09</td>
<td>6.55</td>
<td>3.28</td>
<td>2.06</td>
<td>1.48</td>
</tr>
<tr>
<td>5</td>
<td>22.58</td>
<td>14.45</td>
<td>7.23</td>
<td>3.62</td>
<td>2.27</td>
<td>1.64</td>
</tr>
<tr>
<td>6</td>
<td>24.57</td>
<td>15.73</td>
<td>7.87</td>
<td>3.94</td>
<td>2.47</td>
<td>1.78</td>
</tr>
<tr>
<td>7</td>
<td>26.82</td>
<td>17.17</td>
<td>8.59</td>
<td>4.30</td>
<td>2.69</td>
<td>1.94</td>
</tr>
<tr>
<td>8</td>
<td>29.07</td>
<td>18.61</td>
<td>9.31</td>
<td>4.66</td>
<td>2.92</td>
<td>2.10</td>
</tr>
</tbody>
</table>

B.2 CAN message experiment

In ADSP-21992, there is an indicator for receiver mailbox that there is an unread message there, i.e. CANRMP (Receiver Message Pending). User should reset corresponding bit inside CANRMP manually after reading the message. Beside this flag bit, on the receiver user has two options whether the unread message will be overwritten or not. From transmitter point of view, every successful transmission there will be an interrupt generated.

There are three questions concerning CAN message:

1. What happened if there is unread message and user choose not to overwrite that message? Does the transmitter know this by using its interrupt?
2. Will all mailboxes from same node with filter matching messageID receive the message or only one? If only one mailbox, then what is the rule?
3. How does CAN module organize data bytes in the mailbox?

For this experiment, set up as seen on figure B.1 was used.

Experimental setup:

1. There are two ADSP-21992 EZ-KIT LITE boards, one for transmitter only and the other for receiver only.
2. On the receiver part, several mailboxes will use the same messageID and the Overwrite Protection bits are enabled.
3. Transmitter will send from one mailbox and detect acknowledge signal with transmitter interrupt.
4. If the message is successfully sent, there will be an interrupt generated then transmitter will send the next message.

Result:

1. The first message is received by the highest mailbox number then RMP (Receive Message Pending) for this mailbox is set. This will indicate that there is an unread
message inside that mailbox (user should reset this bit after reading the message). In this experiment, the data is left unread.

2. The second message is sent. The mailbox with the highest number cannot receive this message, so the next mailbox is checked for another matching identifier. If CAN controller found the next match identifier, this mailbox will receive this message and RMP for this mailbox is set. This process goes on until all mailboxes received message. This means all RMPs are set.

3. When the next message is sent again, no mailbox can receive this message (there is no next mailbox with match identifier is found. Therefore, the RML (Receive Message Lost) of the last match mailbox is set. But there is an acknowledge signal at the transmitter.

This experiment also tried to send less than 8 bytes. The order of data should be: DATA3High, DATA3 Low, DATA2High, DATA2Low, DATA1High, DATA1 Low, DATA0High, DATA0Low. When user only needs to send 3 bytes of data, he should save this data on DATA3High, DATA3 Low, DATA2High.

Conclusion:

1. Transmitter interrupt mailbox cannot detect if there is unread message on receiver mailbox and that data cannot be overwritten.
2. If there are many mailboxes with the same message ID, only the mailbox with the highest number will get the data. For example, if mailbox numbers 10 and 9 have the same message ID, then mailbox number 10 will get the data.

3. ADSP-21992 organizes 8 bytes data in CANMBxxDATA3-0. The first byte will go to high byte of CANMBxxDATA3. The second byte will go the low byte of CANMBxxDATA3, and so on.

B.3 One-way configuration without fault tolerance

For one-way configuration in this project, here are the requirements:

- PC with VisualDSP++
- 2 ADSP-21992 EZ-KIT LITE boards with USB cable for each board
- LINIX motor
- Cable for CAN and SPORT connection. These cables have DPST (Double Pole Single Throw) switch to simulate the broken and recovered link
- 1 block of additional circuit and 2 set of indicator LEDs

Before doing the experiment, one needs to connect PCB as shown on figure B.3 to the ADSP-21992 EZ-KIT LITE boards. The position of the jumpers is shown in table A.1. The pin numbers are referred to the real board because several parts of figure A.1 are different from the real board. The procedures below refers to figure B.3 and the ADSP-21992 EZ-KIT LITE board. Here is how to connect the cables:

- The ADC cable goes to board 2, the red cable to pin 8 (VIN1) of P4 and the black cable goes to pin 1 of P4.
- The EIU cable goes to board 2, the Ch.A cable (red) to pin 2 of JP28 and the Ch.B cable (black) to pin 2 of JP29.
- The PWM cable goes to board 1, the red cable to AUX0 of P10 and the black cable to DGND of P10.
- PF15 of P8 on board 2 connects to PF14 of P8 on board 1.
- The encoder cable connects to encoder of the LINIX motor, not encoder at the load. This is flat cable with five different colours. Insert the connector such that the black cable (GND) will be on top.
- For CAN cable, the CANH pin (P12 or P13) of board 1 connects to that of board 2, so does the CANL pin (P12 or P13).
- Red cable for motor connects to positive terminal on the LINIX motor, the black one connects to the other terminal.

