Development of a Fault-Tolerant Bus System suitable for a High-Performance, Embedded, Real-Time Application on SUNSAT's ADCS

Xandri C. Farr

Thesis presented in fulfilment of the requirements for the degree of

Master of Engineering

at the

University of Stellenbosch

Department of Electrical and Electronic Engineering.

Pectora roborant cultus recti

Study leaders: Mr J. Treurnicht and Prof. A. Schoonwinkel

22nd November 2000

Stellenbosch University http://scholar.sun.ac.za

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own, unless stated otherwise, and has not previously been submitted at any university for a degree.

Stellenbosch University http://scholar.sun.ac.za

Opsomming

Die *Attitude Determination and Control System* (ADCS) van *Stellenbosch University Satellite* (SUNSAT I) is 'n geïntegreerde stelsel wat voorsiening maak vir 'n mate van oortolligheid en 'n vermoeë om stelseldata te bestuur vir goeie satellietbeheer. Nietemin, hierdie oortolligheid is nie baie toeganklik nie en daar is 'n gebrek aan aanpasbaarheid tydens die toets en integrasie van individuele modules of moontlike stelseluitbreidings. Die doelwit van hierdie tesis was dus die ontwikkeling van 'n betroubare, aanpasbare, modulêre kommunikasie stelsel wat 'n tipe oortolligheid insluit sodat intydse data bestuur kan word en algehele stelsel ondergang vermy kan word.

Die eerste stap in die projek se ontwikkelings metodiek was om 'n opsomming te verkry van die vereistes en spesifikasies deur die huidige ADCS se argitektuur en databestuur te ondersoek. 'n Ondersoek na die *Controller Area Network* (CAN) protokol het getoon dat hierdie tegnologie aan baie van die vereistes voldoen. Dit het aanleiding gegee tot die ontwerp en implementering van 'n paar konsep ontwerpe gebaseer op CAN. Daarna is 'n demonstrasie model bestaande uit drie prototipe nodusse gebou. Die werksverrigting van die sogenoemde *dual CAN node*, is ondersoek en 'n ekstrapolasie was gemaak om vas te stel of die argitektuur die volkome ADCS kan huisves.

Deur demonstrasie was daar getoon dat die dual CAN node wel genoeg ruimte verskaf om al die verwerkers, aktueerders en sensors van die ADCS te akkommodeer. Daar was terselfdertyd getoon dat betroubaarheid en robuustheid verhoog is deur die verbeterde oortolligheid op 'n node-vlak sowel as op die groter stelsel-vlak. 'n *Dubbele* CAN bus is gebruik vir oortolligheid op 'n node-vlak. Op 'n stelsel-vlak kan die bevel-en-dataversamelings modules (ACP en OBC's) effektief gemultipleks word op die netwerk van aktueerders en sensors. Daar was verder getoon dat die foutopspoorings vermoeë en diagnostiese vermoeë verbeter kan word en die kompleksiteit van die kommunikasie argitektuur en ooreenkomstige kabelharnasse vereenvoudig kan word. Die gevolg is vereenvoudigde toegang tot modules en vergemaklikde opgradering.

Stellenbosch University http://scholar.sun.ac.za

Abstract

The Attitude Determination and Control System (ADCS) of the Stellenbosch University Satellite (SUNSAT I) is an integrated system providing some redundancy and the necessary data management to control the spacecraft. However, the redundancy is not easily accessible and there is a lack in flexibility when testing individual modules during integration or when the system needs to be extended. The objective of this thesis was thus to develop a high reliability, flexible, modular communication system that included some type of redundancy to manage real-time data and to prevent severe malfunctioning of the entire system.

The first step in the project's development methodology was to summarise the requirements and specifications by studying the current ADCS architecture and data management. An investigation into the Controller Area Network (CAN) protocol showed that this technology would fit the requirements very well, leading to the design and implementation of several concept topologies based on CAN. Thereafter, a demonstration model consisting of three prototype nodes was composed. The performance of the so called *dual CAN node* was analysed and an extrapolation was made to determine whether the architecture could support the complete ADCS.

It was demonstrated that the dual CAN node provides enough room to accommodate all the processors, actuators and sensors of the ADCS. At the same time, it was shown that reliability and robustness was increased by enhanced redundancy at a node-level as well as at the greater system-level. A *dual* CAN bus was provided for redundancy at a node-level. At the system-level, the command and data-gathering modules (ACP or OBC's) can now effectively be multiplexed on the network of actuators and sensors. Furthermore, it was shown that error detection capabilities and diagnostics can be enhanced and the complexity of the communication architecture and related wiring harnesses can be reduced. This allows easier access to modules and simplifies development.

Stellenbosch University http://scholar.sun.ac.za

Contents

Acknowledgements ix

List of Acronyms x

List of Figures xiv

List of Tables xvi

1 Introduction **1**

1.1 The Thesis 1

1.1.1 The Meaning of the Title 1

1.1.2 The Objective of the Thesis 2

1.1.3 The Project Methodology 2

1.2 The ADCS on SUNSAT I 3

1.2.1 The ADCS Architecture 4

1.2.2 Number of Effective Nodes on the ADCS 6

1.2.3 Number of bytes per node 6

1.2.4 Data Rates and Efficiency 7

1.2.5 Shortcomings 8

1.3 Requirements for the Upgrading of the ADCS 9

1.3.1 Enhanced Diagnostics 10

1.3.2 Increased Modularity 10

2 Why CAN? **14**

2.1 Possible Architectures, Protocols, and Technologies 14

2.1.1 Can CAN manage the ADCS' Data Load? 15

Stellenbosch University http://scholar.sun.ac.za

2.1.2 Appearance in Space 16
2.1.3 Targeted CAN Device for this Project 17

3 Design Overview 18
3.1 The Proposed Architecture 18
3.1.1 CAN Bridges 19
3.2 Redundancy 20
3.3 The Dual Bus 20
3.3.1 Physical Dual Bus 22
3.3.2 Virtual Dual Bus 22
3.4 Criteria for Designing a Dual CAN Node 22
3.5 Designing the Architecture 23
3.5.1 Design Step One 23
3.5.2 Design Step Two 23
3.5.3 Design Step Three 25
3.5.4 Design Step Four 25
3.5.5 Design Step Five 26
3.5.6 Design Step Six 27
3.6 Application Interfaces and Handshaking of the Dual CAN Node 27
3.6.1 Registers as Interfaces 28
3.7 Hardware 29
3.7.1 Choosing a Reconfigurable Device 29
3.7.2 Program Memory Size 31
3.7.3 Address Decoding 32
3.7.4 Reset Signal 32
3.7.5 CAN Transceivers 34
3.7.6 Dual CAN Node Power Consumption 35
3.8 Software Development 35
3.8.1 Concept Design 35
3.8.2 Initial Development of the CAN Device Drivers 36
3.8.3 Device Drivers and the OSI Model for Software Development 37

Stellenbosch University http://scholar.sun.ac.za

3.8.4 Interprocessor Communication Protocol . . . 38
3.8.5 The Real-Time Kernel . . . 39
3.8.6 Dual Node Switching Algorithm . . . 41
3.8.7 CAN Message Priorities . . . 42
3.8.8 Health & Safety . . . 42
3.8.9 Five Message Window . . . 43
3.9 Interrupts . . . 44
3.9.1 Writing to the Twin Processor's Interrupt Line . . . 44
3.9.2 The EPP Wait Signal . . . 46
3.9.3 Application Writing to the Master's Interrupt Line . . . 48
3.9.4 Application Reading from the Dual CAN Node . . . 49
3.9.5 The Master writing an Address to a Sensor/Actuator Peripheral . . 51
3.9.6 CAN Interrupts . . . 52

4 Analyses **54**
4.1 AC Analyses . . . 56
4.1.1 Extrapolation of the *Useful Processing Time* . . . 62
4.2 DC Analyses . . . 63
4.2.1 Omitting the CPLD . . . 65
4.3 Design Analyses . . . 66
4.3.1 Pull-up Resistors . . . 66
4.3.2 Interrupt latencies . . . 67
4.3.3 Time-loading . . . 67
4.3.4 Memory-Loading . . . 68
4.3.5 Stress Testing . . . 69
4.3.6 Comments on the C505C . . . 70

5 Conclusions and Recommendations **71**
5.1 Conclusions . . . 71
5.2 Recommendations . . . 75

Stellenbosch University http://scholar.sun.ac.za

Bibliography **77**

A Controller Area Network **80**

A.1 BasicCAN and FullCAN 81

A.1.1 BasicCAN 81

A.1.2 FullCAN 81

A.2 Non-Destructive Bitwise Arbitration 82

A.3 Error Detection Capabilities 82

A.4 Throughput efficiency 83

A.5 Bit Stuffing 84

A.6 Calculating and configuring the Bit Timing 84

A.7 Synchronisation 85

A.8 What CAN devices are available? 87

A.9 Serial Linked I/O's 89

A.10 CAN Interrupts 89

A.11 CAN Remote frames 90

A.12 CAN's CRC 91

A.13 CAN's Bus length 91

B Project Details **93**

B.1 Scheduling of Data on the ADCS 93

B.2 Fieldbus Protocols Compared 96

B.3 The CPLD's Thermal Analysis 99

B.3.1 Estimated Power Consumption (P_{EST}) 99

B.3.2 Maximum Power for the Device and Package (P_{MAX}) 101

B.3.3 Comparing the Estimated and the Maximum Power Values 101

B.4 Calculating the Size of the Program Memory 101

B.4.1 Bytes per Instruction 101

B.4.2 Cycles per Instruction 101

B.4.3 Average Number of Times that Instructions are Repeated per Second102

B.5 System Power Consumption 103

Stellenbosch University http://scholar.sun.ac.za

B.6 Test and Evaluation Setup . . . 104
B.6.1 AC Characteristics for the Dual CAN Node . . . 105
B.6.2 Parallel Port Interface . . . 108
B.6.3 PCCAN - Note of Warning . . . 108
B.6.4 Telecommand Reset Pull-up Resistor . . . 109
B.6.5 PC's used as Interface to the Dual CAN Node . . . 109

C PCCAN 110
C.1 PCCAN's Interframe Spacing . . . 111

D Dual CAN node Schematics and PCB layout 112
D.1 Schematics . . . 112
D.2 PCB layouts . . . 117
D.3 Components list . . . 120

E Firmware and software 124
E.1 Processor Firmware . . . 124
E.2 VHDL Code . . . 127

Stellenbosch University http://scholar.sun.ac.za

Acknowledgements

I would like to thank the following contributors for their part in the successful completion of my work:

- My heavenly Father, without whom I am nothing and this project comes to nothing. With His power, given to me through the Holy Spirit, I was able to start and complete this project.
- My parents, grandparents and friends, who have supported me and who are always interested in my work.
- Prof. A. Schoonwinkel for his help and contributions.
- Mr Johann Treurnicht who taught me to look at the bigger picture. I hope that I have made this invaluable asset my own. Without him this project would still be a concept on paper. Thank you that your door was more than a passage to practical help.
- Mr Johan Grobbelaar for masterly designing the PCB's.
- Mr Johan Arendse for teaching me some advanced soldering skills and for his patience during the populating of the PCB's. Thank you for always finding some time for me.
- Rachel for proofreading that leaded to a professional document.

Stellenbosch University http://scholar.sun.ac.za

List of Acronyms

ACP	Attitude Control Processor
ADC	Analogue to Digital Converter
ADC16	16bit Analogue to Digital Converter
ADCS	Attitude Determination and Control System
CAN	Controller Area Network
CCD	Charge Coupled Device
CPLD	Complex Programmable Logic Device
CRC	Cyclic Redundancy Check
DSP	Digital Signal Processor
EDAC	Error Detection And Correction
EEPROM	Electrical Erasable Programmable Read Only Memory
EKF	Extended Kalman Filter
EOM	End Of Message
EPP	Enhanced Parallel Port
FPGA	Field Programmable Gate Array
GPS	Global Positioning System
HSS	Horizon and Sun Sensors
ICP	Interface Control Processor
I/O	Input and Output
ISP	In-System Programmable
JPL	Jet Propulsion Laboratory
JTAG	Joint Test Action Group
LEO	Low Earth Orbit
LSB	Least Significant Bit (Byte)
max.	Maximum
MM	Magnetometer
MSB	Most Significant Bit (Byte)
MSI	Medium Scale Integration
MT	MagnetoTorquer
MUX	Multiplexer
NASA	National Aeronautics and Space Administration
OBC1	OnBoard Computer One
OBC2	OnBoard Computer Two

Stellenbosch University http://scholar.sun.ac.za

OSI	Open Systems Interconnection
OTP	One Time Programmable
PC	Personal Computer
RW	Reaction Wheel
RWSSBUS	Reaction Wheel Star Sensor Bus
RSS	Rough Suncell Sensors
SCP	Star sensor Control Processor
SS	Star Sensor
SSI	Small Scale Integration
SUNSAT	Stellenbosch University Satellite
UART	Universal Asynchronous Receiver/Transmitter

Stellenbosch University http://scholar.sun.ac.za

List of Figures

1.1 The engineering model of the ADCS of SUNSAT 1. 3

1.2 A simplified diagram of the two most important sub-systems that interface with the ADCS, as well as the sensors and actuators involved in orientation control. 4

1.3 A distributed system. Source: Adapted from Lawrenz, 1997. 11

1.4 The ADCS as a distributed system. 12

3.1 This is a concept design of the proposed ADCS architecture which is based on a CAN bus for communication. 18

3.2 Two possible configurations for the dual CAN bus. (a) Physical dual bus. (b) Virtual dual bus. 21

3.3 Step one in the design of a possible interface between the dual network and the application. 24

3.4 Step two in the design of a possible interface between the dual network and the application. 24

3.5 Step three in the design of a possible interface between the dual network and the application. 25

3.6 Step four in the design of a possible interface between the dual network and the application. 26

3.7 Step five in the design of a possible interface between the dual network and the application. 27

3.8 Step six in the design of a possible interface between the dual network and the application. 28

3.9 A block diagram indicating the main components and signals of the dual CAN node. 29

3.10 The RESET circuit for the C505C processors. 33

3.11 Waveforms for the RESET signal. (MAX+plusII) 33

Stellenbosch University http://scholar.sun.ac.za

3.12 Flow diagram of the dual CAN node's Write-cycle. 36

3.13 An illustration of how the hardware and software are partitioned into the layers of the OSI model. 38

3.14 Main event loop of the application software of the dual CAN node. 40

3.15 The five-message-window error detection (a) for two consecutive errors and (b) for errors that are frequent but not consecutive. 43

3.16 Waveforms for handshaking when the Master writes to the Slave in a Master-Checker mode. (MAX+plusII) 45

3.17 Indicates the required wait signal, INTF_WAIT, for the EPP Write cycles. 47

3.18 Indicates the required wait signal, INTF_WAIT, for the EPP Read cycle. . 47

3.19 Enhanced Parallel Port data write cycle 48

3.20 Waveforms for handshaking when the EPP is writing to the dual CAN node. (MAX+plusII) 49

3.21 The interrupt signal, INTF_INT, indicates to the EPP that there is data available in the CPLD. 49

3.22 Waveforms for handshaking when the EPP is reading from the dual CAN node. (MAX+plusII) 51

3.23 Signals involved when the Master writes an address to the application when it is a sensor or an actuator. (MAX+plusII) 52

4.1 The dual CAN node. 54

4.2 The dual CAN network setup consisting of three nodes, an interface to the EPP port and an interface to the diagnostic PC via the CAN bus. 55

4.3 Timing analyses of the CAN dual node. 57

4.4 Typical performance characteristics of some of the dual CAN node's parameters. 59

A.1 Illustration of the relationship of the recent CAN versions. 81

A.2 Standard and Extended CAN Dataframes. 83

A.3 CAN's bit timing definition. Source: Adapted from Siemens, 1997. 85

A.4 Projected transfer rate versus transmission length. Source: Adapted from Hitex, 2000. 92

B.1 Illustration of the test setup for the dual CAN network, its nodes and diagnostic interface. 104

B.2 The interface circuit with the purpose of protecting the PC. 108

Stellenbosch University http://scholar.sun.ac.za

C.1 An approximation of the interframe spaces of the four CAN circuits on the PCCAN card. 111

D.1 Dual CAN node schematics - Page 1 113

D.2 Dual CAN node schematics - Page 2 114

D.3 Dual CAN node schematics - Page 3 115

D.4 Dual CAN node schematics - Page 4 116

D.5 Dual CAN node Legend 117

D.6 Dual CAN node component side. 118

D.7 Dual CAN node solder side. 119

Stellenbosch University http://scholar.sun.ac.za

List of Tables

1.1 The data transferred frequently between modules on the ADCS. 7

2.1 A data exchange summary of table 1.1. 16

3.1 The Altera and Xilinx CPLD devices suitable for this design. 31

3.2 Utilisation of the EPM7128STC100-15 CPLD. 31

3.3 A summary of the protocol used for communication between the two CAN processors. 39

4.1 Legend for Figure 4.3 58

4.2 Summary for the parameter *Second start to processing done.* 63

4.3 D.C. Characteristics for the dual CAN node in a MC configuration. 63

4.4 The derived approximate power dissipations for the dual and single channel CAN nodes, with only the essential components at baud rates of 125kbit/s and 1Mbit/s respectively. 64

4.5 Estimated absolute maximum current dissipation for the interface electronics. 66

4.6 Time-loading as a percentage of one second. 68

A.1 CAN Dataframe efficiency with bit stuffing *excluded.* 84

A.2 CAN Dataframe efficiency with bit stuffing *included.* 84

A.3 Assignments to the Bit timing registers. 86

A.4 Available CAN devices. 87

B.1 Dataflow summary of the ADCS. 93

B.2 A comparison of some fieldbus protocols. 96

B.3 ICP code analysed. 102

B.4 The estimated power consumption of the dual CAN node's evaluation version. 103

Stellenbosch University http://scholar.sun.ac.za

B.5 Thermal resistances for calculating maximum power of the regulator. . . . 104
B.6 A.C. Characteristics 105

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

Introduction

The advancement in satellite technology and the corresponding growth in space applications, stimulate a growing demand for flexible systems. This demand requires standardising of hardware and software products with the objective of increasing the proliferation of the products. The repeated use of these products decrease the risk of failure.

Perhaps the greatest risk to a spacecraft system, and conversely the greatest challenge to the designer, is having unknown system characteristics. Deciding whether a system has a risk factor that makes the system usable in spaceflight or not, is already a complex task. Furthermore, a decision to use ordinary commercial devices (as was done on SUNSAT I) makes this task even more complex and the risk that much greater. For *SUNSAT I* the decision was largely based on the increased cost and very limited availability of military-grade components, and the lower financial risk of "off the shelf" technologies. The risk of failure was however minimised by ensuring that the satellite and its components would receive thorough integration and environmental tests during and after assembly, and secondly, that redundacy was implemented throughout the satellite.

1.1 The Thesis

1.1.1 The Meaning of the Title

What is the meaning of the title: *"Development of a Fault-Tolerant Bus System suitable for a High-Performance, Embedded, Real-Time Application on SUNSAT's ADCS"*?

Stellenbosch University http://scholar.sun.ac.za

A *fault-tolerant* system can be defined as a system that is capable of operating in the presence of faults, due to the reliability of its redundancy. This behaviour should be an acceptable equivalent to its behaviour in the absence of faults. According to Wertz and Larson [19, Wertz, 1995:604] the *Attitude Determination and Control System* (ADCS), "uses attitude errors measured by sensors to automatically activate torquing devices via an onboard computer or analog electronics and thereby maintain the attitude errors within specified limits" despite the external disturbance torques acting on it during the spacecraft's mission. Normally this onboard (*embedded*) computer includes some kind of communication media between itself and the sensors and actuators, collectively called the *bus system*. The data management of this high performance or high precision system is executed in *real-time* or in other words, information is processed at the time that the events occur.

1.1.2 The Objective of the Thesis

The objective of this thesis is to develop a low failure rate, high reliability communication system based on commercial technology. This system should include a type of redundancy to manage the real-time data of an orientation control system. To accomplish this requirement, modules must be adaptable to different types of ADCS payloads with "standard" interfaces to a bus system. This requires a special flexibility concerning the electrical interfaces, and in addition, a modular design.

The *CAN protocol* (See paragraph 2.1) was approached as the preferred communication media on the basis of recommendations from academics and experienced engineers with strong backgrounds in satellite technology. Another factor contributing to the decision to implement CAN is interoperabililty, since projects on other SUNSAT satellite subsystems are also considering CAN technology. The aim of the thesis was thus to *specifically* investigate if and how the *CAN protocol* can be implemented on the ADCS as a communication media. If it can, the advantages and disadvantages it might have must also be taken into consideration.

1.1.3 The Project Methodology

The layout of the document follows the theoretical and practical steps implemented throughout the study. Firstly, the current ADCS was carefully investigated to ensure an understanding of its data managing details. Special emphasis was placed on the ICP, its software and its peripherals. With the knowledge gained from the above investigation, all the project requirements and specifications were summarised (Chapter 1). Thereafter, the CAN protocol was investigated to determine whether it will comply with the requirements of the current ADCS and the ICP (Chapter 2). A prototype consisting of three nodes was designed and built. The software drivers were also developed (Chapter 3). The prototype was then used to analyse the CAN technology's performance. With the data

Stellenbosch University http://scholar.sun.ac.za

gathered from this study, the performance of the prototype was extrapolated to a system that represents a complete ADCS (Chapter 4). From the results obtained, a conclusion was made on the CAN protocol; the architecture developed; the feasibility of the project; and whether the project conforms to the requirements that were specified (Chapter 5).

1.2 The ADCS on SUNSAT I

The ADCS (Figure 1.1) is only one of a number of sub-systems[1] on SUNSAT I. Figure 1.2 shows a simplified diagram of the ADCS and the two most important sub-systems that communicate with it, i.e. the OnBoard Computer 1 (OBC1) and OnBoard Computer 2 (OBC2).

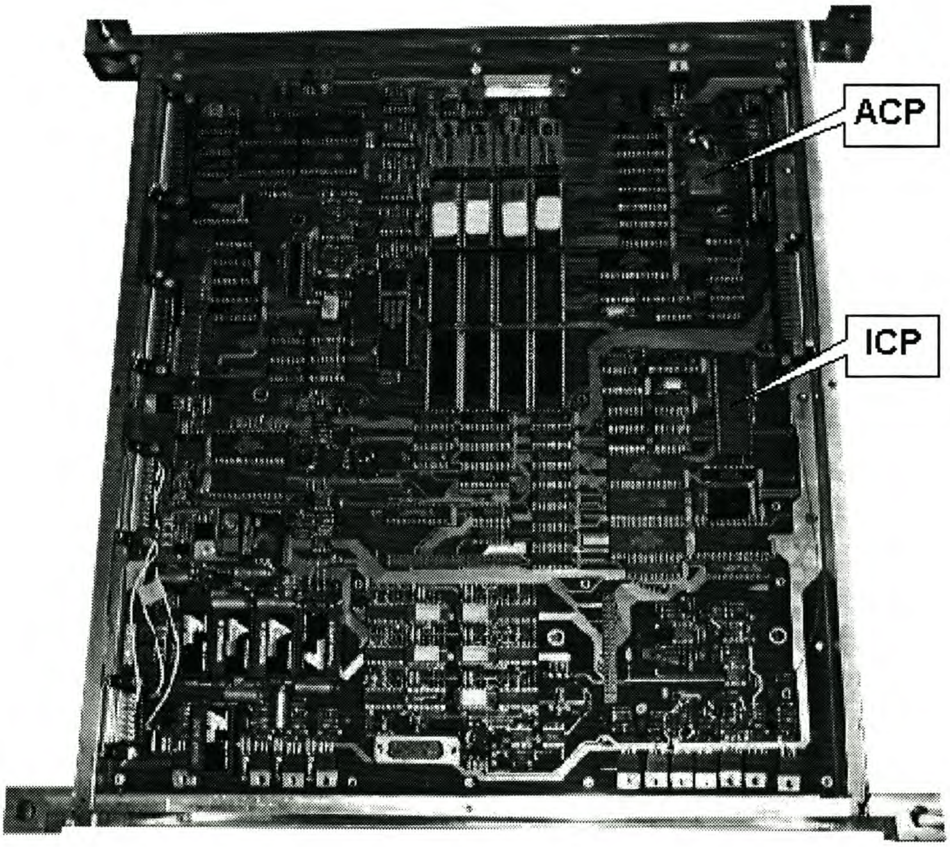


Figure 1.1: The engineering model of the ADCS of SUNSAT 1.

[1] Refer to [27, Müller, 1999] for a systems overview on SUNSAT I.

Stellenbosch University http://scholar.sun.ac.za

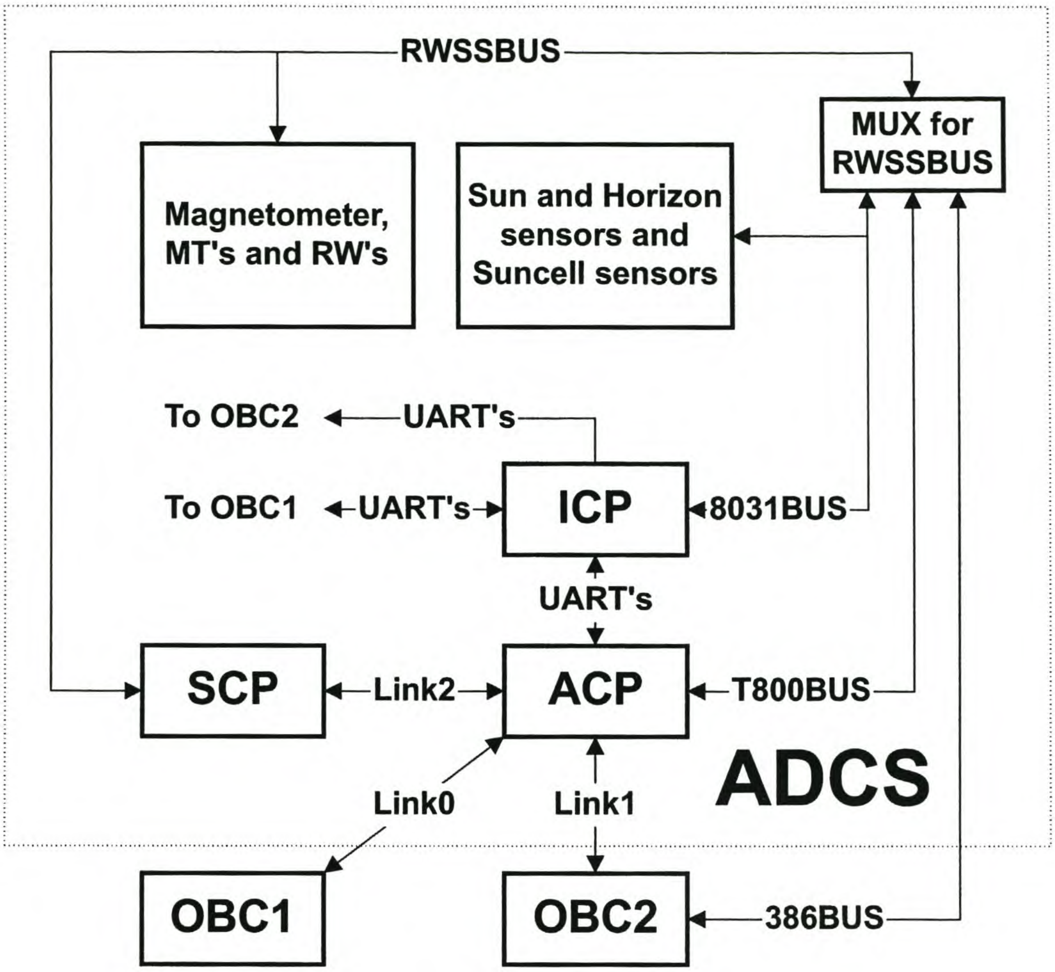


Figure 1.2: A simplified diagram of the two most important sub-systems that interface with the ADCS, as well as the sensors and actuators involved in orientation control.

1.2.1 The ADCS Architecture

The ADCS itself (Figure 1.2) consists of the Attitude Control Processor (ACP) or T800 transputer, the Interface Control Processor (ICP) or 80C52 microcontroller, and the sensors and actuators. The function of the ADCS is to determine the attitude of the satellite (information of the rotation of a spacecraft in a reference coordinate system), and then to control the attitude within an acceptable margin around a desired position (command). The ADCS therefore manages the orientation of the satellite. For the ADCS to fulfil this purpose, input to the ACP concerning the satellite's current orientation is required. From this information, the appropriate reaction (output) can be determined.

Stellenbosch University http://scholar.sun.ac.za

The input to the ACP can come from any one or any combination of the following sensors via the ICP[1]:

- Horizon and Sun Sensors;
- Fluxgate Magnetometer;
- Rough Suncell Sun Sensors; or
- Star Sensor.

The output of the ACP can be targeted to any of the following actuators either directly or via the ICP:

- Magnetotorquers; or
- Reaction Wheels.

The ICP must thus perform the following tasks:

- Initialise the sensors and actuators;
- Gather data from the sensors;
- Send sensor data to any one of the three destinations; ACP, OBC1 or OBC2 (See Figure 1.2);
- Receive commands from the above three sources; and
- Command the actuators.

In short, the ICP manages the flow of data in the integrated architecture of the ADCS. The ACP however, is the heart of the ADCS and interfaces with six different peripherals. Two of the ACP's transputer links are used to communicate with the two OBC's respectively. It's third link is used to communicate with the Star sensor Control Processor (SCP), and the fourth link is used as an external diagnostic interface (which is not indicated in Figure 1.2). The ACP and the ICP communicate via an Universal Asynchronous Receiver/Transmitter (UART) interface, and actuators and sensors can be interfaced via a multiplexed data, address and control bus called the Reaction Wheel Star Sensor Bus (RWSSBUS). The ACP can thus be seen as one node amongst a number of other processor, sensor and actuator nodes in an integrated network of communication routes.

[1] SS data is not managed by the ICP, but by the ACP itself.

Stellenbosch University http://scholar.sun.ac.za

1.2.2 Number of Effective Nodes on the ADCS

One can summarise the number of current ADCS nodes as follows:

- ACP
- OBC1
- OBC2
- Magnetometer
- Magnetotorquers
- Reaction Wheels
- Sun and Horizon Sensors
- Rough Suncell Sensors
- Star Sensor
- Diagnostic port

With the possibility of including a thruster node on future satellites, any new ADCS architecture must provide for a minimum number of eleven nodes.

1.2.3 Number of bytes per node

A detail analysis was performed on the scheduling of messages on the ADCS subsystem. Aspects such as the source and the destination of messages, the number of bytes transferred and the frequency of occurrence of these messages are sumarised in Table B.1. From this table, a new table (Table 1.1) can be extracted indicating only those messages which are classified as messages frequent rather than seldom in occurence. The rest of the information in Table B.1 would be localised to a particular module under a distributed environment, and can be classified as housekeeping for that module.

From the table one can see that a total of 134 data bytes are transferred between *nodes* on the current ADCS. One can however not look only at the number of bytes to be managed, but also at the rate and efficiency at which these bytes are transferred in the current system.

Stellenbosch University http://scholar.sun.ac.za

Table 1.1: The data to be transferred frequently between modules on the ADCS.

Source	*Dest.*	*bytes*	*When (Frequency)*	*Notes*
Reading of MM, HSS's and Suncell Sensors every second.				
ADC	ICP	8	Reading the MM	Read 12 bit ADC values.
HSS's	ICP	6	Reading the HSS's	Read 11 bit HSS values.
ADC16	ICP	16	Reading the RSS's	Read the converted values.
Commanding the RW controllers every second.				
ICP	UART's	8	Commanding RW's	Send speed reference values when RW's are active.
UART's	ICP	16	Commanding RW's	Read data from the RW's. (8bytes*2Msgs)
Commanding the MT's every ten seconds.				
UART	ICP	6	Commanding MT's	Updating MT counters.
Reading the Star Sensor data every second.				
SCP	ACP	2	Star Sensor active	Indicates what the total number of matching stars was. (totnofmatches)
SCP	ACP	72	Star Sensor active	Bytes = totnofmatches* (2vec*3comp*4bytes) where the totnofmatches = [2,3].
		134		

1.2.4 Data Rates and Efficiency

All the UART's on the ADCS are configured as follows, in order to achieve a baud rate of 9600 bit/s:

- 8 bits
- no parity

Stellenbosch University http://scholar.sun.ac.za

- one stop bit
- prescaler of 3
- divider of 8
- external frequency of 3.6864MHz

According to the manufacturers [7, Harris Corporation, 1992:5-29], a double buffering scheme is used on both the receive and transmit sides of the UART's in use on the ADCS. This allows continuous data flow for both the receiption and transmission of data. A customised higher layer protocol is used to transmit data packages via the UART's. Firstly, a message type is sent followed by the length of the package, the data itself and the *End of Message* (EOM) byte. The UART's are accessed with the external data read/write cycles (MOVX-instruction) of the ICP. With an oscillator frequency of 3.6864MHz, the read/write cycles have a duration of 3.21μs each and an instruction cycle 3.3μs long. At the given baud rate it will take a UART 1.04ms to transmit 10 bits of a message. There can thus be several instructions between each IO-cycle to the UART to process the data, preventing an overflow or underflow in the buffers for reception and transmission respectively. With the information given above, one can calculate what the maximum efficiency might be for UART communication. In all, every byte transmitted has 2 bits overhead. Thus for one byte to be transmitted, the efficiency is 80%; while a message with eight data bytes will have an efficiency of 58.2% considering the protocol overhead.

1.2.5 Shortcomings

A number of sensors and actuators cannot be tested as individual modules as if they were connected to a virtual ADCS. This is a direct result of the integrated structure of the ADCS. Futhermore, some of the sensors and actuators cannot even be implemented on a different satellite without being dependent on intelligence from somewhere else or being dependent on a unique interface to the host. Another shortcoming is the fact that the current ADCS does not have any form of error detection capabilities on either the processors or the data between them. This might lead to sporadic and often unwelcomed reboots of the ADCS.

Any system has *single points of failure* unless that system is fully duplicated in itself, or unless a duplication of the system is lying dormant onboard, waiting to be used. In the event of failures, these redundant systems can take over the function of the malfunctioning systems. Such systems take up very expensive space and considerably increase the cost of a particular system. Consequently, the designers of such systems have to take calculated risks in order to minimise the cost and space (and power dissipation as well). The outcome of such a design can therefore *gracefully* degrade to a semi-functional system in the case of failure. However, depending on the system's complexity, these destructive conditions may be recoverable, in which case the problem must be controlled or limited so that one

Stellenbosch University http://scholar.sun.ac.za

can bypass these failures. If this cannot be achieved, the failure will be considered a single point failure.

As one can see from Figure 1.2, there are two possible and very important single points of failure in the architecture of the current ADCS:

- The multiplexer (MUX) to the RWSSBUS;
- The ICP, which gives access to the Horizon and Sun Sensors and Rough Suncell Sensors.

Seeing that the systems that will be isolated due to such failures can be accessed via the Telemetry and Telecommand subsystems of the satellite, these single points of failure are not pure single points of failure. It must be stressed that in order for the already overloaded OBC's to have access to the isolated systems, some of the other satellite subsystems needs to be switched off in order to free some resources. This system reconfiguration will require code uploading to the satellite's OBC's and to the ACP as well. The outcome will be a functional orientation control system, yet a satellite robbed of some of its functionality. Clearly this is not an attractive solution, but under some circumstances one will have no other choice in order to keep the satellite under control.

These shortcomings and single points of failure, when removed or minimised, will not only simplify integration and testing, but will also make the system more reliable and robust.

1.3 Requirements for the Upgrading of the ADCS

In the light of the experience gained during involvement with the ADCS of SUNSAT I, and the knowledge gained from the investigation into the ADCS architecture and data exchange (paragraph 1.2), the following are proposed as the primary set of requirements for the design of the new architecture:

- Reduced complexity of the related wiring harnesses;
- A bus architecture that will ensure higher data transfer rates than that calculated in paragraph 1.2.4;
- A form of error detection capability, which is absent from the current ADCS;
- Removal of possible single points of failure;
- Enhanced diagnostics; and
- Increased modularity.

These requirements were mentioned in the form of shortcomings in paragraph 1.2.5. However, the last two requirements will be elaborated on, seeing that they have not yet been adequately covered in the text. The other points will be covered during the design process.

Stellenbosch University http://scholar.sun.ac.za

1.3.1 Enhanced Diagnostics

As a result of the increased complexity of satellites and satellite systems, diagnostics have become more important in the developement, testing and debugging of these systems. Good diagnostics speed up the development cycle and ensures rapid prototyping. A diagnostic tool therefore had to be developed, which allowed a visual interface of the scheduling of data and possible errors in the system to be implemented. (See Appendix C for details.) This tool would also permit individual nodes to be tested before they were integrated into the system, thus solving the problem of complex debugging and integration procedures of ADCS modules (sensor and actuator nodes).

1.3.2 Increased Modularity

A system in which the sensors and actuators are heavily integrated may be functional with a much lower power consumption. However, issues such as complexity, debugging, testing, single points of failure, upgrading, and expansion make the designer's work difficult. Software can be modified or improved to ease the constraints mentioned above. However, Steyn [18, Steyn, 1995:1-2] stated that: "...the capabilities of existing hardware may be enhanced by making use of more sophisticated control algorithms. It must be stressed [...] that software alone can not cure all the problems of inadequate hardware."

Distributed architectures based on modular design tecniques on the other hand, have disadvantages such as increased costs, component-count, power consumption and space. In contrast to these disadvantages, the distributed architecture brings the designer very attractive advantages such as simplified debugging, easier access, simplified upgrading and well defined, standardised interfaces and drivers. Although modularity is the cause for higher costs during development, it does however reduce cost and time when a single module from a system needs to be replaced, instead of the complete integrated system and its software.

A distributed system can be represented as shown in Figure 1.3. Various processors, actuators and sensors can communicate across the bus to which the nodes (modules) are connected. The number of nodes and the data rate that the system can support are dependent on the type of technology used. Each node contains intelligence to distribute or gather information to or from the application and the bus to which it is connected. A node also carries out system management, which depending on its complexity may include some or many more of the following:

- Checking the status of the application[1];
- Checking the status of the bus to which the node is connected;

[1] Application software should however have no involvement with the system management. It should be concerned only with data which is accessible through mailboxes.

Stellenbosch University http://scholar.sun.ac.za

- Checking the status of the other nodes;
- Establishment of communications;
- Establishment of mailboxes; and/or
- Reception and transmission of messages.

