PERSONAL COMPUTER BASED
DATA ACQUISITION, SENSING AND CONTROL

A Thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Daniel Alan Allwine
March, 1993
Acknowledgements

Before beginning any discussion, the author wishes to give thanks to those who have made it possible for this work to be completed.

First of all, infinite thanks go to Dr. R. Dennis Irwin whose leadership, advice, patience, friendship and general support has been indispensable in this endeavor. Further thanks are extended to Dr. Jerrel R. Mitchell who has on more than one occasion provided insight and motivation without which the author would have been unable to finish.

Many thanks also go to the other Professors and Administration of the Department of Electrical and Computer Engineering at Ohio University. The education and support received by the author from these people has been invaluable throughout his graduate career.

It goes without saying that this work would never have been begun if it were not for the endowment made by Paul and Beth Stocker to Ohio University. For this, the author extends his regards and thanks.

The author also wishes to thank Christine Nichols for the many hours she spent patiently listening while the author discussed the endless details of his graduate work.

Finally, the author gives thanks to his parents, William and Tamara Allwine. They have been a source of immeasurable, unwavering support throughout the author's life. It is to them that this work is dedicated.
Table of Contents

Acknowledgements ........................................................................ i
Table of Contents ...................................................................... ii
List of Figures .......................................................................... iv
List of Tables ........................................................................... vi
Chapter 1: Introduction ............................................................. 1

Chapter 2: The Keithley Data Acquisition System (Model DAS-20) ................................................................ 4
  2.1. Theory of Operation ......................................................... 4
  2.2. Limitations and Their Solutions ....................................... 8
    2.2.1. Using the Software Drivers Provided by Metabyte ........ 9
    2.2.2. Custom Software for the DAS-20 ............................. 10
      2.2.2.1. Combining the Supplied Drivers into a
                 Useable System ........................................ 10
      2.2.2.2. Writing an Interrupt Service Routine to
                 Avoid Data Timing Difficulties ...................... 13
      2.2.2.3. Combining the DMA Capabilities of the
                 DAS-20 with Direct PC Port I/O .................... 28

Chapter 3: Development of a Senior Control Laboratory ............. 31
  3.1. Laboratory Hardware ..................................................... 31
    3.1.1. The Modular Servo System ...................................... 31
    3.1.2. The DAS-20 and PC ............................................... 33
    3.1.3. The I/O Buffer Circuits .......................................... 34
  3.2. Laboratory Software Development .................................. 34
  3.3. The Experiments ............................................................ 35

Chapter 4: The Infra-Red Multi-Position Sensing System (IRMPSS) 50
  4.1. System Hardware .......................................................... 50
    4.1.1. The Hamamatsu S-1200 Large Area Position Sensitive
           Device .............................................................. 50
    4.1.2. Fujinon CF25B TV Lens ......................................... 52
    4.1.3. Hamamatsu C3683 Analog Signal Processing Device ..... 52
    4.1.4. Hamamatsu L2791 Infra-Red Emitting Diode ............. 54
    4.1.5. Power Amplifier for LED Circuit ............................ 55
    4.1.6. Two-Axis Automatic Linear Position Controller ......... 56
    4.2. Characterization Experiments and Software Development ... 58
4.2.1. Generating a Map of the Detector Surface .......................... 58
4.2.2. Sensor Resolution Tests ........................................... 65
4.2.3. System Switching Characteristics .................................. 66
4.2.4. System Noise Characteristics ...................................... 71
4.3. Characterization Results ............................................. 73
4.3.1. Sensor Mapping ...................................................... 73
4.3.2. Resolution Tests ..................................................... 75
4.3.3. Switching Characteristics .......................................... 77
4.3.4. Noise Characteristics ............................................. 78

Chapter 5: Conclusions and Recommendations ............................ 80

References ................................................................. 85

APPENDIX I: ISRSVAR Software Listing (C) ............................... 86
APPENDIX II: XYTEST Software Listing (C) ............................... 99
APPENDIX III: SNOISE Software Listing (C) ............................... 108
APPENDIX IV: DSPLIT m-File (MATLAB) .................................... 113
APPENDIX V: IPC Software Listing (C) ..................................... 114
APPENDIX VI: RESTEST Software Listing (C) ............................... 121
APPENDIX VII: MATLSH.H Software Listing (C) ............................ 123
APPENDIX VIII: DASMAIN Software Listing (FORTRAN) ................. 127
APPENDIX IX: DCCALIB Software Listing (FORTRAN) ..................... 136
APPENDIX X: DCCALIB2 Software Listing (FORTRAN) .................... 140
List of Figures

| Figure 2.1 | DAS-20 Block Diagram | 4 |
| Figure 2.2 | Flowchart of Switched DMA Programming Method | 12 |
| Figure 2.3 | Control System Block Diagram | 15 |
| Figure 2.4 | Flowchart of the ISR Programming Method | 16 |
| Figure 2.5 | Code demonstrating how to start the pacer clocks | 19 |
| Figure 2.6 | Code to perform manual A/D conversion on DAS-20 | 24 |
| Figure 2.7 | Code demonstrating how to send data to the DACs | 26 |
| Figure 2.8 | Flowchart of Paced DMA Programming Method | 29 |
| Figure 3.1 | Block Diagram of Basic Servo System | 31 |
| Figure 3.2 | Actual and Modeled Servo-Motor Velocity Step Response | 38 |
| Figure 3.3 | Actual and Modeled Servo System Step Response | 40 |
| Figure 3.4 | Experimentally Estimated and Modeled Magnitude Response | 41 |
| Figure 3.5 | Experimentally Estimated and Modeled Phase Response | 42 |
| Figure 3.6 | Experimentally Estimated and TFDC Magnitude Response | 43 |
| Figure 3.7 | Experimentally Estimated and TFDC Phase Responses | 44 |
| Figure 3.8 | Actual and TFDC Step Response | 44 |
| Figure 3.9 | Block Diagram of Compensated System | 45 |
| Figure 3.10 | Step Response of Settling Time Compensated System | 46 |
| Figure 3.11 | Step Response of Steady-State Error Compensated System | 47 |
| Figure 3.12 | Step Response of PID Compensated System | 49 |
| Figure 4.1 | Plan View of Detector | 50 |
| Figure 4.2 | Cross-Section of Detector | 51 |
| Figure 4.3 | Equivalent Circuit of Detector | 51 |
| Figure 4.4 | Schematic of Signal Processing Circuit | 53 |
| Figure 4.5 | Schematic of LED Power Amplifier | 55 |
| Figure 4.6 | LPC Test Configuration | 56 |
| Figure 4.7 | EAI Series 1130 Variplotter (EAI Maintenance Series, Page iv.) | 57 |
| Figure 4.8 | Calibration Coordinate Map | 59 |
| Figure 4.9 | Flow Diagram For Position Calibration Software (XYTEST) | 60 |
| Figure 4.10 | Example of LPC Motion | 61 |
| Figure 4.11 | Flowchart of Incremental Position Control Software | 63 |
| Figure 4.12 | Resolution Test Configuration | 65 |
| Figure 4.13 | Flexible Structures Test Facility Concept | 67 |
| Figure 4.14 | X-Axis Slow Switching Diagram | 68 |
| Figure 4.15 | X-Axis Critical Switching Diagram | 69 |
| Figure 4.16 | X-Axis Fast Switching Diagram | 69 |
| Figure 4.17 | Flow Diagram for SNOISE Software | 72 |
| Figure 4.18 | Map of Entire Detector Surface | 74 |
| Figure 4.19 | Map of Linear Portion of Detector Surface | 74 |
| Figure 4.20 | X-Axis Resolution Vs. Distance to Object | 76 |
| Figure 4.21 | Y-Axis Resolution Vs. Distance to Object | 77 |
Figure 4.22: PSD of X-Axis Data .................................................. 79
Figure 4.23: PSD of Y-Axis Data .................................................. 79
List of Tables

Table 2.1: Example of DAS-20 Scan Queue. [Keithley/Metrabyte/Asyst DAS-20 Manual, Page 5-3] ........................................ 6
Table 4.1: SNOISE Data Format ........................................ 71
Table 4.2: Resolution Test Results. ................................. 76
Chapter 1: Introduction

For purposes of experimental research, an economical, reliable, and flexible system for acquiring and manipulating data is invaluable. With the advent of the IBM Personal Computer (PC) and its compatibles, many data manipulation and/or control systems have been introduced that interface with the PC to perform various tasks. These devices claim capabilities that range from simple data acquisition to full real-time control of one or more processes or systems. While many of these products perform a selected set of tasks quite well, few of them maintain a versatile range of applications. These multipurpose systems tend to be rather expensive and thus are only purchased for use in projects with expansive budgets.

It is the purpose of this thesis to introduce some concepts concerning the use of a PC system as the basis for an advanced, low-cost research tool and control platform. While particular brand name components are cited throughout the text, it is important to realize that many similar devices exist that are manufactured by other organizations. The focus on the specific devices simply arises from their use in the development of the capabilities that are presented in this thesis.

Chapter 2 discusses the Keithley/Metrabyte/Asyst model DAS-20 data acquisition system. This is a powerful expansion board for the PC and is used as the backbone for all the development herein. The capabilities and operating principles of the DAS-20 are discussed with a slight focus on possible implementations of the device. Primary emphasis in this chapter is placed on programming the DAS-20 to perform various tasks. The driver software supplied by the manufacturer is discussed briefly, and a complete
exploration of the additional software written by the author is included. Incorporated into the discussion of the programming methods for the DAS-20 is a somewhat in-depth look at some programming considerations for IBM compatible PC's in general.

Chapter 3 discusses an application of the acquisition and control system in which the DAS-20 is used to calibrate and control a servomotor system manufactured by Feedback Instruments Limited. The servo-system is used for the instruction of a senior level feedback control laboratory. The development of the controller itself is performed via the MATLAB software package and implemented through the DAS-20 by the PC. The details of the hardware are presented and the software specific to the application is also briefly examined. Chapter 3 concludes with a discussion of the experiments developed for the laboratory.

Chapter 4 addresses another application in which the PC based system is integrated with additional hardware to develop an infra-red multi-position sensing system. The text presents a detailed discussion of the hardware included in the system as well as the methods by which the DAS-20 is used to characterize the various components. This discussion explores several software concepts and analysis techniques. The system is studied from both spatial accuracy and frequency domain points of view. The results of the calibration tests are examined and attention is paid to the meaning of these results. Data is supplied to support the results.

The general conclusions from the work covered in this thesis are presented in Chapter 5. These conclusions address the issues of usefulness and success and also discuss the problems encountered in the development of the various topics. Recom-
mendations are also made pertaining to possible modifications to existing software and/or hardware. Several additional applications for the position sensing system presented in Chapter 4 are explored to emphasize the usefulness of the sensor system.
Chapter 2: The Keithley Data Acquisition System (Model DAS-20)

The Keithley/Metrabyte/Asyst Model DAS-20 data acquisition system is a high-speed, multimode PC expansion board with A/D (analog to digital) and D/A (digital to analog) capabilities. The board contains several variable rate counter/timers and also supports eight-bit TTL compatible digital I/O. A block diagram of the DAS-20 is shown in Figure 2.1.

![DAS-20 Block Diagram](image)

Figure 2.1: DAS-20 Block Diagram.

2.1. Theory of Operation

The DAS-20 acquires data through an array of programmable input channels. These channels can be hardware configured as either 16 single-ended or 8 differential input lines. Each input channel is also software programmable to allow both unipolar
and bipolar signals as well as instrumentation gains that range from x0.5 to x100. This yields an effective range of input resolution between 4.88mV and 24.4μV.

The actual A/D conversions are performed by an AD774 successive approximation converter with an 8.5 microsecond conversion time. Coupled with a 1400 nanosecond sample and hold delay, the total data acquisition period is about 9.9 microseconds. Hence more than 100,000 samples per second can be achieved on an IBM PC or PC/XT compatible bus as well as many PC/AT compatible machines. On some IBM PC/AT compatibles, the conversion throughput is reduced (about 65,000 samples per second) due to hardware limitations. Even though the AT has a faster system clock, some AT machines use a 16-bit bus in which the clock runs at only 3 MHz as compared to 4.77 MHz on an XT's 8-bit bus. Since the DAS-20 is an 8-bit expansion card, the full 16-bit data path on an AT is not used which results in reduced conversion rates. With the advent of the more sophisticated EISA (Extended Industry Standard Architecture) systems with higher/variable rate data buses, the 100,000 samples per second rate can also be achieved.

In order to take samples from several channels and with variable gains the system employs what is called an ADC control queue. The queue is 2048 bytes of RAM that is used to control the instrumentation amplifier (which selects the input gain) and the analog input multiplexer (which selects the channel to be sampled). Since the queue resides in RAM, the actual channel/gain settings do not interfere with the overall sample rate. Hence complicated schemes of multi-channel/variable-gain scans can be easily
employed. Table 2.1 shows an example of the information contained in the sample queue.

<table>
<thead>
<tr>
<th>SAMPLE #</th>
<th>CHANNEL #</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>±10V</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>±10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0-1V</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>±10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0-1V</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>±10</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>0-1V</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>±5</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>±10</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>±5</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0-1V</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>±5</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>0-1V</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>±10.5</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>0-1V</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>±10.5</td>
</tr>
</tbody>
</table>

Table 2.1: Example of DAS-20 Scan Queue. [Keithley/Metrabyte/Asyst DAS-20 Manual, Page 5-3]

In order for proper timing to occur, the DAS-20 uses an AMD-9513A five-channel system timing controller. Three of the timers (used in a cascaded configuration) can be used to pace the A/D conversions and are driven by an on-board 5 MHz crystal-controlled oscillator. The remaining two timers (also used in a cascaded arrangement) can be used to pace the D/A converters, or can be connected to external ports for other uses.

Data acquisition can be triggered by either software command, external source, or on-board timer. Using the software command method lends itself to several limitations because the acquisition timing period can be interrupted by other PC system functions. Hence the software control method is only useful when data timing is not crucial. The external triggering option allows the DAS-20 to take data when commanded by another device or system. This could be useful in applications such as monitoring a
physical process. External sensors could trigger the DAS-20 to sample the state(s) of a process at different times. Triggering by an on-board pacer clock insures uniformly spaced samples. Also, the ability of the DAS-20 to use interrupt driven or DMA (Direct Memory Access) data transfers allows the entire acquisition system to operate in the background. This enables the computer to "simultaneously" run software controlled applications in the foreground during DAS-20 operation. It should be noted here that when the DAS-20 uses the sampling queue (as in a multi-input system) in conjunction with the ADC pacer clock that the actual sample rate for a specific channel is dependent upon the number of channels being sampled (see Table 2.1). Since the input lines are multiplexed on the DAS-20, the inputs are sampled one at a time with the period between each conversion being equal to the period of the ADC pacer clock. For example, suppose that the sampling queue is configured to acquire data from 5 channels as shown in Table 2.1 and the pacer clock is set at 100 Hz. Then channels 0, 1, 2, and 10 are each sampled at 12.5 Hz while channel 5 is sampled at 50 Hz.

Data transfer from the converter ports to PC memory is accomplished by one of three methods. The first method is software control. This method requires the data transfer to be a foreground task and precludes the use of the DAS-20 in the background. The other two methods as previously mentioned are interrupt transfer and DMA (Direct Memory Access) transfer. Both of these methods allow the transfer operation to occur in the background and thus increase the usefulness of the DAS-20. The interrupt transfer method requires an unused IRQ (interrupt request) line as well as the use of an interrupt service routine. Even though the interrupt service routine is a background operation, it
does require CPU time. The DMA method directly programs the 8237 DMA controller (or its equivalent) in the PC. The data transfer then takes place completely independent of the CPU. It is the DMA capability that allows the DAS-20 to acquire up to 100,000 samples per second. Using the other methods, the transfer rates cannot reliably exceed 5000 samples per second.

The DAS-20's two analog outputs (D/A converters or DACs) are hardware selectable for 0-10v, ±5v, or ±10v output ranges. They are 12-bit converters and are double-buffered to insure no transient glitching during output transition. The DACs can be controlled in much the same way as the ADCs except for the lack of anything similar to the control queue. Using DMA data transfer under pacer clock control, the DACs can achieve an update rate of 130,000 samples per second. Due to the fact that the DAS-20 has only two analog outputs, the card can support I/O for at most, a 16-input/2-output controller.

The DAS-20 also contains 16 TTL-compatible digital I/O lines configured as one 8-bit input port and one 8-bit output port. This gives the system the capability of communicating with external digital devices. As will be discussed in chapter 4, the digital lines may also be used to control the switching of analog devices.

2.2. Limitations and Their Solutions

In order to use the many features supported by the DAS-20, Keithley/Metabyte/Asyst has provided a set of software driver routines. However, the DAS-20 was designed primarily as a data acquisition system. The software routines provided by the manufacturer facilitate its use in this fashion. Hence, these routines are limited when the
hardware is to be used in a closed-loop control situation. The following sections discuss
the use of several of these drivers for achieving closed-loop control. Several methods
of overcoming the software limitations are also examined.

2.2.1. Using the Software Drivers Provided by Metrabyte

The author has programmed the DAS-20 using both C and FORTRAN. The
calls to the DAS-20 driver software (referred to as mode calls) are implemented as
functions in both languages. For convenience, any programming reference will be
specific to C throughout the discussion unless otherwise stated.

One of the most important features of the DAS-20 is its on-board timing
capabilities. These timers allow the user to accurately specify a sampling interval for the
ADCs or an output frequency for the DACs. Both timing functions are programmed in
a similar fashion and allow a timing range of 0.00119 Hz to 100 kHz (ADC timing) or
250 kHz (DAC timing). Both timers function by dividing the output of the on-board 5
MHz oscillator by the product of two 16-bit integer registers. Hence the frequency
resolution is decreased as the desired pacer frequency is increased. While it is important
to be aware of this fact, it yields no difficulty for the applications dealt with here.

In order to set up A/D or D/A conversions the programmer must decide what
kind of triggering and data transfer modes to use. For applications such as spectral
analysis which require only data acquisition, the DAS-20 would typically be programmed
using pacer clock triggering and DMA or interrupt data transfer. This yields jitter-free
sampling intervals which are required for frequency analysis of the data. The use or
need for a gating signal would be specific to the application. Now, if the user wishes
to implement a form of closed loop control in which the system acquires data, operates on the data, and issues an analog output one sample period later, the drivers supplied by the manufacturer will not lend a solution. The packaged software only allows the user to implement either timed data acquisition or timed data output but not both simultaneously. Therefore, a new set of drivers was created to make "simultaneous" I/O possible. They are discussed in the next section. Please note that for most uses, the PC is not capable of truly simultaneous activity. For the purposes at hand, "simultaneous" means that the events are very closely spaced in time. The closeness is relative to the operations being carried out. Hence, when two events are stated as being simultaneous, it means that the interval between them is small compared to the sampling interval being implemented by the system.

2.2.2. Custom Software for the DAS-20

There are three methods for implementing simultaneous I/O on the DAS-20. The first method makes use of the supplied drivers, and requires that the system be calibrated for each PC in which it is used. The two remaining methods are more elegant and make use of the facilities available within the PC itself. They include direct monitoring of the 8237 DMA controller and installation of an interrupt service routine.

