AD-A219 356

DTIC FILE COPY (2)

NCLASSIFIED MASTER COPY - FOR REPRODUCTION PURPOSES

RITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

DTIC ELECTE MAR 13 1990 S D D Cs

EPORT SECURITY CLASSIFICATION: Unclassified

1b. RESTRICTIVE MARKINGS

ECURITY CLASSIFICATION AUTHORITY

3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.

ECLASSIFICATION/DOWNGRADING SCHEDULE

RFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S): ARO 26779.8-EL-AI

6a. NAME OF PERFORMING ORGANIZATION: Center of Excellence in AI, University of Pennsylvania

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION: U. S. Army Research Office

6c. ADDRESS (City, State, and ZIP Code): Dept. of Computer & Information Science, 200 S. 33rd Street, Philadelphia, PA 19104-6389

7b. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211

8a. NAME OF FUNDING/SPONSORING ORGANIZATION: U. S. Army Research Office

8b. OFFICE SYMBOL (If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAL03-89-C-0031

8c. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification): A New Approach to Laboratory Motor Control MMCS: The Modular Motor Control System (MS-CIS-89-17)

12. PERSONAL AUTHOR(S): Peter I. Corke

13a. TYPE OF REPORT: Interim technical

13b. TIME COVERED: FROM ____ TO ____

14. DATE OF REPORT (Year, Month, Day): February 1989

15. PAGE COUNT: 53

16. SUPPLEMENTARY NOTATION: The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Robotics, motion control systems. (SDW)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Many projects within the GRASP laboratory involve motion control via electric servo motors, for example robots, hands, camera mounts and tables. To date each project has been based on a unique hardware/software approach.

This document discusses the development of a new modular, and host independent, motor control system, MMCS, for laboratory use. The background to the project and the development of the concept is traced.

An important hardware component developed is a 2 axis control motor control board that can be plugged into an IBM PC bus or connected via an adaptor to a high performance workstation computer.

To eliminate the need for detailed understanding of the hardware components, an abstract controller model is proposed. Software implementing this model has been developed in a device driver for the Unix operating system. However for those who need or wish to program at the hardware level, the manual describes in detail the various custom hardware components of the system. Keywords: ( )

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE (Include Area Code)

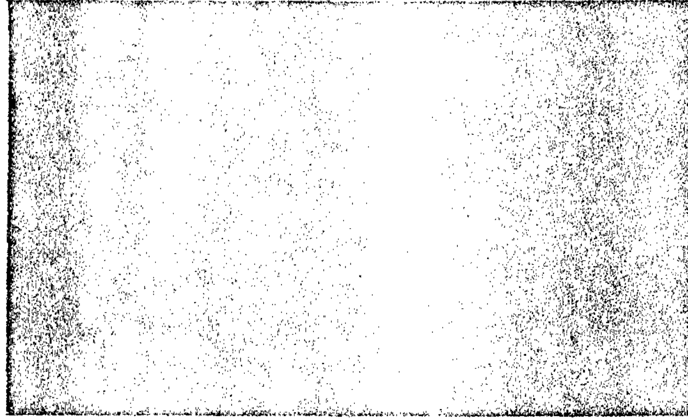
22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE: UNCLASSIFIED

20040721197

UNIVERSITY of PENNSYLVANIA



Department of Computer and Information Science
School of Engineering and Applied Science
Philadelphia, PA 19104-6389

PENN

90 03 13 023

A NEW APPROACH TO LABORATORY MOTOR CONTROL MMCS THE MODULAR MOTOR CONTROL SYSTEM

Peter I. Corke

MS-CIS-89-17
GRASP LAB 175

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

February 1989

Accesion For
NTIS CRA&I
DTIC TAB
By
Distribution /
Dist
A-1

Acknowledgements: This work was supported in part by NSF IRI84-10413-AO2, MCS-8219196-CER, CCR8716975, NSF/DCR grants 8501482, NSF/DMC 8512838, U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

A New Approach to Laboratory Motor Control

MMCS

The Modular Motor Control System

Peter I. Corke[1]
pic@grasp.cis.upenn.edu

Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19104

February 23, 1989

[1]Research Scientist, CSIRO Division of Manufacturing Technology, Melbourne, Australia.

Abstract

Many projects within the GRASP laboratory involve motion control via electric servo motors, for example robots, hands, camera mounts and tables. To date each project has been based on a unique hardware/software approach.

This document discusses the development of a new modular, and host independent, motor control system, MMCS, for laboratory use. The background to the project and the development of the concept is traced.

An important hardware component devloped is a 2 axis control motor control board that can be plugged into an IBM PC bus or connected via an adaptor to a high performance workstation computer.

To eliminate the need for detailed understanding of the hardware components, an abstract controller model is proposed. Software implementing this model has been developed in a device driver for the Unix operating system. However for those who need or wish to program at the hardware level, the manual describes in detail the various custom hardware components of the system.

Contents

1 Introduction **1**
- 1.1 Overview 1
- 1.2 Background and Motivation 2
- 1.3 System overview 4
  - 1.3.1 Control Processor and Software 4
  - 1.3.2 Motor interface 6
- 1.4 Acknowledgements 7

2 Application model of the motor controller **8**
- 2.1 Compensator 9
  - 2.1.1 The general transfer function 9
  - 2.1.2 PID implementation 10
  - 2.1.3 PD implementation 10
  - 2.1.4 PI implementation 10
- 2.2 Control options 11
  - 2.2.1 Velocity servo 11
  - 2.2.2 Torque servo 11
  - 2.2.3 Coulomb friction compensation 11
  - 2.2.4 Feedback source 11
  - 2.2.5 Setpoint source 12
- 2.3 The Unix device driver 12
  - 2.3.1 Configuring the servo 12
  - 2.3.2 Choice of parameters 14
  - 2.3.3 Accessing servo state 15
  - 2.3.4 Error handling 16
  - 2.3.5 Other device driver functions 17
  - 2.3.6 Code example 17
- 2.4 Accessing hardware directly 17
- 2.5 Control synthesis 17

3 mcTool **25**

4 Host adaptor 28
- 4.1 Introduction 28
- 4.2 Adaptors in general 28
  - 4.2.1 Generic specification for adaptors 28
  - 4.2.2 PCbus signals redefined 29
- 4.3 In particular: VMEbus adaptor 29
  - 4.3.1 VME memory Map 29
  - 4.3.2 PC bus access 29
  - 4.3.3 Adaptor Control Registers 32
  - 4.3.4 The Servo Clock 32
  - 4.3.5 Panic signal 33
  - 4.3.6 Interrupts 33
  - 4.3.7 LED indicators 33
  - 4.3.8 Miscellaneous Notes 33

5 The Mark I motor interface card 36
- 5.1 Servo board specification 36
- 5.2 Design aims 36
- 5.3 Description 37
  - 5.3.1 Memory map 37
  - 5.3.2 The latch signal 39
  - 5.3.3 D/A double buffering 39
  - 5.3.4 Calibration 41
  - 5.3.5 Diagnostics 41
  - 5.3.6 Panic signal 41
- 5.4 Board details 42
  - 5.4.1 Switches 42
  - 5.4.2 LED indicators 43
  - 5.4.3 Configuration 43
  - 5.4.4 Pinouts 44

List of Figures

1.1 Notional controller structure 5

2.1 Motor controller block diagram 9
2.2 MMCS code example 20
2.3 MMCS code example 21
2.4 MMCS code example 22
2.5 SunOS code example for direct hardware access 23

4.1 VME Host adaptor memory map (byte addresses shown) 30
4.2 PC bus I/O space address formation 31

5.1 Servo board memory map 38
5.2 Motor interface board layout 42

Chapter 1

Introduction

1.1 Overview

This first chapter discusses the motivation for a new modular, and host independent, motor control system for laboratory use. The background to the project and the development of the concept is traced. A number of possible solutions are proposed and discussed, leading to a general description of the system that has been implemented.

Chapter 2 describes in detail an abstract programmer's model of the axis controller. Details of the servo interface hardware are hidden, allowing the applications programmer to concentrate on higher level control. The software implements position, velocity or torque control, selectable per axis, and the closed loop dynamics may be modified by a digital compensation network.

Chapter 3 describes an interactive graphical tool that allows a user to configure the axis controller, perform diagnostics and perform joint level motions.

The last two chapters are not essential reading for casual programmers, but are essential for those programming at the hardware level.

Chapter 4 describes the function performed by the host bus adaptor and also the axis controller bus, which is the same as IBM/PC bus. Details such as redefinition of some signal lines[1], and the addressing conventions used are covered. It describes in detail the hardware implementation of the VME host to PC bus adaptor that was built.

Chapter 5 describes in detail the hardware and programming details for the Mark I servo interface board.

[1] It's not as bad as it sounds

1.2 Background and Motivation

Many projects within the GRASP lab. involve motion control via electric servo motors, for example robots, hands, camera mounts and tables. Each project has been based on a unique hardware/software approach. In the last few years the approaches have included

- VAL-II control language receiving commands over a serial line from a host computer
- RCCL (Hayward and Paul)
- RFMS multiprocessor (Zhang and Paul)

The first approach is limited by communications speed, and is not suitable for real-time sensor based control. The RFMS controller has proved in practice to be very difficult to program, and does not seem to have realized the full potential of its parallel hardware architecture.

RCCL is a very general robot programming environment and is capable of real-time sensor based control, as has been demonstrated by various projects within the lab. It does have the drawback that it is tightly coupled to the VAX architecture and Unimate robots and their controllers

RCCL provides the programmer with a particular model of the robot and its environment. This model, based on kinematic position equations and cartesian representation using homogeneous transforms, is very powerful, however there are many applications to which it is not well suited. It is at this point that the inherent inflexibility of RCCL becomes a problem, and the application programmer's effort goes increasingly into outwitting and thwarting RCCL "features".

