1991

Computed tomography: experimental data acquisition and parallelization of reconstruction algorithm

Richard K. Powell
Iowa State University

Computed tomography: Experimental data acquisition and parallelization of reconstruction algorithm

by

Richard Karl Powell

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

Iowa State University
Ames, Iowa

1991

DEDICATION

To my parents who have taught, by example, the merits of a formal education and who have instilled the worth of good values. I would especially like to dedicate this thesis to the memory of my mother, who as both mother and teacher shared her love, knowledge, and dedication with many.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

1.1 Need for NDE

Non-destructive Evaluation of components is a need throughout much of the manufacturing industry that involves high value parts. The aircraft companies are a prime example of this need. The fleet of commercial and military aircraft that is in use worldwide today is aging and consequently it has become critical to develop an economical real time inspection of aircraft components such as wing structures, engine mounts, turbine blades, and various other critical components in modern day aircraft. The materials used in aircraft structures vary substantially today, indeed it is commonplace to see exotic composite materials as well as the more conventional materials such as aluminum and titanium.

There are three primary methods used at the Center to do NDE inspections: 1. X-ray based, 2. Ultrasonics based, and 3. Eddy current based. Eddy currents are widely used in industry for the determination of the existence of surface cracks and their lengths in conductors. Ultrasonics is used extensively in the production and in-service inspection of composite panels for problems such as delamination and for cracks in metals. X-ray methods are used during production of composite based materials and with portable generators, for in-service detection of fatigue cracks. X-ray computed tomography image analysis of voids, cracks, and inclusions are currently finding widespread usage in both the aircraft industry and in many other industrial applications as well. Although each of these inspection types has its own merits, it is basic research into some of the X-ray based methods that will be discussed in this thesis.

1.2 Project Overview

The project discussed in this thesis covers the specification of a computational platform for a computer based NDE X-ray inspection research system, followed by the installation of data acquisition and sample positioning equipment. It then continues to include the writing and testing of software to control this hardware.

System capabilities - There are three types of scans done here at the Center, a one dimensional scan (1D scan), a two dimensional scan (2D scan), and a tomographic scan (tomo scan). These three types of scans are to be done automatically under computer control with user input of scan parameters. An explanation of the X-ray lab setup will first be outlined, followed by explanations of the these three scan types outlining the differences in sample positioning for each.

The setup here at the Center has a fixed position x-ray source, and a fixed position photon detector in line with the pencil beam source (refer to Figure 1). In between the source and the detector there is a sample positioner which can move the sample through the pencil beam in several different directions along axes which have been labeled as follows: two linear axes labelled, X for the horizontal axis, and Z for the vertical axis. In addition to these two linear axes, there is one rotational axis called the Θ axis that is used for the tomographic scan. Further explanation of each of these types of scans follows.

1D scan - The one dimensional scan is done on any of the available linear axes. The present setup in the lab makes use of the X and Z linear axes (refer to Figure 1). A 1D scan can be done along either of these axis in either the positive or negative directions. The procedure is as follows: setup the sample to the desired initial position, then acquire photon count data by setting at this position

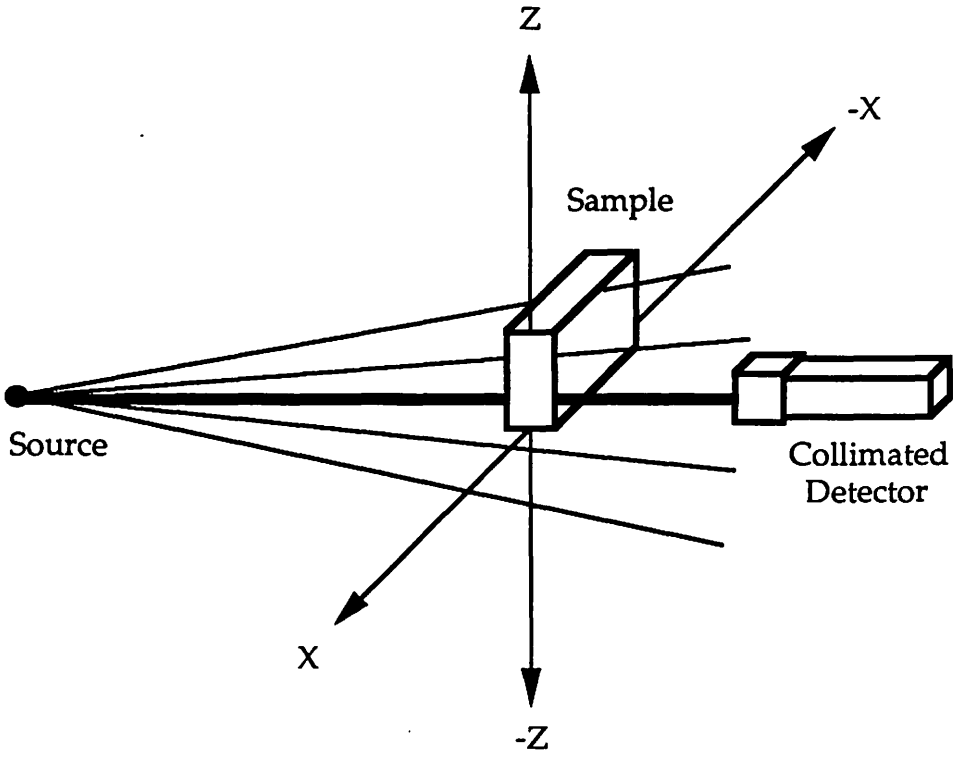


Figure 1. Setup for 1D scan and 2D scan

for a given time interval counting photons, then move the sample one increment in the desired direction and stop and count photons once again. This procedure is repeated until a given total distance on the chosen axis is traversed. At the conclusion of this data acquisition the sample would be returned to its original starting position.

2D scan - The two dimensional scan is similar to the 1D scan with the addition of one more linear axis so the scan is then done in a two dimensional plane (the X-Z plane). As an example, assume a 2D scan along the X and Z axes is chosen with the following parameters: a 2.0 inch total scan distance along both axes, an increment of 0.01 inches also along both axes. This scan will then do the following: referring to Figure 1, it would first do a 1D scan along the X axis, it then would move the sample one increment in the vertical Z axis direction and again do the 1D scan as before. This process would repeat until all of the 2D X-Z plane points have been traversed and photon counts have been acquired at each of these points. This would yield a total of 200 x 200 = 40,000 moves and 201 x 201 = 40,401 photon counts. The large number of counts that are acquired leads to rather large file space requirements as well as program memory considerations. Using this example, there is currently 10 bytes per count in the ASCII GRD output file format (see Appendix C for an explanation of the GRD format), so 10 x 40,401 counts = 404,010 bytes (almost half a megabyte in just one output file). These files can be stored in a binary GRD file representation which dramatically reduces their size (i.e. binary file size is approx. 8% of the ASCII file size). The binary format is not currently used because it is frequently desirable to make changes to these files and this is easily done using a text editor if the file is an ASCII file.

Tomo scan - The motion control for the tomo scan is similar to that of the 2D scan with the exception that one rotational axis is used instead of one of the linear axes. This is an implementation of a first generation tomography scan. Referring to Figure 2, it can be seen that the two axes used presently are the X and Θ axes. This scan is done identically to the 2D scan example above, except that the 'outer' scan axis is now the rotational axis. It is common to sweep the Θ axis through a total of 180 degrees in order to yield a complete set of projection data. This is done to enable data to be taken by a point detector in a first generation computed tomography (CT) scan for CT image analysis. The interested reader is referred to references [1] and [2] for a detailed description of computed tomography and the corresponding theory of image reconstruction from tomographic projections.

General acquisition system needs - The first priority in selecting equipment for this type of acquisition system is to determine exactly what data is to be acquired. The types of equipment needed in a computer based data acquisition system include: a computer, data acquisition equipment (to process raw analog data and put it into digital form), detector (sensing) equipment, and sample positioning system. The data acquisition and positioner equipment must both interface with the computer. The control program needs to detect or sense the occurrence of the events being acquired. Finally, the data must be stored and, as required, processed.

Detectors - A photon detector must be chosen and then connected through its electronics to provide an analog signal representing X-ray photon counts. These analog signals then interface to the computer through data analyzers (containing analog to digital conversion and control logic). There are two types of data analyzers available at the Center, a single-channel analyzer (SCA), and a multi-channel analyzer (MCA). The basic difference is that the multi-channel analyzer is

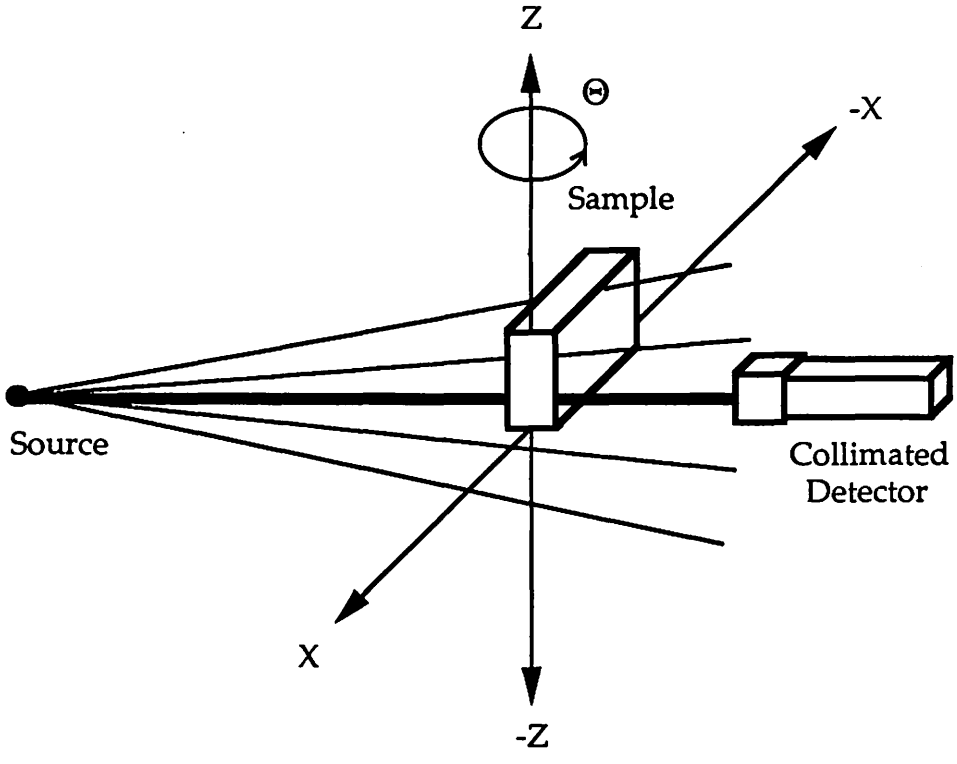


Figure 2. Setup for tomo scan

capable of acquiring multiple bin energy sensitive photon counts where the single-channel analyzer is an integration of photons of all energies within a single energy bin. The MCA is a two slot ISA compatible board that is installed in the computer and communicates via software written by the programmer. Details of this communication are included in Chapter 3.

Sample positioner - The sample positioner consists of a single ISA compatible board that is installed in the computer connected to a microprocessor based motor controller, which is then connected to the motor drivers. Each board controls three axes. Communication between the computer and this board is the responsibility of the programmer. This communication scheme is described in detail in Chapter 3.

Computer - The computer used for this system must be IBM compatible with an industry standard ISA compatible bus. The computer currently used at the Center for this system is a 20 MHz 80386 based IBM compatible personal computer with a 320 Meg. hard disk, 4 Meg. of RAM, and both 3.5" and 5.25" diskette drives. There are interface issues that arise when installing the sample positioner and MCA boards within this computer. These issues will next be addressed.

Interface issues - The physical installation of the positioner and data acquisition boards within the computer presents interfacing problems in trying to get them to coexist with other boards already installed in the system (i.e. an ethernet board, a Mach 2 tape controller board, the video board, disk controller board, and the mouse).

Problems seen in this installation were all related to interrupt request (Irq) and PC I/O bus address conflicts. There are a limited number of Irq lines in an IBM AT compatible machine and some peripherals such as the mouse must use specific

ones. The installation process is a matter of shuffling around the Irq and bus addresses used by the positioner controller and data acquisition boards so that address or Irq conflicts with other peripherals are avoided. Both of these boards have user selectable Irq and bus address switches. This type of problem is seen only when there are numerous peripherals installed.

Another interface issue involves the need to interface this PC to a DEC workstation network through an ethernet board. The interface has the same potential conflict problems, but the most troublesome problem was with the network driver software for the PC. In interfacing this PC the Apollo network, and then later to a DEC network, the primary problem was this software was so new that the bugs were yet to be worked out. One of the problems that arose in the use of this network interface, was in the way that ASCII file transfers are handled. The Apollo software took the approach that in would automatically convert the DOS <CR> <LF> pairs to just <CR> (the Apollos and DECs use only the <CR> for line terminations). However, the DEC software takes the approach that it does not interfere with any file that is transferred. Therefore, it is the responsibility of the person transferring the ASCII files to/from the DEC to run a filter program to convert to the proper line termination format for the target machine. This type of problem became apparent when transferring FORTRAN and C source code from one machine type to the other. The code that was fine on one machine would blow up with syntax error messages on the target machine. The problem was the DEC and Apollo workstations treated the extra <LF> character as a syntax error.

Interfacing to the tape controller board (installed to allow high speed tape archiving of image files) is another case of the conflict issue. The installation again involves the careful selection of Irq and bus addresses such that no conflicts occur.

This project also involved looking into interfacing an existing single channel analyzer through a Tennelec buffer interface for scans that do not need the multi-bin energy sensitive abilities of a multi-channel analyzer. The buffer interface is designed to communicate with a host computer through a RS232 serial port. This requires that an interrupt service routine (ISR) be written to process the serial port data. The best to way to write an ISR such that it would coexist with our operating environments (DesqView's multitasking system) was explored. This particular part of the project was postponed in order to implement the new and more flexible multi-channel interface. The need to use the buffer interface and consequently the serial port interface can be avoided by interfacing the single channel analyzer amplifier directly to a Counter/Timer board installed in the computer bus.

Code implementation - After successful hardware installation, the software needed to control this system in the scan types described had to be written. The language chosen was C for a number of reasons, including the ability of the C language to directly manipulate hardware registers, to do bit manipulations, and its powerful pointer capabilities. Neither FORTRAN or Pascal allow the programmer to get as close to the hardware as does C. This ability is very important in controlling these types of boards. Also, FORTRAN has no pointer capability and Pascal's pointers are not as powerful as those of C. It should be pointed out, though, that either FORTRAN or Pascal could be used to write these interface programs. In fact, there was good direct support for FORTRAN code provided by the board manufacturers. However, C was the language of choice and lent itself well to the task. A detailed description of the program modules that handle the communication and control of this system is included in Chapter 3.

An additional issue encountered in the implementation of the code involved interfacing the original working version of this program to text based menuing user interface code. This took considerable time, as it lead to significant changes in the code and to the introduction of numerous bugs.

Code testing - There were several types of testing that needed to be done to verify the code. The first and most important was to verify that the sample positioning was working properly. This involved placing absolute position markers on the positioner platforms in order to monitor the positions actually moved. Through this type of testing code problems were found. These are discussed further in Chapter 5. Additional testing involved doing example various scans of the different types and then analyzing the output data.

Data storage - As noted earlier, the volume of data generated by the various scan types is large, thus driving the need for archival storage of this data. Several data backup methods were evaluated over the course of this project. These methods included floppy disks with compressed files, however this method became less and less feasible as the volume of files became too large.

We were able to initially get around the purchase of additional hardware for this backup by using the ethernet cards that have been installed in our personal computers. We tied into the Apollo network of workstations to backup our files onto the larger hard disks there. These directories of PC files would then be backed up to tape by the Apollo network administrators.

Eventually, it became desirable to obtain our own tape backup system. The author was involved in shopping for, purchasing, and then installing a tape backup unit. An external unit was chosen so it could be used by any of our PCs. We also purchased the Mach 2 tape controller board to increase throughput in our backups.

The unit purchased and now used is the Mountain Network Systems' Filesafe 8000. Each mini-cartridge will hold 152 MBytes in standard mode and they are capable of holding 304 MBytes per cartridge in compressed mode. A throughput of approximately 6 MBytes/min is realized.

Data analysis - Another task undertaken in this project was to look into the possibility of using a massively parallel computer to aid in speeding up the tomographic reconstruction and image processing algorithms used by our group at the Center. These algorithms are very compute intensive. It was therefore hoped that a machine such as the Connection Machine (built by Thinking Machines, Inc.) would provide a dramatic speedup for these algorithms. As both a test of its abilities and a test of the programming effort required to convert a serial program into a parallel program to run, a considerable amount of time was spent looking into implementing an image processing filter program on the Connection Machine (CM). Further issues involved in parallel processing, additional details of the Connection Machine architecture, and additional information on the filter algorithm can be found in chapter 4 of this thesis.

1.3 Sample Images

Figures 3, 5, 6, and 7 are examples of the data analysis output that is used here at the Center for each of the 3 scan types. Figure 3 is a graph of photon counts versus position for a 1D scan of a resolution gauge used in determining the resolution of a particular image setup (see Figure 4 for a sketch of a resolution gauge). The low portions of the graph immediately after the Y axis (left side of graph) and at the far right of the graph are the photon counts seen through the metal of the sides of the gauge. The high spot in between with bumps show the

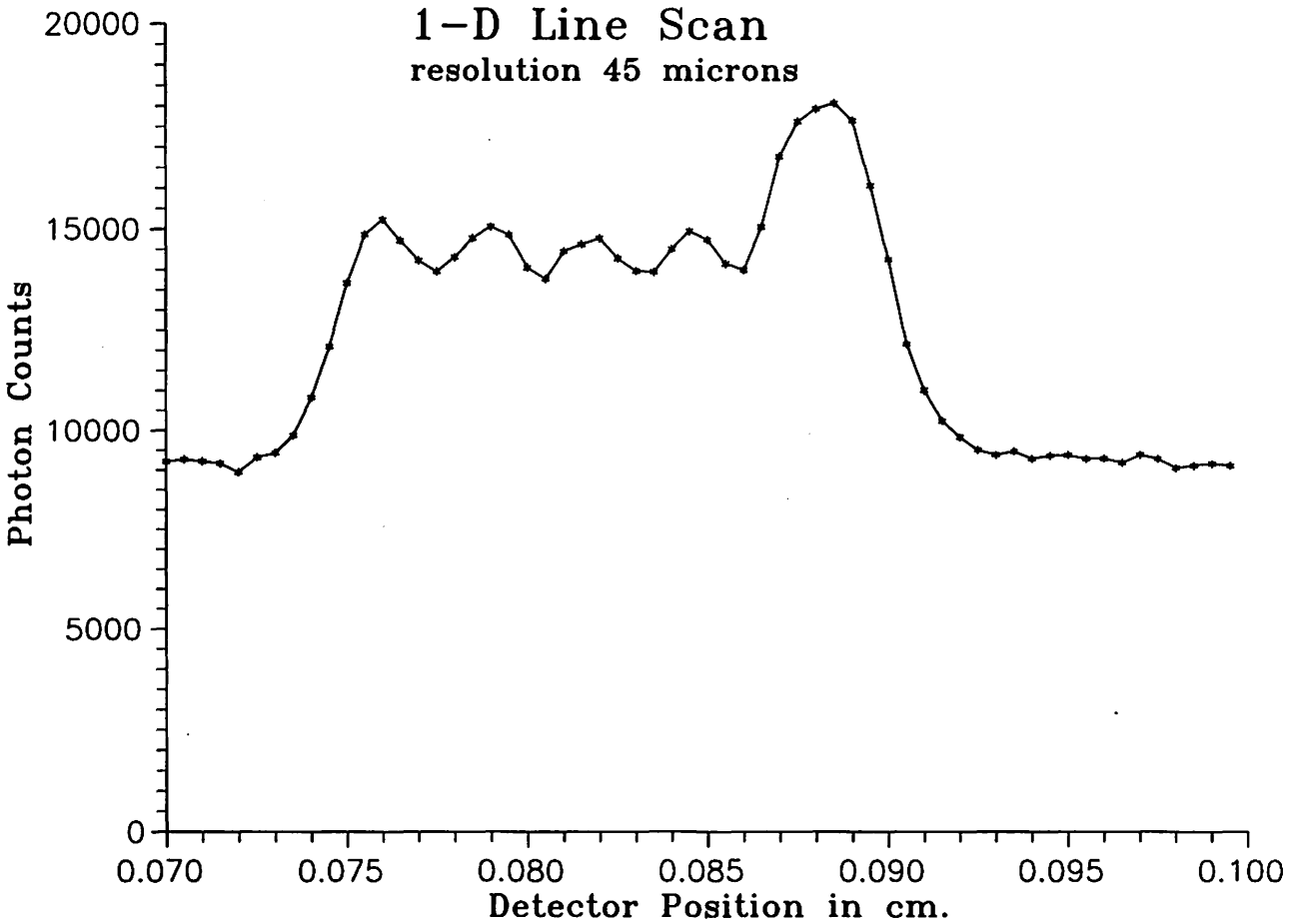
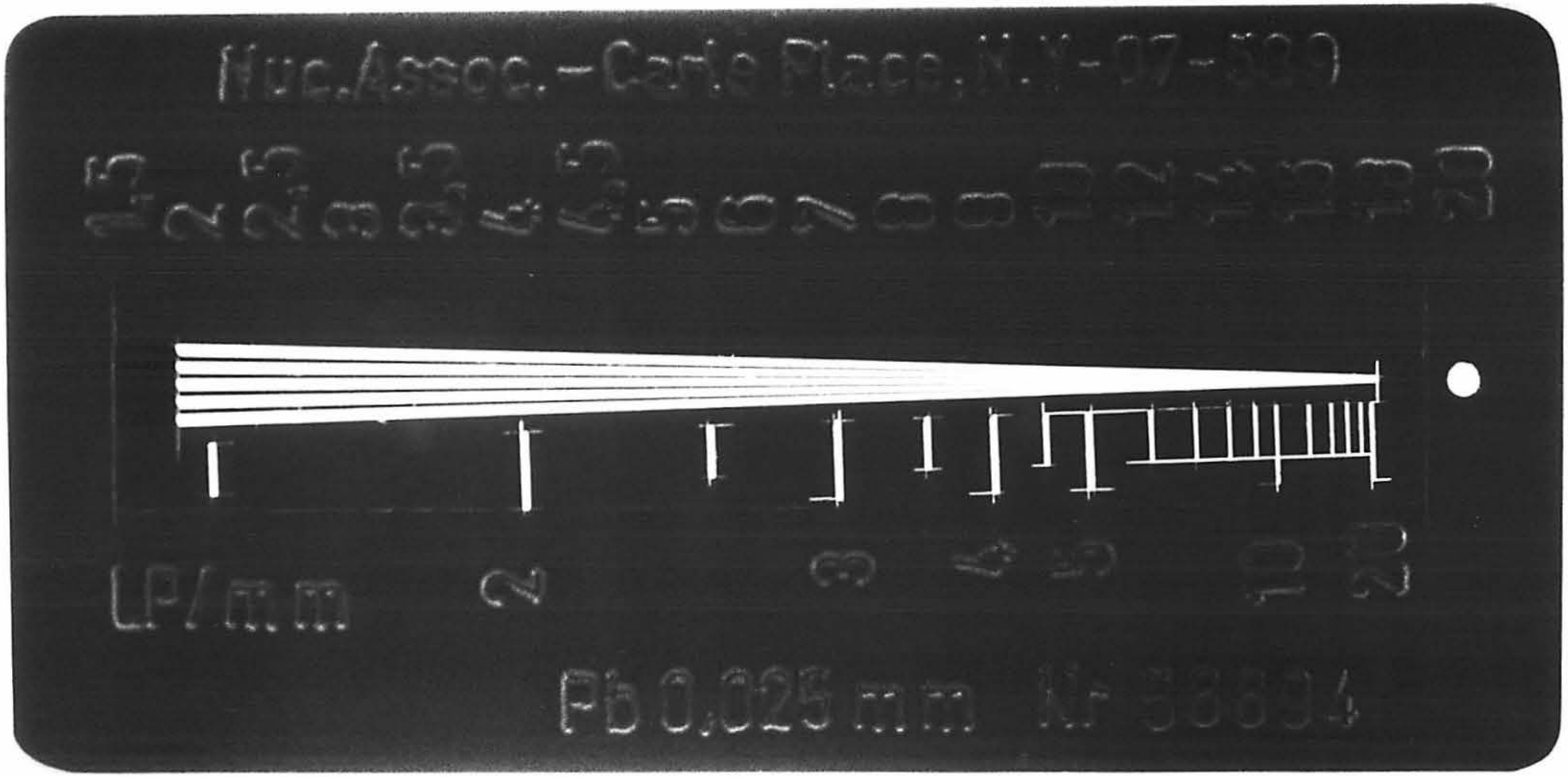


Figure 3. 1D scan of resolution gauge

Figure 4. Picture of resolution gauge.


Nuc.Assoc.-Carle Place, N.Y.
LP/mm
Pb 0,025 mm

photon intensity fluctuations from moving across the gauge openings. First the metal of the gauge is crossed, then back to open air as the beam passes an opening in the gauge and this repeats until the gauge has been traversed and once again the beam is back into the solid gauge portion. This allows us to see the size of a void or crack that can be distinguished with this setup.

Figure 5 is a contour plot of a 2D scan of a kogwheel. It is a top view of the resultant topology in the scanned region of this kogwheel. A couple of the kogwheel teeth are readily visible (towards the upper right corner and in the middle of the left side) and help orient the plot. A flaw in the kogwheel is also visible. The circular shaped contour lines packed closely together gives rise to the existence of a flaw in the kogwheel.

Figure 6 is a graph of the output data taken with a tomo scan (this type of direct graph of the raw output data is called a sinogram). The sample consisted of two aluminum cylinders that contain drilled holes of various sizes and depths. Figure 7 is the resultant tomographic reconstruction image that shows the slice taken through these two cylinders. The slice was chosen to cut through all of the drilled holes and as shown in the reconstructed image (Figure 7), these holes are visible. This technique is powerful and can be used to greatly enhance the ability to locate a void, crack, or inclusion because it yields three dimensional information as to its location.

Chapter 5 contains some conclusions including comments about testing of this software, and recommendations as to what be could be done to improve on the system.

