

University Of Natal

**A Network based Rapid Prototyping System for
Applications in Research and Engineering
Education**

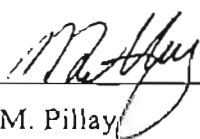
By

Magash Pillay

Submitted in partial fulfilment of the academic requirements for the degree of Master of Science in Engineering, in the Department of Electrical Engineering, University of Natal, Durban, South Africa

February 2001

I hereby declare that all the material incorporated into this thesis is my own original and unaided work except where specific reference is made by name or in the form of a numbered reference. The work contained herein has not been submitted for a degree at any other University.

Signed 
M. Pillay

Date: 03/02/2002

ABSTRACT

Engineering educators the world over are being faced with the dilemma of combining traditional mathematically intensive courses, like Control Systems and Robotics with advances in computational hardware and software. This is because it is impractical to include both software engineering issues as well as conventional course content.

A solution to the problem lies in Rapid Prototyping technology to develop and design software, for application on PC's and embedded systems. Rapid Prototyping, based on automatic code generation, allows users to develop advanced software on high level graphical platforms like Simulink® and LabView®, while “hiding” the underlying layers of complex code. This approach allows the advanced hardware, traditionally reserved for software engineers, to be accessed by a much wider audience and is an ideal educational tool.

This thesis presents the complete development of the Rapid Application Development Environment (RADE). The RADE system customises the Mathworks Real Time Workshop (RTW) revision 11 for application on both standalone and networked DSP cards. The functionality of the RTW is incorporated into the RADE system. This affords the user seamless code generation, downloading, on-line parameter tuning and on-line data visualisation with storage capability. An added advantage of the RADE system is its easy portability to multiple target platforms, which is demonstrated by its implementation on two different DSP cards.

Finally the functionality of the RADE system is demonstrated as an educational tool, with the demonstration of a DC motor speed and position controller.

Dedication

*With deep gratitude and affection to my uncle Authemullam Govender and my anit Visalachee
Kuppasamy for their support and guidance.*

ACKNOWLEDGMENTS

I will like to express my greatest thanks to my supervisor Mr Gregory Diana. Who has endeavoured and succeed in establishing a stimulating and challenging environment at the Motion Control Group in the Department of Electrical Engineering at the University of Natal. His guidance and encouragement throughout this project has been sincerely appreciated.

The following people have also supported me during the course of this work

My family and friends for their understanding and support;

My colleagues Adam Stylo, Myles Walker and Cedric Worthmann for providing a friendly working environment. In addition Adam Stylo requires special mention for laying the groundwork for this project.

The academic and technical staff of the Electrical Department, who are always willing to assist students,.

TABLE OF CONTENTS

CHAPTER ONE: INTRODUCTION	1-1
1.1. General	1-1
1.2. An Overview of Work Presented	1-1
1.3. Thesis Structure	1-3
1.4. Publications and Contributions.....	1-3
CHAPTER TWO: LITERATURE SURVEY.....	2-1
2.1. Introduction	2-1
2.2. Real Time Rapid Prototyping.....	2-1
2.2.1 The Conventional Real-Time Design	2-2
2.2.2 The Rapid Prototyping Approach.....	2-3
2.2.3 The Role of Rapid Prototyping in Education	2-3
2.2.4 Hypersignal	2-5
2.2.5 The Mathworks Rapid Prototyping Framework.....	2-6
2.3. Third Party Tools Based On The Mathworks RTW.....	2-7
2.3.1 dSPACE.....	2-7
2.3.2 Use of dSPACE in Engineering Education	2-8
2.3.3 Complete Experimental Solutions	2-10
2.4. Academic Developments with RTW	2-11
2.4.1 Control System Development Environment.....	2-11
2.4.2 Work at Other Academic Institutions.....	2-13
2.5. Context of Work Presented in this Thesis.....	2-15
2.5.1 Categories of RTW Targets.....	2-16
2.5.2 Summary Of Project Goals.....	2-17
2.6. Conclusion.....	2-18
CHAPTER THREE: OVERVIEW OF SIMULINK AND THE RTW	3-1
3.1. Introduction	3-1

TABLE OF CONTENTS

3.2. Products Available From The Mathworks.....	3-1
3.3. A Overview Of Simulink.....	3-3
3.3.1 An Example with Simulink	3-3
3.3.2 How Simulink Works.....	3-7
3.3.3 An Overview of S-functions.....	3-8
3.4. The Real Time Workshop.....	3-11
3.4.1 An Overview of the RTW	3-12
3.5. Target Language Compiler	3-14
3.5.1 System Target Files	3-16
3.5.2 Block Target Files	3-17
3.6. The Code Build Cycle.....	3-20
3.6.1 Make Utilities	3-20
3.6.2 System Template Make File	3-21
3.6.3 The Build Flow Diagram.....	3-22
3.7. Rapid Prototyping Program Architecture	3-24
3.7.1 System Dependent Layer.....	3-24
3.7.2 System Independent Layer	3-25
3.7.3 Application Layer.....	3-26
3.8. The Mathworks TCP/IP External Mode.....	3-26
3.8.1 Message Frames Between Simulink and Target.....	3-28
3.8.2 Simulink Internals	3-29
3.8.3 Target Internals.....	3-31
3.9. Conclusion	3-32
 CHAPTER FOUR: DESIGN OF THE RADE FRAMEWORK	 4-1
4.1. Introduction	4-1
4.2. Developing the RADE framework	4-1
4.2.1 Mathworks TCP/IP External Mode Architecture	4-1
4.2.2 CSDE External Mode Architecture	4-2
4.2.3 RADE External Mode Architecture	4-3

TABLE OF CONTENTS

4.3. Peripheral Issues.....	4-5
4.3.1 Windows Sockets	4-5
4.3.2 Zuma Toolset for Target Development	4-6
4.4. Modifications to the Simulink Communication Layer.....	4-8
4.4.1 Conversion Functions.....	4-8
4.4.2 Function Registration	4-11
4.5. Server Application.....	4-12
4.5.1 Graphic User Interface	4-14
4.5.2 File Transfer Process	4-15
4.6. Target Run Time Interface.....	4-16
4.6.1 System Dependent Layer.....	4-17
4.7. RADE Communications.....	4-18
4.7.1 Server to Target Protocol.....	4-19
4.7.2 Message Port	4-20
4.7.3 Upload Data Port.....	4-22
4.8. Conclusion.....	4-23
 CHAPTER FIVE: RADE PC32 IMPLEMENTATION	 5-1
5.1. Introduction	5-1
5.2. Description of PC32 Card.....	5-2
5.3. TMS320C32	5-4
5.4. Description of PWM Card.....	5-6
5.5. Device Drivers.....	5-8
5.5.1 ADC's.....	5-9
5.5.2 DAC's.....	5-10
5.5.3 PWM	5-11
5.5.4 Asynchronous Interrupt Support	5-11
5.6. Customising RADE for the PC32.....	5-14
5.6.1 External Mode and the Server to Target Protocol	5-14
5.6.2 System Target File.....	5-17

TABLE OF CONTENTS

5.6.3	Template Make File.....	5-19
5.7.	Conclusion.....	5-20
CHAPTER SIX: RADE ADC64 IMPLEMENTATION.....		6-1
6.1.	Introduction	6-1
6.2.	Description of ADC64 Card	6-1
6.3.	Device Drivers for the RADE ADC64	6-4
6.3.1	ADC's.....	6-5
6.3.2	External Timers	6-6
6.3.3	DAC's, PWM and Interrupt Blocks	6-7
6.4.	Customising RADE framework for the ADC64 Card	6-8
6.4.1	External mode and Server To Target Protocol	6-8
6.5.	Conclusion.....	6-12
CHAPTER SEVEN: DEMONSTRATION OF THE RADE SYSTEM		7-1
7.1.	Introduction	7-1
7.2.	A Case Study: Designing a DC Servo Motor Speed controller	7-2
7.2.1	Motor Model.....	7-2
7.2.2	Design of Current PI Controller	7-3
7.2.3	Simulation of Current Controller.....	7-7
7.2.4	Design and Simulation of Speed PI Controller	7-10
7.3.	Demonstration of the RADE ADC64 System.....	7-12
7.3.1	Real-Time Prototyping with the RADE ADC64.....	7-13
7.3.2	DC Current Controller	7-15
7.3.3	DC Speed Controller	7-21
7.4.	Demonstration of RADE PC32 System	7-26
7.4.1	DC Servo Speed Control	7-26
7.4.2	Position Controller Experiment.....	7-30
7.5.	Conclusion.....	7-34

TABLE OF CONTENTS

CHAPTER EIGHT: CONCLUSION	8-1
8.1. General	8-1
8.1.1 Role of the RADE Framework	8-1
8.2. Suggestions for Further Work	8-2
APPENDIX A: USER GUIDE	1
A.1. Installation Manual for RADE version 1	1
A.1.1 Installation of the RADE Components to Matlab directory	1
A.1.2 Installation of Server Application.....	2
A.2. Sine Wave Example.....	3
A.2.1 RADE PC32	3
A.2.2 RADE ADC64.....	6
APPENDIX B: A PROGRAMMER'S GUIDE TO THE INTERNAL WORKING OF THE RADE SYSTEMS.....	1
B.1. Modifications to The Mathworks External Mode Implementation	1
B.1.1 Default Mathworks External Mode Implementation	1
B.1.2 RADE External Mode.....	3
B.2. Server to Target Protocol.....	5
APPENDIX C: EVALUATION OF MOTOR PARAMETERS	1
APPENDIX D: LISTING OF CODE FOR THE RADE PC32	1
D.1. Conversion Functions ext_convert_c3x.c	1
D.2. System Target File.....	13
D.3. System Template Make File.....	14
D.4. Device Driver Files	18
D.4.1 ADC Blocks.....	18
D.4.2 DAC Block	18
D.4.3 PWM Block	19
D.4.4 Asynchronous Interrupt Support	21

TABLE OF CONTENTS

APPENDIX E: LISTING OF CODE FOR THE RADE ADC64	1
E.1. System Target File	1
E.2. System Template Make File.....	3
E.3. Device Driver Files.....	6
E.3.1 ADC Blocks.....	6
E.3.2 DAC Block.....	7
E.3.3 PWM Block	8
E.3.4 Asynchronous Interrupt Support.....	9
E.3.5 External Timers.....	13
APPENDIX F: DESCRIPTION OF CD	1
REFERENCES.....	1

TABLE OF FIGURES

<i>Fig. 1.1: Mathworks rapid prototyping process</i>	1-2
<i>Fig. 2.1: Conventional real-time design</i>	2-2
<i>Fig. 2.2 : The rapid prototyping design</i>	2-3
<i>Fig. 2.3: Hypersignal environment</i>	2-5
<i>Fig. 2.4: Simplified operation of Mathworks RTW</i>	2-6
<i>Fig. 2.5: dSPACE TDE toolset</i>	2-8
<i>Fig. 2.6: Rapid prototyping laboratory</i>	2-9
<i>Fig. 2.7: Function diagram Quanser system</i>	2-11
<i>Fig. 2.8: Functional operation of CSDE</i>	2-12
<i>Fig. 2.9: Function diagram of ROGER the robot</i>	2-14
<i>Fig. 2.10: Functional diagram for TMS320C30 EVM RTW target</i>	2-15
<i>Fig. 3.1: Product range from The Mathworks</i>	3-2
<i>Fig. 3.2: A Simulink simulation</i>	3-3
<i>Fig. 3.3: A sample of the common Simulink library blocks</i>	3-4
<i>Fig. 3.4: Simulink simulation parameters</i>	3-5
<i>Fig. 3.5: The scope block output of the step response of the system in Fig. 3.2</i>	3-5
<i>Fig. 3.6: An example of a subsystem</i>	3-6
<i>Fig. 3.7: An example of a triggered subsystem</i>	3-6
<i>Fig. 3.8: General model of a Simulink block [</i>	3-7
<i>Fig. 3.9: Flow diagram of Simulink internals</i>	3-8
<i>Fig. 3.10: How Simulink calls into a S-function</i>	3-9
<i>Fig. 3.11: The flow diagram of writing and compiling a S-function</i>	3-9
<i>Fig. 3.12: API function and their calling sequence</i>	3-10
<i>Fig. 3.13: Flow diagram of code generation process</i>	3-12
<i>Fig. 3.14: Object-oriented view</i>	3-14
<i>Fig. 3.15: Operation of the Target Language Compiler</i>	3-15
<i>Fig. 3.16: List of system target files</i>	3-16
<i>Fig. 3.17: Flow diagram of the build process</i>	3-23
<i>Fig. 3.18: Rapid prototyping program framework</i>	3-24
<i>Fig. 3.19: Structure of model code</i>	3-26
<i>Fig. 3.20: TCP/IP implementation of external mode</i>	3-27
<i>Fig. 3.21: Message frame</i>	3-28
<i>Fig. 3.22: Data upload frame</i>	3-28
<i>Fig. 3.23: Simulink internals</i>	3-30
<i>Fig. 3.24: Target internals</i>	3-31
<i>Fig. 3.25: Message Transactions</i>	3-32

TABLE OF FIGURES

Fig. 4.1: Mathworks TCP/IP external mode	4-2
Fig. 4.2: CSDE external mode architecture	4-2
Fig. 4.3: RADE external mode architecture	4-4
Fig. 4.4: Zuma toolset	4-6
Fig. 4.5: Byte format	4-10
Fig. 4.6: Server application	4-13
Fig. 4.7: Server GUI	4-14
Fig. 4.8: Functional representation of the file transfer process	4-15
Fig. 4.9: Flow diagram of file transfer process	4-16
Fig. 4.10: Flow diagram of RTI entry function	4-18
Fig. 4.11: Server to target communications	4-19
Fig. 4.12: Overview of Communication Channel	4-20
Fig. 4.13: Graphical representation of the Message Ports	4-21
Fig. 4.14: Packetisation of external mode messages	4-22
Fig. 4.15: Graphical representation of the Upload Ports	4-23
Fig. 5.1 : Overview of RADE PC32 implementation	5-1
Fig. 5.2: Photo of PC32 card	5-2
Fig. 5.3: Functional diagram of the PC32 card	5-3
Fig. 5.4: ADC triggering	5-4
Fig. 5.5: TMS320C32 block diagram	5-5
Fig. 5.6: Photo of PWM card	5-6
Fig. 5.7: The DSP and PWM plug into the target PC	5-7
Fig. 5.8: Block diagram Of PWM card	5-7
Fig. 5.9: Device driver blocks for PC32	5-9
Fig. 5.10: Parameters for interrupt block	5-12
Fig. 5.11: Flow diagram for the modified Run-Time interface	5-13
Fig. 5.12: Files used for external mode and STP	5-15
Fig. 5.13: RTW build option window	5-18
Fig. 6.1 Photo of the ADC64 DSP card	6-2
Fig. 6.2 Functional diagram of the ADC64 card	6-3
Fig. 6.3: ADC trigger sources for the ADC64 card	6-4
Fig. 6.4: Device Drivers for the ADC64 card	6-4
Fig. 6.5: ADC trigger source selection	6-5
Fig. 6.6: Parameters for external timer Block	6-7
Fig. 6.7: Interrupt block parameters	6-8
Fig. 6.8: A comparison between DPRAM and the PCI bus	6-9
Fig. 6.9: Files used for external mode and STP and the RADE ADC64	6-10

TABLE OF FIGURES

<i>Fig. 7.37: Simulink model</i>	7-27
<i>Fig. 7.38: Subsystem 1 block</i>	7-27
<i>Fig. 7.39: Current response</i>	7-28
<i>Fig. 7.40: ADC64 Current sampling</i>	7-28
<i>Fig. 7.41: Small signal speed response</i>	7-29
<i>Fig. 7.42: Small signal current response</i>	7-29
<i>Fig. 7.43: Large signal speed response</i>	7-30
<i>Fig. 7.44: Large signal current response</i>	7-30
<i>Fig. 7.45: Simulink model</i>	7-31
<i>Fig. 7.46: The controllers subsystem</i>	7-32
<i>Fig. 7.47: Regulating position response</i>	7-33
<i>Fig. 7.48: Under damped response</i>	7-33
<i>Fig. 7.49: Over damped response</i>	7-34

LIST OF ABBREVIATIONS

ADC	Analogue to Digital Converter
API	Application Program Interface
ASIC	Application Specific Integrated Circuit
BSD	Berkeley Software Distribution
DAC	Digital to Analogue Converter
DLL	Dynamic Linked Library
DMA	Direct Memory Access
DPRAM	Dual Port RAM
DSP	Digital Signal Processor
FTP	File Transfer Protocol
GUI	Graphic User Interface
II	Innovative Integration
IPC	Inter Process Communications
ISR	Interrupt Service Routine
MFC	Microsoft Foundation Class
RADE	Rapid Application Development Environment
RAM	Random Access Memory
RPM	Revolution per Minute
RTI	Run-Time Interface
RTW	Real Time Workshop
STP	Server to Target Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TI	Texas Instruments
TLC	Target Language Compiler
VSD	variable speed drive

CHAPTER ONE: INTRODUCTION

1.1. General

In our modern world the use of real-time systems has become commonplace in our homes, cars and workplaces. This is due to the reducing cost of microprocessors coupled with the phenomenal increase in processor bandwidth [AHMED1]. The use of DSP and real-time control is set to continue and engineering educators are therefore required to produce graduates that are equipped to tackle the challenges and technological changes in these fields [TQ2].

Teaching courses that rely on real-time system presents a dilemma to engineering educators, as the implementational details require students to be relatively well versed in software engineering concepts [GAN1]. This is further aggravated by the time constraint of presenting both theoretical course content as well as practical implementations. A point in case is the teaching of Control Systems. This is a challenging theoretical course that is fundamental to many engineering disciplines but is difficult to teach from a practical standpoint, as most modern day implementations rely on complex real-time processing hardware [FENG1]. A plausible solution to this problem is the use of Real-Time Rapid Prototyping Tools.

Rapid prototyping tools allow educators to concentrate on core concepts without being hamstrung by implementational details. Students can use these tools to experimentally validate theoretical assertions without needing to be experienced in software engineering [DSPACE5]. Rapid prototyping allows students from diverse engineering backgrounds to utilise sophisticated digital hardware to control/analyse real-time systems.

1.2. An Overview of Work Presented

The work presented in this thesis deals with the design, development and implementation of a rapid prototyping tool, which is the **Rapid Application Development Environment (RADE)**.

The RADE system consists of three components:

- Simulink ®[MATHWORKS2], which is a widely used graphical simulation package.
- Real Time Workshop (RTW)[MATHWORKS4], an add-on toolbox from The Mathworks, which converts Simulink, models into real-time code

- Target hardware platform that executes the real-time code.

The RADE system, shown in Fig. 1.1 integrates the above components, and offers seamless generation of real-time code from Simulink models. This allows students with little expertise in software engineering, to utilise advanced DSP hardware. The advantage of this approach is that it allows students, to immediately apply concepts taught in courses like control systems, communications and robotics, to real-time systems and see first hand practical verification of theory.

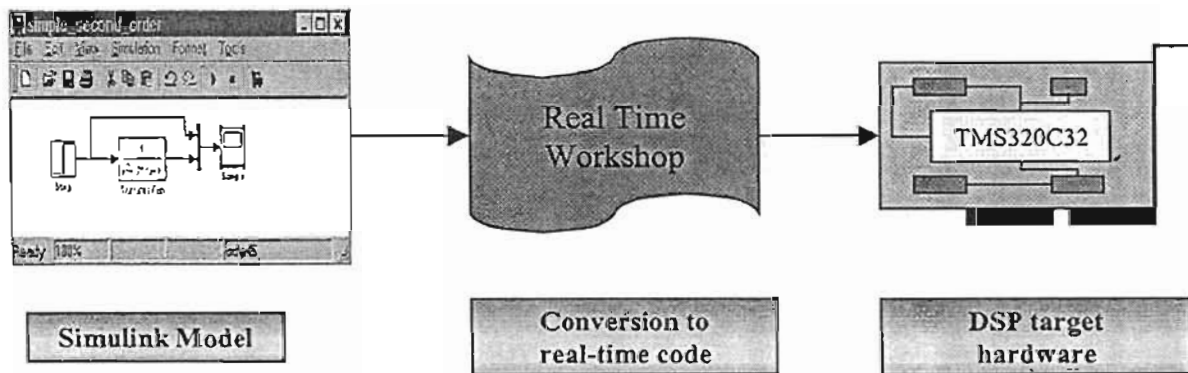


Fig. 1.1: Mathworks rapid prototyping process

The RADE system features the following functionality:

1. Full network and standalone functionality

The system can operate in a standalone mode, whereby Simulink and the target DSP hardware reside on one workstation or in a network mode, whereby multiple users can access a single target DSP hardware. The latter scenario is ideally suited to the educational environment since numerous students can use one DSP card.

2. Online Visualisation

The RADE system is a world first to provide online data visualization using the Simulink scope block, for the TMS320C3x DSP. This allows various signals within a model, to be monitored when it is executing in real-time on the target hardware.

3. Online Parameter Tuning

This feature allows for live changes to parameters within a model, while it is executing on the target hardware. It is especially useful for varying controller parameters and observing system responses. Concepts of controller stability and saturations are easily and quickly demonstrated by varying controller parameters.

4. Multiple Target DSP platforms

The RADE system conforms closely to The Mathworks conventions and is therefore portable to various target platforms. The RADE has been successfully ported to, two DSP cards; the PC32 and ADC64. These cards use the TMS320C32 DSP and are manufactured by Innovative Integration.

5. In house build PWM I/O card

The RADE system was designed to serve a wide spectrum of applications, which also include Motion Control. Therefore a PWM add-on card was designed, to interface directly to the target DSP and provide PWM signal for an inverter. This reduces the processing burden on the target processor and allows for more complex real-time controllers to be implemented.

1.3. Thesis Structure

Chapter 2 presents a literature survey of the current work in the Rapid Prototyping field and concentrates on educational applications. This chapter concludes with the context and purpose of the work presented in this thesis.

Chapter 3 presents background information on Simulink and the workings of the RTW.

Chapter 4 covers the system level design of the RADE framework. It highlights the modifications needed to The Mathworks RTW and details the components of the RADE framework.

Chapter 5 details the implementation of the RADE framework to the PC32 card.

Chapter 6 details the implementation of the RADE framework to the ADC64 card.

Chapter 7 provides a demonstration of the RADE system. An implementation of a DC motor current and speed is presented using the ADC64 and PC32 cards. In addition a DC motor position controller is implemented on the PC32 system.

Chapter 8 concludes this thesis and provides suggestions for further work.

1.4. Publications and Contributions

During the course of the work presented in this thesis the following publications and contributions were made:

M. Pillay Greg Diana "A Design Tool To Facilitate a Matlab/Simulink Simulation, to Run in Real Over a Networked DSP Card", Proceedings of the 8th Southern African Universities Power Electronics Conference, Potchefstroom, South Africa January 1999.

Worlds First Implementation of The Mathworks RTW-3 for the TMS320C32 Target

During the development of the RADE system software bugs were found in the Simulink internals and ,with the help of The Mathworks Support Centre, were corrected. The Mathworks support engineers also stated that this implementation was there first to use a non-PC byte compliant target within The Mathworks TCP/IP External Mode Architecture. Chapter 4 provides more details in this regard.

CHAPTER TWO: LITERATURE SURVEY

2.1. Introduction

The landscape of engineering education is being transformed by the advent of the **Information Age**. The availability of high-speed communication networks, powerful computing technology and advanced software are allowing engineering educators to provide students with a richer learning experience [TQ2]. **Real-Time Rapid Prototyping**¹ is one such technology that is impacting on the teaching of courses like control engineering, DSP and robotics. Engineering educators at numerous institutions across the world are starting to use rapid prototyping tools in their research and teaching syllabuses [KOZICK, STYLO1, GANI].

This chapter describes the use and role of rapid prototyping in the research being conducted by the University of Natal's Motion Control Group and other researchers across the world. It presents a concise overview of The Mathworks rapid prototyping framework and a general review of both commercial and academic implementations derived from this framework. The main aim of this chapter is to review the use of The Mathworks rapid prototyping framework from an educational perspective.

Finally the author contextualises the purpose and role of his work and develops project goals for the development of the RADE system.

2.2. Real Time Rapid Prototyping

In a description of rapid prototyping it is informative to first outline the conventional methodology for designing real-time systems and then compare it to the rapid prototyping approach. This is presented in the next two sections.

¹ Any reference to Rapid Prototyping in this thesis will exclusively deal with Real-Time Rapid Prototyping.

2.2.1 The Conventional Real-Time Design

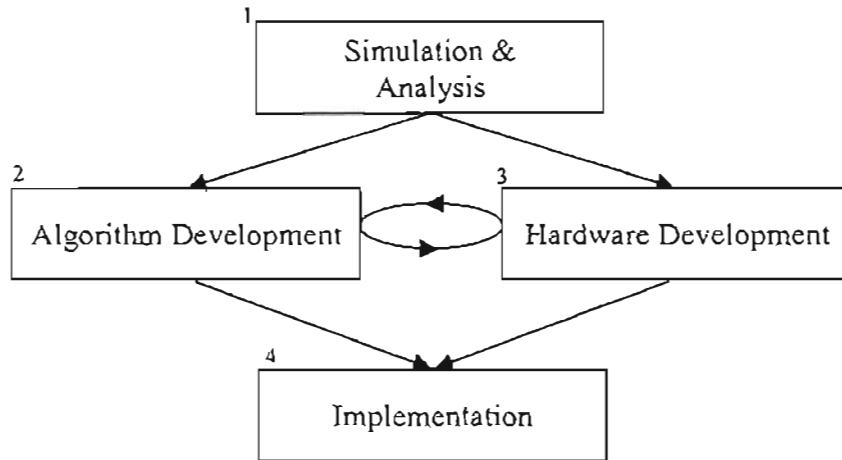


Fig. 2.1: Conventional real-time design

The conventional design process of a real-time system is shown in Fig. 2.1. This normally requires multiple design teams and involves the following four stages:

1. Simulation and Analysis

At this stage the system to be designed is evaluated and different control strategies are simulated. The results are then analysed and a feasible specification is passed on to the next stage.

2. Algorithm development²

From the specifications the software team develops an appropriate algorithms, which traditionally involve C and some low level programming.

3. Hardware development

The hardware team is responsible for the designing of the hardware platform on which the final system will run.

4. Implementation phase

Once the hardware and software teams have produced their components the implementation team is responsible for the integration and testing. If the system fails during the test cycle the design processes is restarted at the stage where the problem is anticipated to be.

In the conventional approach, different tools and expertise are utilised in the development process, with control engineers being used in the first stage and hardware and software engineers being involved in the remaining stages. This approach is very time and labour intensive with each design

² This stage can run in parallel with hardware development.

cycle being costly. A further disadvantage is that code produced by the software teams tend to be difficult to reuse and incorporate in new systems. The advantage of the conventional approach is that the resulting system can be both cost and performance optimised for volume production.

2.2.2 The Rapid Prototyping Approach

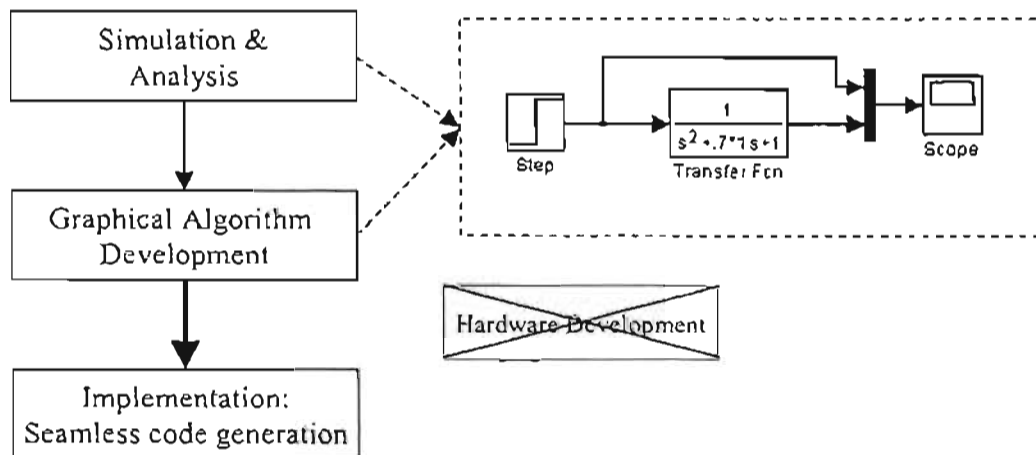


Fig. 2.2 : The rapid prototyping design

In the rapid prototyping approach shown in Fig. 2.2, it is evident, that the hardware stage has been removed allowing a seamless step from algorithm to implementation. Further more the algorithm stage uses a high level graphical language [MATHWORKS6]. The appeal of rapid prototyping over the conventional approach is that:

- The user need not be skilled in software engineering.
- There is no need for multi-disciplinary design teams.
- There is a dramatic time saving.

The rapid prototyping approach eliminates the hardware development stage by using generic hardware platforms. These platforms consists of DSP or microprocessors, with I/O devices that interface to the sensors and actuators on the plant being controlled [DSPACES5].

From the above discussion it is apparent that rapid prototyping is not a replacement for the conventional approach but rather an alternative that allows the designer more flexibility at the simulation and analysis stage. Rapid prototyping applications provide the user with more functionality and less complexity. They do not compete with the conventional approach but are rather an augmentation to it.

2.2.3 The Role of Rapid Prototyping In Education

Rapid prototyping in an educational context is targeted towards courses that rely on real-time systems

[SADASIVA1]. These include control systems, robotics and DSP to name a few. There are both teaching and research applications with research being further split between users and developers of rapid prototyping tools³.

The appeal of rapid prototyping is that it now frees the user from the need to be highly skilled in software engineering and allows advanced hardware to be used by a “lay” audience. Researchers using rapid prototyping are freed to concentrate on their applications rather than being distracted by the peripheral hardware and coding design issues. Educators are able to present better courses, as students are able to immediately evaluate theoretical concepts experimentally.

The teaching of control system is a good example of a theoretically intensive course that benefits from rapid prototyping tools. Lecturers can concentrate on the core concepts that are independent of implementation issues while at the same time allowing students to apply these concepts to a live real-time systems. The traditional design of a motor controller, which normally ends with students merely simulating it, can now be followed up with immediate implementation. Students can also investigate the practical issues of controller stability and plant saturation, which are difficult to appreciate without a live system.

Another aspect that needs consideration in the development of rapid prototyping educational tools is their ability to be networked. Network features provide value adding by maximising utilisation of expensive resources⁴ while also minimising total system costs. Further, with Virtual Laboratories⁵, [HAMMANN1, HUGH1, REID1] based on Internet technologies gaining in popularity, it makes a compelling argument to providing network supported rapid prototyping tools that can be easily incorporated into such laboratories.

Within the rapid prototyping arena there are various tools and in the next two sections products from Hyperception and The Mathworks are reviewed, as they are both being used by the Motion Control Group.

³ The work in this thesis falls into the rapid prototyping development category.

⁴ These resources include the DSP target hardware and external plant being controlled

⁵ Virtual Laboratories use Internet technology to allow student 24-hour access to experimental apparatus.

2.2.4 Hypersignal

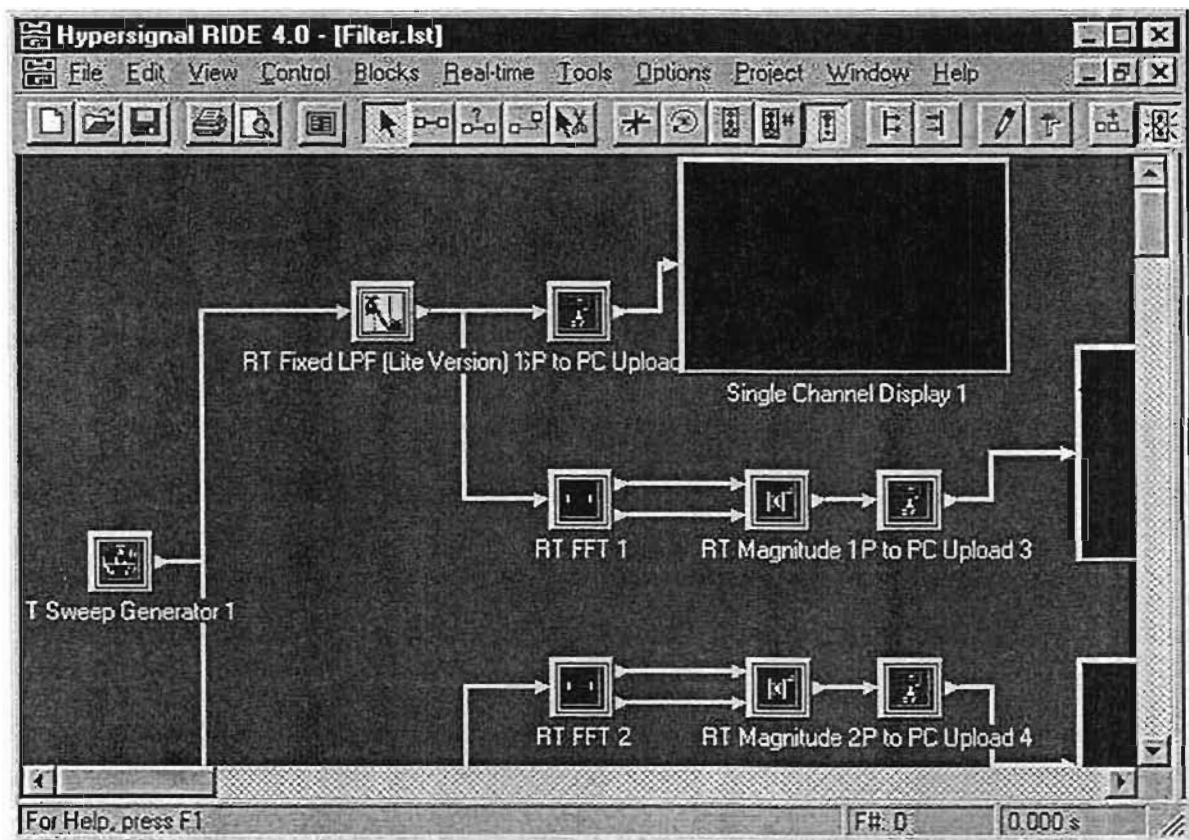


Fig. 2.3: Hypersignal environment

Hyperception Inc. produces the Hypersignal rapid prototyping tool, which uses a graphical environment for algorithm development as shown in Fig. 2.3. It however lacks a comprehensive simulation environment but allows algorithms to be tested on the PC platform before being executed on the intended hardware platform. This tool is targeted mainly for advanced DSP development and is provided with drivers, which support numerous third party DSP cards [BLERK1].

Hypersignal is being used by the Motion Control Group to implement and evaluate motion control applications and tools. A Variable Speed Drive (VSD) test bed tool has been developed to evaluate the performance of VSD's for different load conditions [WALKER1]. Worthmann [WORTHMANN1] evaluated the use of an artificial intelligence algorithm for the control of a boost rectifier. Both these researchers have stated that a considerable portion of their time was spent customising⁶ the Hypersignal environment for their applications. Further, simulations of these systems were also performed using other packages.

⁶ This involves the hand coding of application specific blocks that can then be used with the Hypersignal environment.

The Hypersignal package provides a powerful algorithm development tool that concentrates on traditional DSP development and is ideally suited to research applications. However it lacks simulation and analysis components and requires a steep learning curve, making the system unfeasible as a teaching tool.

2.2.5 The Mathworks Rapid Prototyping Framework

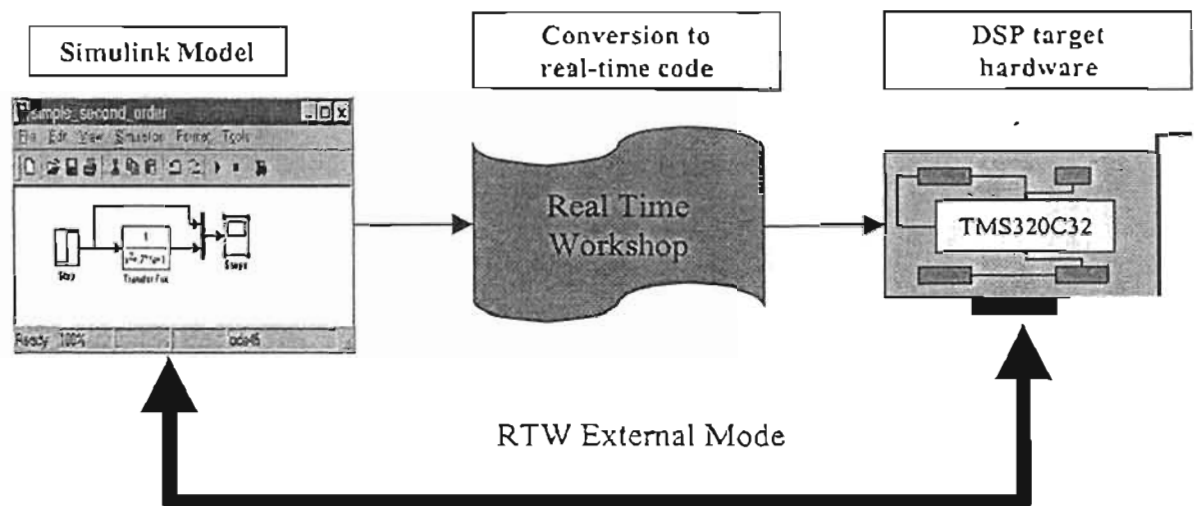


Fig. 2.4: Simplified operation of Mathworks RTW

The Mathworks provide the Real Time Work (RTW) [MATHWORK4, 6], which is integrated with the widely used Simulink environment [PA11, POUS1]; together they form an open architecture rapid prototyping framework. Fig. 2.4 shows a simplified representation of the operation of the RTW, which consists of:

- Simulink, which is a graphical simulation and analysis tool.
- The RTW, which is a toolset and framework responsible for the entire rapid prototyping process.
- Target hardware is the platform on which the generated code runs. The open architecture of the RTW allows users to incorporate third party hardware within the RTW. This process is referred to as RTW targeting⁷.
- RTW External Mode is a feature that allows Simulink to connect to the target platform, whereby target operation may be controlled from within the Simulink environment. This feature allows for:
 - Communication to remote targets.
 - Control of start/stop actions on the target.

⁷ Some of the RTW targets supported by The Mathworks are: DOS real time; Windows 95/98/NT; VxWorks

- On-line parameter tuning and data logging.

Due to The Mathworks open architecture both academic and many commercial developers have extended the RTW to incorporate their custom hardware platforms. From the literature [KOZICK, STYLO1, GANI, TQ1, DSPACE1, 2, 5] there are namely two approaches to use the RTW, either to use commercial products or to develop in house implementation. Both approaches provide viable educational tools, and the Motion Control Group has opted for in house solutions, as development of rapid prototyping tools is one of its active research objectives.

Sections 2.3 and 2.4 respectively, provide a look at some of the commercial and academic solutions available. These sections highlight the operation and general feedback from the use of these tools in both the research and teaching environments.

2.3. Third Party Tools Based On The Mathworks RTW

This section reviews commercial solutions based on the RTW framework and concentrates on companies that are providing viable educational tools.

2.3.1 dSPACE

dSPACE GmbH⁸ is German company that produces the **Total Development Environment (TDE)** [DSPACE1, 4, 5] rapid prototyping toolset. The TDE is based on the RTW and is targeted at both commercial and academic environments. It consists of the five components, shown in Fig. 2.5 and is explained below.

1. The Mathworks RTW has been explained in section 2.2.5

2. Cockpit

This utility replaces Simulink's external mode control, but affords user similar functionality to interface to the target hardware for parameter tuning and visualisation. Using the Cockpit tools users can develop custom GUIs that contain graphs, gauges, slider gains, knobs, etc.

3. Trace

This utility is a digital oscilloscope, which allows the user to get time histories of block outputs.

4. MLIB and MTRACE

Theses are Matlab toolboxes from dSPACE that provide similar features to the Cockpit and Trace utilities from within the Matlab environment. The purpose of these toolboxes is to allow the use of Matlab analysis tools for on-line, real-time data

⁸ www.dspace.com and www.dspace.de

analysis in applications like system identification and control optimisation.

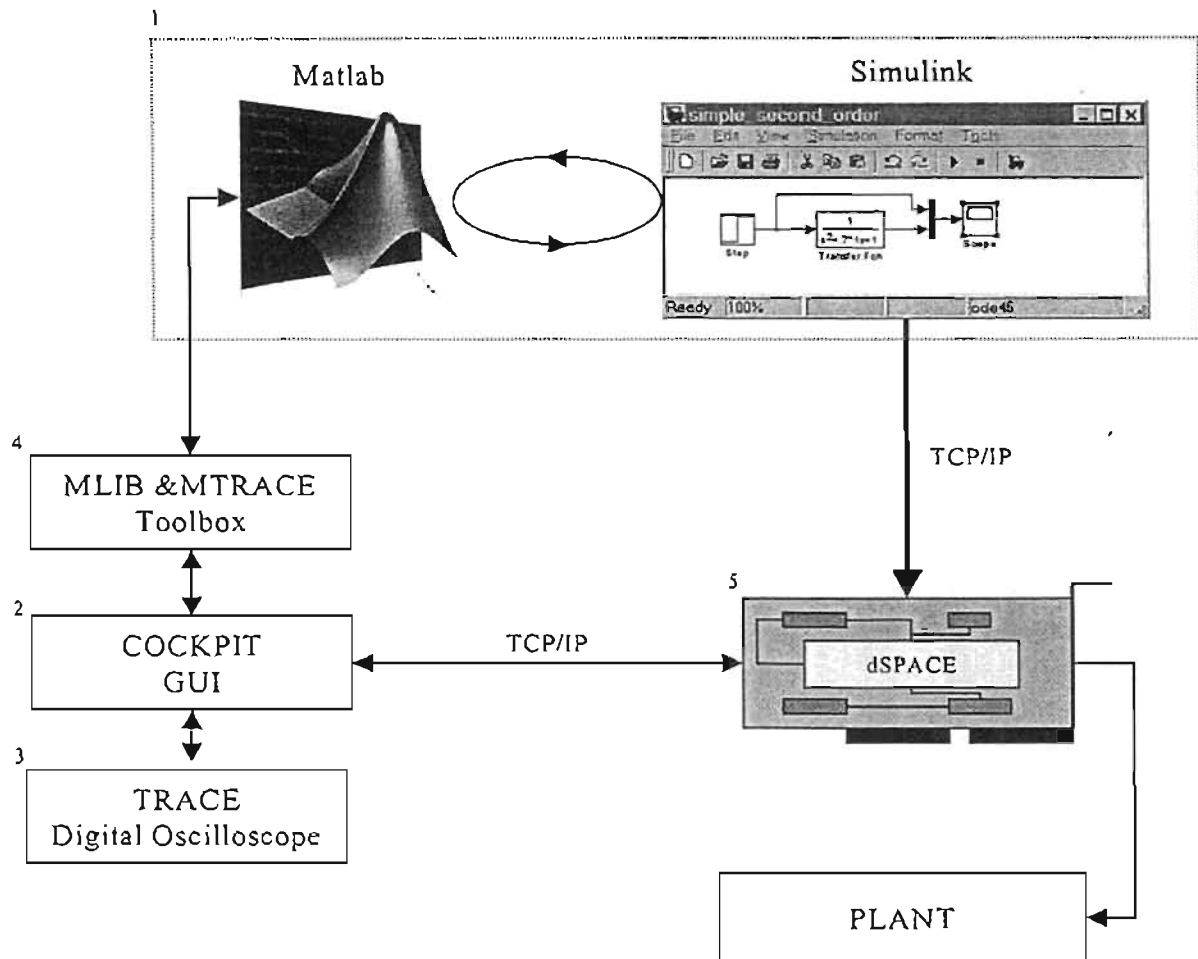


Fig. 2.5: dSPACE TDE toolset

5. Real-Time Hardware

dSPACE provides a range of hardware processor cards that consist of either DSP or microprocessors processing elements, with combinations also available. These cards use either the Texas Instruments DSPs or DEC ALPHA microprocessors [DSPACE2, 3]. In addition I/O cards are provided that interface directly to processor cards, and connect to plant sensors and actuators. There are also Ethernet network cards that allow dSPACE target platform to operate in network environments.

2.3.2 Use of dSPACE in Engineering Education

dSPACE tools are being actively used at numerous academic institutions⁹. This section provides a review of some of this work and gives researcher's and student's opinion of these rapid prototyping tools.

⁹ Bucknell University [KOZICK1], The Mechatronics Laboratory at the Royal Institute of Technology [FENG1], University of Girona, [POUS1], University of Technology, Nanyang Singapore [GAN1].

Bucknell University [KOZICK1] USA has incorporated dSPACE tools into several of their courses, which included:

- Exploring Engineering, a first year introductory course.
- Control systems.
- DSP.

A functional diagram of the rapid processing laboratory is shown in Fig. 2.6. The emphasis at Bucknell is to allow a large volume of student to experiment with a wide variety of real-time application¹⁰. This is accomplished by using a network environment.

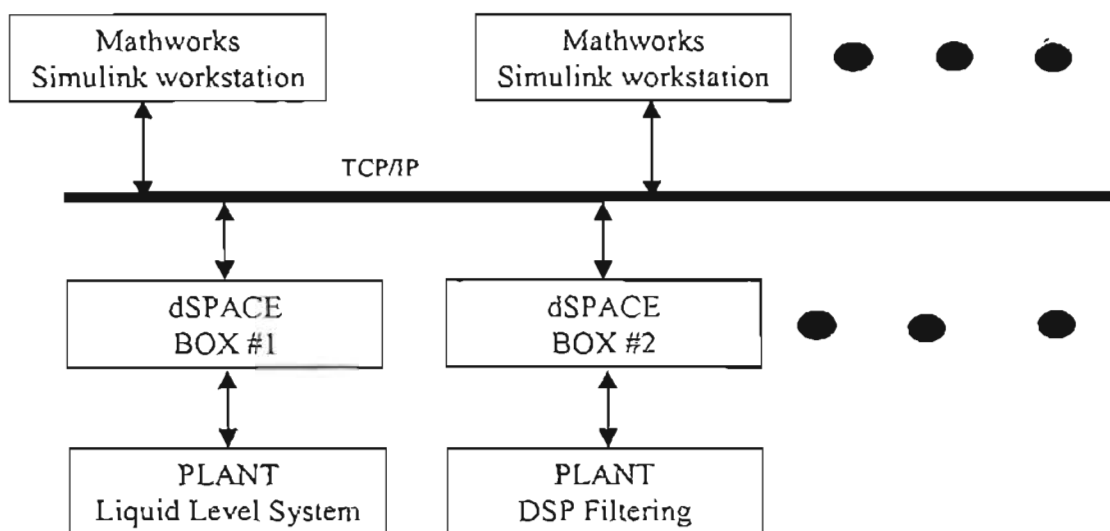


Fig. 2.6: Rapid prototyping laboratory

The experiences at Bucknell have shown that rapid prototyping has allowed under-graduates to access advanced real-time applications, which give the students a greater appreciation and understanding of theoretical courses.

At the Mechatronics Laboratory at the Royal Institute of Technology [FENG1] in Sweden, Prof Jan Wikander has been using dSPACE tools for a masters level control course. His reasons for using rapid prototyping tools are that they bridge the gap between the theory of automatic control and implementation issues; students from a mechanical engineering background, with limited advanced software experience are able to implement real-time control systems. Further more the Simulink environment lends itself to an easier understanding and visualisation of control system models. Feedback from students has shown that they have positively received the dSPACE tools and Prof. Wikander's controls course.

¹⁰ These applications include: a liquid level control system; servomechanisms for position of laser pointing devices; magnetic levitation of a metal ball; various DSP filtering applications.

The work by Sadasiva *etal* [SADASIVA] and Virvalo *etal* [FENG1], presented below, shows a different application for dSPACE tools. These researchers use rapid prototyping in their actual research.

Sadasiva *etal* [SADASIVA1] have used dSPACE tools to evaluate various PWM control algorithms¹¹. dSPACE tools have allowed them to quickly implement different control algorithms and evaluate their performances. Sadasiva *etal* have stated that rapid prototyping tools have dramatically reduced their experimental time from months to a single week.

Prof Virvalo [FENG1] and colleagues from the Institute of Hydraulics and Automation in Tampere, have used the dSPACE toolset to evaluate several controllers, which included; one degree of freedom pneumatic servo drive; one degree of freedom hydraulic servo drive; two degree freedom hydraulic crane. Their feedback shows that the dSPACE toolset provides a powerful platform to implement complicated controllers with relative easy.

From the above discussion it is evident that dSPACE provides a useful rapid prototyping toolset for both research and teaching applications. The only notable drawback with this toolset is cost and learning curve involved with the use of the Cockpit and Trace utilities [STYLO1].

2.3.3 Complete Experimental Solutions

Quanser Consulting, Inc¹² [WINCON1] and TecQuipment Limited¹³ [TQ1] are two of several companies that provide complete (canned) experimental apparatus used in the teaching of control systems and aligned fields. Their solutions are based on the RTW framework and are mainly targeted for teaching applications. Both Quanser and TecQuipment provide similar solutions and it is sufficient for the purposes of this thesis to discuss only one: Quanser was chosen as they provide more information on their products at their website.

Linear Experiments	Rotary Experiments	Specialty Experiments
Linear Position Servo	Rotary Position Servo	3 Degree of freedom (DOF) helicopter
Inverted Pendulum (IP)	Ball and Beam	2 DOF helicopter
Self Erecting IP	Rotary IP	Magnetic levitation
Linear Gantry Crane	Double IP	Planar Rotary IP
Double IP	2 DOF IP	Coupled Tanks

¹¹ These algorithms include: PI voltage controller cascaded with predictive current control; Model based voltage control cascaded with predictive current control; PI voltage controller cascaded with Vector current control.

¹² www.quanser.com

¹³ www.tq.com

Table 2-1: List of experimental apparatus from Quanser

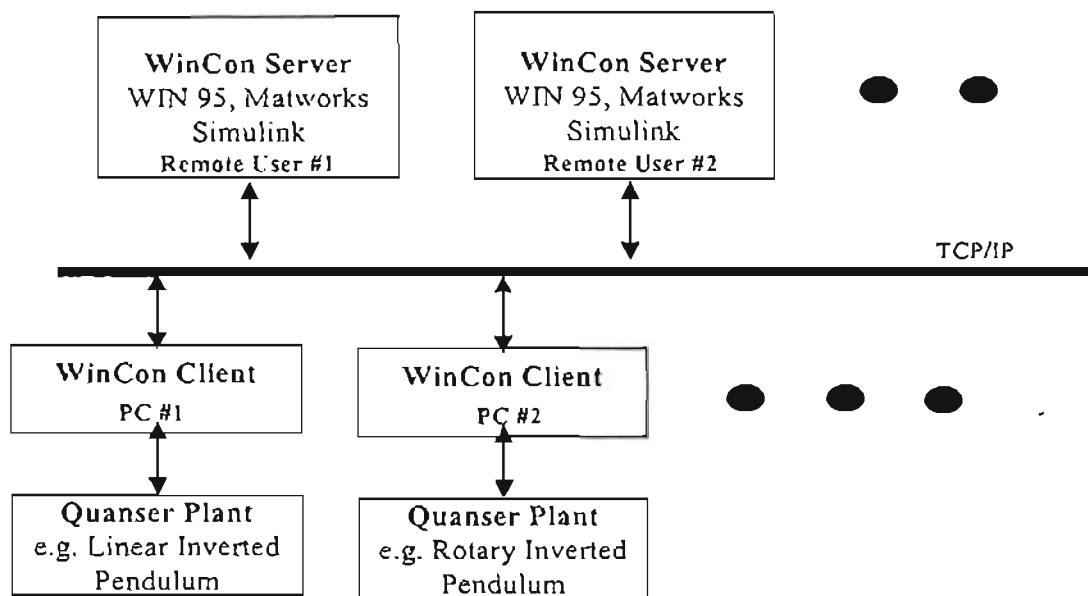


Fig. 2.7: Function diagram Quanser system

Quanser provide a wide range of experiments, which include, linear; rotary; and speciality experiments: a list is shown in Table 2-1. A functional diagram of the Quanser networked WinCon system is shown in Fig. 2.7, which consists of a WinCon server that is responsible for interfacing to Simulink and WinCon client, which is the real-time harness that runs generated model code¹⁴. An interesting feature of the systems from Quanser and TecQuipment is that they both use the soft real-time functionality of the Windows platform to execute target code (PC I/O cards are used to interface to external plants.). This technique minimises system cost and complexity by eliminating the DSP hardware. The drawback of this approach is that:

- The soft real-time makes sampling and latency times unpredictable¹⁵.
- The rigid interface requirements limit the use of user designed custom plants.
- The PC platform also diminishes the application of these systems to courses that require the student to get DSP and embedded system experience.

2.4. Academic Developments with RTW

2.4.1 Control System Development Environment

The Control System Development Environment (CSDE) was developed for the Motion control

¹⁴ WinCon can also run as a standalone system, whereby the WinCon server and client run on one PC.

¹⁵ While the Window platform does not provide hard real-time functionality, it is being increasingly used in industrial applications that do not have strict timing specifications.

Group by Stylo and represents the group's first implementation of an in house rapid prototyping tool, and is therefore used as a reference point to the development of the RADE system. This section reviews the CSDE environment.

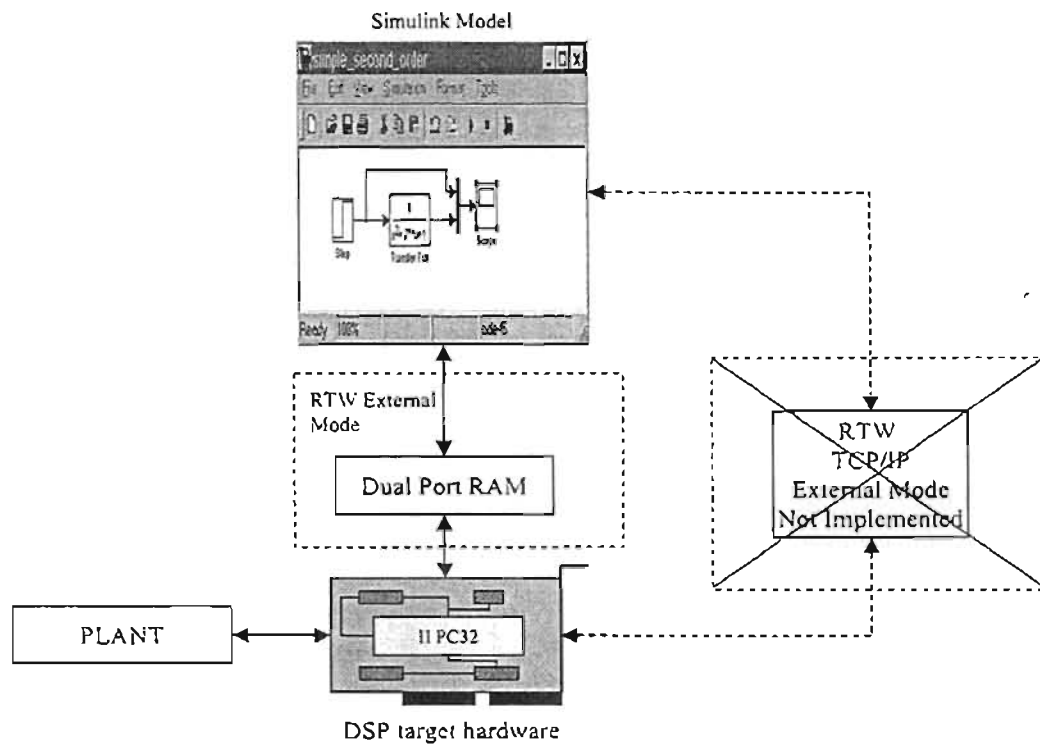


Fig. 2.8: Functional operation of CSDE

The CSDE is a cost effective rapid prototyping environment for both research and teaching application. The CSDE uses Simulink 2.2 and RTW 2.0¹⁶ and provides the following functionality:

- Seamless code generation
- On-line parameter tuning
- External visualization utility
- Support for the Innovative Integration PC32 DSP card
- Asynchronous interrupt support

A functional representation of the CSDE¹⁷ is shown in Fig. 2.8 and from this figure the following observation can be made:

- The network support of the RTW is replaced by DPRAM communication technique, which makes CSDE a solely standalone system.

The standalone impediment did not however affect the operation of CSDE and it was well received by

¹⁶ CSDE does not operate with newer versions of the RTW as The Matworks have updated the internal workings of the RTW.

¹⁷ Further details on the internals of the CSDE system are presented in chapter 4.

under-graduate students that used the system in their final year design course. Shawn Sturgeon [STURGEON1] implemented Field Oriented Control drive for an induction motor and Lynden Moodley [MOODLEY1] implemented a DC motor position controller. Both these projects would not have been feasible if it had not been for rapid prototyping tools. Additional support for rapid prototyping is that both students found control systems a theoretical abstract subject, with very little appreciation for its practical uses. This however changed during the course of their design projects as first hand experimentation allowed them to get to grips with control theory. This new found understanding culminated in them both winning design awards in their respective years

2.4.2 Work at Other Academic Institutions

The Mathworks RTW framework has been well received by other academic institutions, which have also opted to develop in house RTW targets. This section presents a review of some of these developments and how they were incorporated into course syllabuses at these institutions.

At the University of Girona, [POUS1] Catalonia Spain, Pous *et al* implemented a novel approach for a control system laboratory experiment in which they have used a robot named **ROGER**. A functional diagram of ROGER is shown in Fig. 2.9. The image processing system¹⁸ is used to provide XY coordinates of a target object. This information is fed to a Simulink designed controller that provides the actuators signals for the left and right motors. The objective of the controller is to track a moving target while maintaining a specified distance. The ROGER experiment uses a PC platform¹⁹ to implement the real-time target code and is aimed at allowing students to test different control algorithms easily without much emphasis placed on the underlying coding. This experiment incorporates both the simulation and live testing, which is an advantage of the Simulink environment.

¹⁸ This image processor is done on an independent processor and only the XY coordinates are feed to Simulink.

¹⁹ Pous *et al* did not specify if a DOS or Windows real-time target was used. It is the author's opinion that a DOS target was used, as it is a standard component of the RTW and provides hard real-time specifications, with a maximum sampling frequency of 400KHz.

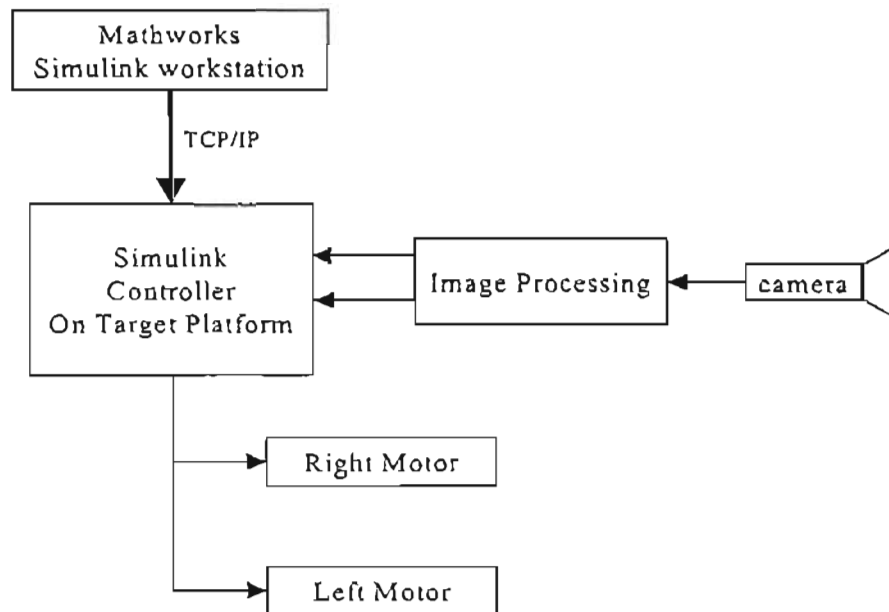


Fig. 2.9: Function diagram of ROGER the robot

At the University Mining and Metallurgy, [GREGA1] Krakow Poland, Grega *et al* have built a RTW windows target. Their implementation targets Windows 95/98/NT platforms with the emphasis of their work to evaluate the real-time performance of the windows platform. This in house windows target achieved a maximum usable sampling time of 10ms²⁰ and a latency of around 80μs. This windows target is feasible for soft real-time²¹ experiment but lacks the precision needed for hard real-time applications. The poor performance of this system is evident by Grega *et al* recommending the use of a commercial windows target, RT-CON by InTeCo Ltd, which provides better timing specifications.

At the University of Technology, Nanyang Singapore [GAN1], Gan *et al* have implemented a RTW target for the Texas Instruments TMS320C30 EVM board. The emphasis of this work was to produce a bare bones low cost DSP rapid prototyping teaching solution. A functional diagram of their solution is shown in Fig. 2.10. From this figure it is evident that the RTW external mode is not supported, as a result no on-line parameter tuning or visualisation will be possible. Gan's *et al* rationale for not supporting external mode is that quick repetitions of the RTW build process can be performed to change parameters, while an external oscilloscope can be used for visualisation. By using this technique a dramatic reduction in target complexity is achieved, albeit at the sacrifice of functionality. This system has been used in a DSP course at Nanyang University and allowed students to implement theoretically challenging DSP applications in short laboratory sessions. This would not have been

²⁰ Using the Windows WIN32 API a maximum of sampling time of 1 ms can be achieved using TIMER objects (ring 3). It is also possible to use PC's 8254 timer for higher sampling time but Grega *et al* did not do this.

²¹ Soft real-time applications don't require strict specification on: sampling time; interrupt latencies; and interrupt pre-emption and priorities.

possible without the use of rapid prototyping tools and student's feedback has shown a positive response to this method of teaching.

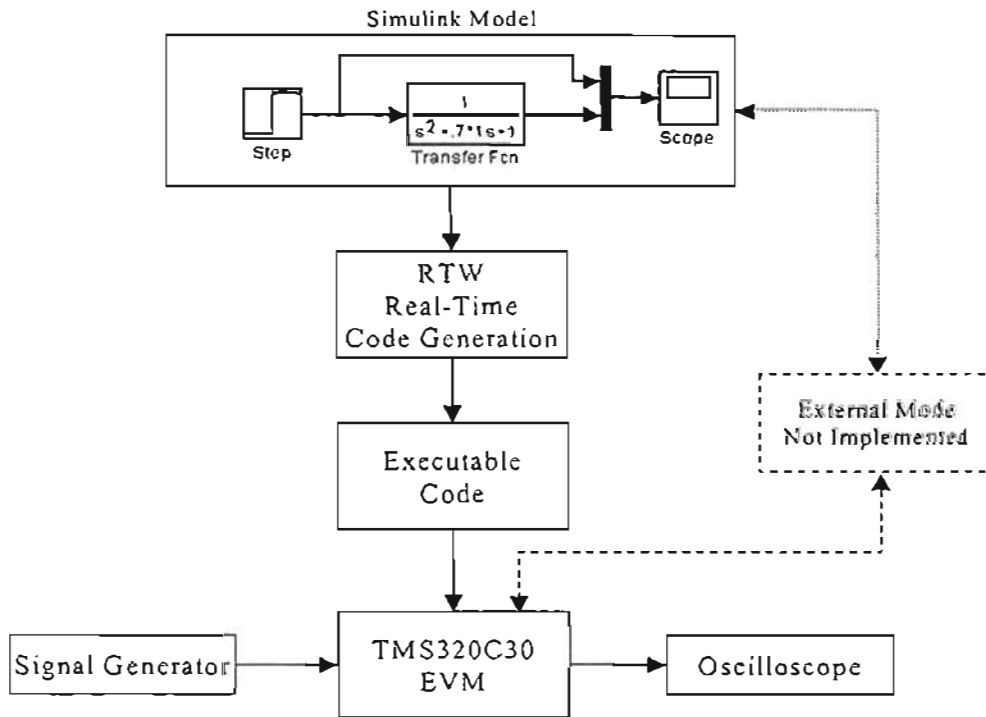


Fig. 2.10: Functional diagram for TMS320C30 EVM RTW target

2.5. Context of Work Presented in this Thesis

From the preceding sections, it is evident that The Mathworks RTW is being actively used in commercial and academic, educational endeavours. The work undertaken in this thesis follows in this vain and is targeted at educational applications. This section highlights the purpose and context of this work and finally ends with a summary of project goals.

Is this work mere duplication?

NO. Rapid prototyping and more specifically the RTW are active research topics, in which the Motion Control Group has been involved in from 1996 with the development of the CSDE system. Rapid prototyping is making significant inroads in engineering education and other institutions are also investigating its uses [GANI, GREGA1, POUS1]. The Motion Control Group as well as other academic institutions are also opting to develop in house rapid prototyping solutions that are based on the RTW framework. This route allows the researcher the flexibility to develop solutions around their courses and they are not hamstrung by rigid solutions similar to those produced by Quanser and TecQuipment.

The RTW workshop as a rapid prototyping framework is allowing interesting developments in the field of engineering education and therefore warrants the attention of any academic institution intent on using modern approaches to teaching. The Motion Control Group is therefore actively contributing to this process by the development of new RTW targets. Notwithstanding this, the RTW itself is an ever-changing framework, which allows greater functionality with every version update and the Motion Control Group is in an ideal position to capitalise on new developments in this field. A point in case is The Mathworks development of rapid prototyping tools for FPGA's. These tools are being incorporated into the RTW framework and in the not too distant future it will help to drastically reduce development time for FPGA and hybrid FPGA and DSP systems²² [GORDON1].

The CSDE was the Motion Control Group's first attempt to use the RTW framework as a rapid prototyping tool, but has been outdated with the 1999 release of Simulink 3 and the RTW 3. The Mathworks constantly upgrades the RTW and there have been significant changes and improvements from RTW version 2 to 3. As a result tools based on the RTW have to be constantly maintained and upgraded. Consequently a methodology is needed to ease version upgrades and this was not provided with the CSDE system. The design approach adopted for the CSDE system was to provide an operational system while minimising system complexity. This resulted in Stylo [STYLO1] removing network support and not adhering to The Mathworks conventions, as a result CSDE met the operational criteria but lacked the framework requirements for revision changes and inclusion of other DSP cards²³. In the development of the CSDE system no attempt was made to document the internal workings of the RTW in respect of the TCP/IP external mode implementation. While this did not impede the operation of the CSDE system it did hinder revision changes, as The Mathworks implements revisions using the TCP/IP external mode framework. This thesis aims to redress these issues and provide a system that adheres closely with Mathworks conventions.

2.5.1 Categories of RTW Targets

For the RADE system to be properly contextualised it is necessary to group work in this field into different categories of RTW targets. This allows for an appreciation of which category the RADE system is targeting and the intended functionality. Developments with the RTW target can be categorised into namely types:

1. High-End Systems

²² The University of Natal's Radio Access and Transmission Centre is currently involved in CDMA research and uses FPGA to implement high-speed signal algorithms. It is therefore envisaged that the RADE framework will be expanded to accommodate FPGA targets.

²³ CSDE only supports the PC32 DSP card

These systems use targets with powerful real-time OS's and have either a DSP or PC processor or a combination thereof. These systems are targeted at advanced commercial and research applications, which include avionics and military uses. In this category issues of hard real-time specification and code performance are well analysed. The systems developed at this level have immediate end product uses and are not solely used for evaluation purposes. Companies producing solutions at this level include dSPACE [DSPACE2, 5] and Wind River Systems [MATHWORKS4].

1. Medium-End Systems

These systems can be broadly categorised as the PC RTW targets segment and are primarily used in applications with soft real-time specification and for algorithm evaluation purposes. The advantage of these systems is that they can be easily networked and can draw on sophisticated PC visualisation techniques. The systems from Quanser [WINCON1] and TecQuipment [TQ1] are good examples of this category of RTW targets

2. Low-End Systems

At this level barebones systems are incorporated into the RTW. These systems include medium to low-end DSP cards with no real-time OS and network support. These systems have adequate bandwidth for evaluation of simple control algorithms and are simple, cost effective platforms for educational applications. This presents a dilemma to commercial developers, as there is little financial benefit in producing products in this category. This has resulted in academics institution filling this gap and producing in house tools. Example of these develops include the work by Gan [GAN1] and Stylo [STYLO1].

The RADE system is a hybrid of the latter two categories, as it aims to incorporate the benefits of both the PC and DSP platforms:

- The PC will provide network support and visualisation.
- The DSP target will provide real-time code execution.

2.5.2 Summary Of Project Goals

1. Incorporate maximum RTW version 3 functionality into the RADE framework with the following support:
 - Seamless code generation and downloading.
 - On-line parameter tuning.
 - On-line visualisation within Simulink.
 - Full network support

2. Document The Mathworks TCP/IP external mode implementation. This will allow for proper understanding of the Mathworks frameworks and facilitate easy version revisions. It will also allow DSP targets to be networked and therefore maximise utilisation of an expensive resource, which makes the RADE system more attractive to teaching applications.
3. The development of the RADE framework must conform to an easily maintainable and scalable framework.
4. Apply the RADE framework to the PC32 and ADC64 target DSP cards from Innovative Integration [INNOVATIVE1, 3].

2.6. Conclusion

This chapter presented an overview of rapid prototyping and its application to the educational environment. It is clear from the literature that both commercial and academic rapid prototyping tools are finding increasing use in courses to provide students with more interactive laboratory experiments. Theoretical concepts, which are traditionally not easy or practical for students to implement in short laboratory sessions are now, becoming common place in rapid prototyping laboratories. In addition researchers who are not skilled in software engineering can use rapid prototyping tools to implement and evaluate complex applications in real-time.

The Mathworks RTW, which was also featured in this chapter, is being used extensively in engineering education applications. The RTW in conjunction with Simulink provides an effective teaching and research platform that is both scalable and customisable. The open architecture of the RTW allows both commercial and academic rapid prototyping tools to be incorporated and is being well received by students and researchers alike.

While this chapter highlighted uses of the RTW, it did not provide a detailed discussion on the internal workings of Simulink and the RTW. These details are necessary for the implementation of RTW targets and are presented in the next chapter.

CHAPTER THREE:

OVERVIEW OF SIMULINK AND THE RTW

3.1. Introduction

This chapter provides an overview of Simulink and the Real Time Workshop, products from The Mathworks, which form the core elements of the RADE system. A large part of the chapter is devoted to RTW conventions, as this information will be required in subsequent chapters. The aim of this chapter is to highlight the important aspects of Simulink and the RTW in the context of work presented in this thesis.

Matlab and Simulink, which form the core products from The Mathworks, are becoming popular tools for modelling and simulation in academic environments [HUGH1]. These products are being widely used in both undergraduate and post-graduate teaching the world over, and are also acquiring a large research user base due to the advanced add-on toolboxes [KOZICK1, GAN1, MATHWORK1]. With the Real Time Workshop add-on extension, Simulink becomes a powerful open architecture, rapid prototyping environment with numerous applications in teaching and research. The RADE system is one such application

3.2. Products Available From The Mathworks

Matlab is the core product from The Mathworks, which is tailored for fast and efficient numerical computation and visualisation. Matlab, which stands for *Matrix Laboratory* is a high performance technical computing and visualisation tool, which provides an easy to use environment where problems and solutions can be expressed in mathematical notation. This product is further enhanced by the availability of numerous advanced application specific toolboxes. The Mathworks also produces a dynamic graphical simulation environment called Simulink, which runs above the core Matlab engine. Simulink is also available with application specific add-on blocksets. Fig. 3.1 shows the entire Mathworks product range and distinguishes Matlab and Simulink components. Matlab and Simulink are well-established software packages, which are being used in both the commercial and academic fields. Its open architecture, which allows for the easy development of customer specific solutions, is a further reason for its wide scale use. Both these packages have become the standard instructional tool for courses in engineering, mathematics and science [MATHWORKS1].

