1992

A modular approach to designing software for real-time actuator control for destructive testing

Poulomi Bharatkumar Damany
Lehigh University

AUTHOR:

Damany,

Poulomi Bharatkumar

TITLE:

A Modular Approach to Designing Software for Real-Time Actuator Conrtol For Destructive Testing

DATE: May 31, 1992

A MODULAR APPROACH TO DESIGNING SOFTWARE FOR REAL-TIME ACTUATOR CONTROL FOR DESTRUCTIVE TESTING

BY

POULOMI BHARATKUMAR DAMANY

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

June 1992

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

April 29, 1992

Date

Advisor

Chairman of Department

ACKNOWLEDGEMENTS

The author would like to express her gratitude to her advisor, Professor Stephen P. Pessiki, for taking a chance on a female, non-Civil Engineer teetering on the brink of financial insolvency. This work would not have been possible without his constant encouragement and guidance throughout this past year. I would be remiss if I did not thank Mark Kaczinski, research engineer at ATLSS, for all his help, insights and patience, CE graduate students John Abruzzo, Alain Nussbaumer and Guy Decorges for seeing that I did not kill myself or anyone else while on the test floor and Dave Scott for rescuing me everytime my program threatened to self-destruct.

Finally, my special thanks to the ATLSS - Engineering Research Center for funding me for my graduate studies and for introducing me to the world of hard hats, steel beams and lean, mean pounding machines - I hope the work I did when not trying to single-handedly destroy all the computers on the floor, will be of some use in the future.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The laboratories for the Center for Advanced Technology for Large Structural Systems (ATLSS) conduct a wide range of experiments (fatigue, quasi-static and dynamic tests) on structures, which have different requirements for actuator control, data acquisition, and data display. Project D2- Computer Controlled and Integrated Experimentation involves developmental work to expand the experimental testing capabilities of ATLSS.

In a computer-controlled test, all aspects of actuator control and data acquisition are controlled by commands issued from a single test computer, with information about the current system state being displayed to the operator. The purpose of the actuator control software is to generate an interface between the Vickers X8700 Digital Closed Loop (DCL) servo controller and the end-user so that the user can easily communicate with the servo controller and send force or motion control parameters, while having information about the actuator on the monitor.

At present, ATLSS is using software written by Test Systems & Simulation, Inc. (TS&S) for actuator control. The software works well in isolation for fatigue testing but imposes limitations on other tests. This is because the software does not communicate easily with other test hardware/software. Actuator control is the primary task of the TS&S software while the actual test is viewed as a sub-task.

The efforts of this project have been towards the development of a software package for actuator control, which can be called within the user-specific test control program as a sub-task. This allows the user to modify the test control program while leaving the basic actuator communication shell intact. In addition to this actuator control software, one user-specific test control program for constant amplitude fatigue testing was also developed.

The software is written in Borland Turbo C to run under MS-DOS version 3.2 or 3.3 on an IBM PC AT with a clock speed of at least 8 MHz. It requires the following minimum hardware configuration:

640K Memory	20 MB Hard Drive
EGA Color Monitor	EGA Graphics Adaptor Card
Math Co-processor	

The memory address map used by the software:

Interrupt vector table - 00000H   BIOS data area - 00400H

Video buffer - B8000H   Burr-Brown I/O-D0000H, CD000H, C0000H

The I/O port addresses used:

Serial I/O to controllers - 3F8H, 2F8H, 3E8H, 2E8H (any one of 4 COM ports).

CHAPTER 1

INTRODUCTION

1.1 Introduction

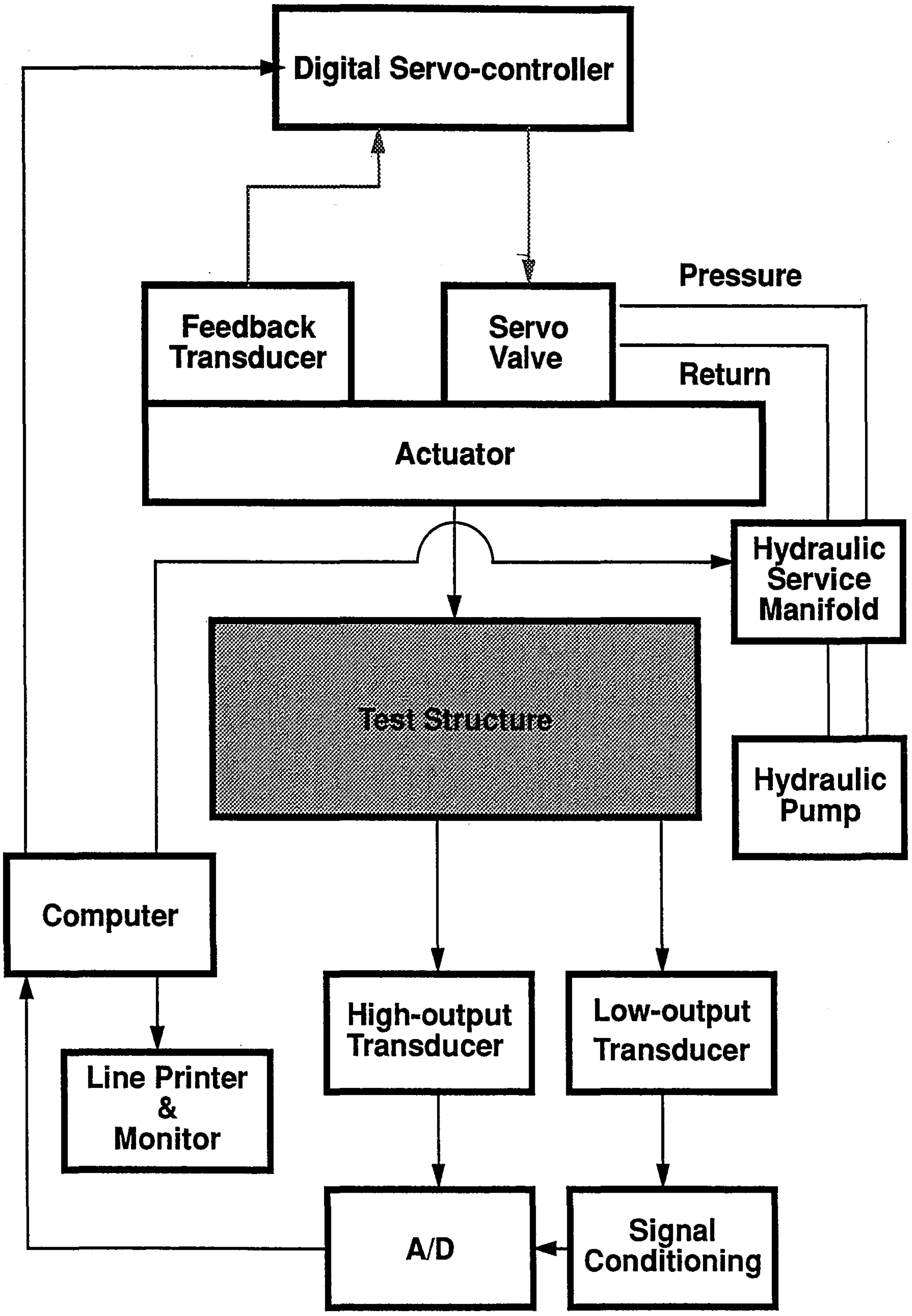
To obtain true "computer-controlled testing" it is necessary to integrate the separate activities of actuator control and data acquisition, as well as other aspects of a test such as data manipulation and display. Figure 1.1 is a schematic diagram of such an integrated test system. In this system, all aspects of actuator control and data acquisition are controlled by the commands issued by the test computer.

It is not essential to have just one computer as shown in Figure 1.1. What is essential is that the information about actuator control and data acquisition are available in some common environment so that their activities can be coordinated, and so that information about the current state of a test are available for manipulation and for presentation to the test operator. This is a key requirement.

At present, ATLSS is using software written by Test Systems & Simulation, Inc. (TS&S) to achieve control of the hydraulic actuators. The TS&S software works well for actuator control in constant amplitude fatigue testing. In theory, the desired command signal, consisting of an amplitude (force or displacement) and frequency, is issued by the computer to the actuator controller, and the computer is then available to perform other tasks. Application of the TS&S software to variable amplitude fatigue testing is somewhat more complicated because of the need to constantly update the command signal. Nonetheless, the software has been used to control variable amplitude fatigue tests.

Use of the TS&S software has imposed severe limitations on tests other than fatigue tests. This is because the software does not easily communicate with other test hardware/software (such as data acquisition hardware/software). Rather than have actuator control as just one task in a computer-controlled test, use of the TS&S software forces all other tasks (such as data acquisition) to be sub-tasks of the activity of actuator control.

The forgoing is not intended to suggest that combined hardware/software test packages are in all aspects undesirable. For example, as noted above, the TS&S software works well for constant amplitude fatigue tests, particularly if data acquisition is not necessary and the test can be monitored with a separate stand-alone data acquisition unit. In addition, there may be many tests performed in



Schematic Diagram Of A Computer Controlled Test System - Figure 1.1

which integration of actuator control and data acquisition is not needed. It can be argued, however, that the sole reliance on packaged test systems (hardware and software) which teat only one component of a test (actuator control or data acquisition) impedes efforts to obtain true integration of a test. Communication between stand-alone software/hardware packages has proved to be an extremely difficult task.

1.2 Objective

The work described in this report was performed as part of the ATLSS project D2 - Computer Controlled and Integrated Experimentation. The objective of the project was the development of a modular software package for actuator control which could be called within the user-specific test control program as a sub-task. This actuator control software would allow the user to modify the test control program while leaving the basic actuator communication shell intact. The modularity of the software would allow inclusion of needed modules only thereby significantly reducing memory requirements and improving overall speed of execution.

1.3 Outline Of The Report

This report combines two manuals- one intended as a users' guide and the other as a programmers' guide. Chapter 2 gives a brief description of a servo feedback controller and the issues involved in using it for position or force control. Chapters 3 and 4 describe the basic structure of the software package, along with a description of all the user display screens and how to operate in them. Chapters 5 through 9 are a programmers' guide to the software and are intended as documentation and reference for future revisors of the code. The following is a brief listing of the contents of each chapter:

Chapter 2 **DESCRIPTION OF A SERVO LOOP CONTROLLER:** This chapter describes a Digital Closed Loop Controller and its various elements.

Chapter 3 **GENERAL DESCRIPTION OF PROGRAM MODULES:** This chapter is intended to give an overview of the software, showing the relationships between the program modules.The modules are described in order of appearance in the system menus.

Chapter 4 **PARALLEL INPUT-OUTPUT**: This chapter describes the Parallel I/O option that allows the user to control external hardware such as panel switches, service manifolds, etc. through computer control of parallel I/O. This feature is most useful in case of an error condition in the control loop that requires the automatic shut down of hydraulic pressure, display warning lights, etc.

Chapter 5 **PROGRAMMING GUIDE**: The next four chapters describe in detail, each module and the lower level routines it calls. The description format is top-down, starting with the main module and going down to the component routines and the functions they call.

Chapter 6 **INITIALIZATION**: This chapter describes the routines executed during start-up. Two separate entities need to be initialized - the serial communication port (8250 UART chip-Appendix A) and the parallel input/output board (Burr-Brown PIO board- Appendix B).

Chapter 7 **SETUP:** This chapter describes the routines which interface between the controller parameters and the user display (screens 1 & 2). They read and write the controller parameters, convert between engineering units and controller counts and display the information onto the screens for the user's benefit.

Chapter 8 **RUN:** In addition to the general purpose software for actuator control, a user-specific program for constant amplitude fatigue testing was also written. This user-specific program is described in this chapter.

Chapter 9 **GRAPHICS:** This chapter describes the routines that have to do with drawing menus, borders and boxes, changing the background and foreground colors, writing to specific portions of the screens and all other screen entity manipulations.

Appendix A **INTEL 8051 UART HARDWARE DESCRIPTION:** This is a pullout from the INTEL Peripheral Components manual and presents a complete description of the hardware and programming features of a NS16450 type Asynchronous Serial Communications controller through which the computer can "talk" to the servo controller on the valve.

Appendix B **PROGRAMMING THE BURR-BROWN PIO BOARD:** This deals with the programming of the PC20001C parallel I/O board in more detail and also expands on the memory addressing scheme of the board.

Appendix C **CONTROLLER PARAMETERS:** This is a listing of the controller parameters along with some information about their engineering units-counts conversion equations and scaling factors, the registers assigned to them and the amount of memory (in bytes) they occupy.

CHAPTER 2

DESCRIPTION OF A SERVO LOOP CONTROLLER

2.1 Introduction

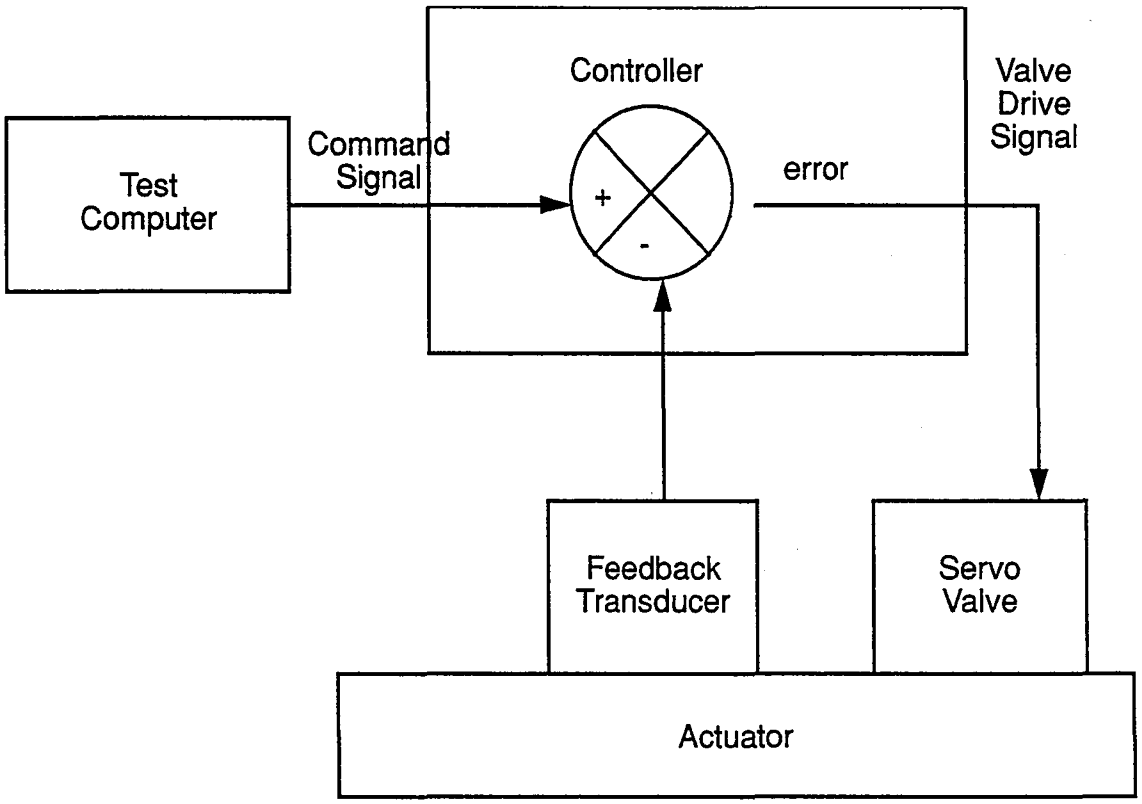
This chapter provides a general description of the operation of closed-loop servo hydraulic control systems such as those used for structural testing in the ATLSS laboratories. This chapter also presents some of the details of the Vickers Xpert DCL servo system, which is the particular system in wide use in the ATLSS laboratories.

2.2 General Description of a Servo Loop:

A servo system is a feedback system in which the output or some function of the output is fed back for comparison with the input and their difference (error) is used to control a source of power[1]. In the ATLSS laboratories, such servo systems are used to control the displacements or forces applied by the hydraulic actuators used in structural tests.

Figure 2.1 is a schematic diagram of a generic control system for the operation of a hydraulic actuator. The main components of this system include the test computer, controller, actuator and feedback transducer. Briefly, the operation of the system is as follows. The test computer issues a command to the controller. The command is the desired force or displacement which is to be implemented by the actuator. The controller also receives from the feedback transducer. Also input to the controller is a feedback signal which is an indication of the current force or displacement applied to the structure by the actuator. The controller computes an error signal which is the difference between the command and feedback signals. The error signal gives rise to a drive signal to the servovalve. The servovalve is a mechanical device used to adjust the flow of oil in to the actuator. The servovalve adjusts this flow of oil in response to the drive signal, porting oil to the actuator in a manner that causes the error signal to reduce to zero, thus implementing the commanded force or displacement.

One end of the hydraulic actuator is normally attached to the test structure, and the other end reacts against the reaction wall, strong floor, or other similar reaction point. The hydraulic actuators are usually operated in one of two modes: load control or displacement control. In the case of load control, the command signal is the desired force, and the feedback signal is the force currently applied by the actuator. The feedback transducer in this case is a load cell. In displacement control, the command signal is the desired displacement, and the feedback signal is the current displacement of the actuator, i.e. how far the piston is currently extended. The feedback transducer in this case is a displacement transducer. This displacement transducer is often mounted internally in the body of the actuator, but may also be external to the actuator. Other type of control are also possible, such as acceleration control or strain control. Finally, the actuators are operated in a closed loop manner, i.e. the valve command is sent automatically without intervention by the test operator.



Schematic DIagram of a Control System - Figure 2.1

2.3 The Vickers DCL Servo System

The rest of this chapter is condensed from the Vickers Xpert DCL User's Manual [2] and is included here for completeness. The Vickers Xpert DCL system is a force and motion control system. It consists of a display board, a microboard containing the software for bi-directional serial communication between the computer and the Controller, a digital closed loop servo valve and a feedback transducer. The servo valve consists of a polarized electrical servo motor and two stages of hydraulic power applications - a pilot flapper/nozzle arrangement and a power sliding spool arrangement (See Figure 2.2). The servo motor converts low level electrical input signals from an amplifier to a mechanical force and motion. When the command signal is applied, the motor is activated and the servo valve provides flow to the actuator proportional to the electrical current applied. The direction of the flow is dependant on the polarity of the DC command signal. Flow from the valve controls actuator movement. Actuator movement is measured by the feedback transducer. The feedback signal from the transducer is compared with the command signal, the difference amplified and a new electrical current sent to the servo valve. The actuator will continue to move until the actuator position approaches that called for by the command.

Controller:

The Vickers controller contains the power display drive board to control the valve driver and the display lights and address switches, the microboard holding the on-board software and the feedback transducer conditioner which converts the signal from the feedback sensor to a digital signal for computer access.

Servo Motor:

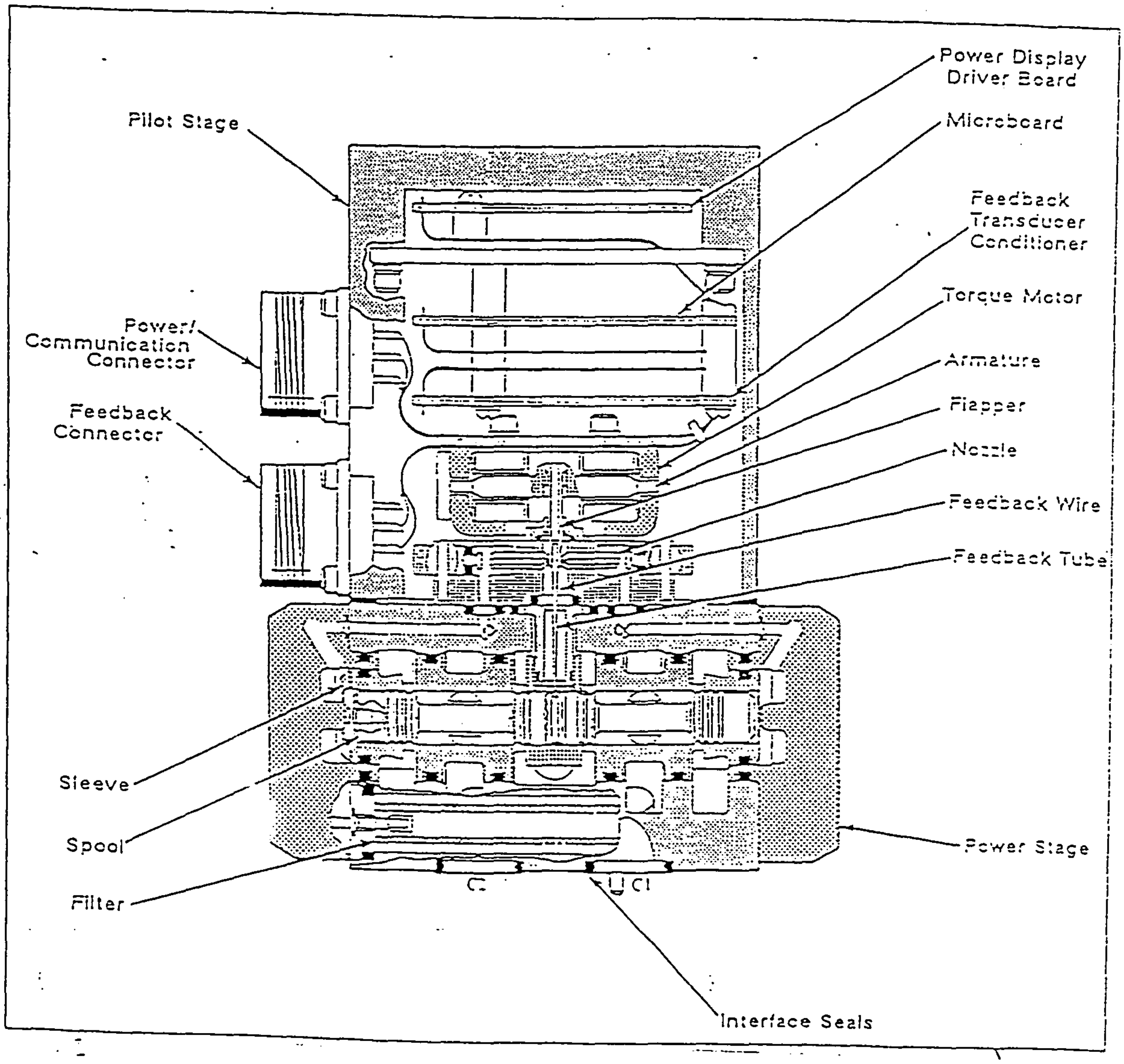
The Vickers motor consists of two coils surrounding an armature. Lead wires from the coils are connected to the power display drive board. A flexible tube supports the armature and also acts as a fluid seal between the hydraulic and electronic areas of the valve.

Pilot Stage:

Attached at the center of the servo motor armature is a flapper with a nozzle on each side, that extends down inside the flexible tube. Flapper movement between the nozzles creates pressure variations, which act on the ends of the power stage spool. Attached to the flapper is a feedback spring which engages into a hole at the center of the spool. When the spool moves, the feedback spring senses the spool movement and a force proportional to this movement is fed back to the flapper.

