University of Natal

A Low Cost, High Performance PC Based Integrated Real-Time Motion Control Development System

by

Adam Wojciech Stylo

Submitted in 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.

December 2000

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 :____Date : ______________

A W Stylo

ABSTRACT

The control of electrical drives, or motion control, is important in modern industry. In order to satisfy the requirements of industry, it is important for tertiary institutions to produce graduates skilled in this field. The theoretical content of a typical electrical engineering course will prepare students to tackle design and offline simulation of a digital motion controller. However, to gain an in-depth understanding of the field, students need to be able to implement and test their designs in practice.

The complete design process of a digital motion controller is an inherently lengthy process requiring a number of diverse skills, for example microprocessor based hardware and software design. While hardware design issues can be minimised by a choice of a commercially available controller board, the coding of real-time software for a complex controller can pose a steep learning curve. At the undergraduate level, students seldom will possess sufficient practical expertise to fully implement a challenging motion control design in the limited time frames allocated for such projects.

This thesis presents a complete rapid prototyping environment for the design of motion control, the Control System Development Environment (CSDE). The CSDE allows a seamless progression of a motion control project through all stages, from initial design and simulation, through real-time implementation to final online tuning and validation. Users are freed from all low-level software and hardware design issues. In the context of undergraduate design projects, the CSDE allows students to design, simulate and prototype challenging solutions in the limited time available. Thus, students can gain in-depth, system level expertise in the design of motion control without being hampered by low-level design issues.

The CSDE has been successfully tested by a number of undergraduate students at the Department of Electrical Engineering at the University of Natal. In particular, the CSDE's effectiveness has been demonstrated by its application during two prize winning final year design projects.

ACKNOWLEDGEMENTS

The work presented in this thesis was supervised by Mr. G Diana of the Department of Electrical Engineering at the University of Natal. I wish to thank Mr. Diana for his support, encouragement and guidance throughout the project.

I also wish to express my gratitude to the following people :

Ms. Melinda Huisman for her love, understanding and continuous support;

My family and friends for their help and support;

My colleagues Mr. Myles Walker, Mr. Cedric Worthmann, Mr. Magash Pillay and Mr. Ravi Govender for their co-operation and providing a friendly working environment;

Final year students Mr. Shaun Sturgeon and Mr. Lynden Moodley for agreeing to field trial the CSDE during their design projects;

The University of Natal, the ESKOM Tertiary Education Support Programme and the Foundation for Research Development for financial support.

TABLE OF CONTENTS

CHAPTER ONE
INTRODUCTION

CHAPTER TWO
REVIEW OF AUTOMATED DEVELOPMENT TOOLS FOR DIGITAL CONTROL

CHAPTER THREE
OVERVIEW OF MATHWORKS TOOLS

CHAPTER FOUR
TARGET HARDWARE PLATFORM

CHAPTER FIVE
CONTROL SYSTEM DEVELOPMENT ENVIRONMENT

CHAPTER SIX
TARGET REAL-TIME SUPPORT COMPONENTS

CHAPTER SEVEN
HARDWARE DEVICE DRIVERS

CHAPTER EIGHT
HOST SUPPORT COMPONENTS AND UTILITIES

CHAPTER NINE
CSDE APPLICATION AND CASE STUDIES

CHAPTER TEN
CONCLUSIONS

Appendix A

Appendix B

Appendix C

Appendix D

Appendix E

Appendix F

Appendix G

Appendix H

Appendix I

Appendix J

APPENDIX K

LIST OF ACRONYMS

A/D	Analogue to Digital
ADC	Analogue to Digital Converter
API	Application Program Interface
ASIC	Application Specific Integrated Circuit
CACSD	Computer Aided Control System Design
CSDE	Control System Development Environment
D/A	Digital to Analogue
DAC	Digital to Analogue Converter
DLL	Dynamic Link Library
DPRAM	Dual Port Random Access Memory
DSP	Digital Signal Processing
FIFO	First In First Out
FOC	Field Oriented Control
GUI	Graphical User Interface
I/O	Input and Output
n	Innovative Integration
ISR	Interrupt Service Routine
MCDS	Motion Control Development System
PC	Personal Computer
PDL	Platform Dependant Layer
PIL	Platform Independent Layer
PWM	Pulse Width Modulation
RIDE	Rapid Integrated Development Environment
ROM	Read Only Memory
RTK	Real-Time Kernel
RTW	Real-Time Workshop
SRAM	Static Random Access Memory
TAG	Transputer Application Group
TI	Texas Instruments

TLC	Target Language Compiler
UND	University of Natal, Durban
VXD	Virtual External Device Driver

CHAPTER ONE INTRODUCTION

1.1 General

In today's world we are surrounded by motion - for example motorcars, trains and escalators. A large proportion of the force that provides this motion in our society is derived from electricity and there is a continuing drive to make the use of this relatively clean electrical power more widespread. Less obvious to the layman, but also crucial to our way of life, is the motion that drives modern industrial processes. In factories today, most of the driving force is derived from electricity. The majority of pumps, fans and conveyor belts, for example, are driven by electrical motors, thus electric drives form a crucial part of any modern day industrial process.

The precise control of speed and position in machinery is often of paramount importance [LINZENKIRCH1]. One example is the paper industry, where the speed of various winding stages needs to be synchronised and maintained to prevent damage to the product and thus subsequent costly interruptions in the manufacturing process [BLERK1]. Another example is precision robotic manipulators which often have to perform complex manoeuvres. Accurate position control needs to be employed as their position has to fall within tight tolerances. Many more areas can be named where control of electric motors is important, and collectively the field is termed *motion control.* Fig. 1.1 shows a schematic of a basic motion control setup. Although

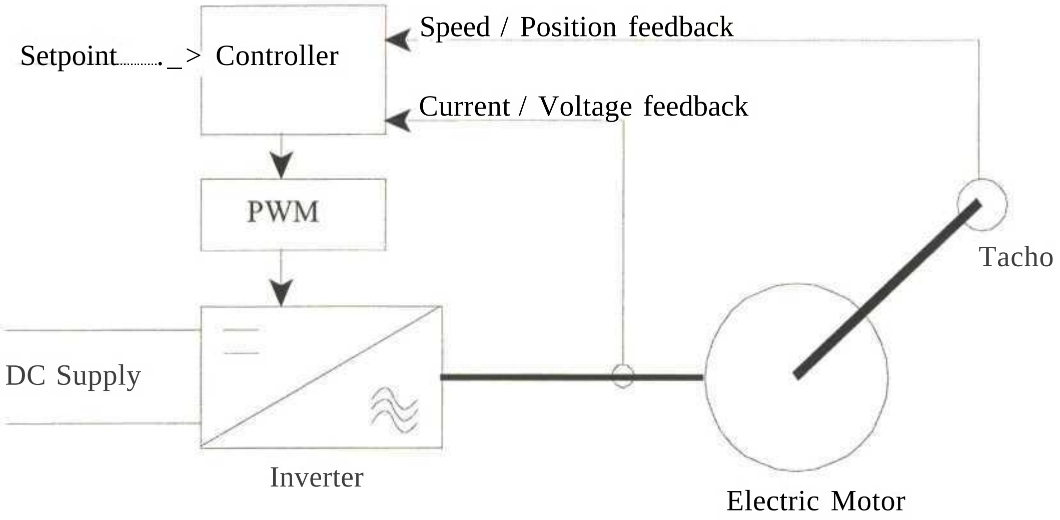

Setpoint..........._> Controller
Speed / Position feedback
Current / Voltage feedback
PWM
Tacho
DC Supply
Inverter
Electric Motor

Fig. 1.1: Block diagram of a typical motion control setup

details might vary with the particular field of application, typically they will be derived from the above.

Motion control forms a subset of the broad control field. A common set of mathematics and techniques is used to model all control problems. Be it a simple temperature controller or a complex chemical plant, we use similar tools to represent them for simulation and subsequently to control them. Although this thesis focuses on issues pertaining to the design and rapid prototyping of motion control, the ideas can easily be applied in other control disciplines.

1.2 Teaching Motion Control

The importance of motion control in the industry draws with it a need for engineering graduates with in-depth skills in the field. Currently, most undergraduate courses in electrical engineering include a sound theoretical grounding in the principles of control and particular attention often is paid to issues specific to the control of electrical machines. However, to apply the theory in practice students need to master a number of other skills including digital hardware and software design. The design of a complete digital motion controller is a lengthy process involving a number of steps :

(i) Modelling of the controlled plant
(ii) Controller design and simulation in non real-time
(iii) Design of the controller hardware
(iv) Coding of the control algorithm software
(v) Prototyping and tuning in real-time
(vi) Final validation and testing

Each of these individual steps forms an important component in a formal design process, but it is equally important to have an overall grasp of the design at the system level.

Author's own experience as well as the input from a number of students at the Department of Electrical Engineering at the University of Natal (UND) indicated that the theoretical content of the course prepared graduates well for points (i) and (ii) above. With the use of commercial microprocessor boards, the design of a controller platform can be greatly simplified. However, the writing of deterministic software to replicate the exact action of the simulated controller seems to be the greatest challenge.

The time allocated for final year projects is typically around 13 weeks. This short time combined with lack of prior experience in some practical design issues means that it is difficult for students to gain sound system level experience in the design and implementation of motion control. Students are seldom able to take challenging motion control projects through a formal design process and typically only focus on a small component of a design. The problem also exists at the postgraduate level, although it is far less pronounced due to the more generous time frames. The complete design of advanced control algorithms, for example field oriented control (FOC), was outside the scope of an undergraduate project and even at postgraduate level usually constituted a substantial portion of the dissertation [HEMME1, MEYER 1, RANDELHOFF1].

1.3 Project Goals and Description

After considering the above points, a need was identified to create an integrated environment which would allow students to conduct their motion control designs entirely at system level. The emphasis should be placed on the students' in-depth understanding of the control issues rather than the low-level details of implementation. With the ability to generate real-time prototypes directly from simulations, more challenging control strategies could be fully implemented and tested during a typical final year design course. In essence, students would be given an opportunity to gain valuable experience of designing, simulating, implementing and verifying advanced control algorithms in practice without first having to become expert software or hardware developers. The tool could prove equally valuable at postgraduate level. Here more emphasis could be placed on verification or comparison of a wide variety of complex control strategies without the time overhead usually associated with designing dedicated controllers based on the various algorithms.

To satisfy the above objectives, the proposed Control System Development Environment (CSDE) has to meet a number of seemingly conflicting requirements :

(i) Low cost - the overall cost of the CSDE will determine whether, and how many, systems can be made available to students.

(ii) High performance - the system has to provide adequate processing power to implement complex control algorithms,

(iii) Ease of use - students must be able to get up to speed with minimum time and effort.

Before deciding on the exact specifications for the CSDE, the author evaluated two commercially available solutions. The findings are presented in Chapter 2. Although the CSDE was designed particularly with motion control in mind it should prove useful in other control applications.

1.4 Thesis Structure

Chapter 2 discusses the issues involved in the development of a digital motion controller and identifies the need for an integrated development environment to simplify the process. Two commercially available solutions are briefly evaluated before a list of requirements is drawn up for the CSDE presented in this thesis.

Chapter 3 introduces the Math Works modelling and simulation software as the foundation of the author's work.

Chapter 4 describes the hardware platform targeted by the CSDE, paying attention to the DSP core, the I/O peripherals as well as custom expansion hardware.

Chapter 5 deals with the overall structure of the software components developed by the author in order for the CSDE to satisfy the requirements as laid out in Chapter 2. The process of automatic code generation is introduced in some detail.

Chapter 6 describes the custom kernel and other real-time support functions developed specifically for the hardware platform from Chapter 4.

Chapter 7 introduces the library of hardware driver blocks created to support the CSDE functionality under the Math Works software platform from Chapter 3.

Chapter 8 covers the components of the CSDE developed for the host PC's Windows environment.

Chapter 9 demonstrates the operation of the CSDE by means of an example and outlines two practical case studies to demonstrate how the CSDE was used by students during their final year design projects.

Chapter 10 concludes the thesis and suggests possible improvements and extensions to the CSDE.

A number of Appendices provide additional information not directly included in the body of the thesis.

1.5 Summary

The CSDE presented in this thesis consists of a number of software and hardware components which were integrated by the author to form a complete design environment. Some of these components were of commercial origin, whilst others were created in collaboration with other researchers. An overview of these components is included in order to provide a complete description of the CSDE. For work which was not entirely the author's own, suitable recognition and references are provided in the text.

Components of the CSDE sourced commercially include :

(i) The host workstation PC running the Windows 95 operating system.
(ii) MATLAB, Simulink and Real-Time Workshop packages from MathWorks.
(iii) PC32 DSP Controller board from Innovative Integration (II).

The choice of Windows 95 as the software platform for the CSDE was dictated by its wide availability on the PC's in the laboratories. PC's equipped with the necessary hardware and running other operating systems were not readily available, thus there was no opportunity to test the CSDE on other software platforms.

A custom expansion card for the PC32, which provides pulse-width modulation (PWM) and incremental tacho support, was designed by Mr. Walker. The technical staff at the Electrical Engineering Department designed and constructed the power electronics and sensor setup necessary for driving electrical machines.

The author's work involved integrating the above components by providing the following software :

(i) Real-time kernel for the PC32

(ii) Standalone visualisation utility

(iii) Library of hardware driver blocks for use in Simulink

(iv) Routines to maintain a bi-directional communication link between the host PC and the PC32

(v) Various template and batch files to automate the code generation and downloading

The work discussed in this thesis resulted in a number of publications which were presented at both local and international conferences [DIANA 1, STYLO 1, 2, 3, 4].

CHAPTER TWO
REVIEW OF AUTOMATED DEVELOPMENT TOOLS FOR DIGITAL MOTION CONTROL

2.1 Introduction

Today there are a host of powerful processing platforms available with which to implement sophisticated motion control algorithms [TRZYNADL01]. But, as the available hardware grows ever more powerful the design process becomes correspondingly more complex, demanding multi disciplinary expertise [FENGl]. The designer not only needs to have in-depth knowledge of the control problem at hand, but also needs to be able to implement the solution in practice. This might involve both low-level microprocessor based hardware design as well as the coding of the control algorithm to execute correctly in real-time [AHMED1]. These practical issues often mean that a complete implementation of a complex motion control solution would extend beyond the time allocated for such projects at undergraduate level.

This Chapter discusses a number of issues involved in the complete design cycle of a digital motion controller, starting with modelling and simulation through to final real-time prototyping and validation. The need for an integrated development environment to shorten the traditionally drawn out design process of an embedded solution [SENESE1] is presented. The discussion is applicable to motion control design at all levels - undergraduate, postgraduate and in industry. Regardless of the design engineers background, similar issues will have to be considered and a structured design method followed.

There are a number of commercially available Computer Aided Control System Design (CACSD) packages to help engineers during the design of digital controllers. The author evaluated two CACSD packages, paying particular attention to their application in motion control. Their strengths and weaknesses are discussed and used to set goals for the CSDE which was developed in order to fill in the gaps left by existing packages.

2.2 Digital Motion Controller Design Process

The design of a digital motion control solution requires a careful and structured approach. The control engineer should follow a number of distinct steps, first getting acquainted with the dynamics of the plant, before proposing a solution and finally testing it by implementing it in real-time [SLIVINSKI1]. A flow chart diagram of the typical steps and iterations in the design of a digital controller is shown in Fig. 2.1. The flow chart shows how the design process progresses from initial modelling of the controlled plant, through simulation and real-time prototyping to a final implementation.

2.2.1 Modelling and Simulation

Before any successful control algorithm can be suggested, it is imperative to be familiar with the transfer characteristics of the plant in question - in most cases this will be a DC machine or an AC induction machine. Physical data needs to be collected from experiments and processed to establish the relationship between input currents, the resultant voltages and the outputs (torque, speed, position) of the given machine. There are modelling software packages available for the PC which can aid the designer in this area, for example MATLAB [MATHW0RKS1, MATHW0RKS7].

Once an adequate model of the machine is constructed and verified, it can then be used by simulation software to test a cross section of control strategies. There are several commercial packages on the market which can help, for example :

(i) Simulink [MATHW0RKS2], an extension of MATLAB, is a capable package. It has a number of specialised toolboxes to allow design and simulation of controllers for a variety of applications in a single graphical user interface (GUI).

(ii) CASED (Computer Analysis and Simulation of Electrical Drives) [KLEINHANS1] is specifically designed with motion control applications in mind. However, it lacks the user friendliness and graphical front end expected of modern software tools.

Once the simulation of a particular controller setup meets required specifications the implementation in real-time can proceed.

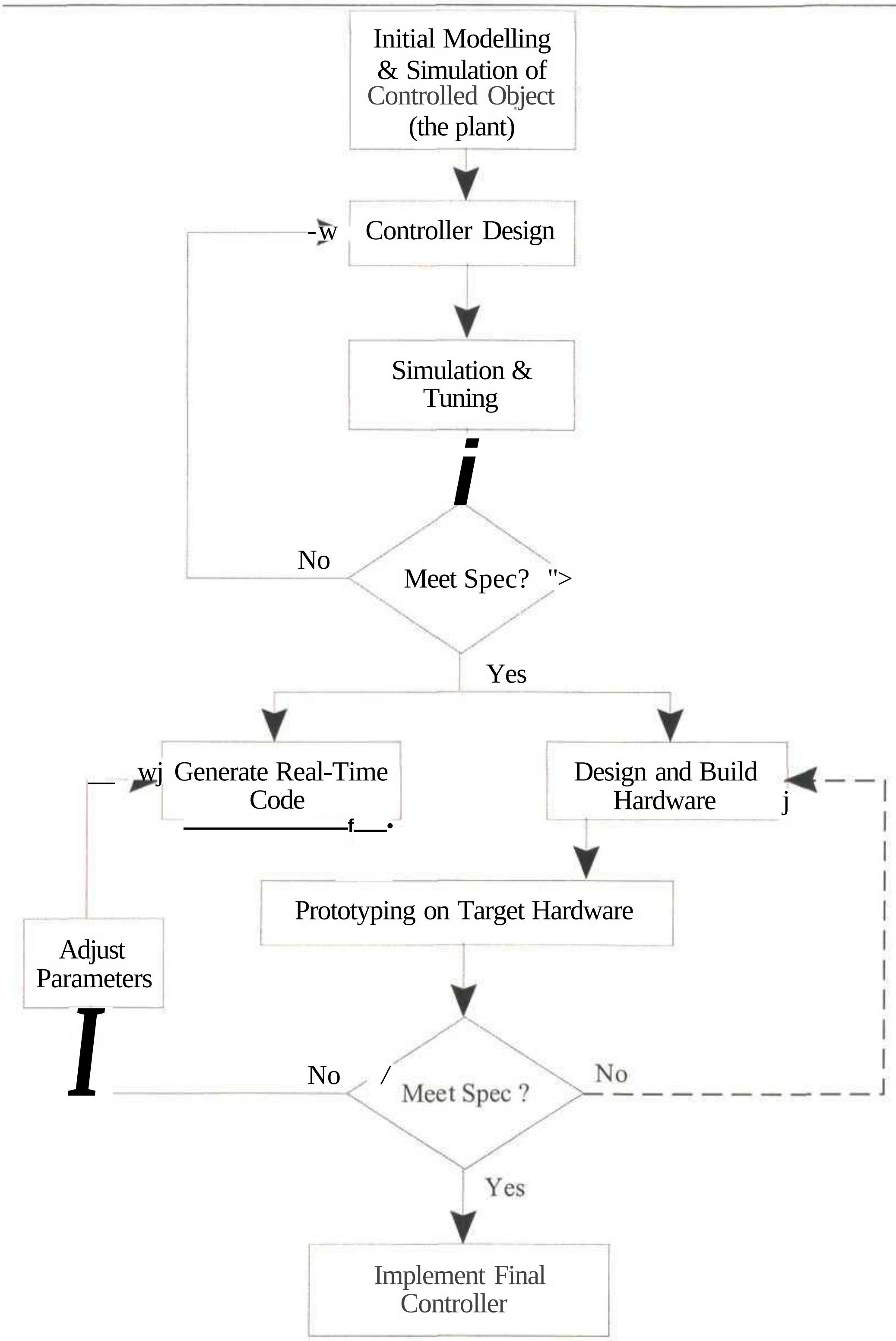

Initial Modelling & Simulation of Controlled Object (the plant)
Controller Design
Simulation & Tuning
No
Meet Spec?
Yes
Generate Real-Time Code
Design and Build Hardware
Prototyping on Target Hardware
Adjust Parameters
No
Meet Spec ?
No
Yes
Implement Final Controller

Fig. 2.1 : Steps and iterations in the design of a controller

2.2.2 Hardware Design

A vital issue in the development of a motion controller is the hardware platform used to execute the controller software developed from a simulation. A commercially available controller board could be used or the designer might choose to design custom hardware around one of many available processors [CRAVOTTA1]. Whatever the choice, the following issues need to be considered:

(i) processing power of the processor (or processors)
(ii) size of available memory
(iii) number of input and output channels
(iv) sampling resolution and conversion times of A/D and D/A converters
(v) other peripherals - eg. PWM generators or incremental tacho support

The simulation results will determine the required number and resolution of I/O channels but can only serve as a guideline to the selection of a processor and memory size. Some additional functions can either be implemented in hardware or later incorporated into the controller software, for example for the switching of power electronics or support for high resolution tachometers.

2.2.3 Software Design

Even with the availability of high-level programming languages and their advanced optimising compilers, designing software to implement complex motion controllers is non-trivial. Most undergraduate students will face a steep learning curve when confronted with the low-level issues of writing and debugging hard real-time code for a modern processor [GANSSLE1]. Even experienced control engineers or postgraduate students might find the software development takes a disproportionate amount of time allocated for the entire motion control project. With the diverse choice of algorithms available the designer might also want to experiment with a number of them before committing to a specific choice. Such experimenting may result in the need to completely, or at least partially, re-write the code. Even after a specific controller setup is chosen, tuning individual control parameters might involve a recompilation each time a small adjustment is made.

By nature human programmers are prone to making errors. As the size of the source code for a controller grows so do the chances of errors creeping in. Two major types of errors encountered in programming are syntax and logical errors. Syntax errors are easily picked up by the compiler and can usually be corrected with minimum effort. Logic errors, on the other hand, often prove to be difficult to locate and correct, as the resultant software might actually execute, but not perform the exact function the programmer had in mind. After successfully designing and simulating a controller it might be found that the hand coded real-time prototype fails to meet the design specifications. The reason for this could either be a logic error in the software or that the plant model used in simulation was incorrect thus producing a badly tuned controller. There could be substantial difficulty in differentiating between these two cases.

The final result of low-level development may be highly efficient code in terms of size and execution times, but a penalty could be paid in terms of the drawn out development time. In an industrial environment, upgrading and maintaining software could also be problematic [BASSETT1], especially with the high turnover of manpower. With the original designer of a particular system leaving, a simple upgrade could easily turn into a complete re-engineering project even if adequate documentation for the software is available. In the academic environment at the undergraduate level, lack of thorough documentation is often a problem. This means that substantial design effort can unnecessarily be duplicated by each subsequent generation of students.

2.2.4 Rapid Prototyping

Rapid prototyping means that the usual cycle of initial modelling, design, simulation and finally real-time prototyping, shown in Fig. 2.1, is made as short as possible. All the traditionally separate steps are combined into a transparent process which involves no manual coding and minimal adjustments to hardware. The engineer can seamlessly progress the design through all stages in a user-friendly, PC based environment, without having to write a single line of code. A rapid prototyping environment for motion control can present a number of advantages :

(i) In industry the length of time from initial idea to marketable product is of utmost importance [SENESE1]. Rapid prototyping has a potential to reduce the overall design time while improving the software quality [G0RD0N1].

(ii) In the tertiary education environment students are allocated limited time to complete their projects. Rapid prototyping can eliminate the tedious, low-level design issues. Thus, students could concentrate their effort on more in-depth investigation of the control theory and observe and verify their designs in practice [K0ZICK1].

An outline of a rapid prototyping system is shown in Fig. 2.2. The design environment is subdivided into layers, helping to isolate the user from low-level design issues. By employing such an environment, the designer could easily evaluate the real-time performance of a number of completely diverse control algorithms in a short time, since the transition from simulation to a functioning prototype would be taken care of automatically in a matter of minutes, or even seconds.

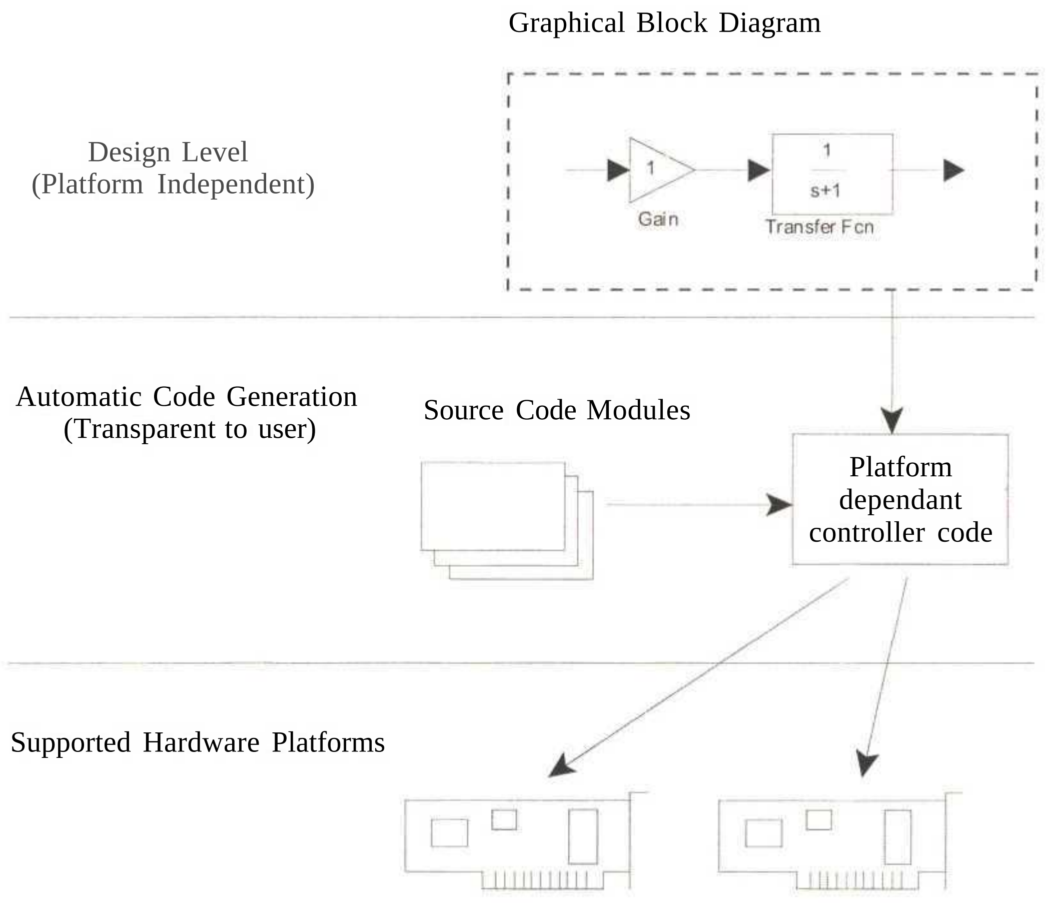


Fig. 2.2 : Levels of a rapid prototyping environment

The automatic code generation involves combining of pre-tested and optimised modules and thus, it has the additional advantage of guaranteeing the absence of both syntax and logic errors from the final controller code. Potential sources of confusion are eliminated when a malfunctioning prototype fails to meet simulation results, making the debugging easier.

Apart from the shortened development cycle, the major advantage is that controller design is kept entirely at a graphical, user-friendly level. This in turn means that the task of documenting a design becomes limited to the documentation of a clear schematic diagram. By employing rapid prototyping methods, software maintenance problems can be reduced [GORDON 1] as all code generation is handled transparently by the rapid prototyping environment.

2.2.5 An Integrated Development System

From the discussion above, a clear picture emerges of a complete environment dedicated to the development of motion control. A practical platform for implementing such an environment would be the widely available IBM compatible personal computer (PC), running a GUI operating system, for example Windows 95 or Windows NT. The targeted embedded hardware would be a selection of commercially available DSP platforms, preferably in the form of standard PC expansion cards. A range of hardware platforms would be decided on based on both price and performance. The proposed environment, the CSDE, would have to provide the following functionality under a unified user-friendly GUI:

(i) modelling;
(ii) controller design, simulation and tuning;
(iii) automatic generation of hard real-time code for a particular hardware platform;
(iv) online tuning of parameters in real-time;
(v) data capture in real-time for visualisation and validation.

A possible setup for such a development environment and the interaction between various components is shown in Fig. 2.3.

2.3 What is Currently Available ?

The ideas presented in the preceding sections of this Chapter are certainly not new or unique. **Before** designing the CSDE, **the** author reviewed some commercial systems currently available. Also, in the past a substantial research effort has gone into creating a complete Motion Control Development System (MCDS) at the University of Natal [MEYER1]. In the following sub-sections a number of systems are introduced and their relative strengths and weaknesses are outlined.

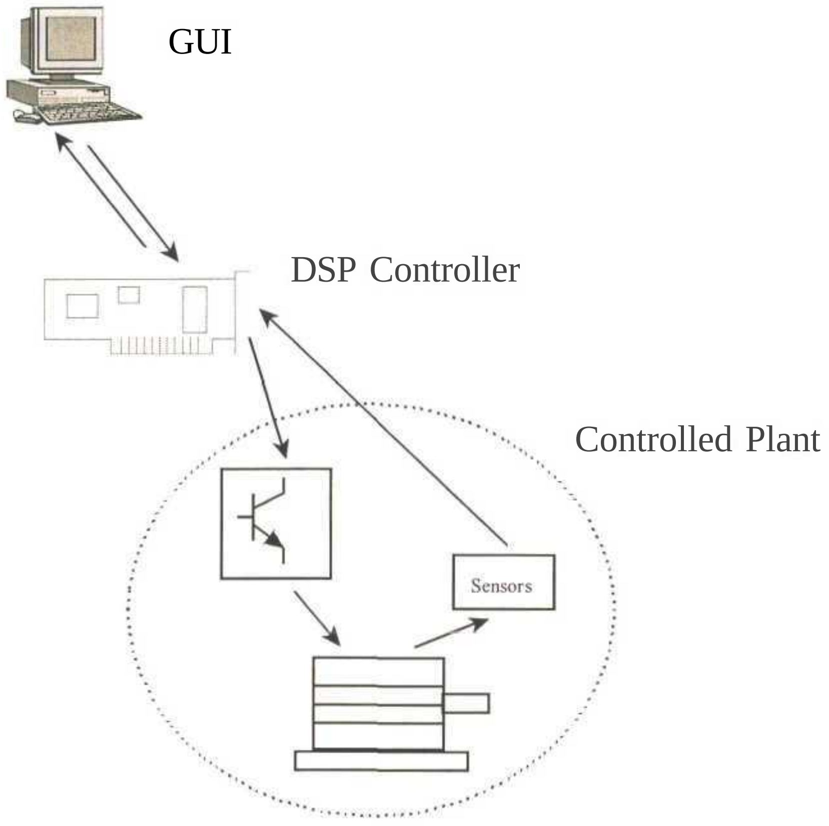


Fig. 2.3 : Interaction between parts of the proposed rapid prototyping environment.

2.3.1 Transputer Based System

The Electrical Engineering Department at the University of Natal has conducted extensive research into the control of high dynamic performance AC motors using parallel processing techniques [WEBSTER1]. The promising results of this early research effort led to the establishment of the Transputer Applications Group (TAG) to design and build transputer based controllers and to provide high-level tools for the design of motion control.

The hardware and software tools developed by TAG [MEYER 1, WOODWARD 1] were used to implement advanced control algorithms in practical applications [RANDELH0FF1]. The system was strong in terms of predictable code generation from graphical block diagrams and the hardware developed formed a capable and highly scalable platform.

The MCDS relied on external software for modelling and simulation and its associated Transputer based platform was specifically targeted at the motion control sector. All hardware and software was a custom in-house development, which meant the project required an ongoing development effort to keep up with trends in industry and the available technology. With a number of the researchers leaving, TAG was dissolved and the MCDS became obsolete in terms of other emerging tools.

2.3.2 MATLAB and Simulink

The MATLAB/Simulink package, from the MathWorks Incorporated, is a universal modelling and simulation environment requiring only a PC to run. Its popularity in the academic as well as industrial environment is mainly due to its modular structure. The core MATLAB engine provides robust support for most mathematical problems. An entire array of associated toolboxes are available to apply this mathematics engine to a number of specific fields. Simulink provides a GUI extension which moves the process of modelling and simulation to a graphical schematic level. The complete MathWorks package forms a comprehensive set of tools for most modelling and simulation requirements. Chapter 3 offers an in-depth discussion on the capabilities of the MATLAB system.

While modelling and simulation are very well covered by the MATLAB software, it provides limited support for generating the embedded hard real-time code necessary to implement motion control successfully. The Real-Time Workshop (RTW) extension, allows raw code to be generated from Simulink graphical diagrams [MATHW0RKS5]. Due to the scripting language used, the Target Language Compiler (TLC) can customise the RTW to provide output in any programming language. However, a significant amount of "glue" code still needs to be manually added before implementation on any particular hardware platform. The generic real-time framework created automatically by RTW does not allow for true real-time, interrupt driven, deterministic code. In many applications, and motion control in particular, software needs to perform predictably and it is vitally important that some hard real-time deadlines are met.

At the time of writing, version 5.2 of MathWorks environment did not provide support for uploading of data from external sources for displaying and visualisation. The designers of Simulink leave a framework open which allows custom code to be attached to provide only a one-way link with an externally executing code. Thus, on-line parameter tuning can be implemented.

Despite its shortcomings the RTW is capable of generating raw high-level code from graphical diagrams and can form a solid starting point for a complete solution to motion control development.

2.3.3 Hypersignal RIDE

Hypersignal Rapid Integrated Development Environment (RIDE), available from Hyperception Incorporated, is a stand alone, graphical environment for the development and real-time implementation of DSP algorithms. A typical Hypersignal block diagram window is shown in Fig. 2.4.

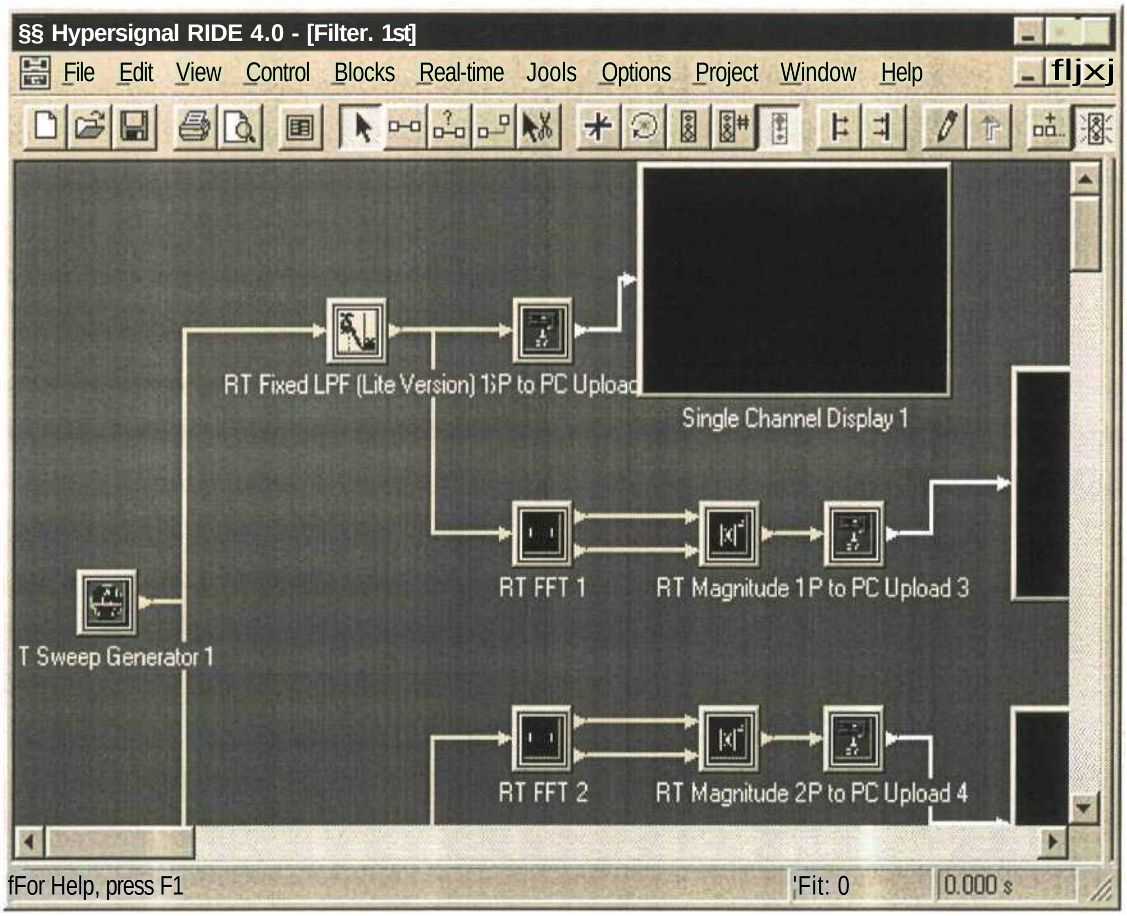


Fig. 2.4 : Typical Hypersignal RIDE diagram

The Hypersignal package supports a wide cross section of DSP hardware, ranging from low end cards to high end multiprocessor solutions. Its true strength lies in the seamless code generation from graphical block diagrams. Individual blocks can be made to run on either the host PC or on the target DSP hardware. Special blocks called *RT DSP to PC Upload* and *RT PC to DSP Download* are used to connect sections of the diagram executing on different platforms. Thus the complete controller can be initially made to operate on the host PC for simulation purposes, and then moved to the target hardware. In the final design some blocks can remain on the PC to provide bi-directional communication with the controller executing in real-time.

As far as the user is concerned the entire process of generating code is completely transparent. Hypersignal also handles the issue of ensuring that processes executing on DSP hardware as well as on the PC remain synchronised. While the block diagram is executing, parameters in most of the blocks can be changed and Hypersignal updates variables in the real-time code without interrupting execution.

Despite the strengths of this package in terms of its real-time support for a wide range of DSP hardware, it does not provide adequate support for modelling and simulation. Hypersignal is mainly targeted at the classical DSP applications and numerous sets of blocks are provided for applications such as communications, speech processing and filter design. Even in these fields Hypersignal does not quite match the functionality of dedicated modelling and simulation packages like MATLAB and Simulink.

Hypersignal is an open environment which lends itself to customisation, if necessary in a particular field. The package has been used to implement control of power electronics at the University of Natal, in particular an artificial intelligence solution for a boost rectifier [WORTHMANN 1] and classical motion controller [WALKER 1]. Researchers involved in both these projects needed to commit considerable effort to extend Hypersignal before using it. However a basic requirement for a truly integrated development environment is that users do not need to write any custom low-level code.

2.3.4 dSPACE

The German company dSPACE GmbH provides a number of DSP based hardware platforms and some of their products are specifically aimed at the control sector. In terms of software support they have the Total Development Environment (TDE) [VATER1].

The TDE is an extension of the MATLAB and Simulink environment introduced above, and as such it inherits all the modelling and simulation power from the MathWorks product family. However, real-time code generation and support are only provided for the dSPACE boards, which limits the choice of controller hardware.

Furthermore, the dSPACE product range is prohibitively expensive when compared to other DSP hardware vendors. For example the DS1102 controller board at an educational discount costs around US$3000, where as a PC32 card from Innovative Integration sells for less then US$1000 and provides similar performance.

The TDE does not currently make use of Simulink's external mode for communication between the block diagram and the code executing on target hardware. Thus downloading parameters from within the Simulink environment is impossible. dSPACE ship a program called *Cockpit* which takes over this functionality. However, the process of connecting various controls on the Cockpit front end with correct parameters in the Simulink diagram is not straight forward. Cockpit seems to use the intermediate files produced by Simulink to bring up a list of available parameters for each block on the diagram. Some Simulink blocks have a number of parameters associated with them which are internal to their operation and should not be modified dynamically. The process of choosing the desired parameters in Cockpit form the available list is not intuitive and could be especially confusing to a novice user. If the incorrect parameters are connected to Cockpit controls and modified online, the problem may be difficult to find.

Not using the standard Simulink external mode also means that there is no way for the user to start and stop the execution of the code once it is generated and downloaded to the target DSP platform. Execution starts immediately as the code is downloaded, and the default set of parameters is used. The user is responsible for ensuring that the initial set of parameters is safe. An error on the user's part could mean the controlled plant runs out of control on downloading the controller code and the only way to stop it from causing or sustaining damage would be a physical intervention, for example cutting off the power supply. There could simply not be enough time to start Cockpit and change the necessary parameters.

Although dSPACE provides all the necessary ingredients for a complete motion control development environment, there is still room for improvement.

2.4 Goals for the Project

Based on the discussion presented in this Chapter, the author decided that there was a need to develop a solution for the design and rapid prototyping of motion control which, while fulfilling all requirements as set out in section 2.2.5, meets following criteria :

(i) Low cost - the developed system needs to be affordable enough to appeal to the wider educational system,

(ii) Ease of use - as far as possible all functionality needs to be packaged in a single environment,

(iii) Computational power - the DSP hardware platforms supported need to provide adequate processing power to implement ambitious control projects.

The CSDE is designed mainly with educational institutions in mind, thus to address the above points it was decided to base the system on the MATLAB and Simulink package. This will reduce costs as most tertiary institutions already make extensive use of these modelling and simulation tools. Further, more ambitious projects may be attempted by students within limited time frames, as they will already be familiar with the software. MATLAB and Simulink are well established CACSD packages and the author's intention is not to modify them, but rather to extend and supplement their functionality as necessary to meet the goals set out section 2.2.5.

The standard external mode of Simulink has to be supported and any functionality not provided for under Simulink can be implemented externally. With the current version of the MathWorks software only the data visualisation needs to be done by a separate utility. The MATLAB and Simulink package is discussed in detail in the next Chapter.

Hardware targeted by the author's proposed system will initially be the PC32 DSP Controller card from Innovative Integration Inc. Choice of hardware was based on the cost to performance ratio. The PC32 card provides 32 bit, floating point power of a 60MHz TMS320C32 DSP processor from Texas Instruments, 16 bit resolution on A/D and D/A channels as well as an open architecture for expansion. This gives enough computing power to implement advanced control algorithms while at the same time being affordable enough (US$ 1000) for wide spread use in the tertiary environment.

While being capable and affordable, the PC32 card does not provide built-in support for motion control specific functionality. This problem was addressed in collaboration with other researchers, in particular Mr. Walker [WALKER1], who designed a custom extension card to the PC32. This custom card provides hardware support for a high resolution tachometer as well as PWM for switching of power electronic devices. This expansion card in not absolutely necessary for implementation of motion control on the PC32, but it does make the task easier by delegating those computationally expensive tasks to dedicated integrated circuits (IC). The hardware employed in the project is described in detail in Chapter 4.

2.5 Conclusion

The issues involved in the design process of a modern digital controller were discussed and a need for a high-level integrated tool was identified. Some available solutions, commercial and otherwise, were discussed and their individual strengths and weaknesses pointed out. From this argument a number of goals were set for the proposed CSDE.

The following Chapter introduces the MATLAB and Simulink environment which was chosen as the starting point for the author's project. Chapter 4 will continue by describing the hardware targeted.

CHAPTER THREE
OVERVIEW OF MATHWORKS TOOLS

3.1 Introduction

Chapter 2 discussed the steps involved in the design process of a motion controller, starting with modelling of the controlled plant, through simulation of the control algorithm to the final real-time implementation. Particular emphasis was placed on motion control design issues in the tertiary environment, and a need was identified to simplify the design process. The idea of an integrated development environment was introduced, to aid the designer by combining all necessary tools under one package.

MATLAB and Simulink from MathWorks Incorporated are well established products and are a defacto standard in terms of modelling and simulation in the academic community [FOSTER 1, GUMAS1, MIR0TZNIK1]. The MathWorks software as well as a number of other design and simulation packages were previously used and evaluated at the Electrical Engineering Department of University of Natal [KLEINHANS1]. Recommendations from researchers experienced with the available software packages led to the selection of Simulink and the RTW to form the foundation of the CSDE.

In this Chapter, an overview will be provided of the MATLAB, Simulink and the RTW packages. The intention is not to give an exhaustive discussion of the complete MathWorks range, but rather to focus on a few points of interest. Specific detail will only be provided on issues relevant to this thesis, in particular Chapters 5, 6, 7 and 8 which present the work undertaken in extending and unifying the MathWorks components into the CSDE.

3.2 MathWorks

MATLAB [MATHWORKS 1, 7] is a high performance mathematical language interpreter for technical computing and is particularly useful for problems which can be formulated in terms of vectors and matrices. Its open architecture lends itself to be customised by users and the system has evolved over a number of years of continuous use in academic and industrial environments. Fig. 3.1 shows the relationship between various MathWorks products. MATLAB forms the foundation of the core mathematical engine for the rest of all the MathWorks range. It provides

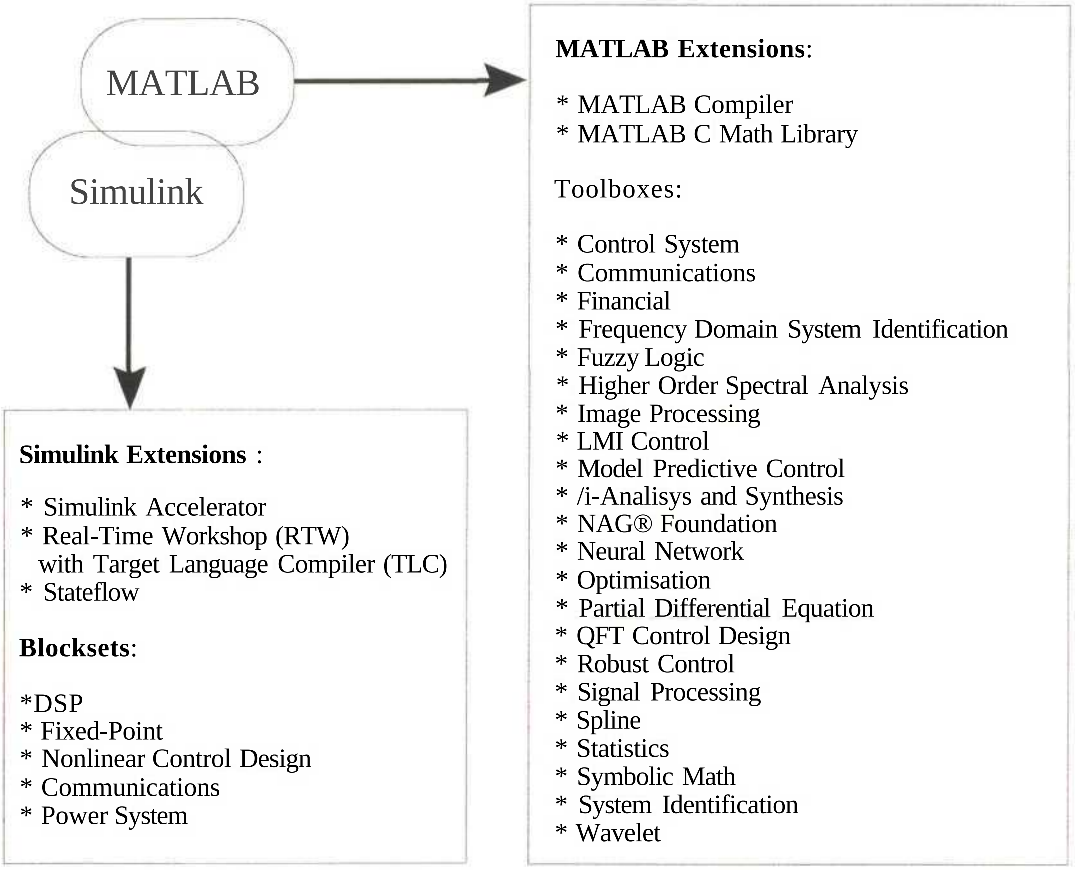


Fig. 3.1: MathWorks Product Family

a programming interface, numeric computation and has advanced data plotting capabilities. **Toolboxes** are collections of MATLAB functions which target specific areas of application. There is a rich selection of such toolboxes available from MathWorks to satisfy most needs. New toolboxes can be written using the high-level MATLAB programming interface or existing ones can easily be modified to meet custom needs. The wide spread use of MATLAB also resulted in a range of custom third party extensions being available. **Simulink** is a GUI extension to MATLAB which allows for dynamic simulation of nonlinear systems in a user-friendly graphical environment. MATLAB's core functionality, including access to toolboxes, is retained while the design is abstracted to a schematic block diagram level. **Simulink Extensions** form a set of tools which extend the functionality of Simulink. The RTW is of particular interest to this thesis as it forms a starting point for transforming the simulated systems into real-time prototypes. **Blocksets** are collections of Simulink blocks which target specific areas of application.

A complete discussion of all the available toolboxes, and extensions is beyond the scope of this thesis. The following sections will concentrate only on those MathWorks products which are directly relevant to the author's work and show how they help to form a complete rapid prototyping environment. Simulink, the RTW and the Target Language Compiler (TLC) will be introduced in more detail.

3.2.1 Host vs. Target Platforms

Before further discussing the MathWorks environment, it is important to define clearly how it integrates into the CSDE in terms of the two distinct hardware platforms. Namely, the host and target platforms.

The platform on which Simulink and the RTW run is denoted as the *host platform.* In the case of the CSDE it takes the form of a personal computer (PC) workstation running the Windows 95 operating system.

The code automatically generated by the RTW is executed on the *target platform.* The target platform could be any of the following examples :

(i) A stand alone embedded microprocessor based controller linked to the host via a serial connection,

(ii) A DSP processor on an expansion card in one of the host's slots, communicating with the host via the ISA or PCI bus.

(iii) Another workstation linked to the host via a local area network (LAN) or the world wide web (WWW)

(iv) The host workstation itself

In the case of the CSDE the target platform is the PC32 controller card installed on the host PC's ISA bus.

3.3 Rapid Prototyping with MathWorks

The set of tools from the MathWorks, as introduced above, form a solid base for modelling, design and simulation of digital controllers. In addition, the RTW provides a framework for extending these tools into a complete rapid prototyping environment which can provide the following functionality:

(i) **Automatic code generation.** The ability to automatically turn a graphical simulation into executable code means users are freed from all software development issues. Controller development takes place in the standard Simulink environment and the RTW handles the transformation into a real-time prototype transparently to the user.

(ii) **On-line parameter tuning.** Controller parameters can be modified in real-time without interrupting the prototype's execution. A feature of Simulink called *external mode* establishes a communication link between the host PC and the controller executing on the target hardware. The Simulink block diagram effectively becomes a GUI for the real-time controller prototype.

(iii) **Visualisation.** Release 10 of the MathWorks environment, which was used as a base for the CSDE, does not support uploading of data into Simulink from external sources. This functionality was implemented by the author outside the MathWorks system as described in Chapter 8. Release 11 from MathWorks provides the necessary functionality, but was shipped too late to be included in the current version of the CSDE.

Fig. 3.2 shows the relationship between the MathWorks components involved in automated code generation. Stateflow allows one to include state-machine logic into Simulink block diagrams, but was not available for inclusion at the time of the development of the CSDE. It could prove to be a valuable tool during the development of real-time code for embedded applications.

The starting point for generating code is a standard Simulink block diagram. Any functionality specific to the target hardware can be included at this level in the form of custom blocks. The RTW processes the block diagram into an intermediate netlist description. This intermediate netlist is parsed and converted to source code by the TLC. The TLC output is passed to a target specific compiler where it is combined with custom real-time support code and results in a stand

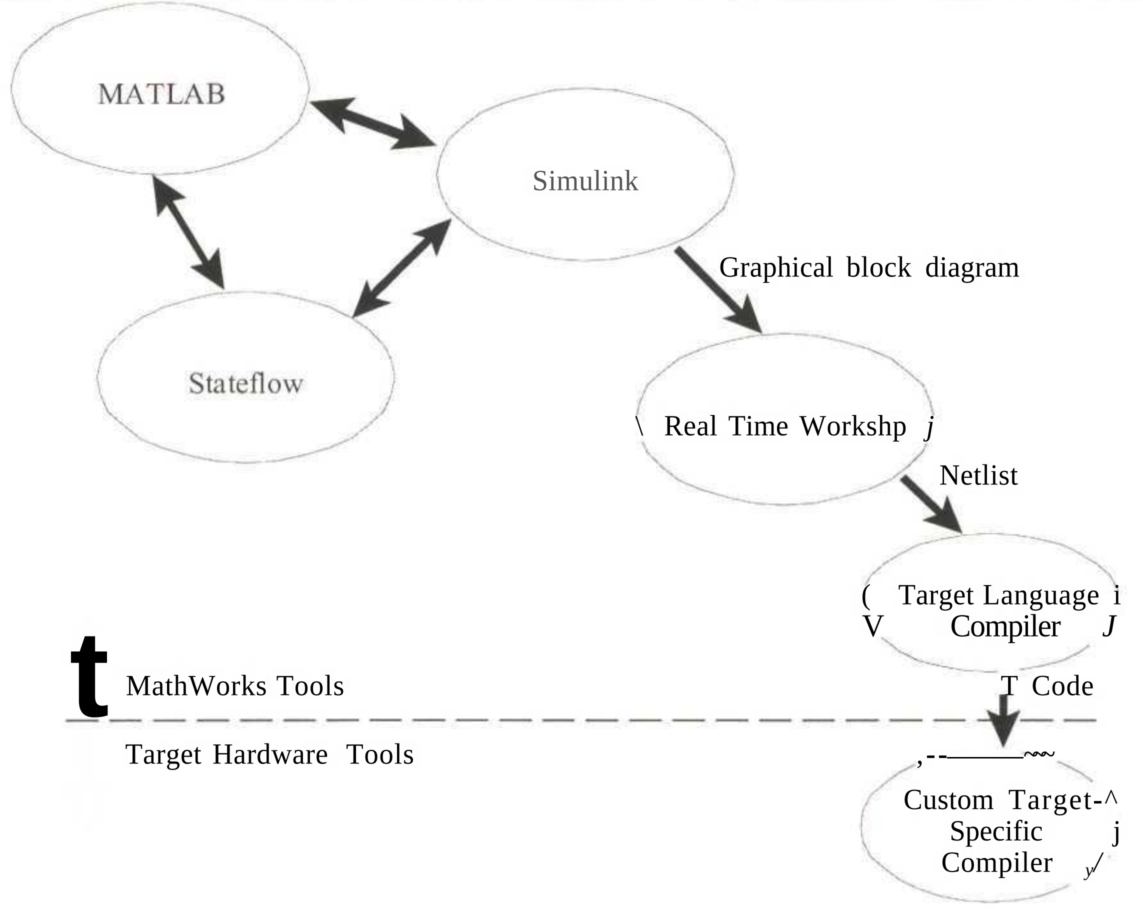


Fig. 3.2 : Relationship between MathWorks tools for code generation

alone executable. The process of converting a Simulink block diagram into executable code is discussed in more detail in sections 3.5 and 3.6.

Automatically generating real-time code is an important requirement of the rapid prototyping process. Equally important, however, is the ability to interact with the generated prototype in real-time without adversely affecting its performance. For this purpose Simulink supports the *external mode.* When placed in external mode, Simulink acts as a remote GUI to the prototype controller, as shown in Fig. 3.3. Any changes to parameters on the Simulink block diagram are automatically communicated to the equivalent real-time executable. This mechanism allows for on-line controller tuning without the need to recompile the controller code after each change.

The above discussion briefly presented two features of the MathWorks environment which allow its expansion into a complete rapid prototyping system, namely automated code generation and external mode. The following sections introduce Simulink, the RTW and the TLC and elaborate on issues important to the design of the CSDE around these components.

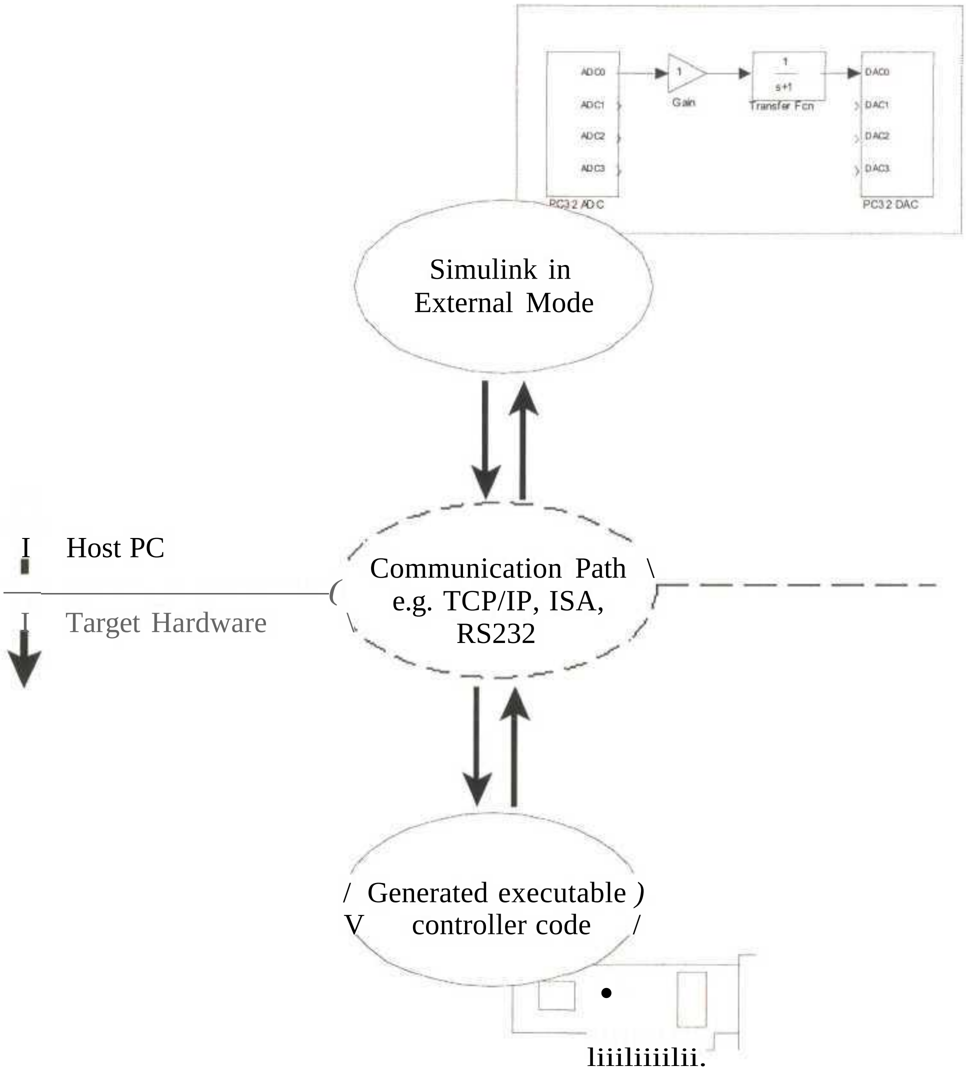

ADC0
ADC1
ADC2
ADC3
PC32 ADC
1
Gain
1
s+1
Transfer Fcn
DAC0
DAC1
DAC2
DAC3
PC32 DAC
Simulink in External Mode
Host PC
Target Hardware
Communication Path e.g. TCP/IP, ISA, RS232
Generated executable controller code

Fig. 3.3 : Interaction with prototype controller via Simulink external mode

3.4 Simulink

Simulink [MATHW0RKS2] is a GUI from MathWorks Inc. It is a software package for modelling, simulation and analysis of dynamic systems and can utilise the proven MATLAB engine. It supports linear and nonlinear systems, modelled in continuous and/or discrete time. Discrete systems can also be multi-rate, i.e. Have different parts that are sampled or updated at different rates.

For modelling, Simulink provides a user-friendly GUI allowing users to intuitively build models as graphical block diagrams, using click-and-drag mouse operations. With this interface users no longer need to formulate equations in a language or programme, but can simply choose from a comprehensive collection of block libraries and additional toolboxes. Simulink also allows for custom blocks and libraries to be added by users to accommodate their specific needs.

Simulink models are hierarchal, thus complex systems can be broken down and approached in a top-to-bottom or bottom-to-top manner. Systems can then be viewed at the top level, and by double-clicking one can navigate down through the lower levels to see increasing amounts of detail.

In the context of the CSDE, Simulink provides a comprehensive graphical environment for the modelling, design and simulation of motion control. It is also a well proven package and is widely used in tertiary institutions, both in South Africa and abroad [FOSTER1, GUMAS1, MIR0TZNIK1]. These factors mean that Simulink is well suited to form a base for the CSDE as discussed in Chapters 1 and 2. The following subsections will introduce a number of practical aspects of Simulink. Particular attention is paid in sections 3.4.4, 3.4.5 and 4.4.6 to issues pertaining to creating custom Simulink blocks, as this is important to the author's work presented in Chapter 7. Simulink's external mode is discussed in section 3.4.7.

3.4.1 Simulink Blocks

Blocks are the basic elements from which Simulink models are built. Virtually any dynamic system can be modelled by interconnecting blocks appropriately. Simulink blocks fall into two basic categories : virtual and non-virtual blocks [MATHW0RKS3]. Non-virtual blocks play an active role in the simulation of a system, thus adding or removing a non-virtual block changes the model's behaviour. Virtual blocks, by contrast, play no active role in the simulation. They simply help to organize a model graphically to make it more readable. Fig. 3.4 illustrates a number of typical Simulink blocks in a diagram and labels each as either virtual or non-virtual.

3.4.2 Libraries and Library Links

Simulink libraries are collections of blocks that allow blocks to be grouped according to their specific area of application. There are a number of standard libraries shipped with Simulink, as shown in Fig. 3.1, and further blocksets can be added. New libraries can be created containing existing blocks, subsystems or completely new custom blocks. Fig. 3.5 shows an example of a typical Simulink library and a logical link with a library block used in a diagram.

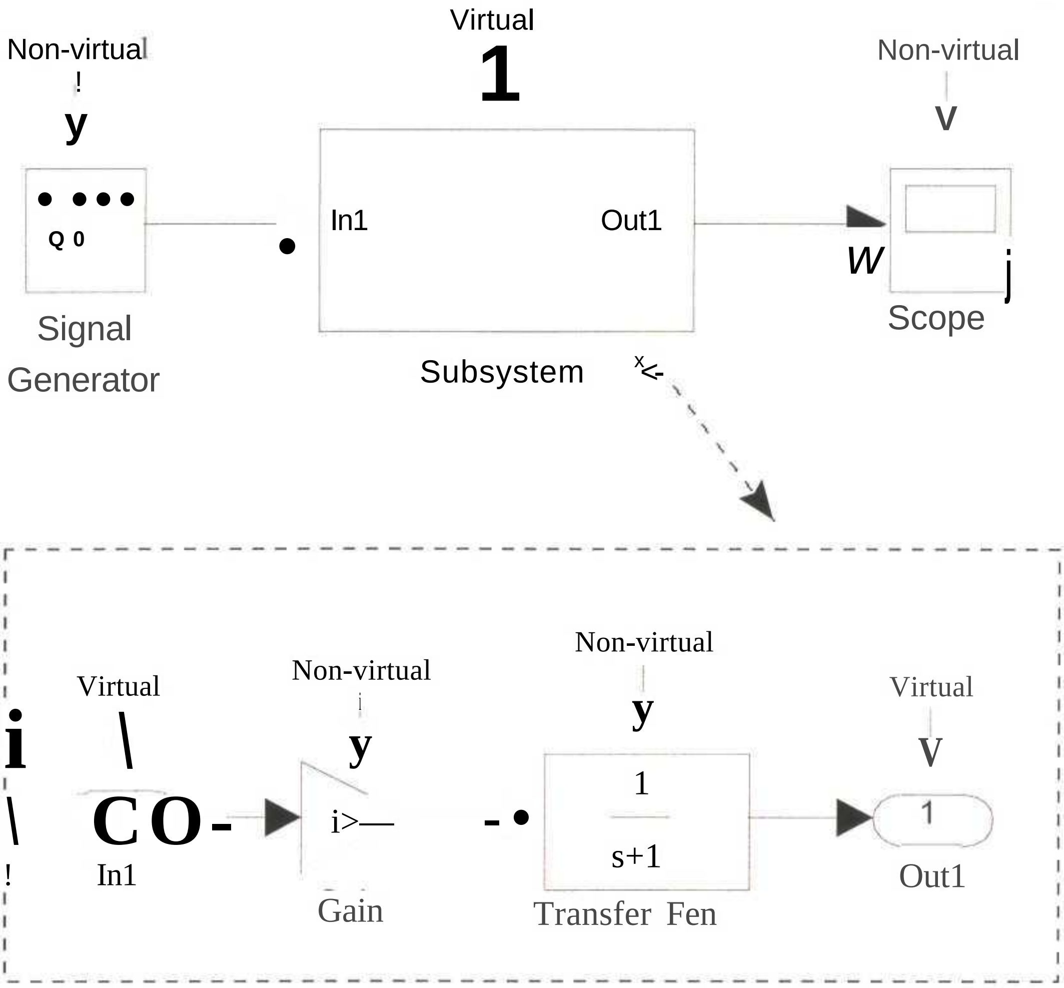


Fig. 3.4 : Virtual and non-virtual Simulnik blocks

A library link is established whenever a library block is used in a Simulink diagram. Any changes to the library block will automatically be reflected in all its instances. Thus, changes to a library block can have a widespread effect on a number of Simulink diagrams.