Based on discussions with robot users in the laboratory the following points were made

1. Robot control hardware. It was considered that the best platform for robot control would be a powerful single processor system like a workstation. A single thread machine is inherently easier to program, and a workstation provides an integrated environment for program development and high speed execution. To allow a workstation to perform robot control an interface is required to the robot's electronic subsystems.

2. Robot interface. The RCCL controllers use a relatively high level interface to the Puma robot. The Unimate controller boxes provide position servo capability, A/D[2] and D/A[3] converters etc. A functionally more general interface was designed for the RFMS project, but the interface was physically limited to use within the RFMS(board size, connectors etc).

[2] Analog to digital
[3] Digital to analog

Professor Paul commisioned a final year project to build a general purpose 6 axis interface for the MicroVAX Qbus, but this was never finished and there is some doubt as to whether MicroVAXs and Qbus are the hardware platform to use in the future.

The author suggested a more general solution, based on the technology developed for the RFMS. The axis controller would be modular, thus allowing it to be expanded easily to cope with changing requirements, for example 7 axis robot, robot + hand, or two cooperating robots. Most importantly the axis controllers would be independent of the host processor bus, whether it be Multibus, VMEbus or Qbus. A simple electronic adaptor would connect the axis controller bus to the host bus, and would represent a relatively small fraction of the total system complexity, thus allowing easy migration to new host computing platforms. It was decided that the axis controller bus should be the IBM-PC bus, due to the variety of compatible products in the marketplace.

3. Robot control software. Based on experience with RCCL and CSIRO's ARCL robot controller[5] it has been decided to redesign the robot control software so as to be very modular, as opposed to the "monolithic" structure of RCCL. The structure looks like comprising a number of simple interfaces and functional blocks, implemented as libraries, and on which the applications programmer can build. The detailed work would be tackled by Gaylord Holder as a Master's project. A number of considerations in the design are:

   - at the lowest level it must be able to interface with the existing RCI interface to Unimate controllers, as well as the new MMCS hardware.
   - at the highest level it must provide a similar level of functionality to the RCCL programming environment, since this is one (despite its limitations) with which many workers are familiar. Within this new programming environments different programming tools will hopefully spring up and eventually replace RCCL.

To restate this, a new motion controller should

- be based on a fast single thread processor
- contain a host independent and modular motor interface
- be accessible via a small and modular software library

The remainder of this document is concerned with the first two points only.

1.3 System overview

This section provides an overview of the hardware and software components of MMCS.

1.3.1 Control Processor and Software

The control processor has two main computations to perform

- High rate servo control loops for the motors
- Slower rate trajectory generation

In the existing Unimate controller the servo control loop functions are performed per axis by an 8 bit 6503 microprocessor, see Figure 1.1. To achieve high joint stiffness and dynamic performance a sample rate approaching 1kHz is desired. There is no way that a process running under Unix (on a 1988 vintage workstation) can achieve this order of response, thus the alternatives are to

1. use a separate processor to perform the high speed servo calculations. The software could run on "bare-metal" or under a real-time operating system.
2. perform the servo computation at interrupt level in a Unix device driver.

The first approach offers the most flexibility, and there are a number of possibilities for a separate processor including

- 680x0 VME CPU cards manufactured from many sources. These processor boards can plug into the VME backplane and communicate with the host via shared memory. Having the same instruction set as the host eliminates the need for software cross development tools. Communications and support software could be developed, or an off-the-shelf package such as VxWorks could be utilized.
- Bell Labs J[illegible]FE[3] processor, which can be plugged into a SUN workstation, and has full software support including a C compiler, and host communication facilities.

The second approach is lower in cost but not as flexible. With an attached processor the user can code up an experimental servo algorithm, download then run it. However, a servo loop in the kernel means that the user would have to rewrite the driver, link a new kernel and boot it. Debugging tools exist for the kernel but they are primitive. More seriously kernel code cannot use floating point arithmetic. Such an approach, under the Xenix operating system, has been described[4].

A variation on this theme is the RCI package written by John Lloyd[9] in which the kernel interrupt handler invokes a user process function in kernel

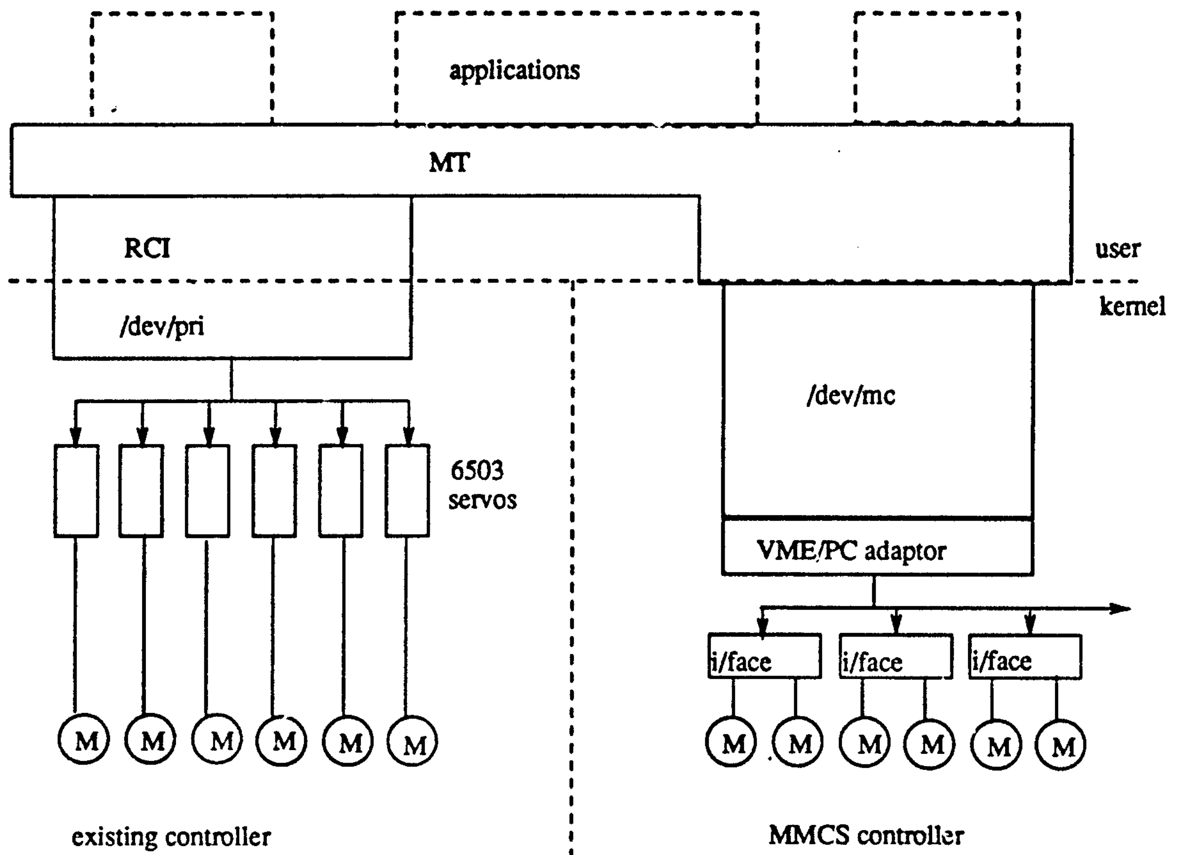


Figure 1.1: Notional controller structure

mode. This provides run time linkage of user code into the kernel, but debugging remains difficult and the user code must obey some strict guidelines.

The approach taken in this project is to build hardware consistent with both approaches, but the first implementation will be use servo loops embedded in a device driver. The driver implements a very general servo loop capable of being configured for position, velocity or torque mode operation. Any application that wishes to can bypass the servo loops and specify motor currents directly (torque mode). In this case the control algorithm is running in a user process and its scheduling cannnot be guaranteed, with possibly serious consequences for stable and smooth control. This is unavoidable when working under Unix.

Comparison of the two approaches in Figure 1.1 shows that the functionality of the six 6503 servo cards in the Unimate controller has been shifted to the host computer. Simulations of the servo software for 6-axis computation time was done for a number of processors and the results are summarized below

Processor	Time (ms)
Sun 3/160	0.43
MicroVAX 3500	0.25
MicroVAX II	0.70

This indicates that Unix device driver based servo loops can provide satisfactory sample rates on most of the GRASP laboratory machines. The simulation does not take into account effects such as adaptor hardware access time, interrupt latency, or service overhead time.

1.3.2 Motor interface

The motor interface was designed with the following design aims

- make it as host bus independent as possible
- make it as modular as possible
- use as much of the proven iSBX design[7] as possible
- control up to 16 axes

To achieve this the hardware has been partitioned into three components

1. Motor interface hardware that provides current drive signal to the motor and processes signals from sensors regarding the motor's state.
2. Axis controller bus into which motor interface cards are plugged.
3. Host adaptor to connect motor interfaces to a host computer that will perform the servo computations.

The motor interface is the electronics that connects the motor to the axis controller bus. It provides an analog drive signal to the motor, and measures shaft angle via an incremental encoder, as well as application specific quantities via a general purpose analog input.

Control of a system with upto 16 axes introduces a number of problems such as timing skew between sampling the first and last axis. If the system is connected to a Unix host computer we cannot rely on software, even at driver level to initiate sampling since interrupt latencies can vary by upto 100 μsec. Thus it was seen to be essential that sampling is controlled by a hardware clock, and the host is notified by interrupt so that it can read and processs the state information. The hardware clock signal, SCLOCK, is common to all motor interface cards.

Since the host is also interrupted by the SCLOCK, by the time the interrupt handler routine is entered all state information is available to be read. This means that A/D conversion time is overlapped with the interrupt service overhead for maximum efficiency.