Power Stage:

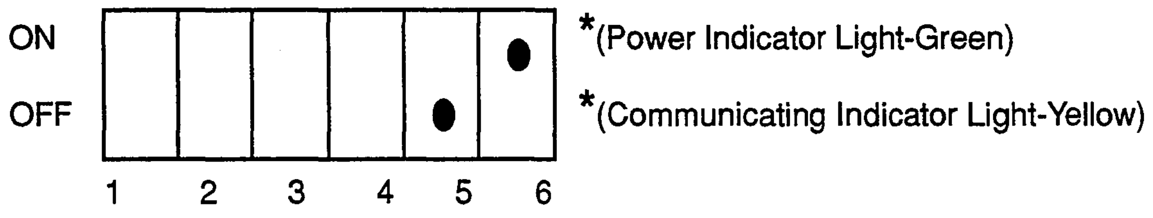
Actual flow in the valve is controlled by a four-way spool that slides within a sleeve. The porting system is arranged so that the spool movement in one direction opens fluid supply to control port #1 and opens port #2 to return. Movement in the opposite direction opens fluid supply to port #2 and opens port #1 to return.



Servo Valve Construction - Figure 1.3

Address Switches:

A given application can contain up to 16 of the above described DCL Controllers on one serial port. To distinguish one from the other, each is assigned an address. This is done by setting the address switches (1-4) on each individual valve . Another setting to be made is that of the baud rate which is the speed with which communication takes place between the valve and the computer. Switches 5 and 6 on each valve have to be set to the desired rate. For example, for a baud rate of 38400 the setting is as shown below in Figure 2.3:


ON
OFF
1
2
3
4
5
6
*(Power Indicator Light-Green)
*(Communicating Indicator Light-Yellow)

Controller Switches-Figure 2.3

2.4 The Vickers Servo Valve Operation

1. An electrical input current is applied to the servo motor.

2. The coils in the motor magnetize the armature.

3. The armature moves clockwise or counterclockwise, depending on the polarity of the current applied.

4 This causes the flapper to move between the nozzles creating a change in fluid pressure at the ends of the power stage spool.

5. As a result of the pressure change, the spool moves away from the high pressure end of the valve.

6. Depending on the direction of spool movement, the pressurized fluid will then flow from either port.

7. As the spool moves, it causes the feedback spring to deflect. This creates a force on the pilot stage flapper opposing the motion caused by the armature.

8. When the force from the feedback spring equals the magnetic force of the armature, the flapper returns to center between the nozzles.

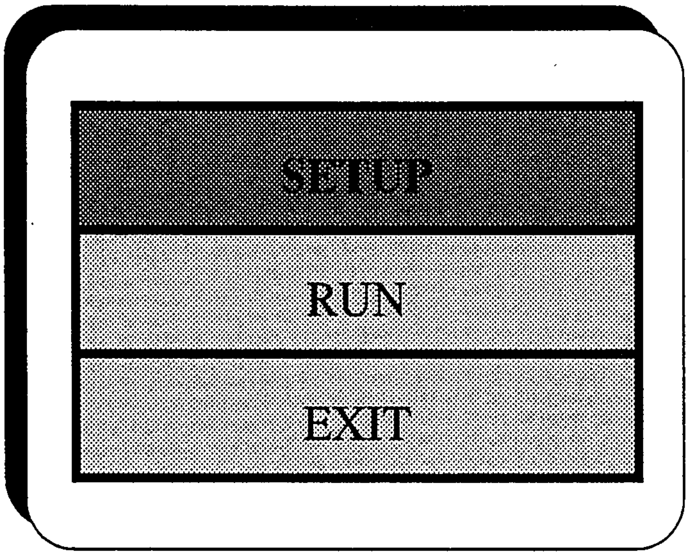
9. The power stage spool stops moving and remains in that position until the input current changes to a new level.

CHAPTER 3

GENERAL DESCRIPTION OF PROGRAM MODULES

3.1 Introduction

This chapter is intended to give an overview of the software, showing the relationships between the program modules. The modules are described in order of appearance in the system menus.



Main Menu- Figure 3.1

3.2 Program Structure

The software is a menu-driven collection of modules with each module performing one aspect of the total controller communication task. The three main modules are *Initialization, Setup and Run.* The Initialization and Setup are constant, general routines which can be used without any changes to establish communication with any controller. The Initialization module is executed as soon as the program starts up and is not a user choice from the main menu (see Figure 3.1). The Run program is intended to be a user test-control program which can use the other two routines for controller communication while running the test at a higher level. The Run program described here is a sample constant-amplitude fatigue test-control program.

The Initialization routine does exactly that - it establishes the communication channel, resets the parallel I/O logic board (for more details see Appendix B) and reads in the current values of the controller parameters. The Setup module is similar to the TS&S Multi-scan program in that it is used to facilitate the storage of controller parameters and to manipulate the controller during setup. Data about the controller can also be saved from this module. The Run module is a sample fatigue test control program with 16-channel configuring capability. The controllers can be run as a group from this routine while still maintaining individual starting/stopping capabilities.

3.3 Installation

To install the package on the hard drive of the host computer, first create a new directory of any name. Copy the following files to the directory - note that all the files must be in the same directory - do not attempt to divide the program files among sub-directories.

The header files are:

1. Async.h	2. Run.h	3. Params.h

The data files are:

1. Chn11.lst	3. Chn21.lst	5. Pioasgn.wgd
2. Chn12.lst	4. Chn22.lst	

The object files are:

1. Menu.obj	3. Run.obj	5. Async.obj
2. Setup.obj	4. Pio.obj	

The graphics drivers are: (TURBO C files)

1. EGAVGA.BGI	2. EGA.BGI	3. BGIOBJ.BGI

The executable files are:

1. DCL.PRJ	2. DCL.EXE

3.4 Initialization And Start-up

To enter the system from DOS, type: DCL <ENTER>

The initialization routine is then activated which sets the following parameters as

ENTRY	DATA FILE	VALUE
Channels	Params.h	0-16
PIO	Pioasgn.wgd	BOI (parallel ports)

Parallel Interface	Params.h	CD00 (Hex address)
Serial Port	Async.h	COM1
Serial Port Type	Pioasgn.wgd	I (input)
Baud Rate	User config.	(9600,19200,38400)

At present, these default values are stored in data files as listed above and cannot be viewed by the user while in the program. The values can be changed by editing these data files from DOS.

The main menu will then appear and the user can select the module to execute from here. If the user choice is *setup*, control is transferred to the setup module (controlling routine scr_setup - see Section 7.2) and if the user selects *run dcls*, the run module is stepped into (controlling routine run_dcl - see Section 8.2).

3.5 Setup

This is a User Interface program that provides the capability to communicate with the controller. The main use of this routine is to setup and tune a channel. It allows communication with only one channel at a time. This module reads and displays all the controller user-configurable and status-only parameters on two screens (toggle between screens using F4). The user is allowed to select the channel to be displayed at the start of the routine (it is assumed that each channel has been assigned with a controller address and the identification switches have been set to the corresponding address). Channels beyond the selected range will cause the routine to routine to quit and go back to the main menu.

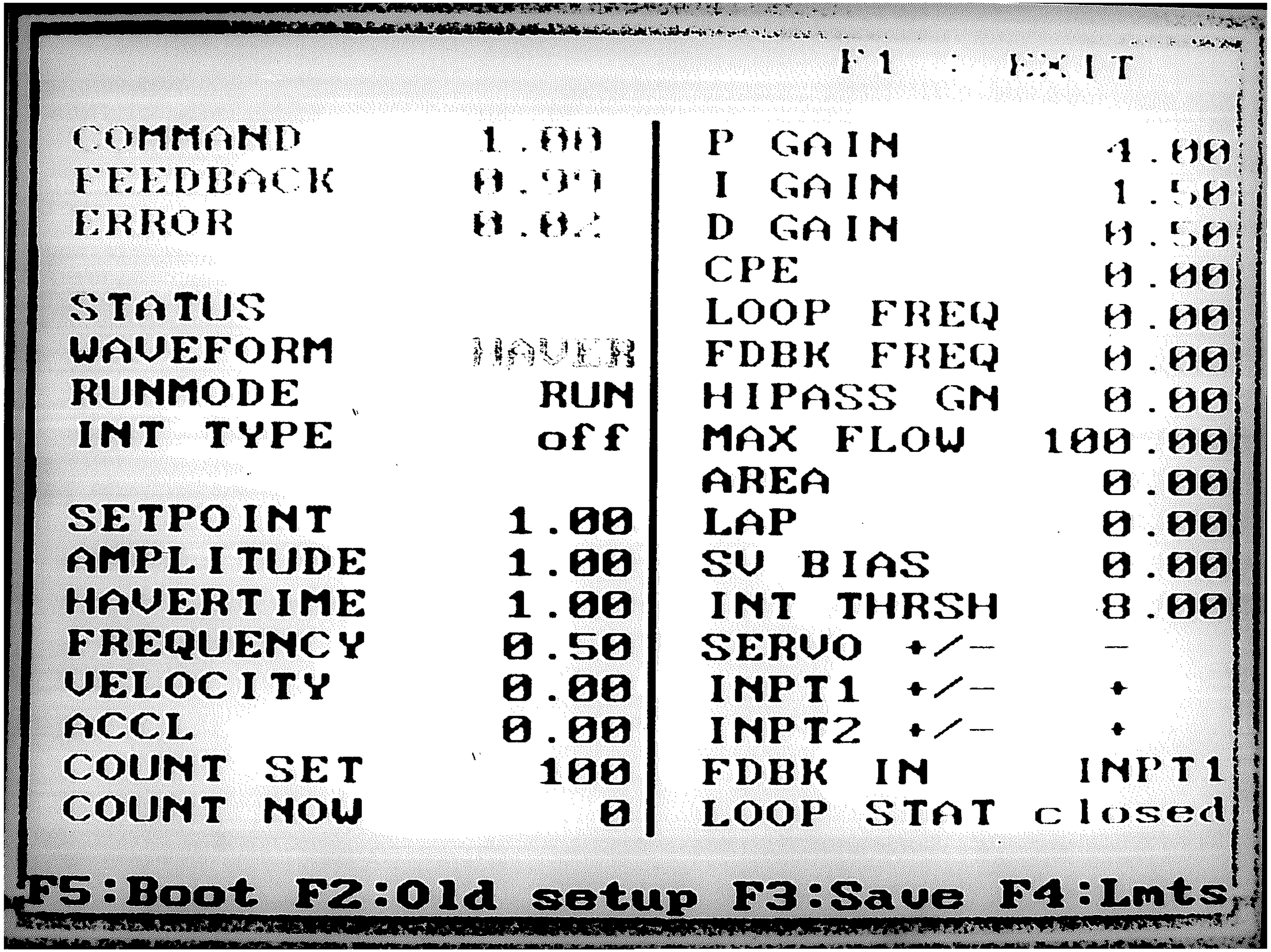
When a valid channel is selected, the screen will show all the values as currently present in the controller for that channel. If the channel has been previously backed up, a file will be present on the hard drive that can be used to recall the data and send it to the controller. When first communicating with the controller, to ensure no erroneous setup parameters are present, it is a good idea to *re-boot* the controller.

All user configurable values can be edited by merely moving the field indicator (using the 4 arrow keys-up, down, right & left), inserting the numeric value or letter (for non-numerical parameters, the first letter of the desired option) and confirming with the ***ENTER*** key. The new value is communicated to the controller *as soon as* the value is set, that is, the controller will be immediately affected by the changes that the user makes.

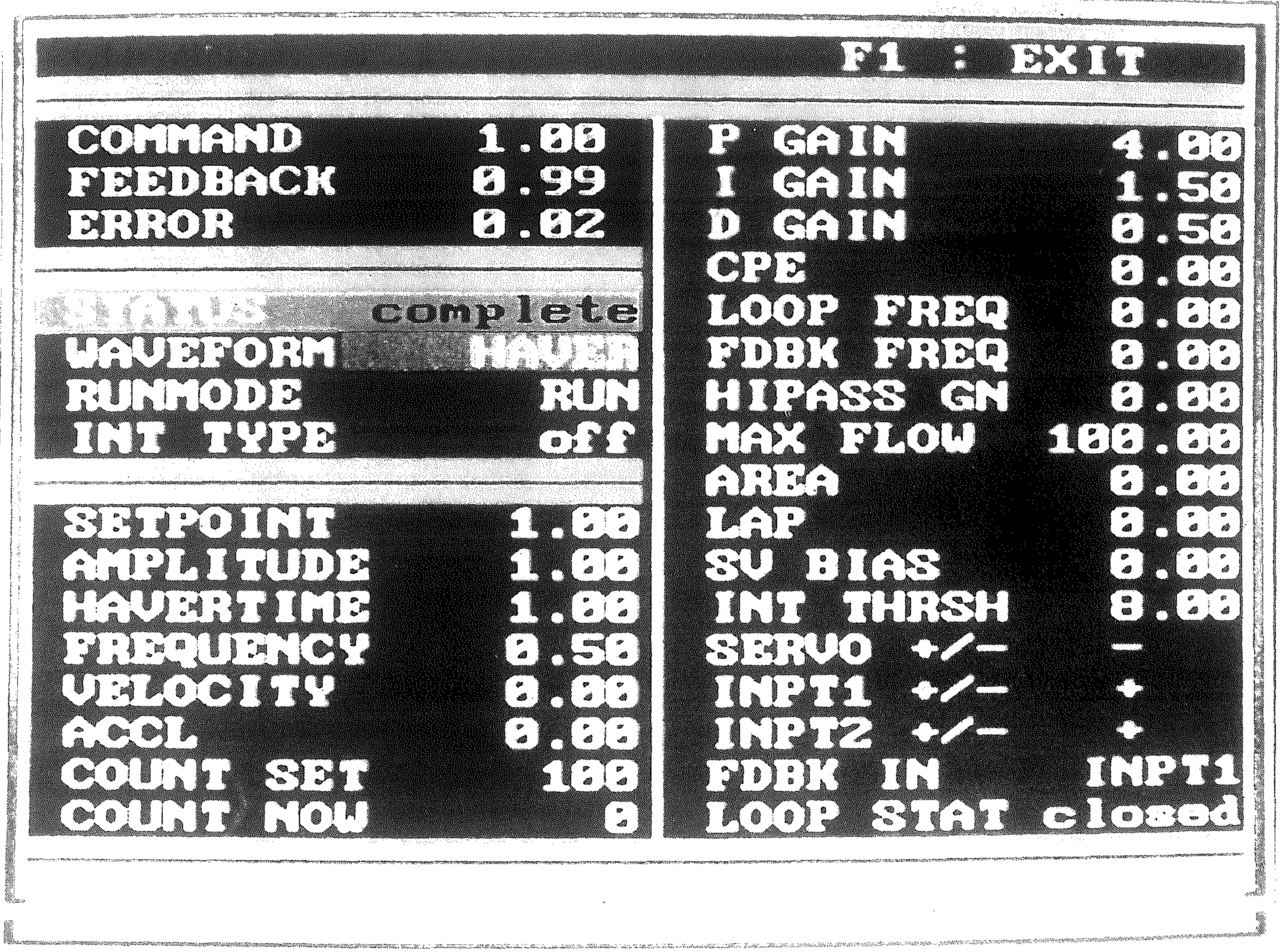
The setup screens also have Function key (F1-F10) definitions displayed at the bottom of each screen. These are used to call initialization routines, recover saved parameter values and other desirable activities.

Screen1 is associated with most of the numerical parameters of the controller (See Figure 3.2). For example, the Proportional Gain may be modified from Screen 1 to improve performance. If the channel is sluggish, one can increase the P GAIN from this screen, if it is unstable, decrease P GAIN. Polarities of the servo loop or the feedback signal can be reversed from here for proper loop closure. Again, to change a value, the user must press the ENTER key after entering the value and before moving onto another menu selection. Alt-F4 from this screen will operate the PIO function. Screen1 toggles to screen2 when F4 is pressed.

Screen2 displays controller limits and abort modes (see Figure 3.3). These are three separate controller monitored limit conditions: the signals that can have limits assigned are the feedback signal, the monitor signal and the control loop error signal. These signals can have thresholds assigned (in the engineering units of the signal being monitored) to set the boundaries for the feedback signal, the monitor signal and the error window. The status of these limit boundaries is also shown (SET if limit exceeded). The user can also assign PIO modules to a limit here by entering a PIO function number in the appropriate column. The PIO functions are used to control hydraulic power units, service manifolds or blocking manifolds. If a PIO function entry is "0" the function will be ignored.

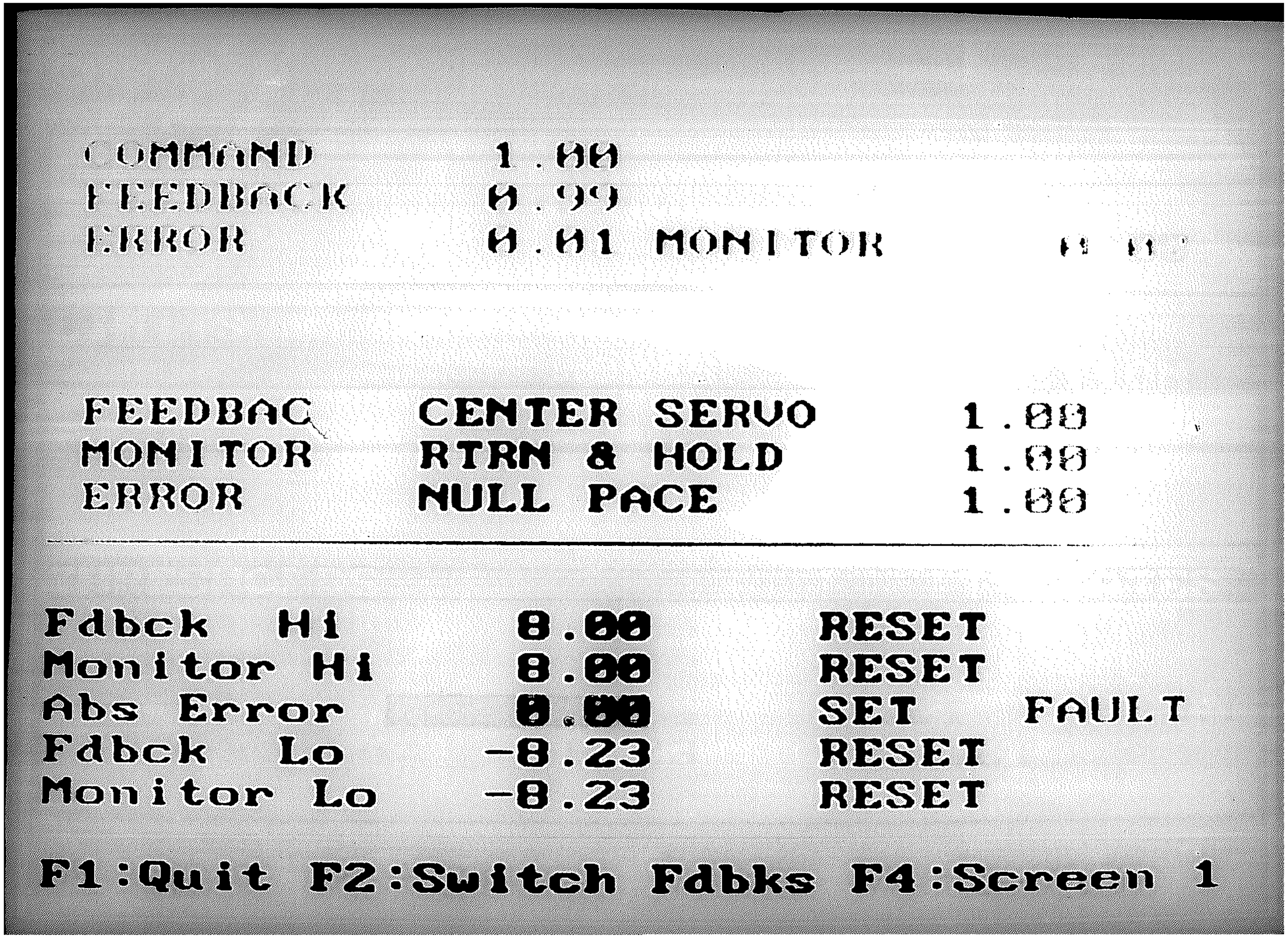


```
                                          F1  :  EXIT

COMMAND              1.00  | P GAIN               4.00
FEEDBACK             0.99  | I GAIN               1.50
ERROR                0.02  | D GAIN               0.50
                           | CPE                  0.00
STATUS                     | LOOP FREQ            0.00
WAVEFORM            HAVER  | FDBK FREQ            0.00
RUNMODE               RUN  | HIPASS GN            0.00
INT TYPE              off  | MAX FLOW           100.00
                           | AREA                 0.00
SETPOINT             1.00  | LAP                  0.00
AMPLITUDE            1.00  | SV BIAS              0.00
HAVERTIME            1.00  | INT THRSH            8.00
FREQUENCY            0.50  | SERVO +/-            -
VELOCITY             0.00  | INPT1 +/-            +
ACCL                 0.00  | INPT2 +/-            +
COUNT SET             100  | FDBK IN              INPT1
COUNT NOW               0  | LOOP STAT            closed

F5:Boot F2:Old setup F3:Save F4:Lmts
```

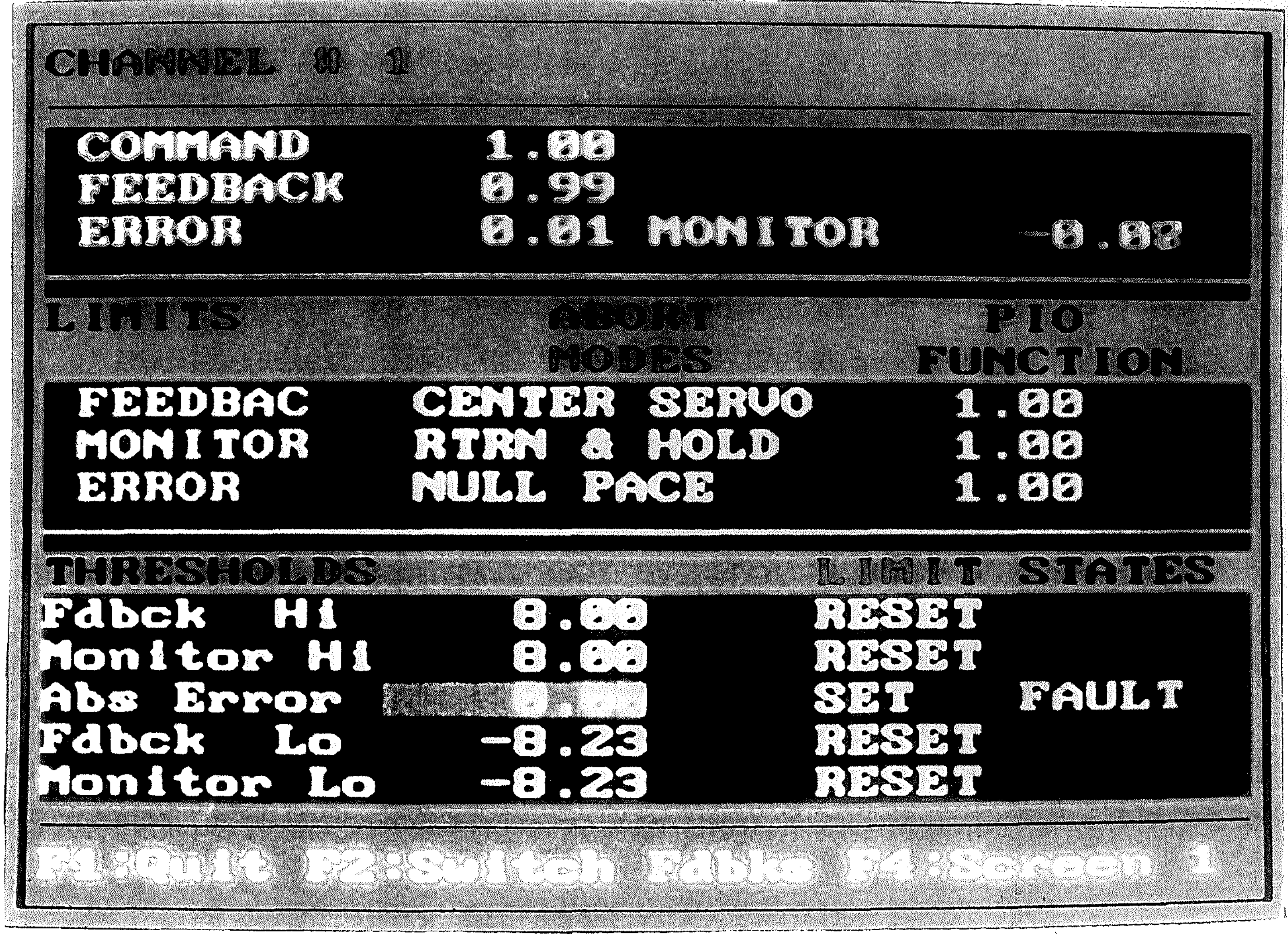


```
                                        F1 : EXIT

COMMAND          1.00        P GAIN               4.00
FEEDBACK         0.99        I GAIN               1.50
ERROR            0.02        D GAIN               0.50
                             CPE                  0.00
STATUS       complete        LOOP FREQ            0.00
WAVEFORM        HAVER        FDBK FREQ            0.00
RUNMODE           RUN        HIPASS GN            0.00
INT TYPE          off        MAX FLOW           100.00
                             AREA                 0.00
SETPOINT         1.00        LAP                  0.00
AMPLITUDE        1.00        SV BIAS              0.00
HAVERTIME        1.00        INT THRSH            8.00
FREQUENCY        0.50        SERVO +/-            -
VELOCITY         0.00        INPT1 +/-            +
ACCL             0.00        INPT2 +/-            +
COUNT SET         100        FDBK IN              INPT1
COUNT NOW           0        LOOP STAT            closed
```