The user uses the potentiometer to set the desired speed for the LINIX motor.

Now it comes the software part. All software in this experiment are stored in the directory \software\embedded\OneWay. The VisualDSP++ for board 1 opens OneWay2_1.dxe file form debug directory. This is the result of building a project file which uses:

- the CT Library (both the C and the header files),
- PID12Main,
- PID12Controller (both the C and the header files),
- CANLinkDriver and AuxPWMLinkDriver (both the C and the header files)

For board 2, the VisualDSP++ will open OneWay2_2.dxe file. This file come from building a project file which uses:
- the CT Library (both the C and the header files),
- PID11Main,
- PID11Controller (both the C and the header files),
- CANLinkDriver, ADCLinkDriver and EIULinkDriver files (both the C and the header files)

The order to run the board is not important because the other party will wait for synchronization. Table B.2 is a copy of Embedded System Realization report for one-way configuration.

Table B.2 Sampling time and frequency for each CAN speed in one-way configuration

<table>
<thead>
<tr>
<th>CAN Speed (kbps)</th>
<th>Sampling time (ms)</th>
<th>Sampling frequency (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.00</td>
<td>20,916</td>
<td>47,8097</td>
</tr>
<tr>
<td>12.50</td>
<td>13,4970</td>
<td>74,0905</td>
</tr>
<tr>
<td>25.00</td>
<td>7,0525</td>
<td>141,7937</td>
</tr>
<tr>
<td>50.00</td>
<td>3,8656</td>
<td>258,6904</td>
</tr>
<tr>
<td>80.00</td>
<td>2,6724</td>
<td>374,1990</td>
</tr>
<tr>
<td>100.00</td>
<td>2,2736</td>
<td>439,8263</td>
</tr>
</tbody>
</table>

Figure B.3 Lay out of the additional circuit (H-bridge and reference signal PCB)

Table B.2 Sampling time and frequency for each CAN speed in one-way configuration
B.4 One-way configuration with fault tolerance

The configuration of the hardware is exactly the same as in the section B.3. The only different is the file for the experiment. The two set of LED indicators are plugged to P8 of the ADSP-21992 EZ-KIT LITE. The pin with resistor is connected to 3.3V pin, the green LED to PF0 and the yellow LED to PF2.

All software for this experiment are stored in the directory \software\thesis\OneWayFT. First, the OneWay1.dxe is opened. This file contains:

- the CT Library (both the C and the header files),
- PID12Main,
- PID12Controller (both the C and the header files),
- RemoteLinkDriver (both the C and the header files),
- CANDeviceDriver (both the C and the header files),
- SPORTDeviceDriver (both the C and the header files),
- AuxPWMLinkDriver (both the C and the header files)

The next step is to open OneWay2.dxe. This file contains:

- the CT Library (both the C and the header files),
- PID11Main,
- PID11Controller (both the C and the header files),
- RemoteLinkDriver (both the C and the header files),
- CANDeviceDriver (both the C and the header files),
- SPORTDeviceDriver (both the C and the header files),
- ADCLinkDriver (both the C and the header files),
- EIULinkDriver (both the C and the header files)

The user can use two switches to break the cable both for the CAN bus and for the SPORT. There are two LED indicators to indicate that the link is available (if the LED is on). The green LED is for the CAN bus, while the yellow LED is for the SPORT.
Appendix C

Source code of ADSP specific parts (context switch and link drivers)

C.1 Context switch implementation

Before implementing context switch with longjmp() and setjmp() function, one needs to do little experiment to see whether those two functions can be used for implementing context switch and how. Simplifying approach used in (Milicev, 1998) minimalistic test kernel has been implemented. Section C.1.1 shows this implementation, while section C.1.2 shows changing in the CT Library. The last section shows snippet of code of the CAN bus and the SPORT Device Driver. Therefore, this section does not show the complete implementation.