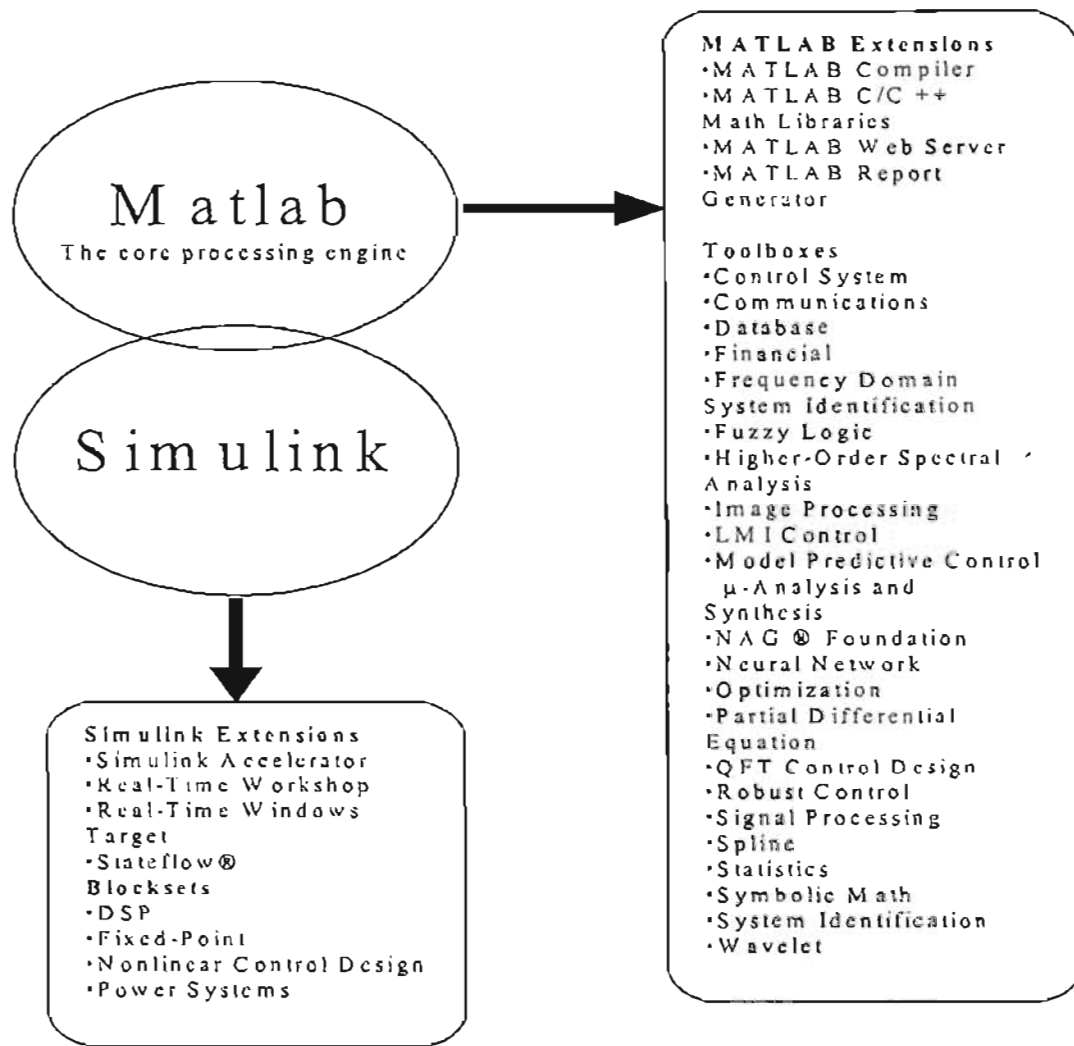


Fig. 3.1: Product range from The Mathworks

The Mathworks provides an extensive set of computing tools with a complete list shown in Fig. 3.1. As shown in Fig. 3.1 The Mathworks extension products, allow Matlab and Simulink functionality to be exported to external programs, by means of the C code or web server applications. There are also two categories of add-on products, toolboxes for Matlab and Blocksets for Simulink. As shown in Fig. 3.1 there are numerous Blocksets and Toolboxes from The Mathworks spanning a wide variety of topics and it is beyond the scope of this thesis to discuss all. Therefore material presented herein will only be that directly relevant to the author's work, of which Simulink and the RTW form a crucial part.

3.3. A Overview Of Simulink

“What is Simulink?”

Simulink is a software package for modelling, simulating and analyzing dynamical systems. It supports linear and non-linear systems, modelled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e. have different parts that are sampled or updated at different rates.”

Using Simulink Version 3,1999

Simulink forms an integral part of the work presented in this thesis, and it is therefore necessary to review this package to enable the reader to gain an understanding of it. A short overview follows in the next section, using a simple example to illustrate the operation of Simulink. It then focuses on the internal operation of this package, which is necessary to gain a proper understanding of the real-time code generation process.

3.3.1 An Example with Simulink

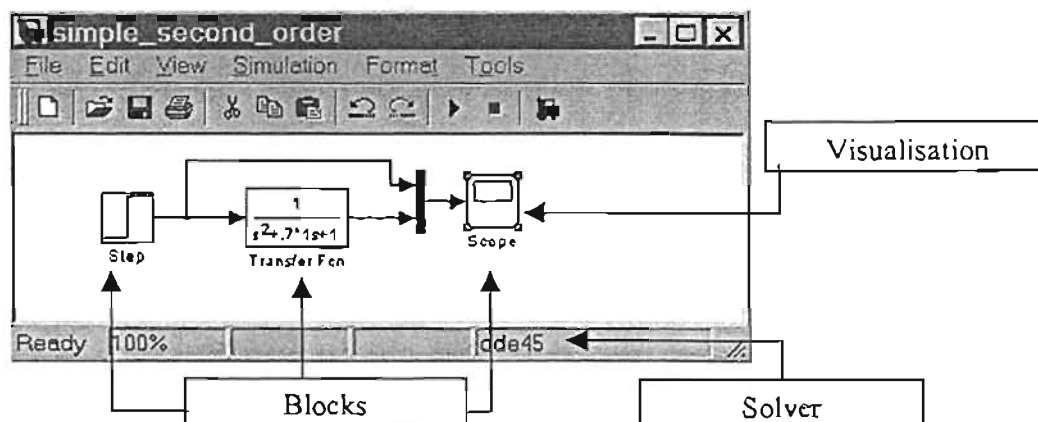


Fig. 3.2: A Simulink simulation

Fig. 3.2 is a Simulink simulation of a simple second order system to a step response and is used to outline the general aspects of a Simulink model. In Fig. 3.2 it is shown that a basic model consists of blocks, an integration solver and a visualisation method. These topics are expanded upon in the following sections

I. Blocks

Blocks are the elementary components of a Simulink model. There are numerous types of blocks ranging from simple mathematical operations to advanced application specific blocks. Fig. 3.3, shows some of the more common blocks available in the Simulink library [MATHWORKS2].

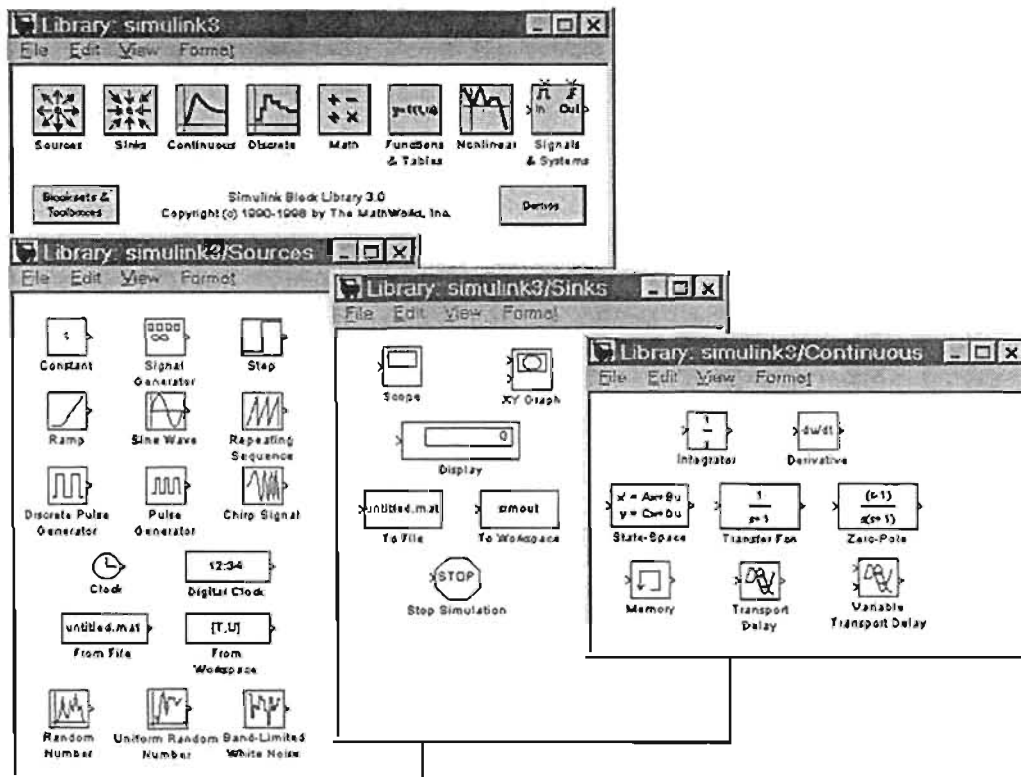


Fig. 3.3: A sample of the common Simulink library blocks

II. Libraries

Simulink uses libraries to group common blocks and they have the added advantage of allowing models that contain library blocks, to automatically update when the source blocks are updated. The common Simulink libraries have already been shown in Fig. 3.3

III. Simulation Parameters

1. The Simulink simulation parameters are set in the parameter window, as shown in Fig. 3.4. The import parameters are:

2. Start and stop time

These parameters are used to set the duration of a simulation.

3. The integration solver

The solver is a fundamental component of Simulink, which is responsible for the numerical integration of the ordinary differential equations of a model. There are various types of solvers varying from fixed to variable step algorithms [MATHWORK2 ,3].

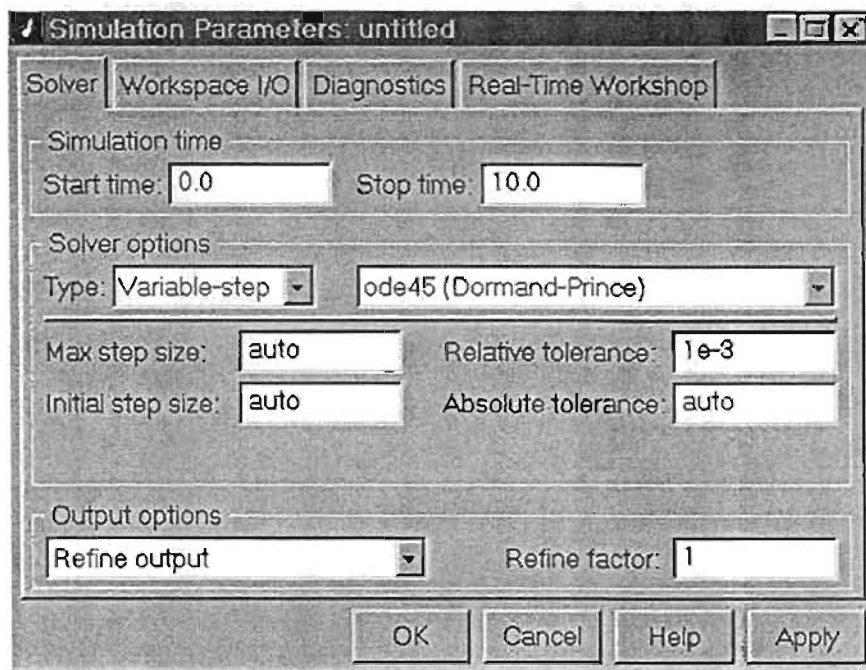


Fig. 3.4: Simulink simulation parameters

IV. Visualisation

Visualisation of simulation data is an important aspect of Simulink with which it allows the user to graphically interpret simulation data. There are two methods to view simulation data, firstly to use either a scope or XY graph block and secondly to pass the data to the Matlab workspace and then use one of the, extensive set of, Matlab¹ visualisation functions. Fig. 3.5 is an example of the typical output of a scope block.

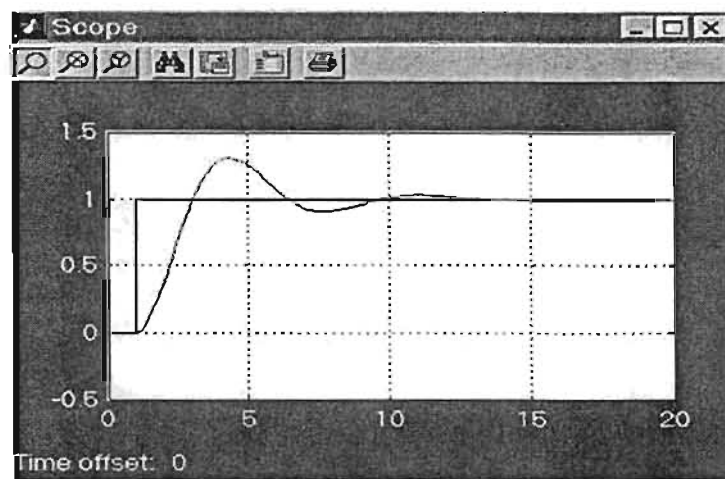


Fig. 3.5: The scope block output of the step response of the system in Fig. 3.2

¹ Simulink also supports interfacing to the Graphic User Interface (GUI) toolbox, which support the use of advanced animation routines.

V. Subsystems and Triggered Subsystems

Subsystems are used in Simulink to build hierarchical models by grouping parts of a model into subsystems². This technique allows models to be constructed in a top-down manner by placing high-level aspects on the first layer and “hiding” model details in lower levels. This makes for easier understanding and navigation of complex Simulink models. Fig. 3.6 details the concept of a subsystem of a simple case.

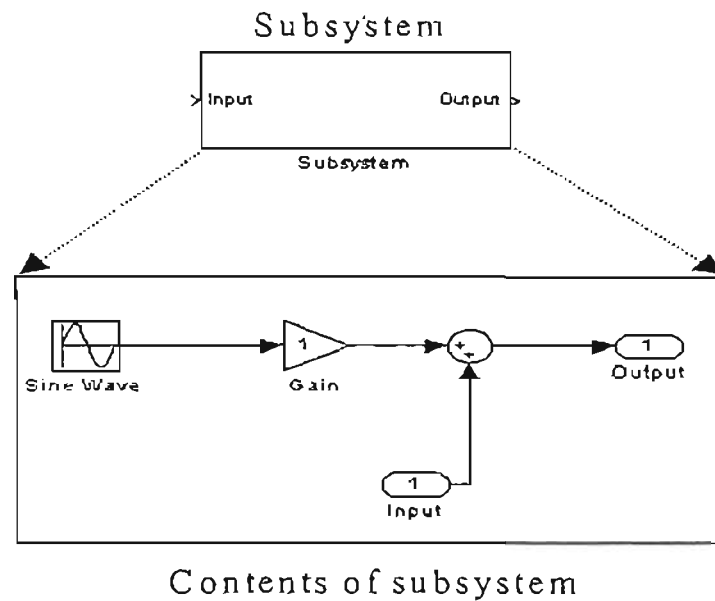


Fig. 3.6: An example of a subsystem

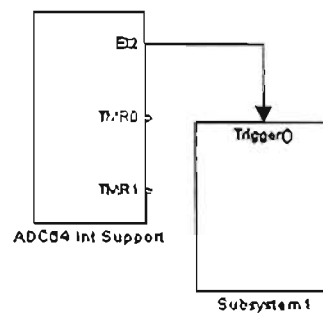


Fig. 3.7: An example of a triggered subsystem

An event driven subsystem is an extension of this process whereby a subsystem is only executed when triggered by an external trigger source. This technique allows a subsystem within a model to execute in an asynchronous mode, and is used in RTW implementations³ to synchronise different sections of a model to external signals. Fig. 3.7 shows a triggered

² A subsystem is a *Virtual Blocks*, these blocks play no active role in a simulation, they are merely a technique to organise a model in a graphically efficient manner.

³ The dSPACE GmbH implementation of the RTW uses this technique [DSPACES]

subsystem.

3.3.2 How Simulink Works

This section briefly outlines the internal operation of Simulink, which will help with the understanding of how the RTW converts a model into real time code. Every block within a model has the standard characteristic detailed in Fig. 3.8, with the mathematical relationship expressed in equations [3-1], [3-2] and [3-3]. It is evident from these equations that Simulink uses a general state space model to implement dynamical blocks [MATHWORKS2, 3].

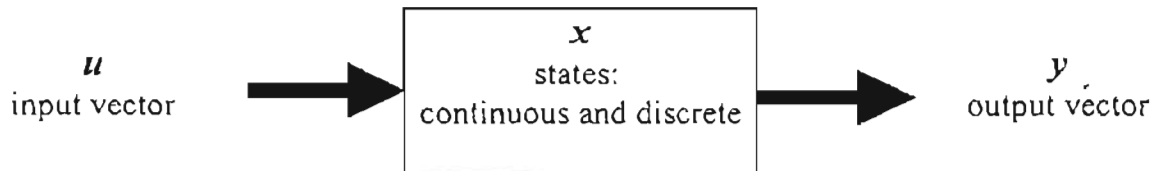


Fig. 3.8: General model of a Simulink block [

$$y = f_0(t, x, u) \quad \dots \text{(output)} \quad [3-1]$$

$$\dot{x}_c = f_d(t, x, u) \quad \dots \text{(derivative)} \quad [3-2]$$

$$x_{d_{k+1}} = f_u(t, x, u) \quad \dots \text{(update for discrete states)} \quad [3-3]$$

$$\text{where } x = x_{\text{continuous}} + x_{\text{discrete}}$$

The simulation process is a multistage process that is detailed in Fig. 3.9:

1. Initialisation

This step consists of:

- Evaluation of block parameters.
- Reducing model hierarchy to a single tier.
- Sorting blocks for execution order
- Verification of correct signal flow between blocks.

2. Simulation stage

The simulation stage involves the repetitive process of finding the state derivative of the model⁴ and then integrating it to find the next state value. This process is further enhanced by, integration algorithms, which support variable stepping and zero crossing detection [MATHWORKS2].

3. Termination

At this the stage termination routines for the model execute and output data is sent to the workspace.

⁴ A model's state and state derivatives can be evaluated because each block within a model conforms to equations [3-1], [3-2], [3-3].

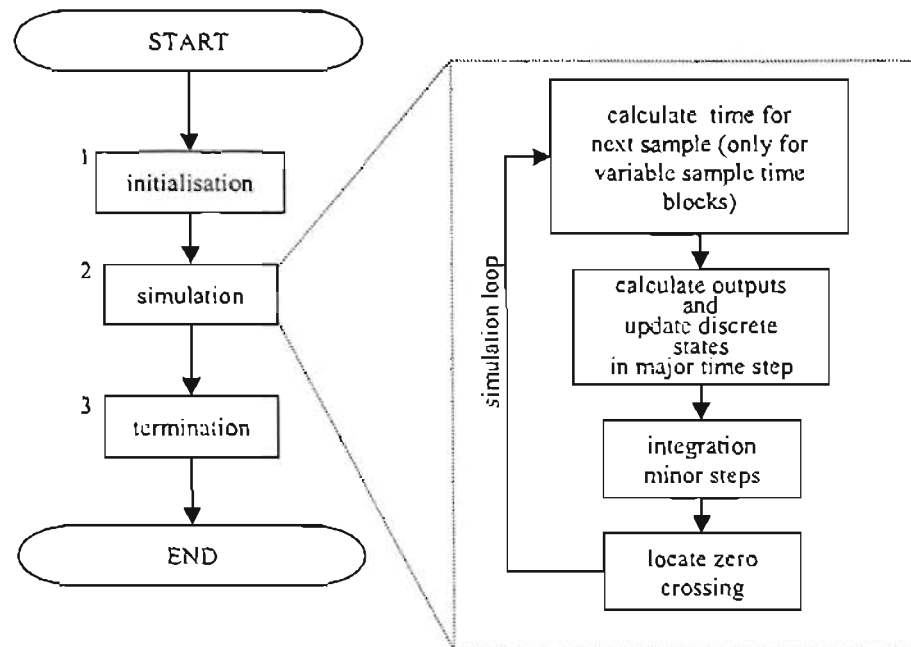


Fig. 3.9: Flow diagram of Simulink internals

3.3.3 An Overview of S-functions

S-functions are a window into Simulink [MATHWORKS3]; they allow third party developers to develop custom blocks for Simulink. It can also be said that all library blocks within Simulink conform to S-function semantics, and therefore understanding S-function allows for a richer understanding of Simulink. This section details the programmatic aspects of S-functions, which has important relevance to the RTW, since custom blocks for the RTW are developed using S-functions and generated code is modelled around S-functions; see section 3.7.3.

Mathworks provides two techniques for writing a S-function, either using a Matlab script or a C-Matlab Executable (C-MEX). Since only C-MEX files can be used with the RTW, they are discussed exclusively in the following sections.

1. How S-functions work

An S-function model is uniform for all Simulink blocks, the input to output relationship is shown in Fig. 3.8 and the mathematical model is described in equations [3-1], [3-2], [3-3]. All S-functions export a standard **Application Program Interface (API)** that Simulink can call into, Fig. 3.10 shows this process. Fig. 3.11 details the step involved in the writing and compiling of a C-MEX S-function. The next section defines The Mathworks conventions for the S-function API.

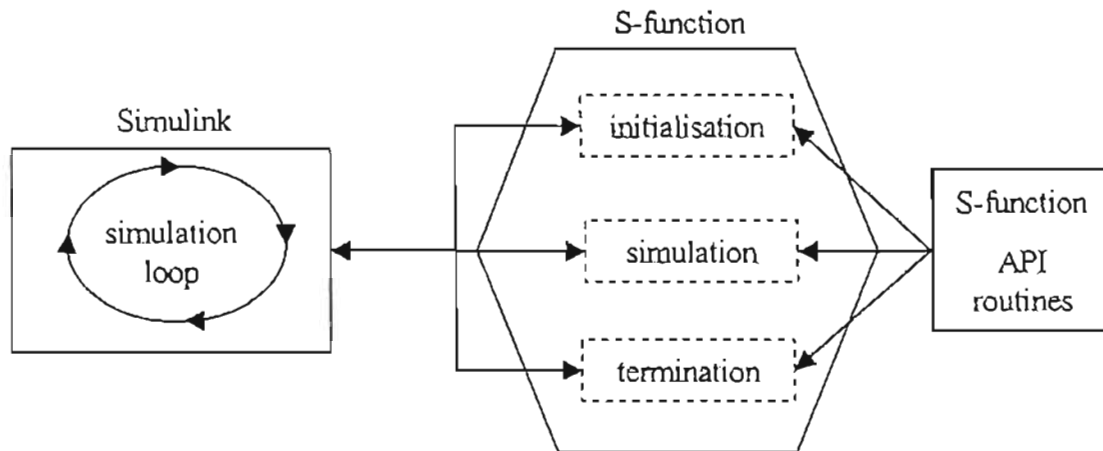


Fig. 3.10: How Simulink calls into a S-function

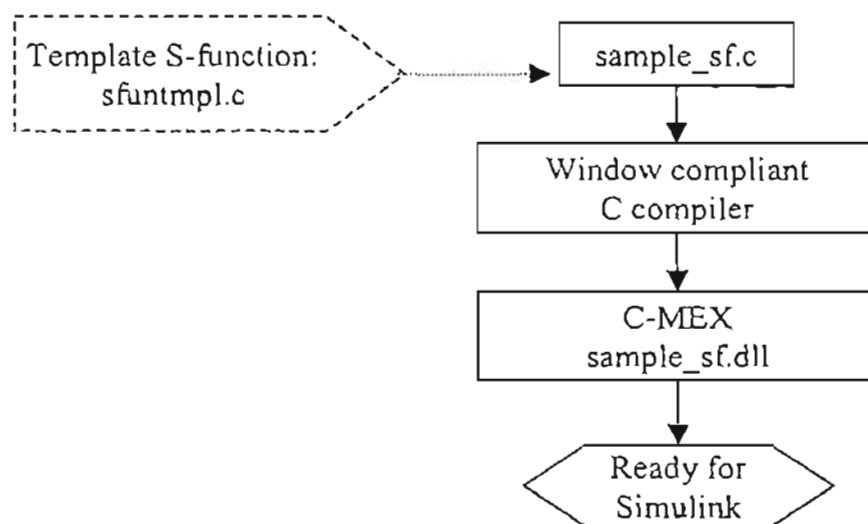


Fig. 3.11: The flow diagram of writing and compiling a S-function

II. The S-function API

Fig. 3.12 details the function calls for a S-function and their sequence of operation⁵.

1. mdlInitializeSizes

This is the first function called by Simulink when interfacing to an S-function. It is used to setup block characteristics like, input and output port sizes, number of parameters, block states, working variables and other block functionality.

2. mdlInitializeSampleTimes

This function is used to setup the sample times for a S-function.

3. mdlStart

This optional function is called once at the initialisation stage. It can perform any additional application specific initialisation.

⁵ A scaled down version of the S-function API is documented, covering only the section relevant to the work presented in this thesis.

4. mdlOutputs

This function calculates the outputs of a block when called. It uses the output equation shown in [3-1].

5. mdlUpdate

This optional function is used to update discrete state of a block.

6. mdlDerivatives

This optional function is used to evaluate the continuous time derivatives of a block, and use the state derivative equation shown in [3-2]

7. mdlTerminate

This function is called at simulation termination and is used to free up memory or

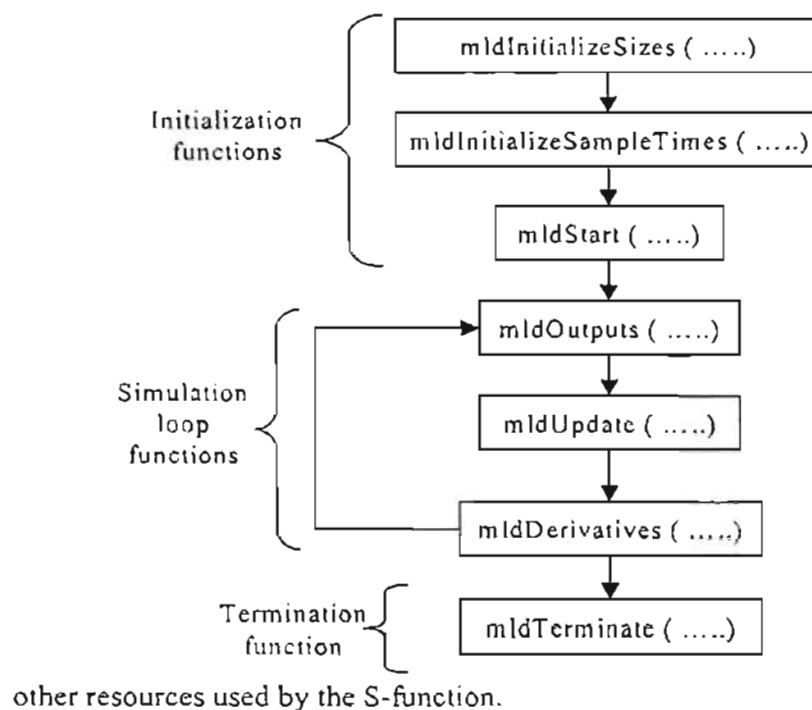


Fig. 3.12: API function and their calling sequence

III. SimStruct, the data object of a S-function

SimStruct is a data structure used by Simulink to manage all aspects of data transactions within a model. The various categories of data stored in SimStruct are detailed in Table 3-1. A Simulink model contains a single parent SimStruct and a child SimStruct for each block within the model. Another feature of SimStruct is data encapsulation, i.e. all data is accessed via macro functions, this allows for easy data manipulation without detailed knowledge of the complex SimStruct structure⁶.

⁶ For reference purposes the entire SimStruct declaration can be found in the `simstruc.h` file found on the accompanying CD.

Field	Data Type Stored
Version	Used for version control.
Parent	Pointer to parent SimStruct of current child SimStruct.
Root	Pointer to model main SimStruct.
Sizes	Stores S-function characteristic.
Parameters	Stores parameters passed to S-function
Work Vectors	Stores working vectors
Timing	Stores timing data
States	Store the states of a S-function
Derivative	Stores derivative data.

Table 3-1: Data stored in SimStruct

3.4. The Real Time Workshop

In this and subsequent sections the **Real Time Workshop (RTW)** (see section 2.2.5) is described, highlighting aspects relevant to the work undertaken in this project. It will concentrate on undocumented internal workings of the RTW.

What is the role of the RTW?

The RTW is responsible for the entire rapid prototyping process, which consists of:

1. Converting a model into real time code.
2. Building code⁷ and downloading to a target processor.
3. Provide a communication protocol for online parameter tuning, data logging and target signalling.
4. Program architecture of code executing on target hardware⁸

⁷ Building code refers to the two-stage process of compiling and then linking code into an executable module.

⁸ The RTW indirectly controls execution on the target hardware by specifying the run-time interface.

3.4.1 An Overview of the RTW

The RTW [MATHWORKS4] is provided with an open architecture, which can be broadly categorised into two parts, code generation and program architecture. The code generation process involves all the physical steps required for converting a Simulink model into a standalone executable module. The program architecture defines the structure of the code generated and deals with the issues of setting up a standard runtime interface, under which generic model code may be executed. With the RTW being inherently complex, a hierarchical approach is used to describe it i.e. a system level description is initially presented in the next two sections with subsequent sections containing the lower levels details.

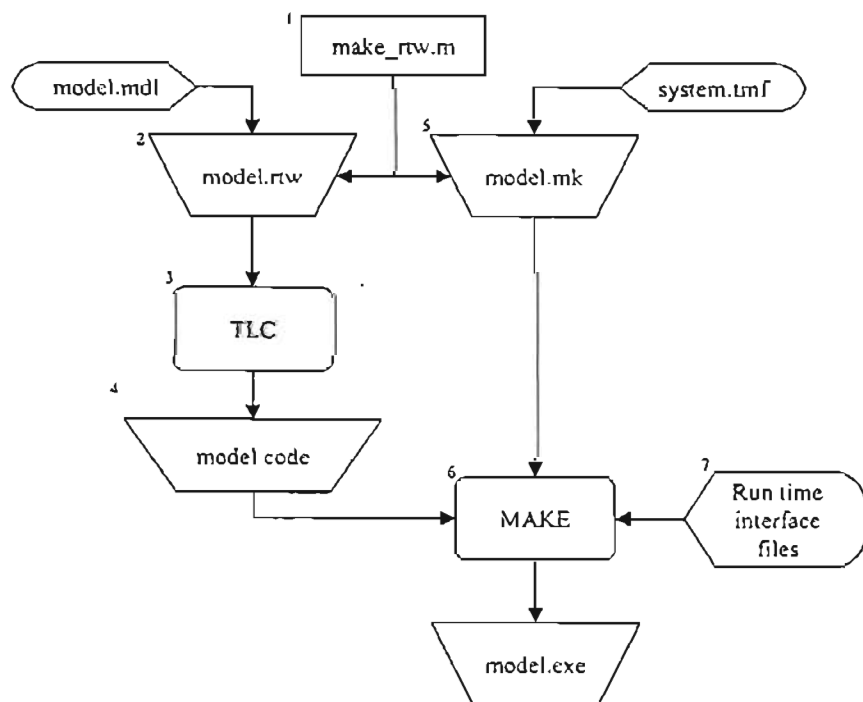


Fig. 3.13: Flow diagram of code generation process

1. Code generation

With the help of Fig. 3.13 the stages involved in the code generation process are detailed as follows:

1. `make_rtw.m`

This is the entry point of the RTW build⁹ process, i.e. it is the first function to be called when a Simulink model is converted to real-time code. The `make_rtw.m` is a

⁹ The entire process of converting a model into real-time code is referred to as the build process by Mathworks conventions, however the process of compiling and linking code into an executable module is also referred to by the same name. For clarity the former shall be referred to as the RTW build process.

Matlab function, or m-file, that controls the execution order of the different components involved in the RTW build process.

2. *model*¹⁰.rtw

This file is generated by a RTW function that converts the *model.mdl*¹¹ into the *model.rtw* file. The *model.rtw* file describes the Simulink model using a text convention (ASCII file), specified by the RTW. The reason for converting a *model.mdl* file into a *model.rtw* is to allow the RTW maximum flexibility, i.e. a *model.rtw* file can then be converted to any language by using the Target Language Compiler (TLC).

3. The Target Language Compiler

The TLC is a proprietary tool that forms part of the RTW, which is solely used for the generation of model code from the *model.rtw*. The model code is generated in the target language¹². Currently Mathworks supports conversion to either ADA or C, for the purpose of this thesis only C is used and therefore any future reference to the TLC will imply conversion to C files. See section 3.5 for more details.

4. Model Code

This represents the model in real time code. A full description of the various C files generated from the TLC is presented in section 3.5.

5. *model.mk*

This file is generated from the **template make file** and is used by the make utility to build the model code into a target executable module. The conversion of the *system.tmf* into the *model.mk* file is controlled by the *make_rtw.m* function. See section 3.6 for more detail.

6. Make Utility

A make utility is a programming tool that automates the build process of projects with numerous files. See section 3.6 for more details.

7. Run-Time Interface

This component of the RTW represents, the hardware specifics, timing scheduler, solvers, and communications layers. This interface represents the point at which the RTW is customised for various target platforms.

II. Program architecture

The program architecture details how the real time mode code executes. The RTW

¹⁰ The term *model* refers to a generic Simulink model.

¹¹ *.mdl file is the native format of a Simulink file,

¹² The **Target Language** refers to the programming language of the target system.

supports two code styles, one suitable for Rapid prototyping¹³ and the other for embedded applications¹⁴. For the purpose of the RADE system the Rapid prototyping style was chosen as it offers the most functionality and flexibility. The rapid prototyping architecture specifies a run-time interface that executes the generated model code. Fig. 3.14 illustrates an object-oriented view of the run time interface interaction with the model code. [MATHWORKS4]. Section 3.7 expands on this topic in more details.

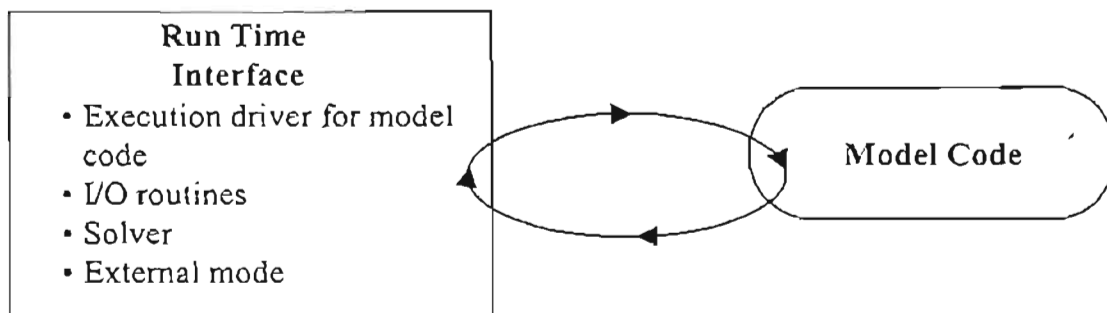


Fig. 3.14: Object-oriented view

3.5. Target Language Compiler

The **Target Language Compiler (TLC)** is a Mathworks tool that parses the intermediate model.rtw file into either ADA or C¹⁵. The generated code produced by the TLC is fully customisable for hardware specifics and performance tuning. The TLC is used to:

- Generate C code from the model.rtw file
- Modify generated code to accommodate for hardware or algorithm specifics
- Optimise code for size or performance
- Generate optimised code for user defined s-functions

The operation of the TLC is detailed in Fig. 3.15

1. sample.rtw file

This is the intermediate text file that represents the model described by the sample.mdl

2. System target file

This is the entry point of the TLC, which is synonymous with the main() function of a C program. The purpose of this file is to setup different implementations of the RTW. Section 3.5.1 provides more details.

¹³ The Rapid Prototyping architecture includes the simstruct data structure, which by default places a performance penalty on the real-time code.

¹⁴ The embedded version strips out the simstruct data structure and external mode functionality for better code performance.

¹⁵ The RADE system only uses C code, and any reference to the TLC will imply generation of C code.

3. Block target files

These files specify how to convert blocks in the model into C code. Section 3.5.2 provides more details.

4. Target language Compiler functions

The TLC uses these functions when generating code [MATHWORKS5].

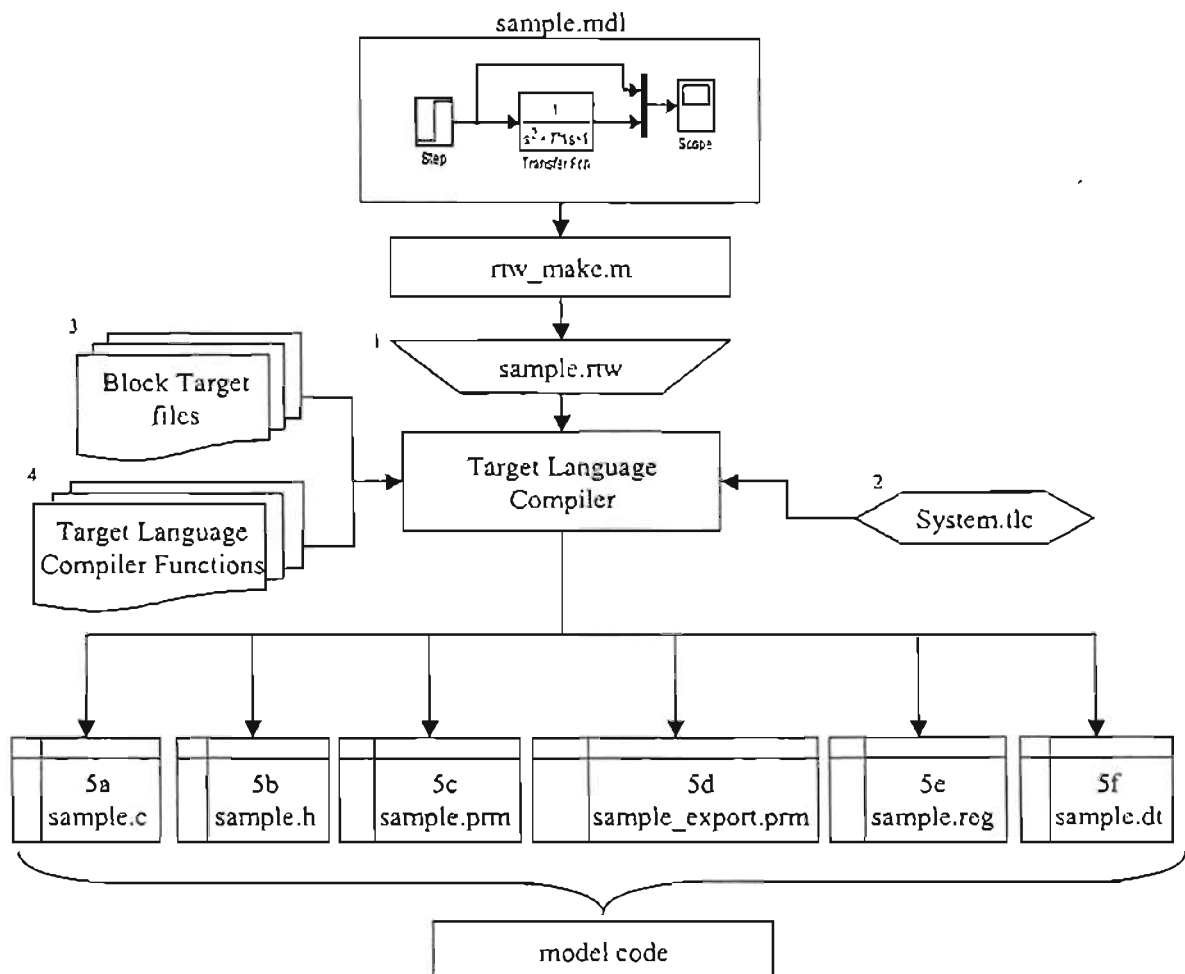


Fig. 3.15: Operation of the Target Language Compiler

5. Model code

The generated code consists of the following files:

a sample.c

Source file for implementing model algorithm.

b sample.h

Header file used for global variable definitions.

c sample.prm

Parameter files that defines models tuneable parameters

d sample_export.prm

Header file that contains source code produced by the user.

e sample.reg

Registration file, used for model registration and initialisation functions.

f sample.dr

This file contains model data types and is only produced if external mode is being used.

The TLC process discussed above shows how the model code is produced and the different files involved. Another important file in the TLC is the System Target File, which is discussed in the next section.

3.5.1 System Target Files

The RTW supports numerous target types and is customisable to include new targets like the RADE system. A system target file is used to identify and implement different real-time targets. The RTW is shipped standard with numerous real time targets and their corresponding system target files are listed in Fig. 3.16.

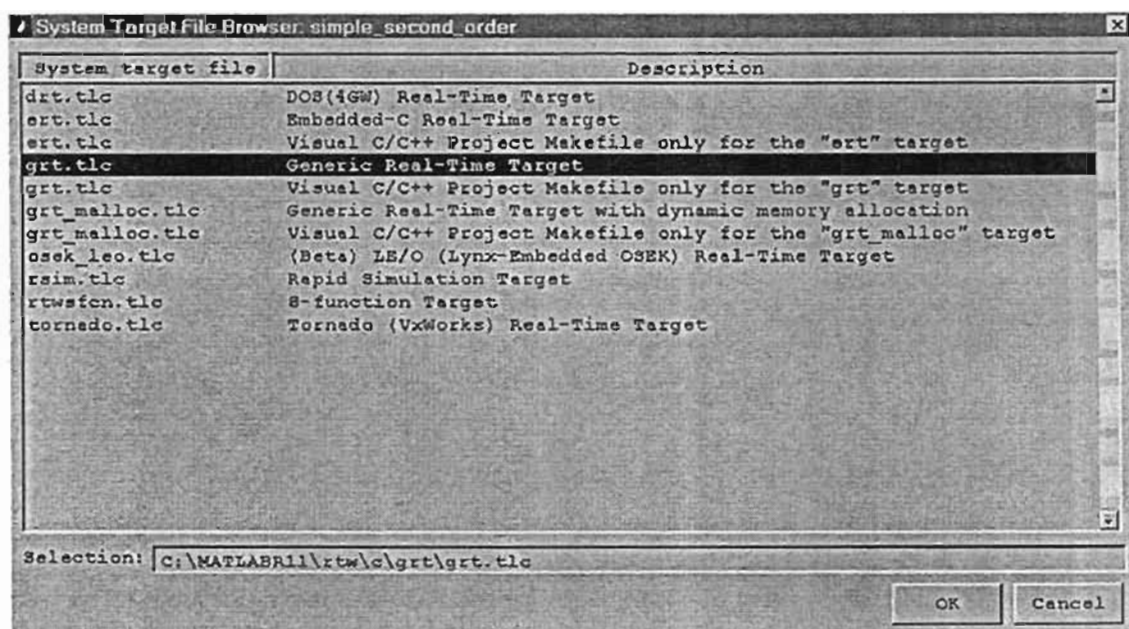


Fig. 3.16: List of system target files

I. Structure of System target file

The simplified structure of a System target file is listed below, which is composed of 4 sections. Section one is used to setup default parameters for the RTW build process. Section two, defines various variables used for the TLC process. Section three is used to include the code generation function, which generates the actual code. Section 4 is used to setup platform defined parameters.

Section 1

```
%%----- Default Values -----
%% SYSTLC: Generic Real-Time Target for PC32 \
```

```
%% TMF: pc32.tmf MAKE: make_rtw EXTMODE: ext_comm_c3x
.....
Section 2
%%----- TLC setup variables-----
%assign MatFileLogging = 1
%assign TargetType = "RT"
....
Section 3
%%----- Include file -----
#include "codegenentry.tlc"
Section 4
%%----- RTW Option Parameters -----
/%
BEGIN_RTW_OPTIONS

rtwoptions(1).prompt      = 'MAT-file variable name modifier';
rtwoptions(1).type        = 'Popup';
rtwoptions(1).default     = 'rt_';
rtwoptions(1).popupstrings = 'rt_|rt|none';
rtwoptions(1).tlcvariable = 'LogVarNameModifier';
rtwoptions(1).tooltip     = [prefix rt_ to variable name,', sprintf('\n'). ...
    'append _rt to variable name.', sprintf('\n'), 'or no modification'];

...
...
...
```

3.5.2 Block Target Files

Block target files (see Fig. 3.15) play an important role in the generation of efficient code and it is imperative to understand their function, as these are the files that have to be modified to accommodate for algorithmic and hardware specific changes.

These files specify how C code is generated for Simulink library blocks, as opposed to using the respective blocks C-MEX S-function directly. This technique of replacing calls into C-MEX S-functions by generated code, is referred to as *S-function in-lining*. The purpose of S-function inlining is to reduce the processing overhead, by removing the use of S-function API. This allows for more efficient and flexible code to be generated¹⁶. S-function inlining applies both to user defined and all library Simulink blocks. In the latter case The Mathworks provides the block target files. The process of S-function in-lining is best demonstrated by an example that follows in the next section.

I. An example of S-function inlining¹⁷

Below is a listing of an S-function for a simple gain block, i.e. $y=u * p$. The first function in this listing is the `mdlInitializeSizes(SimStruct *S)` which is used to setup block specifications, which include number of input and output parameter, number of block states both discrete and continuous, and various other parameters. The second function, `mdlOutputs` implements the intended S-function operation and is called by Simulink when the output of this S-function needs to be evaluated. This S-function is now used by the TLC to produced

¹⁶ Individual blocks no longer require a simstruct of their own.

¹⁷ This example is found in TLC reference guide version 1.2 [MATHWORKS5]

non-inlined code, which is shown below.

```

S-function for gain block
#define S_FUNCTION_NAME foogain
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#define GAIN mxGetPr(ssGetSFcnParam(S,0))[0]
static void mdlInitializeSizes(SimStruct *S)
{
    //Block specification
    ssSetNumContStates (S, 0);
    ssSetNumDiscStates (S, 0);
    ...
    ...
    ...
}
static void
mdlOutputs(SimStruct *S, int_T tid)
{
    real_T 'y = ssGetOutputPortRealSignal(S, 0);
    const InputRealPtrsType u = ssGetInputPortRealSignalPtrs(S, 0);
    y[0] = (*u)[0] * GAIN;
}
static void
mdlInitializeSampleTimes(SimStruct *S){}
static void
mdlTerminate(SimStruct *S) {}
#ifdef MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif

```

From the non in-lined, TLC generated code below, it can be noted that this code calls into the S-function above by using function pointers and the S-function API. This technique is very inefficient, as each block requires its own child simstruc and both the S-function and generated code¹⁸ have to be compiled for the target. There is also a large amount of registration¹⁹ code added to the model.reg file, which further slows down the initialisation of the target system²⁰.

```

Non In-lined code
/*
model.c
*/
real_T untitled_RGND = 0.0; /* real_T ground */
/* Start the model */
void MdlStart(void)
{
    /* (no start code required) */
}
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
    /* Level2 S-Function Block: <Root>/S-Function (foogain) */
    {
        SimStruct *rts = ssGetSFunction(rts, 0);

```

These are calls into the S-function API. SimStruct memory is allocated for the S-function.

¹⁸ Generated code in a non-lined function is merely a wrapper that calls into the base S-function.

¹⁹ This code is used to register parameter, state variables, functions and so forth, in the child simstruc.

²⁰ This code is not reproduced in this thesis as it is of little significance.

```

    sfcnOutputs(rts, tid);
}
}
/* Perform model update */
void MdlUpdate(int_T tid)
{
/* (no update code required) */
}
/* Terminate function */
void MdlTerminate(void)
{
/* Level2 S-Function Block: <Root>/S-Function (foogain) */
{
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
}
}
#include "model.reg"
/* [EOF] model.c */

```

These are calls to the S-function API.

The in-lined generated code that appears below dispenses with the rigours of simulation constraints placed on S-functions and generates code that is called directly. With the in-lined generated code there is no child simstruct and this code is independent of the source S-function file.

```

In-lined generated code
/*
 * model.c
 */
/* Start the model */
void MdlStart(void)
{
/* (no start code required) */
}
/* Compute block outputs */
void MdlOutputs(int_T tid)
/* S-Function block: <Root>/S-Function */
rtB.S_Function = 0.0 * rtP.S_Function_Gain;
}
/* Perform model update */
void MdlUpdate(int_T tid)
{
/* (no update code required) */
}
/* Terminate function */
void MdlTerminate(void)
{
/* (no terminate code required) */
}
#include "model.reg"
/* [EOF] model.c */

```

There are no calls to the S-function API in the inlined version of *model.c*. Also, note that there is no child SimStruct for the S-function.

Inlining eliminates any unnecessary calls to S-function API.

The corresponding block target file used to generate in-line code appears below. It is evident that there is a big saving in both the size and speed of in-lined code, this saving results from a more streamlined function which only implements the required operation: no S-function and simstruct convention are used. It is worthwhile to note that the block target files completely replace their respective C-MEX S-functions and they are, functionally independent of each other. This independence is used extensively in Device Driver Blocks, which is discussed in

the next section.

```
Block Target file
%implements "foogain" "C"
%function Outputs(block, system) Output
/* %<Type> block: %<Name> */
%%
    %assign y = LibBlockOutputSignal(0, "", "", 0)
    %assign u = LibBlockInputSignal(0, "", "", 0)
    %assign p = LibBlockParameter(Gain, "", "", 0)
    %<y> = %<u> * %<p>;
%endfunction
```

II. Device Driver blocks

Device Driver blocks are blocks that allow Simulink models to access peripheral devices, like ADC, DAC, timers and interrupt controllers, on the target hardware. However, these blocks function differently in simulation and real-time mode; therefore block target files are used to allow real-time code to access peripheral devices while the C-MEX S-functions represent “dumb”²¹ Simulink blocks, used in simulation mode. An example of a device driver block target files appears below. The S-function is not listed, as it is merely an “empty” template file.

```
Block Target file for a ADC
%implements "pc32_ad" "C"

%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
/* read in the corrected values from A/D and scale to +/-10 */
{
    %<LibBlockOutputSignal(0, "", "", 0)>=read_adc(BASEBOARD, 0)/(3276.7);
    %<LibBlockOutputSignal(0, "", "", 1)>=read_adc(BASEBOARD, 1)/(3276.7);
    %<LibBlockOutputSignal(0, "", "", 2)>=read_adc(BASEBOARD, 2)/(3276.7);
    %<LibBlockOutputSignal(0, "", "", 3)>=read_adc(BASEBOARD, 3)/(3276.7);
}
%endfunction %% Outputs
```

3.6. The Code Build Cycle

The code build cycle (see Fig. 3.13) is the point at which the RTW uses the target hardware compilations tools to generate an executable module. It accomplishes this task by using a system template make file and a standard make utility. Section 3.6.1 gives an overview of make utilities. Section 3.6.2, discusses the system template make file in more detail, and finally in section 3.6.3 a flow diagram of the code build process is discussed.

3.6.1 Make Utilities

A make utility is used to manage the building of programming projects by automating the compiling and linking stages. A Make utility processes a user make file, which consists of rules on how to build

²¹ These blocks perform no work during simulation mode. See listing of device driver example S-function code.

objects, libraries and executables [GNU1]. In the case of the RTW build process the model.mk file represents the user make file.

Various vendors provide their own make utilities, for example Microsoft use Nmake while GNU²² uses Gmake. The latter tool provides more functionality in general, and is recommended by Mathworks, [MATHWORKS4] it is therefore used with the RADE system. A complete description of a make utility is beyond the scope of this thesis, please refer to the Gmake manual: an electronic copy is found on the accompanying CD.

3.6.2 System Template Make File

The template make file, as the name suggests, is a used to generate the make file that controls the building of the real time executable module. A template make file is made up of five sections, a simplified template make file is listed below with the different sections highlighted. A discussion of these sections follows.

```
Section 1
#----- Comments -----
# Copyright (c) 1994-1998 by The MathWorks, Inc. All Rights Reserved.
#
# File : grt_vc.tmf $Revision: 1.43 $
.....
Section 2
#----- Macros read by make_rtw -----
SYS_TARGET_FILE = grt_c3x.tlc
MAKE            = gmake
HOST            = PC
BUILD           = yes
DOWNLOAD        = yes
Section 3
#----- Tokens expanded by make_rtw -----
MODEL           = |>MODEL_NAME<|
MODEL_MODULES   = |>MODEL_MODULES<|
.....
MATLAB_ROOT     = |>MATLAB_ROOT<|
Section 4
#----- Tool Specifications -----
#----- Include Path -----
#----- C Flags -----
#----- Source Files -----
#----- Rules -----
#----- Dependencies -----
Section 5
#----- Rule for Downloading to Target -----
download :
    ftp_program model.exe
    echo "file sent to target system"
```

I. Comments

This is a standard comments section.

²² GNU is part of Free Software Foundation that produces open source software and development tools. See <http://www.gnu.org>

II. Macros for make_rtw.m function

This section is used to inform the `rtw_make` function: what make utility to use; which real time target is being built; if make is to be executed from the `rtw_make` function; if built module is to be downloaded.

III. Tokens

Tokens are a means of passing variables to the model make file. For example, the following token for the Matlab root directory `|>MATLAB_ROOT<|` will be replaced in the `model.mk` to the location of Matlab, which is `c:\matlabr11` for a default installation.

IV. Build rules

The build rules section is where the most important work gets done. In this section all compiler, linker options, source files and libraries are defined, so to are the rules for compiling and linking.

V. Downloading

This section is only called if the building of the executable module is successful, and the download macro in section two is enabled. The rules in this section call an ftp like routine to download the executable module to the target system.

3.6.3 The Build Flow Diagram

Fig. 3.17 details the flow diagram of the build process:

1. `model.mk`

The `rtw_make` function controls the generation of the `model.mk` file from the system template file.

2. `Gmake`

The `model.mk` file is then used by `Gmake` to build an executable model. The model code and runtime interface are also included in the executable module.

3. Downloading

Once the executable model is built, the `rtw_make` function calls `Gmake` again, which processes the download section of the `model.mk` file. An ftp program is called which sends the executable module to the target system.

4. Target system

Target system receives executable module and waits for a start signal from Simulink.

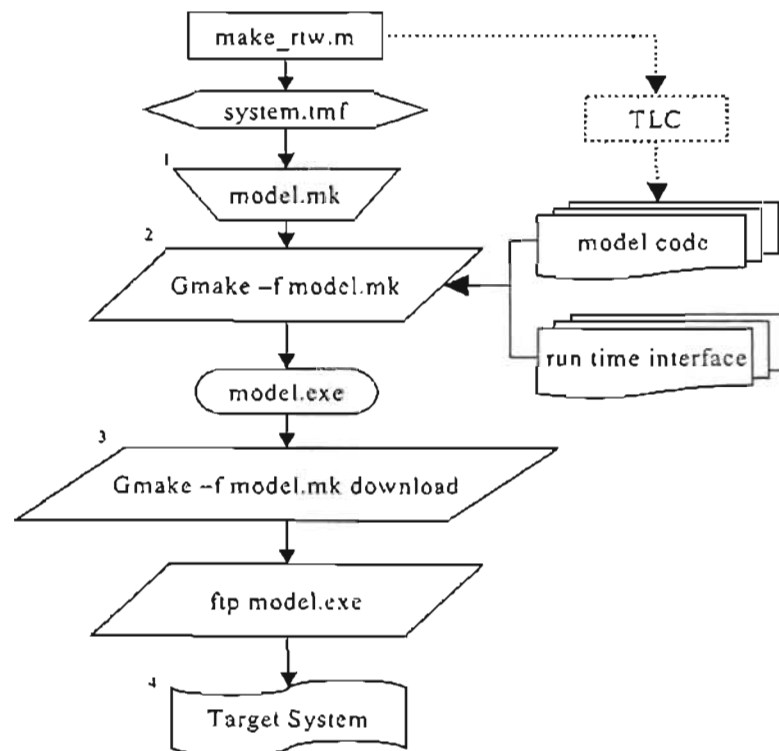


Fig. 3.17: Flow diagram of the build process

3.7. Rapid Prototyping Program Architecture

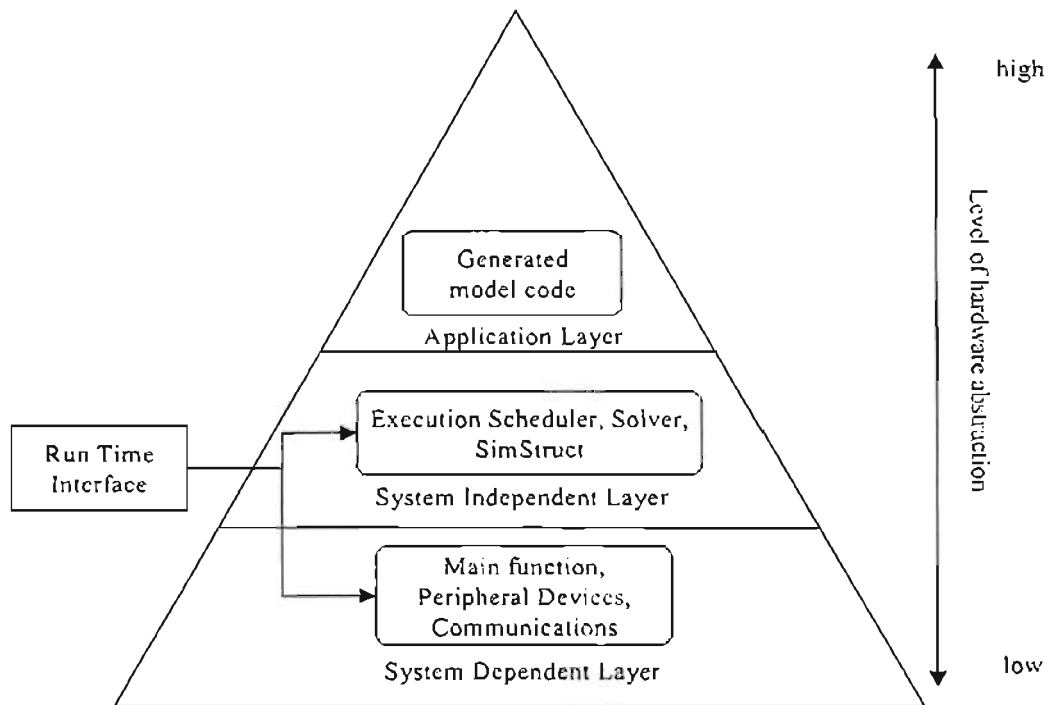


Fig. 3.18: Rapid prototyping program framework

The RTW specifies a program framework for rapid prototyping to allow third party developers to develop real time targets for their specific hardware. This framework specifies the internal working of the target code and how generated model code interacts with the run time interface section. The rapid prototyping framework consists of the three layers shown in Fig. 3.18. These layers are discussed in the next three sections.

3.7.1 System Dependent Layer

This level represents the hardware dependant aspect of the target system, which has to be tailored to the hardware being used. This layer is responsible for the following tasks:

1. `main()` function

This is the entry point for target execution, and is responsible for the initialisation, model execution and program termination.

2. Peripheral devices

The functionality of hardware devices is exported by functions that are callable from the application layer. The initialisation of timers and interrupts are also handled in this section.

3. Communications

The purpose of this section is to incorporate external mode functionality into the target

system. The model code calls into this section to send or receive data. Section 3.8 provides more details.

3.7.2 System Independent Layer

At this level a standard platform is developed, which represents a simulation real-time harness for the application layer above. The code used at this level is provided by The Mathworks and is hardware independent. This level controls the following functionality:

1. Execution scheduler

This section is responsible for the execution of the model code. In real-time code an interrupt source is used to provide accurate timing for the scheduler.

2. Integration solver

The solver is responsible for integration of continuous states in the model code. The RTW provides source files for all the solvers available in Simulink, which are then compiled into the executable target code. It is worthwhile to note that real time code does not use variable step solvers, as the step sizes are normally regulated by interrupts which have strict timing specifications.

3. SimStruct

This section includes the `simstruc.h`, which contains the API for management of `simstruct` data. A model with only in-lined S-function contains only a single root `simstruct`.

3.7.3 Application Layer

Model code resides at this layer. The structure of model code is detailed in Fig. 3.19 and from inspection of this figure and the structure of a S-function, a close correlation can be drawn. The following observations can be made: the model code represents a single S-function²³ with each of the individual S-function sections being merged²⁴ into the respective section in the model code; from a data perspective all child SimStructs are merged into the root SimStruct.

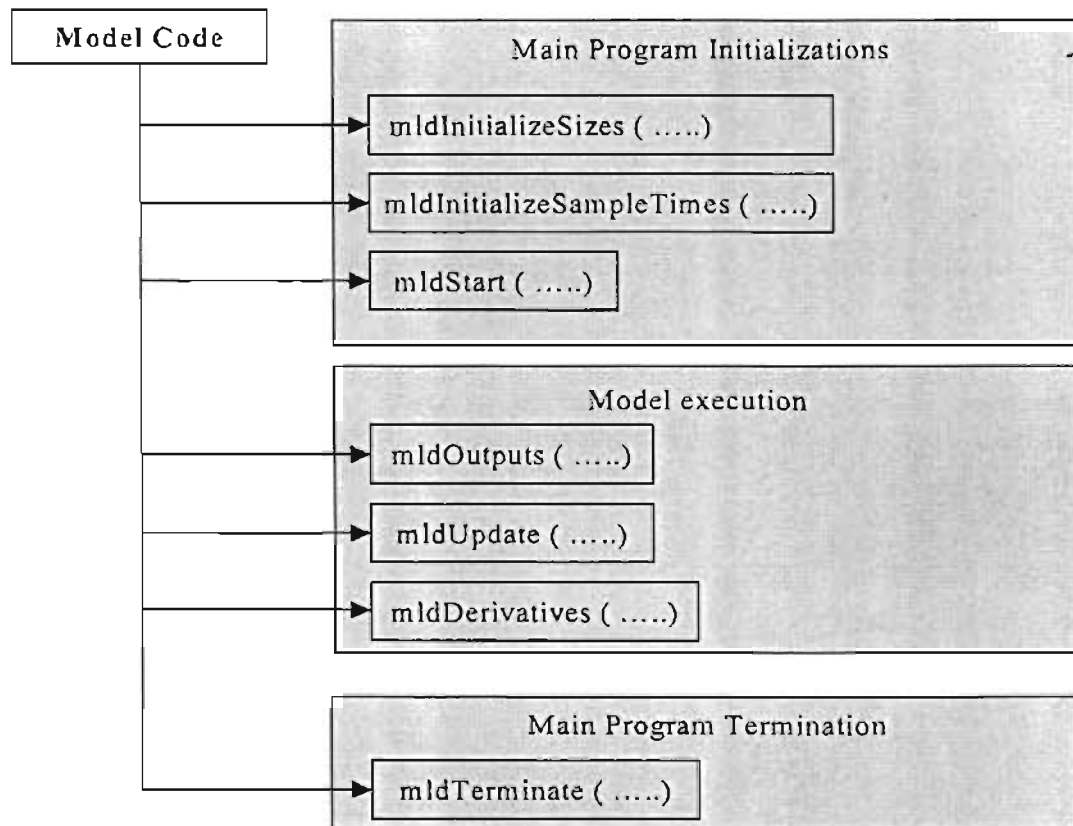


Fig. 3.19: Structure of model code

3.8. The Mathworks TCP/IP External Mode

This section builds on the introduction of RTW External Mode given in chapters one and two. In this section coverage of the RTW External Mode protocol and TCP/IP implementation is provided, as it forms an important part of the RTW and certain aspects are not adequately documented in The Mathworks literature. The details explained in this section are the result of analysis of the code provided with the standard (The Mathworks) external mode implementation. The target system used

²³ This assumes all S-function blocks are in-lined.

²⁴ Merging of code is done by S-function inlining. See section 3.5.2

for the standard external mode implementation is a networked PC running Windows 95.

External mode is a protocol that specifies a communication channel between Simulink and the target. It allows for the control of the target operation from within Simulink. This approach is beneficial to the user as there is no need to move out of the Simulink environment. The external mode protocol allows for:

- Start/ Stop control of the target. After a model has been build and downloaded to the target it can be started or stopped from Simulink.
- Online parameter changes. This feature allows for various Simulink parameters to be changed as the target is executing i.e. parameter are changed in the Simulink environment and downloaded to the target. Tuning PID parameters online is a good example of where this feature is used.
- Online data visualisation and logging. This feature allows data from the target to be displayed on scope blocks within Simulink. In addition data from Scope block can be passed to the Matlab workspace where it can be save and manipulated by other Matlab functions.

Fig. 3.20 details the TCP/IP architecture of external mode where Simulink is the client and the target is the server. A more detailed coverage of the Simulink and target internals is provided in sections 3.8.1 and 3.8.3 respectively.

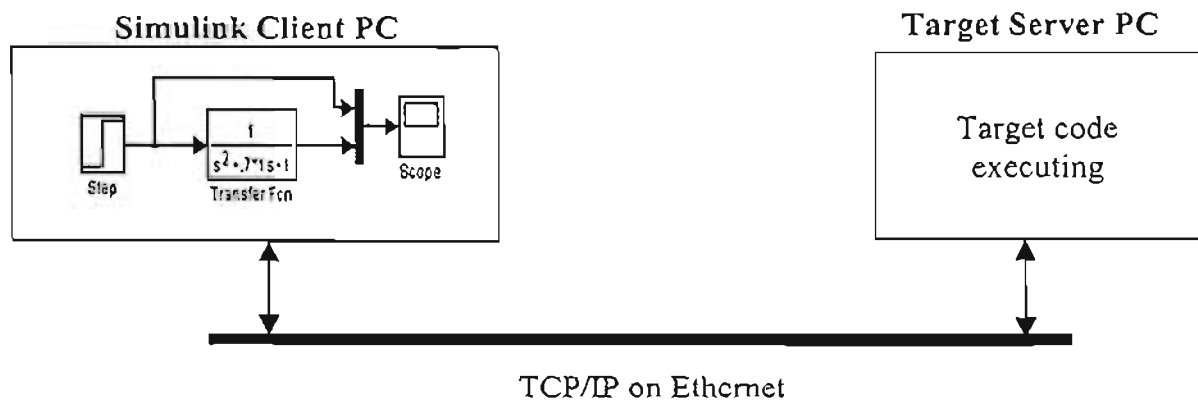


Fig. 3.20: TCP/IP implementation of external mode

3.8.1 Message Frames Between Simulink and Target

During communication between the Simulink client PC, and the target server PC, standard message frames²⁵ are used by the external mode convention. This section describes these frame formats.

During the setup of the external mode channel between Simulink and the target, two stream sockets²⁶ are open: the first is used for messaging and is called the message socket; the second is used for data uploading and is called the upload socket. Fig. 3.21 and Fig. 3.22 represents the message and upload frames respectively. It is worthwhile to note that the message socket is bi-directional while the upload socket is unidirectional. These are not physical limitations but rather external mode conventions.

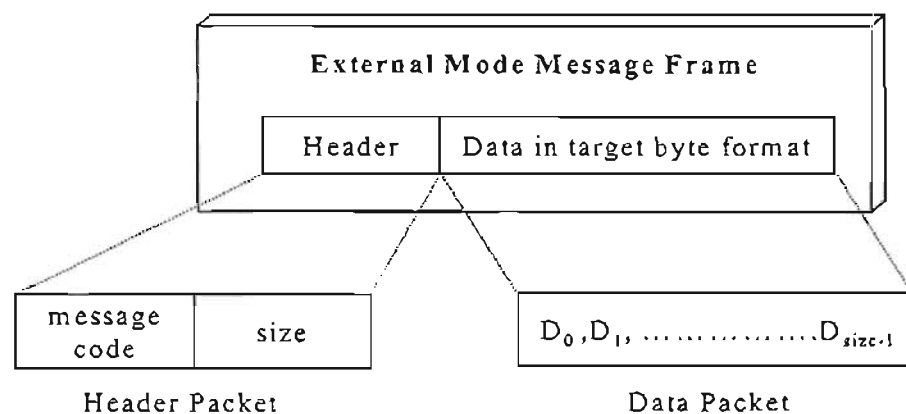


Fig. 3.21: Message frame

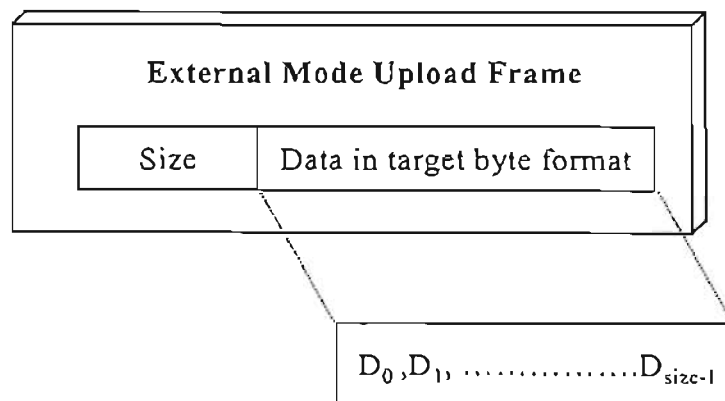


Fig. 3.22: Data upload frame

The message frame (Fig. 3.21) consists of a header portion, which further consists of a message code and the size of the data packet to follow. The message codes used are listed in Table 3-2 with a brief

²⁵ These frames refer to a software abstraction within external mode conventions and must not be confused with frames and packet from the transport layer of a TCP/IP stack.

²⁶ The term Stream Socket refers to a communication end point that guarantees data transmission [WINSOCK1, 2]. See section 4.3.1 for an introduction to the WinSock API.

explanation of their use. Section 3.8.3 also provides more details of the purpose and operation of the various categories of the messages.

Category	Message Code	Purpose
<i>Control actions</i>		
	EXT_CONNECT	The first message sent to the target to setup the external mode communication channel.
	EXT_DISCONNECT	Used to closes the external mode channel but leaves the target running
	EXT_MODEL_START	Used to start target execution.
	EXT_MODEL_STOP	Used to stop target.
<i>Parameter download actions</i>		
	EXT_SETPARAM	Send parameters to target.
<i>Data upload actions</i>		
	EXT_SELECT_SIGNALS	These messages are used to setup the data logging for scope block within a simulation.
	EXT_SELECT_TRIGGER	
	EXT_ARM_TRIGGER	
	EXT_CANCEL_LOGGING	
	EXT_CHECK_UPLOAD_DATA	
<i>Responses signals</i>		
	EXT_CONNECT_RESPONSE	An acknowledge message from target when Simulink tries to connect.
	EXT_SETPARAM_RESPONSE	An acknowledge message from target once parameters are updated.
	EXT_MODEL_SHUTDOWN	Signals target has shutdown.

Table 3-2: List of external mode messages

3.8.2 Simulink Internals

On Simulink's client side of the external mode protocol there are three components, which are shown in Fig. 3.23 and discussed in the following sections.

I. Ext_main.c

This is an intermediate file called from Simulink to send and receive external mode data. It is only responsible for passing function calls from Simulink to the under lying communications layer. The purpose of this module is to allow various communication channels to be integrated into the communication layer. In the TCP/IP implementation ext_main.c calls into ext_comm.c module, which provides the communication layer.

II. Ext_sim.h

This component is the data structure that is used to pass external mode data between Simulink and the communication layer. It manages all aspects of data use, including storage of function pointers for the data conversion functions.