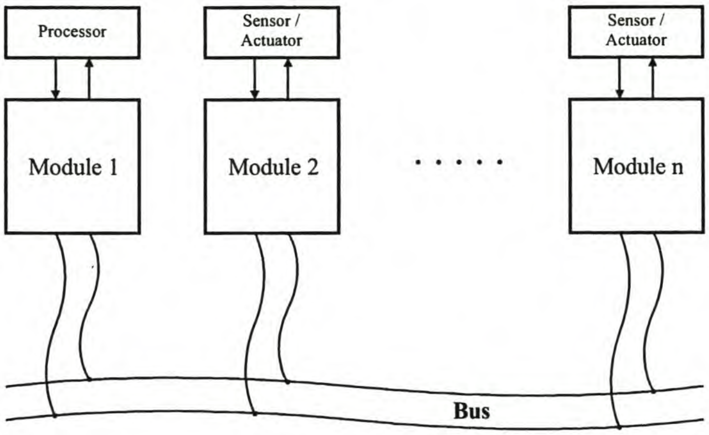


Figure 1.3: A distributed system. Source: Adapted from Lawrenz, 1997.

Modules can communicate in two different ways. The first form of communication is a type of producer communication called "broadcasting". When a node has data ready for transmission, it will broadcast the data. Any node that requires such data can receive it, provided that its mailbox is set up correctly. The second form of communication is a consumer type of communication called "remote transmission requests". When a consumer of information needs a data update, it broadcasts a request. The node responsible for that data will then respond to the request by broadcasting the relevant data.

A further method which can be employed to classify communication is the use of single and multi master typologies. The *single master typology* allows the *master* to sequentially gather and distribute data to or from the nodes, representing a cyclic, more deterministic system. The cyclic system requires more bandwidth in order to accommodate the extra overhead involved in data requests messages. On the other hand, the *multi master typology* permits any node to communicate at any time, corresponding to an event driven system. However, this type of typology lends itself to message collisions. Consequently, some form of arbitration must be put in place in order to control this problem. Furthermore, the event driven system has variable latency times on the messages depending on the load on the network. This makes the prediction of bus traffic a very difficult job. With all the advantages and disadvantages involved using network typologies and protocols, one subsequently has to be extremely careful when choosing the most suitable one.

Stellenbosch University http://scholar.sun.ac.za

ADCS as a Distributed System

Figure 1.4 illustrates how the ADCS would look if it were transformed from the integrated design in Figure 1.2, to a distributed, more modular design. Also notice that provision is made for a secondary processor, thrusters and a data bridge. All the sensor-nodes continuously gather data from the applications of which they are hosts, while the actuators wait for commands from the active processor, relaying them to the client-applications on reception.

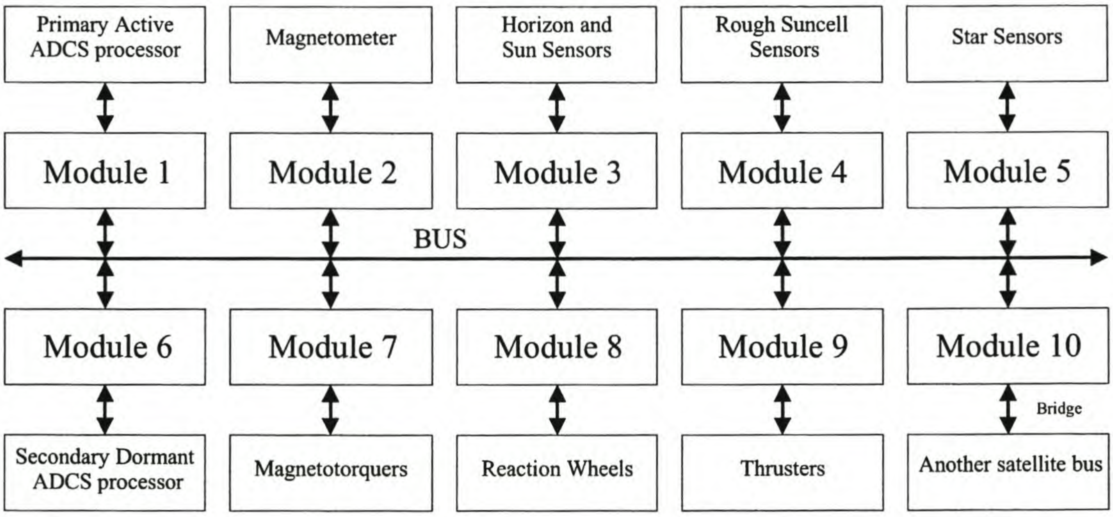


Figure 1.4: The ADCS as a distributed system.

In a single master architecture, the system has a fixed cycle arround which it operates. At the beginning of every cycle the master sends *remote data requests* messages to the sensors and receives the data shortly afterwards. The data is then used in orientation calculations, of which the result will determine how the master will command the actuators. These commands are transmitted in the form of message broadcasts. In a multi-master architecture, the sensors will transmit the new data received from their respective client-applications every cycle, without a request from the processor node. All communication thus takes place autonomously within a broadcasting scheme. Finally, the dormant processor (normally in a powerdown-mode or low power mode) and the data bridge (giving access to other satellite sub-systems), complete the redundancy in the system.

Stellenbosch University http://scholar.sun.ac.za

The requirements under the modularity point can therefore be extended as follows:

- *Increased reliability*[1] *and system robustness.* In case of failure in the communication media or node (module), the rest of the system must continue operation even with reduced functionality - "Graceful degradation". Severe malfunctioning of the entire system must be prevented.

- *Enhanced flexibility.* One must be able to move nodes, remove them or add nodes with ease.

- Each module must be able to *communicate its status.*

- Sensors and actuators with *access to available intelligence* must be put in place where necessary.

- Sensors and actuators are required to be *independent from remote intelligence.*

- The main processor must be *independent of data-manipulation.* Data received must be ready for use in calculations.

- *Standard space interfaces* for ADCS modules are required for increased international interest and rapid prototyping.

- *Testing* of sensors and actuators as *individual modules* as if they were connected to a virtual ADCS[2].

–ooOoo–

[1] A definition of a *reliable* system was derived from work done by Laplante [11, Laplante, 1993:245-246,256]: The "absence of known catastrophic errors; that is, errors that render the system useless; predictable results", and hardware that meets the system requirements by verification through thorough testing.

[2] See paragraph 1.3.1.

Stellenbosch University http://scholar.sun.ac.za

Chapter 2

Why CAN?

2.1 Possible Architectures, Protocols, and Technologies

Networks with increasing intelligent sensors and actuators, demand more standards for their architectures and communication protocols. A number of protocols and architectures could be considered for this project. However, no detailed study was undertaken to establish the most suitable fieldbus[1] for implementation, as this was not the objective of this project[2]. Instead, literature was approached to develop an understanding of how the CAN protocol compares with other fieldbus protocols.[3]

CAN is being used in an increasing number of applications in the automotive world as well as many other industrial applications as a result of its reliability, modularity, simplicity, and robustness. From all the fieldbus protocols investigated, CAN was the preferred choice for its:

- proliferation of technology and development software;
- inexpensive node design;
- flexibility with regard to bit rates and data field content; and
- powerful arbitration and error detection capabilities.

[1] The term "fieldbus" is "applied for inter processor communication as well as for sensor/actuator communication." [13, Lawrenz, 1997:3].

[2] See paragraph 1.1.2

[3] See Appendix B.2 for details.

Stellenbosch University http://scholar.sun.ac.za

Furthermore, CAN gives the designer the following important features[1]: a multi master architecture, a maximum of 32 nodes with standard line drivers, twisted pair media, a maximum communication rate of 1Mbit/s, the possibility of cyclic or event driven communication, and a maximum data length of 8 bytes.

In addition, CAN's communication is based on broadcasting in the sense that the node which has access to the bus broadcasts its information to all nodes at the same time. All nodes receive the information and check for errors. A host (master) node can request data from a client (slave) node by broadcasting what is called a *Remote Request Frame.* The latter then responds immediately to the call by broadcasting the required data. This method of communication can also be used to check whether a suspect node is still functional.

Finally, CAN's extensive error logging makes self-diagnostics a simple task and fault tolerable devices (up to 125kbit/s) are also available where failure in the physical media can be critical. CAN provides a *global acknowledge field* which ensures that at least one node in the network will receive the message or none at all. In this way, a node can detect whether it is still connected to the network. If one node detects an error, it will send out an error frame to all nodes to inform them that the last message must be ignored. This allows for system-wide message consistency.

As can be seen from the summary above, CAN presents attractive features. For example, the requirement of *reduced complexity of wiring harnesses*, can immediately be fulfilled by CAN's twisted pair media. The rest of this chapter will be used to show that CAN can comply with the rest of the requirements as well.

2.1.1 Can CAN manage the ADCS' Data Load?

As the processor-node (Figure 1.4) manages all of the data bytes to be received and transmitted to and from the various sensors and actuators, enough memory and message objects must be available for this purpose. Table 2.1 showes how many bytes every node should be able to store, and the totals indicate how many bytes and message objects the processor-node should be able to store[2]. However, CAN provides only 15 message objects and each message object can store only 8 bytes. It would be extremely advantageous if every message were to have its own message object, as to avoid the re-programing of message objects on the fly, saving CPU time, and leading to a less complex system and better performance.

[1] See Appendix A for CAN specification details.

[2] Notice that this total does not take into account the thrusters, as the number of bytes neccessary for such a node is still to be determined when the thrusters are implemented.

Stellenbosch University http://scholar.sun.ac.za

Table 2.1: A data exchange summary of table 1.1.

Device	*Number of bytes to be stored.*	*Number of CAN Message Objects required.*	*Message direction*
MM	8	1	RX
HSS's	6	1	RX
RSS's	16	2	RX
RW's	8	1	TX
RW's	16	2	RX
MT's	6	1	TX
SS	2	1	TX
SS	72	9	RX
Totals	134	18	

By allocating the required number of message objects to the MM, HSS's, RSS's, RW's and MT's, one can determine that seven message objects remain. The last message object[1] has its own programmable mask for acceptance filtering, allowing a large number of objects to be handled by the system. This allows classes of messages to be received, by masking some of the bits in the mask register. The mask is then AND-ed with the global mask that corresponds to the incoming message [15, Siemens, 1997:134]. Thus, by setting the last message object up to receive all 9 incoming message objects from the SS, and another message object for the outgoing message to the SS, the remaining message objects can be used for other utility messages. One can thus conclude that there are enough message objects in the CAN controller to manage the ADCS' data.

2.1.2 Appearance in Space

The University of Surrey's satellite department has incorporated CAN into their mini-, as well as their microsatellites. There is a CAN-bus operating at 1Mbit/s on the microsatellites and at 380kbit/s on the minisatellites. A standard frame format is used with only 8 bits of the 11 bit identifier implemented. The 8 data bytes are divided into three fields. The first field is one byte long and indicates the *length* of the message. The second field, also one byte long, indicates the *type* of the message and the last field (6 bytes) is the *body* of the message. SUNSAT I has a very similar protocol running between the OBC's, the ACP and the ICP on UART based interfaces. One can deduct from this implementation,

[1] Message object 15 has the highest interrupt priority of all the message objects.

Stellenbosch University http://scholar.sun.ac.za

considering the already low efficiency of the CAN frame[1], that the protocol has a lot of overhead leading to an inefficient data transfer. In contrast, the CAN-bus is exceptionally reliable on Surrey's satellites, and it performs the given tasks inside the specifications. It seldom happens that a message is lost and in the event that this does occur, it is normally due to subsystems which lose synchronism.

For redundancy, a dual CAN-bus (primary and secondary) is implemented per node which includes CAN-bridges to transfer messages between buses. After 5 minutes of zero communication, the system will switch over from the primary bus to the secondary bus. More than one source of intelligence (e.g. the OBC's) can communicate on these buses.

Finally, according to Schofield [31, Schofield, 2000] "The European Space Agency (ESA) is prototyping a CAN controller that will be used in spacecraft control systems."

2.1.3 Targeted CAN Device for this Project

From the above information on fieldbus protocols and other implementations, and the initial software developement on a tutor for the C515C 16-bit 8051-based microprocessor with on-chip CAN by Infineon (Siemens), it was decided that this processor had the neccesary features for the project. The microprocessor-CAN combination can provide the sensors and actuators with some intelligence which can make them operate independently of third party intelligence. The processors will furthermore provide the modules with memory features.

However, this processor had more features than required for such a project and a smaller processor could in fact do the job. Therefore, the C505C microprocessor was chosen to be the nucleus for the design. This processor is an 8-bit 8051-based microprocessor with an on-chip FullCAN CAN controller. The only external device on the CAN-bus's side is a CAN tranceiver. Some semiconductor companies have already indicated that controllers with the transceivers included in the device will be released onto the market soon.

–ooOoo–

[1] See Appendix A for details.

Stellenbosch University http://scholar.sun.ac.za

Chapter 3

Design Overview

3.1 The Proposed Architecture

Considering the discussion in paragraph 1.3.2 and Figure 1.4, Figure 3.1 illustrates the proposed architecture of the ADCS based on the CAN technology.

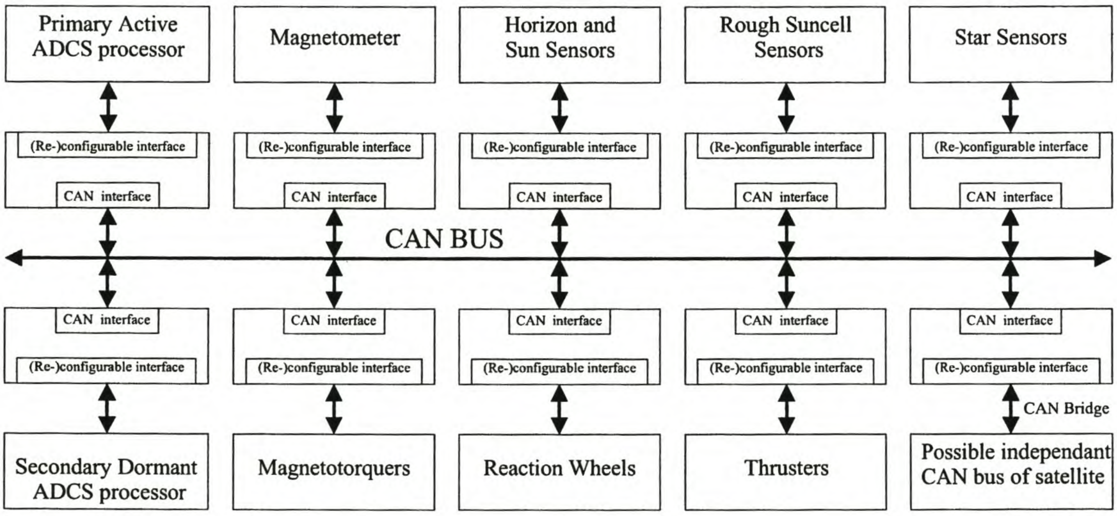


Figure 3.1: This is a concept design of the proposed ADCS architecture which is based on a CAN bus for communication. Note the CAN bridge to interface to another (CAN) bus which operates independently of the CAN bus on the ADCS.

Stellenbosch University http://scholar.sun.ac.za

Since the aim of this project is to move towards modularity, one has to provide for a bridge to convert a custom interface of a sensor, actuator or processor, to an interface which is of a well-defined nature. This bridge is based on the C505C microprocessor (see paragraph 2.1.3).

Considering that this project is a concept design, it was decided that reconfigurable logic will be used. This allows flexibility during the design process and post-design experimentation. Caution must however be exercised when designing for the space environment with reconfigurable logic. The uploading of new software to the satellite is cumbersome and sometimes dangerous because of radiation effects, but it is however cheap considering the replacement cost of a satellite. Notice from the figure that provision is also made for a bridge from the CAN bus on the ADCS to another (CAN) bus on the satellite. This will provide systems such as the Telecommand, Telemetry, OBC1 and OBC2, access to the network of ADCS nodes, and it will ensure redundancy and higher reliability. This communication path will be referred to as the *CAN bridge.*

3.1.1 CAN Bridges

A 16-bit derivative of the Infineon (Siemens) C166 microcontroller family, the C167CS features two on-chip V2.0B FullCAN controllers. These CAN controllers have the same functionality as the 82527 CAN controllers from Philips. Both controllers can operate simultaneously on the same bus (supporting 30 message objects altogether), or on different buses operating at different speeds. The C167CS can serve as a powerful bridge or gateway[1] between two different CAN buses which have different bit rates. Message passing becomes a simple task inside the microcontroller, storing messages either in a FIFO queue or in a buffer with an algorithm allowing higher priority messages to pass first.

TwinCAN[2] is another type of CAN technology that also provides an analysing mode. This mode allows the device to seek for the right bit rate by observing the messages that traverse the CAN bus. An analysing feature like this, can be extremely helpful as an analysing (diagnostic) tool for a CAN network.

[1] "Gateway" is a term used by Wolf and Koller [20, Wolf & Koller, 1998] in an article presented at the 5th International CAN Conference.

[2] A number of applications for this technology, were presented by Barrenscheen at the 5th International CAN Conference [3, Barrenscheen, 1998].

Stellenbosch University http://scholar.sun.ac.za

3.2 Redundancy

Redundancy in the system provides a potential means of recovery from a typical system error, and it is achieved by including redundant hardware. Hardware redundancy can consist of two or more of a certain device. Each device compares its output to its companion's data. If the result is unequal, the system declares itself in error, and the data frame is ignored. If data is continuously in error, a backup system or configuration is switched on. The penalty for hardware redundancy is increased cost, space and power requirements.

A dual CAN bus is proposed for redundancy on the ADCS, thus giving every node two CAN interfaces. In case of failure, or if an error limit is reached, the system can switch from the primary side to the secondary side. Two modes of operating this redundancy on the dual CAN node are considered for the design:

Master-Slave The Master is active, and the Slave is in a dormant state.

Master-Checker The Master and the Slave are both active. The master produces the actual outputs to the application under control, and the Slave shadows the Master. The Slave acts as a checker of the data coming across the CAN buses.

There is only one significant difference in the architecture between the Master-Slave and the Master-Checker configurations: communication between the two microprocessors is only present in the Master-Checker configuration. In order to design a demonstrator that will be able to have the functionality of both the Master-Slave as well as the Master-Checker configurations, one must base the design on the Master-Checker configuration, as it includes both configurations. By disabling the software that deals with the communication between the two processors, one can simulate Master-Slave communication with the help of the Master-Checker hardware.

3.3 The Dual Bus

There are two possible topologies for the dual bus, illustrated by Figure 3.2, a *physical dual bus*, and a *virtual dual bus*. Figure 3.2a demonstrates a physical dual bus - physical because there are two physical networks independent from each other. Figure 3.2b on the other hand demonstrates a virtual dual bus, as there is only one physical network but every node has two interfaces to this network.

Stellenbosch University http://scholar.sun.ac.za

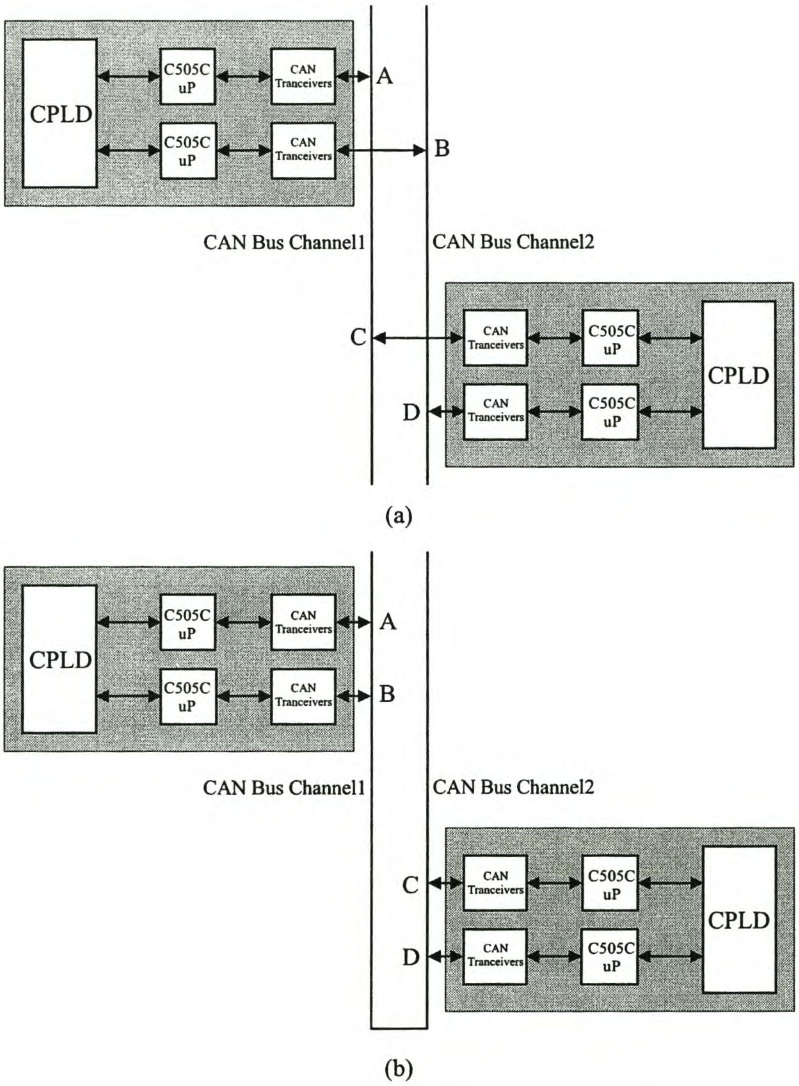


Figure 3.2: Two possible configurations for the dual CAN bus. (a) Physical dual bus. (b) Virtual dual bus.

Stellenbosch University http://scholar.sun.ac.za

3.3.1 Physical Dual Bus

This topology allows communication to continue even if one of the buses fail between two nodes, or one channel per node on the same bus fails (A&C or B&D). Communication on this topology will only fail if both buses fail between two nodes, or one channel per node on two different buses fails (A&D or B&C). The satellite's reliability will be enhanced even further by placing the two buses on separate sides of the satellite body. This topology has some very attractive advantages.

- Message traffic is shared between the two buses.
- True master-slave or master-checker configurations are possible.
- A single message does not need two unique identifiers. (See Virtual dual bus)

3.3.2 Virtual Dual Bus

The advantage of this topology is the fact that the two nodes will never lose communication even if every node in the network loses one channel, unless the physical bus between the nodes fails. The disadvantages of this topology are:

- Bigger delays between Master and the Slave receiving messages.
- True master-slave and master-checker configurations are lost.
- More traffic on single bus. Thus higher bandwidth required.
- Every message must have 2 unique identifiers, leading to unnecessary complexity.

No loss of operational capability, other than redundancy, occurs with a single CAN interface failure in any one of the above redundant bus topologies. The scenario that both the channels of a node fail is not relevant in this discussion, as a node which loses both channels has lost communication whether it is in a physical dual bus or a virtual dual bus. This design will be based on the physical bus topology.

3.4 Criteria for Designing a Dual CAN Node

Only the master should be allowed to communicate with the application at any time. Any potential receiver of data must be warned by means of an interrupt signal. Bidirectional communication must be provided for between the application and master, and between the master and the slave (checker-mode). The Master and the Slave must be capable of swopping modes. It must also be possible to implement a single CAN channel on the dual CAN node. Finally, basic methods of communication between two channels of a dual CAN node must be implemented to manage the different cycle times.

Stellenbosch University http://scholar.sun.ac.za

3.5 Designing the Architecture

The initial design of the architecture was taken through several iterations in order to simplify the design (minimise the number of gates needed to implement the architecture) while keeping it as functional and as sensible as possible. Furthermore, if the design is not carefully reviewed, address-decoding can become very cumbersome and extremely complex while hardware might be functional but not practical. Therefore the design of the architecture was revised on several occasions. Keeping record of the changes helped to avoid making the same mistakes. The proposed design for the dual node interface was reached through a process of six progressive design steps.

3.5.1 Design Step One

Figure 3.3 was the first attempt towards a detailed design. Here, the Master-Checker (/Slave) is determined by the signal $\overline{MASTER}$, that is assumed to be low for the purpose of the explanation. All the interfaces to the two data buses of the respective processors have to be three-state in order to take into acount the instruction fetches from program memory. The Master's three-state latches 1b and 1c are disabled as well as the Slave's bidirectional buffer 2a. The Master, a *bus-master*, then communicates with the Slave and the Application via the bidirectional buffer 1a and message-available signals (which are not indicated here but will be in the form of interrupt signals). The Slave and the Application can thus only communicate with the Master via their respective three-state latches and message-available signals. This topology uses 32 three-state buffers and 48 three-state latches to be implemented in an eight bit system. Address-decoding for such a design is extremely complex and some repetitive hardware can be simplified. It is thus logical to take this concept one step forward.

3.5.2 Design Step Two

The next attempt was to remove both the Master's and the Slave's bidirectional buffers (Figure 3.4). The Master is still the bus-master, but instead of disabling bidirectional buffers, the Master now makes it's bidirectional latches transparent yet retains the three-state capability to prevent bus-conflict. The latches in the opposite channel then serve as the message registers. When the $\overline{MASTER}$ signal is high, the Slave's latches will be transparent and the Master's latches serve as registers. This topology requires 48 three-state latches. However the address-decoding is complex as it must take into account which channel is master, which register is being accessed, and whether the processor is executing a read or write cycle. Subsequently, a third attempt was made.

Stellenbosch University http://scholar.sun.ac.za

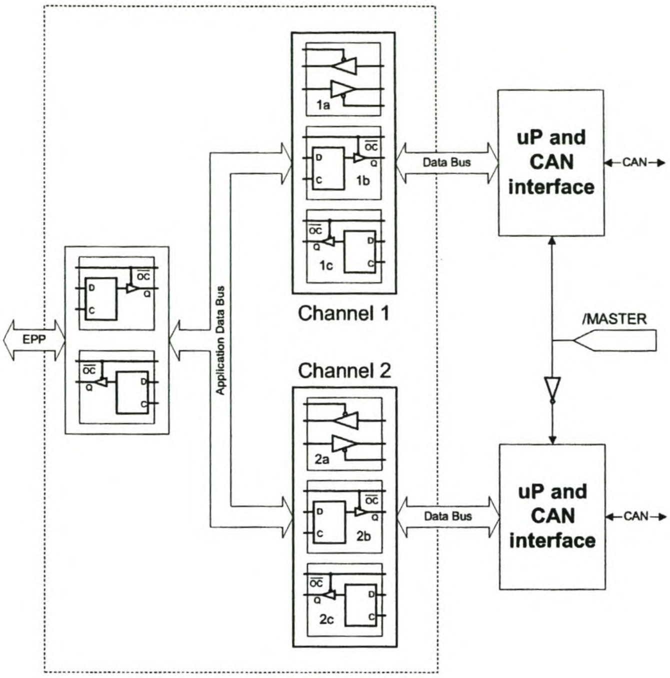


Figure 3.3: Step one in the design of a possible interface between the dual network and the application.

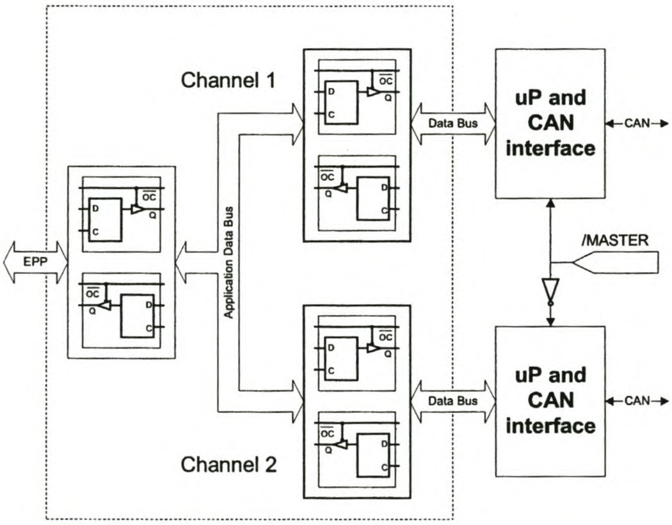


Figure 3.4: Step two in the design of a possible interface between the dual network and the application.

Stellenbosch University http://scholar.sun.ac.za

3.5.3 Design Step Three

For the following attempt (Figure 3.5) the latches on the Master's and the Slave's incoming data were removed. Reading is done in a similar fashion as before, but writing becomes more complex. The incoming data of the Slave cannot be stored by the Slave's interface but must be stored by the Master's interface. For the Slave to read the data, it has to become bus-master for a short while. Thus, the Master does not remain bus-master as required by the criteria for this design. Also, the handshaking overhead created by the slave negotiating for bus access, adds a large amount of undesirable complexity to the system. For the Master to write to the application, the buffer and the latch at the Master's interface have to become transparent. This design not only requires 32 three-state latches and 16 three-state buffers to implement an eight bit system, but the complex address decoding increases the component count even more.

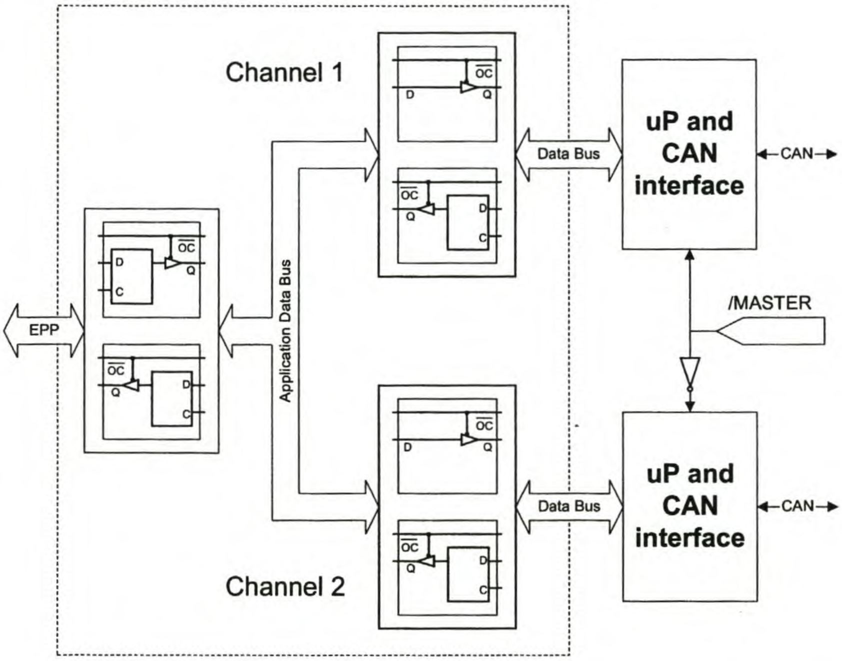


Figure 3.5: Step three in the design of a possible interface between the dual network and the application.

3.5.4 Design Step Four

The next attempt (Figure 3.6) has a similar topology and the same results and conclusions as design step three, but it was considered for control to complement the previous step. Instead of having the data stored by the Master's interface, the data is stored by the

Stellenbosch University http://scholar.sun.ac.za

Slave's interface during the Master's write-cycle and vice versa. When the Slave wants to write data to the Master, the Slave has to become the bus-master, violating the design criteria as before.

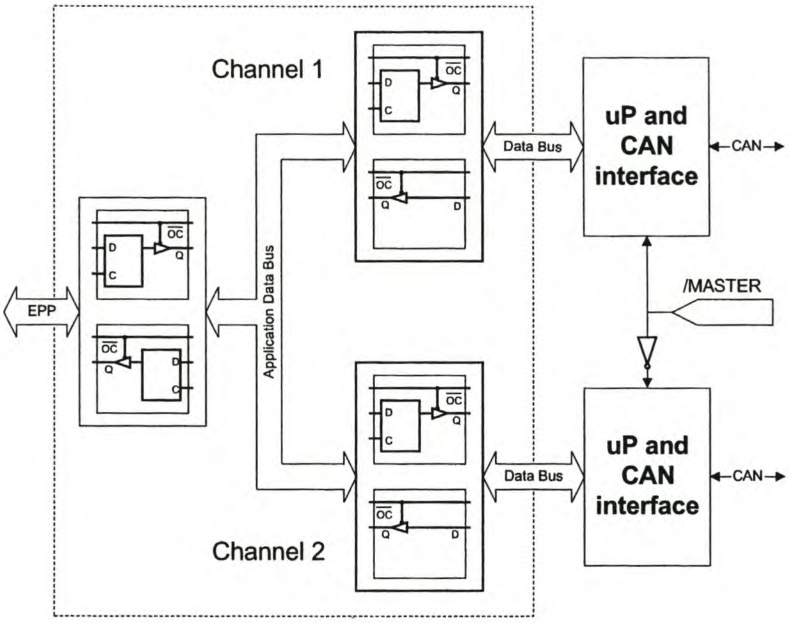


Figure 3.6: Step four in the design of a possible interface between the dual network and the application.

3.5.5 Design Step Five

The fifth attempt (Figure 3.7) simplifies the address-decoding and ensures that the Master remains the bus-master of the application bus by inhibiting the Slave's communication with the application bus. Handshaking thus also becomes a matter of reading from or writing to an external register where the RD and WR signals from the processor act as the acknowledge and message-available signals respectively. Address-decoding is done by decoding the address used by the MOVX-instruction. To implement this topology in an eight bit system will require 32 three-state latches and 32 three-state buffers. This design can be implemented with only a few discrete components and devices such as Motorola's 74HC646, an *Octal Three-state Bus Tranceiver and D Flip-Flops*[1], can be extremely useful in such an application [14, Motorola, 1988:5-498].

[1] Data can be routed real-time from its inputs or flip-flops to the outputs. Each bus has its own set of flip-flops allowing simultaneous storage to their respective flip-flops at any time. The data of the flip-flops can also be routed simultaneously to the outputs at any time.

Stellenbosch University http://scholar.sun.ac.za

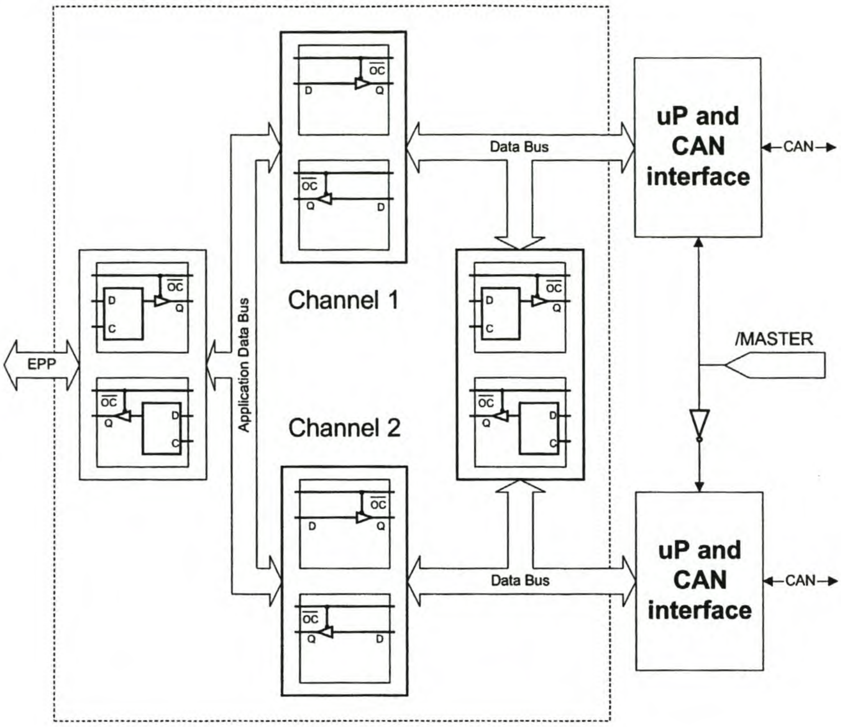


Figure 3.7: Step five in the design of a possible interface between the dual network and the application.

3.5.6 Design Step Six

The last concept topology (Figure 3.8) comes as a result of the aim to make use of a reconfigurable hardware device such as a CPLD (Complex Programmable Logic Device) or FPGA (Field Programmable Gate Array) to build a demonstrator. Implementing the bus as a point-to-point configuration with the help of multiplexers instead of a multi-drop bus implementing three-states, is a better analogy of how a topology like Figure 3.7 will be implemented in a reconfigurable device.

3.6 Application Interfaces and Handshaking of the Dual CAN Node

The interface and handshaking to the application, be it sensor, actuator or processor, will be based on the Enhanced Parallel Port (EPP) standard as used on PC's. (Refer to [29, Peacock] for details on interfacing the EPP.) This standard, can be used to develop a clear and unambiguous inter-connection and interface to the dual CAN node.

Stellenbosch University http://scholar.sun.ac.za

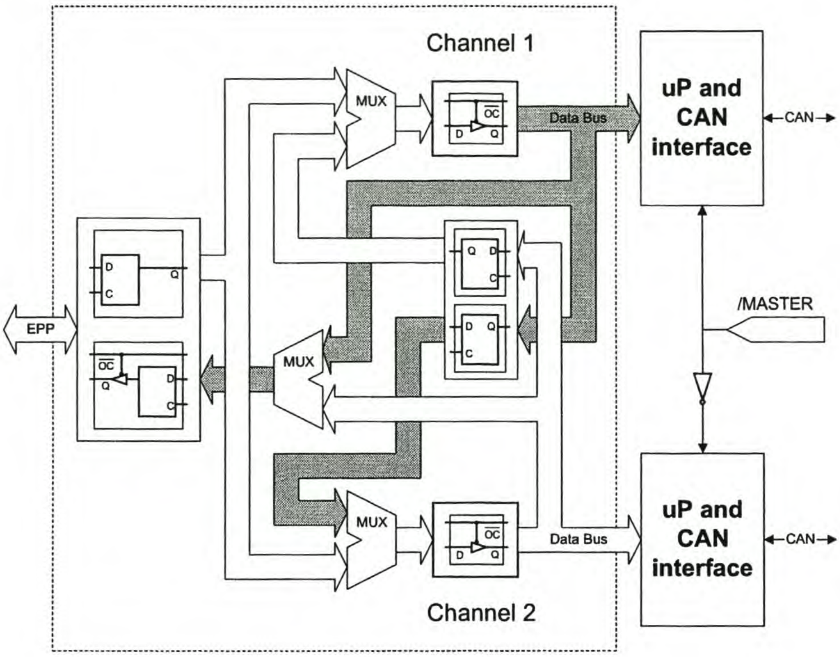


Figure 3.8: Step six in the design of a possible interface between the dual network and the application.

3.6.1 Registers as Interfaces

As data inherently has to travel accross two interfaces to have a data exchange between the dual CAN node and the application, one has to incorporate some mechanism in order to manage these interfaces. Furthermore, the two parties communicating across these interfaces often have different I/O cycle times and/or a phase difference in their communication. Therefore, when the dual CAN node performs an output instruction (I/O write cycle) the data on its bus must be stored in a temporary storage place. Similarly, when an input instruction (I/O read cycle) is executed, the temporary storage place must gate its data onto the data bus lines of the dual CAN node. A similar scenario applies to the application side of the interface. This temporary storage functions very much like a type of Dual Port RAM. The two ports (interfaces) of the register can thus be accessed from two sides independently, and with different access times. Communication between the two controllers of the redundant system can be set-up in a similar way. Each register is given its own address that is used for address-decoding. External I/O cycles are used to transfer data.[1]

[1] Literature significant to this topic can be found in Intel's Embedded Controller Applications Handbook [9, Intel, 1994]. The issues of designing a mailbox memory for two 80C31 microcontrollers using an EPLD device is investigated. As the C505C is an 8051 derivative, this concept design can be very helpful.