Safety considerations indicate that every motor interface should have the ability to indicate its readiness for operation or an error condition. This signal, referred to as PANIC, is also common to all interface cards, anyone of which can assert the line to indicate a system failure. The type of failure can be determined by host software polling all cards.

The axis controller bus needs to be an ordinary computer bus with address, data and control signals, but it also needs to have the SCLOCK and PANIC signals. It was initially decided to define and use a custom bus for this purpose, but later the decision to use the IBM PC bus was made. The PC bus is nearly ideal in that it is simple to interface to, and a wide range of peripheral cards is available for it. The two special signals could have been implemented by a separate ribbon cable linking all boards, but this is not failsafe in that it is possible for cards to be not connected. Instead two signals from the IBM PC bus were "redefined" for these purposes and is discussed further in Chapter 4.

1.4 Acknowledgements

Professor Richard Paul provided the impetus and support for the project.

Dave Feldman did the detailed design of the servo board and design and constructruction of the VME/PC bus adaptor. Filip Fuma and Mat Donham designed and built the iSBX cards used in the RFMS.

Chapter 2

Application model of the motor controller

A general model for servo motor control is proposed. It is capable of performing most commonly required functions such as closed-loop position or velocity control. If this functionality is not required the application can also directly specify motor current demands. The motor controller referred to here comprises

- The motor interface hardware described in detail in Chapter 5.
- Servo software in the kernel of the host computer

Since future interface boards may incorporate more control functionality, an abstract user model allows the software/hardware balance to be changed without applications being recoded.

Figure 2.1 shows the proposed control model. A number of switches S1...S5 control various operating modes of the controller. Feedback can come from one of two sources (S4)

- An incremental shaft angle encoder
- An A/D converter

A derivative block may be switched into the feedback path (S5) which will cause a velocity servo function to be implemented. The setpoint signal may come from either (S1)

- The application program via a write() system call, which is represented by s in Figure 2.1.
- The A/D converter

S3 switches in an optional Coulomb friction compensation block, while S2 bypasses the feedback control and allows the setpoint to control motor current directly.

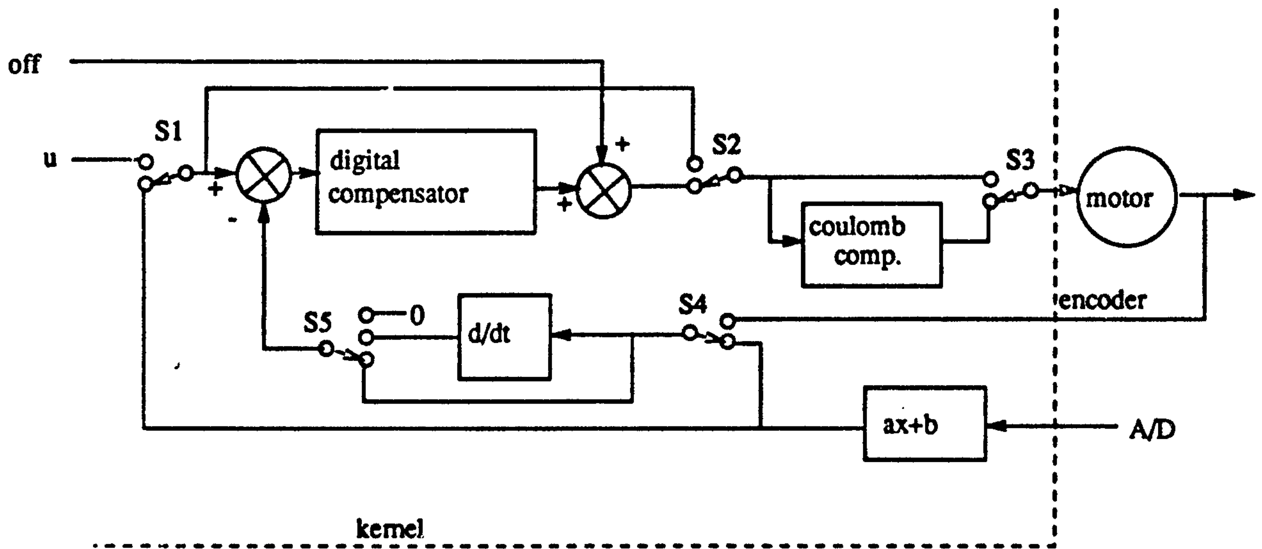


Figure 2.1: Motor controller block diagram

2.1 Compensator

A compensator is included in the forward path to allow users to tailor the dynamic response of the closed-loop system. Typically for a DC electric servo motor, the transfer function is

$$\frac{\Theta}{I} = \frac{k}{s(Js + B)}$$

where θ is motor shaft angle, i is motor current, k is the motor torque constant, J is the motor inertia comprising self and reflected load inertia, and B is viscous friction.

To provide position control, feedback is required, and to achieve good dynamic peformance and disturbance rejection some compensation is required.

2.1.1 The general transfer function

The controller implements a unity gain negative feedback loop on position, with a general discrete transfer function compensator as shown in Figure 1. The transfer function is

$$D(z) = \frac{a_2 z^{-2} + a_1 z^{-1} + a_0}{b_2 z^{-2} + b_1 z^{-1} + b_0}$$

where the coefficients a_i and b_i are programmable by the user. The DC gain is given by $\sum a_i / \sum b_i$. All coefficients and quantities are 32 bit signed integers, so care must be given to the scaling of the integer coefficients.

Many design methodologies may be used to synthesize the compensator coefficients[6]. If a continuous time transfer function is synthesized, perhaps using any of the standard forms discusssed below, techniques such as bilininear transform, Z-transform or pole/zero mapping[6] may be used to generate equivalent discrete time transfer functions. Details of some commonly used control strategies are given below. An example of synthesis for a PID control law is given in Section 2.5.

2.1.2 PID implementation

The classical continuous time PID controller has a transfer function of

$$u = Pe + D\frac{d}{dt}e + I\int edt$$

where e is the error, demanded minus measured plant output. This may be Laplace transformed to

$$\frac{U}{E} = P + Ds + I\frac{1}{s}$$

$$\frac{U}{E} = \frac{Ps + Ds^2 + I}{s}$$

from which it is clear that the transfer function has a pole at the origin, $s = 0$, and a complex pair of zeros affected by the parameters P, I, and D.

2.1.3 PD implementation

The transfer function of a PD controller is

$$\frac{U}{E} = P + Ds$$

which has a zero at $s = -P/D$.

2.1.4 PI implementation

The transfer function of a PI controller is

$$\frac{U}{E} = P + \frac{I}{s}$$

which has a zero at $s = -I/P$, and a pole at the origin.

2.2 Control options

2.2.1 Velocity servo

A velocity servo loop may be implemented by switching in a differentiator (S5) to the position feedback path. The differentiator is implemented by a three point derivative

$$\frac{dy}{dt} \approx \frac{3y_t - 4y_{t-1} + y_{t-2}}{2}$$

to yield a smoother velocity estimate. Note that the velocity units (see S4) are either encoder counts, or transformed A/D units, per sample interval.

2.2.2 Torque servo

For torque control, the compenstation computation is completely switched out (S2), and the user specified value is used directly as motor current demand.

2.2.3 Coulomb friction compensation

Coulomb friction is a non-linear effect, in which an approximately constant torque opposes the motor's torque. The friction torque is not necessarily the same for each direction of rotation, and varies with joint loading, and will thus be somewhat configuration dependant. An optional Coulomb friction feedforward function may be enabled (S3) to compensate for this non-linear effect. The compensator implements the control law

$$i_m = \begin{cases} i + i_{cp} & \text{if } \dot{\theta} > 0 \\ i - i_{cn} & \text{if } \dot{\theta} < 0 \end{cases}$$

where i is the output of switch S2.

If the velocity is zero, and a non-zero current is specified the sign of the current demand (from the digital compensator) is used, since that indicates the direction of desired motion.

i_{cp} and i_{cn} are the currents required to overcome the Coulomb friction torques in the positive and negative rotational directions respectively.

2.2.4 Feedback source

The feedback signal (S4) may come from either the shaft angle incremental encoder θ, or from the A/D converter associated with the axis. The raw data from the A/D converter is processed with a simple linear law

$$\phi = aADC + b$$

that provides scaling and offset before it is used as the feedback signal.

For example, opening the feedback path (S5), and selecting demand from the A/D (S1), the servo will implement a programmable digital filter between A/D and D/A. Application software could also log the raw or filtered signal.

2.2.5 Setpoint source

The setpoint, or demand signal may come from one of two sources (S1) as shown in Figure 2.1. Normally it would be supplied by the user's application program to the device driver via a write() system call. However it may be selected to come from the processed A/D signal, ϕ.

2.3 The Unix device driver

A SunOS device driver (/dev/mc) has been written to implement the application model of the controller, and is described in this section. The mc device driver does not support the many individual features of specific motor interfaces. To access these capabilities it is probably more effective to map to the device hardware from Unix, and directly manipulate control registers as described in section 2.4.