2.2.2.1. Combining the Supplied Drivers into a Useable System

In order to implement a closed-loop digital control system, it is necessary to maintain simultaneous I/O. One method in which to achieve this using a single DAS-20 is to program the card for a series of sequential A/D and D/A conversions. These
conversions are configured to use pacer clock triggering with DMA data transfer to provide control over the sampling period as well as to facilitate the use of the CPU for other tasks during the data transfer sequence. The difficulty encountered here is that the DAS-20 cannot conduct DMA transferred A/D while sustaining DMA transferred D/A conversions. Hence, after each input sample, the software must cancel the DMA input mode and reconfigure the card for the DMA output mode. When the output sample is finished, the output mode must be canceled and the card reconfigured for DMA input once again. Figure 2.2 shows a flowchart of this activity. Note that the flowcharts do not include all error checking decisions and courses of action. These activities were omitted for the sake of size and readability. (The presentation here is to gain an understanding of the primary functions of the software without being concerned with all the tedious details. Complete details of the software are contained in the source code listings in the Appendix.) Now, the drawback of the above programming technique is that when the modes are switched on the DAS-20, the pacer clock timing is disrupted. Thus the frequencies with which to program the pacer clocks must be obtained empirically and are determined by the time required for the PC to make the I/O mode transition. Also, in order to make this a useful process, the computer must implement the controller during each sample period as well. The time required to implement the controller also affects the pacer clock frequency settings. Software has been written on a Compaq 386-20e Deskpro (20 MHz 80386 system) to perform these operations. The system was able to realize a 100 Hz sample rate for a SISO (Single-Input/Single-Output) controller of order 20. A 13TH order controller can operate at 200 Hz.
Figure 2.2: Flowchart of Switched DMA Programming Method
There is another difficulty that arises from this programming technique. Due to the necessity of constant switching between modes, and the requirement that the data acquisition operates via the channel and gain data stored in the sampling queue, it becomes evident that this technique is feasible only for SISO control. For a multi-input system, the inputs are multiplexed as discussed above. Thus, for any given "sampling instant" (defined here to mean the instant when all inputs to the controller have been updated to their current value) the data obtained from each individual channel is actually separated in time by the period of the ADC pacer clock. However, the pacer frequency is empirically determined as explained above and thus has no direct correspondence to the actual desired sampling rate. Hence, unless we are sure that the empirically determined ADC pacer frequency is large compared to the actual sampling rate, we will not be able to use classical control techniques in the system. The difficulty of developing new controller algorithms to take these timing issues into account greatly outweighs the benefits yielded from this method of programming. A method of multi-channel DAS-20 programming has been developed that overcomes this difficulty. This and other techniques will be presented in the following subsections.

2.2.2.2. Writing an Interrupt Service Routine to Avoid Data Timing Difficulties

One method of achieving reliably-timed multi-channel I/O and control using the DAS-20 is to write an interrupt service routine (ISR) or interrupt handler. In order to understand what an interrupt service routine can do, one must first have some background information on the PC and interrupts themselves.
The IBM compatible PC's (as well as most other personal computing systems) use what is called an interrupt driven operating system. An interrupt is a signal generated by either hardware or software that informs the CPU that a particular event has occurred. Each possible interrupt that can occur has a corresponding number associated with it called the interrupt number. When a specific interrupt occurs, the PC looks in a segment of memory called the interrupt vector table to find the starting address of the routine that is to service (or handle) the interrupt. Hence, the interrupt service routine is a piece of software that is executed only when the processor receives a signal to do so. At the time that an interrupt occurs, the CPU puts its current operations "on hold", executes the ISR, and then returns to finish working on the job it was performing before the interrupt happened. When no interrupts are being processed, the CPU executes whatever software the user commands. Now, in order to prevent interrupts from interfering with one another, they are prioritized according to their number. The lower the interrupt number, the higher the priority. Also, it is possible to mask the PC's interrupts on a selective basis by manipulating the interrupt mask register.

This type of processing allows the ISR's to be processed in the background while another application is being executed in the foreground. Hence the computer could implement the control algorithm in the foreground while performing I/O via an interrupt service routine in the background. This type of operation is a form of multi-tasking. True multi-tasking in a single processor environment (such as the IBM PC) is computer operation in which the system switches activity between two or more programs in a sequential fashion. A pre-specified number of system clock cycles are devoted to each
of the software packages. If a particular routine is to have higher priority, then a larger percentage of the cycles (each second) are allocated for that task. As mentioned above, some of the software drivers provided by the manufacturer use interrupt routines to perform certain operations, but these routines do not provide any simultaneous I/O capability which is necessary for closed-loop control. As a consequence a software package that allows virtually simultaneous 2-input/2-output I/O and control using ISR programming was written. The software was specifically designed to act as a forward path compensator in a 2-input/2-output system with the block diagram shown in Figure 2.3. For this particular implementation, the setpoint is generated within the computer hence the summing junction is within the computer as well. Its length is used to determine the length of time that the controller is to operate. This package uses the timing capabilities of the DAS-20 to generate an interrupt. The corresponding interrupt routine then performs all input/output functions through the DAS-20 at the hardware level. The controller actually operates in the foreground and uses several globally declared variables to communicate program status to the ISR. Figure 2.4 shows a flow
Figure 2.4: Flowchart of the ISR Programming Method.
chart of this method of programming. A step by step discussion of the ISR installation software follows. Please refer to Appendix I for a complete listing of the code.

As with any typical software, this code begins by declaring the global variables. Any variable declared as global will be directly accessible to the interrupt handler as well as to the foreground software. This yields a method by which the ISR can communicate with the foreground software. The pointers to the arrays that will be used to contain data are allocated here as HUGE (as specified in Microsoft QuickC version 2.5). The HUGE arrays may occupy more than one data segment but are restricted to meet a power-of-two boundary if they are larger than 128K. In other words, if a HUGE array contains more than 128K of data, then it must be allocated as $2^n$ KBytes where $n$ is an integer. Of course, these limits are also subject to the amount of available RAM. Microsoft QuickC version 2.5 is not directly capable of allocating storage above the 640K DOS base memory barrier.

The special functions necessary for the software are also declared. These functions include all the file I/O routines and the actual ISR (called DASISR). In order to be properly called, the function DASISR is declared (in C) as a type void interrupt routine residing in far memory. The routine is specified to lie in far memory so that the PC has the option of placing it in a code segment other than the one that contains the interrupt installer (main). This will make the software able to implement larger routines if necessary. The routine is declared as type void because it need not return an argument to the installing routine. Another special declaration that should be pointed out is the pointer that stores the existing interrupt service routine address found in the interrupt
vector table. This variable is also type void and is required so that the original contents of the interrupt vector table location corresponding to the specified interrupt can be restored upon program completion.

After all function and global declarations are specified, the program actually enters the main routine. The main begins by allocating all local variables and then by loading the setpoint vector(s) and state-variable representation for the system into globally visible dynamically allocated arrays. All of the data files are written to conform to the MATLAB double precision data file format. This is because the MATLAB software package is used as the primary data analysis and controller design platform. Error checking is used to determine whether the files exist and are in the proper form. Before beginning the actual acquisition/control sequence, the software dynamically allocates memory (also globally visible) to accept the output data. Once again, error checking is used to determine if enough memory is available to perform this task. After the output storage allocation has been completed, the software begins working with the DAS-20. The first operation uses an existing DAS-20 mode call to initialize the card. Recall that a mode call is a call to the driver software supplied by Keithley/Metrabyte. Initialization is required and consists of declaring the base port I/O address for the card and setting up the proper interrupt and DMA levels to be used if necessary. The port I/O base address is the starting address of a 16 element block of ports through which all communication to the DAS-20 is done. As with all portions of this code, error checking will terminate the program if the initialization mode call fails for any reason. The next step is another mode call that initializes the DAC pacer clock to the user-specified fre-
frequency. This frequency value is obtained from the command-line that executed the software. Once the pacer clock has been initialized, the interrupt handler is then installed.

The actual installation of the ISR only requires that the current address in the interrupt vector table be replaced with the address of the properly declared handling routine. Microsoft QuickC was used for the development of this software and provides several functions to facilitate this operation. The first action that must be taken is to disable all interrupts. This prevents the interrupt corresponding to the portion of the interrupt vector table that is being modified from being called while the ISR address is manipulated. The old address is then stored in the specially prepared pointer for later recall and the new address put in its place. Once this has been completed, the system is ready to be activated. Since the system will operate on the interrupts generated by the pacer clock, then starting the pacer clock will effectively start the entire I/O system. The means by which the pacer clock is configured and started involves port level I/O directly to the DAS-20. The software in Figure 2.5 shows how this is done. The first line com-

```c
outp(base+7,0x01);  /* Update the AMD-9513A Data Pointer Register to point to Counter 1 Logic Unit (CTR 1) */
outp(base+6,0xa1);  /* Set LSB of AMD-9513A MODE R (Retriggerable One-shot) */
outp(base+6,0x0b);  /* Set MSB of AMD-9513A MODE R */
outp(base+7,0x02);  /* Update the AMD-9513A Data Pointer Register to point to the Counter 2 Logic Unit (CTR 2) */
outp(base+6,0x21);  /* Set LSB of AMD-9513A MODE E (Rate Generator) */
outp(base+6,0x0b);  /* Set MSB of AMD-9513A MODE E */
outp(base+7,0x43);  /* LOAD CTR's 1 and 2 */
outp(base+7,0x23);  /* ARM CTR's 1 and 2 */
```

Figure 2.5: Code demonstrating how to start the pacer clocks.
mands the AMD-9513A to update the data pointer register to point to the mode register in the counter 1 logic group (refer to AMD-9513A reference guide for details on programming the AMD-9513A). This register determines which timer registers will be visible when communication occurs between the AMD-9513A and the CPU. The next two commands send the 16-bit mode control data to the counter mode register in order to properly configure counter 1. The next three lines perform a similar activity for counter 2. Both counters are then simultaneously loaded and then armed by the remaining two lines of code. Once armed, the clocks begin running. As mentioned earlier, these counters run in a cascaded arrangement to implement the timing functions. Hence, both counters are required to implement the acquisition timing function. After starting the clocks, the I/O and control software is on-line and able to execute in the background. The CPU is also able to continue running other software in the foreground. In the case of this particular package, the foreground operation waits for the n\textsuperscript{TH} sample to be taken and then proceeds to execute one iteration of the state-variable controller algorithm. This checking is done via a global DOK variable that is set (DOK = 1) when the I/O routine has acquired the input data, and is cleared (DOK = 0) when the system is waiting for the I/O routine to operate. Another global variable (ISRERROR) is used to indicate whether the ISR was called during operation of the controller. This can occur if the controller requires more than one sample period to complete its task. In this event, the software will indicate that a timing fault has occurred and halt execution. Providing that a timing fault does not occur, then DOK is cleared upon completion of the control routine, thus preparing for the next cycle. When the next interrupt occurs the ISR
executes once again. The output data is sent to the DACs and the A/D converters perform their operation. The DOK variable is then set to indicate that the next sample is ready. The system continues in this fashion until all points in the setpoint array have been used. When the entire I/O and control sequence has been completed, the foreground software then takes the ISR off-line by restoring the original interrupt vector to its proper place in the interrupt vector table. Once again, the system interrupts are disabled while this change takes place. The acquired data is then stored to disk for use by MATLAB and the program terminates.

The discussion of interrupt service routines would be far from complete if terminate-and-stay-resident (TSR) programs were not discussed. TSR's are probably one of the most widely used forms of ISR implementation, but they require some sophisticated memory handling techniques and a much more than casual understanding of the DOS operating system. A TSR is a program (typically some sort of interrupt service routine) that is installed in memory by a foreground routine (called the installer) in a similar manner to that discussed above. The difference between the above implementation and a TSR is that the foreground installer for the TSR terminates upon successful installation of the service routine. The TSR then remains active until it is intentionally removed, and it is possible to develop a TSR that can remove itself when certain criteria are met. This type of implementation has the advantage that the foreground task that installed the TSR does not continue executing while the TSR operates. Hence the CPU is free to execute other software that is unrelated to the TSR. Recall, in the above discussion, the ISR installation software kept executing in the
foreground to determine when to deactivate the service routine by restoring the original interrupt vector. The reason why a true TSR type package was not developed using the DAS-20 is one of memory allocation. When a program executes on a PC, the amount of memory required to execute the main routine and all of its subroutines or functions (including any interrupt service routines) is set aside by DOS and shielded from use for any other function while the program is executing. When the program terminates, all of the memory is reallocated as useable by DOS. Hence, in order to construct a TSR, one must specifically allocate the memory in which the TSR is to be held in a manner such that it is not reallocated upon termination of the installer. Once this has been completed, then the TSR will be "safe" from being overwritten by any package that conforms to the rules by which DOS manages memory. Since not all DOS compatible software conforms to these standards, then such a package executed in conjunction with the TSR may illegally use the TSR’s memory. While any code of this sort is an example of poorly written software, these types of problems are frequently encountered. These difficulties typically result in a system crash. Hence, it is up to the programmer to incorporate protection schemes into the TSR itself that reduce the chances that such an event will occur. It should be noted that it is not possible to construct a TSR that is completely "bullet-proof" from any operation that may interfere with it. The methods by which a programmer can insure a high degree of protection for the TSR are themselves a lengthy topic and are beyond the scope of this discussion. Hence, while the use of a TSR could yield some advantages, it is not necessary to go to such trouble for the purpose at hand.
The following paragraphs discuss the operation of the interrupt service routine and the state-variable control algorithm. It is these two tasks that are the heart of the interrupt driven data acquisition/control software package.

The interrupt service routine that implements "simultaneous" A/D and D/A is called DASISR. This routine is shown in Appendix I (attached as a subroutine to ISRSVAR). The routine begins by declaring the local variables that are required. These variables are used in intermediate calculations carried out within the service routine. The first actual command is to enable the system interrupts. This ensures that any higher priority interrupts can be executed while the ISR is being processed. The reason that the higher priority interrupts may be unable to execute arises from the fact that there are many algorithms used by the PC that suspend certain interrupts. Since the operation of DASISR is not dependent upon the enabled status of any other interrupts, they are all enabled. This is a common programming practice to help guarantee proper ISR execution and ensure that other system functions such as those that update the system clock are not interfered with. The next step in the ISR checks to see if the specified interrupt was generated by the DAS-20’s onboard timer. This is done by reading the interrupt mode control register (IMCR) on the DAS-20. Reading the register is done in a similar fashion as the port level I/O in 19. Bit 2 of the IMCR will be set if the pending interrupt was generated by the DAS-20. If the bit is not set, the ISR returns control to the main. If set, the pending interrupt status bit (bit 2) of the IMCR is then cleared by a write to the register. This write makes no other changes to the IMCR. Next, the ISR checks to see if DOK = 0. If so, then the state-variable controller has
successfully finished its operation and the current data is valid and waiting to be manipulated by the ISR. If DOK = 1, then the output data is not valid and the I/O routine should not execute. This will only occur if the state-variable controller is unable to complete one iteration in a sampling interval. When this occurs, the variable ISERROR is set to 1 to indicate that a timing fault has occurred. The ISR then terminates and returns control to the main where the error is handled.

Supposing that a timing fault has not occurred, the ISR proceeds to sample new data and sends the output data to the DACs. The hardware level code that commands the DAS-20 to sample the input lines is shown in Figure 2.6. Refer to the Keithley Metrabyte DAS-20 user’s guide for a complete register map of the DAS-20. The first line of the sample sets bit 4 of the ADC (analog-to-digital control) register which tells the DAS-20 to insert a small delay before starting a conversion. This ensures that the instrumentation amplifiers have had sufficient time to settle before beginning the conversion. While timing problems in conjunction with the analog-to-digital conversions

/* Get Channel 0 Data */
outp(base+3,0x10); /* Sets ADC delay timer */
outp(base+2,0x13); /* Sets proper CHANNEL/GAIN queue with proper data and declares end-of-queue (EOQ) */
outp(base+4,0x03); /* Writes to IMCR to clear any pending INT, disables INT’s from DAS-20, and sets up IMCR to reflect when an ADC end-of-conversion takes place */
outp(base+0,0x00); /* Manual conversion start */

get_stat1: /* 1st conversion status check label */
stat = inp(base+4); /* Get conversion status (check IMCR for ADC end-of-conversion */
if(!stat) goto get_stat1; /* If ADC is not done, wait and check again */

samplo = inp(base+0); /* Get low data byte - also contains channel and gain data */
samphix = inp(base+1); /* Get high data byte */

Figure 2.6: Code to perform manual A/D conversion on DAS-20
are not a problem at lower sampling rates, the decision to use the delay at all times was made to preclude any later difficulties if higher rates are used. The second line of code writes the value 03 (03 hex) to the ADC instruction queue. The data specifies that channel 0 is to be used with an instrumentation range of ±10 volts (bipolar) and a gain of x0.5. Also, the end-of-queue bit is set, which informs the control logic that this is the only sample to be taken. Thus the sampling queue index pointer will be returned to the beginning of the queue when the sample has been converted. The third line then writes to the interrupt mode control register (IMCR). This data clears any other pending interrupts from the DAS-20, disables any new DAS-20 interrupts and specifies that the DAS-20 generate an interrupt signal on an ADC end of conversion. This is done so that the IMCR can be used to monitor the completion of the conversion. The interrupt status bit (bit 4 of the IMCR) will be set when the A/D conversion is finished. The actual conversion is manually initiated by a write to the base port address of the DAS-20. The three lines of code following the get_stat1 label shown in Figure 2.6 form a loop that waits for the interrupt status bit to be set. When the bit is set, the conversion has been completed and the data can be read from the ADC registers. Now, since the source of the DAS-20 interrupt has been changed from the pacer clock to the end-of-conversion signal, the conversion timer is unable to generate an interrupt. However, the length of time that this condition exists is very short when compared to the sample period. Also this configuration is used early in the sample interval and the interrupt source is restored to the timer long before the clock can issue another rising edge. Furthermore, as stated above, the ability of the DAS-20 to send its interrupt signal to the CPU is also disabled
during this time. Hence this activity is transparent to the I/O timing functions. The final lines of code read this data in a LSB/MSB (least significant byte, then the most significant byte) format. Since the DAS-20 uses 12-bit converters, the 4 least significant bits of the LSB are used to store a copy of the channel and gain information with which the data was taken. For the applications discussed here, the instrumentation amplifier is always configured in bipolar mode, hence the sampled data is formatted in 2’s complement binary form. If unipolar operation is specified for some reason, the data would be in standard binary format. Once the data has been retrieved from disk in the two byte format, it can be converted to double precision form for use in the controller. A similar activity is performed for the second input channel used in the I/O and control software. When the sampling of the input channels has been completed, the IMCR is reconfigured to again acknowledge interrupts from the system pacer clock.

Following the sampling processes, the DAS-20 must send data to the analog output ports. The data must be converted to two’s complement form and sent to the D/A converters. The actual code that performs the output task for channel 0 is shown in Figure 2.7. These lines perform writes to the analog output ports. The LSB is written first, followed by the MSB. When the MSB is transferred, the data is latched into the DAC and the conversion follows. Recall, the DACs are double buffered to prevent transient errors. Once the data has been written to the output ports, the ISR sets DOK

<table>
<thead>
<tr>
<th>Line 1</th>
<th>Line 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>outp(base+8,lowdbyte);</code></td>
<td>/* Sends low data byte to the DAC */</td>
</tr>
<tr>
<td><code>outp(base+8,hidabyte);</code></td>
<td>/* Sends high data byte to DAC and initiates the conversion */</td>
</tr>
</tbody>
</table>

Figure 2.7: Code demonstrating how to send data to the DACs.
= 1 to inform the main routine that the I/O has been successfully accomplished. The interrupt handler then performs another port write. This single line issues an end-of-interrupt message to the CPU which then performs the necessary stack operations to return control to the main.

The heart of the main routine is the state-variable controller. This is a segment of code that effectively implements the linear set of equations represented by (2.1) and (2.2).

\[ x(k+1) = Ax(k) + Bu(k) \]  \hspace{1cm} (2.1)

\[ y(k) = Cx(k) + Du(k) \]  \hspace{1cm} (2.2)

As previously mentioned, the number of inputs and outputs is limited to two each. However, the dimensions of the other elements are limited only by system memory and the speed at which the computer can perform the operations. For this implementation, a 2-input/2-output controller of order 20 was executed successfully at 98 Hz.

As can be seen from the above discussion, the use of an interrupt service routine yields a large degree of flexibility when designing applications for the IBM PC. In fact the DOS operating system is equipped with many built-in memory management and process handling functions as well as a wealth of callable software tools to support tasks that range from simple port I/O operations to complex interfacing schemes. It is the above method of programming that has yielded the most flexibility from the PC and
DAS-20 system. However, there is yet another method of programming that should be considered, even if just briefly.