Stellenbosch University http://scholar.sun.ac.za

3.7 Hardware

Figure 3.9 is a block diagram of the dual CAN node, indicating the main signals to and from the dual CAN node, the two CAN controllers and their transceivers, the memory, address latches and the CPLD-application interface. From this diagram the blocks can be filled in with selected devices.

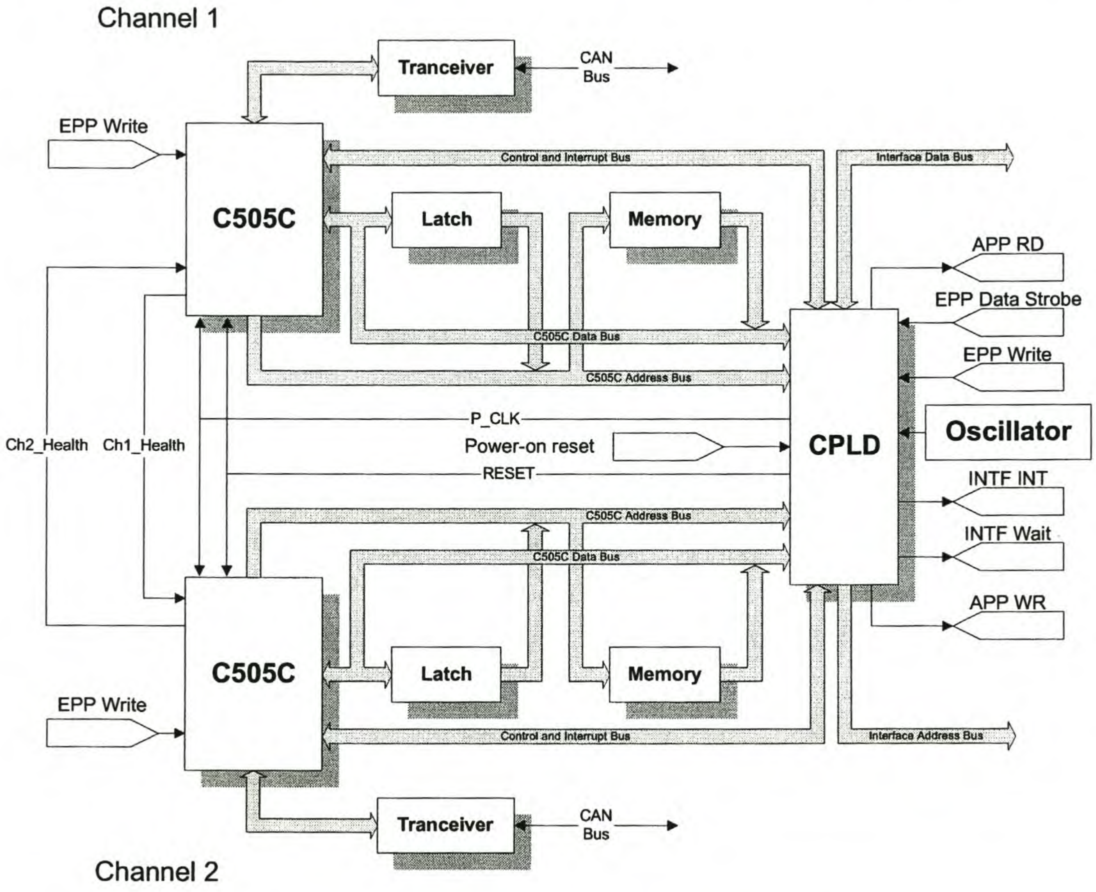


Figure 3.9: A block diagram indicating the main components and signals of the dual CAN node.

3.7.1 Choosing a Reconfigurable Device

From Figure 3.8 one can determine that the interface between the application and the two channels need 4 registers which convert to 32 logic cells in an eight bit system. A further three data buses have to be accommodated, which means another 24 logic cells,

Stellenbosch University http://scholar.sun.ac.za

as every bidirectional and output I/O pin used translates to a logic cell. Three interrupt signals receive 3 more logic cells. Control signals and an address bus to the application which translates to 12 outputs or 12 logic cells must also be provided for. In total, the reconfigurable device must accommodate at least 71 logic cells with an additional 30% for expansion.

A choice between a CPLD- or FPGA-based design had to be made. The FPGA with its large number of logic cells and IO's was very attractive and the speed grade did not matter in this design as propogation delays of even 15ns are negligible. It does however require an external serial configuration Electrical Erasable Programmable Read Only Memory (EEPROM) and an additional clock from the processor, for transferring the data to the FPGA. The CPLD on the other hand does not require external devices or signals to be configured, as it is an EEPROM-based device which can be programmed in-system. High speed devices were available, but as before, propogation delays were negligible. Even though CPLD's have a tendency to be over-consumers of power, it was chosen to be the centre of the design for the following reasons:

- This project is not limited by power consumption as it is demonstrating a concept.
- In-system Programmability (ISP) eases developement.
- Integration is simple.
- Smaller devices are available which are sufficient for the work to be done.

There was also a selection between two possible manufacturers of CPLD devices which had equal availability, namely Altera [1, Altera, 1996:191] and Xilinx [21, Xilinx, 1998:3-1]. To simplify the decision, the manufacturers' families, which cover the requirements of the reconfigurable logic for the design, were compared with each other (Table 3.1).

Part of the decision of which device to use was based on the availability of development and route-and-place software for the devices. For both these devices, the required software was available. Therefore, another more subtle factor guided the decision, the designers experience in the application of the software. With all of the above in mind, the decision was taken to make use of an Altera device.

After writing and simulating the VHDL code for the project, a summary (Table 3.2) was composed with the help of the report file generated by ALTERA's MAX+plusII compiler. EPM7128STC100-15 was the smallest MAX7000S device which could support the requirements.[1] With less than 10% of the logic cells left, the compiler struggled to fit the configuration into the device.[2]

[1] See Appendix B.3 for the thermal analyses of this device.

[2] By the time the final implementation was simulated, the device could not be exchanged for a larger device. Based on the final conclusion on Master-Slave and Master-Checker configurations, future trends and requirements, and problems that were encountered during this project, such as this one, can be taken into account in a future project.

Stellenbosch University http://scholar.sun.ac.za

Table 3.1: The Altera and Xilinx CPLD devices suitable for this design.

Feature	*Altera*	*Xilinx*	*Units*
Family	MAX 7000	XC9500	
Pin-to-pin Propogation delays	5-20	5-15	ns
Counter frequencies (max.)	178.6	125	MHz
Macro cells	32-256	36-288	
User I/O's	36-164	32-192	
Usable gates	600-5000	800-6400	
5V ISP	Yes	Yes	
Program/Erase cycles	100+	10000	
JTAG support	MAX 7000S devices	All devices	
3.3V and 5.0V capability	Yes	Yes	
Output current (max.)	25	24	mA

Table 3.2: Utilisation of the EPM7128STC100-15 CPLD.

	Utilisation	% Utilized
Total dedicated input pins used.	1/4	25%
Total I/O pins used. (27 input, 10 output, 24 bidirectional and 3 reserved pins.)	64/80	80%
Total logic cells used.	117/128	91%

3.7.2 Program Memory Size

The C505C processor has 16 address lines allowing a total of 64kB of program memory to be accessed. Although it is very convenient to implement a 64kB memory device, it is unnecessary. One can make a very good estimation of how much memory will be sufficient for the purposes of the project by doing some calculations (See Appendix B.4 for details.) on the original ICP code as well as an abstract from a typical C-program. It was calculated from the C-program that instructions are implemented with an average of 2.4 bytes/instruction and 1.6 cycles/instruction. Furthermore, the worst case average number of times instructions are repeated in the ICP code was calculated to be 97.2 times in a second.

Operating the microprocessor with a maximum oscillator frequency of 20 MHz allows a maximum number of instructions to be executed per second. Each machine cycle consists of 6 clock periods so that one machine cycle is 0.3μs long. Postulating that all the

Stellenbosch University http://scholar.sun.ac.za

functions of the processor will be completed within 30% of each ADCS cycle (1 second), one can estimate with the help of the information above, the size of the program memory that should be sufficient for the project.

$$Code\ size = \frac{(30\%\ of\ one\ second) \times (Bytes/instruction)}{(Repetition\ of\ instructions) \times (Seconds/cycle) \times (Cycles/instruction)} \tag{3.1}$$

$$Code\ size = \frac{300ms \times 2.4}{97.2 \times 0.3\mu s \times 1.6} \tag{3.2}$$

This gives a code size of 15kBytes, but for safety, a code size of 32kBytes will be implemented.

3.7.3 Address Decoding

By combining the address bus with the RD and WR control signals from the processor, one can produce a system where the MOVX-instruction can be used to write to a variety of different external registers. This concept is used throughout this project and is used to create even the interrupt signals as well as the wait-signals and acknowledge signals for the application hardware.

3.7.4 Reset Signal

Figure 3.10 illustrates how the power-on reset signal will be generated for the C505C processors and the CPLD. Notice the Schmitt trigger on the CPLD's input pin. The purpose of this gate is to have noise rejection on the capacitor voltage as it moves through the threshold voltage. Also notice the hardware reset in the form of a *normaly-open* switch that shorts out the capacitor's terminals when depressed, allowing a normal power-on reset when the switch is released.

Instead of connecting the RESET signal directly to the processors, it is taken via the CPLD where the power-on reset signal (nSTRT_RST) and the reset signal from the Telecommand (nTCM_RST) are combined and synchronised to the rising edge of the system clock. (See the VHDL code below.)

```
Reset_sinch: process (SYS_CLK, nSTRT_RST, nTCM_RST)
begin
    if SYS_CLK'event and SYS_CLK = '1' then
        RESET <= not (nSTRT_RST and nTCM_RST);
    end if;
end process Reset_sinch;
```

Stellenbosch University http://scholar.sun.ac.za

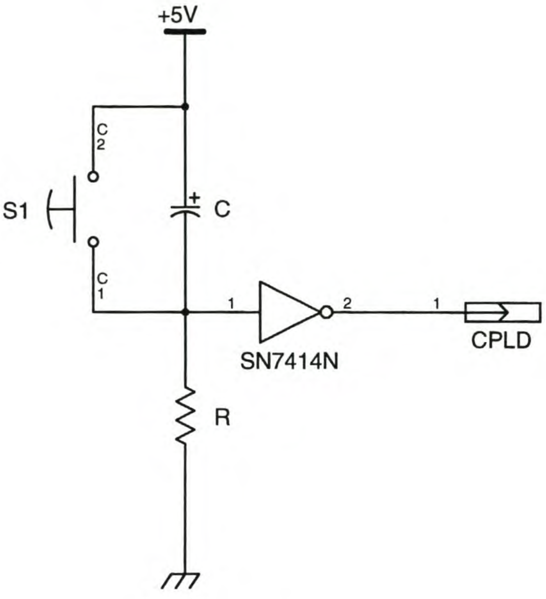


Figure 3.10: The RESET circuit for the C505C processors.

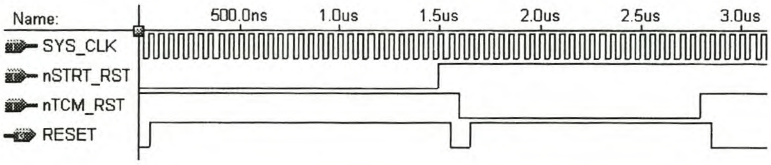


Figure 3.11: Waveforms for the RESET signal. (MAX+plusII)

The C505C's reset signal is active high [15, Siemens, 1997:5-1] and an internal Schmitt trigger is used at the input for noise rejection.[1] Since the reset signal is synchronised internally in the C505C, the RESET signal must be held high for at least two machine cycles (12 oscillator periods) while the oscillator is running. With the oscillator running, the internal reset is executed during the second machine cycle and is repeated every cycle until RESET goes low.

The minimum time required for the power-on reset to be effective is the start-up time for the oscillator plus the two machine cycles as mentioned. Under normal conditions the start-up time for an oscillator is in the order of 10 to 20ms. Since the oscillator's

[1] The C505C processor has an internal pulldown resistor connected between ground and the reset pin allowing only a connection to Vcc via an external capacitor to achieve a power-on reset. This feature is however not used in this application of the C505C processor.

Stellenbosch University http://scholar.sun.ac.za

start-up time is the most dominant value compared to the oscillator frequencies (machine cycles) in the range of 8MHz to 20MHz, it will be used in the calculations. Furthermore, twice the maximum value will be used to calculate what the capacitor and resistor values should be to be sure of a proper reset. The low-level threshold of the external Schmitt trigger is 2.2V [28, National, 1995:2]. With V_O as the Schmitt trigger's threshold voltage, a capacitor value of 10μF and a resistor value of 4k7, one can calculate a power-on reset time of 38.6ms with the help of Equation 3.3. One can calculate a minimum power-on reset time but the value above is sufficient for the purpose of the demonstration model. For the final implementation, the design will not be able to incorporate electrolitic capacitors due to the nature of the component. A military specification tantalum capacitor would rather be used.

$$V_O = V_{CC}e^{\frac{-t}{RC}} \tag{3.3}$$

During configuration of the CPLD, its pins are in a high impedance state. Thus, by inserting a pull-up resistor on the reset pin of the processor, the processor will be placed in a reset state while the CPLD is configuring. The Telecommand reset circuit on the current ADCS does not exist. A reset of the ICP is performed by toggling the power to the device leading to a power-on reset.

3.7.5 CAN Transceivers

The CAN transceiver is responsible for the connection of the CAN node to the physical transmission medium or bus lines. It is designed to apply to the ISO 11898 electrical standard and the $DeviceNet^{TM}$ Specification for the CAN protocol. They are also designed to transmit data with a bit rate of up to 1Mbit/s over a two-wire differential voltage bus.

For this project samples of the PCA82C250 tranceiver [30, Philips, 2000:1] was received and consequently implemented in the design. This transceiver can operate up to a bit rate of 1Mbit/s but it does not have the fault tolerant capability that devices which operate at a maximum bit rate of 125kbit/s do have. A current limiting circuit protects the transmitter output stage against short-circuit to positive and negative supply voltage. Although the power dissipation is increased during this fault condition, this feature will prevent destruction of the transmitter output stage. If the dual CAN node operates within specifications, then implementing the fault tolerant device in future can be done without any complications.

The RS pin (pin 8) of the tranceiver is grounded in order to place the tranceiver in a high speed mode. This mode allows both targeted bit rates. If the RS pin is pulled high, the tranceiver enters a low current standby mode. In this mode, the transmitter is switched off and the receiver is switched to a low current. Because the receiver is slow in standby mode, the first message will be lost when going back to normal operation. It takes the tranceiver 20μs from the time the RS pin is pulled low until it has left the standby mode.

Stellenbosch University http://scholar.sun.ac.za

An unpowered transceiver does not disturb the bus lines. The CANH and CANL lines will be floating and the state of the bus will be recessive in the absence of traffic. In this state, the TXD and RXD pins of the transceiver are in a "don't care" state, as these pins are the signals that go to the CAN controller which controls the RS pin.

3.7.6 Dual CAN Node Power Consumption

The estimated worst case power consumption of the dual CAN node is calculated in Appendix B.5. Taking the *maximum* current dissipation for each device, it was calculated that the dual CAN node will have an estimated absolute maximum power consumption of 1A. With this value, a power supply based on a linear regulator, was designed. This design is available in the same appendix.

3.8 Software Development

3.8.1 Concept Design

The project's software goal was to write one driver for both channels of the dual CAN node. In order to resolve the network, each channel follows an autonomous process to determine its status in the network. Bearing Figure 3.9 in mind, the following flow of information (illustrated in Figure 3.12) is conceptualised: A Write-cycle begins with the transmission of a message to the dual CAN node. On writing each byte of the message to the mailbox (register INTF_IN_REG in the CPLD), the application sends an interrupt request ($\overline{WRITE}$) to the Master (a C505C processor) of the dual CAN node to inform it that a message byte is available. The Master reads the byte from the mailbox and simultaneously acknowledges the message (nCH1_APP_RD). This process continues until the Master has received the complete message. The Master then calculates the CRC of the message by simply adding all the bytes and ignoring the possible carry-out. This CRC is then transmitted to the Slave via the CH1_MSG-register and an interrupt request (nCH1_MSG_RDY). The Slave acknowledges the Master during the reading (nCH2_MSG_RD) of the message from the buffer. The Master and the Slave then transmit their respective messages across the CAN network at approximately the same time. At the respective destinations of the messages, the Slave will transmit the received CRC message to the Master, afterwhich the Master will calculate the CRC of its newly received message and compare it to the CRC received from the Slave. If the CRC's correspond, the Master sends an interrupt request to the application on writing (nCH1_APP_WR) each byte of the message to the buffer (INTF_OUT_REG). The application will in turn perform Read-cycles ($\overline{DSTRB}$) for every interrupt received from the Master, acknowledging on every read cycle.

Stellenbosch University http://scholar.sun.ac.za

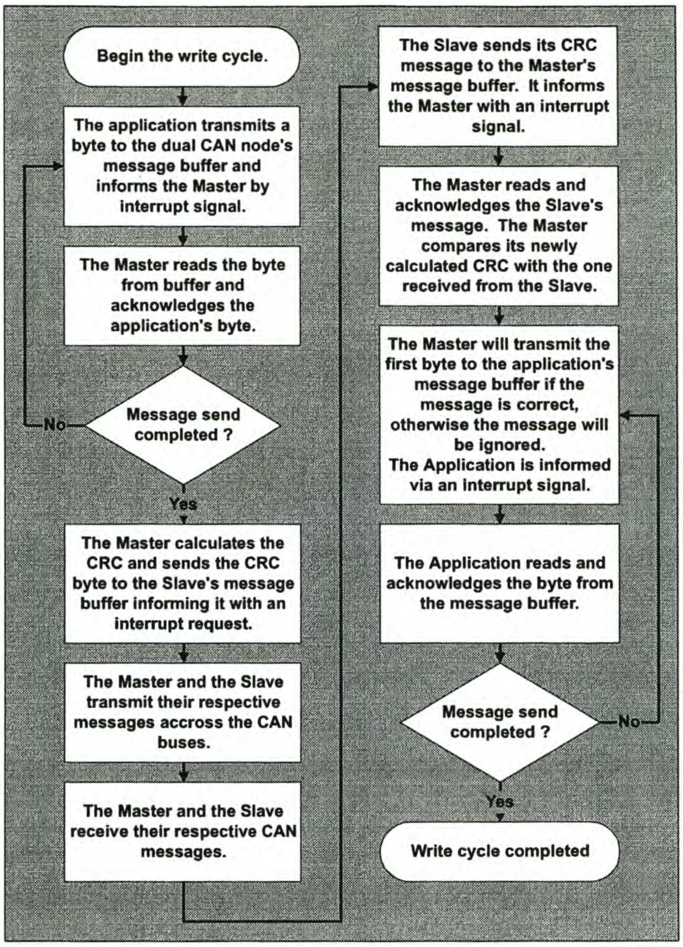

Begin the write cycle.
The application transmits a byte to the dual CAN node's message buffer and informs the Master by interrupt signal.
The Master reads the byte from buffer and acknowledges the application's byte.
Message send completed ?
No
Yes
The Master calculates the CRC and sends the CRC byte to the Slave's message buffer informing it with an interrupt request.
The Master and the Slave transmit their respective messages accross the CAN buses.
The Master and the Slave receive their respective CAN messages.
The Slave sends its CRC message to the Master's message buffer. It informs the Master with an interrupt signal.
The Master reads and acknowledges the Slave's message. The Master compares its newly calculated CRC with the one received from the Slave.
The Master will transmit the first byte to the application's message buffer if the message is correct, otherwise the message will be ignored.
The Application is informed via an interrupt signal.
The Application reads and acknowledges the byte from the message buffer.
Message send completed ?
No
Yes
Write cycle completed

Figure 3.12: Flow diagram of the dual CAN node's Write-cycle.

3.8.2 Initial Development of the CAN Device Drivers

The first development of the CAN software drivers was undertaken with the help of the Siemens (Infineon) C515C 8-bit Starter Kit in conjuction with the PCCAN ISA card by KVASER. (See Appendix C for details.) This was undoubtedly a valuable tool for testing the initial driver software on hardware which was an extremely good working replica of the final system. Furthermore, the C515C and C505C have identical internal CAN controllers.

Stellenbosch University http://scholar.sun.ac.za

3.8.3 Device Drivers and the OSI Model for Software Development

To obtain a higher level of data transfer, a protocol must perform a range of lower level tasks, such as error detection and recovery, synchronisation, etc. The seven layers of the OSI (Open Systems Interconnection) reference model [17, Stallings, 1994:444-450], define a structure or architecture by which these communication tasks and exchange of information can be developed. Using this model, the protocol software is defined in layers, where each layer hides certain properties of the communication channel. Ideally, the layers should be defined so that changes in one layer do not require changes in other layers. To summarise this system a short discription of each layer is given below.

1. **Physical Layer** This layer consists of the mechanical, electrical, functional, and procedural characteristics to access the physical medium - the wires.

2. **Data Link Layer** This layer makes communication accross the physical link reliable. It is responsible for error detection and correction, synchronisation and flow control.

3. **Network Layer** With this layer, the node specifies the destination address of the data and requests certain network facilities such as priority. It is responsible for activating, maintaining and deactivating the communication.

4. **Transport Layer** This layer ensures the delivery of error free data to the destination.

5. **Session Layer** Some nodes demand checkpoints in the messages to allow recovery of faulty messages. This layer provides this function.

6. **Presentaion Layer** This layer defines the message formats.

7. **Application Layer** This layer gives the application access to the OSI environment.

To implement this model a guideline was composed in the form of Figure 3.13. Note that some of the layers are combined to form the device drivers. As a result, selected characteristics of these layers will be carried over to the device drivers. Also note that the application layer in the C505C is in fact a very narrow layer. The two drivers on either side thus virtually become one layer. This application layer is narrow as a result of the lack of data processing taking place in the C505C processor. The C505C can thus be seen as the bridge between the CAN network and the application connected to the dual CAN node.

The Network layer will, in the case of the Master-Checker mode, be responsible for the switching algorithm that determines, with the help of the active high Telecommand signal, MASTER, if a node is the Master or the Slave. Switching from the ill Master to a healthy Slave takes place if the application notices that there are frequent errors in the data, or if there is no communication for some period time. The CAN protocol, according to the CAN specification [4, Bosch, 1991:4], is comprised of all the services and functions of the Physical and Data Link layers.

Stellenbosch University http://scholar.sun.ac.za

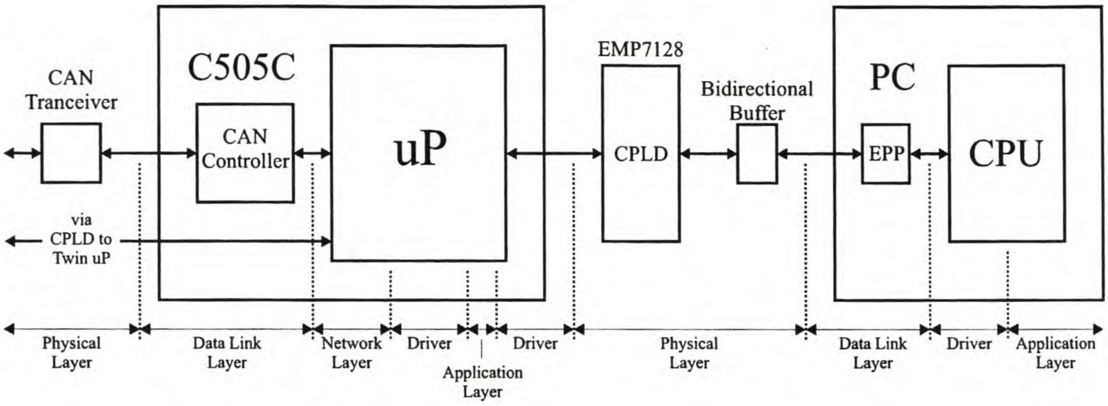


Figure 3.13: An illustration of how the hardware and software are partitioned into the layers of the OSI model. Note that some layers are combined into one for simplicity and functionality.

3.8.4 Interprocessor Communication Protocol

The first step in the design process of a protocol is to have a well-defined problem statement [8, Holtzman, 1991:39]. Thus, for this project it is stated that:

To distinguis between data and error-messages to be transmitted between the processors, both with a maximum length of one byte, a simple protocol has to be implemented.

Because of the level of simplicity of this protocol, the only service it must perform is to code and decode the byte-messages to be transmitted and received accross the physical layer. The internal organisation of this protocol can be structured as follows. Two bytes are transmitted, the first is an ID and the second byte is the data byte. The driver requires four different messages, which implies four ID's (55h, 5Ah, A5h and AAh). If the data to be transmitted is the same as one of the ID's, an escape sequence is to be transmitted consisting of a *key* (FFh) and an additional byte called the *key-number*. With these rules this protocol is both complete and consistent.

By keeping the protocol as simple as descibed above, the designer is relieved from implementing and verifying a fancy protocol. It does also become easier to avoid designing an incomplete set of rules or designing rules that are contradictory. By increasing the number of ID's and key-numbers accordingly, this protocol also becomes easily extendible. The following table (Table 3.3) summarises the protocol.

Stellenbosch University http://scholar.sun.ac.za

Table 3.3: A summary of the protocol used for communication between the two CAN processors.

ID TX-ed	*Data Byte to be TX-ed*	*Actual TX-ed byte*	*Extra byte TX-ed*
55h	Any except one of the ID's	The given byte	-
5Ah	Any except one of the ID's	The given byte	-
A5h	Any except one of the ID's	The given byte	-
AAh	Any except one of the ID's	The given byte	-
One of the above	55h	FFh	01h
One of the above	5Ah	FFh	02h
One of the above	A5h	FFh	03h
One of the above	AAh	FFh	04h
One of the above	FFh	FFh	05h

The efficiency of this protocol is 50% under normal circumstances, but this drops to 33.3% if the message needs encoding. Besides the lack of error detection and correction, low efficiency is the biggest disadvantage of working with such a simple protocol. This is mainly due to the fact that the effective identifier field is being represented by a much larger ID-field than is required. Since one 8-bit port is used for all data transfer between the two processors, be it ID or data, the reduction of the ID-field becomes a complex task. Time restraints prohibited further investigation into this dilemma.

3.8.5 The Real-Time Kernel

This project is an embedded real-time system, and therefore a custom operating system was developed. The kernel responsible for task scheduling will be based on a *Polled loop* system as it is simple to write and debug. "In a polled loop system, a single and repetitive test instruction is used to test a flag that indicates whether or not some event has occured. If the event has not occured, then the polling continues." [11, Laplante, 1993:138] Tasks for which events have occured, are scheduled in a round-robin mode.

Although this scheduler provides an easy method to determine response times, polled loops often have inferior performance when subjected to a burst of events not taken into account during the design. Moreover, polled loops are usually not sufficient when it comes to complex systems and waste processor time if events occur infrequently. It was decided that the polled loop system would be sufficient for the purpose of this project considering the scheduler that was implemented in the ICP. The ADCS has a cycle time of one second, thus every second tasks are scheduled by the ICP to update data. For the new system (see Figure 3.14), the tasks dealing with the transmission and reception of the messages, will be scheduled autonomously until data is updated. Completing all the tasks in a fraction of a second, the scheduler will repeat the updating of data at the start of the new second.

Stellenbosch University http://scholar.sun.ac.za

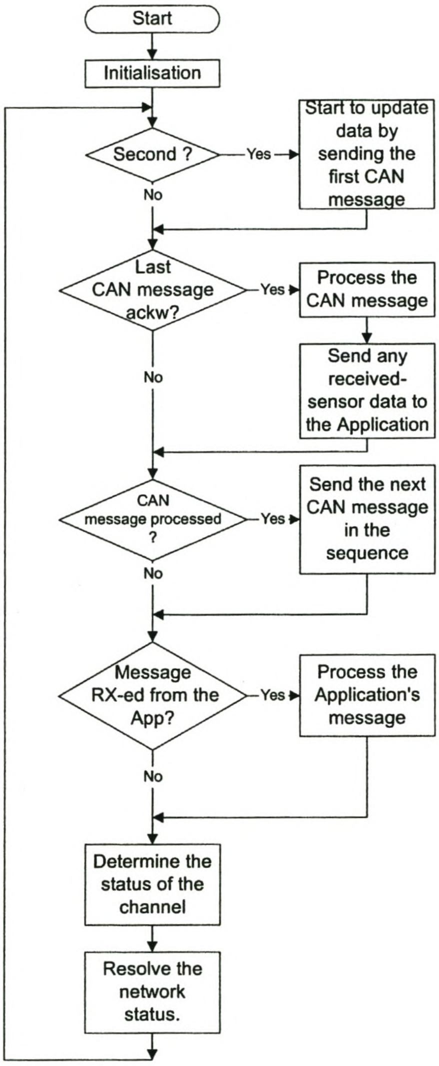

Start
Initialisation
Second ?
Yes
Start to update data by sending the first CAN message
No
Last CAN message ackw?
Yes
Process the CAN message
Send any received-sensor data to the Application
No
CAN message processed ?
Yes
Send the next CAN message in the sequence
No
Message RX-ed from the App?
Yes
Process the Application's message
No
Determine the status of the channel
Resolve the network status.

Figure 3.14: Main event loop of the application software of the dual CAN node.

Stellenbosch University http://scholar.sun.ac.za

3.8.6 Dual Node Switching Algorithm

To demonstrate the dual node concept, one has to demonstrate that the Master and the Slave can switch modes. Two modes in particular will be investigated. In one mode, the Slave becomes the Master when the current Master has some kind of failure. In the other mode the Master becomes the sole CAN node if the Slave fails. Note that mode-swopping should not only depend on the bad data from a particular channel, but also on the good data from the designated channel. Mode swopping should be done locally and it should be autonomous. It should, however, be possible to override it from the groundstation. In cases where some applications are more sensitive to corrupted data than others, care must be taken regarding how the algorithm should switch between channels.

The following are inputs to the switching algorithm:

- The opposite channel's health line. This line is pulled high by a resistor. When power to a processor is switched on, the processor will pull this line low if initialisation of the processor was successful. If not, it will stay high, indicating ill-health to the opposite processor. (See the *Conclusions and Recommendations* chapter in regard to this aspect.)
- The Master/Slave Telecommand signal. This will switch if no communication takes place between the Master and the Application for a predetermined period of time.
- The Slave's CAN error messages. These are transferred from the Slave to the Master via the link between them.
- Time-outs between Master and Slave.
- Differences between the Master and the Slave's CRC data.

Four basic modes of operation can exist:

Mode 1 Channel 1 is the Master and channel 2 is the Slave (Checker).

Mode 2 Channel 2 is the Master and channel 1 is the Slave (Checker).

Mode 3 Channel 1 is the Master and channel 2 is inactive or malfunctioning.

Mode 4 Channel 2 is the Master and channel 1 is inactive or malfunctioning.

During initialisation of the CAN processors, the CAN message objects are configured to be either a message object for the Master consisting of the ID, data length and data matching it, or to be a message object for the Slave ready only for handling CRC messages. When a mode-swop thus takes place, it is necessary to re-initialise the dual CAN node after the swop has taken place.

Stellenbosch University http://scholar.sun.ac.za

3.8.7 CAN Message Priorities

Since there is only one master-processor node in the CAN network which initiates all transfer of messages (data), added to the fact that these messages are transmitted in the same sequential order every second, the use of message priorities becomes irrelevant. It does however become very useful when a higher authority node needs to get its messages across the network without being influenced by traffic. Message priorities do become very important when a system is based on a multi-master topology. This aspect will be considered in the Conclusion and Recommendations chapter.

3.8.8 Health & Safety

The tasks or subroutines which are responsible for finding errors and other single event effects, are dubbed *Health & Safety* in an article by LaBel [26, LaBel, 1996]. These tasks and procedures also performs the necessary actions to solve the problems or to work around them. The code written for the dual CAN node has two such functions. The first function is to monitor the error logging and error counters of the CAN controller, reaching levels of concern. Secondly, messages of the Master and the Slave which do not compare, or a Slave which does not respond, are logged. The action taken for both these cases initiate a mode-swop, thus relieving the faulty channel and continuing communication on the healthy channel. In the case of the Master-Checker configuration, this means switching from Master-Checker to a single Master.

Time-out counters are implemented in both software and hardware. When enabled, a software time-out will occur after 500μs on the Timer1 interrupt for CAN's remote request messages that were not responded to. These remote request messages usually respond within 300μ after transmitting the request. This time-out allows for an exit from the wait-loop in order to transmit the next message.

Hardware time-outs for interprocessor and application communication on the dual CAN node will occur when the message-destination does not respond to a message within a certain period of time. The CPLD will then disable the interrupt signal, which is also the acknowledge signal to the message-source, effectively taking the message-source out of the loop in which it was waiting for the acknowledge. This action does however give the message-source the false impression that the message-destination has successfully read the message. (Comments on this dilemma are made in the Conclusion and Recommendations chapter.) Finally, the last method to make sure that the microprocessor does not get stuck inside a wait-loop is software time-outs; which have a period longer than the hardware time-out period. These time-outs do not give a false impression that messages were acknowledged.

Stellenbosch University http://scholar.sun.ac.za

3.8.9 Five Message Window

To determine what the error status of the Slave is, a *message status window* is implemented in the procedure called *compare_message*. This procedure is also used to determine the validity of the data received across the CAN network. If the Slave is active, the Master shifts the Slave's message-window one to the left and then waits in a loop until the Slave's message arrives. This message can either be a summary of it's errors or the CRC as received across the CAN network. The Slave transmits information also from within its compare_message procedure. The Master then compares the Slave's CRC with the CRC calculated from its own data and if there is a match, the last message in the window is updated as being good and the procedure returns a "good" flag. If, however, the Slave has send an error-status, the last message in the window is updated as being corrupted and a "bad" flag is returned.

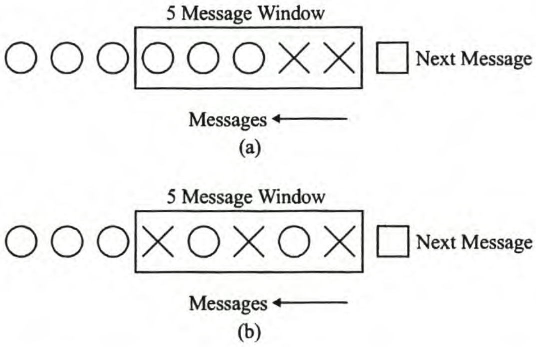


Figure 3.15: The five-message-window error detection (a) for two consecutive errors and (b) for errors that are frequent but not consecutive.

Following this the general status of the Slave is determined by analysing the whole message-window. There are two possible scenarios which will indicate a defective channel (Figure 3.15). As the CAN protocol is reliable, and includes automatic retransmission of bad messages as well as a CRC function build into the hardware, one can assume that data coming across the CAN network has a high integrity. Thus if the Master detects two consecutive errors (Figure 3.15a), it must indicate a serious problem in the system. Another possible situation might be that every alternative message (Figure 3.15b) from the Slave is in error, indicating some frequent error occuring. Since 3 out of the 5 messages have errors, a majority, it also indicates a serious problem. If a time-out has occured while the Master was waiting for the message from the Slave, the Master updates the last message as being bad as it should, and then continues to send the data to the application.

Stellenbosch University http://scholar.sun.ac.za

3.9 Interrupts

Response times for interrupts for the C505C are in the order of 3-9 cycles, depending on the instruction being executed at the moment that the interrupt occurs.

3.9.1 Writing to the Twin Processor's Interrupt Line

The question that one has to ask is:

"Can one write, by means of the MOVX-instruction and the $\overline{WR}$ signal, to another processor's interrupt line which is set to be falling edge triggered, and trigger an interrupt on that processor?"

Firstly, for the processor to recognise a transition, the interrupt signal must be high for at least one machine cycle before the edge, and low for at least one machine cycle after the edge. This is in the light of the fact that the processor only samples the line once every machine cycle. There is, according to the C505C's datasheet, a period of (3CLP - 30)ns[1] between the high-low and the low-high transitions of the $\overline{WR}$ signal of the processor. Taking the maximum processor frequency of 20MHz implies a time of 120ns during which the $\overline{WR}$ signal is low. This is less than one half of a machine cycle of 300ns. As a result, the processor will never recognise an interrupt coming from another C505C processor's $\overline{WR}$ signal, since the interrupt line is sampled at approximately 83% of the machine cycle. Another means of interrupting the processor thus had to be developed.

By feeding both the C505C's address buses and $\overline{RD}$ and $\overline{WR}$ signals to the CPLD and combining them, one can create interrupt signals which will be recognised by the processors. The following code extract, taken from the CPLD's VHDL code, shows how the interrupt signal, nCH1_MSG_INT, is created from the C505C's MOVX-instruction. A new signal, nCH1_MSG_RDY, is created from the $\overline{WR}$ signal and some address decoding on the C505C's address bus. With this signal, the data is written into a message buffer. It simultaneously also activates the nCH1_MSG_INT signal and resets a counter, CH1_ABORT, on the rising edge of the system clock. On the falling edge of nCH1_MSG_INT an interrupt is generated on the other processor which will in return respond by reading from the buffer with a signal, nCH2_MSG_RD, created from it's $\overline{RD}$ signal and address bus decoding. Only at this point does the interrupt signal go inactive. Since the interrupt signal stays active until the processor reads from the buffer from within its ISR, there is enough time for the processor the recognise the interrupt request (Figure 3.16).

[1] CLP is the oscillator's period.

Stellenbosch University http://scholar.sun.ac.za

```
Signal_nCH1_MSG_INT: process (SYS_CLK, RESET, nCH1_MSG_RDY, nCH2_MSG_RD)
                                        --Also ACK for CH1
begin
    if RESET = ’1’ then
        nCH1_MSG_INT <= ’1’;
    else
        if SYS_CLK’event and SYS_CLK = ’1’ then
            if (nCH1_MSG_RDY = ’0’) then
                nCH1_MSG_INT <= ’0’;
                CH1_ABORT <= "00000000";
            elsif (nCH2_MSG_RD = ’0’) or (CH1_ABORT = "11111111") then
                nCH1_MSG_INT <= ’1’;
            else
                CH1_ABORT <= CH1_ABORT + 1;
                                        -- If SYS_CLK = 20MHz, then after
                                        -- 256/20MHz = 12.8us does the
                                        -- interrupt signal reset.
                                        -- If SYS_CLK = 16MHz, then after
                                        -- 256/16MHz = 16us does the
                                        -- interrupt signal reset.
            end if;
        end if;
    end if;
end process Signal_nCH1_MSG_INT;
```

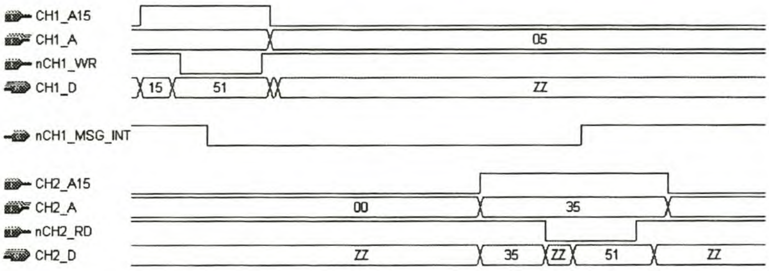


Figure 3.16: Waveforms for handshaking when the Master writes to the Slave in a Master-Checker mode. (MAX+plusII)

The counter that is referred to above, is used to implement the hardware time-outs as described in paragraph 3.8.8. On *every* rising edge, while the clock is running, the counter will unconditionally be incremented. When a write-cycle to the buffer takes place, the counter is reset, and if the value of 256 is reached before the second processor can read from the buffer, the interrupt signal is disabled. This backup is inserted into the system to provide a way for the interrupt signal to be made inactive if the second processor is malfunctioning.