C.1.1 Simple minimal kernel

```c
#include "setjmp.h"
#include <adsp-21992.h>
#include "context.h"

// to make a thread structure
struct Thread{
    int threadNum;
    int* SP;
    jmp_buf context;
    bool isBegining;
    struct Thread* next;
};

// to make a scheduler structure
struct Scheduler
{
    struct Thread* head;   // the first element
    struct Thread* tail;   // the last element
    int size;              // size of the thread
};

// simple run method for the thread
// counter and me->threadNum will be monitored
void Thread_run(struct Thread* me){
    int counter=0;int i=0;
    for(;i<5; i++){
        counter=counter+me->threadNum;
        printf("-");
        dispatch();
    }
```
Fault-tolerance in real time distributed system using The CT Library

```c
)

// to make a thread
void Thread_constructor(struct Thread* t){
    t->isBegining=true;
    t->SP=(int*)malloc(sizeof(int)*100);
    t->SP=(t->SP)+99;
    t->next=0;
}

// to make a scheduler
void Scheduler_constructor(struct Scheduler* me){
    me->head=0;me->tail=0;me->size=0;
}

// to add thread to scheduler
void Scheduler_add(struct Scheduler* me, struct Thread* el){
    if(me->size==0)
        me->head=me->tail=el;
    else {
        me->tail->next=el;
        me->tail=el;
    }
    me->size++;
}

// to get thread from scheduler
struct Thread* Scheduler_get(struct Scheduler* me){
    struct Thread* temp;
    if(me->size==0)
        return 0;
    temp=me->head;
    me->head=me->head->next;
    (me->size)--;
    return temp;
}

struct Thread* running;
struct Scheduler* scheduler;
jmp_buf* mainContext;
struct Thread* t;

void dispatch(){
    if(setjmp(running->context)==0){   // save to stack
        Scheduler_add(scheduler,running);  // add ‘running’ to scheduler
        running=Scheduler_get(scheduler);
        if(running==0) longjmp(*mainContext,1);
        else{
            if(running->isBegining){    // only at the beginning
                running->isBegining=false;  // clear the flag
                asm("I4=%0;" : "c"(running->SP)); // set the Stack Pointer
                Thread_run(running);      // run the thread
            }
            else{
                longjmp(running->context,1); // restore from stack
            }
        }
    }

}
```
int main(){
    int i=0;
    // allocate memory
    scheduler=(struct Scheduler*)malloc(sizeof(struct Scheduler));
    // to make jmp_buff
    mainContext=(jmp_buf*)malloc(sizeof(int)*75);
    Scheduler_constructor(scheduler); // to make 3 threads and add them to the scheduler
    for(;i<3;i++){  
        struct Thread* t=(struct Thread*)malloc(sizeof(struct Thread));
        Thread_constructor(t);
        t->threadNum=i;
        Scheduler_add(scheduler,t);
    }
    if(setjmp(*mainContext)==0){ // save context to stack
        running=Scheduler_get(scheduler); // get thread from scheduler
        if(running->isBegining){
            running->isBegining=false;
            asm("I4=%0; ": "c"(running->SP)) ; // set the Stack Pointer
            Thread_run(running); // run the thread
        }
        else
            longjmp(running->context,1); // restore context from stack
        return 0;
    }

C.1.2 Changes in CT library functions

CONTEXT_SWAPRESTORE macros

#define CONTEXT_SWAPRESTORE      asm("I4=%0;:" "c"(temp));