3.4.3 Subsystems

Subsystems, like virtual blocks, play no active role in a Simulink block diagram. Their main function is to collect blocks together into convenient, logical groups to help break down large systems into a more manageable hierarchal structure, Fig. 3.4 demonstrated the use of a simple subsystem. Five types of subsystems can be identified, as shown in Fig. 3.6 :

(i) *Simple Subsystem.* Its only purpose is to keep block diagrams neat and readable.

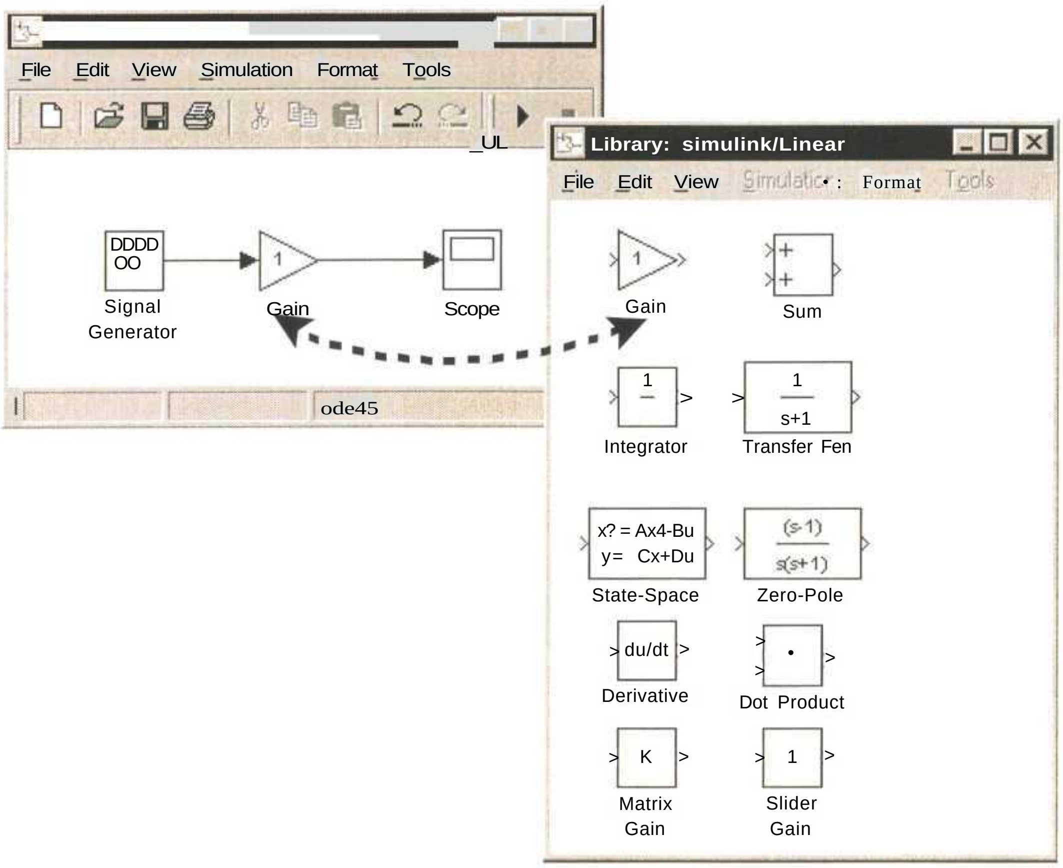


Fig. 3.5 : Example of a library and a library link

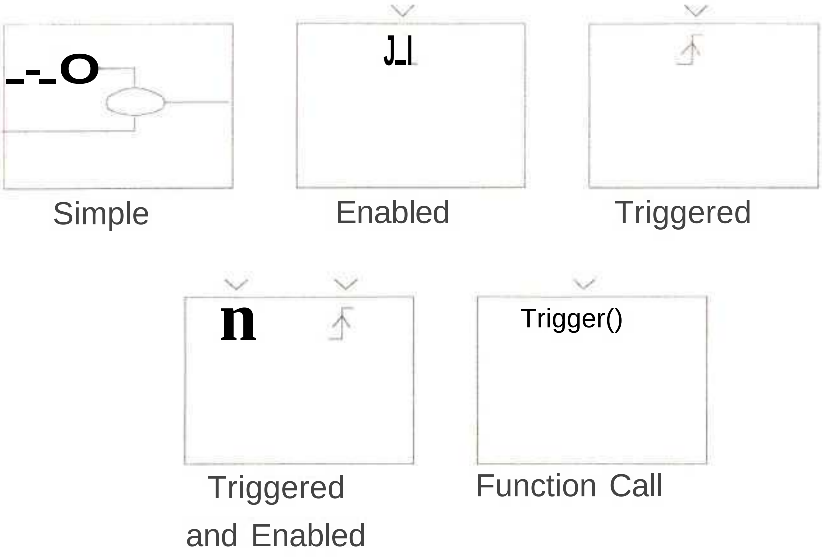


Fig. 3.6 : Subsystem types in Simulink

(ii) *Enabled Subsystem.* An external signal can collectively enable or disable all blocks in a Enabled Subsystem. The execution of the contained blocks continues as long as the control signal remains positive,

(iii) *Triggered Subsystem.* The execution is controlled by the level changes in an external control signal. The contained blocks execute only once for each trigger event,

(iv) *Triggered and Enabled Subsystem.* Execution occurs only if both an enable signal is present and a trigger event occurs,

(v) *Function-call Subsystem.* This type of a subsystem is only used when code is generated from a Simulink diagram using the RTW. The code generated for all contained blocks is placed in a separate function, making it easy to assign parts of the block diagram to external events like interrupts.

3.4.4 S-functions in a Simulink Block Diagram

The standard set of Simulink blocks can be expanded by creating new custom blocks in the form of S-functions. By allowing high-level code to be seamlessly integrated into Simulink block diagrams, S-functions, or *system functions,* provide a powerful way to extend and further customise the capabilities of Simulink. In the CSDE, S-functions are used to create custom driver blocks specific to the PC32 controller and expansion hardware.

Apart for their application in the CSDE for accessing hardware, S-functions can be used in a number of other applications :

(i) Creating new simulation blocks that are not supported in Simulink. Using a high-level programming language can allow a solution in a single S-function block, in place of a potentially complex interconnection of standard blocks.

(ii) Importing m-file or C-code algorithms into a block diagram simulation. This option could save existing and well tested code from having to be re-created using interconnections of standard Simulink blocks.

(iii) Incorporating graphical animations into the simulation.

S-functions, like all standard Simulink blocks, can be fully described in terms of the following equations, where y is the vector of outputs, x is the vector of states and u is the vector of inputs to the block:

(i) $y = f_o(t,x,u)$ -> outputs

(ii) $\dot{x}_c = f_d(t,x,n)$ -> derivatives (in continuous time)

(iii) $x_{IIM} = f_u(t,x,u)$ -> update (discrete)

(iv) $x = x_c + x_d$ -> states

Fig. 3.7 shows how the input, output and states vectors of a block relate to each other. At every simulation step Simulink updates each S-function's vectors via calls to a standardised Application Program Interface (API) [MATHW0RKS4, 5] and makes the results available to other blocks in the diagram. The code contained in the API defined function calls can interact with hardware or other external sources of data. This mechanism allows S-fiinctions to provide an interface between standard Simulink blocks and outside hardware devices.

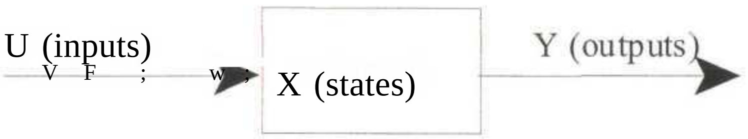


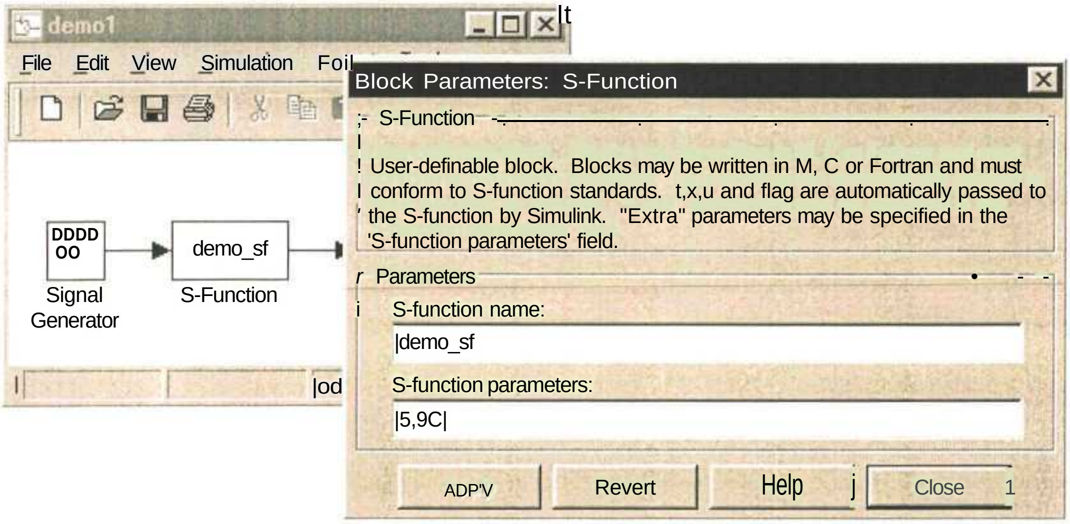
Fig. 3.7 : Relationship between input, output and states vectors of a Simulink block

The code for S-functions can either be written in MathWorks' own m-file format or a third party C compiler can be used to generate executable code specific to the host platform in C-MEX format. The following two subsections will describe how S-functions are created and introduce the necessary API in more detail.

3.4.5 Creating S-functions

Creating a new S-function for use in Simulink requires two steps. Firstly a graphical front-end block is needed. Simulink provides a generic S-function block in the Nonlinear Blocks library which forms a starting point for all new S-functions. Secondly the algorithm has to be coded into a format compatible with the S-function API introduced in section 3.4.6.

The graphical front-end block forms a user interface to the underlying S-function code and allows parameters to be passed which can dynamically influence its behaviour. Simulink provides a generic front-end block, which can then be customised by a process called *masking*. Without masking parameters can be manually formatted and entered as an array in the generic dialog box. Simulink's Mask Editor allows the design of fairly elaborate user interfaces which support familiar Windows based features like drop-down and check boxes. The S-function mask can also be designed to perform range checking of parameters to filter out invalid user input before passing it to the code for processing. Fig. 3.8 shows the generic unmasked block used for the S-function *demo_sf,* while Fig. 3.9 shows the same S-function after it has been masked. The masked interface allows a more intuitive way of entering the two required parameters - gain and phase shift.



Block Parameters: S-Function

S-Function

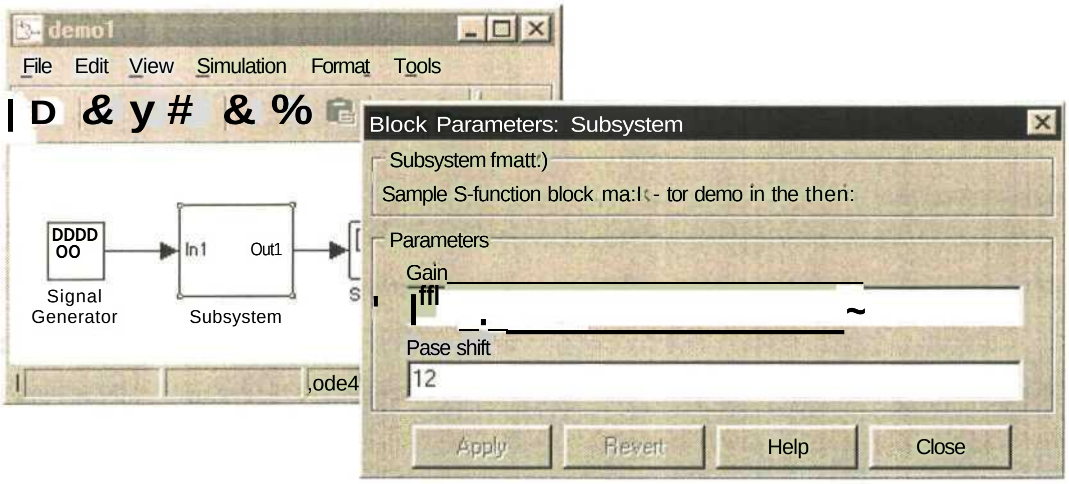
User-definable block. Blocks may be written in M, C or Fortran and must conform to S-function standards. t,x,u and flag are automatically passed to the S-function by Simulink. "Extra" parameters may be specified in the 'S-function parameters' field.

Parameters

S-function name: |demo_sf

S-function parameters: |5,9C|

Fig. 3.8 : A sample S-function block and its parameters dialog box



Block Parameters: Subsystem

Subsystem fmatt:)

Sample S-function block ma:I - tor demo in the then:

Parameters

Gain: ffl

Pase shift: 12

Fig. 3.9 : A sample S-function block with a masked interface

As mentioned, the code for an S-function can be supplied in one of the two supported formats:

(i) A standard MATLAB m-file.

(ii) A C-MEX file compiled for the specific host platform

Both formats have to comply with an API set out by Math Works. A generic template for each of the formats is provided with Simulink to make this integration easy.

Standard m-file S-functions are interpreted at runtime by the MATLAB engine, thus they are fully portable and platform independent. C-MEX S-functions, on the other hand, are executed at runtime by the host processor. In the case of a PC running Windows, the C-MEX file takes the form of a 32-bit Dynamic Link Library (DLL). A number of third party compilers are supported for that purpose:

(i) Watcom C

(ii) Borland C

(iii) Microsoft Visual C/C++

Since the main interest of this thesis lies in the generation of real-time code, only C-MEX functions are relevant. Thus, in the following Chapters all references to S-functions will refer to C-MEX functions and points specific to m-file S-functions will be omitted. Fig. 3.10 demonstrates steps in creating a custom C-MEX S-function and inserting it into a Simulink block diagram.

3.4.6 **S-function API**

To function correctly, S-functions must adhere to the API as defined by MathWorks. Apart from the vector updates, each S-function is responsible for implementing API calls which initialise it at the commencement of a simulation and clean up at the end of a simulation run. A number of the API functions are compulsory, while some calls are optional and their use depends on a particular area of application. Table 3.1 lists a subset of the S-function API relevant to the work presented in this thesis (compulsory functions are marked in bold).

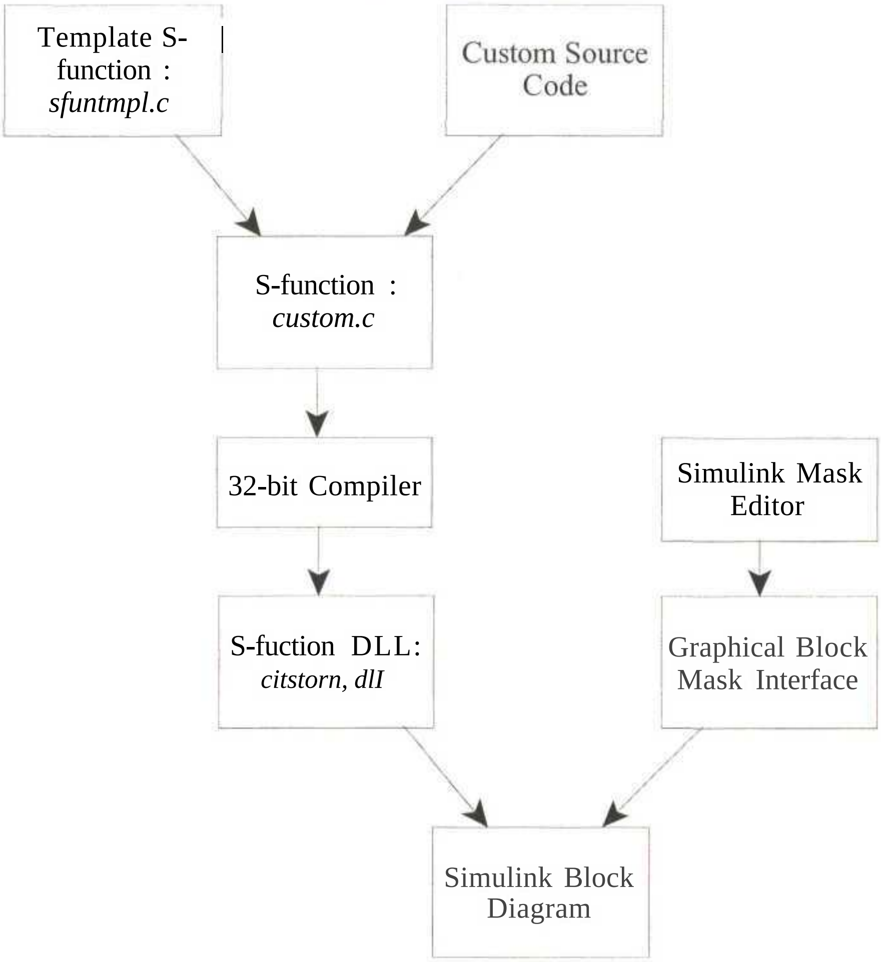


Fig. 3.10 : Incorporating a custom C-MEX S-function into a Simulation

Simulink Request	**API Function Call**
Initialisation at beginning of simulation	**mdllnitialiseSizes** **mdllnitializeSampIeTimes** mdlStart
Calculation of output vector and derivatives and update of states	**mdlOutputs** mdlUpdate mdlDerivatives
Termination and cleanup tasks	**mdlTerminate**

Table 3.1 : Simulink S-function API (compulsory functions in bold)

The functionality of the API calls, listed in Table 3.1, is as follows [MATHW0RKS3, 4] :

(i) mdllnitialiseSizes - it is the first call Simulink performs while interacting with a S-function. The sizes of the input, output and states vectors as well as the number of parameters is specified in this function.

(ii) mdllnitializeSampleTimes - here information about the S-function's sampling is specified. All S-functions used in the CSDE are set to continuous time mode and are thus executed at every sampling step of the model. If discrete time mode is selected the S-function will only be updated at the specified sample hits.

(iii) mdlStart - this optional routine contains any application specific initialisation that needs to be performed once only at the start of the model execution. S-functions which implement hardware specific drivers might use this call to initialise the hardware.

(iv) mdlOutputs - here code is included to compute the S-function's output vector. This function is called at every sampling step of the model.

(v) mdlUpdate - this optional call is performed only once per every major integration step of the model. Typically it is used to update the discrete state vector.

(vi) mdlDerivatives - all code involved in calculating the derivatives should be placed in this optional function.

(vii) mdlTerminate - it is a compulsory function and should contain all termination code for the S-function. Any memory allocated during initialisation should be freed up here and any hardware devices controlled by the S-function should be placed into a safe state.

Apart from the function calls listed above, the S-function API also specifies a common data structure format - *SimStruct* [MATHW0RKS4, 5]. Each S-function in a Simulink model has a corresponding *SimStruct* structure allocated for its private use. In addition there is a single global (root) *SimStruct* defined for the model. The various *SimStructs* in a model are arranged in a hierarchal manner starting at the root (similar to a directory tree). Each structure contains a pointer to its parent as well as a pointers to the root. Thus, the root *SimStruct* will have a NULL parent pointer and all other *SimStructs* will point to it. Table 3.2 shows a broad outline of the *SimStruct* structure. The *SimStruct* is a flexible, pointer based data structure, and is scaled according to the particular needs of the given S-function or model. Some of the fields listed in Table 3.2 can represent relatively complex structures themselves. To simplify access to the data

Field	**Contained Data**
Version	version of the *SimStruct*
Parent	this *SimStrucfs* parent
Root	the root *SimStruct* \
Sizes	all size information - e.g. number of inputs, outputs, states, sample times
Inputs	input arguments passed from the block diagram
Vectors	vectors - e.g. input, output, states, derivatives, parameters
T	simulation time - not used for real-time
TFinal	final simulation time - not used for real-time
TCount	additional counter used for accurate time base
StepSize	sampling time period on which to update
MinorTime StepFlag	flag indicating that a minor time step is taking place
States	additional timing information - e.g. skew time, current and next sample time
Utility	temporary storage and user defined data

Table 3.2 : S-function API data structure - *SimStruct*

contained in the *SimStruct* MathWorks define a host of macros. The subset of macros relevant to the CSDE can be summarized as follows :

(i) ssSetSolverName - selects the solver to use in integration.

(ii) ssSetNumSFcnParams - sets the number of parameters expected from the block mask,

(iii) ssSetSFcnParamNotTunable - used to set some of the parameters as not tunable. Simulink will report an error if the user attempts to change those parameters while simulation is in progress,

(iv) ssSetNumlnputPorts - sets the number of input ports to the block.

(v) ssSetlnputPortWidth - sets the vector width for each port.

(vi) ssSetlnputPortDirectFeedThrough - specifies which of the input ports are used directly in the computation of outputs. Simulink uses this information to set the update sequence for the blocks in a model.

(vii) ssSetNumOutputPorts - sets the number of output ports from a block.

(viii) ssSetOutputPortWidth - sets the vector width for each output port.

(ix) ssSetNumSampleTimes - specifies the total number of sample times for the block,

(x) ssSetT - forces the simulation time to a specific value,

(xi) ssSetSampleTime - configures the sampling characteristics for a particular sample time,

(xii) ssSetTFinal - sets the stop time for a simulation run. Setting this to zero indicates an unlimited time, and is used for most real-time applications,

(xiii) ssSetSolverStopTime - sets the time period for the integration algorithm.

(xiv) ssGetNumSFcnParams - reads the number set by ssSetNumSFcnParams.

(xv) ssGetSFcnParamsCount - returns the actual number of parameters entered by the user.

(xvi) ssGetSampleTime - reads the sample time setting as set by ssSetSampleTime.

(xvii) ssGetDefaultParam - returns a pointer to the parameter array.

(xviii)ssGetStepSize - returns the base sampling period for the model,

(xix) ssGetChecksum - used to access the checksums calculated for the model by Simulink. The checksums are used during code generation to match executables with their corresponding block diagrams,

(xx) ssGetSolverStopTime - reads the value set by ssSetSolverStopTime.

3.4.7 External Mode and Data Logging

The preceding sections introduced the various building blocks of a Simulink block diagram and showed how S-function can be used to add custom functionality. Sections 3.5 and 3.6 will describe how a Simulink diagram can be used by the RTW to automatically produce a real-time prototype. A block diagram can also serve as **a** remote GUI for the generated prototype, thanks to Simulink's external mode. External mode establishes and maintains a communication link between Simulink running on the host platform and the generated prototype code executing on the target platform as was shown in Fig. 3.3. The execution on the target platform can be started or suspended in a similar fashion to the traditional Simulink block diagram simulations. Any parameter changes are automatically downloaded and updated in the target executable's *SimStruct* data structure without interrupting real-time execution. Simulink does not make provision for data logging and visualisation in real-time and this functionality is implemented in the CSDE using a custom stand alone utility, *Scope.* The overall logical interaction is shown in Fig. 3.11.

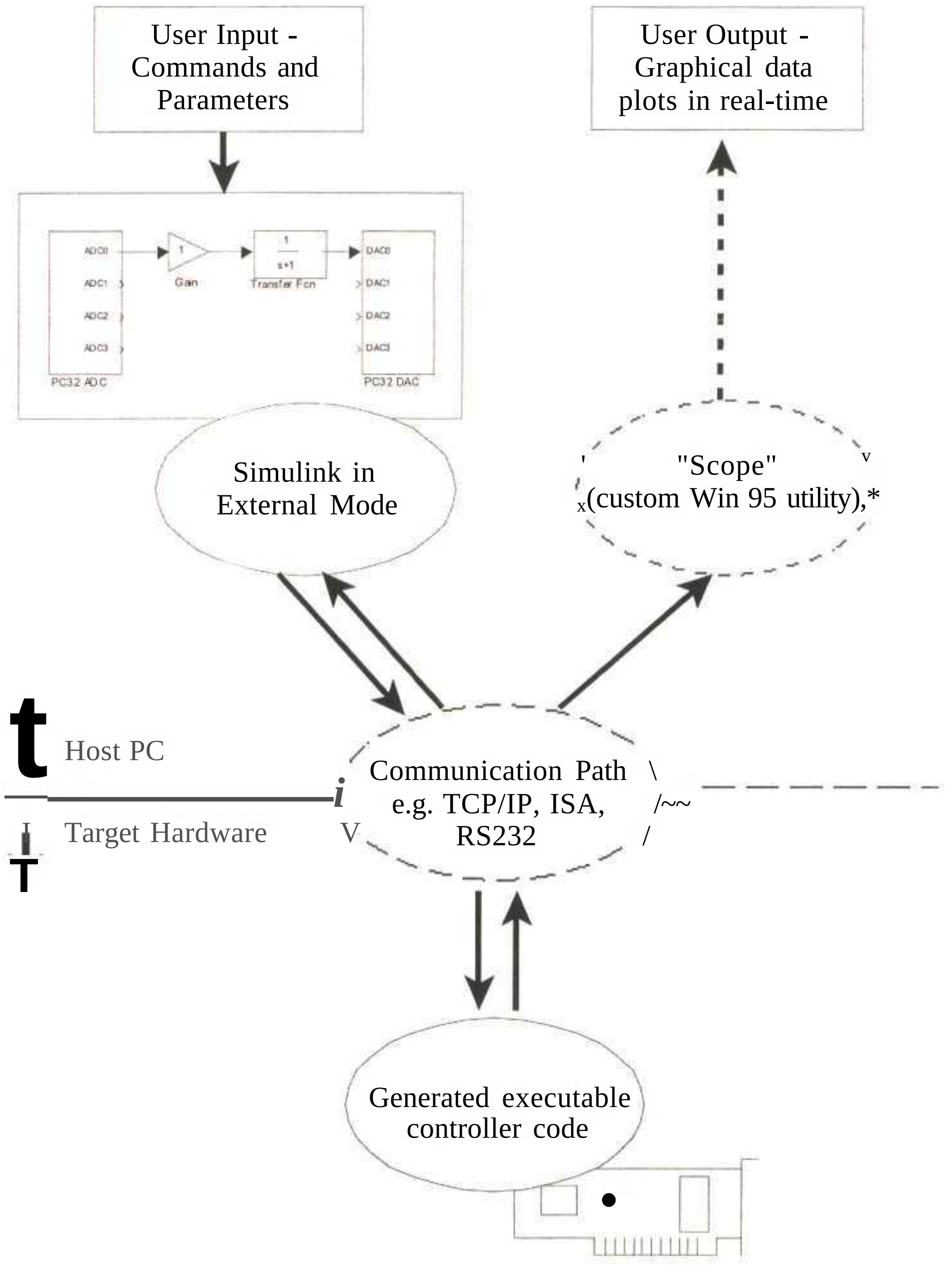

User Input - Commands and Parameters
User Output - Graphical data plots in real-time
ADC0
ADC1
ADC2
ADC3
PC32 ADC
1
Gain
1
s+1
Transfer Fcn
DAC0
DAC1
DAC2
DAC3
PC32 DAC
Simulink in External Mode
"Scope" (custom Win 95 utility)
Host PC
Target Hardware
Communication Path e.g. TCP/IP, ISA, RS232
Generated executable controller code

Fig. 3.11: Simulink external mode and data visualisation

The external mode communication link can be implemented over a variety of interfaces :

(i) Transmission Control Protocol (TCP) - A Local Area Network (LAN) or an Internet connection can serve as a medium for external mode,

(ii) Serial Connection - A traditional RS232 port can connect the target platform to the host PC.

(iii) Dual Port RAM (DPRAM) - targets connected to one of the host's expansion buses can share an area of memory. This is the method used by the CSDE.

The ability to tap into the target executables data structures and control its execution without adversely affecting its performance in real-time presents the user with a powerful tool. The performance of the generated code can be fine tuned on-line while it is responding to real world inputs.

Also the combination of external mode and data logging allows an easy means to verify, in real-time, that the generated code performs equivalently to Simulink simulations. Inputs identical to those used during simulation can be applied to the target and the corresponding outputs can then be compared graphically on the host work station. This way the user can verify that the RTW correctly converted a given block diagram into executable code.

The preceding subsections described the design and simulation environment of Simulink. The following sections will describe how the Simulink diagrams are converted to executable code customised to a specific hardware platform.

3.5 Real-Time Workshop

The RTW extension to Simulink [MATHW0RKS5] forms an environment whereby Simulink block diagrams can be converted directly into executable real-time code. The RTW is a starting point for a true *rapid-development* environment and allows a direct path from system design, through simulation to prototyping and final implementation. It can be applied to a variety of areas :

(i) Real-time control - Control algorithms designed using MATLAB/ Simulink can be compiled into standalone executables and downloaded to target hardware.

(ii) Real-time signal processing - MATLAB and Simulink are well suited to the design of signal processing algorithms which then can be targeted to a DSP platform,

(iii) Hardware-in-the-loop (HIL) simulation - responses of a real life plant can be included into a simulation at an early stage in the controller design,

(iv) Online real-time parameter tuning - Simulink's External Mode can form a graphical front-end to a real-time program. Parameters can be changed, and the effects of such changes observed in real-time, without interrupting controller execution,

(v) High speed stand-alone simulations - performance gains can be achieved by compiling a Simulink diagram to execute directly on the host platform, without having to go through the MATLAB interpreter,

(vi) Generation of raw ANSI C code - Simulink diagrams can be exported for use in other simulation packages.

Within the CSDE, the RTW is used to generate ANSI C code representation of Simulink block diagrams. The following subsections will introduce the code generation process and show how custom source code can be linked in.

3.5.1 Code Generation Process

The RTW goes through a three stage process to convert the graphical Simulink model into a final executable, as shown in Fig. 3.12 :

(i) RTW Build - The graphical block diagram is broken down and described in terms of an intermediate *RTW-file* (*.rtw). This ASCII text file contains all information about the diagram's blocks, hierarchy, interconnections and parameters. A compiler specific *makefile* (*.mk) is also generated.

(ii) TLC Parse - The Target Language Compiler parses through the RTW-file and interprets appropriate TLC-files to create source and header files. A list of all block parameters in the model is also generated and saved as *a parameter file* (*.prm).

(iii) Platform Specific Build - A compiler specific to the target hardware is used to process the generated files and link them to the real-time kernel to produce the final executable.

After this process is completed, the RTW can be configured to automatically download the generated executable to the target hardware and start execution.

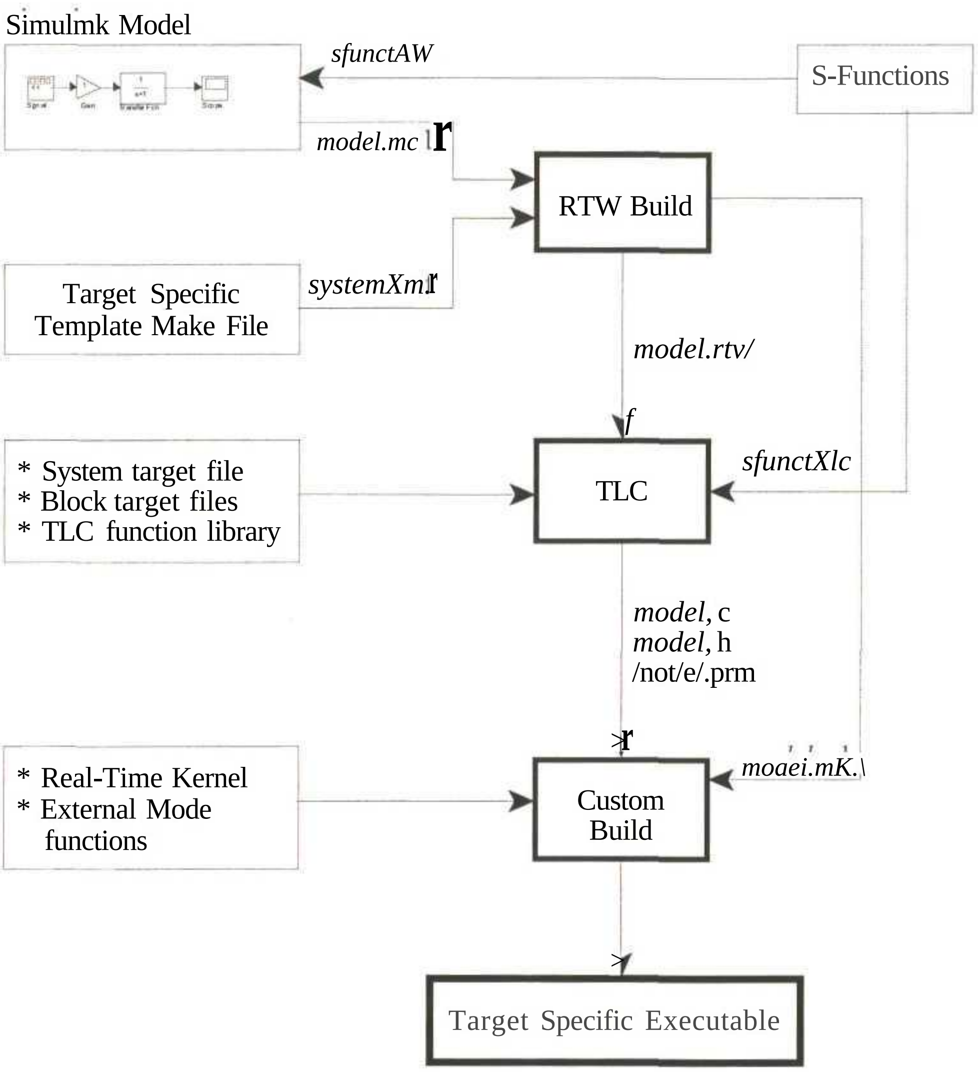


Fig. 3.12 : Real-Time Workshop Code Generation and Build Process

3.5.2 Structure of the Generated Code

The RTW structures the generated source code into two distinct layers :

(i) Platform Dependant Layer (PDL) - the source code for this layer is not automatically generated by RTW, and needs to be manually customised specifically for the targeted platform. The various components of the PDL in the CSDE are introduced in detail in Chapter 6.

(ii) Platform Independent Layer (PIL) - the source code here is directly generated by the RTW from a Simulink diagram and is structured to conform to the S-function API.

Structuring the code in this manner ensures maximum portability - only the PDL needs to be changed to port the design to a particular target platform. The PIL is isolated from hardware specific issues and can be ported to any platform which supports the S-function API. Fig. 3.13 demonstrates the basic structure of a RTW generated program. Conforming to the S-function API means that all data structures in the PIL are kept external to the code using the standard Simulink *SimStruc* data structure. This allows an external process to modify a programme's parameters without interrupting its execution, as mentioned in section 3.4.7.

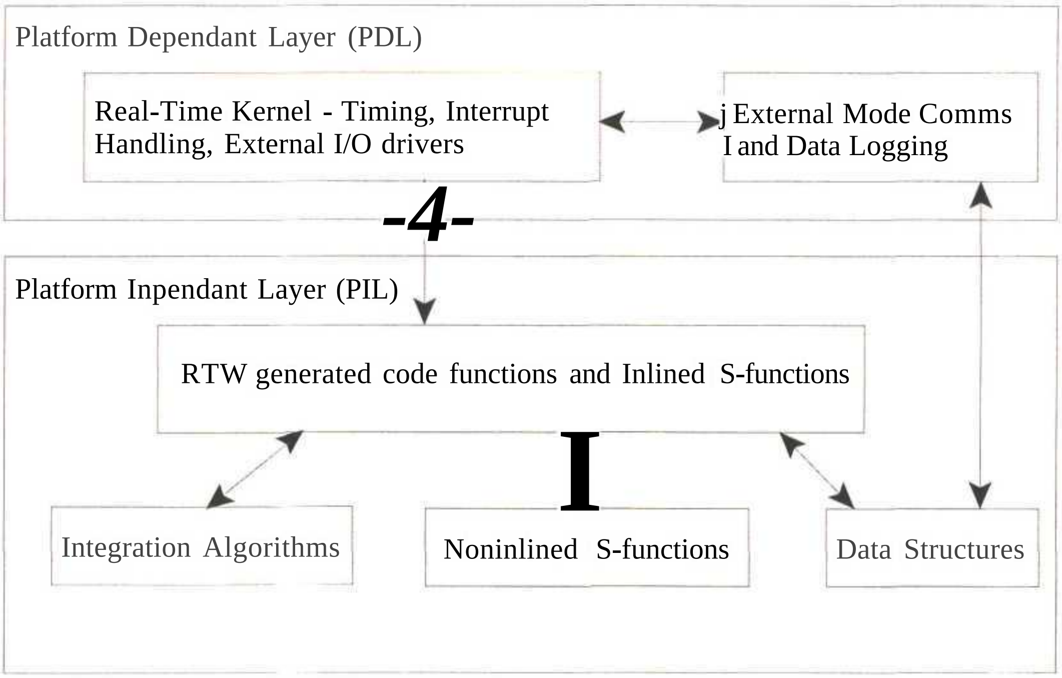

Platform Dependant Layer (PDL)
Real-Time Kernel - Timing, Interrupt Handling, External I/O drivers
j External Mode Comms I and Data Logging
-4-
Platform Inpendant Layer (PIL)
RTW generated code functions and Inlined S-functions
I
Integration Algorithms
Noninlined S-functions
Data Structures

Fig. 3.13 : Basic Structure of a RTW Generated Program

3.5.3 Platform Dependent Layer

The Real-Time Kernel (RTW) forms the backbone of the PDL and is specific to a particular target platform. It serves to isolate the PIL from hardware issues. The RTK needs to ensure that adequate processing resources are available to meet timing deadlines of the generated code. In cases where the target platform does not automatically support multitasking, an emulation scheme *ox pseudo-multitasking* can be implemented.

The RTK is also responsible for reserving memory resources for data structures and dynamically allocating all spare processing time to the external mode comms and data logging which are defined as low-priority background tasks.

The details of the PDL implementation specific to the PC32 hardware platform are provided in Chapter 6.

3.5.4 Platform Independent Layer

To generate code for the PIL, the RTW uses the Target Language Compiler (TLC) which is further introduced in section 3.6. The TLC allows full control over the generated source code. A separate TLC-file is provided for each standard Simulink block and there are also global TLC-files. The set of TLC-files shipped with RTW results in the generation of code compliant with the ANSI C standard. The source code generated for the PIL conforms to the S-function API as set out in section 3.4.

3.5.5 S-functions in RTW

The use of S-functions provides a powerful mechanism for customising the generated code for the particular hardware platform. Particularly, S-functions can be used to incorporate hardware specific driver level code into the final executable. Most hardware platforms targeted by the RTW will incorporate some form of external inputs and outputs (I/O), as well as a range of application specific peripherals. These devices can be represented as graphical blocks on the Simulink block diagram, and the code needed to initialise and drive them can be incorporated into a S-function.

Two types of S-functions can be used by the Real-Time Workshop :

(i) Inlined S-functions - a separate TLC-file is created for the S-function. During the build process the TLC uses this file to incorporate, or inline, target-specific code of the S-function directly into the generated source code.

(ii) Non-inlined S-functions - target-specific code for the S-function is provided in a standard source file conforming to the S-function API.

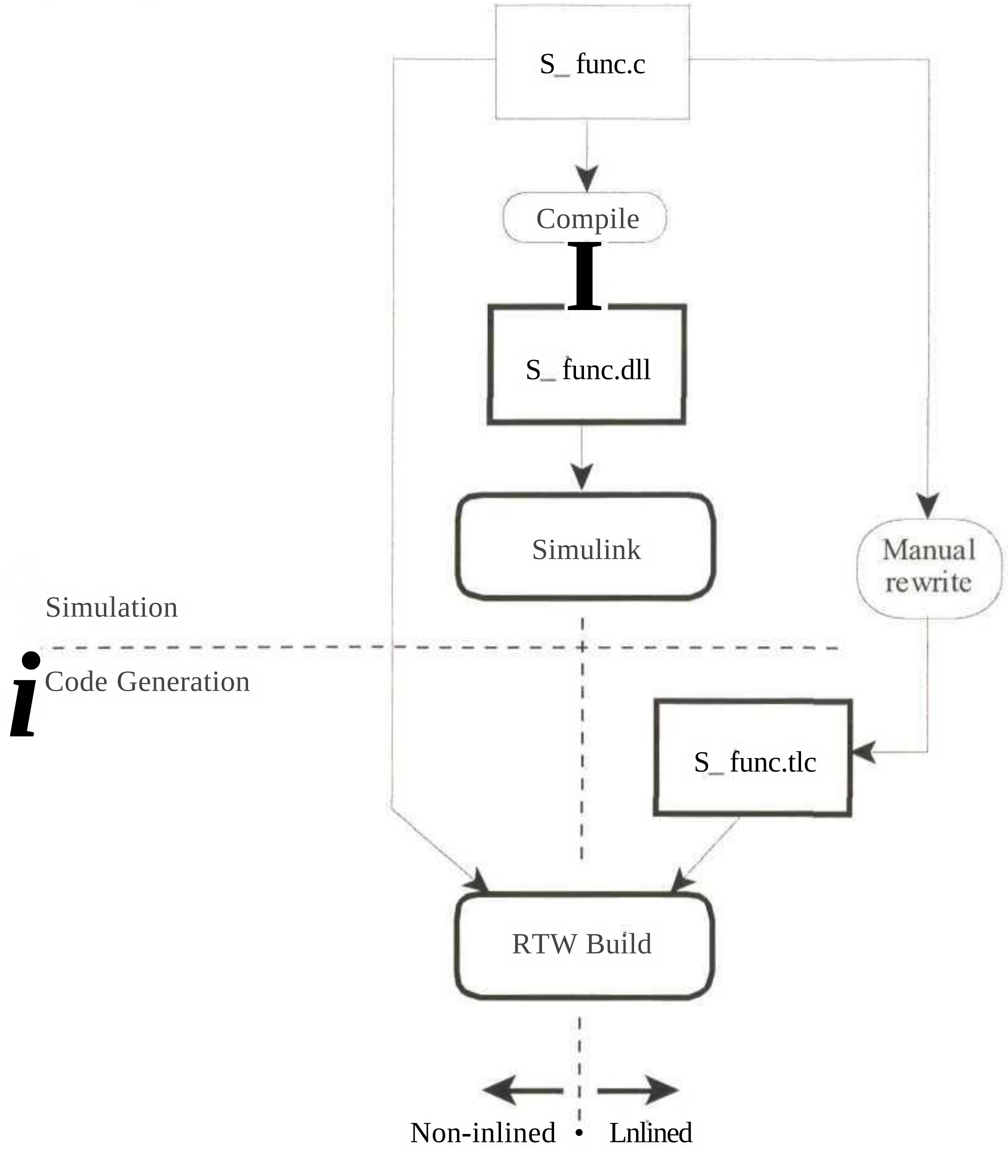


Fig. 3.14 : Including inlined and non-inlined S-functions into generated code

Fig. 3.14 shows the different steps in including inlined and non-inlined S-functions into Simulink diagrams as well as the RTW build. Non-inlined S-functions are simpler to manage but they incur a performance penalty at run-time as all of the API defined procedures have to be called. In most cases the code for an S-function will be concentrated in only one or two routines. However, all calls defined in the S-function API have to be declared even if some of them are left blank. Effectively unused *dummy calls* are created. Some simple S-functions can end up wasting more processing time in switching overheads than performing their intended function.

Inlined S-functions are merged directly into the generated code and thus do not suffer from losses due to switching between sub-routines. However, writing TLC-files requires the user to master the Math Works Target Language and would not prove viable for a once-off project.

When processing an S-function block, the TLC will, as default, first search for a corresponding TLC-file. If found it will be parsed and the code will be inlined along with the standard Simulink blocks. If no TLC-file is found on the MATLAB path, the standard C-MEX S-function source file will be added to the make file to be compiled and linked at the final stage.

3.5.6 RTW Limitations

All standard Simulink blocks can be handled automatically by the RTW build process, with two exceptions :

(1) MATLAB functions and S-function blocks that call m-files first need to be re-written as either C MEX S-functions or TLC files,

(ii) Any Simulink block which relies on absolute time to compute its outputs, e.g. Sine Wave Generator. Depending on the target hardware and the specific real-time kernel, variables used to store absolute time could wrap around and cause unpredictable results.

3.6 Target Language Compiler

The RTW as introduced above is responsible for co-ordinating the automatic transformation of a Simulink block diagram into a complete executable. TLC is a tool included with the RTW and plays an integral role in this process. It parses the intermediate text description of a Simulink model produced by the RTW and generates a number of source and header files which represent the model's functionality in terms of a programming language. By default TLC's output is ANSI C, but it can be customised to any other language.

While parsing through the intermediate RTW-file, the TLC uses a set of TLC-files to generate its output. This process is highlighted in Fig. 3.15. The TLC-files specify what goes into each of the output files and by modifying those files the output of the TLC process can be customised. The changes can range from small platform specific changes to optimise code for size, through

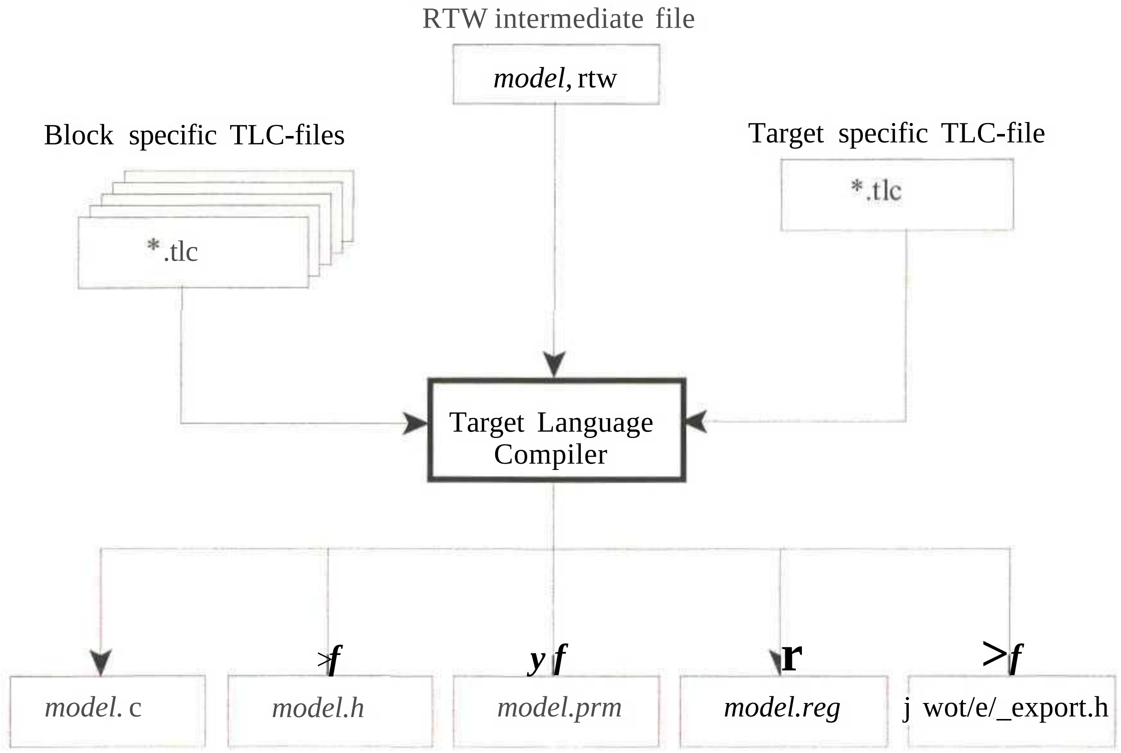


Fig. 3.15 : Target Language Compiler Process

algorithmic changes to optimise for speed, to a complete re-write of all TLC-files to generate output in a different programming language. A RTW extension is available from MathWorks which includes a set of TLC files to generate Ada compliant code. In short the TLC can be used in the following cases :

(i) Changes to the code generated for a particular Simulink block,

(ii) Mining S-functions into the generated code,

(iii) Modify some global aspects of the generated code.

(iv) A complete re-make of the way TLC produces code to target a different programming language.

Assuming that standard ANSI C code is generated, the TLC will generate the following set of files starting with the *model.rtw* intermediate file :

(i) *Model.c* - C file containing all the generated source code.

(ii) *Model.h* - Header file containing all structure definitions needed by the generated code,

(iii) *Model.prm* - Include file containing global data declarations and all default parameter values,

(iv) *Modelseg* - Include file containing all registration functions and initialisation routines.

(v) *Model_export.h* - Header file containing all definitions needed to interface and link the TLC generated code to the Real-Time Kernel.

A complete description of the MathWorks Target Language is beyond the scope of this thesis and the relevant documentation should be consulted for this purpose [MATHWORKS 6]. A brief introduction to the relevant subset of the TLC language is offered in Appendix H.

3.7 Conclusion

This Chapter introduced Simulink and RTW as the foundation of the CSDE. Features of these software packages relevant to author's work were highlighted. Chapter 4 will introduce hardware components of the CSDE. Thereafter, Chapters 5, 6, 7 and 8 will describe how the MathWorks platform was modified and extended to form a complete rapid prototyping system.

CHAPTER FOUR
TARGET HARDWARE PLATFORM

4.1 Introduction

The previous Chapter introduced the Math Works package which forms the software foundation of the CSDE. This Chapter will continue by presenting the hardware targeted by the CSDE. The hardware platform consists of a commercial DSP card, the PC32 from Innovative Integration, as well as a custom PWM / Tacho expansion card designed in-house at the Electrical Engineering Department at University of Natal [WALKER1]. The PC32 card is built around the TMS320C32 floating-point DSP from Texas Instruments (TI). All interfacing between the host PC and the PC32 is done via the PC's ISA bus. The following subsections will introduce the various components and focus on issues relevant to the author's work.

4.2 TMS320C32 DSP

The TMS320C32 processor from TI forms the heart of the PC32 controller card. The TMS320 range of DSPs is very popular due to its good price to performance ratio and has been well documented in literature [CHUNG 1, LIN 1, TRZYNADLO1]. This section does not aim to fully describe the 'C32 but briefly introduces the processor. Fig. 4.1 shows a simplified block diagram of the processor.

4.2.1 'C32 Architecture

The 'C32 employs a modified version of the Harvard architecture [LIN1]. In a strict Harvard architecture, the program and data memories are separated, thus permitting simultaneous fetching of both instruction codes and operands. This setup coupled with pipelining allows the execution of most instructions in a single processor cycle. The TMS320 family of processors modifies this architecture to allow transfers between the two memory areas. Thereby the flexibility is increased while the architecture's inherent processing power is maintained.

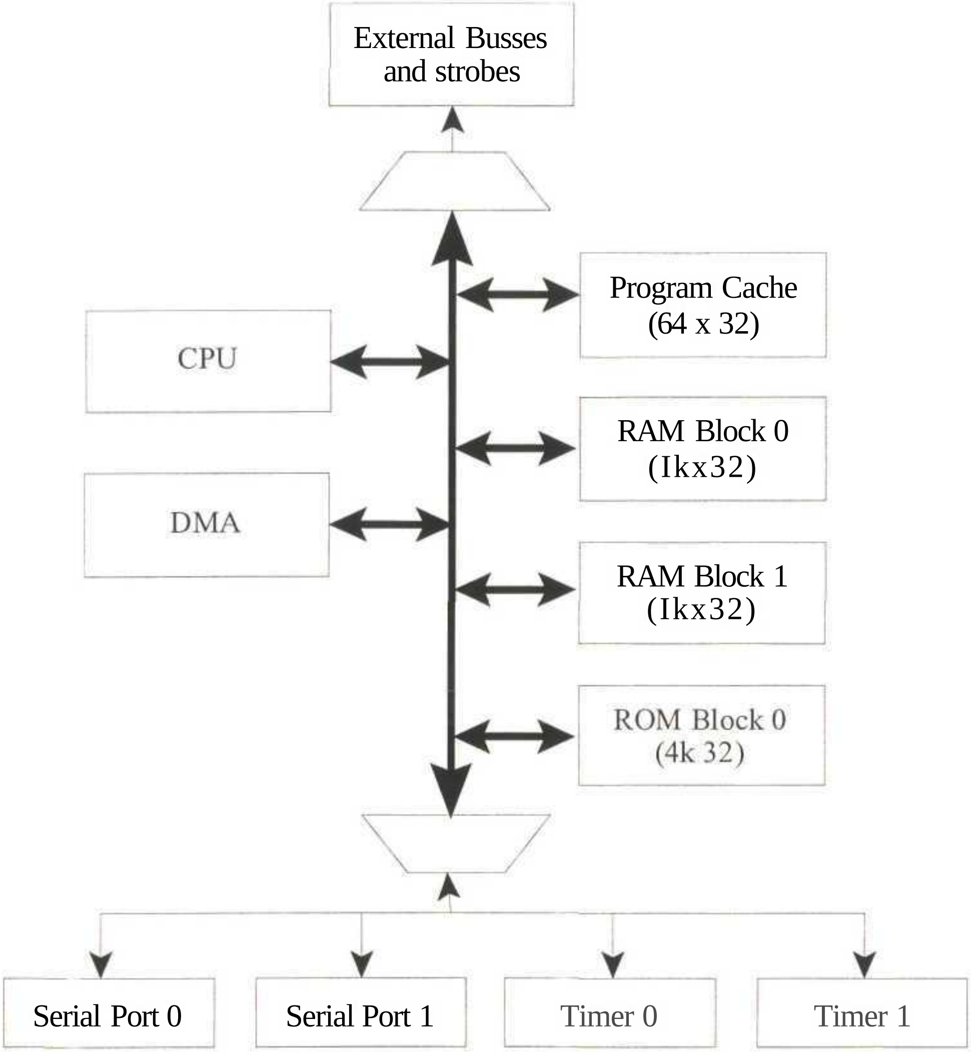


Fig. 4.1 : TMS320C32 Block Diagram

4.2.2 Cache and Pipelining

The 'C32 processor's cache can store up to 64 instructions [TEXASINSTR1]. The cache maintains a list of the most recently accessed instructions. Before a new instruction is fetched from external memory the local list is consulted and the cache copy is used if possible. Only the instruction codes are cached, data memory accesses bypass the cache algorithm. Small, time critical sections of code can benefit in performance due to the cache, but the code timing becomes less deterministic. For applications relying on code timing the cache can be disabled.

A four level pipeline is employed on the 'C32 to maximise the processor core usage [LIN1]. A basic instruction goes through four levels during processing [TEXASINSTR1] :

(i) Fetch Unit (F) - This unit fetches the instruction words from program memory via cache (if enabled).

(ii) Decode Unit (D) - The instruction is decoded and pre-processed,

(iii) Read Unit (R) - If required operands are read from external data memory,

(iv) Execute Unit (E) - The required action is performed, registers updated and if necessary results of previous operations are written to memory.

Each level requires a processor cycle to complete and each instruction can only be at one level at the time. Pipelining allows four consecutive instructions to occupy all the core levels. Table 4.1 demonstrates the pipeline operation graphically for fictitious instructions a, b, c and d. Under perfect overlap conditions the 'C32 can be processing up to four instructions in parallel, and each instruction is at a different stage of its execution.

Cycle \ Unit	F	D	R	E
m-3	a	...	...	...
m-2	b	a	...	...
m-1	c	b	a	...
m	d	c	b	a
m+1	...	d	c	b
m+2	...	...	d	c
m+3	...	...	...	d

Table 4.1: Pipeline Operation for Instructions (a,b,c,d) under Perfect Overlap

4.2.3 Memory Organisation

The 'C32 uses a 32-bit wide data bus and a 24-bit wide address bus. The total addressable memory range therefore is 2^{24} Words (16 Mwords). Some of this memory space is reserved for on-chip peripherals, ROM boot loader and interrupt vectors [TEXASINSTR2]. The remaining address ranges are available for mapping of external memory modules or peripherals. Three strobe signals control access to various areas of memory :

(i) STRB 0 and STRB 1 control the program and data memory regions.

(ii) IOSTROBE controls memory addresses used for mapping external peripherals.

Each strobe can be programmed to have a wait state from 0 to 7. This flexibility in wait state allocation allows for fast program memory to be intermixed with slower data memory chips and also allows a separate setting for the external peripherals. Section 4.4.1 outlines the memory usage particular to the PC32 card.

Internally, all memory accesses are treated as 32-bit, but externally each strobe region can be either 32, 16 or 8 bits wide. The on-chip programmable memory interface of the 'C32 resolves the data width conflicts. There is a trade off in access speed, however. For example, to access a word from a 16-bit wide memory, the processor will perform two consecutive 16-bit reads before using the 32-bit result. Accesses to 8-bit wide devices require 4 bus cycles.

4.2.4 Timers

The 'C32 has two internal programmable 32-bit timer modules. Each module can operate as either a timer or an event counter and can be clocked either by internal or external signals

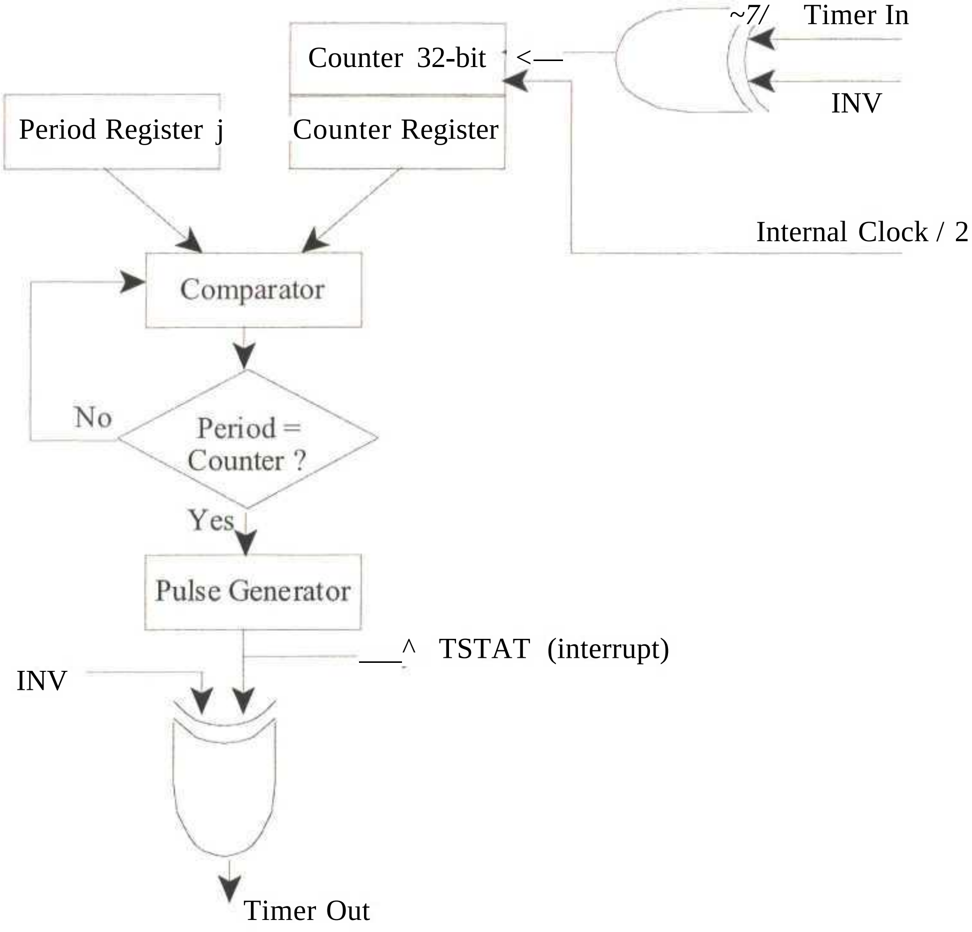


Fig. 4.2 : 'C32 Timer Module Block Diagram

[TEXSASINSTR1]. Fig. 4.2 shows a simplified block diagram of a 'C32 timer module. Each module is controlled by three memory mapped registers, detailed in Appendix B :

(i) Global Control Register - determines the operating mode of the timer, monitors its status and controls the function of the timer's I/O pin (TCLK).
(ii) Period Register - specifies the timers signalling frequency,
(iii) Counter Register - contains the current value of the incrementing counter.

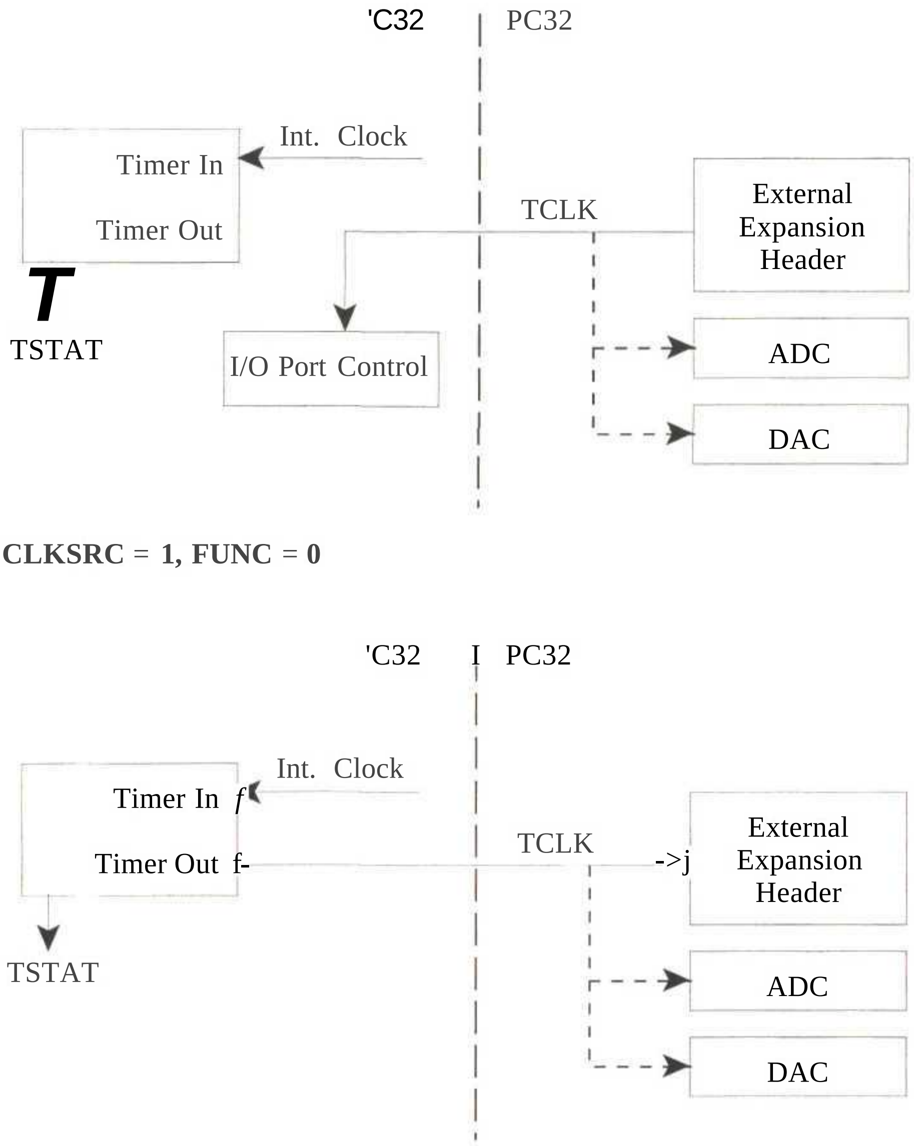
Two flags in the control register specify the timer mode :

(i) FUNC (bit 0) - controls the function of TCLK. If FUNC = 0, TCLK is configured as a general purpose I/O pin. If FUNC = 1, TCLK is used as the timer pin.
(ii) CLKSRC (bit 9) - specifies the source of the timer input. With CLKSRC = 1, the internal chip clock is used at half its frequency. If CLKSRC = 0, an external signal on the TCLK pin will drive the counter.