2.2.2.3. Combining the DMA Capabilities of the DAS-20 with Direct PC Port I/O

While it is not as versatile as the one just discussed, this method of I/O has merits that should be noted for the sake of completeness. This method involves using pacer clock controlled data acquisition with DMA transfer coupled with software controlled data manipulation and output. The technique was not fully developed when it was discarded in favor of ISR programming. Hence this discussion is limited to the results that were obtained before work was discontinued on this method. The theory was that the DAS-20 would be programmed to acquire data at a specific rate. This could be done easily using the drivers supplied by the manufacturer. DMA data transfer was chosen so that the acquisition cycle would run completely independent of the CPU. Hence the CPU could monitor the DMA transfer status to wait for each conversion to be completed. This was also done using a manufacturer supplied routine. After each data transfer cycle, the CPU would fetch the newly acquired data, execute the state-variable control algorithm, and output the result via a direct DAC port write. A flowchart of this concept is shown in Figure 2.8.

Before work was terminated on this idea, a working I/O platform was developed. The PC was able to carry out "simultaneous" I/O using a single channel, and the data rate was easily changed (as opposed to the switched DMA method of section 1.2.2.1 in which each data rate had to be experimentally calibrated). The integration of the state-variable controller was not attempted before this method was discontinued. Hence, no
Figure 2.8: Flowchart of Paced DMA Programming Method
overall speed characteristics were determined. However, data rates for the functioning I/O software could reliably reach 2000 Hz before the CPU failed to keep up with the DMA transfer rate. This rate would be expected to fall dramatically if the controller were included in the software loop.

Development of this method was aborted for several reasons. The major drawback of this method is that it denies the use of anything but a SISO controller for the same reasons as the switched DMA method discussed in section 1.2.2.1. Also, it was found that the method of monitoring the DMA transfer count was prone to return errors. The reason for this is that the DMA transfer count register (a 16-bit register) is accessed via a single 8-bit port on the PC. Hence, in order to determine the DMA count status, the register had to be read twice. The first read would yield the LSB and the second the MSB. Now, there is no protection from reading the DMA counter during a transition, so the data that is received need not necessarily be correct. It is possible to use multiple scans of the DMA count status, but doing so uses more of the time that the CPU requires to implement the control algorithm and write the data to the output port. Hence while this method possesses some potential speed advantages over the other methods due to its method of acquiring data, the lack of a MIMO (multi-input multi-output) capability makes this method unsuitable for the tasks at hand.
Chapter 3: Development of a Senior Control Laboratory

This chapter discusses the development of an educational laboratory. This laboratory is intended for senior level control students and focuses on various aspects of control analysis and design. The instruction is organized from a discrete-time point of view and allows students to perform tasks ranging from component modeling and system identification to compensator design.

3.1. Laboratory Hardware

This section discusses the hardware used in the laboratory. The purpose and basic operation of each component is covered and relationships are developed between these physical elements and the elements contained in the block diagram of the system.

3.1.1. The Modular Servo System

The system under study in the laboratory is the Modular Servo System MS150 Mk2 from Feedback Instruments Limited. A block diagram of this system is shown in Figure 3.1. This is a DC servo-motor system that is designed to allow experimentation.

![Figure 3.1: Block Diagram of Basic Servo System](image-url)
with various types of position and/or rate control. The components of this diagram are discussed in the following paragraphs. Section 3.3 will discuss the experiments in which the various values for these components are determined.

$K_{in}$ and $K_{out}$ represent the input and output potentiometers. Dimensionally these devices represent a change from angular position (degrees) to volts. These devices allow the user to create a position setpoint (by manually setting the input pot) and observe the output (by watching the output pot). The input pot is removed in the sampled-data case, when the computer is used to generate a voltage setpoint that corresponds to an angular position setting.

The summing junction is implemented via a 741 op-amp voltage summing/- differencing circuit. The device allows variable gain settings for different uses. Here we use it in its unity gain form.

$K_p$ represents the gain of the pre-amplifier used by the system. As will be made clear in later discussion, the saturation of this device presents one of the most difficult problems encountered in the laboratory.

$G_1(s)$ is a first-order transfer function that represents the dynamics of the servo-motor from input (voltage) to the velocity output (revolutions per second - rps). This transfer function is of the form:

$$G_1(s) = \frac{K_1}{s\tau + 1}$$

(3.1)

where $K_1$ represents the final speed of the motor due to a unit step input and $\tau$ represents the time constant of the motor (the time required to reach 63% of the final velocity $K_1$).
\( G_2(s) \) is an integrator represented by:

\[
G_2(s) = \frac{K_2}{s}
\]  

(3.2)

where \( K_2 \) is the constant required for dimensional consistency. The output angular position is measured from a secondary output shaft with a 1:30 gear ratio with that of the primary motor shaft. Also, since the output of the previous block \( G_1(s) \) is referenced in revolutions per second, we must convert from revolutions to degrees for the output potentiometer.

The \( K_{\text{tach}} \) block represents the tachometer constant in volts per rps (revolutions per second). For this diagram, the constant represents the dimensional conversion from revolutions per second to volts.

\( K_v \) and \( K_p \) are simply the feedback gains for the velocity and position measurements respectively. They are implemented as simple potentiometers, hence each gain can range from 0 to 1.

3.1.2. The DAS-20 and PC

Ohio University originally purchased the DAS-20 and Compaq PC systems for use as the control system in the senior laboratory. The flexibility of the system immediately lended itself for use not only in implementing the control algorithms but also in performing automated calibration experiments on the servo system components. Chapter 2 covers the use of the DAS-20 for such purposes so further discussion here is not necessary.
3.1.3. The I/O Buffer Circuits

In order to protect the analog output channels on the DAS-20, a 2-channel unity-gain buffer circuit was constructed. This circuit will source a maximum of 25 mA which is sufficient to drive any portion of the servo-motor system circuitry except the motor itself. However, this is not a problem because the motor drive electronics are integrated into the servo-amplifier and thus are not directly accessible.

3.2. Laboratory Software Development

The MATLAB software package was used throughout the laboratory as the primary analysis and design platform. FORTRAN software was written to interface with MATLAB in order to carry out the various calibration and control functions. The software was written in order to function with MATLAB in a manner that is transparent to the user. Hence, the operations that use the DAS-20 appear to be functions that are inherent to MATLAB. These packages are discussed in the following paragraphs.

The software that performs the compensator task uses the switched DMA method covered in section 2.2.2.1. Since the theory behind this software has already been examined, it needs no further discussion here. A listing of the source code, entitled DASMAIN, is included in Appendix VIII.

The software used to perform the various characterization experiments in the laboratory is called DCCALIB. This software was designed for single channel (1 input and 1 output) operation. It performs the same basic operations as DASMAIN except that no controller is involved. Hence, the software simply sends data to one analog output port and samples data from one analog input. A modified version of this software (called
DCCALIB2) was written in order to allow data to be acquired from two analog sources while still sending data out on one output channel. This software facilitates calibration of the pre-amplifier unit in the servo system. A listing of both source code files is included in Appendix IX and X.

It should be noted here that the FORTRAN software is rather limited due to the fact that it employs switched DMA programming. Software written in C using the ISR programming technique is being developed to replace this code. Also, the new software uses more advanced file I/O to greatly reduce the time required for data storage on disk. The FORTRAN method uses ASCII text files to store data while the C method uses a MATLAB compatible binary file format. Other features such as dynamic memory allocation will also enhance the performance of the new software.

3.3. The Experiments

This section will discuss the experiments that have been conducted in the senior laboratory. The experiments and their results are examined sequentially and some general conclusions are reached at the end of this section. Note that most of the experiments rely heavily on the capabilities of the DAS-20.

The first experiment requires no discussion because it includes a battery of exercises intended to familiarize the student with the MATLAB software package. The exercises culminate in the analysis of a contrived linear system.

The second experiment is designed to familiarize the student with the servo system. The system is connected as shown in the above block diagram and the student is encouraged to spend some time getting a feel for how the system operates in a purely
analog sense. The function of each component is presented at this time. Once a certain degree of familiarity is achieved, the system is disassembled in order to study the components on an individual level. Using the computer (and DAS-20) when necessary, the students are instructed to find a transfer function model of each component. These components are discussed in the following paragraphs.

The first components to be analyzed are the input and output potentiometers. The input pot provides a voltage setpoint that is proportional to the angular displacement of the dial. The proportional constant is approximately 0.1 volts per degree of rotation. Since the DC supply is ± 15 volts, it is evident that the setpoint is limited to ± 150 degrees. A rotational stop is inserted here to prevent the device from rotating past these points. The output potentiometer is similar in that it yields 0.1 volts per degree rotation as well. However, since this device is connected to the secondary output shaft (with a 1:30 turn ratio to the primary servo shaft), it is allowed to rotate freely in either direction as the motor turns. The regions from +150 to +180 degrees and -150 to -180 degrees yield a fixed ± 15 volts (with the sign corresponding to the sign of the angular displacement). With this device, the output position of the system can be measured. The dial of the output pot is marked with a timing pattern that allows the angular velocity of the secondary shaft to be measured as well. This is useful when characterizing the tachometer.

The attenuator units are simply calibrated potentiometers that are used to attenuate the position and rate feedback signals. As mentioned before, they provide a gain ranging from 0 to 1.
The tachometer constant (represented by $K_{tach}$) was determined by running the motor at a known, constant speed and measuring the tachometer voltage. For this system $K_{tach}$ is approximately 0.1515 volts per rps (revolution per second).

In this experiment, the summing amplifier was calibrated to ensure zero DC offset. This was done by grounding the input terminals and adjusting the offset control until an output of zero volts was reached.

The next component in question is the pre-amplifier unit. The gain of this unit was tested by commanding the DAS-20 to issue a linear voltage ramp to the pre-amp input while sampling the output. This test revealed that the gain was approximately 50. Due to such a large gain, and the fact that DC supply is limited to $\pm$ 15 volts, the pre-amp saturates at an output of $\pm$ 13 volts with any input whose absolute value is larger than about 0.25 volts. Now, due to the configuration of the system, the pre-amp acts as a gain on the error signal. Since the error signal represents the difference of the actual position and the setpoint position (assuming the motor is not currently rotating, which ensures that the tachometer output is equal to zero), the pre-amp saturates for any angular step greater than 2.5 degrees. This saturation nonlinearity introduces many difficulties in controlling the system. In fact, the system bandwidth cannot be increased any further to decrease the system settling time. In order to prevent this saturation from occurring with such a small error signal, a voltage-divider with a gain of $1/6$ was constructed and inserted between the summing junction and the pre-amp. This reduces the loop gain and decreases the overall system performance. The end result is that a compensator can be
used to recover the lost performance without pushing the system further into saturation. This effectively demonstrates the usefulness of phase-lead and PID techniques.

![Continuous Model vs Actual Response](image)

**Figure 3.2: Actual and Modeled Servo-Motor Velocity Step Response**

The final components to be characterized are the servo-amp and servo-motor unit. The dynamics of these units are determined collectively because the electronics of the two units are integrated into one package. As discussed above, two parameters of the servo-amp/motor system must be determined in order to generate a first order model. These parameters are the steady-state angular velocity response (\(K_v\)) to a step input, and the time constant \(\tau\) which represents the time required to spin up to 63% of the steady-state velocity. Figure 3.2 shows the velocity step response of the servo-amp/motor unit.
From this data, it was found that $K_1$ is 8.1670 rps and $\tau$ is approximately 0.9305. From this, a continuous model was found to be

$$G_1(s) = \frac{8.1670}{0.9305s + 1}. \quad (3.3)$$

Figure 3.2 also shows the step response of the continuous linear model. The step response of this model matches that of the actual system quite well. It should be noted that the actual step response was generated by using a 5 volt step instead of a 1 volt step. This was done because a 1 volt input to the servo-amp was not large enough to overcome the stiction in the motor. Hence, linearity was assumed in reducing the five-volt step response to a unit step response for this characterization.

The value for $K_2$ in (3.2) comes from the conversion from revolutions to volts (where 1 volt corresponds to 10 degrees of rotation) as well as the 1:30 gear ratio in the output shaft of the motor. From this it is found that $K_2 = 1.2$.

The third experiment allows the student to put the model pieces together into an overall transfer function. For $K_v = 0.08$ and $K_p = 0.6$ the transfer function shown in (3.4) is obtained. The student is then directed to compare the step response

$$G(s) = \frac{87.7732}{s^2 + 1.9612s + 52.6639} \quad (3.4)$$

characteristics of the actual system to that of the model. Various feedback gain settings are used in order to allow the student to study their impact on the system.

The student quickly finds that $K_p$ affects both $\zeta$ and $\omega_n$ in a manner such that the product $\zeta\omega_n$ is constant while $K_v$ affects only $\zeta$. Figure 3.3 contains the step response.
of the system for $K_v = 0.08$ and $K_p = 0.6$. Since the experimental step response is that of a sampled-data system, it only makes sense that a discretized version of the continuous linear model is used. This transfer function, denoted $G_s(z)$ to indicate that it comes from discretizing $G(s)$ (with $T = 0.01$) is

$$G_s(z) = \frac{4.358z + 4.330}{1000[z^2 - 1.975z + 0.981]}.$$ \hspace{1cm} (3.5)

The step response of the discretized linear model is also shown in Figure 3.3. As is seen, there are marked differences between the responses. Most of this comes from the nonlinear stiction in the servo-motor as well as the saturation in the pre-amp unit.
Experiment four allows the student to experimentally determine a frequency response estimate of the system. This is done via spectrum estimation in which a gaussian distributed white noise signal is used as the input to the system and the resulting response is sampled by the DAS-20. The input and output data vectors are then manipulated by the SPECTRUM command in MATLAB. SPECTRUM performs power spectrum estimation using the Welch method. The time-domain record contains 10,000 points sampled at 100 Hz. Various FFT lengths are used in order to demonstrate the effects of averaging the data. Figure 3.4 contains the experimentally estimated magnitude response of the system along with the magnitude response of the discretized model. Figure 3.5 shows the corresponding phase data. This estimated response was generated using 1024 point FFT's averaged over 10,000 points (with zero padding) and

![Figure 3.4: Experimentally Estimated and Modeled Magnitude Response](image_url)
Figure 3.5: Experimentally Estimated and Modeled Phase Response

512 point overlapping. As can be seen, the frequency response of the discretized model differs in several ways. In particular, the resonant peak of the discretized model is more pronounced and at a lower frequency. These factors are a direct result of the nonlinear characteristics of the actual system.

The fifth experiment involves system identification using the transfer function determination code (TFDC). This code was developed originally at Mississippi State University by Dr. Jerrel Mitchell and Dr. Dennis Irwin and further enhanced in the Electrical and Computer Engineering Department here at Ohio University. TFDC is used to determine a transfer function given an experimental frequency response (like the one from experiment four). The basic principle of TFDC is that it will develop an N\textsuperscript{th} order transfer function (N is chosen by the user) of a system by calculating a least
squares curve fit to the frequency response data. The student is instructed to generate a transfer function via this method and compare its frequency response to the experimentally determined data. The transfer function $G_{TF}(z)$ is

$$G_{TF}(z) = \frac{3.533z^2 + 0.297z + 12.80}{1000[z^2 - 1.933z + 0.944]}.$$  \hspace{1cm} (3.6)

Figure 3.6 contains the experimentally estimated magnitude response and the magnitude response obtained from the transfer function generated by TFDC. Figure 3.7 contains the corresponding phase responses. Clearly, this frequency response from the model matches the experimentally derived system data better than the component model response. A comparison of the step responses is shown in Figure 3.8. Since the frequency responses match so well, and the step responses still differ as much as they

![Figure 3.6: Experimentally Estimated and TFDC Magnitude Response](image-url)
Figure 3.7: Experimentally Estimated and TFDC Phase Responses

Figure 3.8: Actual and TFDC Step Response
do, it is concluded that the spectral estimate does not characterize the system completely. This comes from the fact that the spectrum estimation theory assumes a linear system. This is clearly not the case here. The TFDC algorithm appears to match the system natural frequency quite well but fails to accurately determine the system damping. Hence, it is made clear that while this method is limited in capability, it is possible to obtain a reasonable transfer function approximation of the actual system.

The sixth and seventh experiments involve the design of digital compensators to enhance the performance of the servo system. In these experiments, the entire servo system becomes the plant and the DAS-20/PC system is used to close a loop around the plant. The resulting block diagram is shown in Figure 3.9.

![Figure 3.9: Block Diagram of Compensated System](image)

Experiment six focuses on two first order designs. The first design demonstrates the ability to reduce the system settling time by increasing the system bandwidth. Steady-state error constraints are not dealt with. Using classical design criteria, the settling time of the system is to be reduced to 0.6 seconds and the percent overshoot restricted to less than 30%. It is decided to implement a minimum phase compensator that introduces 4.91 dB gain and 30 degrees phase at a crossover frequency of 20 radians per second. The transfer function for this compensator is
A plot of the closed-loop step response is shown in Figure 3.10. It is observed that the settling time specification is met but the percent overshoot constraint is not. The failure to meet the percent overshoot specification is due primarily to the nonlinear stiction within the servo-motor.

\[ D_A(z) = \frac{-3.8327z + 3.4193}{-1.4133z + 1}. \] (3.7)

A plot of the closed-loop step response is shown in Figure 3.10. It is observed that the settling time specification is met but the percent overshoot constraint is not. The failure to meet the percent overshoot specification is due primarily to the nonlinear stiction within the servo-motor.

Figure 3.10: Step Response of Settling Time Compensated System

The second design demonstrates concepts relating to steady-state error. Here, the dynamic constraints are ignored and the steady-state error to a ramp constrained to be less than 25%. This yields a type-I system and forces the steady-state error to a step input to be zero. A compensator to do this is
$D_b(z) = \frac{0.0263z}{z - 1}.$ (3.8)

Figure 3.11 shows the step response of the compensated system. Note that zero steady-state error is not achieved. In fact, the system does not settle at all. This is due to the stiction in the servo-motor. When the system stops close to 1 volt (10 degrees), the integrating operation of the compensator eventually energizes the motor enough to overcome the stiction. This forces the motor to rotate past 1 to a point roughly equidistant on the other side. In time, this event occurs again and forces the motor back to its original position. This stiction-induced limit-cycling effect cannot be overcome.
with a compensator such as the one in (3.8). However, the mean value of the oscillation is one which indicates that in the linear case, zero steady-state error would be achieved.

It should be noted that the zero steady-state error constraint and the dynamic constraints of a 0.6 second settling time and 30 percent overshoot can not be met using a first order compensator. To meet the specifications, the compensator would have to introduce more than 90 degrees of phase at the cutoff frequency.

A PID compensator is developed in experiment seven. This design enables the student to meet the combination of steady-state and dynamic constraints not attainable with a first order controller. Using the same design constraints as those from experiment six the following compensator is found:

\[ D_c(z) = \frac{7.0031z^2 - 12.5696z + 5.6065}{z(z-1)} \]  

(3.9)

The closed-loop step response obtained from implementing this controller is shown in Figure 3.12. As can be seen, the percent overshoot specification is reached. The settling time constraint is nearly reached as well. This specification could be more easily reached if we could make larger modifications to the system bandwidth. Elimination of the pre-amp saturation would allow this. As for the steady-state error, we see that it is non-zero. However, we know that this is caused by the nonlinear stiction in the motor.

Even though we cannot exactly meet all of the specifications with the PID controller, it sufficiently demonstrates the theory behind such compensation techniques. If the servo system were more linear, then these techniques would provide more accurate
solutions. Modification of the system as well as the development of nonlinear design techniques may become the topic of further development.
Chapter 4: The Infra-Red Multi-Position Sensing System (IRMPSS)

This chapter covers a practical application of the DAS-20 and its support software. The application is the characterization of an infra-red multi-position sensing system. The chapter begins with a description of the system hardware and continues to discuss the characterization tests with some emphasis placed on the development of the software used to make the experiments possible. The chapter concludes with a discussion of the results of the calibration tests.

4.1. System Hardware

This section discusses the components that make up the infra-red multi-position sensing system (IRMPSS). Primary emphasis is placed on the detection device itself as it will be seen that the remaining components are much less complicated and require a more limited discussion.