Stellenbosch University http://scholar.sun.ac.za

3.9.2 The EPP Wait Signal

It was decided to make use of a Personal Computer (PC) to emulate the ADCS' ACP in the demonstration model. As interface to the dual CAN node, the Enhanced Parallel Port (EPP) of the PC gives standard I/O Read and Write cycles[1], and typical transfer rates in the order of 500kbit/s to 2Mbit/s can be achieved.[2] These high transfer rates are achieved by allowing the EPP's hardware to generate the handshaking signals instead of the software. The only function remaining for the software is to initiate an EPP cycle, by performing an I/O operation to the relevant EPP register.

The Wait signal (INTF_WAIT) is the acknowledge signal for the EPP Read and Write cycles and is therefore not used when the target of the dual CAN node is an actuator or a sensor. The signal must be generated by the peripheral from where the EPP reads from or writes to. In the case of the dual CAN node this is done in the CPLD process called *Wait_SignalGeneration*.

```
Wait_SignalGeneration: process (SYS_CLK, RESET, nWRITE,
                               nD_STRB, nCH1_APP_RD, nCH2_APP_RD)
    variable Temp : std_logic_vector (3 downto 0);
begin
    Temp := nWRITE & nD_STRB & nCH1_APP_RD & nCH2_APP_RD;
    if RESET = '1' then
        INTF_WAIT <= '0';
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            case Temp is
                when "1011" =>              --Acknowledge a EPP-RD-cycle
                    INTF_WAIT <= '1';
                when "0001" =>              --Acknowledge a EPP-WR-cycle
                    INTF_WAIT <= '1';
                when "0010" =>              --Acknowledge a EPP-WR-cycle
                    INTF_WAIT <= '1';
                when "1111" =>              --Terminates a EPP-WR/RD-cycle
                    INTF_WAIT <= '0';
                when others =>
                    NULL;
            end case;
        end if;
    end if;
end process Wait_SignalGeneration;
```

[1] See [29, Peacock, 2000] for details.

[2] The ECP (Extended Capabilities Port) has even higher transfer rates but the hardware requires a dedicated controller or ECP chip to negotiate a reverse channel. This is difficult to achieve with glue logic or limited intelligence.

Stellenbosch University http://scholar.sun.ac.za

Wait signal for the EPP write cycle

The INTF_WAIT signal is asserted during an EPP write cycle when the EPP signals nWRITE and nD_STRB are both low and the Master has started to read the data. (Falling edge of RD-signal nCH1/2_APP_RD.) The wait signal is de-asserted when both the nWRITE and nD_STRB signals are high again (Figure 3.17).

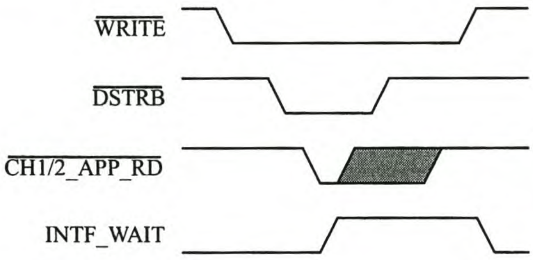

$\overline{\text{WRITE}}$
$\overline{\text{DSTRB}}$
$\overline{\text{CH1/2\_APP\_RD}}$
INTF_WAIT

Figure 3.17: Indicates the required wait signal, INTF_WAIT, for the EPP Write cycles. The rising edge of nCH1/2_APP_RD is valid anywhere in the shaded area. This signal is implemented in the "Wait_SignalGeneration" process.

Wait signal for the EPP read cycle

During the EPP Read cycle the INTF_WAIT signal is asserted by the falling nD_STRB edge, and de-asserted on the rising nD_STRB edge (Figure 3.18). An EPP time-out occurs if the Wait signal is not de-asserted within approximately 10μs after the I/O read or write signals are asserted. This is to prevent the PC from locking up when writing to a port when there is no device connected to the port.

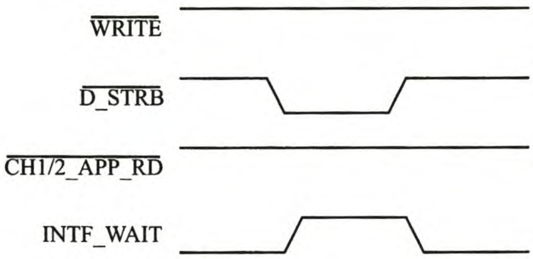

$\overline{\text{WRITE}}$
$\overline{\text{D\_STRB}}$
$\overline{\text{CH1/2\_APP\_RD}}$
INTF_WAIT

Figure 3.18: Indicates the required wait signal, INTF_WAIT, for the EPP Read cycle. This signal is implemented in the "Wait_SignalGeneration" process.

Stellenbosch University http://scholar.sun.ac.za

3.9.3 Application Writing to the Master's Interrupt Line

The interrupt to the Master processor is generated on the falling edge of the active low Write signal (Figure 3.19), indicating that a EPP Write-cycle has started. The data is then stored in the INT_IN_REG register in the CPLD (see VHDL code below) on the falling edge of the active low Data Strobe signal, after which the application waits for an acknowledgement from the dual CAN node. On the rising edge of the Wait signal (see paragraph 3.9.2), indicating the acknowledgement from the Master, the Data Strobe is de-asserted and the Write-cycle terminated. This procedure thus leaves enough time for the interrupt to be registered in the microprocessor since the EPP has to wait for an acknowledgement which comes from within the dual CAN node's ISR before the interrupt signal can be deactivated (Figure 3.20).

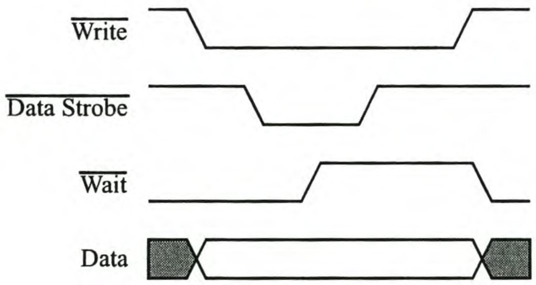


Figure 3.19: Enhanced Parallel Port data write cycle

```
INTF_IN_RegisterControl: process (RESET, SYS_CLK, INTF_D, nWRITE, nD_STRB)
    variable nEN3a : std_logic;
begin
    nEN3a := nWRITE or nD_STRB;
    if RESET = '1' then
        INTF_IN_REG <= "00000000";
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nEN3a = '0' then
                INTF_IN_REG <= INTF_D;
            end if;
        end if;
    end if;
end process INTF_IN_RegisterControl;
```

Stellenbosch University http://scholar.sun.ac.za

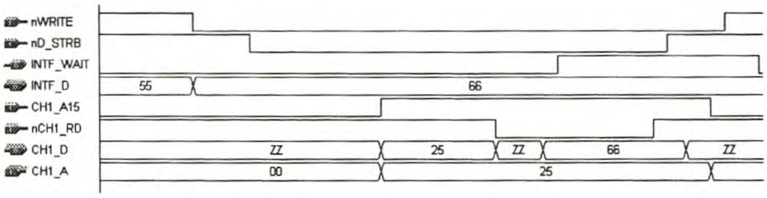


Figure 3.20: Waveforms for handshaking when the EPP is writing to the dual CAN node. (MAX+plusII)

3.9.4 Application Reading from the Dual CAN Node

Since the Data Strobe signal is an EPP ouput signal only, the EPP can read from the dual CAN node but the dual CAN node cannot write to the EPP directly. The dual CAN node can only write to a register (INTF_OUT_REG) in the CPLD from where the EPP can then read the data. (See VHDL code below.)

```
INTF_OUT_RegisterControl: process (SYS_CLK, nCH1_APP_WR,
                                   nCH2_APP_WR, CH1_D, CH2_D)
begin
    if SYS_CLK'event and SYS_CLK = '1' then
        if nCH1_APP_WR = '0' then           --CH1 is the master.
            INTF_OUT_REG <= CH1_D;          --Latch data into the
                                            --application's register.
        elsif nCH2_APP_WR = '0' then        --CH2 is the master
            INTF_OUT_REG <= CH2_D;          --Latch data into the
                                            --application's register.
        end if;
    end if;
end process INTF_OUT_RegisterControl;
```

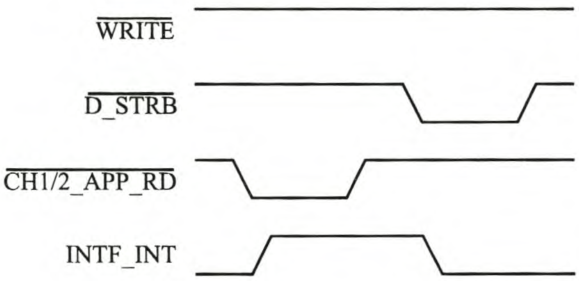

$\overline{\text{WRITE}}$
$\overline{\text{D\_STRB}}$
$\overline{\text{CH1/2\_APP\_RD}}$
INTF_INT

Figure 3.21: The interrupt signal, INTF_INT, indicates to the EPP that there is data available in the CPLD. This signal is implemented in the VHDL process "EPP_INT_Generation".

Stellenbosch University http://scholar.sun.ac.za

On writing the data into the register, the dual CAN node sends an interrupt request to the EPP to indicate that data is available. This interrupt is generated with the help of the WR-signal and address bus of the Master processor. (See the VHDL process called "EPP_INT_Generation" below.) The EPP's interrupt line is an active high interrupt and when the Master writes to the register, signal nCH1_APP_WR or signal nCH2_APP_WR, depending which channel is Master, enables the interrupt signal INTF_INT. (See Figure 3.21 above.) A hardware time-out counter, incremented every clock pulse, is simultaneously reset. When the EPP responds to the request, it will read the data from the register and in doing so reset the interrupt request signal. As the interrupt request signal is also monitored by the Master, this also serves as acknowledgement that the data was read (See Figure 3.22 below.) If a time-out has occurred in the hardware, the interrupt request signal will be reset, giving a false impression to the Master that the data was successfully read by the application. This technique does have an advantage in the fact that it will prevent the Master from waiting in an endless loop for an acknowledgement. (Refer to paragraph 3.8.8 for further details.)

```
EPP_INT_Generation: process (SYS_CLK, RESET, nWRITE, nD_STRB,
                             nCH1_APP_WR, nCH2_APP_WR)
begin
    if RESET = '1' then
        INTF_INT <= '0';
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if ((nCH1_APP_WR = '0') or (nCH2_APP_WR = '0')) then
                INTF_INT <= '1';
                Time_Out <= "00000000";
            elsif (((nD_STRB = '0') and (nWRITE = '1')) or
                                            (Time_Out = "11111111")) then
                INTF_INT <= '0';
            else
                Time_Out <= Time_Out + 1;  -- If SYS_CLK = 20MHz, then after
                                           -- 256/20MHz = 12.8us does the
                                           -- interrupt signal reset.
                                           -- If SYS_CLK = 16MHz, then after
                                           -- 256/16MHz = 16us does the
                                           -- interrupt signal reset.
            end if;
        end if;
    end if;
end process EPP_INT_Generation;
```

Stellenbosch University http://scholar.sun.ac.za

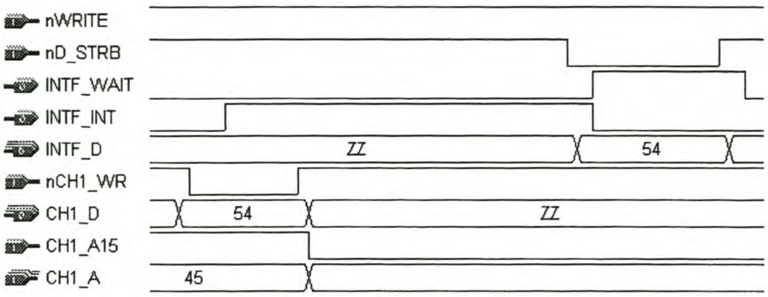


Figure 3.22: Waveforms for handshaking when the EPP is reading from the dual CAN node. (MAX+plusII)

3.9.5 The Master writing an Address to a Sensor/Actuator Peripheral

If the peripheral (application) that is connected to the dual CAN node is a sensor or an actuator, then the dual CAN node might have to write addresses to the peripheral in order to retrieve data from or send data to different registers on the device. This is especially true if the peripheral is dependent on the intelligence from the dual CAN node. The VHDL code used to generate this interface-address and its accompanying signals is shown below, and Figure 3.23 illustrates these signals. Notice that the least significant nibble of the microprocessor's address bus is the address written to the peripheral.

```
InterfaceAddressMux: process (SYS_CLK, nCH1_INTF_ALE,
                              nCH2_INTF_ALE, CH1_A, CH2_A)
begin
    if SYS_CLK'event and SYS_CLK = '1' then
        if nCH1_INTF_ALE = '0' then               --CH1 is Master
            INTF_A <= CH1_A(3 downto 0);          --Write CH1's address
                                                  --to the application.
        elsif nCH2_INTF_ALE = '0' then            --CH2 is Master
            INTF_A <= CH2_A(3 downto 0);          --Write CH2's address
                                                  --to the application.
        end if;
    end if;
end process InterfaceAddressMux;
```

Stellenbosch University http://scholar.sun.ac.za

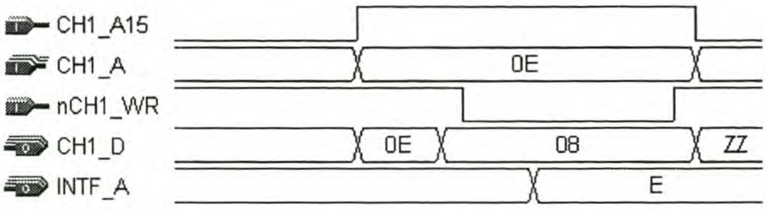


Figure 3.23: Signals involved when the Master writes an address to the application when it is a sensor or an actuator. (MAX+plusII)

3.9.6 CAN Interrupts

On entering the ISR for the CAN controller, all interrupts are disabled. The first task of the processor is to determine what the error status is of the received message. If there is an error in the message the error is logged. If the message has no errors, the processor continues to determine which message object had the event. On finding this message object, the pending interrupt that requested the service, is cleared. The processor then determines if the message was received successfully or transmitted successfully.

If the message was received, an interrupt number is set to indicate which message object contains the data. This number is then used outside the ISR as offset into a CASE-statement wherein the data received is read from the buffers in the CAN controller. Assuming that a second message was to be received before the last one could be processed, the message number will be overwriten by the new message and the previous one will be lost. Provision was thus made to only request new data once the current data is processed.

Unlike the driver that avoids the loss of data, the CAN controller can only indicate the loss of data. Every message object has a bit, NEWDAT, in its Message Control register indicating whether new data has been written into the data buffers of the message object since the bit was last cleared. By allowing the driver to monitor this bit while data is processed, one can determine if a message was lost. Thus in the ISR this NEWDAT flag is also cleared. If the flag is high after processing is completed, it will be an indication that the data was altered during the time the processor read the data. The scenario that the CAN controller can lose data is in fact not relevant since every message object in the system will only receive one message every second. This leaves enough time for processing of data.

If the message was transmitted, an interrupt number is set to indicate which message object transmitted the message succesfully. Furthermore, a flag is set to indicate to the CAN controller that the message may not be transmitted for the moment, and a flag is also set to indicate that the processor is busy with the data. Only when all the interrupts are serviced, should the processor exit from the ISR. If not, the interrupt line from the CAN controller to the processor will stay active, thus inhibiting any other interrupts. On exiting from the ISR all interrupts are enabled again.

Stellenbosch University http://scholar.sun.ac.za

The interrupt number will cause an event in the main loop of the dual CAN node's software. The *Process_Message* procedure called will set a flag to indicate that the message was received, it will stop the time-out timer, calculate the CRC of the received data and then compare it's CRC with the CRC received from the Slave (if the Slave is active). If the *compare_message* procedure returns a "good" flag, the data will be retrieved from the CAN controller's buffers and the valid-data flag will be set indicating that the data can be transmitted to the application (Send_Messages procedure). If the flag returned is "bad", the data is not retrieved from the buffers, and the data-valid flag is cleared. Finally, the message interrupt number is cleared.

–ooOoo–

Stellenbosch University http://scholar.sun.ac.za

Chapter 4

Analyses

The project reached two major milestones at this point: the dual CAN node hardware (Figure 4.1) was completed and the software (C-code and VHDL code) was successfully integrated into the hardware. This was done by making use of the PCCAN interface and diagnostic software throughout to assist in the debugging.

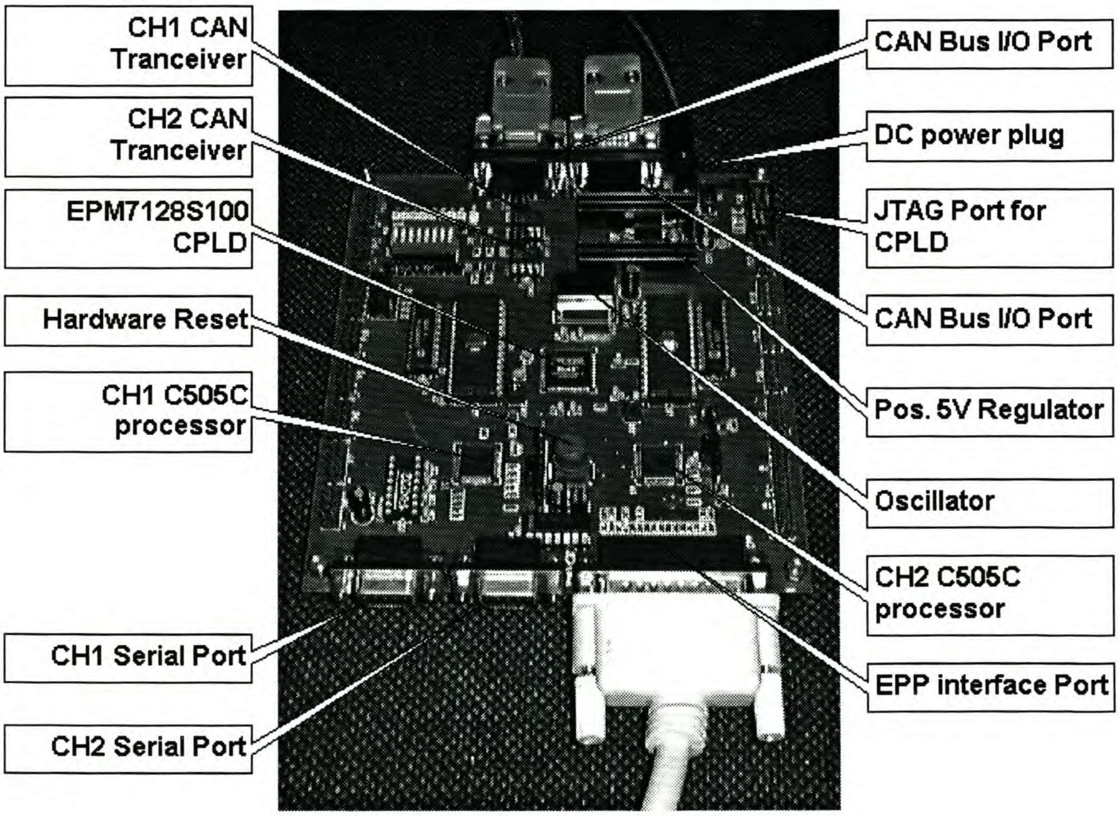


Figure 4.1: The dual CAN node.

Stellenbosch University http://scholar.sun.ac.za

Two things in particular about the software and the hardware can be noted here. Firstly, only one piece of C-code and one piece of VHDL code existed which could be used for all the different nodes (sensors, actuators and processors) as well as both the Master and Checker channels of each node[1]. This made the work of the designer far less complex, as any change in the software would reflect in all of the nodes. It ensured that one could work with consistent responses and results. Secondly, the dual CAN node hardware could support the concept design of both the Master-Slave and Master-Checker configurations.

However, in order to determine the success of the project and whether or not the project has met its objectives, the performance of the system in operation still requires evaluation. Furthermore, in order to make a quantitative comparison between the Master-Checker configuration and the Master-Slave configuration described during the design process, both of the setups had to be investigated. A test and evaluation setup (Figure 4.2, also see Appendix B.6) was made of each configuration on which the analysis was performed. Certain aspects of the design could thus be verify.

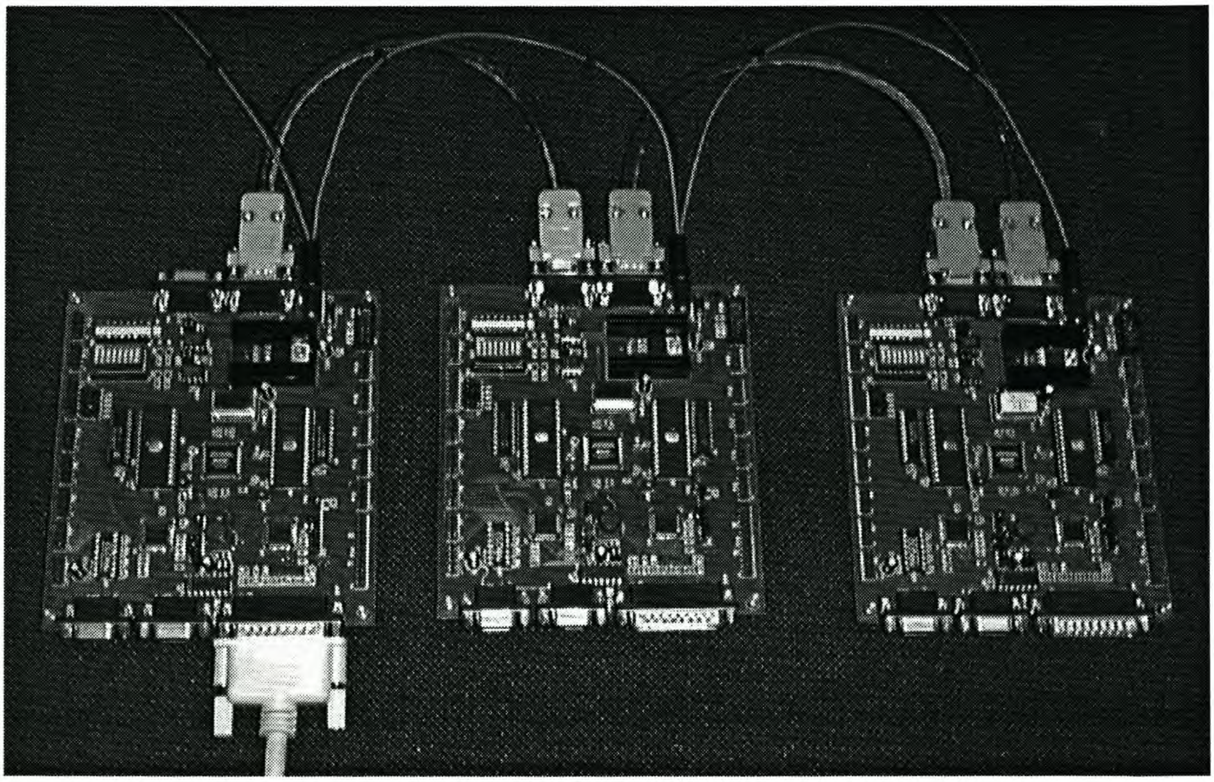


Figure 4.2: The dual CAN network setup consisting of three nodes, an interface to the EPP port and an interface to the diagnostic PC via the CAN bus.

[1] The target application was specified by way of #define compiler directives and the network (Master/Slave) status was resolved with the help of an autonomous process.

Stellenbosch University http://scholar.sun.ac.za

The system analysis methodology that was followed, included an AC and DC analysis and a hardware analyses for both the configurations. Other topics considered include:

- projected power budget;
- interrupt latencies;
- time loading;
- memory loading;
- comments on the C505C CAN Controller-microprocessor;

Each time, an analysis was supported by a short dicussion regarding the important aspects involved with the related topic. In some instances, certain tests in the analyses methodology were not executed because they did not seem necessary or relevant. In these cases reasons are given why this methodology was followed and what impact the decisions made on the analysis.

4.1 AC Analyses

The successfully implemented concept design (paragraph 3.8.1), was analysed to ensure the *smooth* transfer of information amongst nodes. The measured parameters are illustrated in Figure 4.3 with a legend in Table 4.1. Details of these parameters are given by Table B.6 in Appendix B. This data gave one a better understanding of aspects such as

- the time available between two CAN interrupts for processing of data;
- the hardware and software time-outs used to keep a processor from waiting in an endless loop;
- the time left in the ADCS cycle for house-keeping and data processing.

Ultimately, this data was used to make an extrapolation to a system that would be a realistic representation of the actual ADCS.

In Figure 4.3, the system starts off with a reset and initialisation (RESET and can_init() waveforms), completing it at time *'Start of FOREVER'* which indicates the beginning of the repetitive loop. At this point, the ADCS starts it's one second cycle during which all data is to be gathered or distributed. The *'Useful processing during cycle'* waveform indicates the time required by the ACP node to send out all the commands to the actuators and to receive the data from the sensors. Figure 4.4a shows a comparison between the Master-Slave and the Master-Checker configurations for this parameter. Notice that there is hardly a difference between the two configurations. The time available for other utilities or algorithms to be serviced can be expressed as the difference between the ADCS one second cycle and the useful processing time.

Stellenbosch University http://scholar.sun.ac.za

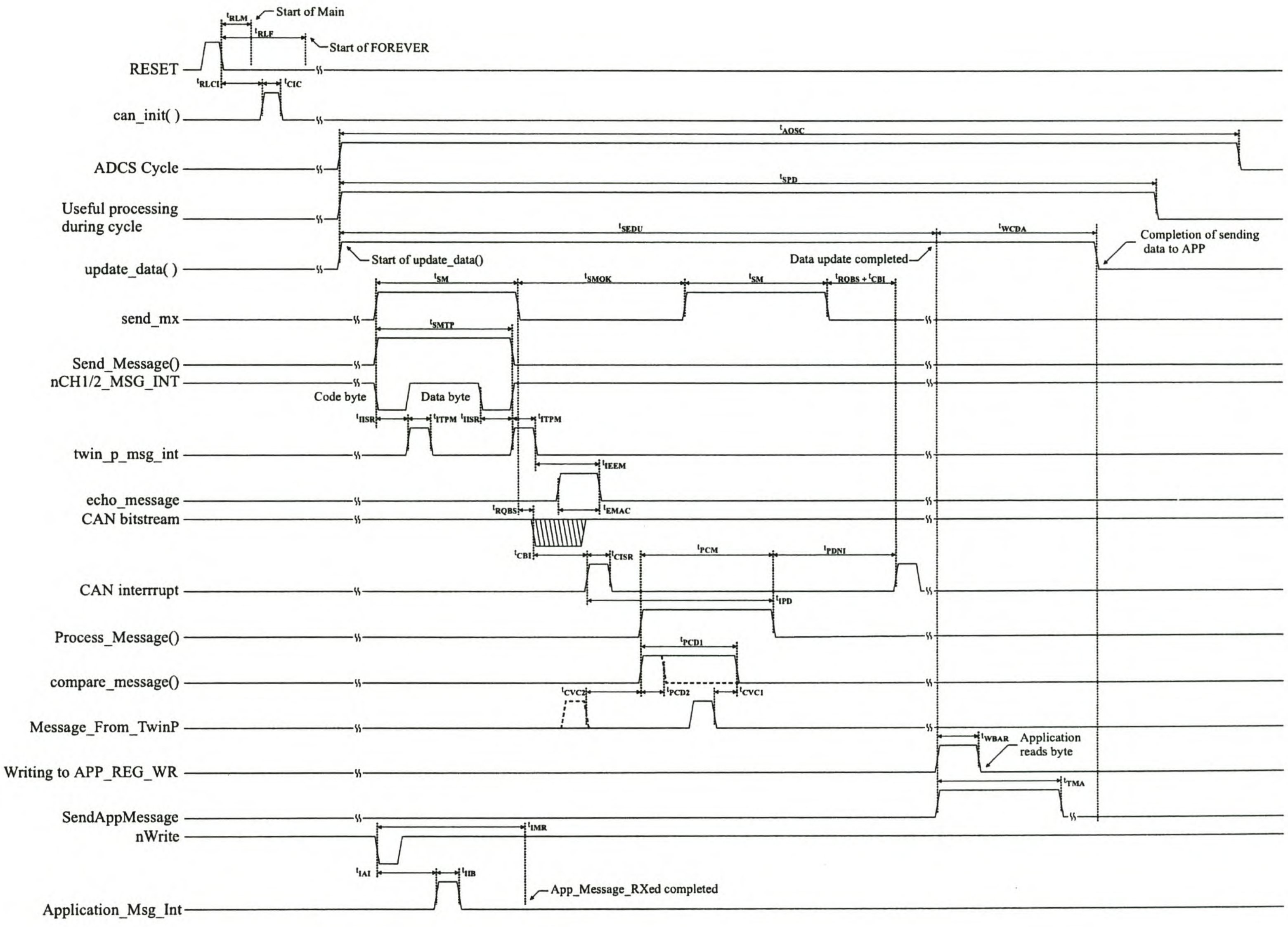


Figure 4.3: Timing analyses of the CAN dual node.

Stellenbosch University http://scholar.sun.ac.za

Table 4.1: Legend for Figure 4.3

Symbol	*Parameter*
t_{RLM}	Reset low to start of the Main-loop
t_{RLF}	Reset low to start of the FOREVER-loop
t_{RLCI}	Reset low to start of CAN init.
t_{CIC}	CAN initialisation cycle
t_{AOSC}	ADCS one second cycle
t_{SPD}	Second start to processing done
t_{SEDU}	Second start to data updated
t_{SM}	Send message across network (send_mx( ))
t_{SMTO}	Send message across network with time-out
t_{SMTP}	Send message to twin processor
t_{IISR}	Interrupt to twin processor
t_{ITHT}	Interrupt to twin processor with HW time-out
t_{ITST}	Interrupt to twin processor with SW time-out
t_{ITPM}	ISR in twin μP receiving data
t_{IEEM}	ISR done to end of echo-message( )
t_{EMAC}	Echo message across CAN network
t_{RQBS}	TXRQ to start of CAN bitstream
t_{CBI}	CAN bitstream start to ISR.[1]
t_{CISR}	CAN ISR receiving a remote or standard frame
t_{PCM}	Processing the CAN messages
t_{IPD}	ISR start to processing done
t_{PDNI}	Processing done to next ISR
t_{PCD1}	Processing start to comparison of M and C message done.[2]
t_{CVC1}	Checker's message valid to end of comparison.[2]
t_{CMTO}	compare_message( ) start to time-out on the Checker's message
t_{PCD2}	Processing start to comparison of M and C message done.[3]
t_{CVC2}	Checker's message valid to start of comparison.[3]
t_{CMCC}	compare_message() start until Checker sends CRC to the Master
t_{SMOK}	Send message to sending next message
t_{WBAR}	Writing a byte to the App. to App. reading it
t_{TMA}	Transmitting a message to the App.
t_{WCDA}	Write collected data to application[4]
t_{IAI}	Interrupt from App. to ISR
t_{IIB}	Inside ISR to receive one byte
t_{IMR}	First interrupt to whole message received (9 bytes)

[1] STD frame with 8 bytes.

[2] Checker's message received after start of compare_message( ).

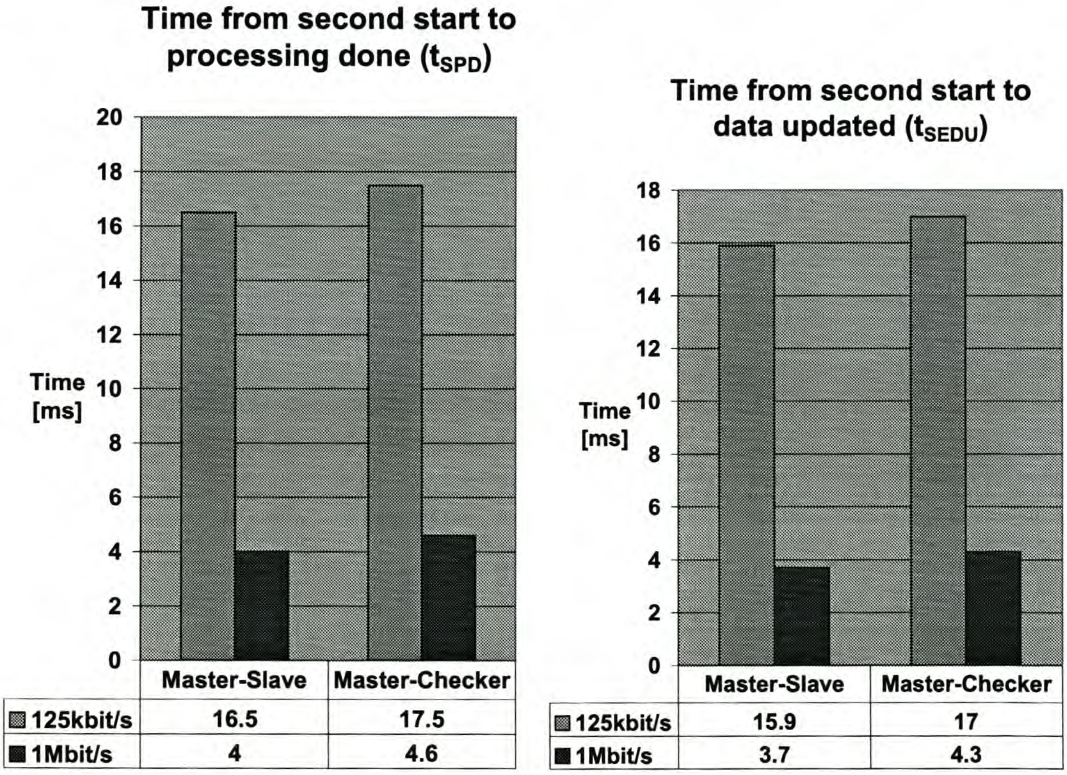
[3] Checker's message received before start of compare_message( ).

[4] This data includes 14 bytes sensor data and one header byte.

Stellenbosch University http://scholar.sun.ac.za

All the data to be received or transmitted across the CAN network is managed between the times labelled *'Start of update_data()'* and *'Data update completed'* in the waveform *'update_data()'*. Again, notice that there is hardly any difference between the two configurations for this parameter illustrated in Figure 4.4b.

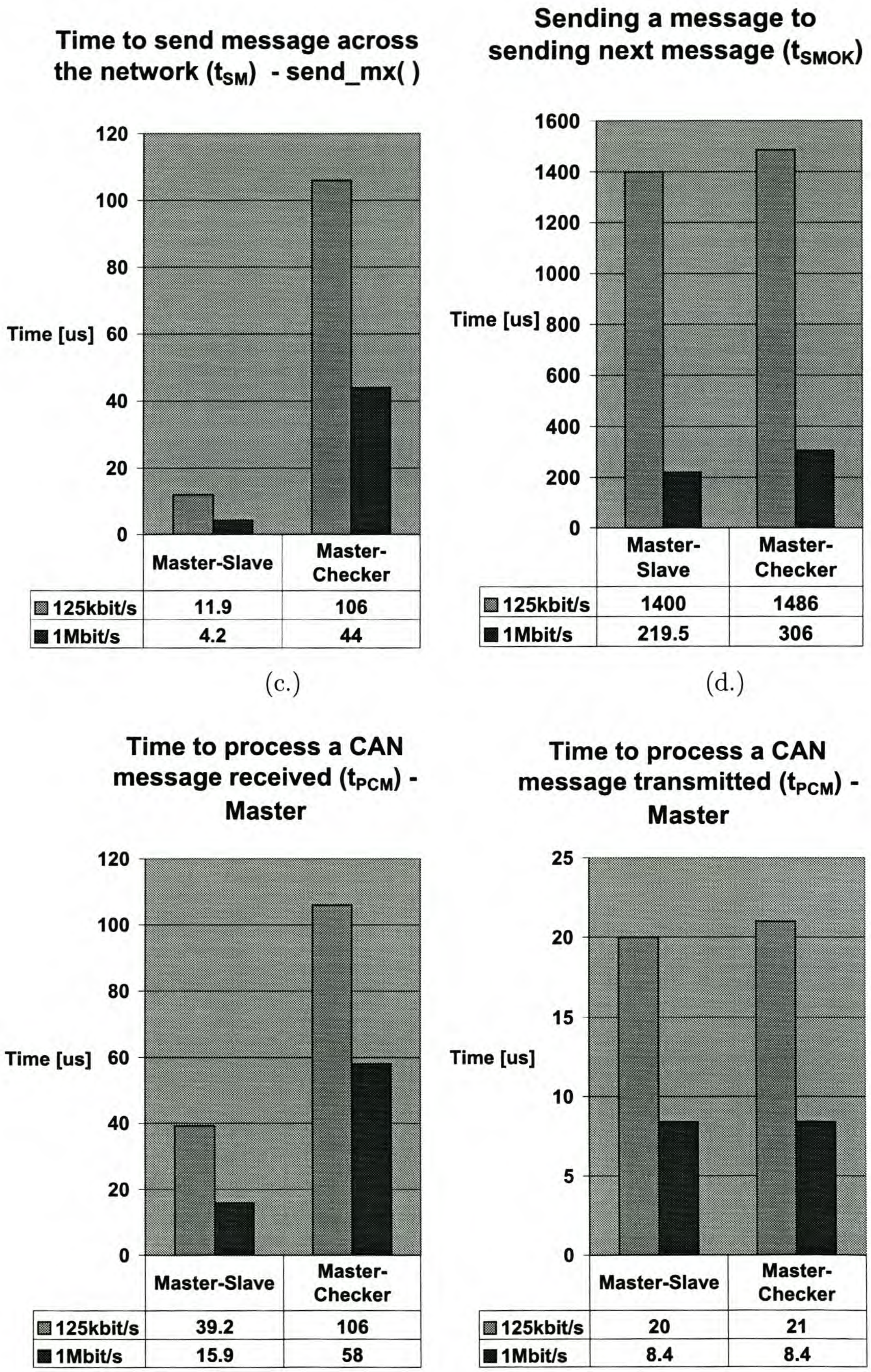
Inside this period of time, the ACP Master transmits either commands to actuators or remote requests to sensors by calling the *'send_mx'* (m = 1,2,....) procedures. The ACP Master contains such procedures for each sensor and actuator. After the successful completion of each procedure, (command transmitted successfully or data received successsfully) the next procedure will be called until all have been serviced. This is where the first noticeable difference between the two configurations can be observed (Figure 4.4c). The extra time required by the Master to communicate with the Checker is responsible for this difference. The reason why this big ratio does not reveal itself in the previous parameter (Figure 4.4b) can be explained with reference to the relatively few times this procedure is called. It can also be explained through the order of the parameter [μs] compared to the order of the other parameter [ms].