Setup Screen1 - Figure 3.2



```
COMMAND          1.00
FEEDBACK         0.99
ERROR            0.01  MONITOR       [illegible]


FEEDBAC     CENTER SERVO        1.00
MONITOR     RTRN & HOLD         1.00
ERROR       NULL PACE           1.00
------------------------------------------------
Fdbck  Hi         8.00      RESET
Monitor Hi        8.00      RESET
Abs Error         0.00      SET      FAULT
Fdbck  Lo        -8.23      RESET
Monitor Lo       -8.23      RESET

F1:Quit F2:Switch Fdbks F4:Screen 1
```



```
CHANNEL # 1

 COMMAND          1.00
 FEEDBACK         0.99
 ERROR            0.01 MONITOR        -0.08

LIMITS           ABORT              PIO
                 MODES              FUNCTION
 FEEDBAC       CENTER SERVO          1.00
 MONITOR       RTRN & HOLD           1.00
 ERROR         NULL PACE             1.00

THRESHOLDS                     LIMIT STATES
Fdbck   Hi          8.00       RESET
Monitor Hi          8.00       RESET
Abs Error     [illegible]      SET     FAULT
Fdbck   Lo         -8.23       RESET
Monitor Lo         -8.23       RESET

F1:Quit F2:Switch Fdbks F4:Screen 1
```

Setup Screen2 - Figure 3.3

The following parameters are displayed on Screen1:

STATUS ONLY:

Channel #:	This is the current channel that is being addressed.
Command:	This is the current output of the internal signal generator for the desired force or motion (The controller does not receive the entire force/motion command; rather it is sent small incremental steps - this is the command).
Feedback:	This is the current output from the controlling transducer (force/motion- See parameter Fdbk In). It is the actual load/position of the actuator.
Error:	This is the difference between command and feedback.
Waveform Status:	This displays the status of the internal waveform generator (either in process or complete).
Counter Now:	This displays the current number of cycles (on Sines).
Fdbk In:	Displays the active (controlling) feedback (1 or 2).

USER CONFIGURABLE:

Waveform Type: (Non-numerical S/R/H)	The controller has 3 options for its internal waveform generator: (S)ine, (R)amp and (H)aversine.
Run Mode: (Non-numerical R/A/Z/H)	There are 4 options: Run, Abort, Zero (return) & Hold.
Integrator Type: (Non-numerical S/R/H/O)	This is the integrating function to the feedback loop. There are 4 options:
(S)ines (On):	This continuously sums the integrator term of the loop into the loop output).

(R)amp (Threshold):	This zeroes the integrator term in the loop output if error is greater than the integrator threshold.
(H)old:	Sends a constant value for the integrator to the loop output.
(O)ff :	No integral action - zero for the integrator term to the loop.
Set Point: (numerical)	This is the endpoint (Haver/Ramp) or the mean value (Sine) of the waveform.
Amplitude: (numerical)	This is the desired amplitude for a Sine waveform.
Havertime: (numerical)	For a Haversine, this is the desired length of time in seconds; for a Sine it is the period.
Frequency: (numerical)	This is the desired frequency of the Sine waveform in Hz.
Velocity: (numerical)	The desired velocity to set the ramp rate for a Ramp.
Acceleration: (numerical)	The desired acceleration for a Ramp waveform.
P Gain: (numerical)	The proportional loop gain.
I Gain: (numerical)	The integral loop gain.
D Gain: (numerical)	The derivative loop gain.
CPE: (numerical)	The feed-forward compensation to adjust the phase shift between command and feedback at a given frequency.
Loop Freq: (numerical)	Desired loop output frequency (3db) in Hertz for the output low-pass filter (zero-filter off).
Fdbk Freq: (numerical)	Desired feedback cutoff frequency (3db) in Hertz for the output low-pass filter (zero-filter off).

Hipass Gn: (numerical)	Desired feedback cutoff frequency (3db) in Hertz for the output high-pass filter (zero-filter off).
Return Time: (numerical)	Haversine time for a profile return.
Lap: (numerical)	The servo valve lap compensation value.
SV Bias: (numerical)	The servo valve electrical null bias value.
Max Flow: (numerical)	The maximum allowable flow limit in percent (0-100)
Int Thresh: (numerical)	The threshold value in controller counts of the integrator function to the loop output.
Fdbk1 +/-: (non-numerical +/-)	Sets the polarity of feedback 1.
Fdbk2 +/-: (non-numerical +/-)	Sets the polarity of feedback 2.
Servo +/-: (nonnumerical+/-)	Sets the servo valve polarity.
Loop Status: (non-numerical O/C)	Set to open or close loop.

At the bottom of screen1 are the Function key assignments:

F1: Exit from the screen - return to main menu.

F2: Recall previously saved backup of controller parameters.

F3: Save current setup to disk.

F4: Toggle to Screen 2.

F5: Boot controller- clear present parameter values and reset limits.

The following parameters are displayed on Screen2: (F4 to toggle between screens1 & 2)

STATUS ONLY:

Command: Same as screen 1.

Feedback: Same as screen 1.

Error: Same as screen 1. **Monitor:** The non-controlling feedback value.

At the bottom of screen2 are the Function key assignments:

F1: Exit from screen 2 - return to main menu.

F2: Switch controlling feedbacks. Only works if the controller is not in process.

F4: Toggle to screen 1.

USER CONFIGURABLE:

Abort Modes: (Non-numerical O/N/R/S)	There are 3 definable controller monitored limit conditions for the user; the signals that can have limit actions assigned are the ***feedback,*** the ***monitor*** and the ***control loop error*** signals. Four independent limit actions can be set for each of these.^{3}
Off:	Limits are checked and fault displayed but not acted on.
Null Pace:	The internal command generator is turned off while the fault exists. Thus the command value at the time of the fault is held until the feedback falls back within set limits. It then **resumes** command sequence unlike the other abort modes. (useful when running Sines between defined limits. **Not** available with monitor/non-controlling feedback).
Return & Hold:	When a fault is detected, the controller generates a haversine to the previous setpoint with a havertime equal to the Return time (on screen 1).
Center Servo:	When a fault is detected, the controller immediately centers the servo valve. The servo valve mechanical null should be adjusted so that the actuator does not drift. (**Not** available with monitor/non-controlling feedback).
Parallel I/O: (numerical 1-32)	PIO can be defined for each of the three signals (feedback, monitor and error).These are used to allow the controller to operate the parallel functions when a limit threshold has been exceeded. (see Appendix B for PIO definitions). The PIO functions can be used to control hydraulic power units, service manifolds, panel

lights, etc. It this entry is less than or equal to zero or greater than 32, it will be ignored.

Thresholds:
(numerical)

The feedback and monitor inputs each have limit threshold window defined by a Hi and Lo value. Abort action is taken if the signal strays outside this window. The error signal only has an absolute threshold (or a Hi value), the crossing of which leads to an abort action specified in the abort mode.

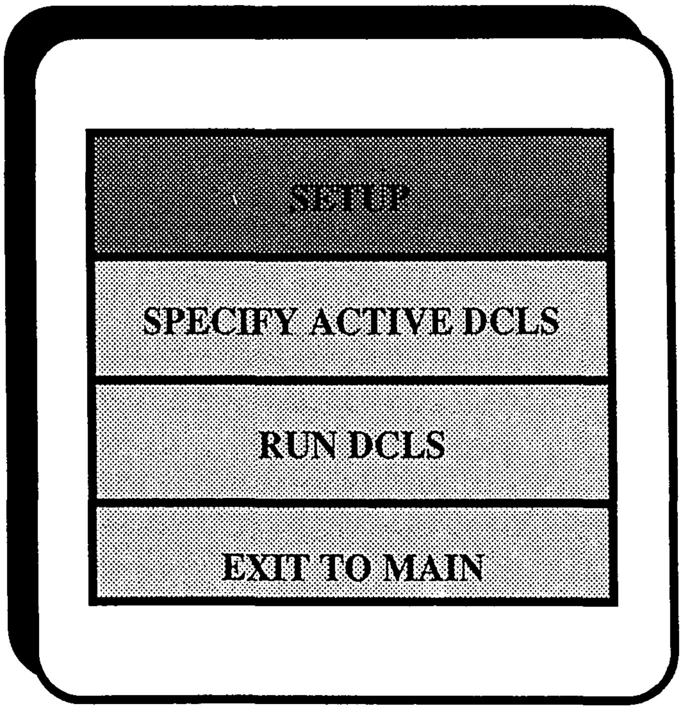
Threshold Status:
(Non-numerical S/R)

Indicates the state of the corresponding threshold for the 3 signals- can be manually SET and RESET. These are automatically updated to reflect the true state of each of the five limits.

3.6 Run

The purpose of this routine is to demonstrate the method in which user test-specific programs will link with the general controller setup and initialization routines and be able to call the screens and routines of the former from within the test program. It is also a valid fatigue test control program. The basic idea was to free the user from the low-level I/O and communication protocols (which are repetitive for every test) by providing easy-to-use commands and means for accessing all parameters. This allows the user to spend time on designing the test sequence instead of having to write the code to communicate with the controllers. The communication mechanism is thus subordinate to the test (Run) program. This module is a constant-amplitude fatigue test control program.

A constant-amplitude



Run Menu- Figure 3.4

fatigue test involves running 1 or more actuators as a group in continuous sinewave mode with either force or displacement as the controlling feedback. The number of sine waves performed by each actuator (counts) have to be constantly monitored and it is desirable to be able to start and stop each actuator individually without affecting the others in the group. The routine first pops up a menu (see Figure 3.4) with four choices: Setup, Specify, Run andExit. The following is a brief description of each.

Setup Individual DCLS:

This allows access to individual channels (setup screen1)

Specify Active DCLS:

This pops another menu shown in Figure 3.5, which allows the user to configure controllers for groups. Up to 16 channels can be specified as **(A)CTIVE** or **(O)FF.**

The function key defined is:

F1:	Exit to the main RUN menu.

Run DCLS:

Starts the test. This displays another screen with the following information for each active channel: (See Figure 3.6)

Channel #:	Number of the active channel.
Count:	Total number of sines sent to the controller-constantly updated.
Limit Modes:	Abort modes for feedback, monitor and error (setup screen 2)
PIO:	The PIO function for the 3 signals (setup screen 2).

The function keys defined are:

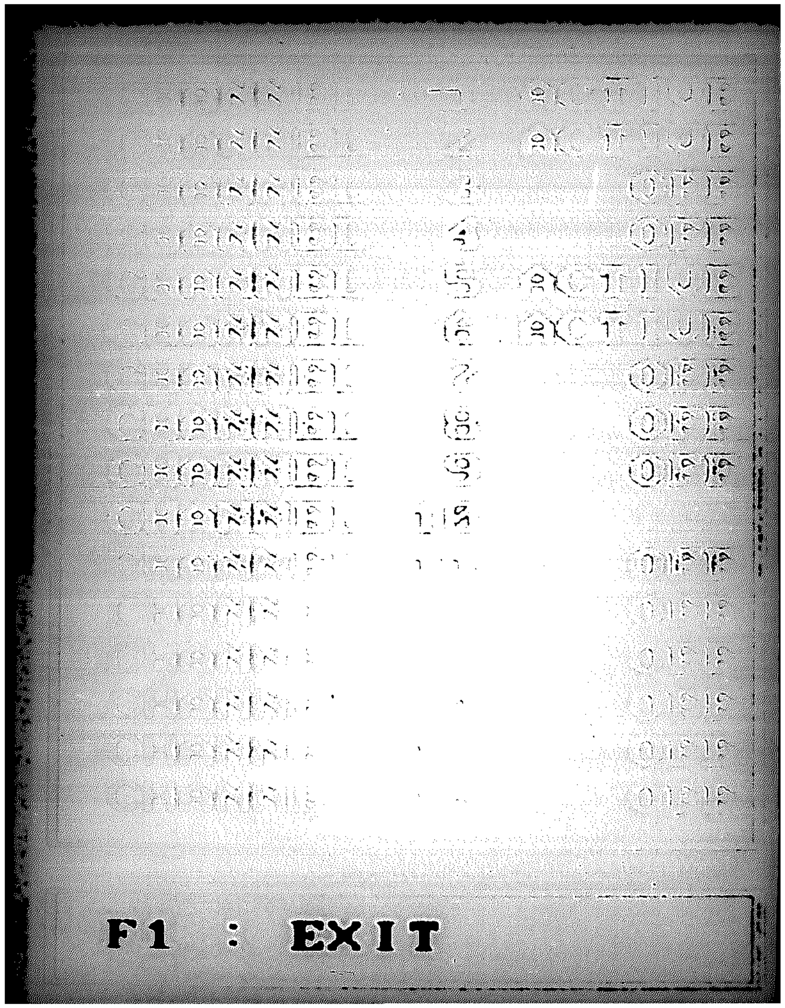
F1: Exit to the main RUN menu.

Counts will be backed up in a file *COUNT.IFT* on the hard drive.

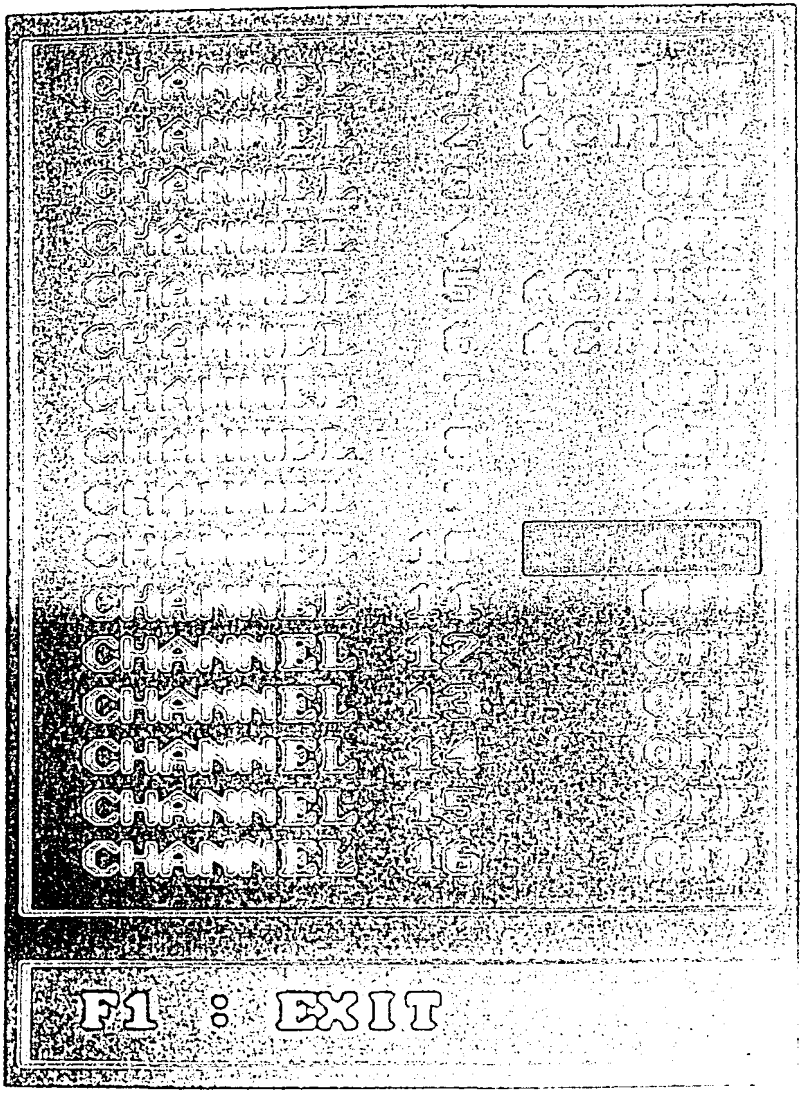
F2: Modify counts (only if controllers are not running).

F3: Start/Stop the test-all actuators.

Exit: Return to the main menu.

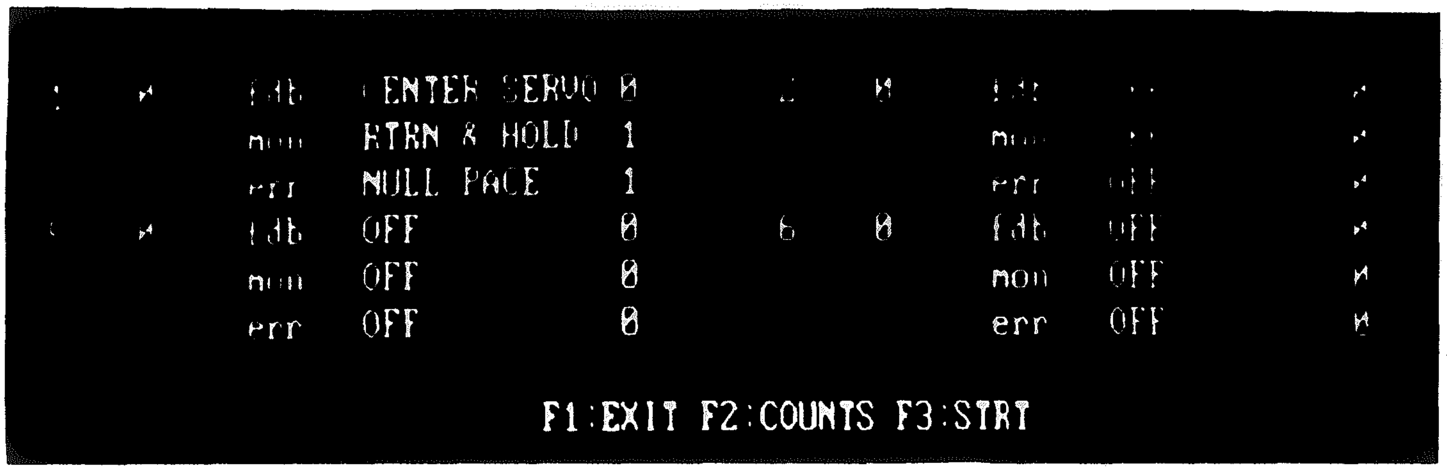


```
CHANNEL  1   ACTIVE
CHANNEL  2   ACTIVE
CHANNEL  3      OFF
CHANNEL  4      OFF
CHANNEL  5   ACTIVE
CHANNEL  6   ACTIVE
CHANNEL  7      OFF
CHANNEL  8      OFF
CHANNEL  9      OFF
CHANNEL 10   [illegible]
CHANNEL [illegible]      OFF
CHANNEL [illegible]      OFF
CHANNEL [illegible]      OFF
CHANNEL [illegible]      OFF
CHANNEL [illegible]      OFF
CHANNEL [illegible]      OFF

F1 : EXIT
```



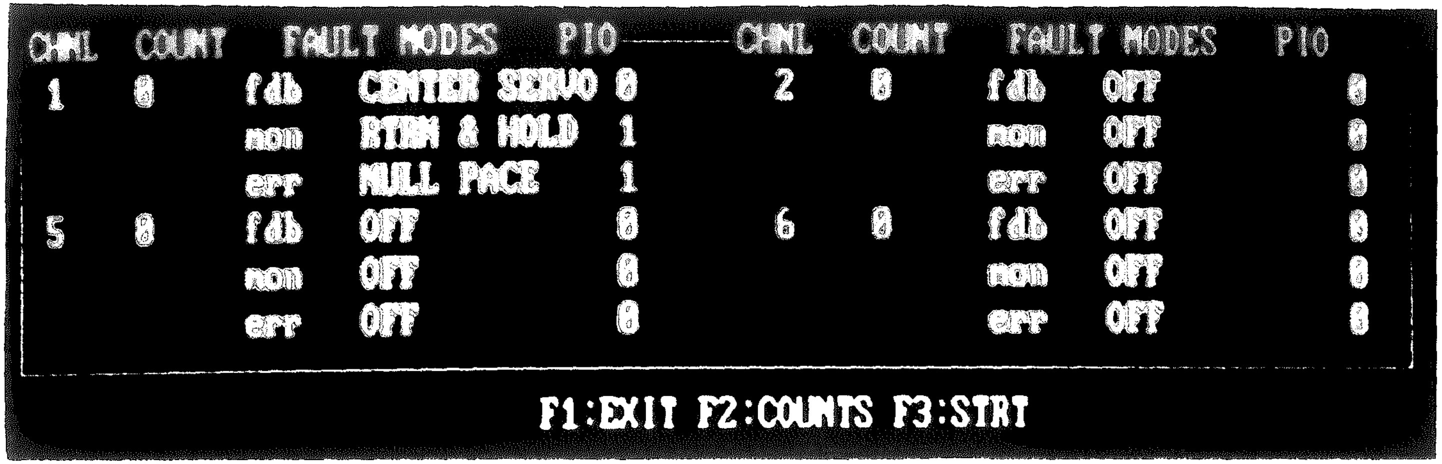
```
CHANNEL  1  ACTIVE
CHANNEL  2  ACTIVE
CHANNEL  3     OFF
CHANNEL  4     OFF
CHANNEL  5  ACTIVE
CHANNEL  6  ACTIVE
CHANNEL  7     OFF
CHANNEL  8     OFF
CHANNEL  9     OFF
CHANNEL 10  [illegible]
CHANNEL 11     OFF
CHANNEL 12     OFF
CHANNEL 13     OFF
CHANNEL 14     OFF
CHANNEL 15     OFF
CHANNEL 16     OFF

F1 : EXIT
```

Run Screen1 - Figure 3.5



```
[illegible]  0   fdb  CENTER SERVO 0  [illegible]  0  fdb  [illegible]  0
                 mon  RTRN & HOLD  1                  mon  [illegible]  0
                 err  NULL PACE    1                  err  OFF          0
[illegible]  0   fdb  OFF          0  6            0  fdb  OFF          0
                 mon  OFF          0                  mon  OFF          0
                 err  OFF          0                  err  OFF          0

              F1:EXIT F2:COUNTS F3:STRT
```

Run Screen2 - Figure 3.6



```
CHNL  COUNT   FAULT MODES       PIO ------ CHNL  COUNT   FAULT MODES     PIO
 1     0     fdb   CENTER SERVO  0          2     0     fdb   OFF          0
             non   RTRN & HOLD   1                      non   OFF          0
             err   NULL PACE     1                      err   OFF          0
 5     0     fdb   OFF           0          6     0     fdb   OFF          0
             non   OFF           0                      non   OFF          0
             err   OFF           0                      err   OFF          0

                     F1:EXIT F2:COUNTS F3:STRT
```

Run Screen2 - Figure 3.6

CHAPTER 4

PARALLEL INPUT-OUTPUT

4.1 Introduction

This chapter describes in detail, the Parallel I/O option that allows the user to control external hardware such as panel switches, service manifolds, etc. through computer control of parallel I/O. This feature is most useful in case of an error being tripped as it allows the user to automatically shut down hydraulic pressure, display warning lights, etc. in the event of a fault.