4.1.1. The Hamamatsu S-1200 Large Area Position Sensitive Device

The detection device used in the IRMPSS is the S-1200 infra-red detector manufactured by the Hamamatsu Photonics Corporation. The S-1200 is a 13 x 13 mm square silicon detection surface mounted in a ceramic case under a glass window. This type of detector is called a tetra-lateral type and has four electrodes on the front.

Figure 4.1: Plan View of Detector
surface as shown in the plan view of Figure 4.1. A cross-section of the surface (Figure 4.2) shows the photodiode structure. This structure consists of an insulating layer (I-layer) sandwiched between a P-doped layer (top) and an N-doped layer (bottom). The entire structure is supported by a planar silicon substrate (not shown). An equivalent circuit for the detector is shown in Figure 4.3.

The S-1200 operates on the principle of optical charge deposition. Light of the proper wavelength (320 - 1100 nm) falling on the detector surface creates an electric charge at the incident position. This electric charge travels through the resistive P-layer and is collected by the electrodes. Now, the resistivity of the P-layer is uniform, so the photocurrent collected by a particular electrode is inversely proportional to the distance between the incident position of the light and that electrode. Thus it is possible to measure the photocurrents and obtain a two dimensional indication of the position of the incident light. Thus far, the discussion has assumed that the light strikes the detector at a single point (of zero dimension). However, in reality the spot of light will have a finite non-zero dimension. Careful examination of the principle by which the device operates...
shows that this fact does not present a problem. Since the electrodes collect current
generated by the charge deposited by the incident light, it is evident that the current
entering any particular electrode is actually proportional to the distance between the
electrode and the centroid of the light spot. The centroid mentioned here is the centroid
from an energy point of view. Hence the position data obtained from the device is
independent of the focus of the light source. Furthermore, since the photocurrents for
a particular axis are collinear, it is easily seen that the intensity of the impinging light
is not a factor if proper signal processing techniques are employed. This topic is
discussed in a later section.

4.1.2. Fujinon CF25B TV Lens

A Fujinon closed-circuit television lens (model CF25B) is mounted in front of the
detector to provide a wider field of view. This lens is designed with a 25 mm fixed focal
length and a 1 inch industry standard image format.

4.1.3. Hamamatsu C3683 Analog Signal Processing Device

Directly connected to the electrodes of the S-1200 are a pair of C3683 analog
signal processing circuits manufactured by Hamamatsu Photonics. Each circuit processes
the signals for a particular axis. A schematic for the device is shown in Figure 4.4.
Each device has two inputs, one for each of the electrodes. Each input channel is tied
to a low-noise operational amplifier that generates a voltage that is proportional to the
photocurrent from the detector. The next stage of the device consists of a pair of low-
offset op-amps that generate the sum and difference of the electrode voltages. The final
Figure 4.4: Schematic of Signal Processing Circuit.
stage of the device consists of an analog divider circuit. This stage generates the quotient of the difference and the sum. Hence the signal processing circuit essentially performs the following operation:

\[
V_o = \frac{V_2 - V_1}{V_1 + V_2}
\] (4.1)

where:

\[
\begin{align*}
V_o &= \text{output voltage of the C3683} \\
V_1 &= \text{voltage proportional to the photocurrent from electrode 1} \\
V_2 &= \text{voltage proportional to the photocurrent from electrode 2.}
\end{align*}
\]

The output voltage generated by the C3683 is directly proportional to the position of the incident light on the detection surface. It is this method of calculation that removes the intensity of the impinging light as a factor in the determination of the position. The output voltage signals from these devices can be directly sampled by the DAS-20.

4.1.4. Hamamatsu L2791 Infra-Red Emitting Diode

The light source used for characterization of the sensing system is the Hamamatsu L2791 infra-red emitting diode with a micro-ball lens. This diode is a GaAlAs device with an epoxy lens mounted to the chip surface. In order to drive the IR diode, it was decided that the TTL channels on the DAS-20 be used. This implementation leaves the analog output channels free for other uses to be discussed later. However, the TTL lines can only source 25 mA and thus can not deliver the needed current of approximately 65
mA. Hence a power amplifier is required to perform this task. The design of this amplifier is discussed in the following section.

![Figure 4.5: Schematic of LED Power Amplifier](image)

4.1.5. **Power Amplifier for LED Circuit**

The power amplifier was designed as a unity gain transistor amplifier with large current sourcing capability. Figure 4.5 shows the schematic of the circuit. Basically, the circuit operates by driving the base of an NPN bi-polar junction transistor with an operational amplifier. The gain control is achieved by feeding the emitter voltage of the transistor back to one input of the op-amp. Hence, the output voltage of the circuit
follows the input voltage. However the 2N2219A transistor allows a maximum continuous current source through the load of 500 mA which more than satisfies the requirements. The bandwidth of the amplifier was experimentally determined and found to be 80 Khz. As later discussion will show, this bandwidth is much higher than that of other system components and need not be further considered.

4.1.6. Two-Axis Automatic Linear Position Controller

In order to facilitate position calibration of the detector, some sort of two-axis automatic position control system was needed. A practical yet economical solution to this problem was to use an existing x-y pen-plotter. An IR LED could be mounted on the plot head and the detector mounted above the plotter. Figure 4.6 shows this configuration. The Department of Electrical and Computer Engineering at Ohio University has in its possession a series 1130 Variplotter manufactured by Electronic Associates, Inc. See Figure 4.7. This device (which is 25 years old) is an analog recorder which responds to x and y-axis analog dc voltage signals. The inputs to each axis have a switch selectable input sensitivity that ranges from 0.5 millivolt per inch to 20 volts per inch. Hence the DAS-20 output channels can
Figure 4.7: EAI Series 1130 Variplotter (EAI Maintenance Series, Page iv.)
be used to position the IR source for testing purposes. As mentioned before, the IR
diode is actually controlled by a TTL output on the DAS-20.

4.2. Characterization Experiments and Software Development

This section discusses the experiments performed to calibrate the IRMPSS and the
development of software routines that make the tests possible. The discussion of the tests
and the software are integrated to make examination of the experimental needs and
practical solutions more clear. The results of these tests are presented in a later section.
For purposes of calibration, most of the routines use the DAS-20 simply as a data
acquisition device and are thus uncomplicated in nature. A few of the routines
implement both input and output on a software controlled level.

4.2.1. Generating a Map of the Detector Surface

In order to gain a preliminary understanding of the detector in terms of position
measurement characteristics, it was necessary to generate a map of the surface of the
detector. This was done by using the two-axis linear position controller (LPC) described
in section 4.1.6. The IR diode was placed on the plot head of the LPC which was
controlled by the computer. Since the LPC is an analog device, the two analog output
channels of the DAS-20 were used. Thus, the computer could position the diode
virtually anywhere in the plotting area of the LPC. Since the physical position of the
diode was known at all times, the surface characteristics could be determined by
analyzing the measurements obtained from the sensor.
The software needed to control this process had to perform a series of tasks as indicated in Figure 4.9. See Appendix II for a complete listing of the code. The first step was to initialize the DAS-20. The position control data (referred to as input data) was then loaded from disk. This data was generated by MATLAB and stored on disk in the form of two vectors. Each vector contained the position control data for a particular axis. The software then allocated the necessary storage for the position measurement data (referred to as output data).

![Calibration Coordinate Map](image)

Figure 4.8: Calibration Coordinate Map

To generate the map, the PC commanded the position controller to move to each point in the grid defined by the coordinates taken from the control data. See Figure 4.8. The diode was moved to each position in such a manner as to overcome the static friction of the LPC. Since the spacing between the adjacent setpoints was relatively small
Figure 4.9: Flow Diagram For Position Calibration Software (XYTEST).
(typically less than 0.05 inches) it was decided that the LPC should move a larger distance to a predetermined base point before moving to the next grid position. This base point was defined to be the endpoint along the axis of motion that is furthest from the destination point. See Figure 4.10 for a graphical description of this concept.

At each grid point, the diode control TTL line from the DAS-20 was toggled in order to illuminate the detector. The position data was then sampled and stored in the PC and the TTL line returned to its inactive state. It should be noted that a 0.5 second delay was introduced before activating the diode in order to allow the LPC sufficient time to settle. Also, the emitter was held active for 0.5 seconds before the ADC’s acquired the position measurement.

Figure 4.10: Example of LPC Motion.
A separate sensor map was generated for regions of several dimensions and at two
different distances between the emitter and the detector. In each configuration, several
maps were to be generated and the resulting data averaged to reduce the effects of LPC
position error and sensor noise. As will later be shown, the sensor noise is virtually
negligible. However, before the desired number of maps were generated, the LPC
suffered hardware difficulties that forced its use to be discontinued. Hence, while the
data in most cases has been averaged at least one time (the most being nine), there will
still be some small effects resulting from the position errors. Nevertheless, it is believed
that the data obtained before LPC failure is quite reliable and the results obtained from
its use are considered acceptable.

As will be shown in the results section, the sensor surface contains a highly
nonlinear outer region surrounding a much more linear inner area. In order to facilitate
study of the boundaries of the linear region, another software package called IPC
(incremental position control) was developed (See Figure 4.11). This also proved most
useful when aligning the LPC with the detector (which had to be done practically every
night after all other people had left the laboratory).

Basically, the IPC software allows the user to control the LPC by selectively
placing the emitter anywhere within the plotter's boundaries (subject to the effects of
quantization). The user can also command the LPC to shift one DAC count at a time
in either direction along both axes. The LPC can be sent to the center of the plot area
(the "zero" condition) at any time with a single keystroke, and the LED toggled on and
Figure 4.11: Flowchart of Incremental Position Control Software.
off with single keystrokes as well. The software also actively updates the position data measured by the sensor after each operation. See Appendix V for a listing of the code.

The main routine basically acts as a switchboard that generates a list of single keystroke commands and waits for a response. When a key is pressed, the main calls the functions necessary for the desired task. The functions are described below.

The POS function moves the LPC one DAC count in the direction specified by the arguments. This is done by issuing a single D/A conversion on the proper channel. The conversion is handled by a mode call to the software provided by Keithley Metrabyte. (See the Keithley Metrabyte (KM) DAS-20 users guide for a list of the software mode calls.) It is possible to assign various position boundaries outside which the plotter will be prohibited from moving. Error checking ensures that the plotter stops at these pre-determined boundaries. For various tests, the boundaries were experimentally determined to facilitate study of the linear/nonlinear boundary. Once set, any subsequent commands to cross the boundary have no effect.

The SET function allows the user to input the desired coordinates of the LPC in inches relative to the zero coordinates (center). This function requires that the LPC is configured for 2 volts per inch. The function uses a KM mode call to issue a D/A conversion that changes the LPC position. This is done for each axis. Error checking is used to prevent illegal position entries.

The SAMPLE function issues a KM mode call to sample the position data for each axis. The data is converted to double precision format from its default 12-bit two’s complement form.
The ZERO function is similar to the SET function except that the LPC is always moved to the origin.

The TOGGLE function toggles the status of the TTL line controlling the diode. This is done via a KM mode call that controls the 8-bit output port.

4.2.2. Sensor Resolution Tests

![Resolution Test Configuration](image)

Figure 4.12: Resolution Test Configuration

The position resolution of the sensor indicates the smallest magnitude of displacement detectable by the system. Clearly this value is proportional to the distance from the emitter to the detector (object distance).

The resolution was tested by precisely positioning the LED in front of the detector. An optical rail system was used to ensure no other movement. Figure 4.12 shows this test configuration. The measurements obtained from the sensor by the PC were monitored while the LED was moved a short distance using the micropositioner. This
displacement was then compared to the change in position voltage as seen by the DAS-20. This effectively yields a resolution measurement for the object distance in question.

The software designed to monitor this position information simply activated the DAS-20, turned on the LED, and displayed the sampled position data on the screen. No timing or memory access/handling functions were required. When the experiment was complete, any keypress turned off the diode and exited the software. For purposes of this experiment, the position data was displayed in its 12-bit two’s complement form. Hence the operator just needed to record the number of ADC counts that corresponded to the distance moved by the LED. The movement that caused a change of one ADC count represents the system resolution for that particular object distance. Appendix VI contains a listing of this code.

4.2.3. System Switching Characteristics

In many situations, it would be convenient or even necessary to be able to measure the position of independently placed objects. This would require more than one optical source, and they would have to be switched on and off in sequence to avoid measuring the optical centroid of all the objects. This type of implementation is called a multi-emitter application. An example of a multi-emitter application is the planned use of the IRMPSS in a flexible structures test facility (see Figure 4.13) in which the system will be used to detect the bending modes of a flexible structure. In this application, the structure (an aluminum rod) will be suspended vertically by a torque controlled two-axis gimbal system. The detector will be placed below the rod and will face upward along the axis of the structure. Independent emitters will be placed at selected points (mounted
helically) along the rod. The emitters will be fired sequentially and the detector will measure each deflection from the axis, thus "mapping" the bending modes of the structure. A further application will involve real-time use of the modal data to add damping to the system. In order to do this, the DAS-20 will be used to control a pair of orthogonally mounted bi-directional cold-gas linear thrusters attached to the free end of the beam.
The switching characteristics of the IRMPSS indicate the maximum speed at which the LED's can be pulsed and still yield a correct measurement from the sensor. If the LED's are pulsed too quickly, the detection surface could become charged with energy from more than one diode and thus yield a position measurement that represents the centroid of multiple sources. This has little use for position control applications. Hence these characteristics must be determined if the device is to be reliably used in such configurations.

Before continuing this discussion, some terms must be identified. The term "positive signal" means that the LED is currently illuminating the detector. The term "positive response" means that the sensor has responded to the positive signal. Similarly, the terms "negative signal" and "negative response" mean that the LED is off and the sensor registers this fact (respectively). These terms imply nothing about voltage polarities or direction of photocurrents.

The method by which the switching characteristics were be tested required no PC control. Actually, the DAS-20 was only used to obtain data with which to make the plots shown in Figure 4.14, Figure 4.15 and Figure 4.16. The results corresponding to

Figure 4.14: X-Axis Slow Switching Diagram.
this data had already been determined. The IR diode was connected to a variable frequency square wave generator (with a 50% duty cycle) and the outputs of the signal processors were connected to a digital storage oscilloscope. The frequency of oscillation was then increased to the point where the sensor was no longer able to respond fast enough to the pulsing of the LED. In other words, the switching of the emitter from negative to positive signal and back again was fast enough that the detection circuitry was unable to switch properly from corresponding positive and negative response states before the next signal cycle.

Figure 4.14 shows the LED signal pulses and the corresponding responses for the x-axis of the detection system. As can be seen, the LED signal is switching at a slow...
enough rate that the sensor is able to respond before the next signal transition. Figure 4.15 shows the same channel after the LED frequency has been increased. For this frequency, it can be seen that the sensor responds quickly to a positive signal state, but responds slowly to the negative signal state. In fact, for this critical frequency, the sensor reaches negative response just as the LED shifts back to a positive signal state. This switching frequency represents the bandwidth of the system for this particular axis. Switching the LED any faster results in the response observed in Figure 4.16. At this speed the sensor response never reaches the true negative response state (which is indicated in Figure 4.14 as the maximum response voltage) before the LED switches back to a positive signal configuration.

As far as software development for this experiment, there really is none. The experiment was conducted manually using a digital oscilloscope. The software used to collect data for the plots is the same software used to obtain noise characteristic data and will be discussed in that section. One important fact about that software that must also be mentioned here is that the software does sample two analog channels. As was found in the discussion from chapter 2, the DAS-20 is not physically able to collect data from multiple channels simultaneously without additional hardware. Hence, the corresponding signal and response data points shown in Figure 4.14, Figure 4.15 and Figure 4.16 are actually separated in time by about 50 microseconds. For the purposes for which this software was developed, this time delay is considered negligible.
4.2.4. **System Noise Characteristics**

Knowledge of the noise characteristics of the sensor system enables the user to better determine the accuracy of acquired data. The test by which these characteristics were determined is a straightforward one in which the emitter was placed in a fixed location and activated at a constant power level. The position measurements generated by the sensor were then sampled at a high frequency by the DAS-20. The data was then analyzed using the MATLAB software package.

The software (called SNOISE) was designed to accept command-line arguments for the data filename, number of points to be sampled, and the rate at which to acquire data. Figure 4.17 shows the flow diagram for SNOISE. See Appendix III for a listing of the code. The software uses this information to allocate the storage needed to hold the samples and set the pacer clock. DMA data transfer was chosen so that extremely high (greater than 50 kHz) sampling rates could be used. The sampling queue is configured to acquire data from channels 0 and 1 on an alternating basis. Due to the alternating feature of this data, the x and y-axis data is always separated in time.

<table>
<thead>
<tr>
<th>CHANNEL1(1)</th>
<th>CHANNEL2(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANNEL1(2)</td>
<td>CHANNEL2(2)</td>
</tr>
<tr>
<td>CHANNEL1(3)</td>
<td>CHANNEL2(3)</td>
</tr>
</tbody>
</table>

Table 4.1: SNOISE Data Format

This is necessary because the DAS-20 cannot acquire data from multiple channels simultaneously. Hence the data is stored as shown in Table 4.1. This data is separated into its two-channel format via an m-file written for MATLAB (shown in Appendix IV).
Figure 4.17: Flow Diagram for SNOISE Software.
by one sample period. However, since the channels are analyzed separately for power spectral density, this fact is of no concern.

Before acquisition actually begins, the DAS-20 activates the LED via the TTL control line. When the acquisition cycle is completed, the diode is turned off. The data is then converted to double precision form and saved to disk for use in MATLAB.

4.3. Characterization Results

This section discusses the results of the characterization experiments. The experiments were conducted as described in the previous section. Data is given where necessary to support the findings.

4.3.1. Sensor Mapping

Figure 4.18 shows a map of the entire surface of the detector. For this particular set of data, the LPC was placed 12.25 inches from the detector and commanded to cover an 81 by 81 point grid corresponding to a 5 by 5 inch square centered about the origin. For each axis, the measurement voltage has been scaled to the maximum deflection of the LPC plot head. As is easily seen, the sensor responds in an extremely nonlinear fashion near the edges, but responds rather well near the center. Hence the study of the sensor was subsequently restricted to this more linear region. Figure 4.19 shows a map of the linear portion of the detector surface. In this case, the LPC was restricted to a 2.5 by 2.5 inch region again centered about the origin. Due to the use of the lens, the area of the linear region is proportional to the distance from the emitter to the detector. Experiments using two different distances from the emitter to the detector have indicated
Figure 4.18: Map of Entire Detector Surface

Figure 4.19: Map of Linear Portion of Detector Surface
that the area of the linear portion of the detector is defined by a square whose dimensions are equal to 20.4 percent of the distance from the lens to the object. Hence, it appears that the IRMPSS can be used as a linear position measurement system as long as the target object remains within the appropriate region. Note that the dimensions of the linear region are also dependent upon the lens being used in the system. A discussion of the optical phenomena introduced by changing the lens used in the system is avoided in this text.

4.3.2. Resolution Tests

As mentioned in the section discussing the experimental procedure used to determine the resolution of the sensor system, the resolution is proportional to the distance from the emitter to the detector. Hence the resolution will increase as the emitter is placed closer to the detector. Also, the lens used in the system will directly effect the observed resolution.

In all tests, the limiting factor in the IRMPSS resolution is the 12-bit resolution of the A/D converters. This is expected since the detector is actually an analog device whose output must suffer the effects of quantization when the computer is to be involved. It should be noted here that Hamamatsu Photonics states that the S-1200 has a maximum resolution of about 10 microns. This is most likely because the system noise makes it virtually impossible to distinguish position changes that are smaller than this value.

The resolution was tested at four separate distances from the emitter to the detector. Table 4.2 contains the results of these tests for both axes. Figure 4.20 graphically illustrates the results for the x-axis and Figure 4.21 does the same for the y-axis.
Table 4.2: Resolution Test Results.