(a.) (b.)

Figure 4.4: Typical performance characteristics of some of the dual CAN node's parameters. Note that these parameters were measured under a reduced ADCS topology for test purposes.

Stellenbosch University http://scholar.sun.ac.za



Time to send message across the network (t_{SM}) - send_mx( )

	Master-Slave	Master-Checker
▨ 125kbit/s	11.9	106
■ 1Mbit/s	4.2	44

(c.)

Sending a message to sending next message (t_{SMOK})

	Master-Slave	Master-Checker
▨ 125kbit/s	1400	1486
■ 1Mbit/s	219.5	306

(d.)

Time to process a CAN message received (t_{PCM}) - Master

	Master-Slave	Master-Checker
▨ 125kbit/s	39.2	106
■ 1Mbit/s	15.9	58

(e.)

Time to process a CAN message transmitted (t_{PCM}) - Master

	Master-Slave	Master-Checker
▨ 125kbit/s	20	21
■ 1Mbit/s	8.4	8.4

(f.)

Figure 4.4: Continued.

Stellenbosch University http://scholar.sun.ac.za

The time between each message transmitted (the time between every send_mx() procedure call) is illustrated in Figure 4.4d. What must be noted here is the fact that the extra time available for the 125kbit/s setup compared to the 1Mbit/s setup, does not signify more time for processing. Since the oscillator frequency of the 125kbit/s setup is 8MHz and the oscillator frequency of the 1Mbit/s setup is 20MHz, processing of instructions was slowed down with the decrease in the oscillator frequency.

Inside a send_mx() procedure, another procedure is called which is responsible for the transmission of either the CRC calculated by the Master on the data of the command, or to request the Slave to transmit a remote request message to a particular node. These messages are transmitted to the Slave via the CH1_MSG-register[1] and an interrupt request, *'nCH1_MSG_INT'* with the help of the protocol discussed in paragraph 3.8.4. Two bytes are transmitted, a code byte and a data byte. When the Slave reads each byte from the buffer with signal nCH2_MSG_RD from within its interrupt service routine, *'twin_p_msg_int'*, it also acknowledges the Master. The Master and the Slave transmit their respective messages across the CAN network at approximately the same time after completion of the interprocessor communication (*CAN bitstream* waveform).

At the respective destinations of the CAN messages, interrupts will occur which will cause the processors to check the status of the interrupt request. Depending on the source of the interrupt, whether caused by an fault in the message or valid data, it will respond to the error or indicate valid data by setting a data valid flag. This flag will cause an event to occur in the main loop with the result that a procedure *'Process_Message()'*, will be called to read the data from the CAN controller's buffers. Figure 4.4e indicates the time needed to process this data if the message was received. Notice here that there is a significant difference between the Master-Slave and Master-Checker parameter.

For transmitted message Figure 4.4f indicates that there is an insignificant difference between the times for the two setups. The processing for transmitted messages entails only a reception of the successful transmission flag and the request to transmit the next message. The processing for received messages involves the validation of the data received from the Checker. Thus, inside this procedure the Master and the Checker call a procedure *'compare_message()'*. In the Checker's case, the processor calculates the CRC from the data that was received and transmits this byte via interprocessor communication to the Master (*'Message_From_TwinP'* waveform). In the Master's case, it also calculates the CRC on its own received data and then waits for the Checker's CRC in order to make the comparison. This message from the Checker may come either before or after the Master calls its compare_message() procedure, depending on how fast the two processors respond to their respective CAN messages. (The Master does not compare its data to the Slave's data in a Master-Slave setup.)

A difference in the CRCs will be logged and the message ignored. If the CRCs match, the data will be transmitted to the application with the procedure *'SendAppMessage()'*. In this procedure-call, data bytes are written to a register *'APP_REG_WR'* (register INTF_OUT_REG in the CPLD) by the Master (*'Writing to APP_REG_WR'* waveform).

[1] For the rest of this discussion it is accepted that channel 1 is the Master.

Stellenbosch University http://scholar.sun.ac.za

This write cycle (nCH1_APP_WR) also sends an interrupt request to the application which will in turn perform Read-cycles ($\overline{DSTRB}$) for every interrupt received from the Master, acknowledging on every read cycle. Data received is passed on to the application between the time *'Data update completed'* and *'Completion of sending data to the ACP'*.

A Write-cycle with respect to the application starts with the transmission of a byte to the dual CAN node. On writing each byte of the message to the mailbox (register INTF_IN_REG in the CPLD), the application sends an interrupt request (*'nWrite'* waveform) to the Master of the dual CAN node to inform it that a message byte is available. The Master reads the byte from the mailbox from within its *'Application_Msg_Int'* interrupt service routine, and acknowledges the message at the same time with it's read signal (nCH1_APP_RD). This process continues until the Master has received the complete message from the application.

4.1.1 Extrapolation of the *Useful Processing Time*

Not all the ACP's remote request messages are answered by the sensor and actuator nodes in the demonstration model. Therefore only 14 of the 118 bytes of sensor data are gathered and then transmitted to the ACP. This is in light of the fact that only 2 dual CAN nodes, representing two sensors, were physically involved in the tests, and not the full complement of ADCS' sensors and actuators. To make an accurate extrapolation for the time *'Second start to Processing Done'* or t_{SPD} for the rest of the sensors and actuators, the timeouts have to be replaced by the time to process and transmit data to the ACP for the individual messages. However, a good estimation of this parameter can be made by using Equation 4.1 and Table B.6. (Table 4.2 summarises the results.)

$$\begin{aligned} t_{SPD\,Extrapolated} &= t_{SPD\,Current} - (MM\,and\,HSS\,messages\,managed) \times t_{SMTO} \\ &+ (13\,CAN\,messages\,to\,be\,transmitted) \times (t_{SM} + t_{SMOK}) \\ &+ t_{WCDA} \times \frac{118\,data\,bytes + one\,code\,byte\,to\,be\,transmitted\,to\,the\,App.}{14\,data\,bytes + one\,code\,byte\,transmitted\,to\,the\,App.} \end{aligned} \tag{4.1}$$

All the commands from the ACP are transmitted across the CAN network and acknowledged by the PCCAN card. Transmission of the commands was thus taken into account in the extrapolation of the useful processing time. Note that interrupt and response times involving the EPP port are not relevant to the final implementation of the dual CAN node because the PC will be replaced with a suitable ADCS processor. Those values from the EPP that are used in the calculations and measurements are however of the same order and can be accepted as sufficient.

Stellenbosch University http://scholar.sun.ac.za

Table 4.2: Summary for the parameter *Second start to processing done.*

Parameter	*Average time*				*Units*
Configuration	*Master-Slave*		*Master-Checker*		
Oscillator frequency	8	20	8	20	MHz
Baud rate	125k	1M	125k	1M	bit/s
Dual CAN node's t_{SPD}	16.5	4.0	17.5	4.6	ms
Extrapolated t_{SPD}	30.2	8.3	33.6	10.0	ms

4.2 DC Analyses

From Table 4.3[1] one can derive what the approximate power dissipation would be for a dual and single channel CAN node with only the essential components at baud rates 125kbit/s and 1Mbit/s respectively (Table 4.4). The current for the CPLD, resistors, LEDs and oscillator is omitted for the derivation because it could not be broken up into its components in order to take only the resistors and oscillator current into account. The CPLD current was omitted because the final product will probably be implemented with SSI and MSI components instead[2] (see paragraph 4.2.1), whereas LEDs are not implemented in flight model products.

Table 4.3: D.C. Characteristics for the dual CAN node in a Master-Checker configuration.

Oscillator frequency of 16MHz, CPU clk = 8MHz, CAN baud rate = 125kbis/s			
Channel	*Device(s)*	*Current*	*Units*
CH1&2	74HC14	50	μA
CH1	C505C	13	mA
CH1	74HC573	36	μA
CH1	C505C, EPROM, 74HC573	18.44	mA
CH1	PCA82C250	13	mA
CH2	C505C	16.6	mA
CH2	74HC573	35	μA
CH2	C505C, EPROM, 74HC573	26.14	mA
CH2	PCA82C250	13	mA
CH1&2	CPLD, resistors, LED's, oscillator, ect.	134	mA
CH1&2	Dual CAN Node (total)	204.63	mA

[1] This table summarises the *measured* currents for the dual CAN node's components, and not the absolute maximum values as given by the respective datasheets.

[2] This will depend on the technology at the time.

Stellenbosch University http://scholar.sun.ac.za

Table 4.3: continued.

Oscillator frequency of 20MHz, CPU clk = 20MHz, CAN baud rate = 1Mbis/s			
Channel	*Device(s)*	*Current*	*Units*
CH1&2	74HC14	58	μA
CH1	C505C	31	mA
CH1	74HC573	40	μA
CH1	C505C, EPROM, 74HC573	40.54	mA
CH1	PCA82C250	13	mA
CH2	C505C	27	mA
CH2	74HC573	42	μA
CH2	C505C, EPROM, 74HC573	34.14	mA
CH2	PCA82C250	12.5	mA
CH1&2	CPLD, resistors, LED's, oscillator, ect.	239	mA
CH1&2	Dual CAN Node (total)	339.24	mA

The measured current dissipation for a 20MHz oscillator was 11mA, while having an absolute maximum of 40 mA [6, Farnell, 1998/99:1598]. Instead of adding this value to the calculations at this point, the current dissipation of the oscillator equivalent, a crystal and its accompanying capacitors, would be investigate. A comparison of the respective current dissipations will indicate the more sufficient value to take into consideration when calculating the total power dissipation of the dual CAN node.

Table 4.4: The derived approximate power dissapations for the dual and single channel CAN nodes, with only the essential components at baud rates of 125kbit/s and 1Mbit/s respectively.

a.) Master-Slave configuration at 125kbit/s

Component(s)	*Current*
74HC14	50 μA
C505C, EPROM and 74HC374	26.14 mA
PCA82C250	13 mA
Total	39.19 mA

b.) Master-Slave configuration at 1Mbit/s

Component(s)	*Current*
74HC14	58 μA
C505C, EPROM and 74HC374	40.54 mA
PCA82C250	13 mA
Total	53.6 mA

Stellenbosch University http://scholar.sun.ac.za

Table 4.4: continued.

c.) Master-Checker configuration at 125kbit/s

Component(s)	*Channel*	*Current*
74HC14	1 & 2	50 μA
C505C, EPROM and 74HC374	1	18.44 mA
PCA82C250	1	13 mA
C505C, EPROM and 74HC374	2	26.14 mA
PCA82C250	2	13 mA
Total		70.63 mA

d.) Master-Checker configuration at 1Mbit/s

Component(s)	*Channel*	*Current*
74HC14	1 & 2	58 μA
C505C, EPROM and 74HC374	1	40.54 mA
PCA82C250	1	13 mA
C505C, EPROM and 74HC374	2	34.14 mA
PCA82C250	2	13 mA
Total		100.24 mA

From this table, one can see that the oscillator frequency has a big influence on the power dissipation of the node, with the biggest contribution noticeable amongst the processors. Also notice the dissipation of the transceiver, which is surely an area of concern. (These aspects will be discussed in the Conclusions.)

4.2.1 Omitting the CPLD

The CPLD on the dual CAN node is responsible for the communication between the two channels. Since this communication is absent in the Master-Slave configuration and because of the simplicity of the interface electronics to the application, the CPLD becomes negligible. Futhermore, power dissipation for the interface electronics will become negligible if implemented with high speed CMOS components. By viewing the CPLD's VHDL code, one can determine what interface components are required to implement a Master-Slave configuration. Using a bidirectional three-state register such as Motorola's 74HC646 (see paragraph 3.5.5) for the data bus, a normal three-state latch for the address bus and an AND- and OR-package for the active low control signals, an estimated absolute maximum current dissipation can be calculated. (See below for details.)

Stellenbosch University http://scholar.sun.ac.za

Table 4.5: Estimated absolute maximum current dissipation for the interface electronics.

Component(s)	*Current*
74HC646	80 μA
74HC573	160 μA
74HC08	20 μA
74HC32	20 μA
Total	280 μA

Adding this total to that measured for the Master-Slave configuration above, the estimated current dissipation of a single CAN channel (at 1Mbit/s) is in the order of 53.9mA.

4.3 Design Analyses

4.3.1 Pull-up Resistors

Pull-up resistors were used to ensure that signals to the CPLD and to the processors are always in a defined state, especially during a power-on cycle and when a signal-source is in a high impedance state. However, these pull-up resistors caused a problem during normal system operation. Provision was made to remove power from a processor in order to test the switching algorithm during mode-swopping. When this was executed, the processor continued to operate as before due to the fact that it drew current from its pins which were pulled up to the supply voltage via the resistors. The only other source of power was the CPLD which could not be removed during this test. All other neighbouring components were removed.

A temporary solution was to *virtually* switch off the processor, so that the other processor had the impression that power was removed. This was achieved by disabling all interrupts, disabling all messages which could automatically respond to remote request frames, and also disabling the transmission of any messages accross the CAN bus and to the other processor. Furthermore, the signal that indicates the health-status of a processor to the other processor, the $\overline{CH1\_Health}$ or $\overline{CH2\_Health}$ line, was set to indicate that the processor was in a disabled or malfunctioning state. Tests could be completed satisfactory with minor changes in the code[1].

A possible solution to this problem is to make use of active-high signals with pull-down resistors. If the power were to be removed from the processor, it would be unable to draw

[1] These changes are marked and can be removed when the system is operating properly.

Stellenbosch University http://scholar.sun.ac.za

power from the ground via the pull-down resistors. For example, the status line of the processor is pulled low via a resistor and if the health of the processor is good, it will make the signal high. If the power were to be removed from the processor, it would be unable to draw power from the ground via the pull-down resistor and being low, the signal will indicate ill-health to the other processor. In future scenarios, such as a malfunctioning processor that still indicates good-health, should be investigated.

4.3.2 Interrupt latencies

Interrupt latency is the inherent delay between the occurence of an interrupt and the reaction of the CPU. One source of interrupt latencies is instruction completion. Therefore, latency times will be dependent on the processor's clock frequency. However, the effect of latency is most pronounced when lower priority interrupts are initiated during the execution of higher priority tasks. It can be a very serious problem when a lower priority interrupt might be lost because a higher priority interrupt is being serviced. Thus, managing interrupts and registering interrupts had to be done with care. Table B.6 gives the interrupt latencies for the CAN interrupt (t_{CBI}), the interrupt from the application (t_{IAI}), and the interrupt to the twin processor t_{IISR}.

Unlike the CAN interrupt, the interrupts from the application and the processor are acknowledged on entering the interrupt service routine and reading the data from the registers. Thus, if these messages are not acknowledged within the time allowed for the receiver to respond to the interrupt, a time-out will occur. These time-outs had to take into account the interrupt latency times of the respective interrupt sources.

4.3.3 Time-loading

According to Laplante [11, Laplante, 1993:13], "Time-loading, or the utilisation factor, is a measure of the percentage of *useful* processing the computer is doing." If the system is near its full capacity (e.g. 98% [11, Laplante, 1993:13]), it becomes difficult to expand the system without risking time-overloading. On the other hand, a system that has a time-loading factor that is under-utilised (e.g. 10% [11, Laplante, 1993:13]), is inefficient because this implies that the system is too powerful for the application. Laplante indicates that a time-loading factor of 70% to 80% is acceptable for systems which do not expect development. Time-loading for the Master-Checker and the Master-Slave configurations can be derived from Table 4.2 and can be given as follows as a percentage of one second.

Stellenbosch University http://scholar.sun.ac.za

Table 4.6: Time-loading as a percentage of one second.

Configuration	*Baud rate*	*Percentage*	
	(bit/s)	Master	Slave
Master-Slave	1M	0.4	
	125k	1.65	
Master-Checker	1M	0.46	0.4
	125k	1.75	1.52

From this summary, one can conclude that the processors are severely under-utilised in terms of processing time. In fact, if it was not for the processors' hardware utilisation and requirements of speed for communication, one could use a much smaller device. To have a higher utilisation factor, the processors can share their computing power with other systems. The application connected to the dual CAN node can for example make use of this computing power very effectively.

4.3.4 Memory-Loading

Memory-loading is typically [12, Laplante, 1997:234] the sum of the memory-loading for the program, stack and RAM areas respectively. That is,

$$M_T = M_P \cdot P_P + M_R \cdot P_R + M_S \cdot P_S \tag{4.2}$$

where M_T is the total memory-loading, M_P, M_R and M_S are the memory-loading for the program, RAM and stack areas respectively and P_P, P_R and P_S are percentages of the total memory allocated to the program, RAM and stack areas respectively. Note however that if any one of the above percentages are over 100%, the system is already memory-overloaded and cannot operate. Thus, with the help of the output of the linker and the code, the total memory loading for this design can be calculated.

The data variables declared in the code take up 130 bytes of the internal RAM space of the C505C processor. The rest of the 256 bytes RAM are allocated to the stack memory of the processor. By investigating what the deepest level of penetration[1] of the software is, and by determining how many bytes are pushed onto the stack during the interrupt service routines, one can get a reasonable idea what the worst case might be for the utilisation of the stack memory of the processor. It was found that *four* was the maximum number of times the processor performs the LCALL instruction in the recursive process described

[1] How many times a procedure calls another procedure and inside that procedure another procedure is called and so forth.

Stellenbosch University http://scholar.sun.ac.za

above. For every LCALL instruction, the two bytes of the program counter are pushed onto the stack, signifying 8 bytes pushed onto the stack. Then the interrupt service routine dealing with messages received from the application[1], pushes another nine bytes onto the stack. Adding these bytes up, one finds that the worst case for the number of bytes pushed onto the stack at any given time, is 17. Furthermore, from the 32kb program memory of the 64kb program memory space allocated, only 6550 bytes are used. With the above information and equation 4.2, it follows that the total memory-loading is:

$$M_T = \frac{6550}{32768} \cdot \frac{32768}{33024} + \frac{130}{130} \cdot \frac{130}{33024} + \frac{17}{126} \cdot \frac{126}{33024} = 20.6\% \qquad (4.3)$$

Notice that the RAM area is 100% loaded because the compiler will assign only the neccessary variables to consecutive positions in the RAM and the rest of the RAM is designated as stack space. This scenario is a special case of the 100% mentioned earlier causing memory-overloading. There can thus only be cause for alarm if the stack space starts to approach 100%. Nevertheless, if the program memory was not extended from the estimated size of 15kB (16kB EPROM)[2] to 32kB, the memory-loading parameter could have been 40.2%. Furthermore, if the allocated program memory space was to be reduced to 8kb the memory-loading percentage can be improved to 79.3%. The possibility of sharing the dual CAN node's resources with the application then becomes limited, but when such an application is implemented, memory-loading is improved.

4.3.5 Stress Testing

Stress testing consists of a burst of messages on the CAN bus followed by a smaller disturbance spread out over a longer period of time [12, Laplante, 1997:269]. Stress testing thus permits a way to test certain aspects of the fault tolerance of a system, and verifies that the facilities in place to manage the loss of data and data corruption, are working properly. Depending on the message ID's (priorities) used, stress testing will also indicate how the system would fare operating under pressure or operating over long periods of time.

All burst-messages with ID's not represented in the message objects of the processors, will be ignored with the effect of not even generating interrupts because of the ID-masks for acceptance filtering. These messages will only be acknowledged and if in error, the neccessary action will be taken by the CAN controller (not CPU) to inform other nodes to ignore the latest data. If the ID does correspond, events will take their normal course and the message object's data will be updated. The remote request will then be transmitted.

However, this test is of higher importance for a multi-master architecture than a single-master architecture. Based on this fact and the lack of time, no stress test was performed on the dual CAN node.

[1] No other ISR will play a role in this calculation, as all interrupts are disabled on entering an ISR.

[2] See paragraph 3.7.2 for details.

Stellenbosch University http://scholar.sun.ac.za

4.3.6 Comments on the C505C

During the development and evaluation of the dual CAN node, some problems were encountered with the C505C processor. These problems are mentioned here in short as reference for future development of the dual CAN node.

Message Object 8

The C505C's message object 8 transmits an ID of 0 (or 000 0000 0000) for any ID programmed in its message object. This was verified with an oscilloscope by monitoring the bits as they were transmitted over the network. A further verification with the same results was obtained with the help of the diagnostic CAN card (PCCAN) (see Appendix C), which has four individual CAN nodes from two different companies (Intel and Philips). Other processors were taken through the same routine with the same results. No statement can be made about these results, because the malfunctioning of this message object might be limited to the batch of processors received from the manufacturers.

Message Object 9

Message object 9 of the C505C does not respond to remote requests. It does however send Standard messages successfully. The same verification steps were performed on this message object as were used with message object 8. The same conclusion must be made about the results.

–ooOoo–

Stellenbosch University http://scholar.sun.ac.za

Chapter 5

Conclusions and Recommendations

At the start of this investigation, the question asked was: *Can the ADCS of SUNSAT I be improved or upgraded?* and the answer was: *The ADCS is an integrated system which provides for the necessary redundancy and data management. There is little that can be improved.* Then after a deeper investigation into new technology, modularity, reliability and robustness, a new question was asked: *What on the ADCS can be improved and what should be improved?* and the answer was: *Anything*!

The objective of this thesis was to develop a modular communication system to manage the real-time data of the ADCS on a new generation SUNSAT satellite. The project's development methodology included an analyses of the current ADCS and CAN protocol. Based on the requirements and specification compiled from this investigation, a demonstration model consisting of three prototype nodes was developed. This demonstration model was analysed, and from the results a conclusion could be made on whether the project has succeeded in this objective.

5.1 Conclusions

One can measure the value gained by the dual CAN node through comparison on how well it complied to the requirements and specifications. With reference to the requirements that were laid on the table for this project (paragraph 1.3), having each module communicating its status was the only requirement that was *not* fulfilled. This failure was due to a lack of time. The other requirements were implemented successfully. A simple extension of the code and messages is all that is required to incorporate the unfinished function. The information to compile this status-message does in fact already exist in the current code as part of the error detection of the system (paragraph 3.8.8).

It was shown with the help of the demonstration model that the project complied to the requirements in the following manner:

Stellenbosch University http://scholar.sun.ac.za

- The complexity of the communication architecture and related wiring harnesses (data, address and control buses) were reduced by way of CAN transceivers and a simple twisted pair that connected the modular nodes (actuators, sensors and processors) together;

- The CAN technology's high data transfer rates, resulting in fast, reliable responses during data management, ensured that tasks and data managed by the ACP-node, were completed within a mere 2% of the ADCS' one second cycle time. The actuator and sensor nodes will even take less time to manage their data since these nodes are only responsible for their own data. This leaves the nodes with processing time available for utility functions and application software. Therefore, CAN technology provides enough room to accommodate all the neccessary processors, sensors and actuators of the ADCS and its data;

- It was shown that error detection capabilities can be incorporated with ease into the current ADCS with the built-in facility of the CAN protocol;

- The single points of failure identified in the integrated system can be removed, but under *certain* configurations of the dual CAN node new single points of failure were created. Both the old ADCS architecture and the new proposed architectures have their single points of failure in the form of a multiplexer. The old architecture multiplexed the ICP, ACP and OBC2's communication on the RWSSBUS, and the new architecture multiplexes the two CAN buses at each dual CAN node's application interface. The new architecture has thus distributed the old multiplexer across the system with the added advantage of having a dual bus system. In the case of a local multiplexer failure, it will be isolated to the affected node while the rest of the system can operate as before. Complete system failure is thus avoided.

- Diagnostics were enhanced from being able to interface to the ADCS via a Link of the transputer gathering sensor data and transmitting commands, to being able to also see exactly what information is traversing the communication routes. The scheduling of data and the possible occurences of errors in the system can be visually observed. The diagnostic tool further allowed for the complete testing of nodes (both channels) before they were integrated into the system;

- Modularity was increased in such a way that, although it caused increased component-count and power consumption, it has led to simplified debugging, easier access, simplified upgrading and well defined, standardised interfaces and drivers. Modules can now be moved, added and removed with only minor changes in software where required. CAN thus makes modules portable, increasing the probability that these sensors and actuators might be implemented on other satellites catering for CAN technology;

- It was shown that the system is very flexible in three aspects. First, the system can operate by broadcasting or requesting messages. (The latter requiring more bandwidth.) Secondly, the system is equally capable of operating in a single master

Stellenbosch University http://scholar.sun.ac.za

as well as multi-master topologies. Although the multi-master topology was never implemented, it was shown what the advantages it would have above the single master topology. Finally, the CAN technology is also extremely flexible with regard to the choice of baud rates;

- The system has increased reliability and robustness. This was achieved with the readily available redundancy in the system to prevent severe malfunctioning of the entire system. Redundancy was achieved on node-level as well as on system-level. On node-level, dual nodes operating on a dual CAN bus, were provided and on system-level multiple access from either attitude processors or onboard computers was provided. Combining this redundancy with a multimaster architecture one can give other interested satellite subsystems (OBC's, Telemetry and Telecommand) access to the ADCS's network of processors, actuators and sensors. This allows for the multiplexing of command and data-gathering modules in an enhanced redundant attitude determination and control system;

- Actuators and sensors will have access to intelligence provided by the local processors and their available processing-time (as was mentioned earlier). With this facility, sensors and actuators, depending on the node, could become independent from intelligence from elsewhere. The Star Sensor node for example requires more processing power than the dual CAN node can provide.

- The CAN protocol might not be a standard *space* protocol yet, but it does show a lot of potential on already implemented systems.

With these results, the reliability of the old system can either be maintained or improved in the dual CAN node. By choosing the most feasible and realistic dual CAN node configuration, one can make a compromise between reliability, the power budget and the risk factor;

Is the dual CAN node architecture overly complex? The CAN protocol itself is not overly complex. In fact, it is very simple to implement, especially on a single CAN bus. A CAN Master-Slave architecture, allowing for simple redundacy with the help of some extra components, is not an overly complex design either. The CAN Master-Checker architecture, although completely functional, giving good results and responses, might be too complex for an interface to ADCS modules. Since the controllers have to code, decode, debug and verify the messages received and transmitted across the dual node network, the Master-Checker's software becomes more complex than that of the Master-Slave's software. Furthermore, the orientation estimation (EKF's) will not diverge because one or two bytes from the sensor data got lost. In comparison with the lack of error detection and correction of messages on the ADCS of SUNSAT I, the CRC and auto-retransmission provided by even a single CAN-bus will be a vast improvement in reliability on the current system. A new ADCS incorporating a primary active CAN-bus and a secondary dormant CAN bus, will increase this reliability even further as a function of redundancy.

Stellenbosch University http://scholar.sun.ac.za

Is the dual CAN node efficient? With regard to power efficiency, comparing the power dissipation of a single CAN channel (approximately 275mW at 20MHz (1Mbit/s) on a Master-Slave architecture) and the power dissipation of the ICP on the ADCS of SUNSAT I (approximately 80mW at 3.6864MHz), one can derive that the dual CAN node needs a re-design with emphasis on power saving. The high value for the power dissipation of the dual CAN node is mainly due to the high clock frequency and the power dissipation of the CAN transceivers (65mW each). The transceivers used in this design was not chosen for its power dissipation, but was received by way of samples. To make matters worse, the power dissipation for the CAN node has to be multiplied with the number of nodes the ADCS will be implemented with, giving an unacceptable 2.2W neccessary for the CAN interfaces alone. The power dissipation of the current ADCS with a bus voltage of 14V, is 6W with the following modules switched on,

- ADCS (128mA);
- ICP (15mA);
- ACP (55mA);
- Star Sensor (65mA); and
- Horizon Sensors (167mA).

To compare the power dissipation of the proposed ADCS and the current ADCS, the current dissipation of the ACP, Star Sensor and Horizon Sensors are added to the current dissipation of three dual CAN nodes. (The ICP above is confined to the current system.) Comparing the calculated 7.5W of the proposed ADCS to the current system's 6W, one can immediately see that modularity certainly brings along with it a serious dilemma in the form of power dissipation. When the final concept design is drawn up, this matter will probably carry more weight than the choice of the reliablity gained. One has to note at this point that aspects such as the clock frequency mentioned above, the possibility to switch nodes off that are not required to operate and the possible implementation of low power CAN transceivers, allow the designer to re-calculate and reconsider the power dissipation dilemma.

The conclusion made from the results above is that the dual CAN node certainly does meet the project's objectives to a very large extent with only a few problems which could not be addressed during the design. The dual CAN node also showed that it can definitely support the requirements of the ADCS. If a compromise can be reached between the reliability gained and the power dissipated, the dual CAN node can certainly be of great service to the ADCS in new generation SUNSAT satellites. The prototype successfully demonstrated a system that can easily incorporate into it the existing ADCS with the general impact of more efficient and robust operation, easier maintenance as well as sufficient room for development with the neccessary CAN diagnostic tools as support. With the CAN diagnostic tools, one can tell exactly what data is on the bus and what the origin and destination are of the data. This architecture allows for individual modules to

Stellenbosch University http://scholar.sun.ac.za

be tested before integrating them into the system. There is only one major disadvantage of this concept, a whole new ADCS motherboard design is required to integrate this concept, since it forms the foundation of the orientation control system. In contrast to this big challenge, one must consider the benefits and the improvement in performance measures of the dual CAN system.

5.2 Recommendations

The following recommendations are made in regard to future developments of the dual CAN node. Some or most of them can be pursued during the possible integration of dual CAN node system into the ADCS of a future satellite.

- The double time-out facility implemented in the hardware and the software to make sure the processors did not hang in loops waiting for responses[1], worked extremely well. However, in the event that a hardware time-out occurs, the processor is deceived to think that the data was acknowledged by the receiver of its message. It will thus, instead of logging an error of *no response*, continue to send messages to a possible malfunctioning receiver. One can remove the hardware time-out, but then the processor might be caught in an endless loop waiting for an aknowledgement from malfunctioning hardware. Further investigation into this problem is recommended.

- With the time available and the initial software planning and structuring, some kind of layering could be established in the form of a physical interface, driver and application. However, the software was not written optimally and can be refined. Focus can be placed on re-usability, forcing the driver to be a clearly separated entity. Care must thus be taken for example when parameter-passing is modified with the aim of removing any global variables.

- Sending a time-synchronisation message from the main module (ACP-application) to the other modules over the network to compensate for interrupts and drift in clock frequencies.

- Instead of making use of the CPLD or glue logic for communication between the two CAN processors, one can also use the serial interface of the processors for interprocessor data transfer. By rearranging some of the signals on the current design, the serial pins can become available for this purpose. This step can considerably simplify the design, particularly the software.

- The Watchdog timer of the C505C was not implemented in this design. It will be to the advantage of the system's reliability to include it in a final implementation of the dual CAN node. A Watchdog is thought of as an "I'm okay" method of error detection, and allows automatic recovery from a software problems. The user

[1] See paragraph 3.8.8 for details.

Stellenbosch University http://scholar.sun.ac.za

software clears the Watchdog periodically within a pre-programmed time period. For example, a message is sent from one location to another and the message was not acknowledged by the destination node. The source of the message, waiting in a loop for the acknowledge, will fail to refresh the timer, and an internal hardware reset will be initiated by the Watchdog. A hardware related problem such as single event upset, can also cause the Watchdog to initiate a reset cycle.

- One can perhaps consider individual reset and oscillator circuits for the two channels of the dual CAN node. This would add redundancy and higher reliability but may also increase power consumption. (See paragraph 4.2.)

- A multi-master architecture can cause a noticable decrease in the size of the software and its complexity compared to the single-master architecture of the current design. This is due to the fact that the ACP-Master does not have to administrate the data collection and distribution of the sensors and actuators. It only receives data from sensors that broadcast data autonomously, and it transmits commands to the actuators via remote request messages received from the actuators. Transmission of data can still be periodic by implementing the same ADCS period on each node.

- Another topology for the dual CAN node can also be implemented with two microcontrollers, each one having *two* on-chip CAN modules. One module of each microcontroller is used to interface to the CAN network, together making up the dual CAN network. Then, the second CAN module of each device can be used for the interprocessor communication. The transceivers can possibly be avoided. This configuration gives the advantage of CAN's error detection capabilities which were not present in the configuration as implemented in this project.

Finally, one must keep the following in mind:

Do not be afraid of paradigm shifts. They keep one out of ruts. However, too large steps should be avoided. Take smaller steps, making sure that a subsystem is working perfectly and in harmony with other subsystems. - X. Farr

–ooOoo–

Stellenbosch University http://scholar.sun.ac.za

Bibliography

[1] Altera Corporation (1996), *Altera Data Book*, Altera Corporation, San Jose, California.

[2] Altera Corporation (1998), *Application note 74 - Evaluating Power for Altera Devices*, Altera CDROM Data Book, Altera Corporation, San Jose, California.

[3] Barrenscheen J. (1998), *TwinCAN, A new milestone for inter-network communication*, Siemens AG, München.

[4] Bosch R. (1991), *CAN Specification Version 2.0*, Robert Bosch GmbH, Stuttgart.

[5] Dunne A., Dillon P., Heffernan D. and Stack P. (1998), *A CAN-based Emergency Light Test Network*, 5th International CAN Conference, San Jose, California.

[6] Farnell (1998), *The Electronic Components Catalogue*, Farnell, Leeds, United Kingdom.

[7] Harris Corporation (1992), *Microprocessor Products for Commercial and Military Digital Applications - Harris Semiconductors*, Harris Corporation, Melbourne, Florida.

[8] Holtzman G.J. (1991), *Design and validation of Computer Protocols*, Prentice Hall, New Jersey.

[9] Intel Corporation (1994), *INTEL - Embedded Controller Applications Handbook*, Intel Corporation, München, Germany.

[10] Kvaser AB (1998), *PCcan 2.0 - Kvaser AB*, Kvaser AB, Kinnahult, Sweden.

[11] Laplante P.A. (1993), *Real-Time Sytems Design and Analysis - An Engineers Handbook*, IEEE PRESS, New Jersey.

[12] Laplante P.A. (1997), *Real-Time Sytems Design and Analysis - An Engineers Handbook 2nd edition*, IEEE PRESS, New York.

[13] Lawrenz W. (1997), *CAN System Engineering: From Theory to Practical Applications*, Springer-Verlag, New York.

Stellenbosch University http://scholar.sun.ac.za

[14] Motorola (1988), *Motorola Semiconductors High Speed CMOS Logic*, Grosvenor Press, Great Britain.

[15] Siemens AG (1997), *C505C 8bit CMOS Microcontroller User's Manual*, Siemens AG, München, Germany.

[16] Siemens AG (1997), *Siemens Microcontroller Starter Kits - Edition 2.1*, Siemens AG, Siemens Semiconductor Group, München, Germany.

[17] Stallings W. (1994), *Data and Computer Communications - Fourth Edition*, Maxwell MacMillan International, New York.

[18] Steyn W.H. (1995), *A Multi-mode Attitude Determination and Control System for Small Satellites*, Ph.D. Thesis, University of Stellenbosch.

[19] Wertz J.R. and Larson W.J. (1995), *Space Mission Analysis and Design Second Edition*, Microcosm, California.

[20] Wolf A. and Koller C. (1998), *16-bit microcontroller with two CAN modules*, Siemens Microelectronics, San Jose.

[21] Xilinx (1998), *Xilinx Programmable Logic Data Book*, Xilinx, San Jose, California.

Electronic references

[22] DeviceNet Europe Technical Support Centre (DETSC) (2000), *CAN FAQ*, http://www.warwick.ac.uk/devicenet/can_faq.htm, September 2000.

[23] Hitex (2000), *Controller Area Networking - The Future Of Industrial Microprocessor Communications?*, http://www.hitex.co.uk/CAN/canarticle.html, September 2000.

[24] Kvaser AB (1999), *CAN Frequently Asked Questions*, http://www.kvaser.com/can/products/index.htm, March 2000.

[25] Kvaser AB (2000), *Other CAN Circuits, etc. - Kvaser CAN Pages, CAN Introduction*, http://www.kvaser.se/can/products/other.htm, July 2000.

[26] LaBel K.A. (1996), *SEECA - Single Event Effect Criticality Analyses*, http://flick.gsfc.nasa.gov/radhome/papers/seeca6.htm, September 2000.

[27] Müller T. and Milne G.W. (1999), *A brief description of SUNSAT's structure and design*, http://sunsat.ee.sun.ac.za/faq1.htm#structure, September 2000.

[28] National Semiconductors (1995), *MM54HC14/MM74HC14 Hex Inverting Schmitt Trigger*, http://www.ncs.com/, June 2000.

[29] Peacock C. (2000), *Beyond Logic*, http://www.beyondlogic.org, September 2000.

Stellenbosch University http://scholar.sun.ac.za

[30] Philips Semicinductors (2000), *PCA82C250 - CAN controller interface*, http://www-eu2.semiconductors.com/pip/PCA82C250, June 2000.

[31] Schofield M.J. (2000), *Controller Area Network - Available Devices*, http://www.omegas.co.uk/CAN/devices.htm, September 2000.

[32] Schofield M.J. (2000), *Controller Area Network - Error Handling*, http://www.omegas.co.uk/CAN/errors.htm#Error Detection, September 2000.

Stellenbosch University http://scholar.sun.ac.za

Appendix A

Controller Area Network

Principle features

- 1Mbit/s transmission on up to 40m length bus.
- 5kbit/s transmission on up to 10km length bus.
- 2032 identifiers for CAN version 2A.
- 2×10^{29} identifiers for CAN version 2B.
- Multimaster system.
- Guaranteed latency times.
- Powerful error detection and handling.
- Non-destructive bitwise arbitration.
- System-wide data consistency.
- Discrimination between temporary and permanent failures at nodes and automatic switching-off of defective nodes.
- Automatic retransmission of frames that have lost arbitration or were disturbed during transmission.
- Can request data when needed.
- Two wire system.

Stellenbosch University http://scholar.sun.ac.za

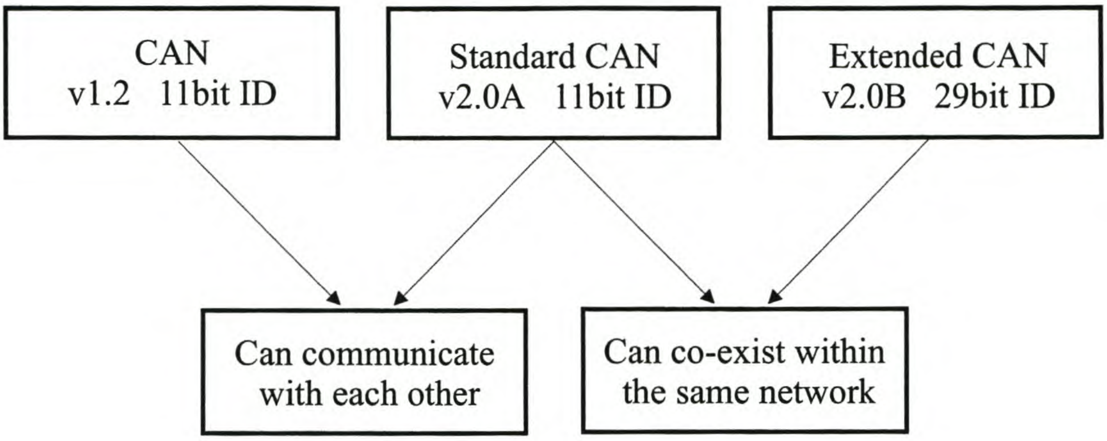


Figure A.1: Illustration of the relationship of the recent CAN versions.