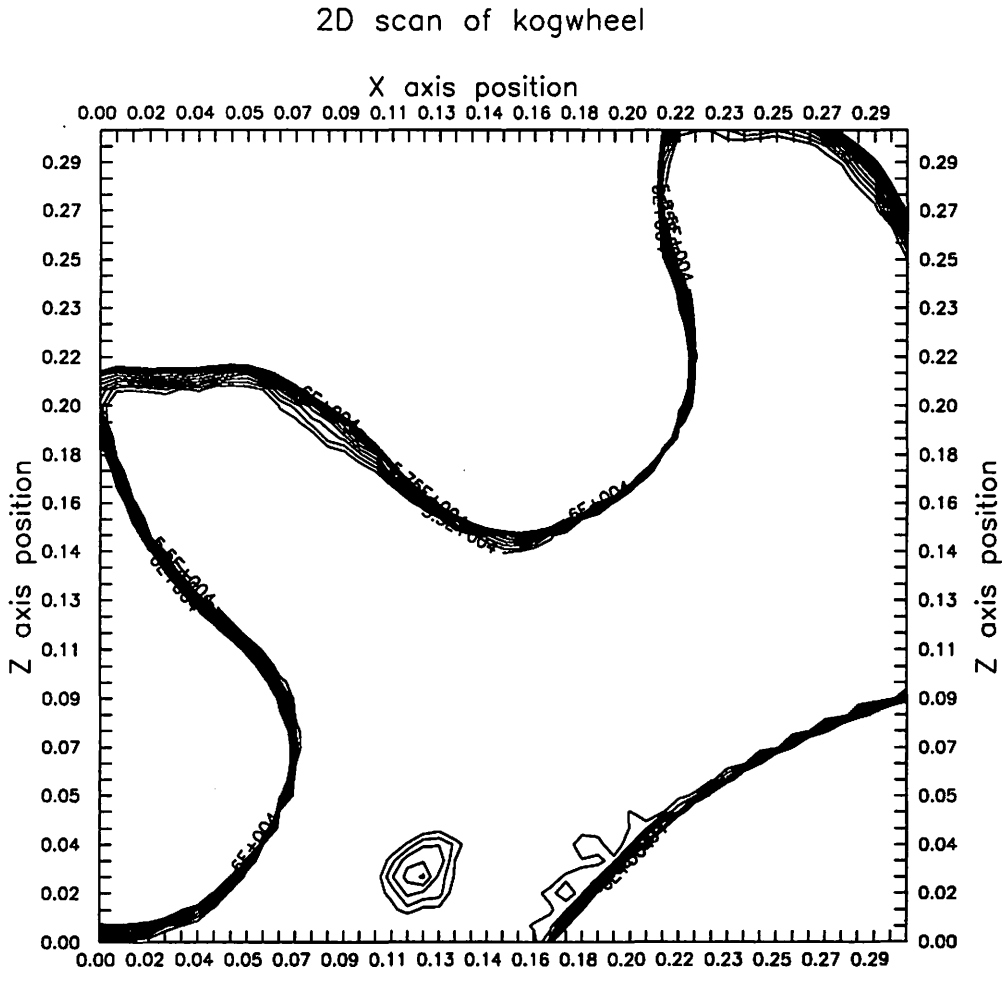


Figure 5. 2D scan of kogwheel, interior void is visible

Figure 6. Sinogram of tomo scan of two cylinders

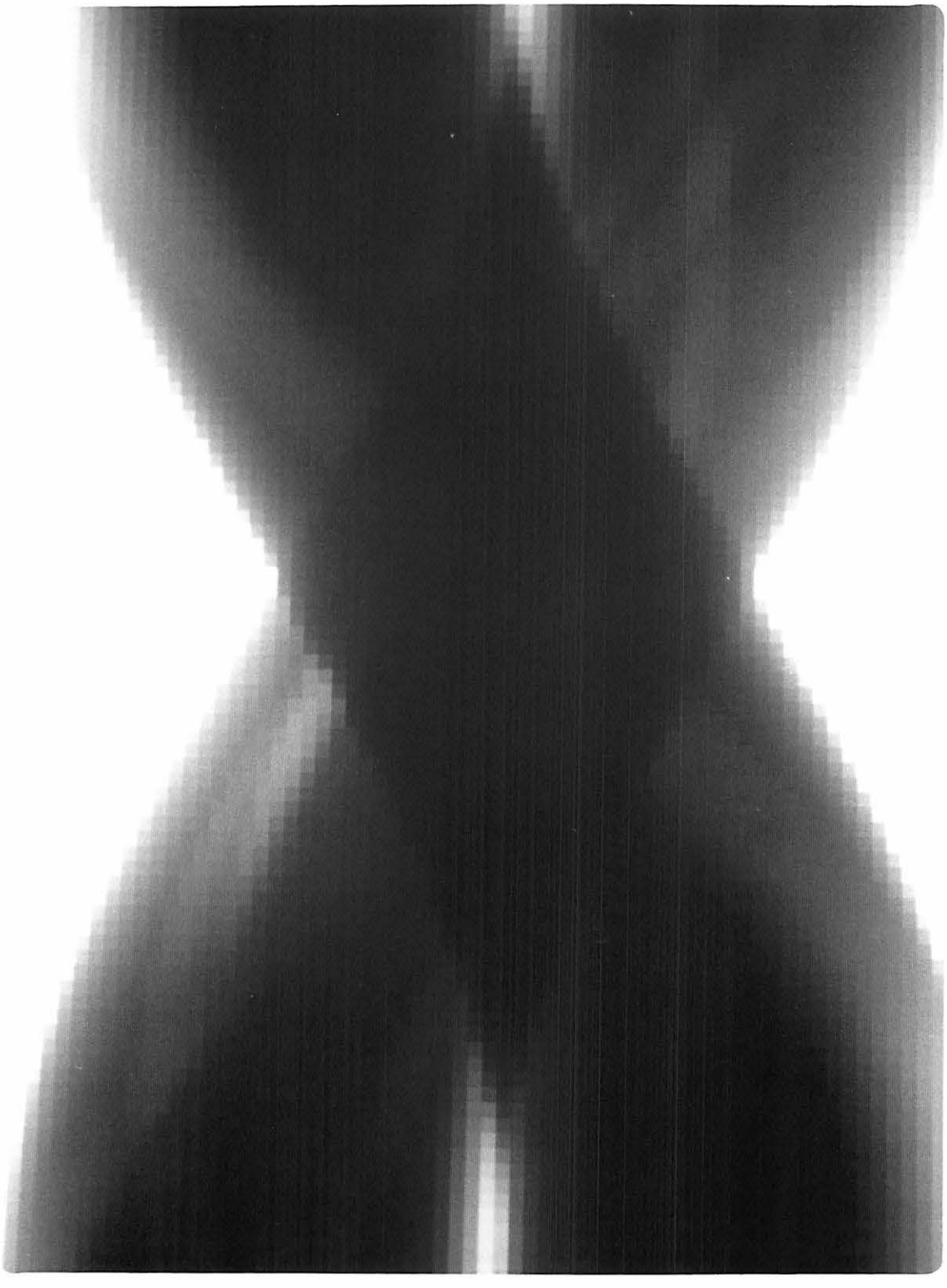
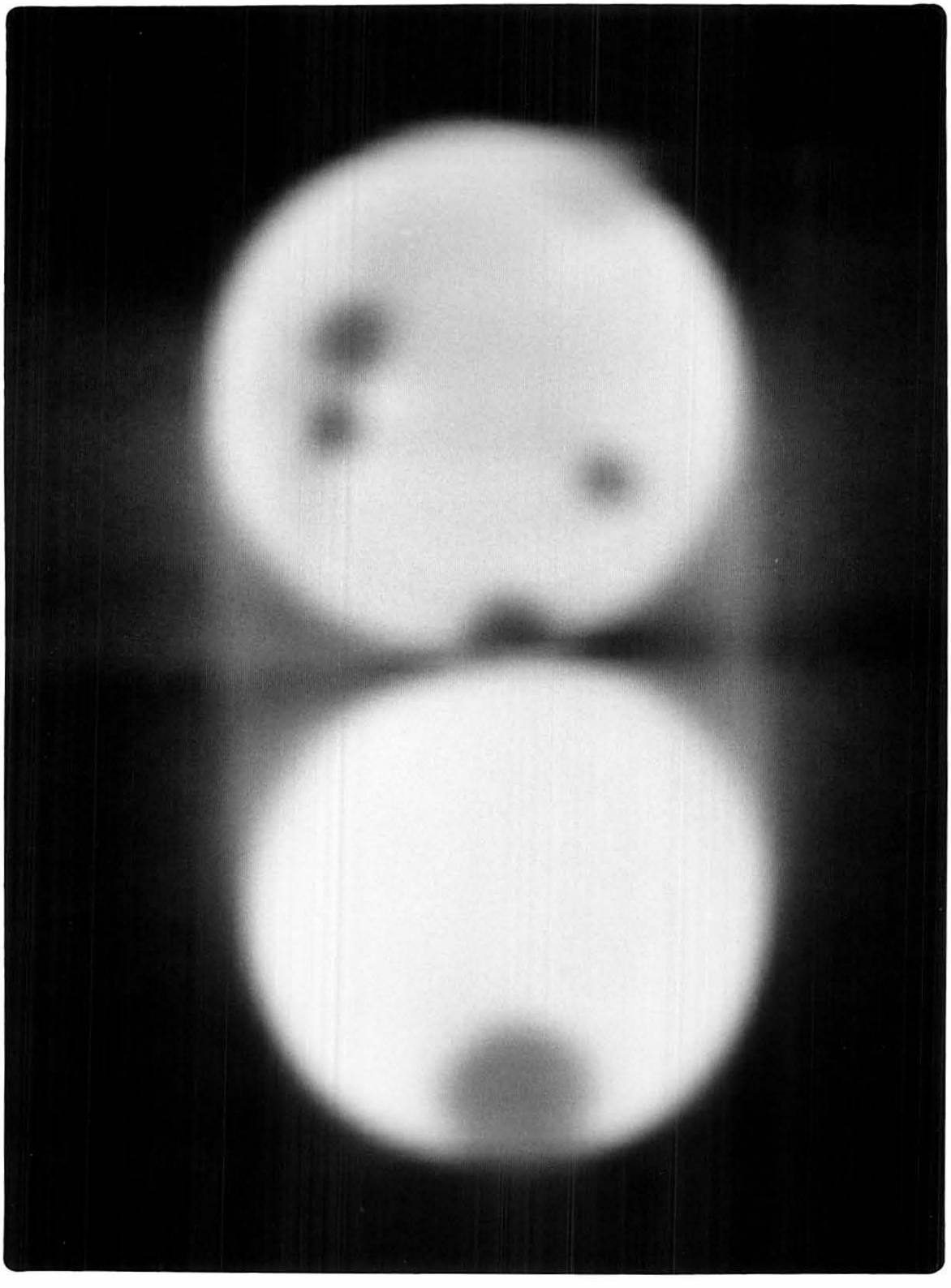


Figure 7. Reconstruction of tomo scan of two cylinders



CHAPTER 2. EQUIPMENT DESCRIPTION

This chapter will describe the equipment used in the Computed Tomography data acquisition system. Referring to Figure 8, the main components of this data acquisition system are: a personal computer, a multi-channel analyzer, a photon detector, and a sample positioner. The following sections describe each of these main components in detail.

2.1 Personal Computer

The personal computer currently used in this system is a 20 MHz 80386 based IBM compatible computer with a 320 Megabyte hard disk, 4 Megabytes of RAM, a VGA plus color video board and monitor, and a standard ISA AT compatible bus. The only requirement for the personal computer to use with this data acquisition hardware and program, is that it must be an IBM compatible PC and must have an ISA AT compatible bus addressing scheme. It must however, have at least three free full size slots in order to physically house the PC23 board (which takes one slot) and the MCA board (takes two slots).

2.2 Multi-Channel Analyzer

The multi-channel analyzer used in this system is an EG&G Ortec Model 916A MCB. A block diagram of this MCA is shown in Figure 9. The 916A MCA consists of an analog to digital converter (ADC), a Z80A microprocessor, program memory, and data memory. The converter is a successive approximation ADC with 2048 channels. A successive approximation ADC is a converter which successively 'guesses' at the digital representation of the sample and hold circuit's analog voltage

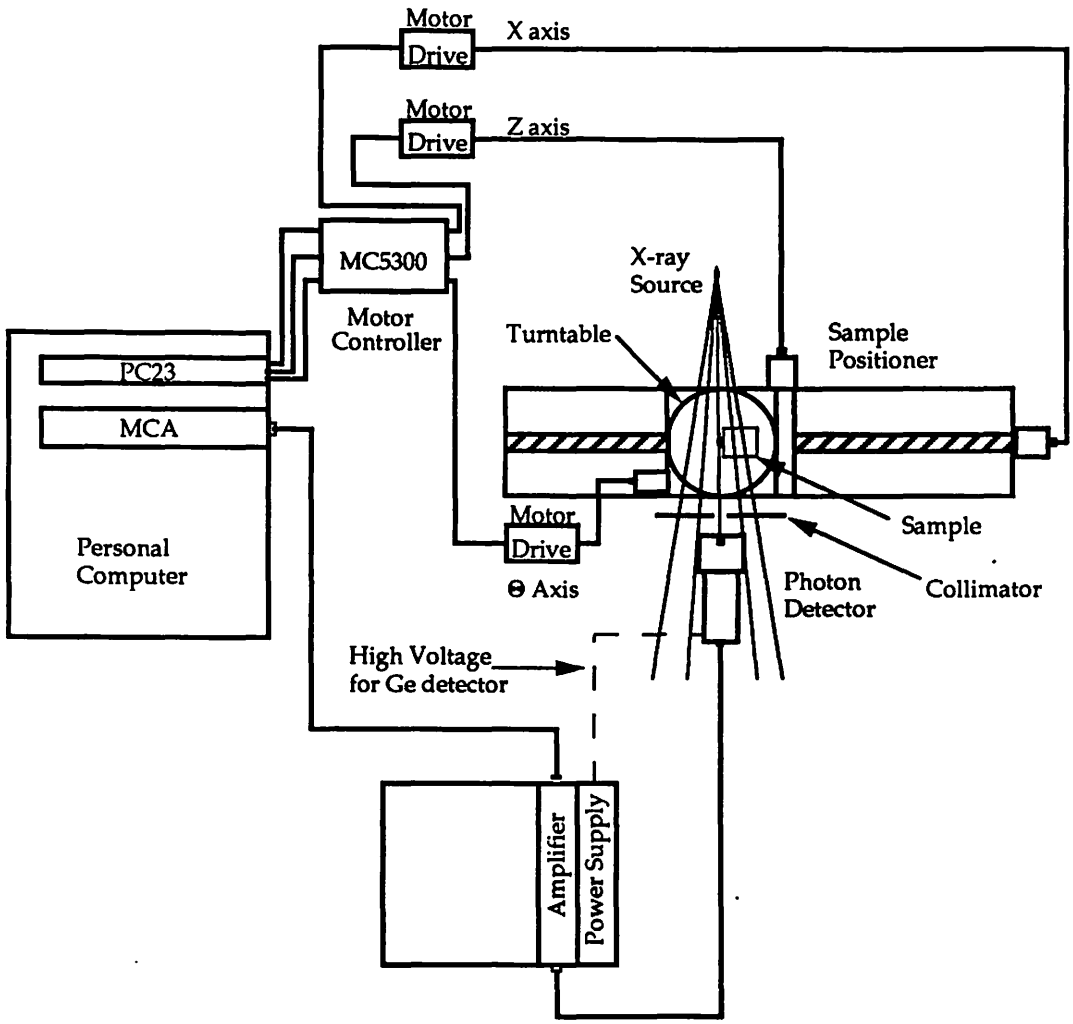


Figure 8. Data acquisition system block diagram

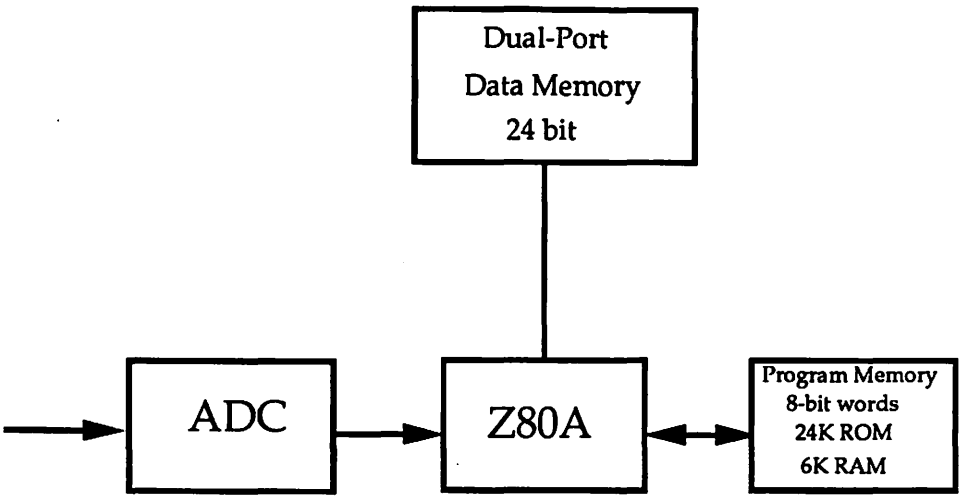


Figure 9. Block diagram of 916A MCA

until the binary representation of that voltage has been determined within a given quantization error. This technique successively moves to its best guess representation of this analog voltage by improving its guess on each approximation interval based on a comparison of the new guess to the known input voltage. There is more involved in this ADC technique which is outside the scope of this thesis. However, interested readers are referred to a digital integrated electronics book for more basic information on this ADC technique such as [3].

The program memory consists of 24K 8-bit words of ROM (read only memory) and 6K 8-bit words of RAM (random access memory). The dual-ported data memory consists of 2048 channels each of which can hold a maximum of $2^{23} - 1$ counts. This dual-ported memory allows the entire spectrum to be transferred from the card memory to the PC memory in milliseconds, thereby allowing real time snapshots of the data to be taken and processed. The specifications for this MCA board are as follows:

PULSE HEIGHT ANALYSIS Successive approximation ADC with 2K channel resolution.
DEAD TIME PER EVENT 15 microseconds, fixed.
MEMORY 2048-channel memory. Conversion gain, software selectable as 512, 1024, or 2048 channels.
INTEGRAL NON-LINEARITY < + or - 0.05% over 99% dynamic range.
DIFFERENTIAL NON-LINEARITY < + or - 1% over top 99% dynamic range.
INSTABILITY Gain < 50 ppm/degree C.
MAXIMUM COUNTS PER CHANNEL $2^{23} - 1$ (23 bit data values).
REGION OF INTEREST (ROI) One flag bit per channel.
PRESETS
- Real Time/Live Time In multiples of 20 ms.
- Region of Interest Peak count.
- Region of Interest Integral count.

Data Overflow Terminates acquisition when data in any channel exceed -1 counts.

ENVIRONMENTAL REQUIREMENTS

Temperature - 15 degrees Centigrade to 30 degrees Centigrade.

Relative Humidity - 20% to 80%.

A programmer's model for this MCA data acquisition board will next be described. The default location of the dual port memory in the IBM PC bus address scheme is D0000h (page D of the PC's memory). This address may be used to set up a pointer to the first data memory location on this board. The output I/O port address is selectable using a jumper on the board and is set to 292h for the default address. This port address is used by the programmer to select one of 8 possible MCA boards (numbered 1-8). In order to select MCA number 1, you must write a zero (MCA# - 1) to this I/O port address. This extra port address is needed to select any of the possible 8 MCA boards that may be in the same PC (the dual port memory address would remain the same for each board, so only the port address is used to uniquely identify each board). Our setup has two MCA boards available, though at present only one is used.

The way the 916A was designed, the count data are contained in the lower three bytes of each double word beginning at the data memory starting address (default = D0000h). Bits 0 -22 are the count data and bit 23 is the region of interest (ROI) bit. This bit can be used to set up a region within the spectrum that you are interested in, ignoring the channels outside the ROI. The high byte (bits 24 -32) contents are undefined. Refer to Figure 10 for a memory map of the dual port memory and to Figure 11 for an overview of the PC's memory map and a description showing the physical location of the MCA pointer and I/O port addresses.

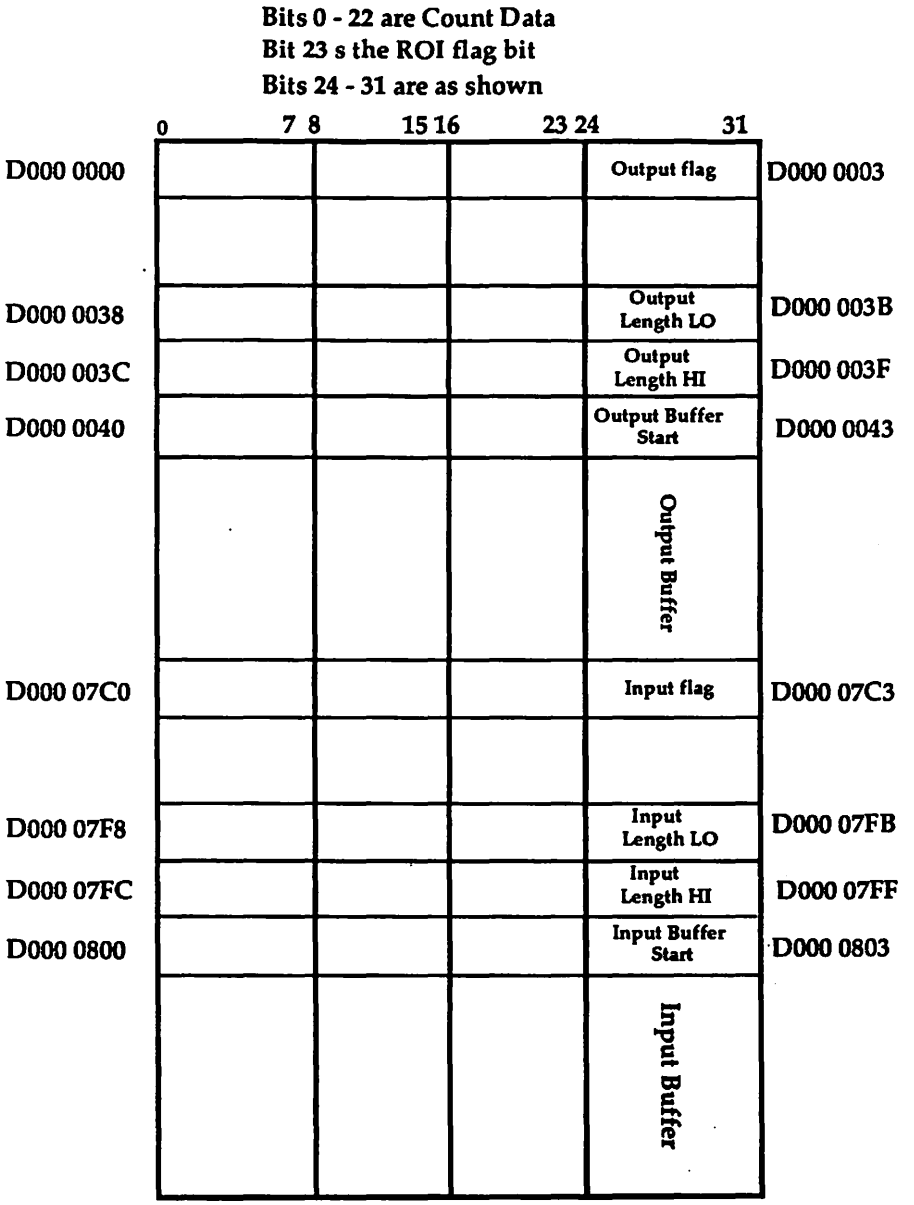


Figure 10. MCA memory map

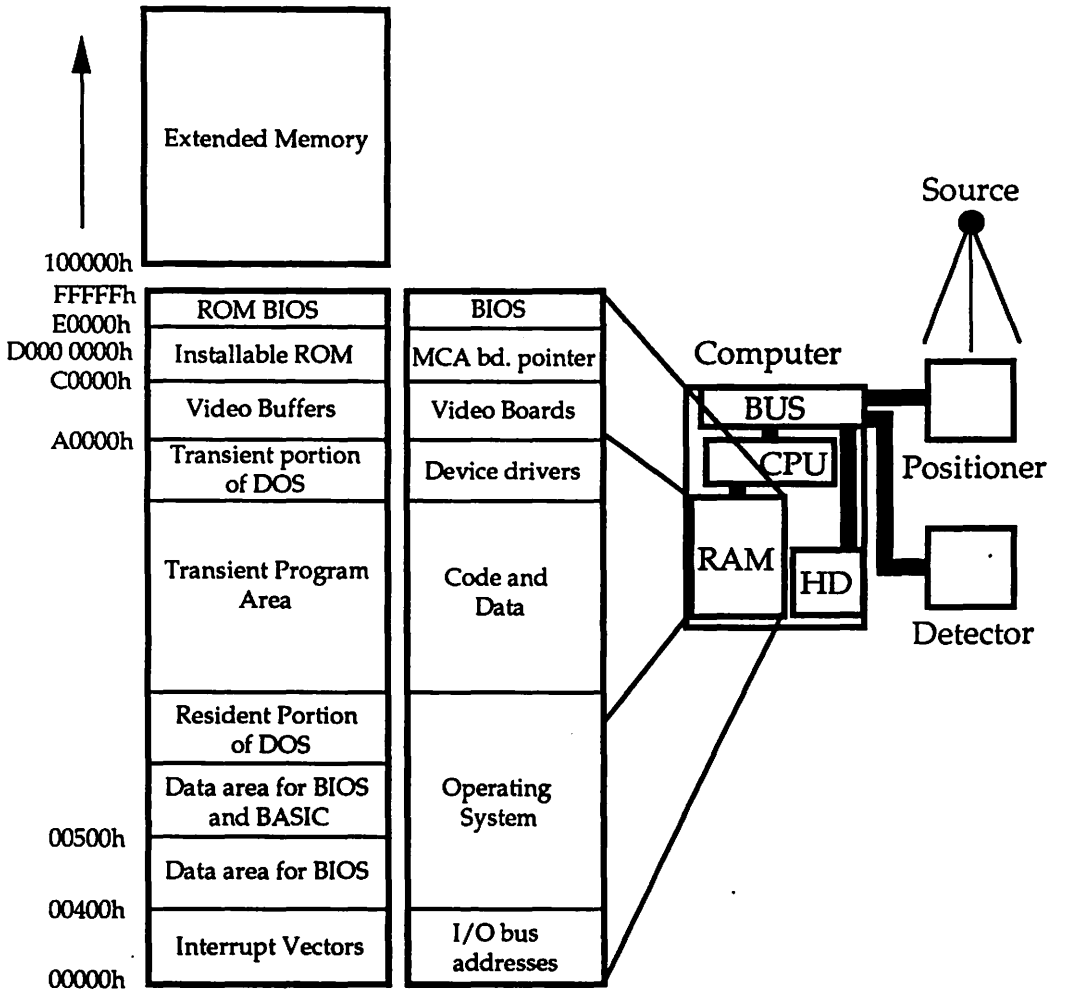


Figure 11. Overview of program interface including PC memory map, shows the location of the MCA base pointer = D000 0000h and the I/O bus address locations for referencing the I/O ports

Communications between the IBM PC and the MCA board uses a message passing mailbox scheme. The addresses used for the MCA to IBM PC mailbox are:

0803h for the message.

07FBh for the length.

07C3h for the flag.

The mailbox operation is as follows. A message to the MCA is placed at address 043h, page D (address D000 0043h), and at every fourth byte until the message has been completed (including the <CR> or <LF>). The length of the message is placed at 03Bh, page D, and a FFh byte is written to 03, page D signalling the MCA that a message is ready. When the MCA accepts this message, the location 03h, page D is set to 00h signalling to the PC that the message has been accepted. In order to read a reply from the MCA, read location 07C3h, page D, until a value other than zero appears; then read location 07FBh for the length, in bytes, of the message. Read the message beginning in location 0803h, page D, and continuing for every fourth byte for the specified length. After the message has been received, the flag at 07C3h, page D, must be set to zero. A list of the commands that the MCA board understands and responds to as well as their syntax can be found in Appendix A.

2.3 Amplifier

The amplifier used in this system is an EG&G Ortec Model 671 high-performance energy spectroscopy amplifier. This amplifier accepts bipolar signals from a detector preamplifier and provides a positive 0 to 10 volt output signal suitable for use with single-channel or multi-channel pulse height analyzers. The

gain is continuously variable from 2.5 to 1500. There is no capability for computer control of the amplifier so I will briefly outline it's specifications, but not go into additional detail on this component. The 671 amplifier has the following specifications:

GAIN RANGE Continuously adjustable form 2.5 to 1500.
INTEGRAL NON-LINEARITY (UNI output) < + or - 0.025% from 0 to +10V.
NOISE Equivalent input noise < 5.0 microvolts rms for gains < 100, and < 4.5 microvolts for gains < 100.
TEMPERATURE COEFFICIENTS (0 to 50 degrees C):
Unipolar Output < + or - 0.005% per degree C for gain, and 7.5 microvolts per degree C for the DC level.
Bipolar Output < + or - 0.0075/degree C for gain, and < + or - 30 microvolts per degree C for the DC level.
OVERLOAD RECOVERY Unipolar and bipolar outputs recover to within 2% of the rated output from a x1000 overload in 2.5 non-overloaded pulse widths using maximum gain.

2.4 Detectors

There are two different type of detectors used in the computed tomography work here at the Center. The first is a NaI(Tl) scintillation detector and the second is a germanium semiconductor detector. Both of these detectors are collimated with lead shielding so that only a 0.6mm diameter pinhole is exposed to the detector itself. These therefore fall into the category of point detectors (i.e. they only detect photons that are excited by the X-ray beam striking the detector at a single point). A close up diagram of the detector is included in Figure 12. In this Figure it shown that these detectors are composed of a detector element and a preamplifier (the Ge detector also has the high voltage filter within this detector system as shown in the diagram).

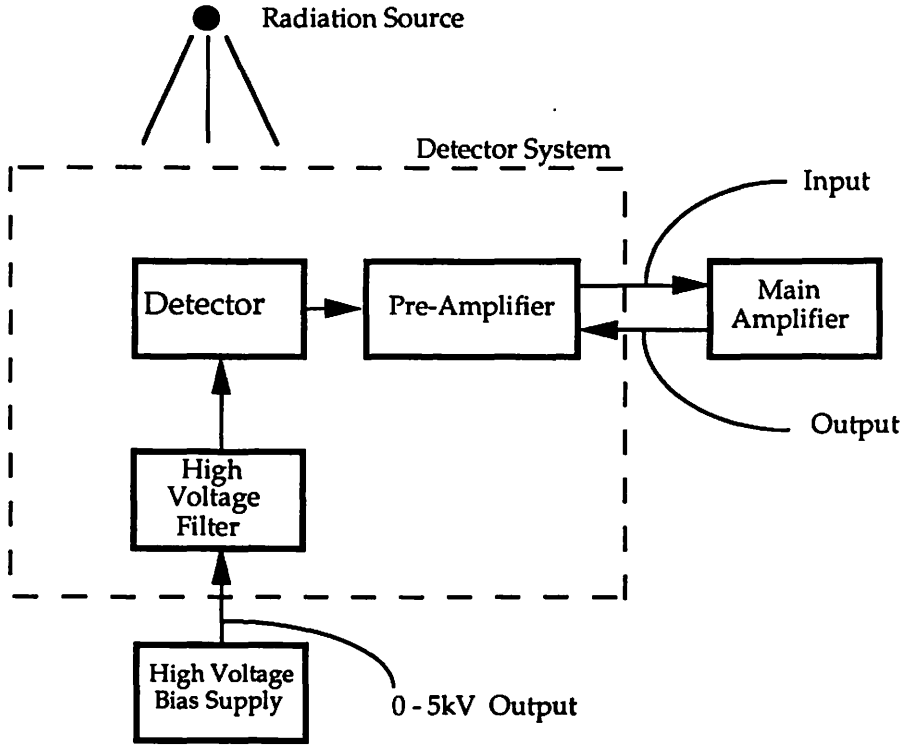


Figure 12. Detector System (Ge detector)

The Ge semiconductor element is composed of a single germanium or silicon crystal that has been made into a diode capable of withstanding high reverse bias voltage at cryogenic temperatures. Under these temperature conditions, electron-hole pairs produced by the absorption of an x-ray or gamma-ray photon are swept to opposite contacts by an electric field. The resulting induced current pulse is integrated by a charge-sensitive preamplifier producing an output voltage pulse with height proportional to the incident photon energy [4]. It is this preamplifier voltage pulse that is hooked to the input of the 671 amplifier when the Ge detector is in use.