<table>
<thead>
<tr>
<th>Dist. to Object (Inches)</th>
<th>X-Axis Resolution (Microns)</th>
<th>Y-Axis Resolution (Microns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.03</td>
<td>52.3</td>
<td>46.5</td>
</tr>
<tr>
<td>19.94</td>
<td>61.7</td>
<td>40.6</td>
</tr>
<tr>
<td>22.91</td>
<td>72.1</td>
<td>61.7</td>
</tr>
<tr>
<td>30.75</td>
<td>96.6</td>
<td>191.1</td>
</tr>
</tbody>
</table>

Figure 4.20: X-Axis Resolution Vs. Distance to Object

data. As can be seen, the resolution increases linearly with the distance from the detector for the x-axis. This was expected because all test points were located within the linear region of the detector. However, the y-axis data yields a nonlinear curve. The reasons for this phenomenon have not yet been precisely determined, but the results have
been verified by further testing. It is suspected that these results indicate some nonlinear optical properties of the lens or detector that were not observed in the mapping results. This data strongly demonstrates that the IRMPSS must be calibrated for each application in which it is to be used.

4.3.3. Switching Characteristics

The maximum speed at which the IRMPSS can reliably switch between multiple emitters was initially determined to be 475 Hz. This was the highest rate that the data from the x-axis would run. The y-axis would run at 800 Hz for the same configuration. It was decided to determine if the signal processing circuitry contributed to the speed limitation so the x and y-axis signal processing channels were swapped. In the new configuration, the y-axis ran at a maximum rate of 520 Hz and the x-axis reached 1530 Hz. Hence it was evident that one signal processing channel responded faster than the other. Since the signal processing circuits provide the means by which the charge is
dissipated from the detection surface, then two possible means of speed improvement became apparent. One method is to switch an active load into the signal processing input after the sample has been taken. This will provide a faster pathway to eliminate the optically deposited charge from the detector. However, without more knowledge of the current handling characteristics of the detector, this method could result in destruction of the device. The second method of speed improvement is to reduce the intensity of the impinging light. In the original configuration, the LED passes about 62.0 mA. The series resistor was increased from 47 ohms to 150 ohms (thus reducing the current to 20.7 mA) and the switching speed tested once again. This time (with the x and y-axis channels returned to their original respective signal processing circuits) the x-axis switched at 610 Hz and the y-axis maintained 1165 Hz. When the signal processing channels were swapped again, the y-axis was able to sustain 625 Hz and the x-axis achieved 2693 Hz. Hence the maximum switching speed increased by a factor of 22 percent by introducing a 66 percent decrease in the current passed by the IR diode. Clearly, this is a nonlinear function, but does yield a small degree of control over the switching speed. Note that increasing the distance from the emitter to the detector has the same effect because less radiant energy from a distant source is captured by the lens than that captured from a closer source.

4.3.4. Noise Characteristics

The data obtained via the SNOISE software was loaded into MATLAB for analysis. The power spectral density (PSD) of the data from each axis was determined and is shown in Figure 4.22 and Figure 4.23. The PSD’s were generated via the Welch
method of spectral estimation. In this particular set of tests, the output of the detector was sampled at 1000 Hz. The maximum gain seen in the x-axis is -60.2 dB volts. When compared to a nominal signal of 1.51 volts, it is clear that the signal-to-noise ratio is rather large. Similarly, the y-axis PSD contains a maximum peak of -67.9 dB volts. When this is compared to a nominal signal of 0.437 volts, a large signal-to-noise ratio is again observed. Hence, the system is not likely to be disturbed a great deal by noise.
Chapter 5: Conclusions and Recommendations

This portion of the text summarizes the concepts and results reached throughout the first four chapters of the paper. The summary discusses the conclusions reached from experimental work and recommends further work that may be done. Special emphasis is placed on the results from Chapter 4 and some discussion is offered with regard to new applications of the infra-red multi-position sensing system.

In Chapter 2, the DAS-20 was introduced and its capabilities studied in some detail. With reference to some of the uses for which it was ultimately intended, several methods of programming were presented. Several software packages resulting from these ideas were developed and their usefulness examined. In particular, the method of using interrupt service routines to increase the flexibility of the DAS-20 beyond its original capabilities was explored in great detail. Other, less capable methods were addressed and their advantages and disadvantages discussed.

For the future, it may be possible to increase the capabilities of the DAS-20 even further by writing code that controls a larger portion of the system directly at the hardware level. This would allow the PC to use that DAS-20 for even more specialized work such as multi-control applications. In such an application, the DAS-20 could use different portions of its I/O capability for simultaneous monitoring and control of several independent systems and processes. Also, better memory handling techniques could be developed to give the DAS-20 access to expanded or extended memory in the PC. This would allow the DAS-20 to be used for better low-frequency analysis in system identification problems by increasing the length of time that data can be acquired.
The purpose of Chapter 3 was to discuss the development of a senior level control laboratory. The DAS-20 system was used in conjunction with the MS150 Modular Servo System to create a working automatic control system. The discussion focused on the hardware included in the system and yielded a brief outline of the experiments designed for the laboratory. The MATLAB software package was used as the analysis and design platform and software was written to allow MATLAB to transparently interface with the servo system through the DAS-20. This software allowed the student to perform characterization experiments on the various system components in order to generate an approximate model of the system. Also, a method of system identification was implemented using the facilities included with the DAS-20 and the PC. Finally, several digital compensators were designed and implemented through the DAS-20. These compensators adequately demonstrated the control theory related to the design techniques, but the fact that the servo system is highly nonlinear introduces many difficulties that are not easy to overcome. In the future, some work needs to be done to help reduce some of the nonlinear effects found in the system. Also, nonlinear studies could be done in order to better understand the system as it is.

While the software originally developed for the control laboratory is limited when compared to what the DAS-20 can now do, its development provided a great deal of information and direction from which later progress arose. The advanced ISR programming topics discussed in Chapter 2 are currently being integrated into a new set of software drivers for the lab. This software should make certain tasks easier and add to the variety of experiments that can be developed. For example, a 2-input 2-output
controller can be implemented which will allow the implementation of advanced full-state feedback techniques. Also, the effects of various sampling rates can be easily studied without the need to recalibrate the software for each new frequency.

Chapter 4 demonstrated that the DAS-20 can be used as to characterize the infrared multi-position sensing system (IRMPSS). Specifically, the DAS-20 was used to obtain information relating to the position measuring capability and frequency characteristics of the system. The position measuring capability was broken down into mapping and resolution tests. The results these tests indicated that the sensor possesses a linear detection region surrounded by a nonlinear area. Also, it was observed that the system is high-resolution but that the resolution appears to be a nonlinear function of the distance from the emitter to the detector. Hence it was concluded that the system must be calibrated for all applications in which it is to be used. The frequency tests involved determining the maximum switching speed of the detection system as well as the noise characteristics of the device. The switching speed was defined as the maximum rate at which the system could respond to a signal and then return to its negative response state. It was found that in the standard configuration the system could switch no faster than 475 Hz. This value was increased by decreasing the amount of energy striking the detector. However, when this level is decreased, the signal to noise ratio will suffer. Another method of increasing the switching speed is to switch in an active load to quickly draw current away from the detection surface once the position information has been sampled. This means will be investigated in the future. In terms of noise characteristics, the
standard system configuration has proven to be highly insensitive to noise effects. As mentioned above, these characteristics are dependent upon the emitter power level.

Overall, it is believed that the IRMPSS is a viable tool for laboratory or industrial use. As was discussed in Chapter 4, plans are underway for the use of the IRMPSS in a flexible structures test facility. The system was also considered for use as a position measuring device in a physical study of planetary ring dynamics. This study is currently being carried out by the Department of Physics and Astronomy at Ohio University and involves the precise measuring of non-elastic collisions in ice crystals. These studies should lend some insight to why the rings of Saturn (composed primarily of ice crystals) behave as they do.

A third possible application of the IRMPSS is that of thickness monitoring in a metal rolling mill. Two emitters could be mounted on the rolling press, one along the axis of each roller. The detector could then be placed some distance away to protect it from the harsh environment near the rolling process. The system could then very accurately monitor the thickness of the metal as it is being rolled. This would eliminate the traditional transport lag that is associated with thickness measurement systems that are located "downstream" from the rolling press.

In conclusion, it has been demonstrated that the DAS-20 is a powerful and versatile tool for use in many applications. These applications range from component and system calibration to process control interfacing. The system is fast, accurate and has thus far proven to be highly reliable. One specific advantage of using such a system
is that the operational aspects of the DAS-20 can be easily changed by simply altering the driver software.

Using the capabilities of the DAS-20, the characteristics of the infra-red multi-position sensing system were investigated in great detail. This detection system has proven to be an efficient and relatively cost effective means of measuring the positions of several independently placed objects.
References


Hamamatsu Corporation (1989), Large-Area PSD Series Technical Data, Hamamatsu Corporation, Bridgewater, NJ.

Hamamatsu Corporation (1990), Opto-semiconductors Condensed Catalog '90 - '91, Hamamatsu Corporation, Bridgewater, NJ.

Hamamatsu Corporation (1990), Operation Manual of Hamamatsu Signal Processing Circuit for P.S.D Type Number C3683-01, Hamamatsu Corporation, Bridgewater, NJ.


APPENDIX I: ISRSVAR Software Listing (C)

/** PROGRAM ISRSVAR.C
** Implements a state-variable control algorithm.
** All I/O is achieved via MAT-files.
**
** Daniel Allwine -- 11/30/92
** Latest Revision - 11/30/92
*/

#include <math.h>
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
#include <dos.h>
#include <time.h>
#include "matlsh.h"
#include "addmult2.h"  /* This header assumes that the main has allocated 
space for the matrix operation solutions */

int loadmat();
int savemat();
void readerr();

int matadd();
int matmult();

void (interrupt far *oldvect)(void);
void interrupt far dasisr(void);

/* ISR vars */
int intcheck;
int statdan;
int samplox,samphix,samploy,samphy;
int imc2adcd;

/* Other vars */
int isrerror;
int count;
int cstop;
int dok;
int xinsamp, yinsamp;
int xoutsamp, youtsamp;
int imcron,imcroff,IntLevel, IntMask, oldmask;
int done;

main()
{
    int mode, error, data[10];
    float rate;
    time_t cstart, cend, ctotl, c2start, c2end, c2totl;
    int temp;
    FILE *fp;
    int i, j;
    int type, imagf;
    char name[20];
    double huge *a, *b, *c, *d;
    double huge *u, *x;
    double huge *y, *dummy;
    double huge *xdot, *usp, *ysp;
    double huge *ax, *bu, *cx, *du;
    int rowa, cola, rowb, colb, rowc, colc, rowd, cold;
    int rowu, colu, rowx, rowy, coly;
    int rowxdot, colxdot, rowusp, colusp, rowysp, colysp;
    int rowax, colax, rowbu, colbu, rowcx, colcx, rowdu, coldu;
    int loaderr;
    int addstat, multstat;
    int numin, numout, order, numpts, sizey;

    IntLevel = 2;

    /* Init Das-20 */
    mode = 0;
    data[0] = 0x300;
    data[1] = IntLevel; /* INT level */
    data[2] = 1; /* DMA level */
    if ((error = das20(mode, data)) != 0)
    {
        printf("\n Mode %d Error = %d\n", mode, error);
        goto endprog;
    }

    /* Set Timer */
    rate:
    mode = 25;
    printf("\n Enter the sampling frequency in Hz: ");
    scanf("%f", &rate);
    /* rate = atof(argv[5]); */
printf("\n RATE = %f",rate);
if (rate >= 76.3)
{
    data[0]=5000000.0/rate;
    data[1]=0;
} else
{
    data[1]=2;
    while((rate+=rate) < 76.3) data[1] += data[1];
    data[0]=5000000.0/rate;
}
if (data[1] > 0)
printf("\nSetting A/D pacer to %f Hz\n",(float)5000000/(float)(data[0]*data[1]));
else
printf("\nSetting A/D Pacer Clock to %f Hz\n",(float)5000000/data[0]);
if ((error = das20(mode,data)) != 0)
{
    printf("\n Mode %d Error = %d\n",mode,error);
    goto endprog;
}

/* Load a from .mat file */
fp=fopen("a.mat","rb");
loaderr=loadmat(fp,&type,name,&rowa,&cola,&imagf,&a,&dummy);
if (loaderr)
{
    readerr(loaderr);
    fclose(fp);
    goto endprog;
}
fclose(fp);

/* Load b from .mat file */
fp=fopen("b.mat","rb");
loaderr=loadmat(fp,&type,name,&rowb,&colb,&imagf,&b,&dummy);
if (loaderr)
{
    readerr(loaderr);
    fclose(fp);
    goto endprog;
}
fclose(fp);
/* Load c from .mat file */
fp=fopen("c.mat","rb");
loaderr=loadmat(fp,&type,name,&rowc,&colc,&imagf,&c,&dummy);
if (loaderr)
    {
        readerr(loaderr);
        fclose(fp);
        goto endprog;
    }
fclose(fp);

;/* Load d from .mat file */
fp=fopen("d.mat","rb");
loaderr=loadmat(fp,&type,name,&rowd,&cold,&imagf,&d,&dummy);
if (loaderr)
    {
        readerr(loaderr);
        fclose(fp);
        goto endprog;
    }
fclose(fp);

;/* Load u from .mat file */
fp=fopen("u.mat","rb");
loaderr=loadmat(fp,&type,name,&rowu,&colu,&imagf,&u,&dummy);
if (loaderr)
    {
        readerr(loaderr);
        fclose(fp);
        goto endprog;
    }
fclose(fp);

/* Check for proper SV representation */
if (rowa != cola)
    {
        printf("A is not square");
        goto endprog;
    }
if (rowa != rowb)
    {
        printf("A and B have different number of rows");
        goto endprog;
    }
if (rowc != rowd)
    {
        printf("C and D have different number of rows");
        goto endprog;
    }
if ((colb != colu) || (cold != colu))
{
    printf("Number of inputs is mismatched to controller");
    goto endprog;
}

numin = colb;
numout = rowc;
order = rowa;
umpts = rowu;

printf("Number of Inputs = %d", numin);
printf("Number of Outputs = %d", numout);
printf("System Order = %d", order);
printf("Number of Points = %d", numpts);

/* Allocate space for state vector */
x = (double huge *)halloc((long)order, sizeof(double));
if (x == NULL)
    {
        printf("Cannot allocate state vector");
        goto endprog;
    }

/* Allocate space for xdot */
xdot = (double huge *)halloc((long)order, sizeof(double));
if (xdot == NULL)
    {
        printf("Cannot allocate state vector");
        goto endprog;
    }

/* Allocate space for single-step input vector */
usp = (double huge *)halloc((long)numin, sizeof(double));
if (usp == NULL)
    {
        printf("Cannot allocate single-step input vector");
        goto endprog;
    }

/* Allocate space for single-step output vector */
ysp = (double huge *)halloc((long)numout, sizeof(double));
if (ysp == NULL)
    {
        printf("Cannot allocate single-step output vector");
        goto endprog;
    }

/* Allocate space for output array */
sizey = numout * numpts;
y = (double huge *)halloc((long)sizey, sizeof(double));
if (y == NULL)
    {
        printf("Cannot allocate state vector");
        goto endprog;
    }
/* Allocate space for ax vector */
ax=(double huge *)halloc((long)order,sizeof(double));
if(ax == NULL)
    { printf("\nCannot allocate state vector");
        goto endprog; }

/* Allocate space for bu vector */
bu=(double huge *)halloc((long)order,sizeof(double));
if(bu == NULL)
    { printf("\nCannot allocate state vector");
        goto endprog; }

/* Allocate space for cx array */
    cx=(double huge *)halloc((long)sizey,sizeof(double));
if(cx == NULL)
    { printf("\nCannot allocate state vector");
        goto endprog; }

/* Allocate space for du array */
    du=(double huge *)halloc((long)sizey,sizeof(double));
if(du == NULL)
    { printf("\nCannot allocate state vector");
        goto endprog; }

dok=0;
cstop=numpts;

/* Install INT handler */
_disable();
oldvect=_dos_getvect(8+IntLevel);
_dos_setvect(8+IntLevel,isr);
IntMask=1;
oldmask=inp(0x21); /* Normally port 0x21 */
printf("\n oldmask = %d",oldmask);
oldmask=(oldmask & ~(IntMask << IntLevel));
printf("\n newmask = %d",oldmask);
outp(0x21,oldmask);  /* Normally port 0x21 */
/* Let it all fly */
_enable();
printf("\nEnabling DAS-20 Level %d INT\n",IntLevel);

/* Set INT mode control register with DAS-20 INTs ENABLED */
imcron = IntLevel;
imcron = imcron << 4;
imcron = imcron | 0x85;
outp(0x304,imcron); /* output INT MODE CTRL Register
   bit 7    -- 0=INT Disabled, 1=INT Enabled
   bits 6,5,4 -- IRQ#
   bits 3,2    -- INT Source
   bit 1    -- Pending INT (cleared by WRITE)
   bit 0    -- 0=16 Sing. Chan, 1=8 Diff. Chan */

Start_up:
/* Start the INT clock - per DAS20.asm MODE 9 - SUland2_1 */
/* Here we assume internal clock with NO gate */
outp(0x307,0x1); /* CTR 1 */
outp(0x306,0xa1);
outp(0x306,0xb0);
outp(0x307,0x2); /* CTR 2 */
outp(0x306,0x21);
outp(0x306,0x2b);
outp(0x307,0x43); /* Load CTRs 1 and 2 */
outp(0x307,0x23); /* Arm CTRs 1 and 2 */

printf("\nEntering the State-Variable loop!!\n\n");

isrerror=0;

/* Perform Algorithm
 ** x. = Ax + Bu
 ** y = Cx + Du */
for (i=0;i<numpts;i++)
{
    checkdok:
    while(dok==0) goto checkdok;

    /* Get single-step input vector */
    for (j=0;j<numin;j++)
    {
        *(usp+j) = *(u+i+numpts*j);
    }
    /* Calculate Ax */
    multstat=matmult(a,rowa,cola,x,order,1,ax);
    if (multstat)
        { printf("\nCannot compute Ax");
          goto endprog; }
    else

{- rowax = rowa;  
  colax = 1; }

/* Calculate Bu */
multstat=matmult(b,rowb,colb,usp,numin,1,bu);
if (multstat)
    { printf("\nCannot compute Bu");  
goto endprog; }
else
    { rowbu = rowb;  
colbu = 1; }

/* Calculate xdot */
addstat=matadd(ax,rowax,colax,bu,rowbu,colbu,xdot);
if (addstat)
    { printf("\nCannot compute xdot");  
goto endprog; }
else
    { rowxdot = rowax;  
colxdot = 1; }

/* Calculate Cx */
multstat=matmult(c,rowc,colc,x,order,1,cx);
if (multstat)
    { printf("\nCannot compute Cx");  
goto endprog; }
else
    { rowcx = rowc;  
colcx = 1; }

/* Calculate Du */
multstat=matmult(d,rowd,cold,usp,numin,1,du);
if (multstat)
    { printf("\nCannot compute Du");  
goto endprog; }
else
    { rowdu = rowd;  
coldu = 1; }

/* Calculate ysp */
addstat=matadd(cx,rowcx,colcx,du,rowdu,coldu,ysp);
if (addstat)
    { printf("\nCannot compute ysp");  
goto endprog; }
else
    { rowysp = rowcx;
      colysp = 1; }

/* Concatenate ysp vector to y array */
for(j=0;j<numout;j++)
    { *(y+numpts*j+i) = *(ysp+j);
    }
/* Set x = xdot */
for (j=0;j<order;j++)
    { *(x+j) = *(xdot+j);
    }
dok=0;

if (isrerror  ==  1) goto endisr;

} /* End of Algorithm */

endisr:
if (done==1) printf("\n Done with numpts");
/* Set INT mode control register with INTs DISABLED */
imcroff=IntLevel;
imcroff=imcroff << 4;
imcroff = imcroff | 0x05;
outp(0x304,imcroff); /* output INT MODE CTRL Register
    bit 7    -- 0=INT Disabled, 1=INT Enabled
    bits 6,5,4  -- IRQ#
    bits 3,2   -- INT Source
    bit 1   -- Pending INT (cleared by WRITE)
    bit 0   -- 0=16 Sing. Chan, 1=8 Diff. Chan */