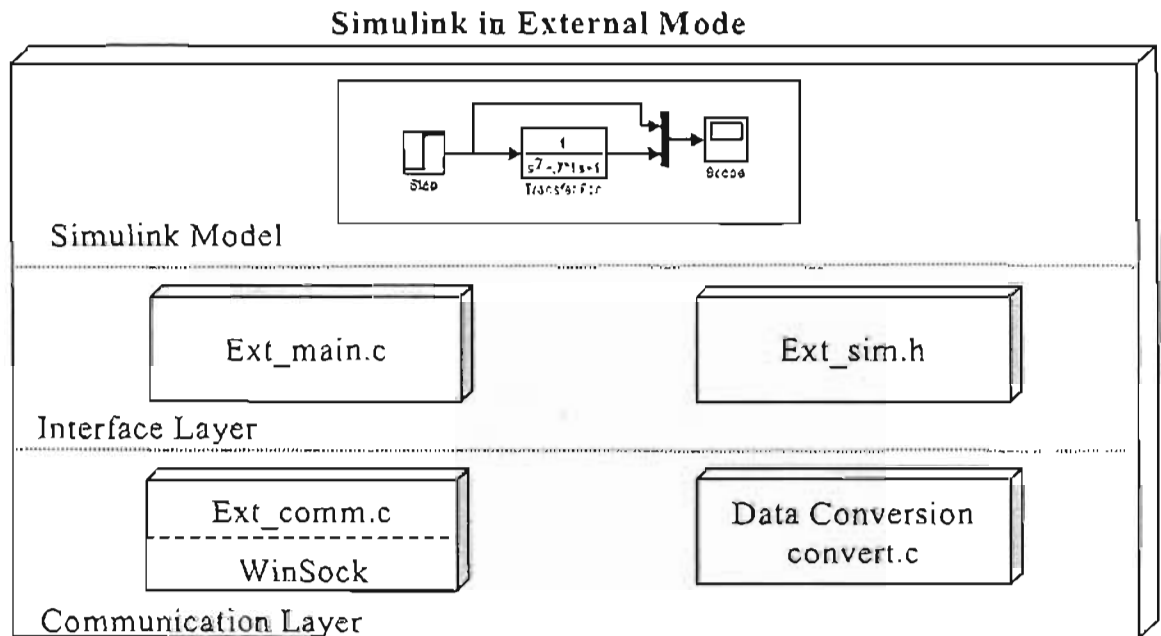


Fig. 3.23: Simulink internals

III. Convert.c

As discussed in section 3.8.1 all communication between Simulink and the target is done using the target data byte format, while this feature is redundant if the target and Simulink have the same byte format it is indispensable when this is not the case. In the case of the RADE system this feature is used as the real-time targets use the TMS320C3X byte format which is not compatible with the PC byte format. See section 4.4

Identity code	Data type
0	Real maximum precision
1	Real 32 bit precision
2	8 bit signed integer
3	8 bit unsigned integer
4	16 signed integer
5	16 unsigned integer
6	32 bit signed integer
7	32 bit unsigned integer
8	Boolean type

Table 3-3: List of Data types and their identity codes

Table 3-3 lists the standard data types, and their identities used by external mode. Each of the data types listed in Table 3-3 have two corresponding data conversion functions, one to

convert from PC to target byte format, and one to do the opposite. These functions are located in the `convert.c` file.

IV. *Ext_comm.c*

This module provides the communication layer by using the Windows Socket (WinSock) API. This module consists of the external mode functions, which send or receive messages.

3.8.3 Target Internals

As described in section 3.7.1 the external mode functionality falls into the system dependant portion of the target, i.e. it is dependent on the type of communication channel used. In The Mathworks TCP/IP implementation the target application uses the *ext_server.c* module. This module uses the WinSock API²⁷ [WINSOCK1, 2] to communicate over the network. Fig. 3.24 highlights the interaction between the *ext_server.c* module and the higher layers of code on the target platform.

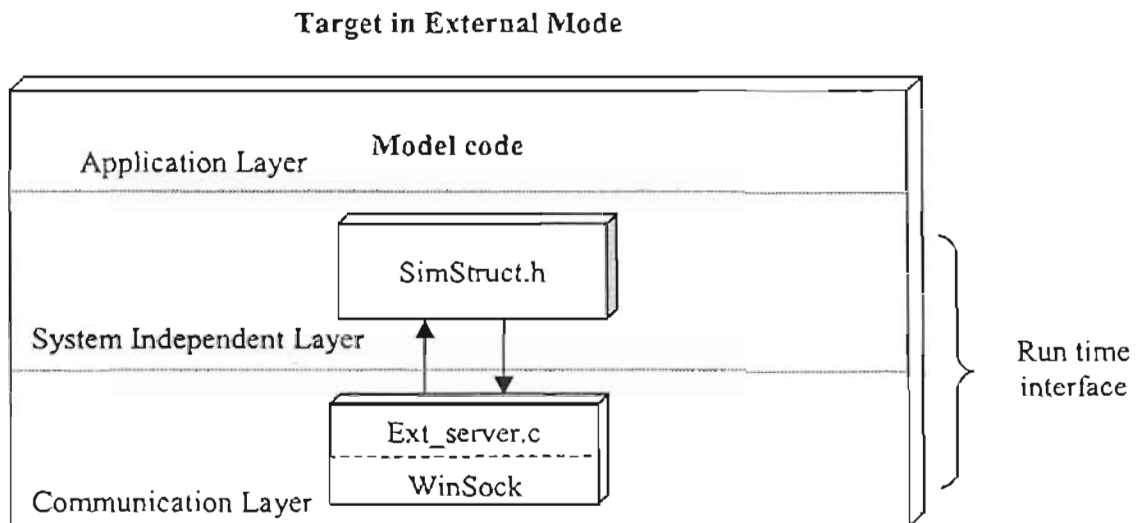


Fig. 3.24: Target internals

Fig. 3.25 elaborates on the message transaction process that occurs between Simulink and the target platform. The *ext_server.c* module is responsible for:

1. Receiving and processing messages received from Simulink.
2. Generating messages and data frames being sent to Simulink.

The four categories of messaging (Table 3-2) that are transacted between Simulink and the target are as follows:

1. Control actions

These messages are received by the *ext_server.c* module and are used to control

²⁷ Chapter four provides a overview of the WinSock API.

execution on the target. They perform action of setting up the communication channel, controlling model code execution status and terminating communications.

2. Parameter changes

When a change parameter message is received it is processed by `ext_server.c` and the appropriate changes are made to parameter data contained in the `simstruct`. The model code then uses the changed parameters when it executes.

3. Data logging

Data logging consists of two parts, the setup and the sending of logged data. The setup messages (Table 3-2) select the signals to be logged, duration and trigger type. Once this is complete signals are sampled at each execution of the model code. This data is then used by the `ext_server.c` module to form data frames, which are sent to Simulink via the upload socket..

4. Response actions

These messages are generated from within the `ext_server.c` module and are used to inform Simulink of the completion of an action on the target. For example the `EXT_SETPARAM_RESPONSE` message is sent to Simulink to acknowledge a parameter change.

For more details on the `ext_server.c` module please refer to Appendix B.

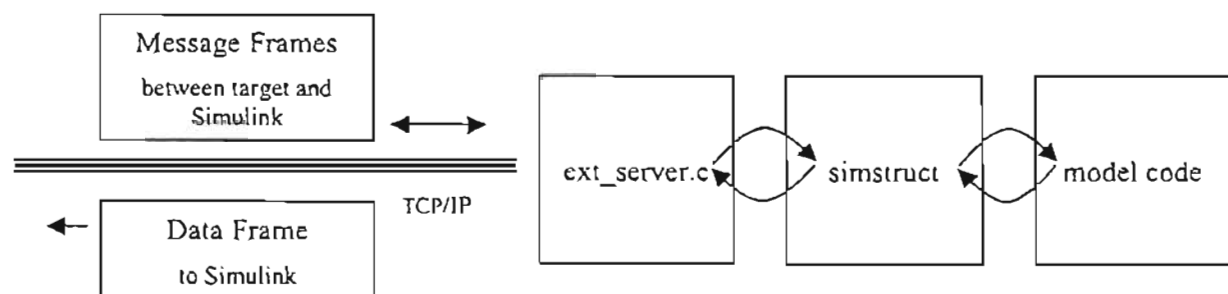


Fig. 3.25: Message Transactions

3.9. Conclusion

The material presented in this chapter provides a foundation for the customisation of the RTW for the RADE system. It highlighted the important components and processes involved with the RTW and the role of Simulink and the target hardware. The issues relating to code generation and program architecture were discussed. It also provided the insight required for the customisation and adaptation of these processes to accommodate various hardware platforms. The Mathworks conventions for the rapid prototyping process will be incorporated into the design of the RADE system and this material is presented in chapter four.

CHAPTER FOUR: DESIGN OF THE RADE FRAMEWORK

4.1. Introduction

Chapter 3 described The Mathworks default TCP/IP external mode implementation that is targeted at platforms that have direct access to a socket API. This chapter draws on this work and presents the RADE framework, which is an adaptation of The Mathworks TCP/IP implementation. The emphasis of this chapter is to provide a functional understanding of the RADE framework independent of the hardware and implementation issues. This approach allows understanding of the framework, which is applicable to any target card¹.

The first part of this chapter reviews the CSDE and The Mathworks systems and then presents the RADE framework within the context of aforementioned systems. Subsequent sections expands on the components of the RADE systems, while also paying attention to aspects drawn from The Mathworks implementation.

4.2. Developing the RADE framework

Before an overview of the RADE framework can be presented it is necessary to first recap on the RTW TCP/IP [MATHWORKS4] and the CSDE [STYLO1] external mode implementations. The reason for concentrating on the external mode protocol is that it represents the area where significant differences exist between both these systems. The RADE framework is then described in perspective of these systems.

4.2.1 Mathworks TCP/IP External Mode Architecture

The Mathworks TCP/IP implementation has already been presented in chapter 3 and this section only highlights the pertinent aspects with regard to the RADE framework. It is worthwhile to note The Mathworks TCP/IP network functionality is built using the WinSock API and therefore references to either are equivalent, i.e. they are used interchangeably in this thesis. Fig. 4.1 shows a functional representation of The Mathworks external mode architecture from, which the following observations can be made:

¹ The RADE implementation issues for the PC32 and ADC64 cards are presented in chapters 5 and 6 respectively.

- The Target system has direct access to a Socket API, which in the case of the Windows platform is Windows Sockets.
- External mode uses the client/server distributed computing model, where the Target is the server and Simulink is the client.

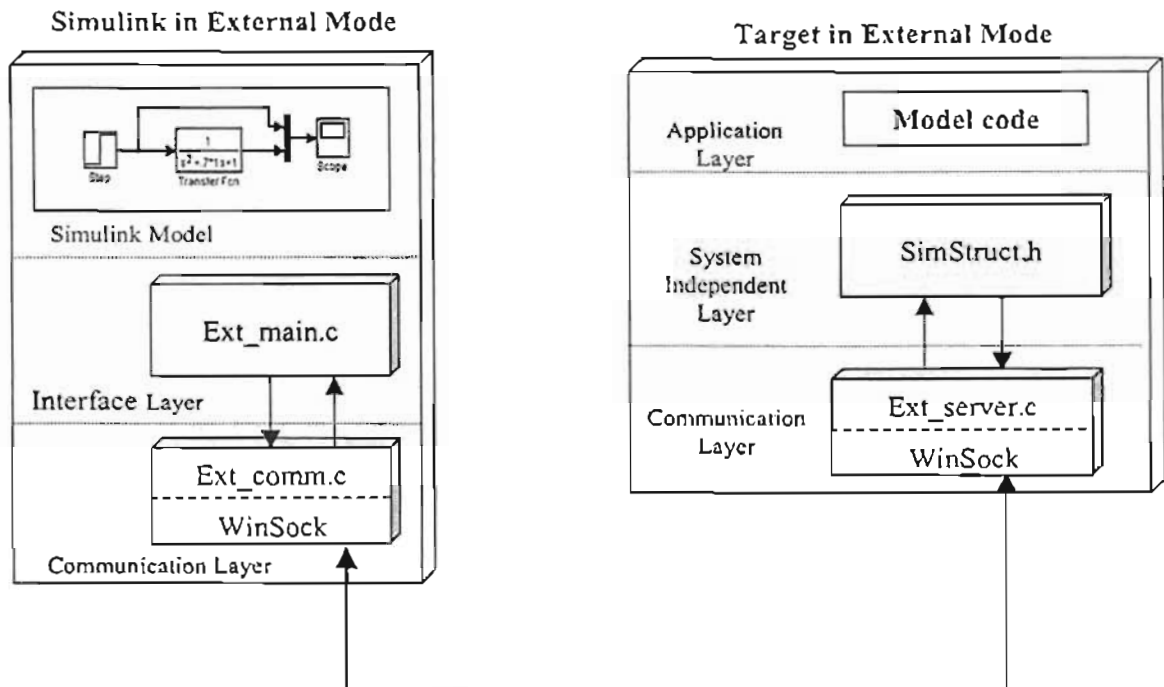


Fig. 4.1: Mathworks TCP/IP external mode

4.2.2 CSDE External Mode Architecture

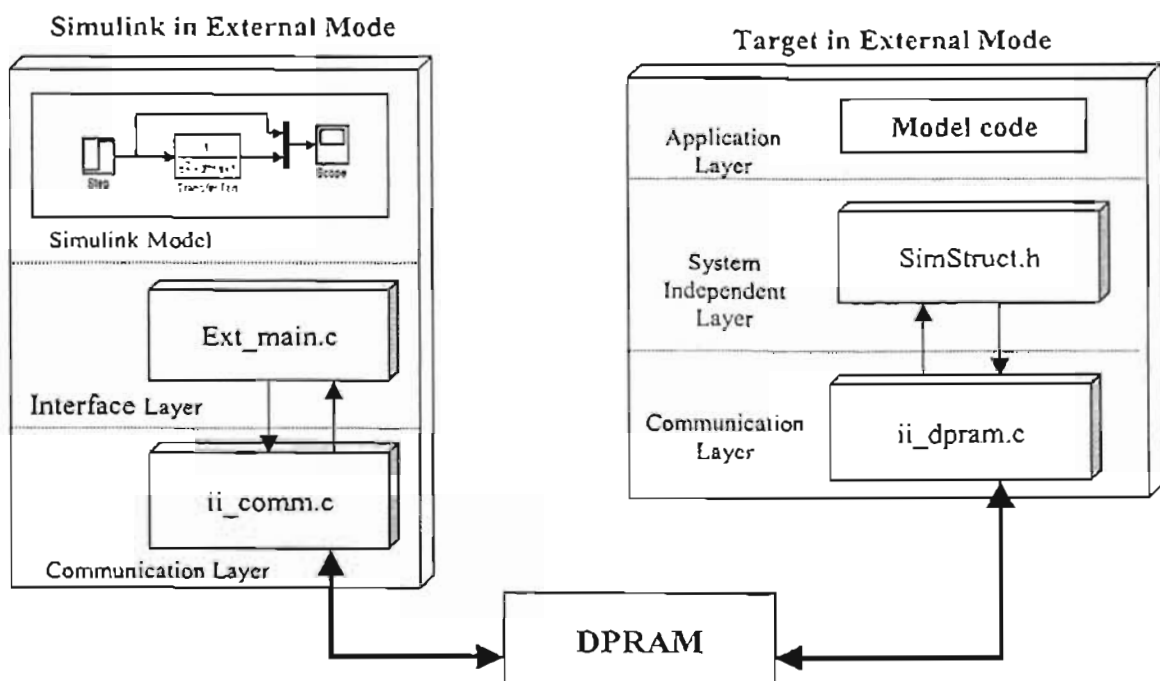


Fig. 4.2: CSDE external mode architecture

The CSDE external mode architecture, developed by Stylo [STYLO1] is shown in Fig. 4.2, and it is evident that the TCP/IP component has been replaced with an in house DPRAM communication protocol. This protocol was developed by Stylo and only works in a standalone mode. On the Simulink side the communication layer (*ii_comm.c*) provides the data conversion functions and the operations of these function do not conform to The Mathworks conventions². The target (*ii_dpram.c*) side is also modified to Stylo's protocol and deviates considerably from The Mathworks protocol. Due to these modifications very little of the source code can be reused with the RADE development and an effective rewrite is needed to incorporate The Mathworks TCP/IP conventions. With this said, it should be reiterated that CSDE was the reference point for the RADE framework, without which development on the RADE framework would have been hampered.

4.2.3 RADE External Mode Architecture

The RADE framework builds on the Mathworks TCP/IP external mode architecture because:

- It allows of for maximum functionality, i.e. full network support, on-line parameter tuning and data logging.
- By adhering to The Mathworks conventions version upgrades of the RTW can be more easily implemented³.
- The Mathworks convention allows for the incorporation of various targets.
- A significant portion of The Mathworks code can be reused.
- The TCP/IP protocol also proved a useful means of Inter Process Communication (IPC)⁴ and allows the RADE system to operate in a standalone mode with no changes.

Due to the complexity of the RADE framework the remaining part of this section is explained using a hierarchical approach i.e. it introduces the various components of the RADE framework and provides references to the relevant sections that provide greater detail. A functional representation of the RADE framework is shown in Fig. 4.3, which comprises of three processes and the server to target communications protocol. These are highlighted below:

1. Simulink: *ext_comms_c3x*

This module provides the communication layer to Simulink and is based on The Mathworks *ext_comm.c* module, besides for some modifications to the data conversion functions.

Further details are provided in section 4.3.

2. Server Application

² Stylo did not implement any of the conversion functions found in the *convert.c* file as describe in chapter 3.

³ The Motion Control Group's experience with CSDE has shown that straying from The Mathworks convention presents significant challenges when faced with version upgrades in the RTW.

⁴ IPC refers to the exchange of data between separate processes executing on a single platform.

This process was developed by the author, to allow target to Simulink communications. The purpose of the server is to allow messages between Simulink and the target to be exchanged seamlessly, i.e. Simulink “believes” it is communicating directly with the target and the server is transparent to it. The reason for using this approach is that the DSP target does not have access to a Socket API. Therefore the server utilises the target PC’s WinSock API to receive or send messages from Simulink. The server application is described in more detail in section 4.5

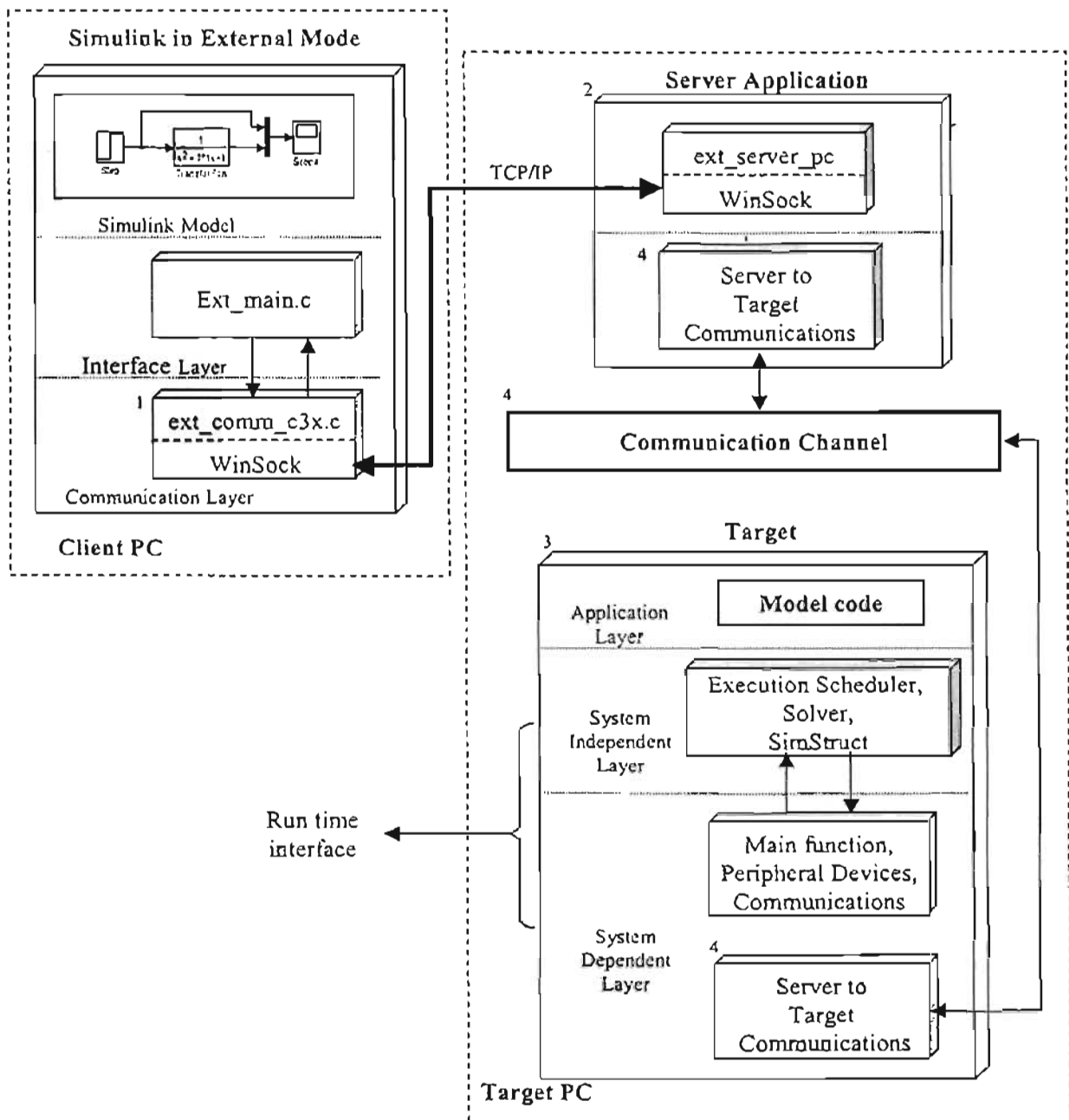


Fig. 4.3: RADE external mode architecture

3. Target Application

This process comprises of two parts, namely the run time interface and the application layer. These processes are adapted from The Mathworks implementation and more details are

provided in section 4.6

4. Server to Target Communication

This protocol was developed by the author to allow for communication between the server and target. It is designed to allow existing code, from The Mathworks, to be reused in the server and target applications while also being independent of the underlying hardware communication channel. Details of this protocol are provided in section 4.7.

4.3. Peripheral Issues

The previous section highlighted the RADE framework in perspective of The Mathworks TCP/IP implementation and subsequent sections in this chapter expand on these topics. However before such a discussion can be presented a review of two peripheral issues needs to be presented. These issues concern the Windows Socket API and the target development toolset by Innovative Integration (II) and their role in the RADE framework. These issues are addressed below.

4.3.1 Windows Sockets

Windows Socket (WinSock) [WINSOCK1, 2] is the API that allows for network programming on the Windows platform and is used by Simulink and the Server applications. The WinSock API is designed to allow application programs to use a standard set of functions, which are conceptually independent of the underlying network protocol, to communicate over a network. The WinSock specification version 1.1 was designed in conjunction with the TCP/IP communication protocol but did not preclude use of other network protocols⁵. This API is the industry defacto standard of network programming and provides an efficient interface to a TCP/IP network [WASHBURN1].

The WinSock API is based on based on the UNIX sockets implementation found in the Berkeley Software Distribution (BSD, release 4.3)[MSDN2]. The WinSock API's basic data object is a socket: this represents a communications endpoint, which is bound to an address and port number. A socket object allows for the bi-directional exchange of data between sockets in the same communication domain i.e. sockets exchanging data must use the same underlying network protocol. There are namely two types of sockets, Stream and Datagram sockets. A Stream socket provides a reliable guaranteed data transmission while a Datagram socket⁶ provides an unguaranteed transmission channel suited for

⁵ Version 2 of the WinSock specification defines a Service Provider Interface, which allows network vendors to provide WinSock support to any network protocol.

⁶ Datagram sockets are used for application that broadcast regular record oriented messages to numerous computers. The synchronisation of system clocks on a network is an example of an application that can use Datagrams.

burst messages. The RADE framework only uses Stream sockets as they provide reliable data transmission.

1. Socket Parameters

This section elaborates on a **Socket Address and Port Number** [MSDN3]. The socket address is associated with the Internet Protocol (IP) address. This address is a 32-bit number that identifies a computer⁷ and can be quoted using the following dot notation X.X.X.X, where X represents an 8-bit number. (Eg. 146.230.192.1)

The Port Number uniquely identifies a socket with a process i.e. multiple sockets can exist simultaneously on a single PC and the port number is used to distinguish which socket belongs to which process. Port numbers for common services like FTP, HTTP and others are reserved. The Port numbers used for the RADE framework are 17725 and 700. Port 17725 is The Mathworks default port used for Simulink to target communication whilst port 700 is used for the automatic downloading of the target application.

4.3.2 Zuma Toolset for Target Development

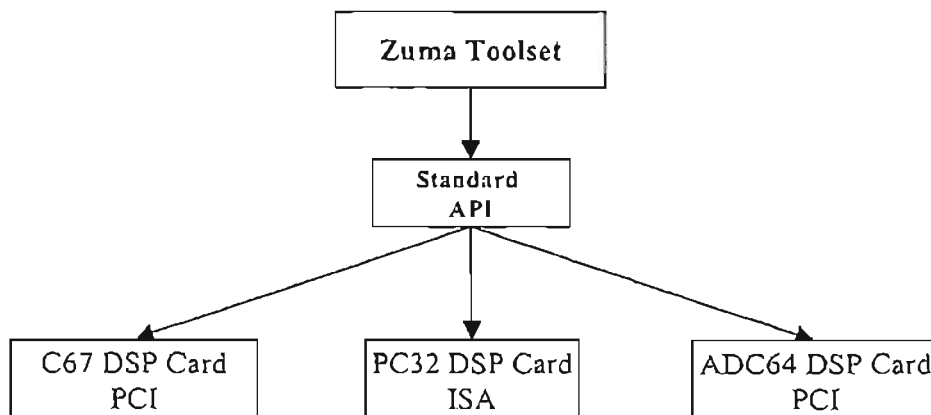


Fig. 4.4: Zuma toolset

An important component of the RADE framework is the DSP target processor. In the work presented in this thesis, the RADE framework is applied to DSP cards⁸ produced by TI, who supply their DSP cards with the Zuma⁹ Toolset. This toolset comprises of an extensive set of functions for both the host¹⁰ and target applications [INNOVATIVE2, 4]. The Zuma toolset, shown in Fig. 4.4 allows for efficient and portable applications to be developed for TI DSP cards and comprises of two components, the host API

⁷ It is also possible to use a computer machine name for example demo_PC.und.ac.za, which is then resolved to an IP address.

⁸ An overview of the DSP cards used is presented in chapters 5 and 6.

⁹ The Zuma Toolset is a propriety development suit provided by TI.

¹⁰ The host is the PC where DSP card is inserted

and target library functions. These are highlighted below.

I. Zuma Host API

The host API provides functions, which allow for easy communications between the host PC and the target DSP. This API also exports core functions that are portable across the Π range of DSP cards. This portability is exploited in the RADE framework because it allows similar code to be used on different Π cards with only the respective DSP card DLL being used. The PC32 card uses the PC32.DLL and the ADC64 uses the ADC64.DLL. The functions exported by the host API allow for:

1. Downloading of target applications.
2. Starting, stopping and resetting the DSP card.
3. Communication to target via Mailboxes. Section 4.3.2-III elaborates on this topic.
4. Block transfer of data between host and target.
5. Data conversions functions.

II. Zuma Target Library Functions

The target library functions allow for the easy and quick development of target applications by allowing the developer to use C style functions for virtually all operations. The target library has a core set of functions that are portable while there are also target specific functions that interface to specialised hardware components. The target library functions afford the developer the following functionality.

1. Functions for Standard I/O.
2. Function for the control of processor features that would normally require assembly language routines. Examples include the setup of timers, ADC's and DAC's.
3. Full host to target communication functions.
4. Registering, enabling and disabling interrupt service routines.

III. Mailbox Operation

When interfacing to a target DSP via the ISA or PCI buses, mailboxes are used to send/receive single data words¹¹ and provide a convenient technique for the control and arbitration of host to target communications. This section outlines the relevant Zuma host and target functions used for mailbox transactions, as mailboxes are used extensively in the server to target communication protocol. (Described in section 4.7) The PC32 and ADC64 cards both have 4 mailboxes and use the same functions definitions, however the underlying implementations differ considerably¹². A few of the host and target functions used to access mailboxes are list

¹¹ The Π implementation for mailboxes uses a signed 32-bit integer.

¹² The PC32 card implements mailboxes in DPRAM. The ADC64 mailboxes are implemented on the S5933 PCI matchmaker IC.

in Table 4-1 and Table 4-2 respectively.

Function Prototype	Description
<code>Int read_mailbox(int target_handle, int box_number);</code>	<code>read_mailbox()</code> returns a 32-bit value from the specified mailbox, when available. The function will wait for data to become available.
<code>void write_mailbox(int target_handle, int box_number, int value);</code>	<code>write_mailbox()</code> writes a 32-bit value to the specified output mailbox and target. Before writing, the function checks to make sure the mailbox is empty (all previous data has been read), and will wait for the target to empty the mailbox if current data is still pending.

Table 4-1: Host mailbox functions

Function Prototype	Description
<code>Int read_mailbox(int box_number);</code>	<code>read_mailbox()</code> waits for data to appear in the specified incoming mailbox, then reads the 32 bit contents of the mailbox.
<code>void write_mailbox(int value, Int box_number);</code>	<code>write_mailbox()</code> waits for the specified outgoing mailbox to become empty, then writes the 32 bit contents of the mailbox with the argument value.

Table 4-2: Target mailbox functions

4.4. Modifications to the Simulink Communication Layer

This section describes the modifications, which were made to the Simulink TCP/IP communication layer (see Fig. 4.3), which will enable Simulink to communicate with a target that does not conform to the PC byte format. The modification involves two parts: the conversion functions and the registration of these functions within the `ext_sim` structure. The following sections expand on these topics.

It is also interesting to note that while The Mathworks built in the functionality of supporting targets with different byte formats, this functionality was never tested. During the development of the RADE framework bugs were identified within Simulink internals. The Mathworks support centre were helpful in rectifying these problems and state that our implementation was in all likelihood the **first in the world** to use the RTW-3 TCP/IP framework, for targets with non-PC compliant byte formats.

4.4.1 Conversion Functions

Conversion functions are needed because in certain instances the target platform and the Simulink PC have different byte formats. Table 3-3 lists the nine data types that can be used in communication between the server and target. Each of these data types requires two-conversion functions, one to

convert data from target to PC format¹³ and the other to convert from PC to target¹⁴ format. Data manipulation only occurs at the Simulink communication layer and all messages sent or received on the Simulink side are done in the target byte format. This approach allows the target to process and generate messages in its native byte format, which saves processing bandwidth.

Table 4-3 lists the TI C3x and corresponding PC byte formats for the relevant data types. The following can be noted:

- All data types on the TI C3x DSP are 32 bits
- Floating point formats differ
- Both platforms use Little Endian¹⁵

With the aid of the information above and Table 4-3 it can be deduced that there are namely two categories of conversion functions: byte ordering and floating-point conversions. These categories are explained in the next two sections.

Data Type	Texas Instruments TMS320C3x			IBM PC		
	Bit Size	Format	Endian L=Little	Bit Size	Format	Endian L=Little
double	32	TI	L	64	IEEE	L
Float	32	TI	L	32	IEEE	L
Char	32	Binary	L	8	Binary	L
unsigned char	32	Binary	L	8	Binary	L
Short	32	2's comp	L	16	2's comp	L
unsigned short	32	Binary	L	16	Binary	L
Int	32	2's comp	L	32	2's comp	L
unsigned int	32	Binary	L	32	Binary	L
BOOL	32	2's comp	L	32	2's comp	L

Table 4-3: Byte formats

1. Byte reordering conversions

Byte reordering is used for most of the data formats list in Table 4-3 excluding the floating-point types. An operational representation of the byte-reordering algorithm is shown in Fig. 4.5, and the following observation can be made, when converting from the PC to target format or vice versa the data is merely repacked into the appropriate memory format. This repacking is achieved by using pointer type casting and is illustrated below with the conversion function used for target to PC *char* conversions.

¹³ These functions are used to interpret messages received from the target.

¹⁴ These functions are used to format messages transmitted to the target.

¹⁵ The Little Endian formats data with the least significant bit at position zero and most significant at position n, where n is the bit size of the data type concerned.

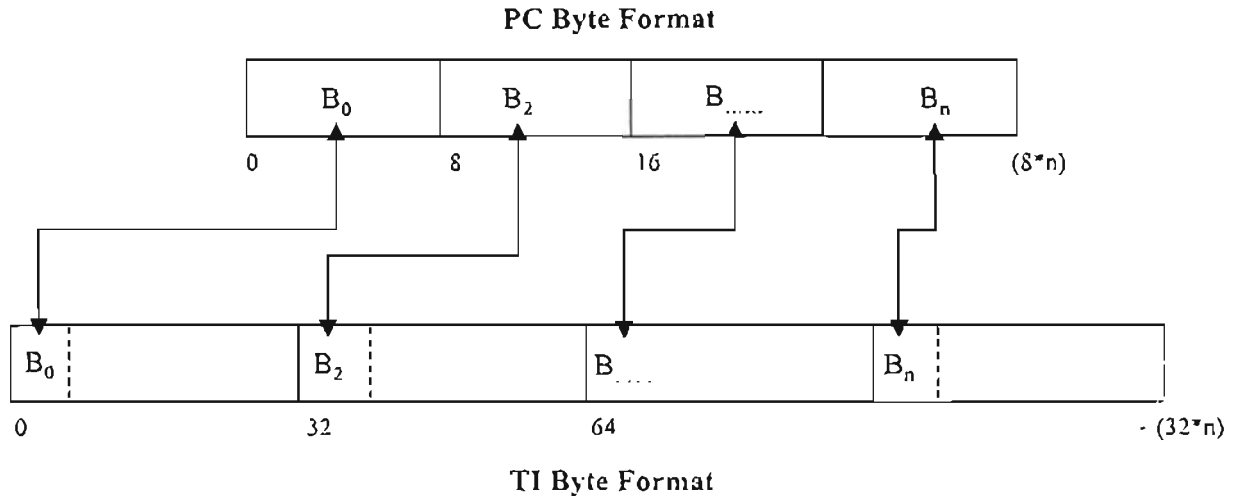


Fig. 4.5: Byte format

```

static void Int8_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    //target has 32 bit format; host 8 bit format

    int32_T *p_src;
    int8_T *p_dst;
    int32_T i;

    p_src=(int32_T*)src; // data from target in 32 bits
    p_dst=(int8_T*)dst; //data typecast to 8 bits for pc
    for(i=0;i<n;i++)
    {
        *p_dst=(int8_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

```

As explained in section 4.4.1, each data type requires two conversion functions and the host to target function for the *char* type is listed below. As the bulk of the function remains the same only the part which differs is listed. It is worthwhile to note that these functions are reciprocals of each other and only repack data: there is no numerical manipulation. The functions for the other data types are based on a similar reordering algorithm and can be found in `ext_convert_c3x.c` file listed in appendix C

```

static void Int8_HostToTarget(.....)
{.....
.....
.....

    p_src=(int8_T*)src; data from PC in 8 bits
    p_dst=(int32_T*)dst; //data typecast to 32 bits for Target
    for(i=0;i<n;i++)
    {
        *p_dst=(int32_T)(*p_src);
        p_src++;
    }
}

```

```

        p_dst++;
    }
}

```

II. Floating point conversions

The two floating-point data types supported within the external mode framework are listed in Table 4-3. The target DSP however only supports a 32-bit TI formatted, floating point data type while the PC supports both a 32-bit and 64-bit IEEE format floating point numbers. The *double* data type is therefore truncated to 32-bits and conversion functions supplied with the Zuma host API are used. These functions are listed in Table 4-4 and convert *float* type data between the IEEE and TI formats. Complete listings of the floating-point conversion function can be found in the `ext_convert_c3x.c` file listed in appendix C.

Function Prototype	Description
<code>int from_ieee(float x);</code>	<code>from_ieee()</code> converts a floating point number to the representation used by the target for 32 bit floating point numbers.
<code>float to_ieee(unsigned int l);</code>	<code>to_ieee()</code> converts a floating point number to the representation used by the host from the representation used by the target for 32 bit floating point numbers.

Table 4-4: Floating-point conversion functions

4.4.2 Function Registration

An important part of The Mathworks external mode specification is the registering of the 18-conversion functions pointer within the `ext_sim` structure. This allows Simulink to directly access these functions for the processing and generating of messages. The code segment that is responsible for this registration is shown below, with the complete function found in the `ext_convert_c3x.c` file listed in appendix C. The registration of each data type entails the calling of a macro to add the required function pointer into the `ext_sim` structure.

```

void ProcessConnectResponse1(ExternalSim *ES, MsgHeader *msgHdr)
{
    .....
    .....
    .....
    .....

    /*
     * Set up fcn ptrs for data conversion - Simulink data types.
     */
    esSetDoubleTargetToHostFcn(ES, Double_TargetToHost);
    esSetDoubleHostToTargetFcn(ES, Double_HostToTarget);

    esSetSingleTargetToHostFcn(ES, Single_TargetToHost); /* assume 32 bit */
    esSetSingleHostToTargetFcn(ES, Single_HostToTarget); /* assume 32 bit */

    esSetInt8TargetToHostFcn(ES, Int8_TargetToHost);
    esSetInt8HostToTargetFcn(ES, Int8_HostToTarget);

    esSetUInt8TargetToHostFcn(ES, UInt8_TargetToHost);

```

```
esSetUInt8HostToTargetFcn(ES, UInt8_HostToTarget);

esSetInt16TargetToHostFcn(ES, Int16_TargetToHost);
esSetInt16HostToTargetFcn(ES, Int16_HostToTarget);

esSetUInt16TargetToHostFcn(ES, UInt16_TargetToHost);
esSetUInt16HostToTargetFcn(ES, UInt16_HostToTarget);

esSetInt32TargetToHostFcn(ES, Int32_TargetToHost);
esSetInt32HostToTargetFcn(ES, Int32_HostToTarget);

esSetUInt32TargetToHostFcn(ES, UInt32_TargetToHost);
esSetUInt32HostToTargetFcn(ES, UInt32_HostToTarget);

esSetBoolTargetToHostFcn(ES, Bool_TargetToHost);
esSetBoolHostToTargetFcn(ES, Bool_HostToTarget);

EXIT_POINT:
    return;
} /* end ProcessConnectResponse1 */
```

4.5. Server Application

This section describes the server application¹⁶ at a system level and is intended to provide the reader with a general understanding of this application within the context of the RADE framework. The Server application is primarily an “invisible” helper that links Simulink to the target DSP as discussed in section 4.2.3. To the best of the author’s knowledge and according to the information available in the open literature the RADE framework is the world’s first RTW implementation to use a hybrid target approach, that incorporates the target PC’s WinSock API (server application), and a DSP target within The Mathworks TCP/IP framework¹⁷. It allows a relatively inexpensive DSP card to be incorporated within the RTW framework with full network support by “piggy backing” on the target PC’s WinSock API.

¹⁶ There are two versions of the server applications: one for the PC32 target and the other for the ADC64 target. Both the Visual C++ project files are contained on the CD attached. Appendix F lists their locations

¹⁷ See section 4.4 for substantiation.

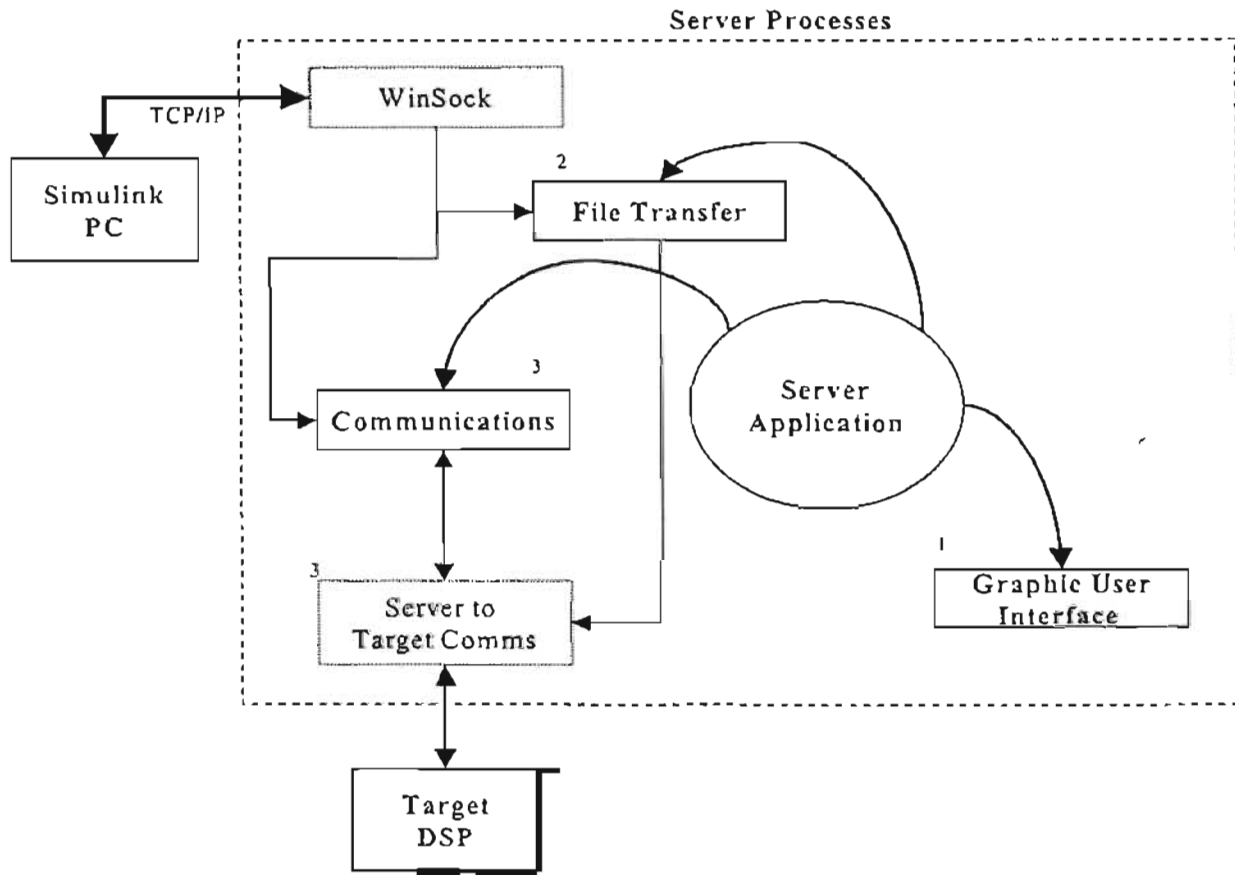


Fig. 4.6: Server application

A simplified functional diagram of the server application is shown in Fig. 4.6, which comprises of three processes that are highlighted below:

1. GUI

The GUI's primary function is to echo text messages from the target, as this is the only reliable means to debug the target¹⁸. It is also used to display status messages of both the target and server and is invaluable during development. The GUI of the server is however of little significance to a user of the RADE system since the messages echoed are of no use to a user. A more detailed description is presented in section 4.5.1.

2. File Transfer Utility

The file transfer utility is used for the automated downloading of target application from the Simulink PC to the target PC. It uses the WinSock API to perform the file transfer and comprises of two applications, `auto_download` and `server`. `Auto_download` is an application developed by the author, which is executed on the Simulink PC when the target application is ready for downloading. The server application in turn receives and stores the

¹⁸ A JTAG debugger is more effective but was only available towards the end of the development, in the latter part of 2000.

target application on the target PC. A more detailed discussion is presented in section 4.5.2.

3. Communications

The communication process of the server application is made up of the WinSock part and server to target communication protocol. The WinSock part is used to send and receive messages between Simulink and the target PCs while the server to target communication is used to transact these messages between the target and server applications. The communication aspects of the server application are a critical part of the RADE framework and therefore warrant the detailed discussion that is presented in section 4.7.

4.5.1 Graphic User Interface

This section describes the graphical attributes of the server application and its purposes. A goal of the RADE framework is to allow for easy upgrading. Therefore a simple technique to view execution status of the target and server application during development is needed: the server GUI provides this. The GUI is shown in Fig. 4.7 and comprises of two display windows. The Server IO WIN32, which displays text messages from the PC and the Target IO c3x, which displays target text messages.

The primary need for the server GUI is to provide feedback from both the server and target applications. This feedback is then used to track bugs or the execution status on either application. Messages from the target platform use the ANSI C *printf* statement while the PC platform uses the MFC *Cstring* object, which allows printf like functionality.

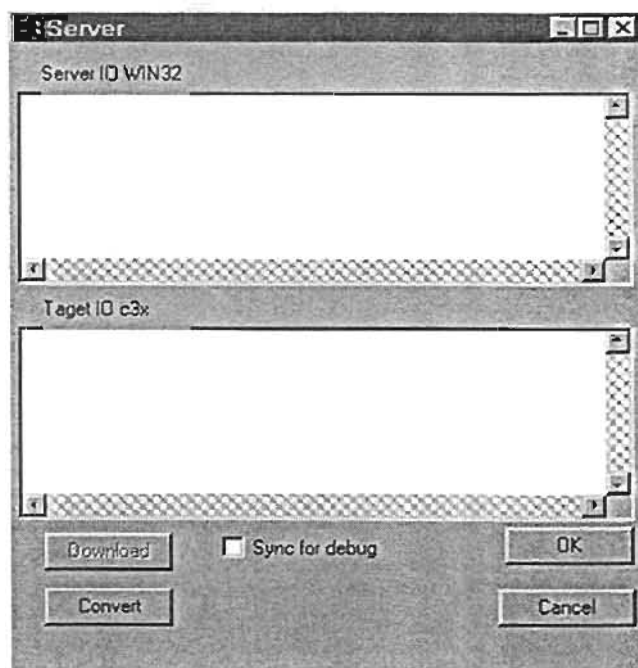


Fig. 4.7: Server GUI

An added feature of the server application is the logging of these text messages to files, which is

useful for message analysis if the server application¹⁹ crashes. The logging feature is intended for use during development only. The server application is prone to crashes during development because it is interdependent on the target system, i.e. a bug on the target system can crash both the target and the server applications. With this said it must be noted that the current server application has been thoroughly tested by the author and found to be extremely stable i.e. it does not crash after repeated Simulink model RTW rebuilds and continuous execution of the target code.

4.5.2 File Transfer Process

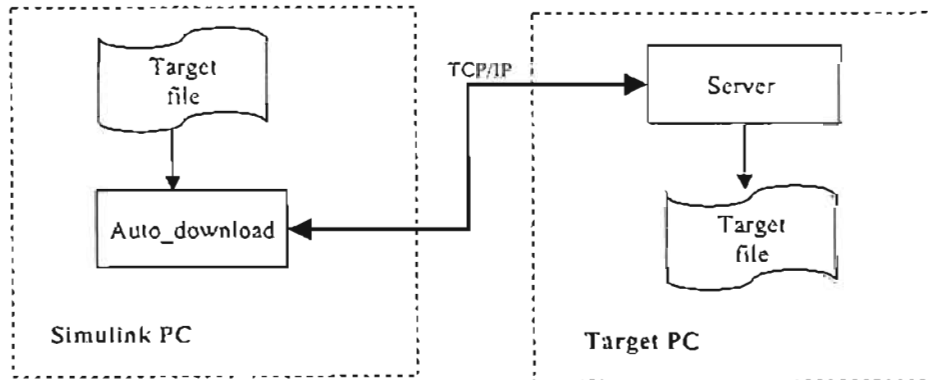


Fig. 4.8: Functional representation of the file transfer process

This section describes how the target application is transferred to the target PC. The Mathworks TCP/IP implementation does not provide a utility to automatically download the target application from the Simulink PC to the target PC²⁰. Therefore the author developed an FTP²¹ like process. The combination of **Auto_download** utility, on the Simulink PC, and the **Server**²² application on the target PC are used. Fig. 4.8 shows the functional operation of these two applications, and it should be noted that the file transfer process on the target PC is a component of the server application as shown in Fig. 4.6, while the **auto_download** utility is an independent MS-DOS console application. The command line parameters for the **auto_download** application are shown below. This application is automatically executed from the RTW build process.

```
Auto_download -f[FILE_NAME] -s[SERVER_NAME] -p[SERVER_PORT]
```

¹⁹ PC and target messages are logged to the `server_data.txt` and `target_data.txt` files respectively. These files are overwritten on each launch of the server application.

²⁰ It does however allow for a third party, FTP utility to be called for the code build cycle. See Chapter 3.

²¹ The protocol used by the routines does not conform to FTP, but from an operational perspective does allow for file transfers between computers.

²² These processes are adapted from **CHATTER** and **CHATSRVR** sample programs by Microsoft [MSDN1]

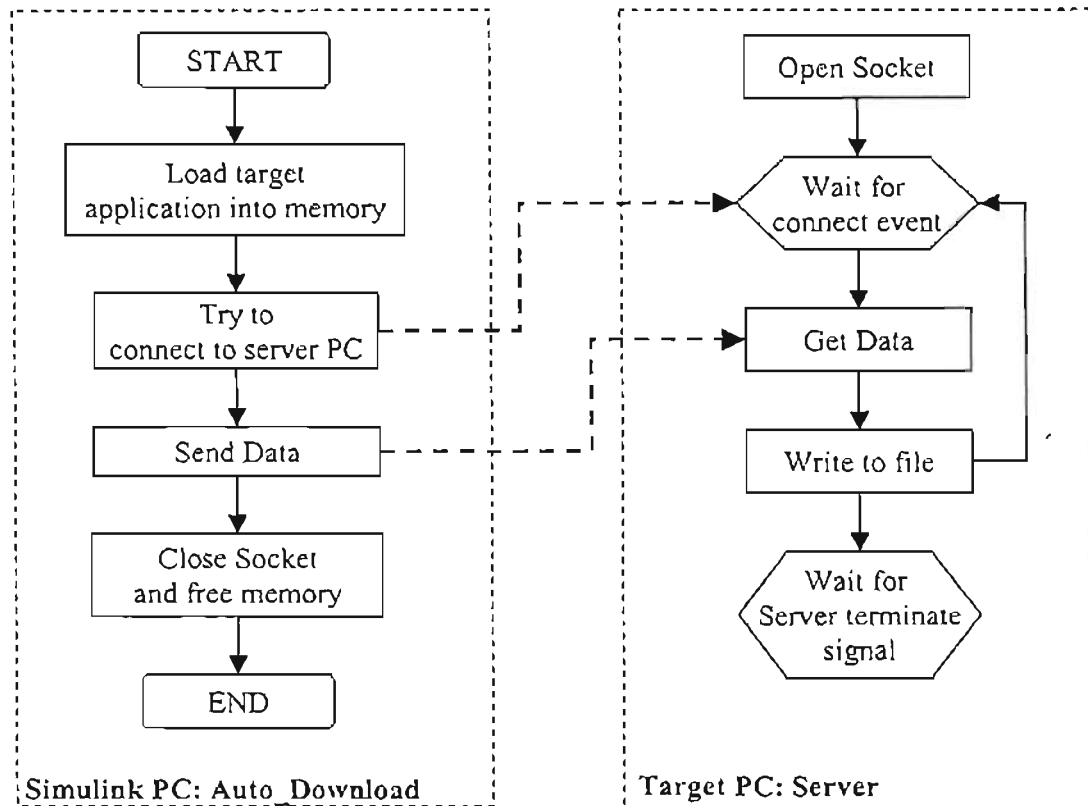


Fig. 4.9: Flow diagram of file transfer process

Fig. 4.9 shows a simplified flow diagram of the file transfer process. The server application opens a socket and then waits for a connection signal from the auto_download application. Once the auto_download application executes it, loads the target file and then proceeds to transmit it to the server application. Once the file is received by the server application it is stored on the target PC. The auto_download application then terminates while the server application returns to the “wait for connect signal” state.

4.6. Target Run Time Interface

This section describes the RTI, which forms part of the target application. The RTI is composed of two parts, the system independent and the system dependent layers, which must be designed to provide a harness on which model code can execute. The system independent layer is hardware independent ANSI C code provided by Mathworks that is merely compiled by the TI C compiler. It therefore does not require discussion here as it has already been discussed in chapter 3. The system dependent layer however, provides the core hub for target execution and is adapted from The Mathworks default implementation. A description is presented below.

4.6.1 System Dependent Layer

The system dependent layer of the RTI is composed of:

1. The entry function *main()*
2. Communication aspects
3. Peripheral device drivers

The entry function is the core hub, around which the rest of the target application executes. The Mathworks specify the general structure, which is shown in Fig. 4.10. The communication aspects are presented in section 4.7 and the peripheral device drives, which hardware specific are described in chapter 5 for the PC32 card and chapter 6 for the ADC64 card.

The simplified flow diagram of the *main()* function is shown in Fig. 4.10, with the individual steps described as follows:

1. This step is used to synchronise the target and host systems at target start-up. This ensures that data sent to the target is correctly processed and not lost due to the target being in a undefined state.
2. The model code initialisation routines are executed and the root simstruct is declared.
3. The target then halts until a start signal is received from Simulink.
4. Once the start signal is received the target registers and enables the ISR routine. This routine is responsible for the real-time execution of the model code. See step 8
5. This is a foreground process that can be pre-empted by the ISR. It is responsible for processing messages being sent or received from the server application.
6. This routine sends logged data to Simulink via the server application. The data being sent is logged during the ISR.
7. This is a standard termination routine that shutdowns the target and resets peripheral devices. Once the target is shutdown the server application holds it in reset until the target code has to be executed again.
8. The ISR routine is model around Simulink's simulation loop. See chapter 3.
 - 8.a This routine maintains the absolute time, which is used by certain Simulink blocks.
 - 8.b These routines are all generated by the RTW and represent the Simulink model being executed.
 - 8.c When data logging is enabled this routine logs each time step of data to a buffer. When this buffer is full it signals the foreground upload process. See step 6

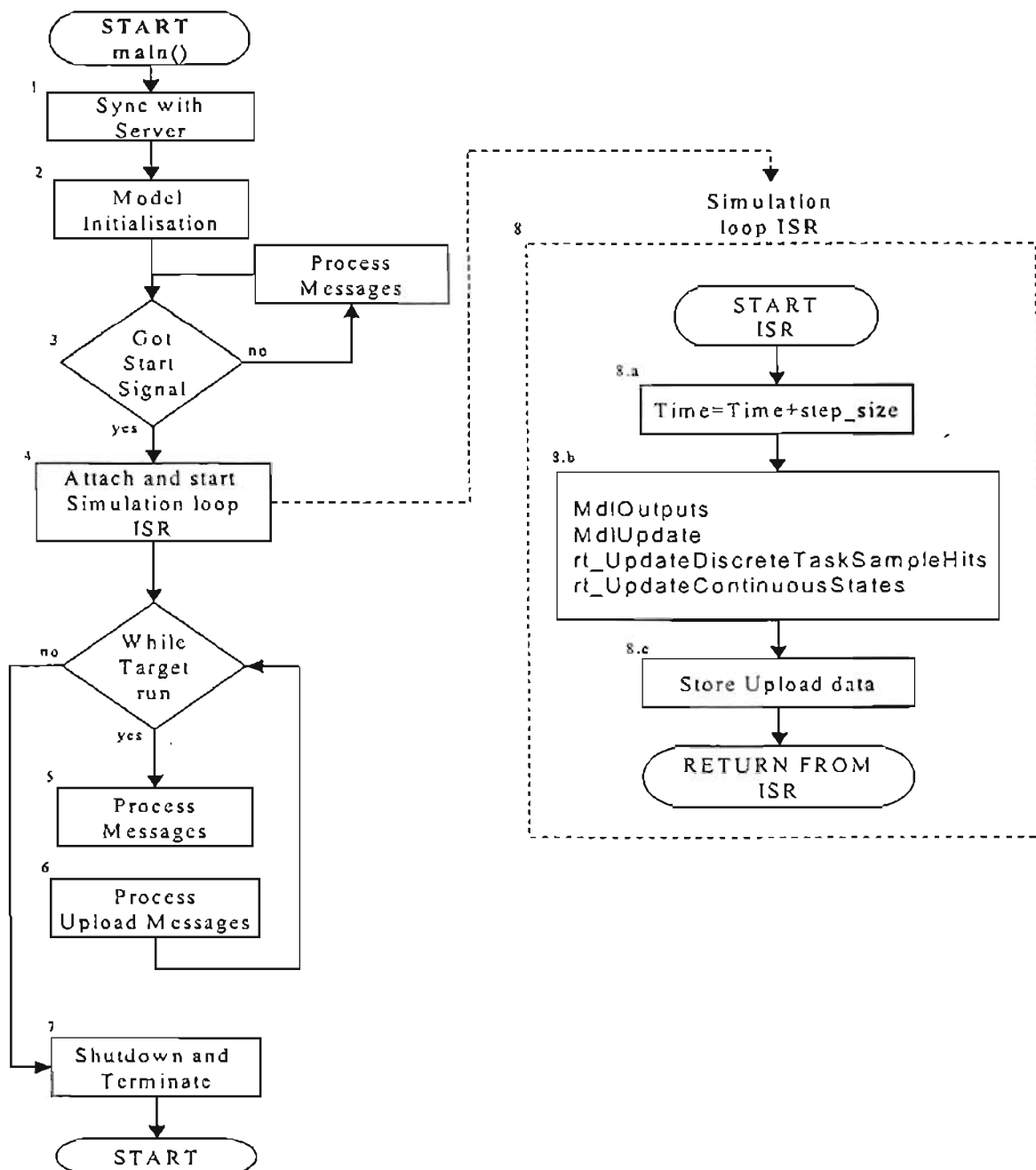


Fig. 4.10: Flow diagram of RTI entry function

4.7. RADE Communications

This section discusses the communication details between Simulink and the target DSP, with the server application being an intermediary. The purpose of this section is to provide a conceptual overview of the communication operations within the RADE framework. The two important aspects, which will be addressed, is the adaptation of The Mathworks TCP/IP code for reuse within the RADE framework and the operation of Server to Target Protocol (STP).

The operation of target PC communication within the RADE framework is shown in Fig. 4.11. The

Mathworks target side TCP/IP external mode implementation²³ which is intended to be executed on one platform is dissected into two parts:

- The WinSock part which runs on the PC
- The external mode message processing and generation part which runs on the target DSP.

This code dissection while not trivial from an implementation perspective allows the bulk of The Mathworks TCP/IP code to be reused and this approach allows for maximum functionality. The PC component of The Mathworks TCP/IP implementation is found in the `ext_srv_pc.cpp` file and the target component is found in the `ext_srv_c3x.c`. These files run on separate platforms and communicate via the STP.

STP is designed to link the PC WinSock component with the target message-processing component through a protocol that abstracts the underlying communication channel to a software layer. The STP allows for flexible communication between the server and target that can be easily ported to various target platforms. The next section elaborates on the STP.

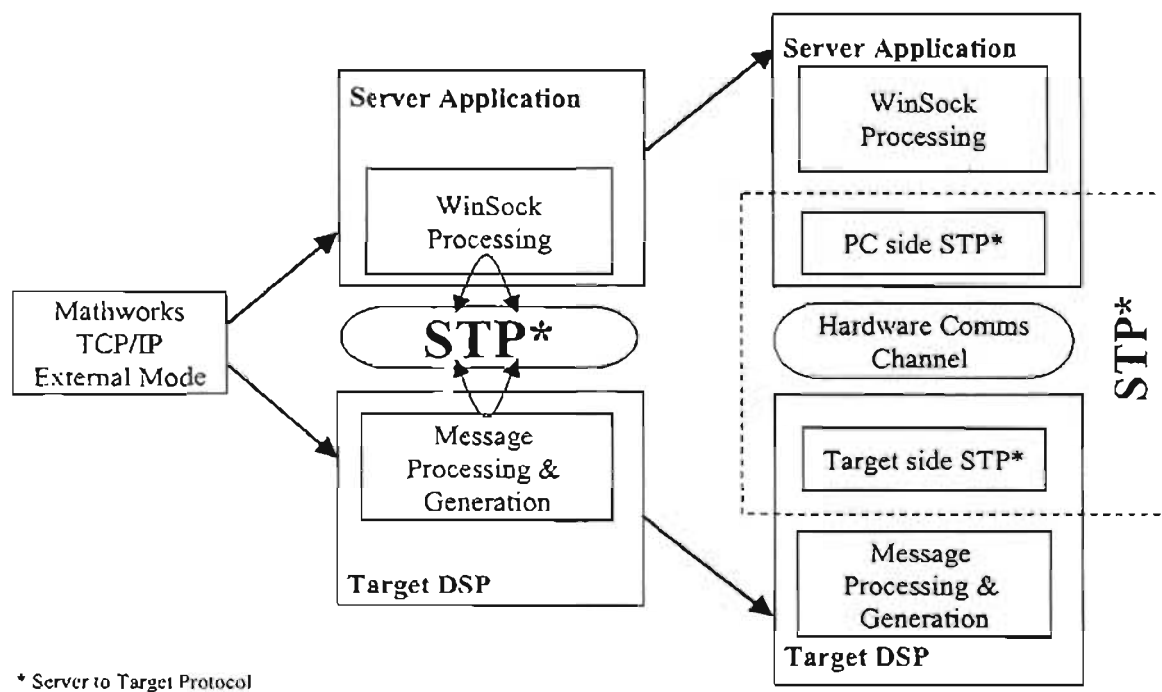


Fig. 4.11: Server to target communications

4.7.1 Server to Target Protocol

The STP is primarily designed to allow for the transfer of external mode messages and upload data between the target and server. It is accomplished by using the communication data structure shown in Fig. 4.12. This structure consists of two **Message Ports** and two **Upload Ports**, which are overlaid

²³ The Mathworks TCP/IP implementation has been described in chapter 3.

onto a shared memory resource. The message ports are used for external mode messages transactions. The reason for having two is to allow for the full-duplex transfer of data i.e. the TO-TARGET port only receives data from the server, while the FROM-TARGET port only sends data to the server. The communication structure also consists of two upload ports, which are used for the uploading of logged data. Two upload ports are used because a “ping-pong”²⁴ technique is used for the uploading of data.

Both the server and target application regularly²⁵ poll the message ports and upload ports to check for new data. A polling technique was employed, as opposed to the interrupt base approach used in the CSDE system because this method does not compromise the real-time operation of the target i.e. the target will not be pre-empted by lower priority messaging overhead. The messaging and data logging function therefore have minimal effect on the targets real-time performance. A further advantage of this method is that it does not use any interrupt resources on the target which is useful in cases where targets do not support PC to target interrupts or have no spare interrupts²⁶.

The message and upload ports are import components of the STP and warrant a detailed description, which is presented in sections 4.7.2 and 4.7.3 respectively.

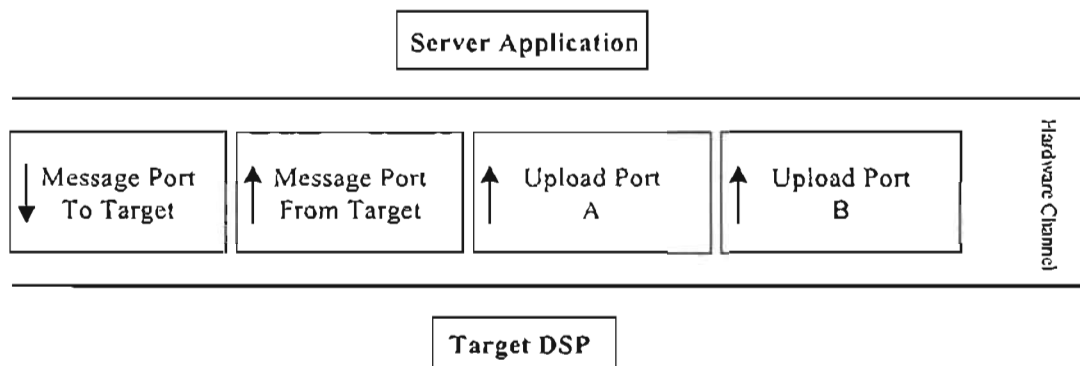


Fig. 4.12: Overview of Communication Channel

4.7.2 Message Port

This section describes the functional operation and purpose of the mail port data structure. Mail ports have two purposes, which are namely to facilitate transfer of external mode messages between the

²⁴ A “ping-pong” technique consists of the target writing data to port A and then once finished the server retrieves this data. Simultaneously while the server is reading port A the target writes port B and then again hands off to the server. This process continues until all the data is transferred. The advantage of this technique is that it allows the server and target simultaneous access to different parts of a share memory resource, thereby allowing for maximum data through put. This method is recommended by II to achieve maximum data through put.

²⁵ The Server application uses a 10ms timer and the target runs the polling routine in the foreground.

²⁶ This applies to the ADC64 card as all its interrupt resources are used by on card peripherals, it does however support PC to target interrupts by multiplexing interrupts, but is clumsy to implement.

server and target and to allow local communication between the target and server.

The detailed structure of the mail ports used is shown in Fig. 4.13. The server application uses the TO-TARGET mail port to send data to the target and the target uses the FROM-TARGET mail port to send data to the server. This allows both applications simultaneous access to the shared memory resource. Each mail port is associated with a mailbox and a status flag. The mailboxes are used to signal the respective applications when data in the mail ports are ready to be received, while the status flags arbitrate port access. Acknowledge signals are used to inform the respective applications when data has been retrieved and to clear the port's busy status for new messages.

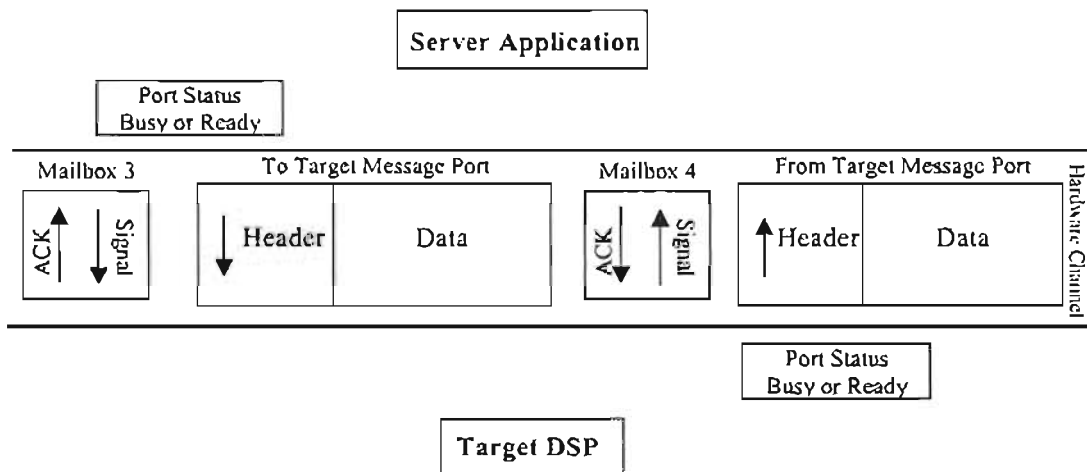


Fig. 4.13: Graphical representation of the Message Ports

A crucial feature of STP is to allow the incoming or outgoing external mode messages to be broken into packets. Message packeting is needed, as the shared memory between the server and target is a limited resource, and the sizes of these messages can be larger than the space allotted. Fig. 4.14 shows the packeting of an external mode message being sent to the target. The message is first received by the WinSock component of The Mathworks TCP/IP code; it is then broken into packets and sequentially sent to the target via the TO-TARGET mail port. On the target side the packets are reassembled and the complete message is passed to the message-processing component of The Mathworks TCP/IP code.

The mail port structure declaration is shown below, and consists of:

```
typedef struct {
    msg_id msg_type;
    int32_T current_size, full_size;
    int32_T spare[2]; //spare data for debugging
    MsgHeader msg_hdr; //external mode header
    int32_T buff[MP_BUF_SIZE];
} Msg_Port, *p_Msg_Port;
```

1. message type (msg_type)

This data type is used to identify to the application reading the mail port what data it is being received and what action to take. This information is normally used to sequence data packets

being received.

2. `current_size`, `full_size`

These variables contain size information about the data buffer section of the mail port

3. `spare[2]`

Two integer variables used to send additional information with the mail port header. It was primarily used during debugging.

4. message header (`msg_hdr`)

This is the external mode header component.

5. buffer (`buf`)

This is a buffer that stores the data being transferred. Its size is limited by the amount of shared memory allotted to the mail port and varies for different targets²⁷.

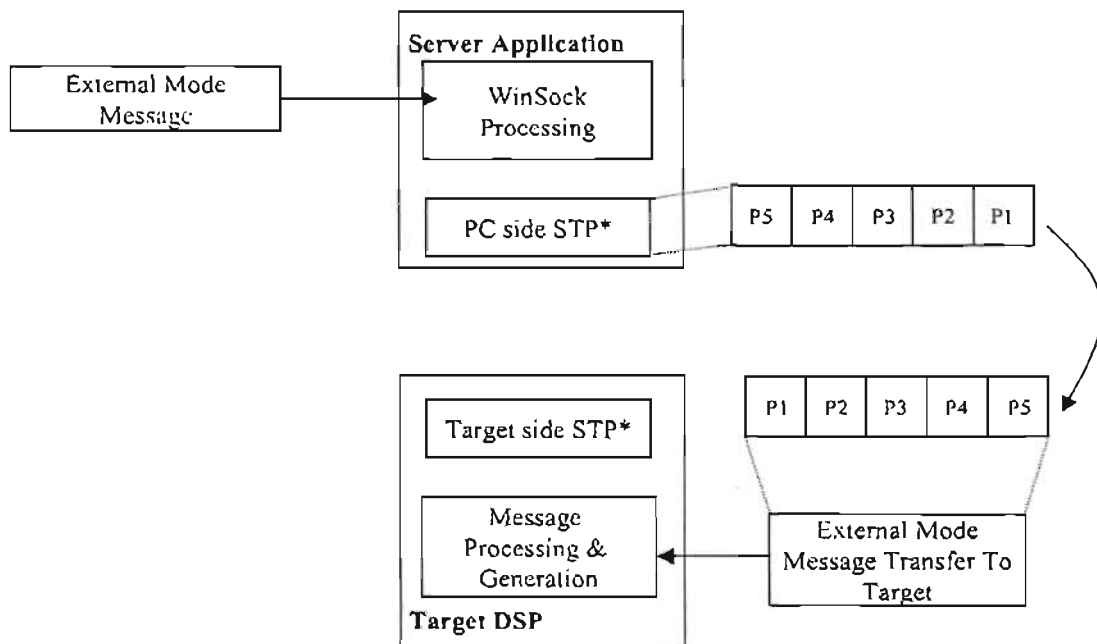


Fig. 4.14: Packetisation of external mode messages

4.7.3 Upload Data Port

This section describes the upload port, which is modelled along similar lines to the message port. The differences are primarily that: data is only transferred from target to server; the amount of upload data is considerably more than message data; and there is also an emphasis on maximising data throughput. A functional diagram of the upload ports is shown in Fig. 4.15; there are two ports A and B, which are

²⁷ On the PC32 the buffers size is 50 while on the ADC64 it is 200.

used in a ping-pong²⁸ fashion to transfer data.

On the target application a logging buffer is filled with data. When this buffer is full it is sent to the server via the upload ports. The buffer is broken into packets and transferred to the server using the ping-pong technique. The server sequences these packets and then transmits the entire buffer to Simulink. This operation is similar to the packeting of the message port as described above and used because the logging buffer is much larger than the size of the upload port's buffer.