4.2 The Burr-Brown PIO Board

A Burr-Brown Parallel Logic board is used to interface between the external devices and the computer. The I/O board contains up to 16 I/O modules each of which can be divided into two ports with 8 points each. There are four types of these modules: DC input, DC output, AC input and AC output. The software can operate 2 Burr-Brown boards for a total of 32 external points. Each port of 8 modules can be specified as input or output by the user. Once the port is defined as input or output, the I/O modules must be chosen to match the configuration. The Parallel Input/Output configuration table has the following fields:

FUNC#	NAME	INIT VAL.	PORT	IN/OUT	BIT MASK
1	" "	0	0	OUT	*.......
2	" "	0	0	OUT	.*......
3	"PRESS "	0	0	OUT	..*.....

and so on up to 32.

FUNC#: This is the numerical designation used by the PIO commands to represent different sets of I/O points. One or more I/O points maybe controlled using a single function number.

NAME: This is the user-defined label for the given function number.

INIT VAL: This is a decimal number between 0 and 255 which is the initial state for a given function (to be set at start-up). It is converted to a binary form and sent to the port.

PORT: This a number between 0 and 3 identifying the port which the function point belongs to. Each I/O module or point is one bit of an 8-bit PIO control word sent through the given port.

IN/OUT: This defines the port direction (read/write) according to the initial system configuration (PIO configuration = BOI means B=Burr-Brown, O = port "0" as output, I = port "1" as input. A maximum of 4 letters can follow the B i.e. 4 output ports can be defined). When a user selects a port # for a particular function, this automatically switches to the type of port defined in the initial configuration.

BIT MASK: This determines which I/O modules are affected by the function value when it is sent to a given port. An asterisk (*) indicates that a module is addressable and a period (.) indicates that it is not. When more than one asterisks are used, (i.e. > 1 modules), the (*) must be in consecutive bit positions. The maximum value that has meaning for a function is determined by the number of asterisks in the bit mask of a function:

MAX VAL = $(2^{\#\text{of asterisks}}) - 1$

e.g. Bit Mask =..***... indicates that the third, fourth and fifth outputs of that PIO port are addressable by the function.

MAX VAl = $2^3 - 1 = 7 = 111$ in binary.

If the value = 5 = 101 is sent to the PIO port, I/O points 3 and 5 will turn on, 4 will be off.

If value > 7 all outputs of that port will turn on.

The information shown above is stored in a file PIOASGN.WGD for the PIO initialization routine. (At present, to change any of these values, one must externally edit the file.In the future, this will be a menu selection from within the program). The control actions defined will occur when a particular PIO function is activated during execution of a motion control waveform. In the above example, Bit zero of output port "0" will be addressed by function 1. When this function is called from a motion control waveform the module will close its contact.

CHAPTER 5

PROGRAMMING GUIDE

5.1 Introduction

The next four chapters describe in detail, each module and the lower level routines it calls. The description format is top-down, starting with the main module and going down to the component routines and the functions they call.

At every level, a functional description of the routine is given, along with the names of the higher level routines that use it, as well as a listing of the routines one level down that it calls. The global data structures that each routine affects or uses are also individually mentioned along with the header files included at the time of compilation. Pertinent local variables that are useful in understanding the purpose behind the routine are also described. Finally, the files opened (for reading or writing) by the routine are mentioned.

5.2 Program Organization

The routines are grouped under four headings on the basis of their function. The groups are: Initialization, Setup, Run and Graphical. The first three are the program modules already described in Chapter 3. The fourth contains routines which initialize for graphics, draw menus, print numbers or strings at specific screen locations, change background color and in other ways affect and control the displaying of information in all the user interface screens and menus.

At start-up, the first routines executed are the initialization routines async-init() and pio_init(). Then the main menu pops up and the setup or the run routines are entered into depending on the user selection. If the user choice is setup, the control is transferred to routine scr_setup(). If the choice is run, routine run_dcl() is executed.

5.3 Program Files

The files which hold these routines are async.c (for all communication routines) , setup.c (for all the setup module routines), run.c (all the run module routines) and menu.c (graphical routines - also holds main() - the main menu).

Header files include async.h (communication default values), params.h (controller parameters) and run.h (data structures associated with run_dcl() like ACTIVE_CHN). In addition, C library header files like stdio.h, stdlib.h, graphics.h, string.h, etc. are included in the appropriate files. These contain useful C library functions (e.g. printf, file open, etc.) and constants (NULL, EOF, etc.) which can be used in the program without redefinition.

5.4 Linking User Test-Control (Run) Programs

One of the objectives of this endeavor was to allow the user freedom of control over the test programs while being able to benefit from the canned communication routines in the package. Because of the modularized structure of the software, the user can incorporate just the required modules in his/her test-specific control program, thus saving memory and improving speed of execution. Listed below are a few examples of how the modules can be called within a user program written in TURBO C - Note the TURBO C compiler must be available to link all the modules.

5.4.1 Linking the setup module with the user program

To be able to access the setup screens 1 & 2 from the user program, the user code must have the lines:

```
        main() {

        /* user variables definition - integer, real etc.  */

        ........

        scr_setup()

        .......

}
```

NOTE: When the program steps into scr_setup(), there will be no updating / checking of user-defined variables and screens until the Function key F1 is pressed to exit from the routine. Standard parameter and pio functions checking will be done by scr_setup itself from setup screen 1 & 2.

The following files must be linked with the user routine at compile-time:

Menu.c Setup.c Async.c Pio.c

To do this internally from the TURBO C editor:

1. create a project file of any name and list in it, the above file names along with the name of the user program file. (one on each line).

2. Then choose *Project File* from the Options menu of the TURBO C editor and enter the name of the created project file in the space provided.

3. Choose *Compile/Run* from the Run menu and an executable file with the same name as the project file but with the extension .EXE will be created.

To compile from outside the editor enter the following lines at the DOS prompt:

tcc -e*executable filename* menu.c setup.c async.c pio.c *user filename* .c

tcc -mx menu.obj setup.obj async.obj pio.obj *user filename*.obj

The -mx option on the second line refers to the memory model size (room the program occupies when executing) where x can be one of t, s, m, l ,h (tiny, small, medium, large or huge - generally medium is the size chosen when the program contains graphics).

5.4.2 Linking individual routines with the user program

The user code must have the lines:

```
main() {

/* user variables definition - integer, real etc. */

........

name of routine (e.g. read_parameter)

.......}
```

The files containing the routine and the lower-level routine it calls must be linked with the user routine at compile-time in the same fashion as described above. For a graphical representation of the program routines and their hierarchy, refer to Figures 6.1, 7.1 , 8.1 and 9.1.

Chapter 6

INITIALIZATION

6.1 Introduction

The initialization module is described in this chapter. Figure 6.1 shows the structure of this module.These are the routines executed during start-up. Two separate entities need to be initialized - the serial communication port (8250 UART chip-Appendix A) and the parallel input/output board (Burr-Brown PIO board- Appendix B).

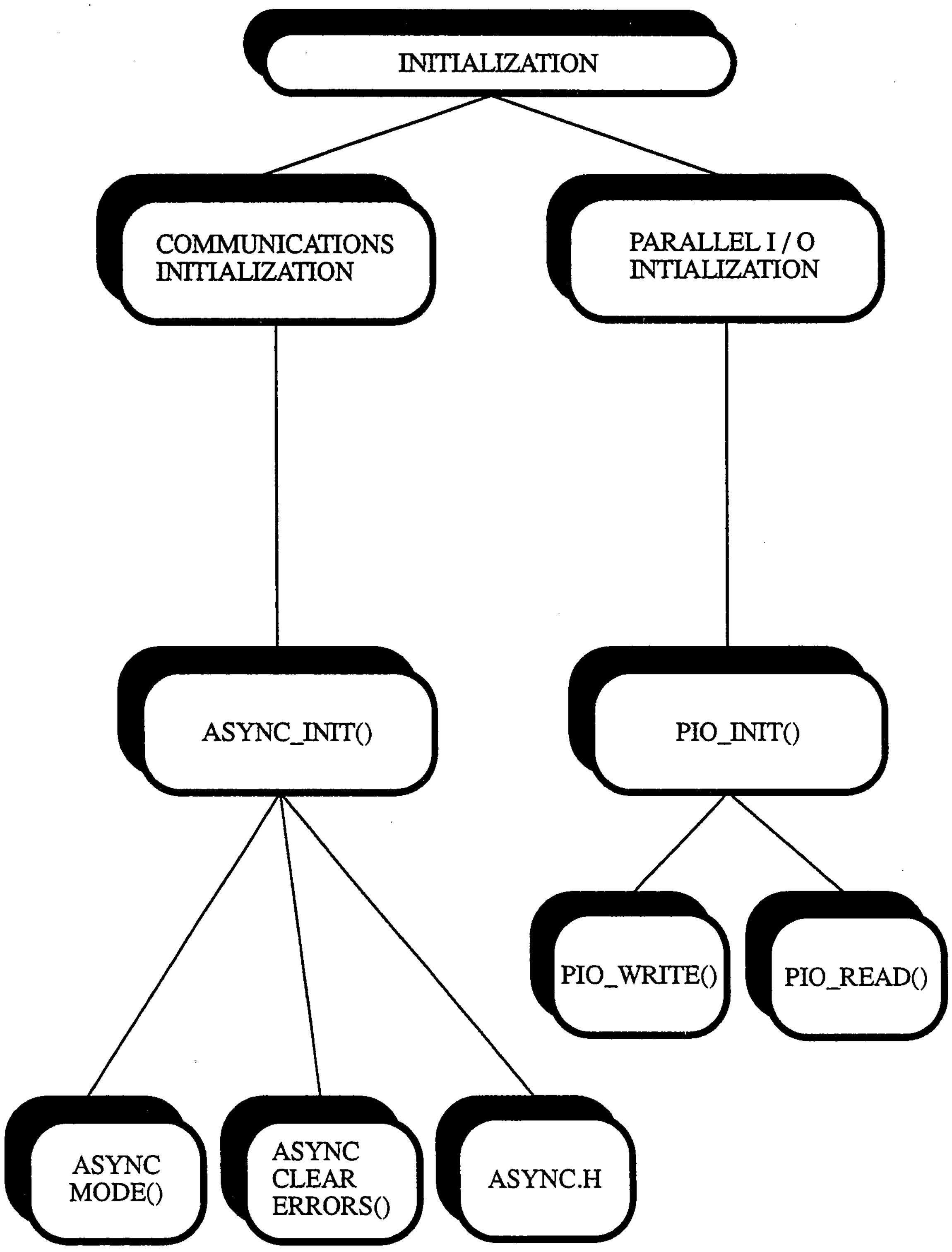
6.2 Communication Port Initialization

The Universal Asynchronous Reciever/Transmitter model 8250 is the serial device which supports up to 16 controllers and located at one of four possible I/O addresses (COM1-4). The serial port allows the CPU to communicate with peripheral devices that recognize serial data signals. Data transmission is interfaced through a 9-pin connector and controlled by an Intel 8250 or equivalent programmable asynchronous communications controller. When serial data is transmitted to a peripheral device, the controller converts the data stream into an 8-bit byte and stores it into a temp register until the CPU executes a read operation. When the CPU writes to the peripherals device, the controller converts the data byte into a serial format in preparation for transmission.

At start-up, the communications port and the 8250 UART chip have to be setup for asynchronous serial receive/transmission. For this, a set of control words pertaining to baud rate, parity, word length and the number of stop bits must be output to the line control register of the UART chip. The modem control register must be set for no modem control and the interrupt enable register for no interrupts. Also, data interrupts must be enabled and the port reset. The higher level routine that does this is async_init().

6.2.1 async_init()

This routine sends all the relevant control words to configure COM1 or COM2 for communication at the user-specified baud rate (B9600,B19200,B38400), parity (even, odd, mark, space, none), word length This



Initialization Module-Figure 6.1

(constants WORD8, WORD7,WORD6,WORD5), and the number of stop bits (STOP1, STOP2).

The default configuration is: (see 6.2.2 for other options)

Port	COM1	(0x03F8)
Baud rate	9600	(0x00C)
Parity	None(n)	(0x00)
Wordlength	Word8	(3)
Stop bits	1	(0)

Header file:	Async.h.
Data file:	None.
Global Var:	None.
Local Var:	Structure **PROTOCOLSTRUCT** ***protocol*** {

Protocol is an example of protocolstruct with the following data fields:

integer **portnum** (1 or 2 whether COM1 or COM2)

unsigned integer **baud** (9600,19200 or 38400)

character **parity** (E,O,M,N,S)

integer **wordsize** (WORD8, WORD7, WORD6 or WORD5)

integer **stopbits** (1 or 2)}

Called by:	main()
Calls:	Async_mode(), async_clear_errors().

6.2.2 async_mode()

This converts the user-specified mode values into the corresponding binary control words and sends them to the UART chip. The routine first breaks up the defined baud rate into a low and high byte and outputs them individually at the UART base address (3F8). It then ORs (binary addition) the parity, word length and stop bits into a mode value and sends it to the line control register (LCR) of the UART chip. The default addresses on the UART as defined in async.h (6.2.4) are:

BAUDLO = Address lower byte of baud sent to = UART_ base = 0x03F8(COM1)

BAUDHI = Address higher byte of baud sent to = UART_base + 1

Line Control Register = LCR = UART_base + 3

Modem Control Register = MCR = UART_base + 4

Modem Status Register = MSR = UART_base + 6

Header file:	Async.h.
Data file:	None.
Global Var:	None.
Local Var:	Structure **PROTOCOLSTRUCT** ***protocol*** (see async_init())
Called by:	Async_init().
Calls:	None.

The Mode Byte Breakdown is:[4]

BIT	**DESCRIPTION**		
7 6 5	4 3	2	1 0
Baud Rate	Parity	Stop bits	Word Length.
0 0 0 - 110	0 0 - None	0 - One	0 0 - Don't care.
0 0 1 - 150	0 1 - Odd	1 - Two	0 1 - Don't care.
0 1 0 - 300	1 0 - Don't care		1 0 - Seven bits.

0 1 1 - 600	1 1 - Even	1 1 - Eight bits.
1 0 0 - 1200		
1 0 1 - 2400		
1 1 0 - 4800		
1 1 1 - 9600		

The above information is sent as the higher order byte of the mode word. For a baud rate greater than 9600, the lower byte is significant - (19200 - control word = 0x006; 38400-control word=0x003).

6.2.3 async_clear_errors()

This enables the programmable interrupt controller (PIC), resets all errors and sets the Data Ready line (DSR) in the UART. To do this, it first disables interrupts, reads the line status register (LSR) to reset errors and then enables the data ready interrupt. It then sends a request to read interrupt throughout the interrupt enable register (IER) and sets the request to send (RTS) line by reading and writing to the modem control register (MCR). At exit, it enables all normal interrupts (like keyboard interrupts) again.

Line Status Register = LSR = UART_base + 5

Interrupt Enable Register = IER = UART_base + 1

Modem Status Register = MSR = UART_base + 6

Header file:	Async.h.
Data file:	None.
Global Var:	None.
Local Var:	None.
Called by:	Async_init().
Calls:	Built-in C functions- disable() and enable() for software interrupts.

6.2.4 async.h

This is the header file containing the values of all the default addresses used in the communication initialization routines. It is included with all the modules at compile-time. Constants for configuring the communications mode like WORD8 (word length), PARNONE (parity), STOP1 (stop bits) and B9600 (baud rate) are defined here. The base addresses of the UART and its registers are also stored here. Functions to read the communications port, the UART registers and port errors are also written in this file.

Header file:	C library header files like stdio.h, bios.h, stdlib.h
Data file:	None.
Used by:	Async_init(), Scr_setup(), Run_dcl() (the main module routines).
Calls:	Built-in C functions, outport(), inport() for reading the comm. port.

6.3 Parallel I/O Initialization

The parallel I/O is a Burr-Brown PCI-20001C located at the user-specified memory address (CD000, C0000 or D0000). The following routines configure and enable the PIO board ports with the appropriate direction controls and define the pins to which external devices are connected. The existing port status is also cleared and the board reset.

6.3.1 pio_init()

This routine executes the following sequence to initialize the PIO board according to the configuration specified in the file ***PIOASGN.WGD***. This file holds the user's choices for the Parallel function. The file is opened and the PIO ports, the set functions (devices attached), their initial values and direction (input/output) are read into an array of structure PARALLEL_IO, PIO[32]. The ports directions are then output to the appropriate control register-83H for ports 1 & 2 and C3H for ports 3 & 4. (0=output, 1=input). The next step is to enable the direction of each port buffer by writing to the enable register, 82H. Finally, all PIO channels are cleared by writing the set initial values to the four output data registers. Digital data can be output through the appropriate I/O register (80H, 81H, C0H, C1H) - to send a HIGH the corresponding bit in the output byte must be set to 1; to send a LOW set to 0. Data can also be read from the above I/O registers. (see Appendix B for details).

Header file: Async.h.

Data file: PIOASGN.WGD.

Global Var: Arrays of Structure ***PIO[32]***

PIO[] - array of structure ***PARALLEL_IO*** and holds the user PIO specifications.

Structure PARALLEL_IO (and each index of PIO[]) has the data fields:

integer fn;	(PIO function number 1-32)
integer init;	(initial value of PIO function 0-1)
integer port;	(function port number 1-4)
integer posn;	(bit number 0-7 of port assigned to the function)
integer bits;	(number of port bits addressable by function)
char star;	(layout of function bits assignment
	* = addressable; · = not addressable)

Local Var: baseadd, offset.

baseadd	assigned memory location of Burr-Brown board
offset	memory location of ports relative to baseadd -128/190.

Calls: pio_read(), pio_write(), peek() and poke() for initializing the PIO board.

6.3.2 pio_read()

This routine reads the channel value in the I/O registers at addresses 80H, 81H, C0H & C1H, one byte at a time. To check the value for a corresponding channel its bit position in the byte must be isolated - if the bit is 0, the input is LOW; if 1, the signal is HIGH.

Header file:	Async.h.
Data file:	PIOASGN.WGD.
Global Var:	Arrays of Structure PIO[].
Local Var:	baseadd, pinno, offset, port_val.

baseadd	see 4.2.1
offset	see 4.2.1.
pinno	channel to be read.
port_val	byte read in from the port

Called by:	pio_init(), scr_setup(), limits, check_chn_faults().
Calls:	peek() and poke() for initializing the PIO board.

6.3.3 pio_write()

This routine writes to the corresponding channel in the I/O registers at addresses 80H, 81H, C0H & C1H, one byte at a time. To output a HIGH level for a corresponding channel its bit position in the byte must be set to 1; for a LOW to 0. The byte is then output from the corresponding port.

Header file:	Async.h.
Data file:	PIOASGN.WGD.

Global Var:	Arrays of Structure PIO[].
Local Var:	baseadd, pinno, offset, port_val.

baseadd	see 4.2.1
offset	see 4.2.1.
pinno	channel to be read.
port_val	byte read in from the port

Called by:	pio_init(), scr_setup(), limits, check_chn_faults().
Calls:	peek() and poke() for initializing the PIO board.

CHAPTER 7

SETUP

7.1 Introduction

This chapter describes the routines which interface between the controller parameters and the user display. (Screens 1 & 2). The main controlling routine is scr_setup() (see Figure 7.1). It reads and writes the controller parameters, converts between engineering units and controller counts and displays the information onto the screens for the user's benefit. It continuously updates the values of the status only parameters and calls different routines based on the sequence of keys pressed by the user. It is essentially a huge case statement which branches on the key pressed; for example: when function key F5 is pressed, the dcl_boot routine is executed. The information about the parameters is stored in the header file params.h. Appendix B has a listing of all the parameters and other information about them.