/* REMOVE INT handler */
printf("\nRemoving INT Handler");
_disable();
IntMask=1;
oldmask=inp(0x21);
oldmask=(oldmask | (IntMask << IntLevel));
outp(0x21,oldmask);
_dos_setvect(8+IntLevel,oldvect);
_enable();
/* Stop pacer clock */
printf("Shutting down Pacer clock");
mode = 26;
data[0]=2; /* Stops BOTH clocks */
if ((error = das20(mode,data))!=0)
{
    printf("Mode %d Error = %d\n",mode,error);
goto endprog;
}

/* Turn off the diode */
/* outp(0x30C,0x00); */

/* Zero both DAC channels */
outp(0x308,0x00); /* Channel 0 LSB */
outp(0x309,0x00); /* Channel 0 MSB -- Causes latched DAC */
outp(0x30A,0x00); /* Channel 1 LSB */
outp(0x30B,0x00); /* Channel 1 MSB -- Causes latched DAC */

/* Save output to .mat file */
if (isrerror == 0)
{
    fp=fopen("y.mat","wb");
savemat(fp,0,"y",numpts,numout,0,y,(double *)0);
fclose(fp);
}

if (isrerror == 0) printf("Algorithm sucessfully completed");
endprog:
printf("Program terminated -- count = %d.",count);
}

void interrupt far dasisr(void)
{
    /* NOTE: "in" refers to obtained data -- to be OUTPUT by the main
    **      "out" refers to output data -- loaded (INPUT) as a setpoint by
    **      the main */
    if (count==cstop) goto endint;
    if (isrerror==1) goto endint;
count++;
    _enable();
imc2adcd = IntLevel;
I*
CHECK INTERRUPT */
intcheck=inp(0x304);
intcheck=(intcheck | 2);
if (intcheck == 0)
goto endint;

if (dok == 0) dok=1;
else
{
    printf("\n ERROR ERROR ERROR !!!");
    isrerror=1;
    goto endint;
}

I* Clear INT */
I* Set INT mode control register with DAS-20 INTs ENABLED */
outp(0x304,imcron); /* output INT MODE CTRL Register
    bit 7 -- 0=INT Disabled, 1=INT Enabled
    bits 6,5,4 -- IRQ#
    bits 3,2 -- INT Source
    bit 1 -- Pending INT (cleared by WRITE)
    bit 0 -- 0=16 Sing. Chan, 1=8 Diff. Chan */

I* Send data to the DACs */
xoutsamp=(int)((float)count/10.*204.8);
youtsamp=(int)(0.25*204.8);
samplox=xoutsamp&255;
samphix=xoutsamp > > 8;
samploy=youtsamp&255;
samphiy=youtsamp > > 8;
outp(776,samplox);
outp(777,samphix);
outp(778,samploy);
outp(779,samphiy);
/* READ from the ADCs */
/* Channel 0 */
outp(0x303,0x10); /* Sets ADC delay timer */
outp(0x302,0x03); /* Sets chan/gain with EOQ to queue control */
outp(0x304,imc2adcd); /* Clears any pending INT, disables INT from DAS-20 and takes INT on End Of ADC */
outp(0x300,0x00); /* Manual conversion start */
get_stat1:
statdan = inp(0x304); /* Get conversion status */
statdan = (statdan & 0x02); /* Check ADC EOC bit */
if (!statdan) goto get_stat1; /* If ADC is not done, wait and check again */

samphix = inp(0x301); /* Get hi data */
samplox = inp(0x300); /* Get low data */

/* Channel 1 */
outp(0x303, 0x10);
outp(0x302, 0x13);
outp(0x304, imc2adcd); /* Clears any pending INT, disables INT from DAS-20 and takes INT on End Of ADC */
outp(0x300, 0x00);
get_stat2:
statdan = inp(0x304); /* Get conversion status */
statdan = (statdan & 0x02); /* Check ADC EOC bit */
if (!statdan) goto get_stat2; /* If ADC is not done, wait and check again */

samphiy = inp(0x301);
samploy = inp(0x300);

/* Shift data to proper form */
samplox = (samplox >> 4);
samplox = (samplox & 0x0f);
samphix = (samphix & 0xff);
xinsamp = samphix*16 + samplox;
samploy = samploy >> 4;
yinsamp = samphiy*16 + samploy;

/* STORE INTEGER DATA */
/* xouti[lenxin-count]=xinsamp; */
/* youti[lenyin-count]=yinsamp; */

/* Reset IMC Register to accept INTs on Timer #2 and enable */
outp(0x304, imcron); /* output INT MODE CTRL Register */
    bit 7 -- 0=INT Disabled, 1=INT Enabled
    bits 6,5,4 -- IRQ#
    bits 3,2 -- INT Source
    bit 1 -- Pending INT (cleared by WRITE)
    bit 0 -- 0=16 Sing. Chan, 1=8 Diff. Chan */

/* End Of Interrupt */
endint:
outp(0x20,0x20);
}

APPENDIX II: XYTEST Software Listing (C)

/* PROGRAM XYTEST.C
** Reads 2 xy-plotter control vectors from .mat files.
** Uses these vectors to position the xy-plotter and take
** corresponding sensor xy-coord voltage samples.
** Returns these voltages to .mat files.
**
** Daniel A. Allwine -- 08/08/92
** Ohio University Department of Electrical and Computer Engineering
** Latest Modification -- 10/22/92
*/
#include <stdio.h>
#include <malloc.h>
#include <time.h>
#include "matlsh.h"

int loadmat();
int savemat();
void readerr();

typedef struct {
    long type;
    long mrows;
    long ncols;
    long imagf;
    long namlen;
} Fmatrix;

main()
{
    /* Declare Vars */
    FILE *fp;    /* File Pointer (used throughout) */
    char name[20];    /* Array Name */
    int type, mrows, ncols, imagf;    /* Array Info */
    int mode, data[10], error;
    int i,j,k;
    int xdim, ydim, xbase, xcoord, asize, ybase, ycoord;
    int *xir, *yir, *xis, *yis;
    double _huge *xr, *xi;    /* Pointers to array elements (real and imag) */
    double _huge *yr, *yi;
    double _huge *xrs, *yrs;
    int loaderr;    /* Checks for errors in loading arrays */
    time_t cstart, dcheck;
/* Initialize DAS-20 */
printf("Initializing DAS-20");
mode=0;  /* Init Mode Number */
data[0]=0x300; /* Base Address (HEX) */
data[1]=5; /* INT Level */
data[2]=1; /* DMA Level */
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
{
    printf("Mode %d Error = %d\n", mode, error);
goto endprog;
}

/* Load x-coord vector */
printf("Loading xcoord.mat");
fp = fopen("xcoord.mat","rb");
loaderr = loadmat(fp,&type,name,&mrows,&ncols,&imagf,&xr,&xi);
if (loaderr)
    {
        readerr(loaderr);
fclose(fp);
goto endprog;
    }
fclose(fp);
xdim = ncols;

/* Load y-coord vector */
printf("Loading ycoord.mat");
fp = fopen("ycoord.mat","rb");
loaderr = loadmat(fp,&type,name,&mrows,&ncols,&imagf,&yr,&yi);
if (loaderr)
    {
        readerr(loaderr);
fclose(fp);
goto endprog;
    }
fclose(fp);
ydim = ncols;

/* Convert x-volts to x-ints */
printf("Converting x-volts to x-ints");
if ( (xir = (int *)malloc((size_t)sizeof(int)*xdim)) == NULL )
    {
        printf("Cannot allocate x-axis INT buffer");
goto endprog;
I printf("\nx-axis INT buffer at %p,",xir);
printf("\nx-axis INT buffer from %p to %p," ,xir,&xir[xdim-1]);
for (i=0;i<xdim;i++)
{
    xir[i]=(int)(xr[i]*204.8);
    printf("\nxir[%d] = %d",i,xir[i]);
}

/* Convert y-volts to y-ints */
printf("\nConverting y-volts to y-ints");
if ( (yir = (int*)malloc((size_t)sizeof(int)*ydim)) == NULL )
{
    printf("\nCannot allocate y-axis INT buffer");
goto endprog;
}
printf("\ny-axis INT buffer at %p," ,yir);
for (i=0;i<ydim;i++)
{
    yir[i]=(int)(yr[i]*204.8);
    printf("\nyir[%d] = %d",i,yir[i]);
}

/* Allocate sample INT buffers */
printf("\nAllocating sample INT buffers (x & y)" );
asize = xdim*ydim;
printf("\nasize = %d",asize);
if ( (xis = (int*)malloc((size_t)sizeof(int)*asize)) == NULL )
{
    printf("\nCannot allocate x-sample INT buffer");
goto endprog;
}
printf("\nx-axis INT sample-buffer at %p," ,xis);
if ( (yis = (int*)malloc((size_t)sizeof(int)*asize)) == NULL )
{
    printf("\nCannot allocate y-sample INT buffer");
goto endprog;
}
printf("\ny-axis INT sample-buffer at %p," ,yis);

/* Allocate sample DOUBLE buffers */
printf("\nAllocating sample DOUBLE buffers (x & y)" );
if ( (xrs = (double huge *) malloc(asize,sizeof(double))) == NULL )
  {
    printf("\nCannot allocate x-sample DOUBLE buffer");
    goto endprog;
  }
printf("\nx-axis DOUBLE sample-buffer at %p",xrs);
if ( (yrs = (double huge *) malloc(asize,sizeof(double))) == NULL )
  {
    printf("\nCannot allocate y-sample DOUBLE buffer");
    goto endprog;
  }
printf("\ny-axis DOUBLE sample-buffer at %p",yrs);

/* Zero the outputs */
printf("\nZeroing the Outputs");

/* x-axis */
mode=7;
data[0]=0;
data[1]=0;
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
  {
    printf("\n Mode %d Error = %d\n",mode,error);
    goto endprog;
  }
/* y-axis */
mode=7;
data[0]=1;
data[1]=0;
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
  {
    printf("\n Mode %d Error = %d\n",mode,error);
    goto endprog;
  }

/* Wait 30 sec */
printf("\nData loop will execute in:\n");
cstart=clock();
wait_30:
dcheck=(((float)clock()-cstart)/CLK_TCK);
  if (dcheck < 30)
printf("\r\t%4.2f\tseconds.",dcheck);
goto wait_30;
}

if ( (((float)clock()-cstart)/CLK_TCK) < 30 ) */

/* Start Taking Data */
printf("\nStart Loop");
startloop:
for (i=0;i<ydimg;i++)
{
  /* Determine base position y-coord */
  if ( i < (float)(ydimg/2) )
    ybase=ydimg-1;
  else
    ybase=0;
  /* Set base y-coord */
ycoord=ybase;
  /* Move to base position (y-axis only) */
  printf("\ny-axis base voltage = %5.3f volts",yr[ycoord]);
  mode=7;
  data[0]=1;
  data[1]=yir[ycoord];
  if ((error = das20(mode,data))!=0) /* Call DAS-20 */
  {
    printf("\n    Mode %d Error = %d\n",mode,error);
    goto endprog;
  }

for (j=0;j<xdim;j++)
{
  /* Determine base position x-coord */
  if ( j < (float)(xdim/2) )
    xbase=xdim-1;
  else
    xbase=0;
  /* Set base x-coord */
xcoord=xbase;
  /* Move to base position (x-axis only) */
  printf("\n  x-axis base voltage = %5.3f volts",xr[xcoord]);
  mode=7;
  data[0]=0;
  data[1]=xir[xcoord];
  if ((error = das20(mode,data))!=0) /* Call DAS-20 */
  {
printf("\n Mode %d Error = %d\n",mode,error);
goto endprog;
}

/* Wait 0.5 sec */
cstart=clock();

wait_a:
if ( (((float)clock()-cstart)/CLK_TCK) < .5)
goto wait_a;

/* Move to x-sample position */
printf("\nx-voltage = %5.3f volts",xr[j]);
mode=7;
data[0]=0;
data[1]=xir[j];
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
{
    printf("\n Mode %d Error = %d\n",mode,error);
goto endprog;
}

/* Move to y-sample position */
printf("\ty-voltage = %5.3f volts",yr[i]);*/
mode=7;
data[0]=1;
data[1]=yir[i];
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
{
    printf("\n Mode %d Error = %d\n",mode,error);
goto endprog;
}

/* Wait 0.5 sec */
cstart=clock();

wait_b:
if ( (((float)clock()-cstart)/CLK_TCK) < .5)
goto wait_b;

/* Fire diode */
mode=15;
data[0]=1;
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
{
    printf("\n Mode %d Error = %d\n",mode,error);
goto endprog;
/* Wait 0.5 sec */
cstart=clock();

wait_c:
    if ( (((float)clock()-cstart)/CLK_TCK) < .5 )
        goto wait_c;

    /* Take samples */
    /* x-axis */
    mode=3;
data[0]=1; /* +/- 10v * 0.5 Bipolar */
data[1]=0; /* x-axis -- Channel 0 */
    if ( (error = das20(mode, data)) != 0 ) /* Call DAS-20 */
        {
            printf("\n Mode %d Error = %d\n", mode, error);
            goto endprog;
        }
    *(xis+xdim*i+j)=data[0]; /* Store data to INT array */
    printf("\nStoring the x-axis INT sample at %p", xis+xdim*i+j);

    /* y-axis */
    mode=3;
data[0]=1; /* +/- 10v * 0.5 Bipolar */
data[1]=1; /* y-axis -- Channel 1 */
    if ( (error = das20(mode, data)) != 0 ) /* Call DAS-20 */
        {
            printf("\n Mode %d Error = %d\n", mode, error);
            goto endprog;
        }
    *(yis+xdim*i+j)=data[0]; /* Store data to INT array */

    /* Wait 0.25 sec */
cstart=clock();

    wait_d:
    if ( (((float)clock()-cstart)/CLK_TCK) < .25 )
        goto wait_d;

    /* Turn off diode */
    mode=15;
data[0]=0;
    if ( (error = das20(mode, data)) != 0 ) /* Call DAS-20 */
        {
            printf("\n Mode %d Error = %d\n", mode, error);
        }
goto endprog;
}

/* Show the input sequences again */
for (k=0;k<xdim;k++)
    printf("Input x-voltage[%d] = %6.3f -- = %d",k,xr[k],xir[k]);
/* Show the output INT array -- x-axis */
for (i=0;i<ydim;i++)
{
    for (j=0;j<xdim;j++)
        printf("ROW %d - COL %d - INT value = %d",i,j,*xis+i*xdim+j);
}
/* Zero the outputs */
printf("\nZeroing the Outputs");
/* x-axis */
mode=7;
data[0]=0;
data[1]=0;
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
{
    printf("\n Mode %d Error = %d\n",mode,error);
    goto endprog;
}
/* y-axis */
mode=7;
data[0]=1;
data[1]=0;
if ((error = das20(mode,data))!=0) /* Call DAS-20 */
{
    printf("\n Mode %d Error = %d\n",mode,error);
    goto endprog;
}
/* Convert INT samples to DOUBLE data for .MAT files */
printf("\nFinished Loop\nNow converting INT samples to DOUBLE samples");
for(i=0;i<ydim;i++)
{
    printf("\nConverting row %d",i);
    for(j=0;j<xdim;j++)
    {
        *(xrs+i*xdim+j)=(float)(*(xis+i*xdim+j))/204.8;
        *(yrs+i*xdim+j)=(float)(*(yis+i*xdim+j))/204.8;
    }
/* Save DOUBLE data to .MAT files */
printf("\nSaving to disk");
/* x-data */
fp = fopen("xdata.mat", "wb");
savemat(fp,type,"xdata",ydim,xdim,0,xrs,(double *)0);
fclose(fp);
/* y-data */
fp = fopen("ydata.mat", "wb");
savemat(fp,type,"ydata",ydim,xdim,0,yrs,(double *)0);
fclose(fp);
endprog:
;
APPENDIX III:  SNOISE Software Listing (C)

/*SNOISE - C program to test sensor characteristics.  *
*  Allows user to take multiple data points at the  *
*  same target location.  These samples are taken at  *
*  a constant rate as defined by the user.  *
*  *  COMMAND LINE:  snoise filename numpts rate  *
*  *  Daniel A. Allwine - 12/21/91  *
*  *  Latest Revision - 12/17/92  */
#include <malloc.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "matlsh.h"
#define MAXSIZE 16384

main(int argc, char *argv[]) {
  FILE *fp;
  int error,mode,trigger,rcycle;
  float rate;
  long dbufsize;
  int noc,numcon;
  int idx;
  int i,j,k,n,kk;
  int data[10];
  int carray[4];
  int far *buffer;
  int far *alloc();
  double huge *matdata;

  /* Init DAS-20 */
  printf("\n Initializing DAS-20 \n");
  mode=0;
  data[0]=0x300;
  data[1]=2;  /* INT Number */
  data[2]=1;  /* DMA Channel */
  if ((error=das20(mode,data)) != 0)
    printf("\n  Mode %d Error = %d",mode,error);
trig_rcyc:
  trig=2;  /* Internal Clock with NO Gating */
  rcyc=1;  /* One Data Run Only (No Repeat) */

/* Get Number of Samples */
num_samples:
  printf("Enter the number of samples to be taken. ");
  scanf("%u", &noc); /*
  noc=atoi(argv[2]);
  numcon=(noc<MAXSIZE)?noc:MAXSIZE;
  printf("Total samples taken will be %u\n",numcon);

/* Allocate INT Buffer */
  printf("Allocating INTEGER Buffer \n");
  if ((buffer = malloc((size_t)(numcon*sizeof(int)))) == NULL) /*
     if ((buffer = alloc(numcon)) == NULL)
     {printf("Cannot Allocate Buffer \n");
      goto endprog;

  } /* Allocate DOUBLE Buffer */
/* Check to see if 128K boundary is exceeded */
dbufsize=(long)numcon;
if(dbufsize > 16384)
  {
    kk=-1;
    do
      {kk++;
       } while((pow((double)2,(double)kk)) < ((double)dbufsize));
    dbufsize=(long)pow((double)2,(double)k);
  } else printf("The DOUBLE array dimension is %ld",dbufsize);

  printf("Allocating DOUBLE Buffer\n");
  if((matdata=(double huge *)halloc((long)numcon,sizeof(double))) == NULL)
     {printf("Cannot Allocate DOUBLE Buffer !!");
      goto endprog;
   }

/* Set Pacer Clock */
rate:
    mode=24;
/* printf("\n Enter the sampling frequency in Hertz \n");
   scanf("%f","rate); */
   rate = atof(argv[3]);
   printf("\n The indicated rate is: %f","rate);
if (rate <= 76.3)
{
   data[0] = 5000000.0/rate;
   data[1] = 0;
}
else
{  
data[1] = 2;
   while ((rate += rate) < 76.3) data[1] += data[1];
   data[0] = 5000000.0/rate;
}
   printf("\n Setting A/D Pacer Clock to %f Hertz
","rate);
   printf("\n Div1 = %u Div2 = %u","data[1],data[0]);
   if ((error=das20(mode,data)) != 0)
       printf("\n   Mode %d Error = %d","mode,error);