The scintillator detector used here at the Center is a NaI crystal based detector. The NaI detector makes use of the ability of a large group of crystal lattice structures to respond to a radiation source by emitting scintillation light. These NaI crystals when laced with a small amount of impurity (thallium iodide), produce an exceptionally large scintillation light output when exposed to a radiation beam. This scintillation light must be then converted to an electrical signal which may then be amplified and counted. Photomultiplier tubes are used to convert the relatively weak light output of a scintillation pulse into a usable current pulse without adding large amount of random noise to the signal. The two major elements to a photomultiplier are a photosensitive layer, called the photocathode which is coupled to an electron multiplier structure. The photocathode converts as many of the incident light photons as possible into low-energy electrons. The photoelectrons produced will be a pulse of similar time duration to the original scintillation light pulse. However, because only a few hundred photoelectrons may be involved in a typical pulse their charge is too small at this point to serve as a convenient electrical

signal. It is the electron multiplier section in the photon multiplier tube that serves as an electron amplifier to greatly increase the number of photoelectrons. After amplification through this multiplication section, a typical scintillation pulse will give rise to 10^7 - 10^{10} electrons which is sufficient to serve as the charge signal representing the original scintillation event.

These two detector types have different sensitivities in that the Ge detector is considerably more energy sensitive than is the NaI detector. Therefore, the Ge detector is able to resolve distinct energies better than the NaI. This fact is easily demonstrated by looking at the response of the two detectors to an isotope source that emits mono-energetic photons of known energies. Figure 13 is a graph of the Ge and NaI detector responses to an Am^{241} isotope. Referring to this Figure it is apparent that the Ge detector can distinguish four distinct energies in this energy window. By comparison, it is not at all apparent that there are four distinct energies present when looking at the NaI detector response.

The trade off when using the Ge detector for better sensitivity, is that it is less efficient and costs considerably more. The Ge detector must also be kept at liquid nitrogen temperatures when in use. Therefore, it is more costly and troublesome to use as well. If the user is not concerned about doing energy sensitive scans, then it is best to use the NaI detector, whereas if energy sensitivity is important, then the Ge detector must be used.

2.5 PC23 Indexer Board

The Compumotor PC23 is a microprocessor based indexer that is designed and built as an add in board for the IBM personal computer ISA AT bus. This board resides in the personal computer and along with the MC5300 motor controller sends pulses to the motor drivers to control the stepper motors in the sample positioning

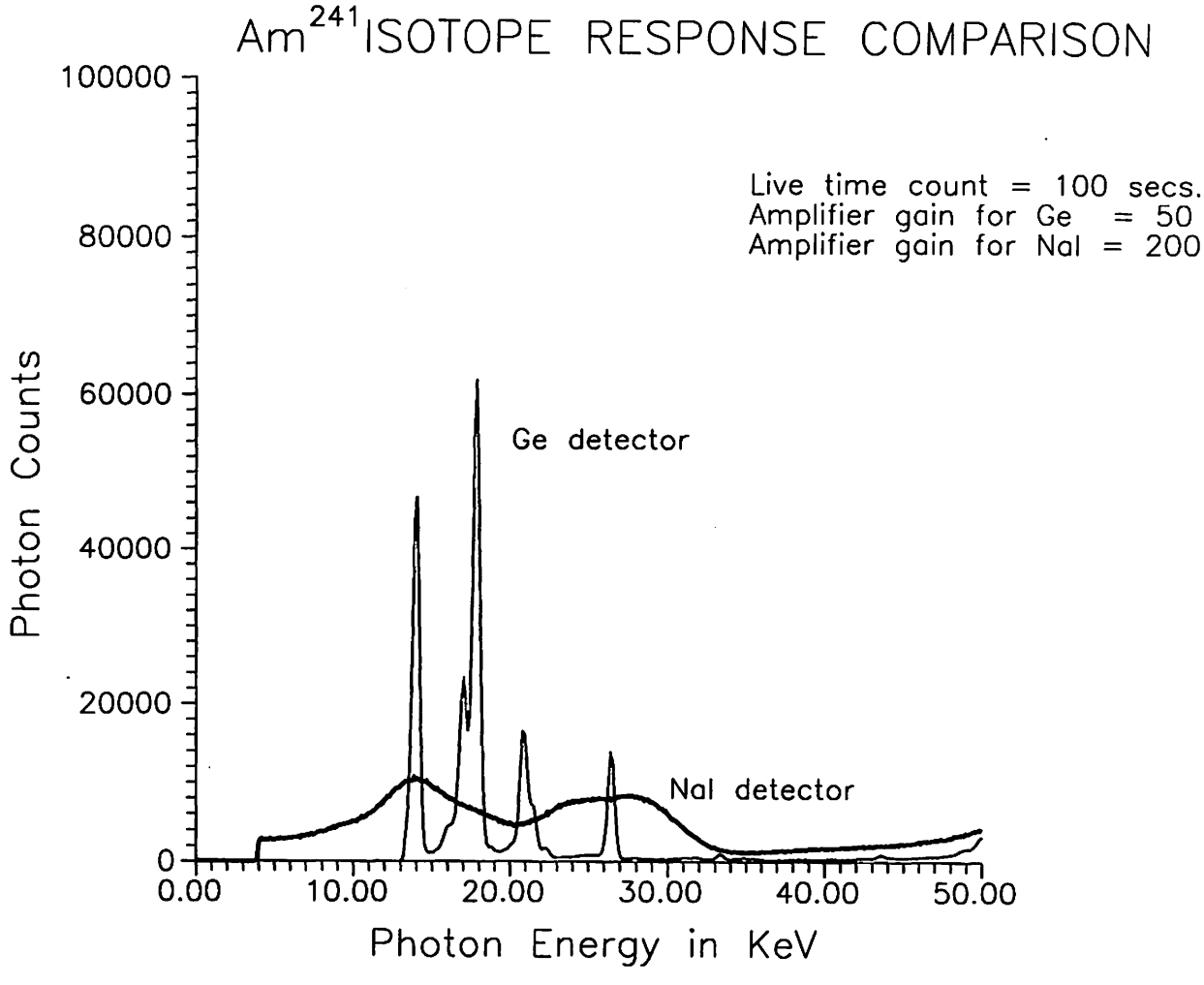


Figure 13. Am 241 isotope response of detectors

system in precise positioning movements (refer to Figure 8). The board is another I/O addressable port as far as the PC is concerned. A structure can be set up to point to the various control and status registers that reside on the board. Once this has been done, it is possible to send motor commands as sequences of ASCII characters to the board and to monitor status signals in order to know what the positioner is doing or has done. A more detailed explanation of the PC23 will now be presented.

Operation of the PC23 is independent of the programming language used to control it. The programmer only need have the means to read from and write to the I/O bus of the computer. This ability to read and write from and to the I/O bus is available in most languages in some form of the Intel assembly language IN and OUT statements. These instructions allow the programmer to send a byte to an I/O bus address (IN) and to receive a byte from an I/O bus address (OUT). The actual name of the functions used to accomplish this communication is language and compiler dependent (example: for Turbo C++, the I/O port communication function names are inportb and outportb).

In order for the computer to control the PC23, it must know where to write instructions and to read responses. This requires that the PC23 have an I/O bus address that the PC can access. The PC23 occupies four addresses on the I/O bus. The base address is selected on the board itself by setting switches to the appropriate positions. The reader is referred to pages 7 and 8 of [5] for more information on setting this address.

Only two of these four address are significant: one for CONTROL and one for DATA, Input and Output operations each use the same address (the other two addresses are duplicates of these two). The DATA address is equal to the base

address of the board set by the switches on the board (default = 300h). The CONTROL address is equal to the base address + 1 (default = 301h).

The PC23 control portion of the program reads from and writes to registers at the base address of the board plus one (default = 301h). This transfer takes place one character at a time. During each character transfer, the program writes control bytes and reads status bytes from registers at this address.

The Control Byte (CB) and Status Byte (SB) provide access to the PC23 operating conditions. The CB allows setting certain operation conditions and the SB reports others. Each refers to an eight bit PC23 register that is accessible on the PC I/O bus, where each of the eight bits is a flag with a specific meaning. Control Byte flags allow the host program to signal the PC23 with messages. The Status Byte flags allow the program to check on the PC23 status such as checking to see if motor X is moving.

Signalling the PC23 involves setting (forcing to binary one) or clearing (forcing to binary zero) control bits or flags. Clearing and resetting a bit means to force it to a binary zero. Setting a bit means to force it to a binary one. All eight Control register bits are forced to one state or the other when the computer writes to the registers address.

Control Byte - Table 2-1 shows the Control Byte flags available to the programmer for signalling the PC23. Immediately following the Table is a more detailed explanation of each of the 8 bits.

Status Byte - The Status Byte (SB) provides information about the status of the PC23 indexer board. The information flags are there meanings are shown in

Table 2-1. PC23 control byte format

Bit	Definition
0	Binary Input, active high (TD mode only)
1	Unused
2	Stop Watchdog Timer (active high pulse)
3	Acknowledge Interrupt (active high pulse)
4	IDB Command Character Ready (active high handshake)
5	Restart Watchdog Timer (active low pulse)
6	Reset Interrupt Output (active low pulse)
7	ODB Message Character Accepted (active high handshake)

Bit 0, when set indicates that the Binary Mode of data input for the TD mode of contouring is under way.
Bit 2, when set, causes the PC23's watchdog timer to time out and stop. When the timer stops, it forces a hardware reset. The reset condition may be cleared by cycling power or restarting the timer. See bit 5 for more info.
Bit 3, when set, tells the PC23 that its interrupt signal to the computer has been noted and is no longer needed. See bit 6 for more info.
Bit 4, when set, tells the indexer that a command character has been put into the Input Data Buffer (IDB). The indexer then clears bit 4 of the SB to indicate that the IDB is unavailable, reads the character in the IDB, and then sets bit 4 of the SB to indicate to the host that the IDB is again ready for a new character.
Bit 5 restarts the watchdog timer. It must first be cleared, then the timer will start up when the bit is set again. This bit should never be toggled unless the timer has timed out.
Bit 6 resets the hardware interrupt latch and thus the interrupt output. The interrupt output cannot be reset unless the interrupt is first acknowledged with bit 3 above. These bits should be cleared during reset or interrupt acknowledge.
Bit 7, when set, tells the indexer that a response character previously placed in the Output Data Buffer (ODB) by the indexer has been received by the host. A new character may then be placed in the ODB.

Table 2-2. The status byte performs two functions; to assist in the communications process, and to provide run-time status information without the need to burden the indexer with routine status request commands.

Reading and Writing Motion control commands and responses are transferred via the Input Data Buffer (IDB) and the Output Data Buffer (ODB) at the PC23 base

Table 2-2. PC23 status byte format

Bit	Definition	Power-Up State
0	Axis 1 Stopped.	Set
1	Axis 2 Stopped.	Set
2	Axis 3 Stopped.	Set
3	ODB Ready.	Cleared
4	IDB Ready.	Set
5	Board Fail.	Cleared
6	Interrupt Active.	Cleared
7	(Reserved)	Cleared

Bits 0, 1, and 2 indicate whether the motors for the three axes are moving. At the beginning of any move, the appropriate bit is cleared. Specifically, these bits indicate whether or not the indexer is sending step pulses to the motor drives.

Bits 3 and 4 are set when their corresponding data buffer is ready: Bit 3 is set when the Output Data Buffer (ODB) contains an output character for the host, signalling the host to read the information it contains; Bit 4 is set when the Input Data Buffer (IDB) is ready, telling the computer it may write a character to the IDB.

Bit 5, when set, tells the PC that the watchdog timer of the PC23 has timed out, possibly indicating an internal failure from which it cannot recover. The only way to clear this bit is to reset the indexer. Exercising the self-test function will also set this bit. In this condition, the motor shutdown output will go on, removing motor torque, and generating an apparent drive fault.

Bit 6 indicates to the host that a conditional interrupt has been armed and that the condition has occurred. If either jumper JU1 or JU2 is installed, then the PC23 has generated a hardware interrupt signal. For more information see the interrupt section in [5].

address (default = 300h). Interface control commands and status information are transferred via the control byte (CB) and the status byte (SB) at address = base address + 1 byte (i.e. if base address = 300h, then the CB and SB address = 301h). The ODB and the SB are read-only registers and the IDB and CB are write-only registers.

Commands sent to the Indexer are strings or sequences of ASCII characters. A list of the commands used to position samples is included in Appendix B. This list is by no means comprehensive, it only contains the commands are presently used in the program. The reader is referred to [5] for a comprehensive list of available commands. Sending commands to the PC23 involves transferring each character in the command, one character at a time. Each character transferred requires that the sender notify the receiver that a character is ready, and the receiver notify the sender that the character has been received. This notification process makes use of single bit flags in the SB and CB registers that are set high (1) or low (0) to denote the ready or busy condition (bits 3 and 4 of the SB and bit 4 of the CB). There are step by step procedures that must be followed in order to program the PC23 board. Every program that controls and monitors the PC23 board must contain routines to do the following:

1. Reset the indexer.
2. Send a command string to the indexer.
3. Receive a character string from the indexer.

The step by step procedures to be followed in order to implement each of the above operations will next be detailed. Note: all anding operations are bitwise logical ands.

Resetting the PC23

1. Write 64h to the Control Port (Base address + 1).
2. Read the Status Port (Base address + 1) until the status byte anded with 20h, is greater than 0.
3. Write 40h to the Control Byte (Base address + 1).
4. Write 60h to the Control Byte (Base address +1).
5. Read the Status Port (Base address + 1) until the status byte anded with 7Fh equals 17h.
6. Write 20h to the Control Port (Base address + 1).
7. Write 60h to the Control Port (Base address + 1).

Reading a Character from the PC23

1. Initialize the ASCII variable to null (0).
2. Read the Status Port (Base address + 1) until the status byte anded with 08h is greater than 0.
3. Read the Data Port (Base address) into the ASCII variable.
4. Write E0h to the Control Port (Base address + 1).
5. Read the Status Port (Base address + 1) until the status byte anded with 08h equals 0.
6. Write 60h to the Control Port (Base address + 1).

Writing a Character to the PC23

1. Convert the character to ASCII (not necessary when programming in C except for binary input mode for axes in the TD mode).
2. Read the Status Port (Base address + 1) until the status byte anded with 10h is greater than 0.
3. Write the ASCII character to the Data Port (Base address + 1).
4. Write 70h to the Control Port (Base address + 1).
5. Read the Status Port (Base address + 1) until the status byte anded with 10h equals 0.
6. Write 60h to the Control Port (Base address + 1).

The control program written for this project uses five functions to implement the above operations. They are called Initialize, ReadChar, ReadAnswer, WriteChar, and WriteCmd. These functions as well as others in the control module are detailed in following chapter.

2.6 Motor Drives

There is a motor drive for each of the three main axes described below:

X_Axis - This axis is the horizontal linear axis for moving the sample left and right through the pencil beam of radiation.

Z_axis - This is the vertical linear axis for moving the sample up and down within the pencil beam of radiation.

Θ_axis - This axis rotates the sample through the radiation beam with the axis of rotation being vertical (parallel to the Z axis).

It is a combination of the X and Q axes that allow us to obtain the full data set for a tomography scan. The type of tomography scan done here at the Center is done with what is called a first generation CT scan configuration, the medical industry is now up to a fifth generation system that is much faster and considerably more costly. The procedure presently used for obtaining a tomographic data set with the point detector is as follows. Scan horizontally (on the X axis) for a specified total X distance, moving the specified X axis increment amount each time and counting photons at each point for a given time (in seconds). Then the sample is rotated one angular increment on the Θ axis and the linear scan direction is reversed and continued. This procedure is repeated until all angular projections have been completed. This method yields a large two dimensional array of data points each containing a photon count representing the X-ray attenuation for a given sample

point. These sample points can then be reconstructed using one of the computed tomography (CT) reconstructions algorithms on the computer which yields a two dimensional view of a cross sectional slice through the sample.

There are two other types of scans used at the Center with this equipment, they are a one-dimensional line scan (1D scan), and a two-dimensional digital radiography scan (2D scan). The 1D scan may involve any of the linear or rotational axes, while the 2D scan is hard coded to only use the X and Z axes.

CHAPTER 3. PROGRAM DESCRIPTION

This chapter will describe the program purpose, structure, and the main functions within the program modules. A flow diagram of this program is presented in Figure 14 and can be used as a reference throughout the description. The functions described will be those that actually do the three main types of data acquisitions and those that are needed to communicate with both the MCA data acquisition board and PC23 Indexer board.

3.1 Program Purpose

This program is designed to do three main types of data acquisition: a one dimensional scan, a two dimensional scan, and a tomographic scan. In each of these types of scans the purpose is to move the sample positioner along the specified axis (or axes) stopping at each delta increment on each axis to collect photon counts for a specified amount time at each point. The sample positioner moves the sample an additional delta increment along the axis and to the new data acquisition position. This procedure is repeated until all of the distance on the given axis or axes has been traversed and all data collected. The program saves these photon counts in an output file in a grd file format to be used by either the Surfer or Grapher software packages. This file can also be used as input for the tomographic reconstruction program.

An option available on any of these types of scans is to collect energy sensitive data in multiple energy bins at each point, as well as obtaining the total photon counts(i.e. the sum of all energy bins). This option when used prompts the user for the number of energy bins and for each energy bin upper and lower values

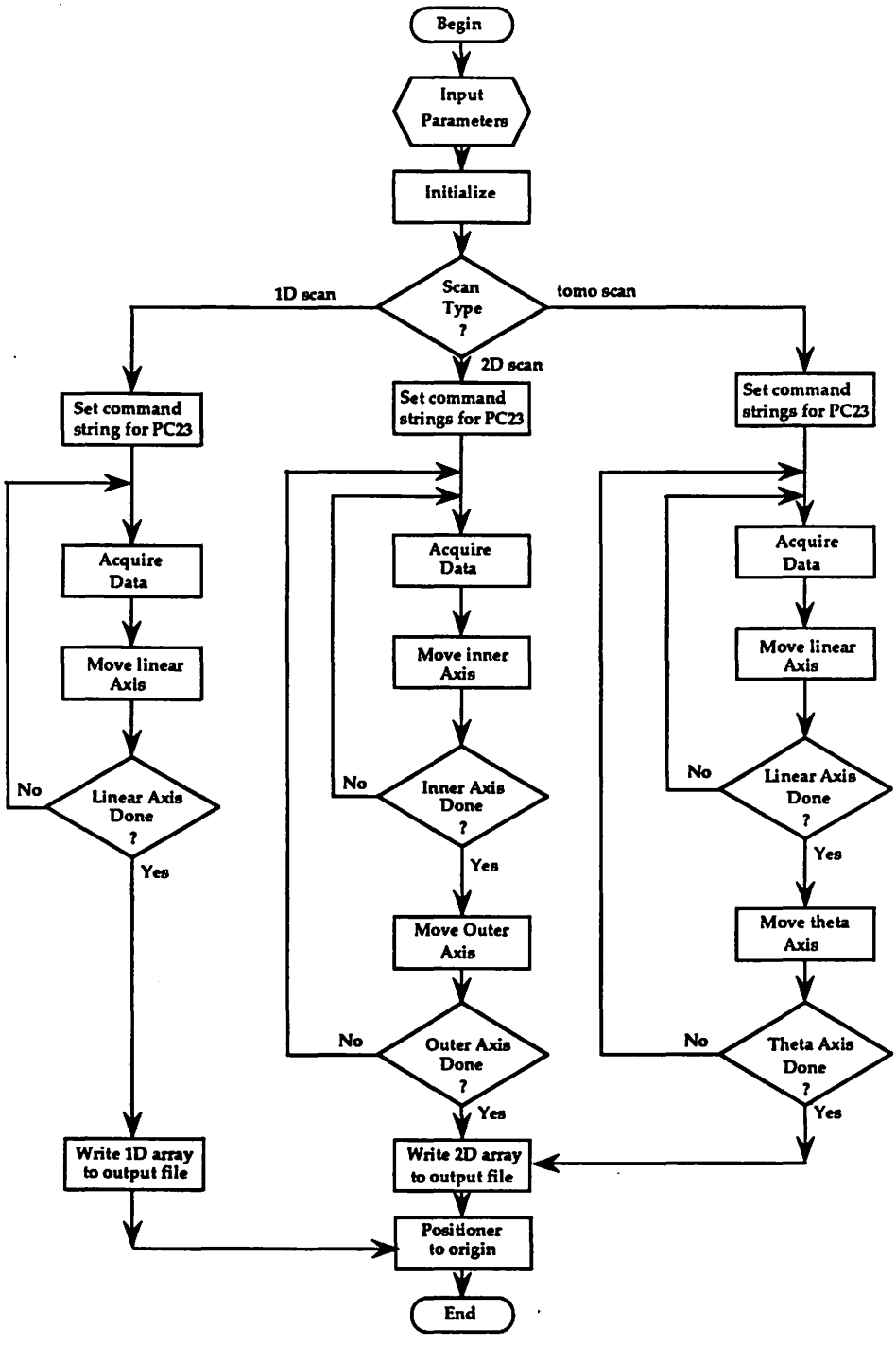


Figure 14. Flowchart of data acquisition program

for each bin. The output for this option is as follows: The total photon count for all bins at each acquisition point is contained in a grd file named by the user. The energy bin data is then stored in a separate file for each energy bin specified by the user in the grd format (see Appendix C for grd ASCII file formats) with following naming convention: The bin files are named using the given output file name without its extension, but using a new extension with the following format: '.en#', where # = 0 to (total number of bins - 1). For example, if the user specified an output file name of 'ceramic.grd' and chose energy bin collection with 4 bins specified, then the sum of all energy bins at each point would reside in output file ceramic.grd and the energy bin data for each bin would be in files: ceramic.en0, ceramic.en1, ..., ceramic.en3 for each bin number 0 to 3. A detailed description of each program module that accomplishes these scan types is now presented.

3.2 Acqray.c module

The acqray.c module contains all of the functions that actually implement the three main scan types. This module also contains an initialization function, a dynamic memory allocation function, functions to write the output data to files, find the minimum and maximum values for the photon counts, test output file name for DOS legality, and a function to handle the multiple energy bin case. A detailed description of these functions within this module follows.

Function InitEverything(void) - This function initializes both the PC23 Indexer board and the MCA data acquisition board so all control registers are cleared and readied to accept command strings. This involves calling the Initialize() function in the moverf.c module to initialize the PC23 board and sending the ASCII string 'Initialize' to the MCA board (see Appendix A for the description of the MCA

commands). InitEverything() also initializes all flags and MCA mailbox pointers used in this module. The flags that are initialized by this module are described in Table 3-1.

The mailbox pointers and their use within the program are described below:

*mcb_outflg => Set TRUE when a command is ready.
*mcb_test => Written to and then read to see if mailbox exists.
*mcb_outlenlo => Low byte of the length of the command string.
*mcb_outlenhi => High byte of the length of the command string.
*mcb_outbuf => Buffer in the 916 MCA for commands.
*mcb_inflg => Set TRUE by 916 MCA when it's response is ready.

Table 3-1. Initial flags and their values

Flag Name	Description of flag	Initialized to:
rot_flg	When TRUE says use the rotational axis.	FALSE
Zflag	When TRUE says use the Z axis, else use X.	FALSE
Live_Time_flag	When TRUE means count photons using the Live time preset within the MCA board.	TRUE
First_Acq	When TRUE says that this acq. point is the first one.	FALSE
redo_flag	When TRUE says to redo the first acquisition.	FALSE
mult_fact	Multiplicative factor to allow us to adjust the photon count off the sample when counting in real time instead of live time.	1

*mcb_inlenlo => Low byte of the length of the response string.
*mcb_inlenhi => High byte of the length of the response string.
*mcb_inbuf => Buffer in the 916 MCA for it's response.

This function needs no input parameters and returns nothing.

Function one_D_scan(char) - This function implements the one dimensional scan operation. The axis chosen is selected by the user in the user interface section and then passed to this function. Any of the following axes may be used: Theta, X, and Z. Within this function we first check to see what axis is used, set the appropriate flag for this axis, and then determine the number of moves on this axis. The number of moves is calculated by obtaining the difference between the ending axis value and the beginning axis value and dividing it by the increment to use on this axis. For example, if we use the X axis and the X start value = 0.0, the X end value = 1.0, and the increment for the X axis is 0.01, then the number of moves = (1 - 0)/0.1 = 10. At this point the axis direction is determined and stored in its direction variable. All axis directions are determined by the arithmetic sign of the difference between the end axis value and the start axis value. If this difference is positive then we move in the positive direction for this axis, if it is negative then we move in the negative direction. We then ensure that the output file name supplied by the user is a valid DOS file name and then create the command strings to be used by this axis. There will be two strings to use, one for the forward direction and one for the reverse direction.

We are now ready to enter the main loop (the only loop that the one axis needs). In this loop we move the sample positioner if this is not the first data acquisition point. We then call the acq_data() function to acquire the photon count data for this point. We continue on with this sequence, move the positioner, acquire

the photon count data, and repeat until the we have moved the total number of axis moves. When finished with the data acquisition (terminate from the main loop), we save off the output data to the output file and then reposition the sample to its original position.

Function tomo_scan(char char) - This function does as its name implies, the tomographic scan. Two values are passed to this function, they are the two axis numbers to use for the tomographic scan (for example, the X axis and the theta axis). This function then determines how many moves must be done on each axis used and how many motor steps for each move. Refer to the one_D_scan function description for an example how these are calculated. Before any data is actually collected, the output file name specified by the user is checked to insure that it is a legal DOS file name, if not, then the user is prompted for a legal name until one is given. Next the function creates command strings for each move to be sent through the PC23 board to its two axes. These strings contain all the motor drive information such as the axis to use, the motor resolution for this axis, the acceleration and velocity for this move, the move distance in motor steps, the axis direction to move in, and finally the go command which says execute this command string. Four command strings are created, two for each axis. The two for each axis are for both the forward and reverse axis directions. At this point, the function calls the allocate_Temp() function to allocate dynamic memory for the huge two dimensional array that will hold the total photon counts for each scan point.

At this point, we are ready to start the main loops that actually accomplish the tomographic scan. The outside loop, the rotational axis loop, controls the rotational axis of the scan and is a for loop going from i = 0 to i = num_rotational_moves. At the end of the inner loop ,called the linear loop, this

rotational loop checks to see if this is an even or an odd numbered scan. If it is an even scan then the linear loop variables are set up to scan on the linear axis in the reverse direction and to store this data in the two dimensional array in reverse form for the next linear axis pass. Otherwise, the linear scan is done in the forward direction. This is done so that the sample does not have to be moved back to its original starting position after each linear scan. The rotational axis loop then moves the rotational axis one more increment and starts the linear loop again.

Within the linear loop, the sample is moved one linear axis increment if it is not the first data acquisition point for each linear scan. This move is either in the forward or the reverse direction as determined by the even or odd rotational scan number (see above for further details). The function then calls the acq_data() function which actually sends other command strings to the MCA data acquisition board. The description for this function is included later in this chapter. This linear loop also contains some of the control statements to handle the multiple energy bins scan case.

At the end of this function we clean up by closing the file pointers for the multiple energy bins if they were opened, storing out the total photon count data, and then repositioning the sample to its original position on both axis. We also store out to an extra file called useful.dat containing the following information: the beginning and ending energy values for each energy bin and then the minimum and maximum count values for each energy bin. This then concludes the tomo_scan function and control is returned to the calling function.

Function two_D_scan(char, char) - This function implements the two dimensional scan, similar to the tomographic scan except this function can be done using any two axes (i.e. it does not require a rotational axis). Otherwise, the layout

of this function is very similar to the tomo_scan function. If one understands the structure of the tomo_scan function then this function is easy to understand. The only differences are in the axes that may be used; this effects the command string creation and sample position return. Except in the creation of the command strings to send to the PC23 board the description for the tomo_scan function can be studied. We refer the reader to Appendix D (the program listing), if any additional specific details are desired.