7.2 Scr_setup()

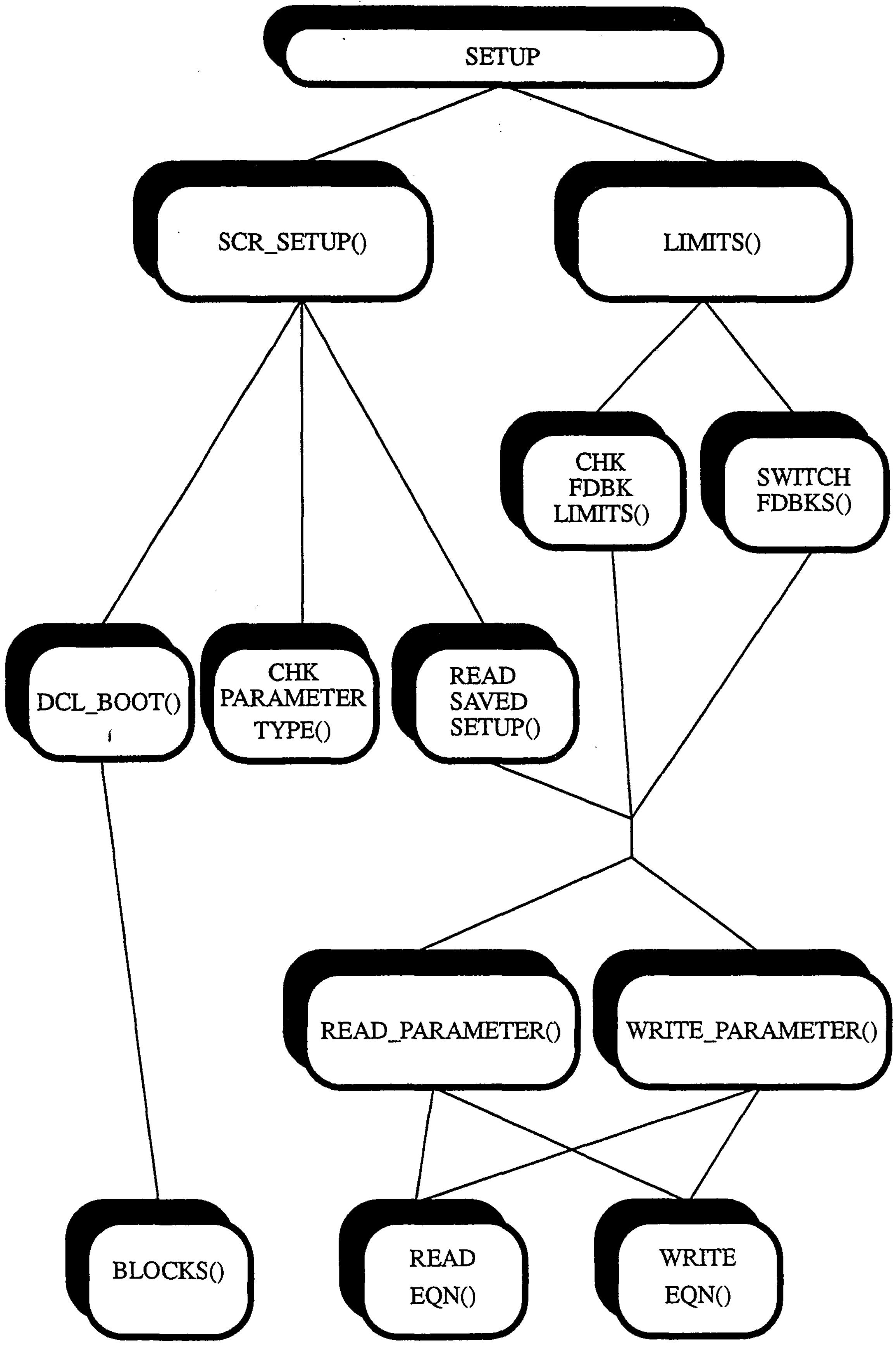
It first draws the valid screen, then reads each parameter value from the controller, converts it to engineering units, and fills it in the appropriate position on the screen. It calls *read_parameter()* and *write_parameter()* to communicate with the controller. While in screen1, the user is permitted to browse through these values using the four arrow keys (up, down, right & left) and can directly modify and send the desired value to the controller from the screen. The numerical values are converted from engineering units to controller counts and written to the correct controller register, while t he non-numerical parameters (e.g. waveform type-Sine, Haver, Ramp) are taken and the corresponding control word generated and conveyed to the controller register (e.g For Sines, the control word is 192) by calling *chk_parameter _type()*. Note that data is sent or read from the controller in blocks of 5 bytes by the routine blocks()

Function key assignments:

F1: return.

F2: read_saved_setup()

F3: Save current setup to disk.



Setup Module-Figure 7.1

F4: Toggle to Screen 2, limits().

F5: dcl_boot().

Header file: Async.h, Params.h, Run.h.

Data file: CHN***(chn#)(fdbk#)***.!ST for saved setup (read/write).

e.g. Setup file for chn=1 with controlling feedback = fdbk1 is ***CHN11.!ST***

Global Var: Arrays of Structures ***PARAMETERS[32], ACTIVE_CHN[16], b[5].***

PARAMETERS[] - array of structure ***param*** and holds all the controller parameters.

Structure param (and each index of PARAMETERS[]) has the data fields:

integer reg;	(register # of controller parameter)
float scale;	(scaling factor for units-counts conversion)
int no_bytes;	(maximum size of parameter value
int bit;	(masking bit)
int eqtyp;	(conversion equation #)
int xpos;	(row position on screen1)
int ypos;	(column position on screen 1)
char name;	(name of parameter)
double val;	(last read value)

ACTIVE_CHN[] defined in Run module.

b[] array of five bytes to communicate a single parameter to the controller.

b[1]	240 + channel# (e.g for chnl1 = 241)
b[2]	reg (for write, parameter register# e.g. for waveform = 150) (for read = 155 always)
b[3])	Data MSB for write / reg# for read.
b[4]	Data LSB for write/ "0" for read
b[5]	checksum.

To read chnl 1 waveform: b[] = {241, 155, 150, 0, checksum}

To write chnl 1 Sine wave: b[] = {241, 150, 192, 0, checksum}

Local Var: Chnl, Keyb, Curr, Prev, Waveform.

Chnl	current channel being addressed
Keyb	last key pressed
Curr	current cursor (at parameter value) position
Prev	last cursor position
Waveform	last written waveform and mode type.

Called by: main().

Calls: chk_parameter_type(), dcl_boot(), read_saved_setup(), read_parameter(), write_parameter(), blocks(), check_chn_faults, pio_write(), pio_read().

7.2.1 chk_parameter_type()

This is called every time a parameter is read from or written to the controller. This routine takes each parameter value, prints it in the set position on the screen if it s numerical or, if non-numerical, converts the control word into a string understandable to the user.

For example, if the setpoint value is to be printed, whatever is held in PARAMETERS[6].val will be printed as a floating point number. To print the waveform type, however, the value in PARAMETERS[3].val is taken and a lookup is done to match it to the correct mode (e.g. sInes = 192) and the matched string is printed (in this case, "Sines" is printed in the screen position).

Header file:	Async.h, Params.h, Run.h.	
Data file:	None.	
Global Var:	Array of structures PARAMETERS[32] (see 7.2 for details).	
Local Var:	chnl, par_no.	
	chnl	current channel being addressed.
	par_no	index for the array PARAMETERS[]
Called by:	scr_setup()	
Calls:	read_parameter().	

7.2.2 dcl_boot()

This is called when function key F5 is pressed when in screen1 of the Setup module. It causes the controller parameters to be reset and all limit thresholds to be cleared. It puts the controller in Hold, so that the reset parameters sent to the controller are not acted upon. This is useful when tuning the controller in the beginning or when restarting from a fault.

Header file:	Async.h.	
Data file:	None.	
Global Var:	None	
Local Var:	chn, b[5].	
	chn	current channel being addressed.
	b[]	(see 7.2. for details)

Called by:	scr_setup()
Calls:	blocks().

7.2.3 read_saved_setup()

This is called when function key F2 is pressed when in screen1 of the Setup module. It is used to transfer previously backed up data (a particular saved configuration) to the controller. It first checks if the controller parameter values have indeed been saved previously for that channel. If so, it opens the file CHN***(chn#)(fdbk#)***.!ST, reads each parameter value into the array PARAMETERS[] and writes it out to the controller by calling *write_parameter(). It* then calls *chk_parameter_type*() to update the values on the screen.

Header file:	Async.h, Params.h, Run.h	
Data file:	CHN***(chn#)(fdbk#)***.!ST (see 7.2 for file naming details)	
Global Var:	PARAMETERS[]	
Local Var:	chn, b[5].	
	chn	current channel being addressed.
	b[]	(see 7.2. for details)
Called by:	scr_setup()	
Calls:	write_parameter(), chk_parameter_type().	

7.2.4 read_parameter()

This is called every time a parameter value has to be read from the controller. Each parameter value is stored in an addressable register in the controller and make take anywhere from one to four bytes of memory. (in the register). When, read, the value is in controller counts and has to be converted to meaningful engineering units using one of six conversion equations (depending on the type of parameter being read). This is done by calling the routine read_eqn(). The converted value is stored at the appropriate index in the array PARAMETERS[] as a double precision floating point number.

Header file: Params.h, Run.h.

Data file: None.

Global Var: None.]

Local Var: chn, b[5], structure PARS.

chn	current channel being addressed.
b[]	(see 7.2. for details).

PARS is an example of structure param (see 7.2 for definition of param).

Called by: scr_setup(), limits(), chk_parameter_type(), switch_fdbks(), run_dcl(), check_chn_faults().

Calls: read_eqn(), blocks().

7.2.5 read_eqn()

This is called by read_parameter() to convert between controller counts and engineering units. Each parameter uses one of six conversion equations and its individual scaling factor along with a preset slope and offset value (fixed when tuning the channel) to do this (see appendix b for a listing of all the parameters, their scaling factors, equation types, etc.).

The six conversion equations are:

1. V# = U#*scale

2. V# = scale/U#

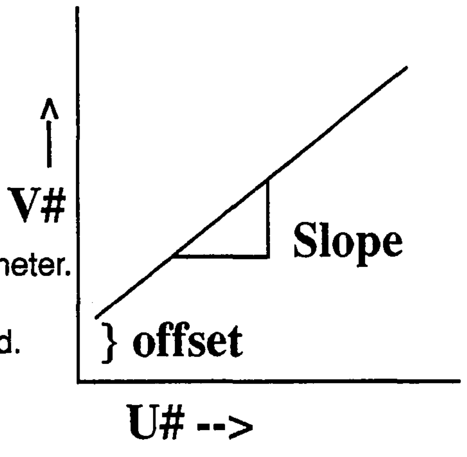
3. V# = U#

4. V# = (U#*scale - offset) * slope

5. V# = U# * slope.

6. V# = U# * scale * slope

where	V# = engineering units
	U# = controller counts
	scale = scaling factor for that parameter.
	offset = offset from zero when tuned.
	slope = slope of tuned signal.



Tuned Signal - Figure 7.2

Header file: Params.h.

Data file: None.

Global Var: offset, slope (defined in params.h)]

Local Var: eqn_id, scal, u.

eqn_id	the no. of conversion equation used by that parameter.
scal	scaling factor of that parameter.
u	counts read from the controller.

Called by: read_parameter.

Calls: None.

7.2.6 write_parameter()

This is called every time a parameter value needs to be written to the controller. Each value has to be first converted from engineering units to controller counts using write_eqn() and sent as bytes of data. If the counts occupy more than n2 bytes two registers (consecutive numbering) need to be addressed with low and high words (1 word = 2 bytes). The data bytes are sent a pair at a time using blocks().

Header file: Params.h, Run.h.

Data file: None.

Global Var: None.]

Local Var: chn, b[5], structure PARS.

chn	current channel being addressed.
b[]	(see 7.2. for details).

PARS is an example of structure param (see 7.2 for definition of param).

Called by: scr_setup(), limits(), chk_parameter_type(), switch_fdbks(), run_dcl(), check_chn_faults().

Calls: write_eqn(), blocks().

7.2.7 write_equation()

Called by write-parameter to convert user entered value from engineering units to controller counts using one of the six conversion equations (see 7.2.6 for equations)

Header file: Params.h.

Data file: None.

Global Var: offset, slope (defined in params.h)]

Local Var: eqn_id, scal, v.

eqn_id	the no. of conversion equation used by that parameter.
scal	scaling factor of that parameter.
v	user input value in engineering. units.

Called by: read_parameter.

Calls: None.

7.2.8 blocks()

This is the routine which performs the actual read/write to the controller. It takes five bytes of data at a time and writes it to the output port. The five bytes are:

	READING	WRITING
BYTE 1	240+channel#	240+channel#
BYTE 2	155	Param. register#
BYTE 3	Param. register#	Most Significant Data Byte (Upper)
BYTE 4	0	Least Significant Data Byte (Lower)
BYTE 5	checksum	checksum (of the first 4 bytes)

For every block of data written to the controller, except for the *warm boot dcl* command*, the controller sends back a confirming block; if the checksum of this confirming block is *NOT equal* to the checksum (BYTE 5) of the written data block, there is a malfunction in the communication channel (usually the program is writing out data faster than the controller can receive it). In such a case, re-initialize the communication port by restarting the program and if the problem still persists, check the UART in the outport port. For writing a parameter value, only the checksum verification is important; for reading, the parameter value is stored in BYTES 3 and 4 of the confirming block and is interpreted by read_parameter()

Header file:	Async.h.
Data file:	None.
Global Var:	None.]
Local Var:	b[5], IN[5].

b[] (see 7.2. for details).

IN[] array of five bytes to returned by the controller.

b[1] 128+ channel# (e.g for chnl1 = 129).

b[2]	parameter register#.
b[3])	Data MSB.
b[4]	Data LSB.
b[5]	checksum.

Called by: scr_setup(), limits(), chk_parameter_type(), switch_fdbks(), run_dcl(), check_chn_faults(), read_parameter(), write_parameter().

Calls: None.

7.3 Limits()

This routine is responsible for drawing and controlling screen 2 of the Setup module. It is called from scr_setup() when function key F4 is pressed and ends when the same key is pressed in screen 2 (scr_setup() is called at exit). The three monitored signals, the feedback, monitor and error, along with their current limit modes, pio functions, status and thresholds are displayed on this screen (2). The present values of the three are constantly updated and browsing through the limit modes, pio functions, status and thresholds is permitted using the *UP* and *DOWN* arrow keys. As in scr_setup(), different specialized routines are called up based on the keys pressed by the user. Again, numerical parameters (pio, hi and lo thresholds) can be modified directly from the screen while non-numerical (abort modes, threshold status) can be set by pressing the first letter of the desired configuration. (see 3.5 for the choices).

Function key assignments:

F1: return.

F2: Switch_fdbks()

F4: Toggle to Screen 1, scr_setup().

Header file: Async.h, Params.h, Run.h.

Data file: None.

Global Var: Arrays of Structures ***PARAMETERS[32], Fbk[3].fbkval[6],***

MONITOR, b[5].

PARAMETERS[] - See 7.2 for details.

Fbk[] is an array of structure Feedback which contains an array ***fbkval[6]*** with six examples of structure param (see 7.2 for definition) as data fields:

fbkval[1]	struct param mode;	(abort mode parameter)
fbkval[2]	struct param pio;	(pio parameter)
fbkval[3]	struct param hi;	(threshold high limit parameter)
fbkval[4]	struct param lo;	(threshold low limit parameter)
fbkval[5]	struct param hi_stat;	(threshold high limit status parameter)
fbkval[6]	struct param lo_stat;	(threshold low limit status parameter)

The above fields are defined for each of the 3 signals-feedback, monitor & error. To reference a field (e.g. value) from any one of the above, the naming convention is:

fbk[control signal# 0-2].fbkval[param# 0-5]->value.

MONITOR is an example of struct param. It holds information about the non-controlling feedback (See 7.2 for details on struct param).

b[] - See 7.2 for details.

Local Var: Chnl, Keyb, Curr, Prev.

Chnl	current channel being addressed
Keyb	last key pressed
Curr	current cursor (at parameter value) position
Prev	last cursor position

Called by: scr_setup().

Calls: chk_fbk_limits(), switch_fdbks(), read_parameter(), blocks(), write_parameter(), pio_write(), pio_read().

7.3.1 chk_fbk_limits()

This is called from screen 2 every time a parameter pertaining to a feedback or error signal is read from or written to the controller. It is identical in function to chk_parameter_type() in screen 1.This routine takes each parameter value, prints it in the set position on the screen if it s numerical or, if non-numerical, converts the control word into a string understandable to the user.

For example, if the threshold value of the error signal is to be printed, whatever is held in fbk[2].fbkval[2]->value will be printed as a floating point number. To print the current abort mode of the error signal, however, the value in fbk[2].fbkval[0]->value is taken and a lookup is done to match it to the correct mode (e.g. Return & Hold = 4) and the matched string is printed (in this case, "Rtrn & Hold" is printed in the screen position for abort modes). If a fault occurs on a particular signal, this routine conveys that information to the user by typing "FAULT" at the top of the screen and also writing out to the pio if a pio function has been set for that signal.

Header file: Async.h, Params.h, Run.h.

Data file: None.

Global Var: fbk[].fbkval[]. (see 7.2 for details).

Local Var: posn.

	posn	index for the array fbk[].fbkval[].

Called by: limits().

Calls: pio_write().

7.3.2 switch_fdbks()

This is called from screen 2 when function key F2 is pressed. This allows the user to switch controlling feedbacks from force to motion control and vice versa. The routine first checks if the controller is in process (i.e. executing a waveform) by checking the status parameter (PARAMETERS[3])- if running, it alerts the user with a beep and does not allow switching. If the controller status is complete, the following sequence is executed. All abort modes are turned off. The controller is set in DIRECT waveform mode. The bias value (PARAMETERS[25])and the average servo out value (controller register# 204)are read and

their difference (servo out - bias) is sent out to the open loop (controller register # ≈ 155) so that the controller does not go full stroke when feedbacks are switched. The loop is then opened (no feedback) and the feedbacks switched. The loop is closed and a backup file is then opened which contains the default parameter values for the new feedback using read_saved_setup(). The parameters are read from the file and written to the controller. The routine the n exits to screen 1. Note that abort modes and thresholds have been reset at this point.

Header file:	Async.h, Params.h, Run.h.	
Data file:	CHN*(chn#)(fdbk#)*.IST (see 7.2 for file naming details)	
Global Var:	fbk[].fbkval[], PARAMETERS[]. (see 7.2 for details).	
Local Var:	chn, b[5],.	
	chn	current channel being addressed.
	b[]	(see 7.2. for details).
Called by:	limits().	
Calls:	read_parameter(),write_parameter(), blocks(), read_saved_setup().	

CHAPTER 8

RUN

8.1 Introduction

This chapter describes the routines which are used for the constant amplitude fatigue test. The main controlling routine is run_dcl() (see Figure 8.1).

8.2 Run_dcl()

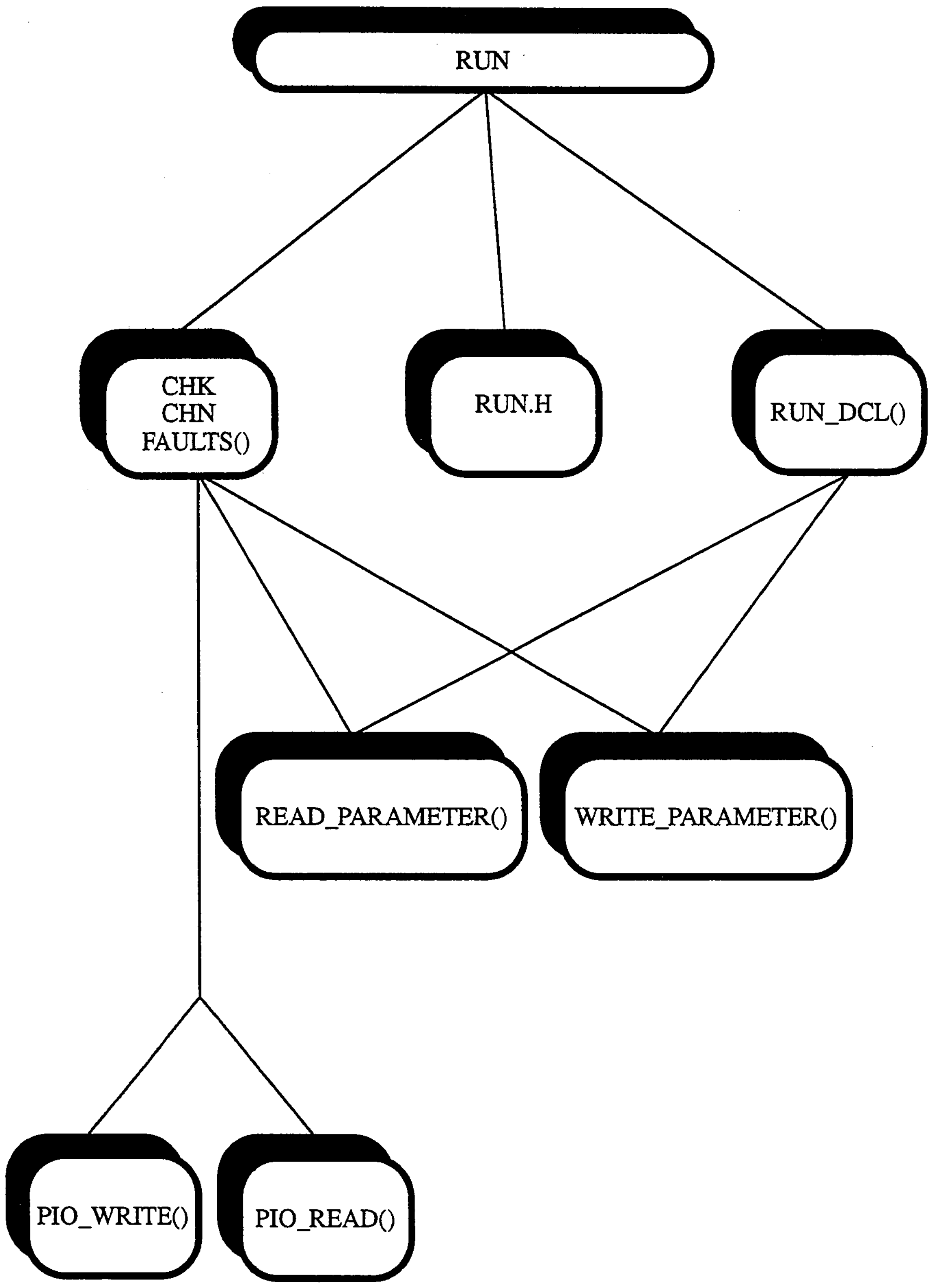
This routine starts up with a menu outlining four tasks for the test:

Run	(running the controllers as a group)
Specify	(set which of the channels from 1 to 16 are to be used)
Setup	(setup individual channel parameters - scr_setup())
Exit	(go to main menu).

Based on the user's selection, the program goes into different run screens. **Specify** pops up run screen 1 with t he list of valid channels and their run status (active or off). **Setup** calls scr_setup and screen 1 of the current channel is displayed. **Exit** draws the main menu. **Run** brings up the fatigue test screen (run screen 2). It is an information screen with the active channels, their current count values, pio functions and preset abort modes.

Run Screen 1: This is used to decide which actuators will be controlled by the test. up to 16 actuators can be active. The user specifies whether a given channel is active (A) or off (O) by pressing the appropriate key. As with all menus, the ENTER key is to be pressed to confirm the selection. If a channel is set as ACTIVE, it is added to the end of a linked list, ACTIVE_CHN[] along with other useful information about it. Browsing is permitted in the up or down direction using the arrow keys of the same name.

Run Screen 2: This is the main display screen for the test. The current active channels, their current counts, pio functions and abort modes are displayed. If the test is running, the counts are recovered from a backup file, COUNT.IFT, and the current cycles added on to this read value. The function key assignments are:



Run Module-Figure 8.1

F1: Exit to main

F2: Modify counts - if the user wishes the counts to start from a number other than zero or the saved counts.

F3: Start /Stop test - toggle.

If no key is pressed, the program sends sets of 100 sinewaves to each of the active controllers and continuously updates the counts on the screen. Faults for all active channels are continuously monitored using check_chn_faults().

Header file: Async.h, Params.h, Run.h.

Data file: COUNT.!FT (The current active channels and their counts are backed up in this file anytime the user steps out of run screen 2 when the test is running.)

Global Var: Array ACTIVE_CHN[16], fbk[].fbkval[], PARAMETERS[], RUNNING.

ACTIVE_CHN[]-array of structure ***chn_arr***, holds run screen2 information

Structure chn_arr (and each index of ACTIVE_CHN[]) has the data fields:

boolean ch;	(channel ACTIVE or OFF)
long int count;	(current count value)
int oldct;	(last count when stepping out of run screen2)
int set;	(no of sine cycles to set at a time-100,500,etc)
int pio;	(set pio function number)
int xpos;	(row position on run screen2)
int ypos;	(column position on run screen 2)
char fmode;	(fault modes)
int next;	(number of next active channel in linked list)

RUNNING-boolean variable set when test starts running (F3)

fbk[].fbkval[], PARAMETERS[]. (see 5.2 for details).

Local Var:	Chnl, Keyb, Curr, Prev.	
	Chnl	current channel being addressed
	Keyb	last key pressed
	Curr	current cursor (at channel #) position
	Prev	last cursor position

Called by: main().

Calls: read_parameter(), write_parameter(), check_chn_faults(), pio_write().