On the PC32 card, introduced in section 4.4, the two signals from timers 0 and 1 can be used to trigger the ADC and DAC peripherals. In a control system it is sometimes important to synchronise I/O sampling to a fixed common clock or to an asynchronous external event as will be shown in Chapter 9. Using FUNC and CLKSRC flags a total of four different timer modes can be set up as demonstrated in Table 4.2. Setups (a) and (b) are of particular interest to the CSDE and are further explained graphically in Fig. 4.3. In both modes the timer input is independent of the TCLK pin and the timer output can be used to generate internal interrupts via TSTAT. In mode (a) the TCLK line is isolated from the timer and can be used to patch trigger

No.	FUNC	CLKSRC	Description
a	0	1	TCLK is a general I/O pin, isolated from the timer. Internal clock drives the timer.
b	1	1	TCLK is the timer output pin. Internal clock drives the timer.
c	0	0	TCLK is a general I/O pin, and the timer is triggered by any activity on the pin, internal clock is ignored.
d	1	0	TCLK is the timer input pin, internal clock is ignored

Table 4.2 : 'C32 Timer Module Modes



(a) CLKSRC = 1, FUNC = 0

(b) CLKSRC = 1, FUNC = 1

Fig. 4.3 : Timer Modes (a) and (b)

signals from the PC32 expansion header to the peripherals. Thus, in this mode the I/O sampling can take place asynchronously to the internal 'C32 clock. In mode (b) the TCLK signal is derived from the timer and the I/O sampling will be synchronised to the internal clock.

```
   *(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
```

G.1.6Source Listing of Upload.tlc

```
%%
%%
%% Abstract:
%%      A TLC file for upload channels.  Implements one channel at a time
%% Author:
%%      Adam Stylo
%% Date:
%%      98/11/03
%%

%implements "Upload" "C"

%include "iilib.tlc"

%function Blocklnstance Setup(block, system) void

  %if EXISTS("UpldSeen")
    %assign ::UpldSeen = ::UpldSeen + 1
  %else
    %assign ::UpldSeen = 1
      %openfile buffer
      extern QUEUE queue[] ;
      extern unsigned int buffer_size[];
      extern unsigned int Down_Sample[] ;
      extern int queue_error;
      extern int NumQueues;
      extern LogData;
      extern channel_map[];
      %closefile buffer
      %<LibCacheDefine(buffer)>
  %endif

%openfile buffer

#ifdef UPLD_YES
 buffer_size[%<UpldSeen>-1] = (unsigned int)%<LibBlockParameter(PI,"",
                                                                    "",0)>;
 Down_Sample[%<UpldSeen>-l] = (unsigned int)%<LibBlockParameter(P3,"",
                                                                    "",0)>;
 NumQueues = %<UpldSeen>;

 if (!queue_init(&queue[%<UpldSeen>-l], (2*buffer_size[%<UpldSeen>-l])))
 {
   #ifdef IO_ENABLE
     printf("Memory allocation error in %<Name>\n");
   #endif
   gueue_error = TRUE;
   LogData = FALSE;
 }
   else
 {
   ttifdef IO_ENABLE
         printf("%d memory words allocated for %<Name>\n",
                                          2*buffer_size[%<UpldSeen>-l]);
   #endif

   LogData = TRUE;
  }

#endif

%closefile buffer

%<LibMdlStartCustomCode(buffer, "trailer")>

%openfile buffer

free(queue[%<UpldSeen>-1].base);

ftifdef IOJENABLE
  printf("Memory freed for %<Name> (%<ParamSettings.FunctionName>)\n");
#endif

%closefile buffer

%<LibMdlTerminateCustomCode(buffer, "trailer")>

%openfile buffer

/* %<Type> Block: %<Name> (%<ParamSettings.FunctionName>) */

if (LogData)
{
 Down_Sample[%<UpldSeen>-1]
 if (Down_Sample[%<UpldSeen>-l]==0)  /*capture this sample*/
 {
  channel_map[%<UpldSeen>-l]=%<LibBlockParameter(P2,"","",0)>;
  •((volatile float*)enqueue_ptr(&queue[%<UpldSeen>-l])) =
                                       %<LibBlockInputSignal(0, "", "", 0)
  Down_Sample[%<UpldSeen>-l] = (unsigned int)%<LibBlockParameter(P3,"",
                                                                  "",0)>;
 }

%closefile buffer

%<LibSystemOutputCustomCode (system, buffer, "trailer")>

%endfunction
```

Appendix H

H.I Target Language Compiler

The TLC is a tool included with the RTW. It parses the intermediate files generated by the RTW build process as well as the necessary set of TLC files to generate source code for the corresponding Simulink model [MATHW0RKS6]. This appendix provides a brief introduction to writing TLC files.

H.2 TLC Directives

All TLC files consist of series of statements in the following form :

```
      %keyword [argumentl, argument2,. . . . .]
```

Where keyword represents one of the TLC directives listed in Fig. H.I and arguments define an expression or parameter. The line below illustrates the use of the directive assign to change the value of the variable myjnum :

```
      %assign my_num = your_num + 1
```

Directive Type	General Syntax
comment	**%% *comment* text** /% comment text %/
TLC expression	%<expression>
conditional inclusion	**%if *expression*** **%else** **%elseif expression** **%endif**
macro definition	**%define identifier argument-list** **%undef identifier**

Fig. H.I : TLC Directives

Directive Type	General Syntax
! multiple inclusion	%foreach *variable* = *expression* %break %continue %endforeach
output file control	%openfile *identifier* %closefile %flushfile identifier %selectfile *identifier*
debug statements	%error *error-text* %warning *warning-text* %trace *trace-text* %exit *exit-text*
identifier definition	%assign [::]*variable* = expression
TLC functions	%function identifier (optional-arguments) [Output|void] %return %endfunction

Fig. H.I : TLC Directives (continued)

H.2.1 Comments and Line Parsing

All lines contained between the /% and %/ directives are treated as comments and omitted during the TLC parse. The %% directive specifies a line based comment - all characters between the directive and the end of the line become a comment.

Non-directive lines outside of comment directives are copied verbatim to the output buffer. Non-directive lines are lines which do not have % as their first non-blank character.

Long TLC statements can be broken down to span a number of lines. To indicate line spanning either the C language \ character or MATLAB ... sequence can be used. This mechanism allows code to be kept more readable.

H.2.2 Expressions

Statements contained within a TLC expression can include a mixture of variables, constants, logic and arithmetic constructs. During the parse the TLC evaluates the complete expression and substitutes it with the result. Expressions appearing on directive lines do not need to be delimited by the %< > directives.

H.2.3 Conditional Inclusions

The %if directive allows a conditional inclusion of TLC code sections based on the value of an expression. If the expression evaluates to zero the statements between the corresponding %if and %elseif or %endif directives are omitted during the parse. Any other result of the expression is considered as true and the statements are included.

The %elseif directive allows the nesting of a number of conditional inclusions. If a %elseif directive appears without a preceding %if it does not cause an error and is simply treated the same as a %if.

H.2.4 Multiple Inclusions

The %foreach directive allows multiple inclusion loops to be generated. The loop variable is incremented from 0 to the value of the expression minus 1. For example :

```
%foreach x = max_loop
```

could be equated to the following ANSI C statement:

```
for(x = 0; x < max_loop; x++)
```

Within the loop the value of the loop variable can be used in expressions. To exit a loop before the loop variable reaches its maximum the **%break** directive can be used. The directive **%continue** forces the next iteration of the loop.

H.2.5 File Output

The **%openfile** directive allows the channelling of the non-directive lines' output to either a local buffer or a external file. A number of buffers and files can be open simultaneously and access to them can be controlled using the **%selectfile** directive. The **%closefile** directive closes the corresponding buffer or file and if the closed file was currently selected, the selection is moved to the last previously selected file.

H.2.6 Debug Messages

The various debug directives allow text to be printed to the MATLAB command window during the TLC parse. This functionality is especially useful during the debugging stages. Various levels of message priority can be specified by using the different debug directives. TLC command line option -v allows the blocking of selected message levels. The debug directive **%exit** always causes its text arguments to be displayed and stops the TLC process.

H.2.7 Macro Definitions

To simplify complicated references macros can be defined. The macros are automatically expanded to their full content during the TLC parse. The **%define** directive allows definition of macros and is similar in use to the ANSI C **#define** compiler directive. The **%undef** directive removes the previously defined macro.

H.2.8 Identifier Definitions

Variable identifiers can be defined and their data type is automatically determined from the initial assignment. The **%assign** directive is used to introduce a new variable aor to modify the contents of an existing one. All defined variables are only visible within the scope of their function. To access variables globally the :: operator is used before the variable name.

H.2.9 TLC Functions

The TLC functions are recursive and have **their** own variable scopes. No output is produced **by the functions** of type **void unless they use the %openfile** directives explicitly. Functions defined as **Output** automatically **channel their** output to the currently selected file or buffer.

Appendix I

I.I External Mode Communication

This appendix list source code for the *ext_PC32.c* file which implements the external mode comms under Simulink on the host side. When compiled it produces *ext_PC32.dll.*

1.1.1 Source Listing of ext_PC32.c

```
/*
External comms .dll for use with Simulink and Innovative Integration PC32
 */
#include <stdlib.h>
#include <math.h>
#include "mex.h"
#include "windows.h"
#include "target.h"
#include "cardinfo.h"

CARDINFO* dsp;

volatile unsigned long* dpram;      /* 4 bytes long = 32bit */

int starting=O;

int handshake (int opt)
{
      int count=0, got_it=0;

      dpram[0] = (OxAAAO + opt);   /* Send the sync + option to target */

      target_interrupt (0) ,-
      while((count<50) && !((dpram[0] & OxFFFO)==0xABC0))
            /* wait for acknowledge */
      {
         count++;
         Sleep(100);
      }
      if (count<50)
      {
        got_it = dpram [0] & OxOOOF;
      }
      return got_it;
}
/* Function: mdlCommlnitiate
=====================================================
 * Abstract:
 *    Inititate communication with the host and pass down initial parameter
 *    values.
 */
int mdlCommlnitiate(
    int          block_param_count, /*total number of block parameters*/
    double          *block_params,      /•length = block_param_count       */
    const  char      *model_name,       /*model name                       */
    const  uint32_T  model_checksum[] , /*128 bit model checksum           */
    int              nrhs,              /*number of additional arguments   */
    const mxArray   *prhs[] ,           /* additional arguments            */
    char             *error_message     /* error message for SIMULINK      */
)
{

   int xxx=0;

   mexPrintf("\n\nExternal mode initiation for model :  %s\n",model_name);

   if  (target_open(0))
   {
      printf("Target Open. \n");
      dsp = target_cardinfo(0);
      dpram =  (volatile unsigned  long*)dsp->DualPort.PhysAddr;

      dpram [1] =model_checksum [0] ,-
      dpram[2]  = model_checksum[1];
      dpram[3]  = model_checksum[2] ;

      dpram[4]  = model_checksum[3];

      mexPrintf("Verifying checksums  ....  ") ;

      if  (handshake (4)  == 4)
      {
        mexPrintf("OK!\n");
        XXX = Θ;
        starting = 1;
      }
      else
      {
        mexPrintf("FAILED!\n");
        XXX = -1;
        mexPrintf("Checksum error.  Recompile/Reload Diagram!\n");
      }
    }

    return xxx;
}  /* end mdlCommlnitiate */

/* Function: mdlGetScopelnputs
=================================================
 * Abstract:
 *    Upload scope inputs from target.
 */
int  mdlGetScopelnputs(
    int            scope_input_cnt,    /* Number of scope inputs      */
    double         *scope_inputs,      /* Pointer to the scope inputs */
    double         *t,                 /* Pointer to time             */
    const  char    *model_name,        /* the model name              */
    const uint32_T model_checksum[],   /* 128 bit model checksum      */
    int            nrhs,               /* # of additional arguments   */
    const  mxArray  *prhs[],           /* additional arguments        */
    char            *error_message     /* error message for SIMULINK  */
)
{
  /* Not supported yet under Simulink 2.2 - for future use.*/
  return 0;  /* no errors */
}

/* Function: mdlGetBlockOutputs
=================================================
 * Abstract:
 *    Upload block outputs from target.
 */

int mdlGetBlockOutputs(
    int             block_output_cnt, /* Number of block outputs    */
    double          *block_outputs,   /* Pointer to block outputs   */
    double          *t,               /* Pointer to time            */
    const char      *model_name,      /* the model name             */
    const uint32_T model_checksum[] , /* 128 bit model checksum     */
    int             nrhs,             /* # of additional arguments */
    const mxArray   *prhs[],          /* additional arguments       */
    char            *error_message    /* error message for SIMULINK*/
)
{
  /* Not supported yet under Simulink 2.2 - for future use.*/
  return 0; /* no errors */
}

/* Function: mdlSetParameters
==================================================
 * Abstract:
 *    Download parameter changes to the target.
 */

int mdlSetParameters(
     int             block_param_count,  /'* number of block parameters •/
     double          •block_params,      /* length = block_param_count */
     int             num_changed,        /'* # of changed parameters    */
     int             •elems,             /'* indices of changed params  */
     const char      •model_name,        /'* the model name             */
     const uint32_T model_checksum[],    /+ 128 bit model checksum     */
     int             nrhs,               /'• # of additional arguments  */
     const mxArray   *prhs[],            /f* additional arguments       */
     char            •error_message      /'* error message for SIMULINK */
)
{
      int  i,xxx=0,count=0,temp;
      union {
            float f;
            unsigned long u;
            } bl_param;

  mexPrintf("\n");

  for (i=0; i<num_changed; i++)
  {
      dpram[l] = elems[i];
      bl_param.f = block_params[elems[i] ];

      dpram[2] = bl_param.u;

      if (handshake(2) == 2)

      } mexPrintf("Changed parameter P[%d] = %g\n",elems[i], bl_param.f);

      else
        mexPrintf("Handshake Failed\n") ;
        xxx=1,•
      }
  }

  if (!xxx & starting)
      /•First time we downloaded params, need to tell target*/
  {
      mexPrintf("Starting execution....");

      if(handshake(1) == 1)
      {
        mexPrintf("OK!\n");
        XXX=0;
        starting=O;

        mexPrintf("Output to Terminal is %s.\n",
                                            (dpram[3] & 1) ? "ON" : "OFF");

        mexPrintf("Timer 0 is %s used for base sample time.\n",
                                            (dpram[3] & 2) ? "" : "NOT");

        mexPrintf("Data Uploads are %s.\n",
      }                                     (dpram[3] & 4) ? "ON" : "OFF");

      else
      {
      } mexPrintf("Failed!\n"); xxx=l;}

  return xxx;
}

/* Function: mdlCommTerminate
====================================================
 * Abstract:
 *    Terminate communication with the host.
 */

int mdlCommTerminate (
    const char     *model_name,      /* the model name              */
    const uint32_T model_checksum[], /* 128 bit model checksum      */
    int            nrhs,             /* # of additional arguments   */
    const mxArray  *prhs[],          /* additional arguments        */
    char           *error_message    /* error message for SIMULINK  */
)
{
   int xxx=0;

   mexPrintf("Terminating Comms .... ") ;

   if (handshake(3) == 3)
   {
      mexPrintf("DONE!\n");
      target_close(0);
      xxx=0;
   }
   else
   {
      mexPrintf("FAILED!\n");
      XXX=-1;
      mexPrintf("Target program did not respond!");
   }

return xxx;
}

#include "ext_main.c"  /* MEX glue */

/* [EOF] ext_PC32.C */
```

Appendix J

J.I Display Utility

This appendix list the source code for the Display utility. The utility was generated using the Microsoft Visual Studio and some of the code is automatically generated, thus only relevant excerpts are listed to conserve space.

J.I.I Source for DisplayChildFrm.cpp

```
// DisplayChildFrm.cpp : implementation file
//

#include "stdafx.h"
#include "Display.h"
#include "DisplayChildFrm.h"
#include "ChildPropDlg.h"
#include "math.h"
ttifdef _DEBUG
  #define new DEBUG_NEW
#undef THIS_FILE
  static char THIS_FILE[] =___FILE__;
#endif

I III 1111 III 11 III III III 11 III I III III III I III 1111IllI III 111 III 1111111111 III I

II CDisplayChildFrm

IMPLEMENTJ3YNCREATE(CDisplayChildFrm, CFrameWnd)

extern CDisplayApp theApp;

CDisplayChildFrm::CDisplayChildFrm()
{
      Create(NULL,    " " ) ;
      num_samples=200;
      y_span=10 0;
      trigg_level=0;
      trigg_reset=FALSE;
      use_trigger=FALSE;
      for (UINT i=0; i<1000; I++)
            samples [i] .x=i;
}

CDisplayChildFrm::-CDisplayChildFrm()
{
}

BEGIN_MESSAGE_MAP(CDisplayChildFrm, CFrameWnd)
      //{{AFX_MSG_MAP(CDisplayChildFrm)
      ON_WM_PAINT()
      ON_WM_SIZE()
      ON_WM_CLOSE()
      ON_WM_LBUTTONDBLCLK()
      ON_WM_SHOWWINDOW()
      //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

IIIIIIIIIIIII IIIIIIIIIIII III IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII II111III
II CDisplayChildFrm message handlers

void CDisplayChildFrm::OnPaint()
{
      CPaintDC dc(this); // device context for painting
      // draw zero line
      dc.MoveTo(0, size.y/2);
      dc.LineTo(size.x, size.y/2);

      dc.Polyline(samples, num_samples);

}

void CDisplayChildFrm::OnSize(UINT nType, int ex, int cy)
{
      CFrameWnd::OnSize(nType, ex, cy) ;
      if (nType!=SIZE_MINIMIZED)
      {
            size.x=cx;
            size.y=cy;
            if (size.x<num_samples) num_samples = size.x;
              for (UINT i=0; i<=num_samples; I++)
              {
                samples[i].x=(i*size.x)/num_samples;
      }       }

}

BOOL CDisplayChildFrm::PreCreateWindow(CREATESTRUCT &cs)
{
      static UINT my_x, my_y;

      cs.cx = 212;
      cs.cy = 10 0;

      return CFrameWnd::PreCreateWindow(cs) ;
}

void CDisplayChildFrm::OnClose()
{
      theApp.active_child[my_num]=FALSE;

      CFrameWnd::OnClose();

}

void CDisplayChildFrm::OnLButtonDblClk(UINT nFlags, CPoint point)
{
      RECT temp_rect;

      ChildPropDlg temp_dlg;

      temp_dlg.DoModal(my_num, point);

      if (size.x<num_samples) //check if we need to expand the window
      {
        size.x=num_samples;
```

```
          GetWindowRect(&temp_rect);
          SetWindowPos(NULL,0,0, size.x, temp_rect.bottom-temp_rect.top,
                                 SWP_NOMOVE | SWP_NOZORDER);
        }

        for (UINT i = 0; i<=num_samples,- I + + )
              samples[i].x=(i*size.x)/num_samples;

        RedrawWindow();

        CFrameWnd::OnLButtonDblClk(nFlags,  point);

}
```

J.I.2 Source for DispIayDlg.cpp

```
// DispIayDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Display.h"
#include "DisplayDlg.h"
#include "DisplayChildFrm.h"

tfifdef _DEBUG
   #define new DEBUG_NEW
#undef THIS_FILE
   static char THIS_FILE[] =___FILE__;
ttendif

1111111111111111111111111111111111111111111111111111111111111
II CAboutDlg dialog used for App About

class CAboutDlg : public Cdialog
{
public:
      CAboutDlg();

// Dialog Data
      //{{AFX_DATA(CAboutDlg)
      enum { IDD = IDD_ABOUTBOX };
      //}}AFX_DATA

      // ClassWizard generated virtual function overrides
      //{{AFXJVIRTUAL(CAboutDlg)
      protected:
      virtual void DoDataExchange(CDataExchange* pDX);
      //}}AFX_VIRTUAL
// Implementation
protected:
      //{{AFX_MSG(CAboutDlg)
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
      //{{AFX_DATA_INIT(CAboutDlg)
      //}}AFX_DATA_INIT
}
```

```
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(CAboutDlg)
      //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CAboutDlg, Cdialog)
      //{{AFX_MSG_MAP(CAboutDlg)
            // No message handlers
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

II1111111II III I III III I III 111 III III I III IIllllIIIII1111II111 III I III 11111111
//CDisplayDlg dialog

CDisplayDlg::CDisplayDlg(CWnd* pParent)
      : CDialog(CDisplayDlg::IDD, pParent)
{
      //{{AFX_DATA_INIT(CDisplayDlg)
      m_num_channels = 0;
      m_target = 0;
      //}}AFX_DATA_INIT
      m_hlcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
void CDisplayDlg::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{(AFX_DATA_MAP(CDisplayDlg)
      DDX_Text(pDX, IDC_EDIT1, m_num_channels);
      DDV_MinMaxUInt(pDX, m_num_channels, 0, 10);
      DDX_Text(pDX, IDC_EDIT2, m_target);
      DDV_MinMaxUInt(pDX, m_target, 0, 5);
      //} }AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CDisplayDlg, Cdialog)
      //{{AFX_MSG_MAP(CDisplayDlg)
      ON_WM_SYSCOMMAND()
      ON_WM_PAINT()
      ON_WM_QUERYDRAGICON()
      ON_BN_CLICKED(IDOK, OnOk)
      ON_WM_CLOSE()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

III I III I III I III III II III III I III I III IIIIlllIII III I III 1111 III 11111 IIIIlll1111
//CDisplayDlg message handlers

BOOL CDisplayDlg::OnInitDialog()
{
      CDialog: :OnInitDialog () ,-

      // Add "About..." menu item to system menu.
      // IDM_ABOUTBOX must be in the system command range.
      ASSERT ( (IDM_ABOUTBOX & OxFFFO) == IDM_ABOUTBOX);
      ASSERT(IDM_ABOUTBOX < OxFOOO);

      CMenu* pSysMenu = GetSystemMenu(FALSE);
      if (pSysMenu != NULL)
      {
            CString StrAboutMenu;
            strAboutMenu.LoadString(IDS_ABOUTBOX),-
            if  (!strAboutMenu.IsEmpty())
            {
                  pSysMenu->AppendMenu(MF_SEPARATOR);
                  pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
                                                     strAboutMenu);
            }
      }

      Setlcon(m_hlcon, TRUE);            // Set big icon
      Setlcon(m_hlcon, FALSE);           // Set small icon

      return TRUE;
}

void CDisplayDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
      if ((nID & OxFFFO) == IDM_ABOUTBOX)
      {
            CAboutDlg dlgAbout;
            dlgAbout.DoModal();
      }
      else
      {
            CDialog::OnSysCommand(nID, lParam);
}     }

void CDisplayDlg::OnPaint()
{
      if (IsIconicO)
      {

            CPaintDC dc(this); // device context for painting

            SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0)

            // Center icon in client rectangle
            int cxlcon = GetSystemMetrics(SM_CXICON);
            int cylcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
            GetClientRect(&rect);
            int x = (rect.Width() - cxlcon +1) / 2;
            int y = (rect.Height() - cylcon +1) / 2;
            // Draw the icon
            dc.Drawlcon(x, y, m_hlcon);
      }
      else
      {
            CDialog::0nPaint();
}     }

HCURSOR CDisplayDlg::OnQueryDragIcon()
{
      return (HCURSOR) m_hlcon;
}
extern CDisplayApp theApp;

void CDisplayDlg::OnOk()
{
      char title [100] ;
      WINDOWPLACEMENT wpl;

      for (UINT i=0; i<m_num_channels; I++)
            if (theApp.active_child[i])
            {
               theApp.child[i]->DestroyWindow();
               theApp.active_child[i]=FALSE;
            }
if(UpdateData())
{
   for (i=0; i<m_num_channels; I++)
   {
      theApp.child[i] = new CDisplayChildFrm;
      sprintf(title,"CH #%d",i);
      theApp.child[i]->ShowWindow(SW_SHOW);
      theApp.child[i]->UpdateWindow();
      theApp.child[i]->SetWindowText(title) ;
      theApp.active_child[i]=TRUE;
      theApp.chiId[i]->my_num= i;
      theApp.queue[i].head = 0;
      theApp.queue[i].trigg = 0;
      theApp.queue[i].tail = 0;

      for (UINT j=0; j<5000; j++)
            theApp.queue [i] .sample [j] = 0;
   }
   theApp.num_children=m_num_channels;

   if(!target_open(m_target))
   {
      sprintf(title, "Unable to open Device Driver for target %d\n"
                             "Check target number setting", m_target);
      MessageBox(title, "FATAL ERROR", MB_ICONINFORMATION);
      PostQuitMessage(0) ;
   }
  // Set up the Virtual ISR
   host_interrupt_install(m_target, EnqueueData, (PVOID)m_target);
   host_interrupt_enable(m_target) ;

  // Set DPRAM addrss
   theApp.dsp = (CARDINFO*) target_cardinfo (m_target) ,-
   theApp.dpratn = (volatile int*)theApp.dsp->BusMaster.Addr;

   if (theApp.pThread==NULL) theApp.pThread = AfxBeginThread(ThreadFunc,
                                                            NULL);

  // tell target we are ready for data
   theApp.dpram [4] &= OxFFFE;

   GetWindowPlacement(&wpl);
   wpl.showCmd=SW_MINIMIZE;
   SetWindowPlacement(&wpl);
}

__inline long enqueued(UINT num)
{
      long depth = theApp.queue[num].head - theApp.queue[num].tail;
      return ((depth < 0) ? (depth + Q_SIZE) : depth);
```

```
}

__inline float dequeue(UINT num)
{
      float value = theApp.queue[num].sample[theApp.queue[num].tail++];
      if (theApp.queue[num].tail == Q_SIZE) theApp.queue[num].tail=0;
      return (value);
}

__inline void enqueue(UINT num, float value)
{
      theApp.queue[num].sample[theApp.queue[num].head++]=value;
      if (theApp.queue[num].head == Q_SIZE) theApp.queue[num].head=0;
}

UINT ThreadFunc(LPVOID pPtr)
{
  while (1)
  {                                            j++)
     for (UINT j=0; j<theApp.num_children;
     if ((theApp.active_child[j]) &&
                 (enqueued(j)>=theApp.child[j]->num_samples))
     {
      if (theApp.child[j]->use_trigger)
      {
           //looking for trigger reset
        while((enqueued(j)>=theApp.child[j]->num_samples)&&
                                 (!theApp.child[j]->trigg_reset))
        {
           if ((int)dequeue(j)<theApp.child[j]->trigg_level)
                   theApp.child[j]->trigg_reset = TRUE;
        }  //looking for new trigger
        while((enqueued(j)>theApp.child[j]->num_samples)&&
                                 (theApp.child[j]->trigg_reset))
        {
           if ((int)dequeue(j)>=theApp.child[j]->trigg_level)
           {
             theApp.child[j]->trigg_reset = FALSE;               .++)
             for (int i=0; i<theApp.child[j]->num_samples; i
             {theApp.child[j]->samples[i].y=
                   ((float)(theApp.child[j]->size.y/2) *
                        (1-(float)dequeue(j) /
                             (float)theApp.child[j]->y_span));
             }
             theApp.child[j]->RedrawWindow();
           }
        }
      }
      else //no trigger needed
      {
        for (int k=0; k<theApp.child[j]->num_samples; k++)
         theApp.child[j]->samples[k].y =
           ((float)(theApp.child[j]->size.y/2) * (1-(float)dequeue(j) /
                        (float)theApp.child[j]->y_span));
        theApp.child[j]->RedrawWindow();

     ,'
     Sleep(0);
  }
}
```

```
VOID EnqueueData (PVOID pvoid)
{
      UINT I;
      long offset;
      unsigned long buffer_size;
      int buffer_num;
      union {
            float f;
            unsigned long u;
            } plot_data;

      offset = 0;
      buffer__num = theApp. dpram [5 + offset++];
      buffer_size = theApp.dpram[5 + offset++];

      while ((buffer_num !=999) && (offset < 1000))
      {
         for (i = 0; i < buffer_size; I++)
         {
            plot_data.u = theApp.dpram [5 + offset + + ] ;
            enqueue(buffer_num, plot-data.f);
         }
         buffer_num = theApp.dpram[5 + offset++];
         buffer_size = theApp.dpram[5 + offset + + ] ;
      }

       //signal to target that we are finished reading
      theApp.dpram[4] &= OxFFFE;
}

void CDisplayDlg: :OnClose ()
{
      for (UINT i=0; i<m_num_channels; I++)
        if (theApp.active_child[i])
        {
            theApp.child[i]->DestroyWindow() ;
            theApp.active_child[i]=FALSE;
        }

      CDialog::OnClose  ();
}
```

J.1.3 Source for ChildPropDIg.cpp

```
// ChildPropDIg.cpp : implementation file
//

#include "stdafx.h"
#include "Display.h"
#include "ChildPropDlg.h"

#ifdef _DEBUG
   #define new DEBUG_NEW
#undef THIS_FILE
   static char THIS_FILE[] =___FILE__;
#endif

IllllllII IIIIlllllllllllllllIII IIIIllllllllllllllllllllllllllllllllllllllllll
II ChildPropDlg dialog
```

```
ChildPropDlg::ChildPropDlg(CWnd* pParent)
      : CDialog(ChildPropDlg::IDD, pParent)
{
      //{{AFX_DATA_INIT(ChildPropDlg)
      m_num_samples = 0;
      m_trigg_level = 0;
      m_y_span = 0;
      m_use_trigger = FALSE;
      //}}AFX-DATA_INIT
}
void ChildPropDlg::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(ChildPropDlg)
      DDX_Text(pDX, IDC_EDIT1, m_num_samples);
      DDV_MinMaxUInt(pDX, m_num_samples, 20, 1000);
      DDX_Text (pDX, IDC_EDIT4, m_trigg_level) ;
      DDV_MinMaxInt(pDX, m_trigg_level, -32000, 32000);
      DDX_Text(pDX, IDC_EDIT2, m_y_span);
      DDV_MinMaxUInt(pDX, m_y_span, 0, 32000);
      DDX_Check(pDX, IDC_CHECK1, m_use_trigger);
      //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(ChildPropDlg, Cdiaiog)
      //{{AFX_MSG_MAP(ChildPropDlg)
      ON_BN_CLICKED(IDOK, OnOk)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

11 III III I III 111111II11 III 11III III III11111111111 III III 111 III 11111 III III III
II ChildPropDlg message handlers

extern CDisplayApp theApp;

void ChildPropDlg::OnOk()
{
      if (UpdateData ())
      {
         theApp.child[m_child_num]->num_samples = m_num_samples;
         theApp.child[m_child_num]->y_span = m_y_span;
         theApp.child[m_child_num]->trigg_level = m_trigg_level;
         theApp.child[m_child_num]->use_trigger = m_use_trigger;
         EndDialog(TRUE);
}     }

int ChildPropDlg::DoModal(UINT child_num, CPoint cPt)
{
      m_child_num=child_num;
      m_y_span=theApp.child[child_num]->y_span;
      m_num_samples=theApp.child[child_num]->num_samples;
      m_trigg_level=theApp.child[child_num]->trigg_level;
      m_use_trigger=theApp.child[child_num]->use_trigger;
      return CDialog::DoModal();
}
BOOL ChildPropDlg::PreCreateWindow(CREATESTRUCT& cs)
{
      cs.x=m_cPt.x;
      cs.y=m_cPt.y,-
      return CDialog::PreCreateWindow(cs) ;
}
```

APPENDIX K

K.1 RTW Build Example

This Appendix presents a step-by-step example of the RTW build. It is shown how a simple block diagram is converted into ANSI C source code and compiled into an executable.

K.2 Block Diagram

The Simulink block diagram *Example* is shown in Fig. L. 1. It is assumed that an external trigger event is connected to the EIO interrupt of the 'C32. Each time EIO is triggered the **AD TRIGGER** block executes, and starts a conversion on the four ADC channels. The end of conversion signal from the ADC in turn triggers the Ell interrupt and the **Process** subsystem executes. The contents of the **Process** subsystem block are shown in Fig. K.I.

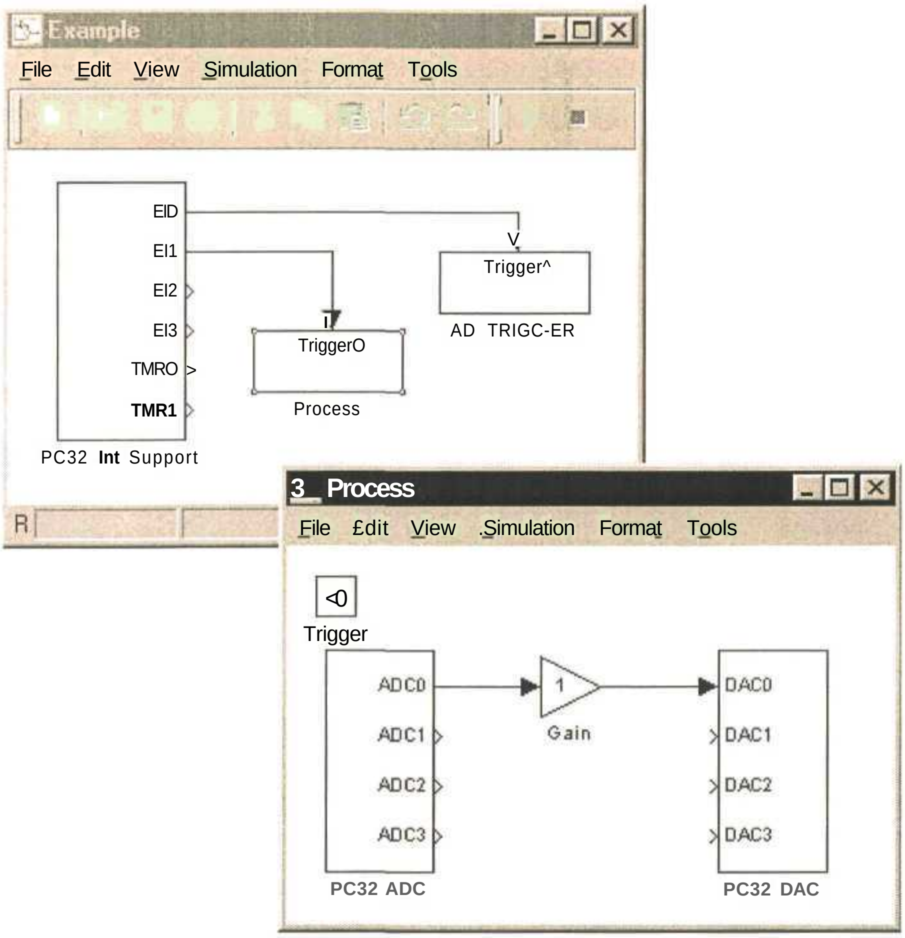


Fig. K.I : Simulink diagram used in the RTW example

The overall effect of this setup is a simple feed-through from ADC channel 0 to DAC channel 0. The sampling rate is determined by the signal driving EIO and the gain can be adjusted dynamically.

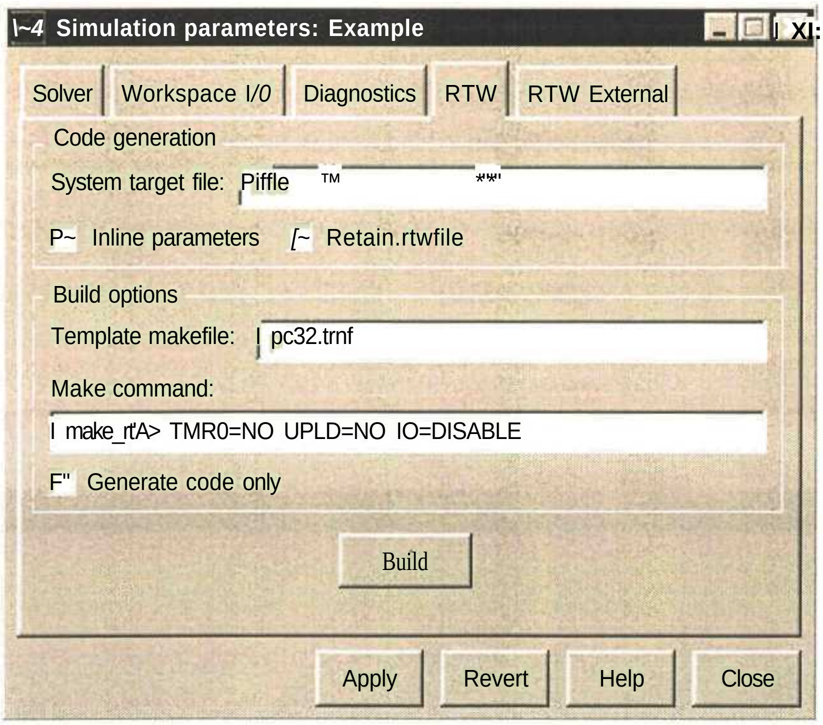
K.2.1 Setting up RTW Parameters

The simulation parameters dialog box for the *Example* diagram is shown in Fig. K.2. The code generation switches are set as follows :

(i) **TMRO** = NO - the diagram will be triggered asynchronously by and external event, thus there is no need to use timer 0 to generate a clock,

(ii) **UPLD** = NO - there are no scope upload blocks used in the diagram, thus all data buffering and uploading functions will be excluded,

(iii) IO = DISABLE - the debug printouts to the terminal emulator will be disabled.



Code generation

System target file: Piffle ™ *"*"

P~ Inline parameters [~ Retain.rtwfile

Build options

Template makefile: pc32.trnf

Make command:

```
make_rt'A> TMR0=NO UPLD=NO IO=DISABLE
```

F" Generate code only

Fig. K.2 : RTW Simulation parameters dialog

K.3 The Build Process

After initiating the RTW build by clicking on the **Build** button, the progress can be followed in the MATLAB command window. The output generated during building of the *Example* diagram is listed below (some lines are truncated due to the limited line length) :

```
»
### Starting RTW build procedure for model: Example
Using fixed-step discrete time solver for model 'Example[1].
### Invoking Target Language Compiler on Example.rtw
tic -r Example.rtw C:\MATLAB\rtw\c\ii\ii.tlc -0. -IC:\MATLAB\rtw\c\ii -Ic:
### Loading TMW TLC function libraries
### Initial pass through model to cache user defined code
Warning - No code will be generated for ISR 2 since it is not connected to any-
Warning - No code will be generated for ISR 3 since it is not connected to any-
Warning - No code will be generated for ISR 4 since it is not connected to any
Warning - No code will be generated for ISR 5 since it is not connected to any
### Creating (RealTime) source file Example.c
### Creating part 1 of registration file Example.reg
### Creating parameter file Example.prm
### Creating model header file Example.h
### Creating part 2 of registration file Example.reg
### TLC code generation complete.
### Creating project marker file: rtwproj.tmw
### Creating Example.mk from pc32.tmf
### Building Example: gmake -f Example.mk TMR0=NO UPLD=N0 IO=DISABLE
c:\fltc\cl30 c:\matlab\rtw\c\ii\ti_fpc\PC32main.c
 [pc32main.c]
c:\fltc\cl30 C:\MATLAB\rtw\c\src\rt_sim.c
 [rt_sim.c]
c:\fltc\cl30 C:\MATLAB\rtw\c\libsrc\rt_matrx.c
 [rt_matrx.c]
c:\fltc\cl30 Example.c
 [example.c]
echo PC32main.o30 rt_sim.o30 rt_matrx.o30 Example.o30  > Example.lin
echo C:\MATLAB\rtw\c\ii\ti_fpc\pc32func.o30 » Example.lin
echo C:\MATLAB\rtw\c\ii\ti_fpc\iiPC32.cmd >> Example.lin
c:\fltc\lnk30 -x -a -cr -heap 0x2000 C:\MATLAB\rtw\c\ii\ti_fpc\vectors.obj
TMS320C3x/4x COFF Linker      Version 4.70
Copyright (c) 1987-1996  Texas Instruments Incorporated
echo Completed Example.out
```

```
Completed Example.out
### Downloading Example: gmake -f Example.mk download TMR0=NO UPLD=NO IO=D
C:\MATLAB\rtw\c\ii\D_Load.exe Example.out
### Successful completion of RTW build procedure for model: Example
```

Firstly, the TLC parse process generates all source code into files *Example.c,* .reg, .h and .prm. The last file generated is the customised make file *Example.mk.* These files will be discussed in more detail in the following sections. Next, it can be seen how the source code files are compiled and linked by the TI tools. The final executable, *Example.out,* is downloaded to the target after the successful completion of all preceding steps.

K.4 Header File

The generated header file, *Example*.h is listed below :

```
/*
 * Example.h
 *
 * Real-Time Workshop code generation for Simulink model "Example".
 *
 * RTW file version           : 2.11 (Aug 28, 1997)
 * RTW file generated on      : Fri Nov 12 08:34:15 1999
 * TLC version                : 1.0 (Dec 12 1997)
 * C source code generated on : Fri Nov 12 08:34:15 1999
 *
 * Relevant TLC Options:
 *   InlineParameters = 0
 *   RollThreshold = 5
 *   FileSizeThreshold = 50000
 *   CodeFormat = RealTime
 */

#ifndef _RTW_HEADER_FILE_
ttdefine _RTW_HEADER_FILE_

ttdefine assert(exp)
#include "simstruc.h"

#define CODEGENERATOR_VERSION "2.2"
```

```
#define MODEL_NAME Example
#define NMODES (0)              /* Number of block mode elements */
#define NSAMPLEJTIMES  (2)      /* Number of sample times */
#define NINPUTS (0)             /* Number of model inputs */
#define NOUTPUTS (0)            /* Number of model outputs */
#define NSTATES (0)             /* Number of states (total) */
ttdefine NDSTATES (0)           /* Number of discrete states */
ttdefine NBLOCKIO (4)           /* Number of data output port signals */
ttdefine NUM_ZC_EVENTS  (0)     /* Number of zero-crossing events */

ttifndef NCSTATES
# define NCSTATES  (0)          /* Number of continuous states */
ttelif NCSTATES != 0
# error Invalid specification of NCSTATES defined in compiler command
#endif

/* include header files for II 0/S calls */
ttinclude "periph.h"
ttinclude "dsp.h"

/*
      define addresses for AD & DA Triggering
 */
ttdefine  ADC0 (volatile  int*)  0x810000
ttdefine  ADC1 (volatile  int*)  0x810800
ttdefine  ADC2 (volatile  int*)  0x811000
ttdefine  ADC3 (volatile  int*)  0x811800

/* define addresses for control registers */
ttdefine GC_CTRL0  (volatile  int*)  0x808020
ttdefine GCCTRL1  (volatile  int*)  0x808030
/*
   redefine arguments for ISR: <Root>/AD TRIGGER
 */
ttdefine Sys_root_AD_TRIG_OutputUpdate(rtS,controlPortldx,tid)   c_int01()
/*
   redefine arguments for ISR: <Root>/Process
 */
ttdefine Sys_root_Process_OutputUpdate(rtS,controlPortldx,tid)   c_intO2 ()

tifndef      TRUE
```

```
# define TRUE (1)
#endif
#ifndef FALSE
# define FALSE (0)
#endif

/*
 * Block I/O Structure
 *
 * Note: Individual field names are derived from the signal name when present,
 * otherwise, field names are derived from the source block name with an
 * optional port number appended to the block name if the block has multiple
 * output ports. The comment to the right of structure field contains the
 * signal source block name.
 *
 */

typedef struct BlocklOtag {
  real_T sl_S_Functionl;                   /* Source: sl_S_Functionl */
  real_T s4_S_Function[4] ;                /* Source: s4_S_Function */
  realT s3_Gain;                           /* Source: s3_Gain */
  realT s2_S_Function[6] ;                 /* Source: s2_S_Function */
} BlocklO;

/*
 * Default Parameters Structure
 *
 * Note: The structure names are derived from the block name and the
 * structure fields are derived from the block parameter name.  The
 * comment to the right of the structure field contains the actual
 * contents of the dialog box entry.
 *
 */

typedef struct Parameters_tag {
  struct {                                 /* Block Type: Gain */
    real_T Gain;                           /* Dialog Entry: 1 */
  } s3_Gain;
  struct {                                 /* Block Type: S-Function */
    realJT PlSize[2];                      /* PISize */
    real_T PI;                             /* PI */
    real_T P2Size[2];                  /* P2Size */
    realJT P2;                         /* P2 */
    real_T P3Size[2];                  /* P3Size */
    real_T P3;                         /* P3 */
    real_T P4Size[2];                  /* P4Size */
    real_T P4;                         /* P4 */
    real_T P5Size[2];                  /* P5Size */
    real_T P5;                         /* P5 */
    real_T P6Size[2];                  /* P6Size */
    real_T P6[6];                      /* P6 */
  } s2_S_Function;
} Parameters;

extern real_T rtlnf;
extern real_T rtMinusInf;
extern real_T rtNaN;
extern real_T rtRealGROUND;

/*
 * System hierarchy
 *
 * <Root>: Example
 * <S1>: Example/AD TRIGGER
 * <S2>: Example/PC32 Int Support
 * <S3>: Example/Process
 * <S4>: Example/Process/PC32 ADC
 * <S5>: Example/Process/PC32 DAC
 */

#endif                                 /* _RTW_HEADER_FILE_ */
```

K.5 C Source File

The generated C source file, *Examplex,* is listed below :

```
/*
 * Example.c
 *
 * Real-Time Workshop code generation for Simulink model "Example".
 *
 * RTW file version          : 2.11 (Aug 28, 1997)
 * RTW file generated on       : Fri Nov 12 08:34:15 1999
 * TLC version                 : 1.0 (Dec 12 1997)
 * C source code generated on  : Fri Nov 12 08:34:15 1999
 *
 * TLC Options:
 *   InlineParameters = 0
 *   RollThreshold = 5
 *   FileSizeThreshold = 50000
 *   CodeFormat = RealTime
 *
 * Simulink model settings:
 *   Solver     : FixedStep
 *   StartTime  : 0 s
 *   StopTime   : 0 s
 *   FixedStep  : 0.0002 s
 */

#include <math.h>
#include <string.h>
#include "Example.h"
#include "Example.prm"

real_T rtRealGROUND = 0.0;

/* Output and update for function-call system: <Root>/AD TRIGGER */
void Sys_root_AD_TRIG_OutputUpdate(void *reserved, int_T controlPortldx,
                                                  int_T tid)
{
  /* (outputs) */
  /* S-Function Block: <S1>/S-Functionl (ADTrigger) */
  /* a write to those addresses triggers a conversion on A/D */
  *(ADC0)=0;
  *(ADCl)=0;
  *(ADC2)=0;
  *(ADC3)=0;
  /* (no update code required) */
}

/* Output and update for function-call system: <Root>/Process */
void Sys_root_Process_OutputUpdate(void 'reserved, int_T controlPortldx,
                                                  int_T tid)
```

```
{
  /* (outputs) */
  /* S-Function Block: <S4>/S-Function (pc32_ad) */
  /* read in the corrected values from A/D and scale to +-10 */
  {
    rtB.s4_S_Function[0]=read_adc(BASEBOARD, 0)/(3276.7);
    rtB.s4_S_Function[l]=read_adc(BASEBOARD, 1)/(3276.7);
    rtB.s4_S_Function[2]=read_adc(BASEBOARD, 2)/(3276.7);
    rtB.s4_S_Function[3]=read_adc(BASEBOARD, 3)/(3276.7);
  }
  /* Gain Block: <S3>/Gain */
  rtB.s3_Gain = rtB.s4_S_Function[0] * rtP.s3_Gain.Gain;

  /* S-Function Block: <S5>/S-Function (pc32_da) */
  /* Start an output conversion*/
  {
    write_(BASEBOARD, 0, rtB.s3_Gain*(3276.7));
    convert_dac(BASEBOARD, 0);
    write_(BASEBOARD, 1, rtRealGROUND*(3276.7));
    convert_dac(BASEBOARD, 1);
    write_(BASEBOARD, 2, rtRealGROUND*(3276.7));
    convert_dac(BASEBOARD, 2);
    write_(BASEBOARD, 3, rtRealGROUND*(3276.7));
    convert_dac(BASEBOARD, 3);
  }
  /* (no update code required) */
}

/* Initialize the model */
void MdlStart(void)
{
  /* User specific code ( function declaration) */

#ifdef IO_ENABLE
  printf("Connecting Interrupts\n");
#endif

  /* Make  edge triggered only if required */
  if ((int)rtP.s2_S_Function.Pl == 1)
  {asm ("   OR 4000h,ST");
#ifdef IO_ENABLE
   printfC'EIO - EI3 Set to edge triggered\n");
#endif
  }
  else
  {asm (•   ANDN 4000h,ST");
#ifdef IO_ENABLE
   printfC'EIO - EI3 Set to level triggered\n");
#endif
  }
  /* End of user specific code (MdlStart function declaration) */

  /* connect ISR system: <Root>/AD TRIGGER */
  if (rtP.s2_S_Function.P6[0]==9)
  {
      /* check that timerO int is unused before assigning new vector */
#ifndef TMR0_YES
    install_int_vector(c_int01, (int)rtP.s2_S_Function.P6 [0]);
    enable_interrupt((int)rtP.s2_S_Function.P6[0]-1);
ttifdef IO_ENABLE
  printf("Vectior installed for INT #%d.\n",(int)rtP.s2_S_Function.P6[0]);
#endif
#endif
  }
  else {
    install_int_vector(c_intOl,(int)rtP.s2_S_Function.P6[0]);
    enable_interrupt((int)rtP.s2_S_Function.P6[0]-1);
#ifdef IO_ENABLE
  printf("Vectior installed for INT #%d.\n", (int)rtP.s2_S_Function.P6 [0]);
#endif
  }
  /* connect ISR system: <Root>/Process */
  if (rtP.s2_S_Function.P6[l]==9)
  {
      /* check that timerO int is unused before assigning new vector */
#ifndef TMR0_YES
    install_int_vector(c_int02, (int)rtP.s2_S_Function.P6 [1]);
    enable_interrupt((int)rtP.s2_S_Function.P6[1]-1);
#ifdef IO_ENABLE
  printf("Vectior installed for INT #%d.\n",(int)rtP.s2_S_Function.P6[1]);
#endif
```

```
#endif
  }
  else {
    installintvector(c_intO2, (int)rtP.s2_S_Function.P6 [1]);
    enableinterrupt((int)rtP.s2_S_Function.P6[1]-1);
ttifdef IO_ENABLE
  printf("Vectior installed for INT #%d.\n",(int)rtP.s2_S_Function.P6[1]);
ttendif
  }
#ifndef TMR0_YES
 /*Only change Timer 0 settings if it isn't used for base sampling rate */
  timer(0, (int)rtP.s2_S_Function.P4);
#endif
  timer(1, (int)rtP.s2_S_Function.P5);
  if ((int)rtP.s2_S_Function.P2 == 2)
  {*GC_CTRL0 = 0x6c3;
#ifdef IO_ENABLE
    printf("TCLKO driven by Timer 0.\n");
ttendif
  }
  else
  {•GCCTRLO = 0x6cO;
ttifdef IO_ENABLE
    printf("TCLKO driven externaly.\n");
ttendif
  }
  if ((int)rtP.s2_S_Function.P3 == 2)
  {•GCCTRL1 = 0x6c3;
ttifdef IOENABLE
    printf("TCLK1 driven by Timer l.\n");
ttendif
  >
  else
  {*GC_CTRL1 = 0x6c0;
ttifdef IOENABLE
    printf("TCLK1 driven externaly.\n");
ttendif
  }
ttifdef IO_ENABLE
  printf("Interrupts Connected & Enabled\n");
ttendif
```

```
  /* End of user specific code (MdlStart function exit) */
}

/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* (no output code required) */
}

/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}

/* Terminate function */
void MdlTerminate(void)
{
  /* User specific code (MdlTerminate function exit) */
  /*   disconnect ISR system: <Root>/AD TRIGGER  */
  if (rtP.s2_S_Function.P6[0]==9)
  {
      /* only disconnect timerO if it was set up here  */
#ifndef TMR0_YES
    disable_interrupt((int)rtP.s2_S_Function.P6[0]     -1);
    deinstall_int_vector((int)rtP.s2_S_Function.P6[0]);
#ifdef IO_ENABLE
    printf("INT #%d disabled.\n", (int)rtP.s2_S_Function.P6 [0]);
#endif
#endif
  }
  else {
    disable_interrupt((int)rtP.s2_S_Function.P6[0]-1);
    deinstall_int_vector((int)rtP.s2_S_Function.P6[0]);
#ifdef IO_ENABLE
    printf("INT #%d disabled.\n",(int)rtP.s2_S_Function.P6[0]);
#endif
  }
  /*   disconnect ISR system: <Root>/Process  */
  if (rtP.s2_S_Function.P6[1]==9)
  {
      /* only disconnect timerO if it was set up here  */
#ifndef TMR0_YES
    disable_interrupt((int)rtP.s2_S_Function.P6[1]-1);
    deinstallintvector((int)rtP.s2_S_Function.P6   [1]);
ttifdef IOENABLE
    printff'INT #%d disabled.\n», (int)rtP.s2_S_Function.P6 [1]);
ttendif
#endif
  >
  else {
    disable_interrupt((int)rtP.s2_S_Function.P6tl]-1);
    deinstallintvector((int)rtP.s2_S_Function.P6[1]);
ttifdef   IOENABLE
    printfC'INT #%d disabled.\n", (int)rtP.s2_S_Function.P6 [1]);
#endif
  }
  /* 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);
  writedac(BASEBOARD, 3, 0);
  convert_dac(BASEBOARD, 3);
  /* End of user specific code (MdlTerminate function exit) */
}

/* model registration */
#include "Example.reg"
```

K.6 Model Parameters File

The model parameter file, *Example.prm*, is listed below :

```
/*
 * Example.prm
 *
 * Real-Time Workshop code generation for Simulink model "Example".
 *
 * RTW file version           : 2.11 (Aug 28, 1997)
 * RTW file generated on      : Fri Nov 12 08:34:15 1999
 * TLC version                : 1.0 (Dec 12 1997)
 * C source code generated on : Fri Nov 12 08:34:15 1999
 *
 * Relevant TLC Options:
 *   InlineParameters = 0
 *   RollThreshold = 5
 *   FileSizeThreshold = 50000
 *   CodeFormat = RealTime
 */

#include <simstruc.h>
/* Default Parameters Structure */
Parameters rtP =
  {
  /* Gain: <S3>/Gain */
  {
    1.0                                /* Gain */
  } ,
  /* S-Function: <S2>/S-Function */
  {
    1.0,                               /* Computed: PlSize[0] */
    1.0,                               /* Computed: PISizefl] */
    1.0,                               /* PI */
    1.0,                               /* Computed: P2Size[0] */
    1.0,                               /* Computed: P2Size[l] */
    1.0,                               /* P2 */
    1.0,                               /* Computed: P3Size[0] */
    1.0,                               /* Computed: P3Size[l] */
    1.0,                               /* P3 */
    1.0,                               /* Computed: P4Size[0] */
    1.0,                               /* Computed: P4Size[l] */
    0.0,                               /* P4 */
    1.0,                               /* Computed: P5Size[0] */
    1.0,                               /* Computed: P5Size[l] */
    1100.0,                            /* P5 */
    1.0,                               /* Computed: P6Size[0] */
    6.0,                               /* Computed: P6Size[l] */
    1.0,                               /* P6[0] */
    2.0,                               /* P6[l] */
    3.0,                                /* P6[2] */
    4.0,                                /* P6[3] */
    9.0,                                /* P6[4] */
    10.0                                /* P6[5] */
  }
};

/* Block I/O Structure */
BlocklO rtB;

/* Parent Simstruct */
SimStruct model_S;
SimStruct *const rtS = &model_S;
```

K.7 Model Registration File

The source code file containing all model registration functions, *Example.reg,* is listed below:

```
/*
 * Example.reg
 *
 * Real-Time Workshop code generation for Simulink model "Example".
 *
 * RTW file version            : 2.11 (Aug 28, 1997)
 * RTW file generated on       : Fri Nov 12 08:34:15 1999
 * TLC version                 : 1.0 (Dec 12 1997)
 * C source code generated on : Fri Nov 12 08:34:15 1999
 *
 *
 * Relevant TLC Options:
 *   InlineParameters = 0
 *   RollThreshold = 5
 *   FileSizeThreshold = 50000
 *   CodeFormat = RealTime
 *
 * Simulink model settings:
 *   Solver     : FixedStep
 *   StartTime  : 0 s
 *   StopTime   : 0 s
 *   FixedStep  : 0.0002 s
 *
 * SOURCES: Example.c
 *
 * MdlUpdate: Required
 */

/* Function to initialize sizes */

void MdllnitializeSizes(void)
{
  ssSetNumContStates(rtS, 0);     /* Number of continuous states */
  ssSetNumDiscStates(rtS, 0);     /* Number of discrete states */
  ssSetNumOutputs(rtS, 0);        /* Number of model outputs */
  ssSetNumlnputs(rtS, 0);         /* Number of model inputs */
  ssSetDirectFeedThrough(rtS, 0); /* The model is not direct feedthrough*/
  ssSetNumSampleTimes(rtS, 2);    /* Number of sample times */
  ssSetNumModes(rtS, 0);          /* Number of mode elements */
  ssSetNumBlocks(rtS, 7);         /* Number of blocks */
  ssSetNumBlocklO(rtS, 4);        /* Number of block outputs */
  ssSetNumBlockParams(rtS, sizeof(Parameters)/sizeof(realT));
                                  /* Number of block parameters */
}

/* Function to initialize sample times */
void MdllnitializeSampleTimes(void)
{
  /* task periods */
  ssSetSampleTime(rtS, 0, 0.0002);
  ssSetSampleTime(rtS, 1, 1);

  /* task offsets */
  ssSetOffsetTime(rtS, 0, 0);
  ssSetOffsetTime(rtS, 1, 0);
}

/* Function to register the model */
SimStruct *Example(void)
{
  static struct _ssMdlInfo mdllnfo;

  memset((char *)rtS, 0, sizeof(SimStruct));
  memset((char *)Smdllnfo, 0, sizeof(struct _ssMdlInfo));

  ssSetMdllnfoPtr(rtS, &mdllnfo);

  {
    uintT I;
    /* timing info */
    {
      static time_T [NSAMPLE_TIMES];
      static time_T [NSAMPLE_TIMES];
      static timeT [NSAMPLETIMES];
      static int_T [NSAMPLEJTIMES] ;
      static booleanT [NSAMPLE_TIMES];

      for(i =0 ; i < NSAMPLE_TIMES; i++) {
        mdlPeriodfi] = 0.0;
        mdlPeriod[i] = 0.0;
        mdlTaskTimes[i] = 0.0;
      }

      memset((char_T *)SmdlTsMap[0], 0, 2 * sizeof(int_T));
      memset((char_T *)&mdlSampleHits[0] , 0, 2 * sizeof(booleanT));

      ssSetSampleTimePtr(rtS, fcmdlPeriod[0]);
      ssSetOffsetTimePtr(rtS, SmdlOffset[0]);
      ssSetSampleTimeTasklDPtr(rtS, SmdlTsMap[0]);
      ssSetTPtr(rtS, SmdlTaskTimes[0]);
      ssSetSampleHitPtr(rtS, imdlSampleHits[0]);
    }

#if defined(MULTITASKING)
    {
      static boolean_T mdlPerTaskSampleHits[NSAMPLETIMES *NSAMPLE_TIMES];
      memset((charT *)SmdlPerTaskSampleHits[0],0,2*2*sizeof(booleanT));
      SsSetPerTaskSampleHitsPtr(rtS, SmdlPerTaskSampleHits[0]);
    }
#endif

    /* model work vectors */
    {
      realJT *b = (real_T *) &rtB;
      ssSetBlockIO(rtS, b);
      for(i = 0; i < sizeof(BlocklO)/sizeof(real_T); i++) {
        b[i] = 0.0;
      }
    }

    /* Model specific registration */
    ssSetRootSS(rtS, rtS);
    ssSetVersion(rtS, SIMSTRUCT_VERSION_LEVEL2);

    ssSetModelName(rtS, "Example");
    ssSetPath(rtS, "Example");

    ssSetTStart(rtS, 0);
    ssSetTFinal(rtS, 0);
    ssSetStepSize(rtS, 0.0002);
    ssSetFixedStepSize(rtS, 0.0002);

    ssSetLogT(rtS, "");
    ssSetLogX(rtS, "");
    ssSetLogY(rtS, "");
    ssSetLogXFinal(rtS, "");
    ssSetLogMaxRows(rtS, 0);
    ssSetLogDecimation(rtS, 1);

    ssSetChecksum0(rtS, 2735119696U);
    ssSetChecksuml(rtS, 2475633512U);
    ssSetChecksum2(rtS, 3952932782U);
    ssSetChecksum3(rtS, 953123913U);

    /* model parameters */
    ssSetDefaultParam(rtS, (real_T *) &rtP);
  }
  return rtS;
}
```

K.8 Make File

```
#*******************************************************************

SYS_TARGET_FILE  = ii.tlc
MAKE             = gmake
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              = Example
MODEL_MODULES      =
MODEL_MODULES_OBJ =
MAKEFILE           = Example.mk
MATLABROOT         = C:\MATLAB
MATLABBIN          = C:\MATLAB\bin
S_FUNCTIONS        =
S_FUNCTIONS_OBJ    =
SOLVER             =
SOLVEROBJ          =
NUMST              = 2
TID01EQ            = 0
NCSTATES           = 0
BUILDARGS          = TMR0=NO UPLD=NO IO=DISABLE
COMPUTER           = PCWIN

#------------------------------H pc32 Definition-------------------------
#
BOARDTYPE          = PC32
DSP_FAMILY          = 3 0
COMPILER           = TIFPC
#-------------------------------------TI Tools----------------------------
#
# You may need to modify the TIROOT  if you have installed the
# Texas Instrument Compiler in a different location.
```

```
#
TI_ROOT     = c:\fltc
TI_FLAGS    = -v$(DSPFAMILY)

CC = §(TI_ROOT)\cl3 0
LD = $(TI_ROOT)\lnk3 0

#-----------------------------------II Tools --------------------------------
#
II_ROOT     = $(MATLABROOT)\rtw\c\ii
II_COMPILER = $(II_ROOT)\ti_fpc

II_CMD      = $(II_COMPILER)\iiPC32.cmd
II_BOOT     = $(IICOMPILER)\vectors.obj

#II_BOARD    = c:\pc32cc

PC32_DOWNLOAD = $(IIROOT)\D_Load.exe

#--------------------------------Include Path-------------------------------

MATLAB_INCLUDES = \
$(MATLABROOT)\simulink\include; \
$(MATLAB_ROOT)\extern\include; \
$(MATLABROOT)\rtw\c\src; \

$(MATLABROOT)\rtw\c\libsrc;

TI_INCLUDES = $(TI_ROOT); $(TI_ROOT)\Include\Target;

II_INCLUDES = $(IIBOARD)\Lib\Target

INCLUDES = .; $(MATLAB_INCLUDES) $(TI_INCLUDES) $(IIINCLUDES)

#-----------------------------------C Flags-----------------------------
# Required Options
REQ_OPTS    = $(TI_FLAGS) -q -eo .0$(DSPFAMILY)

# Optimization Options
OPT_OPTS        = -xO -o3
```

```
# Debug Options
DBGJ3PTS          =

CCOPTS            = $(REQ_OPTS) $(OPT_OPTS) $(DBGOPTS) -dIO_$(10) \
                  -dTMRO_$(TMR0)  -dUPLD_$(UPLD)

CPP_REQ_DEFINES = -dMODEL=$(MODEL) -dRT -dNUMST=$(NUMST) \
                  -dTID01EQ=$(TID01EQ)  -dNCSTATES=$(NCSTATES)



CFLAGS            = $(CC_OPTS) $(CPP_REQ_DEFINES) $(CPPDEFINES)

LDFLAGS           = -x -a -cr -heap 0x2000 $(II_BOOT) -m $(MODEL).map

#---------------------------Source Files ---------------------------

REQ_SRCS          = PC32main.c pc32upld.c rt_sim.c rtmatrx.c $(MODEL).c
OPT_SRCS          =
S_FCN_SRCS        = $(S_FUNCTIONS)
INT_SRCS          = $(SOLVER)

REQOBJS           = $(REQSRCS:.c=.o$(DSPFAMILY))
OPT_OBJS          = $(OPTSRCS:.c=.o$(DSPFAMILY))
S_FCN_OBJS        = $(SFCNSRCS:.c=.o$(DSPFAMILY))
INTOBJS           = $(INTSRCS:.c=.o$(DSPFAMILY))
OBJS              = $(REQ_OBJS) $(OPT_OBJS) $(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
#
COPTION := $(CFLAGS)
C_DIR   := $(INCLUDES); $(CDIR)
C_MODE   = PROTECTED

#-------------------------------- Rules --------------------------------
```

```
$(PROGRAM) : $(OBJS)
      echo $(OBJS) > $(MODEL).lin
      echo $(II_CMD) >> $(MODEL).lin
      $(LD) $(LDFLAGS) -O $@ $(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
%.o$(DSP_FAMILY) : c:\matlab\rtw\c\ii\ti_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 S(MODEL).h
      del S(MODEL).map
      del $(MODEL).o3 0
      del S(MODEL).prm
      del S(MODEL).reg
      $(PC32_DOWNLOAD) $(PROGRAM)
      echo $(DOWNLOAD_SUCCESS) $(PROGRAM)

#____________________________Dependencies ----------------------------
iirt_main.o$(DSP_FAMILY) : $(MODEL).c
$(OBJS)      : $(MAKEFILE)
```

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

BASSETT1 Paul Bassett, "Managing for Flexible Software Manufacturing", *Computer,* VOL. 31, NO. 7, pp. 100-102, IEEE Computer Society, July 1998

BLERK1 Bruce van Blerk, "Development of a Scaled Down Paper Machine to Demonstrate the Principles of Tension Control", *MScEng Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, December 1998

BURRBROWN1 "ADS7805 16-Bit lOus Sampling CMOS Analogue-to-Digital Converter", Burr-Brown Data Sheet.