A.1 BasicCAN and FullCAN

The architectures of controllers are not covered by the CAN standard, so there is a variation in how they are used. There are two general architectures: BasicCAN and FullCAN which should not be confused with CAN1.0 and CAN2.0, or standard identifiers and extended identifiers. The difference is in the buffering of the messages [22, DETSC, 2000].

A.1.1 BasicCAN

In the Basic CAN controller, the architecture is similar to a simple UART, except that complete frames are sent instead of characters. There is typically a single transmit buffer and a double-buffered receive buffer. The processor puts a frame in the transmit buffer and services an interrupt when the frame is sent. The processor receives a frame in the receive buffer, services an interrupt and empties the buffer before a subsequent frame is received. The processor must manage the transmission, reception and the storage of the frames.

A.1.2 FullCAN

In a FullCAN controller, the frames are stored in the controller. A limited number of frames (typically 16) can be dealt with. Because there can be many more frames on the network, each buffer is tagged with the identifier of the frame mapped to that buffer. The processor can update a frame in the buffer and mark it for transmission. Buffers can be examined to see if a frame with a matching identifier has been received.

Stellenbosch University http://scholar.sun.ac.za

The intention of FullCAN is thus to provide a set of "shared variables" in the network. The processor periodically updates the variables to be transmitted and reads the variables received. Status flags are provided to inform the processor that the data was updated. The processor examines these flags after reading from the buffers to ensure that data was not updated while the processor was reading from the buffers.

A.2 Non-Destructive Bitwise Arbitration

The priority of a CAN message is determined by the binary value of its identifier. The numerical value of each message identifier (and thus the priority of the message) is assigned during the initial phase of system design. The identifier with the lowest numerical value has the highest priority. Any potential bus conflicts are resolved by bitwise arbitration in accordance with the wired-and mechanism, by which a dominant state (logic 0) over-writes a recessive state (logic 1). The overall result is the same as if the highest priority message were the only message being transmitted. As soon as any lower priority transmitter loses control of the bus via the arbitration mechanism, it automatically becomes a receiver of the message with the highest priority and will not attempt re-transmission until the bus becomes available again. Therefore, non-destructive bitwise arbitration provides bus allocation on the basis of need. Transmission requests are dealt with in order of priority, with minimum delay, and with maximum possible utilisation of the available capacity of the bus.

A.3 Error Detection Capabilities

"Error detection on CAN is extremely thorough. Global errors which occur at all nodes are 100% detectable. For local errors (i.e. errors which may appear at only some nodes) the CRC check alone has the following error detection capabilities: Up to 5 single bit errors are 100% detectable, even if the errors are distributed randomly within the code word. All single bit errors are detected if their total number within the code word is odd." [32, Schofield, 2000] Five types of errors can be detected by CAN:

- Bit errors
- Stuff bit errors
- CRC errors
- Format errors
- Acknowledge errors

For details on these errors and information on CAN's Error Confinement, refer to [32, Schofield, 2000].

Stellenbosch University http://scholar.sun.ac.za

A.4 Throughput efficiency

With the help of Figure A.2, the efficiency of one dataframe for CAN v2.0A and CAN v2.0B at 1Mbit/s can be calculated over a period of one second. Some simple assumptions are made about the messages being sent on the bus.

- The given message will never wait in a queue.
- The given message has a bounded size.
- The given message has the highest priority.
- There are no overhead due to errors and retransmissions on the bus in the interval considered.

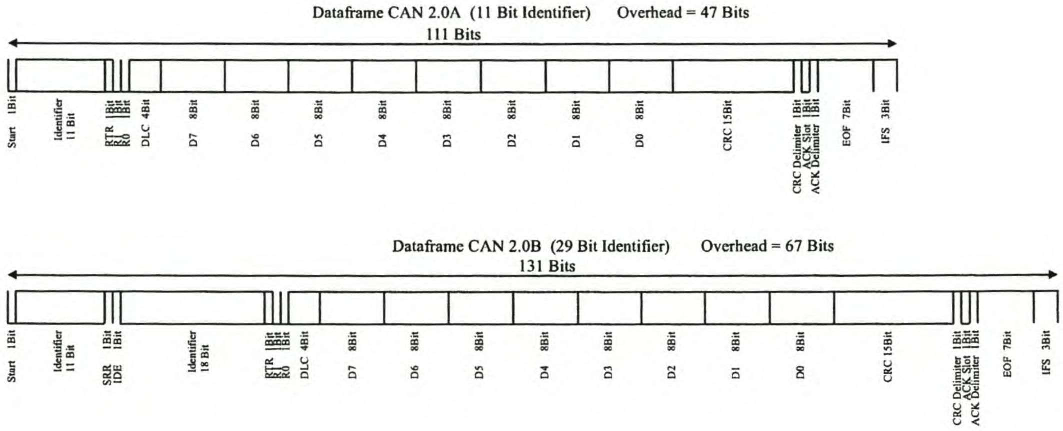


Figure A.2: Standard and Extended CAN Dataframes.

The standard frame has 47 bits overhead which excludes a maximum of 19 stuff bits (see paragraph A.5), and the extended frame has 67 bits overhead which excludes a maximum of 23 stuff bits. In accordance with this, a 57.7% and a 48.9% efficiency can be calculated respectively. In one second, one can thus transmit 72072 bytes of the standard frame format and 61068 bytes of the extended frame format. A similar calculation can be performed for a 125kbit/s data rate. Table A.1 shows a summary of the above-mentioned and Table A.2 can be compiled if the maximum number of possible bit stuffing bits are included in the calculations.

Stellenbosch University http://scholar.sun.ac.za

Table A.1: CAN Dataframe efficiency with bit stuffing *excluded.*

CAN Spec	*Efficiency*	*Data Rate [Bytes/s]*	
		@125kbit/s	*@1Mbit/s*
CAN v2.0A	57.7%	9009	72072
CAN v2.0B	48.9%	7633	61068

Table A.2: CAN Dataframe efficiency with bit stuffing *included.*

		Data Rate [Bytes/s]	
CAN Spec	*Efficiency*	*@125kbit/s*	*@1Mbit/s*
CAN v2.0A	49.2%	7692	61538
CAN v2.0B	41.6%	6493	51948

A.5 Bit Stuffing

After five consecutive identical bits (recessive or dominant) in a bit stream, the controller will insert a complementary bit in place of the fifth bit after which the fifth bit will be sent. This rule only holds for the arbitration, control, data and CRC fields. The controller receiving the message will automatically remove these bits from the bit stream as the bits arrive.

A.6 Calculating and configuring the Bit Timing

The programming of the bit timing depends on the desired baud rate for the CAN network. The *bit time* is subdivided into three segments (Figure A.3 [15, Siemens, 1997:6-90]), each a multiple of the *time quantum*, t_q, with a minimum length of $8t_q$. The synchronisation segment is always one t_q long, while TSeg1 describes the time before the sampling point and TSeg2 describes the time after the sampling point. TSeg1 and TSeg2 have programmable lengths in order to determine the sampling point of the bit time. The bit time is determined by the processor's clock period, CLP, the Baud Rate Prescaler (BRP) and the number of time quanta per bit. The following equations [15, Siemens, 1997:6-91] can be used to program the bit timing.

Stellenbosch University http://scholar.sun.ac.za

$$\begin{aligned}
\text{Bit Time} &= t_{Sync-Seg} + t_{TSeg1} + t_{TSeg2} \\
t_{Sync-Seg} &= 1 \times t_q \\
t_{TSeg1} &= (TSEG1 + 1) \times t_q && \text{TSEG1 min. 3} \\
t_{TSeg2} &= (TSEG2 + 1) \times t_q && \text{TSEG2 min. 2} \\
t_q &= (BRP + 1) \times 2^{(1-CMOD)} \times CLP
\end{aligned}$$

TSEG1, TSEG2 and BRP are programmed by the respective fields of the *Bit Timing Register*. CMOD is a bit in the *SYSCON register* of the processor which must be cleared if the processor's clock has a frequency higher than 10MHz.

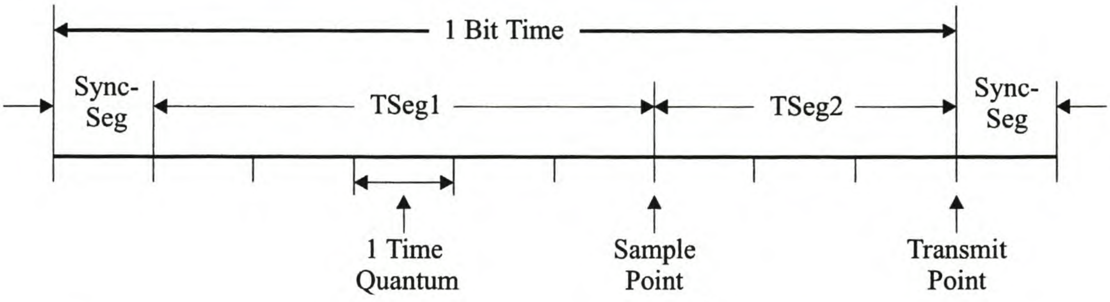


Figure A.3: CAN's bit timing definition. Source: Adapted from Siemens, 1997.

A.7 Synchronisation

When any node receives a data frame or a remote frame, it is necessary for the receiver to synchronise with the transmitter. Because there is no explicit clock signal that a CAN system can use as a timing reference, two mechanisms are used to maintain synchronisation. The first is hard synchronisation and occurs within each receiving controller whenever a falling (recessive-to-dominant) edge is detected during bus-idle time, i.e. at Start-of-Frame (SOF). This is the *only* time when hard synchronisation occurs.

To compensate for oscillator drift, and phase differences between transmitter and receiver oscillators, additional synchronisation is needed. Thus, for subsequent bits in any received frame, if a bit edge does not occur in the synchronisation segment of the bit time, resynchronisation is automatically invoked and will shorten or lengthen the current bit

Stellenbosch University http://scholar.sun.ac.za

time depending on where the edge occurs. The maximum amount by which the bit time is lengthened or shortened is determined by a user-programmable number of time quanta known as the Synchronisation Jump Width (SJW) [15, Siemens, 1997:6-92].

$$t_{SJW} = (SJW + 1) \times t_q \tag{A.1}$$

SJW is also a field in the bit timing register. t_{TSeg1} in fact consists of two fields, the Propogation Time segment and Phase segment1. The Propogation segment compensates for physical delays within the network and Phase segment1 is used for synchronisation. If the edge of the current bit is "late", i.e. it occurs after Synchronisation segment but before the sample point, then Phase segment1 of the current bit is automatically lengthened. If the edge of the next bit is "early", i.e. it occurs during Phase-segment2 (or t_{TSeg2}) of the current bit, then Phase segment2 of the current bit is automatically shortened. For the two baud rates, 1Mbit/s and 125kbit/s, used in this project the following values were assigned to the Bit timing registers.

Table A.3: Assignments to the Bit timing registers.

Baud rate	*Bit timing registers*	*Conditions*

1Mbit/s	BTR0 = 0x40
	BTR1 = 0x25
		f_{OSC} = 20MHz
		f_{CAN} = 10MHz
		CMOD = 0
		SJW = 1
		BRP = 0
		TSEG1 = 5
		TSEG2 = 2

125kbit/s	BTR0 = 0x41
	BTR1 = 0x49
		$f_{OSC} = 16MHz$
		$f_{CAN} = 8MHz$
		CMOD = 0
		SJW = 1
		BRP = 1
		TSEG1 = 9
		TSEG2 = 4

Stellenbosch University http://scholar.sun.ac.za

A.8 What CAN devices are available?

Table A.4: Available CAN devices.

Vendor	*Device*	*Notes*
8-bit Microcontroller with CAN		
Philips	P80C592	ROM-less (PLCC68), 16MHz.
Philips	P87C592	EPROM-based (CLCC68) or OTP (PLCC68), 16MHz.
Philips	P80CE598	ROM-less (PQFP80), 16MHz.
Philips	P87CE598	EPROM-based (CQFP80) or OTP (PQFP80), 16MHz.
Infineon	C505C	2.0B FullCAN controller on an 8bit uP with 15 message objects. Infineon's C500 family extends the power and functionality of the 8051 architecture with greater performance and on chip features integrated. Total redesign of the 8051 architecture that yields substantial increase in performance with clock frequencies of up to 40MHz, lower power and marked EMC improvement.
Infineon	C515C	2.0B FullCAN controller on an 16bit uP with 15 message objects.
Dallas	DS80C390	80C52-compatible CPU, max 40MHz, internal clock multiplier, 4 clocks/machine cycle. Dual CAN 2.0B controllers. 16/32-bit FPU. 256 byte "scratchpad" RAM, 4KB SRAM. Addresses up to 4MB external RAM. Two serial ports. 16 interrupt sources, 6 external. Watchdog. Available in 64-pin QPF and 68-pin PLCC.
Processors with on-chip CAN Controllers		
NS	COP884	2.0A BasicCAN controller. Very low cost uP.
Infineon	SABC167C	2.0B FullCAN controller. Almost identical to the 82527. Messages are odd byte aligned which makes word access difficult.
Infineon	C167CS-32FM	A C167 with two CAN controllers (probably not the TwinCAN module, though).
Infineon	TwinCAN	Two independent CAN modules and gateway capabilities. A total of 32 message buffers which can be concatenated to form a FIFO of various sizes.
Motorola	TOUCAN	2.0A FullCAN controller. Available on a range of 683xxx devices.
Motorola	MSCAN	2.0A BasicCAN controller. Available with a 68HC08 uP
Fujitsu	MB90590	16-bit CPU family, with single voltage flash memory and two CAN controllers on-chip. The MPU core is running at 3V.

Stellenbosch University http://scholar.sun.ac.za

Table A.4: continued.

Vendor	*Device*	*Notes*
Hitachi	SuperH 7055F	Version 2.0B controller on a 32-bit CPU with 32kB RAM and 512kB flash RAM. Dual CAN controllers, with 16 buffers per channel and A/D converter. 256-pin package.
NEC	V850 family	32-bit RISC microcontroller family. Two CAN 2.0B active interfaces (DPRAM type), 50 MHz clock frequency, 1 kByte EEPROM, different memory options (512 kByte flash + 32 kByte RAM, or 256 kByte mask-programmed ROM + 6 kByte RAM), some DSP-like features, 10-bit ADC, etc. The first V85x to use the new CAN module is nick-named ATOMIC and has been presented on the iCC.
Stand-alone CAN Controllers		
Philips	SJA1000	2.0B FullCAN controller. Pin and electrical compatible to the 82C200. Max. baud rate = 1Mbit/s
Infineon	81C90	2.0A FullCAN controller. Able to time-stamp messages.
INTEL	82527	2.0B FullCAN controller. Also in 80196 16 bit uP.
Infineon	SAE81C90	PLCC44
Infineon	SAE81C91	PLCC28
Philips	PCA82C200	DIP28 or SO28
CAN Tranceivers		
Philips	PCA82C250	Standard transceiver, 110 nodes can be connected. For 12V systems in the automotive industry.Max. baud rate = 1Mbit/s
Philips	PCA82C251	Max. baud rate = 1Mbit/s. 24V systems and industrial applications.
Philips	PCA82C252	Low speed fault-tolerant tranceiver. 32 nodes can be connected. Supports unshielded bus wires. High EMC performance. Keeps on operating in a one-wire mode if one of the wires (twisted pair) is cut or shorting. The system operates on relative changes in voltage which implies a higher level of safety. Maximum baud rate = 125kbit/s
Infineon	TLE6252G	"Very similar" to the TJA1053 from Philips. A low-speed fault-tolerant device.
Motorola	MC33388	A CAN driver similar to 82C252 and TJA1053, i.e. low-speed and fault-tolerant.

For more CAN devices, refer to [32, Schofield, 2000] and [22, DETSC, 2000].

Stellenbosch University http://scholar.sun.ac.za

A.9 Serial Linked I/O's

Instead of the conventional all microcontroller-based system, a Serial Linked I/O or SLIO can be introduced into a network. This single chip acts as a dumb input/output interface on a CAN network able to turn messages into digital I/O signals (or switches). Additionally, it can also read I/O pins and transmit the data as a message. Finally, one can make use of its A-to-D converter to generate CAN messages. These devices are extremely cost effective and are ideal for driving remote sensors and actuators which do not need the intelligence from a microcontroller based interface to the CAN network.

The P82C150 is Philips' attempt on making a one-chip CAN node. It has the following features [25, Kvaser, 2000]:

- An integrated clock removes the need for an external crystal.
- 16 configurable digital or analogue I/O pins.
- 3-state outputs available.
- 10-bit A-to-D converter with 6 multiplexed input channels.
- 2 comparators.
- 20kbit/s to 125kbit/s bit rates available with internal oscillator.
- Up to 16 SLIO nodes per network.
- Recovery from bus failure supported.
- Controlled from a single Master.
- Automatic bit-rate detection and calibration.

Although a clever concept in theory, this chip has now a "maintenance status", meaning that it is *not recommended for new designs* [25, Kvaser, 2000]. The question is: "Why did the SLIO fail?" A probable cause can be that it was too simple and thus too difficult to build a system with, for example, advanced self-diagnostics using these chips. There is no real cost advantage either; in reality an inexpensive one-chip device with an on-chip CAN controller proves to be equally cost-effective.

A.10 CAN Interrupts

More than one interrupt can occur for the same CAN message received or transmitted. In order to update the interrupt identifier (INTID) in the Interrupt Register (IR), the interrupt pending (INTPND) bit in Message Control Register of the corresponding message buffer, has to be reset. If further interrupts are pending, the interrupt with the

Stellenbosch University http://scholar.sun.ac.za

next highest priority will appear in the INTID. An INTID code of 00h indicates that all requested interrupts have been correctly serviced and no more interrupts are pending. This should be the condition to leave the CAN interrupt service routine, otherwise the interrupt line to the CPU stays active (the IRQ stays pending) and no further interrupt generation from the CAN module to the CPU can occur.

An INTID value of 01h (highest priority) indicates a status interrupt (if enabled by SIE[1]) or an error interrupt (if enabled by EIE[1]). In the case of a status change due to a successful message transfer, the TXOK or RXOK flag in the CAN Status Register is set. An erroneous message transfer is indicated by the LEC (Last Error Code) bit field in the Status Register. In the case of an error interrupt, at least one of the error flags, EWRN and BOFF, has changed.

An INTID code of 03h to FFh indicates a message specific transmit (TXIE set[2]) or receive (RXIE set[2]) interrupt concerning the message objects 1 to 14 or 'INTID-2' message object. In addition, the global flags TXOK and RXOK can be checked to decide if it was an interrupt on a received or on a transmitted message.

Thus, in order to ensure that all interrupts are correctly serviced, a standard CAN interrupt procedure should respect the following items. Each read operation of the CAN Status Register may have an influence on the INTID value. To avoid errors due to this functionality, this register should be read only at the beginning of the interrupt service routine and then be stored in a variable (e.g. status). Actions, such as testing flags, should only refer to this variable. The same procedure can be used for the variable INTID, which yields the value of the IR. Furthermore, the INTPND bit in the concerned Message Control Register, has to be reset in order to release the interrupt request if it has been activated by one of the message specific bits TXIE or RXIE.

A.11 CAN Remote frames

When a μP requests the transmission of a receive-object, a remote frame will be sent instead of a data frame to request a remote node to send the corresponding data frame. To send a remote frame the Transmit Request (TXRQ) bit[2], simply needs to be set. The New Data (NEWDAT) bit[2] will be set and the TXRQ bit will be cleared by the CAN controller if the data is received before the CAN controller can transmit the remote frame.

On reception of a remote frame with matching identifier and with the message object's Direction (DIR) bit[3] set, indicating transmission, the TXRQ and Remote Pending (RMTPND) bits of this message object are set by the controller. The TXRQ bit will be cleared by the CAN controller along with the RMTPND bit when the message has been successfully transmitted, unless the NEWDAT bit has been set. This bit will be set if the

[1] Bit in the Control Register

[2] Bit in the corresponding Message Control Register

[3] Bit in the Message Config Register (MCFG)

Stellenbosch University http://scholar.sun.ac.za

μP has written new data into the message object while the data was being transmitted. The TXRQ and RMTPND bits will remain in a set state. When the CAN controller stores the remote frame, only the data length code is stored into the corresponding message object. The identifier and the data bytes remain unchanged. Incoming frames can only match with corresponding message objects (standard or extended). Data frames only match with receive-objects and remote frames only match with transmit-objects.

A.12 CAN's CRC

"The CRC for the CAN frame is calculated with a BCH error detecting code best suited for frames with bit counts less than 127." [4, Bosch, 1991:13] The destuffed bit stream consisting of the start of frame, arbitration field, control field, data field and 15 least significant '0' bits, is thought of as a "stream" of serial data bits. The bits in this n-bit block are considered to be the coefficients of a characteristic polynomial, M(X). In this equation, the X-term with the highest exponent has the LSB as coefficient and the X-term with the lowest exponent has the MSB as coefficient. If one or more of the data bits was to change, the polynomial would also change. The CRC is found by applying the following equation:

$$\frac{M(X) \times X^n}{G(X)} = Q(X) + R(X) \tag{A.2}$$

where G(X) is called the generator polynomial, Q(X) the Quotient and R(X) the Remainder. The CRC technique consists of calculating R(X) for the data stream and appending the result to the data field. When R(X) is calculated by the receiver of the data block, the result should be R(X) = 0 indicating that no detectable error were observed. As G(X) is of power 16, the R(X) cannot be of order higher than 15. The CRC can thus be represented by the field given to it in the CAN frame (Figure A.2) irrespective of the length of the data block. CAN's CRC is used in a "detect only" method of error detection and correction. It does not attempt to correct the error, but signals that an error has occurred in the data. The source of the data will then respond to the error signal and retransmit the message which is in error.

A.13 CAN's Bus length

"CAN was originally designed for automotive systems where a bus length would typically be approximately 5 to 10 meters. However, since CAN is now finding its way into many systems where an increased bus length is required, the ultimate transmission range is of increased interest. The standard drivers will be able to drive a bus up to around 1 kilometer." [23, Hitex, 2000] The following figure (Figure A.4 [23, Hitex, 2000]) gives an idea of the expected transfer rate versus transmission length.

Stellenbosch University http://scholar.sun.ac.za

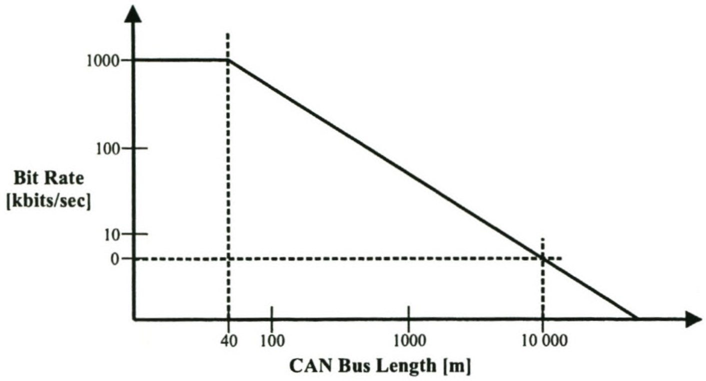


Figure A.4: Projected transfer rate versus transmission length. Source: Adapted from Hitex, 2000.

–ooOoo–

Stellenbosch University http://scholar.sun.ac.za

Appendix B

Project Details

B.1 Scheduling of Data on the ADCS

Table B.1: Dataflow summary of the ADCS.

Source	*Dest.*	*bytes*	*When (Frequency)*	*Notes*
Initialisation of the ICP.				
ICP	MT's	2	Booting the ICP	MT's switched off.
ICP	HSS's	1	Booting the ICP	Select CCD illumination time.
ICP	HSS's	2	Booting the ICP	Selecting pixel 1024 on the CCD.
ICP	UART's	15	Booting the ICP	Configure the 5 UART's.
UART's	ICP's	5	Booting the ICP	Clearing UART buffers.
Reading of MM, HSS's and Suncell Sensors every second.				
ICP	ADC	1	Reading the MM	Put ADC in converting mode.
ICP	ADC	1	Reading the MM	Select MM's X channel.
ICP	ADC	1	Reading the MM	Do a dummy conversion.
ICP	ADC	4	Reading the MM	Select X,Y,Z and T channels respectively.
ICP	ADC	4	Reading the MM	Do conversions.
ADC	ICP	8	Reading the MM	Data read from 12 bit ADC.
ICP	ADC	1	Reading the MM	Place ADC in a Sleep mode.
HSS's	ICP	6	Reading the HSS's	Read 11 bit HSS values.
ICP	ADC16	32	Reading the RSS's	Write addresses of ADC channels to be converted.
ADC16	ICP	16	Reading the RSS's	Read the converted values.

Stellenbosch University http://scholar.sun.ac.za

Table B.1: continued.

Source	*Dest.*	*bytes*	*When (Frequency)*	*Notes*
Commanding the RW controllers every second.				
ICP	UART's	8	Commanding RWC's	Send speed reference values when RW's are active.
ICP	UART's	24	Commanding RWC's	Reset UART's. [1]
UART's	ICP	8	Commanding RWC's	Reset UART Buffers. [1]
ICP	UART's	2	Commanding RWC's	Send End of Message flag.
ICP	UART's	6	Commanding RWC's	Reset UART's. [1]
UART's	ICP	8	Commanding RWC's	Reset UART Buffers. [1]
UART's	ICP	2	Commanding RWC's	Reset UART Buffers.
ICP	UART's	2	Commanding RWC's	Enable UART's to receive data.
UART's	ICP	18	Commanding RWC's	Read data from the RWC's. ((8bytes+EOM)*2Msgs)
ICP	UART's	2	Commanding RWC's	Disable UART's to receive data from the RW's.
Commanding the MT's every ten seconds. [2]				
UART	ICP	6	Commanding MT's	Updating MT counters.
The transmission of Sensor data every second.				
ICP	UART [3]	1	TX Sensor data	Send Message length.
ICP	UART('s)	3	TX Sensor data	Reset UART's. [1]
UART('s)	ICP	1	TX Sensor data	Reset UART Buffers. [1]
ICP	UART [3]	1	TX Sensor data	Send Message type.
ICP	UART('s)	3	TX Sensor data	Reset UART's. [1]
UART('s)	ICP	1	TX Sensor data	Reset UART Buffers. [1]
ICP	UART [3]	46	TX Sensor data	Send Sensor data.
ICP	UART('s)	3	TX Sensor data	Reset UART's. [1]
UART('s)	ICP	1	TX Sensor data	Reset UART Buffers. [1]
ICP	UART [3]	1	TX Sensor data	Send End of Message.
ICP	UART('s)	3	TX Sensor data	Reset UART's. [1]
UART('s)	ICP	1	TX Sensor data	Reset UART Buffers. [1]

[1] If time-out occured.

[2] See the "10ms timer interrupt" summary below.

[3] UART 1, 2 and/or 3.

Stellenbosch University http://scholar.sun.ac.za

Table B.1: continued.

Source	*Dest.*	*bytes*	*When (Frequency)*	*Notes*
The Receive Message Subroutine. (Frequency is seldom.)				
ICP	EEPROM	32	RX Mesg Routine	EEPROM page Programming.
ICP	UART	1	RX Mesg Routine	Acknowledge that EEPROM programming is completed.
ICP	HSS's	2	RX Mesg Routine	Selecting a pixel on the CCD's.
ICP	HSS's	1	RX Mesg Routine	Selecting illumination time.
The 10ms Timer Interrupt.				
ICP	MT's	3	Commanding MT's [1]	Configure MT's polarity and pulsewidth.
ICP	MT's	3	Commanding MT's [1]	Disable MT's if counters are zero.
The Message Received interrupts. (Every second.)				
UART's	ICP	3	Message Received	Test which UART has sent the message.
UART's	ICP	18	Message Received	Read the data from the UART. (10 byte RW ref. and 8 byte MT cmnd., 36 bytes max.)
The Imager Interrupts. (Seldom)				
ICP	RAMTRAY	4	Image taken	For new imager data valid.
The Star Sensor data. (Every second via a transputer link.)				
ACP	SCP	20	Star Sensor active	[1234567890] of type CARDINAL send for synchronisation.
ACP	SCP	24	Star Sensor active	Kepler elements.[2]
SCP	ACP	20	Star Sensor active	[1234567890] of type CARDINAL send for synchronisation.
SCP	ACP	2	Star Sensor active	Indicates what the total number of matching stars was. (totnofmatches)
SCP	ACP	72	Star Sensor active	Bytes = totnofmatches* (2vec*3comp*4bytes) where the totnofmatches = [2,3].

[1] See "Commanding the MT's every ten seconds" summary above.

[2] Satellite Orbital elements are sent to the SCP when it is initialised.

Stellenbosch University http://scholar.sun.ac.za

B.2 Fieldbus Protocols Compared

The following table was extracted from the work done by Lawrenz [13, Lawrenz, 1997:15-25].

Table B.2: A comparison of some fieldbus protocols.

Architecture and Hardware	*Communication Technique*	*Notes*	*Problems*
Process Field Bus Fieldbus Message Specification (Profibus FMS)			
Multi master. Shielded twisted pair with RS485 tranceivers.	Token passing. Slaves can only respond and acknowledge	Nodes must broadcast their functionality during initialisation. Designated for unintelligent sensors and actuators.	Requirement for intelligent nodes cannot be complied to.
Interbus S			
Single-master-multiple-slave. 5 wire bus with 64 nodes (max.)	Cyclic in a ring network at 16bits/cycle. Protocol overhead is low.	Nodes are identified in an identification cycle, and are addressed by there physical location in the network.	Sensitive to failure of any node in the network.
Bitbus			
Single-master-multiple-slave. 30 Slaves (max.)	Slaves are polled by the master	Based on the 8051 architecture with a protocol handler. IEEE approved protocol.	Large frame overhead. Operation is dependant on a single master. Event driven communication is difficult to achieve.
Factory Intrumentation Protocol (FIP)			
Single-master-multiple-slave. 32 Nodes (max.)	Cyclic polling based on a predefined list.	The source and destination addresses are specified by a 24bit field. Data frames can be as long as 128 bytes. Semi-event driven communication is possible.	Protocol efficiency is low for short messages. Nodes must broadcast their functionality during initialisation.

Stellenbosch University http://scholar.sun.ac.za

Table B.2: continued.

Architecture and Hardware	*Communication Technique*	*Notes*	*Problems*
Local Operating Network (LonWorks)			
Single master	Polling. Data exchanged is an analog of variable exchange in the software environment.	Communication details are hidden from the application.	Message collisions and event driven communication cannnot be managed properly. Operation is dependant on a single master. The development language is customized for this protocol. Thus, proliferation can become a problem.
Process Network (PNET)			
Master-Slave. 125 nodes (max.) A node's UART port interfaces to the network. The network is a closed loop. Tranceivers are RS485 devices. The bit rate is 76.8 kbits/s.	Cyclic polling of slaves.	Implemented in software and based on an exchange of variables. Multiple master-slave sets can exist in the same network while a token is passed to the master in control.	Development is in uniquely defined "Process-Pascal" software. Events are not possible.
European Installation Bus (EIB)			
Multiple master. 64 nodes (max.) Transmission is based on the UART of Motorola's 68HC05 processor. The bit rate is 9600bits/s via a 4 wire bus.	Any node can be the master if the bus is idle.	The destination address field can be one or multiple addresses. Bitwise arbitration determines the master in case of collisions.	High power dissipation, cost and availability of hardware.

Stellenbosch University http://scholar.sun.ac.za

Table B.2: continued.

Architecture and Hardware	*Communication Technique*	*Notes*	*Problems*
Actuator/Sensor-interface (ASI)			
Master-slave protocol. 31 slaves (max.) Bus consisting of a 2 wire flat cable that includes a power supply line for the nodes. The bit rate is 167 kbits/s.	Cyclic polling of slaves.		Events are impossible. High cost.
Controller Area Network (CAN)			
Multiple master. 32 nodes (max.) with standard line drivers. Twisted pair media. 1Mbit/s (max.) communication rate. Rate depends on the lenght of the bus.[1]	Cyclic driven communication or event driven communication is possible.	Remote request facility. Automatic retransmission of message that lost arbitration or that were corrupted.	Maximum data length of 8 bytes.

Furthermore, an article by Dunne [5, Dunne, 1998] presented at the 5th International CAN Conference reported on a feasibility study of 22 different fieldbus protocols with the purpose of implementing one for an emergency light test network. These protocols include LonWorks, CAN, SDS, DeviceNet, Profibus FMS, ASI, Interbus S, FIP, EIB, Bitbus, and PNET.

Aspects such as cost, transmission media, performance, development time, commercial availability, and available support tools were considered. The Bitbus, although being relatively well established and stable, lacked in support and availibility was poor. LonWorks, on the other hand, has a limited range of controllers and the costs of development tools were too high. DeviceNet and SDS were considered, but rejected due to a lack in flexibility of the application layer. DeviceNet only supports three baud rates, namely 125kB,

[1] For a 50m bus, the maximum bit rate is 1Mbit/s, and for a 500m bus the maximum bit rate is 100kbit/s.

Stellenbosch University http://scholar.sun.ac.za

250kB and 500kB.

According to the article, protocols were tailored for specific market sectors, particularly at the application layer. Networks were also found to dominate in a national fashion with only LonWorks and CAN having penetrated the global market. Of all the above mentioned fieldbus protocols in this article, CAN was chosen for its:

- proliferation of technology and development software;
- inexpensive node design;
- robust nature of the protocol;
- flexibility with regard to bit rates and data field content; and
- arbitration and error detection capabilities.

B.3 The CPLD's Thermal Analysis

The EPM7128STC100 CPLD has an application note warning that the thermal analysis of the device has to be completed before the actual implementation, in order to prevent it from exceeding the device's maximum allowed junction temperature. The following calculations, divided into three steps, were completed with the help of the Altera application notes [2, AlteraCD, 1998].

- Estimating the power consumption of the application.
- Calculating the maximum power for the device.
- Comparing the estimated and the maximum power values.

B.3.1 Estimated Power Consumption (P_{EST})

The estimated power consumption is given by the following equation,

$$P_{EST} = P_{INT} + P_{IO} = (I_{CCINT} \times V_{CCINT}) + (P_{ACOUT} + P_{DCOUT}) \qquad \text{(B.1)}$$

where P_{INT} is the no-load power and P_{IO} is the power dissipated by the I/O buffers. I_{CCINT} is given by $I_{CCINT} = A \times MC_{TON} + B \times (MC_{DEV} - MC_{TON}) + C \times MC_{USED} \times f_{MAX} \times tog_{LC}$

Thus $P_{INT} = 121.13mA \times 5V = 605.65mW$. P_{DCOUT}, power dissipation by the I/O buffers during steady-state, depends on the logic levels that the outputs have to drive and the resistive load on each output. The power that is dissipated by outputs which drive

Stellenbosch University http://scholar.sun.ac.za

$\mathbf{MC_{TON}}$ = Number of macrocells (LE) with the Turbo Bit option turned on, as reported by the MAX+PlusII Report File.

$\mathbf{MC_{DEV}}$ = Number of macrocells (LE) in the device.

$\mathbf{MC_{USED}}$ = Total number of macrocells (LE) in the design, as reported in the MAX=PlusII Report File.

$\mathbf{f_{MAX}}$ = The highest clock frequency (MHZ) in the device.

$\mathbf{tog_{LC}}$ = Average ratio of logic cells toggling at each clock (typically 0.125).

$\mathbf{A, B, C}$ = MAX 7000 I_{CC} equation constants given by the data sheet. $A = 0.93mA/LE$, $B = 0.4mA/LE$, $C = 0.04mA/(MHz.LE)$

$$\begin{aligned} I_{CCINT} &= 0.93{\times}111{+}0.4{\times}(128{-}111){+}0.04{\times}111{\times}20{\times}0.125 \\ I_{CCINT} &= 121.13mA \end{aligned}$$

CMOS devices, is negligible. However, the outputs driving TTL and resistive loads can not be ignored. Pull-up resistors, according to the application note, with a value of 1k, will dissipate 0.49mW for a low output. Five resistors, with values of 4k7 each, will be used as pull-up resistors in this design, which implies an estimated DC-power dissipation of approximately 0.5mW in total. This is the only dc-load that needs to be taken into account.

P_{ACOUT}, the power dissipated by frequently switching outputs, depends on the capacitive load on each output and the frequency at which each output switches. The application note gives an equation

$$P_{ACOUT} = 0.5 \times OUT \times C_{AVE} \times V_O \times f_{MAX} \times tog_{IO} \times V_{CCIO}) \tag{B.2}$$

for P_{ACOUT} for the average capacitive load, where OUT is the total number of output and bidirectional pins. V_O is given as 3.8V by the data sheet for $V_{CCIO} = 5V$. With a total of 30 output and bidirectional pins, and by taking the average capacitive load as 35pF as an estimation on the basis of an example in the application note, one can calculate an estimated P_{ACOUT} of 24.9mW.

The estimated power consumption (P_{EST}) thus equals 631mW.

Stellenbosch University http://scholar.sun.ac.za

B.3.2 Maximum Power for the Device and Package (P_{MAX})

The following equation is used to calculate the maximum allowed power for the device.

$$P_{MAX} = \frac{T_j - T_A}{\theta_{jA}} \tag{B.3}$$

With a maximum allowed junction temperature T_j of 90°C, an ambient temperature of operation of 70°C and a junction to ambient thermal resistance θ_{jA} of 10°C/W (still air), one can calculate the maximum allowed power for the device as 2W. If the ambient temperature should be lower than the above value, it only suggests that the maximum allowed power can be increased.

B.3.3 Comparing the Estimated and the Maximum Power Values

Comparing the estimated power consumption with the maximum allowable device power, one can clearly see that $P_{EST} < P_{MAX}$.

B.4 Calculating the Size of the Program Memory

Taking an abstract from a typical C-program, one can calculate what the averages are for bytes per instruction and cycles per instructions. The worst case for the average number of times instructions are repeated per second can be calculated from the assembler code of the ICP on the ADCS of SUNSAT I.

B.4.1 Bytes per Instruction

It was found from the C-code that:
38 instructions are 1 byte in size - 10% of the abstract;
72 instructions are 2 bytes in size - 41% of the abstract;
57 instructions are 3 bytes in size - 49% of the abstract;

From this, it was calculated that the average instruction is 2.39 bytes in size.