8.2.1 check_chn_faults()

This is called by both Setup and Run to check for any faults in the current active channels. For each channel, it checks if any of the limit thresholds have been exceeded and reports the fault along with the channel number to the current display screen. It also writes out to the valid pio function.

Header file:	Async.h, Params.h, Run.h.
Data file:	None.
Global Var:	fbk[].fbkval[], ACTIVE_CHN[] (see 6.1 for details).
Local Var:	faulty.
faulty	the channel number of channel where the fault has occurred.
Called by:	run_dcl(), scr_setup().
Calls:	blocks(), pio_write().

CHAPTER 9

GRAPHICS

9.1 Introduction

This chapter describes the routines that have to do with drawing menus, borders and boxes, changing the background and foreground colors, writing to specific portions of the screens and all other screen entity manipulations. See Figure 9.1 for a delineation of the different graphical routines and their hierarchy.

9.2 Initialize()

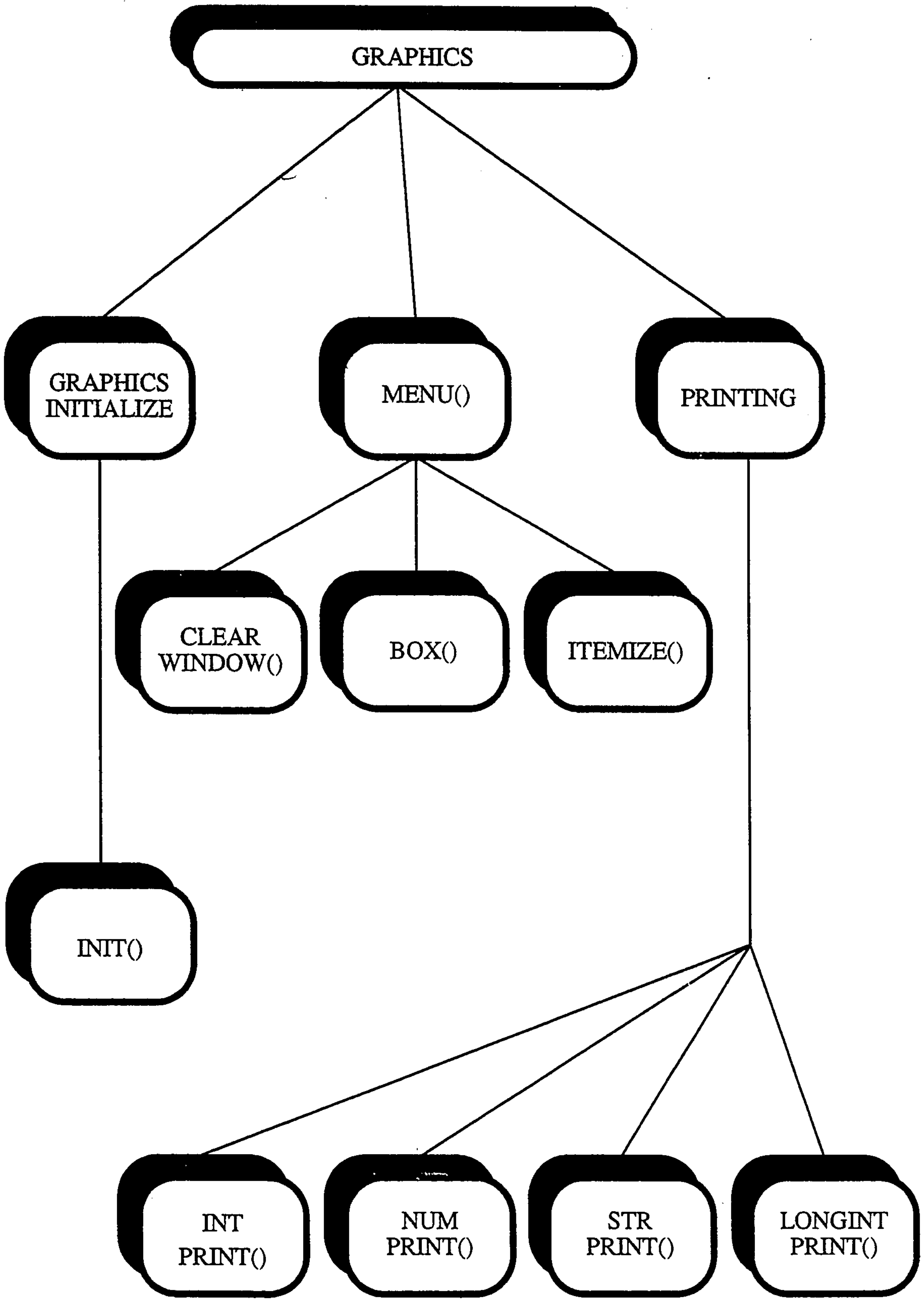
This initializes the screen for graphics mode using a C function initgraph(). It detects the graphics hardware and drivers and selects the screen resolution on the basis

of that. It also checks for the number of colors available.

Header file:	C library file Graphics.h.
Data file:	None.
Global Var:	None.
Local Var:	None.
Called by:	Main().
Calls:	C function initgraph().

9.3 Menu()

This draws the menu outlines and places the choices within. It also returns the user selection from the choices. The menus are scaled according to the size and number of the choices using Itemize(). All menus have the same foreground and background color as defined in a structure mnuAtr which holds all the graphical attributes of the menus. The outline is drawn by calling Box(). This is used to display the main, setup and run menus.



Graphics Module-Figure 9.1

Header file: Async.h, run.h

Data file: None.

Global Var: struct menus.

menus is an example of structure mnuAtr with the following fields:

int fgNormal;	(foreground color for unselected menu choices)
int fgSelect;	(foreground color for selected menu choice)
int fgBorder;	(foreground border color)
long bgNormal;	(background color for unselected menu choice)
long bgSelect;	(background color for selected menu choice)
long bgBorder;	(background border color)
int centered;	(set if menus to be centered on screen)
char nw, ne, se, sw;	(menu outlines constructed from these strings)
char ns, ew;	(menu outlines)

Local Var: char items, xcol, ycol, prev, curr, keyb.

items	array holding text of the menu choices.
xcol	starting x position for the menu to be drawn onscreen
ycol	starting y position for the menu to be drawn onscreen
Keyb	last key pressed
Curr	current cursor (at channel #) position
Prev	last cursor position

Called by: main(), run_dcl(), scr_setup().

Calls: Box(), Itemize(), clear_window().

9.3.1 box()

This draws the menu outline. It uses the two-character strings from structure menus - nw, ne, se, sw, ns, ew which are actually directional lines (ns = “|”, ew = “--”,ne = “|‾”, nw= “‾|”,se = “|_”, sw= “_|”) and constructs a rectangle of height = number of menu choices and width = length of the longest menu choice.

Header file: Async.h, Run.h.

Data file: None.

Global Var: structure menus (see 9.2 for details).

Local Var: xrow, ycol, num, max.

xrow	starting x position for the outline to be drawn onscreen
ycol	starting y position for the outline to be drawn onscreen
hi	number of menu choices.
wid	maximum text length of the longest menu choice.

Called by: menu(), run_dcl(), scr_setup().

Calls: None.

9.3.2 itemize()

This takes each item to be printed in the menu and writes it to the appropriate screen position. All choices have the same length = length of the longest menu choice; the shorter choices are padded with blanks.

Header file: Async.h, Run.h.

Data file: None.

Global Var: structure menus (see 9.2 for details).

Local Var: xrow, ycol, str, max.

xrow	screen x position for the choice text to be written to.
ycol	screen y position for the choice text to be written to.
str	choice text.
len	text length of the current menu choice.

Called by: menu(), run_dcl(), scr_setup().

Calls: None.

9.3.3 clear_window()

This clears the current text window and resets the screen colors to default. It calls a C function, clrscr().

Header file: None.

Data file: None.

Global Var: structure menus (see 9.2 for details).

Local Var: None.

Called by: menu(), run_dcl(), scr_setup().

Calls: C function clrscr().

9.3.4 cursor()

This is a hardware level routine which is used to turn the cursor off or on (blinking). It writes a control value to the 6845 CRT controller in the Z-409 video card, using a programmable video interrupt (INT 10H). The interrupt can be used to program 16 different video I/O functions, based on the value in register AH. The one function used to turn the cursor OFF is function code 1 which sets the cursor size on the basis of the value in registers CH & CL. Bits 0-4 of register CH represent the starting scan line number of the cursor while bits 0-4 in CL, the ending scan line number. Values for these usually range from 0-7 (0x00-0x07). If the starting scan line value is greater than the ending scan line value, the cursor is turned off.[4]

Header file:	None.
Data file:	None.
Global Var:	registers AX, CX.
Local Var:	None.
Called by:	menu(), run_dcl(), scr_setup().
Calls:	C function int86().

9.4 Printing

The next few routines deal solely with printing integers, real numbers and strings on to set locations in the screen text window.

9.4.1 intprint()

Prints an integer on to the screen. Takes a row and column position nad prints up to six digits of a given value. If the value is less than six digits, it pads the screen with blank spaces.

Header file:	None.
Data file:	None.
Global Var:	None.

Local Var: fgcolr, bgcolr, x, y, value.

fgcolr	foreground color of text to be printed.
bgcolr	background color of text to be printed.
x	screen x position for the choice text to be written to.
y	screen y position for the choice text to be written to.
value	integer value to be printed.

Called by: menu(), run_dcl(), scr_setup().

Calls: Cursor().

9.4.2 numprint()

Prints a real number on to the screen. Takes a row and column position and prints up to eight digits of a given value. If the value is less than eight digits, it pads the screen with blank spaces.

Header file: None.

Data file: None.

Global Var: None.

Local Var: fgcolr, bgcolr, x, y, value.

fgcolr	foreground color of text to be printed.
bgcolr	background color of text to be printed.
x	screen x position for the choice text to be written to.
y	screen y position for the choice text to be written to.
value	floating point value to be printed.

Called by: menu(), run_dcl(), scr_setup().

Calls: Cursor().

9.4.3 longprint()

Prints a long integer on to the screen. Takes a row and column position and prints all the digits of a given value. If the value is less than six digits, it pads the screen with

blank spaces.

Header file: None.

Data file: None.

Global Var: None.

Local Var: fgcolr, bgcolr, x, y, value.

fgcolr	foreground color of text to be printed.
bgcolr	background color of text to be printed.
x	screen x position for the choice text to be written to.
y	screen y position for the choice text to be written to.
value	long integer value to be printed.

Called by: menu(), run_dcl(), scr_setup().

Calls: Cursor().

9.4.4 strprint()

Prints a string value on to the screen. Takes a row and column position and prints the string as text.

Header file: None.

Data file: None.

Global Var: None.

Local Var:	fgcolr, bgcolr, x, y, str.	
	fgcolr	foreground color of text to be printed.
	bgcolr	background color of text to be printed.
	x	screen x position for the choice text to be written to.
	y	screen y position for the choice text to be written to.
	str	text to be printed.

Called by:	menu(), run_dcl(), scr_setup().
Calls:	Cursor().

CHAPTER 10

FUTURE WORK

10.1 Introduction

This chapter details some of the tasks still to be worked on. A Brief outline of the effort involved and suggestions on how to proceed with these tasks is also given.

10.2 Tasks Ahead

1. **Include a calibration utility as part of the setup module.**

This allows for setting up minimum and maximum amplitudes (force or displacement) for the actuators which are different from the physical values. Essentially, the signal will be offset from zero by some amount.

Coding this utility might involve the displaying of a screen with the tuned signal (see Figure 7.3) along with the current minmum and maximum values in both engineering units and controller counts. These should be user-configurable and the user-adjusted value should be conveyed to the controller and the tuned signal correspondingly updated to show the offset from zero.

2. **Porting the software to a SUN workstation.**

Running the software on a SUN workstation would not only enhance the speed and performance of the software, but it would also allow for multiple tests to be run from the same computer. This would be a distinct advantage from the point of view of safety as tests could be remote-controlled from another location without having to visit the testing floor during a test.

Tasks involved in transferring to a SUN workstation include:

(1) Confirming that the low-level communication routines between host computer and controller work.

(2) Re-writing the graphical routines (see Chapter 9) for displaying as windows on the screen.

(3) Verifying that the C library functions and interrupts used in the software are available on the SUN. If not, the nreplacing them with their equivalents.

REFERENCES

1. White D., "Servo Tuning with Five Mode Controllers" MOTION, September/October 1988, pp 14-23.

2. Vickers Xpert DCL User's Manual, pp 17-20.

3. TS&S Software User's Manual, pp 3.1 - 3.16

4. Zenith Z-200 PC Series Technical Reference Manual, pp 9.2 -11.7

5. Burr-Brown PCI-20000 System User's Manual, pp 3-5.

6. INTEL Peripheral Components Manual (1986).

7. Borland Turbo C Reference Manual & User's Guide.

APPENDIX A

THE INTEL 8250 UART

(UNIVERSAL ASYNCHRONOUS RECIEVER/TRANSMITTER)

This appendix describes the 8250 UART which is a standard chip used for serial communication in a PC. The rest of the pages of this appendix are condensed from the INTEL Peripheral Components Manual[6]. They outline the chip's function, its internal hardware, its various operating modes and the special issues involved in programming them.

8250 ORGANIZATION

Here we shall examine carefully the 8250 *asynchronous communications element*, which is the controller chip of the adapter of Fig. 10-3. The IC of the 40-pin DIP is fabricated with NMOS technology. With associated circuits, it provides the interface between the parallel data bus of the computer and the serial data lines of the modem. Parallel-to-serial conversion of data is implemented by a shift register for transmission to the modem. A second shift register provides serial-to-parallel conversion when receiving from the modem.

The 8250 is commonly referred to as a *universal asynchronous receiver/transmitter (UART)*. Some communications chips are designed for control of either synchronous or asynchronous transfers and these are called *universal synchronous/asynchronous receiver/transmitter (USART)*.

Registers

The 10 addressable byte registers of the 8250 require 9 port numbers for input (READ) and 8 for output (WRITE). Because the three pins dedicated to internal addressing provide only eight combinations, the MSB of the line control register is used to distinguish between ports with the same numbers. This bit is called the *divisor latch access bit (DLAB)* because it must be set in order to read or write the two divisor latches with port numbers 3F8H and 3F9H. When DLAB is 0, addresses 3F8H and 3F9H relate to other registers. Whenever an address that depends on the state of DLAB is given, it will be followed by the state in parentheses. Figure 10-8 lists the 10 registers and their hexadecimal addresses.

Block Diagram

Figure 10-9 shows a block diagram of the 8250 element, with most of the internal control lines omitted. Immediately above each addressable register is the address including in parentheses, where appropriate, the state of the DLAB. With DLAB low, address 3F8H applies to either the receiver buffer or the transmitter buffer register, depending on whether the operation is READ or WRITE, and with DLAB

Address	Register	Read/Write
3F8 (0)	Tx buffer	Write only
3F8 (0)	Rx buffer	Read only
3F8 (1)	Divisor latch LSB	Read/write
3F9 (1)	Divisor latch MSB	Read/write
3F9 (0)	Interrupt enable	Read/write
3FA	Interrupt ID	Read only
3FB	Line control	Read/write
3FC	Modem control	Read/write
3FD	Line status	Read/write
3FE	Modem status	Read/write

FIGURE 10-8. 8250 registers and addresses.

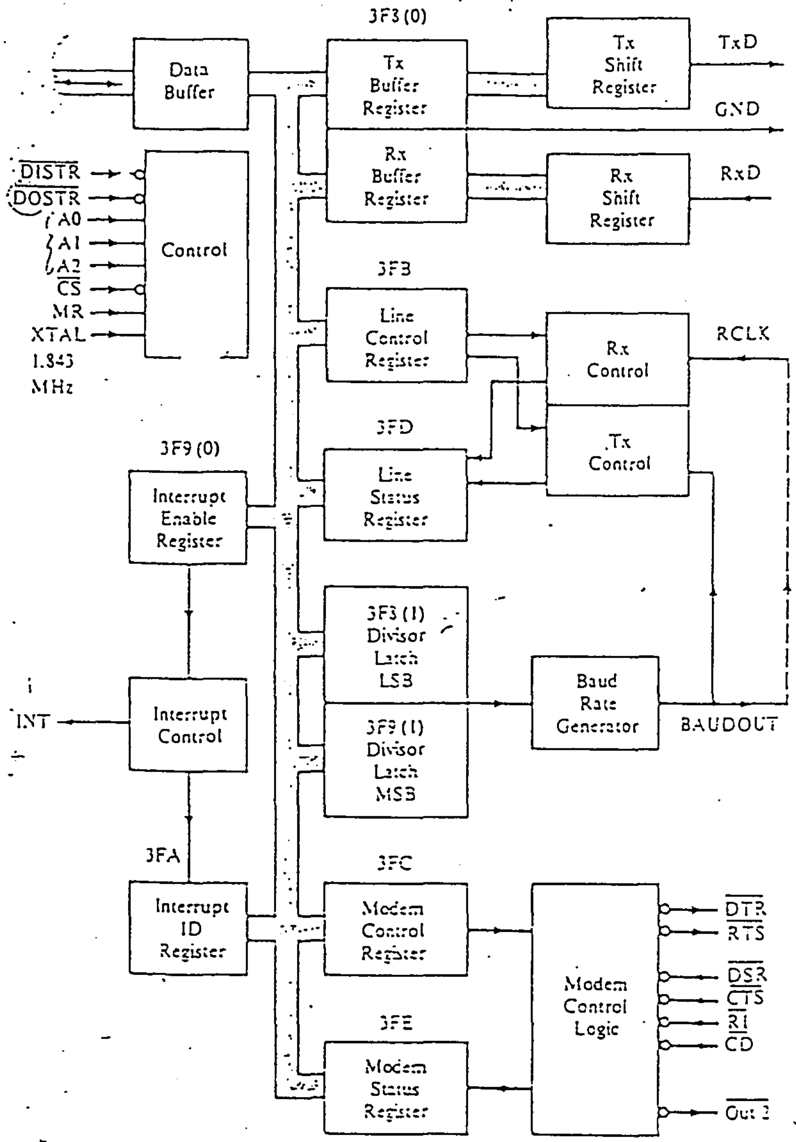


FIGURE 10-9. Block diagram of the INS 8250 asynchronous communications element.

high, the address accesses the low byte of the divisor latch. Address 3F9H is used for the interrupt enable register when DLAB is low, but when the bit is high, the most significant byte of the divisor latch is addressed.

In the block diagram 29 pins are indicated, including the 8 pins of the data bus. In addition, two pins provide the 5-V supply, and the other nine are connected directly to either a logical 0 or a logical 1, or simply not connected.

Connected internally to the transmitter control block is the BAUDOUT line from the *baud rate generator*, which provides a pulse train to the transmitter clock. The vertical dashed line at the right is an external connection of BAUDOUT to the receiver clock (RCLK). The connection makes the bit rate for reception the same as that for transmission, which is the usual condition. The actual bit rate equals the frequency of the pulse train of BAUDOUT divided by 16.

The XTAL input to the control block is a 1.8432-MHz pulse train. This pu train is generated from a circuit having a crystal oscillator that has 10 times t frequency, with the output of the oscillator passed through a divide-by-10 netwo The frequency of BAUDOUT is that of the XTAL input divided by the numbe the two divisor latches.

Baud Rate Generator

Figure 10-9 shows a baud rate generator with an input from the two divisor latch In addition, the generator receives the 1.8432-MHz pulse train from the XT input. Division of this frequency by the 16-bit number of the divisor latc gives the frequency of the pulse train of BAUDOUT. Because the frequency BAUDOUT also equals 16*bps, it follows that the bit rate is

Bit rate = 1,843,200 / (16 * divisor) bps

For example, if the divisor number is 0060H (96 decimal), the bit rate is 1 bps. The divisor must be written into the latches during the initialization of chip prior to communications. The maximum allowed bit rate is 9600 bps.

TxD and RxD Buffer Registers

When a character is written to the *transmitter buffer register,* it is held until *transmitter shift register* has shifted out serially the last character sent to it. T the character of the buffer is moved to the shift register, and the buffer becom empty. Writing to it when it is not empty replaces the prior character, which sho be avoided, of course. The LSB is data bit 0, and it is the first one shifted to communications line. Because the buffer holds a character until it can be tr mitted, it is often called the *transmitter holding register.*

The *receiver buffer register* contains the last character moved to it from *receiver shift register.* Its LSB is the first one that serially entered the shift regi from the communications line. Unless the character is read before the next on received, it is destroyed, which is the *overrun* error. Both the transmit and rece buffer registers have address 3F8H, with DLAB zero. There is no conflict, beca one is an input port and the other is an output port. Because the transmitter a receiver have buffer registers in addition to their shift registers, they are said to *double buffered.*

Line Control Register

The format of the asynchronous data word is specified by writing to the *line con register.* Also, the state of DLAB is defined by bit 7. The contents and encod are shown in Fig. 10-10. The port number is 3FBH, and both writing and read are allowed.

The *set break* bit 6 of the line control register is set when it is desired to a a distant computer or terminal for attention. It simply changes the output line fr the normal mark state to the spacing state. Set break is disabled by writing a 0 bit 6.

7	6	5	4	3	2	1	0
DLAB	Set Break	Stick Parity	Parity Select	Parity EN	Stop Bits	Word Length	

Encoding:

Bit 6:	When 1, the TxD output line is forced to the space state and remains there as long as bit 6 is 1.
Bit 5:	When 1 with bit 3 also 1, the parity bit is 0 if bit 4 is 1, and it is 1 if bit 4 is 0.
Bit 4:	Even parity = 1; odd parity = 0.
Bit 3:	Parity enable = 1; no parity bit = 0.
Bit 2:	Two stop bits = 1 (for 6- to 8-bit words); one stop bit = 0.
Bits 1, 0:	5 bits = 00; 6 bits = 01; 7 bits = 10; 8 bits = 11.

FIGURE 10-10. Line control register.

Three of the bits relate to parity. If bit 3 is 0, there is no parity bit, and bits 4 and 5 are don't-cares. However, if bit 3 is 1, there is a parity bit, with choices of even, odd, 1, or 0. The selection depends on the states of bits 4 and 5. If bit 5 is 0, the parity is either even or odd, as specified by bit 4. Even parity is most often used.

However, with parity enabled and bit 5 set, the parity bit is always 0 if bit 4 is 1, but it is always 1 if bit 4 is 0. Some systems with echoplex do not allow parity error checking and may require a mark parity or perhaps a space parity.

Bit 2 is used to specify either 1 or 2 stop bits for words having lengths of 6 to 8 bits. However, 5-bit words can have only 1 or 1.5 stop bits, and in this special case, a value of 1 for bit 2 gives 1.5 stop bits. Most common is 1 stop bit.

Each transmitted or received serial character can have a specified length of either 5, 6, 7, or 8 bits. Usually, text characters have 7 bits and binary codes have 8 bits. Encoding of bits 1 and 0 specifies the word length.

The proper data must be written into the line control register before communication is possible. Once the word format has been set, it is not necessary to write to the register again, or to read it, except to set or clear DLAB. The instruction sequences that follow will set or clear the bit without changing the other bits of the line control register:

```
MOV DX, 3FBH        MOV DX, 3FBH
IN  AL, DX          IN  AL, DX
OR  AL, 80H         AND AL, 7FH
OUT DX, AL          OUT DX, AL
```

Modem Control Register

Port 3FCH accesses the read/write *modem control register*, which controls the interface with the data set. Indicated in Fig. 10-11 are its contents. Bits 3 and 2 control the outputs of pins $\overline{OUT2}$ and $\overline{OUT1}$, which are available for control as desired. In the PC, an active $\overline{OUT2}$ enables the output buffer of the IRQ4 interrupt, and $\overline{OUT1}$ is not used.