Function one_tomo_scan(char) - This function exists for the sole purpose of doing one extra linear scan for the tomographic scan. This scan is done by first moving the positioner on the rotational axis to an angle equal to (thetaMax - thetaMin)/2 + thetaMax, then doing the line scan and returning the sample positioner to its original position. The reason for this extra scan is to have one more projection data set to use to help determine the center of rotation of the scan. For example, if we are doing projection angles from 0 to 180 degrees, then we would do one extra linear scan at an angle of 180/2 + 180 = 270 degrees. Then this data could be compared with the data taken at 90 degrees along with the 0 degree and 180 degree data sets to enable us to have two data sets in order to better determine the center of rotation.

Function acq_data(void) - This function returns an unsigned long value representing the photon count total of the selected MCA channels. It first sends a clear command to the MCA board, then sets the live count preset to equal the value of the count time entered by the user in the user interface section of the program. This forces the MCA board once started to count for count_time seconds of actual live time, not wall clock (or real time). An explanation of this difference is now in order. Real time counting for 2 seconds would literally be counting for 2 wall clock

seconds; however, if the electronics within the detector/MCA board become saturated due to the large number of photons striking the detector, then it will miss some of these counts. The MCA board has a mechanism for pausing itself so it can catch up and not miss counts. This mechanism is used by setting the live time preset for 2 seconds and this will yield a full 2 seconds of good count time (i.e. will not miss counts due to saturation).

Whenever a command string is sent to the MCA board this function calls the mbxio() function to actually send the command string. The acq_data() function then sends a start command to start the data acquisition and sets up a command string of show_active which when sent to the MCA board will cause this board to send a response record showing the active MCA segments (i.e. those that are still counting). The function then loops until the response record returned is equal to all zeros, saying that all segments are done counting. A pointer is then set up that points to the beginning of the dual port memory in the MCA board. This pointer is then appropriately offset to sum up only the channel photon counts that are to be used (out of the total of 2K channels) If we are doing a multiple energy bin scan, then this function handles the energy bin count selection and summation for each bin. The function then returns either the total photon count, if not doing a multiple energy bin scan, or the total of the last energy bin selected. In the multiple energy bin scan cases, the global array called bin_data[][] is filled with the various energy bin counts by this function.

Function mbxio(char *, char *, char*) - This function handles all of the direct communication to the MCA board and then evaluates and returns all of the responses from this board as well. Refer to the mailbox communication explanation contained in the Chapter 2 MCA section for details as to how the mailbox hardware

is set up for this communication. The parameters are as follows: the command string to be sent to the MCA called command, the response string called response, and then the percent response string called per_response. For more information on the MCA board commands and their response records see Appendix A.

We first initialize both of the message flags and set a start time equal to the current internal PC time and loop until the we either see the appropriate response from the MCA board or have had five seconds elapsed. If the five seconds does elapse before we get the expected response (output flag = 0FFh), then this function returns a error flag = -1. If we do get the appropriate response then we copy the command sent to this function to the output buffer within the MCA, write out the length of the output message, and then set the out flag to say the command is ready. At this point, we call the get_resp() function and copy its return value into the first response string. We check this response to see if it was a percent response i.e., the first character is a percent sign, '%'. If it was and the command sent should cause a response record to be sent back first, then we have an error condition returned from the board. If it is not a percent response, then we copy the per_response string into the response string and go get the next response which should be the percent response returning from the MCA board showing no error condition. This function then returns the response and percent response to the calling function as well as an integer representing whether or not an error was detected.

Function get_resp(void) - This function polls the MCA board for the input flag to be set, if this occurs within 5 seconds, then we go on to obtain the message length from the MCA board, else we return an error condition saying the MCA board is not responding. Assuming, we get the input flag as expected, we then obtain the message length and copy the message from the MCA input buffer to the

resp_buf string within this function. We then reset the input flag and return a pointer to the response string.

Function allocate_Temp(void) - This function attempts to allocate enough dynamic storage to hold all of the output data expected for the scan. It is expected that due to other limitations the maximum scan grid size will be 256 x 256. This size would require 256 x 256 x 4 bytes/element = 256Kbytes of storage, so if in the future this limitation no longer exists, it would not be hard to bump up against the DOS limit of 640K for both data and program. Note: to use the same amount of space with one of the 1D scans you could specify 64K data points on the linear scan. For now this function should not have a problem allocating memory unless other programs or structures have not terminated and still have allocated memory. However, if this does occur and there is not enough memory to allocate for the whole scan the program terminates immediately and does not acquire and then lose data.

Function free_Mem(void) - This function frees up the dynamic storage allocated by the allocate_Temp function.

Function write_2D_to file(void) - This function is called by all two dimensional type scans including the tomo_scan, it then opens the output file for write and writes out the output data from the huge data array called DataTemp to this file.

Function write_1D_to file(void) - This function is called by the one dimensional scans, it then opens the output file for write and writes out the output data from the huge data array called DataTemp to this file.

Function find_limits(int, int, int) - This function searches the huge DataTemp array for the minimum and maximum values and then sets the global variables Min and Max to these values.

Function wait_for_mover(void) - This function polls all three axes on the PC23 board to report the current axis position, which is done only after each axis is with any moves in progress. This is the method used to insure that all moves are completed before continuing on with the data acquisition.

Function make_good_filename(char *, char *) - This function adds an extension passed to it to the DOS filename also passed to it.

Function insure_legal_file_name(void) - This function does a trial file open on the file name contained in the global fileName char string. It gives the user additional chances to type in a legal DOS file name until the open is successful. In this way, it is not possible to have the data acquisition finish and then find out the file name is not legal, terminate the program and lose all of the acquired data.

Function input_energy_bins(void) - Uses global array ene_bins[][] and global variables slope and intercept (which define the calibration curve for the MCA. This function gives the user the chance to change these default slope and intercept values. It then prompts the user for the number of energy bins he or she wishes to specify as well as for the starting and ending energy values for each of these bins. A conversion is done on all of the energy values given by the user so that they are stored as channel numbers instead of energy values. These channel numbers are then used to offset the dual-memory port pointer to obtain the proper photon count data for each bin.

3.3 Moverf.c module

Module Summary - This module contains the functions that actually communicate with the PC23 Indexer board. These functions initialize this board and handle reading and writing to and from the board. The reader is referred to Appendix D for a listing of this module. We will now detail each of the functions contained in this module.

Function Initialize(void) - The initialize function and indeed all of the functions in the moverf.c module work with a structure describing the register locations on the PC23 Indexer board. The variable used is a global variable called board which is a pointer to a structure called Board_struct defined as follows:

```
struct Board_struct   {
                        long base;
                        long Command;
                        long Control;
                        long Status;
                        long Data;
                        }; *board;
```

As illustrated above this board variable is indeed a pointer to the Board_struct structure which contains the base address field and the other fields called Command, Control, Status, and Data. These fields within the structure are initialized to point to the base address of the PC23 board, the Command register, the Control register, the Status register, and the Data field on the PC23 board itself.

Within this function we set up these addresses correctly then output to the I/O port address given by dereferencing the board->Control register address and sending to it a STOP which is a bit pattern defined in the header file mover.h that

this board recognizes as a stop function. We then loop waiting for the board to return the proper status (a 00h when the status register contents are bitwise anded with the FAIL_MASK, 20h). We then output to the control register the normal control byte = 60h, i.e. bits 5 and 6 set), followed by a START command (40h), the control byte used to restart the board. We now loop until the board's status register does not contain a RESTART bit pattern (7Fh anded with 17h) or until we time out without the board returning the proper status. If we do time out then this function returns an error condition, else we again output the control byte 0x60 and return a positive integer signalling a good function return. This function returns an integer either signalling a good return or an error condition.

Function WriteCmd(char *) - This function is used to write a character string representing a PC23 command one character at a time by calling the WriteChar function and passing to it one character at a time of this command string. This function firsts sends an ASCII space character (32), then the command string one character at a time, followed by an ASCII carriage return (13). There is no value returned by this function.

Function WriteChar(char) - The WriteChar function accepts a single ASCII character passed to it and then loops until the PC23 board returns a 0x00 status. Then the single character passed to this function is written out to the PC23 board's command register, followed by a IDB byte (70h, the control byte used to set CB4). We then loop again waiting for a status of 0x00, signalling that the PC23 board is ready to accept a new command. We then send a control byte, CB = 60h, again and wait for the status of 00h to be returned.. At this point this function terminates and returns control to its calling function.

Function_ReadAnswer(char *) - The ReadAnswer function does as its name implies, it reads a string of characters from the PC23 board by relying on the ReadChar function to read one character at a time. It continues to read a single character at a time until a carriage return (ASCII 13) is read. This function continues to place the single character returned by ReadChar into a variable called answer_string which is a pointer to a character (string). This pointer is passed to it by the calling function so that the calling function's string variable is actually modified.

Function_ReadChar(void) - The ReadChar function returns a character read from the PC23 board. It firsts reads the status register and if the byte returned from this register is not equal to 0x00, then it loops until the status byte is equal to 0x00. It then reads a byte in from the data register on the PC23 board, puts it into the return_byte variable, and outputs an acknowledge control byte (0E0h0) to the board's control register. It then waits until the status register again reads 0x00 and outputs a control byte of 0x60. This function then returns the value in the return_byte variable.

3.4 Header files

The following header files contain define statements and global variable and function definitions and prototypes. They set up in the C language tradition to better facilitate changes within the hardware setup. For example, if the user changes the base address of the PC23 board in the personal computer all that need be changed in the data acquisition program is the address in the PC23BASE define statement in the acq.h header file and then recompile the program. There are numerous other examples of things that can be easily modified in this program by

making changes to these header files. A brief description of the type of things included in both the acq.h and mover.h header files follows.

Acq.h - This module contains numerous define statements to define parameters such as the maximum number of bins allowed in a multiple energy bin scan, the number of points that are collected in these multiple energy bin scans before a write to the output file is done, the number of inches per step and the number of degrees per step for the stepper motors, the default axis numbers for the various axes, the default acceleration and velocities for these axes, and the default MCA number (should match the MCA board number selectable on the board itself). There are a number of other parameters defined in this file: the default addresses for the MCA board, the default PC23 board base address, etc. For a complete list refer to the header file listing contained in Appendix F. This header file also contains some function prototypes and global variables used by the user interface modules.

Mover.h - This Mover.h header file contains additional define statements for the PC23 board such as the control byte and mask bit definitions. This header file also contains the definition of the global board structure called Board_struct and the associated structure pointer variable called board, as well as the function prototypes for the functions in the moverf.c module.

CHAPTER 4. CONNECTION MACHINE

This chapter will describe some general parallel architecture issues, some general issues involved in parallelization of software, Thinking Machines Connection Machine, and some details about the way the Connection Machine can be programmed. We will also give an assessment of the success we had and compare it to some of the successes found in the literature. We looked at parallelization of an image processing median filter program because it has in it some inherent issues that make it difficult to parallelise (i.e. a lot of data dependencies that will become more apparent in the discussion of this program). We will first look at the parallel machine architectures to gain some understanding of some of the terms used in describing these architectures.

We looked at parallel architecture machines in order to pursue a method of speeding up the computer intensive image processing and reconstructive algorithms. Both of these algorithm types involve doing the same type of operation on large amounts of data. For example, image processing filter algorithms typically must do pixel by pixel sorts for each pixel in the image (may use image sizes of 256 x 256 pixels).

4.1 Parallel Machine Architectures

When classifying computer architectures in the past, the most prevalent system used is one called Flynn's Taxonomy [6]. This system attempts to classify a computer into one of the following four classes:

SISD - Single Instruction, Single Data stream machines. This describes a standard serial computer with one instruction stream and one data stream.

MISD - Multiple Instruction streams, Single Data stream. This class would involve multiple processors applying different instructions to a single datum. This class though hypothetically possible is generally deemed impractical by the computer industry.

SIMD - Single Instruction stream, Multiple Data streams. Machines in this class involve multiple processors simultaneously executing the same instruction on different data.

MIMD - Multiple Instruction streams, Multiple Data streams. These machines have multiple processors autonomously executing diverse instructions on diverse data.

Although these categories provide a useful mechanism for characterizing computer architectures, they are deemed insufficient for classifying various modern computers. For example, pipelined vector processors merit inclusion as parallel architectures since they exhibit substantial concurrent arithmetic execution by executing hundreds of vector elements in parallel. However, it is difficult to classify these vector processors with Flynn's Taxonomy because vector processors do not have processors executing the same instruction operation simultaneously within a clock cycle such as in SIMD, and they lack the asynchronous autonomy of the MIMD category [7]. Duncan in his paper attempts to expand on this classifying system as shown in Figure 15. He has three main categories called Synchronous, MIMD, and MIMD paradigm.

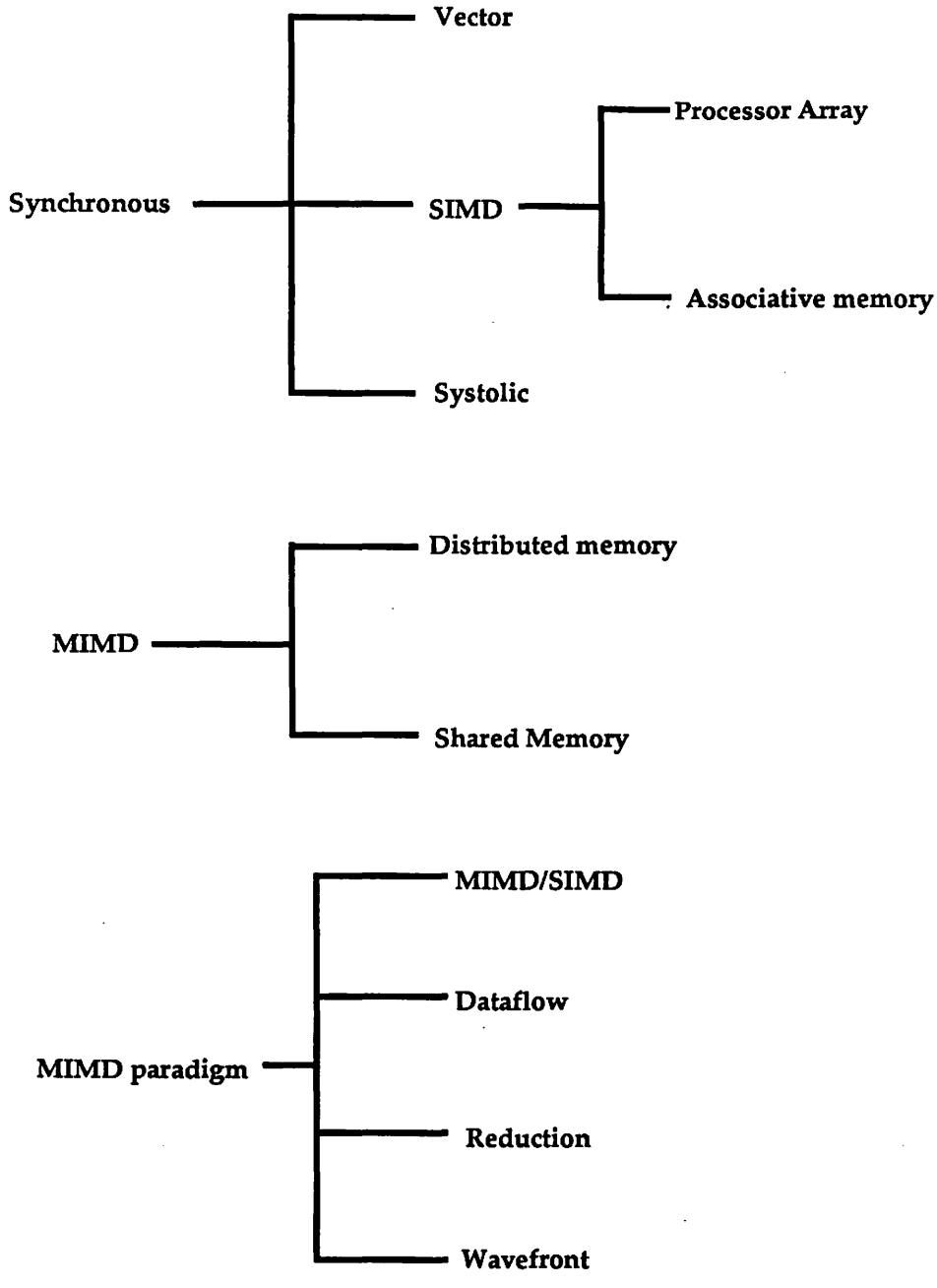


Figure 15. High-level taxonomy of parallel computer architectures [7]

The Synchronous class is defined as parallel architectures that coordinate concurrent operations in lockstep through global clocks, central control units, or vector unit controllers. This class is for the pipelined vector processors previously discussed. Examples of this type of architecture include the Cray 1, Control Data Cyber 205, and the Fujitsu VP-200.

SIMD - The SIMD category of machines typically employ a central control unit, multiple processors, and an interconnection network for either processor to processor or processor to memory communications. The control unit broadcasts a single instruction to all processors which execute the instruction in lockstep fashion on local data (local to the processor). Examples within this category include the Illiac -IV, the Burroughs Scientific Processor, Loral's Massively Parallel Processor, and the Thinking Machines Connection Machine.

Also within the Synchronous category are the systolic array architectures. Systolic array architectures evolved from attempts to get more efficient computing bandwidth from silicon. Systolic arrays can be thought of as a method for designing special-purpose computers to balance resources, I/O bandwidth and computation [8]. Refer to the Duncan paper [7], or a paper by H.T. Kung of Carnegie Mellon University [9] for more information on the systolic array architectures.

MIMD - The MIMD architectures employ processors that can execute independent instruction streams, using local data. MIMD computers then support parallel solutions that require their processors to operate in an autonomous manner. Software processes executing on MIMD machines can communicate by one of two means: 1. Message passing, or 2. Shared Memory.

Message passing MIMD computers have multiple processors that are synchronized by passing messages from one processor to another through an interconnection network. This is the way message passing machines share data amongst their processors.

Shared Memory MIMD computers still have multiple processors, but the communication between them is accomplished through a global shared memory unit which all processors can access at some point. These architectures involve multiple general-purpose processors sharing memory, rather than a CPU and peripheral I/O processors. Problems that must be solved for this type of machine are data access synchronization and cache coherency. These architectures must also have built in hardware support for atomic synchronous mechanisms, such as a test and set instruction, to allow the use of software synchronous tools such as semaphores and/or monitors.

MIMD Paradigms - This category is included to classify MIMD based MIMD/SIMD hybrids such as dataflow architectures, reduction machines, and wavefront arrays. Each of these types is predicated on MIMD principles of asynchronous operation and concurrent manipulation of multiple instruction and data streams. However, each of these architectures is also based on a distinctive organizing principle as fundamental to its overall design as MIMD characteristics.

Dataflow Architectures - The fundamental feature of dataflow architectures is an execution paradigm in which instructions are enabled for execution as soon as all of their data operandi become available. Thus the sequence of execution is based on the flow of data. Examples of this type architecture are the Manchester Data Flow Computer, MIT Tagged Token Data Flow architecture, and the Toulouse LAU System. For more information on these data flow machines refer to [10].

Reduction Architectures - Reduction or demand driven architectures implement an execution paradigm in which an instruction is enabled for execution when its results are required for one of another instructions operands. Reduction program execution consists of recognizing reducible expressions, then replacing them with their calculated values. Therefore, an entire reduction program is ultimately reduced to its result. Further information on this type of architecture can be found in [11].

Wavefront Array Architectures - Wavefront array processors combine systolic data pipelining with an asynchronous dataflow execution paradigm. S-Y. Kung et al. [12] developed wavefront array architectures in the early 1980s to address the following problems - producing efficient, cost-effective architectures for special-purpose systems that balance intensive computations with high I/O bandwidth (the same problems that led to systolic array architecture research). Wavefront and systolic architectures are both characterized by modular processors with regular local interconnection networks. However, wavefront arrays use asynchronous handshaking as the mechanism for coordinating interprocessor data movement instead of using the global clock and explicit time delays used for synchronizing systolic data pipelining. This handshaking mechanism makes computational wavefronts pass smoothly through the array of processors without intersecting as these processors act as a wave propagating medium. The advantages of wavefront architectures over systolic arrays include greatly scalability, simpler programming, and greater fault tolerance. Examples of this architecture have been constructed at Johns Hopkins University and at the Standard Telecommunication and Royal Signals and Radar Establishment in the United Kingdom.

This concludes the discussion of different types of computer architectures and some of their design issues. Next we will describe the architecture of the Thinking Machines Connection Machine called the CM2.

4.2 Connection Machine Architecture

Most computer programs consist of a control sequence and some data elements. Large programs consist of tens of thousands of instructions operating on tens of thousands or even millions of data elements. There is, in theory, an opportunity for parallelism in both the control sequence and in the collection of data elements. In the control sequence we can identify threads of control that can operate independently on different processors. This approach is called control parallelism and is used for programming most multiprocessor computers. The primary difficulty in this approach is the difficulty of identifying and synchronizing these independent threads of control.

Alternatively, we can take advantage of the large number of independent data elements by assigning one processor to each data element and performing all operations on the data in parallel. This approach called data parallelism, works best for large amounts of data [13]. It is this approach that led to the massively parallel architectures containing thousands or even millions of processing elements. Early examples of this type of architecture include ICL's Distributed Array Processor (DAP), Goodyear's Massively Parallel Processor (MPP), Columbia University Non-Von, and now Thinking Machines Connection Machine.

The Connection Machine provides 64K physical processing elements and millions of virtual processor elements with its virtual processing mechanism. It also provides a general-purpose, reconfigurable communications networks.

The Connection Machine is a data-parallel system with integrated hardware software.. Referring to Figure 16, the Connection Machine can have from one to four front end machines connected through a 4x4 Crosspoint switch to from one to four control sequencers. Each sequencer controls up to 16,384 individual processors executing parallel operations. A high performance, data-parallel I/O system (bottom of Figure 16) connects the processors to a peripheral mass storage called the DataVault.

Systems software is based on the operating system or environment of the front-end computer, with some software extensions. Programs have normal sequential control and require no new synchronization structures. Thus it is designed to be relatively easy to develop programs that can take advantage of the Connection Machine hardware.

At the heart of the Connection Machine system lies the parallel-processing units consisting of thousands of processors (up to 64K) each with thousands of bits of memory (4K bits for the CM1 and 64K bits for the CM2). These processors are logically interconnected using several mechanisms to support processor communication.

These communication mechanisms are as follows:

Broadcast Communications - Broadcast communications allow immediate data to be broadcast from the front end computer to all data processors at once.

Global OR - The global OR is a logical OR of the ALU carry output from all data processors. This makes it possible to quickly discover unusual or termination conditions.

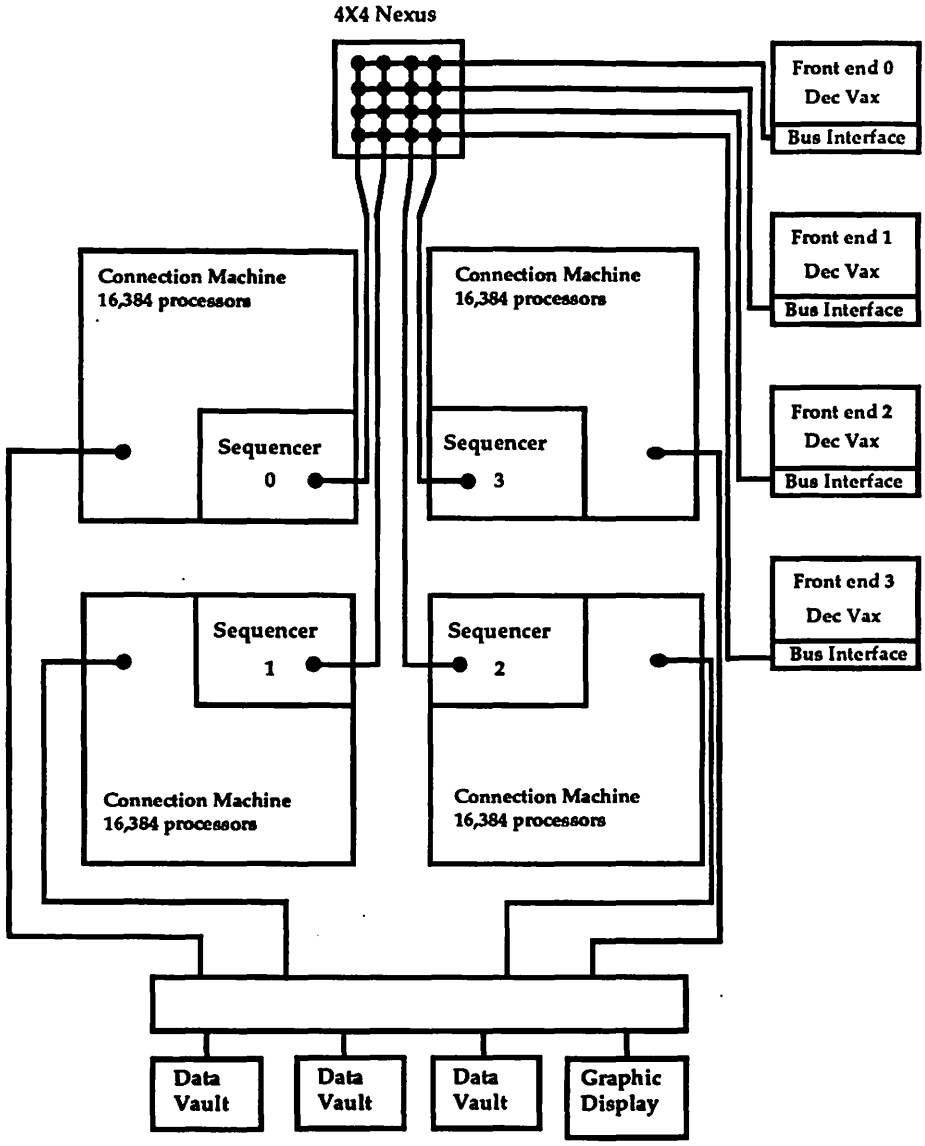


Figure 16. Connection Machine organization

Hypercube Communication - The virtual model supports hypercube communications to form the basis for the router and numerous parallel primitives. Hypercube topology consists of a Boolean n-cube. For a fully configured CM-1, the network is a 12-cube connecting 4096 processor chips (each chip consists of 16 data processors, an ALU, flag bits, and communications interface for the 16 processors) so that each chip is a vertex of a 12-cube.

Router - The router directly implements general pointers following with switched message packets containing processor addresses (the pointers) and data. The router controller uses the hypercube for data transmission, using the standard hypercube nearest-neighbor communication scheme (i.e. each processor is directly connected to its nearest-neighbors - nodes with node I.D.s that differ by only one binary bit). It provides heavily overlapped, pipelined message switching with routing decisions, buffering, and combining of messages directed to the same address, all implemented in hardware.

NEWS grid - The NEWS grid is a two-dimensional Cartesian gird that provides a direct way to perform nearest-neighbor communications. All processors would then communicate in the same direction (North, East, West, or South) so that addresses are implicit and no collisions can occur. This allows NEWS grid communication to be up to 6 times faster than communication using the router for simple regular message patterns.

Virtual Processors - The virtual processor mechanism in the Connection Machine allows the programmer to use more processors than are physically

in the machine. When the Connection Machine is initialized the application specifies the number of processors required. If this number exceeds the number of available physical processors, then the local memory of each processor splits into as many regions as necessary, with the processor automatically time-sliced among the regions.

4.3 Connection Machine Implementation

Converting existing FORTRAN programs to run on the Connection Machine at Argonne National Laboratories (the machine used for this part of the project) is not a terribly difficult task, if all you want to do is to just get them to execute on the Connection Machine. It is another thing all together to get them to execute with performance gain over what would be seen on a single processor RISC based processor (in our case a DEC 5000). These changes involve converting array definitions and usage to take advantage of the array extensions found in the FORTRAN '90 standard. This primarily involves using an arrays like in the following example:

INTEGER A,B : array(10), C: array(5)
A(1:5) = 0 Initializes the first five elements of the A array to 0.
B(6:10) = 1 Initializes the second five elements of the A array to 1.
B(1:5) = 2 Initializes the first five elements of the B array to 2.
B(6:10) = 3 Initializes the second five elements of the B array to 3.
C = A(3:7) + B(4:8)
The above statement then sets:

C(1) = 0 + 2 = 2
C(2) = 0 + 2 = 2
C(3) = 0 + 3 = 3
C(4) = 1 + 3 = 4

$$C(5) = 1 + 3 = 4$$

As this examples shows an array can be treated as if it were a single variable and then the specified operation can be performed on all the specified array elements in parallel. The intention is to make the parallelism as transparent to the programmer as possible. However, as previously stated, it is important to fully understand the interprocessor communications and to utilize the fastest communication technique possible. For more information on the CM FORTRAN style of using arrays and in CM FORTRAN in general the reader is referred to [15].