BURRBROWN2 "DAC712 16-Bit Digital-to-Analogue 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 Taiwan Limited, January 1998

CRAVOTTA1 Nicholas Cravotta, "Buying a Single Board Computer", *Embedded Systems Programming,* pp. 75-83, May 1997

DIANA1 G. Diana, M. Pillay, A.W. Stylo, M.L. Walker, "An Advanced Development Tool for Rapid Prototyping of Motion Control Applications", to be published.

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

F0RSYTHE1 W. Forsythe, R.M. Goodall, "Digital Control - Fundamentals, Theory and Practice", *Macmillan Education Ltd.,* England, 1991, ISBN 0-333-53501-4

FOSTER1 Kenneth R. Foster, "Matrices and much, much more", Software Reviews, *IEEE Spectrum,* VOL. 34, NO. 2, February 1997

GABRIEL1 R. Gabriel, W. Leonhard, C. Nordby, "Microprocessor Control of Induction Motors Employing Field Coordinates", *Proceedings of the 2nd International Conference on Electrical Variable-Speed Drives,^.* 146-150, IEE Power Division, London, September 1979

GANSSLE1 Jack G. Ganssle, "Debuggers for Modern Embedded Systems", *Embedded Systems Programming,* pp. 58-65, November 1998

GORDON 1 V. Scott Gordon, James M. Bieman," Rapid Prototyping : Lessons Learned", *IEEE Software,* VOL. 12, NO. 1, pp. 85-95, IEEE Computer Society, January 1995

GUMAS1 Charles Constantine Gumas, "The DSP Workshop - a compelling CASE", Software Reviews, *IEEE Spectrum,* VOL. 36, NO. 8, August 1999

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

HEMME1 Alexander W.M. Hemme, "A Software Platform for a Transputer Based Embedded Real-Time System for Motion Control Applications", *MScEng Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, April 1992

INNOVATIVE1 "PC32 Hardware Manual", Innovative Integration, May 1995

INNOVATIVE2 "PC32 Developer's Package Software Manual", Innovative Integration, July 1996

KLEINHANS1 C. Kleinhans, G. Diana, R. Harley, M. McCulloch, "The Application of CASED as a Simulation Tool for the Design and Analysis of Variable Speed Drives", *Conference Proceedings of the IEEE Industrial Applications Society Annual Meeting,* IAS, pp. 750-757, Denver, Colorado, 1994

K0ZICK1 Richard J. Kozick, Curtis C. Crane, "An Integrated Environment for Modelling, Simulation, Digital Signal Processing, and Control", *IEEE Transactions on Education,* VOL. 39, No. 2, pp. 114-119, IEEE Professional Technical Group on Education, May 1996

LEM1 LEM Group web site - http://lem.com

LEONHARD1 Werner Leonhard, "Control of Electric Drives", *Springer-Verlag,* Germany, 1990, TSBN 3-540-13650-9

LIN1 Kun-Shan Lin, Gene A. Frantz, Ray Simar, Jr., "The TMS320 Family of Digital Signal Processors", *Proceedings of the IEEE,* Vol. 75, No.9, September 1987

LINZENKIRCH1 Edmund Linzenkircher, "The Importance of Control Engineering in Automation", *South African Instrumentation and Control,* VOL. 15, NO. 7,pp.50-51, July 1999

MATHWORKS1 "Using MATLAB, Version 5", The Math Works inc., June 1997

MATHWORKS2 "Using Simulink, Version 2.2", The MathWorks Inc., January 1998

MATHWORKS3 "Using Simulink, Version 3", The MathWorks Inc., January 1999

MATHWORKS4 "Simulink, Writing S-Functions, Version 3", The MathWorks Inc, October 1998

MATHWORKS5 "Real-Time Workshop, User's Guide, Version 3", The MathWorks Inc., January 1999

MATHWORKS6 "Target Language Compiler Reference Guide, Version 1.2", The MathWorks inc., January 1999

MATHWORKS7 MathWorks web site - http://www.mathworks.com

MEYER1 Benjamin S. Meyer, "A Transputer Based Digital Controller with High Performance I/O for Motion Control Applications", *MScEng Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, December 1992

MIR0TZNIK1 Mark S. Mirotznik, "Translating Matlab programs into C code", Software Reviews, *IEEE Spectrum,* VOL. 33, NO. 2, pp. 63-64, February 1996

M00DLEY1 Lynden Modley, "Position Controller for a DC *Drive",BScEng Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, December 1999.

RANDELHOFF1 Mark C.Randelhoff, "Automated Generation of Predictable Real-Time Code for Motion Control Applications", *PhD Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, January 1995

SENESE1 Brian Senese, "Meeting Aggressive Schedules Through Smart Integration", *Embedded Systems Programming,* pp. 30-45, September 1997

SLIVINSKI1 Charles Slivnski, Jack Borninski, "Control System Compensation and Implementation with the TMS32010", *Digital Signal Processing Applications with the TMS320 Family (Theory, Algorithms, and Implementations),* VOL. 1, Texas Instruments, 1989

STURGEON1 Shawn Sturgeons, "DSP Based Field Oriented Control of an Induction Machine", *BScEng Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, November 1998

STYLO1 Adam Stylo, Greg Diana, "A Low Cost, High Performance PC Based Integrated Real-Time Motion Control Development System", *Proceedings of the 7th Southern African Universities Power Electronics Conference,* Stelenbosch, South Africa, January 1997

STYLO2 Adam Stylo, Greg Diana, "A Low Cost, High Performance PC Based Integrated Real-Time Motion Control Development System", *Proceedings of the IEEE International Symposium on Industrial Electronics*, pp. 127- 130, Pretoria, South Africa, June 1998

STYL03 Adam Stylo, Greg Diana, "Hard Real-Time Code Generation Using Simulink And Real-Time Workshop", *Proceedings of the 8th Southern African Universities Power Electronics Conference*, Potchefstroom, South Africa, January 1999

STYLO4 Adam Stylo, Greg Diana, "An Advanced Real-Time Research and Teaching Tool for the Design and Analysis of Control", *Proceedings of the AFRICON Conference*, Cape Town, South Africa, September 1999.

TEXASINSTR1 "TMS320C3x User's Guide", SPRU031D, Texas Instruments Inc., October 1994.

TEXASINSTR2 "TMS320C32 User's Guide", Addendum to the TMS320C3x User's Guide, Texas Instruments Inc., March 1995

TEXASINSTR3 "Field Oriented Control of 3-Phase AC-Motors", BPRA073, Texas Instruments Europe., February 1998.

TRZYNADLO1 Andrzej M. Trzynadlowski, "DSP Controllers - An Emerging Tool for Electric Motor Drives", *IEEE Industrial Electronics Society Newsletter*, VOL. 45, NO. 3, pp. 11-13, IEEE, September 1998

VATER1 Jörg Vater, "The Need For And The Principle Of High-Resolution Incremental Encoder Interfaces In Rapid Control Prototyping", *Proceedings ofPCIM '97*, Germany, June 1997

WALKER1 Myles Walker, "Test Bed System to Investigate the Energy Efficiency of Variable Speed Drive Systems Under Variable Load Conditions", *MScEng Thesis in preparation*, Dept. of Electrical Engineering, University of Natal, Durban, October 1999.

W0RTHMANN1 Cedric Worthmann, "Feasibility Study of a Neural Network Current Controller for a Boost Rectifier", *MScEng Thesis in preparation,* Dept. of Electrical Engineering, University of Natal, Durban, January 2000.

WEBSTER 1 M.R. Webster, "Field Oriented Control of AC Motors Using Transputers", *MScEng Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, January 1991

WOODWARD 1 D. Woodward, "A Transputer Based High Speed Embedded Controller for Motor Drives", *MScEng Thesis,* Dept. of Electrical Engineering, University of Natal, Durban, November 1992

the AC induction machine can be directly controlled by adjusting the current. The method allows a good transient response to be obtained from an AC machine and is a good candidate for substituting the traditional DC machine in variable speed drive applications.

The implementation of an FOC algorithm in practice is a computationally expensive task [RANDELHOFF1, TEXASINSTR3] and only became viable with the advent of fast programmable microprocessors. Designing software to implement the FOC mathematics on an embedded platform would normally be beyond the time frames allocated for final year design projects. In the past a formal FOC design was only attempted as part of postgraduate level research projects [RANDELHOFF1, HEMME1].

Using the CSDE, Mr. Sturgeon implemented an FOC algorithm [STURGEON1] in 1998 in the time allocated for his 4th year design. The project included all stages of formal control design from initial modelling and simulation right through to demonstrating a working prototype. Mr. Sturgeon was awarded the prize for the best final year design in electrical engineering in 1998 at UND.

The FOC controller topology is shown in Fig. 9.14. This Simulink diagram was used for offline simulation. The resultant controller was then modified, as shown in Fig. 9.15, before being implemented in real-time. All Scope Channel blocks which were used to monitor the feedback signals in Fig. 9.15, are omitted in order to keep the diagram clear. A photograph of Mr. Sturgeon with the CSDE workstation as well as the hardware used to implement and test the FOC design is shown in Fig.9.16.

9.4.1 Feedback and Observations

The FOC design was the first formal student motion control project to use the CSDE and showed that the lack of support for synchronous sampling was a critical problem. The feedback signals were distorted by switching noise from the power electronic devices and the control loops were difficult to tune and stabilise.

To solve the synchronous sampling problem both hardware and software modifications to the CSDE were necessary. On the PWM/Tacho card, the interrupt signal from the PWM ASIC was inverted using a spare gate and patched to the El 0 interrupt line on the PC32 card's expansion header. Section 4.5.2 in Chapter 4 described the PWM ASIC and its interrupt signal. The

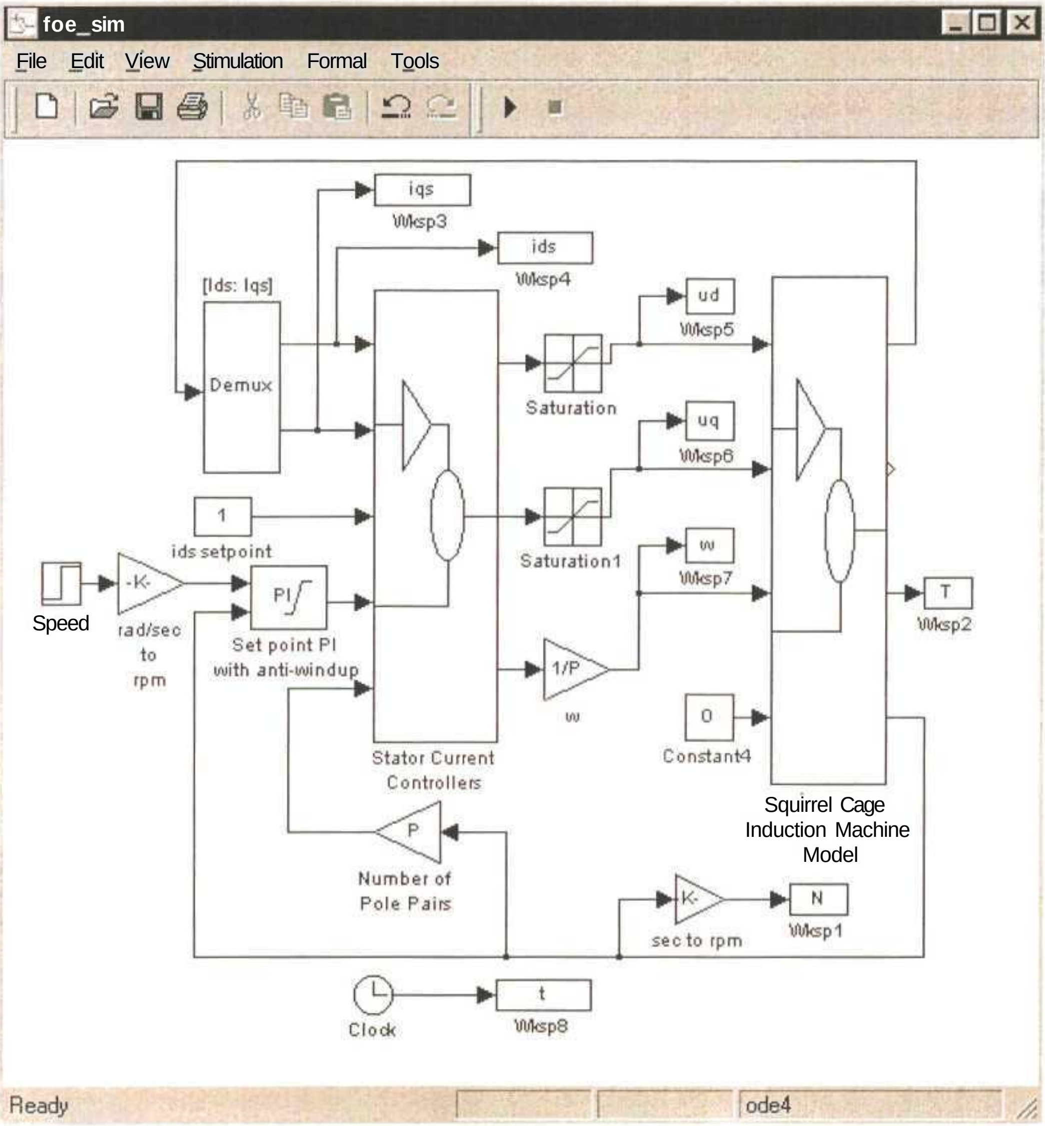


Fig. 9.14 : FOC simulation

software modification involved creating an AD Trigger block to allow software triggering of the ADC channels, as described in section 7.5.1, Chapter 7.

To achieve synchronous sampling the PC32 Int Support block is used to trigger the AD Trigger block via El 0. The ADC end of conversion interrupt (El 1) is then used to trigger the control loop. This procedure ensures that the feedback signals used to update the controller are sampled free of switching noise.

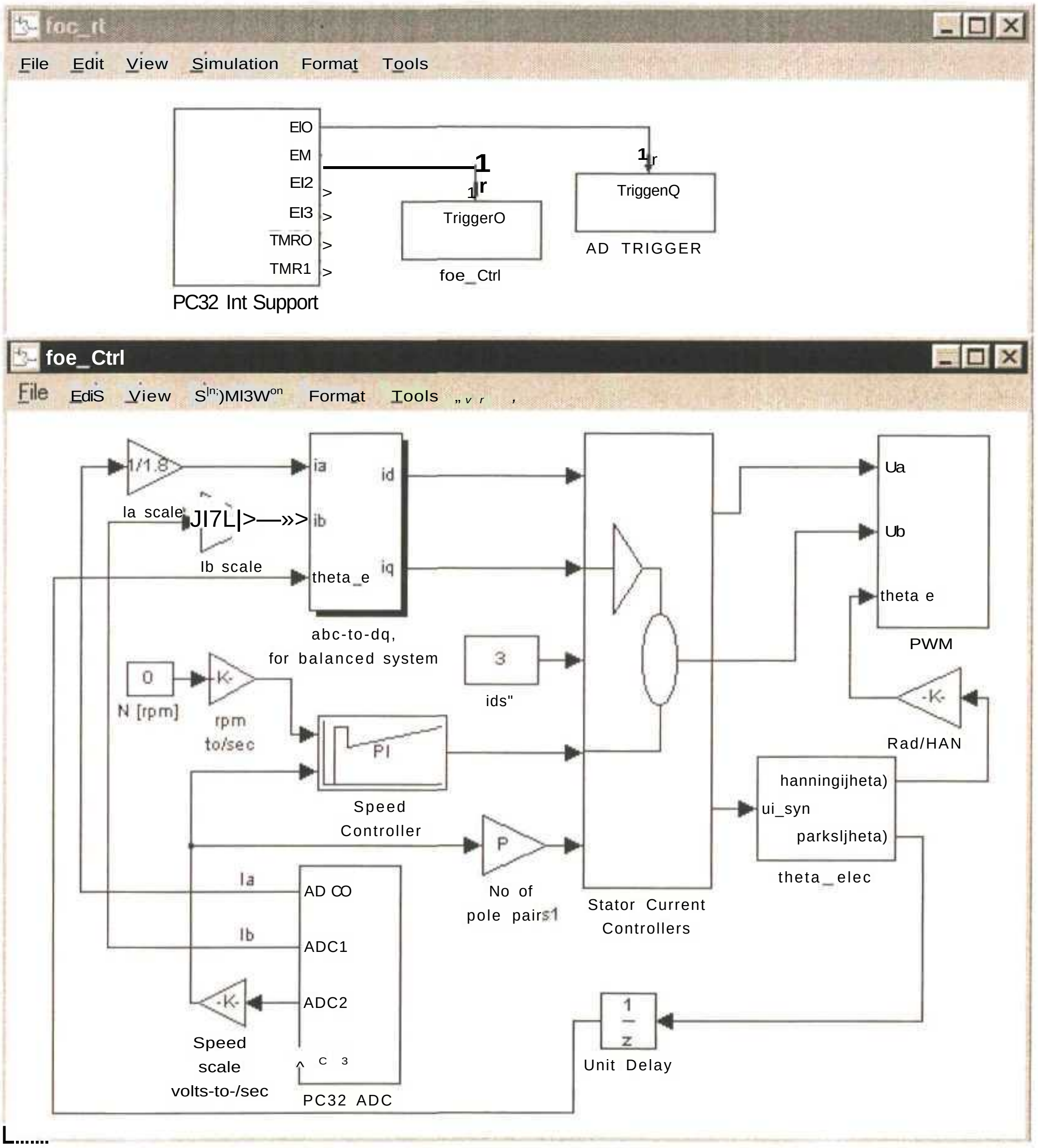


Fig. 9.15 : FOC real-time controller

9.5 DC Machine Controller for a Ball Catcher

The final year design project of Mr. Moodley in 1999 involved the design of a precision position controller for a DC machine using the CSDE [MOODLEYl]. The setup was used to implement an automatic ball catcher. Fig. 9.17 shows a simplified diagram of the mechanical assembly. A tennis ball is placed in the cup at the bottom of its travel. The arm then swings counterclockwise by 180 °. As the ball passes the sensor the cup is moved clockwise back to its bottom position in time to catch the falling ball. After a short delay the cycle is repeated.

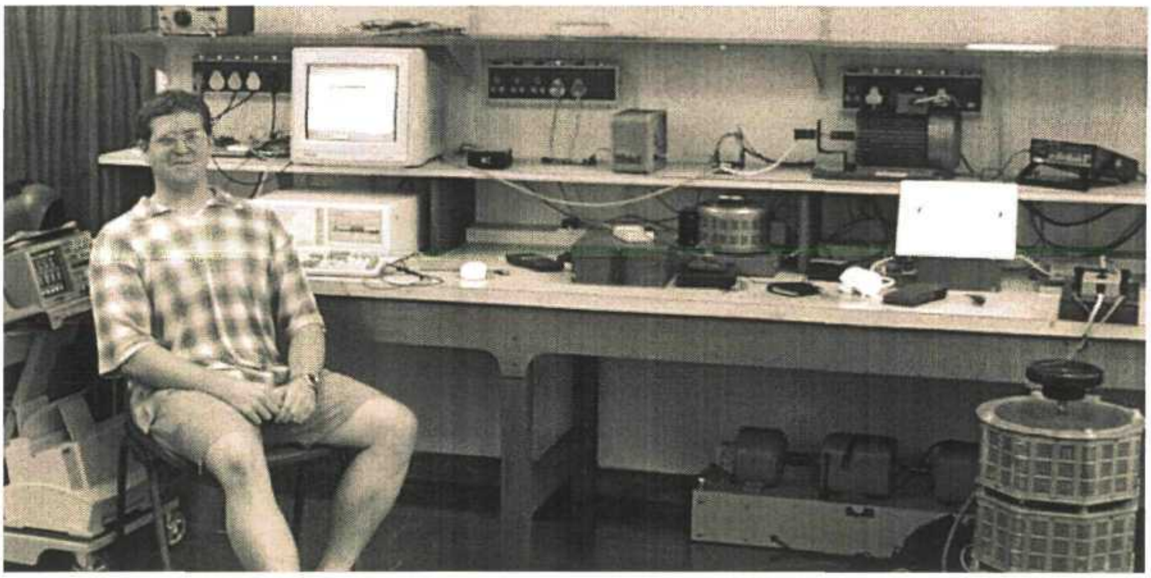


Fig. 9.16 : Photograph of the FOC design layout

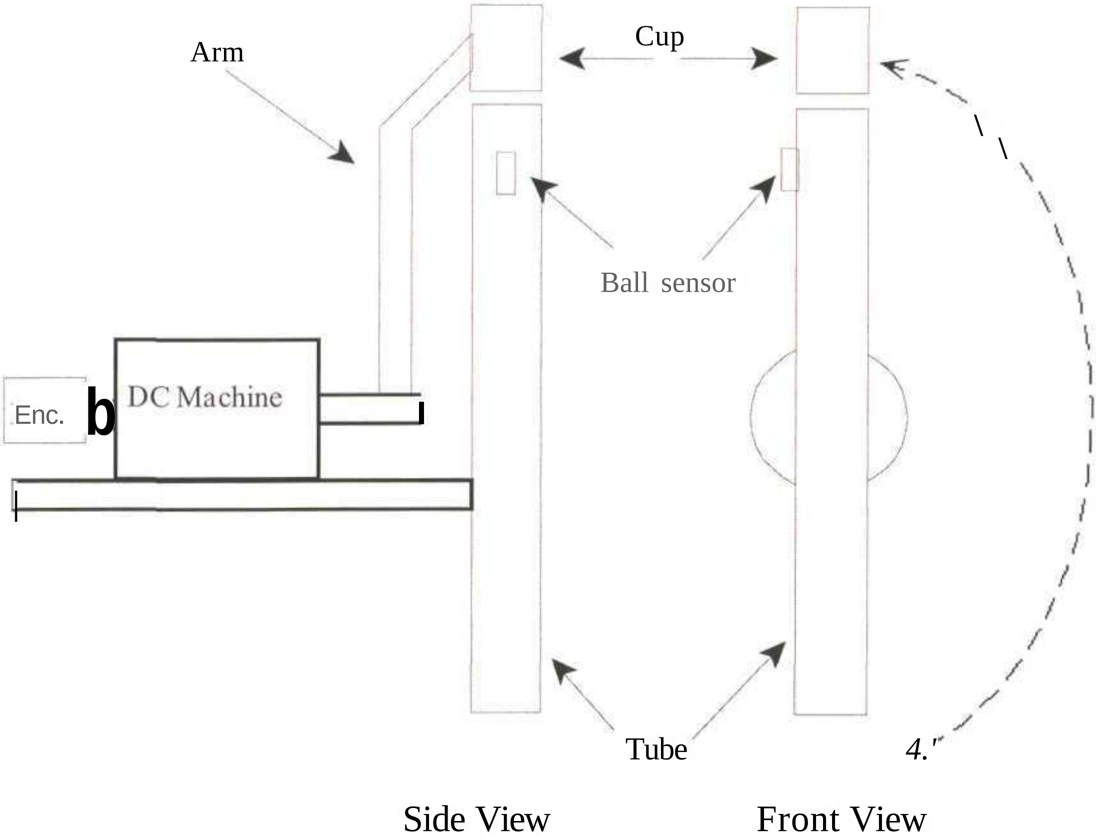


Fig. 9.17 : Ball catcher mechanics

In order to achieve the necessary degree of position control, the design necessitated the use of a high resolution incremental encoder. The TC3005H IC on the expansion card described in section 4.5.4 was used to interface to the encoder. A custom driver block for use with the RTW was created by Mr. Moodley for that purpose in collaboration with the author.

The design process was completed in a structured manner well before the deadline and Mr. Moodley was awarded the top control design prize for his project. A photograph of the complete Ball Catcher assembly is shown in Fig. 9.18.

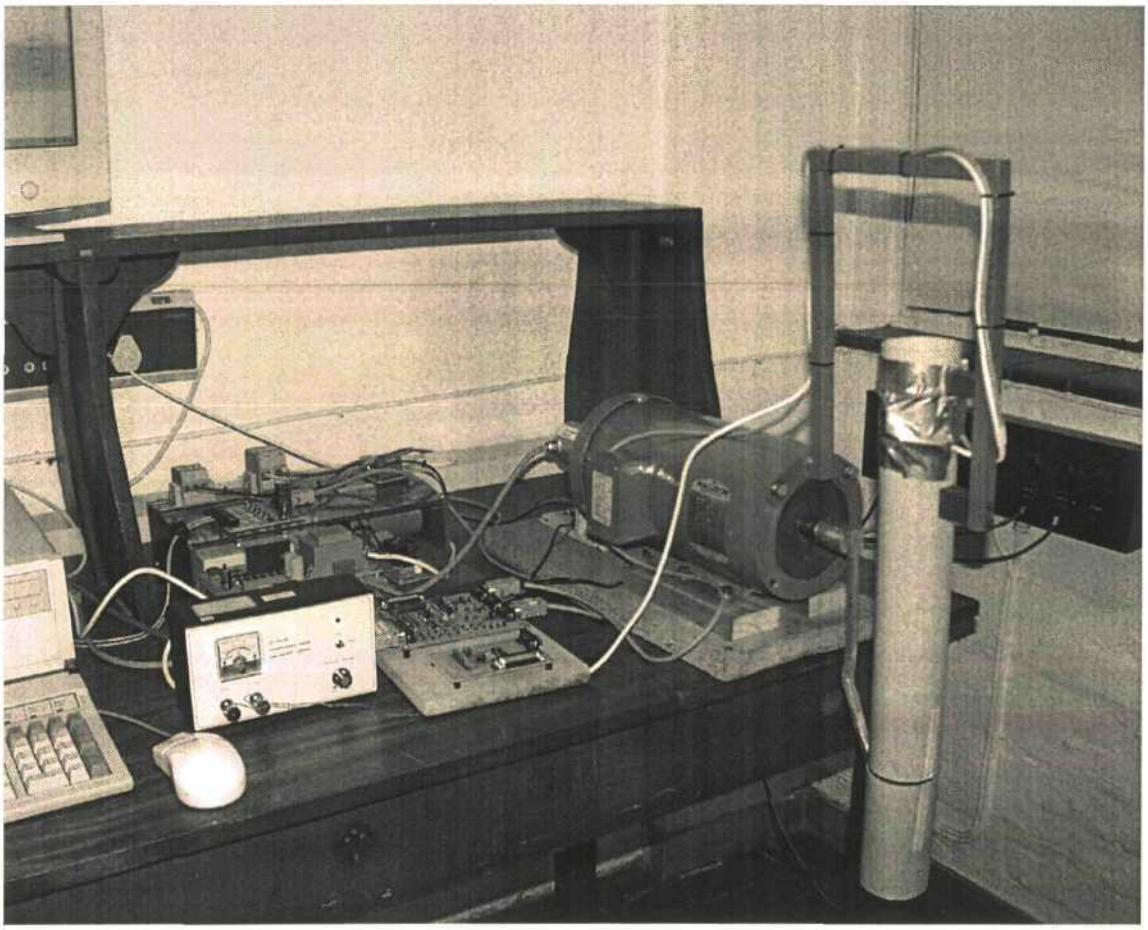


Fig. 9.18 : Photograph of the Ball Catcher setup

Fig. 9.19 shows the Simulink diagram used for offline simulation and tuning of the current, speed and position control loops for the DC motor. These controllers were bundled into a single subsystem block and used to construct the real-time prototype, shown in Fig. 9.20. In the simulation a signal generator is used to provide the position reference signals. In the real time diagram this is replaced by a photo sensor fed into ADC 3 and an Automatic Re-trigger Block which introduces some delay before the catcher arm is swung back to the top. The delay is necessary in order to allow the ball to stop bouncing in the cup after being caught. Another difference between the simulation and real-time diagrams is the replacement of the DC machine model with the PWM driver block and feedback inputs from sensors on the real machine.

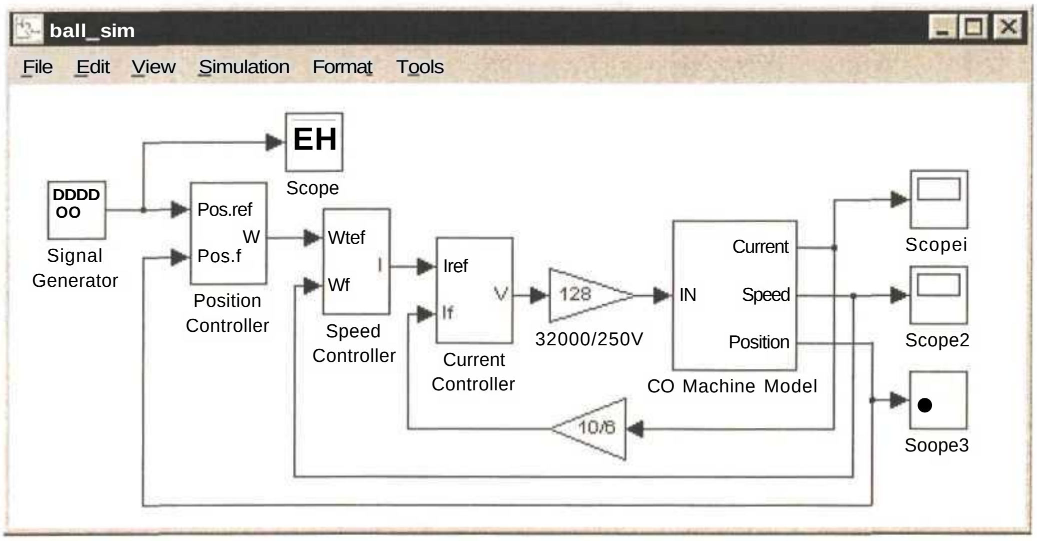


Fig. 9.19 : Ball Catcher controllers simulation

9.5.1 Feedback and Observations

The ball catcher project uncovered a shortcoming in the mechanism used by the CSDE in uploading data for visualisation. The Scope Channel blocks capture and buffer samples of data which are then transferred via the DPRAM to the Display utility on the PC, as described in Chapter 6, section 6.4. The Display utility can only display up to 1000 samples in a window. Since the current control loop for the ball catcher executed at 5 kHz, samples are captured every 0.2 ms and the Display utility can plot at most 200 ms worth of data. To observe the performance of the speed and position control loops much longer capture period is required.

In response to this observation the Scope Channel block was modified to allow for down sampling. Effectively, data is captured at a lower frequency and the display period in the Display utility can be adjusted as described in Chapter 7, section 7.5.6.

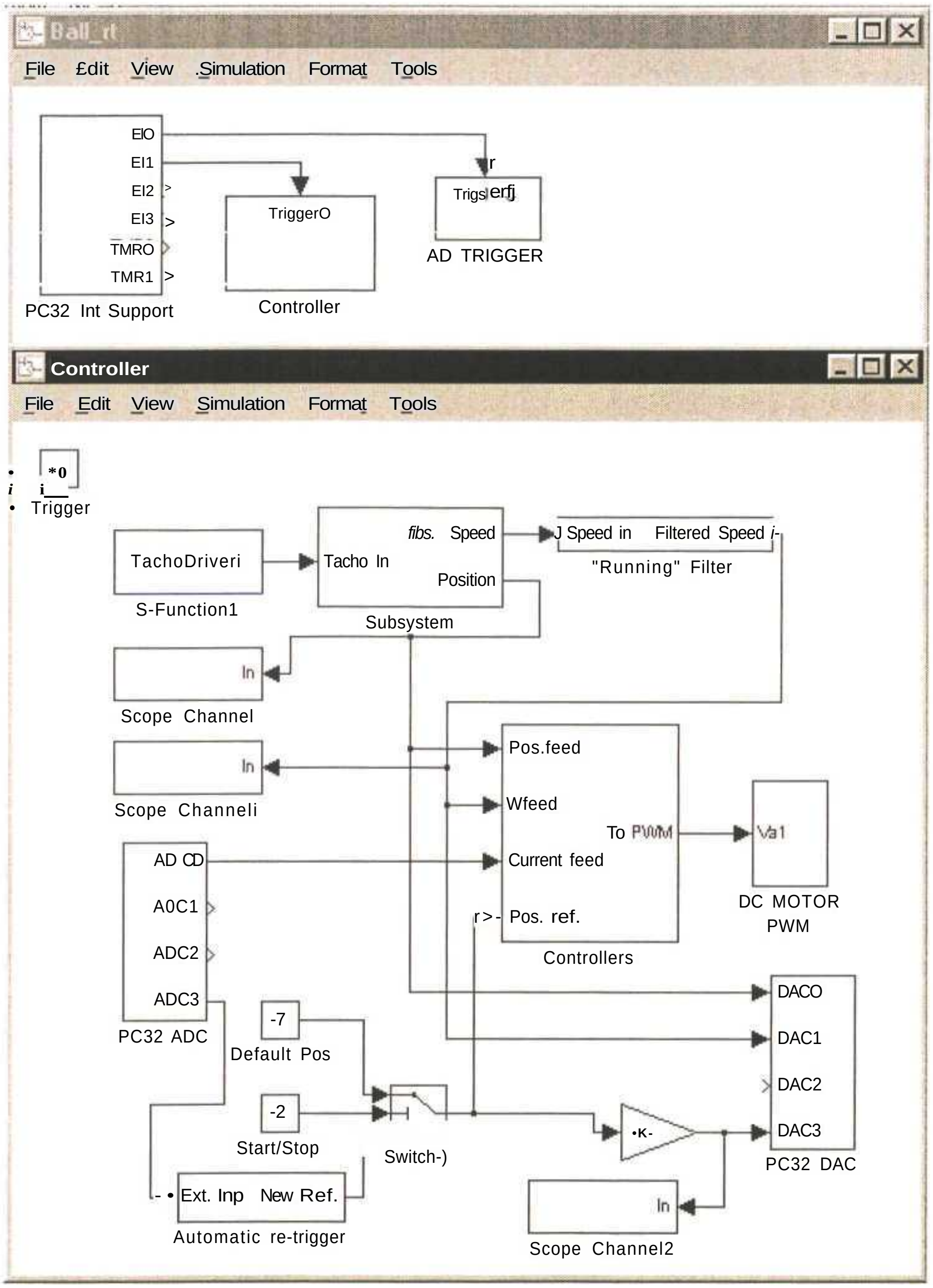


Fig. 9.20 : Ball Catcher real-time controller

9.6 Conclusion

This Chapter verified the usefulness of the CSDE as an aid in the development of motion control. The operation of the CSDE was demonstrated using a DC machine speed controller as an example. The case studies in sections 9.4 and 9.5 show that by using the CSDE undergraduate students can successfully implement challenging motion control projects in the limited time allocated for the design courses.

By using the rapid prototyping environment, students can spend more time exploring the underlying control theory rather that be burdened with the low-level issues of developing and debugging a stable controller. There was an additional spinoff from the use of the CSDE in the number of field trials. The students found it easier to follow a formal and structured approach to the overall design and produced documentation above the quality usually expected at that level.

CHAPTER TEN
CONCLUSIONS

10.1 General

The general trend in motion control is towards microprocessor based solutions [TRZYNADLO1] and as the processing power of the available hardware improves, ever more sophisticated algorithms become practical. The complexity of designing a stable and reliable digital controller means that a structured approach is crucial as outlined below :

(i) Modelling - an accurate model of the machine is necessary before simulation can become useful,

(ii) Design and simulation - a control algorithm is proposed and its feasibility is tested in a offline simulation,

(iii) Hardware design - the controller hardware is designed or a commercially available platform can be used.

(iv) Software design - the simulated algorithm is converted into real-time code,

(v) Prototyping and validation - the controller is tested in real-time and final parameter adjustments take place.

The traditional approach is to program the controller software by hand - a tedious and error prone task. During the design process a number of the above steps might need to be revisited and the process becomes drawn out.

10.1.1 Undergraduate Design

It becomes especially difficult to follow the above design methodology at the undergraduate level where project time frames are short. Most electrical engineering students will have limited embedded programming experience, meaning that the coding and debugging of complex real-time software might take most of the project time. Time pressure and subsequent lack of adequate explorative simulation limit the complexity of control algorithms that can be successfully implemented.

10.1.2 Rapid Prototyping

The application of a rapid prototyping platform, like the CSDE, eliminates the software and hardware issues from the motion control design process. Students can focus their effort on modelling and simulation and observe the results almost immediately in real time on a practical system. In the limited time allocated students can explore advanced algorithms in more depth and verify their actual performance in real life.

10.2 Project Summary

The overall goal of the author's work was to develop a rapid prototyping environment to aid undergraduate students during the design of motion control. Taking into consideration the user group targeted a number of requirements were laid down :

(i) Low cost - the overall cost needs to be kept down for the system to be practical in an undergraduate environment,

(ii) Adequate processing power - the processing platform needs to support complex control algorithms,

(iii) Ease of use - the learning curve associated with the system must be as short as possible so that valuable design time is not further compromised.

10.2.1 The CSDE

To respond to the above requirements, the CSDE was based on the proven and well established modelling and simulation platform from MathWorks [FOSTER1, GUMAS1, MIROTZNIK1]. The MATLAB and Simulink packages are widely available to students and their use is compulsory in a number of courses. As a result final year design students are well versed in the use of those tools for modelling and simulation. An additional benefit to the CSDE is the cost saving - no additional software license need to be purchased.

The hardware platform for the CSDE, the PC32 DSP controller, represents a good balance between price and processing power and to keep further costs down, the additional PWM / Tacho expansion card was designed in-house.

10.2.2 Field Trials

A number of students were selected to use the CSDE during their design courses. The level and quality of the resultant motion control projects demonstrated that the CSDE is a valuable tool which can aid students in tackling more challenging problems than was possible before.

Unfortunately no formal record was kept of the feedback from students who field trialed the CSDE. In all cases the students reported the CSDE to be a valuable design tool which allowed them to produce working prototypes early in their project schedule. The fact that two projects won the best final year design prizes is a good indication that the CSDE does, in fact, fulfill the requirements set out in Chapters 1 and 2.

10.3 Suggestions for Further Work

The CSDE in its present form represents a stable environment for the design and rapid prototyping of motion control. Notwithstanding the positive feedback from a number of student users, there remain a number of areas which can be further extended or improved. The following sections list a number of suggestions for such improvements.

10.3.1 Profiler and Exception Handler

Currently there is no way of predicting whether there is sufficient processing bandwidth to execute the controller designed in Simulink reliably. If the control loop execution time is longer than the sampling time in real-time the controller will fail to execute. The current version of the RTK does not provide for neat handling of such exceptions.

Firstly, the RTK can be extended to include exception handling and to fail safe in such cases. Secondly, a static profiler could be designed to estimate the worst case execution time for the designed control loop. The profiler output could aid in the selection of the maximum sampling frequency for the controller.

10.3.2 Implementation over a Network

The most expensive part of the CSDE setup is the controller hardware as well as the controlled machine. However, during a normal project cycle most time is spent on design and simulation and the real-time validation and prototyping usually only takes a small fraction of the project time. Presently the CSDE is limited to a single standalone system.

In the ideal scenario the controller hardware as well as the electric motor would be shared between a number of design workstations in a laboratory. A number of students could carry out their usual design and simulation on their own workstations and only link up via a network to the central station to verify their controller design and capture results. In this manner the cost of providing rapid prototyping facilities per student would be greatly reduced, while still maintaining the benefits of hands-on verification of final design on real hardware.

10.3.3 Improving Display Utility

Currently the Display utility applies a triggering algorithm to each channel individually and each channel is displayed in a separate window. This means that it is impossible to compare timing between two channels, as was seen in Fig. 9.6. To overcome this, timing information would need to be included into the data uploads from the target to enable the Display utility to synchronise individual channels. To improve the Display GUI further a number of improvements could be introduced. For example grids, axis labels and more flexible zooming options. Another useful feature would be the ability to log the plots to a file in various formats. The data could then be imported into MATLAB and compared directly with simulation results.

Appendix A

A.1 CSDE Users' Guide

This Appendix contains an edited version of the CSDE Users' Guide. Some of the material contained in this Appendix has already been discussed in more depth in the preceding chapters. However, the intention of the Users' Guide is not to summarise the thesis, but to be a practical 'hands-on' introduction to the CSDE.

A.2 Introduction

The Control System Development Environment (CSDE) represents a complete solution for the development of most control and digital signal processing solutions. All phases of development are conveniently bundled under a user friendly Graphical User Interface (GUI), from initial modelling through design, simulation right to final real-time prototyping and validation.

The CSDE integrates the following major components :

(i) MATLAB with Simulink and Real-Time Workshop from MathWorks Inc.
(ii) PC32 DSP Controller Card from Innovative Integration
(iii) Texas Instruments C Compiler for the TMS320C32
(iv) Real-Time Interface Software - custom libraries and visualisation tools

Fig. A.I shows how the various components of the CSDE interact to provide a complete solution. This User's Guide describes how to generate predicable real-time code from Simulink diagrams for execution on the PC32 Controller Card. As far as the user is concerned the only "visible" components of the CSDE are Simulink GUI, the physical I/O of the PC32 and Display which handles data plotting and capture on the PC. Most of the Real-Time Interface Software (RTIS) which forms the "glue" between the various components is essentially transparent to the user.

A.2.1 What is Simulink?

The MATLAB / Simulink environment forms the front-end GUI for the CSDE, as such it is important that the user if familiar with MathWorks' software before proceeding.

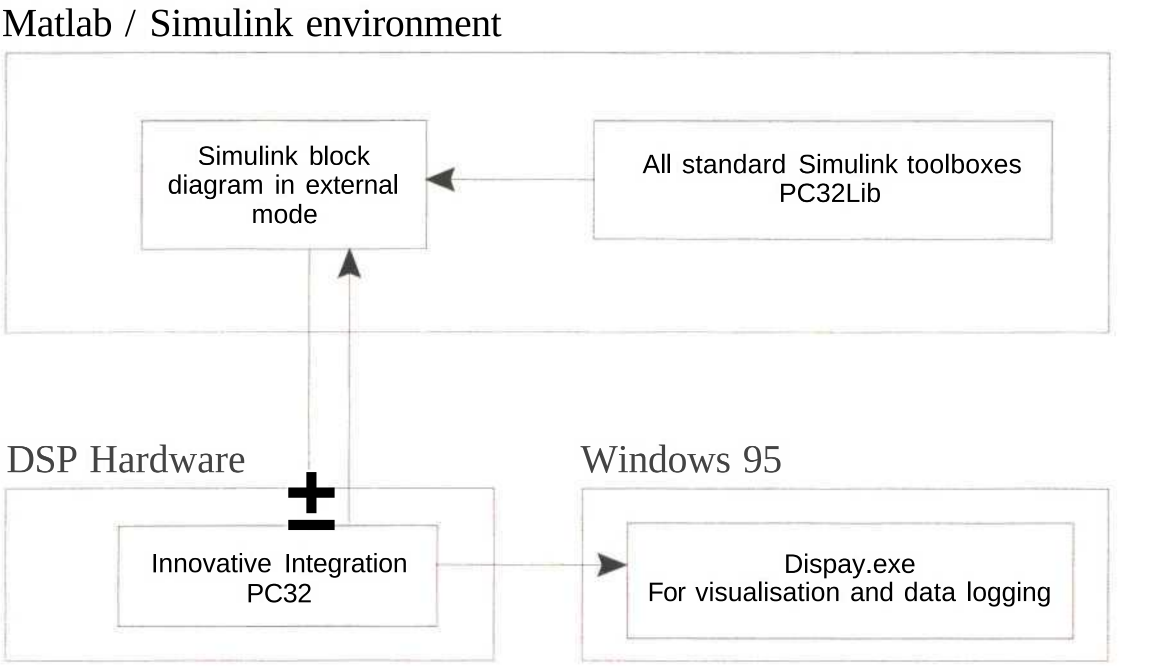

Matlab / Simulink environment
Simulink block diagram in external mode
All standard Simulink toolboxes
PC32Lib
DSP Hardware
Windows 95
Innovative Integration
PC32
Dispay.exe
For visualisation and data logging

Fig. A.I : Interaction of the major components of the CSDE from the users perspective.

Simulink is a graphical user interface (GUI) for the well known MATLAB system from MathWorks Inc. It is a software package for modeling, simulation and analyzing dynamical systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multi-rate, i.e. Have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a user-friendly GUI allowing users to intuitively build models as graphical block diagrams, using click-and-drag mouse operations. With this interface users no longer need to formulate equations in a language or program, but simply choose from a comprehensive collection of block libraries and additional toolboxes. Simulink also allows for custom blocks and libraries to be added by the user to facilitate for their specific needs.

Simulink models are hierarchal, so complex systems can be broken down and approached in a top-to-bottom or bottom-to-top manner. System can then be viewed at a high level, and by double-clicking one can navigate down through the levels to see increasing amount of detail.

For more information on the MATLAB \ Simulink environment please refer to the corresponding manuals shipped with the Math Works Inc products. In the following sections of this manual it is assumed that the reader is familiar with Simulink. Math Works Inc can be contacted at www.mathworks.com.

A.2.2 The PC32 Controller

The PC32 from Innovative Integration, shown in Fig. A.2, is an affordable and capable DSP card. The high performance 32-bit floating point TMS320C32 DSP from Texas Instruments is coupled with full featured analog and digital peripherals to form a complete DSP based acquisition and control system for the PC/AT on a single card.

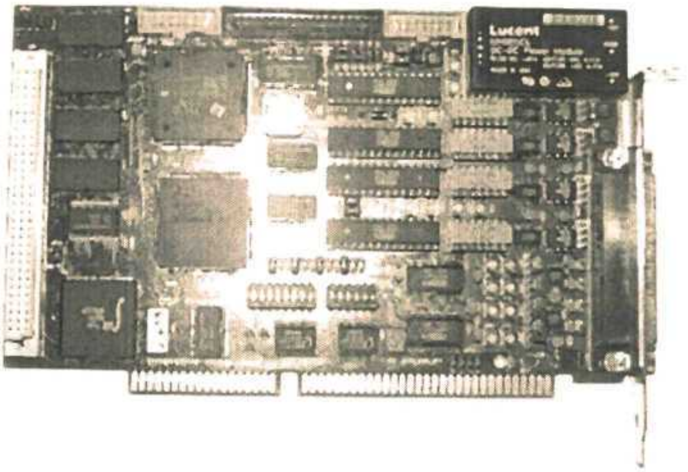


Fig. A.2 : Photograph of the Innovative Integration PC32 card.

The Texas Instruments TMS320C32 is capable of up to 60 MFLOPS/30 MIPS. On-chip peripherals include two 32-bit counter/timers, two flexible DMA controllers, 15 prioritized interrupts, and much more. The 'C32 has a memory range of 16M words total with on-chip memory control for wait states and bus sizing. Memory on the PC32 may be expanded up to 512Kx32 zero wait and 1 Mbyte, 1 wait-state memory for an optimal mix of performance, size and cost.

The PC32 features analog and digital I/O for a wide range of applications in process control, audio processing, data logging and signal processing. Features include four channels of 16-bit, instrumentation-grade analog-to-digital converters with -120 dB/decade tunable anti-alias filters capable of sampling up to 100K samples/sec, and 4 channels of 16-bit, instrumentation-grade digital-to-analog converters at 100khz per channel. Sixteen bits of high-drive digital I/O are also on-card. The PC32 is compatible with the full range of 3xBUS cards for I/O expansion including analog I/O, digital camera interfacing, prototyping and SCSI devices.

The PC32 is a half-card which plugs into a standard 16-bit ISA bus slot. The ISA bus interface provides high-performance dual-ported memory capable of 5 Mbytes/sec, on most platforms.

For more detailed information on the PC32 card please refer to Innovative Integration manuals. Innovative Integration can also be contacted at www.innovative-dsp.com.

A.2.3 Texas Instruments Tools

The CSDE relies on the Real-Time Workshop (RTW) to generate C code from Simulink diagrams. This C code then needs to be compiled and linked into an executable before it can be run on the target hardware.

As far as the user is concerned the Texas Instruments compiler is transparent and the CSDE ensures that all necessary parameters and settings are passed to the TI compiler.

A.2.4 Real-Time Interface Software

What makes the CSDE into a complete package is the glue interface software which ensures that even the most complex diagrams constructed and simulated in Simulink perform as expected once run on the target hardware. All Simulink models for which code can be generated by the RTW can be implemented in CSDE, this includes continuous-time, discrete-time as well as hybrid models.

The Real-Time Interface Software (RTIS) consists of three parts :

(i) PC32Lib, a library containing blocks to incorporate hardware access from within a Simulink block diagram

(ii) Template make file to govern the compilation, source code need to make RTW generated code into an executable and a .dll library to implement external mode communications from Simulink

(iii) Display, a standalone Windows executable to facilitate data uploads from the PC32 target hardware for logging and visualisation

These parts will be discussed in more detail in the following sections.

A.2.5 About the User's Guide

Section A.3 explains the installation of the CSDE

Section A.4 gives a quick introduction to the environment by means of a few simple examples

Section A.5 gives more detailed information on blocks contained in PC32Lib

Section A.6 explains the use of the Display utility

Section A.7 discusses the various modes of operation and the limitations of the CSDE

A.3 Installation

A.3.1 Hardware Requirements

The PC32 DSP Controller is designed to fit into a ISA expansion slot of any IBM/PC compatible machine. However, the software tools require at least a Windows 95 system. Any PC capable of running Windows 95 should be able to support the CSDE, however it is strongly recommended to use a Pentium machine with 32 Mbytes of RAM, but preferably more (e.g. 64 Mbytes).

The installation of the PC32 is as simple as that of any PC expansion card, the PC32 Hardware Manual from Innovative Integration should be consulted before fitting the card into a vacant ISA slot.

For the purposes of the examples in section A.4 the jumper settings on the PC32 should be initially left in their factory default setting :

(i) A/D pair 0 and 1 triggered by software

(ii) D/A 0, 1, 2 and 3 triggered by timer 0

(iii) JP 5 should be set to IRQ 5

The exact function and usage of these (and other) jumpers will be explained in more detail in section A.5 when the components of PC32Lib are discussed.

A.3.2 Software Requirements

The CSDE requires a number of software packages to be installed on the host computer. The CSDE might also function correctly with other versions or releases of these packages but only the following set has been fully tested :

(i) Windows 95

(ii) Texas Instruments ANSI C Compiler Version 4.70

(iii) MATLAB Version 5.2.1.1420
Simulink Version 2.2.1
Real-Time Workshop Version 2.2.1

(iv) Innovative Integration Device Driver and Peripheral Library Version 1.18

(v) RTIS shipped with this manual

A.3.3 Installation Instructions

For the installation of Texas Instruments, Math Works and Innovative Integration software please refer to the respective documentation shipped with these products.

To install the RTIS simply run the setup utility provided on the install disk and follow **the** instructions provided.

It is recommended that during installation of all components the default directory structure is used as far as possible. Should some of the directories be changed RTIS template make file **will** need to be modified to reflect the new locations. In the next sections the following directory structure is expected :

Texas Instruments

(i) C:\fltc\

MathWorks

(ii) C:\MATLAB\

Innovative Integration

(iii) C:\pc32cc\

Real-Time Interface Software

(iv) C:\MATLAB\RTW\c\ii\
(v) C:\MATLAB\RTW\ext_mode

Before proceeding to the next section it is recommended that the correct installation of the PC32 DSP Controller hardware and drivers is verified. An easy way to do this is to run one of the demo programs included in the Innovative Integration package.

A.4 Getting Started

This section provides a quick glance at the CSDE. Since most users will already be familiar with the Simulink GUI, the examples presented should prove straightforward.

Three simple, ready to use examples are presented to demonstrate how Simulink block diagrams is compiled and transferred to the PC32 DSP Controller. The following demos give a chance to experiment with the I/O block as well as the various modes of operation.

A.4.1 Simple Example

To run the first simple model:

(i) Start MATLAB
(ii) Type *pc32_demol*

This opens the block diagram for the first demo. Fig. A.3 shows the Simulink block diagram for the *pc32_demol* example.

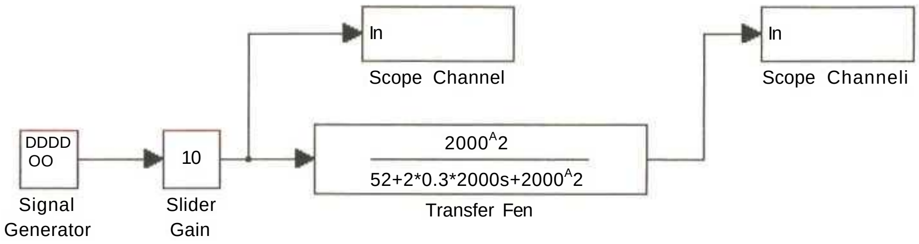


Fig. A.3 : Simulink block diagram for*pc32_demol* example

The block diagram contains a signal generator set to provide a square wave at lOOHz, a slider gain, a second order continuous transfer function and two scope channel upload blocks.

To get things working on the DSP card go to Tools menu and select RTW Build. The build/compile/download process can be followed in the MATLAB main window. When that is complete select Start from the Simulation menu or push the play button on the toolbar of your *pc32_demol* window. At this point in time the code generated from the block diagram begins execution on the DSP target.

Start the Display.exe utility, enter 2 as the number of channels and push OK. Two windows appear - CH #0 and CH # 1. CH #0 displays the square wave output from the signal generator and CH #1 displays the result of putting it through a second order system.

Now, going back to the block diagram, experiment with changing the values of the slider gain, damping factor of the transfer function and settings of the signal generator. The effect of these changes can be seen immediately in the CH #0 and 1 windows.

By double-clicking on the CH # display windows their properties can be adjusted, the OK button on Display's main dialog needs to be clicked each time a simulation is restarted, the various options are discussed in section A.6 when we present the Display utility.

Timer 0 onboard the 'C32 processor is used to generate the base sampling rate for the diagram. To experiment with other sampling rates go to Tools | RTW Options | Solver and adjust the step size. The diagram needs to be recompiled each time the step size changes.

A.4.2 Example with A/D and D/A

In the previous example the input signal was generated on board the DSP card, in a real-world application we are more likely to process some signal captured through a A/D channel. In MATLAB window type *pc32_demo2.* Fig. A.4 shows the block diagram for this example.

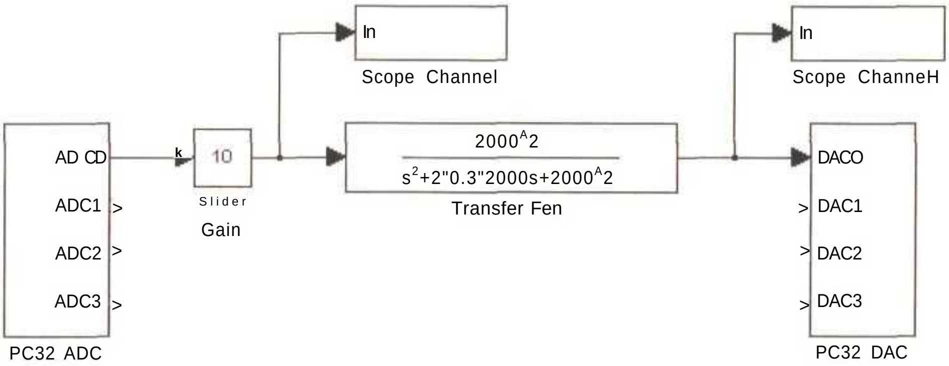


Fig. A.4 : Simulink block diagram for *pc32_demo2* example

Essentially this is *pc32_demol* revisited, with the exception of the signal generator being replaced by an A/D block and the addition of a D/A block. Before compiling and starting this diagram, connect a signal generator to A/D 0 connector on the PC32 card (pin 26 and 6 on the D type connector) and set it to provide a 100 Hz square wave with a 5 volt amplitude. The A/D channels of the PC32 Controller take a maximum of ± 10 V, so care should be take not to exceed these levels. Optionally a oscilloscope can be connected to D/A 0 (pin 16 and 35).

Select RTW Build and after the build process is complete start the simulation. Once again select two channels on the Display utility. Adjusting the signal generator affects the plots in both CH #0and#l windows.

A.4.3 Interrupt Support Example

The final example can be accessed by typing *pc32_demo3* in the MATLAB window, Fig. A.5 shows the block diagram.

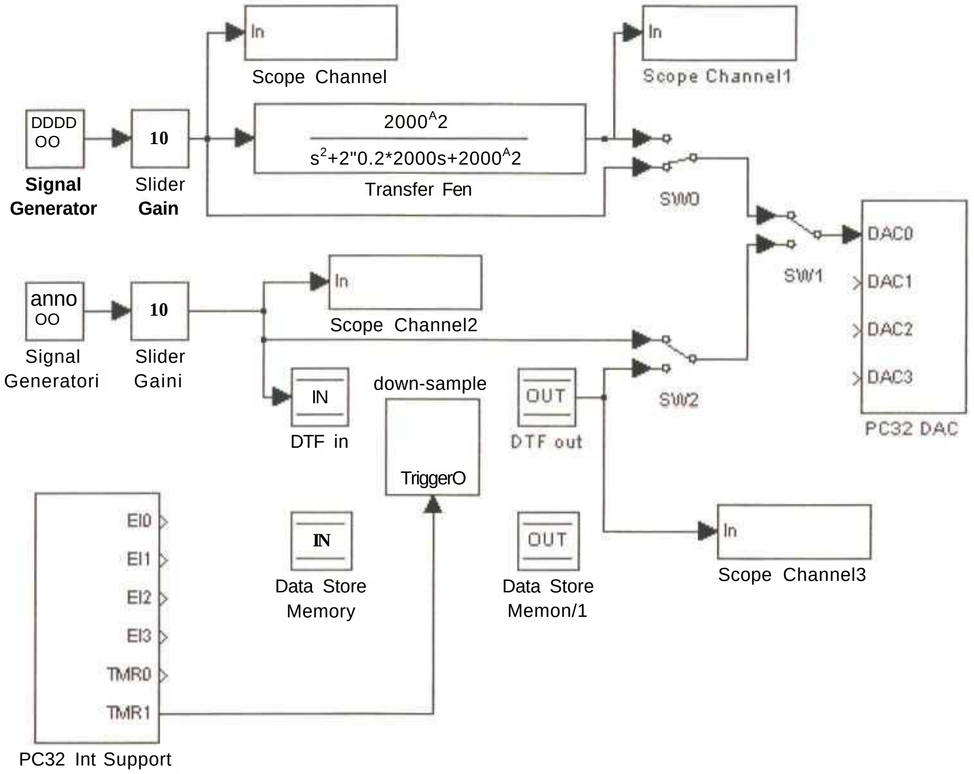


Fig. A.5 : Simulink block diagram for*pc32_demo3* example

At first glance we can see the familiar second order system fed by a signal generator. The new blocks, despite their numbers, perform a very simple task - down-sampling. A second signal generator provides a 100 Hz sine wave which is then passed, through a memory store element, to the down-sample block. All the down-sample block does is copy from the IN memory store to OUT. However, since it is triggered at a different frequency to the rest of the diagram, the signal that passes through it is down-sampled. The down-sample block is triggered by Timer 1 interrupt at 1.1 kHz, while the rest of the diagram is executed by Timer 0 at 5 kHz. Thus effectively the sine wave is down-sampled from 5 kHz to 1.1 kHz and the effect can be observed in CH #2 and 3 windows.

If there is an oscilloscope attached to D/A 0 the various signals can be observed by changing the configuration of switches SWO, 1 and 2.

A.5 PC32Lib

This section describes the custom block contained in the *PC32Lib* blockset for Simulink. These blocks allow low-level hardware access to the PC32 Controller from within a block diagram. Fig. A.6 shows the collection of blocks in the *PC32Lib*. To bring up the library simply type *PC32Lib* in MATLAB window.

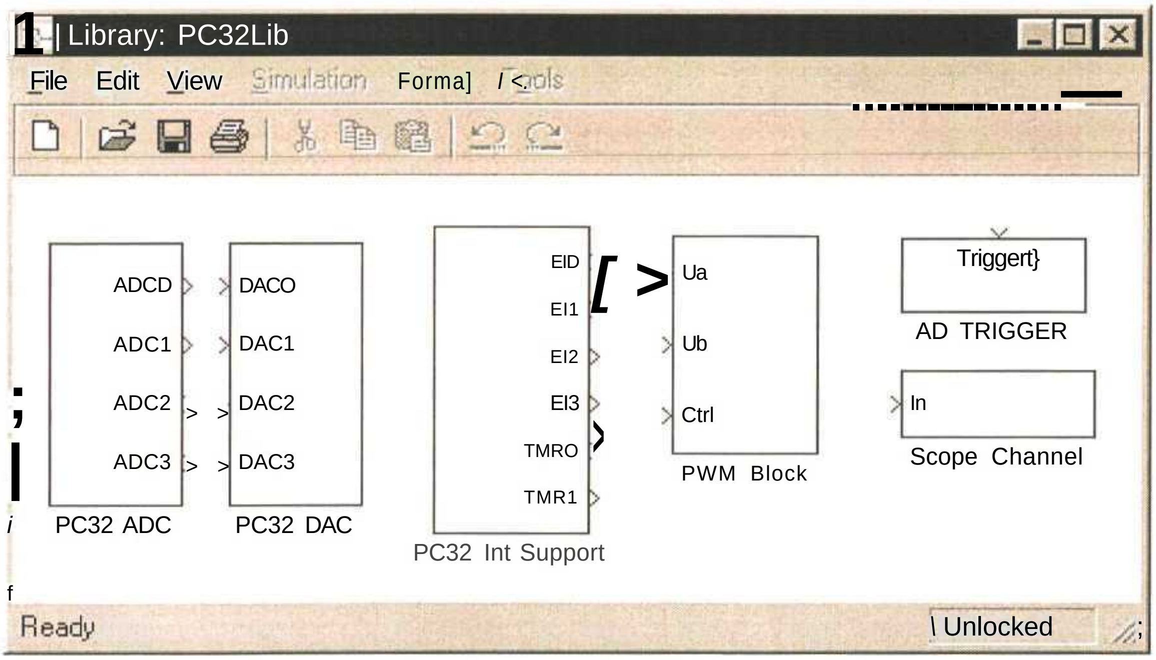


Fig. A.6 : Contents of PC32Lib

A.5.1 PC32 ADC Block

This block allows access to the on-board A/D converters of the PC32 Controller. When making use of this block it is important to understand the way in which the Innovative Integration board triggers the A/D converters.