B.4.2 Cycles per Instruction

From the same C-code it was found that:
68 instructions are 1 cycle long - 40% of the abstract;

Stellenbosch University http://scholar.sun.ac.za

97 instructions are 2 cycles long - 60% of the abstract;

From this, it was calculated that the average instruction is 1.6 cycles long.

B.4.3 Average Number of Times that Instructions are Repeated per Second

The ICP's assembler code was analysed to see how many times the average instruction is repeated. Each subroutine consists of a number of instructions, some of which are repeated in loops. The worst case for the number of instructions executed in these loops are calculated. Repeating this exercise for all of the routines in the code (Table B.3), adding the values and dividing the totals, one can calculate the worst case average repetition for an instruction in the ICP's code.

Table B.3: ICP code analysed.

Routines	*Instructions/second*	*Instructions*
10ms Timer Interrupt	9300	146
UART Receive Interrupts	393	121
Sensor Read Subroutine	291	73
Transmit Sensor Data	11347	58
Receive Message Subroutine	119	170
RW control Subroutine	2362	116
MT control Subroutine	105	105
Main	62500	20
Initialise[1]	-	48
Imager Interrupt[2]	-	18
UART Reset[3]	-	14
Totals	86417	889

The worst case average number of times an instruction is repeated in the ICP code was calculated to be 86417/889 or 97.2 times in a second.

[1] Seldom executed.

[2] Seldom executed.

[3] Included in subroutines above.

Stellenbosch University http://scholar.sun.ac.za

B.5 System Power Consumption

The estimated maximum power consumption of the complete dual CAN node can be calculated by drawing up a table (Table B.4) for the estimated maximum current consumptions for each device on the board.

Table B.4: The estimated power consumption of the dual CAN node's evaluation version.

Quantity	*Device*	$I_C Cmax$ *[mA]*	*Total [mA]*
2	SAFC505CLM-AB	32	64
2	74HC573	0.16	0.32
2	27C256	30	60
1	74HC14	0.04	0.04
1	EPM7128STC100-15	100	100
2	PCA82C250	70	140
	(Standby mode)	0.17	
	(Dominant bit TX)	70	
	(Recesive bit TX)	16	
Total maximum current consumption, MAX232 and LED's excluded.			364.36
1	MAX232A	10	10
15	High brightness LED's	30	450
Total maximum current consumption of the devices.			824.36
Plus 20% for resistors and the oscillator.			165
Total maximum current consumption required.			990

For a standard 7805 (positive 5V) regulator, the minimum supply voltage should be 7.5V. Choosing a supply voltage of 8V and an estimated output current of 1A as calculated in Table B.4, one can calculate the maximum power that the regulator has to dissipate.

$$P_{DEV} = \frac{V_{SUP}-V_{OUT}}{I_{OUT}} = 3W$$

The maximum junction temperature, T_{JMAX}, of the regulator, according to its data sheet, is 125°C/W. With thermal resistances given by Table B.5, and an ambient temperature, T_A, of 25°C, one can calculate the maximum power that the device can dissipate.

$$P_{DMAX} = \frac{T_{JMAX}-T_A}{\theta_{JA}} = \frac{T_{JMAX}-T_A}{\theta_{JC}+\theta_{CS}+\theta_{SA}} = 7W$$

The device can dissipate up to 8W with a 10°C/W heatsink and an ambient temperature of 25°C. Working back from this maximum power for the device, one can calculate that the maximum supply voltage to the regulator can be as high as 12V.

Stellenbosch University http://scholar.sun.ac.za

Table B.5: Thermal resistances for calculating maximum power of the regulator.

Thermal Resistance	*Symbol*	*Value* [°C/W]
Junction to case	θ_{JC}	4
Case to sink	θ_{CS}	0.3
Sink to ambient	θ_{SA}	9.9

B.6 Test and Evaluation Setup

The design phase of the dual CAN node is completed with the successful integration of the software with the hardware and the integration of the nodes into the demonstration setup. During this phase, it was ensured that the individual modules were running at their prescribed rate and that they were functioning correctly. For this purpose a CAN-ISA-bus adapter, or PCCAN card (See Appendix C), was purchased to serve as the initial interface to the first dual CAN node. It proved to be an invaluable tool during software development and debugging. Then the modules were added to the network one by one making sure the system still performed as expected every time. The PCCAN card then served as diagnostic node for the demonstration model. The final test setup is illustrated in Figure B.1. Note that two alone-standing PC's were used because the PCCAN card used Windows-based software, while the PC that interfaced to the dual CAN node via its EPP, used DOS-based software and interrupts. The two PC's furthermore protected the integrity of the demonstration.

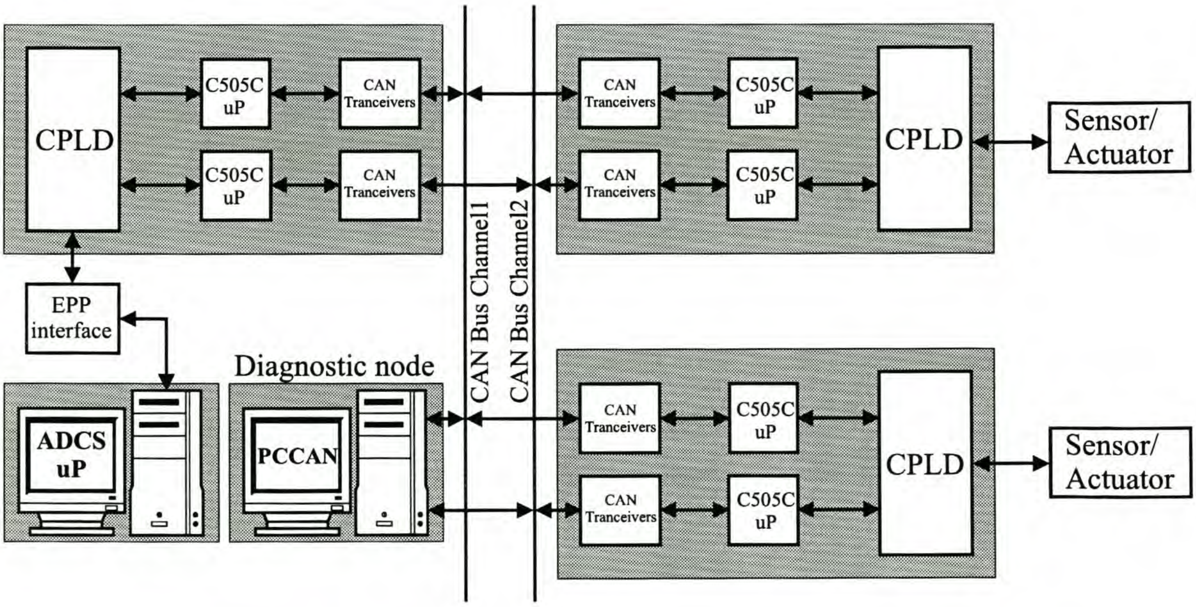


Figure B.1: Illustration of the test setup for the dual CAN network, its nodes and diagnostic interface.

Stellenbosch University http://scholar.sun.ac.za

B.6.1 AC Characteristics for the Dual CAN Node

Table B.6 gives the detailed timing information for the Master-Slave and Master-Checker configurations at baud rates of 125kbit/s and 1Mbit/s respectively. This table also forms the detailed legend for Figure 4.3 in paragraph 4.1. Note that the values that are absent from this table do not exist. These values are irrelevant for the configuration being tested. Also note that these values represent the *averages* of parameters that were measured.

Table B.6: A.C. Characteristics

Symbol	*Parameter*	*Average time*				*Units*
	Configuration	*Master-Slave*		*Master-Checker*		
	Oscillator frequency	8	20	8	20	MHz
	Baud rate	125k	1M	125k	1M	bit/s
t_{RLM}	Reset low to start of the Main-loop	446	177	444	180	μs
t_{RLF}	Reset low to start of the FOREVER-loop	1174	472	1270	512	μs
t_{RLCI}	Reset low to start of CAN init.	511	184	514	187	μs
t_{CIC}	CAN initialisation cycle	84.8	36	91.2	37.5	μs
t_{AOSC}	ADCS one second cycle	1	1	1	1	s
t_{SPD}	Second start to processing done					
	Master	16.5	4.0	17.5	4.6	ms
	Slave			15.2	4.0	ms
t_{SEDU}	Second start to data updated	15.9	3.7	17.0	4.3	ms
t_{SM}	Send message across network (send_mx( ))	11.9	4.2	106	44	μs
t_{SMTO}	Send message across network with time-out	2040	515	2035	505	μs
t_{SMTP}	Send message to twin processor			74.8	29	μs
t_{IISR}	Interrupt to twin processor			13.4	6.3	μs
t_{ITHT}	Interrupt to twin processor with HW time-out			16.3	12.8	μs
t_{ITST}	Interrupt to twin processor with SW time-out			50.2	35	μs

Stellenbosch University http://scholar.sun.ac.za

Table B.6: continued.

Symbol	*Parameter*	*Average time*				*Units*
	Configuration	*Master-Slave*		*Master-Checker*		
	Oscillator frequency	8	20	8	20	MHz
	Baud rate	125k	1M	125k	1M	bit/s
t_{ITPM}	ISR in twin μP					
	Master receives code byte			12.8	5.6	μs
	Master receives data byte			32	13.6	μs
	Slave receives code byte			9.6	12.2	μs
	Slave receives data byte			46	19.7	μs
t_{IEEM}	ISR done to end of echo-message( )			81.4	49.5	μs
t_{EMAC}	Echo message across CAN network			69.8	10.4	μs
t_{RQBS}	TXRQ to start of CAN bitstream	16.6	16.6	16.6	16.6	μs
t_{CBI}	CAN bitstream start to ISR.[1]	980	78	984	77.1	μs
t_{CISR}	CAN ISR					
	Remote frame received	31	13.2	42.1	17.1	μs
	Standard frame received	59.2	24.4	70.4	28.5	μs
t_{PCM}	Processing the CAN messages					
	Master received message	39.2	15.9	106	58	μs
	Slave received message			120	47	μs
	Master transmitted message	20.0	8.4	21	8.4	μs
	Slave transmitted message			21	8.4	μs
t_{IPD}	ISR start to processing done					
	Master received message	91.1	54.8	205	122	μs
	Slave received message			256	82	μs
	Master transmitted message	86.4	41.5	138	41.8	μs
	Slave transmitted message			124	47.5	μs
t_{PDNI}	Processing done to next ISR					
	Master received message	637	89.6	737	126.6	μs
	Slave received message			890	193.5	μs
	Master transmitted message	879	58.4	957	104.8	μs
	Slave transmitted message			1346	104.5	μs

[1] STD frame with 8 bytes.

Stellenbosch University http://scholar.sun.ac.za

Table B.6: continued.

Symbol	*Parameter*	*Average time*				*Units*
	Configuration	*Master-Slave*		*Master-Checker*		
	Oscillator frequency	8	20	8	20	MHz
	Baud rate	125k	1M	125k	1M	bit/s
t_{PCD1}	Processing start to comparison of M and C message done.[1]				49.4	μs
t_{CVC1}	Checker's message valid to end of comparison.[1]				5.9	μs
t_{CMTO}	compare_message( ) start to time-out on the Slave's message			194	107	μs
t_{PCD2}	Processing start to comparrison of M and C message done.[2]			10.4	2.2	μs
t_{CVC2}	Checker's message valid to start of comparison.[2]			242	11.2	μs
t_{CMCC}	compare_message() start until checker sends CRC to the Master			88.6	36	μs
t_{SMOK}	Send message to sending next message	1400	219.5	1486	306	μs
t_{WBAR}	Writing a byte to the App. until the App. reads it.	6.12	5.5	12.7	3.85	μs
t_{WBAH}	Writing a byte to the App. until hardware time-out	16.4	12.8	20.2	12.8	μs
t_{WBAS}	Writing a byte to the App. until software time-out	186	77.8	75.6	77.6	μs
t_{TMA}	Transmitting a message to the App.[3]	26.4	27.3	26.2	24.3	μs
t_{WCDA}	Write collected data to application[4]	444	430	447	370	μs
t_{IAI}	Interrupt from App. to ISR	16	15.9	16	15.9	μs
t_{IIB}	Inside ISR to receive one byte	27.5	12.0	30.0	12.4	μs
t_{IMR}	First interrupt to whole message received (9 bytes)	566	18.1	577	25.8	μs

[1] Checker's message received after start of compare_message( ).

[2] Checker's message received before start of compare_message( ).

[3] A message consists of one header byte and the data byte.

[4] This data includes 14 bytes sensor data and one header byte.

Stellenbosch University http://scholar.sun.ac.za

B.6.2 Parallel Port Interface

Figure B.2 illustrates the interface circuit used to protect the PC's parallel port. Notice how delays were inserted into the signals to generate a valid signal for the bidirectional buffer's directional input.

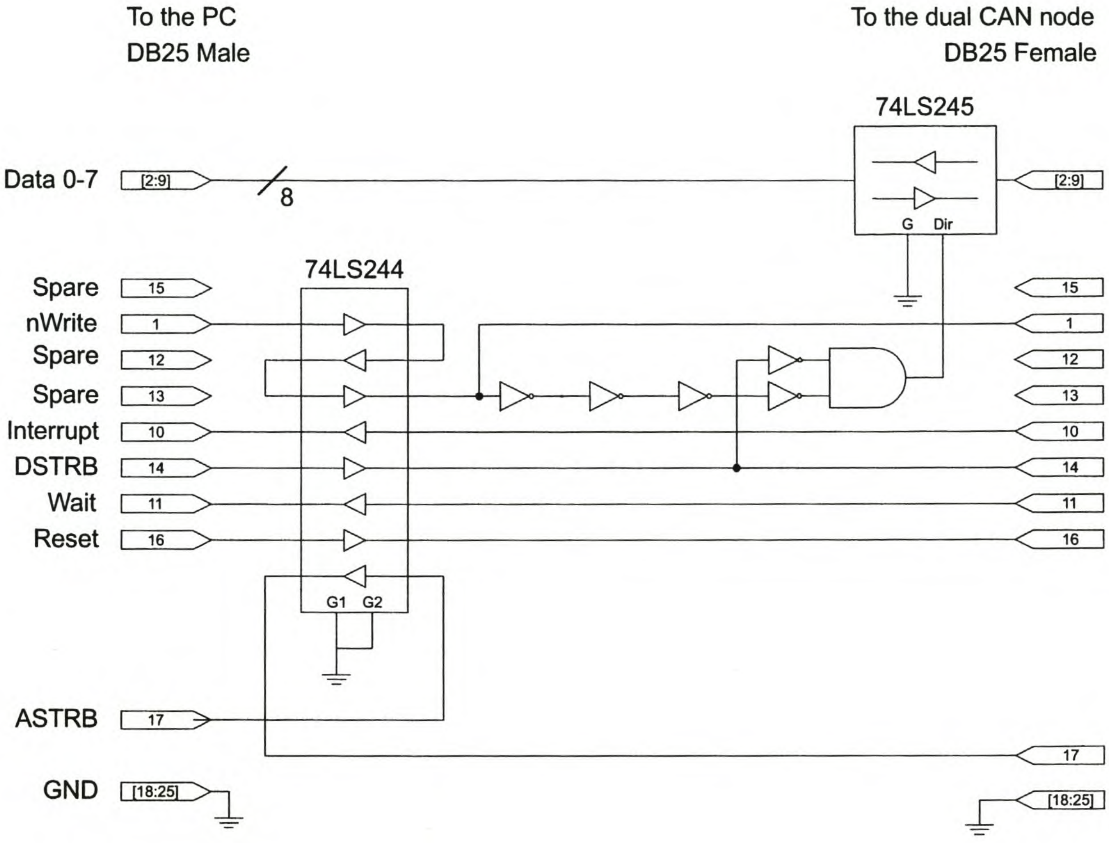


Figure B.2: The interface circuit with the purpose of protecting the PC.

B.6.3 PCCAN - Note of Warning

Since any node, even the PCCAN card, can acknowledge a CAN message, seeing the message on the PC's monitor does not necessarily mean that the destination of the message has received the message. The destination may be malfunctioning.

Stellenbosch University http://scholar.sun.ac.za

B.6.4 Telecommand Reset Pull-up Resistor

Resistor R56 of the dual CAN node was changed from a 1k value to a 4k7 value because it seemed that the EPP's reset could not pull the signal low in order to reset the dual CAN node. The 1k resistor was too small.

B.6.5 PC's used as Interface to the Dual CAN Node

PC's running Windows NT or WIN98 operating systems or newer version may not be able to communicate with the EPP in the customised way required by this design unless specialised software is used. WIN95 and older versions with a BIOS and hardware which allows access to the EPP with specifications 1.7 or 1.9, still provide functionality without any specialised software. Therefore, care must be taken when setting up the test setup. For EPP 1.7 the Data Strobe signal will be inserted independently of the state of the Wait signal; whereas for EPP 1.9 the Data Strobe will only be inserted if the Wait signal is low. Both standards however require that the Wait signal be high to finish the cycle. (Refer to paragraph 3.9.2.)

–ooOoo–

Stellenbosch University http://scholar.sun.ac.za

Appendix C

PCCAN

The PCCAN card is a CAN interface for the ISA bus [10, PCCAN, 1998]. The card has four CAN controllers: two Intel 82527 and two Philips 82C200 controllers. The board occupies 64 bytes of I/O space and the base address can be any one of 200H, 240H, 300H, and 340H. The interrupt signal on the CAN controllers can be connected - under software control - to any one of IRQ2/9, IRQ3, or IRQ5 on the ISA bus. It is also possible to choose between the PC bus clock and an oscillator clock, located on the card, to clock the CAN circuits. The original 16 MHz oscillator may be replaced by another oscillator of a frequency in the range of 1 - 24 MHz if this is required to obtain the required bit rate on the bus. The PCCAN board draws approximately 600 mA at 5V.

The outputs from the CAN circuits are connected to a 25 pin female DSUB via optocouplers and bus drivers (Philips 82C251, conforming to ISO 11898.) It is possible to connect all 4 CAN in/outputs on the card to a common CAN bus, which also is connected to the CAN-connector, but they can also be used separately. An on-board terminating resistor of 120Ω may be connected to the common bus in order to match the cable impedance and suppress noise at recessive level. The 25 pin connector provides the voltage feed for the bus section (optocouplers and drive circuits). Alternatively, the PC is used as a power source; however this gives no galvanic separation between the computer and the CAN bus. Switch3 makes it possible to connect the grounds from the drivers to the ground of the PC.

Note: The two Intel CAN controllers on the PCCAN card do not respond to remote frames, since no drivers were developed for them when this card was purchased. However, the KVASER web-site [24, KVASER, 1999] may be approached to see or enquire about updated drivers.

Stellenbosch University http://scholar.sun.ac.za

C.1 PCCAN's Interframe Spacing

If one sends messages to all four of the CAN channels on the PCCAN card via the ISA bus, one will notice (Figure C.1) a distinct difference in the interframe space of the different CAN channels. This is not due to the CAN protocol or hardware, but this is as a direct result of the hardware responsible for servicing the interrupts on the CAN card. Since all four the channels will generate interrupts at the same time when a message is received, the driver will service the interrupts in an order which is hardwired in the circuit, starting with Intel1, Intel2, Philips1 and finally Philips2.

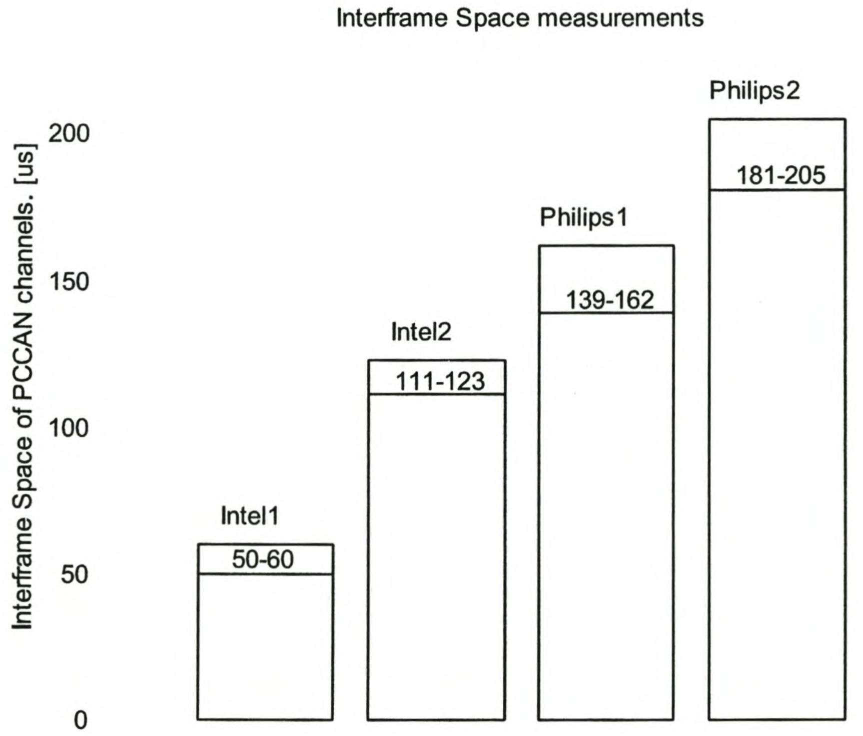


Figure C.1: An approximation of the interframe spaces of the four CAN circuits on the PCCAN card.

–ooOoo–

Stellenbosch University http://scholar.sun.ac.za

Appendix D

Dual CAN node Schematics and PCB layout

D.1 Schematics

Stellenbosch University http://scholar.sun.ac.za

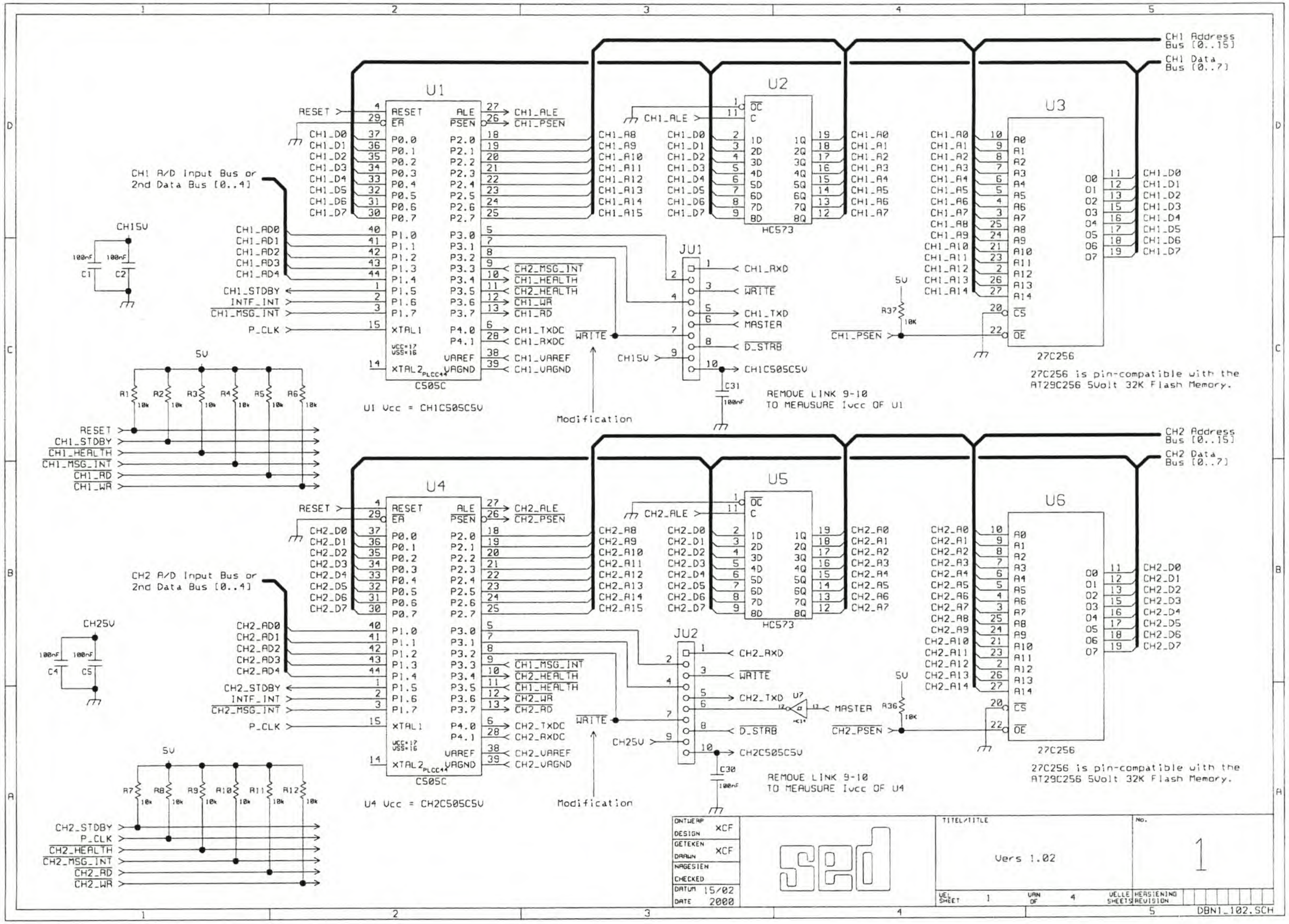


Figure D.1: Dual CAN node schematics - Page 1

Stellenbosch University http://scholar.sun.ac.za

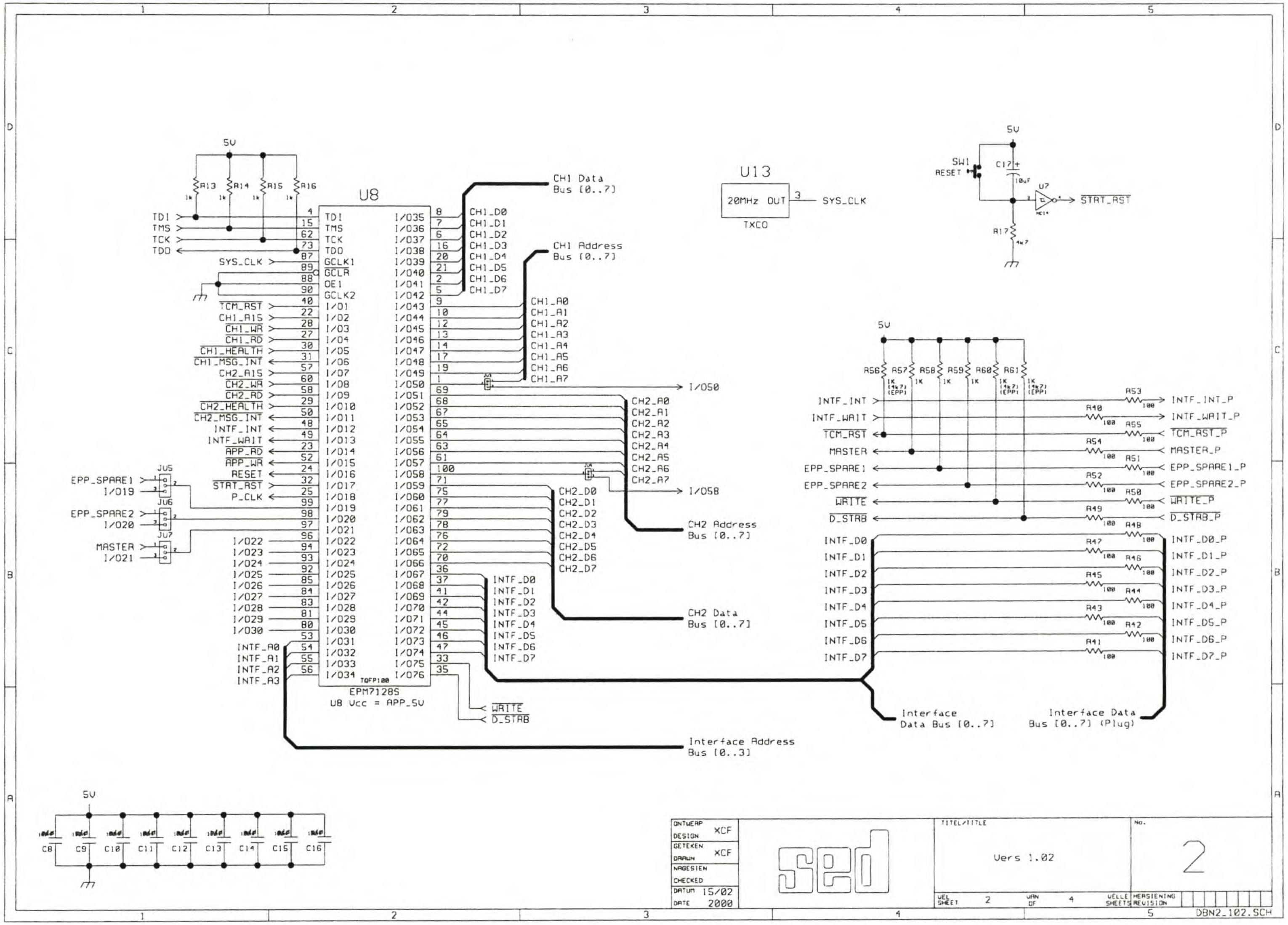
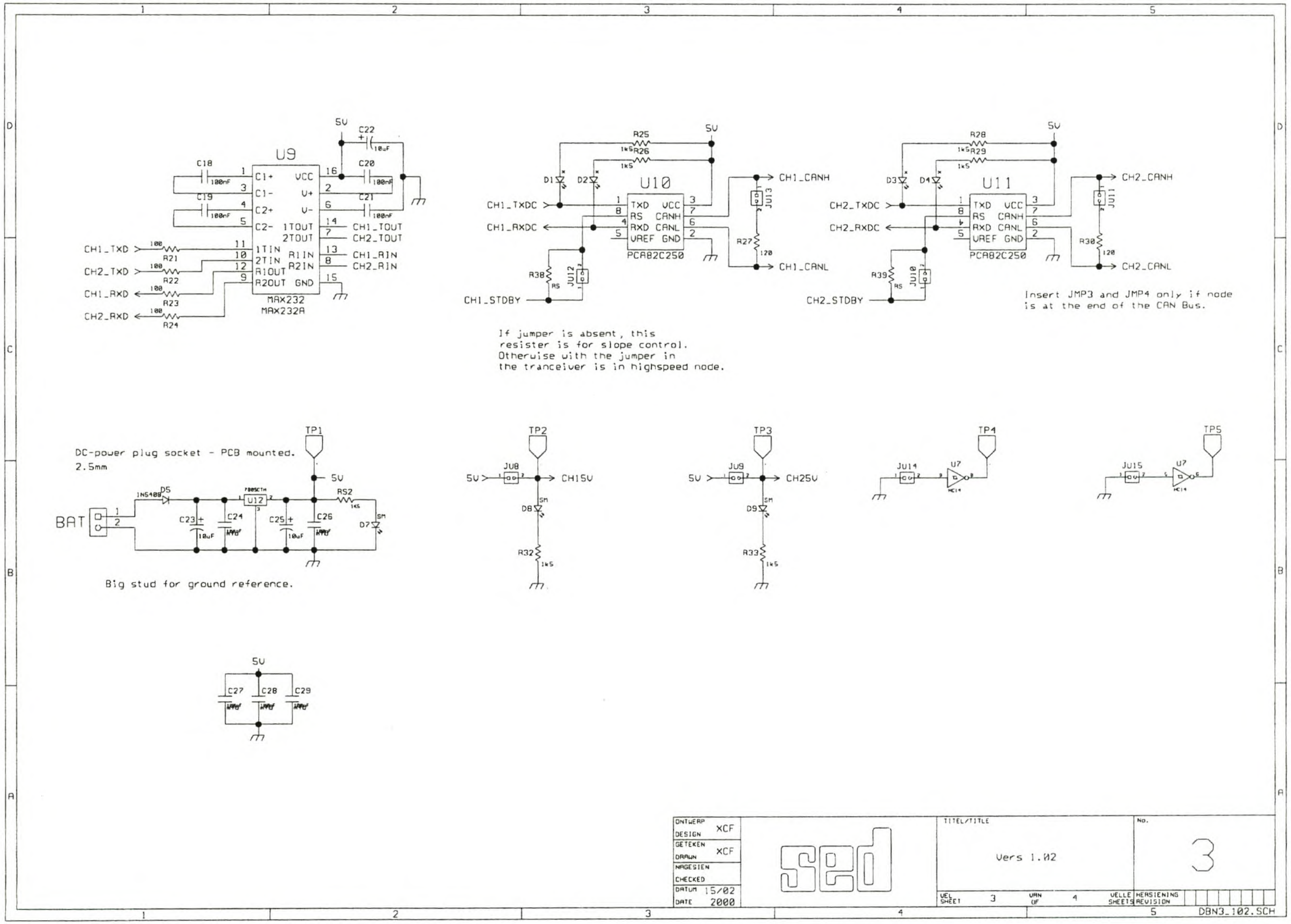


Figure D.2: Dual CAN node schematics - Page 2

Stellenbosch University http://scholar.sun.ac.za



If jumper is absent, this resister is for slope control. Otherwise with the jumper in the tranceiver is in highspeed node.

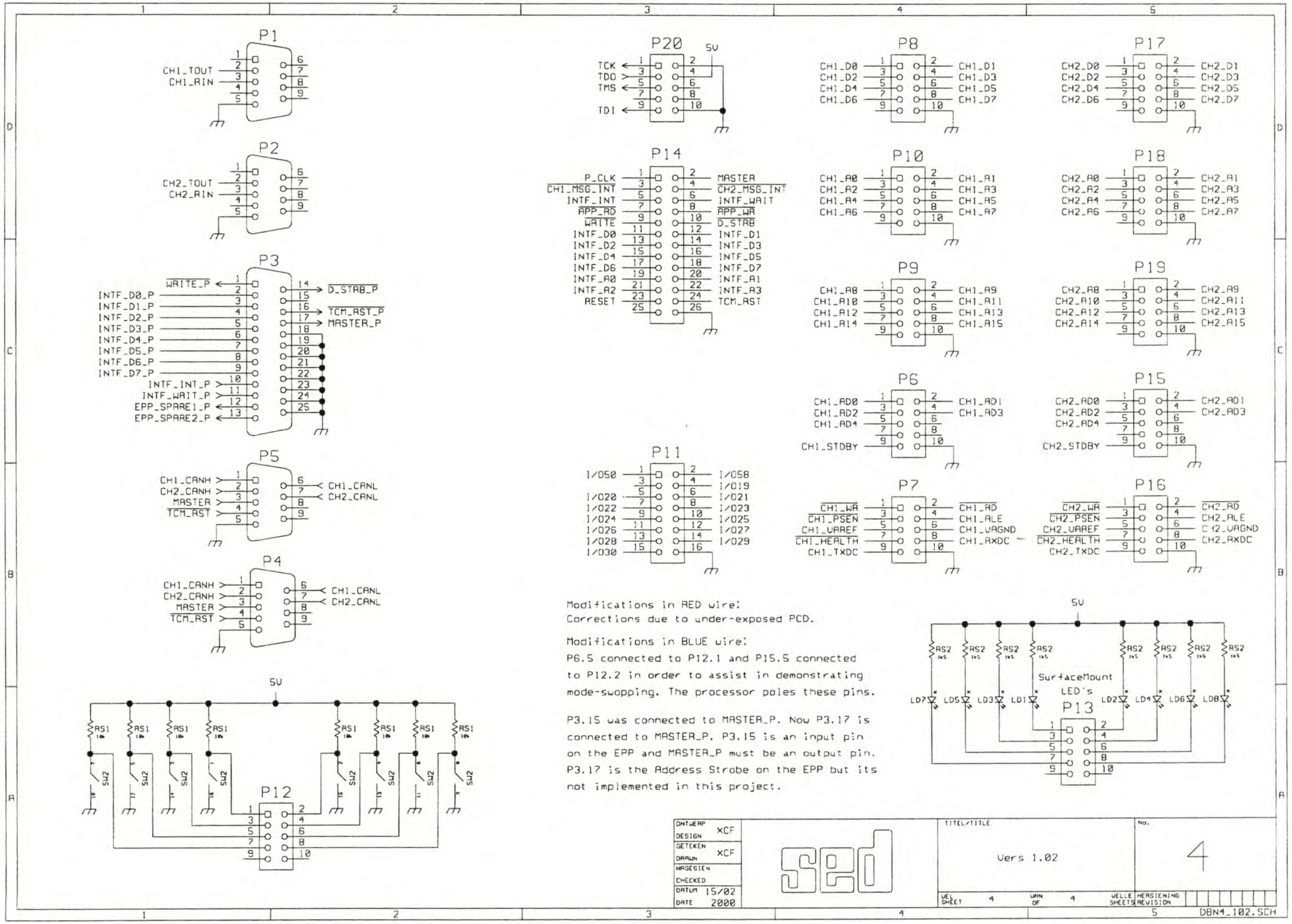
Insert JMP3 and JMP4 only if node is at the end of the CAN Bus.

DC-power plug socket - PCB mounted. 2.5mm

Big stud for ground reference.

Figure D.3: Dual CAN node schematics - Page 3

Stellenbosch University http://scholar.sun.ac.za



Modifications in RED wire:
Corrections due to under-exposed PCD.

Modifications in BLUE wire:
P6.5 connected to P12.1 and P15.5 connected to P12.2 in order to assist in demonstrating mode-swopping. The processor poles these pins.

P3.15 was connected to MASTER_P. Now P3.17 is connected to MASTER_P. P3.15 is an input pin on the EPP and MASTER_P must be an output pin. P3.17 is the Address Strobe on the EPP but its not implemented in this project.