2.3.1 Configuring the servo

Every axis has a parameter structure which describes the mode of operation to the mc device driver.

```
#include <sys/mcdef.h>

/*
 * Per joint parameter structure
 */
struct mc_param {
        int     which;
        int     a2, a1, a0, b2, b1, b0; /* compensator coefficients */
        int     ic_pos, ic_neg;         /* feedforward constants */
        int     mode, clkdivisor;
        int     ilim, ilimmax;          /* current limit */
        int     plo, phi;               /* position limits */
        int     adc_a, adc_b;           /* adc conversion law */
        int     ipole;                  /* current filter pole */
};
```

The units for *ic_pos*, *ic_neg* and *ilim* are D/A converter units. *plo* and *phi* are in the units of whatever feedback source is selected. *clkdivisor* is the number

of hardware clock ticks between servo computations for this axis. That is, each axis may be servoed at a sub-multiple of the hardware clock rate.
Possible values for mode are

MD_OFF	Not servoed
MD_POS	Position control mode
MD_VEL	Velocity control mode
MD_TORQ	Torque (current) control mode
MD_TEST1	Generate triangle waveform

Additional values may be or'd with the mode word, such as

COULCOMP	Enable the Coulomb friction feedforward (S3)
ADFB	Feedback comes from ϕ not θ (S4)
ZEROFB	Zero feedback, that is, open-loop operation (S5)
ADDMD	Demand comes from ϕ not the computer (S1)
ADOFF	Torque offset comes from ϕ not the computer
POSCHK	Check position limits on feedback signal
SOFTERR	Don't shut motors down when error is detected

Note that not all switch combinations are useful, and this is not checked. The servo parameters for an axis are set or examined using an ioctl() system call on the mc device. To retrieve parameters from an axis the *which* element must first be set to indicate which axis the parameters are required for

```
struct mc_param         par;

par.which = axis;
ioctl(mcfd, MGETPARAM, &par);
```

mcfd is the file descriptor for the motor control device, /dev/mc0. To set parameters the parameter structure should be initialized by the user's program, and the *which* element set to indicate which axis the parameters are destined for.

```
struct mc_param         par;

par.which = 3;
par.mode = MD_VEL;
ioctl(mcfd, MSETPARAM, &par);
```

Only one axis may be initialized per ioctl(). All parameters are initialized as shown in Table 2.1 when the device is opened.

Also at open time the driver scans the axis controller bus looking for motor interface cards from axis 0 through axis 15. When the device is closed, all motor currents are set to zero and the PANIC signal asserted.

Structure element	Initial value
a0	1
a1	0
a2	0
b0	1
b1	0
b2	0
ic_pos	0
ic_neg	0
ilim	1/2 maximum current
ilimmax	0
adc_a	1
adc_b	0
mode	MD_OFF
clkdivisor	1
ipole	0

Table 2.1: Initial parameter values

2.3.2 Choice of parameters

It may appear that there is an overwhelming number of parameters to set before anything can be done. This is true, and unavoidable with the general appoach taken. For the Unimate controller many of these issues are handled by the 6503 microprocessors and the code they execute from EPROM. The Unimate servos have been tuned for good performance with the motors and mechanical systems used. For MMCS these parameters must be determined by the user, there is no alternative. The interactive tool *mctool* can let a user adjust controller parameters to obtain good performance. Another approach would be an adaptive control algorithm that modelled the motor from input/output data and computes the compensator parameters for user specifed closed-loop poles.

The controller parameters are very dependent upon the type of motor, the power amplifier, and mechanical drive train. Parameters that work well for one motor may cause instability with another.

However some practical hints are in order

- Determine which direction the encoders change when a positive torque is applied to the motor. If the encoders increase in a negative direction then the DC gain of the compensator must be negative, and both Coulomb friction compensation paramters must be negative.

- For position mode control PD control is appropriate, for velocity mode PI control is appropriate.

- The maximum current should be left at some low value (default is half

maximum) until the controller parameters and application are well behaved.

2.3.3 Accessing servo state

The driver maintains state information for each axis.

```
/*
 * Per joint state and status
 */
struct mc_state {
        int     adcval;         /* latest A/D value (processed) */
        int     encnow;         /* latest encoder value */
        int     inow;           /* last current command issued */
        int     error;          /* latest error value */
        int     fbnow;          /* latest value of feeback quantity */
        int     vel;            /* latest velocity estimate d/dt {fbnow} */
int errcode;        /* current errors on this axis */
};
```

A read() system call on the mc device returns a vector of *mc_state* structures, which may be used by the user as required. The state variables are

adcval The instantaneous value of the A/D after processing via the linear law.

encnow The instantaneous value of the incremental encoder counter.

inow The instantaneous or filtered motor current demand, is related to the torque needed to maintain the position or velocity demand set. It will be related to disturbance forces such as gravity or robot/object interactions. If the parameter *ipole* is zero *inow* is the instantaneous current. If non-zero, then the motor current demand is filtered by a unity gain first order digital filter whose pole is at $ipole/MC_FSCALE$, and the value of *inow* should be divided by MC_FSCALE to convert from fixed point filter arithmetic to real value.

error The instantaneous error between the feeback quantity and the demand.

fbnow The instantaneous value of the feedback quantity, which will always be the same as either *encnow* or *adcval*.

vel The current plant output velocity estimate, in units per sample period.

errcode The last error code that occurred for this axis.

MERR_PLO	Low position limit crossed
MERR_PHI	High position limit crossed
MERR_ILIM	Sustained current overload
MERR_PANIC	Hardware panic detected
MERR_ZINDEX	Zero index detected in CALIB mode
MERR_MASK	Mask for actual error code bits

Table 2.2: Device error codes

State information is always available once the mc device is open. It is updated at every SCLOCK, the sample interval is set by the MSETINTERVAL ioctl() call.

2.3.4 Error handling

The MMCS subsystem can generate a number of error conditions. The error status for each axis is held in the *errcode* element of the motor's state structure. The value may be any of the codes shown in Table 2.2. The bit MERR_POSERR, if set, indicates that a position error, either high or low occurred. Since the state structures are read-only the error status of all axes can only be cleared by the MCLEARERROR ioctl() which will usually be called from the user's error handling code.

On all error conditions, and zero index detection, the application process is notifed by a signal SIGUSR1. If the operting mode of the axis is or'd with SOFTERR then no further action is taken by MMCS, and the user's signal handler is responsible for dealing the condition. If SOFTERR is not set the robot shutdown by giving a zero current demand to all motors, and activating the brakes.

The driver always checks for sustained torque overdrive of the motor. If the current demand exceeds the parameter *ilim* for more than *ilimmax* samples the MMCS is shutdown. Motor current is always clipped to the maximum value allowed by the D/A converter. If *ilimmax* is zero, then motor currents are clipped to *ilim* and no error condition is generated.

If *mode* has the POSCHK bit set then the feedback quantity, from encoder or A/D is checked against the limit parameters *phi* and *plo*. The error condition is generated only when the limits are crossed, not continuously while the limit exists.

A hardware panic is initiated by one of the motor interface cards, or the hand held panic button. The failsafe nature of the system design means that panic will also be asserted if connections such as that between host and MMCS, or MMCS and panic button are broken. The axis number bitfield in the error code is meaningless for this condition.

When an axis is placed in calibrate mode using the MCALIB ioctl(), then the first detected zero index will send a SIGUSR1, and switch that axis out of

calibrate mode. The high order 16 bits are the encoder value at the time of the zero index.

2.3.5 Other device driver functions

Functions, not already discussed, that can be controlled via ioctl() calls are given in Table 2.3.

2.3.6 Code example

Figure 2.4 is a code fragment that illustrates the important steps in controlling motors via the mc device driver.

2.4 Accessing hardware directly

This approach to interfacing, mapping device hardware registers to a Unix process, is very specific to the flavor of Unix being used. Some likely approaches are Ultrix via the /dev/bus device driver, or SunOS via the /dev/vme* device drivers. User's following this path should be very familiar with the material in Chapters 4, 5 and the appendices. A SunOS code fragment is given in Figure 2.5.

For more details consult the Unix manual entries for valloc(2) and mmap(2). If an access is made to a address at which no device resides, the VMEbus times out and a SIGSEGV (segmentation violation) signal is delivered to the user process. A SIGBUS (bus error) signal can be delivered if the device hardware messes up the VME cycle.

Note that accesses to devices via mapped memory cause "non-priviliged" address modifiers to be issued while accesses from a device driver cause "priviliged" address modifiers[1].

2.5 Control synthesis

There are a number of methods of transforming a continuous time system to a discrete time system, such as

- Pole/zero mapping
- Bilinear transformation
- Bilinear transformation with frequency prewarping
- Z-transform
- Z-transform with zero-order hold

Request	Argument	Comments
MGETNUMCARDS	int	Return the number of motor interface cards present in the axis controller bus.
MSETNUMJOINTS	int	Specify the number of axes that will be controlled by the servo code. Returns EINVAL if this number is greater than that supplied by the motor interface cards present.
MGETPARAM	struct mc_param	Return the parameter structure for the axis specified by the *which* element of the passed parameter (read/write argument). Will return EINVAL if *which* is greater than the number of axes, or if *bo* is equal to zero.
MSETPARAM	struct mc_param	Set the parameter structure for the axis specified by the *which* element of the structure. Will return EINVAL if *which* is greater than the number of axes.
MSTOP		Stop all axes, set all motor torques to zero, joint control modes to MD_OFF, remove enable status, and activate brakes.
MENABLE		Check that all boards are operating, and allow all D/A's to be written.
MSETVERBOSITY	int	If M_VERBOSE bit is set then the driver prints additional diagnostic information during operation. If M_ERRORPRINT bit is set then information is only printed during error situations.
MGETVERBOSITY	int	Returns the verbosity flag.
MCLEARERROR		Clears the error status for all axes.
MSETINTERVAL	unsigned int	Set the hardware timer interval in *µsec*. Return EINVAL if timer is incapable of meeting the interval requested. If the time interval is greater than the heartbeat timeconstant in the motor interface board PANIC will be asserted by hardware, see 5.3.6
MGETINTERVAL	unsigned int	Get the hardware timer interval in *µsec*
MSETENC	unsigned int	Set the hardware encoder register to given value. Lower 8 bit specify axis, next 16 bits specify value.
MSETLED	int	Control axis indicator LEDs, each bit in the argument controls one LED associated with the axis. Axis is specified by lower 8 bits, LEDs by bits 8...