startswitch method

protected void Processor__startswitch(void){
    // allocate memory for maincontext
    maincontext = (jmp_buf *)malloc(sizeof(jmp_buf));
    // get running thread from dispatcher
    currentProcessThread = Processor__currentDispatcher->running;
    newProcessThread = currentProcessThread ;
    if (setjmp(*maincontext)==0){  
        // this is only for the first
        newProcessThread->isBegining = false;
        // set the Stack Pointer (I4) to the current thread
        asm("I4=%0;:" "c"(newProcessThread->sp));
        // set the run method
        runmethod = (void *)(*((REGISTER *)(newProcessThread->sp)));
    }
Fault-tolerance in real time distributed system using The CT Library

```
// interrupt can happen here
ENABLE_INTERRUPTS;
// run the method
runmethod(currentProcessThread);
}
}

stopswitch method

protected void Processor_stopswitch(void){
    longjmp(*maincontext,1);
}

c ontextswitch method

private void Processor_contextswitch(void){
    void (*runmethod)(ProcessThread);
    // get current thread and new thread
    tempProcessThread = currentProcessThread;
    currentProcessThread = newProcessThread;

    if (setjmp(tempProcessThread->context)==0){
        if (newProcessThread->isBeginning){
            // only for the first time
            newProcessThread->isBeginning = false;
            // set the Stack Pointer (I4) for the new thread
            asm("i4=%0;"
                ::"c"(newProcessThread->sp));
            // set the run method
            runmethod = (void *)((REGISTER *(newProcessThread->sp)));
            // interrupt can happen here
            ENABLE_INTERRUPTS;
            // run the method
            runmethod(newProcessThread);
        }
        else
            longjmp(newProcessThread->context,1);
    }
}

C.2 Communication Link Drivers

C.2.1 CAN

Transmit method

bool CANDeviceDriver_Transmit(CANDeviceDriver me, Object obj){
    int temp;
    temp = io_space_read(CANMC); // read CAN Master Control Register
    temp &= 0xffe; // clear bit 0
    io_space_write(CANMC, temp); // disable MB 0
    temp = (*(int*)obj); // get the data
    ```
io_space_write(CANMB00_DATA3,temp); // write that data to mailbox

temp = peerNodeID<<8; // place nodeID in the messageID
io_space_write(CANMB00_ID0,temp); // write it to the register

temp = 0xbfff;
io_space_write(CANMB00_ID1,temp); // write 0xbfff to the register
io_space_write(CANMB00_LENGTH,2); // send 2 bytes of data

temp = io_space_read(CANMC); // read CAN Master Control Reg.
temp |= 0x0001; // set bit 0
io_space_write(CANMC,temp); // enable MB 0

io_space_write(CANTRS,0x01); // transmit the data

return true;
}

Receive method

bool CANDeviceDriver__Receive(CANDeviceDriver me,Object obj){
    int temp;
    sysreg_write(sysreg_IOPG, CAN_Page); // set to CAN page

    temp = io_space_read(CANMC); // read CAN Master Control Reg.
temp &= 0xfffd; // clear bit 1
io_space_write(CANMC,temp); // disable MB 1

    *(int*)obj = io_space_read(CANMB01_DATA3); // read the data

io_space_write(CANRMP,0x0002); // it indicates data is read

temp = peerNodeID<<8; // place nodeID in the messageID
io_space_write(CANMB03_ID0,temp); // write it to the register

temp = 0xb7ff;
io_space_write(CANMB03_ID1,temp); // write 0xb7ff to register
io_space_write(CANMB03_LENGTH,0); // send 0 byte of data

temp = io_space_read(CANMC); // read CAN Master Control Reg.
temp |= 0x0002; // set bit 2
io_space_write(CANMC,temp); // enable MB 2

io_space_write(CANTRS,0x0002); // send acknowledge

return true;
}

C.2.2 SPORT

Transmit

bool SPORTDeviceDriver__Transmit(SPORTDeviceDriver me,Object obj){
    // set to SPORT page
    sysreg_write(sysreg_IOPG, SPORT0_Controller_Page);

    // send protocol
    bProtocol = true;
bFirst = true;
bWait = true;

Fault-tolerance in real time distributed system using The CT Library

```c
while((io_space_read(SP0_STATR) & 0x0004)==0x0004);
io_space_write(SP0_TX,0xbfff);

// wait until other party ready
while(bWait);

// send data
SPORTBuffer = (*((unsigned*)obj));
io_space_write(SP0_TX,SPORTBuffer);
return true;
}

Receive

bool SPORTDeviceDriver__Receive(SPORTDeviceDriver me, Object obj){
    // set to SPORT page and read data
    sysreg_write(sysreg_IOPG, SPORT0_Controller_Page);
    *(unsigned*)obj = SPORTBuffer;

    // send acknowledge
    io_space_write(SP0_TX,0xb7ff);
    return true;
}
```
Reference:


Avizienis, A., J.-C. Laprie and B. Randell Fundamental Concepts of Dependability,

CAN in Automation (2003), *CAN in Automation - protocol*, [http://www.can-cia.org/can/protocol/](http://www.can-cia.org/can/protocol/)

Douglass, B.P. (2003), *Real-time design patterns : robust scalable architecture for real-time systems*, Addison-Wesley,

Engelen, T.v. (1999), ARTY - Autonomous Mobile Robot, University of Twente, Enschede.

Hilderink, G.H. Communicating Threads for C - Portability Guide. A White Paper, pp. ISSN:


Kvaser Advanced CAN solutions for hardware, software, consulting and education, [http://www.kvaser.com](http://www.kvaser.com).

Milicev, D. (1998), Object-oriented programming and real-time (in Serbian), University of Belgrade,

Robert Bosch GmBH (1991), *CAN Specification version 2.0*, Robert Bosch GmBH,