ADC 0 and 1 form ADC pair 0. ADC 2 and 3 form pair 1. Each pair can be triggered by one of the following :

(i) A software memory write
(ii) Signal on TCLKO
(iii) Signal on TCLK1

JP 6 and JP 5 control the trigger sources for ADC pair 0 and 1 respectively. Irrespective of the jumper setting a memory write in software always triggers a conversion. Setting the jumper to position 1-2 selects the trigger source to be TCLKO, position 2-3 selects TCLK1. Depending on a software setting signal for each TCLK can come from either :

(i) an external TTL compatible source
(ii) a on-board timer

The source for each TCLK is selected in the PC32 Int Support block (see below). If no PC32 Int Support block is present on the diagram the default setting is for both TCLKO and TCLK1 to be driven by respective on-board timers.

If either, or both JP5 and 6 are completely removed the respective ADC pair can only be triggered by a software memory write. This option is discussed below when the AD Trigger block is introduced.

A.5.2 PC32DAC Block

This block allows access to the on-board D/A converters of the PC32 Controller. When making use of this block it is important to understand the way in which the Innovative Integration board triggers conversions on the D/A channels.

Each DAC channel can be triggered separately by one of the following :

(i) A software memory write
(ii) Signal on TCLKO
(iii) Signal on TCLK1

Jumpers JP 3,4,11 and 10 control trigger sources for DAC 0,1,2 and 3 respectively. In contrast to the A/D triggering each jumper has three possible positions and if it is completely removed the respective DAC channel is net triggered at all. Once again the source for TCLK signals can be selected in the PC32 Int Support block to be one of:

(i) an external TTL compatible source
(ii) a on-board timer

The default source, if PC32 Int Support block is not used, is the respective on-board timer.

The software memory write triggering is build into the PC32 DAC block. In other words, if a jumper is in the DSP position the corresponding D/A channel will be triggered each time a new value is passed to the PC32 DAC block.

A.5.3 PC32 Int Support Block

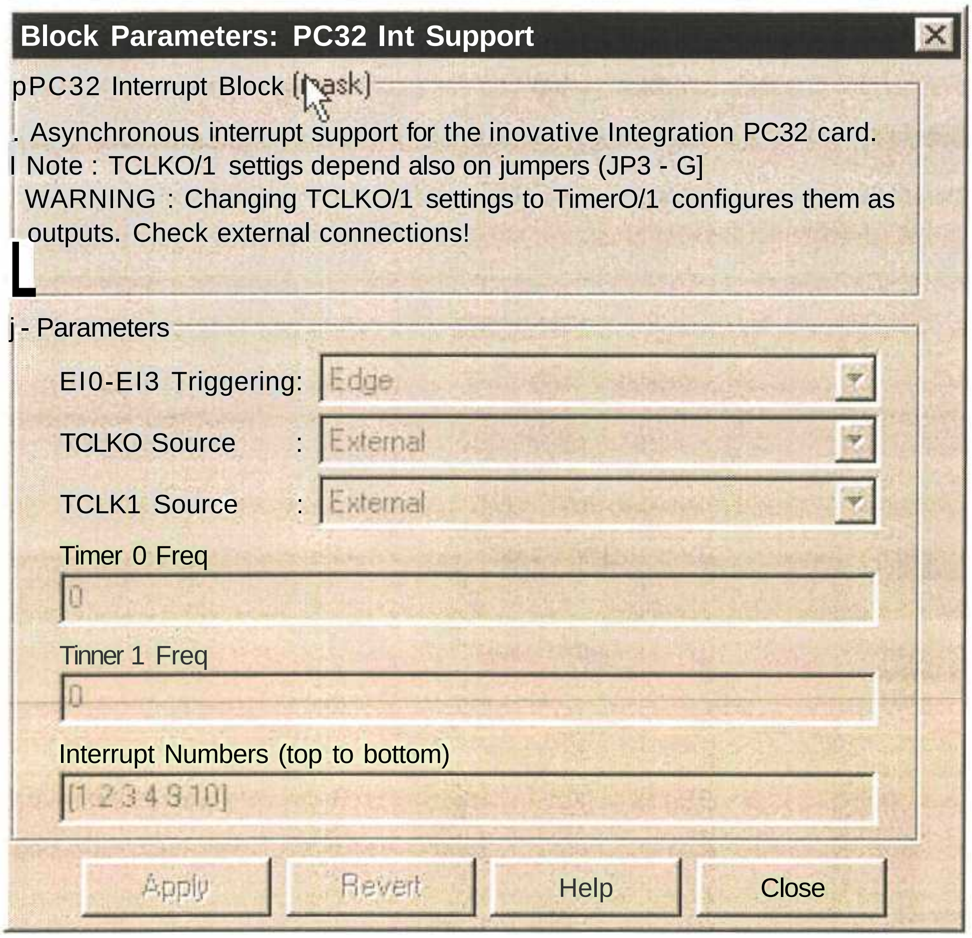
The PC32 Int Support block is used to perform a number of functions. Double-clicking on it brings up the menu shown in Fig. A.7. Let us go through the various options.

EIO - EI3 Triggering, this allows the user to specify how the 'C32 processor responds to external interrupts. In all but the most specific applications the default setting will be adequate.

TCLKO and TCLK1 source, allows to specify whether the signal is drawn from one of the on-board timers or from an external connector. When the default setting is used both TCLKO and 1 are taken from an external pin (JP 15 or U34 memory expansion connector). Care should be taken when changing those settings since the configuration of these pins changes from input to output.

TimerO Freq and Timerl Freq, allow to set frequencies for the on-board TimerO and 1 respectively. TimerO can be used to generate the base sample time for a block diagram (see section A.7 for details). When TimerO is thus engaged, the corresponding menu in Fig. A.7 is disregarded.

Interrupt Numbers, set the numbers of interrupts available form the outside of a PC32 Int Support block. Default setting is for interrupt 1 to 4 (EIO to EI3) and interrupt 9 and 10 (Timer 0 and 1 interrupts). If any additional interrupts need to be trapped Texas Instruments 'C32 Users's



Block Parameters: PC32 Int Support

pPC32 Interrupt Block (mask)

Asynchronous interrupt support for the inovative Integration PC32 card.
Note : TCLKO/1 settigs depend also on jumpers (JP3 - G]
WARNING : Changing TCLKO/1 settings to TimerO/1 configures them as outputs. Check external connections!

Parameters

EI0-EI3 Triggering: Edge

TCLKO Source : External

TCLK1 Source : External

Timer 0 Freq

0

Tinner 1 Freq

0

Interrupt Numbers (top to bottom)

[1 2 3 4 9 10]

Fig. A.7 : Mask for the PC32 Int Support block

Guide should be consulted for the relevant numbers. Should the Interrupt Numbers array be changed, it will be necessary to go under the mask for the PC32 Int Support block and make relevant changes. For details on this topic see Math Works' Using Simulink.

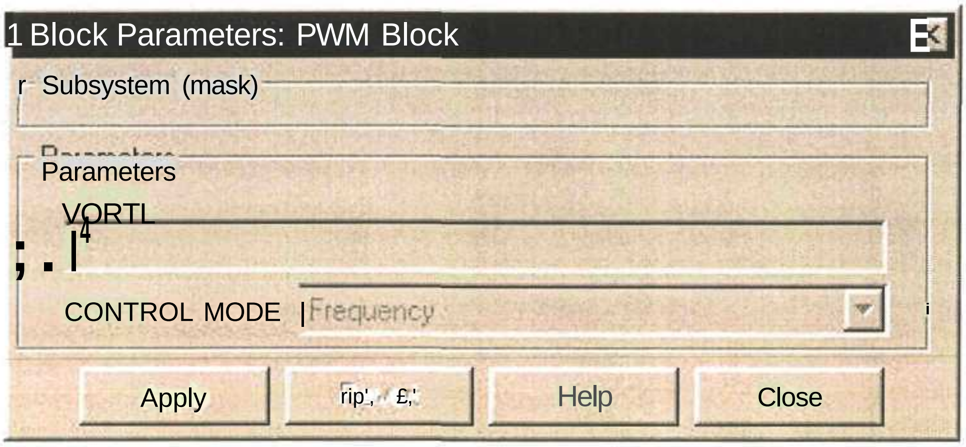
The interrupts specified in Interrupt Numbers array are available the outside of the PC32 Int Support block. These outputs can be used to drive inputs of standard Simulink triggered subsystems. An example of such subsystem is the AD Trigger block.

As a default the external interrupts Ell and EI2 are triggered by the end-of-conversion signals from ADC pair 0 and 1 respectively. JP17 controls the sources for these interrupts.

External interrupt EI3 is reserved by the CSDE for communication from the host PC and should not be used for other purposes.

A.5.4 PWM Block

This block requires a PWM/Tacho expansion card to be connected to the PC32 Controller. For information on the operation of the PWM/Tacho card please refer to the relevant documentation shipped with the card.



Block Parameters: PWM Block

Subsystem (mask)

Parameters

VORTL

4

CONTROL MODE Frequency

Fig. A.8 : Mask for the PWM Block

Fig. A.8 shows the mask for the PWM block. VORTL specifies the frequency of operation of the PWM generator as follows :

(i) 1 = 20 kHz

(ii) 2 =10 kHz

(iii) 3 = 5 kHz

(iv) etc.

Control Mode can be changed from Frequency to Angle depending on the control algorithm employed.

A.5.5 AD Trigger Block

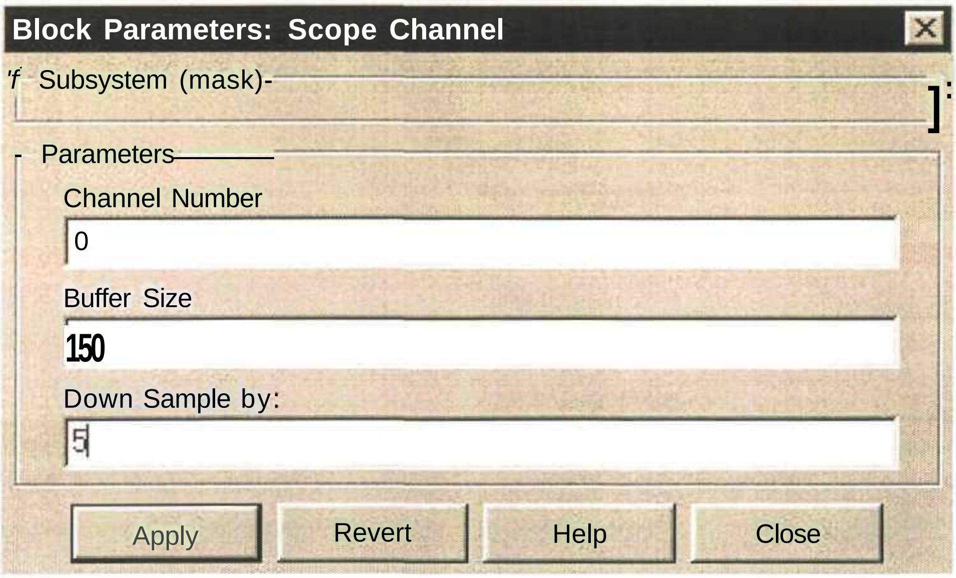
This block causes a conversion to be triggered on both ADC pairs of the PC32 Controller. It is designed to be driven from an output of the PC32 Int Support block.

To synchronise sampling to an external trigger signal feed this signal to EIO. Use EIO output of the PC32 Int Support block to drive the AD Trigger block and use the end-of-conversion signals on Ell and/or EI2 to trigger subsystems which require data from the A/D converters.

This method is particularly useful when the external trigger signal is not directly compatible to hardware trigger the A/D converters (i.e. Pulse width too long), but still provides an acceptable negative edge to trigger an external interrupt.

A.5.6 Scope Channel Block

This block provides a buffer for uploading a channel of data form the PC32 Controller to the Display utility on the host PC. Presently up to 10 such block can be used in a single block diagram. Fig. A.9 shows the mask for the first Scope Channel block. Each such block placed on a block diagram should have a number between 0 and 10 entered under Channel Number. The CSDE does not check for validity of users choice of numbers. Should two (or more) blocks point to the same channel the associated window of the Display utility will show a mixture of the blocks data, which in most cases will be meaningless.



Block Parameters: Scope Channel

Subsystem (mask)

Parameters

Channel Number: 0

Buffer Size: 150

Down Sample by: 5

Fig. A.9 : Mask for the Scope Channel block

Buffer Size specifies the size of **the** packets sent **to the host for a particular block. As** a **rule of** thumb blocks with faster sampling rates should have larger buffers. When a number of Scope Channel blocks is **used** and only some operate correctly it is most likely that the total of **all** buffer sizes exceeds the memory allocated for the purpose. In such case the size of buffers should be decreased.

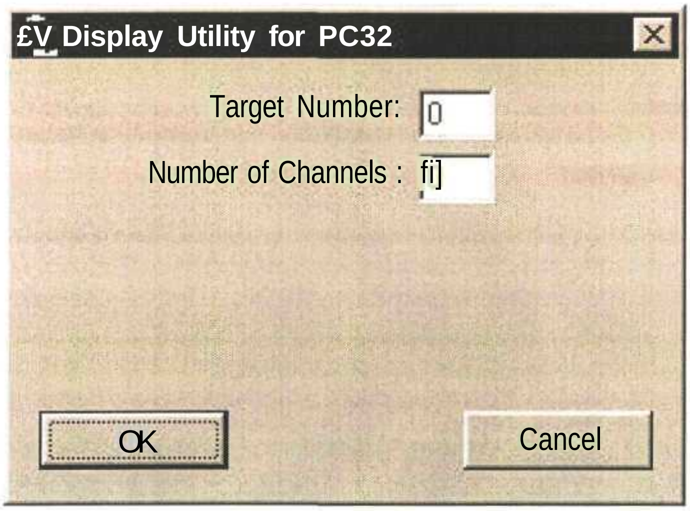
The Down Sample option allows for adjustment of the sampling frequency. The input signal to the Scope Channel block will be sampled at the block's execution frequency divided by the value of Down Sample.

A.6 Display Utility

The Display utility is a stand-alone Win95 application which allows to plot graphically data uploaded from the PC32 Controller in real-time. Up to 10 channels can be captured simultaneously.

A.6.1 Running Display

To start the utility run *Display.exe* from C:\MATLAB\RTW\c\ii. The main dialog of the utility is shown in Fig. A. 10.


£V Display Utility for PC32

Target Number: 0

Number of Channels : fi]

Fig. A. 10 : Main window of the Display utility

The assumption is that a Simulink block diagram containing a Scope Channel block (or blocks) **and** with the UPLD=YES option specified was build and started. The default setting is for Target Number 0, this should not be changed unless more than one PC32 Controller card is installed in the host system. The number of Scope Channel blocks used in the corresponding Simulink block diagram should be entered under Number of Channels. When OK is pressed the main dialog is minimised and the specified number of channel windows appear. Every time a Simulink block diagram is started the OK button will have to be pressed.

A.6.2 Working With Channel Windows

Each channel window has a caption - CH# 0, CH# 1 etc. Those numbers correspond to numbers entered **in** each Scope Channel **block** in the Simulink block diagram. A typical channel window is shown in Fig. A. 11. Settings specific to each window can be modified by double-clicking of the window to bring up a properties dialog shown in Fig. A. 12.

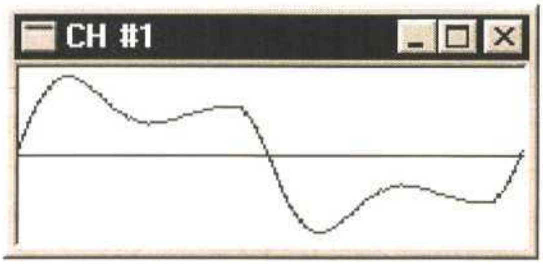
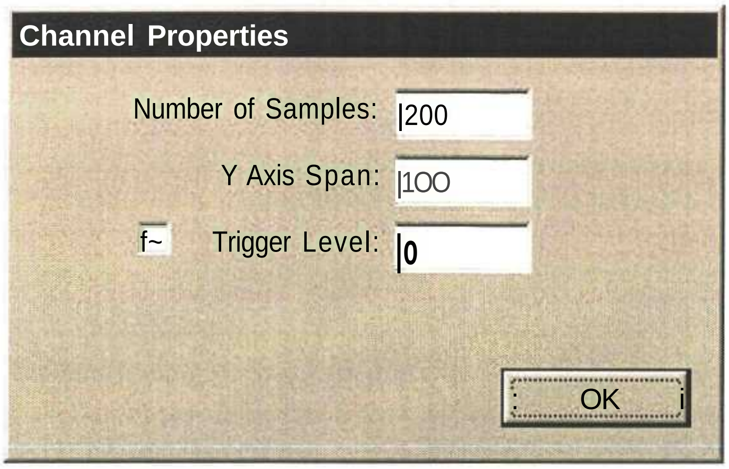


Fig. A. 11 : Typical Channel Window



Number of Samples: 200

Y Axis Span: 100

☑ Trigger Level: 0

Fig. A. 12 : Channel Properties dialog

Default settings are :

(i) Number of Samples = 200

(ii) Y Axis Span = 100

(iii) Trigger Level = 0

(iv) No Triggering (check-box clear)

Number of Samples specifies the total number of sample points visible in the window. If this number is less then the windows physical x-size, adjoining samples are joined by straight lines. Display does not allow the opposite to happen, however. If the x-size of the window is decreased below the specified number of samples, Display automatically decreases this setting.

Y Axis Span is the highest value displayed on both positive and negative y-axis. This setting does not change when a window is resized.

Trigger Level allows to set the triggering for a particular window. For waveforms which cross the x-axis a number of times in each period it might be necessary to increase the trigger level to avoid re-triggering. By default triggering is not enabled, to enable it select the check-box.

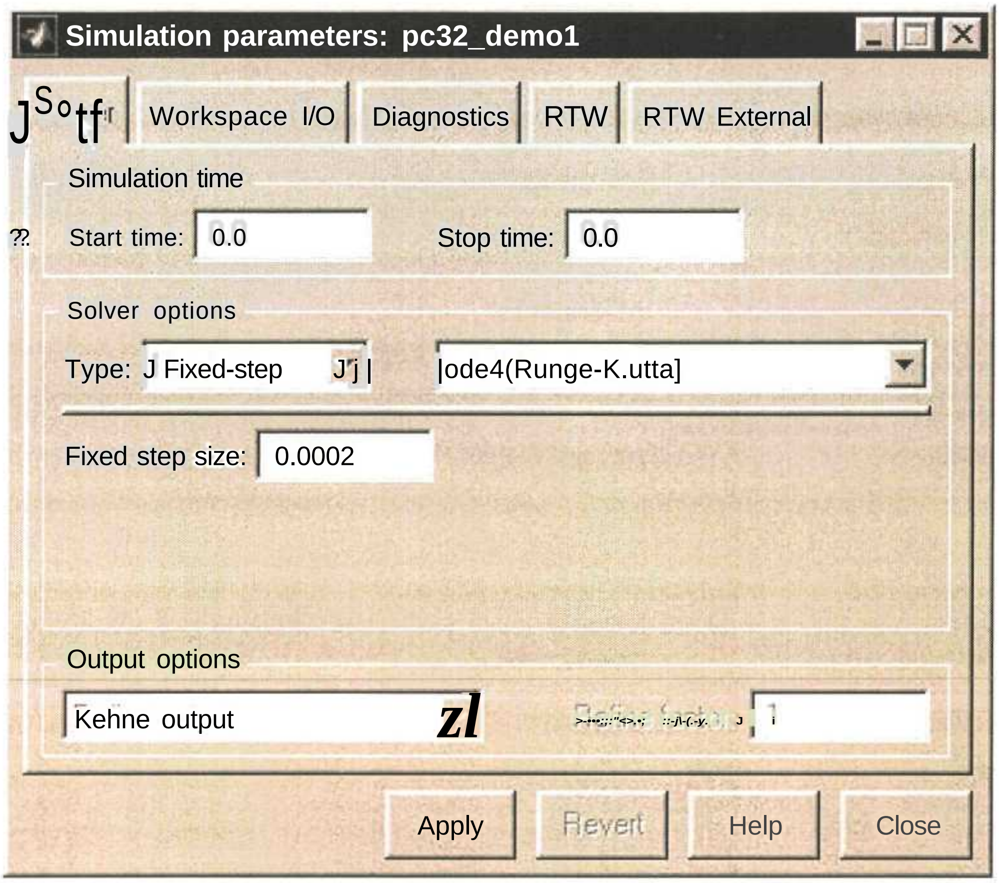
A.7 Operation of the CSDE

As described in preceding sections, the CSDE is a system based around the MATLAB/Simulink environment. Any new design starts with a Simulink block diagram. There are a few limitations as to which blocs may be used (see below) but most standard Simulink blocks (including toolboxes) and blocks in the PC32Lib will function as expected.

A.7.1 Setting Up a Simulink Block Diagram

First step in any new project will most likely be the design and initial simulation of a standard Simulink diagram. The transition from simulation to actual real-time implementation involves a few steps.

Firstly under Simulation Parameters | Solver, as shown in Fig. A. 13, the Solver options have to be set to Fixed-step. If the block diagram contains any continuous-time integration blocks an integration algorithm needs to be specified. A step size has to be set if the block diagram makes



Simulation time

Start time: 0.0 Stop time: 0.0

Solver options

Type: Fixed-step |ode4(Runge-K.utta]

Fixed step size: 0.0002

Output options

Kehne output 1

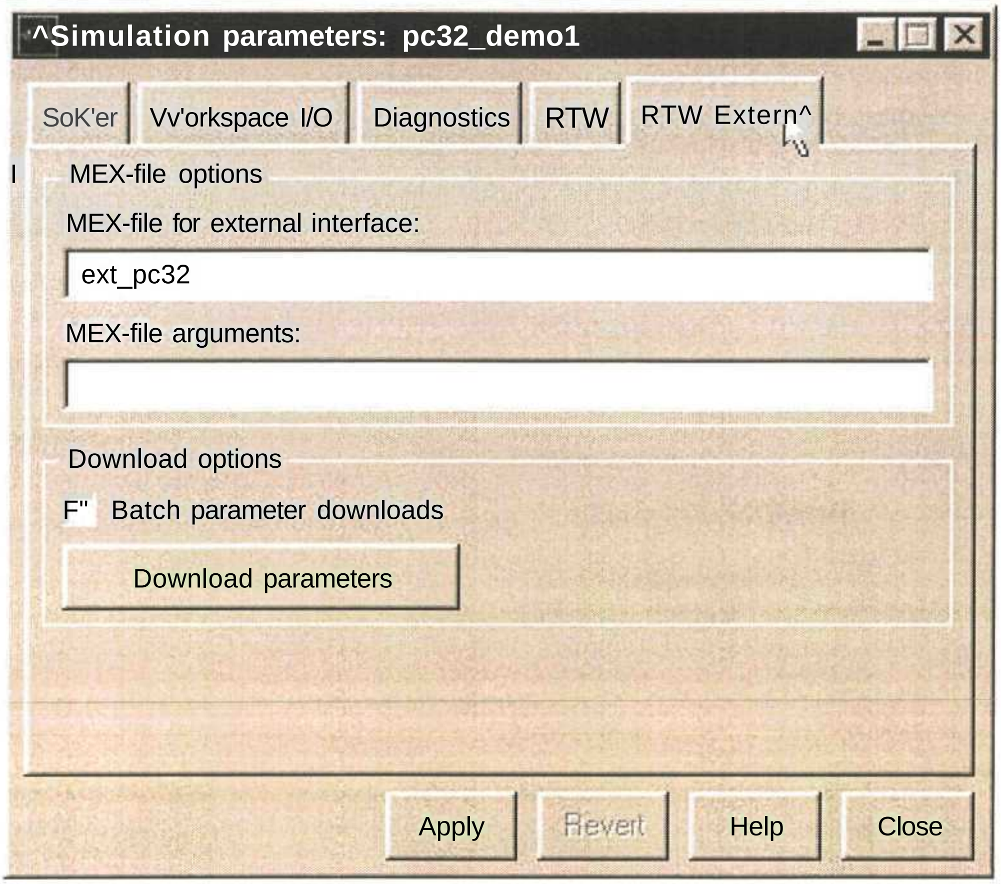
Fig. A. 13 : Simulation Parameters | Solver

use of Timer 0 to generate the base sampling rate (see below). The values of Start and Stop Time are ignored by the RTW.

The name for the external mode .dll needs to be entered in Simulation Parameters | RTW External as in Fig. A.M.

Final changes have to be done in Simulation Parameters | RTW, Fig. A. 15. System Target file is *ii.tlc* and the Template Make File is *pc3 2. tmf.* A number of options can be included behind the make command *make_rtw,* these options have to be specified exactly as listed below :

(i) TMR0=YES. This option sets Timer 0 to generate the base sampling rate for the block diagram at the rate specified under Simulation Parameters. When Timer 0 is used in this way it cannot be used by the PC32 Int Support block. If this option is not used any blocks not contained within triggered subsystems will execute as



MEX-file options

MEX-file for external interface: ext_pc32

MEX-file arguments:

Download options

F" Batch parameter downloads

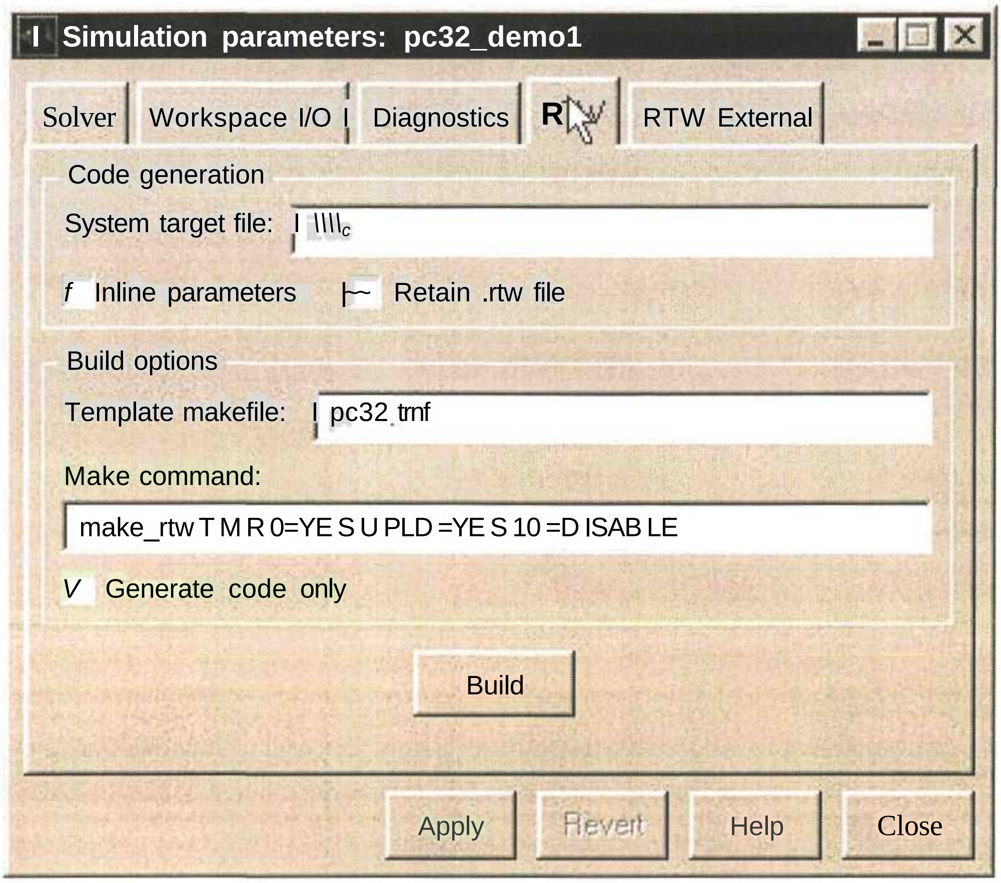
Fig. A. 14 : Simulation Parameters | RTW External

background tasks and their execution frequency cannot be guaranteed. Default is not to use Timer 0.

(ii) UPLD=YES. This option needs to be included if use is made of the Scope Channel block(s) to capture data. If it is omitted any Scope Channel blocks on the diagram will not be serviced. Default is not to service uploads.

(iii) IO=ENABLE. This option is included for debugging purposes. Diagrams compiled with this option will need to be downloaded using Innovative Integration's Terminal program. With this option set any diagram executing on the PC32 Controller prints text to the Terminal window. If Terminal utility is not used when this option is set programs will hang on the DSP hardware. Default is not to give output to the Terminal window.

If the syntax for any of the options is changed or it is omitted all together the default setting for that particular option is used.



System target file: I \\\\c

f Inline parameters |~ Retain .rtw file

Template makefile: I pc32 trnf

Make command:

make_rtw T M R 0=YE S U PLD =YE S 10 =D ISAB LE

V Generate code only

Fig. A. 15 : Simulation Parameters | RTW

A.7.2 Limitations of CSDE

Due to technical reasons some Simulink blocks might not function correctly when compiled to run on real-time hardware :

(i) Any Simulink block which depends on absolute time - e.g. Sine Wave. This is due to the overflowing of the time register if the code is run for extended periods of time.

(ii) All Simulink Sinks blocks and Source blocks reading from files or **MATLAB** workspace. This functionality is implemented using the Scope Channel blocks.

Appendix B

B.I Dual Port Memory Access

This appendix details the protocol used by the CSDE during bi-directional communication between the host PC and the target PC32 card. All data exchange takes place via the DPRAM. To avoid conflicts interrupts are used to arbitrate access to the shared memory regions.

B.I.I DPRAM Allocation

The DPRAM allocation map is shown in Fig.B. 1. The only exception to this allocation is during initial checksum verification when locations 0x0001 to 0x0004 are temporarily used to pass the four checksum values from the host to the target. Location 0x0000 can be read and written by both the host and the target, all other locations are uni-directional as indicated in the **Read** and **Write** columns in Fig. B.I.

Offset	Function	**Write**	**Read**
0x0000	Bits 0 to 7 - host command to target, or target response to host Bits 8 to 31 - command confirmation padding	H/T	H/T
0x0001	Bits 0 to 31 - parameter offset	H	T
0x0002	Bits 0 to 31 - parameter value	H	T
0x0003	Bits 0 to 2 - current target status Bits 3 to 31 - unused	T	H
0x0004	Bit 0 - current host status Bits 1 to 31 - unused	H	T
0x0005 ... 0x0400	Data upload packets from target to host	T	H

Fig. **B. 1** : Memory usage **during normal** operation (H = host, T = target)

B.1.2 Host Commands

To communicate a command to the target the host places the command number within the least significant byte at location 0x0000 and pads the remaining 3 bytes with the value OxAAA. Interrupt EI3 is then signalled to target. The padding is done to add a degree of security to the communication. In the event of a spurious interrupt the target will not process a command unless the security padding is not correct. The possible host commands are listed in Fig. B.2.

Command No.	Description
1	Start Execution - The Simulink generated code is initialised, interrupts installed, and host writes its status to location 0x0003
2	Parameter Update - The parameter offset in SimStruct and its updated value are placed in locations 0x0001 and 0x0002 respectively.
3	Suspend Execution - The model execution is suspended and the termination functions are called. Model can be re-started by command 1.
4	Checksum Verification - Four checksum values passed in locations 0x0001 to 0x0003 are compared against copies embedded in the target executable.
5	Start Data Upload - The upload queues are cleared and the uploads are started.
6	Suspend Data Upload - The uploading of data is stopped. It can be restarted by command 5.

Fig. B.2 : List of host commands

B.I.3 Target Response

After receiving and processing a host command, the target places its response in the least significant byte at location 0x0000 and pads it with OxABC. If processing is successful the command number is echoed, a response of 0 signals failure. After sending a command the host polls DP RAM location 0x0000 until the correct padding value is seen.

B.1.4 Target Status Word

During processing of the **Start Execution** command the target updates its status word at location 0x0003 to signal to host which options were enabled. Fig. B.3 lists the bit allocation in the status word.

Bit	Description
0	Status of the debug printouts (IO_YES define). A 1 signals that debug info is being printed to the terminal emulator, a 0 means that all printouts are disabled.
1	Usage of the on-board timer 0 (TMR0_YES define). A 1 means that timer 0 is used to generate the base sampling rate for the model, a 0 signals that timer 0 is available for other uses.
2	Status of the data uploads (UPLD_YES define). A 1 signals that compiled code supports data uploads, a 0 means that all data upload functionality was excluded during compilation.
3..31	unused

Fig. B.3 : Target Status Word contents

B.I.5 Host Status **Word**

Bit 0 at location 0x0004 signals whether the Display utility on the host PC is ready to accept data uploads. A 1 signals a ready condition. A 0 signals that Display is either not running or it is still busy reading packets out form DPRAM. The remaining bits (1 to 31) in the status word are not used.

B.1.6 Packetized **Data** Uploads

The continuous DPRAM block from location 0x0005 to 0x0400 is reserved for data uploads from the target to host. When enough buffered data is available on the target, the host status word is polled until it is 1. Next packetized data is placed in the DP ARAM in the manner shown in Fig. B.4 and interrupt IRQ5 is signalled to the host PC.

DPRAM offset:	**Description**:
0x0005	Channel Number N
0x0006	Packet Size X_N
0x0007	Packet Data
0x0007 + X_N	Channel Number M
0x0007 + X_N + 1	Packet Size X_M
0x0007 + X_N + 2	Packet Data
0x0007 + X_N + 2 + X_M	Termination 999

Fig. B.4 : DPRAM usage for data uploads

Appendix C

C.I Hardware Registers

This appendix lists the functionality of the timer configuration registers of the TMS320C32 processor, and all registers of the PMB 1/87 ASIC.

C.I.I 'C32 Timer Control Registers

Timer control registers are memory mapped in the general address space of the 'C32 processor as shown in Fig.C. 1. The function of the various flags in the Timer Global Control Registers is explained in Fig. C.2.

Register	Timer 0 address	Timer 1 address
Timer Global Control	0x808020	0x808030
Timer Counter	0x808024	0x808034
Timer Period	0x808028	0x808038

Fig. C.I : Timer Control Register memory locations

C.1.2 PMB 1/87 Registers

All communication with the PWM ASIC is conducted via two memory mapped locations. Assuming the PWM/Tacho card is set to the default settings (Decodes 0 and offset 0x000) the memory mapping is shown in Fig. C.3. Writing to the control word sets various internal functions as well as the read and write addresses for internal registers, as shown in Fig. C.4. During reading the status word contains the IC's status information as shown in Fig. C.5.

The various registers available for writing are listed in Fig. C.7. Registers which can be read back are shown in Fig. C.6.

Bit	Name	Rst. Val	Description
0	FUNC	0	FUNC controls the function of TCLK. If 0, TCLK is a general I/O pin, if 1 TCLK is a timer pin
1	I/O	0	If TCLK is a I/O pin then I/O = 0 configures it as an input, 1 makes it an output
2	DATOUT	0	Data for TCLK in output I/O mode, can be used as input to the timer
3	DATIN	?	Data input for TCLK, write has no effect
4.. 5	Reserved	0	Read as 0
6	GO	0	GO resets and starts the timer. If 1 and timer is not held, counter is reset,started and GO is cleared. GO = 0 has no effect.
7	HLD	0	Counter hold signal. If 0 counter is stopped, if 1 timer carries on running.
8	C/P	0	Clock/Pulse mode control
9	CLKSRC	0	Selects source for the timer. If 1 the internal clock is used. If 0 TCKL pin drives the timer.
10	INV	0	Inverter. If 1 timer input and output signals are inverted. If 0 no effect.
11	TSTAT	0	Timer status. Sends an interrupt to the CPU on each transition from 0 to 1. A write has no effect.
12 ..31	Reserved	0	Read as 0

Fig. C.2 : Timer Global Control Register

Address	Function
0x819800	Data Word
0x819801	Status Word (during a read) Control Word (during a write)

Fig. C.3 : Memory mapping of the PMB 1/87 ASIC

Bit	Name	Description
0	EIN	EIN=1 enables pulse calculation and output, 0 disables
1	IINT	IINT=0 selects external control over current polarity (pins 11..13), 1 internal control via bits IINT1 .. IINT3
2..4	IINT1 ..IINT3	Current polarity in inverter branch 1 .. 3 (l=positive current)
5	Not used	
6	TESTFL	Test flag, should be left at 0
7	BUS 16	BUS 16=1 selects 16-bit bus mode, 0 selects 8-bit bus mode
8.. 11	WAD0 .. WAD3	Write address used by next write to Data Register, automatically increments after each write
12.. 13	RAD0 .. RADl	Read address used by next read from Data Word.
14	RDSTART	Start read cycle.
15	Not used	

Fig. C.4. Control Word during a write

Bit	Name	Description
0	WRFLAG	Data may only be written to registers when WRFLAG = 0
1	RDFLAG	RDSTART = 0 means data read cycle is complete.
2	CALCFLAG	Processing flag indicates when an internal processing cycle is in progress. Read cycle can be started when CALCFLAG = 0
3.. 15	Not used	

Fig. C.5 : Status Word during a read

RAD 1 .. 0	Name	Description
00(0)	PHI0	Phase angle, lower half
01(1)	PHI1	Phase angle, upper half
10(2)	UO	Voltage value
11(3)	Not used	

Fig. C.6 : Readable registers

WAD3 .. 0	Name	Description
0000 (0)	UA	Ua voltage component
0001(1)	UB	Ub voltage component
0010(2)	PHI1	Phase angle, upper half
0011 (3)	DPHI1	Frequency, upper half
0100(4)	PHIO	Phase angle, lower half
0101 (5)	DPHIO	Frequency, lower half
0110(6)	PHIADD	Difference in phase angle
0111 (7)	Not used	
1000(8)	TAUS	Turn-off time
1001(9)	TTOT	Dead-band time
1010(10)	TMIN	Turn-on time
1011 (11)	VORTL	Switching frequency scale value
1100(12)	TSTART	Start of processing cycle
1101 (13)	Not used	
1110(14)	Not used	
1111 (15)	Not used	

Fig. C.7 : Writeable control registers

Appendix D

D.I Template Make File

This appendix list the template make file specifically customised to generate code for the PC32 platform.

D.I.I Source Listing of pc32.tmf

```
#************************************************************************
#------------------------- General Defines ------------------------------

SYS_TARGET_FILE  = ii.tlc
MAKE             = gtnake
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      = j >MODEL_MODULES<|
MODEL_MODULES_OBJ = j >MODEL_MODULES_OBJ<|
MAKEFILE           =  |>MAKEFILE_NAME<|
MATLAB_ROOT       = j >MATLAB_ROOT<|
MATLAB_BIN         =  j>MATLAB_BIN<|
S_FUNCTIONS       = j >S_FUNCTIONS<|
S_FUNCTIONS_OBJ    =  j>S_FUNCTIONS_OBJ<|
SOLVER             =  I>SOLVER<|
SOLVER_OBJ         =  |>SOLVER_OBJ<|
NUMST              =  I>NUMST<|
TID01EQ            =  |>TID01EQ<|
NCSTATES           =  I>NCSTATES<|
BUILDARGS          =  j>BUILDARGS<|
COMPUTER           =  I>COMPUTER<|

#_____________________________II pc32 Definitions ___________________------
#

BOARDJTYPE      = PC32
DSP_FAMILY       =30
COMPILER        = TI_FPC

II_ROOT     = $(MATLAB_ROOT)\rtw\c\ii
II_COMPILER = $(II_ROOT)\ti_fpc
II_CMD      = $(II_COMPILER)\iiPC32.cmd
II_BOOT     = $(II_COMPILER)\vectors.obj

PC32_DOWNLOAD = $(II_ROOT)\D_Load.exe

#_________________________________XI Tools ___________________----------
#
# You may need to modify the TI_ROOT  if you have installed the
# Texas Instrument Compiler in a different location.
#

TI_ROOT      = c:\fltc
TI_FLAGS     = -v$(DSP_FAMILY)

CC = $(TI_ROOT)\cl30
LD = $(TI_ROOT)\lnk30

#--------------------------------Include Path-----------------------------

MATLAB_INCLUDES = \
$(MATLAB_ROOT)\simulink\include; \
$(MATLAB_ROOT)\extern\include; \
$(MATLAB_ROOT)\rtw\c\src; \
$(MATLAB_ROOT)\rtw\c\libsrc;

TI_INCLUDES = $(TI_ROOT); $(TI_ROOT)\Include\Target;

II_INCLUDES = $(II_BOARD)\Lib\Target

INCLUDES = .; $(MATLAB_INCLUDES) $(TI_INCLUDES) $(II_INCLUDES)

#----------------------------Compiler Flags ----------------------------
# Required Options

REQ_OPTS    = $(TI_FLAGS) -q -eo .o$(DSP_FAMILY)

# Optimization Options
OPT_OPTS        = -xO -o3

# Debug Options
DBG_OPTS        =

CC_OPTS         = $(REQ_OPTS) $(OPT_OPTS) $(DBG_OPTS) -dIO_$(IO) \
                      -dTMR0_$(TMR0) -dUPLD_$(UPLD)

CPP_REQ_DEFINES = -dMODEL=$(MODEL) -dRT -dNUMST=$(NUMST) \

                      -dTID01EQ=$(TID01EQ) -dNCSTATES=$(NCSTATES)

CFLAGS          = $(CC_OPTS) $(CPP_REQ_DEFINES) $(CPPJDEFINES)

LDFLAGS         = -x -a -cr -heap 0x2000 $(II_BOOT) -m $(MODEL).map

#-------------------------------Source Files ---------------------------
REQ_SRCS         = PC32main.c pc32func.c rt_sim.c rt_matrx.c $(MODEL).c
OPT_SRCS         =
S_FCN_SRCS       = $(S_FUNCTIONS)
INT_SRCS         = $(SOLVER)
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)
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

#-------------------Compile and Link Rules ----------------------------
$(PROGRAM)  :  $(OBJS)
      echo $(OBJS) > $(MODEL).lin
      echo $(II_CMD) >> $(MODEL).lin
      $(LD) $(LDFLAGS) -o $@ $(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
%.o$(DSP_FAMILY) : c:\matlab\rtw\c\ii\ti_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_IFAMILY) : $(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) $(PROGRAM)
      echo $(DOWNLOAD_SUCCESS) $(PROGRAM)

#---------------------------Dependencies ------------------------------

iirt_main.o$(DSP_FAMILY) : $(MODEL).c
$(OBJS)       : $(MAKEFILE)
```

Appendix E

E.I Real Time Kernel

This appendix list source code for the Real Time Kernel. The functionality is contained in two files **pc32main.c and pc32func.c.**

E.I.I Source Listing of pc32main.c

```
/*********************************************************************
 *  pc32main.c                                                       *
 *    Entry point to code generated with RTW for the                 *
 *    PC32 DSP card.                                                 *
 *       This file is specifically for the TI compiler, V4.6, but    *
 *       will probably work under later versions.                    *
 *                                                                   *
 *     Adam Stylo       (07 Jan 1999)                                *
 *********************************************************************/

#include "pc32main,h"

/* Defines */

#ifndef RT
# error "must define RT"
#endif

#ifndef MODEL
# error "must define MODEL"
#endif

ttifndef NUMST
# error "must define number of sample times, NUMST"
#endif

ttifndef NCSTATES
# error "must define NCSTATES"
#endif

/* External functions */

  extern SimStruct *MODEL(void);
  extern void MdllnitializeSizes(void);
  extern void MdllnitializeSampleTimes(void);
  extern void MdlStart(void) ;
  extern void MdlOutputs(int_T tid);
  extern void MdlUpdate(int_T tid);
  extern void MdlTerminate(void);
  extern void ServiceUploads(void);

#if NCSTATES > 0
  extern void rt_CreateIntegrationData(SimStruct *S);
  extern void rt_UpdateContinuousStates(SimStruct *S);
#else
  #define rt_CreateIntegrationData(S)
      ssSetSolverName(S,"FixedStepDiscrete");
  #define rt_UpdateContinuousStates(S) ssSetT(S,ssGetSolverStopTime(S));
#endif
```

```
/* Global data */
real_T rtlnf;
real_T rtMinusInf,-
real_T rtNaN;

static SimStruct *S;
static real_T     *param,-
volatile uint32_T *dpram;
int int_flag = 0;
int dprx = 0;

/* Global structure for Data Uploads */

QUEUE queue[MAX_QUEUES];
unsigned int buffer_size[MAX_QUEUES];
unsigned int Down_Sample[MAX_QUEUES];
int LogData = FALSE;
int gueue_error = FALSE;
int NumQueues = 0;
unsigned int channel_map[MAX_QUEUES];

/*=================*
 * Local functions *
 *=================*/

#define     PC0_int c_intl4

static void PC0_int(void);

#ifdef TMR0_YES  /* Include rtOneStep function only if we use Timer 0 to
                    trigger it */
      #define     rtOneStep c_intl9
      static void rtOneStep(void);

/* Function: rtOneStep
=====================================================================
 *      Performs one step of the model.                 */

static void rtOneStep(void)
{
    real_T tnext;

    tnext = rt_GetNextSampleHit();
    ssSetSolverStopTime(S,tnext);

    MdlOutputs(0);
    MdlUpdate(0);

    rt_UpdateDiscreteTaskSampleHits(S);

    if (ssGetSampleTime(S,0) == CONTINUOUS_SAMPLE_TIME)
    {
        rt_UpdateContinuousStates(S);
    }
} /* end rtOneStep */

#endif /* TMR0_YES */

/ ----------------------------
 * Interrupt Service Routines *
 *________________________ */

static void PC0_int()
```

```
{
      /* wait for the sync signal from host*/
  if ((!int_flag) && ((dpram[0] & OxFFFO)==0xAAA0))
  {
      int_flag=l;
      dprx=dpram[0] & OxOOOF;}      /* extract the command byte */
  }

 * Visible functions *
 *==================*/

/*
 * main - The main program, which calls the initialization routines and
 *         executes the background task.
 */

int_T main(int_T argc, char_T *argv[])
{
    int rxmb;
    int waiting_for_comms;

    int host_start=0, ChecksumOK=0;

    dpram = (volatile uint32_T* ) kPeriph- >Dpram,-

/* Get going on the PC32 */
      MHZ = detect_cpu_speed();
      timer(0,0),•
      enable_cache();
      enable_clock();
      enable_monitor();

#ifdef IO_ENABLE
      clrscr(),-
      printf("Real Time Code Execution.\n");
ttendif

/* Initialize the model */

    S = MODEL();

    ssSetTFinal(S,0.0);  /* run forever */

    MdllnitializeSizes();
    MdllnitializeSampleTimes();

    rt_InitTimingEngine(S);
    rt_CreateIntegrationData(S);

      /*get address of parameter structure */
    param = ssGetDefaultParam(S);
#ifdef IO_ENABLE
      printf("Standing by to establish comms.\n");
#endif

      /* trap the host interrupt for comms */
      install_int_vector(PC0_int, 4);
      enable_interrupt(3);

      dpram[0]= 0;
      enable_interrupts();

/* infinite loop - but execution can still be stopped by host */
```

```
while (1)
{
   /* dont respond to interrupts 'till we process the current one */
   disable_interrupt(3);
   if (int_flag)
   {
    switch (dprx)
     {
      case 1:  /* command to start execution */
         #ifdef IO_ENABLE
           printf ("External Comms Initiated. Initialising Model.\n") ;
         #endif
           disable_interrupts();
           MdlStart();
           dpram[3] = 0;  /* assume all options are OFF */
         ttifdef TMR0_YES
         /* Timer 0 used to generate base clock for the model */
         /* trap the timerO interrupt for base time*/
           install_int_vector(rtOneStep, 9);
           enable_interrupt(8);
           timer (0, (int)(1.0 / ssGetStepSize(S)));
           dpram[3] += 2;  /* tell host we are using timer 0 */

         #ifdef IO_ENABLE
           printf("Model base sample time is %d Hz (generated by Timer
               0).\n", (int) (1.0 / ssGetStepSize(S)));
         #endif
         ttendif
           enable_interrupts();
           host_start=l;
         ttifdef IO_ENABLE
           printf("Model Initialised.\n");
           dpram[3] += 1; /* tell host if 10 to Terminal is enabled */
         ttendif

         #ifdef UPLD_YES
           dpram[3] += 4;  /* tell host uploading is enabled */
           ClearAllQueues();
         #ifdef IO_ENABLE
           printf("Data upload buffers are ready.\n");
         #endif
         ttendif
           break;

      case 2:  /* accept a parameter from host */
           rxmb=dpram[1];
           param[rxmb] =from_ieee(dpram[2]);

         ttifdef I0_ENABLE
           printf("Param[%d] = %g\n",rxmb,param[rxmb]);
         #endif
           break;

      case 3:  /*suspend execution-run through all termination code */
         ttifdef IO_ENABLE
           printf("Terminating Execution.\n");
         #endif

           disable_interrupts();

         ttifdef TMR0_YES
               /* stop timer 0 int only if it was used */
           disable_interrupt (8);
         ttendif
```

```
                MdlTerminate();
                enable_interrupts();
                host_start=0;
                break;

        case 4:
            /* verify that we have the latest build on both host and DSP*/

            ttifdef IO_ENABLE
              printf("Verifying cheksums....");
            ttendif

              ChecksumOK=l;

              if (dpram[l] != ssGetChecksumO(S)) ChecksumOK=0;
              if (dpram[2] != ssGetChecksuml(S)) ChecksumOK=0;
              if (dpram[3] != ssGetChecksum2(S)) ChecksumOK=0;
              if (dpram[4] != ssGetChecksum3(S)) ChecksumOK=0;

              if (ChecksumOK)
              {
               ttifdef IO_ENABLE
                  printf("OK!\n");
               ttendif
              }
                else
              {
                dprx=0;
               #ifdef IO_ENABLE
                  printf("FAILED!\n");
               #endif
              }
            break;
        case 5:  /* clear queues and start logging data */
            #ifdef IO_ENABLE
              printf("Data logging started.\n");
            #endif

              ClearAllQueues();
              LogData = TRUE;
            break;

        case 6:  /* suspend logging data */
            #ifdef IO_ENABLE
              printf("Data logging suspended.\n");
            ttendif

              LogData = FALSE;
            break;
    }
    dpram[0] = OxABCO + dprx,-  /* send an acknowledge to host */
    int_flag = 0;  /* ready for a new command */
    }

     enable_interrupt(3);   /* lets listen out for comms from host */

    if (host_start)
    {
      #ifdef UPLD_YES
          ServiceUploads();   /* Upload data if neccesary */
      ttendif
```

```
 /* run background blocks (if any) if Timer 0 is not doing it already*/

        #ifndef TMR0_YES
            MdlOutputs(0);MdlUpdate(0);
        ttendif
      }
    }

      /* we never get here, but just in case of a MAJOR problem
         exit (or rather die) gracefully */

  MdlTerminate();
  return(1);

} /* end main */
```

E.1.2 Source Listing of pc32func.c

```
/*******************************************************************
 *  pc32func.c                                                     *
 *    support functions for data uploading via DPRAM on the        *
 *    PC32 DSP card.                                               *
 *      This file is specifically for the TI compiler, V4.6, but   *
 *      will probably work under later versions.                   *
 *                                                                 *
 *     Adam Stylo       (07 Jan 1999)                              *
 *******************************************************************/

#include "pc32main.h"

extern SimStruct* S;
extern QUEUE queue[];
extern unsigned int buffer_size[];
extern volatile uint32_T* dpram;
extern int NumQueues;
extern int queue_error;
extern LogData;
extern channel_map[] ;
extern real_T* param;
extern int int_flag;
extern int dprx;

#define MAXOFFSET 1001
#define BASE_DPRAM 5

void ClearAllQueues(void)
{
      int num;
      /* chuck all enqued data out */
      for (num=0; num<NumQueues; num++)
        while (enqueued(kqueue[num] ) ) dequeuejptr(kqueue[num]);
}


void ServiceUploads(void)
{
      int num=0;
      int offset=0;
      int dpram_full=O;

      int              xxx;
      unsigned int      zzz;

  if ((LogData) && !(dpram[4] & 1))
  {
      while ((num<NumQueues) && (offset<MAXOFFSET))
      {
        if ((enqueued(&queue[num]) >= buffer_size[num]) &&
              (buffer_size[num]<MAXOFFSET-offset))
        {
           dpram[BASE_DPRAM + offset++] = channel_map[num];
            /* put queue number in dpram */

           dpram[BASE_DPRAM + offset++] = buffer_size[num];
            /* put queue size in dpram */

           for (xxx=0; xxx<buffer_size[num]; xxx++)
           {
            dpram[BASE_DPRAM + offset++] =
                  to_ieee(*(volatile float*)dequeue_ptr(&queue [num]));

           }

           dpram_full = TRUE;
        }

        num++;
      }

      /* got something to sent to host, tell him */

      if (dpram_full)
      {
            dpram[BASE_DPRAM + offset] = (int)999;
            dpram [4] += 1;
            host-interrupt();
      }

}

}
```

Appendix F

F.I Driver Block DLL

This appendix list source code for the S-function DLL file for the II interrupt driver block. The differences between the DLL files for other driver blocks are minor, thus to conserve space only this one will be listed as an example.

F.I.I Source Listing of iiinterrupt.c

```
/*   Front-end file for the PC32 interrupt support block.
     There is absolutely no useful code here apart from
     setting the number of parameters to use with the mask.

       Adam Stylo  (January 1999)
 */

#define S_FUNCTION_NAME iiinterrupt
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

#ifndef MATLAB_MEX_FILE

/* Since we have a target file for this S-function, declare an error here
 * so that, if for some reason this file is being used (instead of the
 * target file) for code generation, we can trap this problem at compile
 * time. */

#   error This_file_can_be_used_only_during_simulation_inside_Simulink
#endif


 * S-function methods *
  --------------------- /
static void mdllnitializeSizes(SimStruct *S)
{

    ssSetNumSFcnParams( S, 6 );
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
    {
        return; /* Simulink will report a parameter mismatch error */
    }
    ssSetSFcnParamNotTunable ( S,0),-
    ssSetSFcnParamNotTunable( S, 1);
    ssSetSFcnParamNotTunable( S, 2);
    ssSetSFcnParamNotTunable( S, 3);
    ssSetSFcnParamNotTunable( S, 4);
    ssSetSFcnParamNotTunable( S, 5);

    ssSetNumlnputPorts( S, 1),-
    ssSetlnputPortWidthf S, 0, DYNAMICALLY_SIZED);
    ssSetInputPortDirectFeedThrough( S, 0, 1);

    ssSetNumOutputPorts( S, 1);
    ssSetOutputPortWidth( S, 0, DYNAMICALLY_SIZED);
    ssSetNumlWork( S, 0);
    ssSetNumRWork( S, 0);
    ssSetNumPWork( S, 0) ;

    ssSetNumSampleTimes( S, 1);
    ssSetNumContStates ( S, 0);
    ssSetNumDiscStates ( S, 0) ;

    ssSetNumModes ( S, 0) ;
    ssSetNumNonsampledZCs( S, 0);
    ssSetOptions( S, (SS_OPTION_EXCEPTION_FREE_CODE |
                          SS_OPTION_ASYNCHRONOUS));
}

static void mdllnitializeSampleTimes(SimStruct *S)
{
   int i;
   /* simulation mode - used inside of triggered subsystem */
        ssSetSampleTime(S, 0, 1.0);
        ssSetOffsetTime(S, 0, 0.0);

        for(i = 0; i < ssGetOutputPortWidth(S,0); I++)
        {
              ssSetCallSystemOutput(S, i);
        }

}

static void mdllnitializeConditions(SimStruct *S)
{

}

static void mdlOutputs(SimStruct *S, int_T tid)
{

}

static void mdlTerminate(SimStruct *S)
{

}
/*=============================*
 * Required S-function trailer *
 *=============================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Appendix G

G.I Hardware Drivers TLC Files

This appendix list the TLC source code for the functions which implement hardware access on the target side.

G.I.I Source Listing of ADTrigger .tic

```
%%
%%
%% Abstract:
%%      TLC file for the PC32 A/D Trigger Block.
%%    Performs four writes to A/D memory locations thus
%%    triggering a conversion. Conversion is triggered
%%    regardless of the status of JP5 & 6 on the board
%% Author:
%%      Adam Stylo
%% Date:
%%      98/11/03
%%

%implements "ADTrigger" "C"

%include "iilib.tlc"

%% II define these addr in their headers, but rather safe then sorry...
%openfile buffer
      /*
      define addresses for ADC Triggering
      */

ftdefine ADCO (volatile int*) 0x810000
#define ADC1 (volatile int*) 0x810800
#define ADC2 (volatile int*) 0x811000

ttdefine ADC3 (volatile int*) 0x811800

%closefile buffer

%<LibCacheDefine(buffer)>

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

%endfunction %% Outputs

%% [EOF] ADTrigger.tic
```

G.1.2 Source Listing of iiinterrupt.tlc

```
%%
%%
%% 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
%%

%implements "iiinterrupt" "C"

%include "iilib.tlc"

%% Function: BlocklnstanceSetup
 ===============================================
%% 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 Connects each ISR in the model's start function.
%%
%%      o Enables each ISR at the bottom of the model's start function.
%%
%%      o Disables each ISR in the model's terminate function.
%%
%%      o Saves floating point context in the ISR's critical code section

%assign ::EITrigg = LibBlockParameter(PI,"","",0)
%assign ::Tmr0freq = LibBlockParameter(P4,"","",0)
%assign ::Tmrlfreq = LibBlockParameter(P5,"","",0)
%assign ::Trigg_srcO = LibBlockParameter(P2,"","",0)
%assign ::Trigg_srcl = LibBlockParameter(P3,"","",0)

%function BlocklnstanceSetup(block, system) void

  %% Only allow 1 interrupt block
  %if EXISTS("IIInterruptBlockSeen")
    %assign errTxt = "Only 1 Interrupt block is allowed in " ...
      "model: %<CompiledModel.Name>."
    %exit RTW Fatal: %<errTxt>
  %else
    %assign ::IIInterruptBlockSeen = 1
  %endif

  %% Disable interrupts while setting up
  %openfile buffer

 /* Disable interrupts while setting up */
      #ifdef IO_ENABLE
        printf("Connecting Interrupts\n");
      #endif

 /* Make ints edge triggered only if required */
      if ((int)%<EITrigg> == 1)
```

```
     {
       asm ("     OR 4000h,ST");
       ttifdef IO_ENABLE
        printf("EIO - EI3 Set to edge triggered\n");
       #endif
     }
      else
     {
       asm ("    ANDN 4000h,ST");
       #ifdef IO_ENABLE
        printf("EIO - EI3 Set to level triggered\n");
       ttendif
     }

%closefile buffer

%<LibMdlStartCustomCode(buffer, "header")>

%openfile buffer

/* define addresses for control registers */

#define GC_CTRL0 (volatile int*) 0x808020
#define GC_CTRL1 (volatile int*) 0x808030

%clofc»efile buffer

%<LibCacheDefine(buffer)>

%foreach callldx = NumSFcnSysOutputCalls

%% Get downstream block if there is one
%if "%<SFcnSystemOutputCall[callldx].BlockToCall>" != "unconnected"
%assign ssSysIdx = SFcnSystemOutputCall[callldx] .BlockToCall [0]
%assign ssBlkldx = SFcnSystemOutputCall[callldx] .BlockToCall [1]
%assign ssBlock = CompiledModel.System [ssSysIdx] .Block[ssBlkldx]

%% Check to see if this is a direct connection
%if (ssBlock.ControlInput.Width != 1)
%assign errTxt = "The II 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.Systemldx]

%% redefine arguments of ISR function

%openfile buffer
     /*
     redefine arguments for ISR: %<ssBlock.Name>
     */
  ttdefine
    %<isrSystem.OutputUpdateFcn>(%<tSimStruct>,%<tControlPortIdx>,%<tTID>)
         c_intO%<callIdx+l>()

%closefile buffer
```

```
%<LibCacheDefine(buffer)>

%% Connect the ISR in the model's start function

%openfile buffer
     /*
     connect ISR system: %<ssBlock.Name>
     */
if (%<LibBlockParameter(P6,"","",callldx)>==9)
{
     /*
     check that timerO int is unused before assigning new vector
     */
  #ifndef TMR0_YES
 %if callldx < 9
     install_int_vector(c_into%<callldx+l>,(int)%<LibBlockParameter(P6," ",
           11",callldx)>),-
 %else
     install_int_vector(c_int%<callldx+l>,(int)%<LibBlockParameter(P6,"",
           "",callldx)>);
 %endif
     enable_interrupt((int)%<LibBlockParameter(P6,"","",callldx)>-1);
  #ifdef IO_ENABLE
     printf("Vectior installed for INT
                 #%d.\n",(int)%<LibBlockParameter(P6,"","",callldx)>);
  #endif
  #endif
}
else
{
 %if callldx < 9
     install_int_vector(c_into%<callldx+l>,(int)%<LibBlockParameter(P6, "",
           "",callldx)>),-
 %else
     install_int_vector(c_int%<callldx+l>,(int)%<LibBlockParameter(P6," ",
           "",callldx)>);
 %endif

     enable_interrupt((int)%<LibBlockParameter(P6,"","",callldx)>-l);
 #ifdef IO_ENABLE
    printf("Vectior installed for INT
                 #%d.\n",(int)%<LibBlockParameter(P6,"","",callldx)>);
 #endif
}

%closefile buffer

%<LibMdlStartCustomCode(buffer, "trailer")>

%openfile buffer

     /*
     disconnect ISR system: %<ssBlock.Name>
     */

if  (%<LibBlockParameter(P6,"","",callldx)>==9)
{
     /*
     only disconnect timerO if it was set up here
     */
  #ifndef TMR0_YES
     disable_interrupt((int)%<LibBlockParameter(P6, "", "",callldx)>-l);
     deinstall_int_vector((int)%<LibBlockParameter(P6,"","",callldx)>);
  #ifdef IO_ENABLE
      printf("INT #%d disabled.\n",(int)%<LibBlockParameter(P6,"",
                                                                 "",callldx)>);

  #endif
  #endif
}
else
{
     disable_interrupt((int)%<LibBlockParameter(P6,"","",callldx)>-l);
     deinstall_int_vector((int)%<LibBlockParameter(P6,"","",callldx)>);
  #ifdef IO_ENABLE
     printf("INT #%d disabled.\n",(int)%<LibBlockParameter(P6,"",
                                                                 "",callldx)>);
  #endif
}

%closefile buffer

%<LibMdlTerminateCustomCode(buffer, "trailer")>

%else %% The element is not connected to anything

%assign wrnTxt = "No code will be generated for ISR %<callldx> "\

                              "since it is not connected to anything."