Request	Argument	Comments
MDIAGMODE	int	Specify the diagnostic operating mode. If bit M_ANDIAG is set then analog loopback is enabled, if bit M_ENDIAG is set then encoder loopback is enabled. EINVAL is returned if the board cannot perform the specified diagnostic.
MSETDAC	int	The lower 8 bits of the argument specify which axis D/A is set to the value specified by the next higher 12 bits.
MDACMODE	int	If argument is non-zero then DAC double buffer mode is enabled for all axes, otherwise single buffered mode is enabled.
MGETSTATS	struct mc_stats	Returns a structure of statistics gathered by the driver about interrupts, such as total, those missed, overrun etc.
MZEROSTATS		Zero the driver's statistics structure.
MCALIB	int	Puts the axis specified by the argument into calibration mode. When a zero index is detected, the user's process is notified by a SIGUSR1, and a calibration 'error' status is indicated.

Table 2.3: Ioctl calls for the mc device

```
/*
 * Example program showing use of MMCS system for 2 axis control in
 * velocity mode with random trajectories.
 *
 * pic 1/89
 */
#include        <stdio.h>
#include        <sys/file.h>
#include        "/usr/sys/sundev/mcdef.h"
#include        <signal.h>

int     fd,
        naxis,
        verbose,
        intval = 2;
int     setp[16];
char    *pname;

#define         VAL(b)          (atoi(&b[1]))

main(ac, av)
int     ac;
char    **av;
{
        struct mc_stats         stats;
        struct mc_state         state[16];
        struct mc_param param;
        int             i;
        void            mmcserr();


        pname = av[0];
        /*
        if (ac == 1) {
usage:                  fprintf(stderr, "Usage: %s □\n", av[0]);
                exit(1);
        }
        */

        while (--ac > 0  && **++av == '-') {
                        register char           *p = *av;

                        while (*++p != '\0')
                                switch (*p) {
                                case 'v':       verbose++; break;
                                case 't':       intval = VAL(p); break;
                                }
        }
```

Figure 2.2: MMCS code example

```
srandom(123456);

if ((fd = open("/dev/mc0", O_RDWR)) < 0) {
        perror("open:");
        exit(3);
}
signal(SIGUSR1, mmcserr);

if (verbose) {
        i = M_VERBOSE;
        ioctl(fd, MSETVERBOSITY, &i);
}

if (ioctl(fd, MGETNUMCARDS, &i) < 0)
        perror("ioctl:");
printf("%d cards in system\n", i);

naxis = i*2;
if (ioctl(fd, MSETNUMJOINTS, &naxis) < 0)
        perror("ioctl:");

intval *= 1000;
if (ioctl(fd, MSETINTERVAL, &intval) < 0)
        perror("ioctl:");

/*
 * zero the encoders
 */
for (i=0; i<naxis; i++) {
        setp[i] = 0;
        ioctl(fd, MSETENC, &i);
/*
 * initialize the parameters
 */
param.which = 0;
ioctl(fd, MGETPARAM, &param);
param.a0 = 50;
param.b0 = -1;
param.mode = MD_VEL | SOFTERR | POSCHK;
param.plo = -6000;
param.phi = 6000;
param.ilimmax = 0;
}
```

Figure 2.3: MMGS code example

```
	for (i=0; i<naxis; i++) {
		param.which = i;
		ioctl(fd, MSETPARAM, &param);
	}

	if (ioctl(fd, MENABLE) < 0) {
		fprintf(stderr, "cant enable\n");
		exit(3);
	}

	for (i=0; i<naxis; i++)
		setp[i] = veloc();
	write(fd, setp, naxis * sizeof(int));

/*
 * The program now waits,  all control is done in a signal
 * handler invoked when an axis exceeds its position limits.
 */
	for (;;)
		sigpause();
}

void
mmcserr()
{
	struct mc_state state[16];
	int	i, code;

	read(fd, state, sizeof(state));
	ioctl(fd, MCLEARERROR);

	for (i=0; i<naxis; i++) {
		if ((code = state[i].errcode) == 0)
			continue;
		if ((code & MERR_POSLIM) && (i>=3) && (i<=4)) {
			printf("%s limit on axis %d\n",
				(code & MERR_PLO) ? "low" : "high",
				i
			);
			/*
			 * choose random velocity in opposite direction
			 * for error axis
			 */
			setp[i] = -veloc() * abs(setp[i]) / setp[i];
		}
		else
			printf("error code 0x%x on axis %d\n", code, i);
	}
	write(fd, setp, naxis * sizeof(int));
}

veloc()
{
```

```
#include	<signal.h>
#include	<sys/file.h>
#include	<sys/mman.h>
#include	<sys/types.h>

	int		fd,			/* file descriptor for the bus device */
			len,			/* length of memory window to map */
			off,			/* base of memory window */
			buserr();
	caddr_t		addr, valloc();

	if (len < getpagesize())	/* round len up to a page size */
		len = getpagesize();

	fd = open("/dev/vme16", O_RDWR);	/* open the bus device */
	if (fd < 0)
		perror("open");

	addr = valloc(len);		/* allocate virtual memory */
	if (addr == NULL)
		perror("valloc");

	/*
	 * map bus memory window into user's virtual memory
	 */
	if (mmap(addr, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, off) < 0)
		perror("mmap");

	signal(SIGBUS, buserr);		/* set up signal handlers */
	signal(SIGSEGV, buserr);

buserr()
{
	printf("BUS ERROR\n");
	longjmp(env, 1);
}
```

Figure 2.5: SunOS code example for direct hardware access

Each of these techniques has a different effect on characteristics of the system such as DC gain, frequency response, overshoot etc. Standard control systems texts[8][6] can provide more details.

One simple approach is the bilinear transform

$$s = \frac{2}{T}\frac{z-1}{z+1}$$

where T is the sampling interval. The Laplace transform expression for a general PID controller is

$$\frac{u}{e} = \frac{Ps + Ds^2 + I}{s}$$

where P, D and I are the proportional, derivative and integral gains respectively. Substituting yields

$$\frac{z^{-2}(IT/2 + 2D/T - P) + z^{-1}(IT - 4D/T) + (P + 2D/T + IT/2)}{-z^{-2} + 1}$$

which is in the form implemented by the controller's compensator. The coefficients should be scaled, so that all are greater than 1, since all arithmetic is performed in 32 bit fixed point.

Chapter 3

mcTool

mcTool is an interactive graphical tool that runs under the SunView window environment. The Sunview Programmer's Guide provides some details on the conventions of this interface.

When invoked mctool opens the motor control device, /dev/mc0, and determines the number of motor interface cards. A panel is then built for every axis present. The topmost panel provides global controls for the device, in particular sample interval. The enable button must be pressed to commence servo operation once parameters for the various axes have been set. The *diagmode* switch controls analog and encoder loop back modes of the motor interface board, see Chapter 5.

mctool reads status information from the device driver five times per second and updates the displayed values for encoder count, A/D and motor current.

Many parameters of the device driver may be altered, by typing in new numeric values or adjusting control sliders. Operating modes such as Coulomb friction compensation, LED indicator, feedback source, or servo mode may be controlled by switches.

The *SetpSource* switch allows the driver setpoint to come from either

- the *Setpoint* slider
- the A/D converter
- a square wave whose amplitude is controlled by the *Amplitude* slider which replaces the *Setpoint* slider in this mode.

The *Mode* switch controls the actual servo operating mode. Its values can be any of

Off this axis not servoed in which case the *Setpoint* slider is not

Pos this axis is in position servo mode and the labels of the *Setpoint* slider reflect this. displayed.

SampleInterval(ms) 10 Enable QUIT

DACmode: Normal DiagMode: No

SetpSource: Slider Mode: Off FBSource: Enc Enc: 0
LED: Off CoulComp: Off ADC: 0
P%: [100] 0 2000 ADC_a: 1 ADC_b: 0 Im: 0
D%: [0] 0 200 Ic+: 1 Ic-: 0
I%: [0] 0 200 Ilim: 1

SetpSource: Slider Mode: Off FBSource: Enc Enc: 0
LED: Off CoulComp: Off ADC: 0
P%: [100] 0 2000 ADC_a: 1 ADC_b: 0 Im: 0
D%: [0] 0 200 Ic+: 1 Ic-: 0
I%: [0] 0 200 Ilim: 1

SetpSource: Slider Mode: Off FBSource: Enc Enc: 0
LED: Off CoulComp: Off ADC: 0
P%: [100] 0 2000 ADC_a: 1 ADC_b: 0 Im: 0
D%: [0] 0 200 Ic+: 1 Ic-: 0
I%: [0] 0 200 Ilim: 1

SetpSource: Slider Mode: Off FBSource: Enc Enc: 0
LED: Off CoulComp: Off ADC: 0
P%: [100] 0 2000 ADC_a: 1 ADC_b: 0 Im: 0
D%: [0] 0 200 Ic+: 1 Ic-: 0
I%: [0] 0 200 Ilim: 1

SetpSource: Slider Mode: Off FBSource: Enc Enc: 0
LED: Off CoulComp: Off ADC: 0
P%: [100] 0 2000 ADC_a: 1 ADC_b: 0 Im: 0
D%: [0] 0 200 Ic+: 1 Ic-: 0
I%: [0] 0 200 Ilim: 1

SetpSource: Slider Mode: Off FBSource: Enc Enc: 0
LED: Off CoulComp: Off ADC: 0
P%: [100] 0 2000 ADC_a: 1 ADC_b: 0 Im: 0
D%: [0] 0 200 Ic+: 1 Ic-: 0
I%: [0] 0 200 Ilim: 1

Vel this axis is in velocity servo mode and the labels of the *Setpoint* slider reflect this.