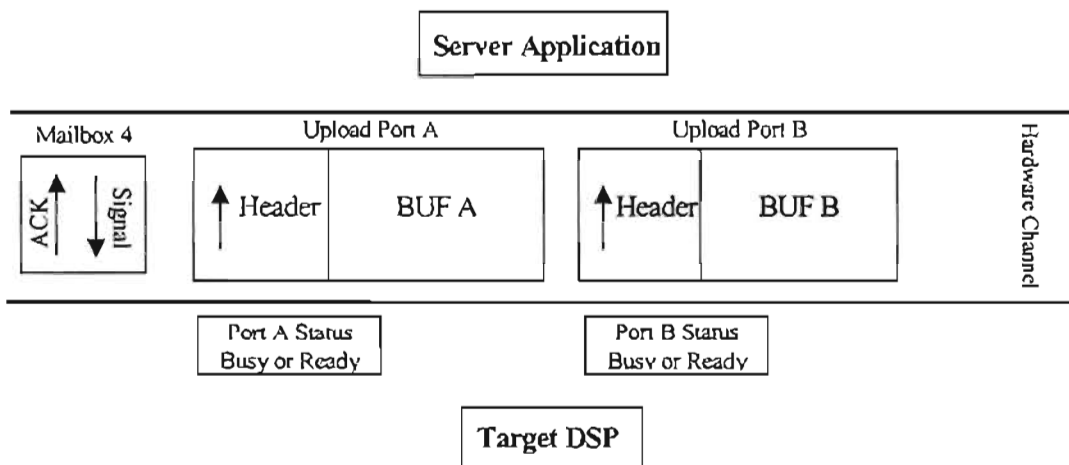


Fig. 4.15: Graphical representation of the Upload Ports

4.8. Conclusion

This chapter presented the RADE framework within the context of The Mathworks TCP/IP implementation. It highlighted and described the four components of the RADE framework which are:

- The Simulink Communications layer
- The Server application
- The STP
- The RTI for the target platform.

The RADE framework presents a methodology to incorporate medium to low end targets within the RTW, which allows full external mode functionality. The next two chapters provide details of application of the RADE framework to the PC32 and ADC64 DSP cards

²⁸ The ping-pong technique is recommended by TI for achieving maximum data throughput.

CHAPTER FIVE:

RADE PC32 IMPLEMENTATION

5.1. Introduction

In chapter 4 a functional overview of the RADE framework was provided, which was mostly independent upon hardware and implementation issues¹. This chapter discusses the implementation of the RADE framework to the PC32 DSP card. This entails the development of Simulink device drives, for the peripheral card I/O and the customising of the RTW files as shown in Fig. 5.1. It also provides an overview of the PC32 DSP card, the Texas Instruments TMS320C32 DSP and PWM card, as they are required for the implementation of the RADE framework.

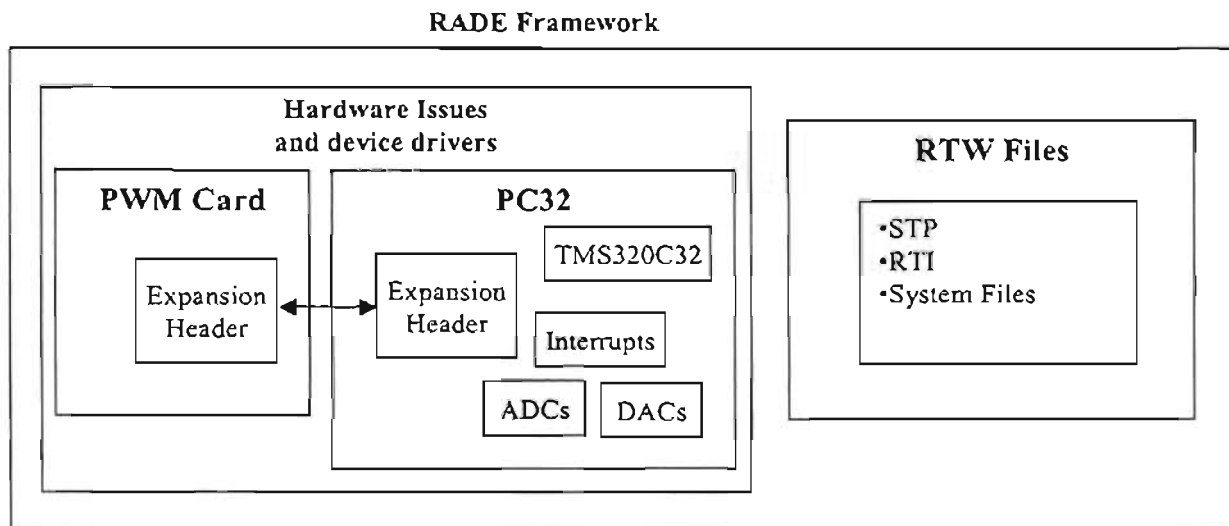


Fig. 5.1 : Overview of RADE PC32 implementation

The implementation of the RADE framework can be broadly categorised into two sections:

- **Hardware issues.** This entails the development of device drivers and is presented in section 5.5. Before device drivers can be developed, an overview of the hardware is needed and sections 5.2 to 5.4 respectively provide information on the PC32 card, TMS320C32 DSP and the PWM card.
- **RTW files,** which cover the development of the RTI, STP and system files is presented in section 5.6.

¹ Chapter 4 provides details of data type conversions function, which are hardware specific.

5.2. Description of PC32 Card

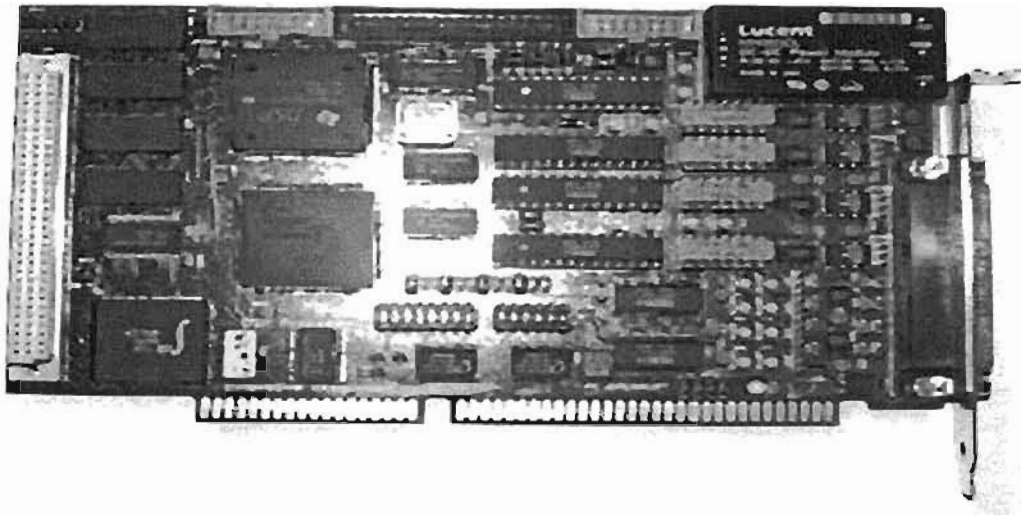


Fig. 5.2: Photo of PC32 card

This section provides an overview of the PC32 DSP card from Innovative Integration (II) [INNOVATIVE1, INNOVATIVE2], shown in Fig. 5.2. It is intended to only describe the functional operation of the card from a rapid prototyping perspective and neglects the complex low-level details, which are not necessary for an understanding of the PC32 RADE implementation².

A block diagram of the PC32 card is shown in Fig. 5.3, which highlights the following aspects: TMS320C32 DSP processor; on-board peripherals; external memory; Dual Port RAM (DPRAM); and an expansion header. The discussion on the TMS320C32 processor is deferred to section 5.3 while the remaining aspects are covered in this section.

As shown in Fig. 5.3 the PC32 card has 4 channels of ADC's consisting of 16-bit Burr Brown ADS7805³ ADC's[BURRBROWN1]. The ADC's are double buffered have a maximum sampling frequency of 100K Hz and can be triggered using three techniques:

- An external trigger signal.
- Either of the internal processor timers.
- By software.

Fig. 5.4 shows the different triggering methods. It should be noted that the ADCs triggers are grouped into two banks as shown in Fig. 5.4 and jumpers 5 and 6 are used to select either of the processor timers for triggering. For the RADE system all the triggering methods are supported but it should be

² An electronic copy of the PC32 software and hardware manuals are found on the CD attached.

³ The datasheet can be found in the PC32 hardware manual.

noted that the external triggering has limited use due to the strict timing conditions specified by Π . The ADC's end conversion signal can also be patched to the processor's external interrupt 2 pin, with the use of a jumper. Π have also included anti-aliasing filters and differential signal amplifiers on each channel to allow for high precision sampling. The sampling voltage range is also adjustable but the default range of $\pm 10V$ is adequate for RADE applications.

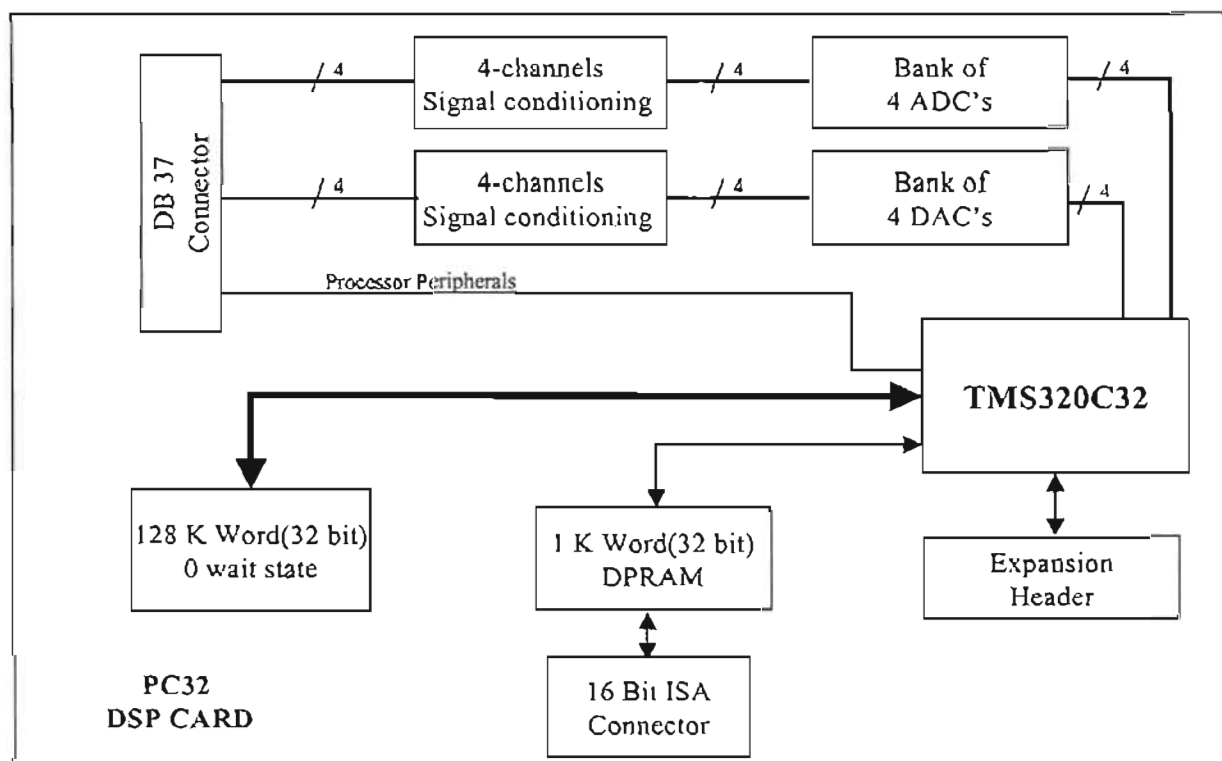


Fig. 5.3: Functional diagram of the PC32 card

There are 4 DAC's channels on the PC32 card, which use the Burr Brown ADS7805² 16-bit DAC IC [BURRBROWN2]. The DAC's are double buffered and are capable of a maximum conversion throughput of 200K Hz. Each channel is connected to a gain stage to allow for different output voltage ranges⁴. The DAC's conversions can be triggered by software or either of the two internal DSP processor timers⁵.

The PC32 card is capable of supporting 128 K Words to 2M Words of static external memory (Fig. 5.3). The cards being used with the RADE system have 128 K Words of external memory. The external memory used, supports zero wait state access, which allows for fast external memory access. A DSP cards performance and ability to run large programs is closely linked to memory sizes i.e. more memory more power. The 128K words, memory size, on the PC32 was sufficient for motion control applications tested on the RADE PC32 system.

⁴ The default range of $\pm 10V$ is used

⁵ The RADE system uses software triggers as this frees up the timers for other uses and reduces the setup complexity for the user.

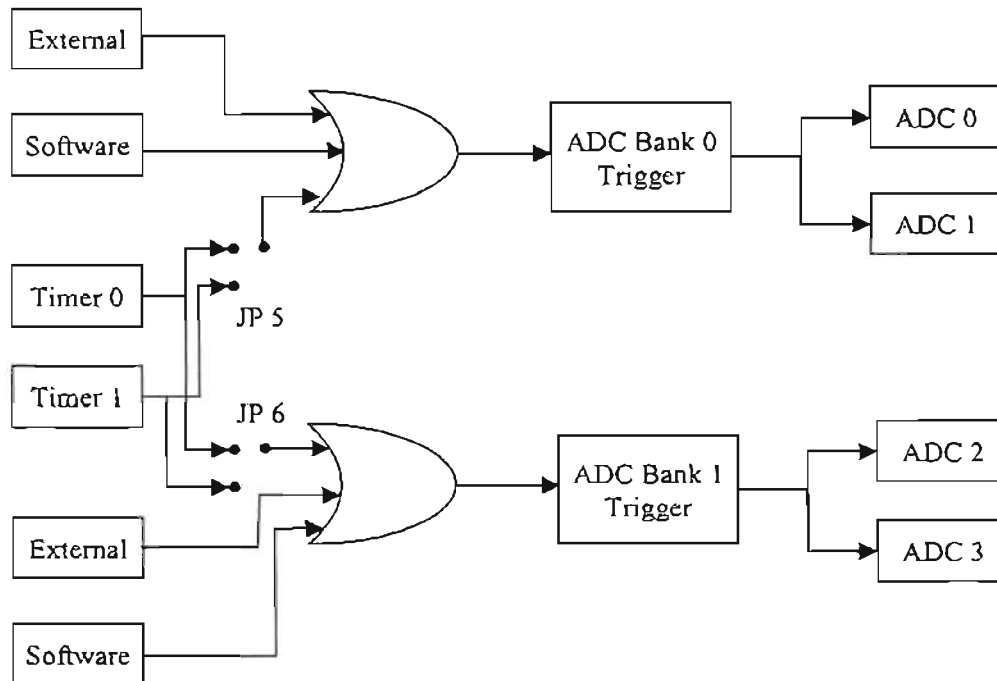


Fig. 5.4: ADC triggering

The PC32 card (Fig. 5.3) also contains 1K Word of DPRAM that is used for target to PC communications. Both the target and the host PC can access this memory and access is arbitrated through the use of four semaphores. The STP overlays the message and uploads ports within this memory.

The PC32 card supports external interrupts from the host PC, ADC's, DAC's and external sources, these signals can be patched to the processor's external interrupt pins with the use of jumpers. The RADE system supports the use of all board interrupts within Simulink, however the PC interrupt has little use within Simulink and is not supported.

5.3. TMS320C32

This section describes the target DSP used on both the PC32 and ADC64 DSP cards. It provides a review of the salient features of the processor pertinent to the RADE system. The low-level details are omitted as all target code written for the RADE system is done in C, which is relatively independent of the internal processor operation.

The TMS320C32 is part of TI C3X family of DSP processors [TEXAS1]. All processors within the C3X family are C code compatible (i.e. have C compilers available) and differ only by internal memory sizes and types of on-chip peripherals. A functional diagram of the TMS320C32 is shown in Fig. 5.5. This processor contains:

- 32 bit data bus.
- 24 bit address bus.
- 32 floating point CPU.
- Two banks of 256 Words of on-chip memory used for either program or data storage.
- 64 words of program cache.
- 2 Direct Memory Access (DMA) channels.
- One serial port.
- Two internal 32-bit timers.
- Four external interrupts.

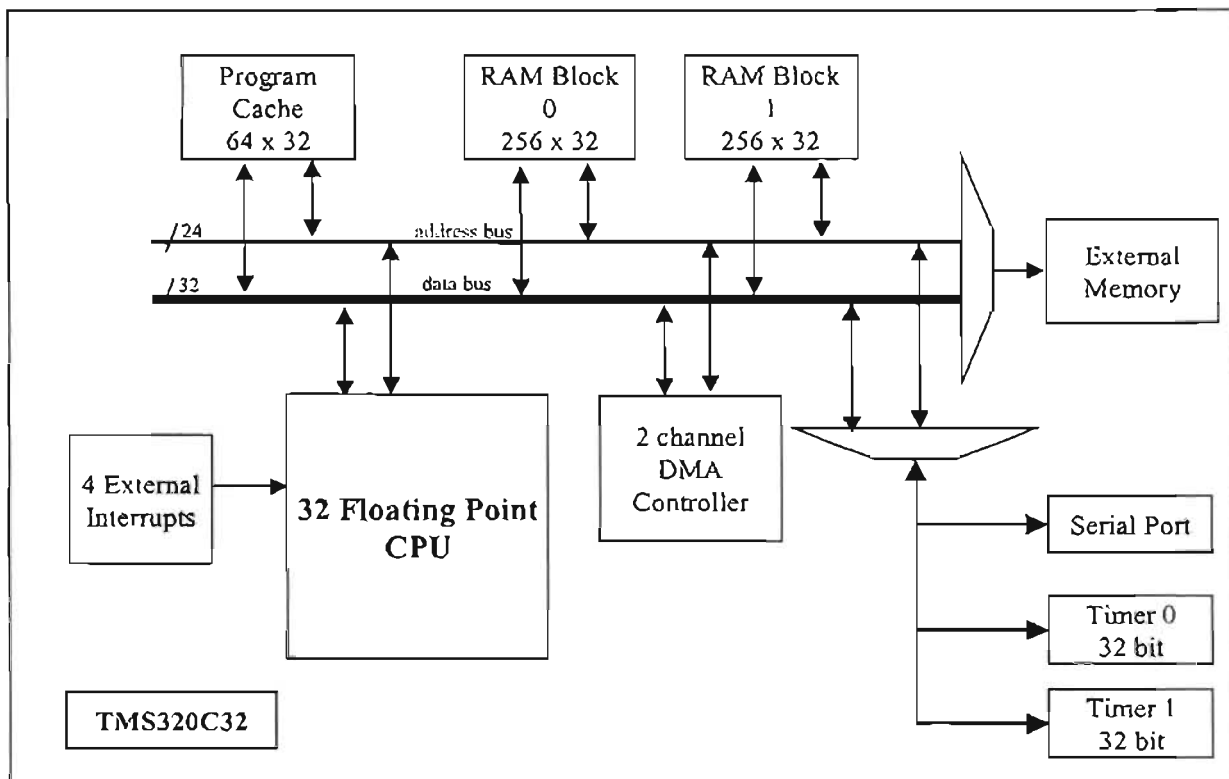


Fig. 5.5: TMS320C32 block diagram

The C32 uses a modified Harvard architecture internally whereby the program address and data buses are separate from data address and data buses. This allows for simultaneous access to internal program and data stores. These buses are multiplexed into a single address and data bus for external memory access. The processor also supports instructions pipelining which allows for one cycle instruction execution provided the pipeline is optimally used⁶. The C32 has an instruction cycle time of 33ns (30M Hz) with a maximum instruction through put of 60 Million Floating Point Operations (MFLOPs)⁷.

⁶ The TI C compiler is designed to produce highly optimised code and ensures minimally pipeline conflicts.

⁷ This processor includes parallel operations in its instruction set and is therefore able to execute two instructions in one machine cycle. This is however not a sustainable level of performance.

In the design of the RADE system, emphasis is placed on rapidly evaluating real-time control strategies and not the efficiency of the code itself. As a result there is no use of on-chip memory and DMA channels in the RADE system. While there may be performance benefits with their use, these are offset by the aim of the RADE system to produce generic code⁸. With processor technology changing rapidly, there is little point to hand optimise code as conventional wisdom dictates that it is easier to use a faster processor. A point in case is the TI 1999 release of the TMS320C33, which is pin compatible with the C32 but 3 to 4 times faster [TEXAS2].

5.4. Description of PWM Card

The RADE system was designed for motion control applications but is applicable to a much wider spectrum of applications, which also included motion control applications. Therefore a PWM add-on card was designed by M. Walker [WALKER1], to interface via the expansion header to the target DSP and provide PWM signal for an external inverter. This reduces the processing burden on the target processor and allows for more complex models to be implemented. The PWM card also supports a Tacho interface, which allows for the easy interfacing of incremental rotary encoders to the target DSP. A photo of the PWM card appears in Fig. 5.6 and a photo of the combined host PC, target DSP and PWM card is shown in Fig. 5.7.

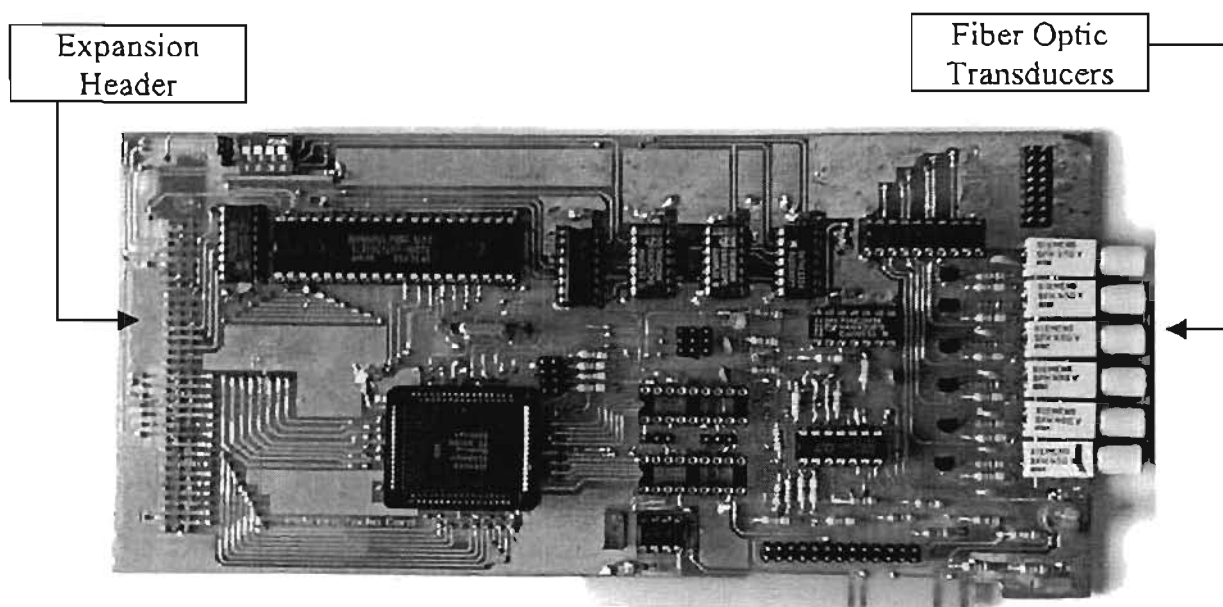


Fig. 5.6: Photo of PWM card

A block diagram of the PWM card is shown in Fig. 5.8. This card primarily consists of two ASIC's by Hanning Elektro-Werke GmbH, the PBM 1/87 for PWM signals [HANNING1] and the TC3005H for

⁸ The RADE system is focused on Educational Value as opposed to efficient real-time code.

the tacho interface [HANNING2]. Both these ASIC's are memory mapped into the target DSP's external memory space and setup by writing command signals to the respective control ports.

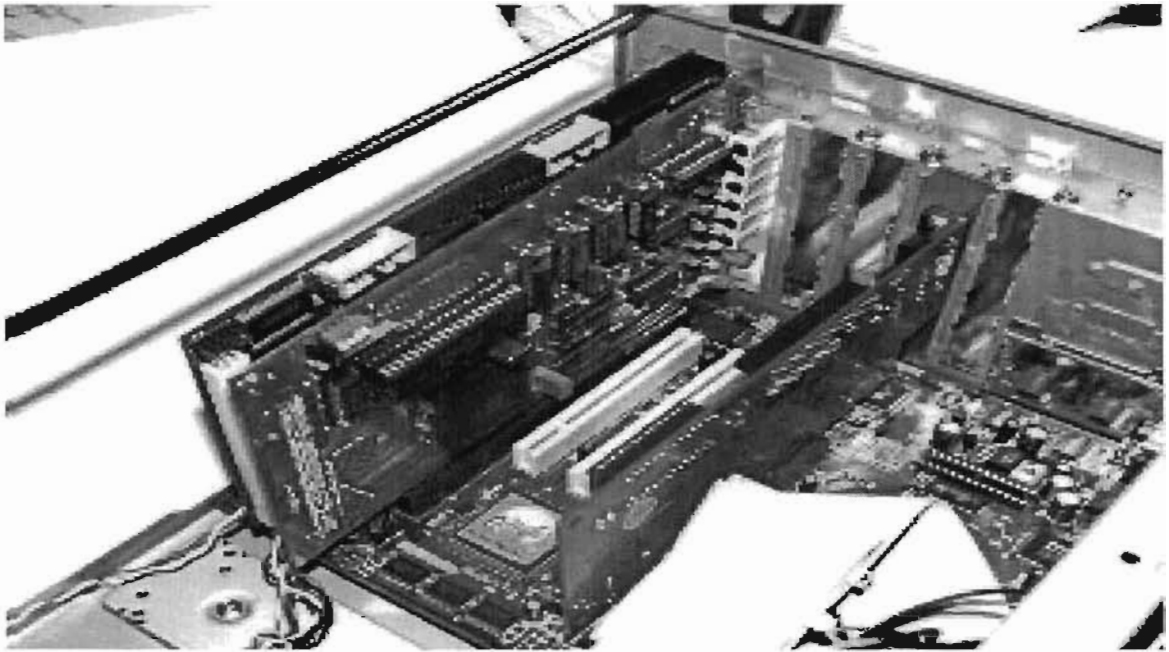


Fig. 5.7: The DSP and PWM plug into the target PC

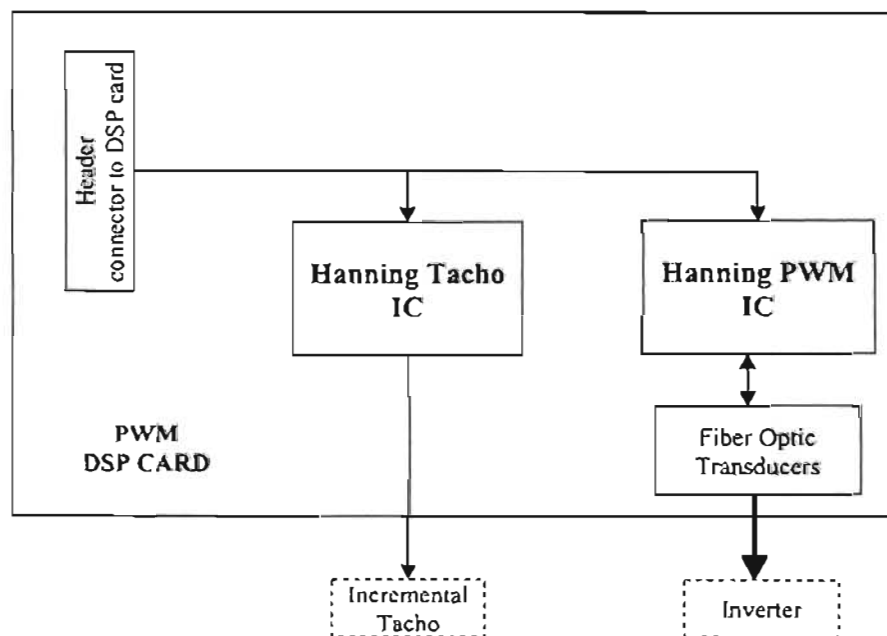


Fig. 5.8:Block diagram Of PWM card

The PBM 1/87 PWM ASIC provides switching signals for a 3-phase⁹ frequency inverter, and is designed to generate a sinusoidal supply at the desired voltage, frequency and phase. The target

⁹ 3-phase signals are intended for induction machines applications. Single phase operation is also possible for DC motor applications

processor is responsible for writing the required PWM setting while the PBM 1/87 performs all the necessary calculations. The PWM ASIC also includes an interrupt feature, which allows for the synchronous sampling of current and voltage waveforms i.e. sampling only occurs when, no inverter switching¹⁰ [STYLO1] is taking place and is therefore *synchronised* to the inverter.

The PWM card also uses high speed optical transducers to enable the PWM signals to be transmitted via fibre optic cable to the inverter. This approach allows for:

- Electrical isolation between the computer and inverter.
- Prevents ground loops.
- Provide better noise immunity.

The use of fibre optic transducers improves the safety of the system and goes a long way to “student proofing” the system, which is a high priority in an educational environment.

The TC3005H tacho ASIC allows for the simultaneous interfacing of two incremental rotary encodes to the target DSP. The tacho ASIC monitors the incremental tacho’s signals and provides both position and speed information. An added advantage of the tacho ASIC is that it allows for both digital and analog incremental tachos to be used. The target DSP communicates with the tacho ASIC using a memory mapped technique, as discussed above and is able to read position data from registers on the tacho chip.

5.5. Device Drivers

An important component of the RADE framework is the development of target specific device drivers for the RTW, as they allow generated code to communicate with target peripherals. This section describes the device drivers used with the PC32 target and explains their operations. The RADE PC32 has the following peripheral incorporated into Simulink:

- ADCs
- DACs
- Processor asynchronous interrupt support and internal timers.
- PWM control block for the PWM card¹¹.

Fig. 5.9 shows the device driver library for the RADE PC32 with the device driver blocks. The individual device driver blocks are now discussed in more detail.

¹⁰ Synchronous sampling allows signals to be sampled relatively free of switching noise.

¹¹ The author acknowledges Stylo's [STYLO1] contribution in the development of the device drivers as the device drives from the CSDE system provided a good framework.

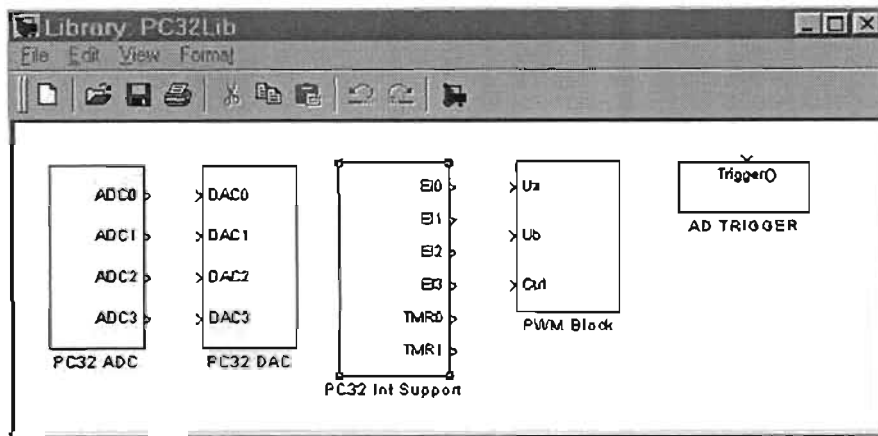


Fig. 5.9: Device driver blocks for PC32

5.5.1 ADC's

The PC32 card contains 4 ADC's and the PC32 ADC (Fig. 5.9) device driver block uses the `read_adc` function to read each of these ADCs. The `read_adc` function is a Zuma toolset function and is described in Table 5-1. The `pc32_ad.tlc` file contains the device driver, of which the important segment is listed below. The value read from the ADCs are 2's complement signed integers and are divided by 3276.7 to convert them into the voltage being sampled i.e. +/- 10V. It is worthwhile to note that after the division the values are stored as 32 floating-point numbers, and therefore, there is minimal degradation to the dynamic range of the data. The `%<LibBlockOutputSignal(0,"", "",0)>` is a TLC function used to write to the output ports of a block [MATHWORKS5]. In the case of the PC32 card, there are four output ports, which correspond to the four ADC's.

```
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
/* read in the corrected values from A/D and scale to +/-10 */
{
  %<LibBlockOutputSignal(0,"", "",0)>=read_adc(BASEBOARD, 0)/(3276.7); %%ADC0
  %<LibBlockOutputSignal(0,"", "",1)>=read_adc(BASEBOARD, 1)/(3276.7); %%ADC1
  %<LibBlockOutputSignal(0,"", "",2)>=read_adc(BASEBOARD, 2)/(3276.7); %%ADC2
  %<LibBlockOutputSignal(0,"", "",3)>=read_adc(BASEBOARD, 3)/(3276.7); %%ADC3
}
%endfunction %% Outputs
```

Function Prototype	Description
<code>read_adc(unsigned int site, unsigned int channel);</code>	<code>read_adc()</code> reads a 16-bit sample from the ADC indicated by site and channel. The result is sign extended to 32 bits.

Table 5-1: The `read_adc` function

I. ADC trigger

The ADCs used on the PC32 card are double buffered i.e. the reading of an ADC value and the triggering¹² are two separate operations. The ADCs device driver discussed above did not perform conversion triggering, which is necessary after each sample. This intended omission is rectified with the ADC trigger device driver that provides this feature. The device driver is found in the *adtrigger.tlc* file and a code segment representing the important part is listed below. The ADCs are triggered by a memory write to their respective memory-mapped address; this does not however affect the data stored from the previous sample. The reason for separating the ADC read and trigger operations is to allow for independent external triggering of the ADCs via one of the processors interrupts. This feature is used in the implementation of a DC motor control, discussed in chapter 7.

```
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
/* a write to those addresses triggers a conversion on A/D */
{
*(ADC0)=0; %% Trigger conversion
*(ADC1)=0; %% Trigger conversion
*(ADC2)=0; %% Trigger conversion
*(ADC3)=0; %% Trigger conversion
}

%endfunction %% Outputs
```

5.5.2 DAC's

The DAC's on the PC32 card are accessed with the *write_dac* function, which is described in Table 5-2. The device driver is found in the *pc32_da.tlc* file and the main segment is listed below. As with the ADC's, the DAC's also use a 2's complement signed integer value. This value is generated by scaling the output¹³ by 3276.7 and then converting it to an integer. The DACs are triggered immediately after a value is written to them by the *convert_dac* function. It should be noted that the DAC's used are double buffered and the writing of data and the triggering of a conversion are two separate operations.

```
%function Outputs(block, system) Output

/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
/* Start an output conversion*/
{
write_dac(BASEBOARD, 0, %<LibBlockInputSignal(0,"",0)>*(3276.7));%%write to latch
convert_dac(BASEBOARD, 0); %% trip conversion
write_dac(BASEBOARD, 1, %<LibBlockInputSignal(0,"",1)>*(3276.7));
convert_dac(BASEBOARD, 1);
write_dac(BASEBOARD, 2, %<LibBlockInputSignal(0,"",2)>*(3276.7));
convert_dac(BASEBOARD, 2);
write_dac(BASEBOARD, 3, %<LibBlockInputSignal(0,"",3)>*(3276.7));
```

¹² ADC trigger sources are discussed in section 5.2.

¹³ The value being passed to the DAC is constrained to the range +/- 10. In simulations were larger signals are written to the DAC's a gain block must precede these signals; this will allow for scaling to the desired range.

```

convert_dac(BASEBOARD, 3);
}
%endfunction %% Outputs

```

Function Prototype	Description
write_dac(unsigned int site, unsigned int channel, int value);	write_dac() delivers a new sample value to the DAC indicated by site and channel. The DAC output will change on the next DAC conversion, triggered by software or hardware.
convert_dac(unsigned int site, unsigned int channel);	convert_dac() triggers a conversion on the DAC indicated by site and channel. A new data sample will be available on the output line as soon as the hardware conversion time has passed.

Table 5-2: The write_dac function

5.5.3 PWM

The PWM block device driver (Fig. 5.9) is used to write the required PWM setting to the PWM ASIC. This device driver is found in the *pwmblock.tlc* file and the main segment is listed below. During real-time code execution the *pollpwm()* function polls the PWM ASIC until it is ready to receive data. When this is true the required output values UA and UB are written to the PWM ASIC. The last setting to be written depends on the control mode (CtrlMode) parameter, which specifies either the frequency or phase angle register of the PWM ASIC.

```

%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
{
    *(Status_word) = 129;
    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 0)>; %%UA
    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 1)>; %%UB

    if ((int)%<CtrlMode> == 1) /* skip three values to write frequency */
    {
        pollpwm();
        *(Status_word) = 897;
    }

    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 2)>; %% CtrlMode
}
%endfunction %% Outputs

```

5.5.4 Asynchronous Interrupt Support

The interrupt support block for the PC32 card provides functionality that allows the user to synchronise execution of subsystems to external events. These events include external interrupts and timer overflows. The parameters used to setup this block are shown in Fig. 5.10 and consist of:

1. External Interrupt Type
This setting configures external interrupts for either edge or level triggering
2. External Timer Pins (TCLKx)

This setting is used to disable/enable external timer signals that are used to trigger ADC's and DAC's. The settings are either **External** to disable timer signals or **Timerx** to enable signals. The purpose of this parameter is to allow use of the internal timers without affecting the triggering of the ADC's or DAC's¹⁴.

3. TimerX

This parameter is used to setup the frequency of timer overflows.

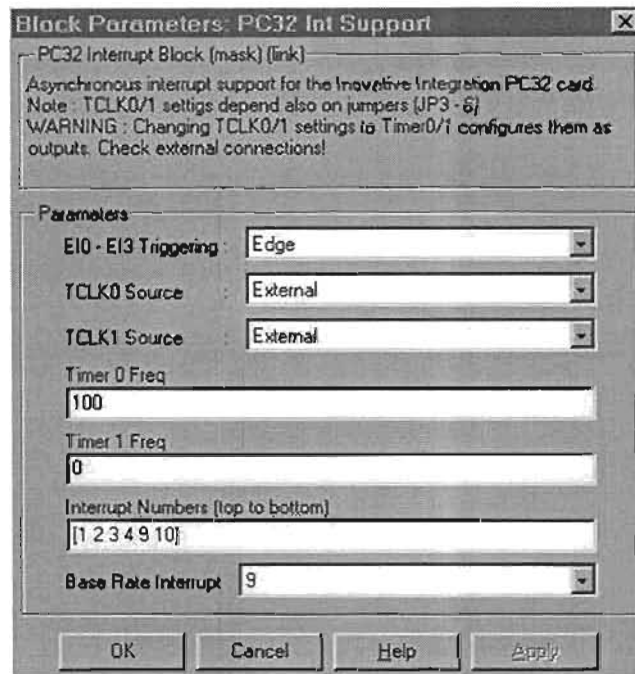


Fig. 5.10: Parameters for interrupt block

4. Interrupt Numbers

This is an array of the interrupt vector addresses for the respective interrupt sources. This parameter is hardware specific and is setup once and does not need to be modified by the user.

The numbers use correspond to the following interrupt sources:

- (1): External Interrupt 0.
- (2): External Interrupt 1.
- (3): External Interrupt 2
- (4): External Interrupt 3.
- (9): Internal Timer 0.
- (10): Internal Timer 1

5. Base Rate Interrupt

This parameter is used to select which interrupt is used to execute the main simulation loop.

The occurrence of this interrupt is expected to be regular as it is used to maintain the absolute

¹⁴ It is also possible to use jumper setting on the PC32 card to manually disconnect these signals however the software approach allows for the same functionality without the need to modified jumper settings.

time, integration of continuous states and data logging operations.

The previous paragraph described the interrupt block from a users perspective and ignored the internal workings, which are necessary for a proper understanding of this device driver as it modifies the workings of the run-time interface.

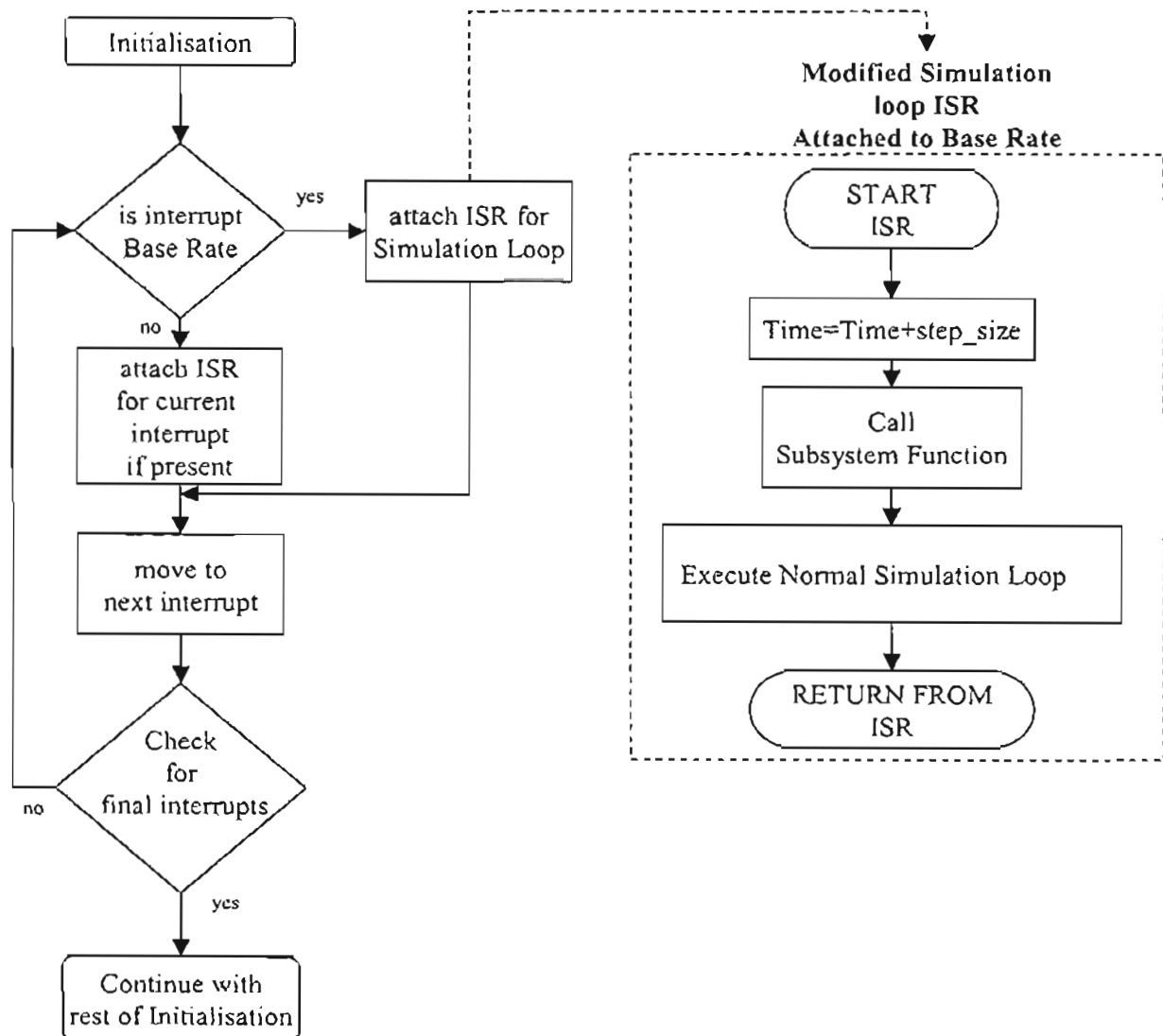


Fig. 5.11: Flow diagram for the modified Run-Time interface

The Mathworks allows Simulink the use of triggered function called subsystems¹⁵, which produce independent functions when real-time code is generated. These functions can be called from the interrupt sources. These functions have to be attached to their respective interrupt sources by the run-time interface. A flow diagram detailing these changes to the *main()*¹⁶ function is shown in Fig. 5.11. There are two parts, the initialisation and the simulation loop ISR that require modifications. The

¹⁵ The Mathworks impose a restriction on triggered subsystem, which prevents them from containing continuous state variables i.e. all block within triggered subsystem with dynamical behaviour must be of the discrete type.

¹⁶ The standard *main()* function is discussed in chapter 4.

initialisation part entails the looping through each of the possible interrupts and installing ISRs for the interrupts that are connected to subsystems. The unused interrupts are ignored. The interrupt that is used to generate the base rate is attached to the simulation loop and the subsystem function is called from within the modified simulation loop. The device driver for the interrupt block is found in the *iiinterrupt.tlc* file

5.6. Customising RADE for the PC32

During the customising of the RADE system for the PC32 card the following aspects needed to be addressed:

1. The Simulink communication layer as discussed in chapter 4.
2. The runtime harness as discussed in chapter 4 and section 5.5.4 above.
3. The implementation of external mode and the STP.
4. The system target file
5. The template make file.

The latter three issues listed above, are discussed in this section. The system files used for the RADE PC32 are listed in Table 5-3 with more details present in the following sections.

System File	Name
System Target File	grt_c3x.tlc
Template Make File	PC32.tmf
Simulink Communications Layer	ext_comm_C3x.dll
Run-time harness	PC32_grtm.c

Table 5-3: Files used for the RADE PC32

5.6.1 External Mode and the Server to Target Protocol

When applying the RADE framework to the PC32 card the implementation of the external mode and the STP represents a large portion of the work. This section documents which files are used on the PC and the target platforms and their respective functions, as shown in Table 5-4. In addition this section also elaborates upon the physical implementation of the STP.

File Name	Purpose
ext_srv_pc.cpp	Implements WinSock part of external mode
ii_comms_pc.cpp	PC side of STP
il_pc32.c	Target side of STP

File Name	Purpose
ext_srv_c3x.c	External mode message processing and data logging routines

Table 5-4: Files used for external mode and STP

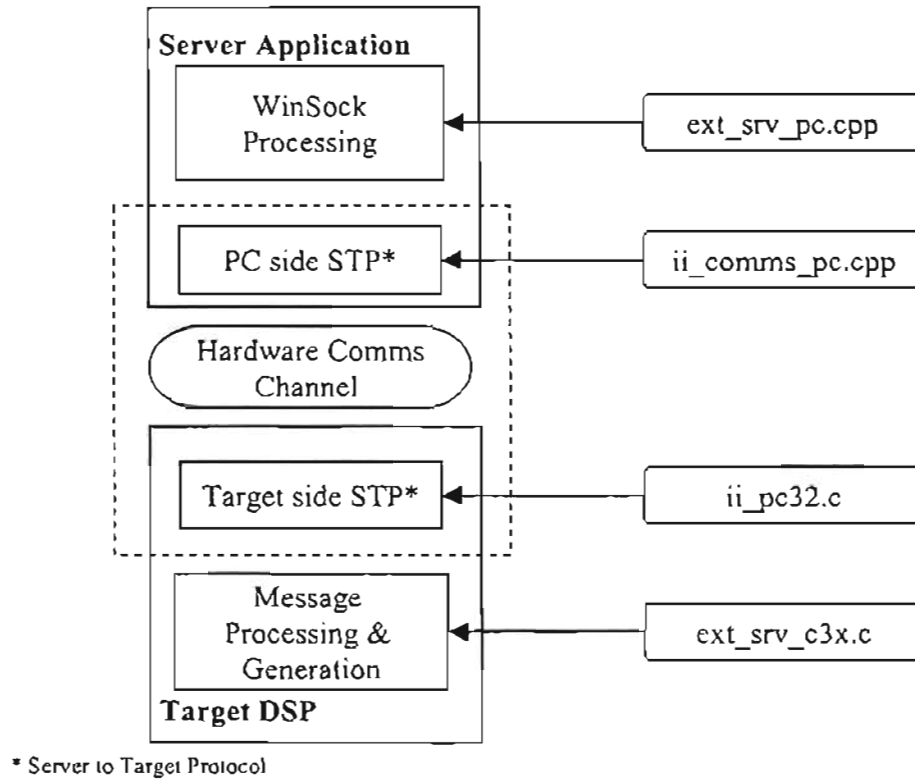


Fig. 5.12: Files used for external mode and STP

Fig. 5.12 shows the entire interaction between the external mode and STP components with the respective files involved. The *ext_srv_pc.cpp* and *ii_comms_pc.cpp* files are used on the PC platform; the former is the WinSock component of The Mathworks TCP/IP external mode implementation, while the latter is the PC component of the STP. On the target platform the *ext_srv_c3x.c* and *ii_pc32.c* files are used. They respectively provide the external mode message processing and the target component of the STP. The STP framework, discussed in chapter 4, explained the operation of both the message and upload ports but neglected the implementational details for the PC32 card. This section now describes these details i.e. the physical characteristics of the message and uploads ports.

The PC32 card as stated before, uses DPRAM to transfer data between the PC and target platforms. This resource is mapped in both the PC and PC32 memory and can be accessed as normal memory from either platform. This allows for the message and upload ports to be directly overlaid in the DPRAM memory segment. The access to the ports themselves is controlled by the use of status flags and mailbox semaphores, which prevents both platforms from accessing the same region of memory

simultaneously¹⁷. The layout for the DPRAM region is shown in Table 5-5. As seen from this table, each port buffers consumes the lion's share of the DPRAM memory, which allows maximum data throughput. The upload port's buffers are significantly larger as to accommodate for the higher volumes of logged data relative to external messages.

Port	Size of port (32 bit words)	Location offset (decimal)	Description
From Server	57 (buffer = 50)	0	This port receives data from the PC
To Server	57 (buffer = 50)	57	This port sends messages to the PC
Upload Buffer A	406 (buffer = 400)	114	This the first of the upload port used for sending logged data to the PC
Upload Buffer B	406 (buffer = 400)	520	This the second of the upload port used for sending logged data to the PC
Total	926		The total size of ports for the transfer of data is 926. 82 words are unused

Table 5-5: DPRAM layout

The code listed below, has been extracted from the *ii_comms_pc.cpp* file, and shows the overlaying of the message and upload ports in DPRAM on the PC platform. This process entails the setting up of pointers to their respective position in DPRAM. On the PC platform the logical address location of DPRAM is non-unique and is found using the ZUMA *target_cardinfo()* function. This address is then copied to the *RAM_START* variable and used as the starting point for the first message port. The remaining message and upload ports are then sequentially overlaid into DPRAM.

```
void Setup_Comms() //PC platform
{
    //used to initionalised global data
    //Over laying two msg port onto DPRAM
    CARDINFO* dsp = (CARDINFO*)target_cardinfo(0);    // (CARDINFO*)isr->cardinfo;
    RAM_START = (int)dsp->BusMaster.Addr;

    To_Target_MP=(p_Msg_Port)RAM_START;
    From_Target_MP=(p_Msg_Port)((UINT)RAM_START+sizeof(Msg_Port));
    //over lay upload buffer onto DPRAM after the two msg ports
    Buf_A=(p_upload_buf)((UINT)RAM_START+2*sizeof(Msg_Port));
    Buf_B=(p_upload_buf)((UINT)RAM_START+2*sizeof(Msg_Port) +sizeof(upload_buf));

    //initialise buffer parameters
    Buf_A->current_size=0;
    Buf_A->free=UPLOAD_BUF_SIZE;
    Buf_A->full_size=0;
}
```

The code listed below, has been extracted from the *ii_pc32.c* file, and details the overlaying of the message and upload ports in DPRAM on the PC32 card. This function is very similar to the one listed above with exception that the starting location of DPRAM on the target platform is fixed at address 0x1000. Aside from this fact both these functions operate in a similar fashion.

```
void Setup_Comms() //target platform
```

¹⁷ It is however possible for different regions of DPRAM to be accessed simultaneously by both the target and PC platforms.


```

{
//used to initlionalised global data
//Over laying two msg port onto DPRAM
  From_Server_MP=(p_Msg_Port)RAM_START;// RAM_PORT=0x1000
  To_Server_MP=(p_Msg_Port)((int)RAM_START+sizeof(Msg_Port));

//over lay upload buffer onto DPRAM after the two msg ports
  Buf_A=(p_upload_buf)((int)RAM_START+2*sizeof(Msg_Port));
  Buf_B=(p_upload_buf)((int)RAM_START+2*sizeof(Msg_Port) +sizeof(upload_buf));

//initialise buffer parameters
  Buf_A->current_size=0;
  Buf_A->free=UPLOAD_BUF_SIZE;
  Buf_A->full_size=0;
}

```

5.6.2 System Target File

The system target file used for the RADE PC32 system is the *grt_C3x.tlc* file and is based on the standard generic real-time system target file provided by The Mathworks (*grt.tlc*) [MATHWORKS4]. The modifications to this file involve two parts; the default parameters and RTW build options

The default parameters are used to setup the:

1. System description
2. Template make file
3. The RTW build function
4. External mode Simulink communication layer

The code segment used to set these parameters is listed below.

```

%% SYSTLC: Generic Real-Time Target for PC32 \
%% TMF: pc32.tmf MAKE: make_rtw EXTMODE: ext_comm_c3x

```

The RTW build options are used to pass user defined parameters to the make file. This is accomplished by adding in new variables into the option window and by modifying the RTW build option section of the system target file. The segment of one of these variables is listed below, with the entire file being listed in appendix C. The parameter being defined in this segment is the server name.

```

rtwoptions(6).prompt      = 'Server name';
rtwoptions(6).type        = 'Edit';
rtwoptions(6).default     = 'magash';
rtwoptions(6).tlcvariable = 'server_name';
rtwoptions(6).makevariable = 'SERVER_NAME';
rtwoptions(6).tooltip     = ['Enter name of server computer'];

```

The complete option window is shown in Fig. 5.13 and consists of:

1. MAT File modifier

This parameter is used when data is being logged to a MAT file and is not supported by the RADE system.

2. External Mode

Used to select an external mode build.

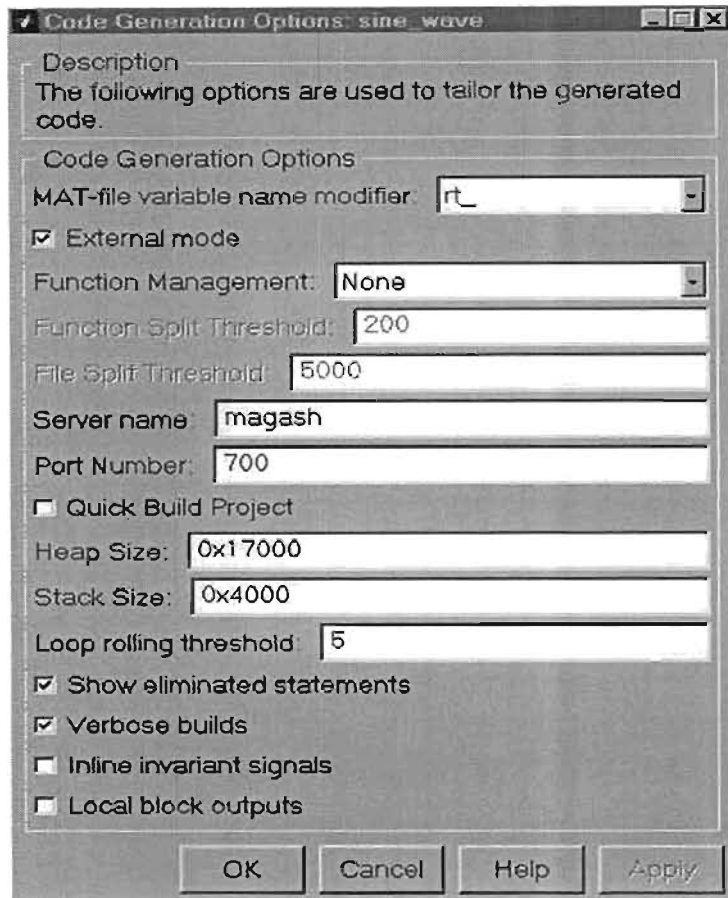


Fig. 5.13: RTW build option window

3. Function Management; Function Split and File Split Threshold

These parameters are a standard feature of the RTW and are used to manage file and function sizes. As this feature has minimal impact on the RADE system it has not been used.

4. Server and Port

These parameters specify the location of the target PC and are used for the automatic downloading of the target application.

5. Quick Build

This flag, when set speeds up repeated model builds by preventing unnecessary compilation of static files.

6. Heap and Stack

These parameters are passed to the linker and setup the required heap and stack sizes.

7. Loop Rolling Threshold

This parameter is a standard feature of the RTW and is used for code optimisation by wrapping TLC algorithms that repeat more than the threshold value, into a for loop.

8. Show Eliminated Statements

Used to comment statements that are eliminated by the TLC optimising process.

9. Verbose builds

Used to enable feedback during the RTW build process.

10. Inline Invariant Signals

Used to inline constant signals that are passed between blocks.

11. Local Block Outputs

Used to place block output variables into a local scope as opposed to a global scope.

5.6.3 Template Make File

The template make file used for the RADE PC32 system is the *pc32.mf* file. This file is responsible for producing the make file that builds the target application. As template make file is well documented, this section only highlights a few of the changes that have been made to this file. The complete file listing is found in appendix C

In the previous section the RTW build parameters were discussed. These parameters are passed to the make file by using tokens. The segment that is used for this process is listed below.

```
SERVER_NAME      = |>SERVER_NAME<|
SERVER_PORT      = |>SERVER_PORT<|
QUICK            = |>QUICK_BUILD<|
HEAP             = |>HEAP_SIZE<|
STACK           = |>STACK_SIZE<|
```

The section responsible for the downloading of code is listed below. Once the target application has been built the make file download section is called and the *auto_download* application is executed. From the listing below it is evident that make utility inserts the appropriate parameters for the download process, which are passed from the RTW options window.

```
II_ROOT = $(MATLAB_ROOT)\rtw\cli
PC32_DOWNLOAD = $(II_ROOT)\bin\auto_download.exe
download :
    $(PC32_DOWNLOAD) -f$(PROGRAM) -s$(SERVER_NAME) -p$(SERVER_PORT)
#the line above expanded typically looks like the line below:
#    c:\mablabr11\rtw\cli\bin\auto_download -fsample_model.out -starget_pc -p700
#
```

5.7. Conclusion

This chapter discussed the implementation of the RADE framework to PC32 card and highlighted the development of the device drives and the STP. It also provided a hardware overview of the PC32 card, the TMS320C32 DSP and the PWM card. This chapter exclusively concentrated on the implementational details and excluded the demonstration of the RADE system to the educational applications, which is present in chapter 7. The next chapter applies the RADE framework to the ADC64 card and draws on this chapter for the common elements that exists between the two systems. These elements include the TMS320C32 DSP, PWM card and a large portion of the system files for the RTW.

CHAPTER SIX:

RADE ADC64 IMPLEMENTATION

6.1. Introduction

The previous chapter described the application of the RADE framework to the PC32 card. This chapter will describes its application to the ADC64 card. It draws upon the RADE PC32 implementation as both the ADC64 and PC32 cards are TMS320C32 DSP based. While this similarity eases the RADE implementation for the ADC64 card, there are significant differences in its implementation.

An example of a difference between the cards is their communication channel; the PC32 uses the ISA bus while the ADC64 uses the PCI bus. This difference serves as an ACID test for the portability of the RADE framework across different targets and illustrates the flexibility of the STP.

The structure of this chapter is similar to chapter 5. It starts with a functional description of the ADC64 card, progresses to the development of device drivers and ends with a discussion of the RADE ADC64 implementational details. A discussion of the processor and PWM card for the ADC64 card is excluded from this chapter as this has already been presented in chapter 5.

6.2. Description of ADC64 Card

This section gives a brief description of the ADC64 card shown in Fig. 6.1. The ADC64 card uses the TMS320C32 DSP and interfaces to the host PC via the PCI bus. It also contains 8 ADC channels, 2 DAC channels, five external timers, and 128K words of external memory. These peripheral components along with the processor are shown in a functional diagram in Fig. 6.2

The ADC64 card supports 8 channels of simultaneous ADC's and uses the 16- bit Burr-Brown ADS7805¹ [BURRBROWN1] ADC's; as with the PC32 card. On the ADC64 card, the triggering of the ADC's are done in pairs i.e. there are four-trigger signal for the 4 banks of ADCs pairs. The trigger signals themselves can be provided by one of the 5 external timers or by an external source², Fig.

¹ The datasheet can be found in the ADC64 hardware manual. An electronic copy is on the CD attached.

² The external trigger signals for the ADC's are negative edge sensitive and do not have the strict timing specification as with the PC32 card.

6.3 shows this. Unlike the PC32 card, which uses jumpers to select the trigger source, the ADC64 used a software programmable trigger matrix that can patch any of the external timer signals to the desired ADC bank. (Section 6.3.1 elaborates on this topic). The ADC64 card also provides differential inputs with anti-aliasing filters and a gain stage on each channel. The default voltage range for sampling is $\pm 10V$.

The ADC64 card supports 2 channels of DACs, which uses the Burr-Brown DAC712 IC [BURRBROWN2]. These channels also contain low-passing filters for analog signal reconstruction. The default output range for the DACs is $\pm 10V$.

As mentioned above the ADC64 card contains 5 external timers, which are implemented using the 82C54 timer IC. These timers are used primarily for the generation of trigger signals for the ADCs and DACs. The use of these timers together with the trigger selection matrix provides an effective method for the implementation of multirate sampling on this card. See Fig. 6.3.

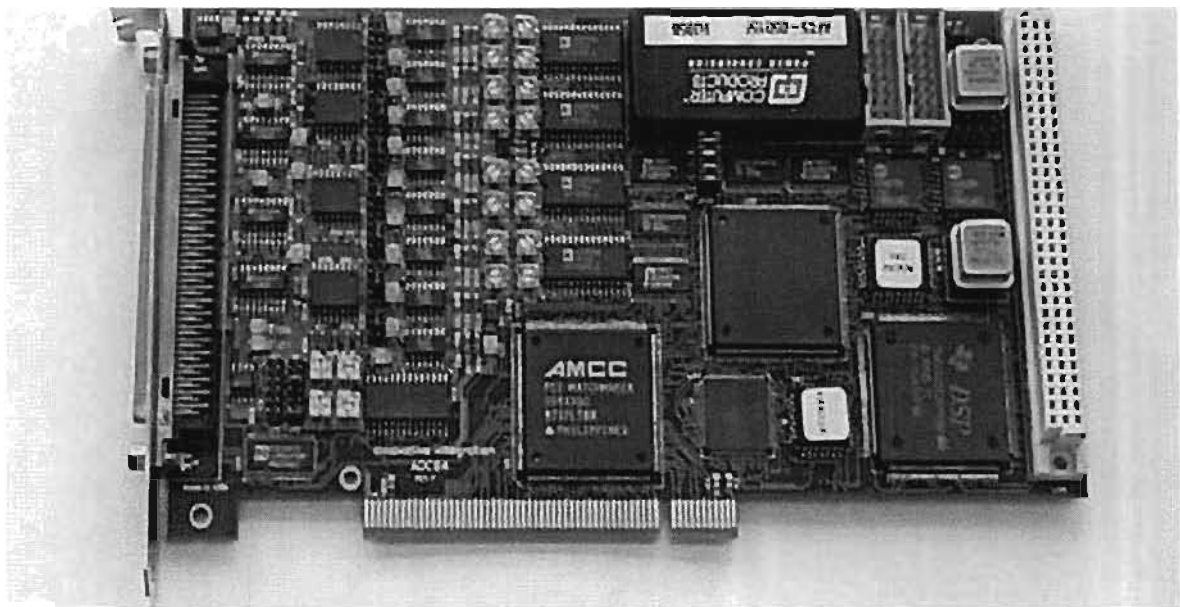


Fig. 6.1 Photo of the ADC64 DSP card

The ADC64 communicates with the host PC via the PCI bus and uses an AMCC S5933 PCI [AMCC1] matchmaker ASIC for this purpose. The DSP on the card communicates to the PCI ASIC via memory-mapped registers. There are two modes of data exchange allowed on the ADC64 card, namely bus master³ transfers or mailbox transactions, with both being supported concurrently. Bus master transfers are ideally suited for the transfer of large banks of data while the mailboxes provide a

³ During bus master transfers the PCI ASIC controls the bus and allows for burst transfers of data between the PC and ADC64 card.

convenient method for the control of communication between the two platforms. The use of the PCI bus significantly improves data throughput on the ADC64 relative to the PC32 as the ADC64 card is rated at 41M-byte/sec-transfer rate while the PC32 only achieves a 400Kbyte/sec-transfer rate. Due to this difference in bus architecture between the ADC64 and PC32 cards the STP implementation for the ADC64 differs and warrants a discussion, which is presented in section 6.4.1.

The ADC64 card also supports end of conversion and PC interrupts which are respectively patched to external interrupts 2 and 3. The first two interrupts EI 0 and EI 1, on the ADC64 card, are used by the PCI ASIC during data transfers. It should also be noted that the PC interrupt is used during bus master transfer. Due to 3 out of the 4 external interrupts being used for on board functionality only the end conversion interrupt is supported for the RADE ADC64 system. Section 6.3.3 elaborates on this topic.

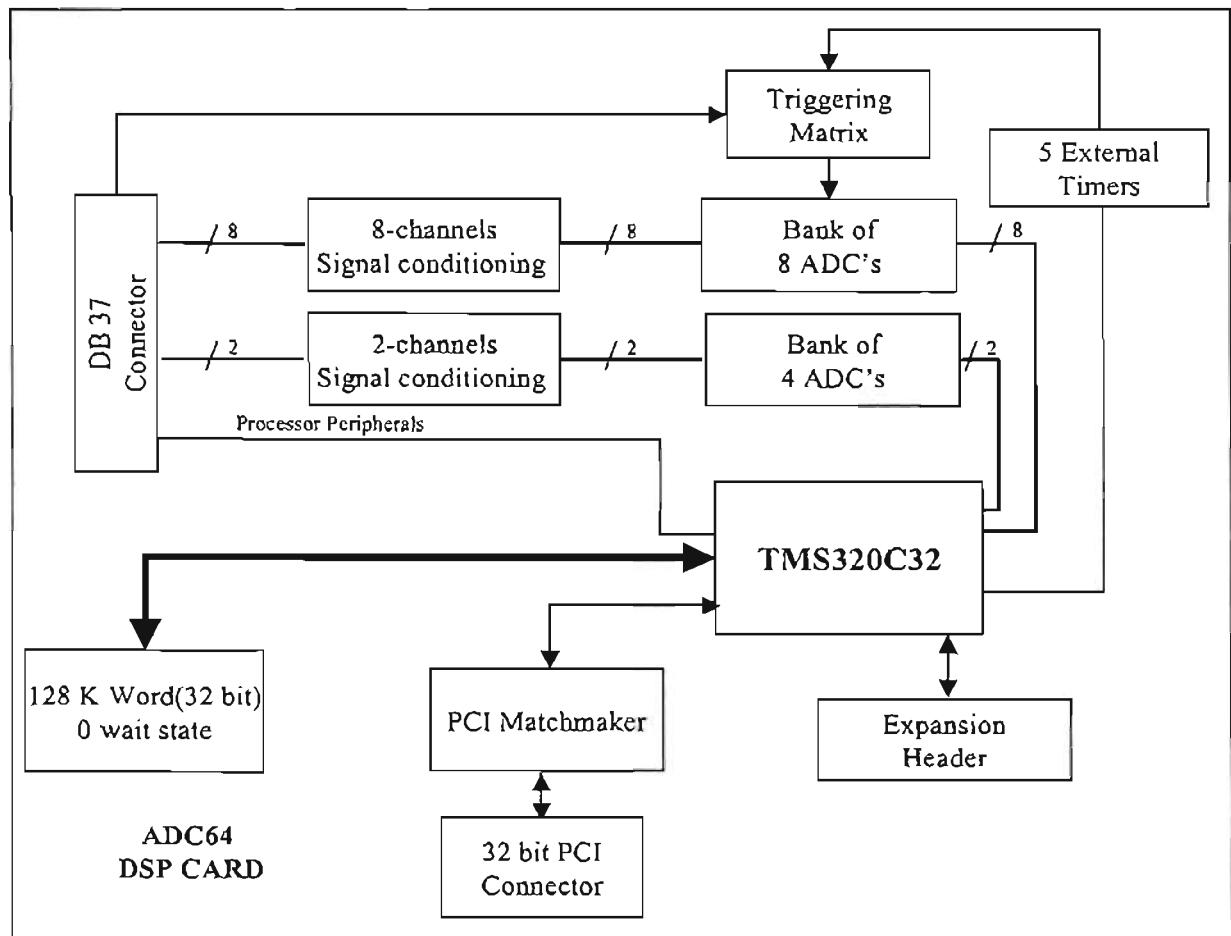


Fig. 6.2 Functional diagram of the ADC64 card

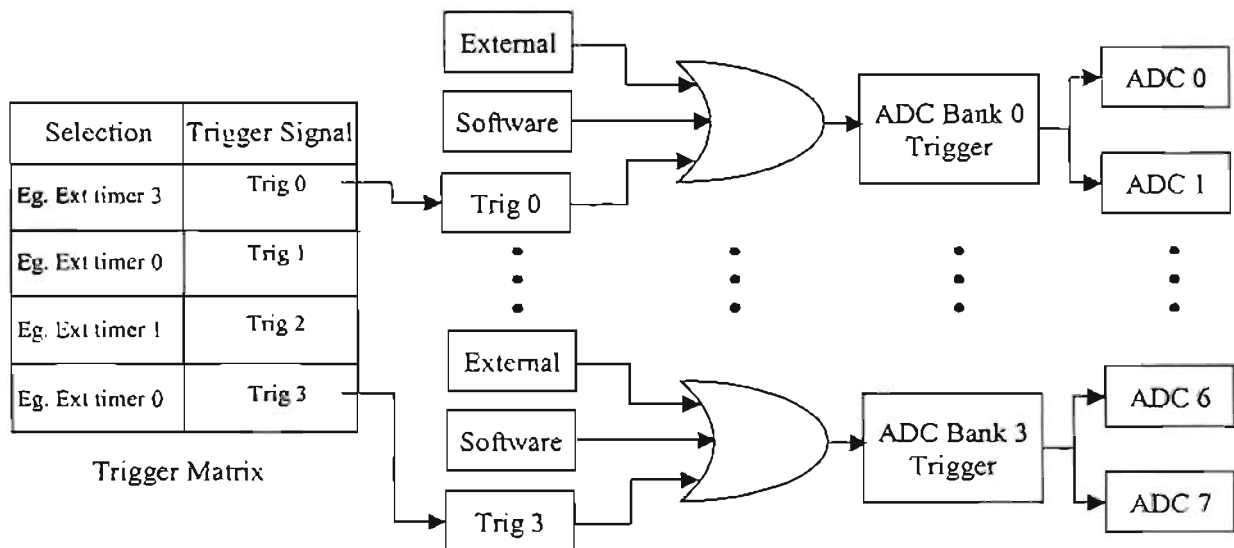


Fig. 6.3: ADC trigger sources for the ADC64 card

6.3. Device Drivers for the RADE ADC64

In chapter 5 the device drivers for the RADE PC32 system were explained and much of this is relevant to the RADE ADC64 system. The peripherals supported within Simulink on both cards fall into similar categories i.e. there are both DACs, ADCs, PWM blocks, and Interrupt blocks. The DAC, PWM and Interrupts device driver blocks are closely based on the PC32 implementation, while the External Timers and ADC blocks have been specifically designed for the ADC64. For this reason the new device driver blocks are discussed in detail while the modified device drive block are explained in perspective of the PC32 implementation.