%warning %<wrnTxt>

%endif

%endforeach

%% Setup timers and enable global interrupts

%openfile buffer

ttifndef TMR0_YES
     /*
     Only change Timer 0 settings if it isn't used for base sampling rate
     */
  timer(0, (int)%<Tmr0freq>);
ttendif
  timer(1, (int)%<Tmrlfreq>);
  if ( (int)%<Trigg_srcO> == 2)
  {
     *GC_CTRL0 = 0x6c3;
   ttifdef IO_ENABLE
      printf("TCLKO driven by Timer 0.\n");
   ttendif
  }
   else
  {
     *GC_CTRL0 = 0x6CO;
   ttifdef IO_ENABLE
      printf("TCLKO driven externaly.\n");
   ttendif
  }

  if ((int)%<Trigg_srcl> == 2)
  {
     *GC_CTRL1 = 0x6c3;
   ttifdef 10 _ENABLE
```

```
      printf("TCLK1 driven by Timer l.\n");
    #endif
   }
    else
   {
      *GC_CTRL1 = 0x6c0;
    #ifdef  IO_ENABLE
      printf ("TCLK1 driven externaly. \n") ,•
    #endif
   }
    #ifdef  IO_ENABLE
       printf("Interrupts Connected & Enabled\nT');
    #endif

 %closefile buffer

 %<LibMdlStartCustomCode(buffer,  "trailer")>

%endfunction

%% [EOF] iiinterrupt.tic
```

G.1.3 Source Listing of pc32_ad.tlc

```
%%
%%
%% 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
%%

%implements "pc32_ad" "C"

%include "iilib.tlc"

%function Blocklnstancesetup(block, system) void

 %% Only allow 1 instance of the A/D block
 %if !EXISTS("Rt_pc32ad")
   %assign :Rt_pc32ad = 1
 %else
   %error Only 1 PC32adn block is allowed in the model.
 %Endif

%endfunction %% BlocklnstanceSetup

%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
```

G.I.4Source Listing of pc32_da.tlc

```
%%
%%
%% 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 "pc32_da" "C"

%include "iilib.tlc"

%function Blocklnstance Setup(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 %% BlocklnstanceSetup

%% 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
```

G.1.5Source Listing of pwmblock.tlc

```
%%
%%
%% 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
%%

%implements "pwmblock" "C"

%include "iilib.tlc"

%assign ::Vortl = LibBlockParameter(PI,"","",0)
%assign ::CtrlMode = LibBlockParameter(P2,"","",0)

%function Blocklnstancesetup(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

 ttifdef IO_ENABLE
   printf("Initializing PWM Block ....\n");
 ttendif
 VORTL = (int)(%<Vortl>);
 TSTART = ((int) (512-(322/(VORTL + 1))) ) ;

 *IOBCR = 0x58;
 *(Status_word) = 128;     /* set up 16 bit addressing mode */
 *(Status_word) = 128;     /* set addres to zero */

 pollpwm();

 *(Data_word) = 0 ;  /* Ua */

 pollpwm();

 *(Data_word) = 0 ;  /* Ub */

 pollpwm();

 *(Data_word) = 0;  /* phil */

 pollpwm();

 *(Data_word) = 0;  /* dphil */

 pollpwm();

 *(Data_word) = 0;  /* phiO */

 pollpwm();

 *(Data_word) = 0;  /* dphiO */

 pollpwm();

 *(Data_word) = 0;  /* phiadd */

 pollpwm();

 *(Data_word) = 0;  /* unused */

 pollpwm();
```

4.2.5 'C32 Development Tools

TI provide a code development suite for the 'C32, including :

(i) Optimising ANSI C compiler - translates standard ANSI C language directly into optimised assembly code,

(ii) Assembler/Linker - converts source mnemonics to executable object code and links all specified modules int a single image file,

(iii) Pre compiled libraries - provide support for specific DSP algorithms as well as standard ANSI C functionality.

4.3 Motion Controller Hardware

Having discussed the 'C32 processor's architecture, the following sections will introduce the two components of the CSDE hardware platform, namely the PC32 DSP controller and the PWM/Tacho card. The PC32 is a commercially available solution while the PWM/Tacho expansion card was developed in-house at UND [WALKERl]. Fig. 4.4 shows the overall structure of the CSDE target hardware.

The PC32 slots into the PC expansion bay and all interfacing is done via the ISA bus. The custom expansion card is "piggy-backed" onto the PC32 and slots into an adjacent ISA slot for mechanical support and also draws its power supply from the PC. The interfacing between the PWM/Tacho card and the PC32 is performed via the PC32's expansion bus. This way the expansion card is mapped directly into the 'C32 address space.

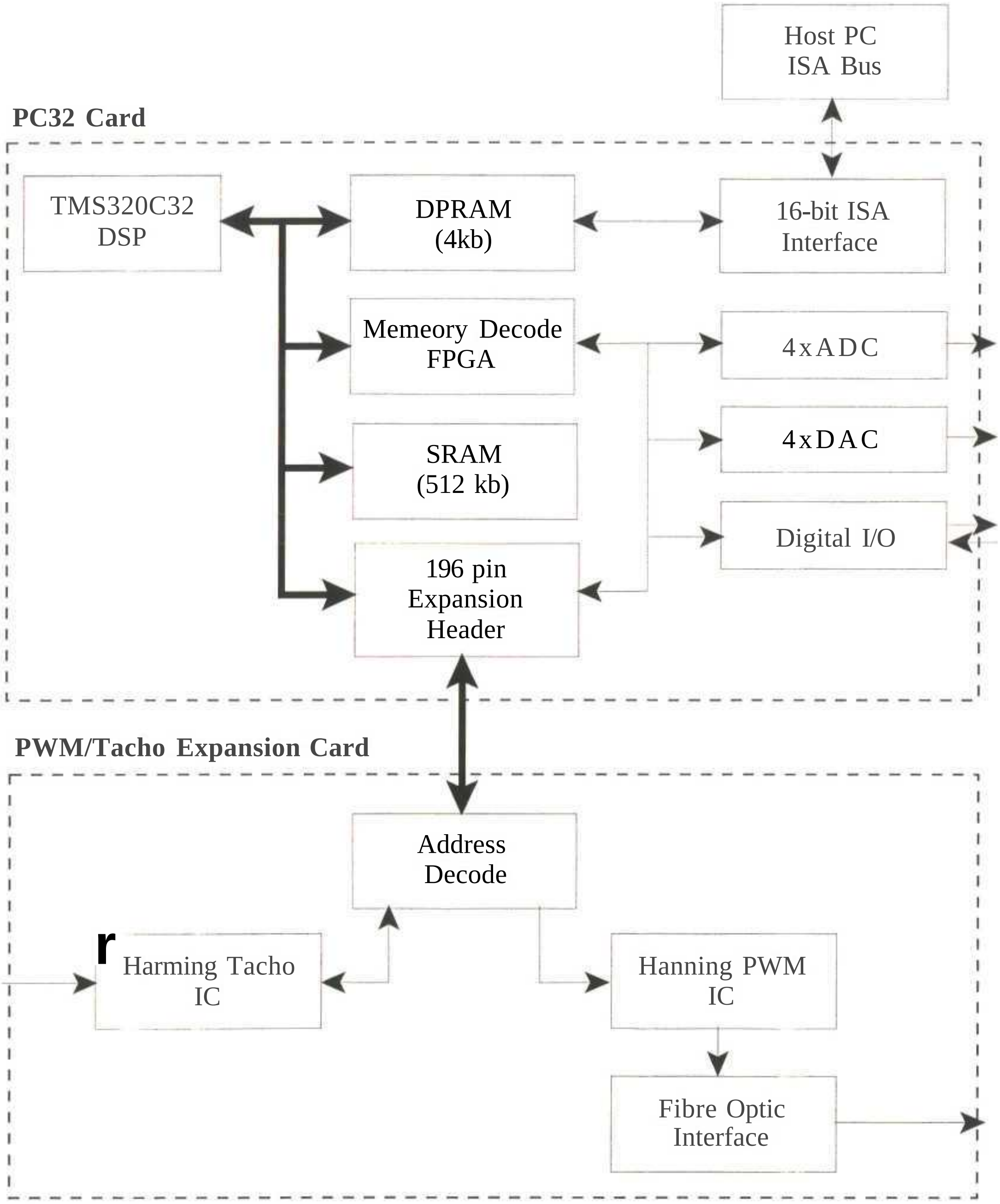


Fig. 4.4 : Overall CSDE Hardware Structure

4.4 PC32 DSP Controller Card

The PC32 card from Innovative Integration [INN0VATIVE1], represents a complete DSP system on a single half-length ISA expansion card and forms the centre of the CSDE target hardware. A photo of the card is shown in Fig. 4.5. The board incorporates a number of memory mapped peripherals, namely:

(i) Four ADC channels

(ii) Four DAC channels

(iii) 16 digital I/O, which can be configured as 16 in, 16 out, or 8 in / 8 out.

(iv) 196 pin expansion header, which gives direct access to the memory bus, interrupts as well as other processor pins,

(v) 4 kb of DPRAM, mapped to a 16-bit ISA interface for communicating with the host PC.

(vi) 512 kb SRAM, up to 2Mb can be supported.

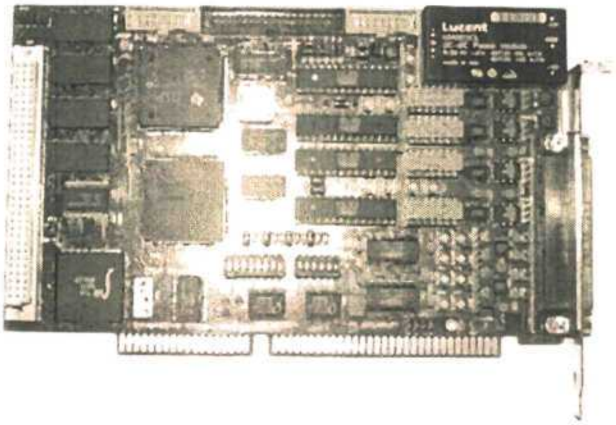


Fig. 4.5 : Photo of the Innovative Integration PC32 card.

4.4.1 Memory

The total address range of 16 MWords (addresses from 0x000000 to OxFFFFFF), is mapped on the PC32 as shown in Fig. 4.6. The PC32 directly supports up to 512 kWords (2 Mb) of zero-wait-state SRAM. Optionally up to 256 kWords (1 Mb) of slower one-wait-state memory can be supported. The 1 kWord (4 kb) of DPRAM and the onboard peripherals are also mapped into the general address range. Internally the 'C32 uses 32-bit data representation. Physically,

'C32 memory strobe :	**Description :**	Address :
	Bootloader	0x000000
STRBO	Dual Port Memory 16-bit physical 32-bit logical	0x001000
STRBO	Data Memory 16-bit physical 32-bit logical	0x200000
STRBO	Expansion 16-bit physical 32-bit logical	0x400000
	Reserved and Internal Peripherals	0x800000
1OSTROBE	External Peripherals	0x810000
	Reserved and Internal RAM	0x830000
STRBO	Expansion 16-bit physical 32-bit logical	0x880000
STRB 1	Fast SRAM 32-bit physical 32-bit logical	0x900000
		OxFFFFFF

Fig. 4.6 : 'C32 Memory Map on the PC32

only the SRAM memory mapped into STRB 1 region is 32-bit wide. Since the SRAM is also zero-wait-state, accesses to it require only a single processor cycle. STRB 0 memory regions are populated with 16-bit, one-wait-state memory devices (DPRAM and data memory). Accessing data in those areas requires a total of four processor cycles, two cycles per 16-bit access due to the wait state and two cycles to emulate a 32-bit transaction. The IOSTROBE region of memory is used to map peripherals external to the 'C32 processor and can also be utilised in applications using the external PC32 expansion header.

4.4.2 Dual Port Memory (DPRAM)

The DPRAM is addressable as a continuous block of 32-bit memory from address 0x1000 to 0x1400. From the PC side it is visible at a conventional memory address specified by DIP switch S2 on the PC32 card, the default setting is 0xD000:0000. DPRAM accesses are not wait stated from the PC side, thus data rates can be significantly faster than the conventional PC I/O bus interface which normally requires three bus cycles per transaction.

The access to DPRAM is not arbitrated in hardware. This means that simultaneous accesses to the same DPRAM address by both the PC and the PC32 will yield unpredictable results. To avoid conflicts, a common protocol has to be defined between the PC and the PC32 application. The use of specific areas of the DPRAM can be arbitrated using one of the following methods [INNOVATIVE2] :

(i) Four hardware semaphores are provided as peripherals on the PC32 and are visible via conventional I/O registers on the host PC. A semaphore can be owned by only one side at a time, and simultaneous requests for ownership are elegantly resolved by hardware. However, semaphore requests might have to be polled until successful, thus incurring a possible performance penalty.

(ii) Interrupt signals passed between the host and target can signal when DPRAM is available for access to either side. A careful use of this method can avoid the overheads involved in semaphore switching and polling.

The CSDE makes use of an interrupt based DPRAM access protocol as defined in Appendix B.

4.4.3 Memory Mapped Peripherals

All PC32 peripherals external to the 'C32 processor are memory mapped. The address range from 0x81000 to 0x81CFFF is reserved for this purpose and the IOSTROBE signal from the 'C32 is used. Due to the address decoding mechanism used on the PC32 each peripheral consumes an address range of 0x800 words. Table 4.3 shows the memory allocation for the peripherals.

Decode 0 and 1 are specifically decoded by the PC32 with custom daughter boards in mind and the resultant signals are available via the expansion header. The PWM/Tacho card, introduced in section 4.5 makes use of these signals to map its devices.

Address	Device
0x810000	ADCO
0x810800	ADC 1
0x811000	ADC 2
0x811800	ADC 3
0x812000	PC Refresh
0x812800 to 0x813FFF	Reserved
0x814000	DACO
0x814800	DAC 1
0x815000	DAC2
0x815800	DAC 3
0x816000	Update DAC 0 output
0x816800	Update DAC 1 output
0x817000	Update DAC 2 output
0x817800	Update DAC 3 output
0x818000	Digital I/O
0x818800	Boot Indication Override to PC
0x819000	Interrupt to PC
0x819800	Decodes 0
0x81a000	Decodes 1
0x81a800	Reserved
0x81b000	Semaphore 0
0x81b800	Semaphore 1
0x81c000	Semaphore 2
0x81c800	Semaphore 3

Table 4.3 : IOSTROBE PC32 Peripheral Mapping

4.4.4 Analogue to Digital Converters

The PC32 offers four independent ADC channels, each with 16-bit accuracy, ± 10V input range and a guaranteed maximum conversion time of 10 */us* [BURRBROWNl]. The analogue inputs are filtered through a 6 pole anti-alias filter preset to have a 50 kHz passband. The filter poles can be adjusted by means of resistor banks. The ADC channels are mapped into the IOSTROBE region of memory as shown in Table 4.3. Reading a 32-bit word form any of the ADC addresses returns the result of the last completed conversion in the lower 16 bits as an unsigned integer. The top 16 bits are undetermined and should be masked off.

ADC conversions can be triggered by a combination of the following methods :

(i) Memory write to the ADC address. The write does not affect the value of the last sample stored,

(ii) An external active low TTL signal with a minimum pulse width of 40 ns. The signals can be connected via PC32's external D-type connector,

(iii) One of the timer signals (TCLK 0 and 1) from the 'C32 processor. Setting up timer modes and outputs is detailed in section 4.2.4.

For the purpose of triggering, the ADC devices are grouped into two pairs. ADC 0 and 1 form pair 0 and ADC 2 and 3 form pair 1. The triggering source for each pair is selected using jumpers JP 5 and JP 6. This selection is logically ORed with the memory writes. Fig. 4.7 demonstrates the triggering logic for ADC pair 0, the triggering setup for pair 1 is identical, but uses JP 6.

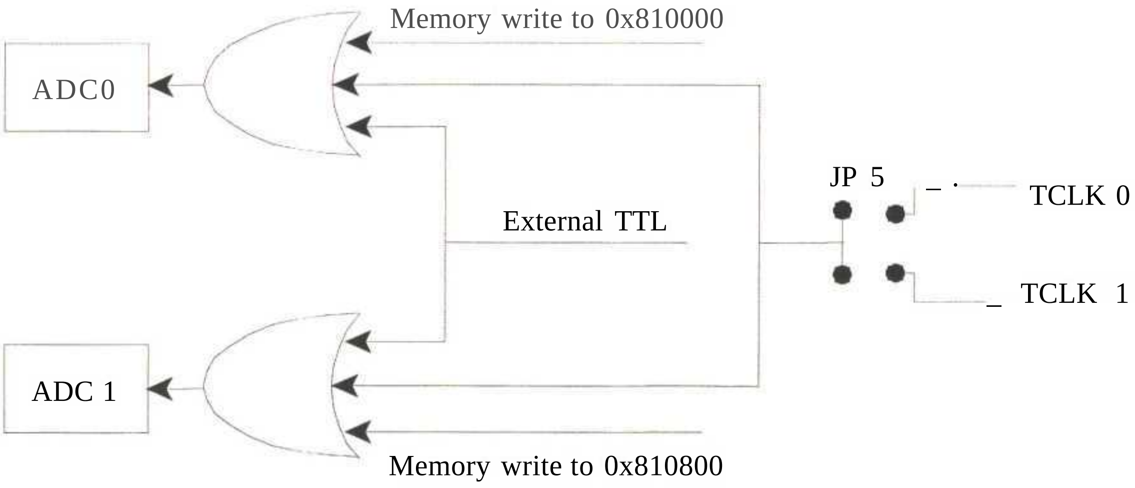


Fig. 4.7 : Triggering for the ADC pair 0

4.4.5 Digital to Analogue Converters

The PC32 has four independent 16-bit DAC channels with a ±10V output swing [BURRBR0WN2]. The maximum output settling time is 10 */J.S* for a full scale 20V step. Analogue outputs are filtered through a 200 kHz filter to remove any digital noise. The DAC data latches are memory mapped as shown in Table 4.3, only the lower 16 bits are significant. Writing to the latches does not trigger a conversion and reading returns an undetermined value.

All DAC channels may be triggered separately by one of the following :

(i) Memory write to the update register,
(ii) TCLK 0 signal,
(iii) TCLK 1 signal.

Jumpers JP 3, JP 4, JP 10 and JP 11 select trigger sources for DAC 0, 1,2 and 3 respectively. No additional logic is provided making the three trigger sources mutually exclusive.

4.4.6 PC32 Interrupts

Four external interrupt pins are available on the 'C32 - El 0, 1,2 and 3. El 1 to 3 can be patched through to selected trigger sources on the PC32 via the jumper header JP 17. Table. 4.4 shows the possible combinations. El 0 to 4 are also directly available via the expansion header. In the CSDE, El 0 is used by the custom expansion card, introduced in section 4.5. The PC_INT 0 and 1 signals can be generated by the host PC via writes to PC32 control registers mapped into the PC's general I/O space. The CSDE uses PC_INT 0, patched through to El 3 as part of the communication protocol between the host PC and the PC32.

Interrupts can also be generated on the host PC by the PC32 card. A memory write to PC32 address 0x819000 generates an interrupt signal which can be patched via jumper JP 7 to trigger IRQ 5, 7,11 or 15 on the host PC. The interrupt signalling between the host PC and the PC32 is shown in Fig. 4.8.

Interrupt	Source	JP 17 setting
Ell	ADC pair 0 end of conversion	1-3
Ell	External digital clock	8-10
EI 1	PCJNT 0	1-2
EI2	ADC pair 0 end of conversion	3-5
EI2	ADC pair 1 end of conversion	5-7
EI2	External digital clock	10-12
EI2	PCJNT 1	11-12
EI3	PCJNT 0	2-4
EI3	ADC pair 0 end of conversion	3-4
EI3	PCJNT 1	9-11
EI3	External digital clock	9-10
EI3	ADC pair 1 end of conversion	7-9

Table 4.4 : PC32 Interrupt Sources Selection

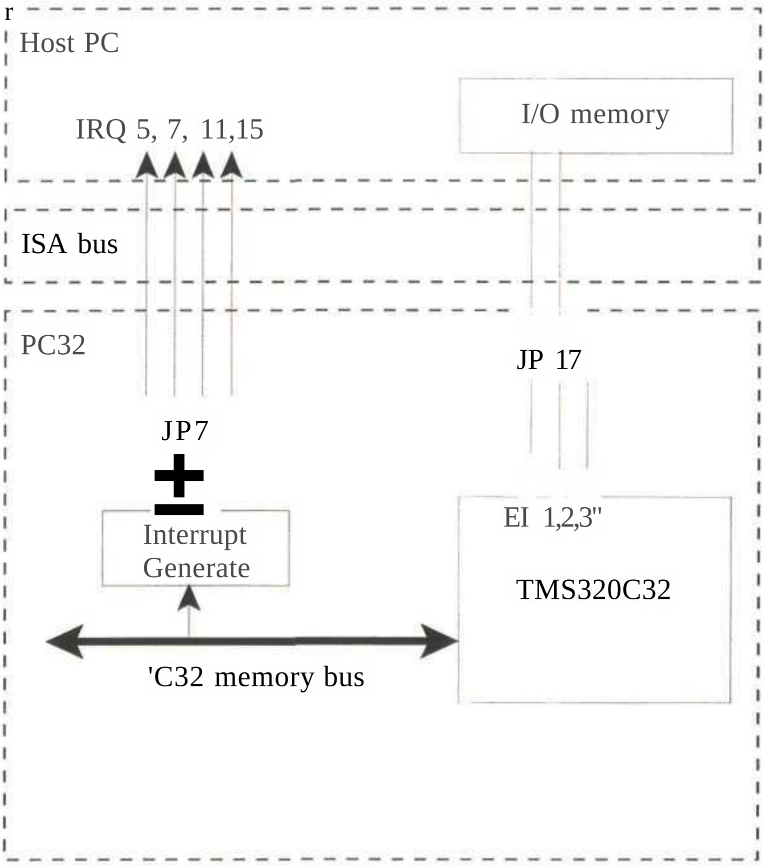


Fig. 4.8 : Interrupt Signalling between the PC and the PC32

4.4.7 Innovative Integration Development Environment

II ship the PC32 controller card with a comprehensive set of support software in the form of both pre-compiled libraries as well as corresponding source code, namely :

(i) Application specific libraries - this includes advanced maths and DSP functions,

(ii) Hardware access - functions implementing easy access to on board peripherals,

(iii) Target-to-host access - functions which allow access to hardware semaphores and mailboxes, and also interrupt signalling to the host PC.

(iv) Host-to-target access - a 32-bit DLL which implements all communication and access to the PC32 target from within the Windows 95 environment.

4.5 PWM/Tacho Expansion Card

Alongside the PC32, as introduced above, the custom PWM/Tacho card forms the second component of the CSDE target hardware. This expansion card for the PC32 was designed by Mr. Walker [WALKER1] to be used in a control system using the RIDE platform. The author added minor modifications to the interrupt signalling to adapt it to the needs of the CSDE. In motion control applications the TWM/Tacho card relieves the 'C32 processor of two computationally intensive and time critical tasks :

(i) Generation of PWM power electronics switching waveforms. The PBM 1/87 ASIC [HANNING1] is a dedicated PWM slave peripheral,

(ii) Decoding signals from incremental speed or position encoder. The ASIC used for this purpose is the TC3005H [HANNING2].

A photograph of the expansion card is shown in Fig. 4.9, while Fig. 4.10 shows the interconnection between the PC32 and the PWM/Tacho card inside the host PC. The two cards occupy two adjacent ISA slots and are joined via the expansion header on the PC32 **card.**

4.5.1 Address Decoding

The PWM/Tacho card is designed to use either the Decodes 0 or 1 region of PC32 memory space, as described in section 4.4.3. Each Decodes region spans an address range of 0x800

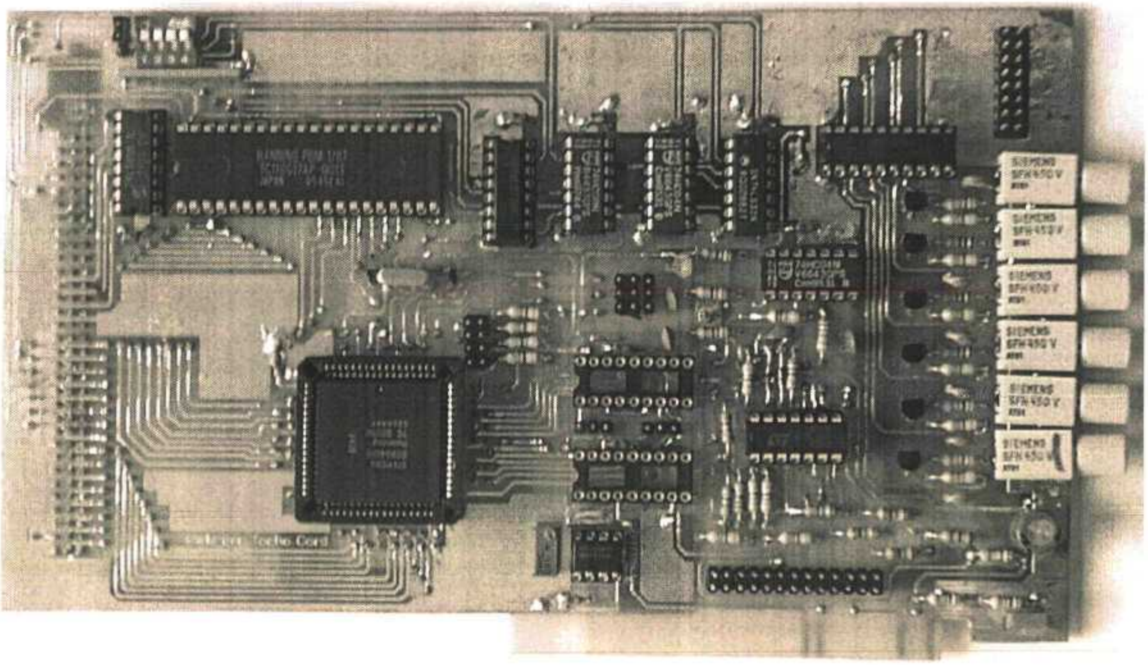


Fig. 4.9 : Photograph of the PWM/Tacho Expansion Card

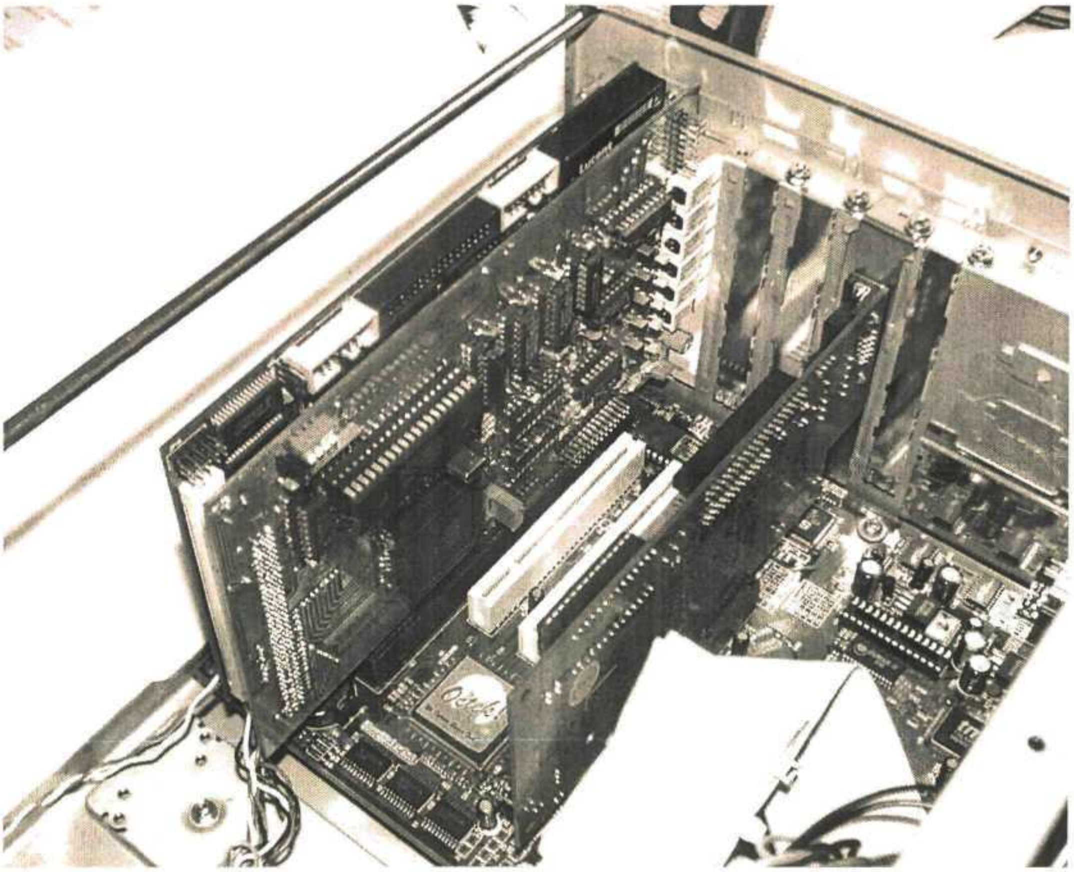


Fig. 4.10 : Photograph of the PC32 and PWM/Tacho cards inside the host PC.

words, and by means of a DIP switch on the expansion card this range is further sub divided into blocks of size 0x100 words. This mechanism allows up to 8 expansion cards in each Decodes region. Thus, a total of 16 PWM/Tacho cards can be addressed by a single PC32 controller. The memory decoding arrangement is shown in Fig. 4.11.

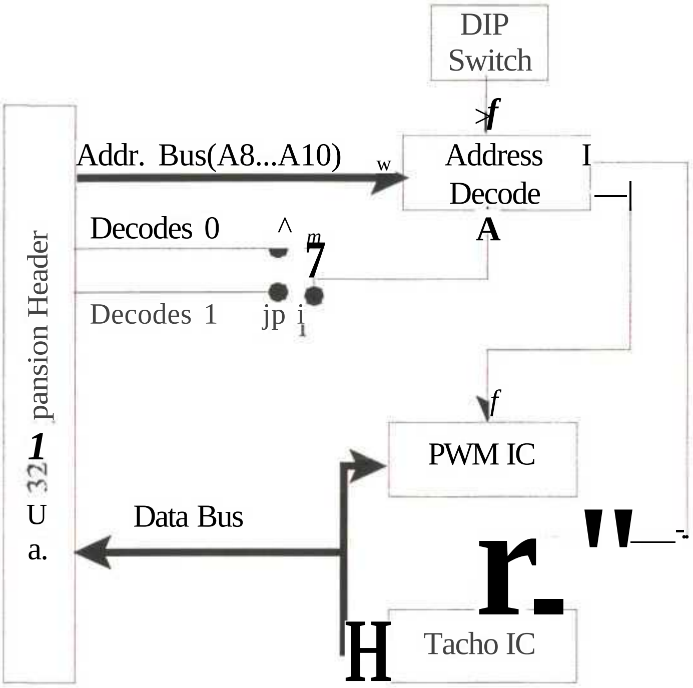


Fig. 4.11: Address Decoding on the Expansion Card

4.5.2 PWM ASIC

The PBM 1/87 is a slave peripheral designed specifically with motion control in mind [HANNING1]. It can generate PWM switching waveforms for a 3-phase inverter to generate a sinusoidal supply at the desired voltage, frequency and phase. The host processor is relieved of any time critical calculations and simply writes the PWM settings to the IC's configuration registers. The IC uses these values to calculate the required switching pattern. The PMB 1/87 registers and their function are listed in Appendix C.

The PMB 1/87 uses "space vector or triangular modulation with a third harmonic injection" [HANNING1] with switching events positioned symmetrically around a switching period. Fig. 4.12 shows a typical waveform for one of the 6 outputs. The switching period T is equal to the sum of the on-time T_{on} and the off-time T_{of}f. The INT signal always occurs when no switching is taking place and can be used to trigger the I/O channels used to sample currents and

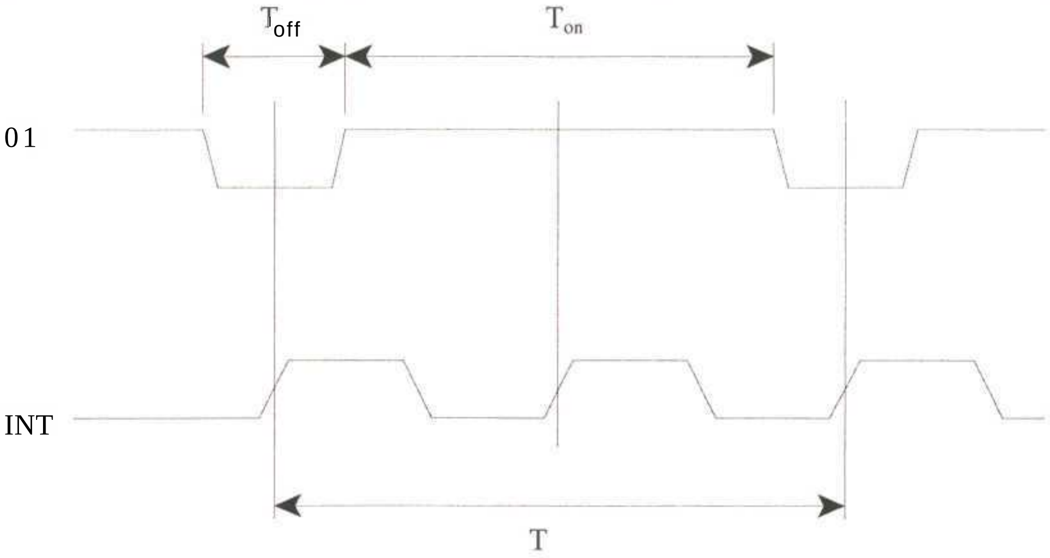


Fig. 4.12: Typical switching waveform for a single output

voltages at the inverter's outputs. Since no switching occurs at the sampling instants, the resultant samples are guaranteed to be free from switching noise.

Writes to the PWM ASIC's configuration registers can only be performed when the calculation cycle is inactive. To ensure all register accesses fall during the idle time of the PWM calculation, the interrupt signal can be used or a flag can be monitored in the IC's status register. These two mechanisms are discussed in chapter 7, section 7.5.5. Values written during a half-cycle are used in the next calculation, but their result is only used to control the switching during the following half-cycle. This is shown in Fig. 4.13, the diagonal arrows indicate how the result of each calculation is delayed by a half-cycle each time.

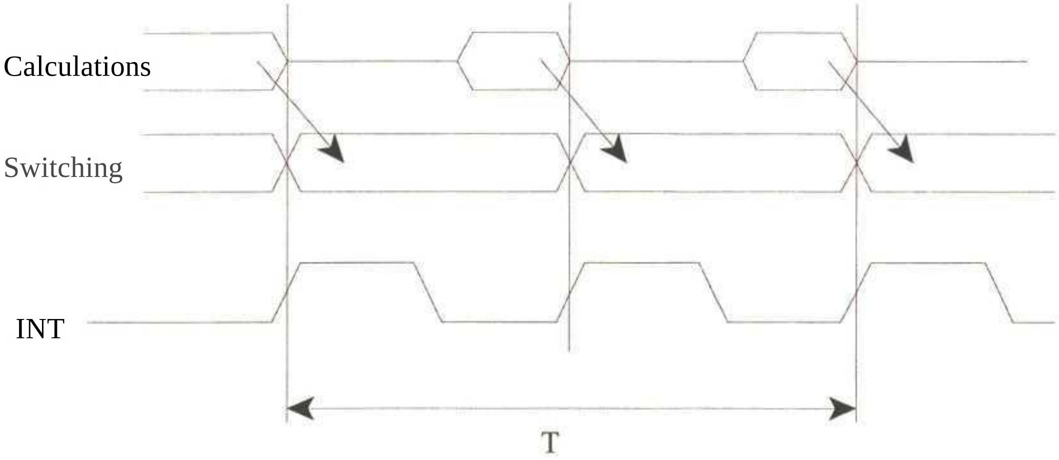


Fig. 4.13 : Calculation and switching cycle timing

4.5.3 Fibre Optic Interface

The six outputs of the PWM IC are used to drive fibre optic transmitters. The use of fibre optic links between the PWM card and the inverter offers a number of advantages over conventional copper wiring:

(i) Electrical isolation - the PC housing the PC32 and PWM/Tacho cards is completely protected from potential faults in the inverter,
(ii) Separate ground - errors due to ground loops are avoided,
(iii) Noise immunity - optic fibre links are inherently immune to induced noise,
(iv) Voltage levels - interfacing problems due to different logic voltage levels on the inverter are avoided.

4.5.4 Tacho ASIC

The TC3005H ASIC [HANNING2] allows simultaneous interfacing for up to two incremental rotary encoders. A typical encoder produces three outputs :

(i) Two sine or square wave signals A and B. These are phase shifted by 90° relative to each other.
(ii) A logic pulse signal, R, indicating a full revolution.

The speed and position information is extracted by counting the number of cycles in signals A and B, the phase difference indicating the direction of rotation. The resolution of an encoder is thus limited by the number of cycles (or quadrants) per revolution. Encoders producing sinusoidal outputs can provide better resolution. The amplitude of the sine signals can be measured thus providing additional position information within each quadrant.

The TC3005H can be connected to two standard resolution encoders as shown in Fig. 4.14. In this configuration each encoder interface is fully independent of the other. Fig. 4.15 shows the connection used for a high resolution encoder. This option requires the use of two external 6-bit ADC devices as well as two comparators to convert the sine signals into square waves. The standard speed and position information can be read out from the second encoder interface while the first one provides additional position information within each quadrant.

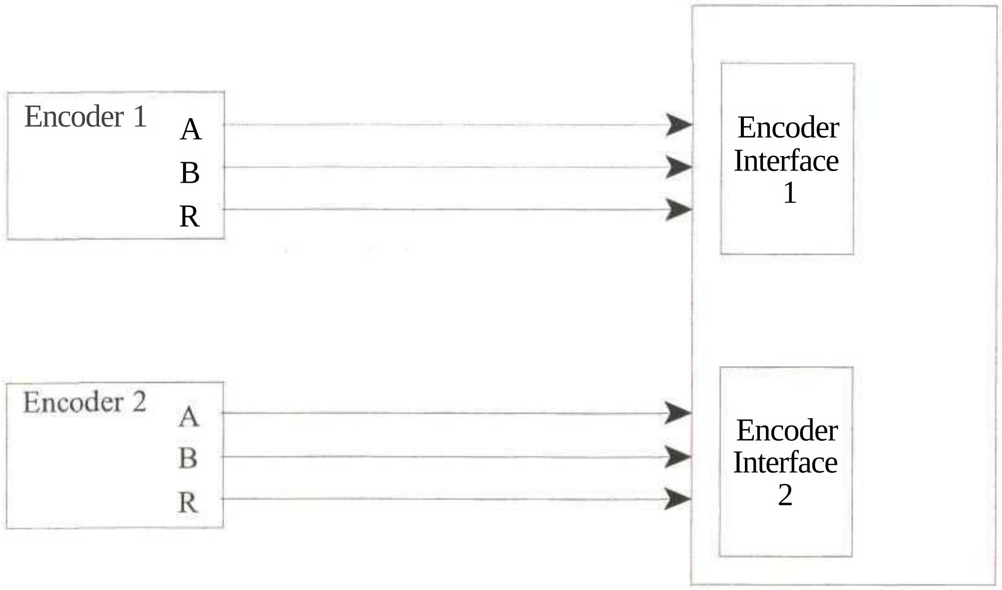


Fig. 4.14 : Interfacing standard resolution encoders

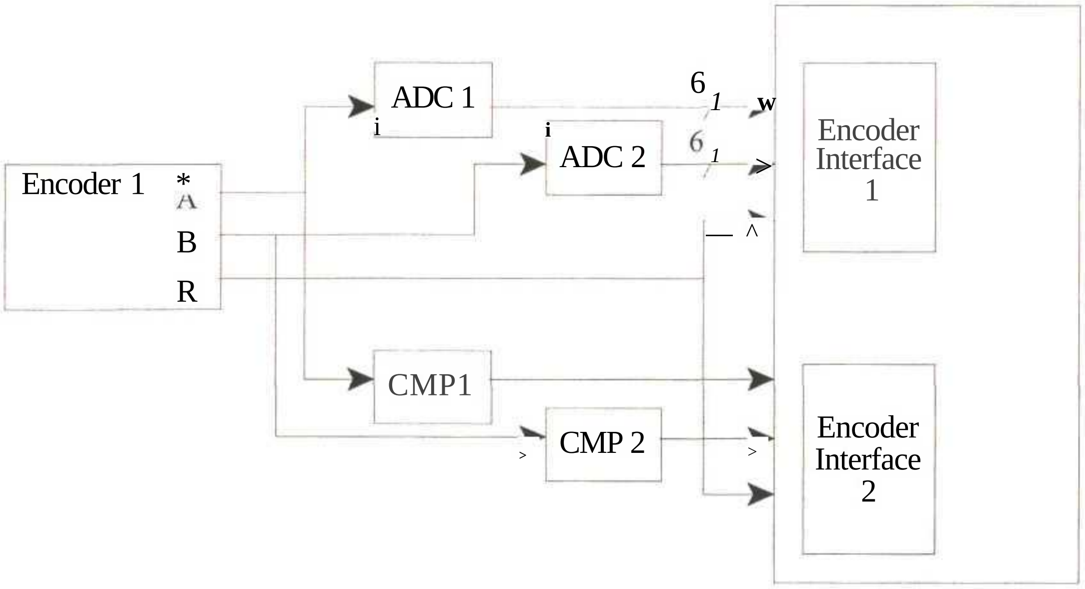


Fig. 4.15 : Interfacing a high resolution encoder

4.6 Conclusion

This Chapter introduced the motion controller hardware platform used by the CSDE. Firstly the TMS320C32 processor which forms the processing core was discussed. Next, the PC32 controller card and its on board peripherals were described. Finally, the custom PWM/Tacho expansion card was introduced. Together these components provide a capable processing platform for implementing digital motion control. Chapters 5, 6, 7 and 8 will now describe how the Simulink and RTW software platform from Chapter 3 was adapted and extended to interact with the target hardware components.

CHAPTER FIVE
CONTROL SYSTEM DEVELOPMENT ENVIRONMENT

5.1 Introduction

In Chapters 1 and 2 a need was identified for an integrated rapid prototyping system for the development of motion control. Subsequently, a number of requirements were set out for the author's proposed CSDE. Chapter 3 introduced Simulink, RTW and the Target Language Compiler from Math Works Inc. as the high-level software tools on which the CSDE is based. The PC32 DSP card as well as custom hardware used during the development of the CSDE were described in Chapter 4. The bulk of the author's work was to integrate the various components and to extend the standard MathWorks products. Features not supported by MathWorks were implemented separately. This Chapter outlines the overall structure of the CSDE and focuses in more detail on the automatic code generation under RTW and TLC.

During the development of the CSDE various components were created in four diverse environments :

(i) GNU make utility
(ii) 'C32 ANSI C
(iii) TLC language
(iv) MS Visual C++

The following sections, as well as Chapters 6, 7, and 8, make reference to code excerpts targeted at these environments. To keep the text readable the complete code listings are placed in the appendices. In particular, sections dealing with the TLC language might not be immediately clear to readers unfamiliar with the RTW development environment. Thus, a practical example is provided in Appendix K in an attempt to clarify the process of transforming a set of TLC files into ANSI C source code. Appendix H offers more detail on the structure and syntax of the TLC language.

5.2 Development of the Global CSDE Structure

The CSDE can be broken down into two distinct parts :

(i) **Host Platform** - A personal computer running the Windows 95 operating system. All algorithm development and simulation takes place here. The host also serves as an interface during real-time prototyping.

(ii) **Target Platform** - The processing platform is a commercial PC32 board from Innovative Integration based around the TMS320C32 processor from Texas Instruments. A dedicated PWM / Tacho card is interfaced to the processor to take away form it the burden of generating power electronics switching waveforms and also provides support for an incremental tachometer.

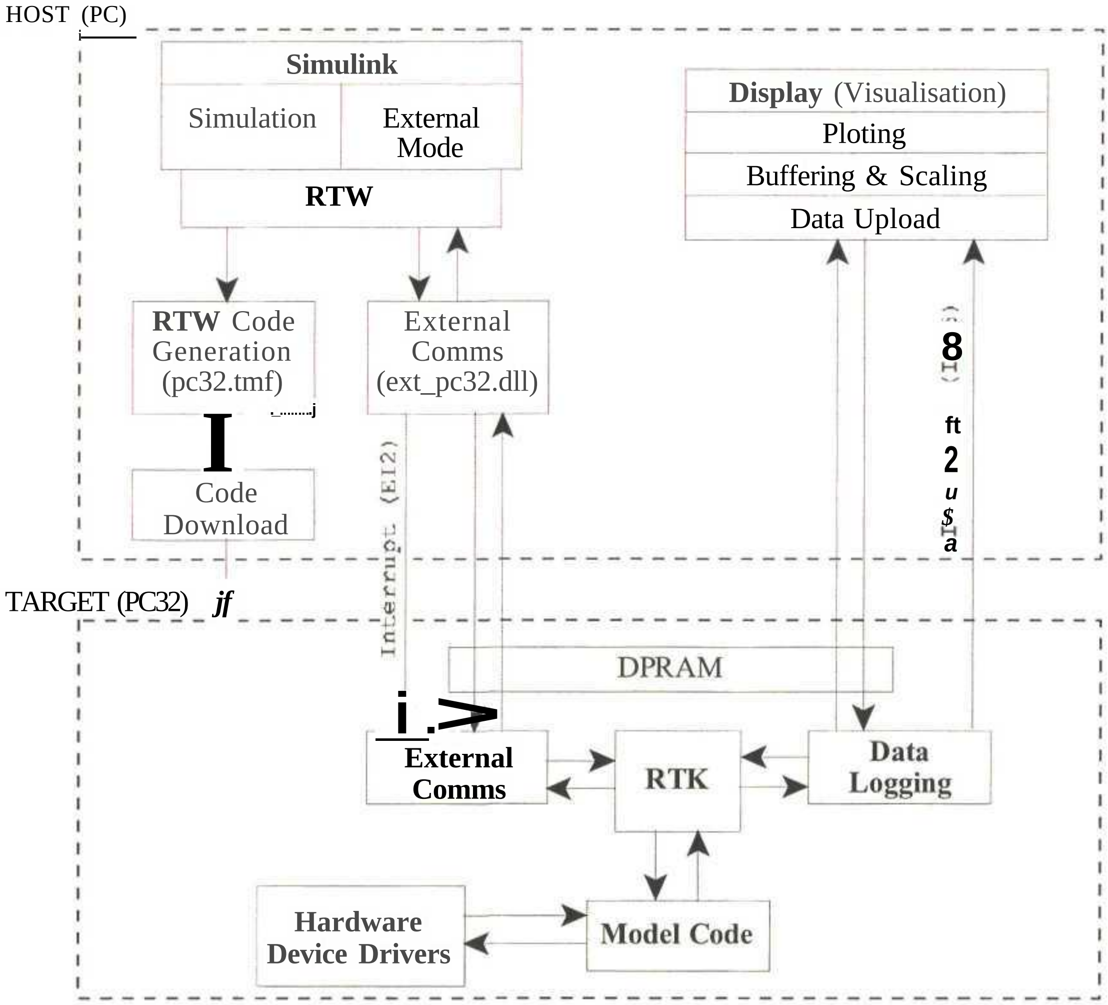


Fig. 5.1 : Logical Diagram of the CSDE

The overall structure of the CSDE is shown in Fig. 5.1. Clearly, the development environment apart from being split over the two physical hardware platforms (host / target), is further divided into a number of distinct logical components. In order to meet the requirements set out in Chapter 2 the author developed the following software components :

(i) Real-Time Code Generation - a Template Make (TMK) File was created which automates the compilation and linking of all necessary components into an executable,

(ii) Real-Time Kernel (RTK) - a basic real-time operating system was written to supervise the execution of generated code on the target PC32 platform,

(iii) External Mode Communication - functions were created to handle both host and target sides of communication. On the host PC a DLL was compiled which is used by Simulink's external mode. On the target, the RTK was extended to receive, process and respond to commands from the host,

(iv) Data Logging - support functions were created to enable the RTK to log and buffer user selected signals and supervise the uploading of the resultant data to the host in real-time,

(v) Hardware Device Drivers - the development of a driver for each hardware device involved both writing source code for the target platform as well as creating a graphical representation for use in Simulink. The resultant Simulink blocks were grouped in a library - *PC32Lib.mdl*

(vi) Code Download - a utility was written which could be called from a DOS batch file and handle downloads to the PC32 target. This replaced the more elaborate utility shipped by II which lacked command line support,

(vii) Visualisation Utility - due to lack of support for plotting of external signals under Simulink ver.2.2, a separate Windows utility was created which uploads data from the target, processes it and plots it in real-time.

As far as the end users of the author's system are concerned the above components are bundled within the CSDE either as executables, compiled libraries, or read-only source code. The components listed above can broadly be grouped according to the development environment in which they were created :

(i) Automated code generation - developed for the GNU make utility, Gmake. This will be introduced further in section 5.3.

(ii) Target Real-Time Support Components - includes the RTK as well as the target side of the external mode communication and data logging. The code was developed using TI 'C32 ANSI C and is introduced in Chapter 6.

(iii) Hardware Device Drivers - most development here took place using the TLC language. Chapter 7 describes this part of the author's work

(iv) Host Support Components and Utilities - includes Simulink's external mode implementation as well as code download and Scope utilities. The development environment here was the MS Visual C++. These components are covered in Chapter 8.

5.3 RTW Code Generation

The generic RTW build process was described in Chapter 3, section 3.5.1. To adapt this process to target the PC32 card, a custom TMK file needs to be provided. The RTW parses the TMK file and produces a final make file specific to the particular build. This final make file contains information about all files needed to compile and link the target executable. It is passed as a command line argument to *gmake.exe* which in turn makes calls to the TI compiler and linker. This process is illustrated in Fig. 5.2. Gmake is a make utility from GNU Software and is distributed as freeware.

5.3.1 Controlling the Build Process

Fig. 5.3 shows the RTW dialog box. The system target file 'ii.tlc' is a global TLC file which configures the MathWorks TLC for generation of ANSI C compliant code. Behind the make command, **make_rtw,** three switches can be specified :

(i) TMRO - 'YES' configures the on-board timer 0 to generate the base sampling rate for the model. Default setting is 'NO', in which case the user is responsible for ensuring correct triggering for all blocks in the model, for example by connecting them to an external interrupt support block.

(ii) UPLD - 'YES' enables data logging support in the generated code. Default setting is 'NO', which excludes all data logging source code from the build process.

(iii) 10 - this option is included mainly for debugging purposes. 'ENABLE' causes information to be printed out to the II terminal emulator at run time. The serial

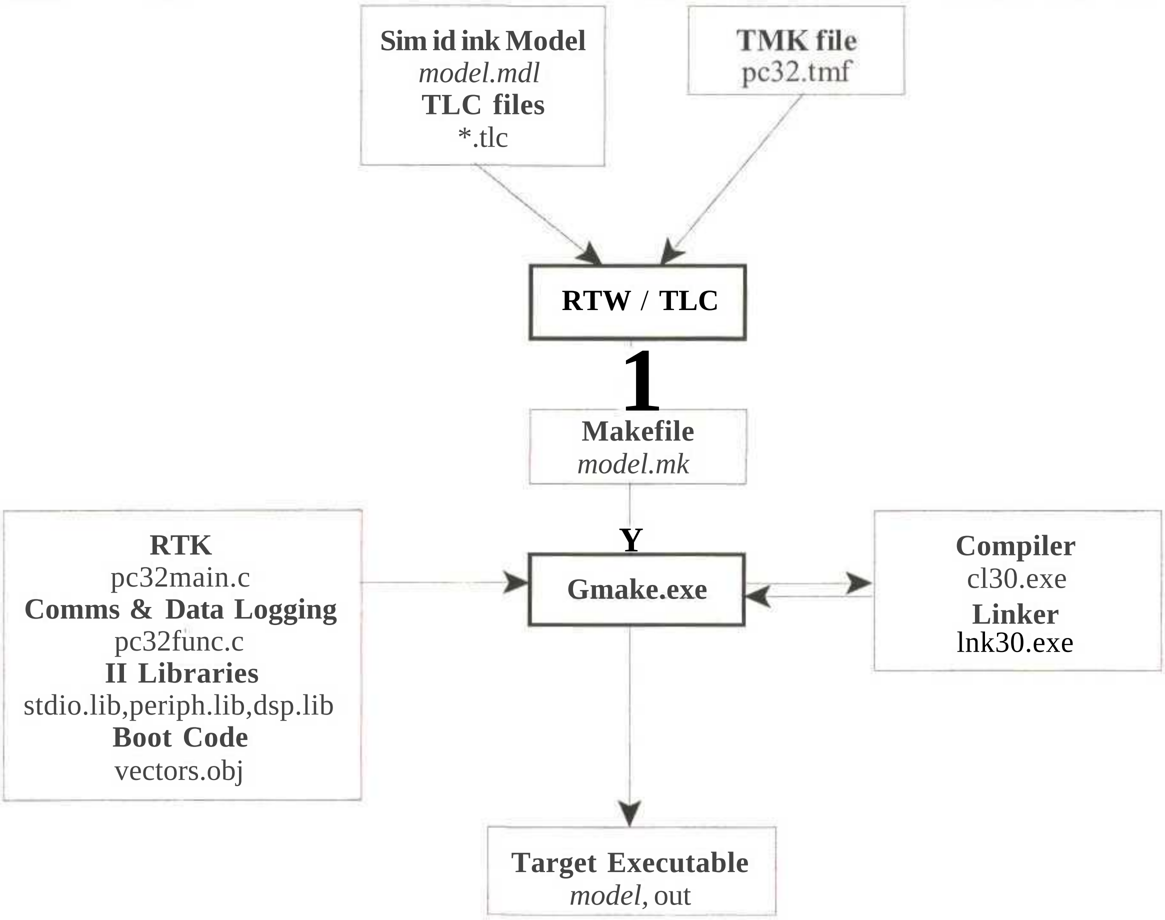
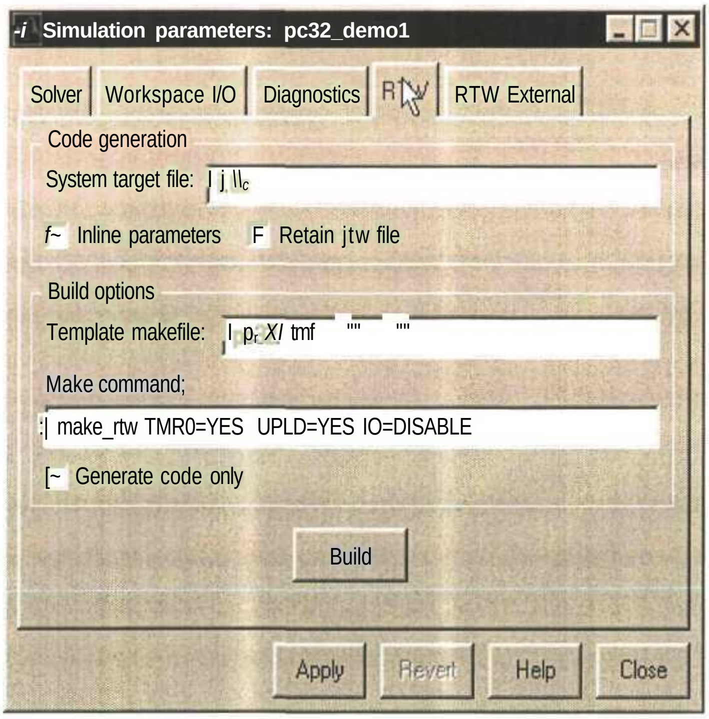


Fig. 5.2 : CSDE Build Process



Code generation

System target file: | j \\c

f~ Inline parameters F Retain jtw file

Build options

Template makefile: | pr XI tmf "" ""

Make command;

:| make_rtw TMR0=YES UPLD=YES IO=DISABLE

[~ Generate code only

Fig. 5.3 : Simulation Parameters | RTW

printing of debug strings slows down the code execution on the PC 3 2 target considerably. The default setting is 'DISABLE'.

During the build process the switches are translated into defines which control the compilation of source code. The following excerpt from the *pc32.tmf* file translates the switches and includes them as define (-d) command line parameters for the TI compiler :

```
... -dIO_$(10) -dTMR0_$(TMRO) -dUPLD_$(UPLD) ...
```

The following lines of source code demonstrate how the **IO** switch is used to control the text output to the terminal emulator :

```
#ifdef IO_ENABLE
   printf("Printing debug info to terminal....\n" );
#endif
```

If **IO_ENABLE** is not defined the **printf** statement will be excluded from the compilation process. In this fashion the defines control which blocks of source code are compiled.

5.3.2 Structure of the Template Make File

In Appendix D a full listing of the *pc32.tmf* file can be found. The TMK file has a fixed structure, as follows :

(i) General Defines - global definitions and settings.

(ii) Customization Macros - RTW passes a number of parameters which are substituted here into defines. These parameters later control the compilation of model source code,

(iii) II PC32 Definitions - defines and path settings for II components. The download utility is also specified here,

(iv) TI Tools - paths to the TI compiler and linker.

(v) Include Path - paths to directories containing all necessary libraries and header files,

(vi) Compiler Flags - command line options for the compiler and linker,

(vii) Source Files - all necessary source code files are listed here,

(viii) Exported Environment Variables - due to the command line length limitation in DOS, parameters to be passed to the compiler and linker are assigned to environment variables,

(ix) Compile and Link Rules - Gmake rules for compiling and linking source code.

(x) Rule for Downloading to Target - commands to be performed after a successful build,

(xi) Dependencies - gmake dependencies. Based on this relationship gmake decides whether it is necessary to recompile and re-link the target executable.

The TMK file, after being parsed by the RTW, carries all information necessary for Gmake and the TI tools to generate the final target executable. The main purpose of the TMK is to list additional source files, for example the integration algorithm, which need to be linked to the RTK in order to replicate the Simulink diagram exactly in software.

5.4 Conclusion

This Chapter started by introducing the overall structure of the CSDE and subdividing it into a number of logical components to be introduced further in Chapters 6, 7 and 8. The process of automatic code generation with the RTW and TLC was described.

CHAPTER SIX
TARGET REAL-TIME SUPPORT COMPONENTS

6.1 Introduction

This Chapter describes components of the CSDE designed to provide support for the automatically generated code in real-time on the PC32 controller. Three sections fall into this category, namely :

(i) RTK - the kernel controls and supervises all execution on the PC32 target platform,

(ii) External Mode Communication - this component is responsible for receiving and processing of commands from Simulink's external mode DLL.

(iii) Data Logging - this set of routines handles data buffered by the Scope Channel blocks (introduced in Chapter 7). The data is arranged into packets and sent to the Display utility for visualisation in real-time.

The above components form the PDL as described in Chapter 3, section 3.5.2. During the automatic code generation process they are linked with the code generated from the Simulink model to create a single executable for downloading to the target. The following sections will introduce each of the three components in turn and show how they fit together.

6.2 The Real-Time Kernel

The RTK is a basic real-time operating system designed to run on the TI 'C32 processor. It controls the execution of the target side CSDE components on the PC32 platform. A full listing of the RTK source code can be found in Appendix E. Math Works provide a generic RTK as well as a host of RTKs specifically tailored to a variety of target platforms. However, none of the generic modules support the II environment based around the 'C32 processor and the author decided to develop his own RTK.

6.2.1 RTK Initialisation

Once the build process is complete, the download utility places the compiled target executable in RAM on the PC32 card and resets the 'C32 processor. Execution starts in the boot strap image *vectors.bin* provided by II. After this low-level initialisation the boot code hands the control over to RTK's **main** function. The first few lines of code perform calls to II library functions:

```
dpram =  (volatile uint32_T*)&Periph->Dpram;

MHZ = detect_cpu_speed();
timer(0, 0);
enable_cache();

enable_monitor();
```

Firstly the DPRAM address is configured. The variable **dpram** is used for all accesses to the DPRAM. The PC32 can be configured with a number of crystals, thus the board speed is not hardcoded but is detected dynamically. The **MHZ** variable is defined globally in the II header files and is used in all timing calculations. The call to **enable_cache** enables the use of the built-in 'C32 hardware caching algorithms. The on-board monitor allows the use of functions to print to the II terminal emulator and is initialised via the call to **enable_monitor.** After this low-level initialisation, the following lines perform calls to Math Works' functions :

```
S = MODEL() ;

ssSetTFinal(S,0.0);  /* run forever */
MdllnitializeSampleTimes();

rt_InitTimingEngine(S) ;
rt_CreateIntegrationData(S) ;
```

The variable S is a pointer to type *SimStruct,* a Simulink defined data structure containing all model settings and parameters. A call to **MODEL** dynamically reserves memory for and initialises the *SimStruct.* It also returns a pointer to the global *SimStruct,* and later this pointer is used for direct access to model parameters in the structure. **MODEL** is a define which is substituted with the actual Simulink model's name by RTW during the build process. Setting model's stop time to zero in the call to **ssSetTFinal,** means it will execute forever. This option is hard-coded since it is uncommon to limit the run period of a real-time controller. Simulink's external mode allows models to be suspended and restarted from within Simulink GUI as will be described in section 6.3 and in Chapter 8.

The call to **MdllnitializeSampleTimes** is part of the Simulink S-function API as introduced in Chapter 3. **MdllnitializeSampleTimes** sets the sampling period for the model to the value entered by the user under Simulation Parameters in Simulink. The call to **rt_InitTimingEngine** initialises the timing data structures in the *SimStruct.*

If there are any continuous time blocks in the Simulink block diagram the RTW will reflect the number of continuous time states with the define **NCSTATES.** The following compiler directive at the top of 'pc32main.c' evaluates depending on the value of this define :

```
#if NCSTATES > 0
      extern void rtCreatelntegrationData(SimStruct *S);
      extern void rtUpdateContinuousStates(SimStruct *S);
#else
      #define rtCreatelntegrationData(S)
                        ssSetSolverName(S,"FixedStepDiscrete");
      #define rt_UpdateContinuousStates(S)
                        ssSetT(S,ssGetSolverStopTime(S));
#endif
```

Thus, if there are no continuous time blocks **NCSTATES** is set to zero and all calls to **rtCreatelntegrationData** are redirected and hard-coded to set the solver to "FixedStepDiscrete". During RTK initialisation the call to **rtCreatelntegrationData** will either setup the continuous time integration data structures in *SimStruc* or it will set the execution mode to be purely discrete.

Only a portion of the *SimStruc* is dedicated to holding model parameters. To speed up access to the parameter array a dedicated pointer, **param** is defined. A call to the macro **ssGetDefaultParam** returns the address of the parameter array within *SimStruc :*

param = ssGetDefaultParam(S);

The above sections of the RTK code perform the hardware initialisation and also setup the Math Works defined structures. Thereafter, the model code is ready for execution.

6.2.2 RTK Execution Loop

After initialisation the RTK enters **an** infinite while loop. Fig. 6.1. shows the logical flow diagram of the main loop.

```
while (1)
{
    ....