Torq this axis is in torq servo mode and the labels of the *Setpoint* slider reflect this.

Test1 this axis is in MD_TEST1 mode which outputs a sawtooth waveform of amplitude equal to the present current limit, Ilim.

Diag this axis is not servoed by any adjustment of the *Setpoint* slider causes that value to be output to the D/A. This is useful for testing the D/As or setting a specific output voltage.

The PID gain sliders have units of %, that is the displayed value divided by 100 since the sliders can only have integer values. As the sliders are adjusted the appropriate compensator coefficients are computed and modified in the driver.

Some command line switches for mctool are

-t*interval* Set the initial value of the interval time to the specified number of milliseconds.

-u Dont update the encoder, A/D and motor current state information.

-z Zero all incremental encoder registers.

Since many processes can open the mc device at the same time, mctool can be used as a monitor of the motor state while other processes are controlling it. In this situation mctool should leave all axes in the

Chapter 4

Host adaptor

4.1 Introduction

A fundamental component of the proposed modular motor control system (MMCS) is the interface between the host bus and the axis controller bus. The first such interface contructed is for VME bus host machines. This chapter describes the hardware details necessary for application program development.

4.2 Adaptors in general

Physically an adaptor consists of two cards linked by a ribbon cable. One board plugs into the host bus, and the other plugs into the axis controller bus.

For future host bus adaptors it is likely that the existing axis controller side of the adaptor could be used, necessitating only a new host bus card.

4.2.1 Generic specification for adaptors

The adaptor provides five main functions

1. A mapping of memory accesses on the host bus to PC bus I/O cycles on the axis controller bus.

2. A programmable source of clock pulses, SCLOCK, used to synchronize the latching of state measurements in all motor interface cards.

3. A facility to interrupt the host processor on two conditions, clock pulse SCLOCK and control bus detected panic condition PANIC.

4. A general purpose clock signal (around 8MHz) must be supplied to the PC bus for use by motor interface boards.

5. Safety features such as interfacing to the handheld panic button, and control of mechanism brakes if present.

4.2.2 PCbus signals redefined

Two PC bus signals have been redefined for this application. From a purist's standpoint this is bad design, but is unavoidable unless a separate cable was run linking all motor interface cards, or we adopted a different bus altogether. Some care was taken in choosing which lines to use for these special purposes.

1. SCLOCK signal, used to control latching on all motor interface cards. This signal uses one of the PC bus DMA request lines DRQ1...DRQ3. It is jumper selectable on the adaptor and motor interface cards, and should be set such that all cards use the same line, and that the line does not conflict with other PC bus devices.

2. PANIC signal, used to indicate that one of the motor interface cards has detected an error condition. This line is a *wired or*, that is, any board can pull it high to assert the PANIC condition, which is then available to all other boards.

4.3 In particular: VMEbus adaptor

Physically the adaptor is two cards linked by an 8 foot, 50way ribbon cable. One board plugs into the VME host's backplane, and the other plugs into the control (PC) bus. The VME end is based upon a Logical Design Group VME-5100D VMEbus prototype card with only 10 extra chips. The PC end is based on a bare wirewrap prototype module.

4.3.1 VME memory Map

The adaptor occupies 2kbytes of VME A16 address space, layed out as shown in Figure 4.1. It comprises two regions, one that is mapped to PC bus requests, and another that is adaptor control registers. The current configurable base address of this segment is 0x6000.

The adaptor only responds to 8(O) data, that is byte data at odd addresses. Short (16 bit) requests to even addresses (odd address-1) will transfer the bottom byte of the short to the adaptor, the top byte is ignored on write, and 0xff on read.

4.3.2 PC bus access

When A10 is 0, all accesses to the adaptor are passed straight through to the PC slave bus. These accesses become I/O space requests on the PC bus and occupy the odd addresses 0x001 through 0x3FF on the VME bus.

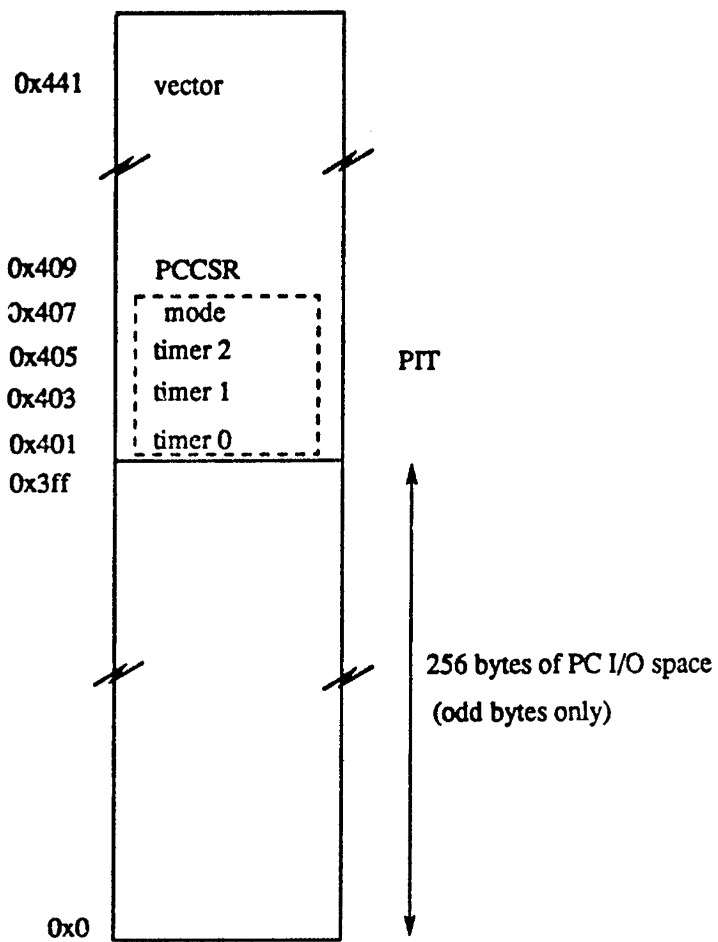


Figure 4.1: VME Host adaptor memory map (byte addresses shown)

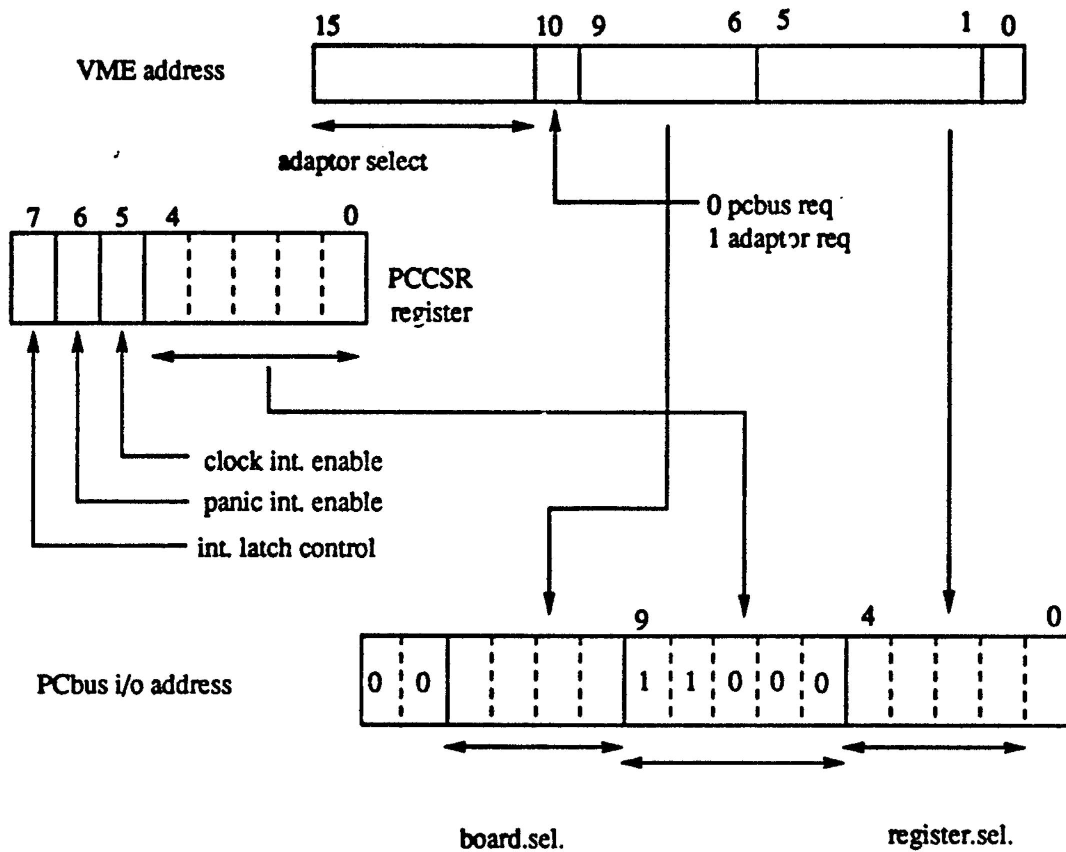


Figure 4.2: PC bus I/O space address formation

The PC bus allows for 16 bit I/O space addresses. IBM however define only unique values or ranges for the least significant 9 bits of the I/O address. The address range for *prototype card*, which we use for motor interface cards, is 0x300 to 0x31f. However this is not enough address bits to support upto 16 interface cards, each with upto 32 bytes of registers. We thus use address bits 10...13 to peform the board select function, and bits 0 through 4 for register select.

The address mapping between VME and PC I/O space is shown diagramaticaly in Figure 4.2. VME address bits A01...A05 are mapped to PC address bits A00...A04. VME address bits A06...A09 are mapped to PC address bits A10...A13. PC address bits A05...A09 are set via the control register. PC address bits A14 and A15 are fixed at 0.