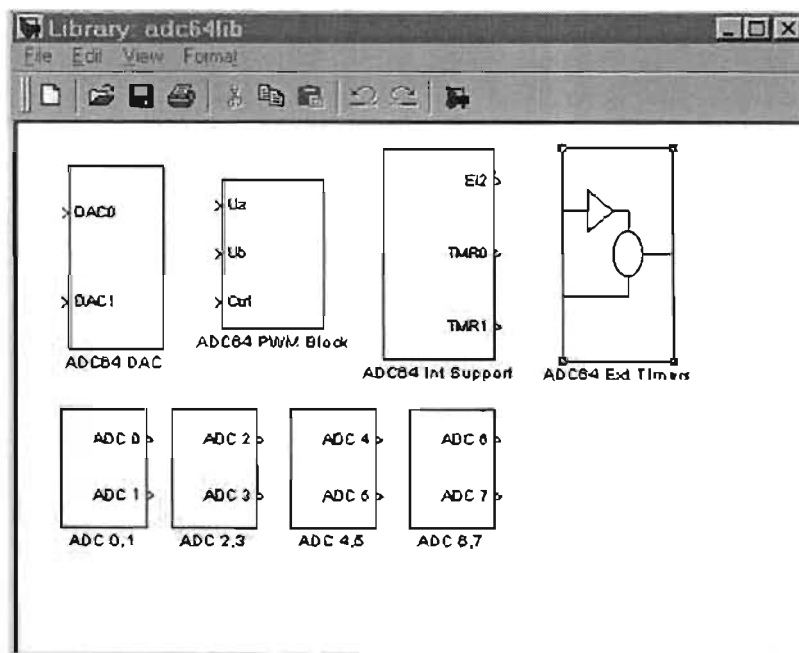


Fig. 6.4: Device Drivers for the ADC64 card

6.3.1 ADC's

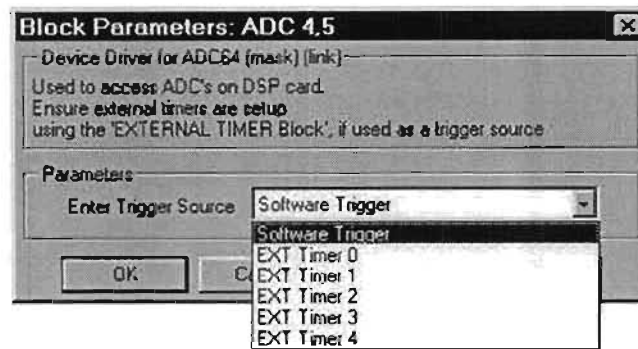


Fig. 6.5: ADC trigger source selection

On the ADC64 card there are 8 ADC channels, which are grouped in pairs for triggering and therefore form four banks of ADCs. Each of these banks is individually implemented in Simulink, as seen in Fig. 6.4 where there are four ADC blocks. Using the respective ADC block's parameter window the appropriate trigger source can be selected as shown in Fig. 6.5. The ADC64 card also supports the use external ADC triggers concurrently with any of the internal triggers i.e. both signals are ORed to produce the final trigger signal. For exclusive external triggering an unused timer must be selected as the trigger source with its frequency set to 0 Hz. (The setting-up of the external timers frequencies is explained in the next section)

The first part of this section described the operational details of the ADC blocks and attention is also given to the implementational details. A single device driver found in the *adc64_ad.tlc* file is used to implement the four ADC blocks. This is done by passing the ADC's bank number parameter to the device driver file, which allows for the physical ADC channel numbers to be generated. The code listed below details this process. The **LibBlockParameterValue (P1,0)** function extracts the ADC bank number passed. The physical ADC channels are generated using this value and are then stored in the *ADC_num0* and *ADC_num1* variables. For example if bank 2 of the ADC was used the physical channel numbers would be 4 and 5 i.e. ADC 4 and 5 make up bank 2. The reading of the ADC then follows a similar procedure to the PC32 implementation and uses the *read_adc* function, previously discussed in chapter 5. The final part of the code implements the ADC software triggering and is only included if trigger source (*Trig_s*) equals 1, which corresponds to software trigger source selection in the parameter window.

```
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
/* read in the corrected values from A/D and scale to +-10 */
%assign Trig_s= LibBlockParameterValue(P2,0)
%assign ADC_num0 = LibBlockParameterValue(P1,0)*2
%assign ADC_num1 = LibBlockParameterValue(P1,0)*2+1
%assign ADC_num0 =CAST("Number",ADC_num0)
%assign ADC_num1 =CAST("Number",ADC_num1)
```

```

{
  %<LibBlockOutputSignal(0,"",0)>=read_adc(BASEBOARD,(%<ADC_num0>))/(3276.7);
  %<LibBlockOutputSignal(0,"",1)>=read_adc(BASEBOARD,(%<ADC_num1>))/(3276.7);
  %if (Trig_s==1)
    convert_adc_pair(BASEBOARD,(int)(%<LibBlockParameterValue(P1,0)>));
  %endif
}
%endfunction %% Outputs

```

Another function of the ADC device driver, is the connection of the ADC bank to the selected external timer. This is done by the code below. This code uses the ZUMA toolset **trigger** function to connect the ADC bank to the selected timer. It is worthwhile to note that the code generated from this segment is inserted in the ADC block's initialisation and only executed once i.e. trigger source cannot be changed during program execution.

```

%function Start(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
%assign Trig_s= LibBlockParameterValue(P2,0)
/* Connect to Trigger source*/
%switch (Trig_s)
  %case 2
    trigger(PIT0_TIMER,(int)(%<LibBlockParameter(P1,"",0)>));
    %break
  %case 3
    trigger(PIT1_TIMER,(int)(%<LibBlockParameter(P1,"",0)>));
    %break
  %case 4
    trigger(PIT2_TIMER,(int)(%<LibBlockParameter(P1,"",0)>));
    %break
  %case 5
    trigger(PIT3_TIMER,(int)(%<LibBlockParameter(P1,"",0)>));
    %break
  %case 6
    trigger(PIT4_TIMER,(int)(%<LibBlockParameter(P1,"",0)>));
    %break
%endswitch

```

Function Prototype	Description
trigger (unsigned int source, unsigned int bank);	trigger() sets the triggering source for a pair of ADC channels in software.

6.3.2 External Timers

The external timer device driver block is used to setup the frequency of the five external timers on the ADC64 card. This device driver produces code that only executes during the initialisation stage of model code and its parameters cannot be changed during execution. Fig. 6.6 shows the parameter window used to enter the timer frequencies.

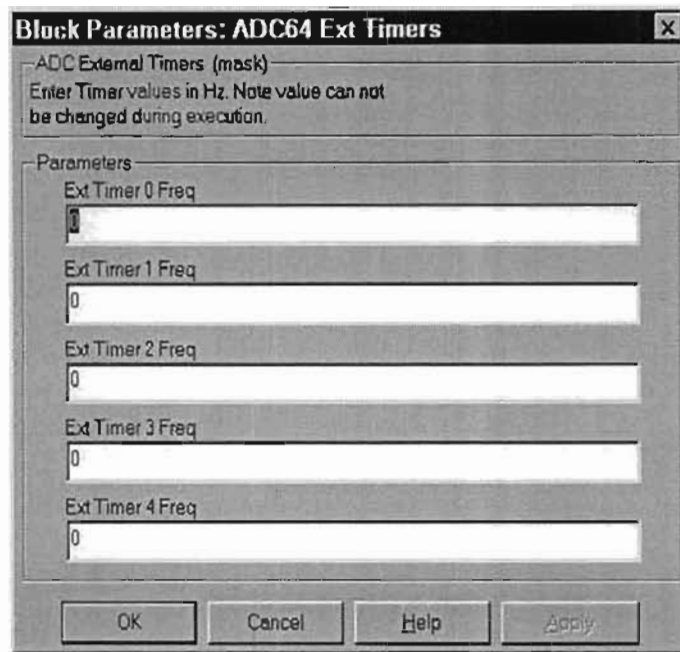


Fig. 6.6: Parameters for external timer Block

The device driver code for this block is found in the *ext_timer_adc.tlc* listed in appendix D with a segment detailing the main part of the device drive listed below. This code uses ZUMA Toolset **timer()** function to setup the respective timers. The variables *tmr0*, to *tmr4* correspond to the values entered in the parameter window of this block.

```
/*setup external timers */
timer(0,(int)%<tmr0>);
timer(1,(int)%<tmr1>);
timer(2,(int)%<tmr2>);
timer(3,(int)%<tmr3>);
timer(4,(int)%<tmr4>);
```

6.3.3 DAC's, PWM and Interrupt Blocks

The device drives for the DAC, PWM and Interrupt blocks are closely based on the PC32 implementation but for a few minor modifications. These modifications entail the changing of the memory location of the respective peripheral if necessary. The device driver file names for these blocks are listed in Table 6-1 with complete file listing in appendix D

Block	Device Driver File
DAC	ADC64_da.tlc
PWM	PWMBLOCK_ADC.tlc
Interrupt Support	iiinterrupt_adc.tlc

Table 6-1: Device driver files

The Interrupt Support block for the ADC64 card only supports 3 interrupts; the two internal timers and one external interrupt used to signal an ADC end conversion event. Fig. 6.7 shows the parameter

window for this block, which is based along PC32 interrupt block already discussed in chapter 5.

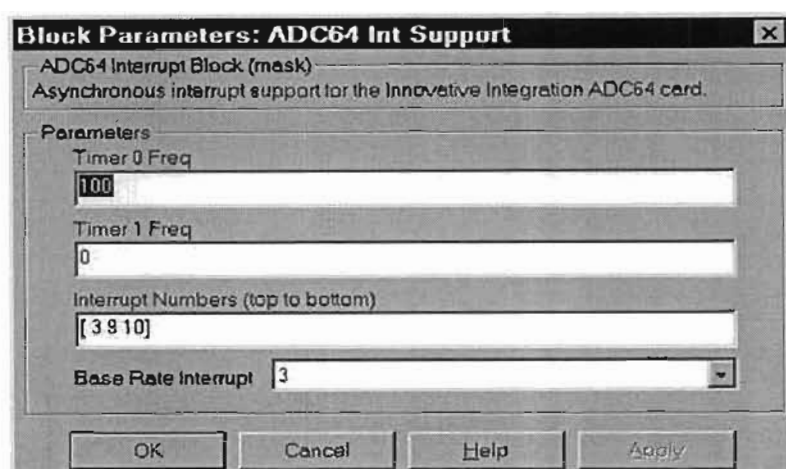


Fig. 6.7: Interrupt block parameters

6.4. Customising RADE framework for the ADC64 Card

To customise the RADE framework for the ADC64 a major portion of the RADE PC32 system can be reused with slight modifications. Table 6-2 lists the system files that are used and summarises the modifications needed. These file have all been renamed to show their association with ADC64 card and prevent any confusion with PC32 system.

System File	Name	Modifications
System Target File	GrI_ADC64.tlc	Change Default parameters
Template Make File	ADC64.lmf	Change files used for STP and run-time harness
Simulink Communications Layer	Ext_comm_C3x.dll	No modifications needed
Run-time harness	adc64_grtm.c	Change initialisation to accomodate for ADC64 peripherals

Table 6-2: Files used for the RADE ADC64 system

The files listed above deal with the RTW build process, run-time harness and Simulink communication layer. The files used for external mode and the STP on the ADC64 have not been discussed as this will be presented in the next section.

6.4.1 External mode and Server To Target Protocol

The RADE framework was designed to encapsulate complexity⁴. This is demonstrated by the use of external mode components from the PC32 system on the ADC64 system. These components are independent of the underlying communication channel and can therefore be reused unchanged on both

⁴ The principle of encapsulation is drawn from object oriented design methods. [BOOCH1]

systems. The principle of encapsulation⁴ allows the changes in communication architecture between the PC32 and ADC64 cards to be restricted to the STP layer. The changes made to the STP layer are discussed below. Table 6-4 summaries the files that are reused and modified.

File Name	Purpose
Ext_srv_pc.cpp	Implements WInSock part of external mode. Same file used for the RADE PC32 system
ll_adc64_pc.cpp	PC side of STP. Modified for ADC64 based on the RADE PC32 system
ll_adc64.c	Target side of STP. Modified for ADC64 based on the RADE PC32 system.
Ext_srv_c3x.c	External mode message processing and data logging routines. Same file used for the RADE PC32 system

Table 6-3: File used for external mode and STP on the RADE ADC64

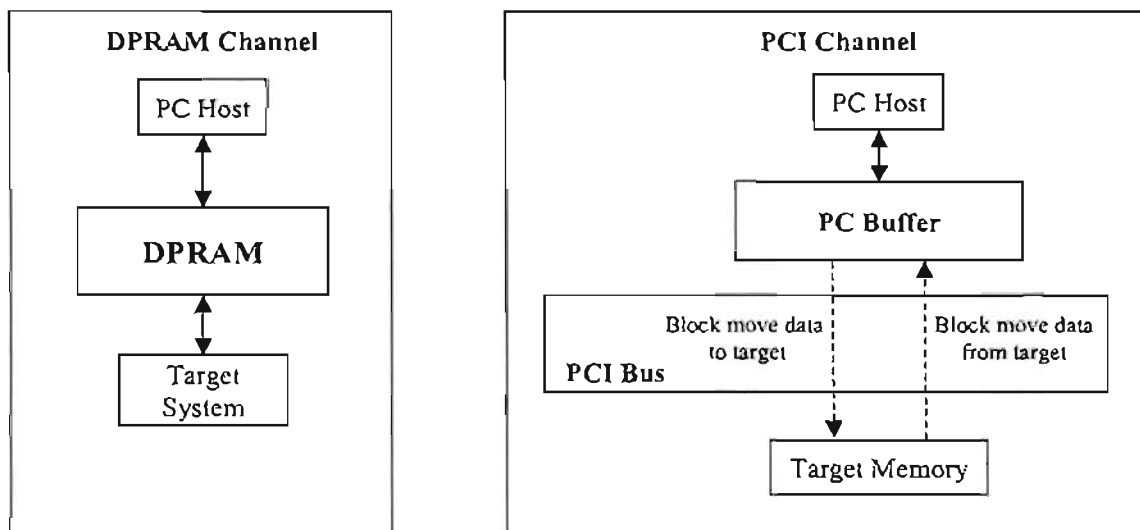


Fig. 6.8: A comparison between DPRAM and the PCI bus

Before the STP implementation for the ADC64 card can be discussed it is advantageous to review the physical differences and similarities between the DPRAM and PCI bus architectures. Fig. 6.8 shows a graphical representation of both these communication channels. In the case of DPRAM both the PC and target platforms have direct read/write access to the DPRAM memory segment. The STP message and upload ports are overlaid in this region and both platforms can read and write the necessary data. In the case of the PCI bus only a PC buffer exists and the target can only block move data to and from between its memory and the PC buffer. This means that the target cannot directly access the PC buffer and memory moves are needed.

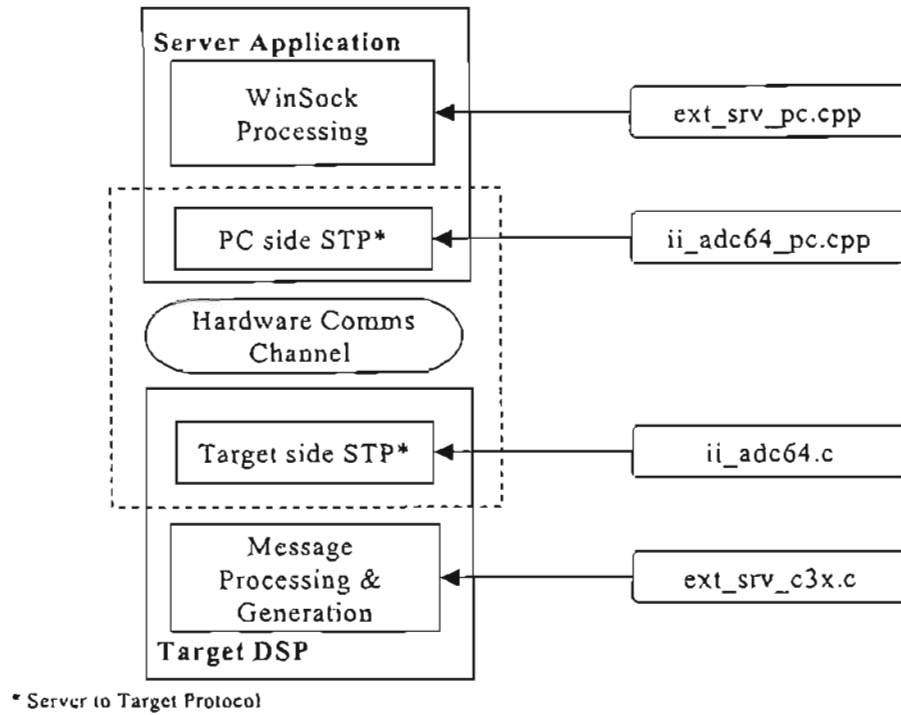


Fig. 6.9: Files used for external mode and STP and the RADE ADC64

The modifications to the STP needed for the PCI bus are shown in Fig. 6.10. Unlike the PC32 card there is no buffer that can be accessed directly by both platforms, as shown in Fig. 6.8. There is, however a PC side 32K word (32Bits) buffer that is used to overlay the message and upload ports. A mirror of this buffer is then maintained on the target and transfers between these buffers allow data to be moved across platforms. This mirroring of the PC buffer on the target allows the PCI bus architecture to resemble DPRAM and allows the reuse of STP code from the RADE PC32 system.

The size of the PC buffer is one-fourth the size of the total memory on the target and it is unfeasible to mirror the entire PC buffer on the target. This problem is solved by only mirroring the message ports and the header part of upload ports. The sizes of the message port and upload ports are listed in Table 6-4. From this table it is evident that the upload ports buffers consume the lion share of memory and eliminating them on the target results in a significant saving of memory. The effective size of the mirror buffer on the target platform is now reduced to around 1K word of memory and does not affect the performance of the target.

The part that does not now tie in is, if the target upload port buffers are removed how is data uploaded to the PC? By referring to Fig. 6.10 it is evident that an upload buffer is already present on the target and data from this buffer can be directly transferred to the PC. It should be noted that this upload buffer is not part of the STP and is declared and controlled by the data logging routines that execute on the target. The upload port headers on the target are used to control the packetisation of this buffer.

This section shows that the STP can be easily modified to fit the PCI bus architecture and is therefore flexible for the porting to different target platforms.

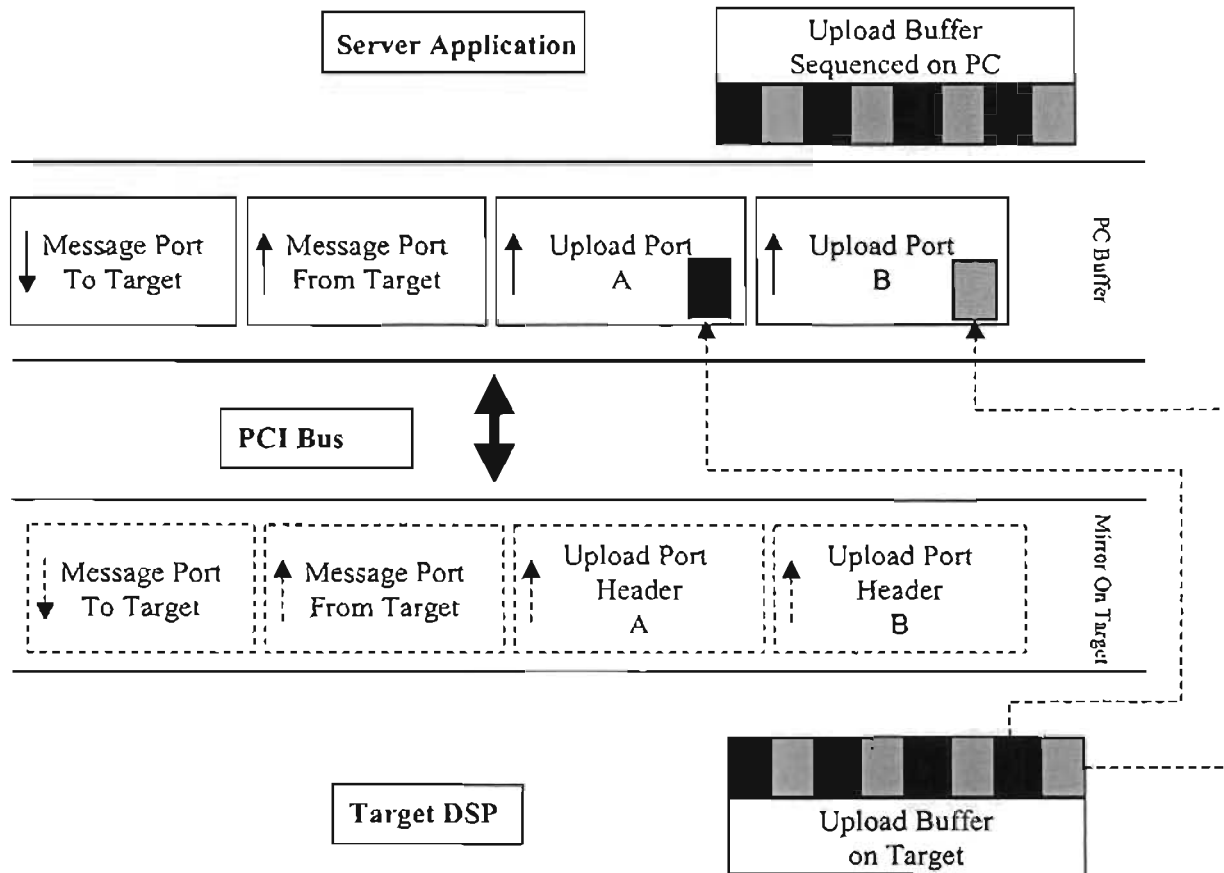


Fig. 6.10: Modified STP for PCI bus Architecture

Port On PC Buffer	Size of port (32 bit words)	Location offset (decimal)	Description
From Server	512 (buffer = 502)	0	This port receives data from the PC
To Server	512 (buffer = 502)	512	This port sends messages to the PC
Upload Buffer A	15872 (buffer = 15862)	1024	This the first of the upload port used for sending logged data to the PC
Upload Buffer B	15872 (buffer = 15862)	16896	This the second of the upload port used for sending logged data to the PC
Total	32768		The total size of ports for the transfer of data is 32768 with no unused space

Table 6-4: Size of ports used for the RADE ADC64 system

6.5. Conclusion

This chapter presented the implementation of RADE ADC64 system. It covered a description of this card, details of how the device drivers were developed and the modifications needed for the STP. It highlighted the portability of the RADE framework between different target platforms. The next chapter demonstrates the RADE ADC64 system and illustrates the use of this system to educational applications.

CHAPTER SEVEN:

DEMONSTRATION OF THE RADE SYSTEM

7.1. Introduction

The previous three chapters detailed the internal operation of the RADE system. This chapter now demonstrates the RADE system as an effective rapid prototyping and teaching tool and describes its use with motor control experiments. Bearing in mind that the primary purpose of the RADE system is to allow students to easily evaluate simulated system with live real-time systems, this chapter is presented in the vein of a student practical and emphasises the controls problems from the student's perspective.

A complete design of a DC servo motor speed controller is presented that covers the entire process from simulation to final implementation. The justification for using a DC servo system is that it is currently being used in both the third year Controls Systems and Electrical Design courses, at the University of Natal's Electrical Engineering department. The Controls Systems course concentrates on the theoretical design issues and uses solely simulation methods to illustrate designs, while the Electrical Design courses emphasises the implementational issues using a micro controller. This chapter now demonstrates how the RADE system can unify both the simulation implementation issues within one course without requiring students to be well versed in software engineering techniques¹.

This chapter rapidly prototypes a theoretically designed speed and current controller and uses the results to verify the performance of both the RADE ADC64 and PC32 systems and demonstrates how students can implement real-time control systems without being preoccupied by complex software engineering issues. Further the theoretical concepts used in the design of the controller will be practically verified.

Topics also included are a demonstration of the concepts of plant saturation, controller stability and integrator windup. Finally a position control experiment is presented with the RADE PC32 system.

¹ It must be noted that the RADE system does not make the Electrical Design courses obsolete as these courses cover lower level implementational specifics, which are hidden by the RADE system.

7.2. A Case Study: Designing a DC Servo Motor Speed controller

This section presents the complete design of a DC servo motor speed controller, which entails the development of a plant model, design of controller parameters and the simulation of the resulting system. Cascaded speed and current loop PI controllers are used, as this control architecture yields good results for both regulation and disturbance rejection specifications [BLERK1]. The controller parameters are designed using a root locus technique, as this is a typical design method taught to under-graduate students in control systems courses².

This section forms the theoretical basis for the evaluation of the RADE systems as an educational tool. The simulated results presented in this section are used to evaluate the real-time results produced by the RADE ADC64 and PC32 systems in sections 7.3 and 7.4.

7.2.1 Motor Model

The block diagram of a separately excited motor is shown in Fig. 7.1 [OGATA1] and is used for the simulation stages. The input to the model is the armature voltage (V_{arm}) and the outputs are armature current (I_a) and shaft speed (W). The parameters for the motor are listed in Table 7-1 and the evaluation is shown in appendix C.

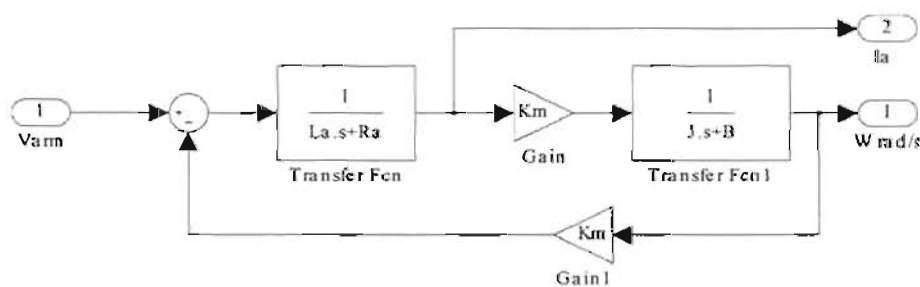


Fig. 7.1: Motor model

² The Universities of Natal and Durban Westville teach root locus design methods in control systems courses.

Parameter	Value	Unit
Armature inductance (L_a)	4.6	mH
Armature resistance (R_a)	3.36	Ω
Rotor moment of inertia (J)	4.889×10^{-3}	Kgm^2
Rotor viscous damping (B)	4.291×10^{-4}	Kgm^2/s
Motor Constant (K_m)	0.834	

Table 7-1: Motor parameters

7.2.2 Design of Current PI Controller

The regulation specification for this design is loosely stated as, “good rise time with minimal overshoot and steady state error”, for a $\pm 8\text{A}$ 10 Hz reference square wave signal. Plant limitations must also be taken into account i.e. maximum supply voltage of 128V. The purpose of such an ambiguous design specification is to allow students the scope to investigate different controller behaviour. The root locus design of the controller is done by using The Mathworks Root Locus Tool (rltool) [MATHWORKS7].

The first requirement for the design is the development of the plant transfer function, which is shown in [7-1] and the corresponding bode plot is shown in Fig. 7.2. This transfer function is derived from Fig. 7.1 with the plant input being the armature voltage (V_{arm}) and the plant output being armature current (I_a) [OGATA1]. The transfer function used for the PI controller is shown in [7-2] [OGATA1]

$$P(s) = \frac{\frac{1}{L_a}(s + \frac{B}{J})}{s^2 + (\frac{B}{J} + \frac{R_a}{L_a})s + K^*} \quad [7-1]$$

$$\text{where } K^* = \frac{BR_a + KeK_t}{LaJ}$$

$$G(s) = \frac{Kp(s + Ki)}{s} \quad [7-2]$$

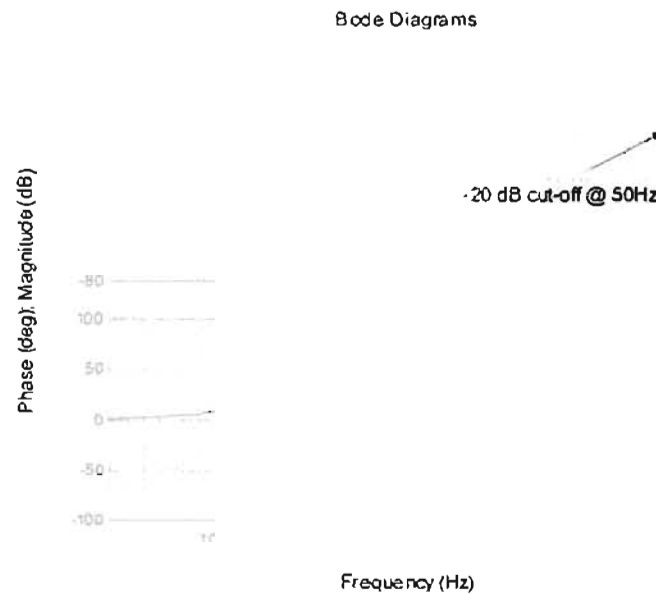


Fig. 7.2: Bode plot of [7-1]

The next part of the design process details the choices of the two controller parameters. Three root locus diagrams are presented showing the plant's response to various controller parameters. In all the subsequent root loci diagrams the controller zero is positioned after (higher absolute radian frequency) the plants complex open loop poles as this configuration allows the current controller integrator to operate faster than the plant's electrical dynamics.

Fig. 7.3 shows the resulting root locus with $K_p=6$ and $K_i=100$ and the corresponding unit step response. From the step response (Fig. 7.3) it seems that a steady state error exists but a closer analysis of the root locus diagram shows a slower first order dominated pole (Fig. 7.4). This dominant pole slows the unit step response down considerably as shown in Fig. 7.4 and makes this design unfeasible to meet specifications. This design however illustrates an over damped response with a "steady state error" and will be verified on the real-time systems. The next step in the design is to reduce the effect of the first order dominant pole.

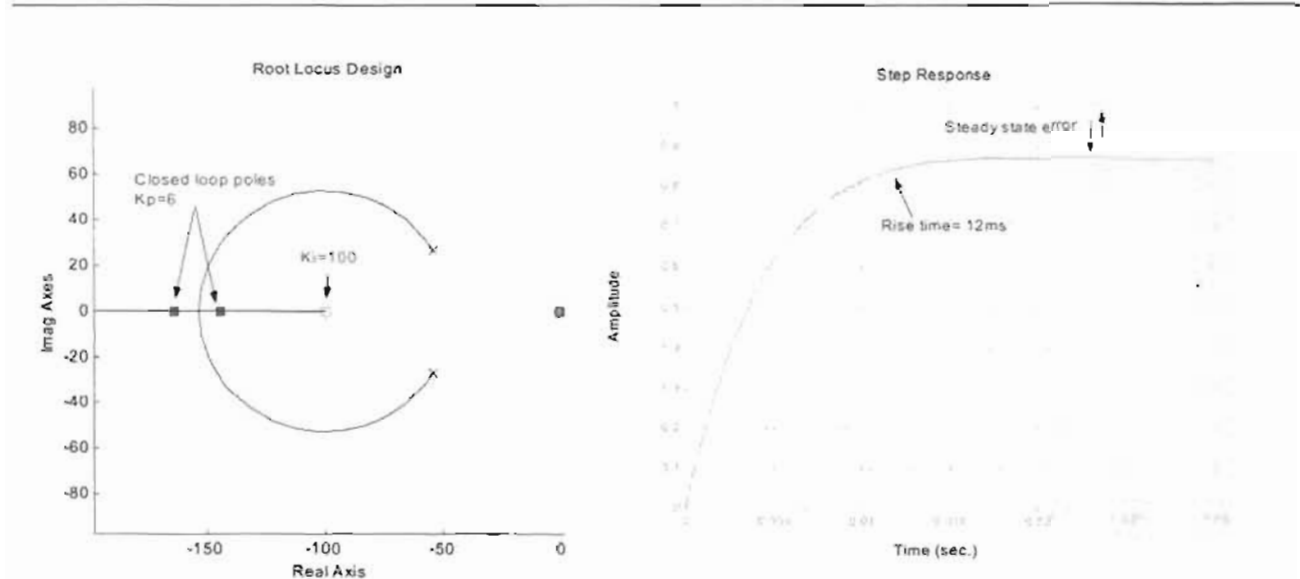


Fig. 7.3: Root locus for over damped response

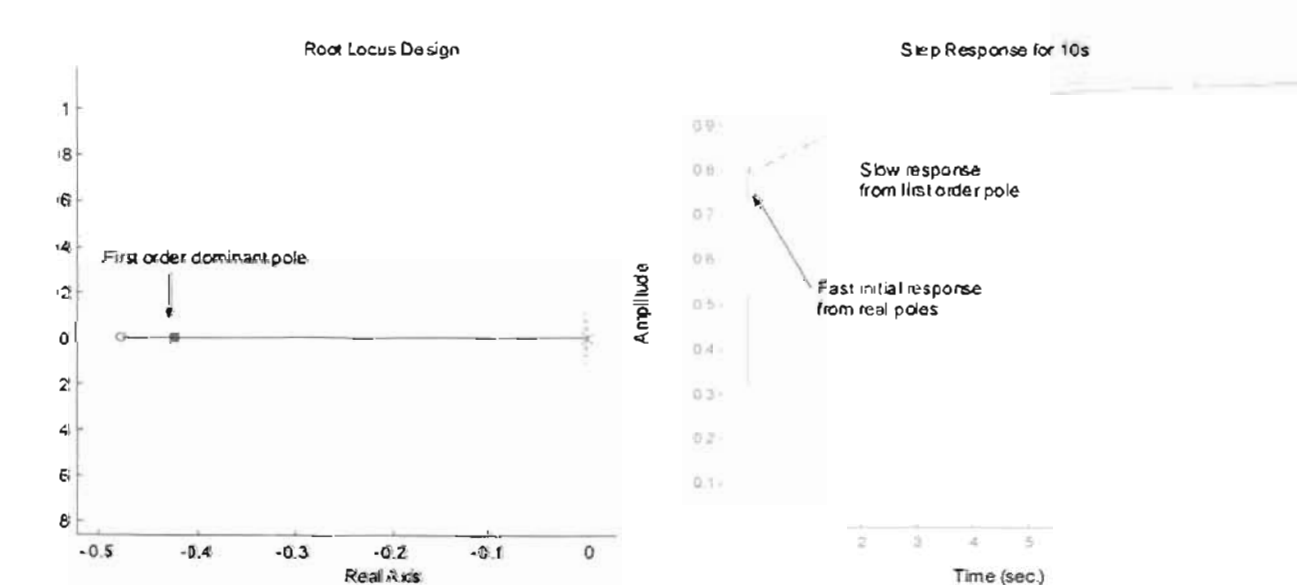


Fig. 7.4: First order dominant pole

The effect of the first order pole can be reduced in two ways:

- By increasing controller gain, the dominated first order pole moves closer to the zero, resulting in a pole zero cancellation. This is unfeasible because the required controller gain of around 30 will saturate the current controller. A maximum error of 8A will demand an armature voltage of 240V, which exceeds the armature voltage that is limited to 128V [AHMEEDI].
- The other option is to make the controller integrator faster as this will integrate out the effect of the slower first order pole [OGATAI].

The controller integrator constant is increased to 1000 ($K_i=1000$) and the proportional constant is reduced to 5 ($K_p=5$)³ with resulting root locus and step response shown in Fig. 7.5. This figure shows

³ K_p was reduced to make the overshoot more pronounced and to demonstrate an under damped response.

an under damped current response with a 36% overshoot and a rise time of 2.7 ms. The faster integrator ($K_i=1000$) has removed the effect of the first order pole and made the current step response faster but has introduced an excessive overshoot coupled with a relatively large settling time. The next iteration in the design will need to reduce the overshoot and improve the settling time.

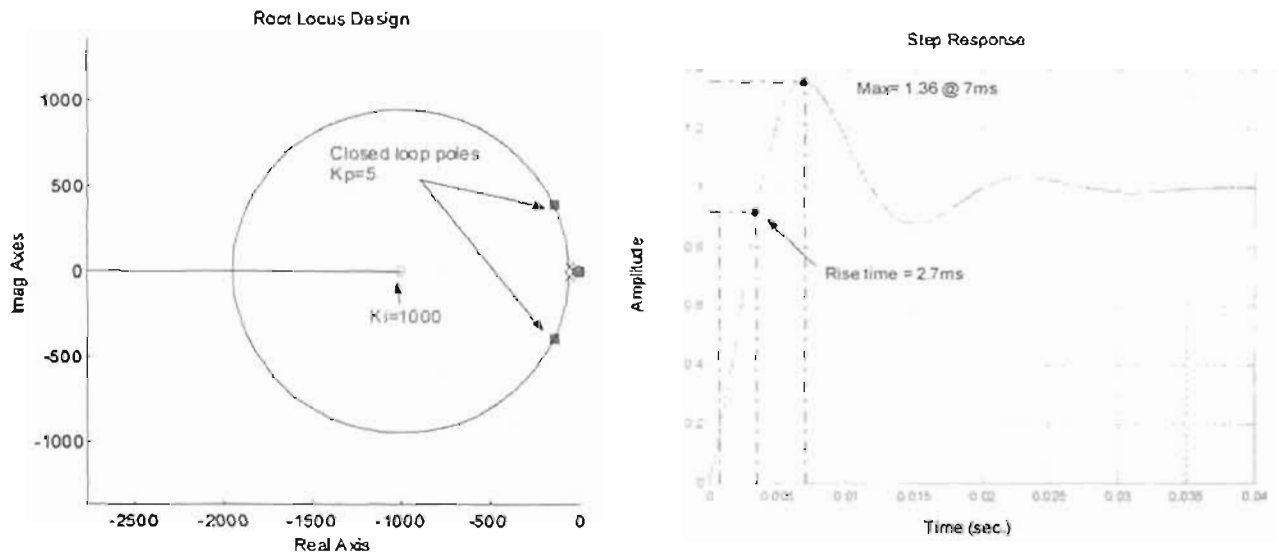


Fig. 7.5: Root locus for under damped response

The excessive overshoot is reduced by decreasing the integrator constant to 200 ($K_i=200$) and increasing the proportional gain to 14.5 ($K_p=14.5$). The resulting design is shown in Fig. 7.6 and meets the specified performance with a rise time of 3.5 ms and a 7% overshoot. This design from this point forward is referred to as the "specified response".

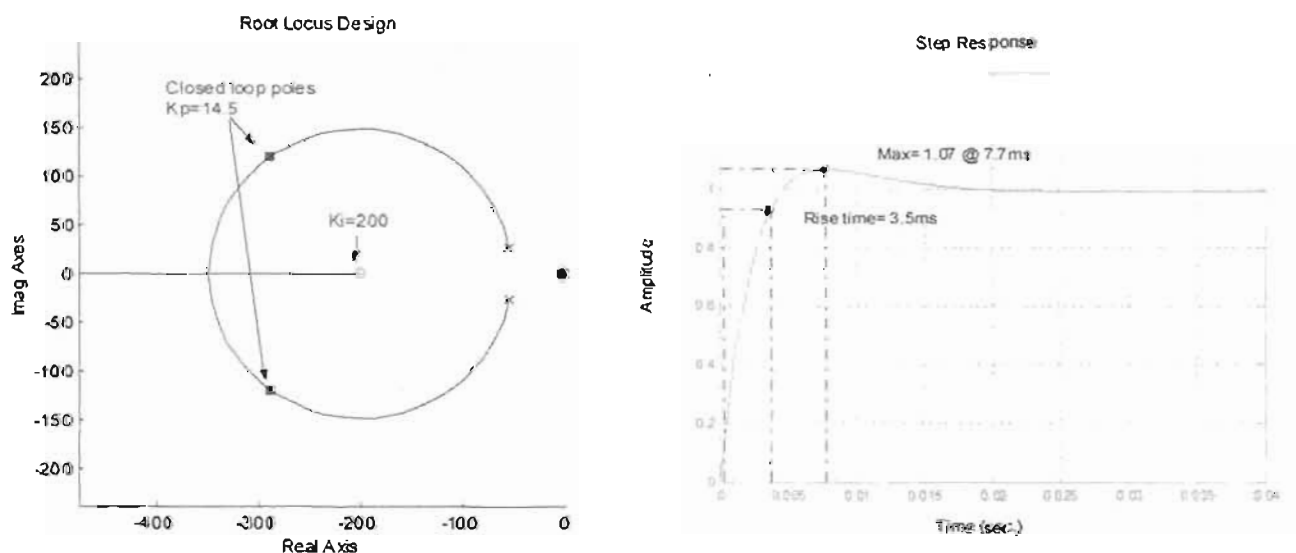


Fig. 7.6: Root locus for specified response

The closed loop bode plot of both the independent plant (i.e. no controller) and the controller and plant

is shown in Fig. 7.7, which demonstrates the benefit of the PI controller on the overall system performance. The overall system bandwidth (-20 dB) improves from 48 Hz to 775 Hz with an infinite gain margin and a 148 -degree phase margin. The controlled system has a relatively flat response (0 dB) up to 70 Hz, which helps improve the regulating performance of the system.

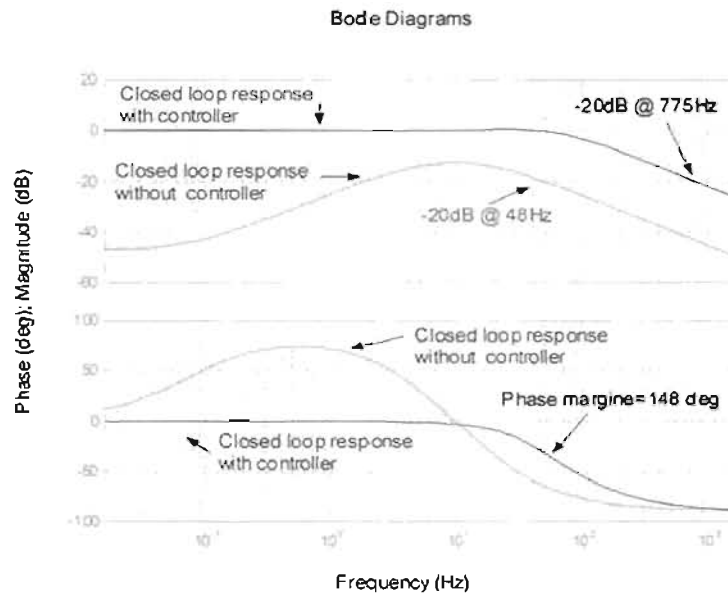


Fig. 7.7: Bode plot of closed loop responses

7.2.3 Simulation of Current Controller

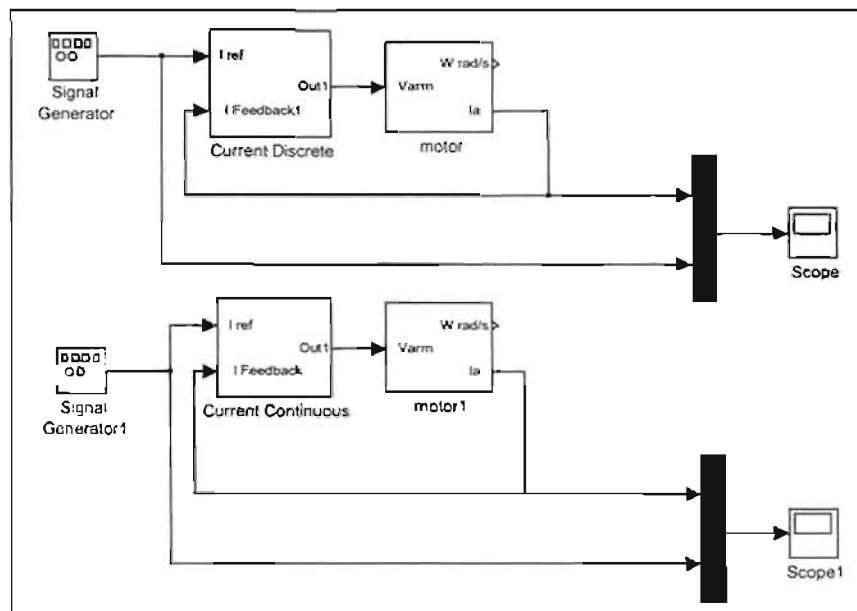


Fig. 7.8: Simulink model for current controller and plant

The controller parameters developed in the previous section are used to investigate simulated plant responses. In this stage of the design process both continuous and discrete PI controllers are

simulated⁴. The continuous PI controller is discretised with the Tustin transform as this yields a good approximation of the continuous system [OGATA2] providing the sampling time is 10 times greater than the -20dB bandwidth of the plant. This sampling constraint is easily met and exceeded with the 8K Hz ⁵ sampling time used noting that the plant's bandwidth is around 50 Hz (Fig. 7.2).

The Simulink model used for the simulation of the current controllers is shown in Fig. 7.8. This model consists of both a continuous and a discrete controller attached to the plant models. The internal workings of the controllers are shown in Fig. 7.9 and Fig. 7.10. Both these controllers contain saturation blocks, which limit the output voltage applied to the motor to $\pm 128\text{V}$ and also support integrator anti-windup. The discrete controller shown in Fig. 7.10 is an implementation of the difference equation [7-4], which is derived from the discrete controller in [7-3] [OGATA2]. Equation [7-3] is the Tustin transform of the continuous PI controller shown in [7-2], where T_s is the sampling time. It should be noted that the zero order holds are omitted from the simulation, since a fixed integration step of $125\text{ }\mu\text{s}$ (8K Hz) is used. This matches the sampling frequency used on the real-time system. In addition, with the inverter not being simulated, the integration step of $125\text{ }\mu\text{s}$ is small enough to simulate system dynamics.

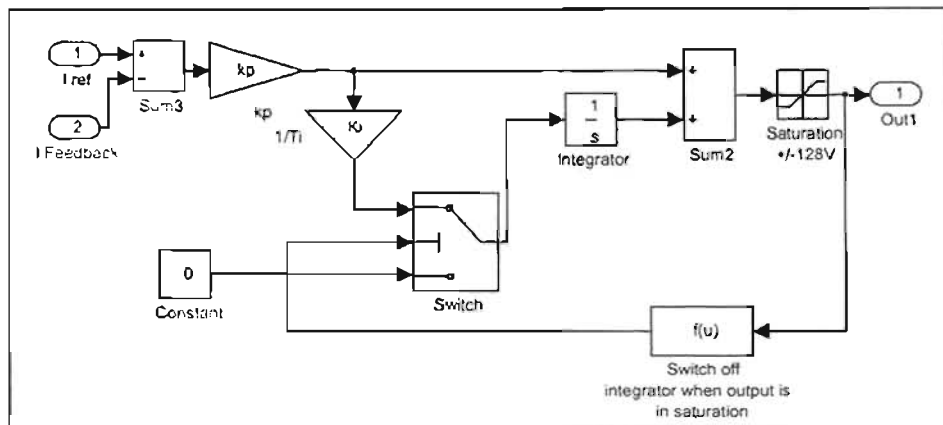


Fig. 7.9: Continuous PI controller

⁴ The discrete controllers are simulated as the controller models used with the RADE system are of this type. Section 7.3.2 provides more details.

⁵ The 8 KHz sampling frequency is used since 4 KHz PWM signals are used to control the power inverter on the real-time system. See section 7.3.1.

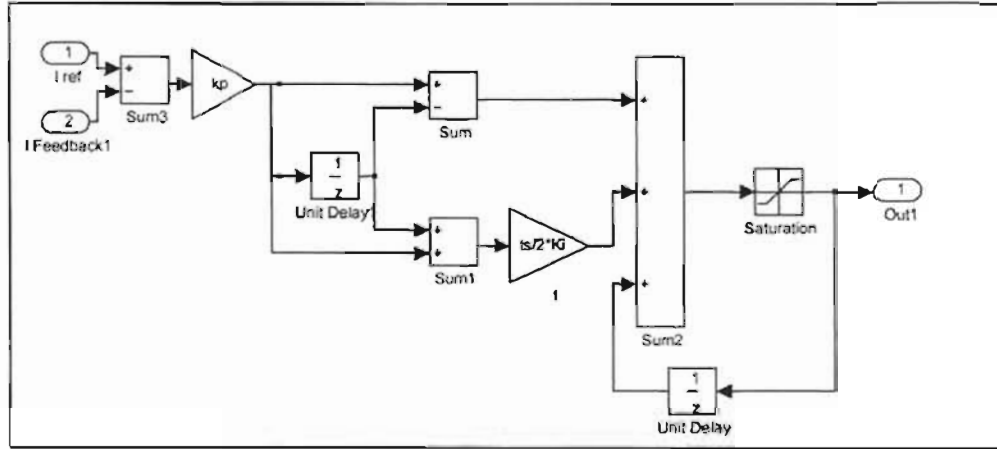


Fig. 7.10: Discrete PI controller

$$G(z) = \frac{kp(1 + \frac{KiTs}{2})z - kp(1 - \frac{KiTs}{2})}{z - 1} \quad [7-3]$$

$$y_n = kp \left((x_n - x_{n-1}) + \frac{KiTs}{2} (x_n + x_{n-1}) \right) + y_{n-1} \quad [7-4]$$

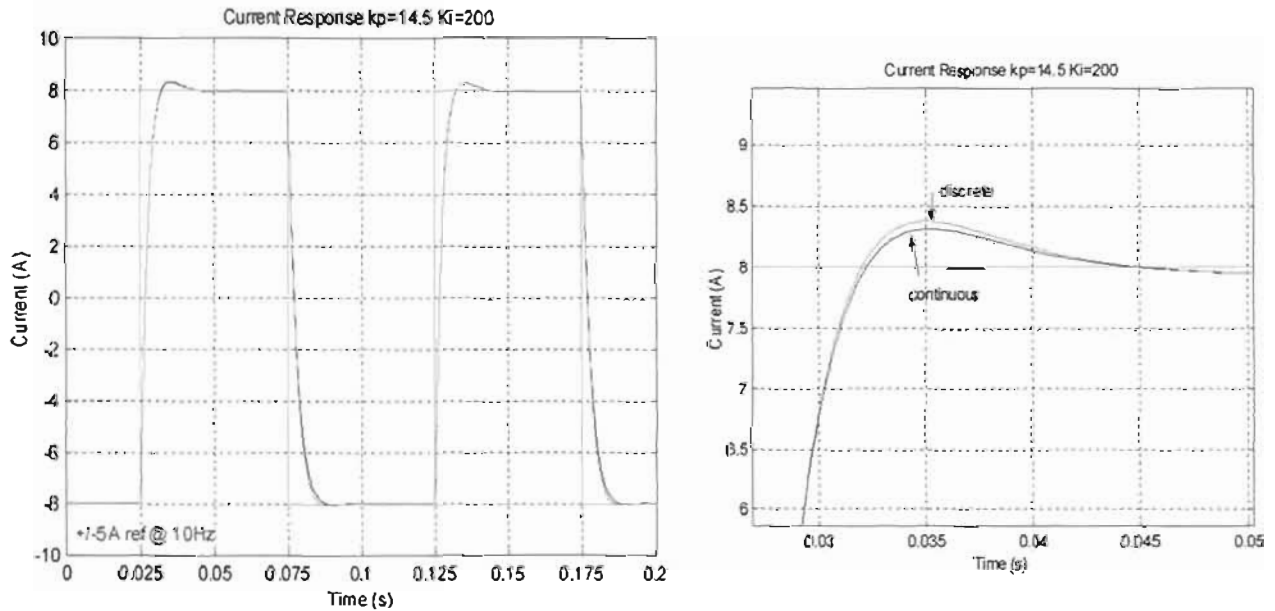


Fig. 7.11: Simulation results continuous and discrete controllers

The specified current controller response ($K_p=14.5$, $K_i=200$) is shown in Fig. 7.11 together with both the continuous and discrete results over-laid and a zoom of the response overshoot also shown. From Fig. 7.11 it is evident that the discrete controller is a good approximation of the continuous controller. The results for the controller over damped and under damped response are shown in Fig. 7.12. It should be noted the under damped response is subjected to a reduced reference of $\pm 5A$ as the motor

current should not exceed 10A on the real-time system. The simulated results presented in this section will be used to evaluate the real-time results produced by the RADE systems. The next part of the design entails the development of a speed loop and this is presented in the next section.

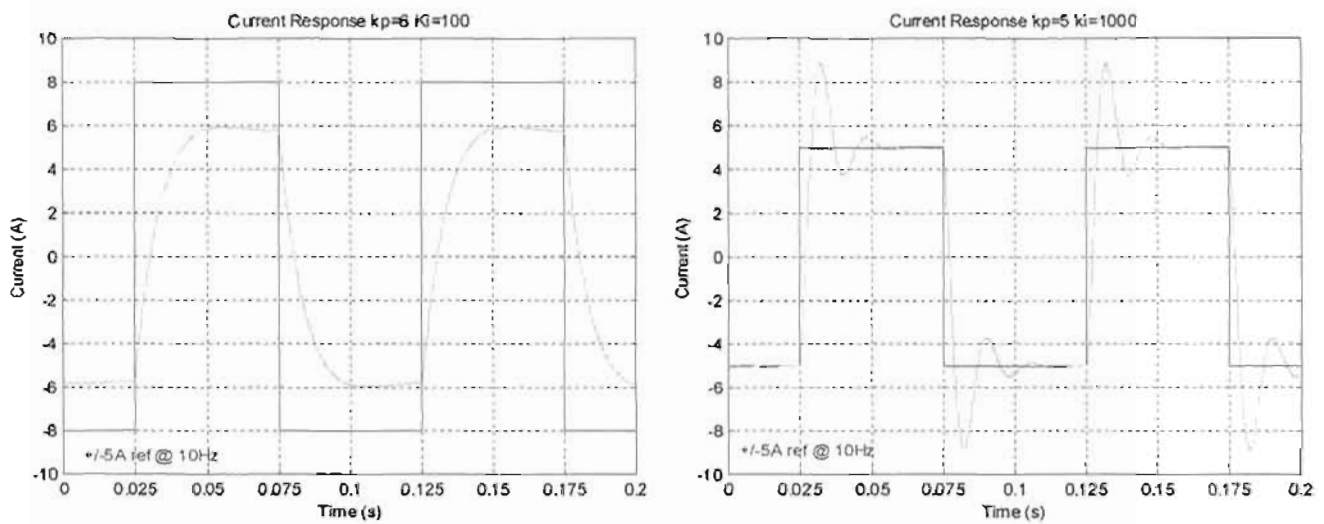


Fig. 7.12: Simulation results for discrete controller over and under damped responses

7.2.4 Design and Simulation of Speed PI Controller

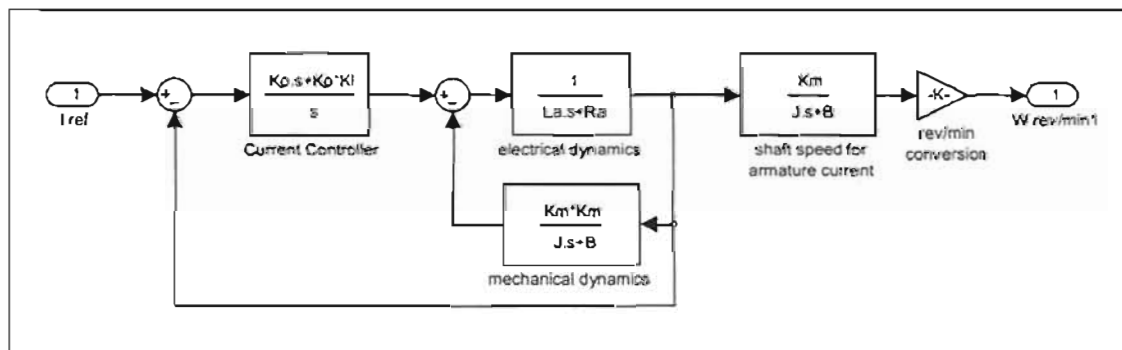


Fig. 7.13: Modified motor model

In the previous section the current loop was designed and is included in the system when designing the speed loop. Fig. 7.13 shows the effective model of the system⁶ used for the speed loop design. The input to the system is current with the output being speed. The regulation specification for the speed loop is loosely stated, as “good rise time with minimal overshoot and steady state error”, for a +/-800 rpm 2 Hz reference square wave signal.

The root locus of this system combined with the speed PI controller is shown in Fig. 7.14. From this figure the mechanical and electrical dynamics of the system can be seen and when designing the speed loop only the mechanical dynamics are of interest. The zoomed mechanical dynamics of the system

⁶ The current controller parameters are set at $K_p=14.5$ and $K_i=200$

presented in Fig. 7.14 shows the speed controller zero for $K_i=20$ and the resulting closed loop complex poles for $K_p=0.03$. The corresponding step response is shown in Fig. 7.15, where a 20% overshoot and 26ms rise time can be observed, which meet specifications. This system is now simulated for these parameters.

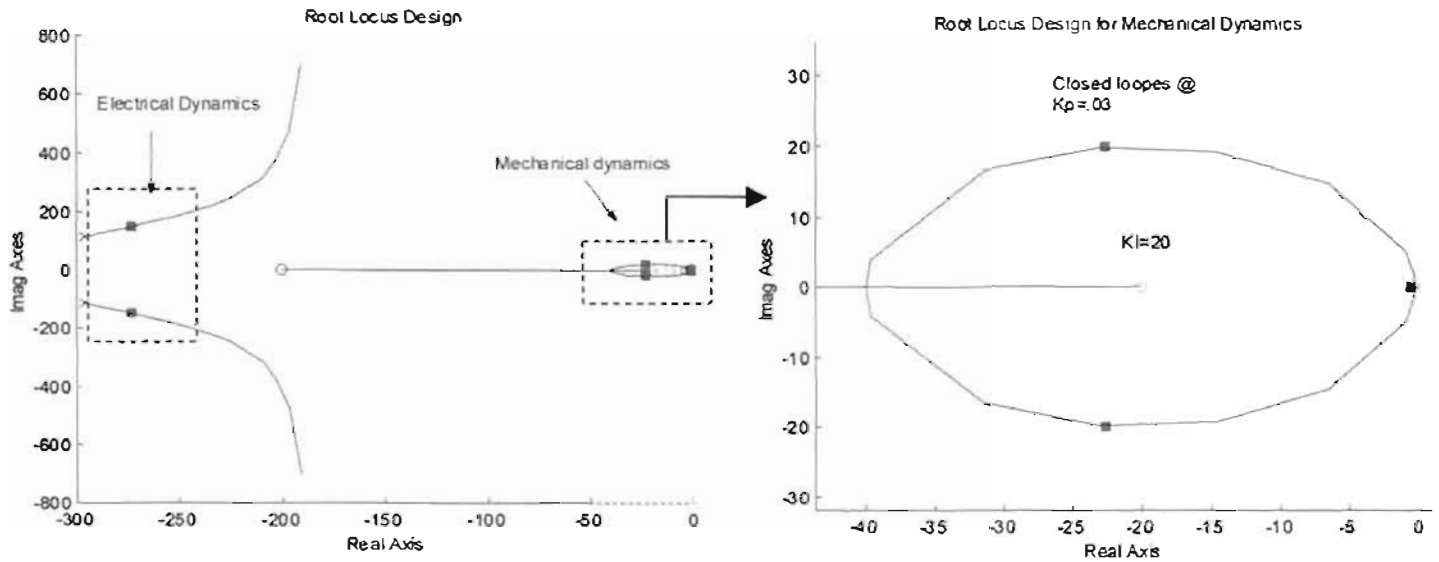


Fig. 7.14: Root locus of system and zoomed mechanical dynamics

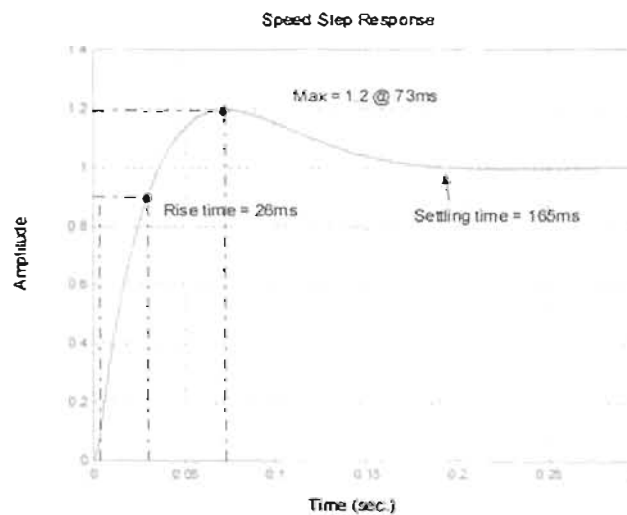


Fig. 7.15: Speed Step Response

The Simulink model used to simulate the speed loop is shown in Fig. 7.16. This model consists of a discrete controller for both the speed loops and uses a fixed integration step of $125\mu s$. The speed controller is limited to $\pm 10A$ and the current controller is limited to $\pm 128V$. The simulated responses are shown in Fig. 7.17. Both the small signal (± 300 rev/min) and large signal (± 800 rev/min) responses are shown. The smaller signal is used to demonstrate the system in a linear region of operation while the large response shows the non-linear operation of the system. These results will be used to evaluate the real-time results produced by the RADE systems.

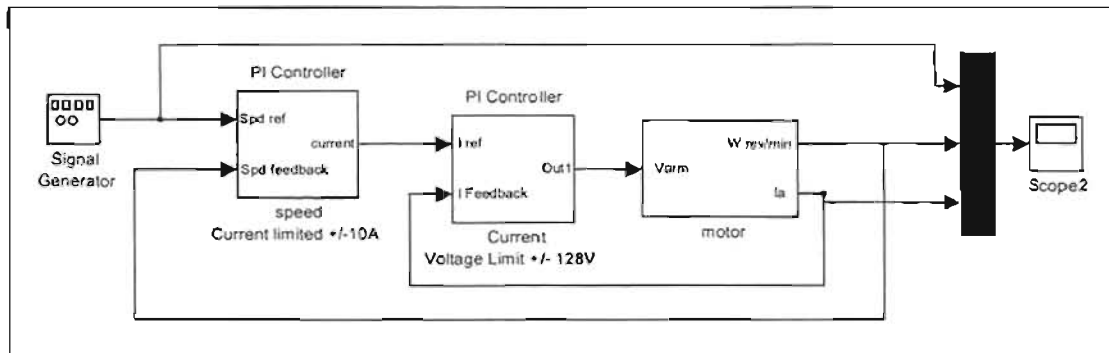


Fig. 7.16: Cascaded speed and current PI loops

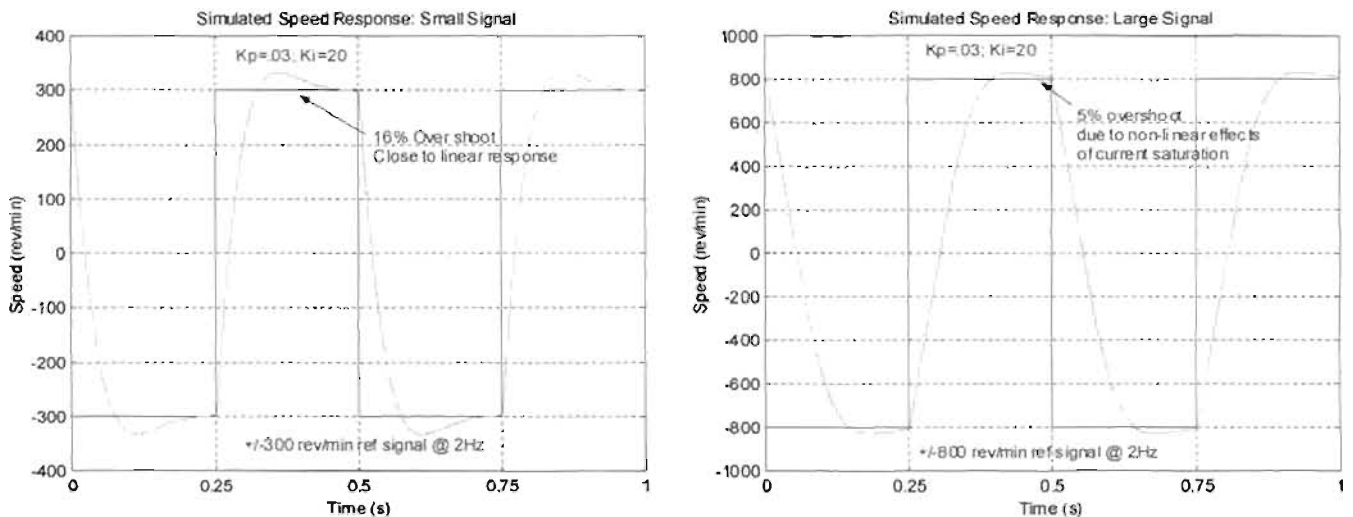


Fig. 7.17: Simulated results with discrete controller

7.3. Demonstration of the RADE ADC64 System

This section presents the rapid-prototyping of the current and speed controller, designed in the last section, on the RADE ADC64 system. It demonstrates how the RADE system allows students to interactively investigate the range of real-time plant behaviour simulated in the previous section. The practical use of the on-line parameter tuning and data logging features of the RADE ADC64 system are also demonstrated.

The first part of this section deals with the experimental setup used to rapid prototype the speed and current controllers. Thereafter the real-time results for the current and speed controllers are presented and evaluated against simulated results from the previous section. Finally this section concludes with a demonstration of saturation and the effect of integrator windup on the speed loop.

7.3.1 Real-Time Prototyping with the RADE ADC64

The experimental setup for demonstrating the real-time rapid prototyping of the current controller is shown in Fig. 7.18. It consists of:

- The Simulink PC.
- The Target PC, which contains the ADC64 DSP and PWM cards. The PWM card provides the 4K Hz switching signals to the inverter and an 8K Hz interrupt signal that is used to synchronise the ADC sampling.
- An H-bridge power inverter, which is connected to 128V DC battery supply.
- An analog tacho and LEM module current provide motor feedback.

The Simulink PC is used for both the simulation and rapid prototyping stages of the controller development. Once the current controller has been designed in Simulink it is converted into a target application via the RTW and downloaded to the target PC. The target PC, which is executing the server application, receives the target application and stores it. Simulink is then used in external mode to control target execution.

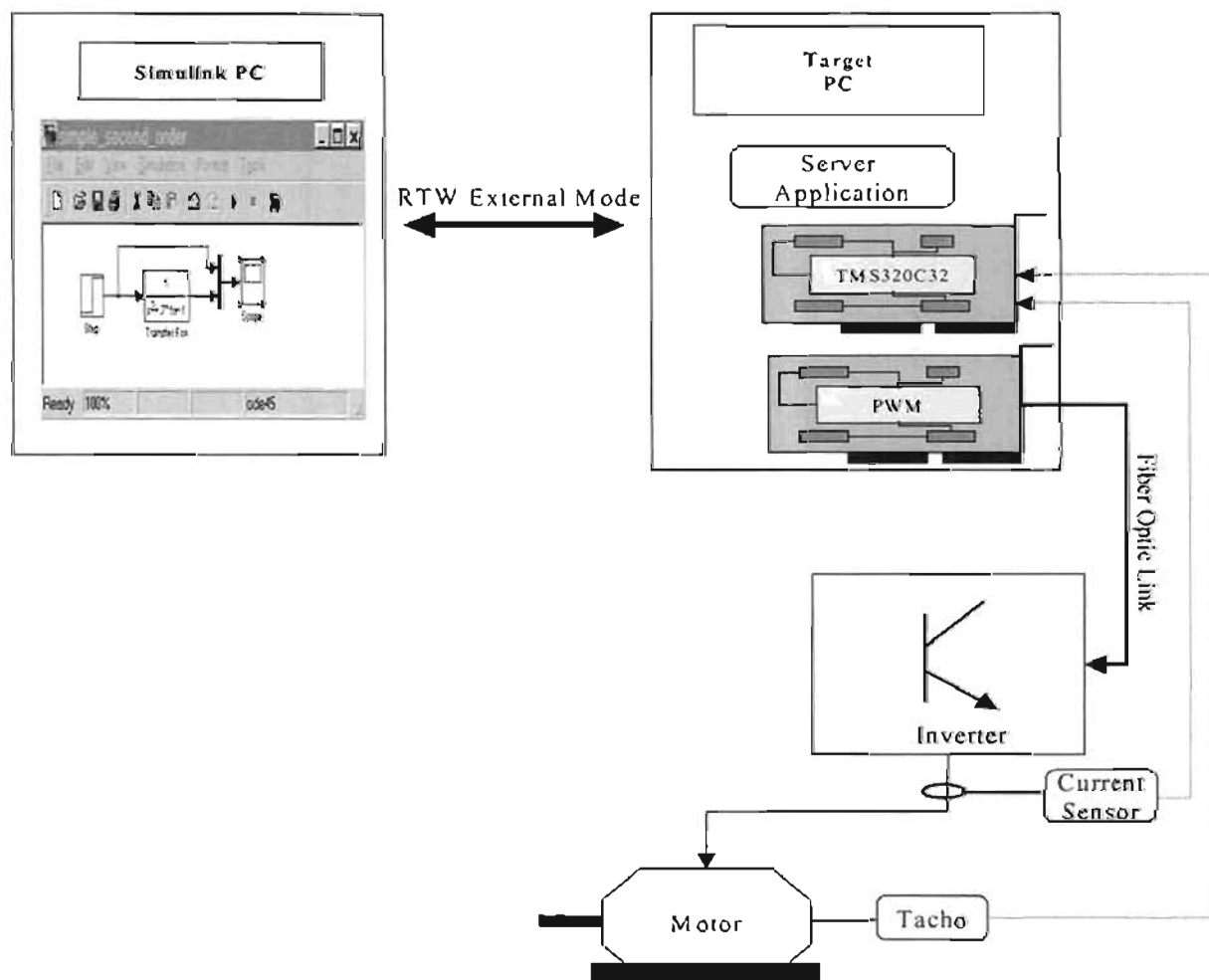
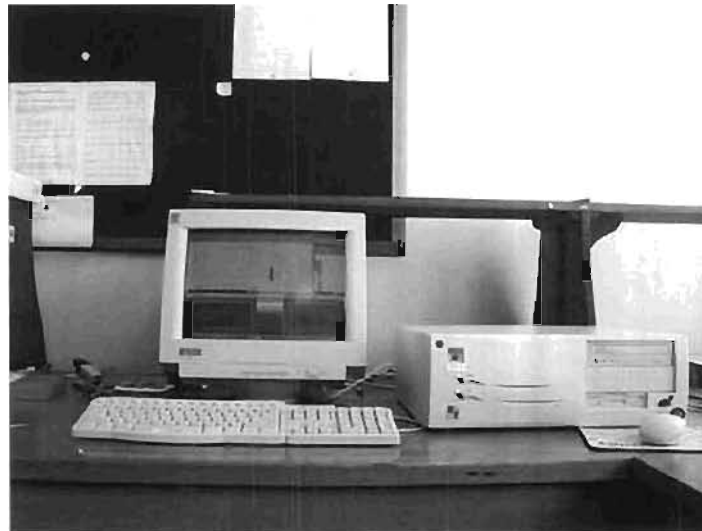


Fig. 7.18: Diagram of experimental setup

The current controller example as well as the following speed controller example both used the same

apparatus and also exploit the network functionality of the RADE ADC64 system. Fig. 7.19 shows the independent Simulink and target PC's and for a matter of interest, both these PC's were operated in two separate rooms.