We had hoped that converting these image processing and tomographic reconstruction programs to run on the Connection Machine with performance improvement, would be a easy task. It was also hoped that it would not require much in the way of program changes to take advantage of this machines inherent ability to handle parallel data problems. However, as the course of this part of the project progressed, we discovered that the effort required to get a significant performance improvement would quickly go beyond the scope of this MS thesis. A source sited in the Connection Machine literature indicates that a man-year of programming is needed to successfully convert a program [14, pp 33].

Further, this conversion project was dropped when we obtained a DSP co-processor board for our data acquisition machine. This board is capable of a peak rate of 100 MFLOPS and has the advantage of being more available.

There are several programming issues that need to be discussed when ever a parallel programming project is started. Conversion from serial programming styles involves the identification of data dependencies , critical variables, and that attention be paid to machine dependent mappings. First we attempted to implement an image processing median filter program using CM FORTRAN.

Figure 17 shows a pictorial representation of the input image array and the wrapped image array for a median filter with a window size of 5 x 5. We use this wrapped array to step through each of the original image array's pixels and then looking at the 5 x 5 window surrounding this pixel and calculating the median pixel value for these 25 pixels within the 5 x 5 window. This process is repeated for all pixels in the original image and gives a smoother resultant image, filtering out high noise characteristics.

This type of algorithm is quite compute intensive especially for an image size of 512 x 512. This size of an image requires 262,144 (512^2) calls to the sort routine that sorts half of the window size until the median pixel value is found. The hope was to be able to parallelise as much of these sort calls as possible. Machines such as the Maspar (an architecture similar to the Connection Machine, but with more powerful and faster processing elements) have been used by others to solve problems that do not involve such huge arrays. They use arrays whose sizes are less than or equal to the largest array size that will fit into memory (i.e. 64 x 64). The huge array sizes we use in our programs are a huge implementation problem.

In order to implement this algorithm on the Connection Machine, we tried to create the wrapped array and locate each pixel so that is was as close as possible to it's surrounding window data to minimize the communication overhead. This should allow us to do the partial sorts to find the median value within each window in parallel. However, we saw a significant communication overhead so that the program would take over 5 minutes to run for an image size of 128 x 128. This median filter program would run considerably faster on the single processor DEC 5000 workstation (in a little under a minute). Most probably the problem was that

89	90	81	82	82	84	85	86	87	88	89	90	81	82
99	100	91	92	93	94	95	96	97	98	99	100	91	92
9	10	1*	2	3	4	5	6	7	8	9	10	1	2
19	20	11	12	13	14	15	16	17	18	19	20	11	12
29	30	21	22	23	24	25	26	27	28	29	30	21	22
39	40	31	32	33	34	35	36	37	38	39	40	31	32
49	59	41	42	43	44	45	46	47	48	49	50	41	42
59	69	51	52	53	54	55	56	57	58	59	60	51	52
69	70	61	62	63	64	65	66	67	68	69	70	61	62
79	80	71	72	73	74	75	76	77	78	79	80	71	72
89	90	81	82	83	84	85	86	87	88	89	90	81	82
99	100	91	92	93	94	95	96	97	98	99	100	91	92
9	10	1	2	3	4	5	6	7	8	9	10	1	2
19	20	11	12	13	14	15	16	17	18	19	20	3	4

* Original image array indices shown by inner double border. Example image size = 10 x 10 with a median filter window size of 5 x 5, yeilding a wrapped image size of 14 x 14.

Figure 17. Filter program wrapped image array

we were not able to properly set up the wrapped array within the Connection Machine virtual machine structure so that this communication was minimized.

CHAPTER 5. CONCLUSIONS

This thesis describes a data acquisition system for an X-ray based NDE inspection research facility. A chapter explaining some background information on some of the requirements for this type of system, an equipment description chapter, a program description chapter, and a parallel processing chapter have been included thus far in this thesis. This chapter will include some comments about some of the problems encountered in testing this data acquisition system, and some insight into some suggested areas to explore further in order to improve on the existing system.

5.1 General Conclusions

The system described in the preceding chapters is installed and working at the Center to enable us to collect energy sensitive photon count data for the 1D, 2D, and tomo scans.

In testing out this data acquisition system (hardware and software alike), a number of problems were found. The more troublesome of these will be briefly discussed in the following paragraphs.

A problem was encountered with the sample positioner not accurately moving a given distance, especially on the X axis. This problem was puzzling for a while because the distances were all correct when running the program through the debugger. It was eventually found that the acceleration and velocity values that were used were too fast so that the positioner motors would slip when trying to move the large weight of the positioner platform for the X axis. Lowering the velocity and acceleration values for the X-axis fixed this particular problem.

Another problem encountered was seen several months after using the program. The symptoms for this problem were that the sample was not returning to the original position after a scan when using particular axis input parameter values. The problem was hard to understand, but basically it amounted to a floating point round off error. This problem when tested showed that for a particular series of statements with a particular set of axis values, there were unexpected answers in some floating point calculations. The program statement that was failing could be reduced such that the problem was easier to understand. In a short test program written to help evaluate this problem the following code fragment produces the unexpected result:

```
float   float1 = 1.0,
        float2 = 0.0,
        float2 = 0.1;
int     int_value;
int_value = (int)((fabs(float1 - float2))/float3);
```

Testing this through the debugger showed that for this code fragment the integer portion of (1- 0)/0.1 was equal to 9. This was rather disturbing and originally thought to be a problem peculiar to the compiler used. However, further testing showed that this problem occurred on a DEC 5000 workstation using a completely different C compiler and having a larger default integer size (32 bits as opposed to 16 bits for the PC). It would fail exactly the same way. This problem was traced to a floating point rounding problem and has been fixed such that it is no longer a problem. Apparently what was happening was that 0.0 was not 0.0 when cast to a double (i.e. it was probably a "dirty" zero for instance 0.0000000000004987430). Well

1.0 - 0.0000000000004987430 is approx. 0.99999999..... so that when divided by 0.1 and cast to an integer the result was truncated to a 9!

Currently the program forces all integers to be rounded to the next higher integer when converting from a floating point or double value. This illustrates the need for a careful, thorough test for the complete data acquisition program. The following represents the test devised for the various scans. Absolute position markers were used on the positioner itself and then trial scans were run and the resultant positions compared to these absolute position markers. This enabled us to verify the positioning controlled by the acquisition program.

5.2 Future Work

Some suggestions come to mind for further work to be done to improve on the system as it exists today.

20 slot PC - In order to work around the problem of running out of slots in the standard PCs used here at the Center, we shopped for and have now ordered a 20 slot industrial AT compatible computer. This computer has not yet arrived and when it does it will be interesting to see if we can implement all of the data acquisition and image processing boards within a single computer and sill get everything to work (i.e. the possible conflicts with Irqs. and I/O bus addresses, and the possible bus timing problems as bus signals are fanned out to more locations).

Ariel board - As previously mentioned (Chapter 4), we have purchased a high speed DSP processor board - an Ariel MM96 board with two Motorola 96010s and 16 Meg of on board RAM. The literature on this board claims a peak speed of 100 MFLOPS. This board when installed and programmed should provide a drastic performance improvement in the image processing algorithms and enable progress

in real time image acquisition and analysis. It is also going to be interesting to see what kind of throughput we can get in implementing the tomography reconstruction algorithm on this DSP board. We might see improvement in this algorithm's throughput in a environment which is much more available to us. It probably will prove to be easier to implement algorithms on this board than on a massively parallel machine because the existing serial program version of this program will not have to be drastically converted to a parallel algorithm version. It will, however, have to be converted to the C language as the current version is in FORTRAN and the only compiler available for the Ariel board is a C compiler.

Counter/Timer board - The recent acquisition of a counter/timer board will enable us to replace the old serial port communication scheme we had to use with the single channel analyzer (SCA). This will allow us to completely avoid the problems with the old interrupt service routine previously in use. So in the near future, we hope to be able to use the SCA by installing this board and writing a more trivial communication routine for it. The SCA is capable of energy sensitive acquisitions, though it is only capable of collecting one energy bin per scan, whereas the MCA can collect up to 2048 energy bins in one scan. However, the SCA has a higher saturation value leading to increased efficiency and thus improved speed. This can dramatically reduce the times required for completion of the longer 2D and tomo scans when using the MCA.

Parallel Processors - Although we determined that the time to convert a program to its parallel equivalent is too prohibitive for our current needs, it would be interesting, if the time does become available, or if newer and faster conversion methods become available, to reinvestigate the usage of the Connection Machine or

perhaps even better the MASPAR (which is available on campus here at Iowa State University).

Vector Processors - The recent emergence of relatively cheap vector processor boards for a PC ISA bus make it a potentially attractive alternative for increasing the speed of the reconstruction and image processing algorithms. Some of these board use the Intel i860 RISC processor which is highly touted for its processing speed. This type of board would be more likely to lend itself to speed improvement for the reconstruction algorithms than would the Ariel DSP board. It would be an interesting project to determine the results for each of these types of options.

Program Improvements - There are improvements that could be made to the data acquisition program. The user interface needs to be improved enhancing the data entry and error reporting abilities. It would also be worthwhile investigating the options for getting around the DOS program size limitation of 640K. Investigating the use of OS/2 or Unix would be worthwhile, but may require programming device drivers for the MCA, SCA, and PC23 boards. Other alternatives worth investigating are the use of protected mode compilers that allow DOS programs to use extended memory thus greatly increasing the allowable program size, and rewriting the program to be a Windows 3.0 program. The latter option involves a considerable learning curve, but also offers the advantage of an improved (and standard) user interface.

BIBLIOGRAPHY

[1] Barrett, H. and Swindell, W. Radiological Imaging Volume 2: The Theory of Image formation, Detection, and Processing. New York: Academic Press, Inc. 1981.

[2] Herman, G.T. Topics in Applied Physics Volume 32: Image Reconstruction from Projections Implementations and Applications. New York: Springer-Verlag. 1979.

[3] Taulb, H. and Schilling, D.L. Digital Integrated Electronics. New York: McGraw Hill. 1977.

[4] EG&G Ortec, GLP Series HPGe (High-Purity Germanium) Low-Energy Photon Spectrometer. Oak Ridge, TN: EG&G Ortec, [n.d.].

[5] Parker Hamilton Corportation, PC-23 Indexer Operator's Manual. Petaluma, CA: Compumotor Division, [n.d.]

[6] Flynn, M.J. "Very High Speed Computing Systems." Proc. IEEE 54 (1966): 1901-1909.

[7] Duncan, R., "A Survey of Parallel Computer Architectures." Computer (Feb. 1990): 5-16.

[8] Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantative Approach. San Mateo, CA: Morgan Kaufmann Publishers, Inc. 1990.

[9] Kung, H.T. "Why Systolic Architectures?." Computer 15 (1982): 37-46.

[10] Srini, V. "An Architectural Comparison of Dataflow Systems." Computer 19 (1986): 68-88.

[11] Treleaven, P., Brownbridge, D., and Hopkins, R. "Data Driven Computer Architecture." ACM Computing Survey 14 (1982): 93-143.

[12] Kung, S.Y., Lo, S.C, Jean, S.N., Hwang, J.N. "Waveform Array Processors - Concept to Implementation." Computer 20 (1987): 18-33.

[13] Hillis, W.P. and Steele, G. L. "Data-Parallel Algorithms." Communications of the ACM 29 (12): 18-33.

[14] Tucker, L.W., Robertson, G.G. "Architecture and Applications of the Connection Machine." Computer (August 1988): 26-33.

[15] Using the Connection Machine System (CM FORTRAN). Technical Memorandum No.118, Rev. 1.Vol. 2. Argonne, Ill: Argonne National Laboratory, 1989.

[16] EG&G Ortec, 916A Hardware Manual. Revision 4. Oak Ridge, Tennessee [n.d.].

ACKNOWLEDGEMENTS

I would especially like to thank my thesis advisor/research coordinator, supervisor, and friend, Joe Gray, for his never ending enthusiasm for his research and for his complete trust in the opinions of his students. Though I might have been able to obtain this degree without Joe's guidance and support, I most certainly would never have been able to enjoy learning as much.

I would also like to thank all the people at the Center for NDE for their unrelenting help. I would especially like to thank: Terry Jensen, Liz Siwek, Jason Ting, Ron Roberts, Mark Kubovich, Ed Doering, Steve Nugen, Neil Johnson, Libby Bilyeu, Connie Nessa, Sarah Jaqua, Diane Miller, Don Thompson, and all the others for their help and guidance.

Thanks are also in order to my other committee members, Terry Smay and Charles Wright. Without their availability and advice, it would not have been possible to obtain this degree.

A special thanks goes out to Nancy, Matthew, and Kristen, whose love and support keep me going and give me the ultimate purpose in all things I pursue. I would never have had the enthusiasm for learning if it weren't for my parents, I thank them from the bottom of my heart.

This work was performed at Ames Laboratory under contract no. W-7405-eng-82 with the U. S. Department of Energy. The United States government has assigned the DOE Report number IS-T 1551 to this thesis.

APPENDIX A. MCA COMMANDS

This appendix contains the MCA board commands used in the data acquisition program described in this thesis. The command strings are built in the acq_data() function and sent by the mbxio() function. The command responses are read and monitored by the get_resp() function. All three of these function reside in the acqray.c program module. For further information on these (or other) MCA commands the reader is referred to [16].

The following is a detailed description of the commands used:

CLEAR
Sets data memory and counters for Live Time and Real Time to zero in the widow-of-interest. CLEAR is equivalent to the CLEAR_DATA and CLEAR_COUNTERS commands. CLEAR cannot be used during data acquisition in the selected MCA segment.

SET_LIVE_PRESET <VALUE>
Sets the Live Time preset value for the selected segment. "Value", in units of 20 milliseconds, is a decimal number expressed as ten ASCII characters. It is converted by the MCA to a 32 bit unsigned binary integer.

START
Starts the data acquisition in the MCA.

SHOW_ACTIVE
Reports whether or not the MCA is acquiring data. The 16 bit answer is transmitted as response record with the following format:

$C xxxxx CCC <DL>

16 bits of data expressed as a five digit (xxxxx) decimal value. The upper limit of the decimal value is 65,535. The final three characters (CCC) are the three digit decimal checksum. <DL> is the delimiting character.

The five digit decimal value returns a 00000 only when there is no active segment.

STOP
Stops the data acquisition in the MCA.

There are numerous commands that the MCA recognizes but are not currently used by the data acquisition program. Interested readers are encouraged to refer to the manual [16] for a comprehensive list of these commands and their corresponding syntax and responses.

APPENDIX B. PC23 COMMANDS

This appendix contains the PC23 board commands used in the data acquisition program described in this thesis. The command strings are built in each of the scan type functions, one_D_scan() , two_D_scan(), and tomo_scan(). They are then sent by the WriteCmd() and WriteChar() functions. The three scan functions reside in the acqray.c program module, and the write and read functions reside in the mover.c program module. For further information on these (or other) PC23 commands the reader is referred to [5].

All PC23 commands are upper case ASCII characters of the form:

[device address][command][parameters][delimiter]

where: device address is a single digit from 1 to 3 (the axis number);

command is a two or three digit command beginning with an upper case ASCII letter (A-Z);

parameters are optional command specific numbers or letters; and

delimiter is a SPACE or CARRIAGE RETURN.

The following symbols and conventions are used in the command descriptions:

a = device address (1-3),
d = delimiter (space or carriage return),
n = ASCII digit in the range 0 - 9 unless otherwise specified in the command summary.
s = sign (+ or -),
Character inside <> are optional.

A Syntax: <a>Annn.nnd

Set acceleration; where nnn.nn is in the range of 0.01 - 999.99 revolutions per second/second. Motor will not move until a non-zero acceleration below 999.99 is defined. Remains set until changed or PC23 is reset.

V Syntax: <a>Vnn.nnd

Set velocity; Where nn.nn ranges from 0.01 to 20.00 revolutions per second.

D Syntax: <a>D<s>nnnnnnnd

Set distance; where nnnnnnn = 0 to 99,999,999 steps. The distance value set with this command will remain the default distance until a new distance is defined, or the indexer is reset. Preset moves will not be executed until the distance is defined as a non-zero value.

MR Syntax: <a>MRnnnd

Motor resolution select. This command sets the motor resolution in the PC23 so that all commands specified in units of revolution (i.e. acceleration and velocity commands) will be yield the correct value for the parameter being specified. Where nn = 00 to 17. See Table B-1 for available motor resolutions.

Table B-1. Motor resolution values to use in the MR command

nn	Resolution*	nn	Resolution*
00	200	10	25000
01	400	11	25400
02	800	12	36000
03	1000	13	50000
04	1600	14	50800
05	3200	15	4096
06	5000	16	12800
07	6400	17	25600
08	10000	18	12500
09	21600	19	16384
		20	20000

*Units are in steps/revolution.

H Syntax: <a>Hsd
Set direction according to s = "+" (CW) or "-" (CCW).

G Syntax: <a>Gd
Go - make a move using the previously entered parameters. It is not necessary to re-enter A, V, and D for each move unless a change in one of the parameters from the last move is necessary.

All of the above commands except the **G** (Go) command are what Compumotor calls parameter commands. This means that these commands sets up a parameter for operation, but does not cause operation. Parameter commands are stored in the PC23 command buffer and executed in sequential order when and execute command is received. The **G** (Go) command is one of these execute commands.

The following example uses all of these commands to move the positioner one move:

1MR20 1A1.0 1V5.0 1D20000 1H+ 1G

This command then sets up the motor resolution to 20,000 steps per revolution, the acceleration to 1.0 rev./sec./sec., the velocity to 5.0 rev./sec., the distance to 20,000 steps (1 rev.), and the direction to CW for axis 1. It then sends a go to axis one to execute the 20,000 step move.

There are numerous other commands that the PC23 recognizes but are not currently used by the data acquisition program. Interested readers are encouraged to refer to the manual [5] for a comprehensive list of these commands and their corresponding syntax.

APPENDIX C. OUTPUT FILE FORMATS

This appendix contains the ASCII GRD file format description and a brief description of the ASCII DAT file format. This GRD format is used as the output file format for all 2 dimensional scans (2D and tomo). The ASCII DAT file format is used as the output file format for the 1D scan.

ASCII GRD file format

For this format description assume that we are doing a tomo scan (i.e. using the X and Θ axis), then the format is as follows:

DSAA
X dimension Θ dimension
X axis beginning value X axis ending value
Θ axis beginning value Θ axis ending value
Minimum output data value Maximum output data value

DATA row 1
DATA row 2
DATA row 3
DATA row 4
.
.
.
.

ASCII DAT file format:

DATA value 1
DATA value 2
DATA value 3
DATA value 4
.
.
.
.