Figure D.4: Dual CAN node schematics - Page 4

Stellenbosch University http://scholar.sun.ac.za

D.2 PCB layouts

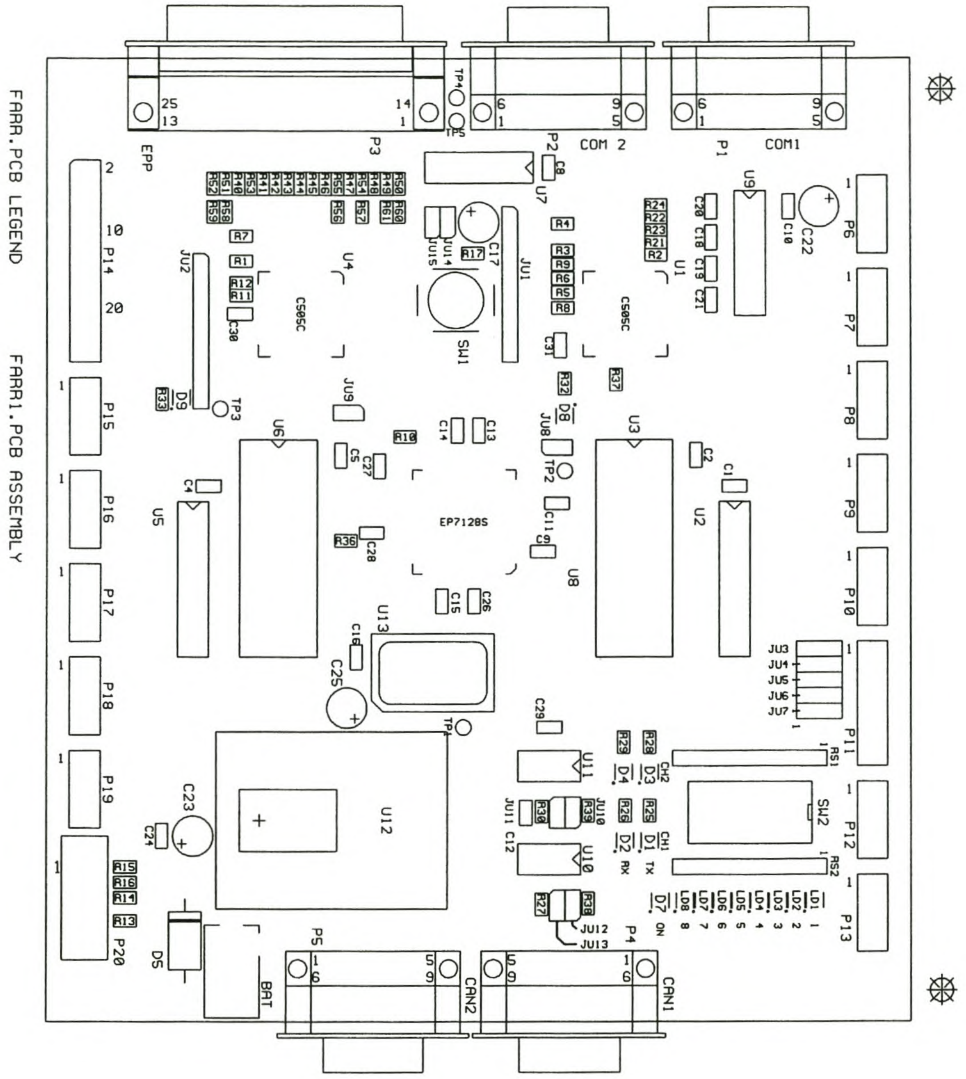


Figure D.5: Dual CAN node Legend

Stellenbosch University http://scholar.sun.ac.za

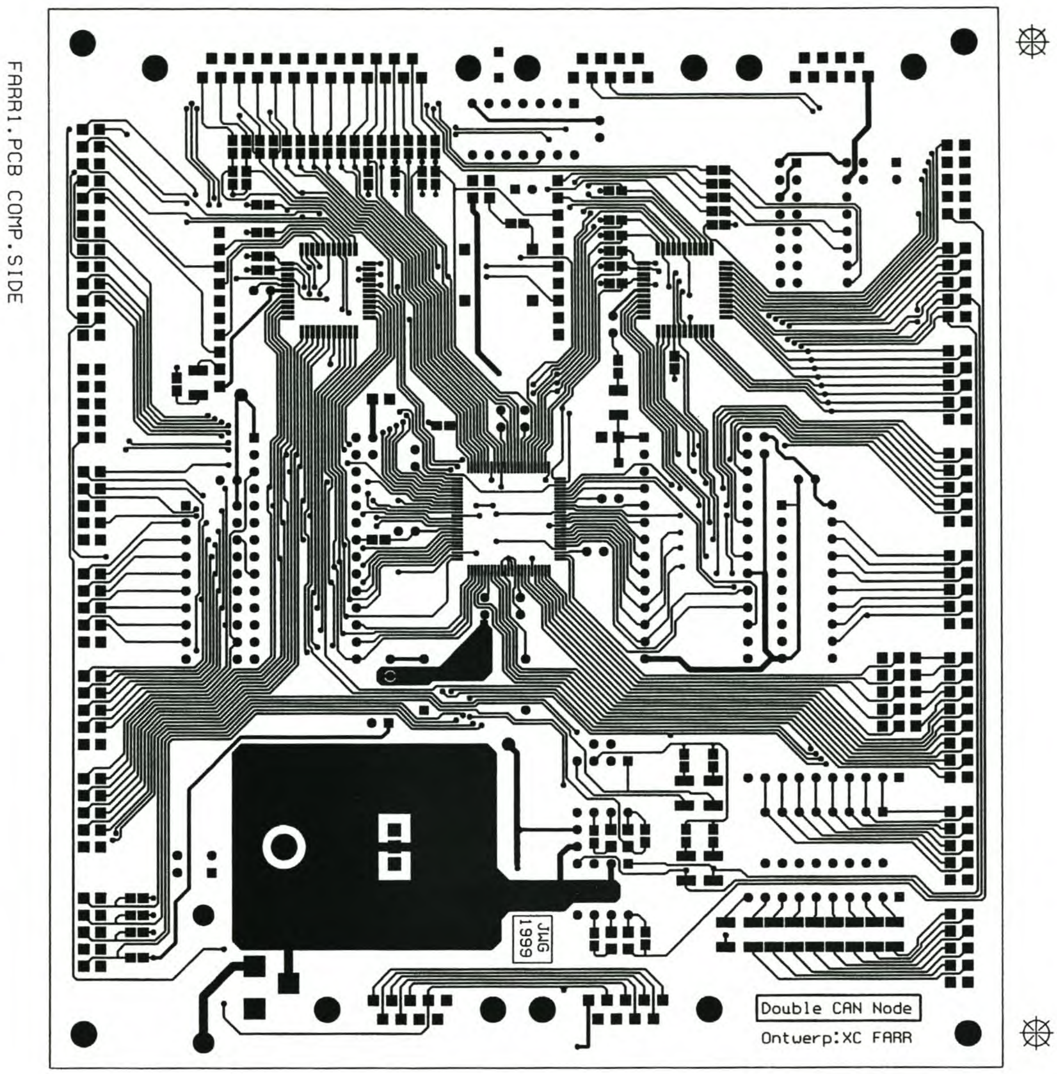


Figure D.6: Dual CAN node component side.

Stellenbosch University http://scholar.sun.ac.za

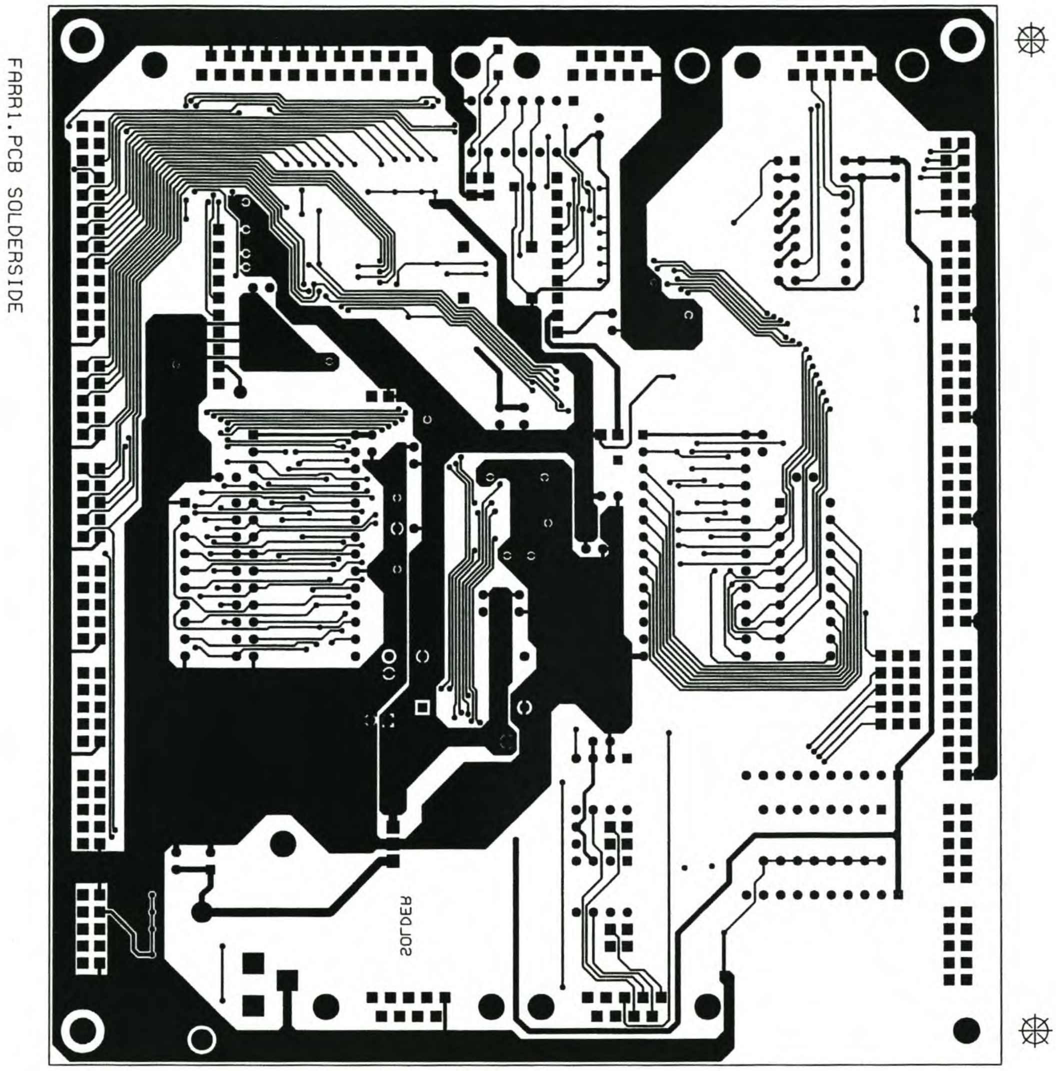


Figure D.7: Dual CAN node solder side.

Stellenbosch University http://scholar.sun.ac.za

D.3 Components list

```
%*********************************************************************
%                                                                    *
%     Program  :  PC-FORM VERSION 8.0                               *
%     Date     :  Feb 16 2000                                       *
%     Time     :  11:37:15 AM                                       *
%     File In  :  dbn.xnl                                           *
%     File Out :  dbn.mat                                           *
%     Format   :  P-CAD MATERIALS LIST                              *
%                                                                    *
%*********************************************************************

ITEM  SQTY PQTY  COMP-NAME  REFERENCE-DESIGNATOR        DESCRIPTION
----  ---- ----  ---------  -------------               -----------


1      2    2    hc573      U2 U5


2      2    2    c505c      U1 U4


3      2    2    27c256     U3 U6


4      20   20   capv       C5 C2 C4 C30 C1 C31 C9
                            C11 C12 C13 C14 C15
                            C16 C10 C8 C24 C29 C28
                            C27 C26


5      4    1    hc14       U7 U7 U7 U7


6      2    2    pin10s1    JU1 JU2
```

Stellenbosch University http://scholar.sun.ac.za

7	31	31	resv	R36 R8 R11 R10 R1 R9 R7 R12 R5 R4 R3 R37 R6 R2 R14 R15 R16 R13 R56 R57 R58 R59 R60 R61 R17 R33 R32 R39 R38 R27 R30
8	1	1	ep7128sa	U8
9	24	24	resh	R55 R54 R53 R52 R51 R50 R49 R48 R47 R46 R45 R44 R43 R42 R41 R40 R23 R29 R25 R26 R24 R22 R21 R28
10	5	5	pin3	JU4 JU3 JU5 JU7 JU6
11	1	1	txco	U13
12	3	3	polcapv	C17 C25 C23
13	1	1	spstpb	SW1
14	4	4	caph	C20 C18 C19 C21
15	1	1	polcaph	C22
16	15	15	ledv	D7 D9 D8 D1 D2 D4 D3

Stellenbosch University http://scholar.sun.ac.za

				LD8 LD5 LD3 LD2 LD6 LD1 LD4 LD7
17	8	8	pin2	JU15 JU14 JU9 JU8 JU10 JU13 JU11 JU12
18	5	5	tp	TP4 TP2 TP1 TP3 TP5
19	1	1	7805cth	U12
20	1	1	max232	U9
21	1	1	diodeh	D5
22	2	2	82c250	U10 U11
23	17	2	rsip10a	RS2 RS1 RS1 RS1 RS1 RS1 RS1 RS1 RS2 RS2 RS2 RS2 RS2 RS2 RS2 RS2 RS1
24	1	1	pin2s	BAT
25	3	3	db9sr	P1 P2 P4
26	13	13	pin10	P16 P17 P18 P15 P19 P7 P8 P10 P6 P9 P12 P13 P20

Stellenbosch University http://scholar.sun.ac.za

27	8	1	sw-8a	SW2 SW2 SW2 SW2 SW2 SW2 SW2 SW2
28	1	1	pin16	P11
29	1	1	db25pr	P3
30	1	1	db9pr	P5
31	1	1	pin26	P14

–ooOoo–

Stellenbosch University http://scholar.sun.ac.za

Appendix E

Firmware and software

The following files[1] are included on the CD attached to this document:

- Firmware for the dual CAN node;
- Turbo Pascal files for the EPP interface;
- Delhpi application files and driver files for the PCCAN interface;

E.1 Processor Firmware

The following files are included on the CD:

regc505c.h	SFR declarations for C515C;
intc505c.h	Interrupt definition file;
canreg.h	CAN definition file;
driver.h	Header file for the CAN driver routines;
driver.c	C-file for the CAN driver routines;
def_msgs.h	Header file for the definitions of all CAN message objects;
def_msgs.c	C-file for the definitions of all CAN message objects;
send_msg.h	Header file for the ACP's "send_msg"-routines;
send_msg.c	C-file for the ACP's "send_msg"-routines;
dbnp.c	The main file for the dual CAN node.

The preamble to the main application software is given here as introduction to the rest of the software on the CD.

[1] Other documents that can be helpful in future, are included on this CD as well.

Stellenbosch University http://scholar.sun.ac.za

File : dbnp.c - Preamble

```
/***************************************************************************/
/* Program name:          DBNP.C                                           */
/* Compiler used:         Keil C51 Compiler                                */
/* Last modifications:    11 November 1999                                 */
/* Author:                X.C. Farr                                        */
/* Target device:         SAB-C505C-SAF                                    */
/***************************************************************************/
/* Function: This firmware represents the application and driver software*/
/*           for the dual CAN node designed during a thesis titled:       */
/*           Development of a fault-tolerant bus system suitable for a    */
/*           high-performance, embedded, real-time application on         */
/*           SUNSAT's ADCS.                                               */
/***************************************************************************/
/* Small Memory model => All variables and the stack is in internal RAM. */
/* Change the following for the C515C:   "regc505c.h"                    */
/*                                       "intc505c.h"                    */
/* Note1: Not really part of the driver. This is a modification to assist*/
/*        in demonstrating mode-swopping.                                 */
/***************************************************************************/
#pragma SMALL
#pragma DEBUG OBJECTEXTEND CODE              // Command line directives.
#include "regc505c.h"                        // SFR declarations for C515C
#include "intc505c.h"                        // Interrupt definition file.
#include "canreg.h"                          // CAN definition file
#include "driver.h"                          // Def. of the CAN driver
                                             // routines.
#include "def_msgs.h"                        // Def. of all CAN message
                                             // objects.
```

Stellenbosch University http://scholar.sun.ac.za

```
#include "send_msg.h"                           // The ACP's "send_msg"-
                                                // routines.
#define FOREVER for (;;)                        // Endless loop
/**************************************************************************/
/*                   GLOBAL DECLARATION OF VARIABLES                      */
/**************************************************************************/
/...CD
```

Stellenbosch University http://scholar.sun.ac.za

E.2 VHDL Code

The VHDL code for the CPLD is given here, while the rest of the associated software to compile successfully is available on the CD.

```
---------------------------------------------------------------------------
-- Filename                  : D_Node_L.vhd
-- Author                    : X. Farr
-- Date                      : 14-12-1999
-- Target Device             : CPLD for Double-CAN-node processor interface
--                           : EPM7128STC100-15
---------------------------------------------------------------------------
-- Functions : Manages the interprocessor communication between the two
--             C505C microcontrollers when the dual CAN node is operating
--             in a Master-Checker configuration.
--           : Manages the communication between the Master of the dual
--             CAN node and the application (peripheral) connected to it.
--           : Combines the reset signal comming from the power-on reset
--             and the Telecommand reset to form a synchronised reset
--             signal for the two C505C microcontrollers.
--           : The oscillator signal is taken via the CPLD to the uP in
--             the event that the clock signal must be divided for the
--             purpose of the tests to be performed on the dual CAN node.
---------------------------------------------------------------------------
-- Notes : Those processes and signals commented out are used when the
--         peripheral connected to the dual CAN node is either a sensor
--         or an actuator. The two processes called "Processor_Clock" are
--         used for dividing the clock either by two or by a specific
--         value depending on the tests performed.
--       : Processes are divided into three groups:
--         1. Channel1 processes - interrupts, address decoding, registers.
```

Stellenbosch University http://scholar.sun.ac.za

```
--         2. Channel2 processes - interrupts, address decoding, registers.
--         3. Application processes - interrupts, registers, reset, clock.
---------------------------------------------------------------------------
library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_unsigned.all;

entity D_Node_L is
   Port(CH1_D        : InOut  std_logic_vector (7 downto 0);
                                                    --Data Bus
        CH1_A        : In     std_logic_vector (6 downto 0);
                                                    --Address Bus
        nCH1_MSG_INT : Out    std_logic;            --Interrupt signal to
                                                    --CH2
        CH1_A15      : In     std_logic;            --MS Address Bus Bit
        nCH1_RD      : In     std_logic;            --Read signal
        nCH1_WR      : In     std_logic;            --Write signal
        nCH1_Health  : In     std_logic;            --Health indication

        CH2_D        : InOut  std_logic_vector (7 downto 0);
                                                    --Data Bus
        CH2_A        : In     std_logic_vector (6 downto 0);
                                                    --Address Bus
        nCH2_MSG_INT : Out    std_logic;            --Interrupt signal to
                                                    --CH1
        CH2_A15      : In     std_logic;            --MS Address Bus Bit
        nCH2_RD      : In     std_logic;            --Read signal
        nCH2_WR      : In     std_logic;            --Write signal
        nCH2_Health  : In     std_logic;            --Health indication
```

Stellenbosch University http://scholar.sun.ac.za

```
--      nAPP_RD      : Out    std_logic;
--      nAPP_WR      : Out    std_logic;
        INTF_D       : InOut  std_logic_vector (7 downto 0);
                                                     --Interface Data Bus
        INTF_A       : Out    std_logic_vector (3 downto 0);
                                                     --Interface Address Bus
        INTF_INT     : Out    std_logic;             --EPP Intterupt signal
        INTF_WAIT    : Out    std_logic;             --EPP wait signal
        nWRITE       : In     std_logic;             --Read/Not-Write EPP
                                                     --signal
        nD_STRB      : In     std_logic;             --EPP data strobe
        RESET        : Buffer std_logic;             --Reset to the uP's.
        nTCM_RST     : In     std_logic;             --Active low RST signal
        nSTRT_RST    : In     std_logic;             --Reset on Power-on
        SYS_CLK      : In     std_logic;             --System clock
        P_CLK        : Buffer std_logic);            --Processor clock
end D_Node_L;

architecture BEHAVIORAL of D_Node_L is
        signal nCH1_INTF_ALE : std_logic;            --Interface ALE
        signal nCH1_MSG_RDY  : std_logic;            --Message Ready
        signal nCH1_APP_RD   : std_logic;            --Application Read
        signal nCH1_MSG_RD   : std_logic;            --Message Read
        signal nCH1_APP_WR   : std_logic;            --Application Write
        signal CH1_MSG_REG   : std_logic_vector (7 downto 0);
        signal CH1_ABORT     : std_logic_vector (7 downto 0);

        signal nCH2_INTF_ALE : std_logic;            --Interface ALE
```

Stellenbosch University http://scholar.sun.ac.za

```
        signal nCH2_MSG_RDY  : std_logic;           --Message Ready
        signal nCH2_APP_RD   : std_logic;           --Application Read
        signal nCH2_MSG_RD   : std_logic;           --Message Read
        signal nCH2_APP_WR   : std_logic;           --Application Write
        signal CH2_MSG_REG   : std_logic_vector (7 downto 0);
        signal CH2_ABORT     : std_logic_vector (7 downto 0);

        signal INTF_IN_REG   : std_logic_vector (7 downto 0);
        signal INTF_OUT_REG  : std_logic_vector (7 downto 0);
        signal Time_Out      : std_logic_vector (7 downto 0);
--      signal divider       : integer range 0 to 10;

begin
---------------  -----------------------------------------------------------
-- CHANNEL 1 --  -----------------------------------------------------------
---------------  -----------------------------------------------------------
Signal_nCH1_INTF_ALE: process (RESET, nCH1_HEALTH, SYS_CLK, nCH1_WR,
                               CH1_A15, CH1_A)
    variable nCH1_IOW_ENA : std_logic_vector (4 downto 0);
begin
    nCH1_IOW_ENA := nCH1_WR & CH1_A15 & CH1_A(6 downto 4);
                                                    --Gate signal
    if (RESET = '1') or (nCH1_HEALTH = '1') then
        nCH1_INTF_ALE <= '1';                       --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH1_IOW_ENA = "01000" then          --Address=8000H
                nCH1_INTF_ALE <= '0';               --Enable signal
            else
```

Stellenbosch University http://scholar.sun.ac.za

```
                nCH1_INTF_ALE <= '1';               --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH1_INTF_ALE;


Signal_nCH1_MSG_RDY: process (RESET, nCH1_HEALTH, SYS_CLK, nCH1_WR,
                              CH1_A15, CH1_A)
    variable nCH1_IOW_ENA : std_logic_vector (4 downto 0);
begin
    nCH1_IOW_ENA := nCH1_WR & CH1_A15 & CH1_A(6 downto 4);
                                                    --Gate signal
    if (RESET = '1') or (nCH1_HEALTH = '1') then
        nCH1_MSG_RDY <= '1';                        --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH1_IOW_ENA = "01001" then          --Address = 8010H
                nCH1_MSG_RDY <= '0';                --Enable signal
            else
                nCH1_MSG_RDY <= '1';                --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH1_MSG_RDY;


Signal_nCH1_APP_RD: process (RESET, nCH1_HEALTH, SYS_CLK, nCH1_RD, CH1_A15,
                             CH1_A)
    variable nCH1_IOR_ENA : std_logic_vector (4 downto 0);
begin
```

Stellenbosch University http://scholar.sun.ac.za

```
    nCH1_IOR_ENA := nCH1_RD & CH1_A15 & CH1_A(6 downto 4);
                                                --Gate signal
    if (RESET = '1') or (nCH1_HEALTH = '1') then
        nCH1_APP_RD <= '1';                     --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH1_IOR_ENA = "01010" then      --Address = 8020H
                nCH1_APP_RD <= '0';             --Enable signal
            else
                nCH1_APP_RD <= '1';             --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH1_APP_RD;

Signal_nCH1_MSG_RD: process (RESET, nCH1_HEALTH, SYS_CLK, nCH1_RD, CH1_A15,
                             CH1_A)
    variable nCH1_IOR_ENA : std_logic_vector (4 downto 0);
begin
    nCH1_IOR_ENA := nCH1_RD & CH1_A15 & CH1_A(6 downto 4);
                                                --Gate signal
    if (RESET = '1') or (nCH1_HEALTH = '1') then
        nCH1_MSG_RD <= '1';                     --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH1_IOR_ENA = "01011" then      --Address = 8030H
                nCH1_MSG_RD <= '0';             --Enable signal
            else
                nCH1_MSG_RD <= '1';             --Disable signal
```

Stellenbosch University http://scholar.sun.ac.za

```
            end if;
        end if;
    end if;
end process Signal_nCH1_MSG_RD;

Signal_nCH1_APP_WR: process (RESET, nCH1_HEALTH, SYS_CLK, nCH1_WR, CH1_A15,
                             CH1_A)
    variable nCH1_IOW_ENA : std_logic_vector (4 downto 0);
begin
    nCH1_IOW_ENA := nCH1_WR & CH1_A15 & CH1_A(6 downto 4);
                                                        --Gate signal
    if (RESET = '1') or (nCH1_HEALTH = '1') then
        nCH1_APP_WR <= '1';                             --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH1_IOW_ENA = "01100" then              --Address = 8040H
                nCH1_APP_WR <= '0';                     --Enable signal
            else
                nCH1_APP_WR <= '1';                     --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH1_APP_WR;

Signal_nCH1_MSG_INT: process (SYS_CLK, RESET, nCH1_MSG_RDY, nCH2_MSG_RD)
                                                        --Also ACK for CH1
begin
    if RESET = '1' then
        nCH1_MSG_INT <= '1';                            --Reset signal
```

Stellenbosch University http://scholar.sun.ac.za

```
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if (nCH1_MSG_RDY = '0') then             --Enable signal if uP
                nCH1_MSG_INT <= '0';                 --wrote to the register
                CH1_ABORT <= "00000000";             --Reset the counter
            elsif (nCH2_MSG_RD = '0') or (CH1_ABORT = "11111111") then
                nCH1_MSG_INT <= '1';                 --Disable interrupt
            else
                CH1_ABORT <= CH1_ABORT + 1;          --If SYS_CLK = 20MHz,
                                                     --then after 256/20MHz
                                                     -- = 12.8us does the
                                                     --interrupt signal
                                                     --reset.
                                                     --If SYS_CLK = 16MHz,
                                                     --then after 256/8MHz
                                                     -- = 16us does the
                                                     --interrupt signal
                                                     --reset.
            end if;
        end if;
    end if;
end process Signal_nCH1_MSG_INT;

CH1_MSG_RegisterControl: process (SYS_CLK, nCH1_MSG_RDY, CH1_D)
begin
    if SYS_CLK'event and SYS_CLK = '1' then
        if nCH1_MSG_RDY = '0' then                   --On writing to the
            CH1_MSG_REG <= CH1_D;                    --register latch the
                                                     --data.
```

Stellenbosch University http://scholar.sun.ac.za

```
        end if;
    end if;
end process CH1_MSG_RegisterControl;

CH1DataBusControl: process (nCH1_APP_RD, nCH1_MSG_RD, INTF_IN_REG,
                            CH2_MSG_REG, nCH1_HEALTH)
begin
    if ((nCH1_APP_RD='1') and (nCH1_MSG_RD='1')) or (nCH1_HEALTH='1') then
        CH1_D <= "ZZZZZZZZ";                        --Place the port into
    elsif nCH1_APP_RD = '0' then                    --high impendance when
        CH1_D <= INTF_IN_REG;                       --not writing to it,
    elsif nCH1_MSG_RD = '0' then                    --otherwise multiplex
        CH1_D <= CH2_MSG_REG;                       --the application and
    end if;                                         --CH2 registers.
end process CH1DataBusControl;

---------------  -----------------------------------------------------------
-- CHANNEL 2 --  -----------------------------------------------------------
---------------  -----------------------------------------------------------
Signal_nCH2_INTF_ALE: process (RESET, nCH2_HEALTH, SYS_CLK, nCH2_WR,
                               CH2_A15, CH2_A)
    variable nCH2_IOW_ENA : std_logic_vector (4 downto 0);
begin
    nCH2_IOW_ENA := nCH2_WR & CH2_A15 & CH2_A(6 downto 4);
                                                    --Gate signal
    if (RESET = '1') or (nCH2_HEALTH = '1') then
        nCH2_INTF_ALE <= '1';                       --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
```

Stellenbosch University http://scholar.sun.ac.za

```
            if nCH2_IOW_ENA = "01000" then           --Address = 8000H
                nCH2_INTF_ALE <= '0';                --Enable signal
            else
                nCH2_INTF_ALE <= '1';                --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH2_INTF_ALE;

Signal_nCH2_MSG_RDY: process (RESET, nCH2_HEALTH, SYS_CLK, nCH2_WR,
                              CH2_A15, CH2_A)
    variable nCH2_IOW_ENA : std_logic_vector (4 downto 0);
begin
    nCH2_IOW_ENA := nCH2_WR & CH2_A15 & CH2_A(6 downto 4);
                                                     --Gate signal
    if (RESET = '1') or (nCH2_HEALTH = '1') then
        nCH2_MSG_RDY <= '1';                         --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH2_IOW_ENA = "01001" then           --Address = 8010H
                nCH2_MSG_RDY <= '0';                 --Enable signal
            else
                nCH2_MSG_RDY <= '1';                 --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH2_MSG_RDY;

Signal_nCH2_APP_RD: process (RESET, nCH2_HEALTH, SYS_CLK, nCH2_RD, CH2_A15,
```

Stellenbosch University http://scholar.sun.ac.za

```
                             CH2_A)
    variable nCH2_IOR_ENA : std_logic_vector (4 downto 0);
begin
    nCH2_IOR_ENA := nCH2_RD & CH2_A15 & CH2_A(6 downto 4);
                                                    --Gate signal
    if (RESET = '1') or (nCH2_HEALTH = '1') then
        nCH2_APP_RD <= '1';                         --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH2_IOR_ENA = "01010" then          --Address = 8020H
                nCH2_APP_RD <= '0';                 --Enable signal
            else
                nCH2_APP_RD <= '1';                 --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH2_APP_RD;

Signal_nCH2_MSG_RD: process (RESET, nCH2_HEALTH, SYS_CLK, nCH2_RD,
                             CH2_A15, CH2_A)
    variable nCH2_IOR_ENA : std_logic_vector (4 downto 0);
begin
    nCH2_IOR_ENA := nCH2_RD & CH2_A15 & CH2_A(6 downto 4);
                                                    --Gate signal
    if (RESET = '1') or (nCH2_HEALTH = '1') then
        nCH2_MSG_RD <= '1';                         --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH2_IOR_ENA = "01011" then          --Address = 8030H
```

Stellenbosch University http://scholar.sun.ac.za

```
                nCH2_MSG_RD <= '0';                  --Enable signal
            else
                nCH2_MSG_RD <= '1';                  --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH2_MSG_RD;

Signal_nCH2_APP_WR: process (RESET, nCH2_HEALTH, SYS_CLK, nCH2_WR,
                             CH2_A15, CH2_A)
    variable nCH2_IOW_ENA : std_logic_vector (4 downto 0);
begin
    nCH2_IOW_ENA := nCH2_WR & CH2_A15 & CH2_A(6 downto 4);
                                                     --Gate signal
    if (RESET = '1') or (nCH2_HEALTH = '1') then
        nCH2_APP_WR <= '1';                          --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nCH2_IOW_ENA = "01100" then           --Address = 8040H
                nCH2_APP_WR <= '0';                  --Enable signal
            else
                nCH2_APP_WR <= '1';                  --Disable signal
            end if;
        end if;
    end if;
end process Signal_nCH2_APP_WR;

Signal_nCH2_MSG_INT: process (SYS_CLK, RESET, nCH2_MSG_RDY, nCH1_MSG_RD)
                                                     --Also ACK for CH2
```

Stellenbosch University http://scholar.sun.ac.za

```
begin
    if RESET = '1' then
        nCH2_MSG_INT <= '1';                              --Reset signal
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if (nCH2_MSG_RDY = '0') then                  --Enable signal if uP
                nCH2_MSG_INT <= '0';                      --wrote to the register
                CH2_ABORT <= "00000000";                  --Reset the counter
            elsif (nCH1_MSG_RD = '0') or (CH2_ABORT = "11111111") then
                nCH2_MSG_INT <= '1';                      --Disable interrupt
            else
                CH2_ABORT <= CH2_ABORT + 1;               -- If SYS_CLK = 20MHz,
                                                          -- then after 256/20MHz
                                                          -- = 12.8us does the
                                                          -- interrupt signal
                                                          -- reset.
            end if;
        end if;
    end if;
end process Signal_nCH2_MSG_INT;

CH2_MSG_RegisterControl: process (SYS_CLK, nCH2_MSG_RDY, CH2_D)
begin
    if SYS_CLK'event and SYS_CLK = '1' then
        if nCH2_MSG_RDY = '0' then                        --On writing to the
            CH2_MSG_REG <= CH2_D;                         --register latch the
                                                          --data.
        end if;
    end if;
```

Stellenbosch University http://scholar.sun.ac.za

```
end process CH2_MSG_RegisterControl;


CH2DataBusControl: process (nCH2_APP_RD, nCH2_MSG_RD, INTF_IN_REG,
                            CH1_MSG_REG, nCH2_HEALTH)
begin
    if ((nCH2_APP_RD='1') and (nCH2_MSG_RD='1')) or (nCH2_HEALTH='1') then
        CH2_D <= "ZZZZZZZZ";                          --Place the port into
    elsif nCH2_APP_RD = '0' then                      --high impendance when
        CH2_D <= INTF_IN_REG;                         --not writing to it,
    elsif nCH2_MSG_RD = '0' then                      --otherwise multiplex
        CH2_D <= CH1_MSG_REG;                         --the application and
    end if;                                           --CH1 registers.
end process CH2DataBusControl;


-----------------  ---------------------------------------------------------
-- APPLICATION --  ---------------------------------------------------------
-----------------  ---------------------------------------------------------
--nAPP_RD <= nCH1_APP_RD and nCH2_APP_RD;             --Sensor-Actuator Read
                                                      --signal
--nAPP_WR <= nCH1_APP_WR and nCH2_APP_WR;             --Sensor-Actuator Write
                                                      --signal
P_CLK <= SYS_CLK;                                     --Oscillator frequency
                                                      --is not divided.


--Processor_Clock: process (SYS_CLK)                    --Divide the oscillator
--begin                                                 --frequency by the value
--    if SYS_CLK'event and SYS_CLK = '1' then           --of "divider".
--        if divider = 4 then
--            divider <= 0;
```

Stellenbosch University http://scholar.sun.ac.za

```
--            P_CLK <= not P_CLK;
--        else
--            divider <= divider + 1;
--        end if;
--    end if;
--end process Processor_Clock;

--Processor_Clock: process (SYS_CLK)                     --Divide the oscillator
--begin                                                  --frequency by two.
--    if SYS_CLK'event and SYS_CLK = '1' then
--        P_CLK <= not P_CLK;
--    end if;
--end process Processor_Clock;

Wait_SignalGeneration: process (SYS_CLK, RESET, nWRITE, nD_STRB,
                                nCH1_APP_RD, nCH2_APP_RD)
    variable Temp : std_logic_vector (3 downto 0);
begin
    Temp := nWRITE & nD_STRB & nCH1_APP_RD & nCH2_APP_RD;
                                                        --Gating signal.
    if RESET = '1' then
        INTF_WAIT <= '0';                               --Reset the signal.
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            case Temp is
                when "1011" =>                          --Acknowledge a
                    INTF_WAIT <= '1';                   --EPP-RD-cycle.
                when "0001" =>                          --Acknowledge a
                    INTF_WAIT <= '1';                   --EPP-WR-cycle.
```

Stellenbosch University http://scholar.sun.ac.za

```
                when "0010" =>                      --Acknowledge a
                    INTF_WAIT <= '1';               --EPP-WR-cycle.
                when "1111" =>                      --Terminates a
                    INTF_WAIT <= '0';               --EPP-WR/RD-cycle.
                when others =>
                    NULL;
            end case;
        end if;
    end if;
end process Wait_SignalGeneration;

EPP_INT_Generation: process (SYS_CLK, RESET, nWRITE, nD_STRB, nCH1_APP_WR,
                             nCH2_APP_WR)
begin
    if RESET = '1' then
        INTF_INT <= '0';                            --Reset the interrupt.
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if ((nCH1_APP_WR = '0') or (nCH2_APP_WR = '0')) then
                INTF_INT <= '1';                    --Enable the interrupt.
                Time_Out <= "00000000";             --Reset the counter.
            elsif (((nD_STRB = '0') and (nWRITE = '1')) or
                  (Time_Out = "11111111")) then
                INTF_INT <= '0';                    --Disable the interrupt
            else
                Time_Out <= Time_Out + 1;           --If SYS_CLK = 20MHz,
                                                    --then after 256/20MHz
                                                    -- = 12.8us does the
                                                    --interrupt signal
```

Stellenbosch University http://scholar.sun.ac.za

```
                                                    --reset.
                                                    --If SYS_CLK = 16MHz,
                                                    --then after 256/16MHz
                                                    -- = 16us does the
                                                    --interrupt signal
                                                    --reset.
            end if;
        end if;
    end if;
end process EPP_INT_Generation;

Reset_sinch: process (SYS_CLK, nSTRT_RST, nTCM_RST)
begin
    if SYS_CLK'event and SYS_CLK = '1' then         --Combine and sync. the
        RESET <= not (nSTRT_RST and nTCM_RST);      --power-on and
    end if;                                         --Telecommand reset
                                                    --signals.
end process Reset_sinch;

INTF_IN_RegisterControl: process (RESET, SYS_CLK, INTF_D, nWRITE, nD_STRB)
    variable nEN3a : std_logic;
begin
    nEN3a := nWRITE or nD_STRB;                     --Gating signal
    if RESET = '1' then
        INTF_IN_REG <= "00000000";                  --Reset the register.
    else
        if SYS_CLK'event and SYS_CLK = '1' then
            if nEN3a = '0' then
                INTF_IN_REG <= INTF_D;              --Latch the appl.'s
```

Stellenbosch University http://scholar.sun.ac.za

```
            end if;                                 --data into the reg.
        end if;
    end if;
end process INTF_IN_RegisterControl;

INTF_OUT_RegisterControl: process (SYS_CLK, nCH1_APP_WR, nCH2_APP_WR,
                                   CH1_D, CH2_D)
begin
    if SYS_CLK'event and SYS_CLK = '1' then
        if nCH1_APP_WR = '0' then                   --CH1 is the master.
            INTF_OUT_REG <= CH1_D;                  --Latch data into the
                                                    --application's reg.
        elsif nCH2_APP_WR = '0' then                --CH2 is the master
            INTF_OUT_REG <= CH2_D;                  --Latch data into the
                                                    --application's reg.
        end if;
    end if;
end process INTF_OUT_RegisterControl;

INTF_DataBusControl: process (INTF_OUT_REG, nWRITE, nD_STRB)
    variable nOE3a : std_logic;
begin
    nOE3a := (not nWRITE) or nD_STRB;               --Gating signal for the
                                                    --appl.'s data bus.
    if nOE3a = '1' then
        INTF_D <= "ZZZZZZZZ";                       --When not writing to
    else                                            --the port, keep it in
        INTF_D <= INTF_OUT_REG;                     --a high impendace
    end if;                                         --state, otherwise port
```

Stellenbosch University http://scholar.sun.ac.za

```
                                                    --gets the value of the
                                                    --register.
end process INTF_DataBusControl;

InterfaceAddressMux: process (SYS_CLK, nCH1_INTF_ALE, nCH2_INTF_ALE, CH1_A,
                              CH2_A)
begin
    if SYS_CLK'event and SYS_CLK = '1' then
        if nCH1_INTF_ALE = '0' then                 --CH1 is Master
            INTF_A <= CH1_A(3 downto 0);            --Write CH1's address
                                                    --to the application.
        elsif nCH2_INTF_ALE = '0' then              --CH2 is Master
            INTF_A <= CH2_A(3 downto 0);            --Write CH2's address
                                                    --to the application.
        end if;
    end if;
end process InterfaceAddressMux;

end BEHAVIORAL;

configuration CFG_D_Node_L_BEHAVIORAL of D_Node_L is
    for BEHAVIORAL
    end for;
end CFG_D_Node_L_BEHAVIORAL;
```

–ooOoo–