/* Preset A/D Control Queue */
queue:
   printf("\n Defining A/D control queue \n");
/* Set Channel 0 */
   mode=1;
   data[0]=0;    /* Channel 0 */
   data[1]=1;    /* Gain +/- 10 volts x0.5 */
   data[2]=2;    /* Init Counters */
   if ((error=das20(mode,data)) != 0)
       printf("\n   Mode %d Error = %d","mode,error);
/* Set Channel 1 */
   mode=1;
   data[0]=1;    /* Channel 1 */
   data[1]=1;    /* Gain +/- 10 volts x0.5 */
   data[2]=1;    /* Add EOQ marker */
   if ((error=das20(mode,data)) != 0)
       printf("\n   Mode %d Error = %d","mode,error);
/* Turn on LED */
led:
   printf("\n Turning on LED \n");
outp(0x30C,1); /* Use TTL Channel 0 (Digital Output) */

set_a_to_d:
   printf("Starting A/D Sampling\n");
   mode=6;
   data[0]=numcon;
   data[1]=segadr(buffer);
   data[2]=trig;
   data[3]=rcyc;
   if ((error=das20(mode7data)) != 0)
      printf("Mode %d Error = %d",mode,error);

wait:
   printf("The sample number is: ");
   mode=12;
   data[1]=1;
   while (data[1] != 0)
   {
      printf("\t\t%d\n" ,data[2]);
      if ((error=das20(mode7data)) != 0)
         printf("Mode %d Error = %d",mode,error);
   }

led_off:
   printf("Turning off LED\n");
   outp(0x30C,0);

data_fix:
   printf("Beginning FOR-loop. Loop length is %d\n",numcon);
   idx=0;
   for(idx = 0; idx < numcon; idx++)
   {
      /* printf("Loop location %d\n",idx); */
      /* printf("darray[%d] = %d\n",idx,darray[idx]); */
      buffer[idx]=buffer[idx] > 4;
      matdata[idx]=(double)buffer[idx]*(double)(10.0/2048.0);
      printf("matdata[%d] = %6.4f\n",idx,matdata[idx]);
   }

dump_to_mat:
   printf("Dumping to Hard Disk\n");
   fp = fopen(argv[1],"wb");
   savemat(fp, 0, argv[1], numcon, 1, 0, matdata, (double *)0);
fclose(fp);

printf("\n Stop - Program Completed. \n");

endprog:
;
}

function [x, y] = dsplit(x)
N = length(x)/2;
y = zeros(1,N);
z = y;
for i = 1:N
    z(i) = x((2*i)-1);
    y(i) = x((2*i));
end
x = z;
end
APPENDIX V: IPC Software Listing (C)

/* PROGRAM IPC.C
**
** This program allows the user to look at the x-y coordinate voltages from the sensor's signal processing boards.
** The user is able to increment or decrement both the x and y position command voltages using the keyboard.
**
** Dan Allwine -- 09/03/92 */
#include <stdio.h>
#include <conio.h>

int pos();
int set();
int sample();
int zero();
int toggle();

main()
{
    int mode,error,data[10];
    float xsv,ysv,xinch,yinch,xdat,ydat;
    int xaxis,yaxis;
    int diode;
    int ret;
    printf("\n\nWELCOME TO THE IPC PROGRAM... ");
    /* Init DAS-20 */
    mode=0;
    data[0]=0x300;
    data[1]=2;
    data[2]=1;
    if ((error=das20(mode,data))!=0)
    {
        printf("\n\tMode %d Error %d",mode,error);
        goto endprog;
    }
    diode=0;
    zero(&xaxis,&yaxis);
    cmd:
    printf("\n\nCommands: ");
    printf("\n\tx(m)ty-axis up(down) ");
    printf("\n\tj(k)tx-axis left(right) ");

printf("\n\tt\ttoggle diode");
printf("\n\tc\tset channel and voltage");
printf("\n\ts\ttake samples");
printf("\n\tq\tquit\n");
keywait:
switch(getch())
{
    case 'i':
        ret=pos(&yaxis,1,1);
        if (ret)
            goto endprog;
        ret=sample(&xdat,&ydat);
        if (ret)
            goto endprog;
        break;
    case 'm':
        ret=pos(&yaxis,1,-1);
        if (ret)
            goto endprog;
        ret=sample(&xdat,&ydat);
        if (ret)
            goto endprog;
        break;
    case 'j':
        ret=pos(&xaxis,0,-1);
        if (ret)
            goto endprog;
        ret=sample(&xdat,&ydat);
        if (ret)
            goto endprog;
        break;
    case 'k':
        ret=pos(&xaxis,0,1);
        if (ret)
            goto endprog;
        ret=sample(&xdat,&ydat);
        if (ret)
            goto endprog;
        break;
    case 't':
        ret=toggle(&diode);
        if (ret)
            goto endprog;
        ret=sample(&xdat,&ydat);
if (ret)
    goto endprog;
break;
case 'z':
    ret = zero(&xaxis, &yaxis);
    if (ret)
        goto endprog;
    ret = sample(&xdat, &ydat);
    if (ret)
        goto endprog;
    break;
case 'c':
    ret = set(&xaxis, &yaxis);
    if (ret)
        goto endprog;
    ret = sample(&xdat, &ydat);
    if (ret)
        goto endprog;
    break;
case 's':
    ret = sample(&xdat, &ydat);
    if (ret)
        goto endprog;
    break;
case 'q':
    ret = zero(&xaxis, &yaxis);
    if (ret)
        goto endprog;
    goto shutdown;
break;
default:
    goto keywait;
break;

printf("Settings and measurements:");
xisv = (float)xaxis/204.8;
yisv = (float)yaxis/204.8;
xinch = xisv/2;
yinch = yisv/2;
printf("nx-axis set at %7.5f volts (%d DAC) -- %7.4f", xisv, xaxis, xinch);
printf("ny-axis set at %7.5f volts (%d DAC) -- %7.4f", yisv, yaxis, yinch);
printf("nx-axis measurement = %7.4f volts", xdat);
printf("ny-axis measurement = %7.4f volts", ydat);
if (diode)
printf("\n\nThe diode is ON");
else
    printf("\n\nThe diode is OFF");
goto cmd;

shutdown:
mode = 15;
data[0] = 0;
if ((error = das20(mode, data)) != 0)
{
    printf("\n\tMode %d Error %d", mode, error);
goto endprog;
}
endprog:
printf("\n\nThank you for using the IPC ROUTINE...\n\t\tBye-bye");
}

int pos(axis, chan, inc)
int *axis;
int chan, inc;
{
int mode, error, data[5];
if ((*axis + inc) > 2047)
    *axis = *axis;
else
    if ((*axis + inc) < -2048)
        *axis = *axis;
else
    *axis = *axis + inc;
mode = 7;
data[0] = chan;
data[1] = *axis;
if ((error = das20(mode, data)) != 0)
{
    printf("\n\tMode %d Error %d", mode, error);
    return 1;
}
return 0;
}

int toggle(lite)
int *lite;
{
int mode, error, data[5];
if (*lite == 1)
    *lite = 0;
else
    *lite = 1;
mode = 15;
data[0] = *lite;
if ((error = das20(mode, data)) != 0)
    {
        printf("\n\tMode %d Error %d",
               mode, error);
        return 1;
    }
return 0;

int sample(xdata, ydata)
float *xdata;
float *ydata;
{
    int mode, error, data[5];
    int xi, yi;
    mode = 3;
data[0] = 1; /* Gain code */
data[1] = 0; /* x-axis */
if ((error = das20(mode, data)) != 0)
    {
        printf("\n\tMode %d Error %d",
               mode, error);
        return 1;
    }
x1 = data[0];
mode = 3;
data[0] = 1;
data[1] = 1;
if ((error = das20(mode, data)) != 0)
    {
        printf("\n\tMode %d Error %d",
               mode, error);
        return 1;
    }
y1 = data[0];
*xdata = (float)xi / 204.8;
*ydata = (float)yi / 204.8;
return 0;
}

int set(xax, yax)

int *xax;
int *yax;
{
    int mode, error, data[5];
    float xinch, yinch;
    printf("Enter the desired x-axis position in INCHES\t\tW); scanf("%f", &xinch);
    printf("Enter the desired y-axis position in INCHES\t\tW); scanf("%f", &yinch);
    *xax = (int)(xinch * 2 * 204.8);
    if (*xax > 2047)
        *xax = 2047;
    else
        if (*xax < -2048)
            *xax = -2048;
    mode = 7;
data[0] = 0;
data[1] = *xax;
if ((error = das20(mode, data)) != 0)
{
    printf("\n\tMode %d Error %d", mode, error);
    return 1;
}

*yax = (int)(yinch * 2 * 204.8);
if (*yax > 2047)
    *yax = 2047;
else
    if (*yax < -2048)
        *yax = -2048;
    mode = 7;
data[0] = 1;
data[1] = *yax;
if ((error = das20(mode, data)) != 0)
{
    printf("\n\tMode %d Error %d", mode, error);
    return 1;
}
return 0;
}

int zero(xaxis, yaxis)
int *xaxis;
int *yaxis;
i

int mode, error, data[5];
*xaxis = 0;
*yaxis = 0;
mode = 7;
data[0] = 0;
data[1] = *xaxis;
if ((error = das20(mode, data)) != 0)
{
  printf("\n\tMode %d Error %d", mode, error);
  return 1;
}

mode = 7;
data[0] = 1;
data[1] = *yaxis;
if ((error = das20(mode, data)) != 0)
{
  printf("\n\tMode %d Error %d", mode, error);
  return 1;
}
return 0;
APPENDIX VI:  RETEST Software Listing (C)

/* PROGRAM RETEST.C
** Enables testing of the IRMPSS resolution.
** Turns the LED on and displays the x & y axis voltages
** while the user adjusts the positioning micrometers.
**
** Daniel A. Allwine -- 12/15/92
** Latest Revision -- 12/15/92
**
** Ohio University
** Department of Electrical and Computer Engineering
*/

#include <stdio.h>
#include <math.h>
#include <conio.h>

main()
{
    double xscale=1.5704;
    double yscale=1.5205;
    int mode, error, data[10], xsamp, ysamp;

    /* Init DAS-20 */
    mode = 0;
    data[0]=0x300; /* BASE Address */
    data[1]=2;    /* INT Level = 2 */
    data[2]=1;    /* DMA Level = 1 */
    if (error = das20(mode,data))!=0)
    {
        printf("\nMODE %d ERROR = %d\n",mode,error);
        goto endprog;
    }

    /* Turn on diode */
    outp(0x30c,0x01);

    do
    {
        /* Take x-axis sample */
        mode = 3; /* Command Single A/D Conversion */
        data[0]=1; /* +/- 10v Bipolar *0.5 */
data[0] = 0; /* Channel 0 -- x-axis */
if ((error = das20(mode, data)) != 0)
{
    printf("\nMODE %d ERROR = %d\n", mode, error);
    goto endprog;
}
xsamp = data[0];

/* Take y-axis sample */
mode = 3; /* Command Single A/D Conversion */
data[0] = 1; /* +/- 10v Bipolar *0.5 */
data[1] = 1; /* Channel 0 -- y-axis */
if ((error = das20(mode, data)) != 0)
{
    printf("\nMODE %d ERROR = %d\n", mode, error);
    goto endprog;
}
ysamp = data[0];

printf("\nxdata = %4d : ydata = %4d", xsamp, ysamp);
}
while (!kbhit());

/* Turn OFF diode */
outp(0x30c, 0x00);

endprog:
printf("\nSTOP - Program Terminated.");
APPENDIX VII:  MATLSH.H Software Listing (C)

typedef struct {
    long type;    /* type */
    long mrows;  /* row dimension */
    long ncols;  /* column dimension */
    long imagf;  /* flag indicating imag part */
    long narnlen; /* name length (including NULL) */
} Fmatrix;

loadmat(fp, type, pname, mrows, ncols, imagf, preal, pimag)
FILE *fp;    /* File pointer */
int *type;   /* Type flag: see reference section of guide */
int *mrows; /* row dimension */
int *ncols; /* column dimension */
int *imagf; /* imaginary flag */
char *pname; /* pointer to matrix name */
double huge **preal; /* pointer to real data */
double huge **pimag; /* pointer to imag data */
{
    char *malloc(); /*
    Fmatrix x;
    int mn, naml;

    /*
    * Get Fmatrix structure from file
    */
    if (fread((char *)&x, sizeof(Fmatrix), 1, fp) != 1) {
        return(1);
    }
    *type = x.type;
    *mrows = x.mrows;
    *ncols = x.ncols;
    *imagf = x.imagf;
    naml = x.narnlen;
    mn = x.mrows * x.ncols;

    /*
    * Get matrix name from file
    */
    if (fread(pname, sizeof(char), naml, fp) != naml) {
        return(2);
    }
}
/ * Get Real part of matrix from file */
if (!(*preal = (double huge *)halloc((long)mn,sizeof(double)))) {
    printf("Error: Variable too big to load\n");
    return(3);
}
if (fread(*preal, sizeof(double), mn, fp) != mn) {
    free(*preal);
    return(4);
}

/*
 * Get Imag part of matrix from file, if it exists
*/
if (x.imagf) {
    if (!(*pimag = (double huge *)halloc((long)mn,sizeof(double)))) {
        printf("Error: Variable too big to load\n");
        free(*preal);
        return(1);
    }
    if (fread(*pimag, sizeof(double), mn, fp) != mn) {
        free(*pimag);
        free(*preal);
        return(1);
    }
}
return(0);

savemat(fp, type, pname, mrows, ncols, imagf, preal, pimag)
FILE *fp;      /* File pointer */
int type;   /* Type flag: Normally 0 for PC, 1000 for Sun, Mac, and */
            /* Apollo, 2000 for VAX D-float, 3000 for VAX G-float */
            /* Add 1 for text variables. */
            /* See LOAD in reference section of guide for more info. */
int mrows;   /* row dimension */
int ncols;   /* column dimension */
int imagf;   /* imaginary flag */
char *pname; /* pointer to matrix name */
double huge *preal; /* pointer to real data */
double huge *pimag; /* pointer to imag data */
{
    Fmatrix x;
int mn;

x.type = type;
x.mrows = mrows;
x.ncols = ncols;
x.imagf = imagf;
x.namlen = strlen(pname) + 1;
mn = x.mrows * x.ncols;

fwrite(&x, sizeof(Fmatrix), 1, fp);
fwrite(pname, sizeof(char), (int)x.namlen, fp);
fwrite(preal, sizeof(double), mn, fp);
if (imagf) {
    fwrite(pimag, sizeof(double), mn, fp);
}
}

void readerr(int loaderr)
int loaderr;
{
    printf("\nREAD ERROR !!\n");
    switch(loaderr)
    {
    case 1:
        printf("\nCannot get Fmatrix structure");
        break;
    case 2:
        printf("\nCannot get matrix name");
        break;
    case 3:
        printf("\nCannot retrieve REAL part");
        break;
    case 4:
        printf("\nREAL size mismatch");
        break;
    case 5:
        printf("\nCannot retrieve IMAG part");
        break;
    case 6:
        printf("\nIMAG size mismatch");
        break;
    default:
        printf("\nError NOT defined");
        break;
    }
APPENDIX VIII: DASMAIN Software Listing (FORTRAN)

PROGRAM DASMAIN
C DMA ASSISTED PROGRAM CONTROLLED I/O
C DMA INPUT WITH DMA OUTPUT
C USED AS OUTER LOOP OF SERVO-SYSTEM
C ALLOWS DATA TRANSFER BETWEEN FORTRAN AND MATLAB
C DATA TRANSFERS ARE DONE IN FLAT ASCII FILES
C THIS CODE IS FOR 20TH ORDER AT 100 HZ (MAX)
C DANIEL ALLWINE -- 02/04/91
C LAST MODIFIED 05/12/91
C VARIABLES
 INTEGER*2 PARAM(10),DATA(1),CHAN,GAIN
 INTEGER*2 BASE,INTLEV,DMALEV,IMODE,RCODE
 INTEGER*2 DAS20,OFFADR,SEGPTR
 INTEGER*2 ITRATE,I,CRATE
 INTEGER*4 BUFFER,ALLOC
 REAL*8 SETPNT(10001),STORAG(10001),RDATA
 REAL*8 A(20,20),B(20,20),C(20,20),D(20,20)
 REAL*8 X(20,20),Y(20,20),U(20,20)
 INTEGER*2 N,ORDER
C NOTE: N=20 (FOR TIMING PURPOSES) BUT ORDER IS THE ACTUAL ORDER
C BEGIN
 PARAMETER (N=20,CRATE=6000)
 120 FORMAT(' MODE ', I2, ' , ERROR = ', I4)
C CONSTANTS
 BASE = #300
 INTLEV = 2
 DMALEV = 1
C CLEAR STORAG & SETPNT
 DO 5 I=1,10001
  STORAG(I)=0.00
  SETPNT(I)=0.00
5 CONTINUE
C INIT DAS20
 IMODE=0
 PARAM(1)=BASE
 PARAM(2)=INTLEV
 PARAM(3)=DMALEV
 RCODE = DAS20(IMODE,PARAM)
C ALLOCATE BUFFER
 BUFFER = ALLOC(1)
 IF (BUFFER.EQ.0) THEN
WRITE(6,*)'CANNOT ALLOCATE BUFFER'
ENDIF
C SET UP SCAN QUEUE
CHAN=0
GAIN=1
C LOAD SCAN QUEUE (MODE 1)
IMODE = 1
PARAM(3) = 2
PARAM(1) = CHAN
PARAM(2) = GAIN
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C INIT PACERS
PARAM(1)=CRATE
PARAM(2)=0
IMODE = 24
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
PARAM(1)=CRATE
PARAM(2)=0
IMODE = 25
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C INITIALIZE MATRICES
WRITE(6,*)'BEGINNING MATRIX INITIALIZATION'
CALL INIT (A,B,C,D,X,Y,U,ORDER,SETPNT,ITRATE)
C RUN DATA
DO 100 I = 1,ITRATE
C SET UP DMA INPUT
IMODE = 6
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 2
PARAM(4) = 1
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C CHECK FOR END OF DMA
98 CONTINUE
IMODE = 12
PARAM(1) = 1
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
IF (PARAM(2).NE.0) GO TO 98
C PUT DATA IN ARRAY
IMODE = 13
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 0
PARAM(4) = OFFADR(DATA)
PARAM(5) = -1
PARAM(6) = 1
PARAM(7) = 0
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C MANIPULATE DATA
IF (DATA(1).LT.2048) THEN
   RDATA = DATA(1)/204.8
ELSE
   RDATA = (DATA(1)-4096)/204.8
ENDIF
STORAG(I)=RDATA
CALL MANIP(A,B,C,D,RDATA,SETPNT,X,Y,U,I)
IF (RDATA.GE.0.00) THEN
   DATA(1)=RDATA*204.8
ELSE
   DATA(1)=(RDATA*204.8)+4096
ENDIF
IF (I.GE.ITRATE) THEN
   DATA(1) = 0.0
ENDIF
C PUT DATA IN MEMORY
IMODE = 8
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 0
PARAM(4) = OFFADR(DATA)
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C SET UP DMA OUTPUT
IMODE = 10
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 2
PARAM(4) = 1
PARAM(5) = 0
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C CHECK FOR END OF DMA
99 CONTINUE
  IMODE = 12
  PARAM(1) = 1
  RCODE = DAS20(IMODE, PARAM)
  IF (RCODE. NE. 0) WRITE(*,120) IMODE, RCODE
  IF (PARAM(2). NE. 0) GO TO 99
100 CONTINUE
C DUMP FOR POST ANALYSIS
C   WRITE(6,*)'DO YOU WISH TO DUMP DATA FOR MATLAB?'
C998   FORMAT(A1)
C   READ(5,998) CHOICE
C   IF ((CHOICE.EQ.'Y').OR.(CHOICE.EQ.'y')) THEN
C     CALL MATDMP(STORAG, ITRATE)
C   ENDIF
   STOP
END
C ........................................................................
SUBROUTINE INIT(A,B,C,D,X,Y,U,ORDER,SETPNT,RECNUM)
C STATE-VARIABLE MATRIX INITIALIZATION
C VARIABLES
C   A: SYSTEM MATRIX (N X N)
C   B: INPUT MATRIX (N X R)
C   C: OUTPUT MATRIX (M X N)
C   D: DIRECT COUPLING MATRIX (M X R)
C   X: STATE VECTOR
C   Y: OUTPUT VECTOR
C   U: INPUT VECTOR
C   SETPNT: SETPOINT VECTOR
C   RECNUM: NUMBER OF SETPOINT ELEMENTS
REAL*8 A(20,20), B(20,20), C(20,20), D(20,20), X(20,20), Y(20,20)
REAL*8 U(20,20)
REAL*8 SETPNT(10001), RLORD
INTEGER*2 I, J, N, ONE, ORDER, RECNUM
PARAMETER (N = 20, ONE = 1)
C BEGIN
C CLEAR ALL ARRAYS AND VECTORS
C CLEAR A MATRIX
C   CALL CLRMAT(A,N,N)
C CLEAR B MATRIX
C   CALL CLRMAT(B,N,ONE)
C CLEAR C MATRIX
C   CALL CLRMAT(C,ONE,N)
C CLEAR D MATRIX
C   CALL CLRMAT(D,ONE,ONE)
C CLEAR X VECTOR
    CALL CLRMAT(X,N,ONE)
C CLEAR Y VECTOR
    CALL CLRMAT(Y,ONE,ONE)
C CLEAR U VECTOR
    CALL CLRMAT(U,ONE,ONE)
C READ INPUT VECTOR
    OPEN(UNIT=2,FILE='INVEC.ASC')
    REWIND 2
    DO 784 I=1,10001
       RECNUM=I-1
       READ(2,*,END=785)SETPNT(I)
    784 CONTINUE
    785 CONTINUE
C READ ARRAYS FROM DISK
C GET SYSTEM ORDER
    OPEN(UNIT=2,FILE='ORDER.ASC')
    REWIND 2
    READ(2,*)RLORD
    ORDER=RLORD
C READ A MATRIX
    OPEN(UNIT=2,FILE='AMAT.ASC')
    REWIND 2
    DO 790 I=1,ORDER
       READ(2,*)(A(I,J),J=1,ORDER)
    790 CONTINUE
    CLOSE (UNIT=2)
C READ B ARRAY
    OPEN(UNIT=2,FILE='BMAT.ASC')
    REWIND 2
    DO 792 I=1,ORDER
       READ(2,*)B(I,1)
    792 CONTINUE
    CLOSE (UNIT=2)
C READ C ARRAY
    OPEN(UNIT=2,FILE='CMAT.ASC')
    REWIND 2
    READ(2,*)(C(l,J),J=1,ORDER)
    794 CONTINUE
    CLOSE (UNIT=2)
C READ D ARRAY
    OPEN(UNIT=2,FILE='DMAT.ASC')
    REWIND 2
    READ(2,*)D(1,1)
CLOSE (UNIT=2)
C ASK IF WISH TO VIEW MATRICES
C WRITE(6,*)'DO YOU WISH TO SEE ARRAYS? (Y/N)'
C WRITE(6,*)'NOTE: IF YOU DESIRE'
C WRITE(6,*)'USE <PAUSE> TO HALT DISPLAY.'
C WRITE(6,*)'ANY OTHER KEY CONTINUES.'
C 999 FORMAT(A1)
C READ(5,999)CHOICE
C IF(CHOICE.EQ.'Y') THEN
C WRITE A MATRIX
C WRITE(6,*)'A MATRIX'
C CALL WRTMAT(A,ORDER,ORDER)
C WRITE B ARRAY
C WRITE(6,*)'B ARRAY'
C CALL WRTMAT(B,ORDER,ONE)
C WRITE C ARRAY
C WRITE(6,*)'C ARRAY'
C CALL WRTMAT(C,ONE,ORDER)
C WRITE D ARRAY
C WRITE(6,*)'D ARRAY'
C CALL WRTMAT(D,ONE,ONE)
C ENDIF
WRITE(6,*)'INITIALIZATION COMPLETED'
RETURN
END
C........................................................................
SUBROUTINE WRTMAT(Z,XX,YY)
C VARIABLES
C Z: MATRIX TO BE WRITTEN
C XX: ROW DIMENSION
C YY: COLUMN DIMENSION
    REAL*8 Z(20,20)
    INTEGER*2 XX,YY,I,J
    900 FORMAT(' ',20(1PD8.2,1X))
    DO 910 I=1,XX
        WRITE(6,900)(Z(I,J),J=1,YY)
    910 CONTINUE
    RETURN
END
C........................................................................
SUBROUTINE CLRMAT(Z,ROW,COL)
C VARIABLES
C Z: MATRIX TO BE WRITTEN
    REAL*8 Z(20,20)
INTEGER*2 I,J,ROW,COL
C BEGIN
   DO 920 I=1,ROW
   DO 930 J=1,COL
      Z(I,J) = 0.00
   930 CONTINUE
   920 CONTINUE
RETURN
END