In the full-duplex mode, bits 0 and 1 are written with 1s, thereby turning ON the data-terminal-ready and request-to-send signals of output pins DTR and RTS. These remain ON at all times during communications. When half-duplex is used, the RTS bit is set when the computer wants to send data, but once the transmission has ended, it is reset to return the modem to the receive mode. This operation is called *line turnaround*.

7	6	5	4	3	2	1	0
0	0	0	Loop	OUT2	OUT1	RTS	DTR

Encoding:

Bits, 7, 6, 5:	Always 0.
Bit 4:	When 1, the output of the transmitter shift register is looped back into the receiver shift register.
Bit 3:	The complement of the bit goes to pin $\overline{OUT2}$.
Bit 2:	The complement of the bit goes to pin $\overline{OUT1}$.
Bit 1:	The complement of the bit goes to pin $\overline{RTS}$.
Bit 0:	The complement of the bit goes to pin $\overline{DTR}$.

FIGURE 10-11. Modem control register.

To enable interrupt IRQ4 from the 8250, bit 3 must be set. Resetting th disables the interrupt line out of the buffer.

Bit 4 provides a loopback feature for diagnostic testing of the 8250. For no communications, the bit is reset. When it is set, however, the output of the tr mitter shift register is fed internally into the receiver shift register, so that a w written to the transmitter buffer register returns immediately via the receiver b register. Transmitting a character and then comparing it with the one rece provides a way of verifying the transmit and receive data paths of the chip.

When the loopback feature is activated, the serial output line TxD is set to mark (1) state, and the serial input line RxD and inputs $\overline{DSR}$, $\overline{CTS}$, $\overline{RI}$, and from the modem are disconnected. In addition, outputs $\overline{DTR}$, $\overline{RTS}$, $\overline{OUT1}$, $\overline{OUT2}$ are internally connected to respective inputs $\overline{DSR}$, $\overline{CTS}$, $\overline{RI}$, and $\overline{CD}$ the diagnostic mode, all interrupts are operational and can be tested.

The Divisor Latches

Port 3F8H is that of the least significant byte (LSB) of the *divisor latch*, and 3F9H is that of the MSB. The DLAB bit must be set prior to addressing, and reading and writing are allowed. The latches are programmed prior to comm cations to give the desired bit rate, which is 1,843,200 divided by the produ 16 and the divisor.

Line Status Register

The read/write *line status register* with address 3FDH provides status inform concerning data transfer. Its contents are indicated in Fig. 10-12. Bit 7 is al 0, and bit 6 is read-only, not affected by write operations.

Whenever a character is moved from the transmitter buffer register to the tr mitter shift register, bit 6 is reset. It is set only after the character has been se shifted out. Thus, state 1 signifies an empty shift register.

State 1 in the bit 5 position indicates that the transmitter buffer (holding) reg is empty and ready to accept a new word for transmission. *This bit should alw be examined prior to writing to the Tx buffer register in order to avoid wr over a character that is waiting to be transmitted.* In addition, when the bit is the output interrupt pin is active if bit 1 of the interrupt enable register is set.

The *break-interrupt (BI)* bit 4 is set when the RxD input line is held in the sp state for longer than a full word transmission time. When set, it generates

7	6	5	4	3	2	1	0
0	TSRE	THRE	BI	FE	PE	OE	DR

Encoding:

Bit 6:	Tx shift register empty (TSRE) indicated by 1.
Bit 5:	Tx holding register empty (THRE) indicated by 1.
Bit 4:	Break interrupt (BI) indicator.
Bit 3:	Framing error (FE) indicator (no stop bit).
Bit 2:	Parity error (PE) indicator
Bit 1:	Overrun error (OE) indicator
Bit 0:	Receiver data ready (DR) indicator.

FIGURE 10-12. Line status register.

interrupt, provided bit 2 of the interrupt-enable register is set. Recall that bit 6 of the line control register can be set to force the serial output line to the space state to alert another computer. The BI bit of the line status register is the detector of such a signal on the serial input line. BI is reset when the serial input line returns to the normal mark condition.

Bits 3, 2, and 1 are error indicators. A *framing error (FE)* occurs when a received character is without a stop bit, in which case the character is improperly framed. A *parity error (PE)* is detected when the parity is not the one specified. and an *overrun error (OE)* results when a character is transferred into the receiver buffer register before the last one was read. Each of these errors causes the corresponding bit of the line status register to be set, and each generates an interrupt if bit 2 of the interrupt-enable register is set. Reading the line status register clears the error indicators.

The *data-ready (DR)* bit 0 is set whenever a complete incoming character has been received in the receiver buffer register. It is reset when the character is read by the CPU, and it can also be reset by writing to the line status register. Bit DR should always be examined before reading the receiver buffer register in order to determine if a valid character is present. When bit 0 is set, an interrupt is generated provided bit 0 of the interrupt-enable register is set.

Modem Status Register

Port 3FEH is that of the read/write *modem status register*. The current state of each of the control lines CTS, DSR, RI, and CD from the modem to the CPU can be read from a bit of the high nibble of this register.

In addition, each bit of the low nibble of the register reveals whether or not the corresponding input signal *has changed state since the last reading of the modem status register*. Whenever a control input changes state, its associated low-nibble bit is set, and whenever the modem status register is read, it is reset. Also, a bit that is set generates an interrupt if bit 3 of the interrupt enable register is set. The contents of the modem status register are shown in Fig. 10-13.

7	6	5	4	3	2	1	0
CD	RI	DSR	CTS	Delta CD	Delta RI	Delta DSR	Delta CTS

FIGURE 10-13. Modem status register.

Status bits 7-4 of the figure are the complements of the bits received at the input pins. The word *Delta* denotes a change. For the bit 2 ring indicator, the Delta RI bit is set only when the RI input to the chip goes from ON to OFF, and it is reset when the register is read.

Prior to a transmission, the modem status register should be read to ascertain that the data set is ready and that a character can be sent. Bits 5 and 4 are the indicators to be examined. Also, bit 5 of the line status register should be checked to determine that the transmitter buffer register is empty. *Before reading a character from the receiver buffer register, the modem status register should be read for inspection of the DSR bit.* Then the data-ready bit 0 of the line status register should be checked.

Interrupt Enable Register

Port 3F9H with DLAB zero is the address of the read/write *interrupt-enable register.* The bits of the high nibble are always 0, but those of the low nibble provide for enabling individually the four allowed types of interrupts, with a bit set to 1 for enable. The contents of the interrupt enable register are indicated at the top of Fig. 10-14, and the interrupt priorities are shown in the table of the figure. Note that the bit numbers of the table are out of order.

All interrupts are disabled by writing 00H to the register; when this is done, line IRQ4 is inactive. Interrupts can also be inhibited by a reset of the OUT2 bit (3) of the modem control register, which disables the external buffer that feeds line IRQ4. The different interrupt types can be selectively enabled, of course, by setting the OUT2 bit and writing to the interrupt enable register.

Interrupt ID Register

The read-only *interrupt identification (ID) register* with port number 3FAH has 5 high-order bits that are always 0, as shown in Fig. 10-15. Thus, only the 3 lowest bits have significance. Bit 0 is 0 whenever an interrupt is pending, which implies that pin INT is active. When it is 1, there is no pending interrupt.

The output of the INT pin of the PC is used to generate a hardware interrupt through level 4 of the 8259 controller, implementing INT 0CH in the PC. An alternate method is to poll bit 0 periodically with software to determine if an interrupt is pending. The hardware method provides a more efficient use of CPU time.

7	6	5	4	3	2	1	0
0	0	0	0	Modem Status INT EN	Line Status INT EN	Tx Buffer INT EN	Data Ready INT EN

Priority	Bit	Interrupt Type	Activated by
Highest	2	Line status	Bits 1–4, line status
Second	0	Data ready	Bit 0 of line status
Third	1	Tx buffer empty	Bit 5 of line status
Lowest	3	Modem status	Bits 0–3, modem status

FIGURE 10-14. Interrupt-enable register and interrupt priorities.

7	6	5	4	3	2	1	0
0	0	0	0	0	INT ID bits		Pending

Encoding:

Bits 2, 1:	ID of pending interrupt of highest priority.
Bit 0:	0 if an interrupt is pending: otherwise 1.

FIGURE 10-15. Interrupt ID register.

Bits 2 and 1 simply identify which pending interrupt has the highest priority. Bit values 11, 10, 01 and 00 correspond to priorities 1, 2, 3, and 4, respectively, with the priorities defined in Fig. 10-14. A reading of the interrupt ID register allows a program to determine whether or not an interrupt is pending, and if so, to ascertain the highest priority.

10-3 MODEMS

Modems can be classified by speed, housing design, type of coupling, features and intelligence, and protocol. Usually modems with bit rates below 1200 bps are considered *low speed*, from 1200 to 9600 bps is *medium speed*, and over 9600 is *high speed*. Both *stand-alone* and *expansion board* modems are available. The former is a separate unit that has LED status indicators and can be used with any system, whereas the latter requires an expansion slot and is designed for a particular type of computer.

The expansion board modem is less expensive, can be located on the board with the communications adapter circuits, is more portable, and requires neither an RS-232-C cable nor an external power connection. However, it increases the internal heat, has no visible indicators, and cannot be transferred to a dissimilar computer. Still, it is quite popular for use with microcomputers. Various classifications are shown in Fig. 10-16.

Coupling is either direct or acoustic. A *direct-coupled* modem connects directly through a cable to the wall jack of the telephone line. An *acoustic coupler* is coupled by means of sound waves. After a number is dialed, a data switch of the acoustic coupler is turned on, and the telephone handset is placed firmly into two cylindrical rubber grommets of a cradle containing a small speaker and a microphone. Data transmission is from the speaker to the telephone mouthpiece via audio tones, and reception is from the telephone receiver through sound waves to the microphone of the cradle. In general, acoustic couplers are less expensive, but they

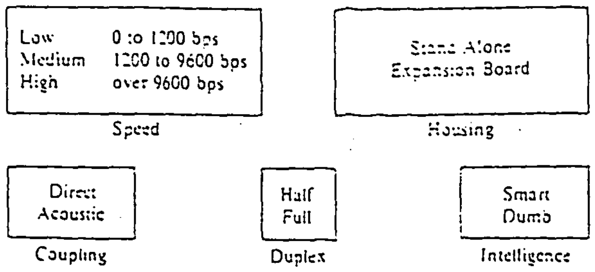


FIGURE 10-16. Various classifications of data sets.

APPENDIX B

PROGRAMMING THE BURR-BROWN PC- 20001C PIO BOARD

This appendix deals with programming of the PC20001C in more detail and also explains the memory addressing scheme for the board.

The PC20001C supports up to 32 channels of TTL-compatible parallel digital information to/from the computer. These channels maybe used to monitor/ control external devices like relays or switches. The 32 lines are arranged as four ports of eight bits each. Each port may be configured as a group of eight inputs or eight outputs.[5] The direction enabling reading and writing to the bits in the ports is dealt with in this section.

Addressing:

Address switches on the board allow it to be mapped to any 1-KByte block in the PC's 1-MByte memory space. When its base address is chosen care must be taken to select a location in memory not used by any other hardware. The base address is selected by switches 1 through 10 on the board with switch 1 = address bit 10 and switch 10 = address bit 19. A setting of OFF = bit value 1 and ON = bit value 0. The addressing scheme used by the PIO board is the same as that of the 8086 microprocessor family with a 16-bit SEGMENT register in conjunction with a 16-bit OFFSET to specify a 20-bit address as:

ADDRESS = SEGMENT * 16 + OFFSET.

The remaining pages of this appendix are taken from the Burr-Brown PCI-20001C programming manual[5].

Programming Notes

The 8086 microprocessor family addresses memory using a 16-bit SEGMENT register in conjunction with a 16-bit OFFSET to specify a 20-bit ADDRESS, as follows:

ADDRESS = SEGMENT * 16 + OFFSET

Most programming languages reflect this addressing scheme in their provisions for absolute memory reference. Consult your programming manual to find out how to read and write absolute memory locations. It does not matter how you divide the address specification between SEGMENT and OFFSET, but for simplicity we will assume that the SEGMENT is chosen so that the base address of the Carrier is at OFFSET 0. When OFFSET is 0, the 20-bit ADDRESS will correspond exactly to the setting of the base address switches on the Carrier.

In the programming procedures that follow, single-byte references are identified as BYTE (address), and two-byte references are identified as WORD (address). For example, a word at local register offset 02 (Hex) on a Carrier would be identified as WORD (02H). A byte at local register offset 40 (Hex) on a Carrier would be identified as BYTE (40H).

The programming language you use must have the capability of absolute memory reference, since the PCI-20000 Modules and Carriers are configured as memory-mapped I/O.

The hardware registers on Carriers and Modules can be read and written as if they were random-access memory, with the following limitations. Registers may be read-only, write-only, or read-write. In some cases the same register may be used for two different functions, depending on whether it is read or written. In such a case, or if a register is write-only, you will not be able to read back data that was previously written to the register. You must save the data in program memory if you will need to use it again. In some cases, just reading or writing a register may trigger an event, such as an analog conversion, regardless of the data involved. Be careful not to inadvertently read or write command registers.

Programming Procedures: PCI-20001C-2A with 32 Bits of Digital I/O

Port Configuration Procedure

The following procedure will configure an eight-channel digital port as inputs or outputs. Note that the control and enable registers cannot be read. If you change the direction of a single port, your program must remember the configuration of the other ports so that they can be reconfigured correctly.

Step 1. Write the control code to set the port direction to the appropriate control register, BYTE (83H) or BYTE (C3H). Since each control register controls two ports, you must be sure to set both ports correctly when you write this register.

Step 2. Write the enable register, BYTE (82H) to set the direction of the port buffer and enable it. Since this register controls all four ports, you must be sure to set all bits correctly when you write this register.

Step 3. Write all four output data registers. The reprogramming procedure resets all output lines, so you must re-write all output data any time you change the direction of a port.

Digital Output Procedure

The following procedure will write data to an output port.

Step 1. Set up the output byte. For each channel to be programmed with a HIGH level, set the corresponding bit in the output byte to 1; to program a LOW level, set the bit to a 0. The lowest-numbered channel of the port corresponds to the low-order bit of the output byte.

Step 2. Write the output byte to the appropriate I/O register, BYTE (80H), BYTE (81H), BYTE (C0H), or BYTE (C1H).

Digital Input Procedure

The following procedure will read data from an input channel.

Step 1. Read the appropriate I/O register, BYTE (80H), BYTE (81H), BYTE (C0H), or BYTE (C1H).

Step 2. Test the bit corresponding to the desired channel. If the bit is 0, the input signal is LOW; if the bit is 1, the input signal is HIGH.

Offset C1   Input/Output Port 3.

If programmed as an input, reading this location will transfer the states of the bits of Port 3 to the data bus. If programmed as an output, writing to this location will transfer the contents of the data bus to Port 3.

Offset C2   Not Used

Offset C3   Port 2, 3 Control.

This register controls the direction assigned to digital I/O ports 2 and 3. After assigning port direction, buffer direction and enable must then be assigned at Offset 82. Bit assignments are as follows:

Bit	Function	Comments
7	Bit value is always 1.	
6	Bit value is always 0.	
5	Bit value is always 0.	
4	Port 2 direction	1 = input; 0 = output.
3	Bit value is always 0.	
2	Bit value is always 0.	
1	Port 3 direction	1 = input; 0 = output.
0	Bit value is always 0.	

Note: Due to hardware restrictions, Ports 0 and 1 must be configured before Ports 2 and 3 can be used even if Ports 0 and 1 are not used.

Offset 82 Buffer Direction and Enable.

This location controls the direction and enabling for the Module's input/output buffers. Before setting buffer direction, port direction must be set at Offset 83 for ports 0 and 1, and at Offset C3 for ports 2 and 3. The format of the Buffer Direction and Enable register is as follows:

Bit	Function	Comments
7	Direction 3	Direction Bits:
6	Direction 2	0 = input
5	Enable 3	1 = output
4	Enable 2	
3	Direction 1	
2	Direction 0	Enable Bits:
1	Enable 1	0 = enabled
0	Enable 0	1 = disabled

Offset 83 Port 0, 1 Control.

This register controls the direction assigned to digital I/O ports 0 and 1. After assigning port direction, buffer direction and enable must then be assigned at Offset 82. Bit assignments are as follows:

Bit	Function	Comments
7	Bit value is always 1.	
6	Bit value is always 0.	
5	Bit value is always 0.	
4	Port 0 direction	1 = input; 0 = output.
3	Bit value is always 0.	
2	Bit value is always 0.	
1	Port 1 direction	1 = input; 0 = output.
0	Bit value is always 0.	

Offset 84 - Offset BF Not Used

Offset C0 Input/Output Port 2.

If programmed as an input, reading this location will transfer the states of the bits of Port 2 to the data bus. If programmed as an output, writing to this location will transfer the contents of the data bus to Port 2.

Carrier I.D. Bits					Carrier Type
4	3	2	1	0	
1	1	0	1	1	PCI-20001C-1A Plug-in Modules. No on-board Digital I/O.
1	1	1	0	1	PCI-20001C-2A Plug-in Modules. On-board Digital I/O.

Offset 01 - Offset 03F   Not Used

Offset 40   Module Interrupt Status.

Reading this register will return the state of the IRQ0 line from each of the three Modules plugged into the Carrier. The meanings of the status bits is specific to each Module, but generally a state of "0" indicates that some event of significance has occurred on the Module in question. The format of the byte returned is as follows:

Bit	Function
7	Interrupt Status, Module 1
6	Interrupt Status, Module 2
5	Interrupt Status, Module 3
4 - 0	Not Used

Offset 41 - Offset 7F   Not Used

Offset 80   Input/Output Port 0.

If programmed as an input, reading this register will transfer the states of the bits of Port 0 to the data bus. If programmed as an output, writing to this register will transfer the contents of the data bus to Port 0.

Offset 81   Input/Output Port 1.

If programmed as an input, reading this location will transfer the states of the bits of Port 1 to the data bus. If programmed as an output, writing to this location will transfer the contents of the data bus to Port 1.

Register Offsets

All register addresses are expressed in hexadecimal, and are shown below as offsets from the Carrier's base address. To determine the absolute addresses of these registers in the computer's memory space, refer to the Carrier Addressing section above, and to the Programmimg Notes below.

Carrier Register Offsets

Register Offset	Function (Read/Write)
C3	Control ports 2, 3 (W)
C2	Not Used
C1	Digital I/O port 3 (R/W)
C0	Digital I/O port 2 (R/W)
BF - 84	Not Used
83	Control ports 0, 1 (W)
82	Buffer direction and enable (R/W)
81	Digital I/O port 1 (R/W)
80	Digital I/O port 0 (R/W)
7F - 41	Not Used
40	Module interrupt status (R)
3F - 01	Not Used
00	Carrier I.D.; module present (R)

A detailed description of the function of each register follows:

Offset 00 Carrier I.D.; Module Present.

Reading this register returns the Carrier I.D. in the low-order five bits, and the three Module Present bits in the upper three bits. A "0" in a Module Present bit indicates that a Module is plugged into the corresponding slot on the Carrier. The format of the register is as follows:

Bit	Function	Comments
7	Module 1 Present	Module is present
6	Module 2 Present	only if bit value
5	Module 3 Present	is 0
4	Carrier I.D. bit 4	
3	Carrier I.D. bit 3	See below for
2	Carrier I.D. bit 2	Carrier types
1	Carrier I.D. bit 1	
0	Carrier I.D. bit 0	Value is always 1

The 8086 microprocessor family addresses memory using a 16-bit SEGMENT register in conjunction with a 16-bit OFFSET to specify a 20-bit ADDRESS, as follows:

ADDRESS = SEGMENT * 16 + OFFSET

Therefore, in the above example, the 20-bit absolute address of a Module's register, which was calculated to be C0203 (Hex), could be expressed by a programming language as

SEGMENT C000:OFFSET 0203

Following are some Carrier and Instrument Module addressing examples. Note that Carrier #2, with its switches set to a base address of C0400 (Hex), is shown with two different ways for a program to address it; one way is with the SEGMENT address at C000 (Hex), and the other is with the SEGMENT address at C040 (Hex). It does not matter how the address specification is divided between SEGMENT and OFFSET, as long as the SEGMENT is evenly divisible by 16 (or 10 (Hex))--the choice is up to the programmer.

Carrier #1 (Address switches set to C0000)

Carrier Base Address SEGMENT : OFFSET	Module 1 Block Address	Module 2 Block Address	Module 3 Block Address
C000 : 000	100	200	300

Carrier #2 (Address switches set to C0400)

Carrier Base Address SEGMENT : OFFSET	Module 1 Block Address	Module 2 Block Address	Module 3 Block Address
C000 : 400	500	600	700
-- or --			
C040 : 000	100	200	300

APPENDIX C

CONTROLLER PARAMETERS LISTING

Below is a list of all the addressable controller parameters. Information about the parameter register, its conversion between engineering units and counts, its initial value etc. is given here exactly in the format accessed by the program. This is stored in the array PARAMETERS (see 6.1 - scr_setup) at start-up. The different fields are:

REG: register on the controller where the parameter is stored.

SCALE: scaling factor for units-counts conversion

#BYTE: number of bytes the parameter occupies on the controller. Values from 1-7.

1- Upper byte of register.

2- Upper & lower bytes of register.

3- Upper byte of register + both bytes of immediately preceding register, reg-1

4- Both bytes of register and of preceding register (reg-1).

5- Both bytes of register + Lower byte of preceding (reg-1).

6- Lower byte of register.

7- Upper byte of register multiplied by mask (bit#).

BIT#: for parameters governed by bits in a single register, this provides a mask to isolate the relevant bit number.

EQTYP: units-counts conversion equation number. Values from 1-6.

1. V# = U#*scale

2. V# = scale/U#

3. V# = U#

4. V# = (U#*scale - offset) * slope

5. V# = U# * slope.

6. V# = U# * scale * slope

V# = engg. units U# = counts scale = scaling factor offset = offset from zero

slope = slope of tuned signal.

NOTE: The values of offset and slope are from the controlling feedback (feedback in) and CANNOT be used to convert the non-controlling (MONITOR) feedback- a separate offset and slope are read in for the non-controlling feedback when tuned. Currently, tuning is only possible using the TS&S software where the values are: slope = (d5-d3) / (d6-d4) where d3, d4, d5 and d6 are the feedback values in engg. units (physical 1 & 2 - d3 and d5) and counts (dcl values 1 & 2 - d4 and d6) in the TS&S calibration utility (Function key F9 from the main TS&S menu) offset is the value of d4 (count 1) when d3 (physical 1 is at zero).

XPOS:	row position on screen1
YPOS:	column position on screen 1
NAME:	name of parameter/
INITVAL:	initialization value when rebooting.