For the motor interface cards, VME address bits A01...A05 will specify a device register byte on an interface card. VME address bits A06...A09 specify which controller card is being selected on the PC bus. Thus, a bank of up to sixteen controller cards is mapped as a contiguous space of addresses on the VME bus.

4.3.3 Adaptor Control Registers

When VME addresss bit A10 is 1, an adaptor register is selected. These registers include the interrupt vector register, the PC control/status register (PCCSR), and the 8254 Programmable Interval Timer (PIT).

When address bit A06 is a 1 on the VME bus, the vector register is selected. This puts the vector register address at 0x0441. A single byte may be written to this register to supply a vector number used during interrupt service on the VME bus. This register may not be read.

When address bit A06 is 0, one of the other registers is selected. When VME addresses A05...A03 are (respectively in binary) 000, the PIT is selected. In this case, A02 and A01 specify the register on the PIT. Thus, the PIT registers are; timer 0, 0x401; timer 1, 0x403; timer 2, 0x405; control register, 0x407.

When A06 is low and A05...A03 are 001 (binary), the PCCSR is selected. Thus, its address is 0x409. The bitfields of this register are shown in Table 4.3.3.

4.3.4 The Servo Clock

The servo clock, SCLOCK, is the output of timer 1 on the 8254. It is clocked by timer 0, which is in turn clocked by the 8 MHz system clock which is derived from the VME bus. The system clock may be configured down to 4, 2, or 1 MHz, and is also passed to the PC bus. Timers 0...2 are permanantly gated on.

Since timers 1 and 2 are cascaded, allowing a count of 32 bits to be made at 8 MHz, a very wide range of servo loop times is achievable. SCLOCK clocks an edge triggered latch which can cause a VME bus interrupt request. The servo

clock is also passed to the PC bus on the DRQ 3 line, for use by the motor interface cards.

The SCLOCK signal is also fed to timer 2 which may be used to by driver software to check that interrupts are not being missed, that is at each interrupt timer 2 is only 1 different from its value at the last interrupt.

4.3.5 Panic signal

The panic signal from the PC bus is logically combined with the status of the handheld emergency stop button, and used to generate the system PANIC signal. The rising edge of this signal clocks an edge triggered latch which can cause a VME bus interrupt request.

4.3.6 Interrupts

Interrupts are controlled by the PC control register (PCCSR). When bit 7 of the PCCSR is low, the two interrupt latches are forced to clear. They will remain cleared until bit 7 is set high. In this state, a low-to-high transition of SCLOCK will set the servo clock interrupt latch, and a low-to-high transition on PANIC sets the panic interrupt latch. They will remain high until cleared by clearing bit 7. Both latches are cleared together, so an interrupt service routine must poll both interrupts and service the appropriate ones before clearing bit 7.

The status of the latches is read back from bits 5 and 6. To enable the latches to interrupt the VME bus, a 1 must be written into bit 5 or 6. If interrupts are used, don't forget to set the vector latch, see section 4.3.3.

Currently the VME-5100D is set to interrupt at VME level 5, which is at the same level as the clock and Unix scheduler for a Sun 3[2]. To alter the level both jumpers J5 and J7 on the VME-5100D must be altered. Be sure to remove the interrupt acknowledge (IACK) jumper from the backplane for the slot containing the VME-5100D. A "spurious level ? interrupt" message on the Sun console indicates that the IACK daisy chain is incorrectly jumpered or that J5 and J7 on the VME-5100D are inconsistent.

4.3.7 LED indicators

The adaptor card has two LED indicators. The green LED is the SCLOCK signal, while the red LED is the status of the user interrupt request to the VME-5100D interrupter logic.

4.3.8 Miscellaneous Notes

VME bus resets is passed to the PC bus. The reset signal also resets the PCCSR to 0x00 (interrupts off).

The PC side of the adaptor may be powered down while the system is running without affecting the VME side. To disconnect the VME side from a SUN 3, the SUN must be shut down, powered off, and the VME-5100D board and 3U-2U adapter must be removed from the SUN backplane, and the IACK jumper installed.

Bit	Write operation	Read operation
4-0	the PC bus address bits A05 through A09	what was written to those bits
5	when set, enables interrupts from SCLOCK	Bit 5 is 1 if an SCLOCK interrupt has been latched (but not necessarily passed to the VME bus)
6	when set, enables panic interrupts from the PC bus	Bit 6 is a 1 if a panic interrupt has been latched (also, not necessarily passed to the VME bus)
7	controls the clearing of the two interrupt latches. When it is 0, the latches get cleared. When it is a 1, the latches may get set	returns what was written to it.

Table 4.1: PCCSR bitfields

Chapter 5

The Mark I motor interface card

The motor interface is the electronics that connects the motor to the axis controller bus. It provides an analog drive signal to the motor, and measures shaft angle via an incremental encoders, as well as application specific quantities via a general purpose analog input.

This chapter provides hardware specifications and details to those needing to program the device hardware directly.

5.1 Servo board specification

5.2 Design aims

One of the most important design features is to allow for the non-deterministic timing of the host computer, targetted in the first instance to be a SUN workstation running Unix. The use of an additional CPU running a real-time operating system would provide better performance, albeit at greater cost.

At the short sample times required (less than 10ms) a Unix process cannot be reliably scheduled to respond. However a device driver working at high hardware priority will respond within 100μs, but will not be deterministic due to interrupts being locked out in critical regions of the Unix kernel. Thus, it was not desirable for the data sampling to be controlled directly by the host computer.

Instead, sampling is controlled by an axis controller bus signal SCLOCK, and at each sample all A/D converters and encoder registers are latched, and the host notified by interrupt that new data has arrived. The host device driver can then read the robot state and compute new setpoints to be written to the

D/A converters. State is thus sampled at a fixed intervals with zero timing error. Additionally the D/A converter updates can occur in a double buffered mode in which the new values are not output until the next sample time.

5.3 Description

Each board provides all the functionality required to control two servo axes, referred to here as joint 1 and joint 2. Each controller consists of an A/D, a D/A, and an incremental encoder interface (IEI).

- AD574 12 bit analog to digital converter. This may be used to input application specific signals such as joint angle, torque, current etc.
- AD667 12 bit digital to analog converter for specifying motor current or torque. The device has the ability to double buffer the digital commands.
- HCT2000 incremental encoder interface (IEI) to decode quadrature signals from incremental encoders. This device has 16 bit resolution, and can be read or written at anytime. It also has the facility for the count to be latched by an external signal, which we use for calibration purposes.
- i8255 to provide control of all board operating modes and to provide status information. The 8255 device provides 24 programmable input or output lines. The usage in this interface is summarized in Table 5.1.

The D/As contain two cascaded 12 bit latches. The D/A must be accessed twice to load a new value into the first latch, once for the low byte, and once for the high nibble. D/A data is 12 bit offset binary. The loading of the second latch, and hence the D/A output, is controlled by port B bit 3 of the 8255 device to occur either when the high nibble is written, or on the edge of the SCLOCK signal.

The A/Ds convert all 12 bits on the edge of the SCLOCK. The 12 bit offset binary values must be read in two byte accesses. The status of the conversion can be read via port C bit 4 of the 8255, which when set, indicates both converters are done.

Data sheets on these devices are provided in an Appendix.

5.3.1 Memory map

The memory map of the motor interface card given in Figure 5.1 shows the addresses as seen from the PC bus. All of the addresses given are relative to the board's base address, and are five bits only.

The board resides at address 0x300 to 0x31F in 9 bit (IBM standard) PC I/O space. However this is not enough address bits to support upto 16 interface cards, each with upto 32 bytes of registers. We thus use address bits 10...13 to

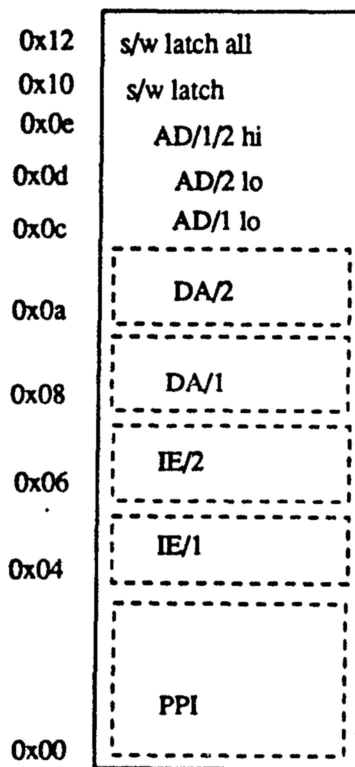


Figure 5.1: Servo board memory map

peform the board select function, and bits 0 through 3 for register select. Thus, 0x0300 is the base of board 0, and 0x3F00 is the base of board fifteen. The board address is set by a 4 bit DIPswitch on the card, when a switch is on, or closed, it corresponds to a 0 bit.

The 8255 programmable peripheral interface (PPI) is at addresses 0x00 through 0x03.

The HCT2000 for joint 1 is at addresses 0x04 through 0x05, while that for joint 2 is at addresses 0x06 through address 0x07. The most significant byte occupies the lower address. The HCT2000 may be written to initialize the 16 bit up/down counter. It may be read at anytime to give the instantaneous counter value, however the first read after a latch command will yield the counter value at the time of the latch. The operating mode of the HCT2000 is set via port A of the 8255, and would normally be mode 5, although a number of other modes such as frequency and interval measurement may be useful. The HCT2000's two byte registers should always be accessed in the same order after reset, see the applications note for more details.

The D/As are at addresses 0x08 through 0x0B. These locations are write-only. Location 0x08 is the low byte of D/A 1, and 0x09 is the high nibble of D/A 1. Similarly, 0x0A and 0x0B write to D/A 2.