C *************************************************
SUBROUTINE MANIP(A,B,C,D,DATA,SETPNT,X,Y,U,I)
C VARIABLES
C   A: SYSTEM MATRIX (N X N)
C   B: INPUT MATRIX (N X 1)
C   C: OUTPUT MATRIX (1 X N)
C   D: I/O DIRECT COUPLING MATRIX (1 X 1)
C   DATA: SAMPLED VALUE (IN VOLTS)
C   SETPNT: SETPOINT ARRAY
C   X: STATE VECTOR (N X 1)
C   Y: OUTPUT VECTOR (1 X 1)
C   U: CURRENT INPUT VECTOR (1 X 1)
C   UOLD: PREVIOUS INPUT VECTOR (1 X 1)
C   N: NUMBER OF STATES (ORDER)
C   M: NUMBER OF OUTPUTS
C   R: NUMBER OF INPUTS
C   AX: PRODUCT OF A AND X (N X 1)
C   BU: PRODUCT OF B AND U (N X 1)
C   CX: PRODUCT OF C AND X (1 X 1)
C   DU: PRODUCT OF D AND U (1 X 1)
C   XK1: STATE VECTOR AT TIME (K+1)
C   ONE: 1
   REAL*8 A(20,20),B(20,20),C(20,20),D(20,20),X(20,20),Y(20,20)
   REAL*8 U(20,20),SETPNT(10001),DATA,XK1(20,20)
   REAL*8 AX(20,20),BU(20,20),CX(20,20),DU(20,20)
   INTEGER*2 N,ONE,I,J
   PARAMETER (N=20,ONE=1)
C BEGIN
C SET UP CORRECT VARIABLES
   U(1,1) = SETPNT(I)-DATA
C GENERATE X(K+1)
C MULTIPLY A BY X
   CALL MATMLT(A,N,N,X,ONE,AX)
C MULTIPLY B BY U
   CALL MATMLT(B,N,ONE,U,ONE,BU)
C ADD AX TO BU
   CALL MATADD(AX,BU,N,ONE,XK1)
C GENERATE Y(K)
C MULTIPY C BY X
   CALL MATMLT(C,ONE,N,X,ONE,CX)
C MULTIPY D BY U
   CALL MATMLT(D,ONE,ONE,U,ONE,DU)
C ADD CX TO DU
   CALL MATADD(CX,DU,ONE,ONE,Y)
DATA = Y(1,1)
C INCREMENT X
   DO 111 J=1,20
       X(J,1)=XK1(J,1)
   111 CONTINUE
RETURN
END
C ******************************************************
SUBROUTINE MATMLT(A,R1,C1,B,C2,PD)
C VARIABLES
C A: MULTIPLIER MATRIX
C R1: ROW DIM OF A
C C1: COLUMN DIM OF A
C B: MULTIPLICAND MATRIX
C C2: COLUMN DIM OF B
C PD: PRODUCT MATRIX (R1 X C2)
C
C NOTE : #COL IN A = #ROW IN B
C NOTE : C(I,J) = A(IROW)*B(JCOL)
C
REAL*8 A(20,20),B(20,20),PD(20,20)
INTEGER*2 R1,C1,C2,I,J,K
C BEGIN
   DO 10 I=1,R1
      DO 20 J=1,C2
         PD(I,J) = 0.0
      20 CONTINUE
   10 CONTINUE
RETURN
END
C ******************************************************
SUBROUTINE MATADD(A,B,ROW,COL,SUM)
C VARIABLES
C A: FIRST MATRIX FOR ADDITION
C B: SECOND MATRIX FOR ADDITION
C ROW: ROW DIMENSION
C COL: COLUMN DIMENSION
C SUM: SUM OF A AND B (ROW X COL)
REAL*8 A(20,20),B(20,20),SUM(20,20)
INTEGER*2 ROW,COL,I,J
C BEGIN
   DO 10 I=1,ROW
       DO 20 J=1,COL
           SUM(I,J)=0.0
           SUM(I,J)=A(I,J)+B(I,J)
20 CONTINUE
10 CONTINUE
RETURN
END

C SUBROUTINE MATDMP(STORAG,ITRATE)
C VARIABLES
C STORAG: ARRAY CONTAINING SAMPLED DATA
C ITRATE: NUMBER OF ITERATIONS PERFORMED
REAL*8 STORAG(10001)
INTEGER*2 ITRATE,I
C BEGIN
   WRITE(6,*)'DUMPING DATA TO "DASDAT.ASC" IN FLAT ASCII FORMAT'
   OPEN(UNIT=2,FILE='DASDAT.ASC')
   REWIND 2
   DO 10 I=1,ITRATE
       WRITE(2,*)STORAG(I)
10 CONTINUE
   CLOSE(UNIT=2)
   WRITE(6,*)'DATA TRANSFER COMPLETE'
RETURN
END
APPENDIX IX: DCCALIB Software Listing (FORTRAN)

PROGRAM DCCALIB
C DMA ASSISTED PROGRAM CONTROLLED I/O
C DMA INPUT WITH DMA OUTPUT
C USED AS DC GAIN CALIBRATION SYSTEM
C ALLOWS DATA TRANSFER BETWEEN FORTRAN AND MATLAB
C DATA TRANSFERS ARE DONE IN FLAT ASCII FILES
C DANIEL ALLWINE -- 02/22/91
C LAST MODIFIED 04/05/91
C VARIABLES
  INTEGER*2 PARAM(10),DATA(I),CHAN,GAIN
  INTEGER*2 BASE,INTLEV,DMALEV,IMODE,RCODE,STORAG(16384)
  INTEGER*2 DAS20,OFFADR,SEGPTR
  INTEGER*2 ITRATE,I,CRATE
  INTEGER*4 BUFFER,ALLOC
  REAL*8 SETPNT(16384)
C BEGIN
  PARAMETER (CRATE=23500)
  120 FORMAT(' MODE ',I2,' ',ERROR = ',I4)
C CONSTANTS
  BASE = #300
  INTLEV = 2
  DMALEV = 1
C CLEAR STORAG & SETPNT
  DO 5 I=1,16384
     STORAG(I)=0
     SETPNT(I)=0.00
  5 CONTINUE
C INIT DAS20
  PARAM(1)=BASE
  PARAM(2)=INTLEV
  PARAM(3)=DMALEV
  RCODE = DAS20(IMODE,PARAM)
C ALLOCATE BUFFER
  BUFFER = ALLOC(1)
  IF (BUFFER.EQ.0) THEN
     WRITE(6,*)'CANNOT ALLOCATE BUFFER'
   ENDIF
C SET UP SCAN QUEUE
  CHAN=0
  GAIN=1
C LOAD SCAN QUEUE (MODE 1)
  IMODE = 1
PARAM(3) = 1
PARAM(1) = CHAN
PARAM(2) = GAIN
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C INIT PACERS
PARAM(1)=CRATE
PARAM(2)=0
IMODE = 24
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
IMODE = 25
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C GET SETPOINT VECTOR
OPEN(UNIT=2,FILE='CALVOLT.ASC')
DO 96 I=1,16384
   READ(2,*END=97)SETPNT(I)
   RECNUM=I
96 CONTINUE
97 CONTINUE

C RUN DATA
ITRATE=RECNUM
DO 100 I = 1,ITRATE+1

C SET UP DMA INPUT
IMODE = 6
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 2
PARAM(4) = 1
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C CHECK FOR END OF DMA
98 CONTINUE
IMODE = 12
PARAM(1) = 1
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
IF (PARAM(2).NE.0) GO TO 98

C PUT DATA IN ARRAY
IMODE = 13
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 0
PARAM(4) = OFFADR(DATA)
PARAM(5) = -1
PARAM(6) = 1
PARAM(7) = 0
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C MANIPULATE DATA
STORAG(I)=DATA(I)
DATA(1)=SETPNT(I+1)
IF (I.GE.(ITRATE+1)) THEN
  DATA(1) = 0.0
ENDIF

C PUT DATA IN MEMORY
IMODE = 8
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 0
PARAM(4) = OFFADR(DATA)
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C SET UP DMA OUTPUT
IMODE = 10
PARAM(1) = 1
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 2
PARAM(4) = 1
PARAM(5) = 0
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C CHECK FOR END OF DMA
99 CONTINUE
IMODE = 12
PARAM(1) = 1
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
IF(PARAM(2).NE.0) GO TO 99
100 CONTINUE

C DUMP TO FLAT ASCII FILE
   CALL MATDMP(STORAG,ITRATE)
STOP
END

C **********************************************
SUBROUTINE MATDMP(STORAG,ITRATE)
C VARIABLES
C STORAG: ARRAY CONTAINING SAMPLED DATA
C ITRATE: NUMBER OF ITERATIONS PERFORMED
   INTEGER*2 STORAG(16384),ITRATE,I
C BEGIN
   WRITE(6,*)'DUMPING DATA TO "DASDAT.ASC" IN FLAT ASCII FORMAT'
   OPEN(UNIT=2,FILE='DASDAT.ASC')
   REWIND 2
   DO 10 I=1,ITRATE
      WRITE(2,*)STORAG(I)
   10 CONTINUE
   CLOSE(UNIT=2)
   WRITE(6,*)'DATA TRANSFER COMPLETE'
   RETURN
END
APPENDIX X: DCCALIB2 Software Listing (FORTRAN)

PROGRAM DCCALIB2
C 2 CHANNEL -- 5000 SAMPLES MAX
C DMA ASSISTED PROGRAM CONTROLLED I/O
C DMA INPUT WITH DMA OUTPUT
C USED AS DC GAIN CALIBRATION SYSTEM
C ALLOWS DATA TRANSFER BETWEEN FORTRAN AND MATLAB
C DATA TRANSFERS ARE DONE IN FLAT ASCII FILES
C DANIEL ALLWINE -- 03/30/91
C LAST MODIFIED 04/05/91
C VARIABLES
    INTEGER*2 PARAM(10),DATA(2),CARRAY(2),GARRAY(2)
    INTEGER*2 BASE,INTLEV,DMALEV,IMODE,RCODE
    INTEGER*2 STOR1(5000),STOR2(5000)
    INTEGER*2 DAS20,OFFADR,SEGPTR
    INTEGER*2 ITRATE,I,CRATE
    INTEGER*4 BUFFER,ALLOC
    REAL*8 SETPNT(5000)
C BEGIN
   PARAMETER (CRATE=23500)
120 FORMAT(‘ MODE ’, I2, ’ , ERROR ’, I4)
C CONSTANTS
    BASE = #300
    INTLEV = 2
    DMALEV = 1
C CLEAR STORAG & SETPNT
   DO 5 I=1,5000
       STOR1(I)=0
       STOR2(I)=0
       SETPNT(I)=0.00
5 CONTINUE
C INIT DAS20
   IMODE=0
   PARAM(1)=BASE
   PARAM(2)=INTLEV
   PARAM(3)=DMALEV
   RCODE = DAS20(IMODE,PARAM)
C ALLOCATE BUFFER
   BUFFER = ALLOC(2)
   IF (BUFFER.EQ.0) THEN
       WRITE(6,*)(‘CANNOT ALLOCATE BUFFER’)
   ENDIF
C SET UP SCAN QUEUE
WRITE(6,*)'SETTING CHANNEL = 0  GAIN = 1'
WRITE(6,*)'SETTING CHANNEL = 1  GAIN = 1'
WRITE(6,*)'BIPOLAR (+/-10 VOLTS)'
CARRAY(1)=0
CARRAY(2)=1
GARRAY(1)=1
GARRAY(2)=1
N=2
C LOAD QUEUE (MODE 1)
WRITE(6,*)'LOADING QUEUE'
IMODE = 1
PARAM(3) = 2
DO 601 I = 1,N
   PARAM(1) = CARRAY(I)
   PARAM(2) = GARRAY(I)
   RCODE = DAS20(IMODE,PARAM)
   IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
   PARAM(3) = 0
   IF (I.EQ.(N-1)) PARAM(3) = 1
601 CONTINUE
C CHECK CHAN/GAIN
WRITE(6,*)'CHANNEL SELECT = ',CARRAY(1),CARRAY(2)
WRITE(6,*)'GAIN SELECT = ',GARRAY(1),GARRAY(2)
C INIT PACERS
PARAM(1)=CRATE
PARAM(2)=0
IMODE = 24
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
IMODE = 25
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C GET SETPOINT VECTOR
OPEN(UNIT=2,FILE='CALVOLT.ASC')
DO 96 I=1,5000
   READ(2,*,END=97)SETPNT(I)
   RECNUM=I
96 CONTINUE
97 CONTINUE
C RUN DATA
ITRATE=RECNUM
DO 100 I = 1,ITRATE
   C SET UP DMA INPUT (THIS GETS ONE SAMPLE FROM EACH CHANNEL)
      IMODE = 6
PARAM(1) = 2
PARAM(2) = SEGPTR(BUFFER)
PARAM(3) = 2
PARAM(4) = 1
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C CHECK FOR END OF DMA
98 CONTINUE
  IMODE = 12
  PARAM(1) = 1
  RCODE = DAS20(IMODE,PARAM)
  IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
  IF (PARAM(2).NE.0) GO TO 98

C PUT DATA IN ARRAY (TRANSFERS 2 SAMPLES)
  IMODE = 13
  PARAM(1) = 2
  PARAM(2) = SEGPTR(BUFFER)
  PARAM(3) = 0
  PARAM(4) = OFFADR(DATA(1))
  PARAM(5) = 0
  PARAM(6) = 1
  PARAM(7) = 0
  RCODE = DAS20(IMODE,PARAM)
  IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C MANIPULATE DATA
  STOR1(I) = DATA(1)
  STOR2(I) = DATA(2)
  DATA(1) = SETPNT(I+1)
  IF (I.GE.(ITRATE+1)) THEN
    DATA(1) = 0.0
  ENDIF

C PUT DATA IN MEMORY
  IMODE = 8
  PARAM(1) = 1
  PARAM(2) = SEGPTR(BUFFER)
  PARAM(3) = 0
  PARAM(4) = OFFADR(DATA(1))
  RCODE = DAS20(IMODE,PARAM)
  IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE

C SET UP DMA OUTPUT
  IMODE = 10
  PARAM(1) = 1
  PARAM(2) = SEGPTR(BUFFER)
  PARAM(3) = 2
PARAM(4) = 1
PARAM(5) = 0
RCODE = DAS20(IMODE,PARAM)
IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
C CHECK FOR END OF DMA
99 CONTINUE
   IMODE = 12
   PARAM(1) = 1
   RCODE = DAS20(IMODE,PARAM)
   IF (RCODE.NE.0) WRITE(*,120) IMODE,RCODE
   IF(PARAM(2).NE.0) GO TO 99
100 CONTINUE
C DUMP TO FLAT ASCII FILE
   CALL MATDPl(STOR1,ITRATE)
   CALL MATDP2(STOR2,ITRATE)
STOP
END
C ******************************************************
SUBROUTINE MATDPl(STORAG,ITRATE)
C VARIABLES
C STORAG: ARRAY CONTAINING SAMPLED DATA
C ITRATE: NUMBER OF ITERATIONS PERFORMED
   INTEGER*2 STORAG(5000),ITRATE,I
C BEGIN
   WRITE(6,*)'DUMPING DATA TO "DASDAT1.ASC" IN FLAT ASCII
   FORMAT'
   OPEN(UNIT=2,FILE='DASDAT1.ASC')
   REWIND 2
   DO 10 I=1,ITRATE
      WRITE(2,*)STORAG(I)
   10 CONTINUE
   CLOSE(UNIT=2)
   WRITE(6,*)'DATA TRANSFER COMPLETE'
RETURN
END
C ******************************************************
SUBROUTINE MATDP2(STORAG,ITRATE)
C VARIABLES
C STORAG: ARRAY CONTAINING SAMPLED DATA
C ITRATE: NUMBER OF ITERATIONS PERFORMED
   INTEGER*2 STORAG(5000),ITRATE,I
C BEGIN
   WRITE(6,*)'DUMPING DATA TO "DASDAT2.ASC" IN FLAT ASCII
   FORMAT'
OPEN(UNIT=2, FILE='DASDAT2.ASC')
REWIND 2
DO 10 I=1, ITRATE
   WRITE(2,*) STORAG(I)
10 CONTINUE
CLOSE(UNIT=2)
WRITE(6,*) 'DATA TRANSFER COMPLETE'
RETURN
END