REG	SCALE	#BYTE	BIT#	EQTYP	XPOS	YPOS	NAME	INITVAL
139	1	2	0	4	0	0	COMMAND"	0
153	1	2	0	4	0	0	"FEEDBACK"	0
238	1	2	0	5	0	0	"ERROR"	0
148	1	1	255	3	0	0	"STATUS"	0

REG	SCALE	#BYTE	BIT#	EQTYP	XPOS	YPOS	NAME	INITVAL
150	1	7	BIT7	3	0	0	"WAVEFORM"	0
150	1	7	BIT5	3	0	0	"RUNMODE"	0
150	1	7	BIT3	3	0	0	"INT TYPE"	0
152	1	2	0	4	0	0	"SETPOINT"	0
223	1	2	0	5	0	0	"AMPLITUDE"	0
222	50382.	3	0	2	0	0	"HAVERTIME"	0
222	$9.9x10^{-6}$	3	0	1	0	0	"FREQUENCY"	0
158	0.0051	4	0	6	0	0	"VELOCITY"	0
160	1.3021	4	0	6	0	0	"ACCL"	0
219	1	3	0	1	0	0	"COUNT SET"	0
220	1	5	0	1	0	0	"COUNT NOW"	0
255	0.00153	2	0	1	0	0	"P GAIN"	0
254	0.00153	2	0	1	0	0	"I GAIN"	0
253	0.00153	2	0	1	0	0	"D GAIN"	0
249	0.00153	2	0	1	0	0	"CPE"	0
252	0.185	2	0	1	0	0	"LOOP FREQ"	0
252	0.185	2	0	1	0	0	"FDBK FREQ"	0
251	0.392	2	0	1	0	0	"HIPASS GN"	0
161	0.625	1	255	1	0	0	"MAX FLOW"	0
251	1	2	0	1	0	0	"AREA"	0
250	0.625	2	0	1	0	0	"LAP"	0
250	0.625	2	0	1	0	0	"SV BIAS"	0
164	1	1	255	1	0	0	"INT THRSH"	0

REG	SCALE	#BYTE	BIT#	EQTYP	XPOS	YPOS	NAME	INITVAL
149	1	6	BIT5	3	0	0	"SERVO +/-"	0
149	1	6	BIT4	3	0	0	"INPT1 +/-"	0
149	1	6	BIT3	3	0	0	"INPT2 +/-"	0
149	1	6	BIT1	3	0	0	"FDBK IN"	0
150	1	1	BIT1	3	0	0	"LOOP STAT"	2
143	1	2	0	4	0	0	"MONITOR	0
205	1	7	BIT5	3	0	0	"FDBK MODE	0
201	1	6	255	1	0	0	"FDBCK PIO"	0
241	1	2	0	4	0	0	"FDBCK HI "	10.0
244	1	2	0	4	0	0	"FDBCK LO "	0
149	1	1	BIT4	3	0	0	"FBK HI STAT"	0
149	1	1	BIT1	3	0	0	"FBK LO STAT"	0
205	1	7	BIT3	3	0	0	"MON MODE"	0
201	1	1	255	1	0	0	"MON PIO"	0
242	1	2	0	4	0	0	"MON HI"	0
243	1	2	0	4	0	0	"MON LO"	0
149	1	1	BIT3	3	0	0	"MON HI STA"	0
149	1	1	BIT2	3	0	0	"MON LO STA"	0
205	1	7	BIT7	3	0	0	"ERR MODE"	0
200	1	6	255	1	0	0	"ER PIO"	0
240	1	2	0	5	0	0	"ABS ERROR "	10
240	1	2	0	5	0	0	"ERROR HI"	0
149	1	1	BIT5	3	0	0	"ERR HI STAT"	0
149	1	1	BIT5	3	0	0	"ERR HI STAT"	0

PARAMETER DEFINITIONS

This section defines the function and location of the user accesable parameters. Parameters range in size from one bit to 32 bits. Parameters with sizes greater than 16 bits require two write cycles to the DCL. For read / write parameters, an example WRITE BLOCK is shown. For read only parameters, an example READ BLOCK is shown. Any parameter can be read using the READ parameter defined in the following section.

Definitions of terms used in this section:

Data range - Maximum and minimum values for each parameter

DCL address - Current address setting of the DCL via switch settings or software override

Parameter size - Number of bits in this parameter

Data type - Defines the parameter as read only or read / write

Data format - Parameter formats are defined as follows:

A) **Unsigned binary** - Natural binary in the range of 0 to 255 for 8 bit data, 0 to 65535 for 16 bit data, 0 to 16777215 for 24 bit data and 0 to 4294967295 for 32 bit data.

B) **Signed binary** - The uppermost bit of the parameter is used as the sign bit for simplicity (Not 2's complement). A '0' represents a positive number and a '1' signifies a negative number. Since the uppermost bit is used for sign, the range is -127 to +127 for 8 bit data and -32767 to +32767 for 16 bit data. No 24 or 32 bit signed parameters are used.

Parameter scaler - Many parameters need a conversion factor to convert engineering units to binary to load or read DCL parameters. Divide the engineering units by the scaler to obtain the proper binary number to send to the DCL. Multiply the binary number received from the DCL by the scaler to obtain the engineering units. When converting to binary, truncate the fractional portion of the result.

SETPOINT - Address 152 (098H)

The SETPOINT has different uses depending on the profile type currently selected.

Profile type	Use of setpoint
Ramp	Defines the endpoint
Haversine	Defines the endpoint
Sine	Defines the mean level

Parameter size - 16 bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 65535
Parameter scaling - None

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 152 (098H) - Register address for SETPOINT
Byte 3 - SETPOINT MSB data
Byte 4 - SETPOINT LSB data
Byte 5 - Checksum

FEEDBACK - Address 153 (099H)

The FEEDBACK parameter contains the actual feedback sensor value.

Parameter size - 16 Bit
Data type - Read only
Data format - Unsigned binary
Data range - 0 to 65535 (Depending on type of sensor)
Parameter scaling - None

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 153 (099H) - Register address for FEEDBACK
Byte 3 - FEEDBACK MSB data
Byte 4 - FEEDBACK LSB data
Byte 5 - Checksum

COUNTER - Address 219 (0DBH) LSB
Address 220 (0DCH) MSB,LSB

COUNTER is the actual number of cycles completed when the sine profiler is active.

Parameter size - 24 Bit
Data type - Read only
Data format - Unsigned binary
Data range - 0 to 8388607

Example WRITE BLOCK (two are required):

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 155 (09BH) - Register address for READ
Byte 3 - 219 (0DBH) - Read COUNTER (Upper 8 bits)
Byte 4 - 0
Byte 5 - Checksum

The DCL response to the above WRITE BLOCK is a READ BLOCK as follows:

Byte 1 - 128 (080H) + DCL address
Byte 2 - 219 (0DBH) - COUNTER data being read
Byte 3 - XXX (0XXH) - Ignore this byte (Except for checksum)
Byte 4 - COUNTER (Highest 8 bits)
Byte 5 - Checksum

Second WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 155 (09BH) - Register address for READ
Byte 3 - 220 (0DCH) - Read COUNTER (Lower 16 bits)
Byte 4 - 0
Byte 5 - Checksum

The DCL response to the above WRITE BLOCK is a READ BLOCK as follows:

Byte 1 - 128 (080H) + DCL address
Byte 2 - 220 (0DCH) - COUNTER data being read
Byte 3 - COUNTER (Middle 8 bits)
Byte 4 - COUNTER (Lowest 8 bits)
Byte 5 - Checksum

INTEGRATOR TIMER ----- Address 164 (0A4H) MSB
INTEGRATOR THRESHOLD - Address 164 (0A4H) LSB

The DCL Integrator has a 'Smart' mode which allows integration to be used only when required to eliminate the overshoot and instability commonly found when using continous integrators. The INTEGRATOR TIMER adjusts the minimum time between Integrator ON / OFF and OFF / ON cycles. The INTEGRATOR THRESHOLD sets maximum velocity at which the integrator will operate.

INTEGRATOR TIMER	INTEGRATOR THRESHOLD
Parameter size: 8 Bit	Parameter size: 8 Bit
Data type: Read / Write	Data type: Read / Write
Data format: Unsigned bin.	Data format: Unsigned bin.
Data range: 0 - 765 ms	Data range: 0 - 765 bits/ms
Parameter scaling: 3.000	Parameter scaling: 3.000

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 164 (0A4H) - Reg. address for INTEGRATOR TIMER and INTEGRATOR THRESHOLD
Byte 3 - INTEGRATOR TIMER data
Byte 4 - INTEGRATOR THRESHOLD data
Byte 5 - Checksum

HAVERSINE TIME - Address 221 (0DDH) MSB,LSB
Address 222 (0DEH) MSB

HAVERSINE TIME sets the haversine endpoint to endpoint time.

Parameter size - 24 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 330 seconds
Parameter scaling - 0.00001984835

Example WRITE BLOCK:

Byte 1 - 176 (0B0H) + DCL address
Byte 2 - 221 (0DDH) - Register address for upper 16 bits of HAVERSINE TIME
Byte 3 - HAVERSINE TIME (Highest 8 bits)
Byte 4 - HAVERSINE TIME (Middle 8 bits)
Byte 5 - Checksum

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 222 (0DEH) - Register address for lower 8 bits of HAVERSINE TIME
Byte 3 - HAVERSINE TIME (Lowest 8 bits)
Byte 4 - 0
Byte 5 - Checksum

RAMP VELOCITY - Address 157 (09DH) MSB,LSB
Address 158 (09EH) MSB,LSB

RAMP VELOCITY is used when the ramp profiler is active.

Parameter size - 32 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 21845000 bits/second
Parameter scaling - 0.0050862

Example WRITE BLOCK:

Byte 1 - 176 (0B0H) + DCL address
Byte 2 - 157 (09DH) - Register address for upper 16 bits of RAMP VELOCITY
Byte 3 - RAMP VELOCITY (Highest 8 bits)
Byte 4 - RAMP VELOCITY (Upper middle 8 bits)
Byte 5 - Checksum

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 158 (09EH) - Register address for lower 16 bits of RAMP VELOCITY
Byte 3 - RAMP VELOCITY (Lower middle 8 bits)
Byte 4 - RAMP VELOCITY (Lowest 8 bits)
Byte 5 - Checksum

RAMP ACCELERATION - Address 159 (09FH) MSB,LSB
Address 160 (0A0H) MSB,LSB

RAMP ACCELERATION is used when the ramp profiler is active.

Parameter size - 32 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 5592348066 bits/second ^{2}
Parameter scaling - 1.30207

Example WRITE BLOCK:

Byte 1 - 176 (0B0H) + DCL address
Byte 2 - 159 (09FH) - Register address for upper 16 bits of RAMP ACCELERATION
Byte 3 - RAMP ACCELERATION (Highest 8 bits)
Byte 4 - RAMP ACCELERATION (Upper middle 8 bits)
Byte 5 - Checksum

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 160 (0A0H) - Register address for lower 16 bits of RAMP ACCELERATION
Byte 3 - RAMP ACCELERATION (Lower middle 8 bits)
Byte 4 - RAMP ACCELERATION (Lowest 8 bits)
Byte 5 - Checksum

AMPLITUDE - Address 223 (0DFH)

When the sine profiler is active, AMPLITUDE is the peak (not peak to peak) amplitude for the sine.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Signed binary
Data range - -32767 to +32767

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 223 (0DFH) - Register address for AMPLITUDE
Byte 3 - AMPLITUDE MSB data
Byte 4 - AMPLITUDE LSB data
Byte 5 - Checksum

FREQUENCY - Address 221 (0DDH) MSB,LSB
Address 222 (0DEH) MSB

FREQUENCY is used for the sine profiler.

Parameter size - 24 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 80 hz
Parameter scaling - 0.0000099341

Example WRITE BLOCK:

Byte 1 - 176 (0B0H) + DCL address
Byte 2 - 221 (0DDH) - Register address for upper 16 bits of FREQUENCY
Byte 3 - FREQUENCY (Highest 8 bits)
Byte 4 - FREQUENCY (Middle 8 bits)
Byte 5 - Checksum

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 222 (0DEH) - Register address for lower 8 bits of FREQUENCY
Byte 3 - FREQUENCY (Lowest 8 bits)
Byte 4 - 0
Byte 5 - Checksum

COUNTER SET - Address 218 (0DAH) MSB,LSB
Address 219 (0DBH) MSB

COUNTER SET selects the number of cycles to run when the sine profiler is active.

Parameter size - 24 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 8388607

Example WRITE BLOCK (Two are required):

Byte 1 - 176 (0B0H) + DCL address
Byte 2 - 218 (0DAH) - Register address for the upper 16 bits of COUNTER SET
Byte 3 - COUNTER SET (Highest 8 bits)
Byte 4 - COUNTER SET (Middle 8 bits)
Byte 5 - Checksum

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 219 (0DBH) - Register address for COUNTER SET (Lower 8 bits)
Byte 3 - COUNTER SET (Lowest 8 bits)
Byte 4 - 0
Byte 5 - Checksum

FLOW LIMIT - Address 161 (0A1H) MSB

FLOW LIMIT sets the maximum possible servovalve spool displacement.

Parameter size - 8 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 100 %
Parameter scaling - 0.625

Example WRITE BLOCK:

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 161 (0A1H) - Register address for FLOW LIMIT
Byte 3 - FLOW LIMIT DATA
Byte 4 - 0
Byte 5 - Checksum

OPEN LOOP - Address 203 (0CBH)

OPEN LOOP sets the servovalve spool position when the control loop is set to OPEN.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Signed binary
Data range - -100 to 100 %
Parameter scaling - 0.0048828125

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 203 (0CBH) - Register address for OPEN LOOP
Byte 3 - OPEN LOOP MSB data
Byte 4 - OPEN LOOP LSB data
Byte 5 - Checksum

COMMAND - Address 139 (08BH)

COMMAND is the actual servo command entering the summing junction of the closed loop controller.

Parameter size - 16 Bit
Data type - Read only
Data format - Unsigned binary
Data range - 0 to 65535

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 155 (09BH) - Register address for READ
Byte 3 - 139 (08BH) - Read COMMAND
Byte 4 - 0
Byte 5 - Checksum

The DCL response to the above WRITE BLOCK is a READ BLOCK as follows:

Byte 1 - 128 (080H) + DCL address
Byte 2 - Address of DCL register being READ
Byte 3 - COMMAND MSB data
Byte 4 - COMMAND LSB data
Byte 5 - Checksum

FOLLOWING ERROR - Address 238 (0EEH)

FOLLOWING ERROR is the actual difference between the COMMAND and FEEDBACK.

Parameter size - 16 Bit
Data type - Read only
Data format - Signed binary
Data range - -32767 to 326767

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 155 (09BH) - Register address for READ
Byte 3 - 238 (0EEH) - Read FOLLOWING ERROR
Byte 4 - 0
Byte 5 - Checksum

The DCL response to the above WRITE BLOCK is a READ BLOCK as follows:

Byte 1 - 128 (080H) + DCL address
Byte 2 - Address of DCL register being READ
Byte 3 - FOLLOWING ERROR MSB data
Byte 4 - FOLLOWING ERROR LSB data
Byte 5 - Checksum

PROPORTIONAL GAIN - Address 255 (0FFH)

The PROPORTIONAL GAIN is the main gain control used in the closed loop control equation.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 100%
Parameter scaling - 0.001525902

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 255 (0FFH) - Register address for PROPORTIONAL GAIN
Byte 3 - PROPORTIONAL GAIN MSB data
Byte 4 - PROPORTIONAL GAIN LSB data
Byte 5 - Checksum

INTEGRAL GAIN - Address 254 (0FEH)

The INTEGRAL GAIN is the offset compensator term used in the closed loop control equation.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 100%
Parameter scaling - 0.001525902

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 254 (0FEH) - Register address for INTEGRAL GAIN
Byte 3 - INTEGRAL GAIN MSB data
Byte 4 - INTEGRAL GAIN LSB data
Byte 5 - Checksum

DERIVITIVE GAIN - Address 253 (0FDH)

The DERIVITIVE GAIN is the high speed damping term used in the closed loop control equation.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 100%
Parameter scaling - 0.001525902

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 253 (0FDH) - Register address for DERIVITIVE GAIN
Byte 3 - DERIVITIVE GAIN MSB data
Byte 4 - DERIVITIVE GAIN LSB data
Byte 5 - Checksum

FEEDBACK FILTER - Address 252 (0FCH) MSB
FORWARD FILTER -- Address 252 (0FCH) LSB

The FEEDBACK FILTER sets the cutoff frequency for the feedback sensor before the summing junction. The FORWARD FILTER sets the cutoff frequency in the forward loop after the summing junction. Setting a filter to 0 bypasses the filter.

FEEDBACK FILTER	FORWARD FILTER
Parameter size - 8 Bit	Parameter size - 8 Bit
Data type - Read / Write	Data type - Read / Write
Data format - Unsigned bin.	Data format - Unsigned bin.
Data range - 0 to 47.1 hz	Data range - 0 to 47.1 hz
Parameter scaling - 0.184894	Parameter scaling - 0.184894

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 252 (0FCH) - Register address for FILTERS
Byte 3 - FEEDBACK FILTER data
Byte 4 - FORWARD FILTER data
Byte 5 - Checksum

GAIN RATIO ---- Address 251 (0FBH) MSB
FEEDBACK LEAD - Address 251 (0FBH) LSB

The GAIN RATIO provides compensation when single ended actuators are used. The FEEDBACK LEAD allows an additional damping signal to be superimposed on the feedback signal at the summing junction. Setting either parameter to 0 will bypass the parameter.

GAIN RATIO	FEEDBACK LEAD
Parameter size - 8 Bit	Parameter size - 8 Bit
Data type - Read / Write	Data type - Read / Write
Data format - Signed bin.	Data format - Unsigned bin.
Data range - +/- 100 %	Data range - 0 to 100 %
Parameter scaling - 0.787402	Parameter scaling - 0.392156

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 251 (0FBH) - Reg. address for G. RATIO & F. LEAD
Byte 3 - GAIN RATIO data
Byte 4 - FEEDBACK LEAD data
Byte 5 - Checksum

BIAS ---- Address 250 (0FAH) MSB
OVERLAP - Address 250 (0FAH) LSB

The BIAS parameter has the same effect as the mechanical null adjust on the servovalve. The overlap parameter can be used to compensate for an overlap condition on a non-standard servovalve.

BIAS	OVERLAP
Parameter size - 8 Bit	Parameter size - 8 Bit
Data type - Read / Write	Data type - Read / Write
Data format - Signed bin.	Data format - Unsigned bin.
Data range - +/- 100 %	Data range - 0 to 100 %
Parameter scaling - 0.787402	Parameter scaling - 0.787402

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 250 (0FAH) - Register address for BIAS & OVERLAP
Byte 3 - BIAS data
Byte 4 - OVERLAP data
Byte 5 - Checksum

CPE - Address 249 (0F9H)

CPE adjusts the closed loop system following error.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 100%
Parameter scaling - 0.001525902

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 249 (0F9H) - Register address for CPE
Byte 3 - CPE MSB data
Byte 4 - CPE LSB data
Byte 5 - Checksum

LIMIT 1 - Address 244 (0F4H)

LIMIT 1 is the threshold for the low outer feedback limit.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 65535

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 244 (0F4H) - Register address for LIMIT 1
Byte 3 - LIMIT 1 MSB data
Byte 4 - LIMIT 1 LSB data
Byte 5 - Checksum

LIMIT 2 - Address 243 (0F3H)

LIMIT 2 is the threshold for the low inner feedback limit.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 65535

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 243 (0F3H) - Register address for LIMIT 2
Byte 3 - LIMIT 2 MSB data
Byte 4 - LIMIT 2 LSB data
Byte 5 - Checksum

LIMIT 3 - Address 242 (0F2H)

LIMIT 3 is the threshold for the high inner feedback limit.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 65535

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 242 (0F2H) - Register address for LIMIT 3
Byte 3 - LIMIT 3 MSB data
Byte 4 - LIMIT 3 LSB data
Byte 5 - Checksum

LIMIT 4 - Address 241 (0F1H)

LIMIT 4 is the threshold for the high outer feedback limit.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 65535

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 241 (0F1H) - Register address for LIMIT 4
Byte 3 - LIMIT 4 MSB data
Byte 4 - LIMIT 4 LSB data
Byte 5 - Checksum

LIMIT 5 - Address 240 (0F0H)

LIMIT 5 is the threshold for the servo error limit.

Parameter size - 16 Bit
Data type - Read / Write
Data format - Unsigned binary
Data range - 0 to 65535

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 240 (0F0H) - Register address for LIMIT 5
Byte 3 - LIMIT 5 MSB data
Byte 4 - LIMIT 5 LSB data
Byte 5 - Checksum

BIT MAP

The remainder of the DCL parameters are in bit form to allow on-off selection of various functions. Bit 0 is the lowest bit (2^0) and bit 7 is the highest (2^7). Since many on-off functions exist in a given register, the 16 bits must be read, modified and written back ensuring that other on-off parameters are not disturbed.

LIMIT STATUS - Address 149 (095H) MSB

Bit 0 of is set to a '1' state when a transducer fault has occured.

Bits 1 through 5 corrospond to limits 1 through 5 as described previously. Each bit is latched into a '1' state when the limit has been exceeded. Writing a '0' to the appropoiate bit position will clear the limit, provided the fault no longer exists.

Bits 6 and 7 of LIMIT STATUS must be set to '0'.

Example WRITE BLOCK:

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 149 (095H) - Register address for LIMIT STATUS
Byte 3 - LIMIT STATUS
Byte 4 - 0
Byte 5 - Checksum

I/O POLARITY - Address 149 (095H) LSB

Bit 5 of I/O POLARITY sets the servovalve polarity. All other bits must be set to '0'.

Example WRITE BLOCK:

Byte 1 - 240 (0F0H) + DCL address
Byte 2 - 149 (095H) - Register address for I/O POLARITY
Byte 3 - 0
Byte 4 - I/O POLARITY
Byte 5 - Checksum

LOOP CONTROL - Address 150 (096H) MSB

Bits 7 and 6 select the type of waveform as follows:

Bit 7	Bit 6	Waveform
1	1	Sine
1	0	Direct
0	1	Haversine
0	0	Ramp

Bits 5 and 4 control the waveform as follows:

Bit 5	Bit 4	Action
1	1	Abort
1	0	Zero
0	1	Hold
0	0	Run

Bits 3 and 2 control the integrator as follows:

Bit 3	Bit 2	Type of integrator
1	1	Continous
1	0	Smart
0	1	Hold
0	0	None

Bit 1 sets the loop status as follows:

Bit 1	Loop status
0	Open
1	Closed

Bit 0 must be set to '0'.

Example WRITE BLOCK:

Byte 1 - 224 (0E0H) + DCL address
Byte 2 - 150 (096H) - Register address for LOOP CONTROL
Byte 3 - LOOP CONTROL DATA
Byte 4 - 0
Byte 5 - Checksum

VITA

Poulomi Damany, scheduled to be born on April 1, 1969, decided to fool her parents, Drs. Bharat and Saroj Damany, and arrived into this world (smiling up to the moment the obstetrician smacked her bottom) a day later on April 2 at the Civil Hospital in Ahmedabad, India, much to the relief of the above mentioned doctor who hadn't slept in 36 hours on her account.

She has managed to keep this sense of humor through ten years of a convent education at the St. Annes' High School, Bombay, India (run by nuns - need I say more?) and fours years as one of four female EE majors thrown in with a class of eighty-six Indian male chauvinists at the University of Bombay and two years in the graduate program of the Electrical Engineering & Computer Science department of Lehigh University (formerly an all-male school - pure coincidence).

Her future plans include back-packing through Europe, buying a car, getting a job in the land of opportunity and end with sailing away into the sunset in her all-payed for, fully-owned yacht.