Simulink PC



Target PC and Motor

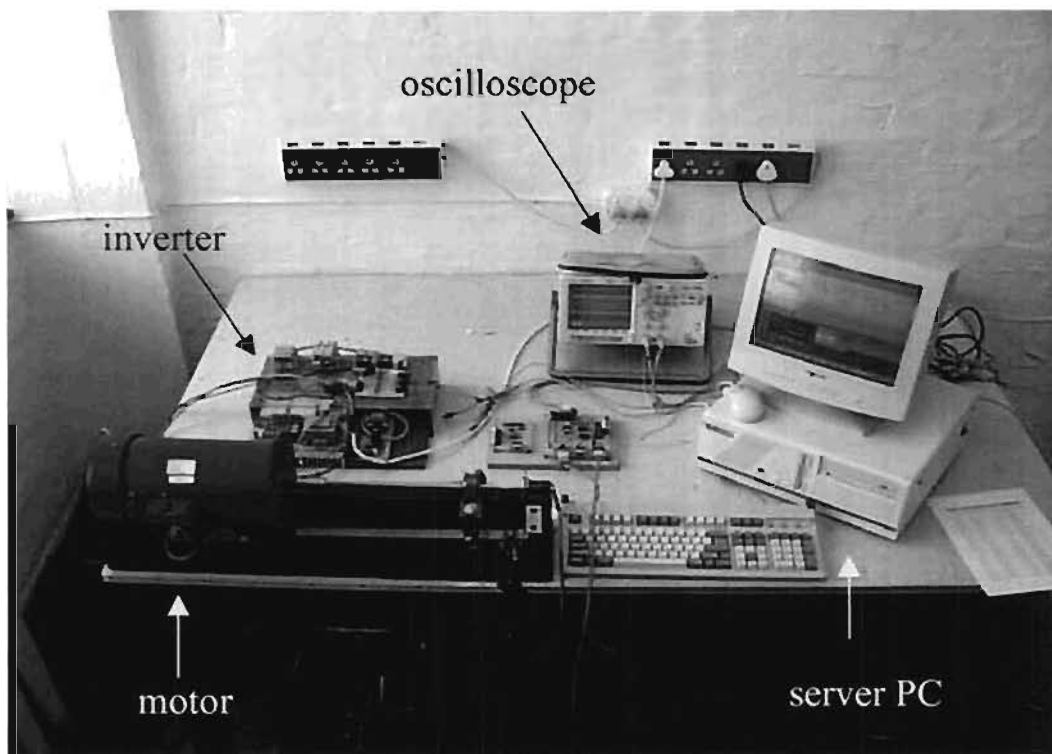


Fig. 7.19: Photo of target PC

7.3.2 DC Current Controller

Before the current controller shown Fig. 7.8 can be rapid-prototyped a few modification are needed, these include:

1. Inclusion of ADC devices which sample plant sensor data.
2. The addition of the PWM block, which controls the inverter.
3. The Interrupt Block is also used to provide synchronous sampling
4. The plant from the original model is removed as the live plant is controlled in the rapid prototyping case.

Fig. 7.20 shows the revised Simulink model, which is ready for rapid-prototyping and shall be referred to as the real-time model. While this model looks considerably different from the original model in Fig. 7.8 it is in essence functionally identical. The interrupt block is added to allow the controller in subsystem 1 to be synchronised with the PWM signals i.e. the PWM ASIC provides a 8K Hz ADC trigger signal and the ADC in turn trips the external interrupt 2 at the end of the conversion. With each end conversion signal, subsystem 1 is executed, which ensures the controller is executed at 8K Hz. The ADC64 Ext Timer block is used to set the external timers on the ADC64 board to 0 Hz as to prevent them from triggering the ADC's.

Subsystem 1 is a triggered system and is restricted by The Mathworks from containing continuous states i.e. all controller used in subsystem 1 have to be discrete and this is the reason for simulating the discrete controller in section 7.2.3. The controller is contained in the subsystem block, shown in Fig. 7.20, consists of:

- ADC 0,1 which is used to sample the current sensor. This value is then scaled by 13/6 A/V, which transforms the ADC voltage signal into a current value.
- The PWM block is used to write PWM setting to the PWM ASIC.
- The sine wave and relay block⁷ combination provide the 10 Hz square wave signal, as the function generator block cannot be used in real-time models. (A 10 Hz +/-8A square wave reference signal is used for the evaluation of the current controller)
- The PI current controller block subsystem implements the discrete controller shown in Fig. 7.10.

⁷ The positive and negative threshold points for the relay block are respectively set at $\pm 0.1 \times 10^{-3}$

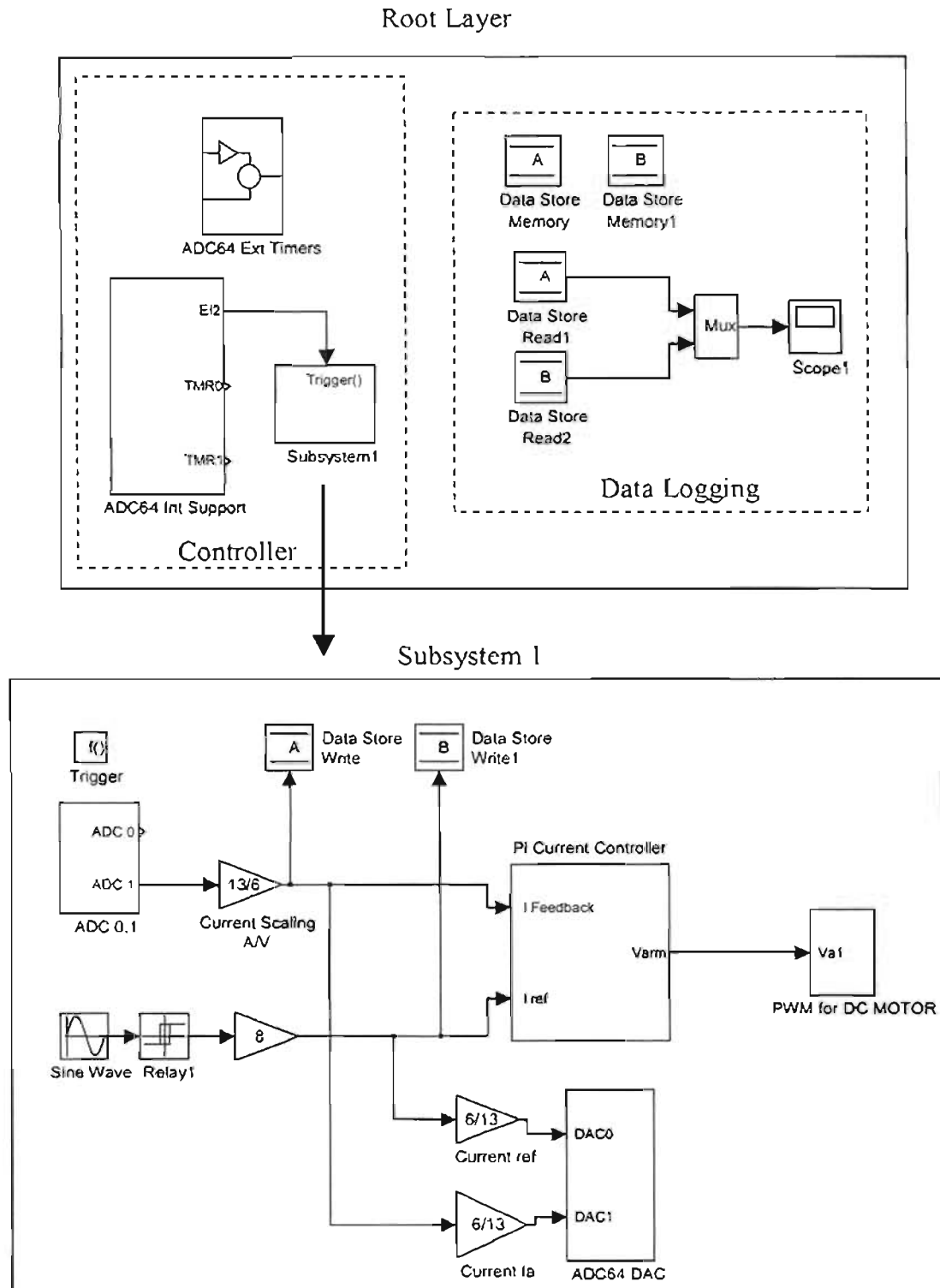


Fig. 7.20: Simulink model for the current controller

The Data Store, Read and Write blocks, shown in Fig. 7.20, are used to pass data between the subsystem1 block and the root layer. These blocks are needed to supply data to the scope blocks, which are restricted from being placed in trigger subsystems[§]. The scope blocks are used to provide on-line data visualisation and data logging. The scope block shown in Fig. 7.22 highlights this

[§] This restriction is imposed by The Mathworks.

functionality.

The proceeding section presents the results of the real-time current controller. The controller is investigated in three regions of operation that were designed and simulated in section 7.2.2:

1. Designed behaviour.
2. Under damped behaviour
3. Over damped behaviour

Students would use the on-line parameter-tuning feature of the RADE ADC64 system to change controller parameters on the target system, which allows changes in plant behaviour to be immediately observed. Fig. 7.21 illustrates how a controller parameter can be changed on-line. By double clicking on the Kp block, its gain can be manually adjusted which results in the seamless change to the equivalent parameter on the real-time target system. This allows parameters to be easily changed without interrupting the execution of the target system.

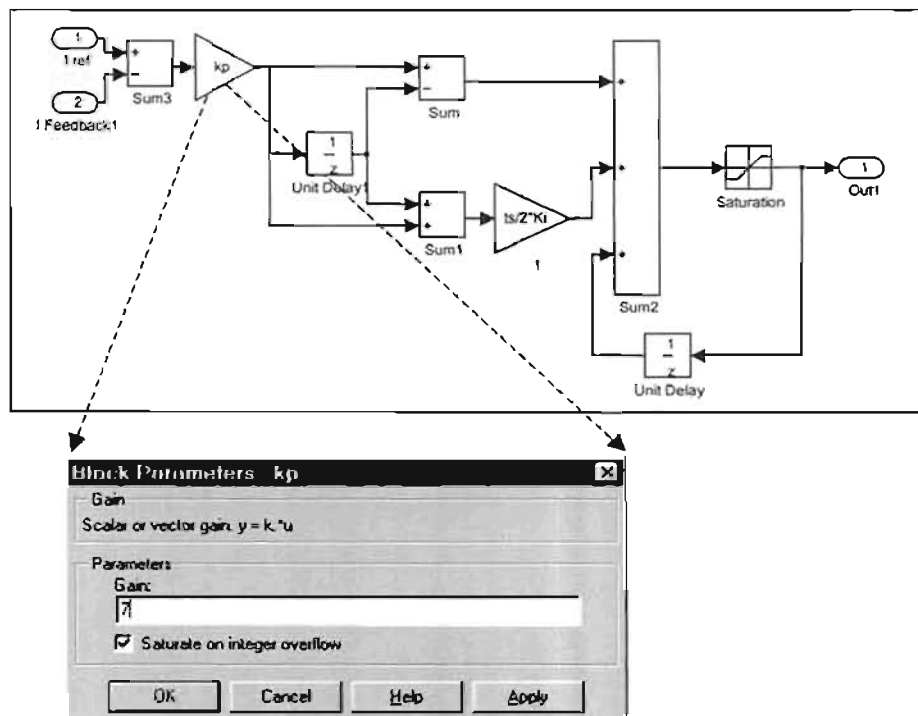


Fig. 7.21: On-line parameter tuning

The results presented in this section have been logged using the scope block as alluded to earlier in this section. This data is passed to the Matlab workspace and is presented in one of two ways, either by directly using a bit map image of the scope block (Fig. 7.22) taken during simulation execution (on-line visualisation) or, by using the Matlab *plot* command (Fig. 7.24) on data passed to the workspace (batched data visualisation). The scope block image method is used to demonstrate the system from a user perspective while the plot method is favoured when data analysis is needed⁹.

⁹ The *plot* function allows for more flexibility and produces vector graphic images that have better resolution for

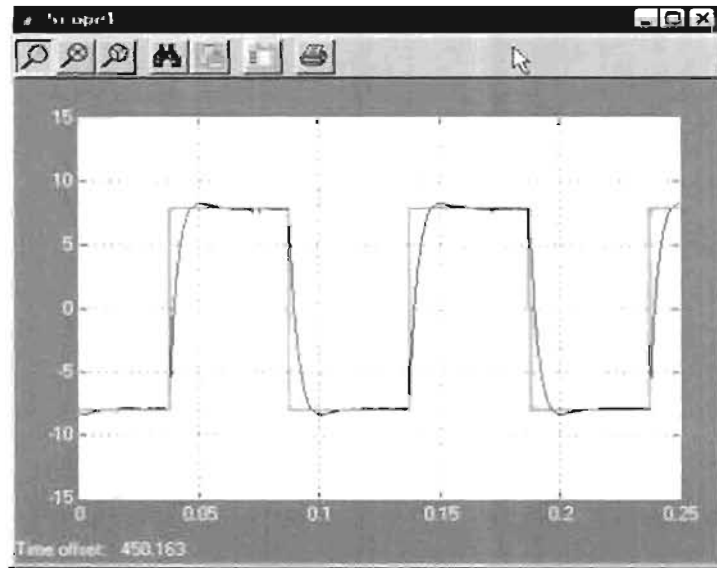


Fig. 7.22: Scope Block Results for $K_p=14.5$, $K_i=200$

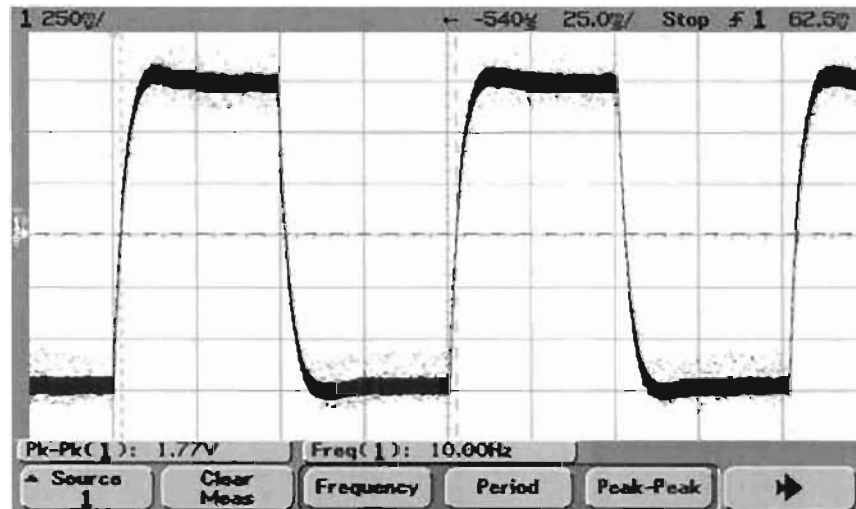


Fig. 7.23: Digital scope results (current probe scaling 100mV/A)

The first test of the real-time current controller will use the controller values designed in section 7.2.2 i.e. $K_p=14.5$, $K_i=200$. The real-time scope block results shown in Fig. 7.22 are verified against both simulation results from Fig. 7.11 and data captured by an external digital scope, Fig. 7.23. The data from the external digital scope is scaled¹⁰ by 10A/V and superimposed on the real-time result captured by the scope block and is shown in Fig. 7.24. From Fig. 7.24 it is evident that the results measured by the Simulink scope block and external digital scope agree with the exception of switching noise. This is an expected discrepancy as the Simulink scope uses data samples that are synchronised to the inverter while the external digital scope does not.

incorporation into printed documents.

¹⁰ The current probe scales measured current by 100mV/A. The measure voltage signal has to be multiplied by 10A/V to get the absolute current measured.

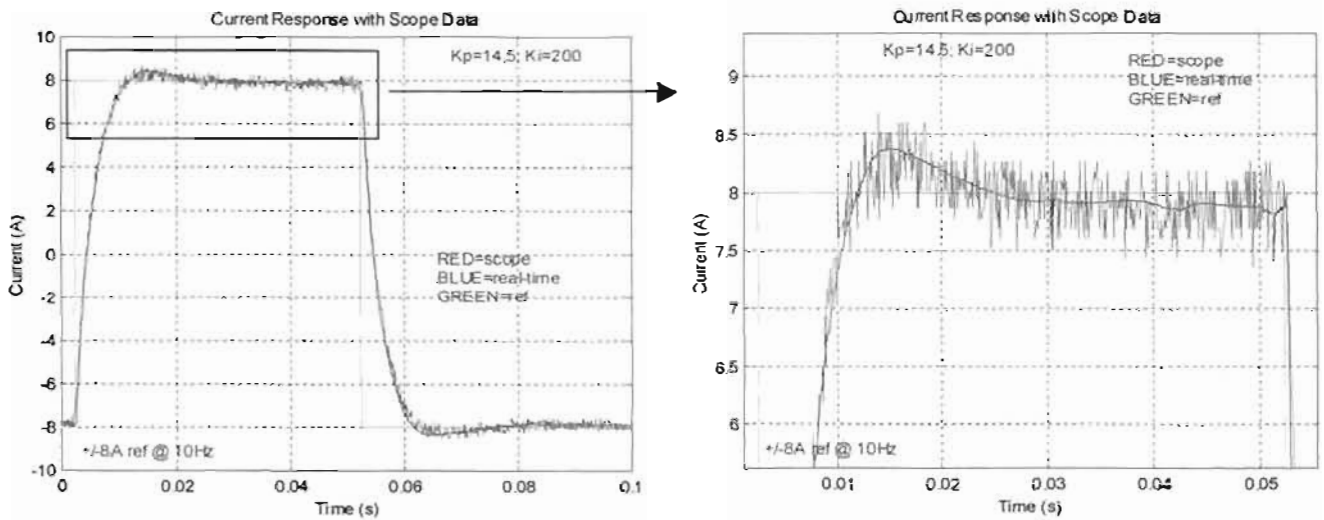


Fig. 7.24: Comparison of results (scope data has been scaled by 10 A/V)

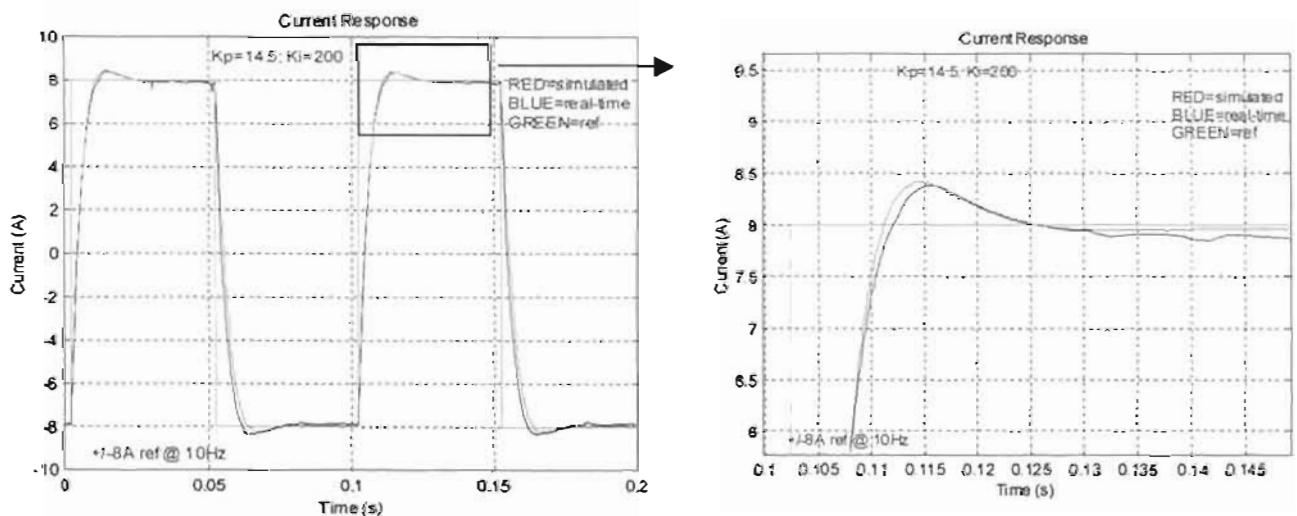


Fig. 7.25: Comparison of real-time and simulation results

The graphs of simulation and real-time data are superimposed in Fig. 7.25 and it is evident that there is good correlation between the two systems. This figure shows that Simulink simulation can be easily rapid prototyped to practically observe real-time control of systems.

For the under damped case $K_p=5$ and $K_i=1000$, the corresponding plant response compared to the simulation response is shown in Fig. 7.26. From this figure it is evident that simulated behaviour and real-time results agree, aside for some switching noise on the real-time system¹¹.

¹¹ Though synchronise sampling is used, switch noise cannot entirely be removed.

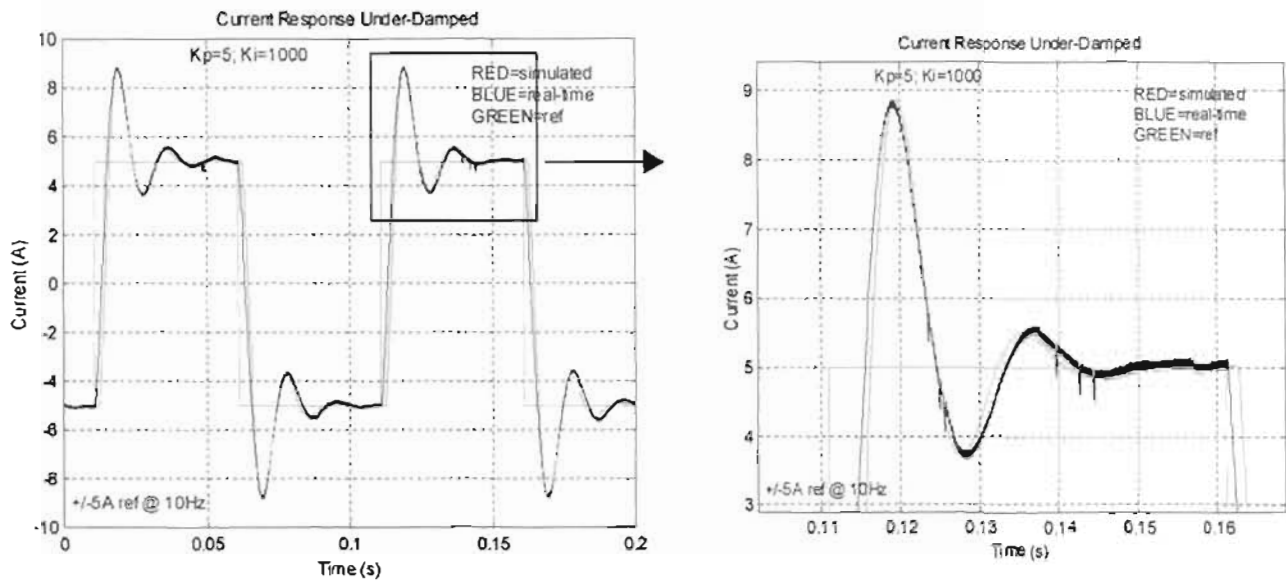


Fig. 7.26: Under damped current controller response

The next part of the investigation involves the changing of controller parameters to observe an over damped plant response. The parameters used are $K_p=6$ and $K_i=100$. The real-time and simulated responses are compared in Fig. 7.27, which also demonstrate good correlation between the systems.

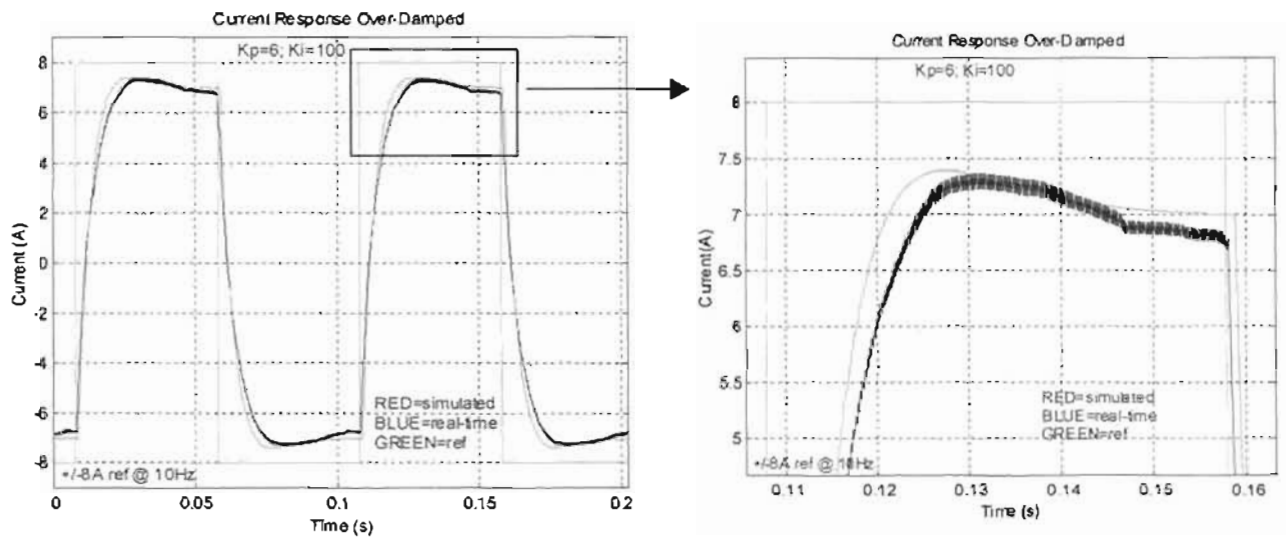


Fig. 7.27: Over damped current controller response

This section demonstrated that the implementation of a current controller with the RADE ADC64 system produces real-time results that agree closely with simulated results. The next section introduces the rapid prototyping of the speed loop.

7.3.3 DC Speed Controller

In this section a speed loop is added to the current controller loop implemented in the previous section. The Simulink model used for the current controller (Fig. 7.20) does not change aside for the subsystem 1 block with the modified block shown in Fig. 7.28 and new controller subsystem block shown in Fig. 7.29. The changes made include the addition of:

- The ADC4,5 block. Used to sample the speed data from the tacho sensor. This signal is then scaled by $1000\text{rpm}/2.6\text{V}$ to convert the voltage signal into a speed signal.
- The sine wave and relay block¹² combination provide the 2 Hz reference square wave signal, as the function generator block cannot be used in real-time models.
- The data store blocks A, B and C respectively, pass current, speed and reference signal data back to the root layer for on-line visualisation.
- The controllers subsystem contains the cascaded speed and current controller and is shown in Fig. 7.29. This block consists of two discrete PI controllers, Speed controller and Current controller with internal models shown in Fig. 7.10.

The experimental setup (Fig. 7.18) does not change and the tacho sensor provides speed feedback. The PWM switching frequency remains at 4K Hz with an 8K Hz sampling time.

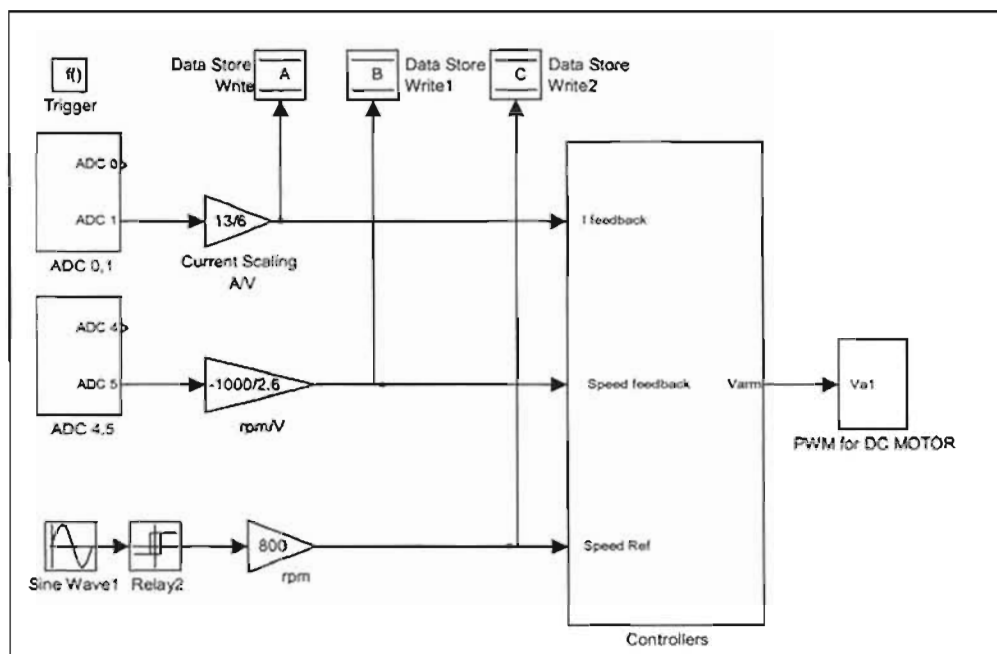


Fig. 7.28: Subsystem model for speed controller

¹² The positive and negative threshold points for the relay block are respectively set at $\pm 0.1 \times 10^{-3}$

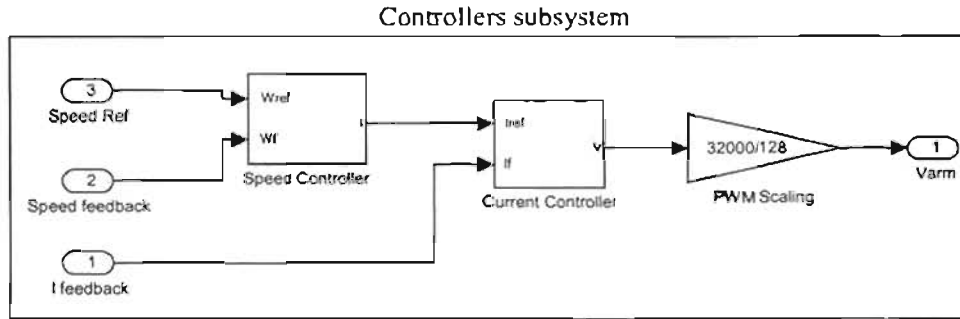


Fig. 7.29: Controllers subsystem block

The controller parameters used for the speed loop are $K_p=0.03$ and $K_i=20$. The plant's small signal speed response compared with simulated data is shown in Fig. 7.30. This figure shows that the real-time results agree with simulated data, aside for speed jitter of the shaft. This resulted from the inverter switching noise and its effect is more conspicuous at low speeds as the tacho has a low output voltage. For example at 300rpm the tacho voltage is 0.78V and with 70mV switching noise observed on this signal this translates to a 27 rpm error which can be seen in Fig. 7.30. The corresponding current waveform compared with simulated data is shown in Fig. 7.31. Here again results agree if the effect of noise is ignored it is also evident that the current controller is not held in saturation which allows the system to operate in a linear region.

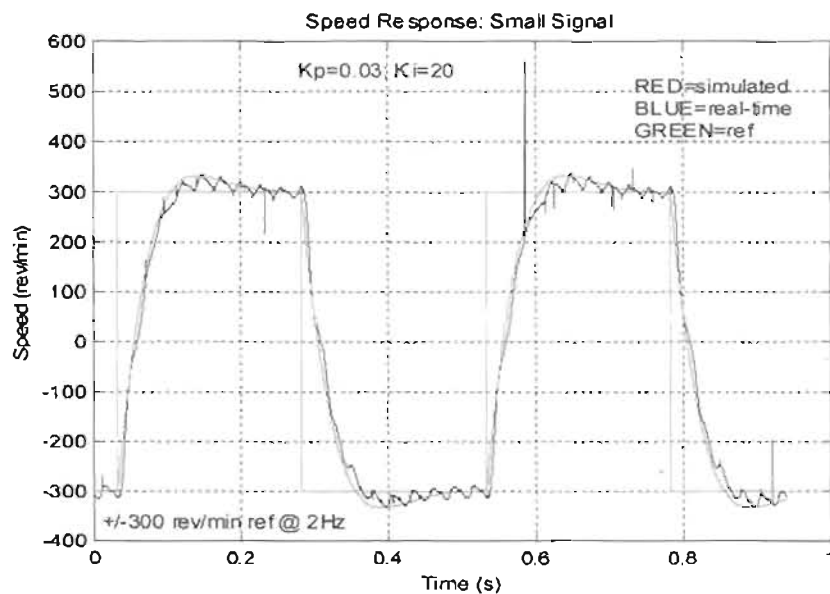


Fig. 7.30: Small signal speed response

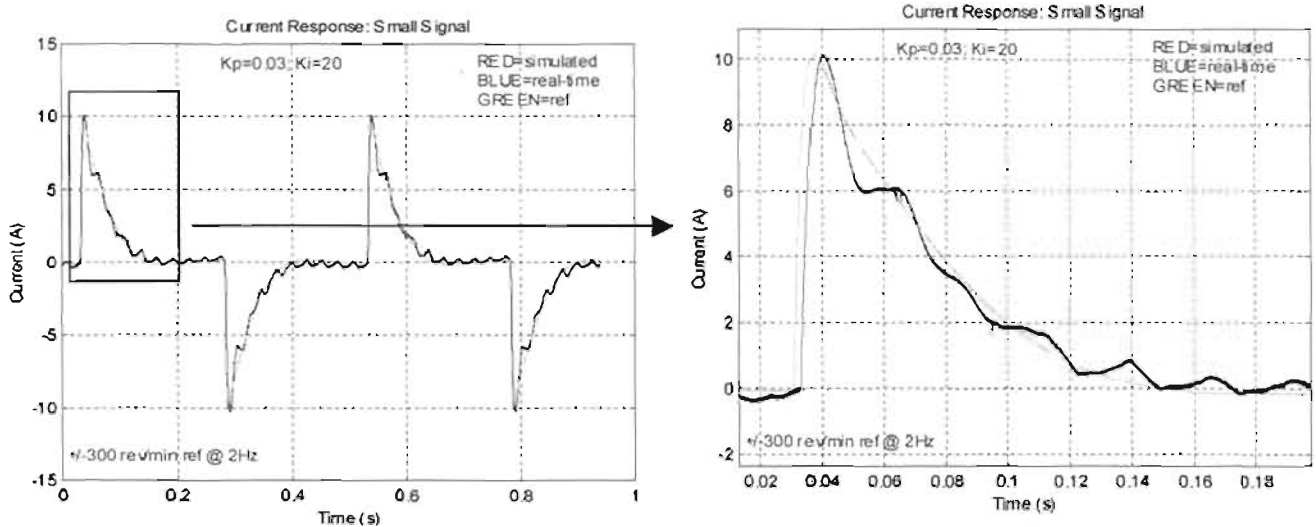


Fig. 7.31: Small signal current response

With the controller running the reference signal is changed to ± 800 rpm the resulting speed and current waveforms are shown in Fig. 7.32 and Fig. 7.33. Both these figures have simulated data overlaid, which confirm that the real-time and simulated results agree.

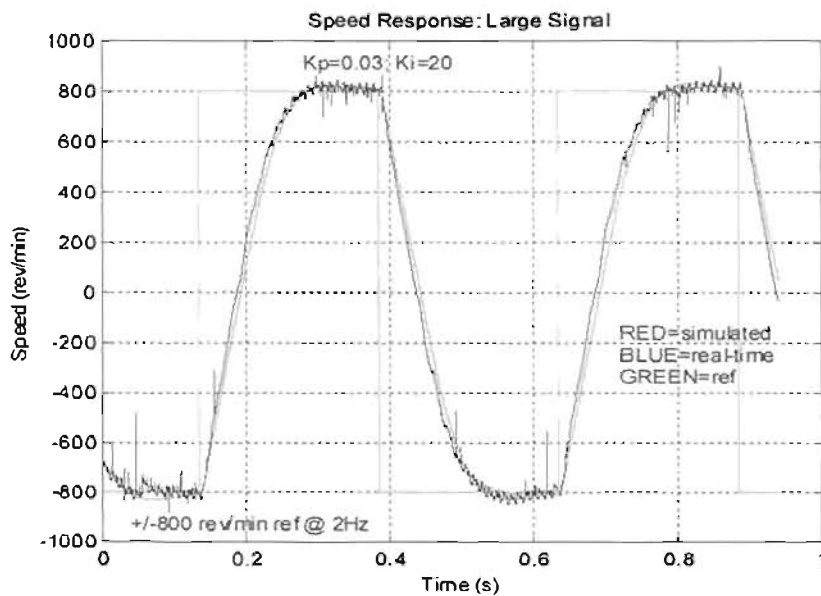


Fig. 7.32: Large signal speed response

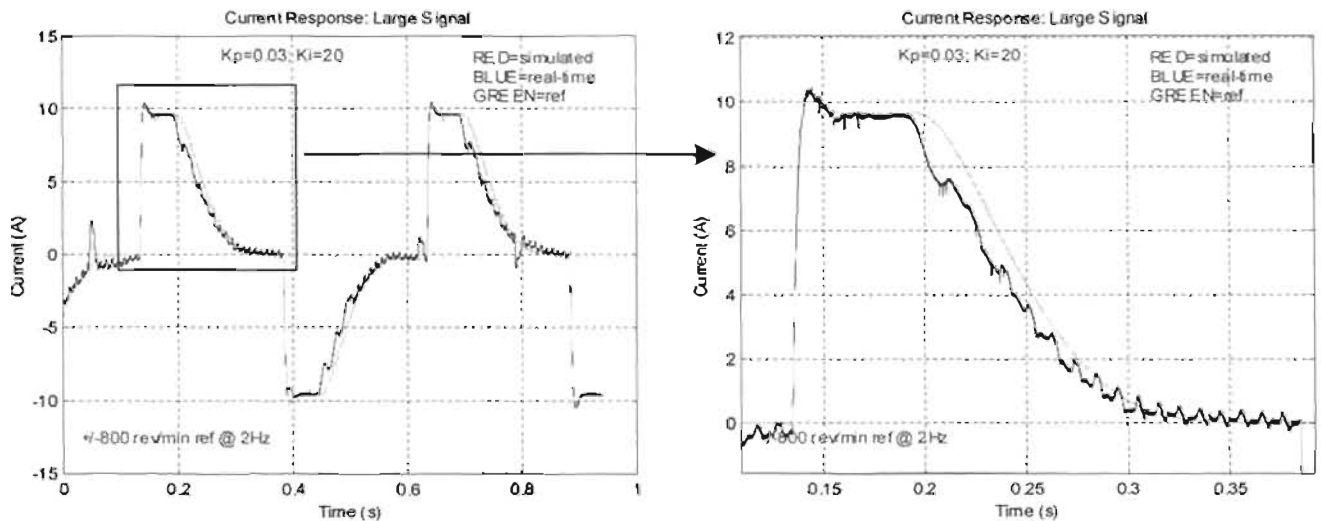


Fig. 7.33: Large signal current response

This part now demonstrates how the RADE system allows students to qualitatively investigate the effects of plant saturation and integrator windup. While the system is running the reference signal is changed from ± 800 rpm at 2 Hz to ± 2000 rpm at 1.2 Hz. The resulting speed response is shown in Fig. 7.34. This figure demonstrates how the motor can only rotate at a speed physically constrained by supply voltage i.e. 1500 rpm. This example while trivial, allows students to get a tangible understanding and visualisation of plant saturation.

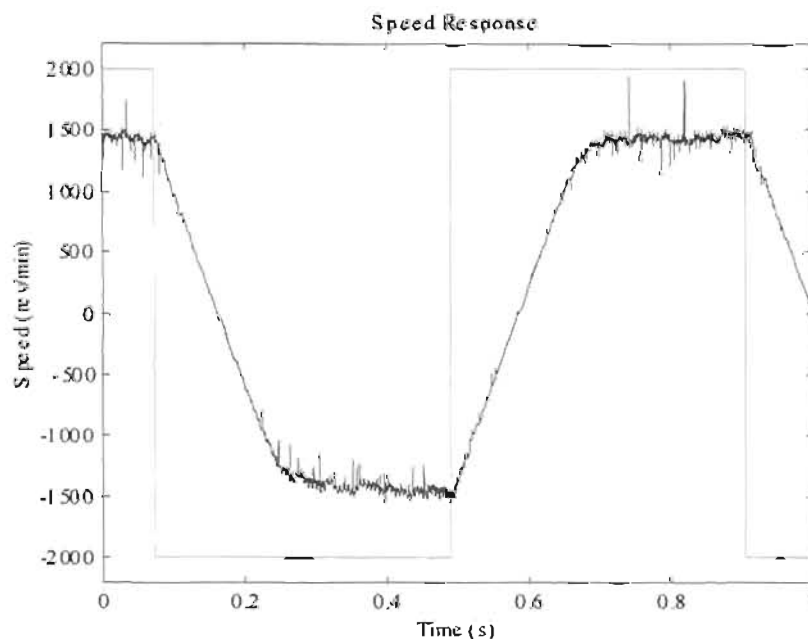


Fig. 7.34: Plant saturation speed response

The concept of integrator windup is closely related to plant saturation and can also be demonstrated to students. A “manual” switch block, shown in Fig. 7.35, is inserted into the speed controller loop to switch in or out the integrator windup.

It works on the principle of changing the input to the output delay term of the controller. If this input is taken before the saturation block, the delay term or integrator is able to windup, and conversely if taken after the saturation block, the maximum value attainable by the integrator is constrained to the saturation block limits. It is worthwhile to note that the manual switch position can be changed during real-time code execution. When the integrator windup has been disabled the resulting speed response is shown in Fig. 7.36. The controller parameters are unchanged from the ones used for the response in Fig. 7.32 i.e. $K_p=0.03$ and $K_i=20$. From Fig. 7.36 it can be seen how integrator windup affects the predicted system responses. This now concludes the demonstration of the RADE ADC64 and the next section focuses on the RADE PC32

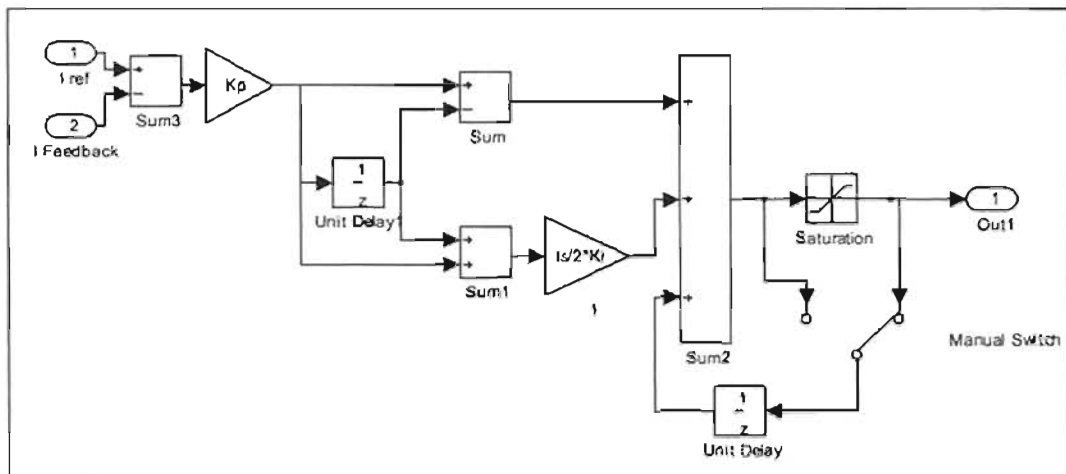


Fig. 7.35: Discrete PI controller with windup switch

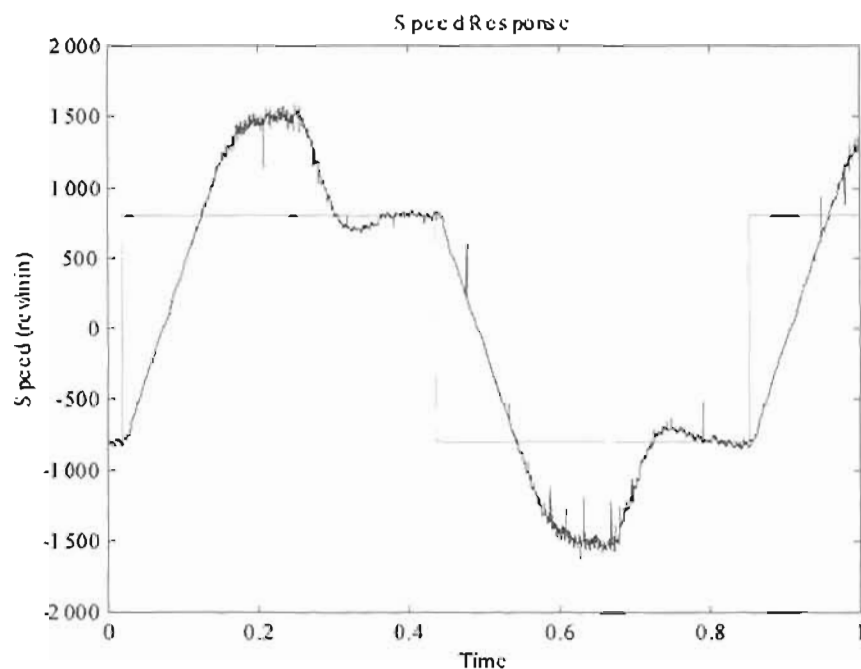


Fig. 7.36: Speed response with no anti-windup

7.4. Demonstration of RADE PC32 System

This section demonstrates rapid prototyping with the RADE PC32 system. The first part of the section rapidly prototypes the current and speed controllers designed in sections 7.2.2 and 7.2.4. These results are compared against both simulated and RADE ADC64 results. This section then concludes with a demonstration of a rapid prototyped position controller.

7.4.1 DC Servo Speed Control

The DC servo implemented in this section is based on the Simulink models used on the RADE ADC64 system. The tuned current and speed loop results presented in this section, are used to evaluate the RADE PC32 system against simulated and RADE ADC64 data.

The operation of the RADE PC32 system is functionally equivalent to RADE ADC64 and the similar Simulink model used on both these system. Fig. 7.38 and Fig. 7.28 respectively show the PC32 and ADC64 versions of the subsystem 1 block; only the device driver blocks are changed. This illustrates the flexibility of the RADE system to reuse models across different targets. The experimental setup, shown in Fig. 7.18, does not change except for the target PC using the PC32 card.

Fig. 7.37 shows the root layer of the DC servo speed controller model and consists of:

- AD Trigger block which is used to software trigger ADCs on external interrupt 0. The interrupt signal is generated by the PWM card and occurs at 8K Hz synchronised to the 4K Hz PWM signals.
- The subsystem 1 block contains the controller model (Fig. 7.38) and is synchronised to the ADC end conversion signal which trips interrupt 1. This is identical to ADC64 version (Fig. 7.28) except for the ADC and PWM device driver being changed.
- The PC32 Int Support block used to synchronise AD Trigger block to external interrupt 0 and the subsystem 1 to external interrupt 1
- The Data Store Memory, Data Store Read and scope blocks are used for on-line visualisation. Data Store Read block B and C provide speed data and A provides current data.

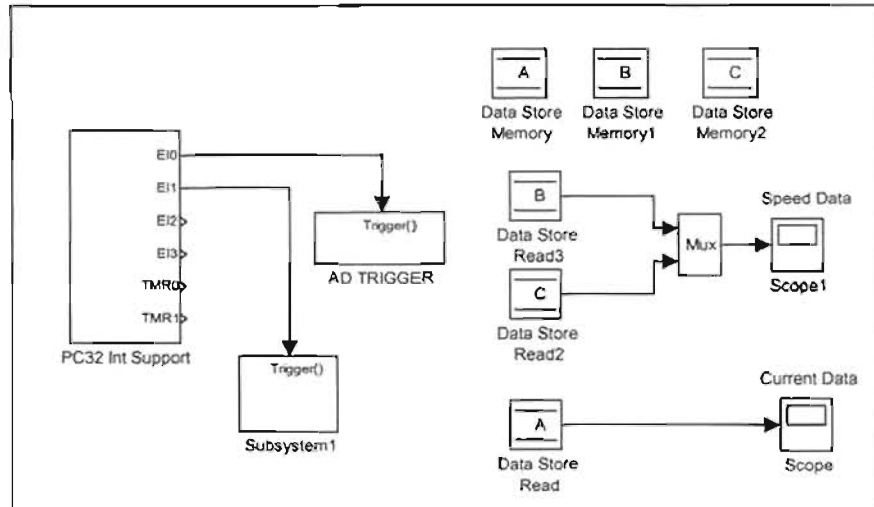


Fig. 7.37: Simulink model

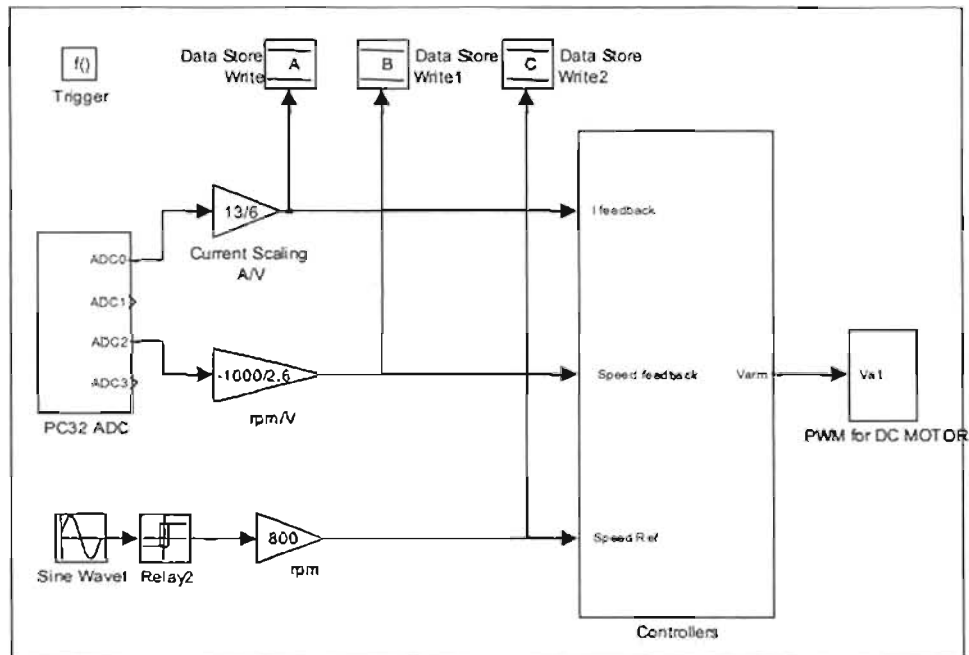


Fig. 7.38: Subsystem 1 block

The current loop response (without speed loop) is shown in Fig. 7.39 and it is evident that simulated and real-time results agree, aside for the switching noise. The zoomed responses in Fig. 7.39 and Fig. 7.40 show the ADC64 and PC32 results¹³. A visual comparison of these results reveals that the PC32 system is more susceptible to noise and this can be explained by analysing the ADC triggering used on these systems. On the ADC64 system the trigger signals are supplied directly by the PWM card and there will only be signal propagation delay. On the PC32 system the PWM signal can not be directly

¹³ These results have not been overlaid, as the switching noise makes it difficult to differentiate signals.

used to trigger ADC conversions¹⁴ and is instead patched to the processor external interrupt 0 pin which in turn trips software ADC conversions. In this case there is the interrupt latency and the processing delays, which skews sampling and results in more, switching noise being sampled.

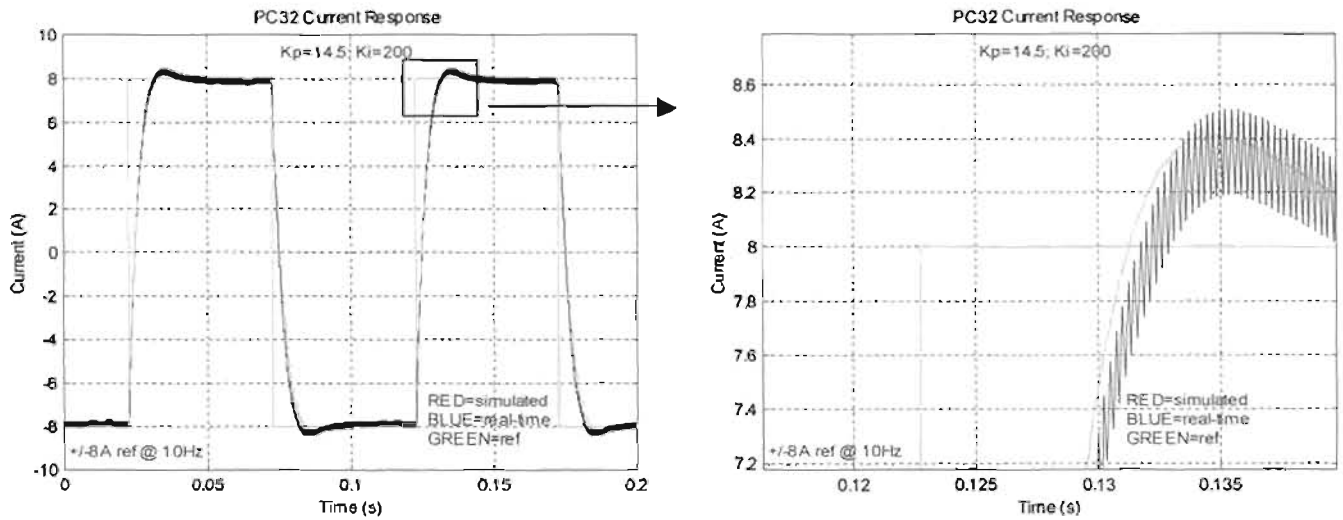


Fig. 7.39: Current response

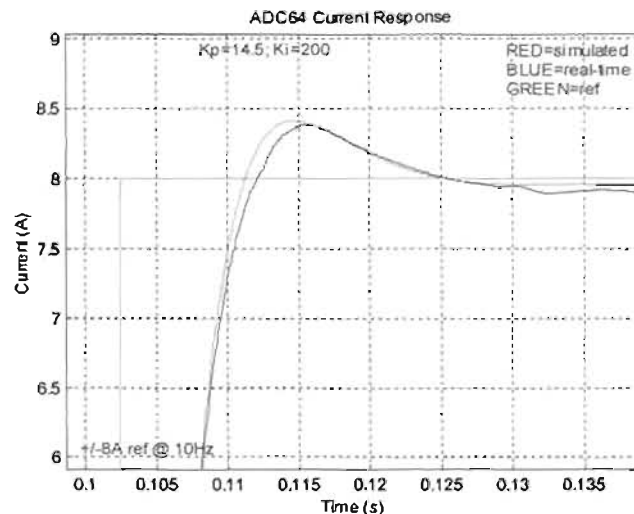


Fig. 7.40: ADC64 Current sampling

The speed loop small and large signal results are presented below and also confirm that the RADE PC32 system produces results that closely match simulated and ADC64 data. Table 7-2 summarises these results. The next section demonstrates the RADE PC32 system for a position controller.

¹⁴ The external trigger signals for the ADCs on the PC32 card have strict timing restriction that are not met by the interrupt signal generated from the PWM ASIC.

Result Type	Kp	Ki	Comment	Figure
Small signal speed	0.03	20	Simulated ADC64 results agree.	Fig. 7.41
Small signal current	0.03	20	Simulated ADC64 results agree	Fig. 7.42
Large signal speed	0.03	20	Simulated ADC64 results agree	Fig. 7.43
Large signal current	0.03	20	Simulated ADC64 results agree	Fig. 7.44

Table 7-2: Summary of RADE PC32 results

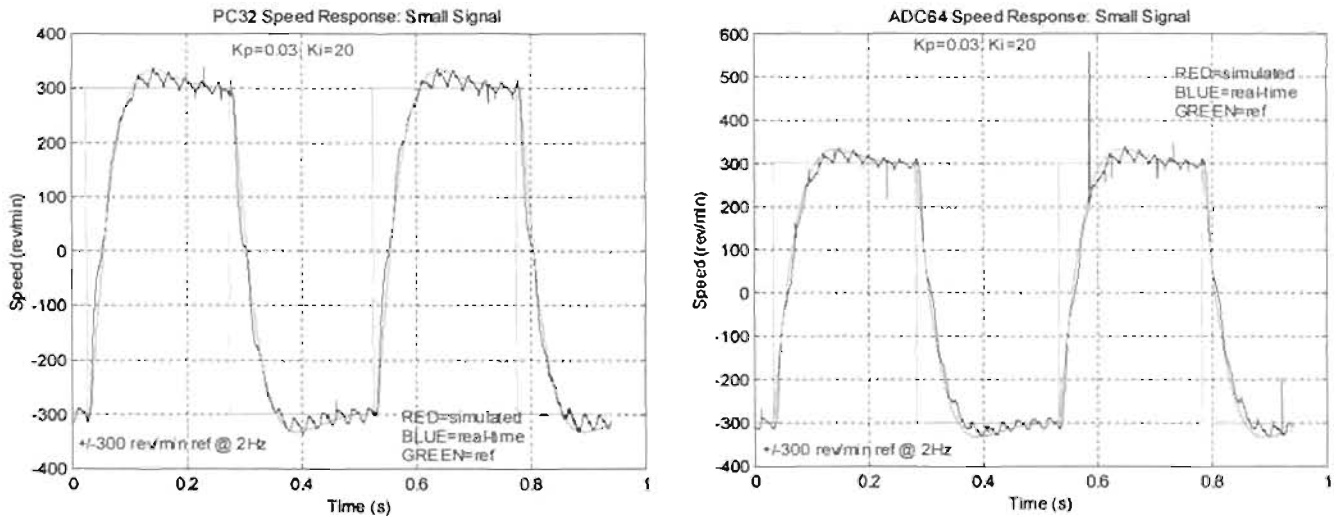


Fig. 7.41: Small signal speed response

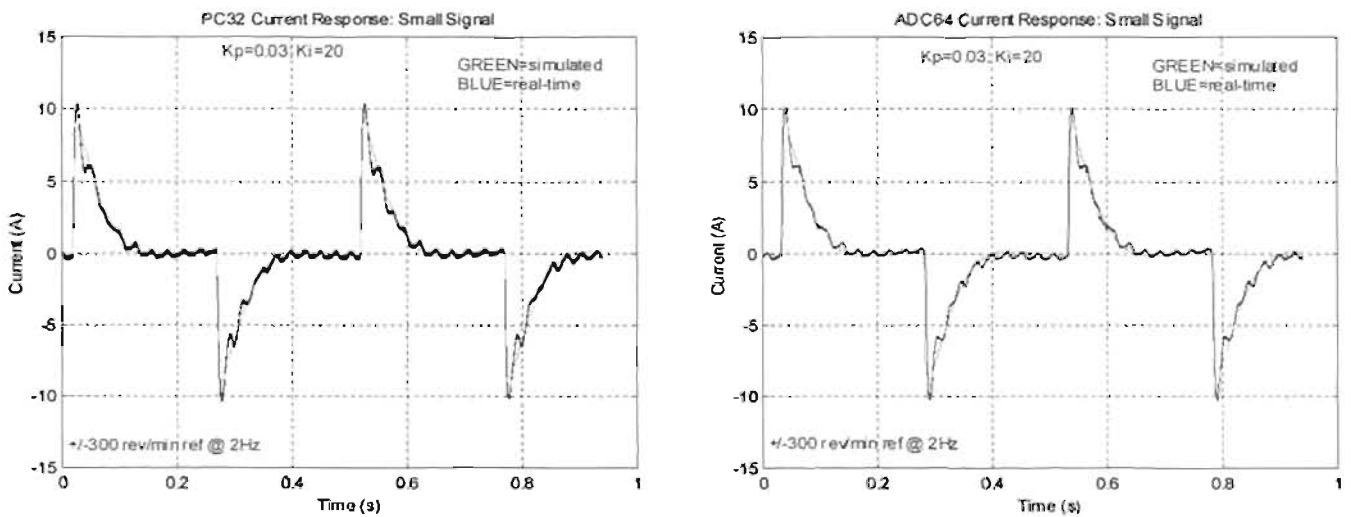


Fig. 7.42: Small signal current response

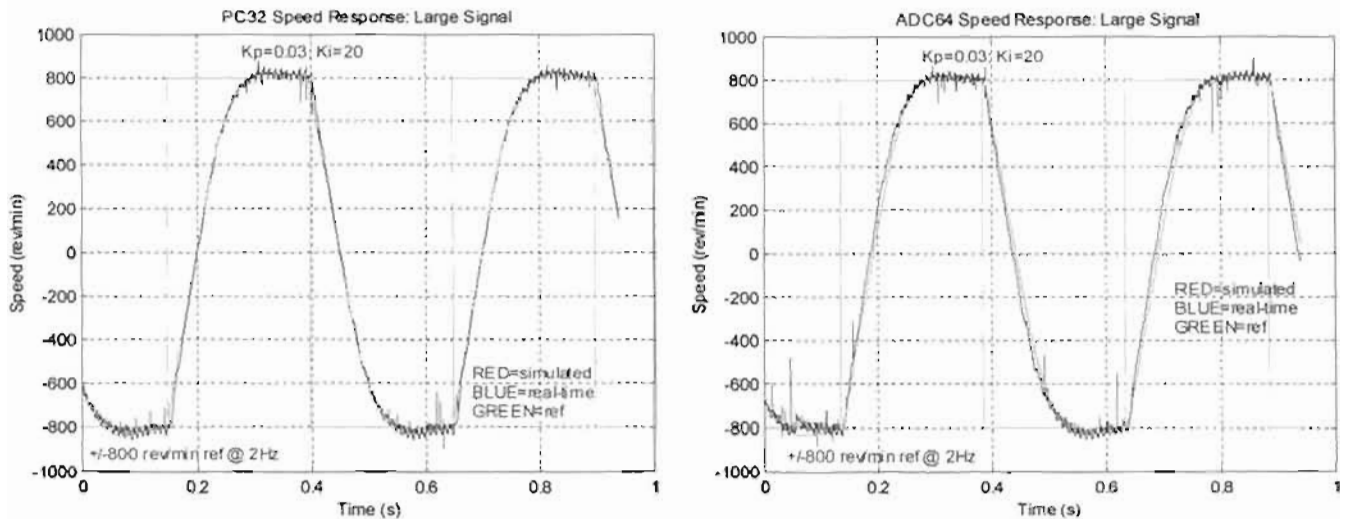


Fig. 7.43: Large signal speed response

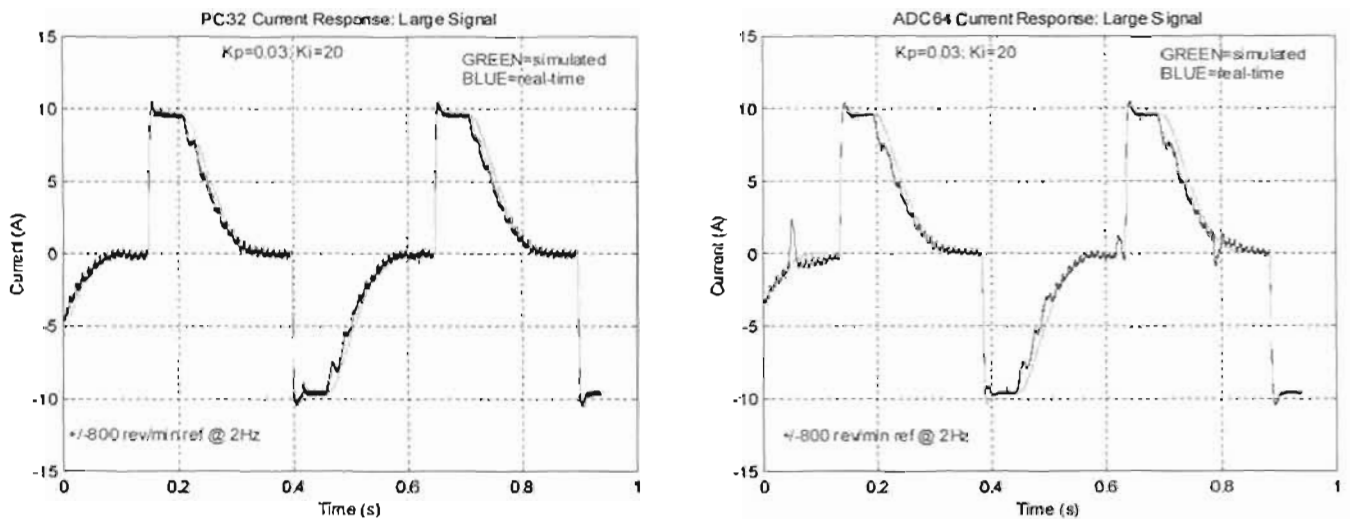


Fig. 7.44: Large signal current response

7.4.2 Position Controller Experiment

The RADE PC32 system, from an operational standpoint, functions in a similar fashion to the RADE AD64 system and this section illustrates its use with a position control experiment. This experiment is presented in a qualitative manner, as it is envisaged that students would use a supplied model to investigate plant responses. This experiment could therefore be used in introductory courses with students having little controls backgrounds. An added advantage of using a position control experiment is that changes in plant responses can be easily observed. For example in the under damped case the student can observe how the shaft overshoots the set point and then converges.

The experimental setup for this experiment is based on the system used for the RADE ADC64 system (Fig. 7.18) with a different motor and the tachometer replaced with an incremental encoder. The Simulink model used is shown in Fig. 7.45 and Fig. 7.46.

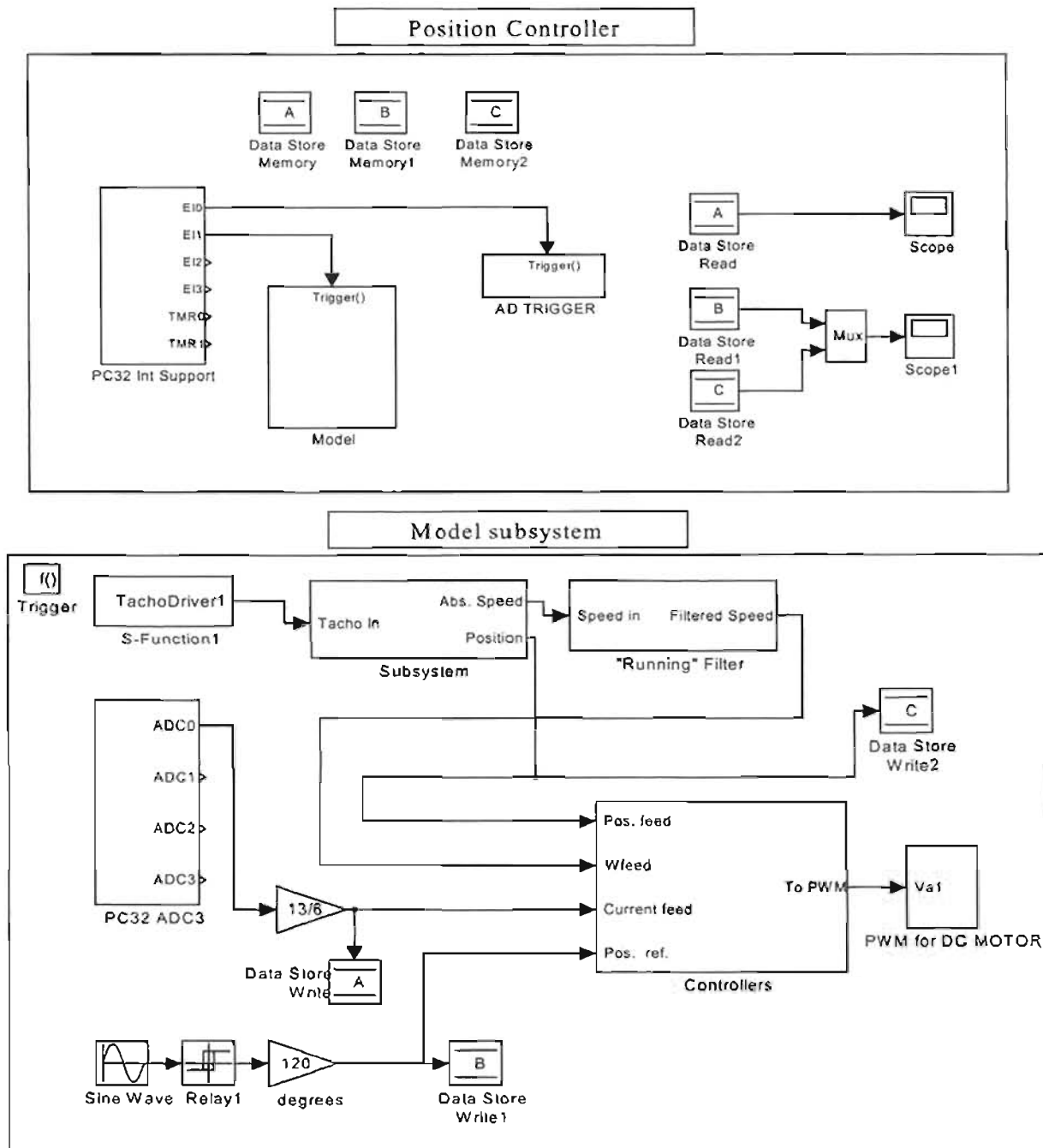


Fig. 7.45: Simulink model

The PC32 Interrupt block is used to synchronise controller execution to PWM ASIC as done with RADE ADC64 system. On the PC32 card the PWM ASIC's interrupt signal cannot be used directly to trigger ADC conversion¹⁵ and is instead used to trip external interrupt 1 that in turn triggers a software ADC conversion. At the end of the conversion the ADCs trip the external interrupt 2, which runs the model subsystem code.

¹⁵ The external trigger signals for the ADCs on the PC32 card have strict timing restriction that are not met by the interrupt signal generated from the PWM ASIC.

In the model subsystem, shown in Fig. 7.45, the ADC block is used to sample the current sensor signal, the Tacho driver is used to input position data from the tacho ASIC on the PWM card. The position data is then used to generate speed signal¹⁶, which used for speed feedback. The sine and relay combination are used to provide a reference square wave signal for the position control loop.

The controller subsystem, shown in Fig. 7.45 is expanded in Fig. 7.46. It consists of 3-cascaded discrete PI controllers. The inner most control loop regulates the armature current. This is followed by the speed loop and then the position loop. For the purpose of this discussion the inner two loops are assumed to be tuned and the student would only need to modify position loop parameters to observe different plant responses.

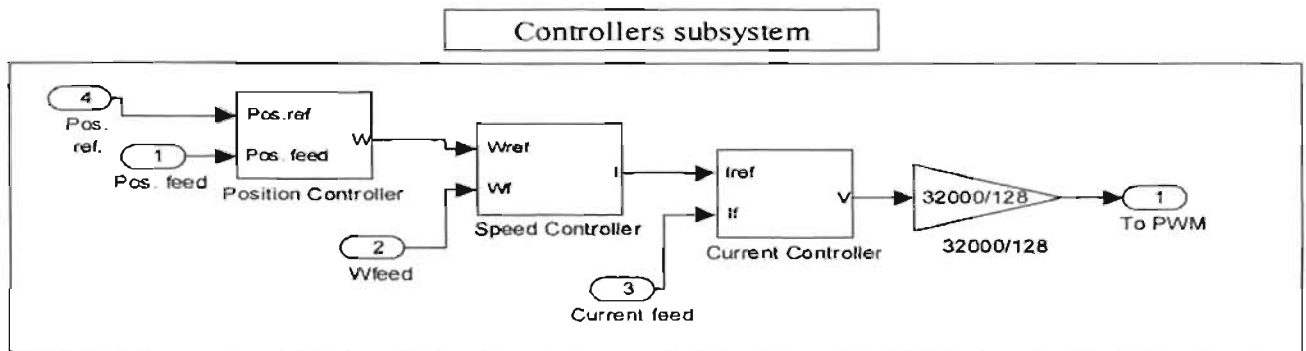


Fig. 7.46: The controllers subsystem

With the plant running, the student can enter different position controller parameters. Table 7-3 lists three examples of K_i and K_p parameters for the position control loop and their corresponding responses. These responses show a regulating, under damped and over damped plant/controller behaviour. The controller parameters for the speed and current loop are shown in Table 7-4, note these parameters are not modified.

Kp Position	Ki Position	Description	Figure
2	7	Good regulation	Fig. 7.47
2	30	Under damped plant response	Fig. 7.48
4	1	Over damped plant response	Fig. 7.49

Table 7-3: Controller parameters for various plant responses

Kp Speed	Ki Speed	Kp Current	Ki Current
45/30	20	30	300

Table 7-4: Fixed controller parameters for the speed and current loops

¹⁶ This is done by differentiating the position signal.