      Host command processing

    ....


    if (host_start)
    {
      ttifdef     UPLD_YES
           ServiceUploads();
      ttendif

      ttifndef    TMR0_YES
           MdlOutputs(O);
           MdlUpdate(O);
      #endif
    }
}
```

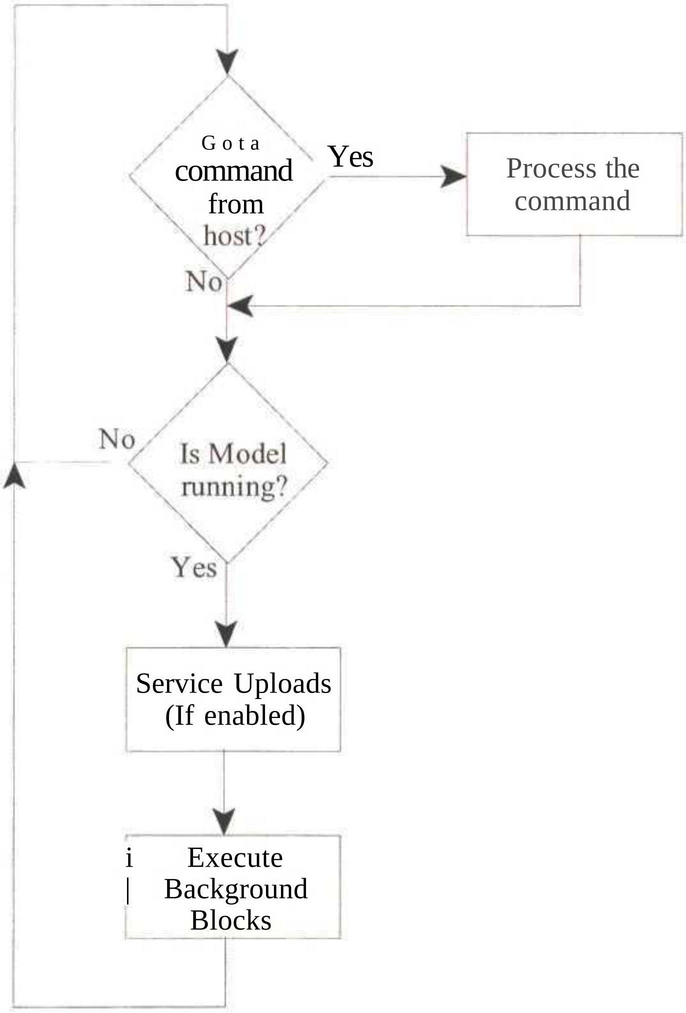


Fig. 6.1 : RTK Execution Loop flow diagram

The RTK main loop is effectively a background process as no attempt is made to prevent interrupts of any priority from interrupting its execution. Within the loop, however, the following subsections have equal priority :

(i) Processing of commands from host
(ii) Servicing of upload buffers
(iii) Executing code for blocks in the root of Simulink's diagram

Processing of commands from the host forms a large part of the RTK main loop and is outlined separately in section 6.3. Likewise, data uploading is discussed in section 6.4. The compiler **#ifdef** directives make use of defines introduced in Chapter 5 to control the remainder of the loop. If no data uploading is selected the call to the relevant **ServiceUpIoads** function will be omitted. Likewise, **MdlOutputs** and **MdlUpdate** will only be called if the calls are not already attached to the timer 0 interrupt.

Due to the low priority of the main loop, its execution frequency cannot be guaranteed and the model update functions called from the loop will be called at irregular intervals. Thus, any blocks which depend on execution frequency are likely to produce results inconsistent with simulation if executed in the background loop. Diagrams containing blocks like that should be executed using interrupts as described in the following section.

6.2.3 RTK Timer Interrupt

The only way the RTK can guarantee a fixed execution rate for the model is to call the update functions from an interrupt service routine (ISR). The ISR can be triggered either by a timer or by a hardware interrupt via the PC32 Int Support block, described in Chapter 7. If **TMR0_YES** is defined the following code is included during the external mode initialisation and attaches the **rtOneStep** function to timer 0 interrupt:

```
#ifdef TMRO_YES
  install_int_vector(rtOneStep, 9);
  enable_interrupt (8);
  timer (0, (int)(1.0 / ssGetStepSize(S)));
 .
 .
 .
#endi£
```

Timer 0 period is set up with a call to the **ssGetStepSize** macro which returns the model time step in seconds. On each overflow of timer 0 interrupt 8 is triggered and the following ISR executes:

```
static void rtOneStep(void)
{
    real_T tnext;

    tnext = rtGetNextSampleHit();
    ssSetSolverStopTime(S,tnext);

    MdlOutputs (0) ;
    MdlUpdate(O);

    rtUpdateDiscreteTaskSampleHits(S);

    if (ssGetSampleTime(S,0) == CONTINUOUS_SAMPLE_TIME)
    {
        rtUpdateContinuousStates(S);
    }
} /* end rtOneStep */
```

The above ISR performs a complete update of the model code. The variable **tnext** is used to set the step time for the integration routines. The code as above should perform correctly even if the model update frequency changes dynamically, however, the CSDE currently only supports fixed model step times.

As with any ISR it is necessary to ensure that the processor context is not changed after the routine terminates. The TI compiler automatically generates code to save all registers used by the ISR on the stack on entry to the routine and also to restore them on exit. The requirement for this mechanism is that the ISR naming has to comply with the following convention [TEXASINSTR1]:

```
c_intXX
```

Where XX is a two digit decimal number in the range 01 to 99. To ensure that all references **to** the **rtOneStep** function comply with the ISR naming convention the following define is included :

```
#define     rtOneStep   c_intl9
```

6.2.4 Model Code in the RTK

The above sections described the operation of the RTK. The kernel ensures the correct execution of the model code generated directly from the Simulink diagram and allocates remaining processing time to background tasks. The model code will be concentrated in a number of routines conforming to the S-fiinction API as described in Chapter 3, section 3.4.6. The RTK then interacts with the model code via this API. The following two sections will introduce the External Mode Communication and Data Logging as the two background tasks in the RTK.

6.3 External Mode Communication

After a code is generated from a Simulink block diagram and downloaded to the PC32 target, Simulink can be placed in external mode, as described in Chapter 3, section 3.4.7. In this mode a communication channel is established between the executable on the target and Simulink. This communication link between the host PC and the PC32 target is physically implemented via the DPRAM. The user can then utilise the Simulink GUI to issue commands and modify parameters online in the controller executing on the DSP card.

Appendix B shows the details of memory mapping and protocols used in implementing the communication link via the DPRAM. A list of the possible command codes which the RTK can interpret is shown in Table 6.1. For a reliable communication link two sets of functions are necessary :

(i) On the host PC - a DLL is needed to implement Simulink's external mode requests across the DPRAM. This component is introduced in Chapter 8.

(ii) On the target PC32 - functions have to be included into the RTK to receive and process commands from the host as outlined in the following section.

Command Code	Action
1	Start Model Execution
2	Single Parameter Update
3	Suspend Model Execution
4	Verify Checksums
5	Initialise Data Logging
6	Suspend Data Logging

Table 6.1 : List of External Mode commands

6.3.1 Target Side External Communications

Processing of commands received by the target from the host is done in two steps within the RTK as shown in Fig. 6.2 :

(i) An ISR receives the command and buffers it.

(ii) The buffered command is decoded and processed within the RTK background loop.

During the RTK initialisation interrupt El 3 is trapped for host command reception and mapped to an ISR :

```
install_int_vector(PCO_int, 4);
enable_interrupt(3);
```

The routine **PC0_int** below checks if a valid synch value was placed in DPRAM by Simulink's DLL, extracts the command byte and places it in a global buffer variable **dprx :**

```
static void PCOintO
{
      /* wait for the sync signal from host*/
  if ((lintflag) && ((dpram[0] & OxFFFO)==0xAAA0))
  {
      int_flag=l;
  }   dprx=dpram[O] & OxOOOF;}      /* extract the command byte */
}
```

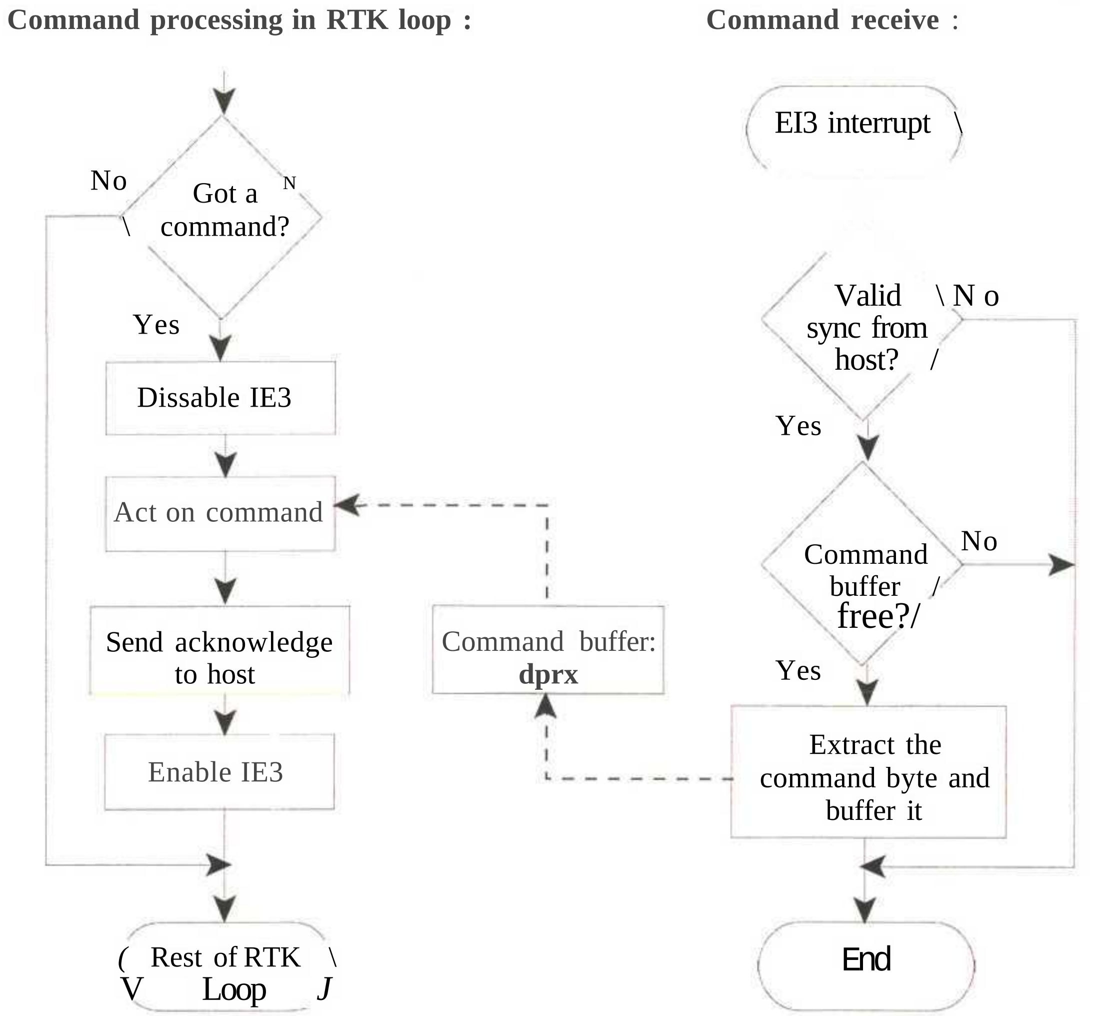


Fig. 6.2 : Flow diagram of the command receive and processing on target

The command processing forms part of the background process and is integrated into the RTK loop as shown in Fig. 6.1. The commands are decoded in a switch statement, as the following source code excerpt demonstrates :

```
disable_interrupt(3);
if  (int_flag)
{
 switch  (dprx)
 {
   case 1:  /* command to start execution */
      .
      .
      .
 }
 dpram[0] = OxABCO + dprx;  /* send an acknowledge to host */
 int_flag = 0;  /* ready for a new command */
}
enable_interrupt(3);    /* lets listen out for comms from host */
```

The switch statement has six case subsections each implementing a single command code as listed in Table 6.1. Command code 1 performs model initialisation by calling **MdlStart** and if necessary attaches the **rtOneStep** ISR to the timer 0 interrupt. The setting of the three build switches introduced in Chapter 5 is reported back to the host.

Command code 2 reads two values from DPRAM :

(i) At offset 0x0001 the index of the parameter to be written to *SimStruc*
(ii) At offset 0x0002 the value of the parameter in IEEE floating point format

The value read out from DPRAM is then converted to TI floating point format via a call to the function **from_ieee** in II library :

```
    rxmb = dpram[l];
    param[rxmb] = from_ieee(dpram[2] ) ;
```

Command code 3 executes the following code to suspend execution :

```
    disable_interrupts();

  #ifdef TMRO_YES
        /* stop timer 0 int only if it was used */
    disable_interrupt (8);
  #endif

    MdlTerminateO ;
    enable_interrupts();
    host_start=0;
```

All interrupts are disabled globally to ensure that no part of the model code is running while the call to **MdlTerminate** is performed. **MdlTerminate** destroys global data structures and it is imperative that these structures are not accessed while the termination process is active. Setting host_start to 0 disables any model updates in the RTK background loop.

Command code 4 reads four checksums placed by the Simulink DLL in DPRAM and compares them to the local copies :

```
ChecksumOK=l;

if (dpram[l] != ssGetChecksumO(S)) ChecksumOK=0;
if (dpram[2] != ssGetChecksuml(S)) ChecksumOK=0;
if (dpram[3] != ssGetChecksum2(S)) ChecksumOK=0;
if (dpram[4] != ssGetChecksum3(S)) ChecksumOK=0;

if (ChecksumOK)
{
 #ifdef IOENABLE
    printf("OK!\n");
 #endif
}
  else
{
  dprx=0;
 #ifdef IOENABLE
    printf("FAILED!\n");
}ttendif
```

If a discrepancy is found between local checksums and the ones sent from the host, an error is signalled back to the host. This mechanism is implemented in order to verify that the correct version of generated executable is present on the target before model execution can be started.

Command codes 5 and 6 (upload start and stop) are currently not used since external mode under Simulink 2.2 does not support data logging. The uploading of data is started automatically during data queues initialisation.

6.4 Data Logging

The Scope Channel blocks continuously queue up data in circular buffers as described in Chapter 7, section 7.5.6. In the background RTK process loop the function **ServiceUploads** is called to monitor the queues and to send the buffered data up to the host when full packets become available. Appendix E lists the source code for the data upload functions contained in *pc32func.c* and Fig. 6.3 shows a flow chart of the data transfer process on the target. The standalone Display utility on the host PC monitors interrupt IRQ 5, uploads data from the DPRAM and displays it as it becomes available. The Display utility is introduced in Chapter 8.

The following code excerpt from the **ServiceUploads** function first checks that data logging is enabled. Next, a flag at offset 0x0004 in DPRAM is tested to make sure the host has finished reading the previous set of packets. The while loop then steps through each active queue and a packet is copied into the DPRAM provided two conditions hold true :

(i) There is enough data buffered in the queue to fill a packet, and

(ii) There is sufficient free memory left in DPRAM to hold the resultant packet.

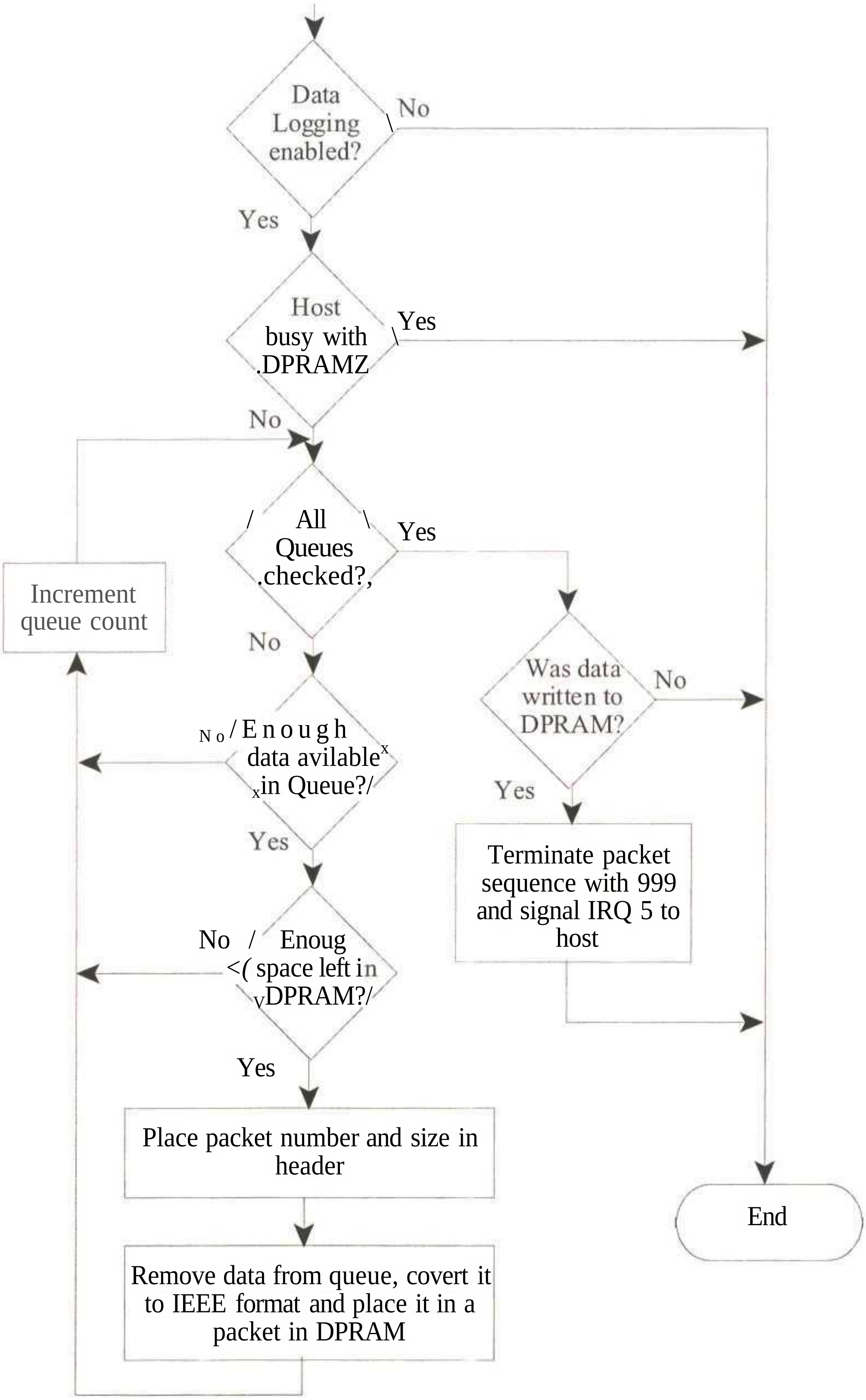

Data Logging enabled?
No
Yes
Host busy with DPRAM?
Yes
No
All Queues checked?
Yes
Increment queue count
No
Was data written to DPRAM?
No
Enough data avilable in Queue?
No
Yes
Terminate packet sequence with 999 and signal IRQ 5 to host
Enoug space left in DPRAM?
No
Yes
Place packet number and size in header
End
Remove data from queue, covert it to IEEE format and place it in a packet in DPRAM

Fig. 6.3 : Flow diagram of the Data Upload mechanism on target

```
if ((LogData) && !(dpram[4] & 1))
{
    while ((num<NumQueues) && (offset<MAXOFFSET))
    {
      if ((enqueued(&queue[num]) >= buffer_size[num]) &&
            (buffer_size[num]<MAXOFFSET-offset))
      {
         dpram[BASE_DPRAM + offset++] = channel_map[num];
          /* put queue number in dpram */

         dpram[BASE_DPRAM + offset++] = buffer_size[num];
          /* put queue size in dpram */

         for (xxx=0; xxx<buffer_size[num]; xxx++)
         {
          dpram[BASE_DPRAM + offset++] =
                to_ieee(*(volatile float*)dequeue_ptr(&queue[num]));
         }

         dpram_full = TRUE;
      }

      num++;
    }
}
```

The II library call to_ieee takes a TI floating point type as an argument and returns a 32-bit value formatted as a IEEE floating point. After all queues are checked and if any data was placed in DPRAM the packet sequence is terminated with the value 999 and the host is notified that data is available with and interrupt on IRQ 5 :

```
if  (dpram_full)
{
     dpram[BASE_DPRAM + offset] = (int)999;
     dpram[4] += 1;
     host_interrupt ();
}
```

The function **Clear AllQueues** is also included in *pc32func.c*, it is called during the model start procedure to clear and initialise the queues in memory.

6.5 Conclusion

This Chapter introduced the RTK as well as its background tasks. External mode communication forms one of the background processes and handles all commands issued by Simulink via its external mode DLL, as introduced in Chapter 8. A second background task monitors the data queues of all Scope Channel blocks and sends the resultant data packets to the Display utility on the host PC via the DPRAM. The Scope Channel block is introduced with the other CSDE hardware device driver blocks in Chapter 7, while Chapter 8 describes the standalone Display utility.

CHAPTER SEVEN
HARDWARE DEVICE DRIVERS

7.1 Introduction

This Chapter introduces the practical issues of developing the hardware driver blocks for use in Simulink. All source code for the hardware drivers is included in the Simulink diagrams in the form of inlined S-functions. This method of including custom source code into the RTW build process requires three components for each device driver :

(i) A DLL file for use by Simulink in the graphical model representation

(ii) A Simulink block mask to make entering parameters for the driver more intuitive.

(iii) A TLC file describing how to inline the actual source code for the driver during the RTW build process.

Based on the above points, the steps in creating a hardware driver block for Simulink will be described in the following sections. Each of the CSDE driver blocks will be treated individually in Section 7.5.

7.2 Generating a Driver Block

The process of creating an inlined S-function block is illustrated in Fig. 7.1. The block's DLL file does not contain any useful code but serves simply as a guide for the block's graphical appearance in a Simulink block diagram. It serves to indicate to Simulink the number of inputs, outputs and parameters for the driver block. The block's mask is the user interface and is included for blocks which require user interaction. Creating a mask is further discussed in section 7.3. All the actual source code for a driver is placed in the corresponding block's TLC file to be inlined during the build process. Section 7.4 introduces this.

The DLL file for each of the CSDE driver blocks is derived from a blank S-function template. The source code for a typical driver DLL file is listed in Appendix F. The TLC source code files for the various driver blocks are listed in Appendix G. The following two sections will describe a number of issues common to all driver blocks.

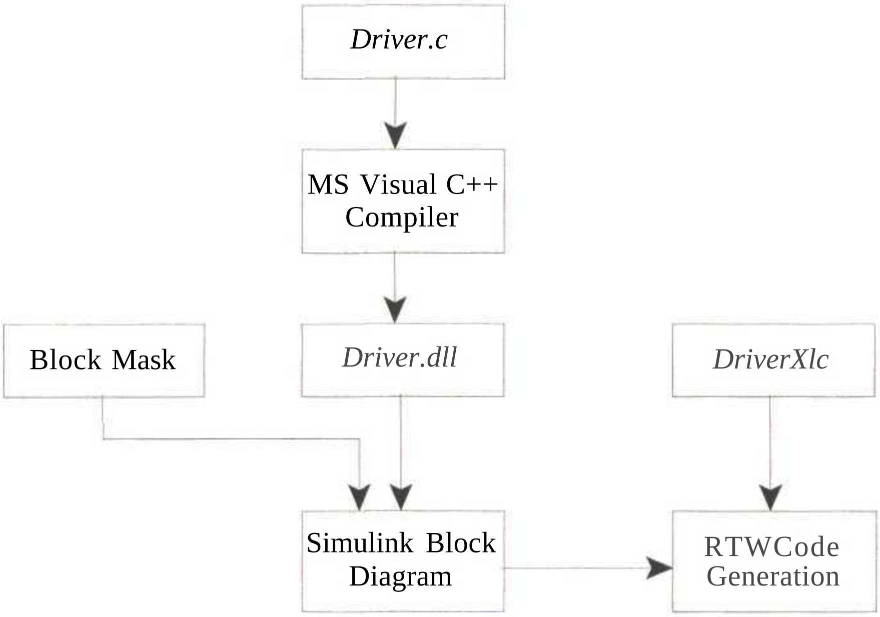
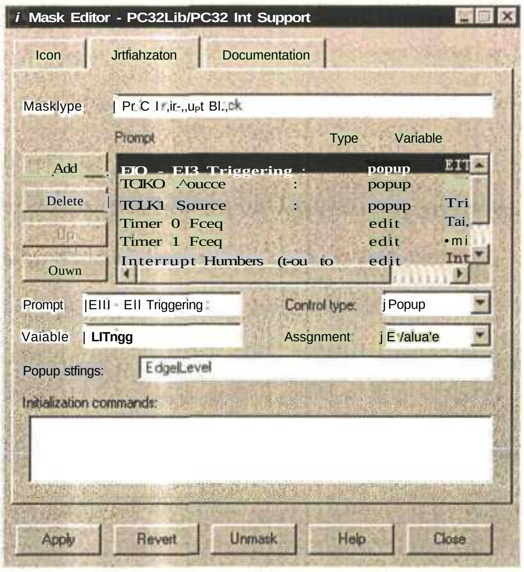


Fig. 7.1 : Process in development of hardware drivers

7.3 Driver Block Masking

The ability to mask Simulink blocks and subsystems allows creation of custom user interfaces, as described in Chapter 3, section 3.4.5. A well designed block mask can provide additional information to users and provide help with entering parameters. The Simulink Block Mask Editor is shown in Fig. 7.2, the Interrupt Support block's mask is used as an example. The 'Initialisation' tab of the editor allows the design of a custom parameter entry interface. Under



Masklype | Pr C I r,ir-,,upt Bl.,ck

Prompt	Type	Variable
EIO - EI3 Triggering :	popup	EIT
TCIKO .^oucce :	popup	
TCLK1 Source :	popup	Tri
Timer 0 Fceq	edit	Tai,
Timer 1 Fceq	edit	•mi
Interrupt Humbers (t-ou to	edit	Int

Prompt |EIII - EII Triggering

Control type: j Popup

Vaiable | LITngg

Assgnment j E /alua'e

Popup stfings: EdgelLevel

Initialization commands:

Fig. 7.2 : Simulink Block Mask Editor

the 'Documentation' tab a brief help screen for the block can be created. The resultant user interfaces for the driver blocks which have masks are shown in section 7.5. A step-by-step description of the masking process is well documented in the MathWorks literature [MATHW0RKS2, 3,4].

Only a DLL file is necessary to incorporate an S-function block into a Simulink block diagram. A corresponding mask is optional. However, for the RTW to be able to generate code for the driver correctly, a TLC file needs to be provided. The following sections describe this.

7.4 Inlining Driver Code

The TLC splits code generated for a model into five functions, which can be grouped into three sections :

(i) Initialisation - **MdlStart**
(ii) Model Step- **MdlOutputs,**
MdlUpdate,
MdlDerivatives
(iii) Termination - **MdlTerminate**

The above functions conform to the S-function API, as described in Chapter 3 in section 3.4.6. The TLC file for each individual driver block can specify source code to be placed into any of these pre-defined global functions, or it can create additional functions. Definitions and variable declarations can also be placed into a common model header file. A more in-depth discussion of the TLC and its command set is offered in Appendix H.

Mining the source code for a driver is advantageous to including it as a separate module as discussed in Chapter 3, section 3.5.5. The inlined code executes faster since there is no unnecessary function switching overheads, thus improving overall system performance. Also the final code becomes more compact and readable as it is no longer split over multiple source code and header files. The following section will describe the development of the individual block TLC files in more detail.

7.5 Driver Block TLC Files

The following driver blocks are provided with the CSDE for operation with the PC32 and expansion cards as introduced in Chapter 4 :

(i) AD TRIGGER - allows software triggering of the individual ADC channels.

(ii) PC32 ADC - support for the four onboard ADC channels, including scaling of the inputs to the ± 1OV range,

(iii) PC32 DAC - support for the four onboard DAC channels, including scaling of the outputs to the ± 10V range,

(iv) PC32 Int Support - allows mapping of various interrupt signals on the PC32 to triggered subsystems on the Simulink diagram. This block's mask also provides a way to configure a number of interrupt related registers in the 'C32 processor,

(v) PWM Block - provides an interface to the PWMIC on the custom expansion card,

(vi) Scope Channel - this block buffers its inputs and sends the resultant data via the DPRAM to the Scope utility for displaying.

The following sections will discuss each of the above blocks individually in more detail. For convenience all CSDE driver blocks are grouped into a Simulink library, shown in Fig.7.3. One component which is clearly missing from the library is a driver block to support the tachometer interface on the custom expansion card. The high resolution tachometer hardware necessary to develop and fully test support for the tachometer interface only became available very late into the author's project. Although a driver for a high resolution tachometer was designed to support one of the students' projects described in section 9.5, the driver block has not formally been tested and included into the CSDE by the author.

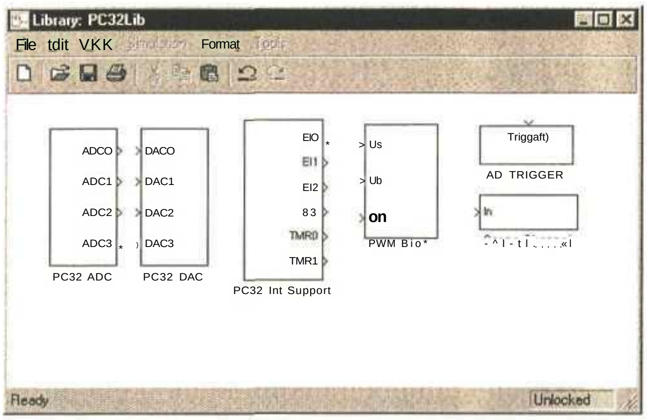


Fig. 7.3 : PC32 hardware driver block library

Some of the driver blocks can only be used once in a given model block diagram. Multiple instances of source code for these blocks could cause unpredictable results :

(i) PC32 ADC

(ii) PC32 DAC

(iii) PC32 Int Support

(iv) PWM Block

The TLC files for those blocks report an error should there be multiple instances found during the TLC parse. The following TLC code excerpt illustrates an example of the error trapping mechanism :

```
%function BlocklnstanceSetup(block, system) void

 %% Only allow 1 instance of the A/D block
 %if .'EXISTS("Rt_pc32ad")
   %assign ::Rt_pc32ad = 1
 %else
   %error Only 1 PC32adn block is allowed in the model.
 %endif

%endfunction %% BlocklnstanceSetup
```

While the above discussion was applicable to all CSDE driver blocks, the following sections will describe each individual block in turn.

7.5.1 ADC Trigger Block

The ADC Trigger block is designed to be attached to one of the outputs of the asynchronous interrupt support block. When triggered, a write is performed to each of the four memory mapped ADC devices starting a conversion :

```
*(ADC0)=0;
*(ADC1)=O;
*(ADC2)=0;
*(ADC3)=0;
```

In Chapter 4 the memory mapping of the various peripherals was discussed. The addresses **for** the ADC devices on the PC32 card are defined as follows in the header file :

```
#define ADCO (volatile int*) 0x810000
#define ADC1 (volatile int*) 0x810800
#define ADC2 (volatile int*) 0x811000
#define ADC3 (volatile int*) 0x811800
```

This block is particularly useful when sampling of the ADC inputs needs to be synchronised with an external trigger signal. In motion control applications, the associated power electronics switches can introduce substantial amount of noise into the current and voltage feedback signals. However, the effect of the switching noise can be eliminated if the signals are sampled during periods when the switches are inactive. The PWM IC provides a trigger signal in the centre of each switching interval and using it to trigger sampling of the feedback channels ensures minimum interference from switching noise. In Chapter 9 an example is presented which illustrates the use of the AD TRIGGER block.

7.5.2 ADC Input Block

The PC32 ADC block reads the result of the last conversion from the onboard ADC, latches and makes the values available at its output. The ADC reads are performed via calls to the II library:

```
%<LibBlockOutputSignal(0,"","",0)>=read_adc(BASEBOARD, 0)/(3276.7);
%<LibBlockOutputSignal(0,"","",1)>=read_adc(BASEBOARD, l)/(3276.7);
%<LibBlockOutputSignal(0,"","",2)>=read_adc(BASEBOARD, 2)/(3276.7);
%<LibBlockOutputSignal(0,"","",3)>=read_adc(BASEBOARD, 3 ) / (3276.7);
```

The TLC translates the <LibBIockOutputSignaI> macros into variable names during its parse process. Calls to the **read_adc** function return a 16-bit signed integer value representing the full range of the ADC. For convenience the range is scaled down to ± 1OV. No loss of resolution occurs since the variable on the lefthand side of the equal sign is of 32-bit floating point type.

7.5.3 **DAC Output Block**

On the PC32 a write to the onboard DAC devices takes two steps. Firstly the value is written to the memory mapped latch. Conversion only takes place after a subsequent memory write to a control register. The write_dac and **convert_dac** function provided in the II library provide an elegant way to perform these steps :

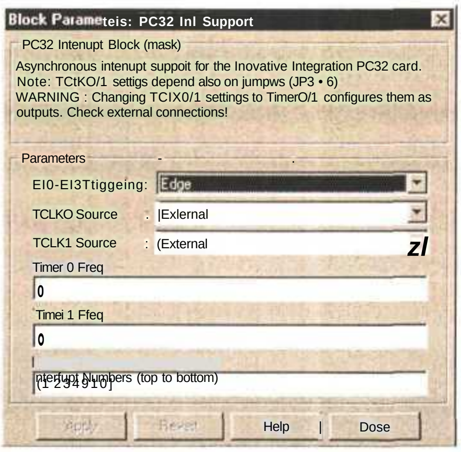
```
write_dac(BASEBOARD, 0, %<LibBlockInputSignal(0,"","",0)>*(327 6.7));
convertdac(BASEBOARD, 0 ) ;
write_dac(BASEBOARD, 1, %<LibBlockInputSignal(0,"","",1)>*(3276.7));
convert_dac(BASEBOARD, 1 ) ;
write_dac(BASEBOARD, 2, %<LibBlockInputSignal(0,"","",2)>*(3276.7));
convert_dac(BASEBOARD, 2 ) ;
write_dac(BASEBOARD, 3, %<LibBlockInputSignal(0,"","",3)>*(3276.7));
convert_dac(BASEBOARD, 3 ) ;
```

Before being written the values are scaled up form the ± 1OV range used internally by the model. On termination of model execution it is important to zero the DAC outputs. For this purpose the following code is inlined into the **MdlTerminate** function :

```
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);
```

7.5.4 Asynchronous Interrupt Support Block

The PC32 Int Support block allows subsystems on the Simulink block diagram to be triggered by PC32 interrupts. Also timer frequencies and hardware interrupt triggering modes can be set up. The corresponding block mask is shown in Fig. 7.4.



Block Parameteis: PC32 Inl Support

PC32 Intenupt Block (mask)

Asynchronous intenupt suppoit for the Inovative Integration PC32 card.
Note: TCtKO/1 settigs depend also on jumpws (JP3 • 6)
WARNING : Changing TCIX0/1 settings to TimerO/1 configures them as outputs. Check external connections!

Parameters

EI0-EI3Ttiggeing: Edge

TCLKO Source : |Exlernal

TCLK1 Source : (External

Timer 0 Freq

0

Timei 1 Ffeq

0

nterfupt Numbers (top to bottom)

(1 2 3 4 9 10]

Fig. 7.4 : Block mask for the PC32 Int Support block

The source code generated for triggered subsystems is placed in separate functions and not included into the model's main update functions. This fact is used by the PC32 Int Support block to attach these functions as individual ISRs. During the TLC parse all outputs of the PC32 Int Support block are probed for attached subsystems as shown in the flow diagram in Fig. 7.5. It is verified that only one connection exists per output port. If the connection leads directly to **a** subsystem its update function is mapped as an ISR and code is generated to support that interrupt at runtime.

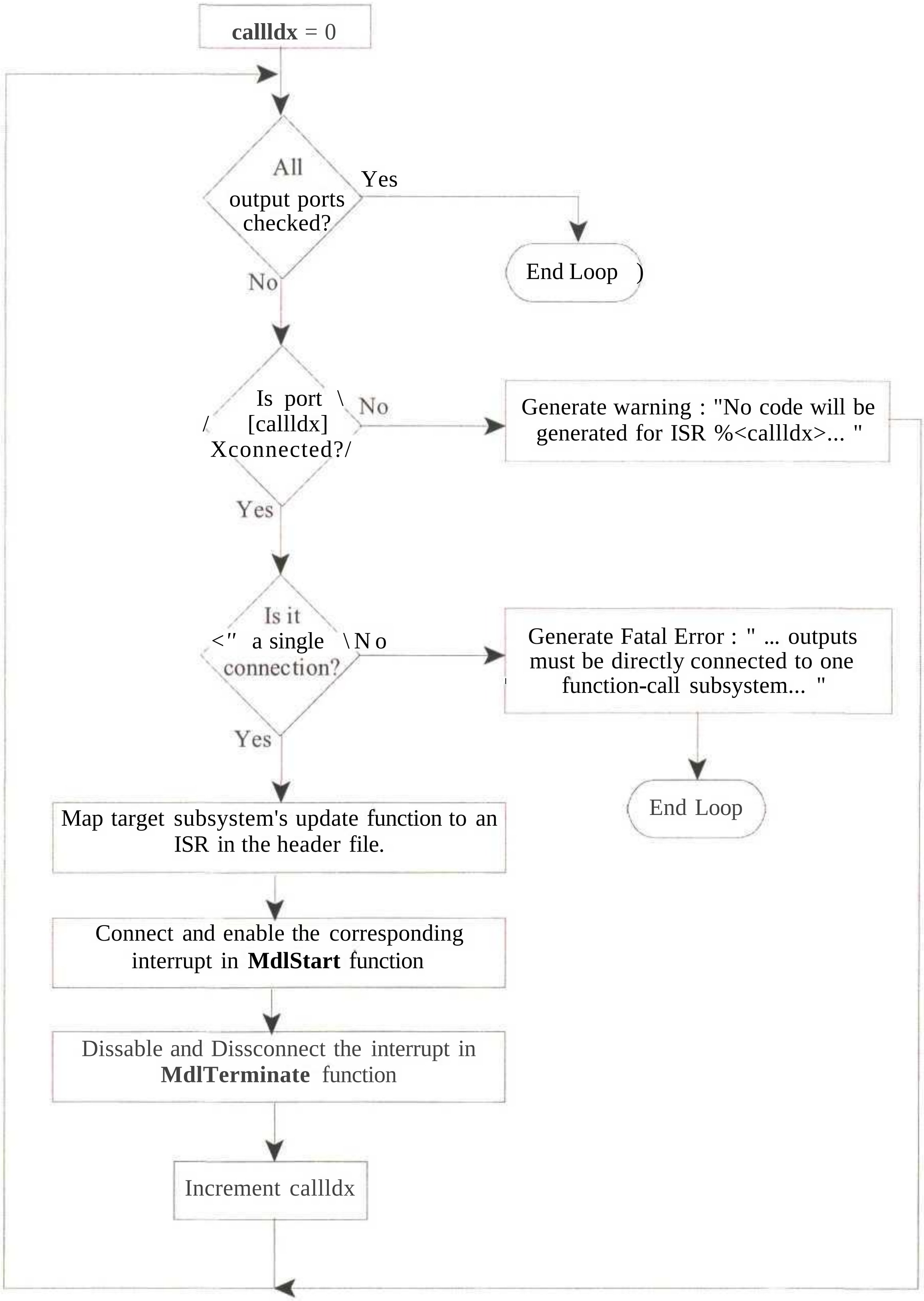

callIdx = 0
All output ports checked?
Yes
End Loop
No
Is port [callIdx] connected?
No
Generate warning : "No code will be generated for ISR %<callIdx>... "
Yes
Is it a single connection?
No
Generate Fatal Error : " ... outputs must be directly connected to one function-call subsystem... "
End Loop
Yes
Map target subsystem's update function to an ISR in the header file.
Connect and enable the corresponding interrupt in MdlStart function
Dissable and Dissconnect the interrupt in MdlTerminate function
Increment callIdx

Fig. 7.5 : TLC parse loop for processing subsystems connected to a PC32 Int Support block

The following TLC code excerpt demonstrates the loop structure and the error trapping algorithm. The TLC loop variable **calllдx** is used to step through all outputs of the interrupt support block :

```
%foreach calllдx = NumSFcnSysOutputCalls

%% Get downstream block if there is one
%if "%<SFcnSystemOutputCall[calllдx] .BlockToCall>" != "unconnected"
 %assign ssSysIdx = SFcnSystemOutputCall[calllдx] .BlockToCall [0]
 %assign ssBlkldx = SFcnSystemOutputCall[calllдx] .BlockToCall [1]
 %assign ssBlock = CompiledModel.System[ssSysIdx].Block[ssBlkldx]

%% Check to see if this is a direct connection
 %if (ssBlock.ControlInput.Width != 1)
  %assign errTxt = "The II 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


     .
     .
     .
          process the valid subsystem
     .
     .
     .


 %else  %% The element is not connected to anything

  %assign wrnTxt = "No code will be generated for ISR %<calllдx> "\
                               "since it is not connected to anything."
  %warning %<wrnTxt>
 %endif

%endforeach
```

All valid subsystems' update functions need to be mapped as ISRs.The following TLC code ensures the correct definitions are placed in the header file :

```
          .
          .
          .
     %assign isrSystem = System[ssBlock.ParamSettings.Systemldx]
          .
          .
          .
     ttdefine   %<isrSystem.OutputUpdateFcn>(%<tSimStruct>,
                                             %<tControlPortIdx>,
                                             %<tTID>)
                                             c_intO%<callIdx+l>()
          .
          .
          .
```

This define ensures that the resultant ISR naming adheres to the TI convention as discussed in Chapter 6, section 6.2.3, and also ignores any parameters passed to the function. The three default function parameters can be safely disregarded for any triggered subsystem since :

(i) Simulink only allows discrete blocks in triggered subsystems, thus the *SimStruc* parameter is not needed since no timing information is necessary.

(ii) No direct inputs or outputs to a triggered subsystem are allowed, thus *ControlPortldx* has no meaning.

(iii) Since the blocks within a triggered subsystem execute asynchronously to the rest of the model the sample hit time, *TID*, is not needed either.

In the **MdlStart** function the ISRs are attached to their corresponding interrupts as specified by the user in the block mask and the interrupts are enabled :

```
                .
                .
                .
  install_int_vector(c_intO%<callIdx+l>,(int)%<LibBlockParameter(P6,"",
                                                              "",callldx)>);

  enable_interrupt^ (int)%<LibBlockParameter(P6,"","",callldx)>-l);
                .
                .
                .
```

Correspondingly, in the **MdlTerminate** function the interrupts are disabled and the ISR vectors freed up :

```
                .
                .
                .
     disable_interrupt((int)%<LibBlockParameter(P6,"","",callldx)>-l);
     deinstall_int_vector((int)%<LibBlockParameter(P6,"","",callldx)>);
                .
                .
                .
```

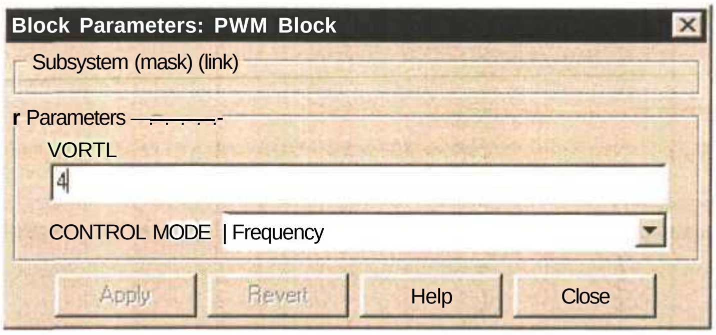
Apart from attaching subsystem blocks as interrupts the PC32 Int Support block generates code in the **MdlStart** function to set hardware interrupts to either level or edge triggering. The source for TCLKO and TCLK1 signals, described in Chapter 4, can be specified too. Code is also generated to setup onboard timer frequencies. To effect the above settings the user input is read from the block's mask.

The TLC file for the interrupt support block generates source code only into **MdlStart** and **MdlTerminate** functions, thus it only affects the model's initialisation and termination and has no effect on the model update functions since the ISRs execute independently once initiated.

7.5.5 PWM Card Driver Block

The PWM block is designed to facilitate access to the custom PWM board, described in Chapter 4. The mask for this block is shown in Fig. 7.6. The Harming PWM IC is a memory mapped peripheral, and following addresses are defined to allow reads and writes to its registers :

```
#define Status_word (volatile int*) 0x81a001
ttdefine Data _word (volatile int*) 0x81a000
```



Block Parameters: PWM Block

Subsystem (mask) (link)

Parameters

VORTL

4

CONTROL MODE | Frequency

Fig. 7.6 : Block mask for the PWM Block

All writes to the Hanning IC have to be performed outside its processing cycle. This can be done using two methods :

(i) Polling - the WRFLAG flag in the status register of the PWM IC is polled and writes are only performed while it is low.

(ii) Interrupt - the PWM IC issues an interrupt signal at the end of each processing cycle. The write operations will be successful after this signal providing that they are complete before the start of the next processing cycle.

During initialisation default values are written to all registers and the polling method is used. The **pollpwm** function is called before each write access :

```
void pollpwm( void )
{
          while (*(Status_word) & Oxl);
}
```

If a processing cycle is active this function will effectively suspend execution on the target until the Harming IC is ready for data. Another side effect of the polling method is that there is no guarantee that subsequent writes will fall during the same idle cycle of the Harming IC. During startup there are no pressing time deadlines and the target code can afford the resultant delays. Since the PWM outputs are disabled it is not critical if the writes are split over a number of cycles. During normal operation, however, it is important that registers required for output calculations are all written during one cycle. Otherwise, the PWM IC will calculate its outputs based on an incorrect set of data and the results will be unpredictable. The user has to ensure that the PWM driver block is placed in a subsystem triggered by the interrupt signal from the PWM IC. The example in Chapter 9 illustrates a safe use of the PWM block.

The following lines perform updates of the PWM IC registers during execution of the model:

```
*(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, "", 1I"/ 2)>;
```

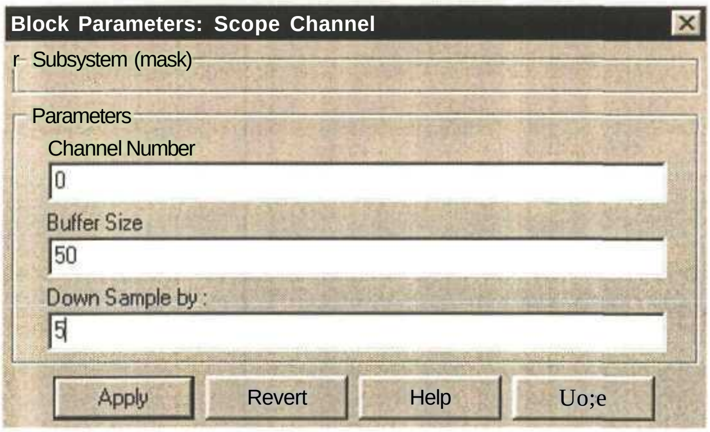
Block inputs 1 and 2 are written to the UA and UB registers respectively. The variable **CtrlMode** is passed dynamically as parameter from the PWM block's mask and specifies whether the third input signal value is written into the frequency or phase angle register. The **pollpwm** function is called between the writes simply as a safety precaution. Assuming the block is executed on an interrupt signal from the PWM IC, the **pollpwm** function calls should always return true immediately. At model termination the output of the PWM IC is disabled by writting 0 to the status register :

```
*(Status_word) = 0;
•(Status_word) = 0;
```

7.5.6 Scope Channel Block

The Scope Channel block is used to queue its input data in a circular buffer. The corresponding block mask is shown in Fig. 7.7. The data buffered by the Scope Channel block is then uploaded to the host in packets by a background process of the RTK, as described in Chapter 6. This mechanism ensures that minimum time is wasted during model updates on data capture. There is no theoretical limit to the number of Scope Channel blocks in any Simulink diagram. However, there is a physical limitation since memory for each queue is dynamically allocated for the individual queues on the program stack on model startup. To establish the number of Scope Channel blocks in the model the following code is included in each TLC file :

```
%if EXISTS("UpldSeen")
  %assign ::UpldSeen = ::UpldSeen + 1
%else
  %assign ::UpldSeen = 1
    %openfile buffer
    extern QUEUE queue[];
    extern unsigned int buffer_size[];
    extern int queue_error;
    extern int NumQueues;
    extern LogData;
    extern channel_map[];
    %closefile buffer
    %<LibCacheDefine(buffer)>
%endif
```



Block Parameters: Scope Channel

r Subsystem (mask)

Parameters

Channel Number: 0

Buffer Size: 50

Down Sample by : 5

Fig. 7.7 : Block mask for the Scope Channel Block

The above TLC code ensures that global variable definitions are only included once to avoid multiple declarations. At the end of the TLC parse process the variable UpldSeen holds the number of Scope Channel blocks present. The value is then used to dynamically allocate resources for the correct number of upload queues :

```
ttifdef UPLDYES
  buffer_size[%<UpldSeen>-l] = (unsigned int)
                                  %<LibBlockParameter(PI,"",•"',0)>;
  NumQueues = %<UpldSeen>;

  if  (!queue_init(&queue[%<UpldSeen>-l],(2*buffersize[%<UpldSeen>-l])))
  {
          .
          .
          .

ttendif
```

On termination of the model the memory is freed up again :

```
free(queue[%<UpldSeen>-l].base);
```

The Scope Channel blocks and the data queues are numbered according to the value of UpldSeen, however the block mask allows each Scope Channel to have a user defined logical number as well. Data packets sent to the host are numbered using this logical numbering scheme, while within the target code the physical numbering is used. Thus, all Scope Channel blocks will have a unique physical number, however, no checking is performed to ensure that the logical numbers are unique as well. Should a user assign the same logical number to multiple Scope Channel blocks, their data packets will be mixed and treated as a single stream by the visualisation utility, Display.

The down sample option allows adjustment of the sampling frequency for the Scope Channel block. The Display utility limits the number of samples which can be displayed at a single time to 1000. This limitation can make it difficult to view waveforms with frequencies much lower than the execution frequency of the Scope Channel block. In a typical motion control project the controller execution frequency might be in the region of 5 kHz, while the frequency of the speed or position reference is not likely to be above 1 Hz. This means that only up to 200 ms (1 / 5 kHz) of the captured waveform would be visible at any time. The down sampling option allows to divide the sampling frequency internally in the Scope Channel block and effectively to extend the time visible in the Display utility. For example, with a Scope Channel block executing at 5 kHz and with down sampling set to 5, a full period of a 1 Hz waveform could be viewed.

The following code excerpt illustrates how data samples are queued up :

```
if (LogData)
{
 DownSample[%<UpldSeen>-l]--;
 if (DownSample[%<UpldSeen>-l]==0)  /*capture this sample*/
 {
  channelmap[%<UpldSeen>-l]=%<LibBlockParameter(P2, "","",0)>;
  •((volatile float*)enqueue_ptr(&queue[%<UpldSeen>-l])) =
                                 %<LibBlockInputSignal(0, •"', "", 0)>;
  Down_Sample[%<UpldSeen>-l] = (unsigned int)
                                 %<LibBlockParameter(P3,"", "",0)>;
 }
}
```

Each time a sample is captured into a queue, the value of **Down_Sample** is reloaded with the current value from the corresponding Scope Channel block's dialog box. **Down_Sample** is then decremented each time the Scope Channel block executes and a sample is captured when it reaches zero.

7.6 Conclusion

The above sections described the details of operation of the various driver blocks included in the CSDE. Each block has a corresponding graphical block for use in **the** Simulink block diagram and a TLC file which is used during the build process to include the block's code into the final executable. Blocks requiring parameter input for the user also have a mask. Chapter 8 will describe the implementation details of the standalone utilities created as part of the CSDE.

CHAPTER EIGHT
HOST SUPPORT COMPONENTS AND UTILITIES

8.1 Introduction

This Chapter describes three components of the CSDE which reside in the Windows 95 environment on the host PC, namely :

(i) External Mode Communication DLL - this module provides an interface between Simulink in external mode and the RTK on the PC32 target.

(ii) Code Download Utility - this is a command line based utility designed to download executables to the PC32 card.

(iii) Display Utility - this standalone Windows executable receives data packets from the target in real-time and displays the data graphically.

All three components were created using MS Visual C++ and will be introduced individually in the following sections.

8.2 External Mode Communication

The external mode link between Simulink on the host PC and the RTK on the PC32 target is implemented via the DPRAM. Section 6.3 in Chapter 6 focussed on the subsection of the RTK responsible for receiving and executing commands issued from Simulink. In order to implement Simulink's external mode on the host PC a custom DLL file *(ext_PC32.dll)* was created and is introduced in detail in the following section. The DPRAM memory mapping and protocols used for communication are listed in Appendix B.

8.2.1 Host Side External Communications

The DLL file to implement the Simulink External Mode needs to conform to an API as shown in Table 8.1. Appendix I lists the source code which is compiled to obtain the *ext_PC32.dll* file. Apart from the three default functions specified in the Simulink External Mode API, a common function **handshake** is implemented. Fig. 8.1 shows the flow diagram for sending a single

Simulink Request	Function **Call**
External Mode comms initialisation	**mdlCommInitiate**
Send changed parameter to target	**mdlSetParameters**
Terminate comms	**mdlCommTerminate**

Table 8.1 : Simulink External Mode API

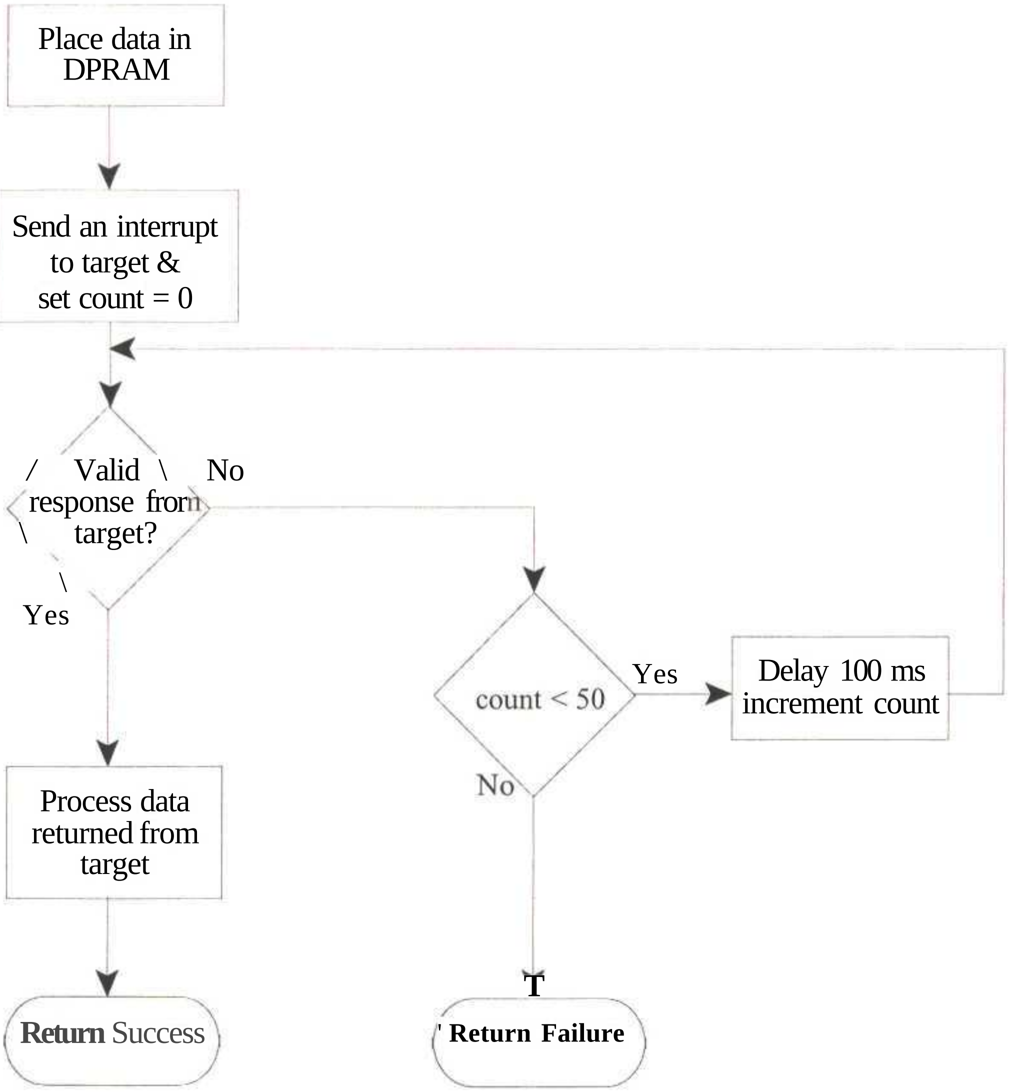

Place data in DPRAM
Send an interrupt to target & set count = 0
Valid response from target?
No
Yes
count < 50
Yes
No
Delay 100 ms increment count
Process data returned from target
Return Success
Return Failure

Fig. 8.1 : Flow Diagram of Command Communication on the Host

command to the target. Each of the API functions first sets up data in DPRAM and then calls **handshake** to initiate the data transfer. If **handshake** returns successfully, the DPRAM contains the target's response and the corresponding API function can process it and send the result back to Simulink. If no response is received from the target within 5 seconds (50 x 100 ms) a comms failure is signalled back to Simulink. The relatively long time before a comms timeout is necessary since the RTK on the target implements its communication routines within a low priority background task and response time cannot be guaranteed.

The API function **mdlCommlnitiate** performs two steps :

(i) Initialisation of the II VXD driver and a virtual pointer to DPRAM on the PC

(ii) Checksum verification between the host and target. This is done to ensure the executable about to be started on the target corresponds to the block diagram in Simulink.

The following calls to the II driver ensure the VXD is initialised and that the local pointer, **dpram,** points to the virtual address where DPRAM is mapped on the host PC :

```
if  (target_open(0))
{
      printf("Target Open. \n");
      dsp = target_cardinfo(0);
      dpram = (volatile unsigned long*)dsp->DualPort.PhysAddr;
      .
      .
}
```

If the above initialisation is successful local block diagram's checksums are passed to the target executable for comparison:

```
dpram[1] = model_checksum[0];
dpram[2] = model_checksum[1];
dpram[3] = model_checksum[2];
dpram[4] = model_checksum[3];
mexPrintf("Verifying checksums .... ");
if (handshake(4) == 4)
{
  mexPrintf("OK!\n");
  xxx = 0;
  starting = 1;
}
else
{
  mexPrintf("FAILED!\n");
  xxx = -1;
  mexPrintf("Checksum error. Recompile/Reload Diagram!\nM);
}
```

If the checksum comparison fails, -1 is returned to Simulink and an error is printed in the MATLAB command window. During parameter updates no range checking is performed on the pointers into *SimStruc* passed to the target. Invalid pointers could cause memory to be corrupted on the target and result in unpredictable results or even a complete software crash. Thus it is crucial at startup to ensure that the compiled executable on the target corresponds exactly to the block diagram used as the external mode interface.

After a successful call to **mdlCommlnitiate** Simulink passes a complete set of default parameters to **mdlSetParameters** to be downloaded to the target. The function **mdlSetParameters** sends the parameters serially in a for loop :

```
for (i=0; i<num_changed; i++)
{
 dpramtl] = elems[i];
 blparam.f = blockparams[elems[i]];
 dpram[2] = bl_param.u;

 if (handshake(2) == 2)
 {
   mexPrintf("Changed parameter P[%d] = %g\n",elems[i], blparam.f);
 }
 else
 {
   mexPrintf("Handshake Failed\n");
   xxx=l;
 }
}
```

The floating point parameter values passed by Simulink in the array **block_params** are mapped to 32-bit unsigned integers before being placed in DPRAM using a union definition :

```
union {
      float f;
      unsigned long u;
      } bl_param;
```

This conversion is necessary due to the different floating point representations used on the host PC and the TI 'C32 target processor. Execution of the model is only started after a successful download of default parameters. The connection between host and target is terminated in the **mdlCommTerminate** function via the call to **target_close** in the II library. The flow diagram in Fig. 8.2 demonstrates all the processes during Simulink's External Mode. Using codes defined in Table 6.1 the command sequence in a typical external mode run could be : 4, 2,... 2, 1,2,. ..2,3.

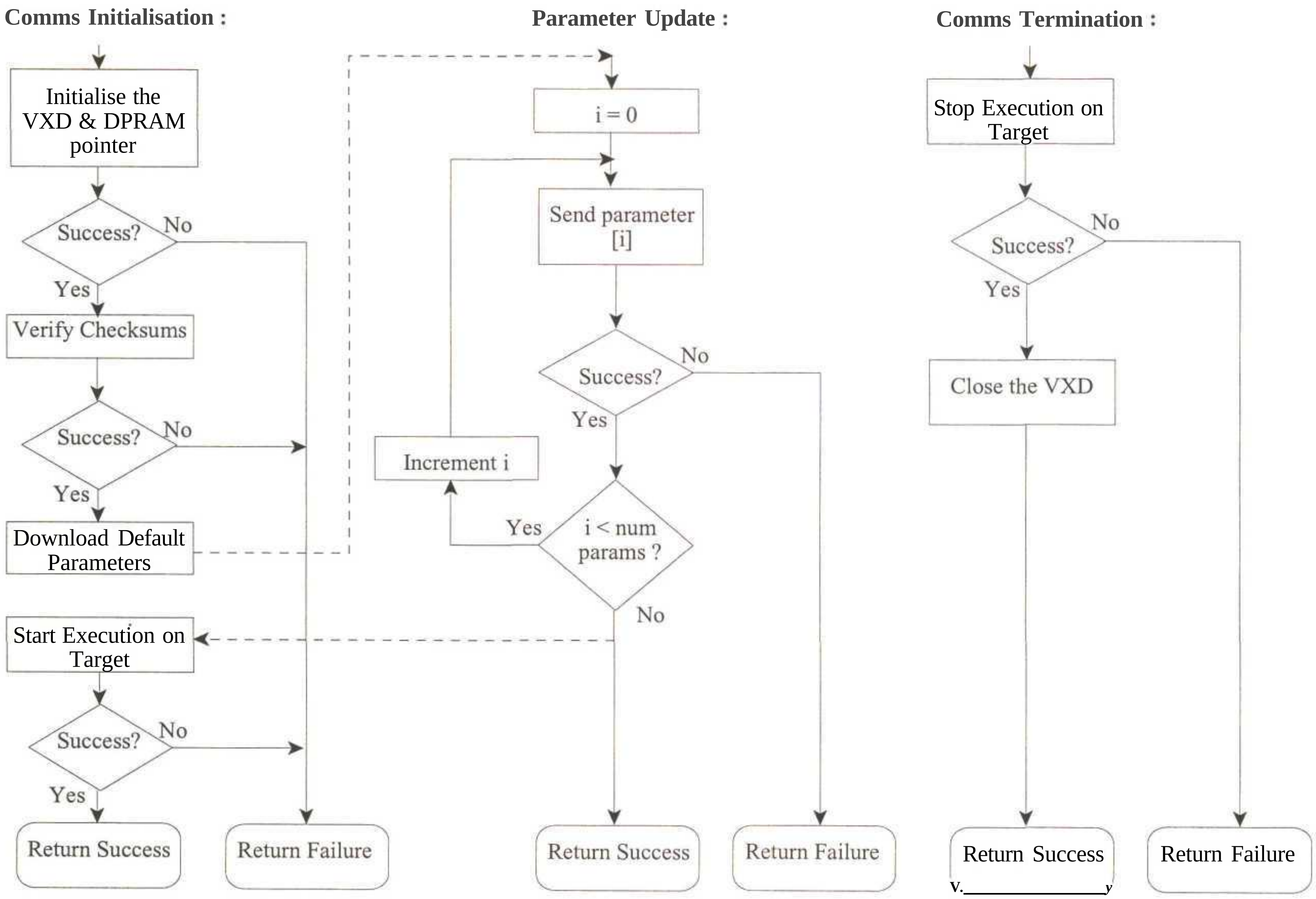


Figure 8.3 : Flow diagram of the link external mode communication

The communication process as described above, as well as in section 6.3 of Chapter 6, allows a user to use the Simulink GUI to interact with the controller executing on the PC32 target in real-time. Execution of the target code can be started or suspended and parameters can be modified online. The following two sections will describe two standalone Windows based utilities which also form a part of the CSDE.

8.3 Code Download Utility

The download utility, *D_Load.exe* is a tool which allows downloading and starting of executable code from the host PC to the PC32 card. It is a command line based program with the following calling convention :

D_Load *parameter 1*

Where *parameter!* is the name of the file to be downloaded, including the full path. Internally, D_Load uses calls to a library supplied by II to communicate with the PC32 card.

As part of the development package, II ship a download tool - *download.exe.* Although this Windows based utility offers a number of extra options, it requires an external dongle to operate. The use of CSDE on PCs not fitted with one of II's dongles does not violate any of the company's license agreements. Thus, the author decided to develop a scaled down download utility to accommodate CSDE users who opt not to install the complete development package from II, but want to make full use of the CSDE.

8.4 Display Utility

The visualisation utility, *Display.exe,* is a Windows based program that uploads raw data from a CSDE compiled executable running on the PC32 card and plots it in a number of windows. Up to ten channels are supported simultaneously. Section 6.4 in Chapter 6 describes how the RTK handles the process of formatting and sending the data on the PC32. The packetised data is transferred via the DPRAM, buffered in the Display application's local memory, processed according to user-specified settings and plotted to scale. The logical interaction of various components of Display is shown graphically in Fig. 8.3. The listing of all relevant source code for the Display utility can be found in Appendix J.

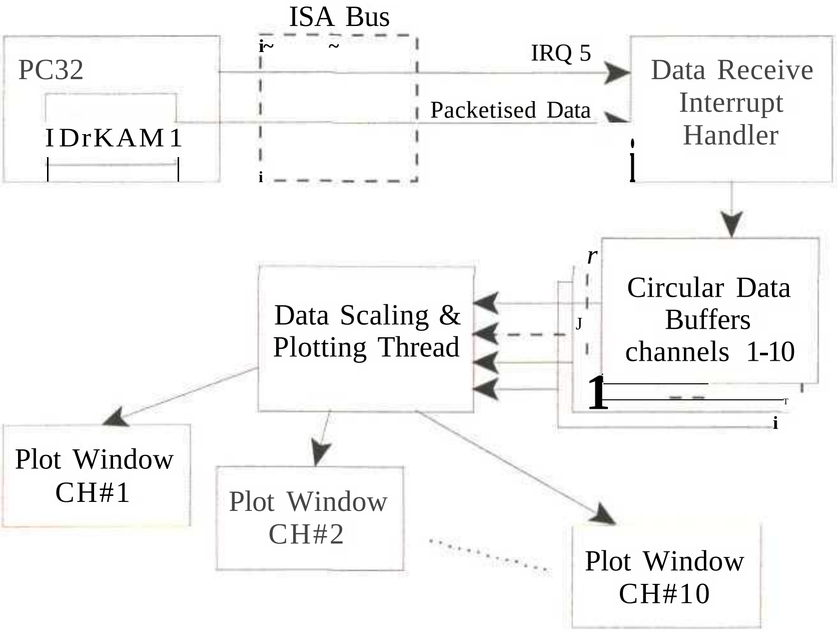

ISA Bus
PC32
IDrKAM1
IRQ 5
Packetised Data
Data Receive Interrupt Handler
Circular Data Buffers channels 1-10
Data Scaling & Plotting Thread
Plot Window CH#1
Plot Window CH#2
Plot Window CH#10

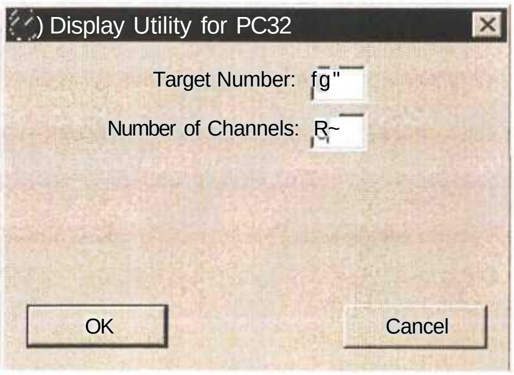
Fig. 8.3 : Logical Diagram of the Display Utility

8.4.1 Display Initialisation

When started, Display comes up with the dialog box shown in Fig. 8.4. Unless there are multiple PC32 cards installed, the target number defaults to 0. The number of channels has to be entered manually and should correspond to the number of channel blocks on the corresponding Simulink diagram.

After the user clicks on 'OK', Display performs its initialisation. Firstly, should there be any channel windows open they are closed to prevent multiple instances of the same window.