APPENDIX D. MODULE LISTINGS

acqray.c module listing:

```
/*************************************************************************/
/*** FILE: acqray.c, Rick Powell;          Last Modif. Date: 05/11/91 ***/
/*** This program is used to acquire x-ray photon count data for either a ***/
/*** 1D scan, a 2D scan, or a tomographic scan. It must be run on an IBM ***/
/*** PC compatible personal computer with an ISA compatible I/O bus. It ***/
/*** is hard coded to use the EG&G Ortec 916A multichannel analyzer and ***/
/*** the Compumotor PC23 Indexer and MC5300 motor controller. However, ***/
/*** the functions that read and write to and from the PC23 Indexer card ***/
/*** do not require that only the PC23 model of Indexer be used. Any ***/
/*** indexer card that accepts ASCII strings as commands and uses an I/O ***/
/*** bus addressing scheme should work with only a minor modification to ***/
/*** the code (i.e. the define statement that sets up the base address of ***/
/*** the board will need to be changed to point to the new board address).***/
/*** This of course assumes that the register structure of the Indexer ***/
/*** board remain the same. See the moverf.c module for more details on ***/
/*** this structure. The mca mailbox communications functions are not ***/
/*** as transportable to other EG&G Ortec mcas. The mailbox communication ***/
/*** structure changes significantly for the different models requiring ***/
/*** more work be done before changing mca board types. It is still not ***/
/*** a major problem to update the code, just more difficult that for the ***/
/*** Indexer card.                                                    ***/

/*** This version of x-ray is setup to be compiled and linked using ***/
/*** Turbo C++ and its linker, as well as talking to the mca board ***/
/*** directly (i.e. not using Ortec's routines) using text mode ***/
/*** windows and drop down menus. This version deletes threshold useage. ***/
/*** This module must be linked with the user interface modules which ***/
/*** provide the scan parameter information via global and passed ***/
/*** variables.                                                       ***/
/*************************************************************************/

/*************************************************************************/
/***                    INCLUDE FILES                                  ***/
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <dos.h>
#include <fcntl.h>
#include <graphics.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
```

```
#include <conio.h>
#include <string.h>
#include <time.h>
#include <process.h>
#include <errno.h>
#include <alloc.h>
#include "acq.h"
#include "mover.h"

/*************************************************/
/***  FUNCTION  PROTOTYPES for this module      ***/
/*************************************************/
void  InitEverything(void);
void  tomo_scan(char, char);
void  two_D_scan(char, char);
void  one_D_scan(char);
void  one_tomo_scan(char);
unsigned long  acq_data(void);
void  sendString(char *cmdString, char callString[15]);
void  allocate_Temp(void);
void  allocate_mem(void);
void  freeMem(void);
void  write_2D_to_file(void);
void  write_1D_to_file(void);
void  find_limits(int, register int, register int);
int  wait_for_mover(void);
void  make_good_filename(char *, char *);
int  getint(int *);
int  getulong(unsigned long *);
int  getfloat(float *);
void  cls(void);
int  clrkbd(void);
int  newkbhit(void);
int  mbxio(char *, char *, char *);
void insure_legal_file_name(void);
void input_energy_bins(void);
void early_terminate(void);
static  traverse_heap(void);
static  check_heap(void);
extern  int Initialize(void);
extern  void WriteCmd(char *);
extern  int ReadAnswer(char *);


/*************************************************************************/
/***                 Global variables                                  ***/
/*************************************************************************/
extern float param[8][5];
/***  paramater array contains the following:                          ***/
/***     X axis    ===> x1_srt, x1_end, x1_inc, x1_vel, x1_acl
```

```
        Y axis    ===> y1_srt, y1_end, y1_inc, y1_vel, y1_acl
        Z axis    ===> z1_srt, z1_end, z1_inc, z1_vel, z1_acl
        Theta axis ===> t1_srt, t1_end, t1_inc, t1_vel, t1_acl
        Phi axis  ===> p1_srt, p1_end, p1_inc, p1_vel, p1_acl
        X2 axis   ===> x2_srt, x2_end, x2_inc, x2_vel, x2_acl
        Y2 axis   ===> y2_srt, y2_end, y2_inc, y2_vel, y2_acl
        Theta axis ===> t2_srt, t2_end, t2_inc, t2_vel, t2_acl  ***/

extern float  slope,    /*** slope of calibration curve for mca bd.    ***/
          intercept; /*** intercept of the same curve for mca bd.   ***/

extern int  ene_bins[10][2]; /*** array to hold energy bin values ***/

int Zflag = FALSE;       /*** global to this module only ***/
int  ans,              /*** type of scan to be done       ***/
                    /***  1 = move only, no acq.      ***/
                    /***  2 = tomographic scan        ***/
                    /***  3 = 1D scan (X or Z)        ***/
                    /***  4 = 2D scan (X and Z)       ***/
    num_X_moves,       /*** Number of X axis moves to be done.    ***/
    num_Z_moves,       /*** Number of Z axis moves to be done.    ***/
    num_theta_steps,   /*** Number of theta axis moves.           ***/
    num_points = 0,    /*** Number of data points taken since     ***/
                    /*** last write of mult. energy bin counts. ***/
    num_bins = 0,      /*** Number of energy bins to used.        ***/
    beg_chan = 0,      /*** Beginning mca channel number.         ***/
    end_chan = MAXCHAN,  /*** Ending mca channel number.          ***/
    rot_flag = FALSE,   /*** Flag to show if rotation axis is used. ***/
    Live_Time_Flag = TRUE,  /*** Acquire counts using live time acq. ***/
    First_Acq = FALSE,     /*** Flag indicating first data point.  ***/
    redo_flag = FALSE,     /*** redo first acquisiton if true.     ***/
    mult_bins_flag = FALSE, /*** TRUE if selected part energy from menu ***/
    mult_fact=1;          /*** multiplicative factor.             ***/

float Xmin,             /***  Starting X axis value     ***/
    Xmax,             /***  Ending X axis value       ***/
    deltaX,           /***  X axis increment value    ***/
    Zmin,             /***  Starting Z axis value     ***/
    Zmax,             /***  Ending Z axis value       ***/
    deltaZ,           /***  Z axis increment value    ***/
    thetaMin,         /***  Starting Theta axis value ***/
    thetaMax,         /***  Ending Theta axis value   ***/
    deltaTheta,       /***  Theta axis increment value ***/
    count_time = 0.0,   /***  Photon count time in secs. ***/
    tot_time=0.0;

unsigned long  X_steps_per_move,
          Z_steps_per_move,
          theta_steps_per_move,
          threshold,     /***  Photon count threshold.          ***/
```

```
        huge **DataTemp, /*** huge array for total channel counts. ***/
        bin_data[MAX_BINS][POINTS_PER_WRITE], /*** energy bin data ***/
        Min = ULONG_MAX,
        Max = 0,
        eLimits[MAX_BINS][2];  /*** [][0] == Min, [][1] == Max ***/


char Xdir,            /***  Direction of X axis scan     ***/
   Zdir,             /***  Direction of Z axis scan     ***/
   thetaDir,          /***  Direction of Theta axis scan ***/
   *fileName,         /***  Output file name             ***/
   XdirRev,
   ZdirRev,
   thetaDirRev,
   energy_choice;     /*** 1 = All energy bins (channels    ***/
                      /***    250 -1950) inclusive.       ***/
                      /*** 2 = A band of energy specified   ***/
                      /***    by the user.                ***/
                      /*** 3 = A single energy bin specified ***/
                      /***    by the user.                ***/
unsigned char mcb;

static char far  *mcb_outflg,  /* Set true when command is ready */
       far  *mcb_test,    /* Written and read to see if mailbox exists */
       far  *mcb_outlenlo, /* Low byte of length of command string */
       far  *mcb_outlenhi, /* High byte of length of command string */
       far  *mcb_outbuf,  /* Buffer in 916 for commands */
       far  *mcb_inflg,   /* Set TRUE by 916 when response is ready */
       far  *mcb_inlenlo,  /* Low byte of length of response string */
       far  *mcb_inlenhi,  /* High byte of length of response string */
       far  *mcb_inbuf;   /* Buffer in 916 for responses */

/*** External variables found in user interface modules. ***/
extern  float r1d_ctime;
extern  float r2d_ctime;
extern  float tomo_ctime;
extern  float tomo_thd;

/*** Function InitEverything                                    ***/
/*** Initializes all flags, mca mailbox pointers, the PC23 Indexer     ***/
/*** card, and the MCA board.                                   ***/
void InitEverything()
{
  int  result;
  char cmdString[80],
      resp[512],
      per_resp[512];

  /* Make sure the PC23 is alive and well */
  if ((result = Initialize()) < 0)
```

```
  {
    printf("Can't Initialize the PC23\n");
    printf("result = %d\n", result);
    exit(-1);
  }

  /* Initialize pointers */
  mcb_outflg = (char far *)OUTFLG;
  mcb_test = (char far *)TEST;
  mcb_outlenlo = (char far *)OUTLENLO;
  mcb_outlenhi = (char far *)OUTLENHI;
  mcb_outbuf = (char far *)OUTBUF;
  mcb_inflg = (char far *)INFLG;
  mcb_inlenlo = (char far *)INLENLO;
  mcb_inlenhi = (char far *)INLENHI;
  mcb_inbuf = (char far *)INBUF;

 mcb = (unsigned char)(DATAMCB - 1);
 (void)outportb(0x292, mcb);


 /*** Initialize the 916 mca data acquisition board and then set ***/
 /*** the gain_conversion for the mca board to MAXCHAN + 1      ***/
 sprintf(cmdString, "initialize");
 if (mbxio(cmdString, resp, per_resp) == -1)
 {
   printf("ERROR: the command string initialize returned an error!\n");
   printf("resp = %s, per_resp = %s\n", resp, per_resp);
   exit(1);
 }

 sprintf(cmdString, "set_gain_conversion %d", (MAXCHAN + 1));
 if (mbxio(cmdString, resp, per_resp) == -1)
 {
   printf("ERROR: set_gain_conversion returned an error!\n");
   printf("resp = %s, per_resp = %s\n", resp, per_resp);
   exit(1);
 }

 /*** Initialize all flags to defaults ***/
 rot_flag = FALSE;
 Zflag = FALSE;
 Live_Time_Flag = TRUE,
 First_Acq = FALSE,
 redo_flag = FALSE,
 mult_fact=1;

} /*** end of InitEverything ***/
```

```
/*** cls function : Clears the screen. ***/
void cls()
{
  union REGS  r;

  r.h.ah = 6;
  r.h.al = 0;
  r.h.ch = 0;
  r.h.cl = 0;
  r.h.dh = 24;
  r.h.dl = 79;
  r.h.bh = 7;
  int86(0x10, &r, &r);
} /*** end of cls ***/


/*************************************************************************/
/************************** Function: tomo_scan **************************/
/*************************************************************************/
/*** This function implements the tomographic scan. It is hard coded at ***/
/*** this time to use theta for it's rotational axis. It scans linearly ***/
/*** in one direction, rotates one theta axis delta value, then scans   ***/
/*** in the reverse linear direction, rotates again and continues for a ***/
/*** total of (num_X_moves * num_theta_steps) moves.                    ***/
/***    If a multiple energy bin tomo scan is chosen, then this function ***/
/*** writes the multiple energy bin data to num_bins number of files.   ***/
/*** The file names for these files are determined as follows. Use all  ***/
/*** all characters of the original file name up to the '.' and extension ***/
/*** append on an extension of 'en#', where # = the energy bin number   ***/
/*** (0 to num_bins-1). Example: if the user types in an output file name***/
/*** of "ceramic.grd", for the total channel count data, then the energy ***/
/*** bin file names for 3 bins would be: ceramic.en0, ceramic.en1, and  ***/
/*** ceramic.en2 (currently we are limited to ten energy bins).         ***/
/*** Parameters: Pass the two axis numbers of the axis to use for scan. ***/
void tomo_scan(char axis1, char axis2)
{
  register int  i, j, k;
  char  Xcmd[80],
        XcmdRev[80],
        thetaCmd[80],
        thetaCmdRev[80],
        callfile[20],
        chbuffer[20],
        X_ptr[20],
        Theta_ptr[20],
        ch,
        *p;
  int  evenFlag = FALSE,
        testCh,
        files_open,
```

```
    X_index = num_X_moves,
    ii,
    jj,
    kk;
FILE *outTemp,
    *fp[10],
    *usefull;
fpos_t  *pos;

fileName = tomo;
count_time = tomo_ctime;
threshold = tomo_thd;
ans = 2;

if ((param[0][1]-param[0][0])!=0.0)
{
  Zflag = FALSE;
}
else if ((param[2][1]-param[2][0])!=0)
{
  Zflag = TRUE;
}
rot_flag = TRUE;
if (Zflag)
{
  num_Z_moves = (int)(ceil((fabs(param[2][1]-param[2][0]))/param[2][2]));
  Z_steps_per_move = (unsigned long)(param[2][2] / INCHES_PER_STEP);
}
else
{
  X_steps_per_move = (unsigned long)(param[0][2] / INCHES_PER_STEP);
  num_X_moves = (int)(ceil((fabs(param[0][1]-param[0][0]))/param[0][2]));
}

num_theta_steps = (int)(ceil((fabs(param[3][1]-param[3][0]))/param[3][2]));
theta_steps_per_move = (unsigned long)(param[3][2] / DEGREES_PER_STEP);

if ((param[0][1]-param[0][0]) >= 0)
{
  Xdir = '+';       /*** Right  ***/
  XdirRev = '-';    /*** Left ***/
}
else
{
  Xdir = '-';       /*** Left  ***/
  XdirRev = '+';    /*** Right ***/
}

if ((param[3][1]-param[3][0]) >= 0)
{
```

```
    thetaDir = '+';   /*** CCW   ***/
    thetaDirRev = '-'; /*** CW   ***/
  }
  else
  {
    thetaDir = '-';   /*** CCW   ***/
    thetaDirRev = '+'; /*** CW   ***/
  }

  X_index = num_X_moves;
  insure_legal_file_name();

  one_tomo_scan('2');  /*** Do one extra scan angle at large theta angle ***/

  window(45,15,79,16);
  gotoxy(1,1);
  textcolor(LIGHTGRAY);
  textbackground(RED);
  cprintf("Indices_____Photon Count\r\n");

  ultoa(X_steps_per_move, X_ptr, 10);
  ultoa(theta_steps_per_move, Theta_ptr, 10);

  sprintf(Xcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
    MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
    X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, Xdir, X_AXIS, GO);

  sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
    MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
    X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS, GO);

  sprintf(thetaCmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", THETA_AXIS,
    MOTOR_RESOLUTION, THETA_AXIS, param[3][4], THETA_AXIS, param[3][3],
    THETA_AXIS, DISTANCE, Theta_ptr,THETA_AXIS, DIR_DEF, thetaDir,
    THETA_AXIS, GO);

  sprintf(thetaCmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", THETA_AXIS,
    MOTOR_RESOLUTION, THETA_AXIS, param[3][4], THETA_AXIS, param[3][3],
    THETA_AXIS, DISTANCE, Theta_ptr, THETA_AXIS, DIR_DEF, thetaDirRev,
    THETA_AXIS, GO);

  allocate_Temp();

#ifdef DEBUG
  check_heap();
  traverse_heap();
#endif

  if ((outTemp = fopen("tempFile.grd", "w")) == NULL)
  {
```

```
    printf("fopen of temporary output file tempFile.grd failed!!\n");
    exit(0);
  }

  fprintf(outTemp, "DSAA\n%d %d\n", num_X_moves+1, num_theta_steps);
  fprintf(outTemp, "%f %f\n%f %f\n", Xmin, Xmax,
      (thetaMin/DEGREES_PER_RADIAN), (thetaMax/DEGREES_PER_RADIAN));
  fclose(outTemp);

        window(45,16,79,24);
  gotoxy(1,1);
  textcolor(LIGHTGRAY);
  textbackground(RED);

/*********************************************************/
/********* Beginning of main loop for tomo_scan *********/
/*********************************************************/
  again:   for (i=0; i<= num_theta_steps; i++)
  {
    for (j=0; j<= num_X_moves; j++)
    {

      if ((i==0) && (j==0))
        First_Acq = TRUE;
      else
        First_Acq = FALSE;

      if ((i % 2) == 0)
        evenFlag = TRUE;
      else
        evenFlag = FALSE;

      /*** acquire photon count data for count_time secs. ***/
      if (j > 0)   /*** move sample in the X dir if not first scan ***/
      {
        if (evenFlag)  /*** if i is even (even theta increment) ***/
        {
          WriteCmd(Xcmd);
                if (wait_for_mover()!=0)
          {
            printf("***ERROR: timeout occurred in wait_for_mover!!\n");
            printf("Program saving data and exiting now--BYE.....\n");
            if (i == 0)
              num_X_moves = j + 1;
            num_theta_steps = i + 1;
            files_open = flushall();
            if (mult_bins_flag)
              for (jj=0; jj<num_bins; jj++)
                fclose(fp[jj]);
            write_2D_to_file();
```

```
        freeMem();
        exit(0);
      }
    }
    else
    {
      WriteCmd(XcmdRev);
      if (wait_for_mover()!=0)
      {
        printf("***ERROR: timeout occurred in wait_for_mover!!\n");
        printf("Program saving data and exiting now--BYE.....\n");
        if (i == 0)
          num_X_moves = j + 1;
        num_theta_steps = i + 1;
        files_open = flushall();
        if (mult_bins_flag)
          for (jj=0; jj<num_bins; jj++)
            fclose(fp[jj]);
        write_2D_to_file();
        freeMem();
        exit(0);
      }
    } /*** end else i % 2 ***/
  } /*** end if j > 0 ***/

  if (evenFlag)
  {
    if ((i >= 0) && (j >= 0) && (j <= num_X_moves)
        && (i <= num_theta_steps))
    {
      DataTemp[i][j] = acq_data();
    }

    else
    {
      printf("Index: %d, %d into temp array not legal!\n", i, j);
      write_2D_to_file();
      freeMem();
      exit(0);
    }
  }
  else
  {
    if ((X_index >= 0) && (i >= 0) && (X_index <= num_X_moves)
        && (i <= num_theta_steps))
    {
      DataTemp[i][X_index] = acq_data();
      X_index--;
    }
    else
```

```
  {
    printf("Index: %d, %d into temp array not legal!\n",
          i, X_index);
    write_2D_to_file();
    freeMem();
    exit(0);
  }
}

textcolor(LIGHTGRAY);
textbackground(RED);
if (evenFlag)
{
  if (mult_bins_flag)
    cprintf("%3d %3d %7lu %7lu %7lu\r\n", i, j,
          bin_data[0][j], bin_data[1][j],
          bin_data[2][j]);
  else
    cprintf("%3d %3d    %7lu\r\n", i, j, DataTemp[i][j]);
}
else
{
  if (mult_bins_flag)
    cprintf("%3d %3d %7lu %7lu %7lu\r\n", i,
          X_index+1, bin_data[0][j],
          bin_data[1][j], bin_data[2][j]);
  else
    cprintf("%3d %3d    %7lu\r\n", i, X_index+1,
          DataTemp[i][X_index+1]);
}

if (kbhit())
{
 if ((testCh = getch()) == ESC)
 {
    printf("Program saving data and exiting now--BYE.....\n");
    if (i == 0)
      num_X_moves = j + 1;
    num_theta_steps = i + 1;
    if (mult_bins_flag)
      for (jj=0; jj<num_bins; jj++)
        fclose(fp[jj]);
    write_2D_to_file();
    freeMem();
    exit(0);
 }
}

if (redo_flag)
  break;
```

```
    if (mult_bins_flag)
    {
      for (jj=0; jj<num_bins; jj++)
      {
        for (ii=0,p=fileName; (*p!='.')&&(*p!='\0');p++,ii++)
        {
          callfile[ii] = *p;
          if ((*(p + 1) == '.')&&(*(p+1)!='\0'))
            callfile[ii+1] = '\0';
        }  /*** end for ii ***/
        sprintf(chbuffer, "%s.en%d", callfile, jj);
        if (First_Acq)
        {
          if ((fp[jj] = fopen(chbuffer, "w")) == NULL)
          {
            printf("fopen of output file %s failed!!\n", chbuffer);
          }
          find_limits(ans, num_theta_steps, num_X_moves);
          fprintf(fp[jj], "DSAA\n%d %d\n", num_X_moves+1,
                num_theta_steps+1);
          fprintf(fp[jj], "%f %f\n%f %f\n", fabs(param[0][0]),
                fabs(param[0][1]),
                (fabs(param[3][0])/DEGREES_PER_RADIAN),
                (fabs(param[3][1])/DEGREES_PER_RADIAN));
          fprintf(fp[jj], "%lu %lu\n", eLimits[jj][0],
                eLimits[jj][1]);
        }  /*** end if First_Acq ***/
      }  /*** end for jj ***/
    }  /*** end if mult_bins_flag ***/

    num_points++;

  }  /* <======================== end for j ============================*/

  if (mult_bins_flag)
  {
    /*** Write out the current energy bin counts.        ***/
    /*** Write out each projection's  (one line's)       ***/
    /*** energy bin photon counts to each file.          ***/
    /*** Then close all files.  There should be num_bins  ***/
    /*** files.                                          ***/
    for (jj=0; jj<num_bins; jj++)
    {
      for (kk=0; kk<=num_X_moves; kk++)
        fprintf(fp[jj], "%10lu", bin_data[jj][kk]);
      fprintf(fp[jj], "\n");
    }
    files_open = flushall();
  }
```

```
num_points = 0;
if (redo_flag)
  break;

if (evenFlag)
  X_index = num_X_moves;  /*** reset X_index for each even ***/
                  /*** move of the theta axis     ***/

if (i<num_theta_steps)
{
  WriteCmd(thetaCmd);
      if (wait_for_mover()!=0)
  {
    printf("***ERROR: timeout occurred in wait_for_mover!!\n");
    printf("Program saving data and exiting now--BYE.....\n");
    if (i == 0)
      num_X_moves = j + 1;
    num_theta_steps = i + 1;
    files_open = flushall();
    if (mult_bins_flag)
      for (jj=0; jj<num_bins; jj++)
        fclose(fp[jj]);
    write_2D_to_file();
    freeMem();
    exit(0);
  }
}

/*********************************************************************/
/*** Write counts to temporary file after each line scan is complete. */
/*********************************************************************/
if ((outTemp = fopen("tempFile.grd", "a")) == NULL)
{
  printf("fopen of temporary output file tempFile.grd failed!!\n");
  exit(0);
}

if (i < 1)
{
  find_limits(ans, i, num_X_moves);
  if (fgetpos(outTemp, pos) != 0)
    printf("****************** fgetpos error!! **************\n");
  fprintf(outTemp, "%lu %lu\n", Min, Max);
}
else
{
  find_limits(ans, i, num_X_moves);
}
if (i == num_theta_steps-1)
```

```
  {
    if (fsetpos(outTemp, pos) !=0)
      printf("****************** fsetpos error!! ***************\n");
    fprintf(outTemp, "%lu %lu\n", Min, Max);
  }

  for (k=0; k<=num_X_moves; k++)
  {
    fprintf(outTemp, "%12lu", DataTemp[i][k]);
  }

  fprintf(outTemp, "\n");

  fclose(outTemp);

} /* <========================== end for i ============================*/

for (jj=0; jj<num_bins; jj++)
  fclose(fp[jj]);

if (redo_flag)
  goto again;

/*****************************************************************/
/************** Reposition the sample at the origin **************/
/*****************************************************************/
if (evenFlag)
{
  /*** Need to reposition along both axis ***/
  ultoa((unsigned long)(ceil((num_X_moves*param[0][2])/INCHES_PER_STEP)),
                X_ptr, 10);

  sprintf(Xcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
       MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
       X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, Xdir, X_AXIS, GO);

  sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
       MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
       X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS, GO);

  WriteCmd(XcmdRev);
  if (wait_for_mover()!=0)
  {
    printf("***ERROR: timeout occurred in wait_for_mover!!\n");
    printf("However, tomo_scan done anyway!!!\n");
   }
}
/*** Almost always reposition the theta axis ***/
ultoa((unsigned long)(ceil(360.0 -
       ((num_theta_steps*param[3][2])/DEGREES_PER_STEP))), Theta_ptr, 10);
```

```
sprintf(thetaCmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", THETA_AXIS,
      MOTOR_RESOLUTION, THETA_AXIS, param[3][4], THETA_AXIS,
      param[3][3], THETA_AXIS, DISTANCE, Theta_ptr, THETA_AXIS, DIR_DEF,
      thetaDir, THETA_AXIS, GO);

ultoa((unsigned long)(ceil((num_theta_steps*param[3][2])
                  /DEGREES_PER_STEP)), Theta_ptr, 10);
sprintf(thetaCmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", THETA_AXIS,
      MOTOR_RESOLUTION, THETA_AXIS, param[3][4], THETA_AXIS, param[3][3],
      THETA_AXIS, DISTANCE, Theta_ptr, THETA_AXIS, DIR_DEF, thetaDirRev,
      THETA_AXIS, GO);

if ((fabs(param[3][1] - param[3][0])) != 360.0)
{
  if ((fabs(param[3][1] - param[3][0])) > 180.0) /*** > 180 degrees ***/
  {
    /*** Rotate back to origin in same direction as tomo_scan. ***/
    WriteCmd(thetaCmd);
    if (wait_for_mover()!=0)
    {
      printf("***ERROR: timeout occurred in wait_for_mover!!\n");
      printf("However, tomo_scan done anyway!!!\n");
    }
  } /*** end of if thetaMax - thetaMin ***/
  else
  {
    /*** Rotate back to original position in opposite direction. ***/
    WriteCmd(thetaCmdRev);
        if (wait_for_mover()!=0)
    {
      printf("***ERROR: timeout occurred in wait_for_mover!!\n");
      printf("However, tomo_scan done anyway!!!\n");
    }
  } /*** end of else thetaMax - thetaMin ***/
}

if ((usefull = fopen("usefull.dat", "w")) == NULL)
{
  printf("fopen of usefull.dat file failed!!\n");
  exit(0);
}
fprintf(usefull, "enebin#___start_____end___\n");
for (i=0; i<num_bins; i++)
{
  fprintf(usefull, "%7d %9d %9d\n", i, ene_bins[i][0], ene_bins[i][1]);
}

fprintf(usefull, "enebin#___Min_____Max____\n");
for (i=0; i<num_bins; i++)
{
```

```
    fprintf(usefull, "%7d %10lu %10lu\n", i, eLimits[i][0], eLimits[i][1]);
  }
  fclose(usefull);
  cls();
  write_2D_to_file();
  exit(0);

} /*------------------------- end of tomo_scan -------------------------*/

/*--------------------------------------------------------------------------*/
/*------------------------- Function: two_D_scan -------------------------*/
/*--------------------------------------------------------------------------*/
/*** This function implements the 2D scan operation. The two axis chosen ***/
/*** are selected by the user in the user interface section. At this time***/
/*** any two of the following axis may be chosen: X, Z, and theta.      ***/
/*** Currently there is no option for energy bin scans, this will be    ***/
/*** implemented at a later date.                                       ***/
void two_D_scan(char axis1, char axis2)
{
  register int  i, j, k;
  char  Xcmd[80],
        XcmdRev[80],
        Zcmd[80],
        ZcmdRev[80],
        X_ptr[20],
        Z_ptr[20],
        ch;
   int  evenFlag = FALSE,
           testCh,
           files_open,
        X_index = num_X_moves;
   FILE *outTemp;

  fileName = r2d;
  count_time = r2d_ctime;
  threshold = tomo_thd;
  ans = 4;
  if ((param[2][1]-param[2][0])!=0.0)
    Zflag=TRUE;
  else
    Zflag=FALSE;
  rot_flag=FALSE;

  num_Z_moves = (int)((fabs(param[2][1] - param[2][0])) / param[2][2]);
  Z_steps_per_move = (unsigned long)(param[2][2] / INCHES_PER_STEP);
  X_steps_per_move = (unsigned long)(param[0][2] / INCHES_PER_STEP);
  num_X_moves = (int)((fabs(param[0][1] - param[0][0])) / param[0][2]);

  if ((param[0][1]-param[0][0]) > 0)
  {
```

```
  Xdir = '+';    /*** Right ***/
  XdirRev = '-'; /*** Left ***/
 }
 else
 {
  Xdir = '-';    /*** Left ***/
  XdirRev = '+'; /*** Right ***/
 }

 if ((param[2][1]-param[2][0]) > 0)
 {
  Zdir = '+';    /*** Down ***/
  ZdirRev = '-'; /*** Up   ***/
 }
 else
 {
  Zdir = '-';    /*** Up ***/
  ZdirRev = '+'; /*** Down ***/
 }

 insure_legal_file_name();

        window(45,15,79,16);
 gotoxy(1,1);
 textcolor(LIGHTGRAY);
 textbackground(RED);
 cprintf("Indexes_____Photon Count\r\n");


 ultoa(X_steps_per_move, X_ptr, 10);
 ultoa(Z_steps_per_move, Z_ptr, 10);

 sprintf(Xcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
      MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
      X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, Xdir, X_AXIS, GO);

 sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
      MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
      X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS, GO);

 sprintf(Zcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", Z_AXIS,
      MOTOR_RESOLUTION, Z_AXIS, param[2][4], Z_AXIS, param[2][3],
      Z_AXIS, DISTANCE, Z_ptr, Z_AXIS, DIR_DEF, Zdir, Z_AXIS, GO);

 sprintf(ZcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", Z_AXIS,
      MOTOR_RESOLUTION, Z_AXIS, param[2][4], Z_AXIS, param[2][3],
      Z_AXIS, DISTANCE, Z_ptr, Z_AXIS, DIR_DEF, ZdirRev, Z_AXIS, GO);

#ifdef DEBUG
 printf("Xcmd = %s\n", Xcmd);
```

```
  printf("Zcmd = %s\n", Zcmd);
#endif

  /*** Need to setup commands to the mca board(s) as well ***/

  allocate_Temp();

#ifdef DEBUG
  check_heap();
  traverse_heap();
#endif


  if ((outTemp = fopen("tempFile.grd", "w")) == NULL)
  {
    printf("fopen of temporary output file tempFile.grd failed!!\n");
    exit(0);
  }

  fprintf(outTemp, "DSAA\n%d %d\n", num_X_moves, num_Z_moves);
  fprintf(outTemp, "%f %f\n%f %f\n", Xmin, Xmax, Zmin, Zmax);
  fclose(outTemp);

        window(45,15,79,24);
  gotoxy(1,1);

  again1:  for (i=0; i<= num_Z_moves; i++)
  {
    for (j=0; j<= num_X_moves; j++)
    {
      if ((i==0) && (j==0))
        First_Acq = TRUE;
      else
        First_Acq = FALSE;

      if ((i % 2) == 0)
        evenFlag = TRUE;
      else
        evenFlag = FALSE;

      /*** acquire photon count data for count_time secs. ***/
      if (j > 0)   /*** move sample in the X dir if not first scan ***/
      {
        if (evenFlag)  /*** if i is even (even Z increment) ***/
        {
          WriteCmd(Xcmd);
          if (wait_for_mover()!=0)
          {
            printf("***ERROR: timeout occurred in wait_for_mover!!\n");
            printf("Program saving data and exiting now--BYE.....\n");
```

```
            if (i == 0)
              num_X_moves = j + 1;
            num_Z_moves = i + 1;
            files_open = flushall();
/***          if (mult_bins_flag)
              for (jj=0; jj<num_bins; jj++)
                fclose(fp[jj]); ***/
            write_2D_to_file();
            freeMem();
            exit(0);
          }
        }
        else
        {
          WriteCmd(XcmdRev);
          if (wait_for_mover()!=0)
          {
            printf("***ERROR: timeout occurred in wait_for_mover!!\n");
            printf("Program saving data and exiting now--BYE.....\n");
            if (i == 0)
              num_X_moves = j + 1;
            num_Z_moves = i + 1;
            files_open = flushall();
/***          if (mult_bins_flag)
              for (jj=0; jj<num_bins; jj++)
                fclose(fp[jj]);  ***/
            write_2D_to_file();
            freeMem();
            exit(0);
          }
        } /*** end else i % 2 ***/
      } /*** end if j > 0 ***/

      if (evenFlag)
      {
        if ((i >= 0) && (j >= 0) && (j <= num_X_moves)
            && (i <= num_Z_moves))
        {
          DataTemp[i][j] = acq_data();
            }

        else
        {
          printf("Index: %d, %d into data array not legal!\n", i, j);
          exit(0);
        }
      }
      else
      {
        if ((X_index >= 0) && (i >= 0) && (X_index <= num_X_moves)
```

```
          && (i <= num_Z_moves))
        {
          X_index--;
          DataTemp[i][X_index] = acq_data();
        }
        else
        {
          printf("Index: %d, %d into data array not legal!\n", i,
                X_index);
          exit(0);
        }
      }

      textcolor(LIGHTGRAY);
      textbackground(RED);
      if (evenFlag)
       cprintf("%d %d      %12lu\r\n", i, j, DataTemp[i][j]);
      else
       cprintf("%d %d      %12lu\r\n", i, X_index,
            DataTemp[i][X_index]);

    if (kbhit())
    {
      if ((testCh = getch()) == ESC)
      {
        if (i == 0)
          num_X_moves = j + 1;
        num_Z_moves = i + 1;
/***    if (mult_bins_flag)
        for (jj=0; jj<num_bins; jj++)
          fclose(fp[jj]);          ***/
        write_2D_to_file();
        printf("Program saving data and exiting now, BYE.....\n");
        freeMem();
        exit(0);
      }
    }

    if (redo_flag)
      break;

    } /*** end for j ***/

    if (evenFlag)
      X_index = num_X_moves;  /*** reset X_index for each even ***/
                      /*** move of the Z axis     ***/

    if (i < num_Z_moves)
    {
      WriteCmd(Zcmd);
```

```
      if (wait_for_mover()!=0)
      {
        printf("***ERROR: timeout occurred in wait_for_mover!!\n");
        printf("Program saving data and exiting now--BYE.....\n");
        if (i == 0)
          num_X_moves = j + 1;
        num_Z_moves = i + 1;
        files_open = flushall();
/***          if (mult_bins_flag)
             for (jj=0; jj<num_bins; jj++)
               fclose(fp[jj]);  ***/
        write_2D_to_file();
        freeMem();
        exit(0);
      }
    }

    /*********************************************************************/
    /** Write counts to temporary file after each line scan is complete. **/
    /*********************************************************************/
    if ((outTemp = fopen("tempFile.grd", "a")) == NULL)
    {
      printf("fopen of temporary output file tempFile.grd failed!!\n");
      exit(0);
    }

    if (i < 1)
    {
      find_limits(ans, i, num_X_moves);
      fprintf(outTemp, "%lu %lu\n", Min, Max);
    }
    else
    {
      find_limits(ans, i, num_X_moves);
    }

    for (k=0; k<=num_X_moves; k++)
    {
      fprintf(outTemp, "%12lu", DataTemp[i][k]);
    }

    fprintf(outTemp, "\n");

    fclose(outTemp);

    if (redo_flag)
      break;

  } /*** end for i ***/
```

```
  if (redo_flag)
    goto again1;

  /*****************************************************************/
  /*************** Reposition the sample at the origin ***************/
  /*****************************************************************/
  if (evenFlag)
  {
    /*** Need to reposition along both axis ***/
    ultoa((unsigned long)((fabs(param[0][1]-param[0][0]))/INCHES_PER_STEP), X_ptr, 10);

    sprintf(Xcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
          MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
          X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, Xdir, X_AXIS, GO);

    sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
          MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
          X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS, GO);

    WriteCmd(XcmdRev);
    if (wait_for_mover()!=0)
    {
      printf("***ERROR: timeout occurred in wait_for_mover!!\n");
      printf("However, two_D_scan done anyway!!!\n");
    }
  }
  /*** Always reposition the Z axis ***/
  ultoa((unsigned long)((fabs(param[2][1]-param[2][0]))/INCHES_PER_STEP),
              Z_ptr, 10);
  sprintf(Zcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", Z_AXIS,
        MOTOR_RESOLUTION, Z_AXIS, param[2][4], Z_AXIS, param[2][3],
        Z_AXIS, DISTANCE, Z_ptr, Z_AXIS, DIR_DEF, Zdir, Z_AXIS, GO);

  sprintf(ZcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", Z_AXIS,
        MOTOR_RESOLUTION, Z_AXIS, param[2][4], Z_AXIS, param[2][3],
        Z_AXIS, DISTANCE, Z_ptr, Z_AXIS, DIR_DEF, ZdirRev, Z_AXIS, GO);
  WriteCmd(ZcmdRev);
  if (wait_for_mover()!=0)
  {
    printf("***ERROR: timeout occurred in wait_for_mover!!\n");
    printf("However, two_D_scan done anyway!!!\n");
  }

  write_2D_to_file();

} /*---------- end of two_D_scan ----------*/


/*---------- Function: one_D_scan ----------*/
/*---------- 1-D scan and acquisition ----------*/
```

```
/*** This function implements the 1D scan operation.  The axis chosen    ***/
/*** is selected by the user in the user interface section.  At this time ***/
/*** any one of the following axis may be chosen: X, Z, or theta.        ***/
/*** Currently there is no option for energy bin scans, this will be     ***/
/*** implemented at a later date.                                        ***/
void  one_D_scan(char axis1)
{
  register int  i, j;
  char  Xcmd[80],
        XcmdRev[80],
        X_ptr[20],
        Zcmd[80],
        ZcmdRev[80],
        Z_ptr[20],
        thetaCmd[80],
        thetaCmdRev[80],
        Theta_ptr[20],
        ch;
  int  testCh,
       endfor,
       endforTemp;
  float deltaVal;
  FILE  *outTemp;

  fileName = r1d;
  count_time = r1d_ctime;
  threshold = tomo_thd;
  ans = 3;
  if ((param[2][1]-param[2][0])!=0.0)
    Zflag=TRUE;
  else
    Zflag=FALSE;
  rot_flag=FALSE;
  if (Zflag)
  {
    num_Z_moves = (int)((fabs(param[2][1] - param[2][0])) / param[2][2]);
    Z_steps_per_move = (unsigned long)(param[2][2] / INCHES_PER_STEP);
  }
  else
  {
    X_steps_per_move = (unsigned long)(param[0][2] / INCHES_PER_STEP);
    num_X_moves = (int)(ceil((fabs(param[0][1]-param[0][0]))/param[0][2]));
  }

  if ((param[0][1]-param[0][0]) > 0)
  {
    Xdir = '+';    /*** Right ***/
    XdirRev = '-'; /*** Left ***/
  }
  else
```

```
{
  Xdir = '-';    /*** Left  ***/
  XdirRev = '+'; /*** Right ***/
}

if ((param[2][1]-param[2][0]) > 0)
{
  Zdir = '+';    /*** Down  ***/
  ZdirRev = '-'; /*** Up    ***/
}
else
{
  Zdir = '-';    /*** Up  ***/
  ZdirRev = '+'; /*** Down ***/
}


insure_legal_file_name();

window(45,15,79,16);
gotoxy(1,1);
textcolor(LIGHTGRAY);
textbackground(RED);
for (i=0;i<8;i++)
{
  cprintf("                          \r\n");
}
gotoxy(1,1);
cprintf("Index_____Photon Count      \r\n");

if (ans != 1)
  allocate_Temp();

if ((outTemp = fopen("tempFile.grd", "w")) == NULL)
{
  printf("fopen of temporary output file tempFile.grd failed!!\n");
  exit(0);
}

fclose(outTemp);

if ((param[0][1]-param[0][0]) != 0)
{
  ultoa(X_steps_per_move, X_ptr, 10);
  sprintf(Xcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
       MOTOR_RESOLUTION, X_AXIS, param[0][3], X_AXIS, param[0][3],
       X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, Xdir, X_AXIS, GO);

  sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
       MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
```

```
        X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS, GO);
    endfor = num_X_moves;
    endforTemp = (int)(num_X_moves * 0.10);
    deltaVal = deltaX;
  }
  else
  {
    ultoa(Z_steps_per_move, Z_ptr, 10);
    sprintf(Zcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", Z_AXIS,
        MOTOR_RESOLUTION, Z_AXIS, param[0][3], Z_AXIS, param[0][3],
        Z_AXIS, DISTANCE, Z_ptr, X_AXIS, DIR_DEF, Xdir, X_AXIS, GO);

    sprintf(ZcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", Z_AXIS,
        MOTOR_RESOLUTION, Z_AXIS, param[0][4], Z_AXIS, param[0][3],
        Z_AXIS, DISTANCE, Z_ptr, Z_AXIS, DIR_DEF, ZdirRev, Z_AXIS, GO);
    endfor = num_Z_moves;
    endforTemp = (int)(num_Z_moves * 0.10);
    deltaVal = deltaZ;
  }

  if (ans == 1)
    endfor--;

  window(45,16,79,24);
  gotoxy(1,1);


  again2:  for (i=0; i<= endfor; i++)
  {
    if (i==0)
      First_Acq = TRUE;
    else
      First_Acq = FALSE;

    if ((i > 0) || (ans == 1))
    /*** move sample in the appropriate dir if not first scan ***/
    {
      if ((param[0][1]-param[0][0]) != 0)
      {
        WriteCmd(Xcmd);
        if (wait_for_mover()!=0)
        {
          printf("***ERROR: timeout occurred in wait_for_mover!!\n");
          printf("Program saving data and exiting now--BYE.....\n");
          num_X_moves = i + 1;
/***         if (mult_bins_flag)
            for (jj=0; jj<num_bins; jj++)
              fclose(fp[jj]); ***/
          write_1D_to_file();
          freeMem();
```

```
          exit(0);
        }
      }
      else
      {
        WriteCmd(Zcmd);
        if (wait_for_mover()!=0)
        {
          printf("***ERROR: timeout occurred in wait_for_mover!!\n");
          printf("Program saving data and exiting now--BYE.....\n");
          num_X_moves = i + 1;
/***           if (mult_bins_flag)
            for (jj=0; jj<num_bins; jj++)
              fclose(fp[jj]); ***/
          write_1D_to_file();
          freeMem();
          exit(0);
        }
      }
    }

    if (endforTemp < 1)
      endforTemp = 1;

    /*** If ans != 1, acquire photon count data for count_time secs. ***/
    if (ans != 1)
    {
      DataTemp[0][i] = acq_data();

      if (((i % endforTemp) == 0) && (i != 0))
      {
        /*** Write current data to temporary file. ***/
        if ((outTemp = fopen("tempFile.grd", "a")) == NULL)
        {
          printf("fopen of temp output file tempFile.grd failed!!\n");
          exit(0);
        }

        for (j=i-endforTemp; j<=i; j++)
        {
          fprintf(outTemp, "%10f, %12lu", (j * deltaVal),
                DataTemp[0][j]);
        }

        fclose(outTemp);

      } /*** end of if i % endforTemp ***/

      textcolor(LIGHTGRAY);
      textbackground(RED);
```

```
      cprintf("%d       %10lu\r\n", i, DataTemp[0][i]);

      if (kbhit())
      {
        if ((testCh = getch()) == ESC)
        {
          if (Zflag)  num_Z_moves = i+1;
          else      num_X_moves = i+1;
          write_1D_to_file();
          printf("Program saving data and exiting now, BYE.....\n");
          freeMem();
          exit(0);
        } /*** end of if testCh = ***/
      } /*** end of if kbhit ***/
    } /*** end of if ans != 1 ***/

    if (redo_flag)
      break;

  } /*** end of for i ***/

  if (redo_flag)
    goto again2;

  if (ans != 1)
  {
    if (!Zflag)
    {
      /*** Reposition the sample to it's original position. ***/
      ultoa((unsigned long)(ceil((num_X_moves*param[0][2])
          /INCHES_PER_STEP)), X_ptr, 10);
      sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
          MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
          X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS,
          GO);

#ifdef DEBUG
      printf("\n\n(Xmax - Xmin)/INCHES_PER_STEP =  %f\n",
          (Xmax - Xmin)/INCHES_PER_STEP);
      printf("\n\nafter casting to a long, it =  %ld\n",
          (long)((Xmax - Xmin)/INCHES_PER_STEP));

      printf("While X_ptr = %s\n", X_ptr);
#endif


      WriteCmd(XcmdRev);
      if (wait_for_mover()!=0)
      {
        printf("***ERROR: timeout occurred in wait_for_mover!!\n");
```

```
        printf("Program one_D_scan done anyway!!!\n");
      }
    }
    else
    {
      ultoa((unsigned long)(ceil((num_Z_moves*param[2][2])
            /INCHES_PER_STEP)), Z_ptr, 10);
      sprintf(ZcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", Z_AXIS,
          MOTOR_RESOLUTION, Z_AXIS, param[0][4], Z_AXIS, param[0][3],
          Z_AXIS, DISTANCE, Z_ptr, Z_AXIS, DIR_DEF, ZdirRev, Z_AXIS,
          GO);

#ifdef DEBUG
      printf("\n\n(Zmax - Zmin)/INCHES_PER_STEP =  %f\n",
          (Zmax - Zmin)/INCHES_PER_STEP);
      printf("\n\nafter casting to a long, it =  %ld\n",
          (long)((Zmax - Zmin)/INCHES_PER_STEP));

      printf("While Z_ptr = %s\n", Z_ptr);
#endif


      WriteCmd(ZcmdRev);
      if (wait_for_mover()!=0)
      {
        printf("***ERROR: timeout occurred in wait_for_mover!!\n");
        printf("Program one_D_scan done anyway!!!!\n");
      }
    } /*** end of if !Zflag ***/
  } /*** end of if ans != 1 ***/
  write_1D_to_file();

} /****************** end of one_D_scan function ************************/

/*-------------------- Function: one_tomo_scan ------------------------*/
/*---------------------- 1-D scan and acquisition ----------------------*/
/*** Move positioner to theta = (thetaMax-thetaMin)/2 + thetaMax  ***/
/*** Then do one linear scan at this angle, then move positioner  ***/
/*** back to original start position and start tomo scan!!!       ***/
/*** Example:  If you are doing a 0 to 180 degree tomographic scan ***/
/*** then we move the positioner to 90+180=270 degrees for this   ***/
/*** initial linear scan.  This allows us to obtain additional    ***/
/*** useful data from an angle that we would not ordinarily get.  ***/
void  one_tomo_scan(char axis1)
{
  register int  i, j;
  char  Xcmd[80],
      XcmdRev[80],
      X_ptr[20],
      Zcmd[80],
```

```
    ZcmdRev[80],
    Z_ptr[20],
    thetaCmd[80],
    thetaCmdRev[80],
    Theta_ptr[20],
    tempFile[80],
    ch;
int  testCh,
    endfor,
    endforTemp;
float deltaVal;
FILE *outTemp;

strcpy(tempFile, fileName);
strcpy(fileName, "extra.dat");
count_time = tomo_ctime;
threshold = tomo_thd;
ans = 3;   /*** temporarily make scan type 1D ***/
Zflag=FALSE;
rot_flag=FALSE;
X_steps_per_move = (unsigned long)(param[0][2] / INCHES_PER_STEP);
num_X_moves = (int)(ceil((fabs(param[0][1]-param[0][0])) / param[0][2]));

if ((param[0][1]-param[0][0]) > 0)
{
  Xdir = '+';    /*** Right ***/
  XdirRev = '-'; /*** Left ***/
}
else
{
  Xdir = '-';    /*** Left ***/
  XdirRev = '+'; /*** Right ***/
}

insure_legal_file_name();

window(45,15,79,16);
textcolor(LIGHTGRAY);
textbackground(RED);
cprintf("Index_____Photon Count\r\n");

if (ans != 1)
  allocate_Temp();

ultoa(X_steps_per_move, X_ptr, 10);
sprintf(Xcmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
     MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
     X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, Xdir, X_AXIS, GO);

sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
```

```
      MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
      X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS, GO);
endfor = num_X_moves;
endforTemp = (int)(num_X_moves * 0.10);
deltaVal = param[0][2];

ultoa((unsigned long)((fabs(param[3][1]-param[3][0])/2 +
    param[3][1])/DEGREES_PER_STEP), Theta_ptr, 10);
sprintf(thetaCmd, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", THETA_AXIS,
      MOTOR_RESOLUTION, THETA_AXIS, param[3][4], THETA_AXIS,
      param[3][3], THETA_AXIS, DISTANCE, Theta_ptr,THETA_AXIS, DIR_DEF,
      thetaDir, THETA_AXIS, GO);

sprintf(thetaCmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", THETA_AXIS,
      MOTOR_RESOLUTION, THETA_AXIS, param[3][4], THETA_AXIS,
      param[3][3], THETA_AXIS, DISTANCE, Theta_ptr, THETA_AXIS, DIR_DEF,
      thetaDirRev, THETA_AXIS, GO);

WriteCmd(thetaCmd);
if (wait_for_mover()!=0)
{
  printf("***ERROR: timeout occurred in wait_for_mover.\n");
  printf("Trying the theta axis big move for extra tomo scan!!!\n");
  printf("Program exiting now--BYE, no data to save!!!!!\n");
  freeMem();
  exit(0);
}

if (ans == 1)
  endfor--;

window(45,16,79,24);
gotoxy(1,1);
textcolor(LIGHTGRAY);
textbackground(RED);

again2:  for (i=0; i<= endfor; i++)
{
  if (i==0)
    First_Acq = TRUE;
  else
    First_Acq = FALSE;

  if ((i > 0) || (ans == 1))
  /*** move sample in the appropriate dir if not first scan ***/
  {
    WriteCmd(Xcmd);
    if (wait_for_mover()!=0)
    {
      printf("***ERROR: timeout occurred in wait_for_mover!!\n");
```

```
        num_X_moves = i + 1;
        write_1D_to_file();
        printf("Program saving data and exiting now, BYE.....\n");
        freeMem();
        exit(0);
      }
    }

    if (endforTemp < 1)
      endforTemp = 1;

    /*** If ans != 1, acquire photon count data for count_time secs. ***/
    if (ans != 1)
    {
      DataTemp[0][i] = acq_data();
      textcolor(LIGHTGRAY);
      textbackground(RED);
      cprintf("%d      %10lu\r\n", i, DataTemp[0][i]);

      if (kbhit())
      {
        if ((testCh = getch()) == ESC)
        {
          num_X_moves = i + 1;
          write_1D_to_file();
          printf("Program saving data and exiting now, BYE.....\n");
          freeMem();
          exit(0);
        } /*** end of if testCh = ***/
      } /*** end of if kbhit ***/
    } /*** end of if ans != 1 ***/

    if (redo_flag)
      break;

  } /*** end of for i ***/

  if (redo_flag)
    goto again2;

  /*** Reposition the sample to it's original position. ***/
  ultoa((unsigned long)(ceil((num_X_moves*param[0][2])/INCHES_PER_STEP)),
        X_ptr, 10);
  sprintf(XcmdRev, "%s%s %sA%f %sV%f %s%s%s %s%s%c %s%s", X_AXIS,
        MOTOR_RESOLUTION, X_AXIS, param[0][4], X_AXIS, param[0][3],
        X_AXIS, DISTANCE, X_ptr, X_AXIS, DIR_DEF, XdirRev, X_AXIS, GO);

#ifdef DEBUG
      printf("\n\n(Xmax - Xmin)/INCHES_PER_STEP =  %f\n",
          (Xmax - Xmin)/INCHES_PER_STEP);
```

```
      printf("\n\nafter casting to a long, it =  %ld\n",
          (long)((Xmax - Xmin)/INCHES_PER_STEP));

      printf("While X_ptr = %s\n", X_ptr);
#endif


  WriteCmd(XcmdRev);
  if (wait_for_mover()!=0)
  {
    printf("***ERROR: timeout occurred in wait_for_mover!!\n");
    printf("***Doing extra tomo scan, no data to save!!!\n");
  }

  WriteCmd(thetaCmdRev);
  if (wait_for_mover()!=0)
  {
    printf("***ERROR: timeout occurred in wait_for_mover!!\n");
    printf("***Doing extra tomo scan, no data to save!!!\n");
  }

  write_1D_to_file();

  strcpy(fileName, tempFile);
  ans = 2;      /*** set global scan type back to tomo scan ***/
  rot_flag = TRUE;

} /*---------- end of one_tomo_scan function ----------*/

/*---------- Function: acq_data ----------*/
/*-------- This function starts and stops the data acqusition then reads --*/
/*-------- dual port memory on the mca board itself and returns the data --*/
/*-------- result to the calling function.  The photon count returned -----*/
/*-------- is either the sum of all the requested channel's photon counts -*/
/*-------- or the multiple energy bin photon counts if multiple energy ----*/
/*-------- bins are desired. ----------*/
unsigned long  acq_data()
{
  unsigned long  temp_photon_cnt = 0,
            live_time_count = 0,
            real_time_count = 0,
            far  *channel_ptr;  /*** pointer to dual port memory ***/

  register int  j, k;

  int  mca=DATAMCB,
      segment=01,
      result,
      comm_err = 0;
```

```
  unsigned long  real_temp=0L, live_temp=0L;

  float realtime=0.0,
      livetime=0.0;

  char  cmdString[80],
      resp[512],
      per_resp[512];

  realtime = 0.0;
  livetime = count_time;

  real_temp = (unsigned long)(realtime/20*1000);
  live_temp = (unsigned long)(livetime/20*1000);

  sprintf(cmdString, "clear");
  if (mbxio(cmdString, resp, per_resp) == -1)
  {
    printf("ERROR: the command string clear returned an error!\n");
    printf("resp = %s, per_resp = %s\n", resp, per_resp);
    printf("cmdString = %s\n", cmdString);
    exit(1);
  }


  sprintf(cmdString, "set_live_preset %lu", live_temp);
  if (mbxio(cmdString, resp, per_resp) == -1)
  {
    printf("ERROR: the command string set_live returned an error!\n");
    printf("resp = %s, per_resp = %s\n", resp, per_resp);
    printf("cmdString= %s\n",cmdString);
    exit(1);
  }

#ifdef DEBUG
  sprintf(cmdString, "show_live_preset");
  if (mbxio(cmdString, resp, per_resp) == -1)
  {
    printf("ERROR: command string show_live_preset returned an error!\n");
    printf("resp = %s, per_resp = %s\n", resp, per_resp);
    printf("cmdString = %s\n", cmdString);
    exit(1);
  }
  printf("show_live_preset resp = %s\n", resp);

  sprintf(cmdString, "show_live");
  if (mbxio(cmdString, resp, per_resp) == -1)
  {
    printf("ERROR: the command string show_live returned an error!\n");
    printf("resp = %s, per_resp = %s\n", resp, per_resp);
```

```
    printf("cmdString = %s\n", cmdString);
    exit(1);
  }
#endif

  sprintf(cmdString, "start");
  if (mbxio(cmdString, resp, per_resp) == -1)
  {
    printf("ERROR: the command start returned an error!\n");
    printf("resp = %s, per_resp = %s\n", resp, per_resp);
    printf("cmdString = %s\n", cmdString);
    exit(1);
  }

  sprintf(cmdString, "show_active");
  do
  {
#ifdef DEBUG
    sprintf(cmdString, "show_live");
    if (mbxio(cmdString, resp, per_resp) == -1)
    {
      printf("ERROR: the command string show_live returned an error!\n");
      printf("resp = %s, per_resp = %s\n", resp, per_resp);
      printf("cmdString = %s\n", cmdString);
      exit(1);
    }
    printf("show_live resp = %s\n", resp);
    sprintf(cmdString, "show_active");
#endif
    if (mbxio(cmdString, resp, per_resp) == -1)
    {
      printf("ERROR: the command string show_active failed!\n");
      printf("resp = %s, per_resp = %s\n", resp, per_resp);
      printf("cmdString = %s\n", cmdString);
      exit(1);
    }
#ifdef DEBUG
    printf("show_active resp = %s\n", resp);
    printf("In the do loop waiting for mca bd. to go inactive!!\n");
#endif
  } while (strnicmp(resp, "$C00000", 7) != 0);

  sprintf(cmdString, "stop");
  if (mbxio(cmdString, resp, per_resp) == -1)
  {
    printf("ERROR: the command stop returned an error!\n");
    printf("resp = %s, per_resp = %s\n", resp, per_resp);
    printf("cmdString = %s\n", cmdString);
    exit(1);
  }
```

```
  channel_ptr = (unsigned long far *)BASEADD;

/***   (void)outport(0x292, DATAMCB-1);  ***/

  if (mult_bins_flag)
  {
    for (k=0; k<num_bins;k++)
    {
      for (j=ene_bins[k][0];j<ene_bins[k][1]; j++)
        temp_photon_cnt += (MASK & *(channel_ptr + j));
      bin_data[k][num_points] = temp_photon_cnt;
      if (temp_photon_cnt < eLimits[k][0])
        eLimits[k][0] = temp_photon_cnt;
      if (temp_photon_cnt > eLimits[k][1])
        eLimits[k][1] = temp_photon_cnt;
      temp_photon_cnt = 0;
    }
  }
  else
  {
    for (j=beg_chan; j<=end_chan; j++)
    {
      temp_photon_cnt += (MASK & *(channel_ptr + j));
#ifdef DEBUG1
      printf("current photon count = %12lu, ",
            (MASK & *(channel_ptr + j)));
      printf("count = %10lu, pointer = %lu\n", temp_photon_cnt,
           (channel_ptr + j));
#endif
    }
  } /*** end of else ***/

  return temp_photon_cnt;


} /*---------------------- end of acq_data ------------------------*/


/*********************************************************************/
/*** Function mbxio: This function sends a command to the mailbox ***/
/*** and requests its response.                                    ***/
/*********************************************************************/
int mbxio(command, response, per_response)

char *command,      /* Command to send to 916 */
   *response,     /* Response from 916, if any */
   *per_response;  /* Percent response from 916 */

{
```

```
extern char *get_resp(void);

int  counter,   /* General loop counter */
     errflg;    /* Timeout error flag */

time_t  start_time, /* Used to calculate timeout */
        present_time;

errflg = -1;          /* Init as error until we are successful */
*mcb_outlenlo = '\0'; /* Send a zero length message to sync mailbox */
*mcb_outlenhi = '\0';
*mcb_outflg = TRUED;

*mcb_inflg = FALSED;
start_time = time(NULL);
present_time = time(NULL);
while((*mcb_outflg == '\377')&&(difftime(present_time, start_time)<5.0))
{
  *mcb_inflg = FALSED;
  present_time = time(NULL);
}
if (difftime(present_time, start_time) >= 5.0)
{
  printf("MCB not responding!!!\n");
  printf("command = %s\n", command);
  return(errflg);
}

/* Put command in output buffer */
  for (counter = 0; counter < strlen(command); counter++)
    *(mcb_outbuf + (4 * counter)) = *(command + counter);

/* Write out length of command */
  *mcb_outlenlo = (char)(strlen(command) % 256);
  *mcb_outlenhi = (char)(strlen(command) / 256);

/* Set the out flag to say the command is ready */
  *mcb_outflg = TRUED;

/* Get the first response record */
  strcpy(per_response, get_resp());
  if (strcmpi(per_response, "err") == 0)
    return(errflg);

/* See if it was a percent repsonse */
  if (strncmp(per_response, "%", 1) == 0)
  {
    strnset(response, '\0', 1);
    errflg = 0;      /* Good return */
    return(errflg);
```

```
  }

/* It wasn't a percent response, so copy it and */
/* go get the percent response */
  strcpy(response, per_response);
  strcpy(per_response, get_resp());
  if (strcmpi(per_response, "err") == 0)
    return(errflg);

  errflg = 0;      /* Good return */
  return(errflg);

}


/*************************************************************************/
/*   Function get_resp  This function gets the response from the MCA  ***/
/*************************************************************************/
char *get_resp()
{
  char  resp_buf[512];

  int   counter,
        num_chars;

  time_t  start_time,
          present_time;

/* Wait for MCB response */
  start_time = time(NULL);
  present_time = time(NULL);
  while ((*mcb_inflg==FALSED)&&(difftime(present_time, start_time) < 5.0))
    present_time = time(NULL);
  if (difftime(present_time, start_time) >= 5.0)
  {
    printf("MCB not responding!!\n");
    strcpy(resp_buf, "err");
    return(resp_buf);
  }

/* Get number of characters in response and read */
  num_chars = (int)*mcb_inlenlo + 256 * (int)*mcb_inlenhi;
  memset(resp_buf, '\0', 512);
  for (counter = 0; counter < num_chars; counter++)
    resp_buf[counter] = *(mcb_inbuf + (4 * counter));

/* Reset input buffer flag and return response address */
  *mcb_inflg = FALSED;
  return(resp_buf);
```

```
}

/*-------------------------------------------------------------------------*/
/*---------------------- Function: sendString ----------------------------*/
/*-------------------------------------------------------------------------*/
/*** This function sends a command string to DOS.                    ***/
void sendString(cmdString, callString)
char * cmdString;
char callString[15];
{
  int result;

  if ((result = system(cmdString)) == -1)
  {
    printf("ERROR: reported on the call to %s\n", callString);
    if (errno == ENOMEM)  printf("No memory available!\n");
    if (errno == ENOEXEC) printf("Not executable file!\n");
    if (errno == ENOENT)  printf("Either path or file not found!\n");
    if (errno == EINVAL)  printf("Modeflag is invalid!\n");
    if (errno == E2BIG)   printf("See manual\n");
    exit(0);
  }
} /*** end of sendString ***/



/*-------------------------------------------------------------------------*/
/*------------------------- Function: allocate_Temp ----------------------*/
/*-------This function allocates memory for the 2-d array called DataTemp --*/
/*-------------------------------------------------------------------------*/
void allocate_Temp()
{
  register int  i;

  int  mx=0,
       my=0;

  if ((ans == 3) && (Zflag))
    my = num_Z_moves + 1;
  else
    my = num_X_moves + 1;
  if (ans == 2)
    mx = num_theta_steps + 1;
  if (ans == 4)
    mx = num_Z_moves + 1;

  if (ans != 3)
  {
                if ((DataTemp=(unsigned long huge **)malloc((unsigned long)mx *
            sizeof(unsigned long huge *))) == NULL)
```

```
                {
                        printf("malloc called with mx = %d and sizeof long huge * = %d\n",
                                        mx, sizeof(long huge *));
                        printf("Allocation of mem for the x dim. of DataTemp failed!\n");
                        exit(0);
                }

    for (i=0; i<mx; i++)
    {
                        if ((DataTemp[i]=(unsigned long huge *)malloc((unsigned long)my *
        sizeof(unsigned long))) == NULL)
                        {
                                printf("malloc called with my = %d and sizeof long = %d\n",
                                                my, sizeof(long));
                                printf("Allocation of mem for the y dim. of DataTemp failed!\n");
                                freeMem();
                                exit(0);
                        }
                }
        }
        else
        {
                if ((DataTemp = (unsigned long huge **)malloc((unsigned long)1 *
      sizeof(unsigned long huge *))) == NULL)
                {
                        printf("malloc called with my = %d and sizeof long huge * = %d\n",
          my, sizeof(long huge *));
    printf("Allocation of mem for the x dim. of DataTemp failed!\n");
    exit(0);
    }

                if ((DataTemp[0]=(unsigned long huge *)malloc((unsigned long)my *
              sizeof(unsigned long))) == NULL)
                {
                        printf("malloc called with my = %d and sizeof long = %d\n",
                                        my, sizeof(long));
                        printf("Allocation of mem for single dim. of DataTemp failed!\n");
                        freeMem();
                        exit(0);
                }
        }
} /*---------------------- end of allocate_Temp ----------------------*/


/*------------------------------------------------------------------*/
/*---------------------- Function: freeMem() -----------------------*/
/*---------- Frees up the memory allocated by allocateTemp() ---------*/
/*------------------------------------------------------------------*/
void freeMem()
{
```

```
  register int  i;

  int  mx=0,
       my=0;

  mx = num_X_moves + 1;

  if (ans == 3)  free(DataTemp[0]);
  else
  {
    for (i=0; i<=mx; i++)
      free(DataTemp[i]);
  }

  free(DataTemp);

} /*--------------------------- end freeMem() ---------------------------*/


/*-------------------------------------------------------------------*/
/*---------------------- Function: write_2D_to_file ----------------------*/
/*------------ write the 2-d array called DataTemp to file: nameFile ------*/
/*-------------------------------------------------------------------*/
void  write_2D_to_file()
{
  register int  i,j;
  int  endfor;

  FILE *outfile;


  window(45,15,79,24);
  gotoxy(1,1);

  for (i=0;i<10;i++)
  {
    cprintf("                          \r\n");
  }

  if ((outfile = fopen(fileName, "w")) == NULL)
  {
    outfile = fopen("outsave.grd", "w");
    printf("fopen of output file %s failed!!\n", fileName);
  }


  if (ans == 2)
  {
    find_limits(ans, num_theta_steps, num_X_moves);
    fprintf(outfile, "DSAA\n%d %d\n", num_X_moves, num_theta_steps);
```

```
    fprintf(outfile, "%f %f\n%f %f\n", fabs(param[0][0]), fabs(param[0][1]),
          (fabs(param[3][0])/DEGREES_PER_RADIAN),
          (fabs(param[3][1])/DEGREES_PER_RADIAN));
    fprintf(outfile, "%lu %lu\n", Min, Max);
    endfor = num_theta_steps;
  }
  else if (ans == 4)
  {
    find_limits(ans, num_Z_moves, num_X_moves);
    fprintf(outfile, "DSAA\n%d %d\n", num_X_moves, num_Z_moves);
    fprintf(outfile, "%f %f\n%f %f\n", fabs(param[0][0]), fabs(param[0][1]),
          fabs(param[2][0]), fabs(param[2][1]));
    fprintf(outfile, "%lu %lu\n", Min, Max);
    endfor = num_Z_moves;
  }

  for (i=0; i<=endfor; i++)
  {
    for (j=0; j<=num_X_moves; j++)
    {
      fprintf(outfile, "%12lu", DataTemp[i][j]);
    }
    fprintf(outfile, "\n");

  }

  fclose(outfile);

  freeMem();

  window(10,20,79,24);
  gotoxy(1,1);
  textcolor(LIGHTGRAY);
  textbackground(BLUE);

  cprintf("\n\n\n\n\n\nWrote data to file: %s\n", fileName);

} /*---------------------- end of write_2D_to_file ---------------------*/


/*--------------------------------------------------------------------*/
/*---------------------- Function: write_1D_to_file ------------------*/
/*---------- write the 1-d array called DataTemp to file: fileName ----*/
/*--------------------------------------------------------------------*/
void  write_1D_to_file()
{
  register int  i,
          endfor;

  float deltaVal;
```

```
FILE *outfile;

window(45,15,44,24);
gotoxy(1,1);
for (i=0;i<10;i++)
{
  cprintf("                          \r\n");
}

if ((outfile = fopen(fileName, "w")) == NULL)
{
  outfile = fopen("outsave.dat", "w");
  printf("fopen of output file %s failed!!\n", fileName);
}

find_limits(ans, num_Z_moves, num_X_moves);

if (rot_flag)
{
  deltaVal = param[3][2];
  endfor = num_theta_steps;
}
else if (Zflag)
{
  deltaVal = param[2][2];
  endfor = num_Z_moves;
}
else
{
  deltaVal = param[0][2];
  endfor = num_X_moves;
}

for (i=0; i<=endfor; i++)
{
  fprintf(outfile, "%10f %10lu\n", (i * deltaVal), DataTemp[0][i]);
}

fprintf(outfile, "\n");

fclose(outfile);

freeMem();

window(10,20,44,24);
gotoxy(1,1);
textcolor(LIGHTGRAY);
textbackground(BLUE);
```

```
  cprintf("\n\n\n\n\n\nWrote data to file: %s\n", fileName);

} /*---------------------- end of write_1D_to_file ---------------------*/


/*-------------------------------------------------------------------*/
/*------------------------ Function: find_limits --------------------*/
/*---------------- find the min. and max photon counts in DataTemp----------*/
/*-------------------------------------------------------------------*/
void  find_limits(mode, endfori, endforj)
int   mode;
register int   endfori;
register int   endforj;
{
  register int   i,
                 j;

  if (mode == 3)
  {
    if (rot_flag)
      endforj = num_theta_steps;
    else if (Zflag)
      endforj = num_Z_moves;
    else
      endforj = num_X_moves;

    for (j=0; j<=endforj; j++)
    {
      if (DataTemp[0][j] < Min)
        Min = DataTemp[0][j];
      if (DataTemp[0][j] > Max)
        Max = DataTemp[0][j];
    }
  } /*** end of if mode == 3 ***/
  else
  {

    for (i=0; i<endfori; i++)
    {
      for (j=0; j<=endforj; j++)
      {
        if (DataTemp[i][j] < Min)
          Min = DataTemp[i][j];
        if (DataTemp[i][j] > Max)
          Max = DataTemp[i][j];
      }
    }

  } /*** end of else ***/
```

```
} /*---------------------- end of find_limits ----------------------*/


/*************************************************************************/
/*** This function polls all three axis on the PC23 to report the current ***/
/*** axis position, which is done only after each axis is with any moves  ***/
/*** in progress. This is a slick/crude method of determining when all  ***/
/*** positioning is finished.                                           ***/
int wait_for_mover(void)
{
char x_temp[80];
char y_temp[80];
char z_temp[80];

  WriteCmd("1P\r");
  if (ReadAnswer(x_temp)!=0)
    return(-1);
  WriteCmd("2P\r");
  if (ReadAnswer(y_temp)!=0)
    return(-1);
  WriteCmd("3P\r");
  if (ReadAnswer(z_temp)!=0)
    return(-1);
  return(0);
}
/*=========================================================================
*/


/*** Function make_good_filname: This function adds an extension to     ***/
/*** the DOS filename pointed to by ptr. The extension added is passed  ***/
/*** to this function as well.                                          ***/
void make_good_filename(ptr,extension)
char *ptr,*extension;
{
char *dot_ptr;

    /* Find out if there is a '.' in the filename already */
    dot_ptr = strchr(ptr,'.');
    /* If not, find end of string */
    if (dot_ptr == NULL)
      for(dot_ptr = ptr;*dot_ptr != '\0';dot_ptr++);
    else
      return;
    /* No extension, so put the extension on the end of this string */
    *dot_ptr = '\0';
    strcat(ptr,extension);
}
/*=======================================================================*/
```

```
/*** This function clears the keyboard type-ahead buffer prior to reading ***/
/*** a single character from the keyboard. The keyboard input routine is  ***/
/*** interrupt 0x21, function 0x08 which waits for a key to be entered.  ***/
/*** Return value:    int        the character entered              ***/

int clrkbd(void)
{
  union REGS ireg;

  ireg.h.ah = 0x0c;          /* Function 12                */
  ireg.h.al = 0x08;          /* Prepare for function 8        */
  intdos(&ireg, &ireg);

  return ireg.h.al;
}


/***                  Function: getint()                    ***/
/*** This function accepts input from the keyboard to form an integer    ***/
/*** number. When the input is entered, the value is converted to a long  ***/
/*** integer and checked against the symbolic constants as defined in the ***/
/*** (ANSI) limits.h header file. If the value entered does not fall     ***/
/*** within the limits as defined in the header file, an error condition  ***/
/*** is returned.                                          ***/
/*** Argument list:   int *ptr    a pointer to the integer that will   ***/
/***                    hold the integer value on success    ***/
/*** Return value:    int        0 on error, 1 if successful        ***/
int getint(int *ptr)
{
  char buff[10];
  long tempint;

  gets(buff);
  tempint = atol(buff);
  if ( (tempint > (long) INT_MAX) || (tempint < INT_MIN) ) {
    *ptr = 0;
    return 0;
  } else {
    *ptr = (int) tempint;
    return 1;
  }
}

/*** This function accepts input from the keyboard to form an unisgned   ***/
/*** long number.                                          ***/
/*** Argument list:   unsigned long *ptr  a pointer to the ulong that   ***/
/***                  will hold the ulong value on success    ***/
/*** Return value:    int        0 on error, 1 if successful        ***/
int getulong(unsigned long *ptr)
```

```
{
  char buff[20];
  unsigned long templong;

  gets(buff);
  templong = (unsigned long)atol(buff);
  if ((templong > (unsigned long) ULONG_MAX))
  {
    *ptr = 0L;
    return 0;
  }
  else
  {
    *ptr = templong;
    return 1;
  }
}

/*** This function accepts input from the keyboard to form a floating    ***/
/*** pointg number.                                                      ***/
/*** Argument list:   float *ptr  a pointer to the float that          ***/
/***                   will hold the float value on success  ***/
/*** Return value:    int        0 on error, 1 if successful          ***/
int getfloat(float *ptr)
{
  char buff[40];
  double tempfloat;

  gets(buff);
  tempfloat = atof(buff);
  if ((tempfloat > (double)(1.0E20)) || (tempfloat < (double)0.0))
  {
    *ptr = 0.0;
    return 0;
  }
  else
  {
    *ptr = (float) tempfloat;
    return 1;
  }
}

/***              Function: newkbhit()                              ***/
/*** This function reads a character from the keyboard--if one is ready-- ***/
/*** using interrupt 0x21. If not, the function returns zero. If a char  ***/
/*** is read, it is not echoed to the screen. If an extended key code is ***/
/*** read the value returned is the scan code plus 0x100. If an ASCII    ***/
/*** Return value:    int       0        -- no key pressed          ***/
/***                    1 < c < 128 -- ASCII character          ***/
/***                    c > 256    -- Extended key code         ***/
```

```
int newkbhit(void)
{
  int zflag;
  union REGS ireg;

  ireg.h.ah = 0x06;        /* Function 6                  */
  ireg.h.dl = 0xff;        /* We want to read it, not output  */
  zflag = intdos(&ireg, &ireg);

  if ( (zflag & ZEROFLAG) == 0) {  /* A character?          */
    if (ireg.h.al == 0) {   /* Extended keycode?          */
      ireg.h.ah = 0x06;
      ireg.h.dl = 0xff;
      intdos(&ireg, &ireg);
      return (ireg.h.al + 0x100);
    }
    return ireg.h.al;      /* An ASCII character          */
  }
  return 0;              /* No character                */
}

/***        Function name:  insure_legal_file_name          ***/
/*** This function just does a trial file open on the file name   ***/
/*** contained in the global fileName char string.  It gives the  ***/
/*** user additional chances to type in a legal DOS file name     ***/
/*** so that when the data acquisition is finished, we are assured ***/
/*** that the data will be stored off into a file before the      ***/
/*** program terminates and the data is lost!!!                   ***/
void insure_legal_file_name(void)
{
  FILE *testfile;

  do
  {
    if ((testfile = fopen(fileName, "w")) == NULL)
    {
      printf("***************** WARNING WARNING **************\n");
      printf("fopen of output file %s failed!!\n", fileName);
      printf("There are probably some illegal chars. in the \n");
      printf("output file name--Please re-enter the file \n");
      printf("name now so that no data will be lost !!! ===>");
      gets(fileName);
    }
  } while (testfile == NULL);

  fclose(testfile);

} /*** insure_legal_file_name ***/
```

```
/*** Function:  input_energy_bins   Uses global array ene_bins[][] ***/
/*** and global variables slope and intercept (which define the    ***/
/*** calibration curve for the mca. Gives the user the chance to   ***/
/*** change the default values for the slope and intercept. This   ***/
/*** function then prompts the user for the number of energy bins he ***/
/*** or she wishes to specify as well as for the starting and ending ***/
/*** energy values for each of these bins.                         ***/
void input_energy_bins(void)
{
  register int  i, j;

  char  answer[3];

  mult_bins_flag = TRUE;

  window(2,20,44,24);
  gotoxy(1,1);
  textcolor(BLACK);
  textbackground(WHITE);

  for (i=0; i<MAX_BINS; i++)
  {
    eLimits[i][0] = ULONG_MAX;
    eLimits[i][1] = 0;
  }

  cprintf("\nDo you wish to change the default");
  cprintf(" calibration slope = %f and", slope);
  cprintf(" intercept = %f ? Enter y or n for  ");
  cprintf("yes or no ==>", intercept);
  gets(answer);
  if (strncmpi(answer, "y\0", 1) == 0)
  {
    cprintf("Please enter the new slope and intercept ==>");
    scanf("%f %f", &slope, &intercept);
  }

  do
  {
    cprintf("How many energy bins do you wish");
    cprintf(" to specify? ==>");
    scanf("%d", &num_bins);
    cprintf("\n");
  } while(num_bins>10);

  for (i=0; i<num_bins; i++)
  {
    cprintf("Enter the beginning and ending");
    cprintf(" energies for bin #%d ==>", i+1);
    scanf("%d %d", &ene_bins[i][0], &ene_bins[i][1]);
```

```
    /*** Convert these energies to channel numbers using the current ***/
    /*** calibration slope and intercept for the mca.              ***/
    ene_bins[i][0] = (int)((ene_bins[i][0] - intercept)/slope);
    ene_bins[i][1] = (int)((ene_bins[i][1] - intercept)/slope);
  }
} /*** End of input_energy_bins ***/
```