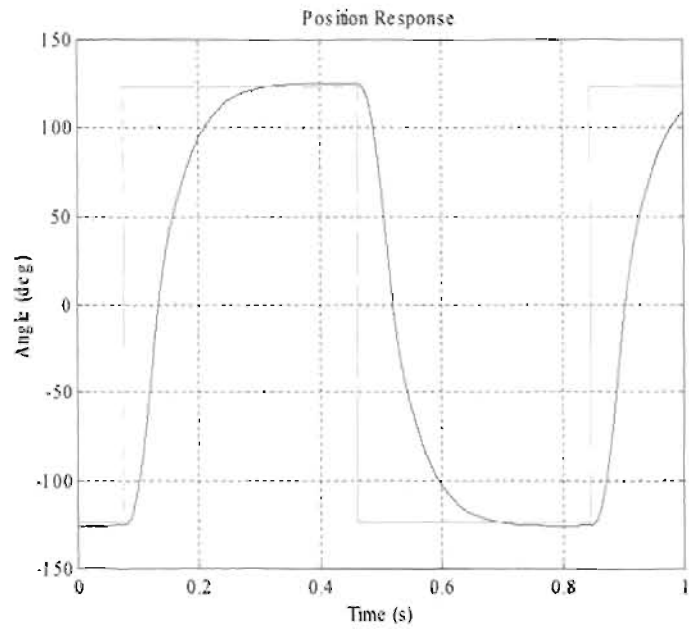
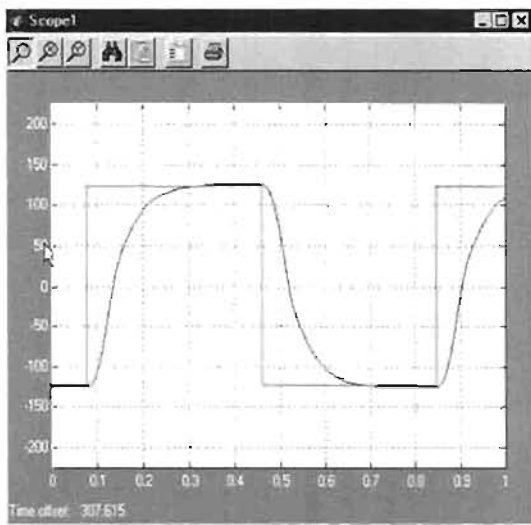


Fig. 7.47:Regulating position response

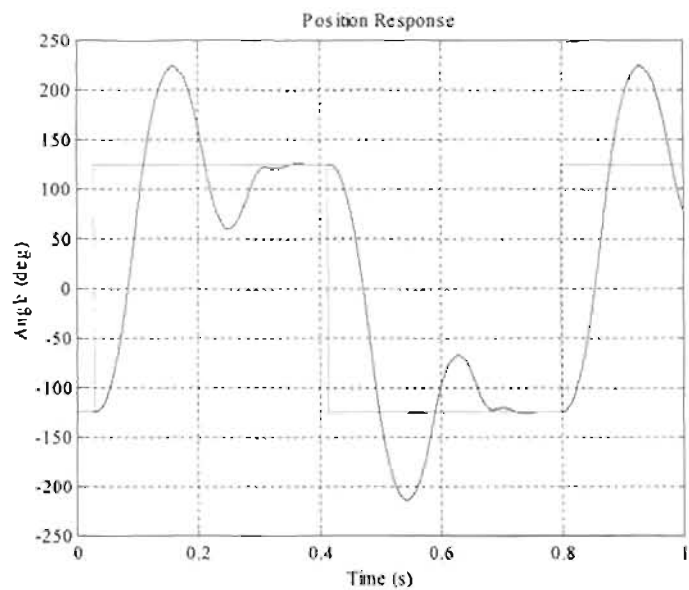
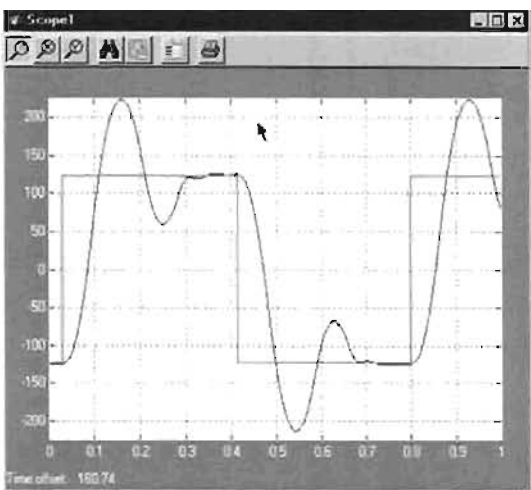


Fig. 7.48:Under damped response

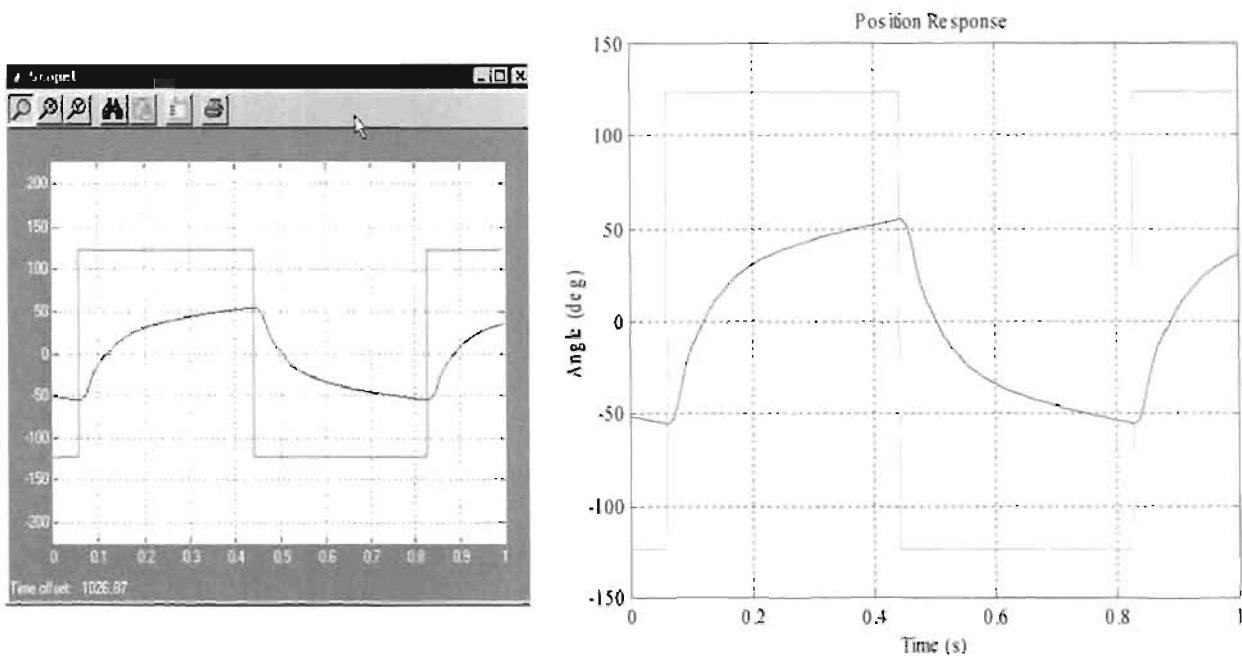


Fig. 7.49: Over damped response

7.5. Conclusion

This chapter demonstrated the use of the RADE ADC64 and PC32 systems with the implementation educational motion control applications and highlighted the use of rapid prototyping technique for the teaching of theoretically challenging courses. The theoretically designed controller yielded a real-time controller with predicted response. While this chapter concentrated on motion control examples the RADE system has immediate applications to the teaching of DSP and communications courses.

The on-line parameter and data-logging feature of the RADE system illustrates how students can interactively investigate plant/controller responses. An added benefit of the Simulink system is that students can perform quick qualitative experiments at the introductory stage of courses and then progress to the implementation of quantitatively designed controllers. This system allows students to practically explore real-time systems and gain an appreciation of the theoretical control topic without being bewildered by the complexities of the lower levels of DSP real-time code.

Both the RADE ADC64 and PC32 have been demonstrated to provide an effective rapid prototyping platform for educational applications.

CHAPTER EIGHT: CONCLUSION

8.1. General

The practical work for this thesis produced a rapid prototyping tool that is suited for the teaching of controls and allied fields. It was based on The Mathworks RTW and implemented the RADE framework for two Π DSP cards i.e. the PC32 and ADC64 cards.

This thesis also presented a cross section of the activity within the rapid prototyping arena, focused on educational application. The work of [GANI, GREGAL, KRONISIC1] shows the positive impact rapid prototyping can make on engineering education. In the South Africa context of poor pass rates and disparaging backgrounds of undergraduate students [JAWITZ1], rapid prototyping present a possible avenue to assist these students. The major advantage of rapid prototyping is that it promotes authentic cognitive¹ learning and affords the students the opportunity to interactively be involved with experiments.

While rapid prototyping allows for new and improved teaching methods it still faces a few hurdles, which will have to be overcome, in order to become a mainstream teaching tool. These include equipment cost and development time. However these pale in comparison to the attitudes of the educators themselves. A point in case is the author's engineering department, while the Motion Control Group has done a sterling job of producing rapid prototyping tools very few of the other academics are willing to trial or contribute to the development process. It seems that the appeal of rapid prototyping is not clear to them and this mindset needs to be changed in-order for rapid prototyping to be an effective teaching aid.

8.1.1 Role of the RADE Framework

The Motion Control Group's first rapid prototyping tool was CSDE developed by Stylo. The RADE framework built from this start and produced a system that:

- Closely conforms to The Mathworks specifications.
- Provides full network support.

¹ Authentic cognitive learning refers to the learning process whereby the individual develops knowledge by forming and refining concepts in a personal idiosyncratic manner [SQUIRES1]

- Provides a scalable framework that can be ported to various target cards.
- Supports on-line parameter tuning and data logging with capture features.

The RADE framework while an effective rapid prototyping tool, is not the final point in the development cycle. It rather represents a step in the Motion Controls Groups on going development strategy. The RADE framework will aid this process and help future developers produce more effective and user-friendly rapid prototyping tools.

8.2. Suggestions for Further Work

As mentioned above the rapid prototyping tools being developed at the Motion Control Group is an on going process and the author during the course of his work has identified the following areas as possible research topics for further work.

Addition of a Real-Time OS on the target

Currently the target platforms used in the RADE framework are at the lower to medium end of DSP processor available and are not suitable to run a real-time OS (RTOS) due to bandwidth constraints. It is envisaged that more sophisticated/powerful processor platforms² will be incorporated into the RADE framework and these platforms will have the necessary bandwidth needed to run a RTOS. The benefits of using a real-time OS are:

- It allows for multitasking.
- More features can be built into the target.
- It will be possible to directly connect to the target platform onto a network as most RTOSs include network support.

Analysis of course impact

An aim of the RADE system is to improve the teaching of controls systems, DSP and aligned courses. This aim needs to be quantitatively evaluated as feedback from lecturers and students are necessary to develop the RADE framework further. Currently the use of the RADE system has shown qualitative benefits, which needs to be backed up by proper analysis of the course impact, which such tools make.

Local DSP cards

A major hurdle that rapid prototyping tools face in general is the cost factor, which is amplified, in the South African context. A possible solution to this impediment is the development of a local DSP

² A TMS320C67 card is scheduled for incorporation into the RADE framework in the near future.

platform. A collaborative effort among local universities will be an ideal way to develop a cost effective platform that can be utilised by all role payers.

Revision Control issue

The RADE framework is inherently complex due to the incorporation of numerous tools and packages. There are about 20 to 30 files that make up each implementation of the RADE framework and it is difficult and near impossible to keep track of revisions. It is therefore advisable for revision control software to be used for future developments. Packages like Visual Source Safe from Microsoft or CVS from GUN will be ideal.

APPENDIX A:

USER GUIDE

A.1. INSTALLATION MANUAL FOR RADE VERSION 1

The installation for the RADE system is inherently a complex task, due to the use of three different packages. Namely:

- a. TI compiler
- a. Innovative Integration Zuma tool set
- b. Matlab release R11

It is recommended that the TI and II Zuma toolset are installed using the II installation manual with the following additional comment

1. TI compiler

- a. Use default directory. c:\fltc for ver 4.7 and c:\c3xtools for ver 5.10
- b. Note there is a patch for the TI compiler ver 5.10, which upgrades it to version 5.11.
Run self-extraction file 3xwinp5x.exe found on RADE installation CD, in the \c3xtools directory. Then run the install.bat file.
- c. CSDE uses only compiler ver 4.7
- d. The following header files should be copied from RADE installation CD (\C3xtools\include) to the respective TI directory on the installation PC. This applies to version 5.10 only. For version 4.7 use the header file found in the \fltc directory on the RADE installation CD.
 - 1.d.1. bus30.h
 - 1.d.2. bus32.h
 - 1.d.3. dma30.h
 - 1.d.4. dma32.h
 - 1.d.5. timer30.h

2. Zuma toolset

- a. Use the newest II installation ver 2.28, as the libraries files for older version are not compatible with TI compiler 5.11
- b. For the PC32 card Adam has modified library files, these files reside in the Matlab directory and are copied to the target PC in the following steps.

A.1.1 INSTALLATION OF THE RADE COMPONENTS TO MATLAB DIRECTORY

1. Use the default directory of c:\matlabr11 for Matlab installation on target PC.

2. Copy the following directories from the RADE installation CD.
 - a. CD:\matlabr11\rtw\c\ii
 - b. CD:\matlabr11\rtw\c\grt_c3x
 - c. Optional demo directories CD:\matlabr11\work\adc_demo and D:\matlabr11\work\PC32_demo
3. Bug fixes for R11.
 - a. Make backup of the c:\bin\licence.dat file to the c:\matlabr11 directory on the target installation PC.
 - b. Copy the d:\matlabr11\bin directory from the RADE installation CD to the target PCs c:\matlabr11\bin directory.
 - c. Replace the licence.dat file backup at step 3.a above.
4. Moving files on target installation PC
 - a. copy the files in directory c:\matlabr11\rtw\c\ii\src_mod to c:\matlabr11\rtw\c\src
 - b. copy the files in directory c:\matlabr11\rtw\c\ii\tlc_mod to c:\matlabr11\rtw\c\tlc
5. Setup Matlab path with the following paths.
 - a. c:\matlabr11\rtw\c\ii\devices
 - b. c:\matlabr11\rtw\c\ii\rti_fpc
 - c. c:\matlabr11\rtw\c\ii\ext_mode
 - d. c:\matlabr11\rtw\c\ii\adc\devices
 - e. c:\matlabr11\rtw\c\ii\adc\src

A.1.2 INSTALLATION OF SERVER APPLICATION.

There are two versions of the server application for the respective DSP cards. In addition there are two different builds of each server application: (all in all four server applications)

1. Dynamical linked. This version uses MCF DLL's dynamically and works only if visual C/C++ is installed.
2. Static linked. This version uses the MFC DLL's statically and works on machines without the need for visual C/C++
3. For a standalone system, i.e. Matlab and DSP target on same PC
 - a. The server applications is found in the c:\matlabr11\rtw\c\ii\bin directory
 - b. Create shortcuts to the respective applications.
4. For a DSP card at a remote workstation
 - a. Make a server directory on PC
 - b. Copy the server application from the RADE CD (c:\matlabr11\rtw\c\ii\bin) to c:\server directory.

You are now ready to use the RADE system

A.2. SINE WAVE EXAMPLE

The following examples are to be used in conjunction with RTW manual. They are intended to demonstrate how to setup the RADE system and are suitable to use as a control measure to ensure the system is working. Thereafter refer to chapters 5 and 6 for information on uses of specific device drivers

A.2.1 RADE PC32

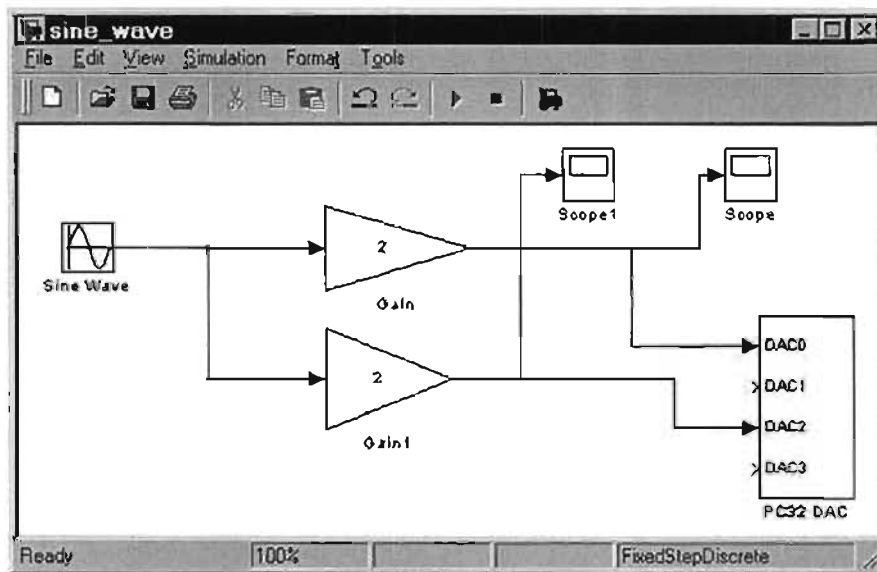


Fig. A. 1: Simulink diagram

The Simulink model shown in Fig. A. 1 is uses a sine wave block to output a sine wave on two DAC channels. The gain blocks are used to alter the signal amplitude on each of the DAC channels. It should be note that both the gain and sine wave blocks can be modified on-line.

The setup of Simulink parameters involves a five-stage process¹:

1. Setup Simulation Parameters

When a model is being converted into real-time code only fixed step size can be used. It possible to use either a continuous or discrete solver. Fig. A. 2 shows the parameters used.

2. RTW Options

¹ The RTW manual provides detailed information on the setup process and purpose of the respective parameters. This section is only intended to highlight the RADE PC32 parameters used.

The RTW options are used to select the target type. In the case of the RADE PC32 the `grt_c3x.tlc` file is used. The options used are shown in Fig. A. 3.

3. RTW Build Options

The RTW build options are used to setup option used for the code build cycle. The options used are shown in Fig. A. 4.

4. External Mode Parameters

These parameters are used to setup external mode. Fig. A. 5 shows the parameters used.

5. Data-Logging

The two scope blocks shown in Fig. A. 1 can be used for data logging and are setup in the External Signal & Triggering window shown in Fig. A. 6. It should be noted that a maximum of 8000 samples are allowed, due target memory constraints.

Once these parameters have been setup, the model can be built. The results obtained are shown in Fig. A. 7. Note the server program must be operating on the target PC.

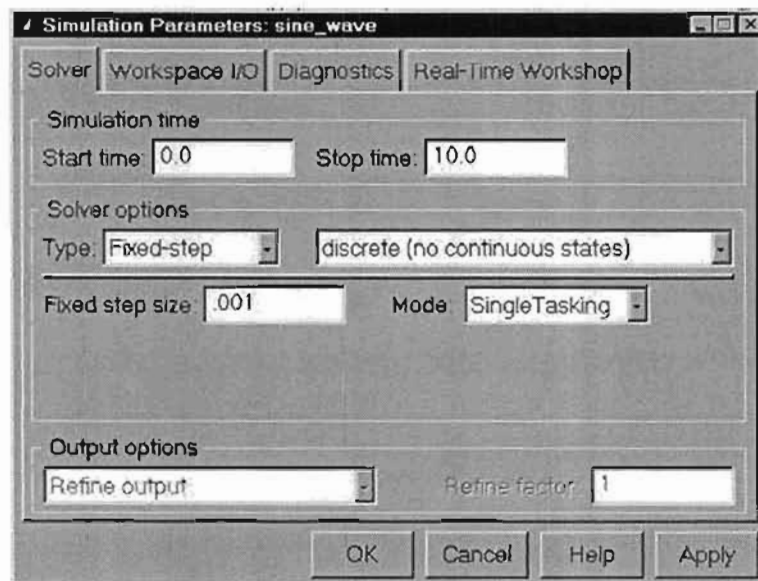
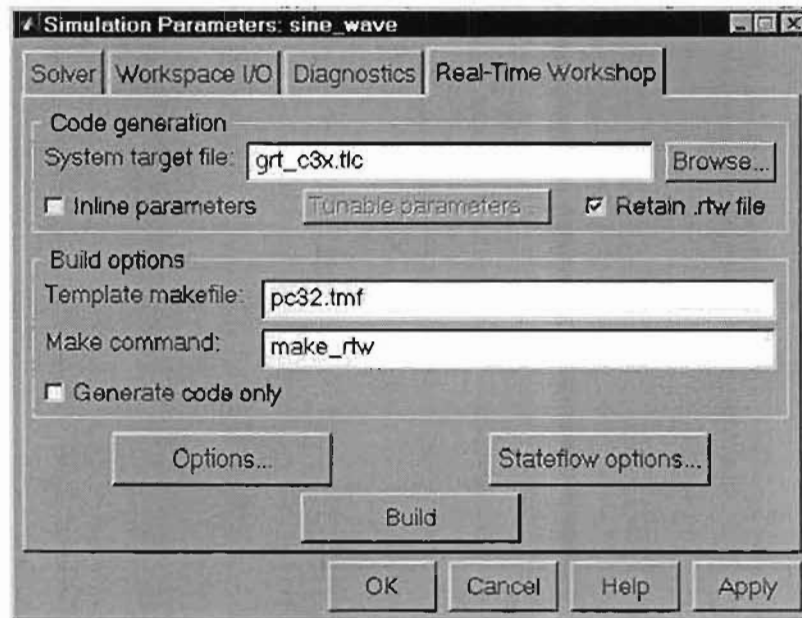
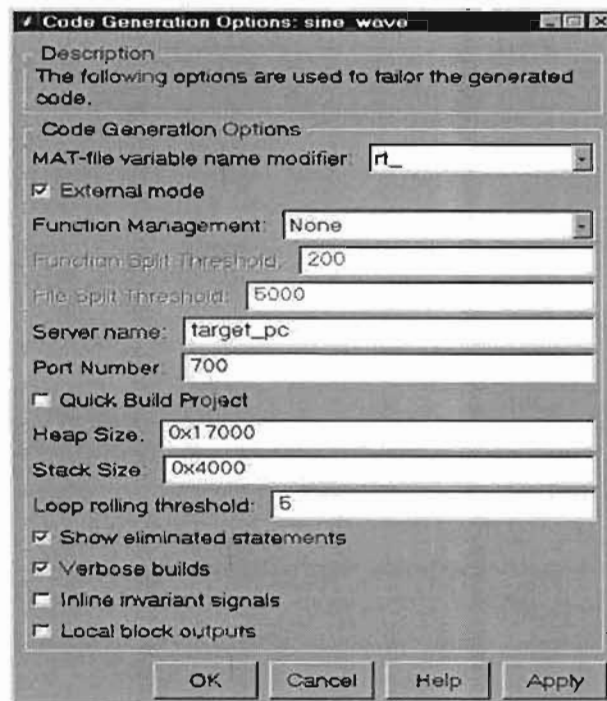
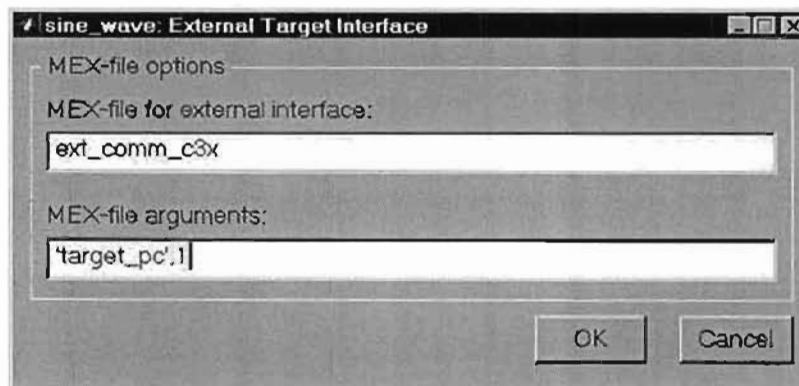


Fig. A. 2: Simulation parameters

*Fig. A. 3: RTW options**Fig. A. 4: RTW build options**Fig. A. 5: External mode setup*

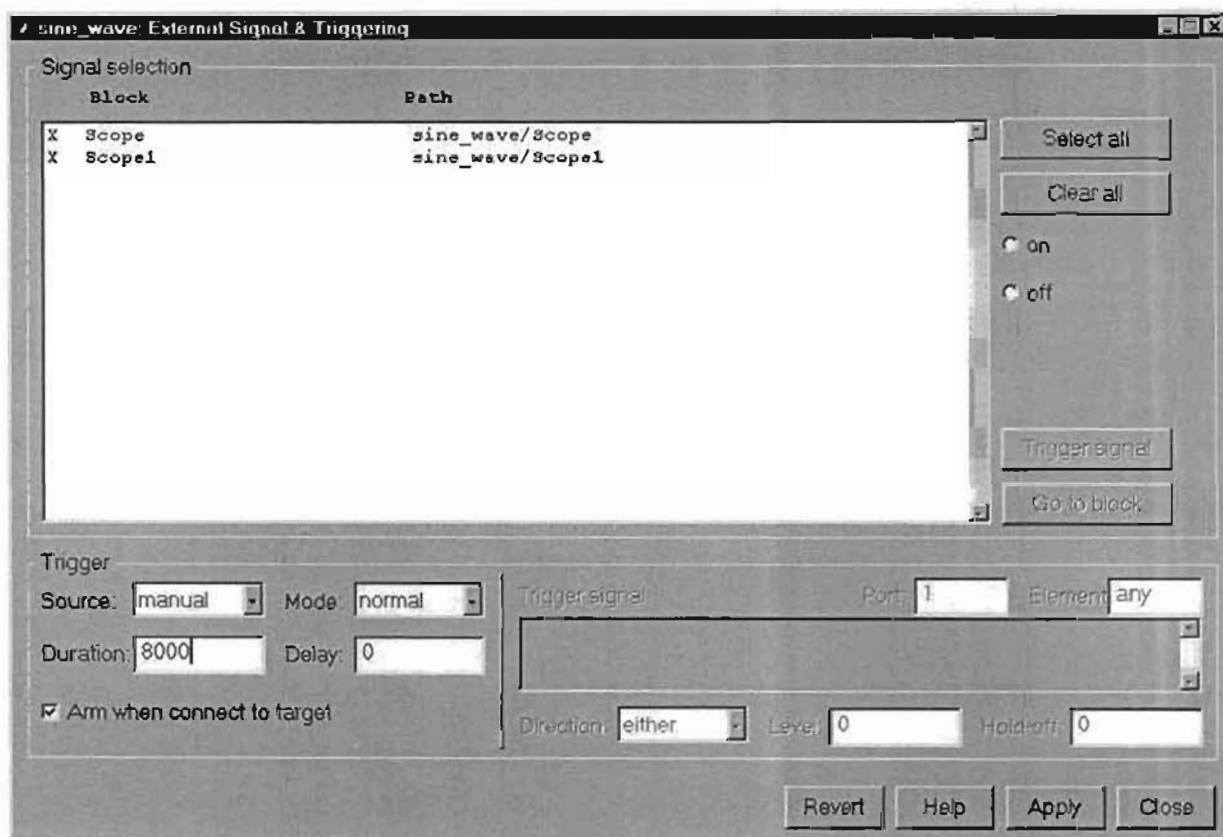


Fig. A. 6: Data logging setup

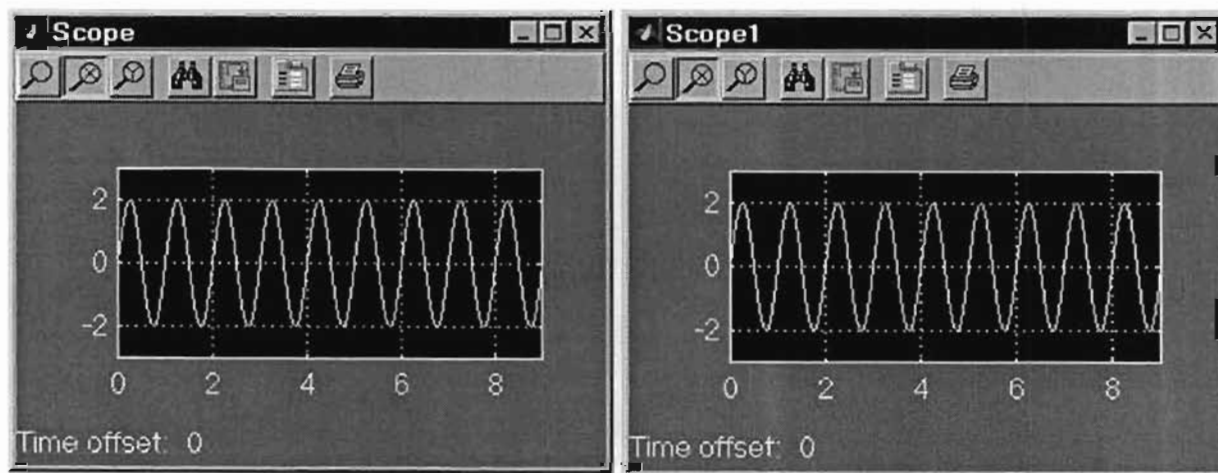
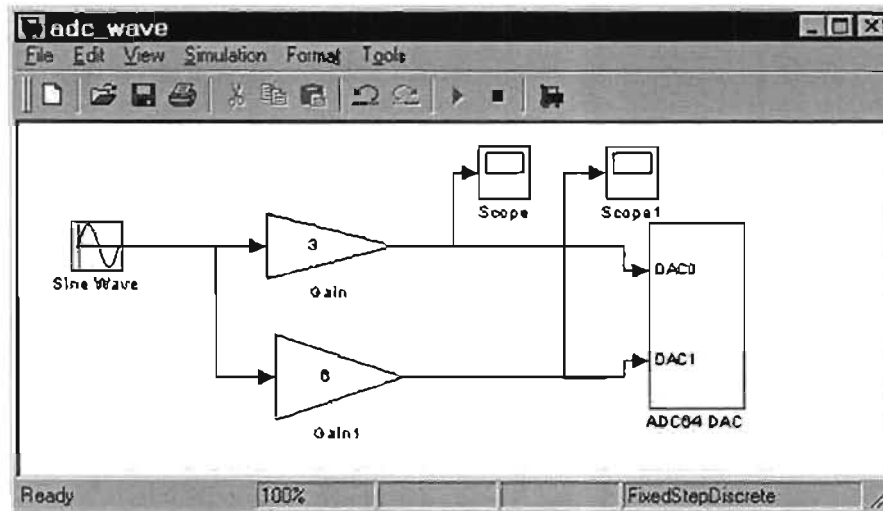
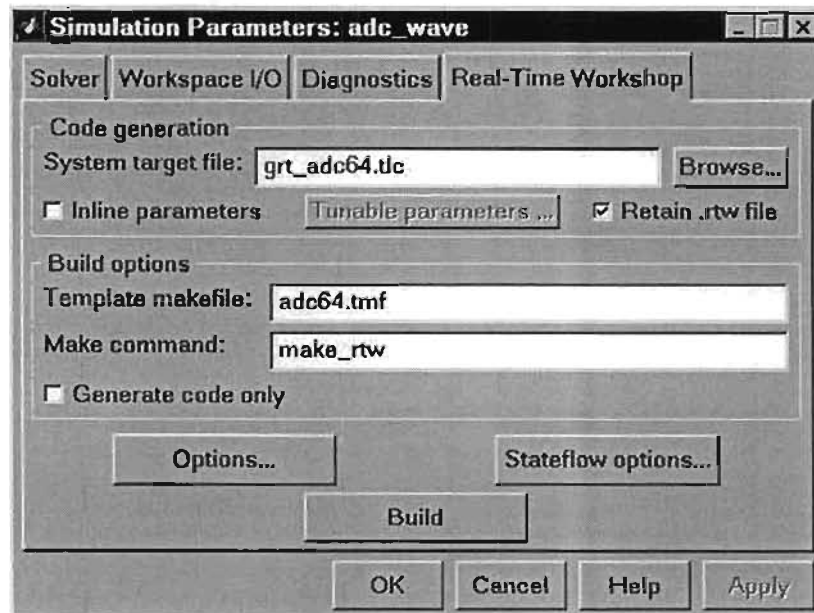
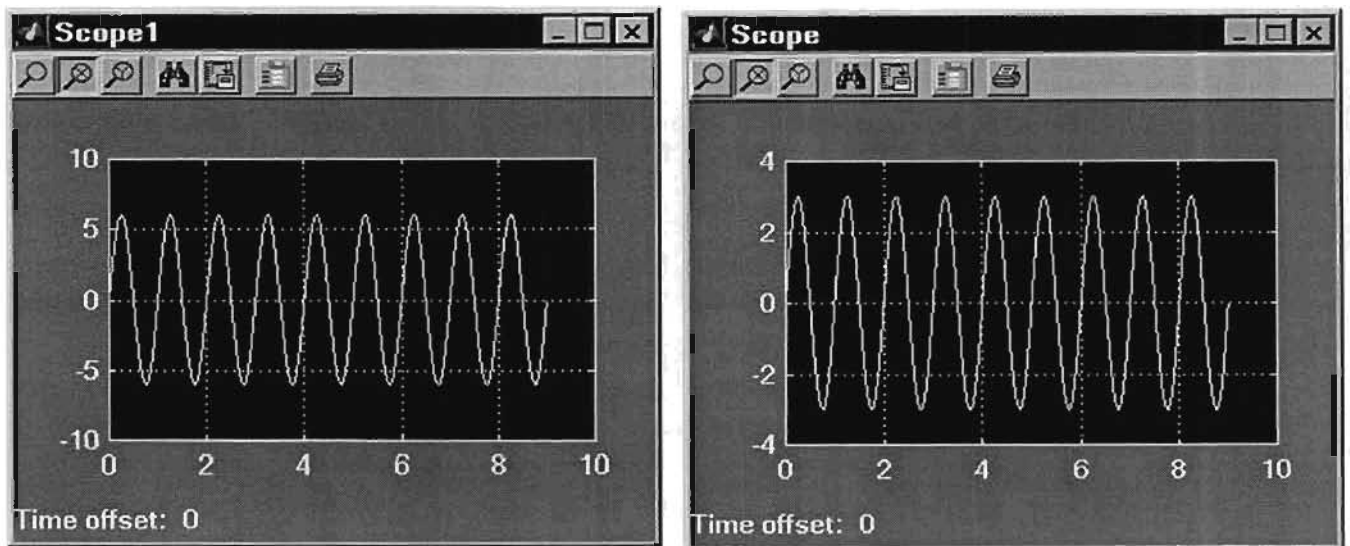


Fig. A. 7: Results form scope block

A.2.2 RADE ADC64

The use of the RADE ADC64 system is functionally very similar to the RADE PC32 and only requires the ADC64 device drivers to be used with a few changes to the RTW options. The Simulink model use, shown in Fig. A. 1, is equivalent to the RADE PC32 example demonstrated above. The RTW options that have changed are shown in Fig. A. 9. Once these parameters have been change the model is ready to be built. With the server executing on the target PC the results shown in Fig. A. 10 should be obtained.

*Fig. A. 8: Simulink diagram**Fig. A. 9: RTW options*

*Fig. A. 10: Scope block results*

APPENDIX B:

A PROGRAMMER'S GUIDE TO THE INTERNAL WORKING OF THE RADE SYSTEMS

B.1. MODIFICATIONS TO THE MATHWORKS EXTERNAL MODE IMPLEMENTATION

This section is intended to provide programmatic details on the modifications made to The Mathworks external mode implementation. The emphasis of this section is to highlight the various functions used in the default implementation and then explains the changes needed for the RADE implementation. It is recommended that the reader have a proper understanding of the RTW and workings of the RADE framework before attempting to make modifications.

B.1.1 DEFAULT MATHWORKS EXTERNAL MODE IMPLEMENTATION

The default Mathworks external mode implementation consists of two major files and two minor files; Table B 1 lists these files. Ext_comm.c module which implements the Simulink communication layer has already been adequately covered in chapter 3 and 4 and further explanation is not necessary. Hard copies of these files are not listed in this thesis, as electronic copies are available on the CD attached.

File Name	Related Files	Description
Ext_comm.c		Simulink communication layer module. See chapter 3
	Ext_convert.c	Conversion module. Converts data between target and host types.
Ext_srv.c		External mode module that run on the target side.
	Updown.c	This is a addition file that contains all the data procesing function. Function from ext_srv call into this module.

Table B 1: Default external mode files

The external server module consists of three major functions:

1. Main loop
Used to manage the external transaction
2. Message Processing
Used to process Simulink messages.
3. Upload Processing

Used to processing the uploading of logged data

These functions all reside in the `ext_srv.c` module and pseudo code listing are presented below. These listing convey the general algorithms for the respective functions and need to be read in conjunction with the C implementation for a full understanding to be gained.

Function Main execution loop

/*This is function represent the execution loop that manages external mode transactions. It is hypothetical function that has been extracted from MAIN function in run time interface i.e. it only represents details pertaining to the external mode aspects */

```
{
    Call rt_ExtModeInit(SimStruct *S, int_T port)
        /*This function setup a listening socket on the target PC*/

    Execution loop /*runs in foreground and can be pre-empted by rt_onestep and other
        interrupts*/
        Start
        Call int_T rt_MsgServerWork(SimStruct *S)
            /*This function is used to receive and process messages sent from
            Simulink*/

        Call int_T rt_UploadServerWork(SimStruct *S)
            /*Used to send upload data if present*/
        end
}
```

Function int_T rt_MsgServerWork(SimStruct *S)

/* Used to receive and process messages sent from Simulink*/

```
{
    If Comms NOT CONNECTED then
    {
        Is Simulink try to Connect
        If YES then
        {
            Call static int_T ConnectToHost(void)
            /*used to open message and upload sockets*/
            Set Comms=CONNECTED
        }
        else
        {
            Waits till Simulink connects to target
            Return
        }
    }
    else /* comms CONNECTED*/
    {
        If first message Then
            Set message code to EXT_CONNECT
            Get data from message socket

        Case Message
        {
```

```
        If EXT_CONNECT
            Call ProcessConnectMsg(S);
            /*used to connect to Simulink client*/
        If EXT_SETPARAM
            Call ProcessSetParamMsg(S, msgHdr.size);
            /*updates parameters*/
        ..
        ..
        ..
        /* All external mode messages are processed here */
    }
End Case
} /*end else*/

} /* end function */

Function int_T rt_UploadServerWork(SimStruct *S)
{
    If data available
    {
        For all buffers
            Call send(upFd, bufMem->section1, bufMem->nBytes1, 0);
            /*WinSock API function to send data*/
        Next
    } /*end if */
} /*end function*/
```

B.1.2 RADE EXTERNAL MODE

On the RADE system all the function described above are used in a modified form i.e. the WinSock parts resides on the PC while the rest resides on the target. The files used for RADE framework external mode implementation are listed in Table B. 2. The Simulink communication layer has already been discussed in chapter 4 and is not revisited here.

The RADE implementation conforms closely to The Mathworks conventions and therefore also consists of three major functions. However these functions are split between the PC (server application) and target platforms. Pseudo code is listed below for these functions.

File Name & Platform	Related Files	Mathworks default file	Description
Ext_comm_c3x.c PC platform		Ext_comm.c	Simulink communication layer module. See chapter 4
	Ext_Convert_c3x.c	Ext_Convert.c	Conversion module. Converts data between target and host types. See chapter 4
Ext_srv_pc.cpp PC platform		Ext_srv.c	

File Name & Platform	Related Files	Mathworks default file	Description
Ext_srv_c3x.c Target Platform		Ext_srv.c	
	Updown_c3x.c	Updown.c	No functional change only added in debugging printf statements.

*Table B. 2: Files used for external mode***Function Main Execution Loop PC Side**

```

/*Found in ext_srv_pc.cpp file and is attached to a timer running at 10ms*/
{
    Call Process_Target_Msg_Work()
        /*used to manage STP transactions */
    Call rt_MsgServerWork()
    Call Thread_Upload(LPVOID pParam)
        /*used to send upload data to Simulink. Found in the STP file, see next section*/
}

```

Function int_T rt_MsgServerWork() /*PC side*/

```

/*Used to receive Simulink messages. ext_srv_pc.cpp */
{
    if comms=CONNECTED then
    {
        Get data from message socket
        Call Send_Data((char*)&msgHdr,sizeof(MsgHeader),TO_TARGET_HDR);
            /*This is a STP function. See next section*/
        Call Send_Data(recv_buf,size*4,TO_TARGET_MSG);
            /*This is a STP function. See next section*/
    }
}

```

Function Main Execution Loop Target Side

```

/*manages external mode execution on the target. Ext_srv_c3x.c*/
{
    Call rt_MsgServerWork(S);
        /*This function process messages that have been sent to the target*/
    Call Process_Msg_Work();
        /*This function manages STP. See next section*/
    Call rt_UploadServerWork(S);
        /*This function sends data to the Server application using STP functions*/
}

```

Function int_T rt_MsgServerWork(SimStruct *S) /*target side*/

```

{
    If message to be processed
    {
        If first message Then
            Set message code to EXT_CONNECT
        Case Message
        {
            If EXT_CONNECT
                Call ProcessConnectMsg(S);
                /*used to connect to Simulink client*/
            }
        }
    }
}

```

```
    If EXT_SETPARAM
        Call ProcessSetParamMsg(S, msgHdr.size);
        /*updates parameters*/
    ..
    ..
    ..
    /* All external mode messages are processed here */
}
End Case
}

Function int_T rt_UploadServerWork(SimStruct *S)
{
If data available
{
    For all buffers
        Call Send_Upload_Data(bufMem->section1, bufMem->nBytes1)
        /*STP function to send data*/
    Next
}/*end if */
}
```

B.2. SERVER TO TARGET PROTOCOL

STP as explained in chapter 4 is the channel through which the server and target application communicate. STP exports standard function that the RADE external mode implementation uses, as describe in the last section. Table B. 3 list the file used of the STP on both the PC32 and ADC64 platforms.

Target Type	File Name	Platform	Description
PC32	ii_comms_pc.cpp	PC	Implements PC side of SPT based on a DPRAM architecture.
	ii_pc32.c	TMS320C32	Implements target side STP
ADC64	ii_adc64_pc.cpp	PC	Implements PC side SPT based on a PCI bus architecture
	ii_adc64.c	TMS320c32	implments target side STP

Table B. 3

The STP is made up of three major functional aspects:

1. The overall management of STP transactions. This entails the checking of mailboxes for incoming data.
2. The sending of data. This part is mirrored on both the PC and target sides of the STP. When data is sent from the target the STP management on PC receives the data and vice versa when

the PC is sending data to the target.

3. The uploading of data. The target sends upload data and the PC STP management function receives, sequences and sends data to Simulink.

The functions used to implement the STP are listed in Table B. 4.

Functional area	Function	Platform	File	Description
PC side SPT management	Process_Target_Msg_Work()	PC32	ii_comms_pc.cpp	Checks from target mail port for data and receives data
	Process_Target_Msg_Work()	ADC64	li_adc64_pc.cpp	Checks from target mail port for data and receives data
	Thread_Upload(LPVOID pParam)	Both PC32 and ADC64	ii_comms_pc.cpp + li_adc64_pc.cpp	This thread check if upload data is ready to be sent to Simulink
PC send functions	Send_Data(char *p_msg,int len,msg_id id)	PC32	ii_comms_pc.cpp	Send external mode messages to the target
	Send_Data(char *p_msg,int len,msg_id id)	ADC64	li_adc64_pc.cpp	Send external mode messages to the target
Target STP management	Process_Msg_Work()	PC32	ii_pc32.c	Receives data for server
	Process_Msg_Work()	ADC64	ii_adc64.c	Receives data for server
Target Send function	Send_Data(char *p_msg,int len,msg_id id)Send data	PC32 +ADC64	ii_pc32.c+ ii_adc64.c	Sends external mode messages to server
	Send_Upload_Data(char *buf, int len)	PC32 +ADC64	ii_pc32.c+ ii_adc64.c	Sends upload data to server application

Table B. 4

APPENDIX C: EVALUATION OF MOTOR PARAMETERS

The motor parameters are found using a no-load test and The Mathworks Non-linear Design Control Blockset [MATHWORKS8]. The motor used is a permanent magnet type shown in Fig. C. 1.

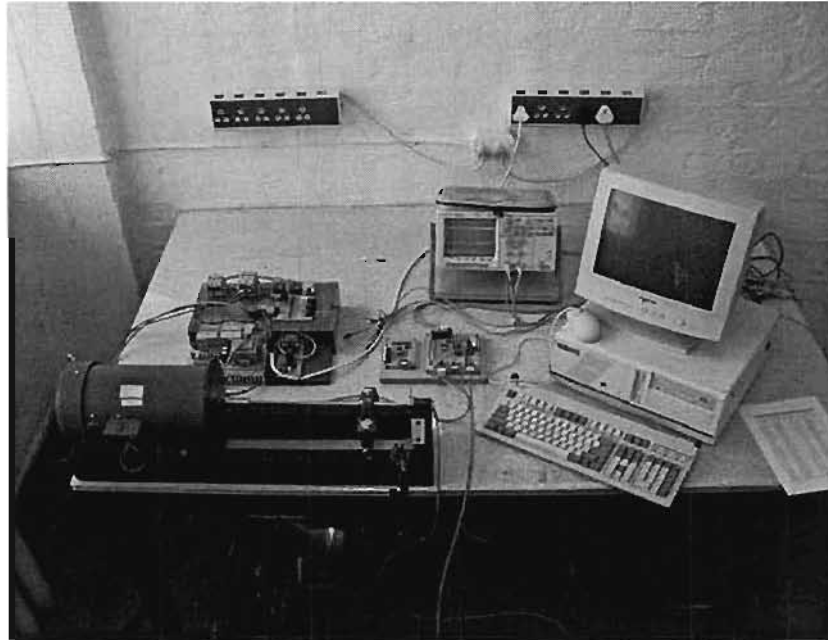


Fig. C. 1: Photo of motor

The motor armature is subject to a 85V DC voltage step and the captured current and speed responses are shown in Fig. C. 2. This data is used with the NCD block [MATHWORKS8] to perform a system identification analysis on the motor.

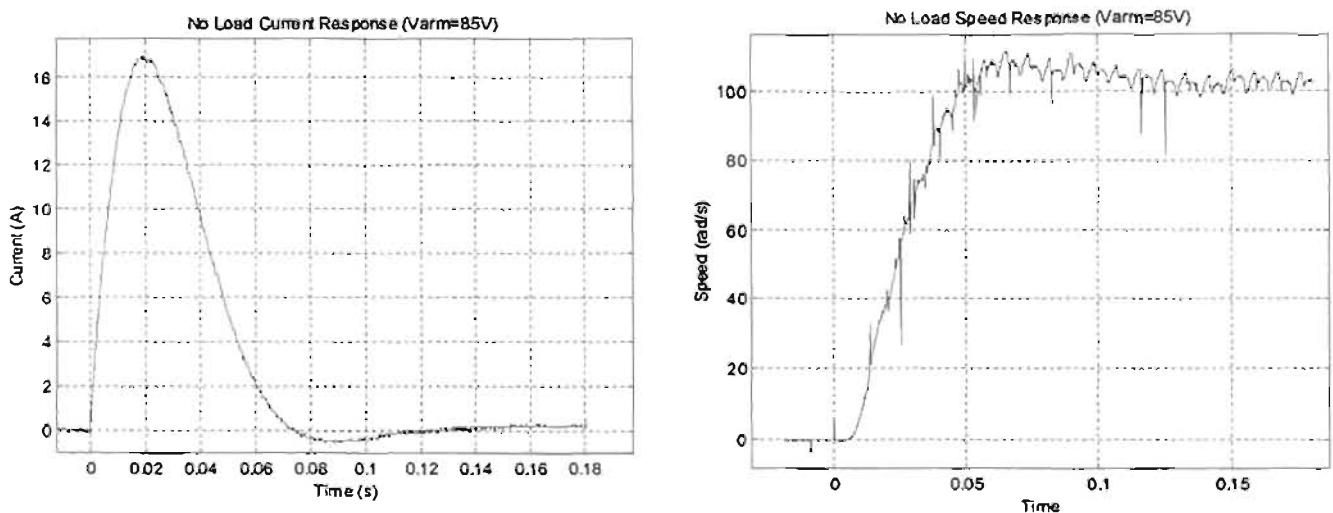


Fig. C. 2; Measured no-load motor responses

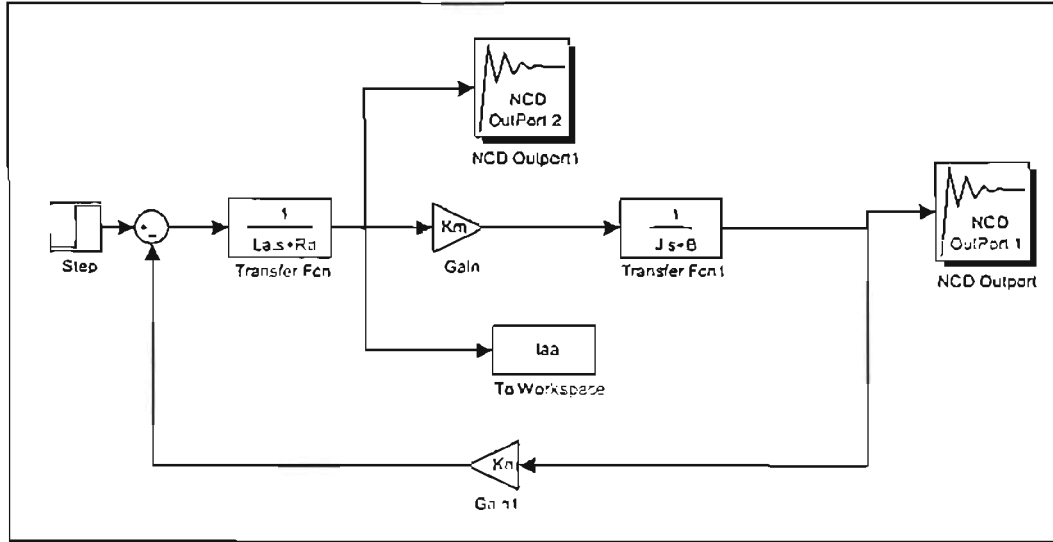
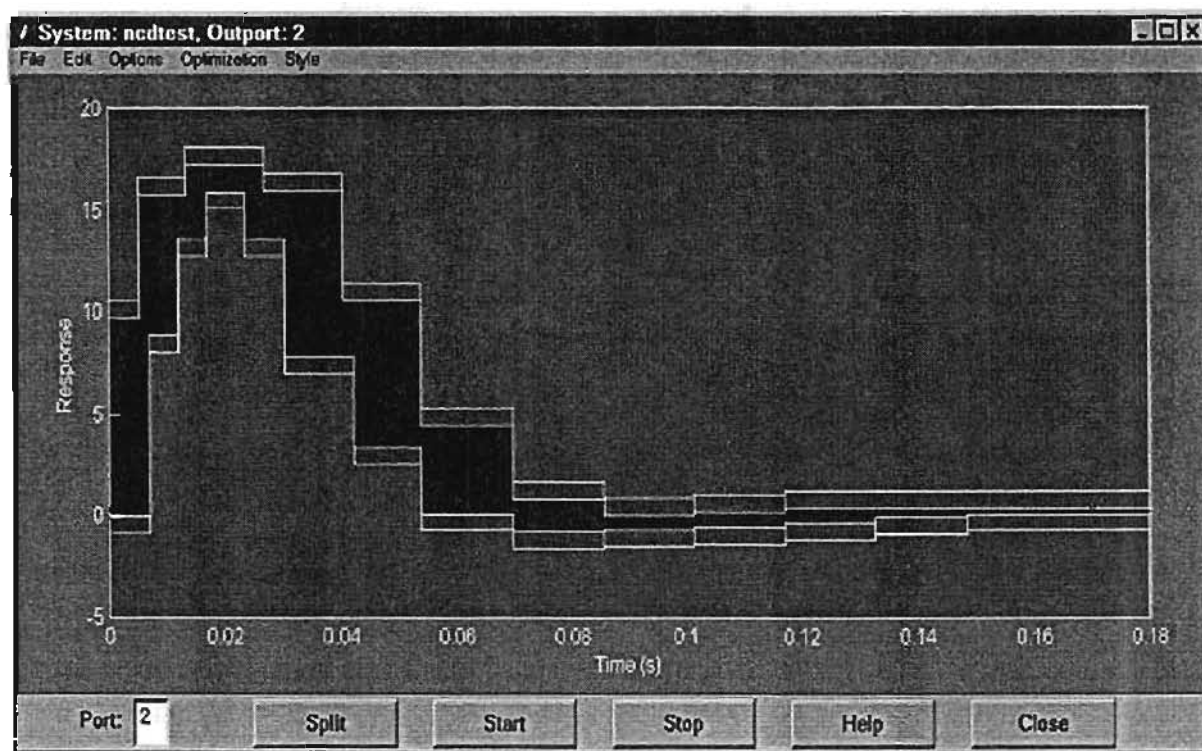
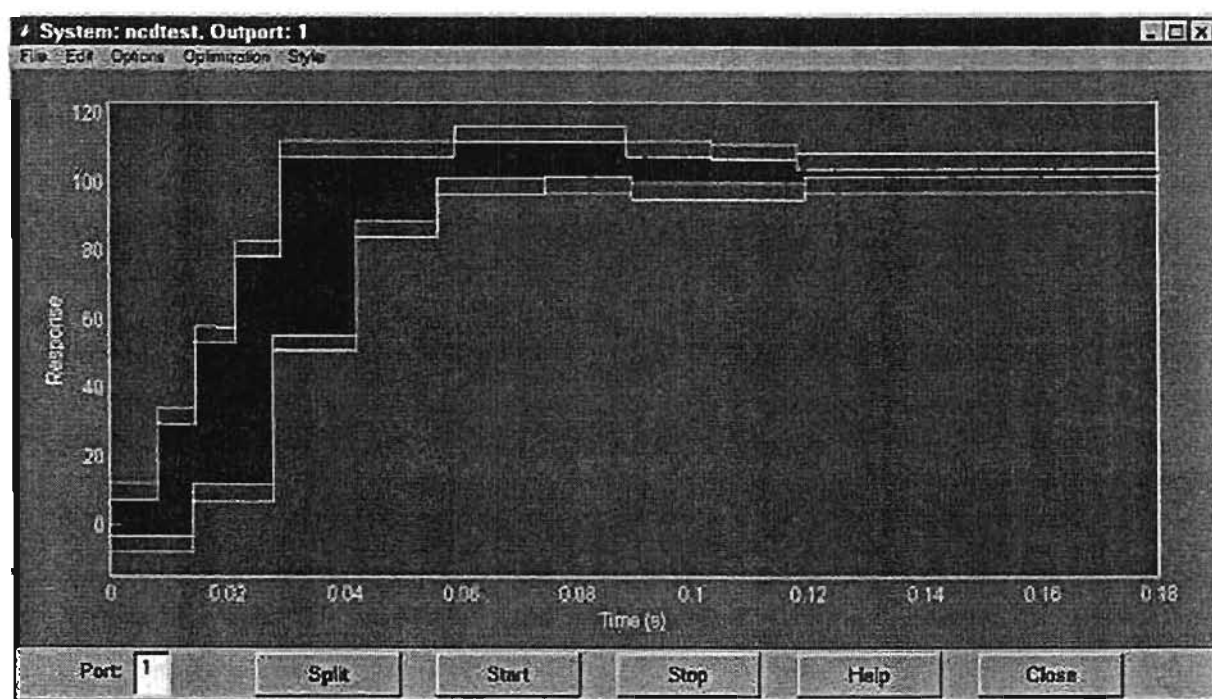


Fig. C. 3: Simulink model used for system Identification

The NCD block operates within the Simulink environment and requires the theoretical model of the motor to be implemented, as shown in Fig. C. 3. This model uses the variables to represent the parameter of the system that need to be identified, which are:

- Armature inductance L_a
- Armature resistance R_a
- Motor rotational moment J
- Motor viscous damping constant B
- Motor constant K_m

The current response is loaded into NCD outport 2 and the plant constraints are set using the NCD environment, and Fig. C. 4 shows this. A similar process is used for the speed response and Fig. C. 5 shows this. The motor parameters to be estimated are setup using the optimisation window shown in Fig. C. 6. The motor parameters estimated by the NCD block are summarised in Fig. C. 7.

*Fig. C. 4: Current constraints**Fig. C. 5: Speed constraints*

Optimization Parameters

Tunable Variables: Km J B Ra La

Lower bounds (rational): 0.1 0.00001 0.00001 1e-3

Upper bounds (rational): 33 0.1 0.1 50e-3

Discretization interval: 0.001

Variable Tolerance: 1e-006

Constraint Tolerance: 0.001

☒ Display optimization information

☐ Stop optimization as soon as the constraints are achieved

☐ Compute gradients with better accuracy (slower)

Note: Simulation parameters are entered in the SIMULINK system.

Fig. C. 6: Tuneable parameters

Parameter	Value	Unit
Armature inductance (La)	46	mH
Armature resistance (Ra)	3.36	Ω
Rotor moment of inertia (J)	4.889×10^{-3}	Kgm^2
Rotor viscous damping (B)	4.291×10^{-4}	Kgm^2/s
Motor Constant (Km)	0.834	

Fig. C. 7: Motor parameters estimated

APPENDIX D: LISTING OF CODE FOR THE RADE PC32

D.1. CONVERSION FUNCTIONS EXT_CONVERT_C3X.C

```
/*
 * Utility functions for ext_comm.c.
 */

#include <string.h>
#include <windows.h>
#include "tmwtypes.h"
#include "mex.h"
#include "extsim.h"
#include "extutil.h"

//Magash 13/09/1999
//li header
//note directory
#include "target.h"

static void Single_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType);
static void Single_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType);
static void Double_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType);
static void Double_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType);
static void Int8_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType); /* internal Simulink data type id */

static void Int16_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType); /* internal Simulink data type id */

static void Uint32_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType);
static void Uint32_TargetToHost(
    ExternalSim *ES,
```



```
void      *dst,
const char *src,
const int  n,
const int  dType); /* internal Simulink data type id */
static void Int32_TargetToHost(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

static void Uint8_TargetToHost(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

static void Uint16_TargetToHost(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

static void Int8_HostToTarget(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

static void Int16_HostToTarget(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

static void Int32_HostToTarget(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

static void Uint8_HostToTarget(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

static void Uint16_HostToTarget(
    ExternalSim *ES,
    void      *dst,
    const char *src,
    const int  n,
    const int  dType); /* internal Simulink data type id */

/* Function: Bool_HostToTarget =====
 * Abstract:
 *   Convert Simulink (hosts) bool value (uint8_T) to target boolean_T value.
```

```

* No assumptions may be made about the alignment of the dst ptr.
* The src pointer is aligned for type uint8_T. As implemented, this function
* supports either uint8_T boolean values on the target or uint32_T booleans
* on the target (for dsps that support only 32-bit words).
*/
static void Bool_HostToTarget(
    ExternalSim *ES,
    char *dst,
    const void *voidSrc,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    boolean_T swapBytes = esGetSwapBytes(ES);
    const uint8_T *src = (const uint8_T *)voidSrc;

    int sizeofTargetBool = esGetSizeOfTargetDataTypeFcn(ES)(ES, dType) *
        esGetHostBytesPerTargetByte(ES);

    int i;
    char *dstPtr = dst;

    for (i=0; i<n; i++)
    {
        uint32_T tmp = (uint32_T)src[i];
        (void)memcpy(dstPtr, &tmp, 4);
        dstPtr += 4;
    }

} /* end Bool_HostToTarget */

/* Function: Bool_TargetToHost =====
* Abstract:
* Convert target bool value to host bool value (uint8_t). No assumptions may
* be made about the alignment of the src ptr. The dst pointer is aligned for
* type uint8_T. As implemented, this function supports either uint8_T boolean
* values on the target or uint32_T booleans on the target (for dsps that
* support only 32-bit words).
*/
static void Bool_TargetToHost(
    ExternalSim *ES,
    void *voidDst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    #define MAX_ELS (1024)
    boolean_T swapBytes = esGetSwapBytes(ES);
    uint8_T *dst = (uint8_T *)voidDst;

    int sizeofTargetBool = esGetSizeOfTargetDataTypeFcn(ES)(ES, dType) *
        esGetHostBytesPerTargetByte(ES);

    int i;
    uint32_T *tmp = (uint32_T *)src;
    for (i=0; i<n; i++)
    {
        dst[i] = (uint8_T)(*tmp++);
    }

#undef MAX_ELS
} /* end Bool_TargetToHost */

/* Function: Generic_HostToTarget =====
* Abstract:
* Convert generic data type from host to target format. This function

```

```

* may be used with any data type where the number of bits is known to
* be the same on both host and target (e.g., int32_T, uint16_t, etc).
* It simply copies the correct number of bits from target to host performing
* byte swapping if required. If any other conversion is required, then
* a custom HostToTarget function must be used.
*/
static void Generic_HostToTarget(
    ExternalSim *ES,
    char *dst,
    const void *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    int dTypeSize = esGetSizeOfDataTypeFcn(ES)(ES, dType);
    boolean_T swapBytes = esGetSwapBytes(ES);

    // slCopyNBytes(dst, src, n, swapBytes, dTypeSize);
} /* end Generic_HostToTarget */

/* Function: Generic_TargetToHost =====
* Abstract:
* Convert generic data type from target to host format. This function
* may be used with any data type where the number of bits is known to
* be the same on both host and target (e.g., int32_T, uint16_t, etc).
* It simply copies the correct number of bits from host to target performing
* byte swapping if required. If any other conversion is required, then
* a custom TargetToHost function must be used.
*/
static void Generic_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    int dTypeSize = esGetSizeOfDataTypeFcn(ES)(ES, dType);
    boolean_T swapBytes = esGetSwapBytes(ES);

    // slCopyNBytes(dst, src, n, swapBytes, dTypeSize);
} /* end Generic_TargetToHost */

/* Function: Copy32BitsToTarget =====
* Abstract:
* Copy 32 bits to the target. It is assumed that the only conversion needed
* is bytes swapping (if needed) (e.g., uint32, int32). Note that this fcn
* does not rely on the Simulink Internal data type id.
*/
void Copy32BitsToTarget(
    ExternalSim *ES,
    char *dst,
    const void *src,
    const int n)
{
    //just use uint32

    Uint32_HostToTarget(ES, dst, src, n, 0);
} /* end Copy32BitsToTarget */

/* Function: Copy32BitsFromTarget =====
* Abstract:
* Copy 32 bits from the target. It is assumed that the only conversion needed
* is bytes swapping (if needed) (e.g., uint32, int32). Note that this fcn
* does not rely on the Simulink Internal data type id.
*/

```

```

void Copy32BitsFromTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n)
{
    UInt32_TargetToHost(ES,dst,src,n,0);
}
/* end Copy32BitsFromTarget */

/* Function
=====
* Process the first of two EXT_CONNECT_RESPONSE messages from the target.
* This message consists of nothing but a message header. In this special
* instance we interpret the size field as the number of bits in a target
* byte (not always 8 - see TI compiler for C30 and C40).
*
* This function is responsible for deducing the endian format of the target,
* validating that the number of bits per target byte and setting up pointers
* to data conversion functions.
*
* NOTE: The caller must check that the error status is clear after calling
* this function (i.e., esIsErrorClear(ES)).
*/
void ProcessConnectResponse1(ExternalSim *ES, MsgHeader *msgHdr)
{
    /*
    * Deduce the endian-ness of the target.
    */
    if (msgHdr->type == EXT_CONNECT_RESPONSE) {
        esSetSwapBytes(ES, FALSE);
    } else {
        const boolean_T swapBytes = TRUE;

        //commented out by Magash
        //slCopyFourBytes(msgHdr, msgHdr, NUM_HDR_ELS, swapBytes);
        if (msgHdr->type != EXT_CONNECT_RESPONSE) {
            esSetError(ES, "Invalid EXT_CONNECT_RESPONSE message.\n");
            goto EXIT_POINT;
        }
        esSetSwapBytes(ES, swapBytes);
    }

    /*
    * Process bits per target byte.
    */
    {
        int_T bitsPerTargetByte = msgHdr->size;
        int_T hostBytesPerTargetByte = bitsPerTargetByte/8;

        assert(bitsPerTargetByte%8 == 0);
        esSetHostBytesPerTargetByte(ES, hostBytesPerTargetByte);
    }

    /*
    * Set up fcn ptrs for data conversion - Simulink data types.
    */
    esSetDoubleTargetToHostFcn(ES, Double_TargetToHost);
    esSetDoubleHostToTargetFcn(ES, Double_HostToTarget);

    esSetSingleTargetToHostFcn(ES, Single_TargetToHost); /* assume 32 bit */
    esSetSingleHostToTargetFcn(ES, Single_HostToTarget); /* assume 32 bit */

    esSetInt8TargetToHostFcn(ES, Int8_TargetToHost);
    esSetInt8HostToTargetFcn(ES, Int8_HostToTarget);

```

```

    esSetUInt8TargetToHostFcn(ES, UInt8_TargetToHost);
    esSetUInt8HostToTargetFcn(ES, UInt8_HostToTarget);

    esSetInt16TargetToHostFcn(ES, Int16_TargetToHost);
    esSetInt16HostToTargetFcn(ES, Int16_HostToTarget);

    esSetUInt16TargetToHostFcn(ES, UInt16_TargetToHost);
    esSetUInt16HostToTargetFcn(ES, UInt16_HostToTarget);

    esSetInt32TargetToHostFcn(ES, Int32_TargetToHost);
    esSetInt32HostToTargetFcn(ES, Int32_HostToTarget);

    esSetUInt32TargetToHostFcn(ES, UInt32_TargetToHost);
    esSetUInt32HostToTargetFcn(ES, UInt32_HostToTarget);

    esSetBoolTargetToHostFcn(ES, Bool_TargetToHost);
    esSetBoolHostToTargetFcn(ES, Bool_HostToTarget);

EXIT_POINT:
    return;
} /* end ProcessConnectResponse1 */

/* Function
=====
* Process the data sizes information from the second EXT_CONNECT_RESPONSE
* messages. The data passed into this function is of the form:
*
* nDataTypes - # of data types      (uint32_T)
* dataTypeSizes - 1 per nDataTypes (uint32_T[])
*
* NOTE: The caller must check that the error status is clear after calling
*       this function (i.e., esIsErrorClear(ES)).
*/
void ProcessTargetDataSizes(ExternalSim *ES, uint32_T *bufPtr)
{
    uint32_T i;

    /* nDataTypes */
    if (esGetNumDataTypes(ES) != *bufPtr++) {
        esSetError(ES, "Unexpected number of data types returned from host.\n");
        goto EXIT_POINT;
    }

    /* data type sizes */
    for (i=0; i<esGetNumDataTypes(ES); i++) {
        esSetDataTypeSize(ES, i, (*bufPtr++));

        //correction by magash
        //mul by 4
        //removed because of updated files
    }

EXIT_POINT:
    return;
} /* end ProcessTargetDataSizes */

void PC_Format(ExternalSim *ES)
{
    /*
    * Process bits per target byte.
    */
    esSetHostBytesPerTargetByte(ES, 1);

```

```
    esSetDataTypeSize(ES, 0, 8); //correction by magash

EXIT_POINT:
    return;
} /* end ProcessTargetDataSizes */

/*
Magash 13/09/1999
All mods for conversion routines
*/
static void Single_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    int dTypeSize = esGetSizeOfDataTypeFcn(ES)(ES, dType);
    boolean_T swapBytes = esGetSwapBytes(ES);
    //assume data is 32 bits in target format

    float *p_dst;
    uint32_T *p_src;
    int32_T i;
    p_src=(uint32_T*)src;
    p_dst=(float*)dst;

    for(i=0;i<n;i++)
    {
        *p_dst=to_ieee(*p_src);
        //printf("call to single to target %f\n",*p_dst);
        p_dst++;
        p_src++;
    }
}

static void Single_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    int dTypeSize = esGetSizeOfDataTypeFcn(ES)(ES, dType);
    boolean_T swapBytes = esGetSwapBytes(ES);
    //assume data is 32 bits in target format

    float *p_src;
    uint32_T *p_dst;
    int32_T i;
    p_src=(float*)src;
    p_dst=(uint32_T*)dst;

    for(i=0;i<n;i++)
    {
        *p_dst=from_ieee(*p_src);
        p_dst++;
        p_src++;
    }
}

static void Double_HostToTarget(
```

```

ExternalSim *ES,
void *dst,
const char *src,
const int n,
const int dType) /* Internal Simulink data type id */
{
    int dTypeSize = esGetSizeOfDataTypeFcn(ES)(ES, dType);
    boolean_T swapBytes = esGetSwapBytes(ES);
    //assume data is host 64 bits and 32 bits in target format

    double *p_src;
    //float *p_src;
    int32_T i;
    float *tmp;
    tmp=malloc(n*4);
    p_src=(double*)src;
    //convert all src data to single

    for(i=0;i<n;i++)
    {
        tmp[i]=(float)(*p_src);
        //mexPrintf("the matlab double is %f, float is %f\n",*p_src,tmp[i]);
        p_src++;
    }
    Single_HostToTarget(ES,dst,(char*)tmp,n,dType);
    tmp[0]=to_leee(*(int*)dst);
    //mexPrintf("the target data is %f\n",tmp[0]);

    free(tmp);
}

extern int Convert_Status;

static void Double_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    int dTypeSize = esGetSizeOfDataTypeFcn(ES)(ES, dType);
    boolean_T swapBytes = esGetSwapBytes(ES);
    //assume data is host 64 bits and 32 bits in target format

    double *p_dst;
    int32_T i;
    float *tmp;
    tmp=malloc(n*4);

    if (Convert_Status)
    {
        //no need to do conversion, just copy over data
        memcpy(dst,src,n*8);
        //mexPrintf("call to convert double target to host, number"
        //          " of conversions %d \n"
        //          "src data is %f , dst data is %f\n",n,*(double*)src,*(double*)dst);
    }
    else
    {
        p_dst=(double*)dst;

        //convert all src data to single in host format
        //mexPrintf("call to convert double target to host, number of conversions %d \n",n);
        Single_TargetToHost(ES,(char*)tmp,src,n,dType);
    }
}

```

```
        for(i=0;i<n;i++)
        {
            *p_dst=(double)tmp[i];
            //mexPrintf("number as float is %f, number as double is %f\n",tmp[i],*p_dst);
            p_dst++;
        }
    }
    free(tmp);
    //mexPrintf("call to double to host passed \n");
}

static void Int8_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    //target has 32 bit format host 8 bit format

    int32_T *p_src;
    int8_T *p_dst;
    int32_T i;

    p_src=(int32_T*)src;
    p_dst=(int8_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(int8_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Int16_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    //target has 32 bit format host 8 bit format

    int32_T *p_src;
    int16_T *p_dst;
    int32_T i;

    p_src=(int32_T*)src;
    p_dst=(int16_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(int16_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Int32_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    //can optimize with memcpy target and host same format
```



```
int32_T *p_src;
int32_T *p_dst;
int32_T i;

p_src=(int32_T*)src;
p_dst=(int32_T*)dst;
for(i=0;i<n;i++)
{
    *p_dst=*p_src;
    p_src++;
    p_dst++;
}

static void Uint8_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    //target has 32 bit format host 8 bit format

    uint32_T *p_src;
    uint8_T *p_dst;
    int32_T i;

    p_src=(uint32_T*)src;
    p_dst=(uint8_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(uint8_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Uint16_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    uint32_T *p_src;
    uint16_T *p_dst;
    int32_T i;

    p_src=(uint32_T*)src;
    p_dst=(uint16_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(uint16_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Uint32_TargetToHost(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* Internal Simulink data type id */
{
    uint32_T *p_src;
    uint32_T *p_dst;
    int32_T i;
```

```
p_src=(uint32_T*)src;
p_dst=(uint32_T*)dst;
for(l=0;i<n;i++)
{
    *p_dst=*p_src;
    p_src++;
    p_dst++;
}
}

static void Int8_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    //target has 32 bit format host 8 bit format

    int32_T *p_dst;
    int8_T *p_src;
    int32_T i;

    p_src=(int8_T*)src;
    p_dst=(int32_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(int32_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Int16_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    int32_T *p_dst;
    int16_T *p_src;
    int32_T i;

    p_src=(int16_T*)src;
    p_dst=(int32_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(int32_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Int32_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* internal Simulink data type id */
{
    int32_T *p_dst;
```

```
int32_T *p_src;
int32_T i;

p_src=(int32_T*)src;
p_dst=(int32_T*)dst;
for(i=0;i<n;i++)
{
    *p_dst=*p_src;
    //      mexPrintf("matlab data is %d, target data is %d\n",*p_src,*p_dst);

    p_src++;
    p_dst++;
}
}

static void Uint8_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* Internal Simulink data type id */
{
    //target has 32 bit format host 8 bit format

    uint8_T *p_src;
    uint32_T *p_dst;
    int32_T i;

    p_src=(uint8_T*)src;
    p_dst=(uint32_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(uint32_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Uint16_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* Internal Simulink data type id */
{
    uint16_T *p_src;
    uint32_T *p_dst;
    int32_T i;

    p_src=(uint16_T*)src;
    p_dst=(uint32_T*)dst;
    for(i=0;i<n;i++)
    {
        *p_dst=(uint32_T)(*p_src);
        p_src++;
        p_dst++;
    }
}

static void Uint32_HostToTarget(
    ExternalSim *ES,
    void *dst,
    const char *src,
    const int n,
    const int dType) /* Internal Simulink data type id */
{

```

```

uint32_T *p_src;
uint32_T *p_dst;
int32_T i;

p_src=(uint32_T*)src;
p_dst=(uint32_T*)dst;
for(i=0;i<n;i++)
{
    *p_dst=*p_src;
    p_src++;
    p_dst++;
}
}

/* [EOF] ext_util.c */

```

D.2. SYSTEM TARGET FILE

```

%% SYSTLC: Generic Real-Time Target for PC32 \
%%   TMF: pc32.tmf MAKE: make_rtw EXTMODE: ext_comm_c3x
%%
%selectfile NULL_FILE

%assign MatFileLogging = 1
%assign TargetType = "RT"
%assign Language = "C"
%assign DSP32=1
%assign BlockIOSignals=0
%include "codegenentry.tlc"

%include "codegenentry.tlc"

%% The contents between 'BEGIN_RTW_OPTIONS' and 'END_RTW_OPTIONS' are strictly
%% written by the standard format. We need to use this structure in RTW
%% options GUI function rtwoptionsdlg.m file.
%%
/%
BEGIN_RTW_OPTIONS

rtwoptions(1).prompt      = 'MAT-file variable name modifier';
rtwoptions(1).type        = 'Popup';
rtwoptions(1).default     = 'rt_';
rtwoptions(1).popupstrings = 'rt_|rt|none';
rtwoptions(1).tlcvariable = 'LogVarNameModifier';
rtwoptions(1).tooltip     = ['prefix rt_ to variable name,', sprintf('\n'). ...
    'append _rt to variable name,', sprintf('\n'), 'or no modification'];

rtwoptions(2).prompt      = 'External mode';
rtwoptions(2).type        = 'Checkbox';
rtwoptions(2).default     = 'on';
rtwoptions(2).tlcvariable = 'ExtMode';
rtwoptions(2).makevariable = 'EXT_MODE';
rtwoptions(2).tooltip     = ['Adds TCP/IP communication support for', ...
    'use with', sprintf('\n'), 'Simulink external mode'];

rtwoptions(3).prompt      = 'Function Management';
rtwoptions(3).type        = 'Popup';
rtwoptions(3).popupstrings = {'None'|Function Splitting|File ', ...
    'Splitting|Function and File Splitting'];
rtwoptions(3).default     = 'None';
rtwoptions(3).tlcvariable = '';
rtwoptions(3).tooltip     = 'Limit size of generated files and functions';
rtwoptions(3).callback     = 'callback_function_management';
rtwoptions(3).opencallback = ({userData = get(gcbbf, "UserData");' ...
    'hModel_local = userData.model;' ...
    'get_value_of_fields(hModel_local,dialogFig,"open");' ...
    'tmp = compute_value_from_rtwoptions(hModel_local);' ...