The A/Ds are at addresses 0x0C through 0x0F. These locations are read-only. Location 0x0C returns the low byte of A/D 1. Location 0x0D returns the low byte of A/D 2. Location 0x0E returns the high nibbles of both A/Ds. The high nibble of A/D 1 will reside on bits 0 through 3 of this byte, and A/D 2 will use bits 4 through 7.

5.3.2 The latch signal

The latch signal is used to start the conversion of the A/Ds, enable the output latch on the D/As, and cause the IEs to latch their counts.

This latch signal may come from either the rising edge of SCLOCK (a back-plane signal from the host adaptor) or be generated via addressing (see Figure 5.1). To enable latch on the rising edge of SCLOCK port B bit 2 of the 8255 must be asserted.

Any access to location 0x10 will cause a latch signal to the A/Ds and IEIs, just as the SCLOCK signal does. Any access to 0x12 will do the same, except that it will cause all boards in the controller to simultaneously latch. Jumper E021...E023 (see Section 5.4.3) control whether or not the D/A responds to the software latch signals.

5.3.3 D/A double buffering

The D/A converters are capable of working in normal or double buffered mode. In normal mode, the D/A analog output reflects exactly what is written to it. In double buffered mode, the analog output changes to the last written value

Bit	Comments
	Port A:(OUTPUT)
0-2	Control the operating mode for both IEIs (HCT2000 chips).
3	Diagnostic LED 1.
4	Diagnostic LED 2.
5	Artificial encoder A signal for IEIs.
6	Artificial encoder B signal for IEIs.
7	Artificial encoder I signal for IEIs.
	Port B:(OUTPUT)
0	When 0, the artificial encoder signals are passed to the HCT2000s. When 1, encoder signals from the joints are passed through.
1	When 0, the output of the D/As is looped back to the A/Ds. When 1, the output of the D/As is switched to the joints.
2	This bit must be a 1 for the external servo clock on the PC bus to affect this board.
3	This bit controls the output mode of the D/As. When 0, writing to the high address of a D/A causes its output to be updated. When 1, the internal servo clock, conditioned by Bit 2, causes both D/A outputs to be updated.
4	When 0, the index latches (see Port C, bits 1 and 3) are cleared. When 1, the index latches get set on the next index pulse. Also, when this bit is a 1, index pulses cause the IEIs to latch.
5-7	Unassigned
	Port C:(INPUT)
0	This returns the status of the heartbeat for joint 1. A 1 indicates that all is well. The heartbeat gets retriggered when the low byte of the D/A gets written to.
1	This is a 1 if the index for joint 1 has been latched. The index gets latched when the index latch enable is set to 1 (Port B, bit 4) and a high level is seen on the index input.
2	The same as Bit 0, but for joint 2.
3	The same as Bit 1, but for joint 2.
4	This signal is high when the A/Ds are not converting. A low to high transition on this signal means that good data is in the data latches for the A/Ds.

Table 5.1: PPI bits

synchronously with the latch signal. Thus guarantees a fixed one sample time delay between reading state and providing a new setpoint value.

5.3.4 Calibration

Calibration of incremental encoders is important for control of many manipulators. Frequently the procedure involves driving the axis so that the encoder's zero index is detected, and measuring the shaft angle with some low resolution absolute transducer (such as a potentiometer), and then computing the correct value for the encoder register. In RFMS the axes were driven, the zero index status polled, and the motor stopped when the index is detected. It was found necessary to drive the motors very slowly else the index would be missed.

The new interface provides a calibration mode, in which the motor can be driven at any speed, and when the index pulse is detected, the instantaneous encoder value is latched and the host notified. To enable this mode port B bit 4 of the 8255 should be asserted. Port C bits 1 and 3 indicate that the IEIs for joints 1 and 2 respectively have been latched. The IEIs can then be read to determine the encoder count at which the index pulse occured, another encoder read to obtain the current encoder counts, and determination of motor potentiometer voltages provides all the information needed to determine the absolute motor angle.

Note that subsequent index pulses are not locked out, and will also latch the encoder count. Detection of an index pulse does not stop the motor, this must still be done by host software upon detection of the index latched status.

5.3.5 Diagnostics

The board has a number of diagnostic test facilities built in. Firstly the input of the A/D converters can be "looped back" to the D/A outputs, which allows testing of the A/D, D/A devices and the analog signal paths. The loopback is done via a relay, so some short time should be allowed for the relay contacts to close. This mode is enabled by setting 8255 port B bit 1.

Secondly, synthetic encoder signals A, B and I can be fed into the IEIs to test their operation. The synthetic signals are the same for both axes, and come from 8255 port A bits 5...7. This mode is enabled by clearing 8255 port B bit 0.

5.3.6 Panic signal

One signal on the backplane is the active high PANIC line, that may be asserted by any board that detects a failure. The only failure detected by this interface card is a failure to regularly update the D/A converters with new setpoint values. The D/A write signal drives a oneshot which has a 50ms timeconstant. The outputs of the oneshot (one per axis) are called heartbeats (since they indicates

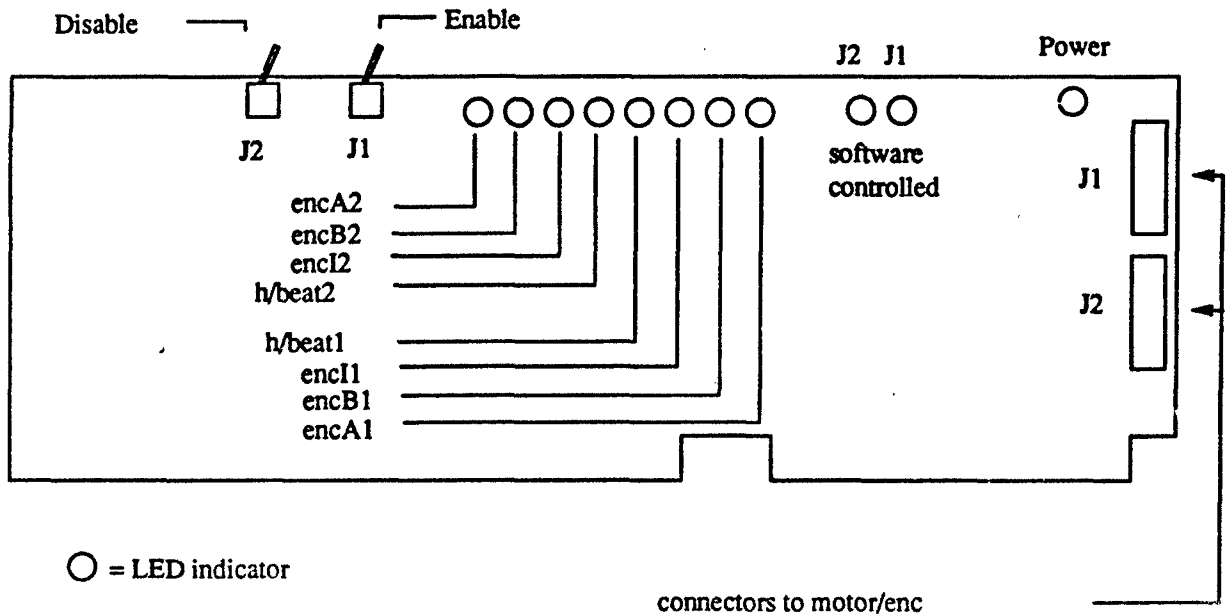


Figure 5.2: Motor interface board layout

something is going on), and can be checked by onboard LED indicators and by software via port C bits 0 and 2 of the 8255.

Note that the "panic" condition will not go away until all the D/A converters have been written to initially. Note also that for SCLOCK intervals longer than the heartbeat timeconstant PANIC will be asserted regularly, it was not considered that servoing at less than 20Hz would be useful.

5.4 Board details

Details of the motor interface board layout are given in Figure 5.2.

5.4.1 Switches

There are two switches on the card, one per axis. Their purpose is to take an axis "out" of the controller without having to remove a card, or half a card. In the enable position everything works as described. When disabled, the analog current drive for the axis is switched from the D/A output to ground, and the heartbeat signal is ignored, thus *panic* won't be caused when that axis's D/A is not updated.

5.4.2 LED indicators

LED	Color	Comment
encA1	Green	Encoder A signal for joint 1.
encB1	Green	Encoder B signal for joint 1.
encI1	Green	Encoder index signal for joint 1.
encA2	Green	Encoder A signal for joint 1.
encB2	Green	Encoder B signal for joint 1.
encI2	Green	Encoder index signal for joint 1.
h/beat1	Green	Heartbeat signal for joint 1.
h/beat2	Green	Heartbeat signal for joint 2.
gp1	Red	General purpose (software controllable) indicator for joint 1.
gp2	Red	General purpose (software controllable) indicator for joint 2.
power	Red	Board 5V supply is OK.

Table 5.2: Motor interface LED indicators

5.4.3 Configuration

There are four groups of jumpers, as described in by Table ??

From	To	Comment
E001	E002	A/D1 10V range
	E003	A/D1 20V range
E004	E005	A/D2 10V range
	E006	A/D2 20V range
E007	E008	If closed D/A1 10V span
E009	E010	If closed D/A2 10V span
E023	E021	D/A does not respond to software latch
	E022	D/A responds to software latch
E020	E017	Sample clock is DRQ1
	E018	Sample clock is DRQ2
	E0189	Sample clock is DRQ3
E011	E012	Panic is IRQ3
	E013	Panic is IRQ4
	E014	Panic is IRQ5
	E015	Panic is IRQ6
	E016	Panic is IRQ7

Table 5.3: Mark 1 configuration jumpers

They are used to set voltage scaling for the A/D and D/A converters as well as to select which PC bus lines are used for the SCLOCK and PANIC signals.