MOVERF.C MODULE LISTING

```
/* ---------------------- PC23 CONTROL ---------------------- */
/* This module contains low level control routines for the   */
/* PC23 board. Each function contained here was adapted from the */
/* Pascal source code in the PC23 instruction manual. They are: */
/*                                                           */
/* Initialize(board)     (reset the PC23)        */
/* WriteChar(board, char)    (write a char to PC23)    */
/* WriteCmd(board,string)    (write string to PC23)    */
/* ReadChar(board)       (read a PC23 character)   */
/* ReadAnswer(address,string)  (read a PC23 ans. string) */
/*                                                           */
/* This module should be included in any C program needing   */
/* to control the PC23.                  */
/*                                                           */
/* ---------------------------------------------------------- */



#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <time.h>
#include "mover.h"


/* ------------------------ INITIALIZE ------------------------ */
/*                                                              */
/* Initialize the PC23 residing at the address 'board'. A return */
/* value of -1 indicates an error resulted and the program should stop. */
/*                                                           */
/* ---------------------------------------------------------- */
int Initialize(void)
/*** struct Board_struct *board; ***/
{
int       count=0;
unsigned char status_byte;

  board->Control = board->base + 1;
  board->Status  = board->base + 1;
  board->Command = board->base;
  board->Data    = board->base;
  outportb((short) board->Control,STOP);

  do
    status_byte = inportb((short)board->Status);
  while( (status_byte & FAIL_MASK) == 0);
```

```
  outportb((short)board->Control,CB);
  outportb((short)board->Control,START);

  do
  {
    status_byte = inportb((short)board->Status);
    count++;
    delay(1.0);
  } while( ((status_byte & START_MASK) != RESTART) && (count < MAXINT) );


  if (count == MAXINT)
    return(ERROR_CONDITION);


  outportb((short)board->Control,INTCLR);
  for(count=0;count<MAXINT;count++);  /* delay for a bit */

  outportb((short)board->Control,CONTROL_BYTE);

  return(100);
} /* End of Initialize */



/* ---------------------------- WRITECHAR ---------------------------- */
/*                                                                      */
/*  This function writes a single character to the PC23 board.          */
/*                                                                      */
/* -------------------------------------------------------------------- */
void WriteChar(alpha)
/*** struct Board_struct *board; ***/
char    alpha;
{
unsigned char status_byte;

  do
    status_byte = inportb((short)board->Status);
  while( (status_byte & IDB_MASK) == 0);
  outportb((short)board->Command,alpha);
  outportb((short)board->Control,IDB);
  do
    status_byte = inportb((short)board->Status);
  while ( (status_byte & IDB_MASK) != 0);
  outportb((short)board->Control,CB);
  do
    status_byte = inportb((short)board->Status);
  while ( (status_byte & IDB_MASK) == 0);
```

```
} /* End of WriteChar() */



/* ---------------------------- WRITECMD ---------------------------- */
/*                                                                    */
/* This function writes a string of characters to the PC23 board.  */
/* It is dependent on the WriteChar function.                         */
/*                                                                    */
/* ------------------------------------------------------------------ */
void WriteCmd(cmd)
/*** struct Board_struct *board; ***/
char    *cmd;
{
int  i;

  WriteChar(32);
  for(i=0;*(cmd + i) != 0;i++)
    WriteChar(*(cmd + i) );
  WriteChar(13);
} /* End of WriteCmd() */


/* ---------------------------- READCHAR ---------------------------- */
/*                                                                    */
/* This function reads a character in from the PC23 and returns it */
/* as the value of the function.                                      */
/*                                                                    */
/* ------------------------------------------------------------------ */
char ReadChar(void)
/*** struct Board_struct *board; ***/
{
unsigned char status_byte,return_byte=0;
time_t   start_time, /* Used to calculate timeout */
       present_time;

  status_byte = inportb((short)board->Status);
  if ( (status_byte & ODB_MASK) != 0)
  {
    start_time = time(NULL);
    present_time = time(NULL);
    do
    {
            status_byte = inportb((short)board->Status);
      present_time = time(NULL);
    } while(((status_byte & ODB_MASK)==0) &&
           (difftime(present_time,start_time)<30.0));
    if (difftime(present_time, start_time)>=30.0)
```

```
      return(-1);
           return_byte = inportb((short)board->Data);
         outportb((short)board->Control,ACK);
    start_time = time(NULL);
    present_time = time(NULL);
    do
    {
      status_byte = inportb((short)board->Status);
      present_time = time(NULL);
    } while(((status_byte & ODB_MASK)!=0)&&
              (difftime(present_time, start_time)<30.0));
    if (difftime(present_time, start_time)>=30.0)
      return(-1);
    outportb((short)board->Control,CB);
  }
         return(return_byte);
} /* End of ReadChar() */



/* ------------------------ READANSWER ------------------------ */
/*                                                               */
/*  This function reads a string of characters from the PC23.  It  */
/* relies on ReadChar() for this.                                */
/*                                                               */
/* ------------------------------------------------------------- */
int ReadAnswer(answer_string)
/*** struct Board_struct *board; ***/
char  *answer_string;
{
unsigned char status_byte,ascii;
char        *ptr = answer_string;
time_t   start_time, /* Used to calculate timeout */
       present_time;

  status_byte = inportb((short)board->Status);
  if (status_byte & ODB_MASK != 0)
  {
    start_time = time(NULL);
    present_time = time(NULL);
    do
    {
      ascii = ReadChar();
      if (ascii != 0)
        *(ptr++) = ascii;
      if (ascii == -1)
        printf("******* ERROR: ReadChar time out!!!! ******\n");
      present_time = time(NULL);
    }
```

```
    while((ascii != 13)&&(difftime(present_time, start_time)<300.0));
    if (difftime(present_time, start_time)>=300.0)
      return(-1);
    *ptr = 0;
  }
  return(0);
} /* End of ReadAnswer() */
```