```

```

'o = findobj(dialogFig, "Tag", "Function Management_PopupFieldTag");' ...
'set (o, "Value", tmp); callback_function_management(dialogFig);'];
rtwoptions(3).closecallback = ([ 'userData = get(gcbox, "UserData");' ...
'hModel_local = userData.model;' ...
'get_value_of_fields(hModel_local, dialogFig, "close");']);

rtwoptions(4).prompt    = 'Function Split Threshold';
rtwoptions(4).type      = 'Edit';
rtwoptions(4).default   = '200';
rtwoptions(4).tlcvariable = 'FunctionSizeThreshold';
rtwoptions(4).tooltip   = ['Split the generated functions after specified threshold.'];
rtwoptions(4).enable    = 'off';

rtwoptions(5).prompt    = 'File Split Threshold';
rtwoptions(5).type      = 'Edit';
rtwoptions(5).default   = '5000';
rtwoptions(5).tlcvariable = 'FileSizeThreshold';
rtwoptions(5).tooltip   = ['Split the generated files after specified threshold.'];
rtwoptions(5).enable    = 'off';

rtwoptions(6).prompt    = 'Server name';
rtwoptions(6).type      = 'Edit';
rtwoptions(6).default   = 'magash';
rtwoptions(6).tlcvariable = 'server_name';
rtwoptions(6).makevariable = 'SERVER_NAME';
rtwoptions(6).tooltip   = ['Enter name of server computer'];

rtwoptions(7).prompt    = 'Port Number';
rtwoptions(7).type      = 'Edit';
rtwoptions(7).default   = '700';
rtwoptions(7).tlcvariable = 'server_port';
rtwoptions(7).makevariable = 'SERVER_PORT';
rtwoptions(7).tooltip   = ['Enter port number of server computer'];

rtwoptions(8).prompt    = 'Quick Build Project';
rtwoptions(8).type      = 'Checkbox';
rtwoptions(8).default   = 'on';
rtwoptions(8).tlcvariable = 'QUICK_BUILD';
rtwoptions(8).makevariable = 'QUICK_BUILD';
rtwoptions(8).tooltip   = ['Used to speedup repetitive builds :', ...
'', sprintf('\n'), 'Warning!! Disable if mods to RTW internals are being made'];

rtwoptions(9).prompt    = 'Heap Size';
rtwoptions(9).type      = 'Edit';
rtwoptions(9).default   = '0x10000';
rtwoptions(9).tlcvariable = 'heap_size';
rtwoptions(9).makevariable = 'HEAP_SIZE';
rtwoptions(9).tooltip   = ['Enter heap size for compiler'];

rtwoptions(10).prompt   = 'Stack Size';
rtwoptions(10).type     = 'Edit';
rtwoptions(10).default  = '0x5000';
rtwoptions(10).tlcvariable = 'stack_size';
rtwoptions(10).makevariable = 'STACK_SIZE';
rtwoptions(10).tooltip  = ['Enter stack size for compiler'];

END_RTW_OPTIONS
%/

```

D.3. SYSTEM TEMPLATE MAKE FILE

```

#*****
SYS_TARGET_FILE = grt_c3x.tlc
MAKE            = |>MATLAB_ROOT<|\rtw\c3x\bin\gmake_3_71
HOST            = PC

```

```
BUILD      = yes
DOWNLOAD   = yes
BUILD_SUCCESS = Completed
DOWNLOAD_SUCCESS = Downloaded

#----- Customization Macros -----
#
# The following set of macros are customized by the make_rt program.
#
MODEL      = |>MODEL_NAME<|
MODEL_MODULES = |>MODEL_MODULES<|
MODEL_MODULES_OBJ = |>MODEL_MODULES_OBJ<|
MAKEFILE    = |>MAKEFILE_NAME<|
MATLAB_ROOT = |>MATLAB_ROOT<|
MATLAB_BIN  = |>MATLAB_BIN<|
S_FUNCTIONS = |>S_FUNCTIONS<|
S_FUNCTIONS_OBJ = |>S_FUNCTIONS_OBJ<|
SOLVER      = |>SOLVER<|
SOLVER_OBJ  = |>SOLVER_OBJ<|
NUMST       = |>NUMST<|
TID01EQ     = |>TID01EQ<|
NCSTATES    = |>NCSTATES<|
BUILDDARGS  = |>BUILDDARGS<|
COMPUTER    = |>COMPUTER<|
SERVER_NAME = |>SERVER_NAME<|
SERVER_PORT = |>SERVER_PORT<|
QUICK       = |>QUICK_BUILD<|
HEAP        = |>HEAP_SIZE<|
STACK       = |>STACK_SIZE<|

QUICK:=$(strip $(QUICK))
#----- II PC32 Definition -----
#
BOARD_TYPE   = PC32
DSP_FAMILY   = 30
COMPILER     = TI_FPC
#----- TI Tools -----
#
# You may need to modify the TI_ROOT if you have installed the
# Texas Instrument Compiler in a different location.
#
#set old_c=1 for compiler 4.7 or 0 for compiler 5.10
old_c = 0
ifeq ($(old_c),1)
TI_ROOT = c:\fltc
else
TI_ROOT = c:\c3xtools\bin
endif
#TI_ROOT = c:\ti\eval3x\c3x4x\cgtools
TI_FLAGS = -v$(DSP_FAMILY)

CC = $(TI_ROOT)\cl30
#CC = $(TI_ROOT)\bin\cl30

LD = $(TI_ROOT)\lnk30

#----- II Tools -----
#
II_DIR = c:\pc32cc
#II_DIR is the dir were II Zuma tool set is installed
II_ROOT = $(MATLAB_ROOT)\rtw\clii
II_COMPILER = $(II_ROOT)\ti_fpc

II_CMD = $(II_COMPILER)\iIPC32.cmd
II_BOOT = $(II_COMPILER)\vectors.obj

ifeq ($(old_c),1)
```

```

II_INCLUDES = $(II_DIR)\include\target;
else
II_INCLUDES = $(II_DIR)\include\target; c:\c3xtools\include;c:\c3xtools\lib
endif

#----- DOWNLOAD Tool -----

PC32_DOWNLOAD = $(II_ROOT)\bin\auto_download.exe

#----- Include Path -----

MATLAB_INCLUDES = \
$(MATLAB_ROOT)\simulink\include; \
$(MATLAB_ROOT)\extern\include; \
$(MATLAB_ROOT)\rtw\c\src; \
$(MATLAB_ROOT)\rtw\c\iitl_fpc; \
$(MATLAB_ROOT)\rtw\c\libsrc;

TI_INCLUDES = $(TI_ROOT);

INCLUDES = .; $(MATLAB_INCLUDES) $(TI_INCLUDES) $(II_INCLUDES)

#----- C Flags -----

# Required Options
REQ_OPTS = -s -ma -mf -g $(TI_FLAGS) -pf -q -eo .o$(DSP_FAMILY)

ifneq ($(old_c),1)
REQ_OPTS := $(REQ_OPTS) -ml -o0 -x2 -op0 -on1
endif

# Optimization Options
OPT_OPTS = -x
# Debug Options
DBG_OPTS =

CC_OPTS = $(REQ_OPTS) $(OPT_OPTS) $(DBG_OPTS) -dio_$(IO) \
          -dTMR0_$(TMR0) -dUPLO_$(UPLO) -dDSP32 -dEXT_MODE -dio_ENABLE \
          -dTARGET_SYSTEM -dVERBOSE

CPP_REQ_DEFINES = -dMODEL=$(MODEL) -dRT -dNUMST=$(NUMST) \
                  -dTID01EQ=$(TID01EQ) -dNCSTATES=$(NCSTATES)

CFLAGS = $(CC_OPTS) $(CPP_REQ_DEFINES) $(CPP_DEFINES)

LDFLAGS := -g -x -cr -heap $(HEAP) -stack $(STACK) $(II_BOOT) -m $(MODEL).map

#----- Source Files -----

REQ_SRCS = ii_pc32.c $(MODEL).c rt_slm.c rtwlog_c3x.c pc32_gdm.c rt_matrx.c
updown_c3x.c ext_srv_c3x.c
OPT_SRCS =
S_FCN_SRCS = $(S_FUNCTIONS)
INT_SRCS = $(SOLVER)

#PC32_OBJS = $(MATLAB_ROOT)\rtw\c\iitl_fpc\pc32func.o30
REQ_OBJS = $(REQ_SRCS:.c=.o$(DSP_FAMILY))
OPT_OBJS = $(OPT_SRCS:.c=.o$(DSP_FAMILY))
S_FCN_OBJS = $(S_FCN_SRCS:.c=.o$(DSP_FAMILY))
INT_OBJS = $(INT_SRCS:.c=.o$(DSP_FAMILY))
OBJS = $(REQ_OBJS) $(OPT_OBJS) $(S_FCN_OBJS) $(INT_OBJS)
OBJS1 = $(REQ_OBJS) $(OPT_OBJS)
OBJS2 = $(S_FCN_OBJS) $(INT_OBJS)

```

```

PROGRAM      = $(MODEL).out

#----- Exported Environment Variables -----
#
# Because of the 128 character command line length limitations in DOS, we
# use environment variables to pass additional information to the
# Compiler and Linker
#
C_OPTION := $(CFLAGS)
C_DIR    := $(INCLUDES); $(C_DIR)
C_MODE   = PROTECTED

#----- Rules -----

$(PROGRAM) : $(OBJS)
    echo $(OBJS1) > $(MODEL).lin
ifneq ($(strip $(OBJS2)),)
    echo $(OBJS2) >> $(MODEL).lin
endif
    echo $(IL_CMD) >> $(MODEL).lin
    $(LD) $(LDFLAGS) -o $@ $(MODEL).lin
    del $(MODEL).lin
    echo $(BUILD_SUCCESS) $(PROGRAM)
# Compile existing code if it exists in current dir
%.o$(DSP_FAMILY) : %.c
    $(CC) $<

# Call to PC32 rt_main.c
# edit mags using token for matlab root
%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\cli\rti_fpc\%.c
    $(CC) $<

# Call to simulink files
%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\simulink\src\%.c
    $(CC) $<

# Call compile RTW files
%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\c\src\%.c
    $(CC) $<

%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\c\libsrc\%.c
    $(CC) $<

#----- Rule for Downloading to Target -----

download :
#   del $(MODEL).lin
#   del $(MODEL).c
#   del $(MODEL).h
#   del $(MODEL).map
#   del $(MODEL).o30
#   del $(MODEL).prm
#   del $(MODEL).reg
    $(PC32_DOWNLOAD) -f$(PROGRAM) -s$(SERVER_NAME) -p$(SERVER_PORT)
    echo $(DOWNLOAD_SUCCESS) $(PROGRAM)

#----- Dependencies -----
pc32_grtm.o$(DSP_FAMILY) : $(MODEL).c

# ifneq $(QUICK),1)
$(OBJS)      : $(MAKEFILE)
# endif
# ii_pc32.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\cli\rti_fpc\il_comms.h \
#                        $(MATLAB_ROOT)\rtw\c\src\ext_srv_c3x.h

```



```
#ext_srv_c3x.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\cli\iti_fpc\cli_comms.h \
# $(MATLAB_ROOT)\rtw\clsrc\ext_srv_c3x.h
```

D.4. DEVICE DRIVER FILES

D.4.1 ADC BLOCKS

```
%%
%%
%% Abstract:
%%   TLC file for the PC32 A/D Block.
%%   This file is used to generate code to read
%%   values from the A/D converters and scale them to +-10.
%% Author:
%%   Adam Stylo
%% Date:
%%   98/11/03
%% Revised By Magash Pillay
%% 2000/01/20

%%
%implements "pc32_ad" "C"

#include "iilib.tlc"

%function BlockInstanceSetup(block, system) void
%% Only allow 1 Instance of the A/D block
%if IEXISTS("Rt_pc32ad")
%assign ::Rt_pc32ad = 1
%else
%error Only 1 PC32adn block is allowed in the model.
%endif
%endfunction %% BlockInstanceSetup

%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
/* read in the corrected values from A/D and scale to +-10 */
{
%<LibBlockOutputSignal(0,"",0)>=read_adc(BASEBOARD, 0)/(3276.7);
%<LibBlockOutputSignal(0,"",1)>=read_adc(BASEBOARD, 1)/(3276.7);
%<LibBlockOutputSignal(0,"",2)>=read_adc(BASEBOARD, 2)/(3276.7);
%<LibBlockOutputSignal(0,"",3)>=read_adc(BASEBOARD, 3)/(3276.7);
}
%endfunction %% Outputs
```

D.4.2 DAC BLOCK

```
%%
%%
%% Abstract:
%%   TLC file for the PC32 D/A Block.
%%   This file is used to generate code to write
%%   values to the D/A converters. At termination
%%   all outputs are set to 0.
%% Author:
%%   Adam Stylo
%% Date:
%%   98/11/03
%% Revised By Magash Pillay
%% 2000/01/20
%%

%implements "pc32_da" "C"

#include "iilib.tlc"
```

```

%function BlockInstanceSetup(block, system) void

%% Only allow 1 Instance of the D/A block
%if !EXISTS("Rt_pc32da")
    %assign ::Rt_pc32da = 1
%else
    %error Only 1 PC32dan block is allowed in the model.
%endif
%endfunction %% BlockInstanceSetup

%% Function: Outputs
=====
%%
%% Abstract:
%%     Generate inlined code to perform one D/A conversion.
%%
%function Outputs(block, system) Output

/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
/* Start an output conversion */
{
    write_dac(BASEBOARD, 0, %<LibBlockInputSignal(0,"",0)>*(3276.7));
    convert_dac(BASEBOARD, 0);
    write_dac(BASEBOARD, 1, %<LibBlockInputSignal(0,"",1)>*(3276.7));
    convert_dac(BASEBOARD, 1);
    write_dac(BASEBOARD, 2, %<LibBlockInputSignal(0,"",2)>*(3276.7));
    convert_dac(BASEBOARD, 2);
    write_dac(BASEBOARD, 3, %<LibBlockInputSignal(0,"",3)>*(3276.7));
    convert_dac(BASEBOARD, 3);
}
%endfunction %% Outputs

%openfile buffer
/* reset D/A outputs to 0 at termination. */
write_dac(BASEBOARD, 0, 0);
convert_dac(BASEBOARD, 0);
write_dac(BASEBOARD, 1, 0);
convert_dac(BASEBOARD, 1);
write_dac(BASEBOARD, 2, 0);
convert_dac(BASEBOARD, 2);
write_dac(BASEBOARD, 3, 0);
convert_dac(BASEBOARD, 3);
%closefile buffer
%<LibMdlTerminateCustomCode(buffer, "trailer")>

```

%% EOF: PC32dan.tlc

D.4.3 PWM BLOCK

```

%%
%%
%% Abstract:
%%     TLC file for the PWM Block. Generates code used to
%%     control a PWM/Tacho add on card.
%% Author:
%%     Adam Stylo
%% Date:
%%     98/11/03
%%
%% Revised By Magash Pillay
%% 2000/01/20

%implements "pwmblock" "C"

%include "iilib.tlc"

```

```
%assign ::Vortl = LibBlockParameter(P1,"",0)
%assign ::CtrlMode = LibBlockParameter(P2,"",0)

%function BlockInstanceSetup(block, system) void
%% Only allow 1 pwm block
%if EXISTS("IIPWMBlockSeen")
%assign errTxt = "Only 1 Interrupt block is allowed in " ...
"model: %<CompiledModel.Name>,"
%exit RTW Fatal: %<errTxt>
%else
%assign ::IIPWMBlockSeen = 1
%endif

%openfile buffer
#define Status_word (volatile int*) 0x81a001
#define Data_word (volatile int*) 0x81a000

#define TAUS (0)
#define TTOT (0)
#define TMIN (0)
int VORTL,TSTART;

void pollpwm( void )
{
    while (*(Status_word) & 0x1);
}

%closefile buffer
%<LibCacheDefine(buffer)>

%openfile buffer
#ifdef IO_ENABLE
printf("Initializing PWM Block ....\n");
#endif

VORTL = (int)(%<Vortl>);
TSTART = ((int)(512-(322/(VORTL+1))));

*IOBCR = 0x58;
*(Status_word) = 128; /* set up 16 bit addressing mode */
*(Status_word) = 128; /* set address to zero */

pollpwm();
*(Data_word) = 0; /* Ua */
pollpwm();
*(Data_word) = 0; /* Ub */
pollpwm();
*(Data_word) = 0; /* phi1 */
pollpwm();
*(Data_word) = 0; /* dphi1 */
pollpwm();
*(Data_word) = 0; /* phi0 */
pollpwm();
*(Data_word) = 0; /* dphi0 */
pollpwm();
*(Data_word) = 0; /* phiadd */
pollpwm();
*(Data_word) = 0; /* unused */
pollpwm();
*(Data_word) = TAUS; /* turn off time */
pollpwm();
*(Data_word) = TTOT; /* dead band */
pollpwm();
*(Data_word) = TMIN; /* turn on time */
pollpwm();
*(Data_word) = VORTL; /* switching frequency scale value */
```

```

pollpwm();
*(Data_word) = TSTART; /* start of processing cycle */

*(Status_word) = 129;

%closefile buffer
%<LibMdiStartCustomCode(buffer, "trailer")>

%openfile buffer
/* dissable the PWM board at terminate */
*(Status_word) = 0;
*(Status_word) = 0;
%closefile buffer
%<LibMdiTerminateCustomCode(buffer, "trailer")>
%endfunction

%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */

{
    *(Status_word) = 129;
    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 0)>;
    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 1)>;

    if ((int)%<CtrlMode> == 1) /* skip three values to write frequency */
    {pollpwm();
      *(Status_word) = 897;}

    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 2)>;
}
%endfunction %% Outputs

```

D.4.4 ASYNCHRONOUS INTERRUPT SUPPORT

mod on 17/04/2000

problem with formatted c file produced

```

%%
%%
%% Abstract:
%%   TLC file for the PC32 Asynchronous Interrupt Block.
%%   This file is used to generate code to support asynchronous
%%   interrupts on the PC32.
%% Author:
%%   Adam Stylo
%% Date:
%%   98/11/03
%% Revised By Magash Pillay
%% 2000/01/20
%%

```

%implements "iinterrupt" "C"

%Include "iilib.tlc"

```

%% Function: BlockInstanceSetup =====
%% Abstract:
%%   Find all the function-call subsystems that are attached to the
%%   interrupt block and hook-in the necessary code for each routine.
%%   This function
%%
%%   o Connect each ISR in the model's start function.
%%
%%   o Enable each ISR at the bottom of the model's start function.
%%

```

```

%%      o Disable each ISR in the model's terminate function.
%%
%%      o Save floating point context in the ISR's critical code section

%assign ::EITrigg = LibBlockParameter(P1,"",0)
%assign ::Tmr0freq = LibBlockParameter(P4,"",0)
%assign ::Tmr1freq = LibBlockParameter(P5,"",0)
%assign ::Trigg_src0 = LibBlockParameter(P2,"",0)
%assign ::Trigg_src1 = LibBlockParameter(P3,"",0)

%function BlockInstanceSetup(block, system) void
%% Only allow 1 interrupt block
%if EXISTS("InterruptBlockSeen")
    %assign errTxt = "Only 1 Interrupt block is allowed in " ...
        "model: %<CompiledModel.Name>."
    %exit RTW Fatal: %<errTxt>
%else
    %assign ::InterruptBlockSeen = 1
%endif

%openfile buffer
#ifdef IO_ENABLE
    printf("Connecting Interrupts\n");
#endif
/* Make ints edge triggered only if required */
if ((int)%<EITrigg> == 1)
    {asm ("        OR 4000h,ST");
    #ifdef IO_ENABLE
        printf("EI0 - EI3 Set to edge triggered\n");
    #endif
    }
else
    {asm ("        ANDN 4000h,ST");
    #ifdef IO_ENABLE
        printf("EI0 - EI3 Set to level triggered\n");
    #endif
    }
%closefile buffer
%<LibMdlStartCustomCode(buffer, "header")>

%openfile buffer
/* define addresses for control registers */
#define GC_CTRL0 (volatile int*) 0x808020
#define GC_CTRL1 (volatile int*) 0x808030
/* Define a interrupt_block */
#define INTERRUPT_BLOCK
#include "pc32main_comms.h"

%closefile buffer
%<LibHeaderFileCustomCode(buffer,"trailer")>

%foreach callIdx = NumSFcnSysOutputCalls

%% Get downstream block if there is one
%if "%<SFcnSystemOutputCall[callIdx].BlockToCall>" != "unconnected"
%assign ssSysIdx = SFcnSystemOutputCall[callIdx].BlockToCall[0]
%assign ssBlkIdx = SFcnSystemOutputCall[callIdx].BlockToCall[1]
%assign ssBlock = CompiledModel.System[ssSysIdx].Block[ssBlkIdx]
%% Check to see if this is a direct connection
%if (ssBlock.ControllInputPort.Width != 1)
%assign errTxt = "The ll Interrupt block '%<block.Name>' " ...
    "outputs must be directly connected to one function-call subsystem. " ...
    "The destination function-call subsystem block '%<ssBlock.Name>' " ...
    "has other inputs."
%exit RTW Fatal: %<errTxt>
%endif
%% Assume it is a subsystem block(Simulink checked for a f-c subsys already).

```

```

%assign isrSystem = System[ssBlock.ParamSettings.SystemIdx]

%<LibForceOutputUpdateFcn(isrSystem)>
%% NO need to redefine since function calls are from dummy interrupt functions
%openfile buffer
void c_int0%<callIdx+1>(void);
%closefile buffer
%<LibCacheDefine(buffer)>

%openfile buffer
/*notes on P7
range 1 to 6 ,coresspondes to interrupt number.
*/
%if callIdx == ( %<LibBlockParameterValue(P7,0)>-1)
    %%call system function call not using TI interrupt convention
    %if callIdx==4
        %openfile temp
        #define TMR0_BASE_RATE
        %closefile temp
        %<LibCacheDefine(temp)>
        %endif
        void Base_Rate_Function()
        {
            %<isrSystem.OutputUpdateFcn>(rtS,1,0);
        }
    %else
        /*
        ISR for : %<ssBlock.Name>
        */
        %if callIdx < 9
            void c_int0%<callIdx+1>()
            {
                /* call subsystem block
                Using TID =0 since, single tasking simulation
                */
                %<isrSystem.OutputUpdateFcn>(rtS,1,0);
            }

            %else
                void c_int%<callIdx+1>()
                {
                    /* call subsystem block
                    Using TID =0 since, single tasking simulation
                    */
                    %<isrSystem.OutputUpdateFcn>(rtS,1,0);
                }
            %endif
        %endif
    %closefile buffer
    %<LibSourceFileCustomCode(buffer,"trailer")>

    %openfile buffer
    %% controlPortIdx will never get used when only one f-c control input
    /* use base tid Inside an ISR for any blocks accessing task time*/
    #define %<tTID> 0

    %closefile buffer
    %<LibSystemOutputCustomCode(isrSystem, buffer, "declaration")>
    %openfile buffer
    #undef %<tTID>
    %closefile buffer
    %<LibSystemOutputCustomCode(isrSystem, buffer, "trailer")>

    %% Connect the ISR In the model's start function

```

```

%openfile buffer
/* connect ISR system: %<ssBlock.Name> */
//int registration
%if callIdx == ( %<LibBlockParameterValue(P7,0)>-1)
{
    /* assign rt_onestep from real time kernel to interrupt */
    install_int_vector(rtOneStep,(int)%<LibBlockParameter(P6,"", "", callIdx)>);
    enable_interrupt((int)%<LibBlockParameterValue(P6,callIdx)>-1);
    #ifdef IO_ENABLE
        printf("Vectlor installed for INT #%%d, Base rate Interrupt.\n"
            ,(int)%<LibBlockParameter(P6,"", "", callIdx)>);
    #endif
}
%else
{
    %if callIdx < 9
        install_int_vector(c_int0%<callIdx+1>,(int)%<LibBlockParameterValue(P6,callIdx)>);
    %else
        install_int_vector(c_int%<callIdx+1>,(int)%<LibBlockParameterValue(P6,callIdx)>);
    %endif
    enable_interrupt((int)%<LibBlockParameterValue(P6,callIdx)>-1);
    #ifdef IO_ENABLE
        printf("Vectlor installed for INT #%%d.\n", (int)%<LibBlockParameter(P6,"", "", callIdx)>);
    #endif
}
%endif

%closefile buffer
%<LibMdlStartCustomCode(buffer, "trailer")>

    %openfile buffer
/* disconnect ISR system: %<ssBlock.Name> */
if (%<LibBlockParameter(P6,"", "", callIdx)>==9)
{
    /* only disconnect timer0 if it was set up here */
    #ifndef TMR0_YES
        disable_interrupt((int)%<LibBlockParameter(P6,"", "", callIdx)>-1);
        deinstall_int_vector((int)%<LibBlockParameter(P6,"", "", callIdx)>);
    #ifdef IO_ENABLE
        printf("INT #%%d disabled.\n", (int)%<LibBlockParameter(P6,"", "", callIdx)>);
    #endif
    #endif
}
else
{
    disable_interrupt((int)%<LibBlockParameter(P6,"", "", callIdx)>-1);
    deinstall_int_vector((int)%<LibBlockParameter(P6,"", "", callIdx)>);
    #ifdef IO_ENABLE
        printf("INT #%%d disabled.\n", (int)%<LibBlockParameter(P6,"", "", callIdx)>);
    #endif
}
}
%closefile buffer
%<LibMdlTerminateCustomCode(buffer, "trailer")>
%else %% The element is not connected to anything
%assign wrnTxt = "No code will be generated for ISR %<callIdx> "\
"since it is not connected to anything."
%warning %<wrnTxt>
%endif
%endforeach

%% Setup timers
%openfile buffer
Interrupt_Block=1;
if ((int)%<Trigg_src0> == 2)
{
    *GC_CTRL0 = 0x6c3;
    #ifdef IO_ENABLE
        printf("TCLK0 driven by Timer 0.\n");
    #endif
}

```

```
#endif
}
else
{ *GC_CTRL0 = 0x6c0;
#ifdef IO_ENABLE
printf("TCLK0 driven externaly.\n");
#endif
}
if ((int)%<Trigg_src1> == 2)
{ *GC_CTRL1 = 0x6c3;
#ifdef IO_ENABLE
printf("TCLK1 driven by Timer 1.\n");
#endif
}
else
{ *GC_CTRL1 = 0x6c0;
#ifdef IO_ENABLE
printf("TCLK1 driven externaly.\n");
#endif
}

#ifdef TMR0_BASE_RATE
/*simulation step size take precedence over TMR0 freq when set as base rate */
timer(0, (int)(1.0 / ssGetStepSize(rtS)));
#else
/* Only change Timer 0 settings if it isn't used for base sampling rate */
timer(0, (int)%<Tmr0freq>);
#endif
timer(1, (int)%<Tmr1freq>);

#ifdef IO_ENABLE
printf("Interrupts Connected ,Waiting for start Signal...\n");
#endif
%closefile buffer
%<LibMdlStartCustomCode(buffer, "header")>
%endfunction

%% [EOF] iiiinterrupt.tlc
```


APPENDIX E: LISTING OF CODE FOR THE RADE ADC64

E.1. SYSTEM TARGET FILE

```

%% SYSTLC: Generic Real-Time Target ADC64 \
%%   TMF: adc64.tmf MAKE: make_rtw IO=DISABLE EXTMODE: ext_comm_c3x
%%
%selectfile NULL_FILE
%addincludepath "c:\matlab\rtw\cui\devices"

%assign MatFileLogging = 1
%assign TargetType = "RT"
%assign Language = "C"
%assign DSP32=1
%assign BlockIOSignals=0
%include "codegenentry.tlc"

%% The contents between 'BEGIN_RTW_OPTIONS' and 'END_RTW_OPTIONS' are
strictly
%% written by the standard format. We need to use this structure in RTW
%% options GUI function rtwoptionsdlg.m file.
%%
/%
BEGIN_RTW_OPTIONS

rtwoptions(1).prompt      = 'MAT-file variable name modifier';
rtwoptions(1).type        = 'Popup';
rtwoptions(1).default     = 'rt_';
rtwoptions(1).popupstrings = 'rt_|rt|none';
rtwoptions(1).tlcvariable = 'LogVarNameModifier';
rtwoptions(1).tooltip     = ['prefix rt_ to variable name,', sprintf('\n'), ...
    'append _rt to variable name,', sprintf('\n'), 'or no modification'];

rtwoptions(2).prompt      = 'External mode';
rtwoptions(2).type        = 'Checkbox';
rtwoptions(2).default     = 'on';
rtwoptions(2).tlcvariable = 'ExtMode';
rtwoptions(2).makevariable = 'EXT_MODE';
rtwoptions(2).tooltip     = ['Adds TCP/IP communication support for', ...
    'use with', sprintf('\n'), 'Simulink external mode'];

rtwoptions(3).prompt      = 'Function Management';
rtwoptions(3).type        = 'Popup';
rtwoptions(3).popupstrings = ['None|Function Splitting|File ', ...
    'Splitting|Function and File Splitting'];
rtwoptions(3).default     = 'None';
rtwoptions(3).tlcvariable = '';
rtwoptions(3).tooltip     = 'Limit size of generated files and functions';
rtwoptions(3).callback    = 'callback_function_management';
rtwoptions(3).opencallback = ([ 'userData = get(gcbbf, "UserData");' ...
    'hModel_local = userData.model;' ...
    'get_value_of_fields(hModel_local, dialogFig, "open");' ...
    'tmp = compute_value_from_rtwoptions(hModel_local);' ...
    'o = findobj(dialogFig, "Tag", "Function Management_PopupFieldTag");' ...

```

```

    'set (o, "Value", tmp); callback_function_management(dialogFig);));
rtwoptions(3).closecallback = ([ 'userData = get(gcbf, "UserData");' ...
    'hModel_local = userData.model;' ...
    'get_value_of_fields(hModel_local,dialogFig,"close");']);

rtwoptions(4).prompt    = 'Function Split Threshold';
rtwoptions(4).type      = 'Edit';
rtwoptions(4).default    = '200';
rtwoptions(4).tlcvariable = 'FunctionSizeThreshold';
rtwoptions(4).tooltip    = ['Split the generated functions after specified threshold.'];
rtwoptions(4).enable     = 'off';

rtwoptions(5).prompt    = 'File Split Threshold';
rtwoptions(5).type      = 'Edit';
rtwoptions(5).default    = '5000';
rtwoptions(5).tlcvariable = 'FileSizeThreshold';
rtwoptions(5).tooltip    = ['Split the generated files after specified threshold.'];
rtwoptions(5).enable     = 'off';

rtwoptions(6).prompt    = 'Server name';
rtwoptions(6).type      = 'Edit';
rtwoptions(6).default    = 'magash';
rtwoptions(6).tlcvariable = 'server_name';
rtwoptions(6).makevariable = 'SERVER_NAME';
rtwoptions(6).tooltip    = ['Enter name of server computer'];

rtwoptions(7).prompt    = 'Port Number';
rtwoptions(7).type      = 'Edit';
rtwoptions(7).default    = '700';
rtwoptions(7).tlcvariable = 'server_port';
rtwoptions(7).makevariable = 'SERVER_PORT';
rtwoptions(7).tooltip    = ['Enter port number of server computer'];

rtwoptions(8).prompt    = 'Quick Build Project';
rtwoptions(8).type      = 'Checkbox';
rtwoptions(8).default    = 'off';
rtwoptions(8).tlcvariable = 'QUICK_BUILD';
rtwoptions(8).makevariable = 'QUICK_BUILD';
rtwoptions(8).tooltip    = ['Used to speedup repetitive builds :', ...
    '', sprintf('\n'), 'Warning!! Disable if mods to RTW internals are being made'];

rtwoptions(9).prompt    = 'Heap Size';
rtwoptions(9).type      = 'Edit';
rtwoptions(9).default    = '0x10000';
rtwoptions(9).tlcvariable = 'heap_size';
rtwoptions(9).makevariable = 'HEAP_SIZE';
rtwoptions(9).tooltip    = ['Enter heap size for compiler'];

rtwoptions(10).prompt    = 'Stack Size';
rtwoptions(10).type      = 'Edit';
rtwoptions(10).default    = '0x5000';
rtwoptions(10).tlcvariable = 'stack_size';
rtwoptions(10).makevariable = 'STACK_SIZE';
rtwoptions(10).tooltip    = ['Enter stack size for compiler'];

```

```
END_RTW_OPTIONS
%/
```

E.2. SYSTEM TEMPLATE MAKE FILE

```
#-----
SYS_TARGET_FILE = grt_adc64.llc
MAKE             = |>MATLAB_ROOT<|rtw\c\lib\bin\gmake_3_71
HOST             = PC
BUILD            = yes
DOWNLOAD        = yes
BUILD_SUCCESS   = Completed
DOWNLOAD_SUCCESS = Downloaded

#----- Customization Macros -----
#
# The following set of macros are customized by the make_rt program.
#
MODEL            = |>MODEL_NAME<|
MODEL_MODULES    = |>MODEL_MODULES<|
MODEL_MODULES_OBJ = |>MODEL_MODULES_OBJ<|
MAKEFILE         = |>MAKEFILE_NAME<|
MATLAB_ROOT      = |>MATLAB_ROOT<|
MATLAB_BIN       = |>MATLAB_BIN<|
S_FUNCTIONS      = |>S_FUNCTIONS<|
S_FUNCTIONS_OBJ  = |>S_FUNCTIONS_OBJ<|
SOLVER           = |>SOLVER<|
SOLVER_OBJ       = |>SOLVER_OBJ<|
NUMST            = |>NUMST<|
TID01EQ          = |>TID01EQ<|
NCSTATES         = |>NCSTATES<|
BUILDARGS        = |>BUILDARGS<|
COMPUTER         = |>COMPUTER<|
SERVER_NAME      = |>SERVER_NAME<|
SERVER_PORT      = |>SERVER_PORT<|
QUICK            = |>QUICK_BUILD<|
HEAP             = |>HEAP_SIZE<|
STACK            = |>STACK_SIZE<|

QUICK:=$(strip ${QUICK})
#----- II PC32 Definition -----
#
BOARD_TYPE      = ADC64
DSP_FAMILY      = 30
COMPILER        = TI_FPC
#----- TI Tools -----
#
# You may need to modify the TI_ROOT if you have installed the
# Texas Instrument Compiler in a different location.
#
#set old_c=1 for compiler 4.7 or 0 for compiler 5.11
old_c=0
ifeq (${old_c},1)
TI_ROOT = c:\fltc
else
TI_ROOT = c:\c3xtools\bin
endif
TI_FLAGS = -v$(DSP_FAMILY)

CC = $(TI_ROOT)\cl30
LD = $(TI_ROOT)\lnk30

#----- II Tools -----
```

```

#
II_DIR    = c:\adc64cc
#II_DIR is the dir were II Zuma tool set is installed
II_ROOT   = $(MATLAB_ROOT)\rtw\c\ii\adc64
II_COMPILER = $(II_ROOT)\src

II_CMD    = $(II_COMPILER)\iiadc64.cmd
#II_BOOT   = $(II_COMPILER)\vectors.obj

ifeq ($(old_c),1)
II_INCLUDES = $(II_DIR)\include\target; $(II_DIR)\lib\target
else
II_INCLUDES = $(II_DIR)\include\target;$(II_DIR)\lib\target; c:\c3xtools\include;c:\c3xtools\lib
endif

#----- DOWNLOAD Tool -----

PC32_DOWNLOAD = $(II_ROOT)\bin\auto_download.exe
#using this location during development only
#PC32_DOWNLOAD = c:\projects\matlab_nell\auto_download\debug\auto_download.exe
#----- Include Path -----

MATLAB_INCLUDES = \
$(MATLAB_ROOT)\simulink\include; \
$(MATLAB_ROOT)\extern\include; \
$(MATLAB_ROOT)\rtw\c\src; \
$(MATLAB_ROOT)\rtw\c\ii\adc64\src; \
$(MATLAB_ROOT)\rtw\c\libsrc;

TI_INCLUDES = $(TI_ROOT);

INCLUDES = .; $(MATLAB_INCLUDES) $(TI_INCLUDES) $(II_INCLUDES)

#----- C Flags -----

# Required Options
REQ_OPTS = -s -ma -mf -g $(TI_FLAGS) -pf -q -eo -o$(DSP_FAMILY)

ifneq ($(old_c),1)
REQ_OPTS := $(REQ_OPTS) -mi -o0 -x2 -op0 -on1
endif

# Optimization Options
OPT_OPTS = -x
# Debug Options
DBG_OPTS =

CC_OPTS = $(REQ_OPTS) $(OPT_OPTS) $(DBG_OPTS) -dIO_$(IO) \
          -dTMR0_$(TMR0) -dUPLD_$(UPLD) -dDSP32 -dEXT_MODE -dIO_ENABLE \
          -dTARGET_SYSTEM -dVERBOSE -dADC64

CPP_REQ_DEFINES = -dMODEL=$(MODEL) -dRT -dNUMST=$(NUMST) \
                  -dTID01EQ=$(TID01EQ) -dNCSTATES=$(NCSTATES)

CFLAGS = $(CC_OPTS) $(CPP_REQ_DEFINES) $(CPP_DEFINES)

LDFLAGS := -g -x -c -heap $(HEAP) -stack $(STACK) $(II_BOOT) -m $(MODEL).map

#----- Source Files -----

REQ_SRCS = ii_adc64.c rt_slm.c rtwlog_c3x.c adc64_grtm.c rt_matrx.c $(MODEL).c \
          ext_srv_c3x.c updown_c3x.c
OPT_SRCS =

```

```

S_FCN_SRCS    = $(S_FUNCTIONS)
INT_SRCS      = $(SOLVER)

#PC32_OBJS    = $(MATLAB_ROOT)\rtw\c\liti_fpc\pc32func.o30
REQ_OBJS      = $(REQ_SRCS:.c=.o$(DSP_FAMILY))
OPT_OBJS      = $(OPT_SRCS:.c=.o$(DSP_FAMILY))
S_FCN_OBJS    = $(S_FCN_SRCS:.c=.o$(DSP_FAMILY))
INT_OBJS      = $(INT_SRCS:.c=.o$(DSP_FAMILY))
OBJS          = $(REQ_OBJS) $(OPT_OBJS) $(S_FCN_OBJS) $(INT_OBJS)
OBJS1         = $(REQ_OBJS) $(OPT_OBJS)
OBJS2         = $(S_FCN_OBJS) $(INT_OBJS)

PROGRAM       = $(MODEL).out

#----- Exported Environment Variables -----
#
# Because of the 128 character command line length limitations in DOS, we
# use environment variables to pass additional information to the
# Compiler and Linker
#
C_OPTION := $(CFLAGS)
C_DIR    := $(INCLUDES); $(C_DIR)
C_MODE   = PROTECTED

#----- Rules -----

$(PROGRAM) : $(OBJS)
    echo $(OBJS1) > $(MODEL).lin
    ifneq ($(strip $(OBJS2)),)
        echo $(OBJS2) >> $(MODEL).lin
    endif
    echo $(LI_CMD) >> $(MODEL).lin
    $(LD) $(LDFLAGS) -o $@ $(MODEL).lin
    del $(MODEL).lin
    echo $(BUILD_SUCCESS) $(PROGRAM)

# Compile existing code if it exists in current dir
%.o$(DSP_FAMILY) : %.c
    $(CC) $<

# edit mags using token for matlab root
%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\c\litiadc64\src\%.c
    $(CC) $<
%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\c\liti_fpc\%.c
    $(CC) $<
# Call to simulink files
%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\simulink\src\%.c
    $(CC) $<

# Call compile RTW files
%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\c\src\%.c
    $(CC) $<

%.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\c\libsrc\%.c
    $(CC) $<

#----- Rule for Downloading to Target -----

download :
# del pc32main.o30
# del $(MODEL).lin
# del $(MODEL).c
# del $(MODEL).h
# del $(MODEL).map

```

```
# del $(MODEL).o30
# del $(MODEL).prm
# del $(MODEL).reg
$(PC32_DOWNLOAD) -f$(PROGRAM) -s$(SERVER_NAME) -p$(SERVER_PORT)
echo $(DOWNLOAD_SUCCESS) $(PROGRAM)

#----- Dependencies -----
adc64_grtm.o$(DSP_FAMILY) : $(MODEL).c
ifneq ($(QUICK),1)
$(OBJS) : $(MAKEFILE)
endif
#il_pc32.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\cli\liti_fpc\li_comms.h \
# $(MATLAB_ROOT)\rtw\c\src\ext_srv_c3x.h

#ext_srv_c3x.o$(DSP_FAMILY) : $(MATLAB_ROOT)\rtw\cli\liti_fpc\li_comms.h \
# $(MATLAB_ROOT)\rtw\c\src\ext_srv_c3x.h
```

E.3. DEVICE DRIVER FILES

E.3.1 ADC BLOCKS

```
%% Abstract:
%% TLC file for the ADC64 A/D Block.
%% This file is used to generate code to read
%% values from the A/D converters and scale them to +-10.
%% Author:
%% Magash Pillay
%% Date:
%% 2000/08/11
%%
%implements "adc64_ad" "C"

#include "ilib_adc64.tlc"

%function Start(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
%assign Trig_s= LibBlockParameterValue(P2,0)
/* Connect to Trigger source*/
%switch (Trig_s)
    %case 2
        trigger(PIT0_TIMER,(int){%<LibBlockParameter(P1,"",0)>});
        %break
    %case 3
        trigger(PIT1_TIMER,(int){%<LibBlockParameter(P1,"",0)>});
        %break
    %case 4
        trigger(PIT2_TIMER,(int){%<LibBlockParameter(P1,"",0)>});
        %break
    %case 5
        trigger(PIT3_TIMER,(int){%<LibBlockParameter(P1,"",0)>});
        %break
    %case 6
        trigger(PIT4_TIMER,(int){%<LibBlockParameter(P1,"",0)>});
        %break
%endswitch

%endfunction %% BlockInstanceSetup
```

```

function Outputs(block, system) Output
    /* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
    /* read in the corrected values from A/D and scale to +/-10 */
    %assign Trig_s= LibBlockParameterValue(P2,0)
    %assign ADC_num0 = LibBlockParameterValue(P1,0)*2
    %assign ADC_num1 = LibBlockParameterValue(P1,0)*2+1
    %assign ADC_num0 =CAST("Number",ADC_num0)
    %assign ADC_num1 =CAST("Number",ADC_num1)

    {
        %<LibBlockOutputSignal(0,"",0)>=read_adc(BASEBOARD,(%<ADC_num0>))/(3276.7);
        %<LibBlockOutputSignal(0,"",1)>=read_adc(BASEBOARD,(%<ADC_num1>))/(3276.7);
        %if (Trig_s==1)
            convert_adc_pair(BASEBOARD,(int)(%<LibBlockParameterValue(P1,0)>));
        %endif
    }
endfunction %% Outputs

```

E.3.2 DAC BLOCK

```

%%
%%
%% Abstract:
%%   TLC file for the PC32 D/A Block.
%%   This file is used to generate code to write
%%   values to the D/A converters. At termination
%%   all outputs are set to 0.
%% Author:
%%   Adam Stylo
%% Date:
%%   98/11/03
%%
%implements "adc64_da" "C"

#include "iilib_adc64.tlc"

function BlockInstanceSetup(block, system) void

    %% Only allow 1 instance of the D/A block
    %if IEXISTS("Rt_pc32da")
        %assign ::Rt_pc32da = 1
    %else
        %error Only 1 PC32dan block is allowed in the model.
    %endif
endfunction %% BlockInstanceSetup

%% Function: Outputs
=====
%%
%% Abstract:
%%   Generate inlined code to perform one D/A conversion.
%%
function Outputs(block, system) Output

    /* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
    /* Start an output conversion*/
    {
        write_dac(BASEBOARD, 0, %<LibBlockInputSignal(0,"",0)>*(3276.7));
        convert_dac(BASEBOARD, 0);
        write_dac(BASEBOARD, 1, %<LibBlockInputSignal(0,"",1)>*(3276.7));
        convert_dac(BASEBOARD, 1);
    }
endfunction %% Outputs

%openfile buffer
/* reset D/A outputs to 0 at termination. */
write_dac(BASEBOARD, 0, 0);

```

```

convert_dac(BASEBOARD, 0);
write_dac(BASEBOARD, 1, 0);
convert_dac(BASEBOARD, 1);
%closefile buffer
%<LibMdlTerminateCustomCode(buffer, "trailer")>

```

```

%% EOF: ADC64_da.tlc

```

E.3.3 PWM BLOCK

```

%%
%%
%% Abstract:
%%   TLC file for the PWM Block. Generates code used to
%%   control a PWM/Tacho add on card.
%% Author: Magash Pillay
%%
%implements "pwmblock_adc" "C"

#include "ilib_adc64.tlc"

%assign ::Vortl = LibBlockParameter(P1,"",0)
%assign ::CtrlMode = LibBlockParameter(P2,"",0)

%function BlockInstanceSetup(block, system) void
%% Only allow 1 pwm block
%if EXISTS("IIPWMBlockSeen")
  %assign errTxt = "Only 1 Interrupt block is allowed in " ...
    "model: %<CompiledModel.Name>."
  %exit RTW Fatal: %<errTxt>
%else
  %assign ::IIPWMBlockSeen = 1
%endif

%openfile buffer
%%DECODE0
#define Status_word (volatile int*) 0x818801
#define Data_word (volatile int*) 0x818800

%%DECODE1
%%#define Status_word (volatile int*) 0x819001
%%#define Data_word (volatile int*) 0x819000

#define TAUS (0)
#define TTOT (0)
#define TMIN (0)
int VORTL,TSTART;

void pollpwm( void )
{
    while (*(Status_word) & 0x1);
}

%closefile buffer
%<LibCacheDefine(buffer)>

%openfile buffer
#ifdef IO_ENABLE
printf("Initializing PWM Block ....\n");
#endif

VORTL = (int)(%<Vortl>);
TSTART = ((int)(512-(322/(VORTL+1))));

*IOBCR = 0x58;

```



```

*(Status_word) = 128; /* set up 16 bit addressing mode */
*(Status_word) = 128; /* set address to zero */

pollpwm();
*(Data_word) = 0; /* Ua */
pollpwm();
*(Data_word) = 0; /* Ub */
pollpwm();
*(Data_word) = 0; /* phi1 */
pollpwm();
*(Data_word) = 0; /* dphi1 */
pollpwm();
*(Data_word) = 0; /* phi0 */
pollpwm();
*(Data_word) = 0; /* dphi0 */
pollpwm();
*(Data_word) = 0; /* phiadd */
pollpwm();
*(Data_word) = 0; /* unused */
pollpwm();
*(Data_word) = TAUS; /* turn off time */
pollpwm();
*(Data_word) = TTOT; /* dead band */
pollpwm();
*(Data_word) = TMIN; /* turn on time */
pollpwm();
*(Data_word) = VORTL; /* switching frequency scale value */
pollpwm();
*(Data_word) = TSTART; /* start of processing cycle */

*(Status_word) = 129;

%closefile buffer
%<LibMdlStartCustomCode(buffer, "trailer")>

%openfile buffer
/* dissable the PWM board at terminate */
*(Status_word) = 0;
*(Status_word) = 0;
%closefile buffer
%<LibMdlTerminateCustomCode(buffer, "trailer")>
%endfunction

%function Outputs(block, system) Output
/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */
{
    *(Status_word) = 129;
    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 0)>;
    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 1)>;

    if ((int)%<CtrlMode> == 1) /* skip three values to write frequency */
    {
        pollpwm();
        *(Status_word) = 897;
    }

    pollpwm();
    *(Data_word) = (int)%<LibBlockInputSignal(0, "", "", 2)>;
}
%endfunction %% Outputs

```

E.3.4 ASYNCHRONOUS INTERRUPT SUPPORT

mod on 17/04/2000

%%

```

%%%
%%% Abstract:
%%% TLC file for the ADC64 Asynchronous Interrupt Block.
%%% This file is used to generate code to support asynchronous
%%% interrupts on the ADC64.
%%% Author:
%%% Magash Pillay
%%% Date:
%%% 2000/08/11
%%%

%Implements "liinterrupt_adc" "C"

#include "ilib_adc64.tlc"

%%% Function: BlockInstanceSetup =====
%%% Abstract:
%%% Find all the function-call subsystems that are attached to the
%%% interrupt block and hook-in the necessary code for each routine.
%%% This function
%%%
%%% o Connect each ISR in the model's start function.
%%%
%%% o Enable each ISR at the bottom of the model's start function.
%%%
%%% o Disable each ISR in the model's terminate function.
%%%
%%% o Save floating point context in the ISR's critical code section

%assign ::Tmr0freq = LibBlockParameter(P1,"",0)
%assign ::Tmr1freq = LibBlockParameter(P2,"",0)

%function BlockInstanceSetup(block, system) void
%%% Only allow 1 interrupt block
%if EXISTS("liInterruptBlockSeen")
%assign errTxt = "Only 1 Interrupt block is allowed in " ...
"model: %<CompiledModel.Name>."
%exit RTW Fatal: %<errTxt>
%else
%assign ::liInterruptBlockSeen = 1
%endif

%openfile buffer
#ifdef IO_ENABLE
printf("Connecting Interrupts\n");
#endif

%openfile buffer
/* Define a interrupt_block */
#define INTERRUPT_BLOCK
#include "adcmain_f.h"

%closefile buffer
%<LibHeaderFileCustomCode(buffer,"trailer")>

%foreach callIdx = NumSFcnSysOutputCalls

%%% Get downstream block if there is one
%if "%<SFcnSystemOutputCall[callIdx].BlockToCall>" != "unconnected"
%assign ssSysIdx = SFcnSystemOutputCall[callIdx].BlockToCall[0]
%assign ssBlkIdx = SFcnSystemOutputCall[callIdx].BlockToCall[1]
%assign ssBlock = CompiledModel.System[ssSysIdx].Block[ssBlkIdx]
%%% Check to see if this is a direct connection
%if (ssBlock.ControlInputPort.Width != 1)
%assign errTxt = "The li Interrupt block '%<block.Name>' " ...
"outputs must be directly connected to one function-call subsystem. " ...
"The destination function-call subsystem block '%<ssBlock.Name>' " ...

```

```

    "has other inputs."
%exit RTW Fatal: %<errTxt>
%endif
%% Assume it is a subsystem block(Simulink checked for a f-c subsys already).
%assign isrSystem = System[ssBlock.ParamSettings.SystemIdx]

%<LibForceOutputUpdateFcn(isrSystem)>
%% NO need to redefine since function calls are from dummy interrupt functions
%openfile buffer
void c_int0%<callIdx+1>(void);
%closefile buffer
%<LibCacheDefine(buffer)>

%openfile buffer
/*notes on P4
range 1 to 3 ,coresspondes to interrupt number.
*/
%If callIdx == 0 %% this is an end of conversion, analog interrupt
    %%enable one interrupt mask
    %openfile buffer1
    // enable all ADC's interrupt
    write_analog_interrupt_mask(BASEBOARD, 0x1);
    %closefile buffer1
    %<LibMdlStartCustomCode(buffer1, "trailer")>

%endif
%if callIdx == ( %<LibBlockParameterValue(P4,0)>-1)
%% checking which int is base rate generator
    %%call system function without using TI interrupt convention
    %%ie wrap it in Base_Rate Function
    %if callIdx==1
    %openfile temp
    #define TMRO_BASE_RATE
    %closefile temp
    %<LibCacheDefine(temp)>
    %endif
    void Base_Rate_Function()
    {
        %<isrSystem.OutputUpdateFcn>(rtS,1,0);
    }
%else
/*
ISR for : %<ssBlock.Name>
*/
%if callIdx < 9
void c_int0%<callIdx+1>()
{
/* call subsystem block
Using TID =0 since, single tasking simulation
*/
%<isrSystem.OutputUpdateFcn>(rtS,1,0);
}

%else
void c_int%<callIdx+1>()
{
/* call subsystem block
Using TID =0 since, single tasking simulation
*/
%<isrSystem.OutputUpdateFcn>(rtS,1,0);
}

%endif
%endif
%closefile buffer
%<LibSourceFileCustomCode(buffer,"trailer")>

%openfile buffer

```

```

%% controlPortIdx will never get used when only one f-c control input
/* use base tid inside an ISR for any blocks accessing task time*/
#define %<ITID> 0

%closefile buffer
%<LibSystemOutputCustomCode(isrSystem, buffer, "declaration")>
%openfile buffer
#undef %<ITID>
%closefile buffer
%<LibSystemOutputCustomCode(isrSystem, buffer, "trailer")>

%% Connect the ISR in the model's start function

%openfile buffer
/* connect ISR system: %<ssBlock.Name> */
//int registration
%if callIdx == (%<LibBlockParameterValue(P4,0)>-1)
{
    /* assign rt_onestep from real time kernel to interrupt */
    install_int_vector(rtOnestep,(int)%<LibBlockParameterValue(P3,callIdx)>);
    enable_interrupt((int)%<LibBlockParameterValue(P3,callIdx)>-1);
    #ifdef IO_ENABLE
        printf("Vectior installed for INT #%%d, Base rate interrupt.\n",
            (int)%<LibBlockParameter(P3,"", "", callIdx)>);
    #endif
}
%else
{
    %if callIdx < 9
        install_int_vector(c_int0%<callIdx+1>,(int)%<LibBlockParameterValue(P3,callIdx)>);
    %else
        install_int_vector(c_int%<callIdx+1>,(int)%<LibBlockParameterValue(P3,callIdx)>);
    %endif
    enable_interrupt((int)%<LibBlockParameterValue(P3,callIdx)>-1);
    #ifdef IO_ENABLE
        printf("Vectior installed for INT #%%d.\n", (int)%<LibBlockParameter(P3,"", "", callIdx)>);
    #endif
}
%endif

%closefile buffer
%<LibMdlStartCustomCode(buffer, "trailer")>

    %openfile buffer
/* disconnect ISR system: %<ssBlock.Name> */
if (%<LibBlockParameter(P3,"", "", callIdx)>==9)
{
    /* only disconnect timer0 if it was set up here */
    disable_interrupt((int)%<LibBlockParameter(P3,"", "", callIdx)>-1);
    deinstall_int_vector((int)%<LibBlockParameter(P3,"", "", callIdx)>);
    #ifdef IO_ENABLE
        printf("INT #%%d disabled.\n", (int)%<LibBlockParameter(P3,"", "", callIdx)>);
    #endif
}
else
{
    disable_interrupt((int)%<LibBlockParameter(P3,"", "", callIdx)>-1);
    deinstall_int_vector((int)%<LibBlockParameter(P3,"", "", callIdx)>);
    #ifdef IO_ENABLE
        printf("INT #%%d disabled.\n", (int)%<LibBlockParameter(P3,"", "", callIdx)>);
    #endif
}
%closefile buffer
%<LibMdlTerminateCustomCode(buffer, "trailer")>
%else %% The element is not connected to anything

```

```

%assign wrnTxt = "No code will be generated for ISR %<callIdx> \
"since it is not connected to anything."
%warning %<wrnTxt>
%endif
%endforeach

%% Setup timers
%openfile buffer
Interrupt_Block=1;
#ifdef TMR0_BASE_RATE
/*simulation step size take precedence over TMR0 freq when set as base rate */
timer(6, (int)(1.0 / ssGetStepSize(rS)));
#else
/* Only change Timer 0 settings if it isn't used for base sampling rate */
timer(6, (int)%<Tmr0freq>);
#endif
timer(7, (int)%<Tmr1freq>);

#ifdef IO_ENABLE
printf("Interrupts Connected ,Waiting for start Signal...\n");
#endif
%closefile buffer
%<LibMdlStartCustomCode(buffer, "header")>
%endfunction

%% (EOF) iinterrupt.tlc

```

E.3.5 EXTERNAL TIMERS

```

%%
%%
%% Abstract:
%% TLC file for the external timers. Generates code used to
%% control external timers.
%% Author:
%% Magash Pillay
%% Date:
%% 2000/08/14
%%

%implements "ext_timers_adc" "C"

#include "iilib_adc64.tlc"

%assign ::tmr0 = LibBlockParameter(P1,"",0)
%assign ::tmr1 = LibBlockParameter(P2,"",0)
%assign ::tmr2 = LibBlockParameter(P3,"",0)
%assign ::tmr3 = LibBlockParameter(P4,"",0)
%assign ::tmr4 = LibBlockParameter(P5,"",0)

%function BlockInstanceSetup(block, system) void
%% Only allow 1 pwm block
%if EXISTS("ITIMERS")
%assign errTxt = "Only 1 Timer block is allowed in " ...
"model: %<CompiledModel.Name>,"
%exit RTW Fatal: %<errTxt>
%else
%assign ::ITIMERS = 1
%endif

%openfile buffer
#ifdef IO_ENABLE
printf("Initializing PWM Block ....\n");
#endif

```

```
/*setup external timers */
timer(0,(int)%<tmr0>);
timer(1,(int)%<tmr1>);
timer(2,(int)%<tmr2>);
timer(3,(int)%<tmr3>);
timer(4,(int)%<tmr4>);

%closefile buffer
%<LibMdlStartCustomCode(buffer, "traller")>

%endfunction
```

APPENDIX F: DESCRIPTION OF CD

Category	Sub Category	Comments	File Location
			cd-rom drive =e:
Documents			
	TI	TMS320C3x User's Guide	e:\documents\texas
	II	Software and Hardware Manuals	e:\documents\ii
	WinSock	Windows Sockets API documents	e:\documents\winsock
	Mathworks	Mathworks manuals	e:\matlabr11\help\pdf_doc
RADE PC32			
	Server	Server executable	E:\matlabr11\rtw\cli\bin\server_PC3
	software	System files	E:\matlabr11\rtw\cli\ti_fpc
	device drives	Block TLC files	E:\matlabr11\rtw\cli\devices
RADE ADC64			
	Server	Server executable	E:\matlabr11\rtw\cli\bin\server_ADC64
	software	System files	E:\matlabr11\rtw\cli\adc64\src
	device drives	Block TLC files	E:\matlabr11\rtw\cli\adc64\devices
Server Projects			
	PC32	Visual C project workspace	E:\visual_c\server_pc32
	ADC64	Visual C project workspace	E:\visual_c\server_adc64
Mathworks Patch		Newer binaries supplied by Mathworks to fix external mode bugs	E:\Mathworks patch
Innovative Integration			
	PC32 library	Library files used for the target system	E:\pc32cc
	ADC64 library	Library files used for the target system	E:\adc64cc
TI compilers			
	Version 4.7	Older compiler used with CSDE	E:\fltc
	Version 5.11	Latest compiler used for RADE system	E:\c3xtools

REFERENCES

- AHMED1 Ifran Ahmed, "Implementation of PID and deadbeat controllers with the TMS320 family", Digital Signal Processing Applications with the TMS320 family (Theory, Algorithms, and Implementations), VOL 2, Texas Instruments, 1990
- AMCC1 "Bus Mastering with the S5933 PCI Matchmaker", Applied Micro Circuits Corporation, February 1996
- BLERK1 Bruce van Blerk, "Development of a Scaled Down Paper Machine to Demonstrate the Principles of Tension Control" *MSc Thesis*, Dept. of Electrical Engineering, University of Natal, Durban 1998
- BOOCH1 Grady Booch "Object-Oriented Analysis and Design: with applications", Second Edition, Addison-Wesley Publishing Company, 1994
- BURRBROWN1 "ADS7805 16-bit 10us sampling CMOS analogue-to-digital converter", Burr-Brown data sheet
- BURRBROWN2 "DAC712 16-bit digital-to-analog converter with 16-bit bus interface", Burr-Brown data sheet
- CHUNG1 Kai M. Chung, Astro Wu, Tresna Hidajat, "Using the TMS320C24X DSP controller for optimal digital control", Application report:SPRA295, Texas Instruments, January 1998
- DANIEL1 Daniel J. Wisheart, "Debugging Embedded Systems", C/C++ User Journal, June 1999
- DSPACE1 Herbert Schutte, "TDE: An integrated toolset for real-time control applications", dSPACE GmbH, Proceedings of the MOVIC 1998, August 1998

- DSPACE2 Susanne Kohl, Peter Bechberger, "In-flight simulators and Stationary flight simulator with dSPACE development tools", dSPACE GmbH, Proceedings of the EADC, France, June 1999
- DSPACE3 Rainer Otterbach, Thomas Pohlmann, Andreas Rukgauer, Jorg Vater. "DS1103 PPC controller board: Rapid prototyping with combined RSIC and DSP power for motion control", dSPACE GmbH, Proceedings of PCIM 1998, Germany, May 1998
- DSPACE4 Jorg Vater, "The need for and the principle of high-resolution incremental encoder interfaces in rapid control prototyping" dSPACE GmbH, dSPACE GmbH, Proceedings of PCIM 1997, Germany, June 1997
- DSPACE5 Rainer Otterbach, Robert Leinfellner, "Real-Time Simulation: Requirments and the State of Technology", Translation from "Vituelles Ausprobieren" Elektronik, August 1999
- FENG1 Henry Feng, Martin Torngren, Bengt Eriksson, "Experiences Using dSPACE Rapid Prototyping Tools For Real-Time Control Applications", Proceedings of the DSP Scandinavia Technical Conference, Sweden, June 1997
- GAN1 Woon-Seng Gan, Yong-Kim Chong, Wilson Gong, Wei-Tong Tan, "Rapid prototyping system for teaching real-time digital signal processing", IEEE transactions on education, VOL 43, NO 1, February 2000
- GANSSLE1 Jack G. Ganssle, "Debuggers for Modern Embedded Systems", Embedded Systems Programming, pg 58-65, November 1998
- GNU1 Richard M. Stallman, Roland McGrath " GNU Make. A Program for Directing Recompilation", Free Software Foundation, May 1998
- GORDON1 V. Scott Gordon, James M. Bieman, "Rapid Prototyping: Lessons Learned", IEEE Software VOL 12, NO 1, January 1995
- GREGA1 Wojciech Grega, Krzysztof Kolek, Andrzej Turnau, " Rapid prototyping

environment for real-time control education", IEEE 1999, 0-7695-0134-6/99

HAMANN1	Hamann Jerry C, Muknahallipatna Suresh, "Distributed instrumentation and computation: a look at what's out on the end of the Internet", ASEE Annual Conference Proceedings, Washington, DC, USA, 1998
HANNING1	"Pulse width modulator, PBM 1/87", Hanning Elektro-Werke GmbH, data sheet, Rev 2.0
HANNING2	"Incremental Rotary Encoder Interface, TC3005H", Hanning Elektro-Werke GmbH, data sheet
HUGH1	Hugh Jack, Michael Karlesky, "A virtual manufacturing Laboratory", ", ASEE Annual Conference Proceedings, Washington, DC, USA, 1998
INNOVATIVE1	"PC32 Hardware Manual", Innovative Integration, 1999
INNOVATIVE2	"PC32 Developer's Software Manual", Innovative Integration, 1999
INNOVATIVE3	"ADC64 Hardware Manual", Innovative Integration, 1999
INNOVATIVE4	"ADC64 Developer's Software Manual", Innovative Integration 1999
JAWITZ1	Jeff Jawitz, " It takes more than just lecturing: Developing engineering education in South Africa", 9 th Annual South African Universities Power Engineering Conference (SAUPEC), University of Natal, January 2000
KOZICK1	Richard J. Kozick, Curtis C. Crane, "An integrated environment for the modelling, simulation, digital signal processing and control", IEEE transactions on education, VOL. 39, NO 2, May 1996
MATHWORKS1	"Using Matlab, Version 5.3", The Mathworks Inc, January 1999
MATHWORKS2	"Using Simulink, Version 3", The Mathworks Inc, January 1999
MATHWORKS3	"Simulink, Writing S-Function, Version 3", The Mathworks Inc, October 1998
MATHWORKS4	"Real-Time Workshop, User's Guide, Version 3", The Mathworks Inc, January 1999

MATHWORKS5	"Target Language Compiler Reference Guide, Version 1.2", The Mathworks Inc, January 1999
MATHWORKS6	"Real-Time Workshop, User's Guide, Version 4", The Mathworks Inc, September 2000.
MATHWORKS7	"Control System Toolbox User's Guide, Version 4.2", The Mathworks Inc, January 1999
MATHWORKS8	"Non-Linear Design Control Blockset, User's Guide. Version 5", The Mathworks Inc, April 1997
MOODLEY1	Lynden Moodely, "Position Controller for a DC Drive", B.Sc. Eng Thesis, Dept. of Electrical Engineering, University of Natal, Durban 1999
MSDN1	"CHATTER and CHATSRVR sample programs", MSDN Microsoft 1999
MSDN2	"Windows Sockets for Network Programming: Overview", MSDN Microsoft 1999
MSDN3	"Windows Sockets: Ports and Socket Addresses", MSDN Microsoft 1999
MSDN4	"IPC and Windows 95", MSDN Microsoft 1999
OGATA1	Katsuhiko Ogata, "Modern Control Engineering, Second Edition", Prentice-Hall International, Inc, 1990
OGATA2	Katsuhiko Ogata, "Discrete-Time Control Sytems", Prentice-Hall International, Inc, 1987
PAI1	Pai Devdas, Kelkar Ajit, Layton Richard, Schulz Mark, "Vertical integration of the undergraduate learning experience" ", ASEE Annual Conference Proceedings, Washington, DC, USA, 1998
POUS1	C. Pous, A. Oller, J. Vehi, J.L. de la Rosa, " Using Matlab Real-Time Workshop in teaching control design techniques", IEEE 1996, 0-8186-7649-3/96

- REID1 Reid, Richard J, "Virtual laboratory for the introductory engineering course", ASEE Annual Conference Proceedings, Washington, DC, USA, 1998
- SADASTVA1 Indu Sadassiva, Frank Flinders, Wardina Oghanna, " A graphical based automatic real-time code generator for power electronic control applications", Proceedings of the ISIE 1997, Guimaraes Portugal, 1997
- SQUIRES1 David Squires, Jenny Preece, "Predicting quality in educational software: Evaluating for learning, usability and synergy between them", Interacting with Computers 11, Elsevier, 1999
- STURGEON1 Shaun Sturgeon, "DSP based Field Oriented Control of an Induction Machine", B.Sc. Eng, Dept. of Electrical Engineering, University of Natal, Durban 1998
- SLIVINSKI1 Charles Slivinski, Jack Borninski, "Control Systems Compensation and Implementations with the TMS32010", Digital Signal Processing Application with the TMS320 Family, VOL 1, Texas Instruments 1989
- STYLO1 Adam W. Stylo, " A low cost, high performance PC based integrated real-time motion control development system", M.SC. Thesis, Dept. of Electrical Engineering, University of Natal, Durban 2001
- TEXAS1 "TMS320C3X User's Guide", SPRU031E, Texas Instruments, July 1997
- TEXAS2 "TMS320C33 Product Information", Texas Instruments, January 2000
- TQ1 Product information from TQ Education and Training LTD web site www.tq.com
- TQ2 Peter Wellstead, "Control Systems Engineering: A core skill for Engineers in a Changing World", TQ Education and Training LTD

References

- WALKER1 Myles Walker, " Test Bed System to Investigate the Energy Efficiency of Variable Speed Drives Systems Under Variable Load Conditions", M.Sc. Thesis in preparation, Dept. of Electrical Engineering, University of Natal, Durban 2000
- WASHBURN1 K. Washburn, J.T. Evans "TCP/IP Running a Successful Network", Addison-Wesley, 1993
- WINCON1 "WinCon 3.0 Description ", Quanser Consulting Inc, www.wincon.quanser.com/description.html
- WINSOCK1 "Windows Sockets: An Open Interface for Network Programming under Microsoft Windows", Version 1.1, January 1993 (www.stardust.com)
- WINSOCK2 "Windows Sockets 2: Application Programming Interface", Revision 2.10, January 1996.
- WOEHR1 Woehr Jack J, "A Conversation with Glenn Reeves: Really remote debugging for real-time systems", Dr Dobb's Journal, November 1999
- WORTHMANN1 Cedric Worthmann, "Feasibility Study of a Neural Network Current Controller for a Boost Rectifier", M.Sc. Thesis in preparation, Dept. of Electrical Engineering, University of Natal, Durban 2000