```
for (UINT i=0; i<MAX_NUM_CHANNELS; i++)
  if  (theApp.active_child[i])
  {
    theApp.child[i]->DestroyWindow();
    theApp.active_child[i]=FALSE;
  }
```



Display Utility for PC32

Target Number: fg"

Number of Channels: R~

Fig. 8.4 : Display utility dialog box

Next Display creates a separate child window for each channel, and initiates its title to reflect the corresponding channel number. Also the circular buffer for each active channel is cleared and initialised.

```
for (i=0; i<m_num_channels; I++)
{
  theApp.child[i] = new CDisplayChildFrm;
  sprintf(title,"CH #%d",i);
  theApp.child[i]->ShowWindow(SW_SHOW);
  theApp.child[i]->UpdateWindow();
  theApp.child[i]->SetWindowText(title);
  theApp.active_child[i]=TRUE;
  theApp.child[i]->my_num=i;
  theApp.queue[i].head = 0;
  theApp.queue[i].trigg = 0;
  theApp.queue[i].tail = 0;
  for (UINT j=0; j<5000; j++) theApp.queue[i].sample[j] = 0;
}
```

The following initialisation step checks whether a PC32 card is present and opens the Innovative Integration VXD driver. If there is no response from the selected PC32 target, Display terminates with an error message.

```
if(!target_open(mtarget))
{
  sprintf(title, "Unable to open Device Driver for target %d\n"
                 "Check target number setting", mtarget);
  MessageBox(title, "FATAL ERROR", MBICONINFORMATION);
  PostQuitMessage(0);
}
```

The DPRAM is physically installed on the PC32, but is also mapped to a logical area of the host PC memory through the ISA bus. This logical address is accessed via calls to the II drivers. To respond to the signals from the target on IRQ 5, Display installs an ISR routine. Further, a thread is launched to monitor the channel buffers and to process, scale and plot the data.

```
//  Set DPRAM addrss
theApp.dsp = (CARDINFO*)targetcardinfo(mtarget);
theApp.dpram = (volatile int*)theApp.dsp->BusMaster.Addr;

//  Set up the Virtual ISR
host_interrupt_install(m_target, EnqueueData, (PVOID)m_target);
host_interrupt_enable(m_target);

// Set up the Thread function
if (theApp.pThread==NULL)
  theApp.pThread = AfxBeginThread(ThreadFunc, NULL);
```

Finally, bit 0 is cleared at the first location in DPRAM reserved for data logging (see Appendix B for details). This value flags the target data upload routine that Display is ready to receive data. The main Display window is then minimised out of the way :

```
// tell target we are ready for data
theApp.dpram[4] &= 0xFFFE;

// Minimise the main window
GetWindowPlacement(&wpl);
wpl.showCmd = SWMINIMIZE;
SetWindowPlacement(&wpl);
```

After the above initialisation code completes successfully, the IRQ 5 interrupt signal from the target is monitored and data uploaded as shown in the following section.

8.4.2 Upload Mechanism

To avoid the need for use of semaphores to arbitrate access to DPRAM, the area from offset 0x0004 up to offset 0x1000, is reserved solely for the uploading of data for visualisation. The memory mapping for DPRAM as well as data transfer protocol are described in detail in Appendix B.

The data logging function on the PC32 places raw packets of data in DPRAM and signals an interrupt (IRQ 5) to the PC when finished, as described in section 6.4 in Chapter 6. Display's interrupt handling routine then decodes the packets and places their contents in the correct channel's buffer. The following excerpt shows the source code for the ISR routine **EnqueueData,** the flow diagram of the process is shown in Fig. 8.5 :

```
VOID EnqueueData (PVOID pvoid)
{ UINT I;
  long offset;
  unsigned long buffer_size;
  int buffer_num;
  union
   {
      float            f;
```

```
        unsigned long     u;
     } plot_data;

   offset = 0;
   buffer_num = theApp.dpram[5 + offset++];
   buffersize = theApp.dpram[5 + offset++];

   while ((buffer_num !=999) && (offset < 1000))
   {
     for (i = 0; i < buffersize; i++)
     {
       plot_data.u = theApp.dpram[5 + offset++];
       enqueue(buffernum, plot_data.f);
     }
     buffer_num = theApp.dpram[5 + offset++];
     buffer_size = theApp.dpram[5 + offset++];
   }
}
   theApp.dpram[4] &= OxFFFE;
       //signal to target that we are finished reading
```

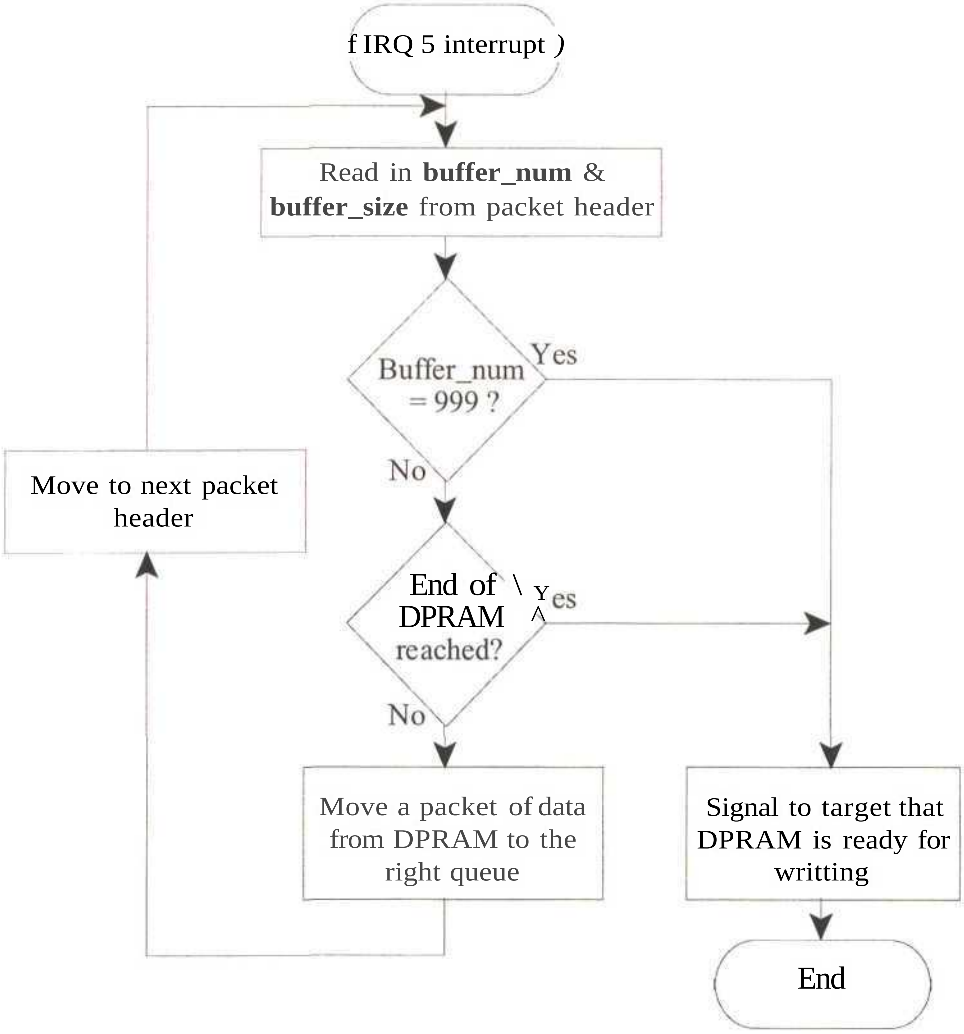

f IRQ 5 interrupt)
Read in buffer_num & buffer_size from packet header
Buffer_num = 999 ?
Yes
No
End of DPRAM reached?
Yes
No
Move to next packet header
Move a packet of data from DPRAM to the right queue
Signal to target that DPRAM is ready for writting
End

Fig. 8.5 : Flow diagram of the data upload and buffering on host

The intermediate variable **plot_data** is defined as a union of types *unsigned long and float*, which means the the physical 32 bit of data stored under **plot_data** can be accessed as either of the types. This mapping of data types is necessary to account for the difference between floating point number representation in the compilers for the target and host platforms. Thus raw data from the DPRAM is read out as an *unsigned long* which means it will be read out 'as is' on a bit by bit basis. The data is meaningless in this format until it is mapped to a floating point type.

Each packet header consists of two values :

(i) **Buffer_num** - the channel number for which this packet's data is destined.

(ii) **Buffer_size** - the number of data points in the packet, not including the header. The value of **buffer_size** also points to the beginning of the next packet in the sequence. The value of 999 (decimal) signals the end of a packet sequence in DPRAM.

The data points read out from each packet are queued in the corresponding channel's queue by a call to the function **enqueue.** After the last packet is processed by the **EnqueueData** function, bit 0 at DPRAM offset 0x0004 is cleared to hand access to DPRAM back to the target for writing.

8.4.3 Data Buffering

Three functions are defined to facilitate access to the channel buffers :

(i) **Enqueue** - this function is used to add new entries to a circular buffer.

(ii) **Dequeue** - is used to read from a buffer.

(iii) **Enqueued** - returns the number of entries currently stored in a buffer.

All three functions are defined as *Mined* in order to optimise the speed at which Display can access the buffers. Defining functions as *inlined* could mean a sacrifice in terms of the size of the resultant executable, but it improves the execution speed by removing the overheads involved in switching between functions. In the case of the Display utility, file size and memory usage on the PC is not a major consideration. Slow execution, however, could create a bottleneck in the transfer of data from the target and result in loss of information.

The channel buffers are implemented in a circular fashion which means each buffer has both **a** head and a tail pointer which wrap around when the maximum size is reached. The circular buffering mechanism was chosen by the author as it allows for the fast data access while maintaining the first-in-first-out (FIFO) principle.

The following code implements the **enqueue** function. An entry is added at current queue head, the head pointer is incremented and wrapped around if necessary.

```
__inline void enqueue(UINT num, float value)
{
  theApp.queue[num].sample[theApp.queue[num].head++]=value;
  if (theApp.queue[num].head == Q_SIZE) theApp.queue[num].head=0;
}
```

To remove an entry from a queue the tail pointer is used. It is then incremented and wraps around if necessary.

```
__inline float dequeue(UINT num)
{
  float value = theApp.queue[num].sample[theApp.queue[num].tail++];
  if (theApp.queue[num].tail == QSIZE) theApp.queue[num].tai1=0;
  return (value);
}
```

To check the number of entries in a given queue the difference between the tail and head pointers is calculated.

```
__inline long enqueued(UINT num)
{
  long depth = theApp.queue[num] .head - theApp.queue [num] .tail;
  return ((depth < 0) ? (depth + QSIZE) : depth);
}
```

The above sections describe the mechanism by which the Display utility receives and buffers data sent by the target in real-time. The following section shows how the buffered data is processed before being plotted.

8.4.4 Scaling and Plotting

The uploaded data is dequeued and plotted individually for each channel in a background thread function **ThreadFunc.** The flow diagram in Fig. 8.6 demonstrates this process.

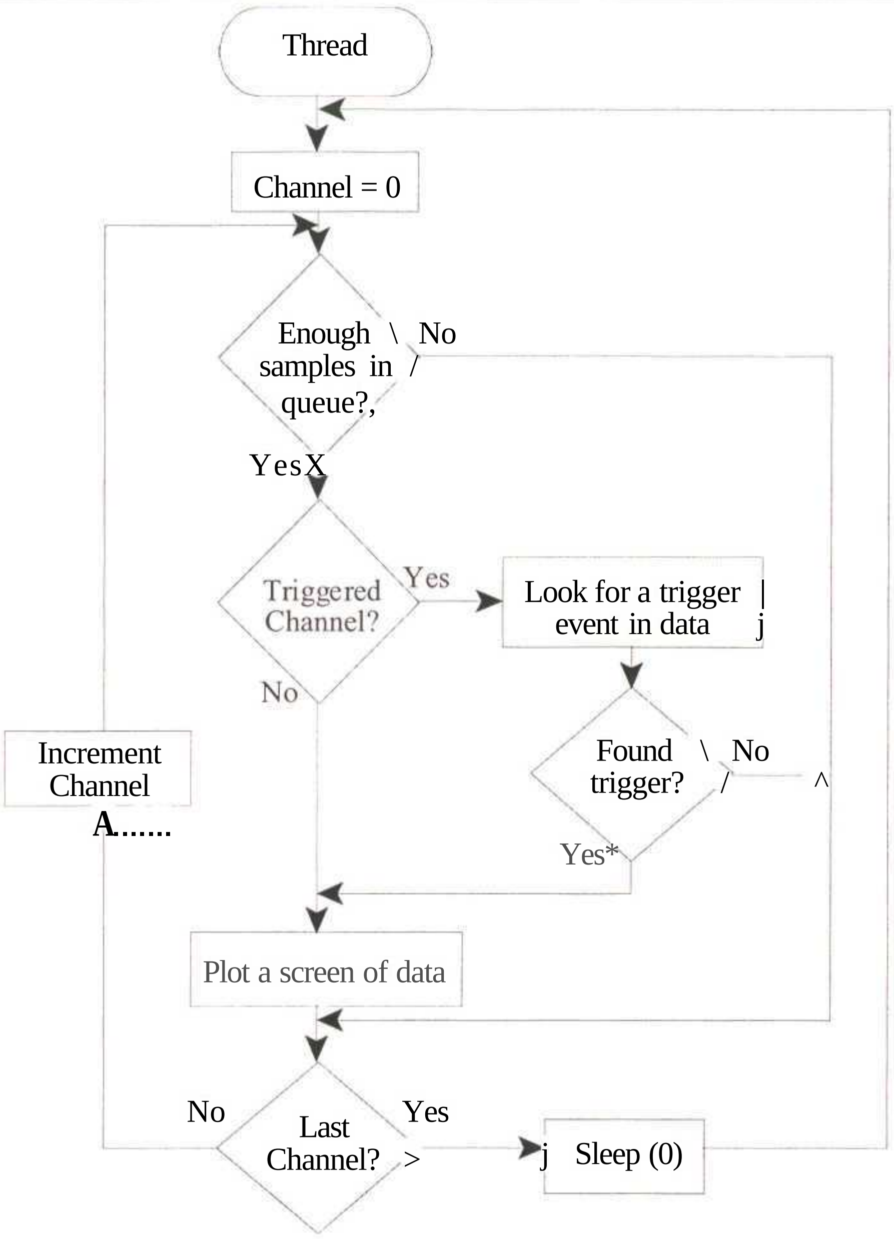


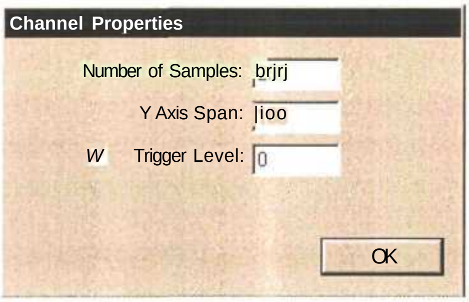
Fig. 8.6 : Flow diagram of the data processing on host

The following source code excerpt demonstrates the structure of the function :

```
UINT ThreadFunc (LPVOID pPtr)
{
  while(1)
  {
        .
      process queue data
        .
    Sleep(0);
  }
}
```

The infinite while loop forms the body of the **ThreadFunc** function. A call to the system function Sleep at every iteration ensures that other Windows processes are not deprived of processing time. Each individual child window's settings can be set by the user via a properties popup window shown in Fig. 8.7. The number of samples to be plotted and the maximum Y axis range can be specified. The number of samples in a child window is used to determine when there is enough data in a queue to fill the window. If the trigger level check box is selected the **ThreadFunc** algorithm will look for a trigger event in the data before plotting it. The leftmost sample plotted in the window will be the first sample found by an trigger finding algorithm shown in Fig. 8.8. Every time a window's plot is updated it is forced to redraw itself via the object call :

```
theApp.childtj]->RedrawWindow();
```



Channel Properties

Number of Samples: brjrj

Y Axis Span: |ioo

W Trigger Level: 0

Fig. 8.7 : Child window properties

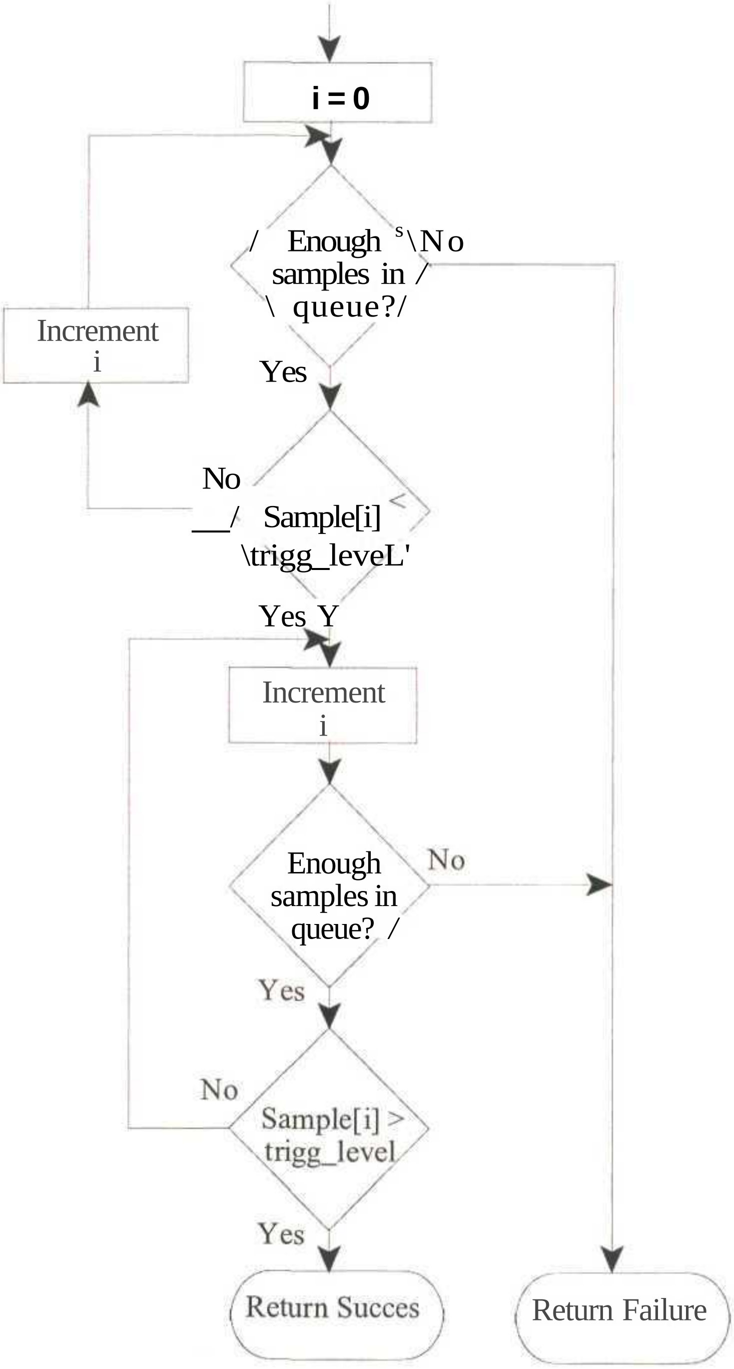


Fig. 8.8 : Flow diagram of the trigger find algorithm

8.5 Conclusion

This Chapter has discussed the software developed as part of the CSDE which is resident on the host PC. These components, as well as the components described in Chapters 5, 6 and 7, form the author's practical contribution to the complete CSDE. Chapter 9 will describe an example of the application of the CSDE to a motion control problem. Two case studies will also be presented of students who applied the CSDE during their undergraduate design projects.

CHAPTER NINE
CSDE APPLICATION AND CASE STUDIES

9.1 Introduction

The preceding Chapters have built up a description of the various components of the CSDE. The aim was to create a complete rapid prototyping tool for the development and real-time implementation of motion control. The particular focus of the project was to aid final year electrical engineering students in tackling challenging design problems in the field of motion control, as introduced in Chapters 1 and 2. By using the CSDE, students are able to gain an in-depth experience in the development of real-time digital controllers without being slowed down by the traditionally drawn out design process.

No such development project can be complete without a field trial to validate whether the objectives have been met. This Chapter shows how the CSDE is used to address issues raised in Chapters 1 and 2. It should be noted that the main aim here is to focus on the transition of controller designs into real-time prototypes. Suitable references are provided to cover controller design and simulation in greater detail.

First example presented is the design of a simple digital controller for a DC machine. The controller itself is straight forward but the example serves well to demonstrate and prove the operation of the CSDE. Next, two case studies are presented of students at the Electrical Engineering Department at UND who used the CSDE to implement complex motion control projects. Mr. Sturgeon implemented a field oriented controller (FOC) on a 3-phase induction machine [STURGE0N1]. His project was judged to be the best final year design in 1998. Mr. Moodley's final year design in 1999 involved the design of a ball-catcher [M00DLEY1] and was judged to be the best control project.

9.2 Motion Controller Design Using the CSDE

The steps in the design of a digital motion controller were outlined in Chapter 2. In a rapid prototyping environment, like the CSDE, they can be summarised as follows :

(i) Modelling - gathering data and creating a model of the controlled machine.

(ii) Simulation - proposing a control algorithm and simulating its performance in non real-time,

(iii) Hardware setup - connecting the machine and required sensors to the controller platform

(iv) Generating prototype - converting the simulation block diagram to a real-time prototype and generating code.

(v) Online tuning - running the controller in real-time, verifying its performance and final adjustment of parameters.

The issues involved in modelling and simulation of motion control are well covered in literature [FORSYTHE1, LEONHARD1]. And in particular the application of MathWorks tools in this field is covered in [FENG1, MATHW0RKS1, 2, 3]. This Chapter will, therefore, not revisit those topics but rather focus on points (iii) to (v) above.

9.2.1 Hardware Setup

The hardware used for the demonstration is a DC machine fed by an H-bridge inverter. A tachometer and a LEM current transducer module [LEM1] are used to provide speed and current feedback respectively. The PC32 controller and a PWM / Tacho expansion card are housed inside the host PC case and form the controller platform. A signal generator is used to provide a speed profile for the controller. Fig. 9.1 shows the schematic diagram of the overall hardware setup used. Fig. 9.2 shows a photograph of the actual setup in the lab.

The speed and current feedback signals are captured by the PC32 on-board ADC channels and displayed in real-time by the Display utility. To verify the correct operation the signals are also made available at the DAC outputs and can be monitored using a digital storage oscilloscope.

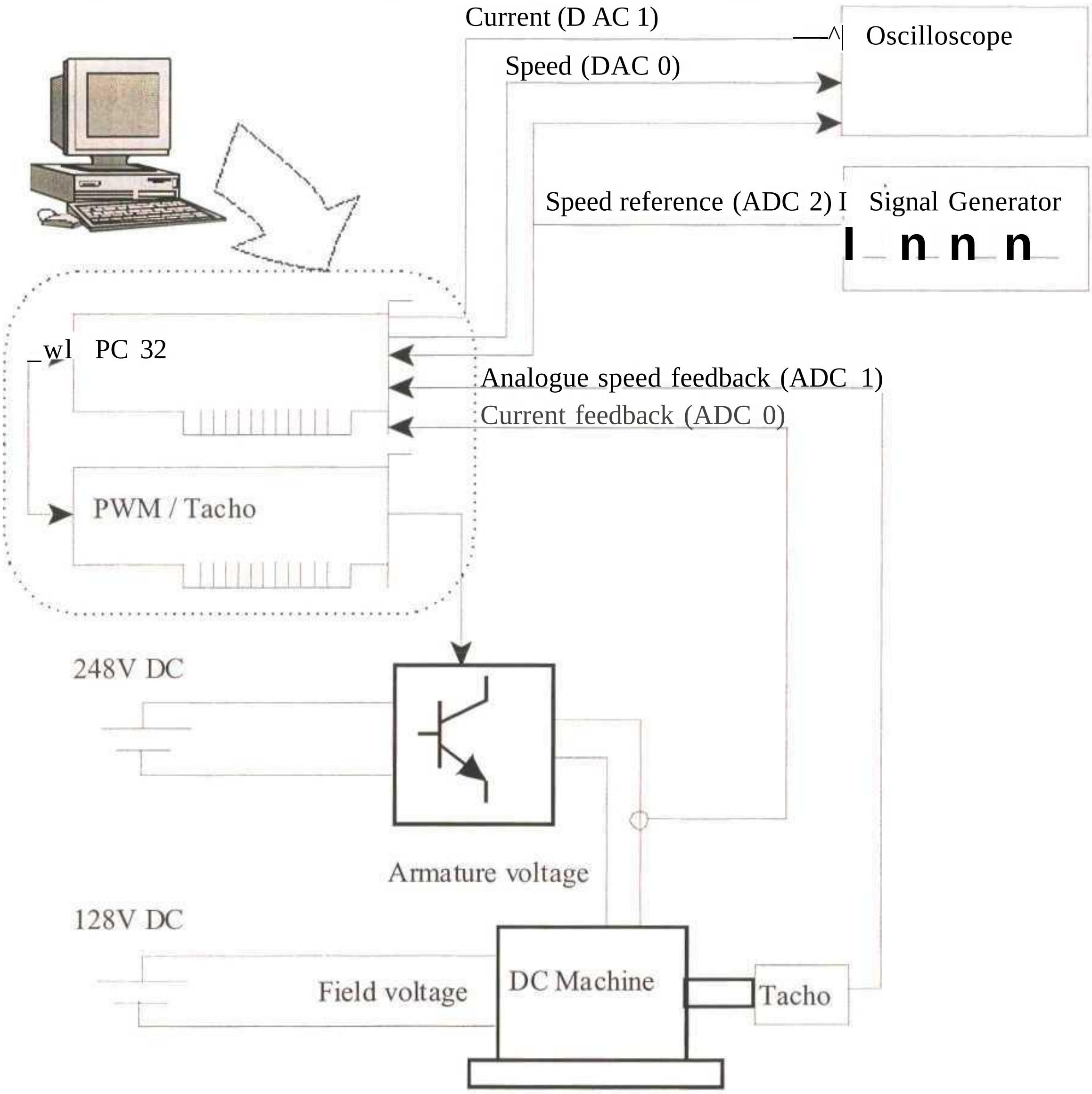


Fig. 9.1 : CSDE Demonstration hardware setup

9.2.2 Generating a Real-Time Prototype

After the controller has been simulated in Simulink, the diagram has to be modified to include hardware driver blocks before a real-time prototype can be generated and executed on **the** hardware platform. Fig. 9.3 shows the simulation diagram while Fig. 9.4 shows the modified real-time version.

The conversion of a simulation diagram into real-time involves replacing **the** DC machine model block with external inputs and outputs to the actual machine. Standard Simulink scope blocks are replaced with custom CSDE scope channel blocks which allow uploading of data in

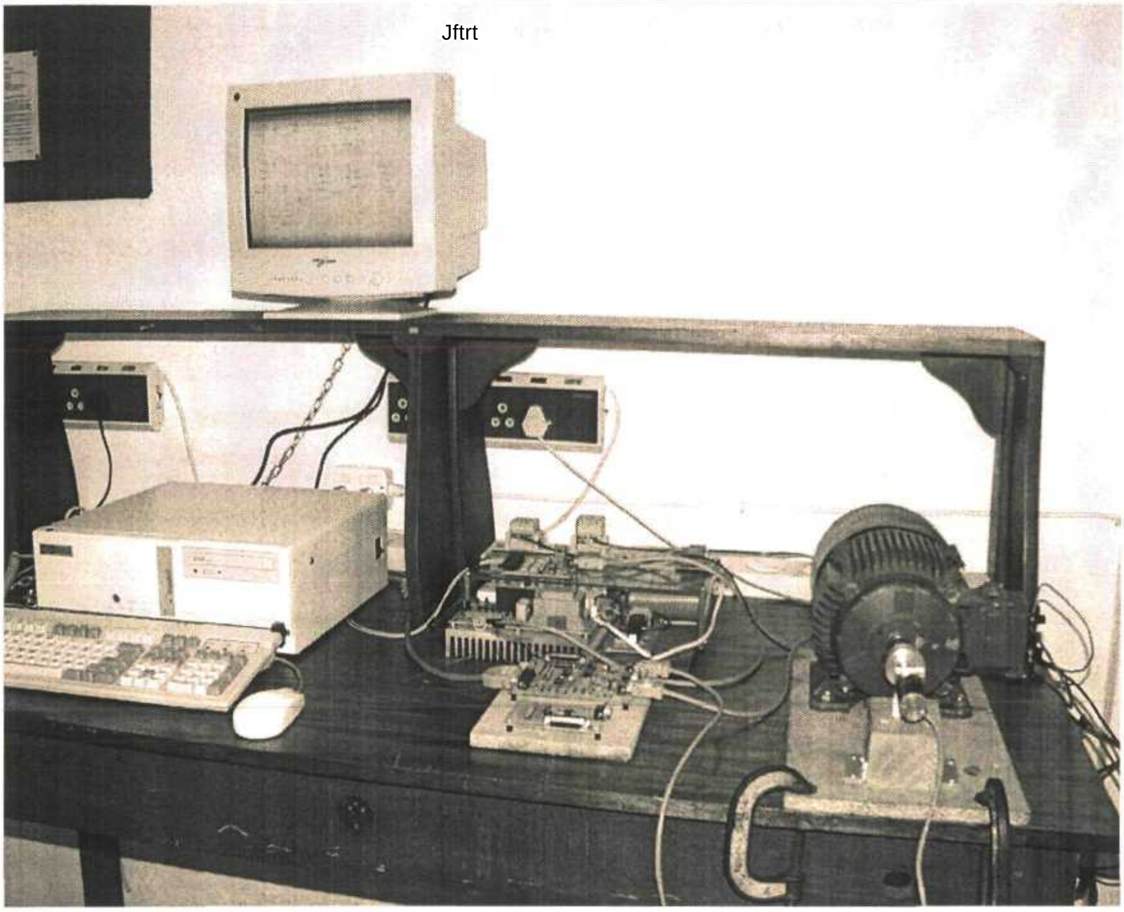


Fig. 9.2 : Photograph of the DC machine demonstration setup

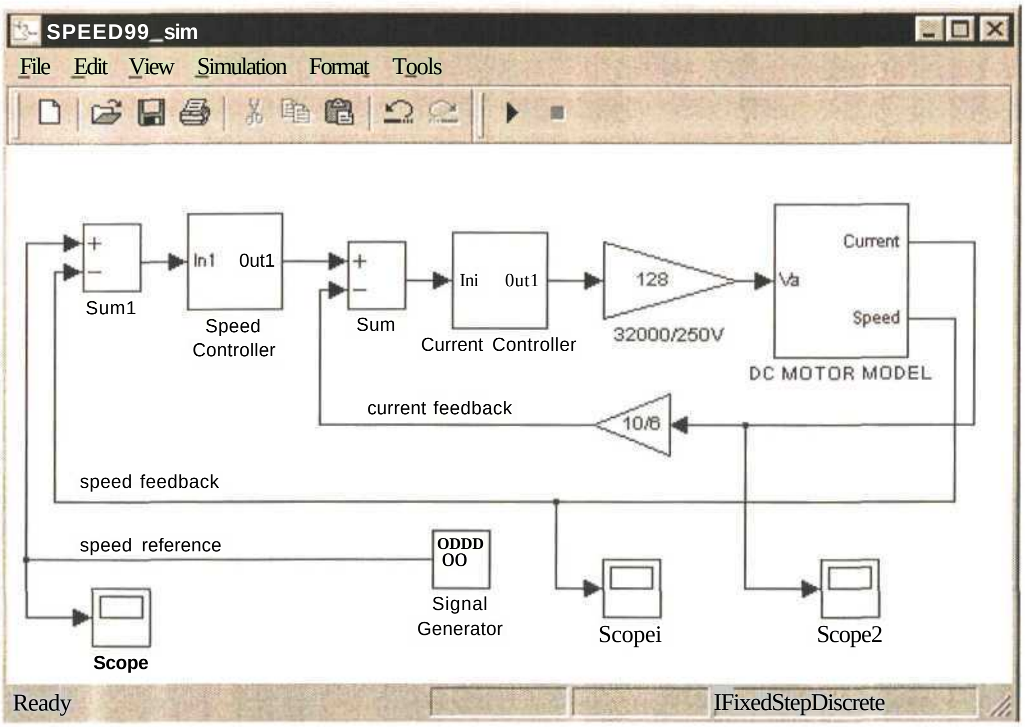


Fig. 9.3 : Simulation diagram for the DC machine controller

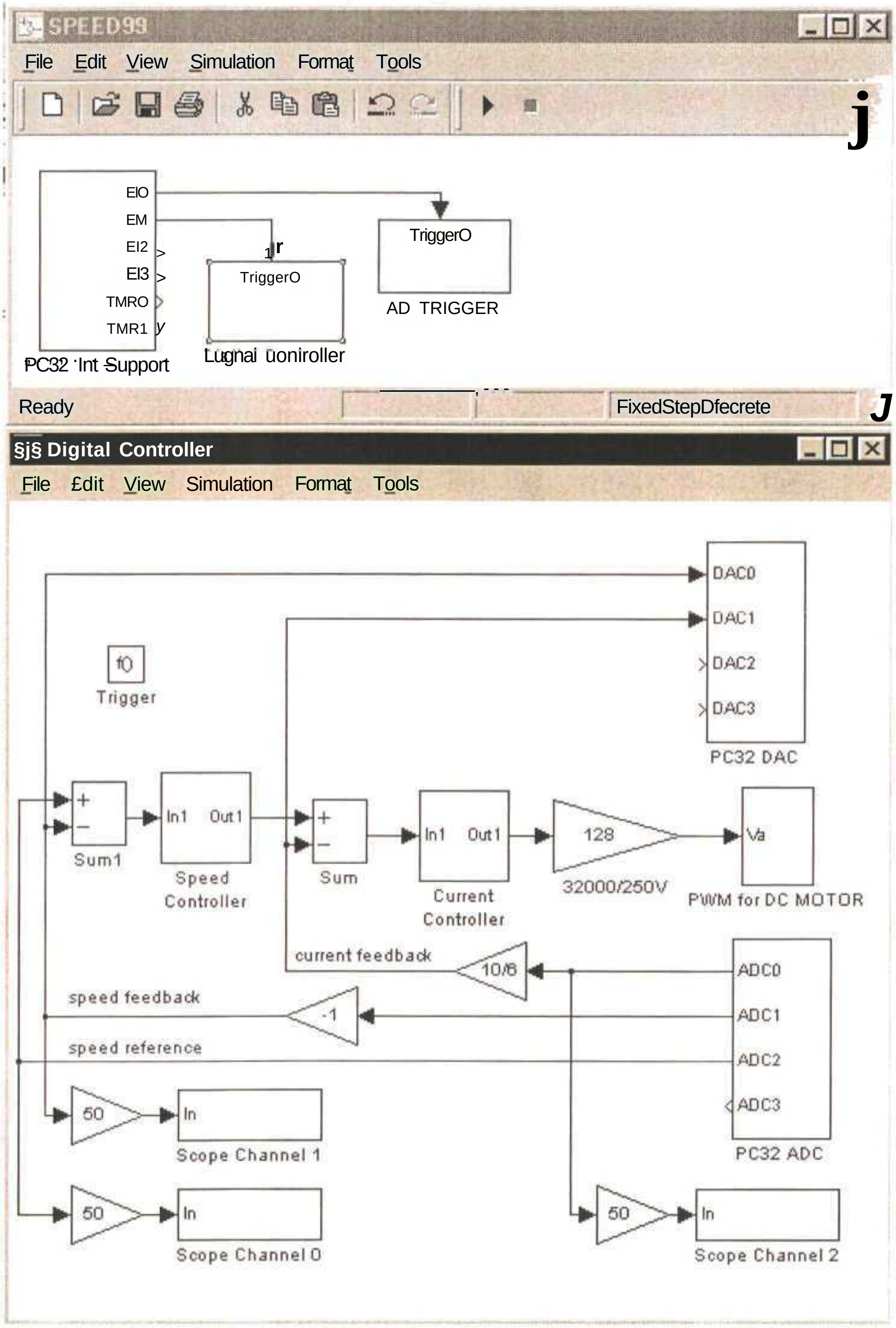


Fig. 9.4 : Real-time controller diagram

real-time to the Display utility. To ensure synchronous sampling, the interrupt signal (El 0) from the Hanning PWM IC is used to trigger the ADCs. In turn, the end of conversion signal (El 1) from the ADC's is used to trigger the control loop. This arrangement ensures that the feedback signals are free of power electronics switching noise as they are always sampled at the centre of the PWM switching period as described in Chapter 4, section 4.5.2.

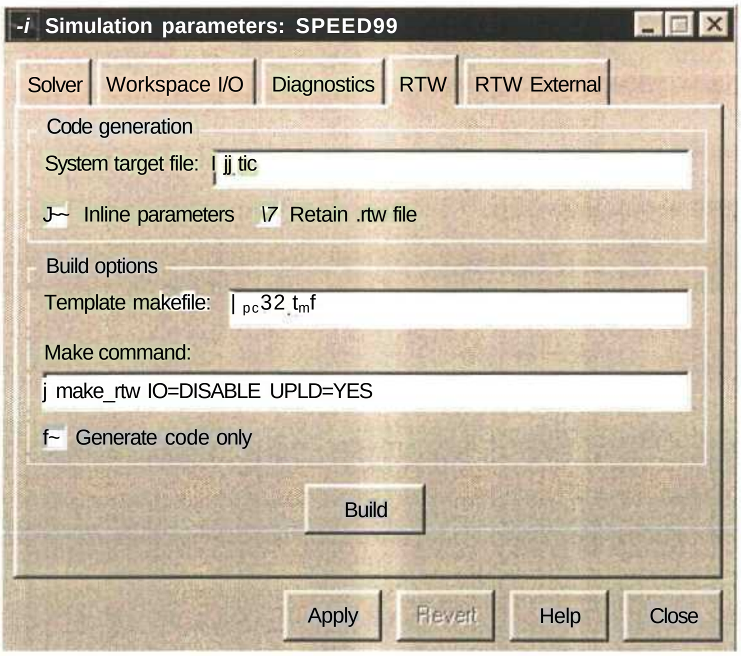
The next step is to set up the RTW parameters. Fig. 9.5 shows the corresponding RTW dialog box. Uploads are enabled since there are a number of scope channel blocks used. Upon pressing the *Build* button RTW generates the controller code and downloads it to the PC32. After the diagram is started the Scope utility can be initiated. Fig. 9.6 shows the plots obtained :

(i) CH#0 displays the speed reference

(ii) CH#1 shows the resultant speed feedback, and

(iii) CH#2 shows the armature current feedback

The signals in Fig. 9.6 are also made available at the DAC outputs and were captured using a digital oscilloscope to give the speed plot in Fig. 9.7 and the current waveform in Fig. 9.8. There is a clear timing skew between the plots in Fig. 9.6 when compared to the results obtained using an external oscilloscope. This is due to the fact that the Display utility does not attempt to synchronise the display channels, but rather triggers each channel individually as described in section 8.4.4.



Code generation

System target file: I jj tic

J~ Inline parameters \7 Retain .rtw file

Build options

Template makefile: | pc32 tmf

Make command:

j make_rtw IO=DISABLE UPLD=YES

f~ Generate code only

Fig. 9.5 : RTW parameters

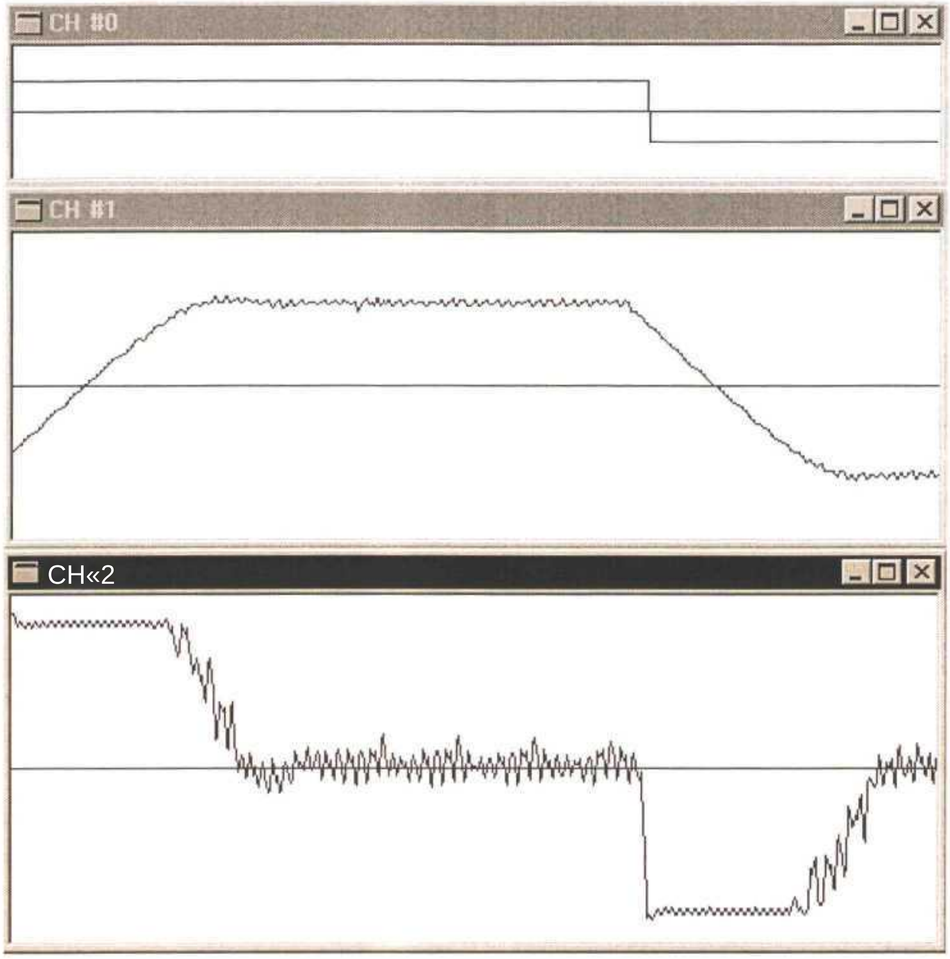


Fig. 9.6 : Speed and current plots with optimal parameters

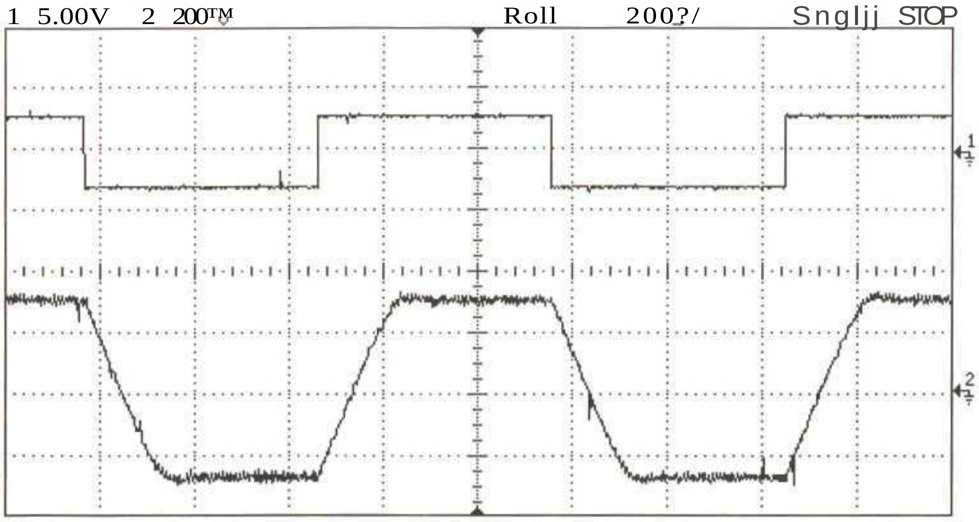


Fig. 9.7 : Speed signal

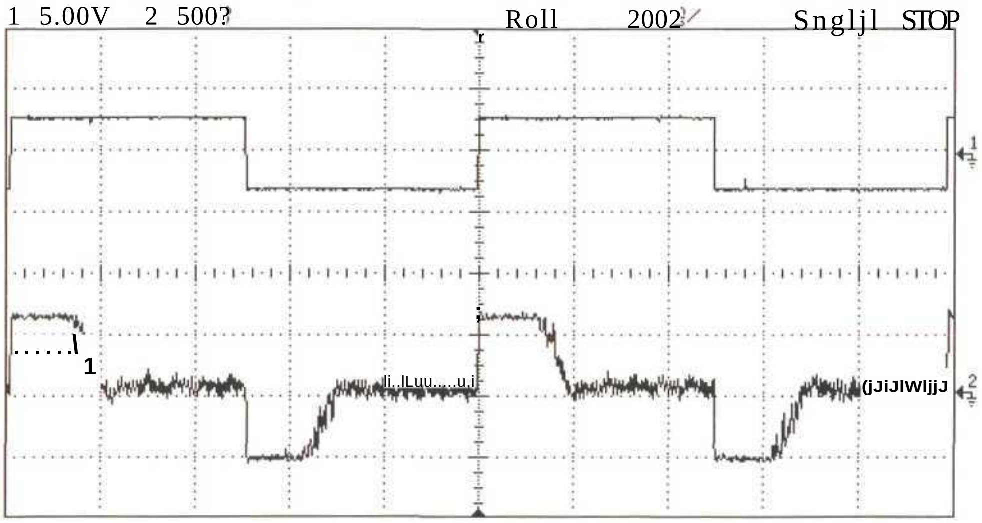


Fig. 9.8 : Armature current waveform

The plots in Fig. 9.6, Fig. 9.7 and Fig. 9.8 correspond to what can be expected from the experiment. The armature current is at its maximum during acceleration and minimum during deceleration. During periods of constant speed the current it almost zero since the machine is running without a load. The above plots are obtained for a well tuned controller. The following section will demonstrate how the controller can be tuned online.

9.2.3 Online Parameter Tuning

A powerful feature of the is the ability to modify controller parameters in real-time without interrupting its execution. Fig. 9.9 shows the discrete controller subsystems used for the speed and current control loops respectively. While any parameter on the diagram can be modified, for the purpose of demonstration only the gains Kp are adjusted. The plots in Fig. 9.6 were obtained with Kp set to the values determined from simulation, namely 15 for the speed and 40 for the current controller. Fig. 9.10 shows the dialog box used to modify parameters. After a new value is entered and **Apply** is pressed the CSDE automatically updates the parameter in the real-time prototype and the effect can be observed on the Display plots.

Table 9.1 lists the various Kp settings and the figures which show the corresponding plots obtained. These plots clearly illustrate the effect of the parameter changes on the controller response. The important point to keep in mind is that the various settings were tried out in real-time in a matter of a few minutes. There was no need to rewrite or regenerate any code and the machine was constantly online during the experiment.

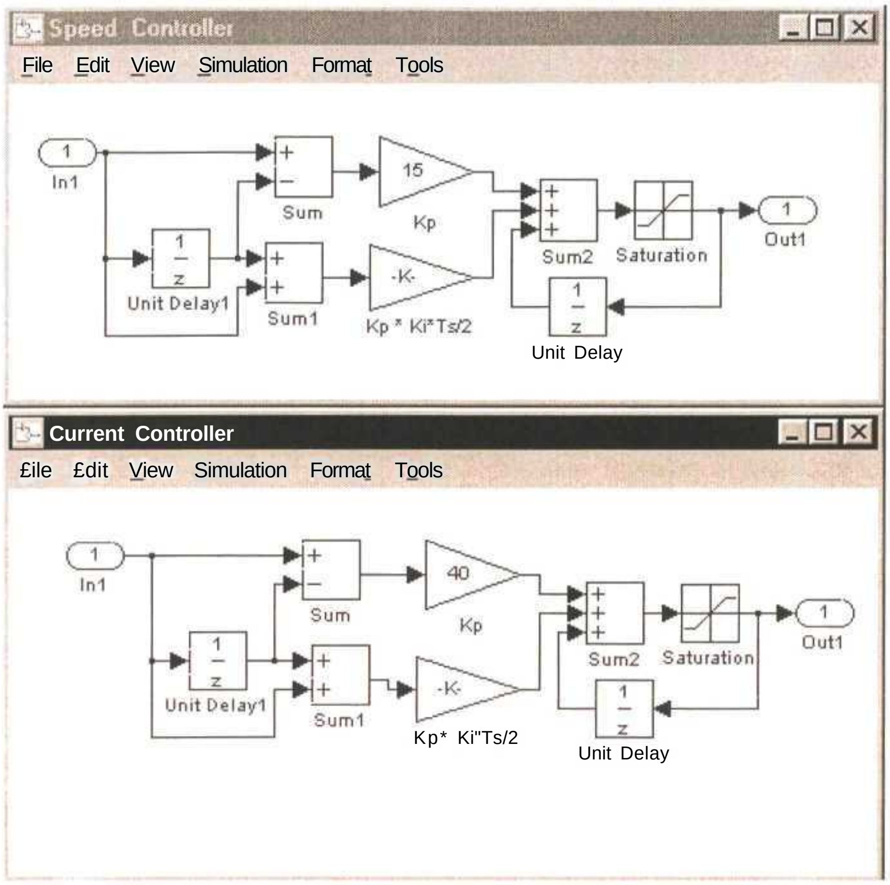
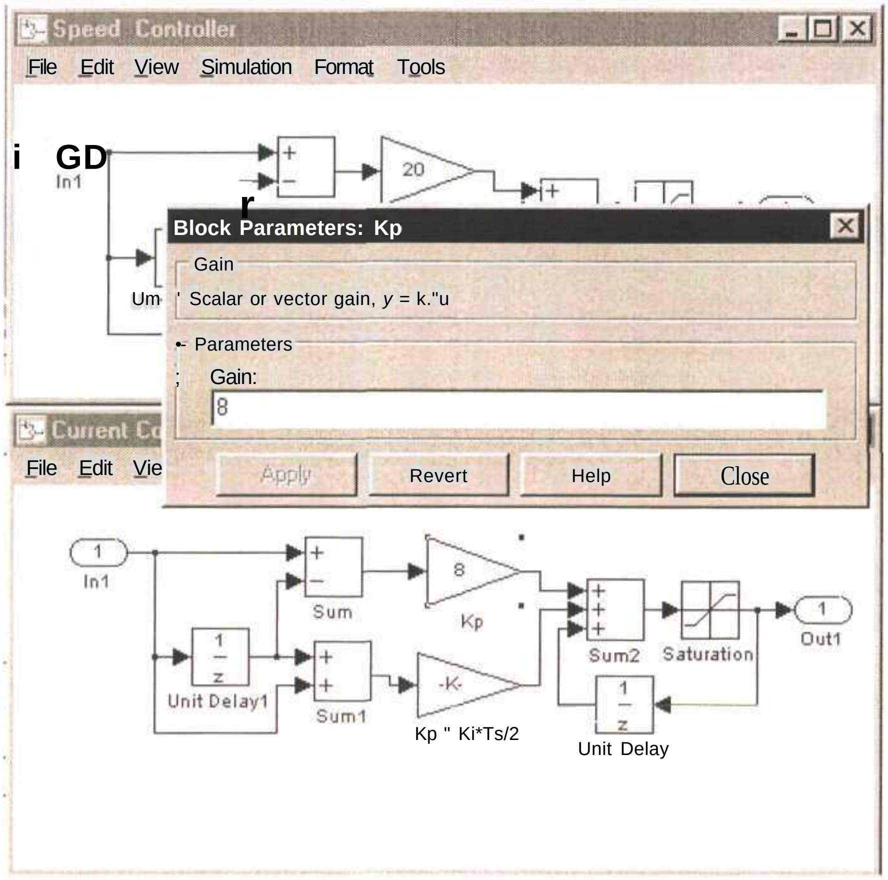


Fig. 9.9 : Discrete controller subsystems

Speed controller Kp	Current controller Kp	Figure
20	**40**	Fig. 9.6
10	**40**	Fig. 9.11
4	**40**	Fig. 9.12
20	**8**	Fig. 9.13

Table 9.1 : Various controller gain combinations



Block Parameters: Kp

Gain

Scalar or vector gain, *y* = k."u

Parameters

Gain: 8

Fig. 9.10 : Modifying parameters online

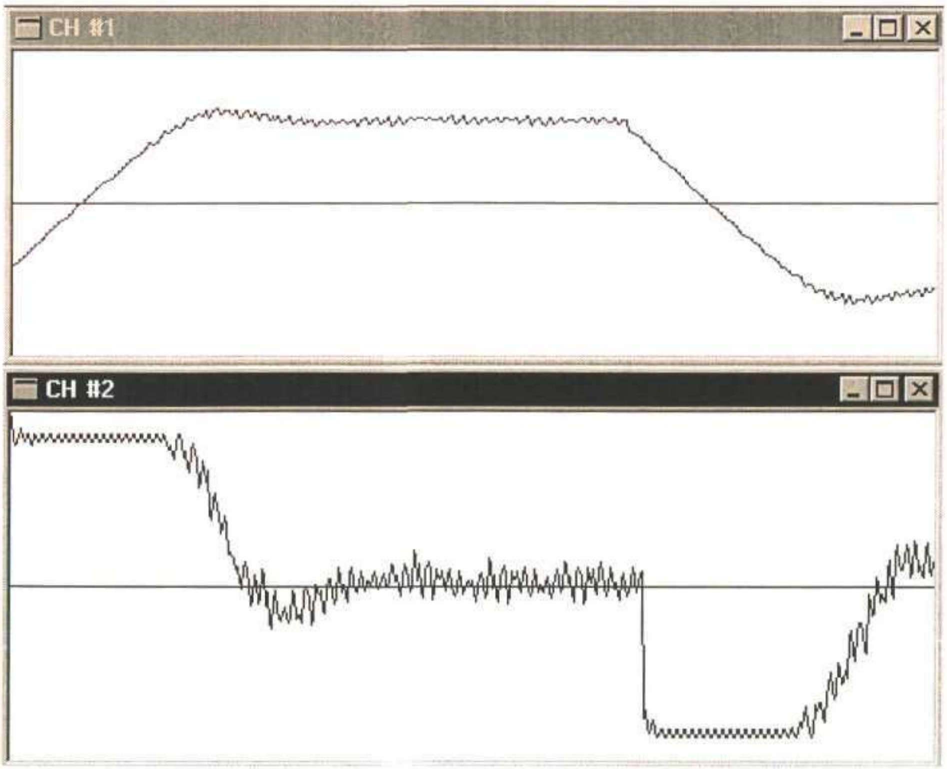


Fig. 9.11 : Speed Kp = 10, Current Kp = 40

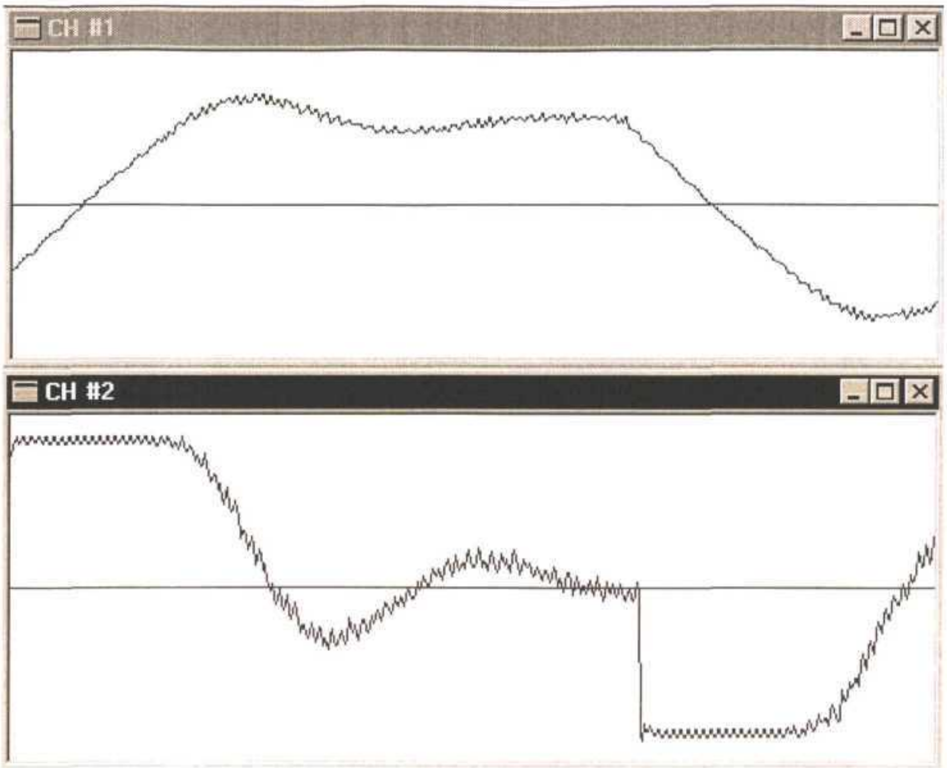


Fig. 9.12: Speed Kp = 4, Current Kp = 40

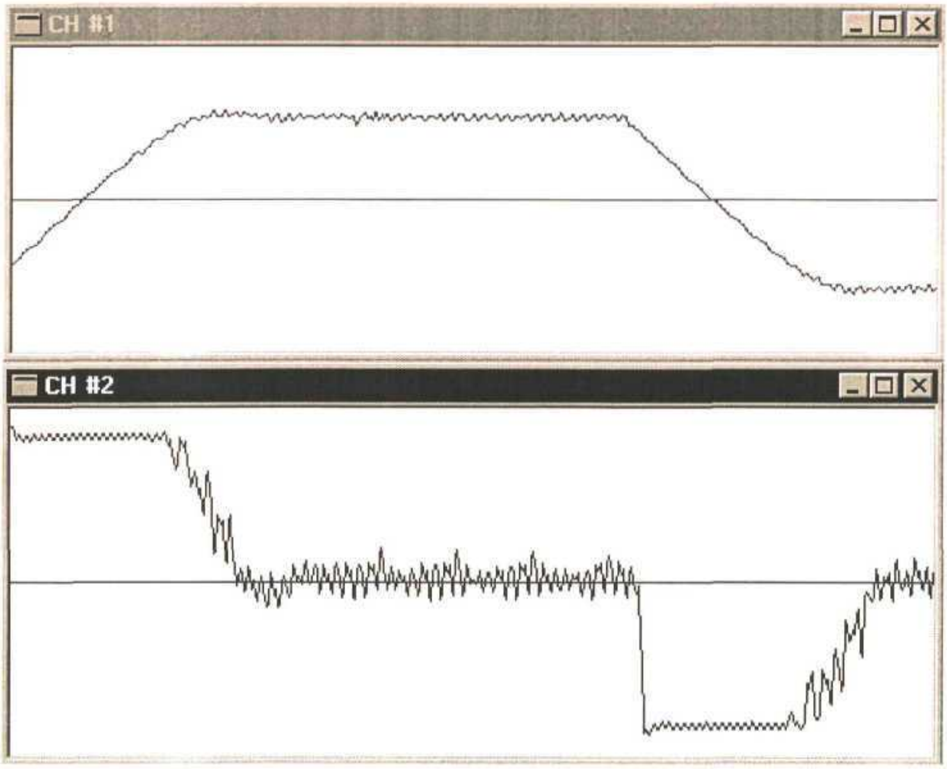


Fig. 9.13 : Speed Kp = 20, Current Kp = 8

9.3 Student Projects - Case Studies

The demonstration in section 9.2 shows that the CSDE functions as intended by the author and fulfills all requirements laid down in Chapter 2. However, the main purpose of the project was to create a research tool aimed at the undergraduate students. Thus, the author felt it was important to allow a number of students to use the CSDE during their design courses in order to verify its usefulness in this environment.

Due to limited resources only one complete rapid prototyping station could be made available to undergraduate students. The setup included the following :

(i) Pentium PC running Windows 95, with the full CSDE software installed
(ii) PC32 DSP Controller card
(iii) PWM / Tacho expansion card
(iv) H-bridgc inverter
(v) Current and speed sensors
(vi) A choice of either a DC motor or a 220 V 3-phase induction motor

During the 1998 and 1999 academic years a number of 3rd and 4th year students used the above setup to implement their motion control designs at the Department of Electrical Engineering at UND. Although no formal record was kept, in all cases the feedback was positive and the students felt that the CSDE allowed them to complete more challenging projects than their colleagues in less time. The feedback from the field trials allowed a number of improvements to be made to the CSDE, as listed in sections 9.4.1 and 9.5.1.

Two of the final year designs which made use of the CSDE are briefly introduced in the following sections. An in-depth discussion of these projects is beyond the scope is of this thesis, and only an overall outline is provided. Both projects were supervised by Mr. Diana.

9.4 Field Oriented Control of an Induction Machine

The method of FOC [LEONHARD1] attempts to change the complex characteristics of an induction machine to be more like the simpler DC counterpart. The algorithm attempts to establish a fixed angular relationship between the stator current vector and the rotor flux vector and in doing so decouples the two values. Effectively, if the flux is kept constant the torque of