Moverf.c module listing:

```
/* ---------------------- PC23 CONTROL ---------------------- */
/* This module contains low level control routines for the   */
/* PC23 board. Each function contained here was adapted from the */
/* Pascal source code in the PC23 instruction manual. They are: */
/*                                                           */
/* Initialize(board)     (reset the PC23)        */
/* WriteChar(board, char)     (write a char to PC23)    */
/* WriteCmd(board,string)     (write string to PC23)    */
/* ReadChar(board)       (read a PC23 character)   */
/* ReadAnswer(address,string)  (read a PC23 ans. string) */
/*                                                           */
/* This module should be included in any C program needing   */
/* to control the PC23.                                      */
/*                                                           */
/* ------------------------------------------------------------ */



#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <time.h>
#include "mover.h"


/* ------------------------ INITIALIZE ------------------------ */
/*                                                           */
/* Initialize the PC23 residing at the address 'board'. A return */
/* value of -1 indicates an error resulted and the program should stop. */
/*                                                           */
/* ------------------------------------------------------------ */
int Initialize(void)
/*** struct Board_struct *board; ***/
{
int       count=0;
unsigned char status_byte;

  board->Control = board->base + 1;
  board->Status  = board->base + 1;
  board->Command = board->base;
  board->Data    = board->base;
  outportb((short) board->Control,STOP);

  do
    status_byte = inportb((short)board->Status);
  while( (status_byte & FAIL_MASK) == 0);
```

```
  outportb((short)board->Control,CB);
  outportb((short)board->Control,START);

  do
  {
    status_byte = inportb((short)board->Status);
    count++;
    delay(1.0);
  } while( ((status_byte & START_MASK) != RESTART) && (count < MAXINT) );


  if (count == MAXINT)
    return(ERROR_CONDITION);


  outportb((short)board->Control,INTCLR);
  for(count=0;count<MAXINT;count++);  /* delay for a bit */

  outportb((short)board->Control,CONTROL_BYTE);

  return(100);
} /* End of Initialize */



/* ------------------------ WRITECHAR ------------------------ */
/*                                                            */
/*  This function writes a single character to the PC23 board.  */
/*                                                            */
/* ---------------------------------------------------------- */
void WriteChar(alpha)
/*** struct Board_struct *board; ***/
char    alpha;
{
unsigned char status_byte;

  do
    status_byte = inportb((short)board->Status);
  while( (status_byte & IDB_MASK) == 0);
  outportb((short)board->Command,alpha);
  outportb((short)board->Control,IDB);
  do
    status_byte = inportb((short)board->Status);
  while ( (status_byte & IDB_MASK) != 0);
  outportb((short)board->Control,CB);
  do
    status_byte = inportb((short)board->Status);
  while ( (status_byte & IDB_MASK) == 0);
```

```
} /* End of WriteChar() */



/* ---------------------------- WRITECMD ---------------------------- */
/*                                                                     */
/* This function writes a string of characters to the PC23 board.  */
/* It is dependent on the WriteChar function.                          */
/*                                                                     */
/* ------------------------------------------------------------------- */
void WriteCmd(cmd)
/*** struct Board_struct *board; ***/
char    *cmd;
{
int  i;

  WriteChar(32);
  for(i=0;*(cmd + i) != 0;i++)
    WriteChar(*(cmd + i) );
  WriteChar(13);
} /* End of WriteCmd() */


/* ------------------------ READCHAR ------------------------ */
/*                                                            */
/* This function reads a character in from the PC23 and returns it */
/* as the value of the function.                              */
/*                                                            */
/* ---------------------------------------------------------- */
char ReadChar(void)
/*** struct Board_struct *board; ***/
{
unsigned char status_byte,return_byte=0;
time_t   start_time, /* Used to calculate timeout */
       present_time;

  status_byte = inportb((short)board->Status);
  if ( (status_byte & ODB_MASK) != 0)
  {
    start_time = time(NULL);
    present_time = time(NULL);
    do
    {
            status_byte = inportb((short)board->Status);
      present_time = time(NULL);
    } while(((status_byte & ODB_MASK)==0) &&
          (difftime(present_time,start_time)<30.0));
    if (difftime(present_time, start_time)>=30.0)
```

```
      return(-1);
           return_byte = inportb((short)board->Data);
         outportb((short)board->Control,ACK);
    start_time = time(NULL);
    present_time = time(NULL);
    do
    {
      status_byte = inportb((short)board->Status);
      present_time = time(NULL);
    } while(((status_byte & ODB_MASK)!=0)&&
              (difftime(present_time, start_time)<30.0));
    if (difftime(present_time, start_time)>=30.0)
      return(-1);
    outportb((short)board->Control,CB);
  }
        return(return_byte);
} /* End of ReadChar() */



/* ------------------------- READANSWER ------------------------- */
/*                                                                 */
/*  This function reads a string of characters from the PC23.  It  */
/*  relies on ReadChar() for this.                                 */
/*                                                                 */
/* ------------------------------------------------------------- */
int ReadAnswer(answer_string)
/*** struct Board_struct *board; ***/
char  *answer_string;
{
unsigned char status_byte,ascii;
char        *ptr = answer_string;
time_t   start_time, /* Used to calculate timeout */
       present_time;

  status_byte = inportb((short)board->Status);
  if (status_byte & ODB_MASK != 0)
  {
    start_time = time(NULL);
    present_time = time(NULL);
    do
    {
      ascii = ReadChar();
      if (ascii != 0)
        *(ptr++) = ascii;
      if (ascii == -1)
        printf("******* ERROR: ReadChar time out!!!! ******\n");
      present_time = time(NULL);
    }
```

```
    while((ascii != 13)&&(difftime(present_time, start_time)<300.0));
    if (difftime(present_time, start_time)>=300.0)
      return(-1);
    *ptr = 0;
  }
  return(0);
} /* End of ReadAnswer() */
```

APPENDIX E. HEADER FILES.

acq.h header file:

```
/* Header File - acq.h */
/* Header File for X-Ray User Interface Program */

/* Peter Jeong and Rick Powell */
/* 01/09/1991 */
/* CNDE, ISU */

#define TRUE 1
#define FALSE 0
#define MAX_BINS 10 /*** NOTE: if this value is increased, you may ***/
                    /*** need to increase the number of file buffers ***/
                    /*** in the config.sys file and reboot.        ***/
#define POINTS_PER_WRITE 50 /*** number of scan points before write ***/
#define DEAD_TIME_PER_ACQ 3.0/60.0 /* 3.0 sec dead time / 60 secs per min */
#define INCHES_PER_STEP 0.00001
#define DEGREES_PER_STEP 0.0001
#define DEGREES_PER_RADIAN 57.295828
#define X_AXIS "2\0"
#define Z_AXIS "1\0"
#define THETA_AXIS "3\0"
#define VEL_DEFAULT "V2\0"
#define ACCEL_DEFAULT "A0.2\0"
#define MOTOR_RESOLUTION "MR20\0"
#define DISTANCE "D\0"
#define DIR_DEF "H\0"
#define GO "G\0"
#define BASEADD 0xD0000000L
#define MASK 0x00FFFFFFL
#define BEGCHAN 250 /*** default beginning channel number ***/
#define ENDCHAN 1950 /*** default ending channel number ***/
#define MAXCHAN 2047
#define DATAMCB 2
#define PC23BASE 0x330
#define REAL_TIME_ADJUST 0.1
#define ZEROFLAG 0x40
#define OUTFLG    0xD0000003L /* Segment:Offset D000:0003 message flag */
#define TEST      0xD00000F3L /* Segment:Offset D000:00F3 test to mca */
#define OUTLENLO  0xD000003BL /* Segment:Offset D000:003B mess. to mca */
#define OUTLENHI  0xD000003FL /* Segment:Offset D000:003F mess. length */
#define OUTBUF    0xD0000043L /* Segment:Offset D000:0043 m. from mca */
#define INFLG     0xD00007C3L /* Segment:Offset D000:07C3 mess. length */
#define INLENLO   0xD00007FBL /* Segment:Offset D000:07FB message flag */
#define INLENHI   0xD00007FFL /* Segment:Offset D000:07FF message flag */
#define INBUF     0xD0000803L /* Segment:Offset D000:0803 input buffer */
#define TRUED '\377'
```

```
#define  FALSED '\0'

#define L_ARRO 75
#define R_ARRO 77
#define U_ARRO 72
#define D_ARRO 80
#define INSERT 82
#define DEL 83
#define ESC 27
#define ENTER 13

/* FUNCTION PROTOTYPES */

/* Format.c */

extern void  top_pad( void );
extern void  Multy_1( int, int, int, int );
extern void MltPad_1( int, int, int, int );
extern void  Multy_2( int, int, int, int );
extern void MltPad_2( int, int, int, int );
extern void  Single( int, int, int );
extern void SinglPad( int, int, int );
extern void   Scr_1( int, int, int );
extern void ScrPad_1( int, int, int );
extern void   Scr_2( int, int, int );
extern void ScrPad_2( int, int, int );

/* level_1a.c */

extern void disp_1(void);
extern void word_1a( int, int, int );
extern void word_1b( int, int, int, int );
extern void dot(void);

/* level_2a.c */

/* level_2b.c */

extern void act_2b(void);
extern void disp_2b(void);
extern void word_2b1( int, int );
extern void word_2b2( int, int, int );

/* level_2c.c */

extern void act_2c(void);
extern void disp_2c(void);
extern void word_2c1( int, int );
extern void word_2c2( int, int, int );
```

```
/* level_2d.c */

extern void act_2d(void);
extern void disp_2d(void);
extern void word_2d1( int, int );
extern void word_2d2( int, int, int );

/* level_2e.c */

extern void act_2e(void);
extern void disp_2e(void);
extern void word_2e1( int, int );
extern void word_2e2( int, int, int );

/* level_3a.c */

extern void act_3a(void);
extern void disp_3a(void);
extern void word_3a1( int, int );
extern void word_3a2( int, int, int );

/* level_3b.c */

extern void act_3b1(void);
extern void act_3b2(void);
extern void act_3b3(void);

/* level_3c.c */

extern void act_3c1(void);
extern void act_3c2(void);
extern void act_3c3(void);

/* level_3d.c */

extern void act_3d1(void);
extern void act_3d2(void);
extern void act_3d3(void);
extern void act_3d4(void);
extern void disp_3d4(void);
extern void word_3d4a( int, int );
extern void word_3d4b( int, int, int );
extern void act_4d4(void);
extern void act_3d5(void);

/* level_3e.c */

extern void act_3e1(void);
extern void act_3e2(void);
extern void act_3e3(void);
```

```
/* level_4a.c */

extern void act_4a(void);
extern void disp_4a(void);
extern void disp_4b(void);
extern void word_4aa(void);
extern void word_4ab( int, int, int, int );
extern void word_4ac( int, int, int, int );
extern void reset(void);

/* level_4b.c */

extern void act_4b(void);
extern void disp_4b(void);
extern void disp_4b(void);
extern void word_4ba(void);
extern void word_4bb( int, int, int, int );
extern void word_4bc( int, int, int, int );



/* GLOBAL VARIABLES */

/* acqray.c */



/* format.c */

extern void *buff_w1;
extern void *buff_p1;
extern void *buff_w2;
extern void *buff_p2;
extern void *buff_w3;
extern void *buff_p3;
extern void *buff_w4;
extern void *buff_p4;
extern unsigned buff_size;

/* set-up */

extern int  dotpos[8][5];
extern float param[8][5];

/* x1_srt, x1_end, x1_inc, x1_vel, x1_acl
  y1_srt, y1_end, y1_inc, y1_vel, y1_acl
  z1_srt, z1_end, z1_inc, z1_vel, z1_acl
  t1_srt, t1_end, t1_inc, t1_vel, t1_acl
  p1_srt, p1_end, p1_inc, p1_vel, p1_acl
```

```
  x2_srt, x2_end, x2_inc, x2_vel, x2_acl
  y2_srt, y2_end, y2_inc, y2_vel, y2_acl
  t2_srt, t2_end, t2_inc, t2_vel, t2_acl */

/* raster-1d */

extern int r1d_disp;
extern float r1d_ctime;

/* raster-2d */

extern int r2d_disp;
extern float r2d_ctime;

/* tomography */

extern float tomo_ctime;
extern float tomo_thd;
extern int   tomo_engy;
extern int   tomo_disp;

/* MFB */

extern int micro_disp;
extern float micro_ctime;

/* Other Variables */

extern int _wscroll;
extern int directvideo;

extern int code;
extern int xbox,xtemp;
extern int sbox,stemp;
extern int tbox,ttemp;
extern int fbox,ftemp;
extern int gbox,gtemp;
extern int pbox,ptemp;
extern int kbox,ktemp;

extern int left_2b,top_2b;
extern int left_2c,top_2c;
extern int left_2d,top_2d;
extern int left_2e,top_2e;

extern int left_3a,top_3a;

extern int left_3b1,top_3b1;
extern int left_3b2,top_3b2;
extern int left_3b3,top_3b3;
```

```
extern int left_3c1,top_3c1;
extern int left_3c2,top_3c2;
extern int left_3c3,top_3c3;

extern int left_3d1,top_3d1;
extern int left_3d2,top_3d2;
extern int left_3d3,top_3d3;
extern int left_3d4,top_3d4;
extern int left_3d5,top_3d5;
extern int left_4d4,top_4d4;

extern int left_3e1,top_3e1;
extern int left_3e2,top_3e2;
extern int left_3e3,top_3e3;

extern int left_4a,top_4a;
extern int left_4b,top_4b;

/* 4th level */

extern int d_srt;
extern int d_end;
extern int ch_srt;
extern int ch_end;


/* ARRAYS */

/* 1st level */

extern char *item1a[ 8];
extern char *item1b[11];
extern char *item1c[11];
extern char *item1d[12];
extern char *item1e[ 5];
extern char *item1f[ 7];
extern char *item1g[ 6];
extern char  **list1[7];
extern int    light1[7];
extern int     num1[7];
extern int   pos1[7][2];

/* 2nd level */

extern char *item2a[9]; /* Set-up */
extern char *item2b[6]; /* Raster-1D */
extern char *item2c[6]; /* Raster-2D */
extern char *item2d[8]; /* Tomography Scan */
extern char *item2e[6]; /* MFB (Micro-Focused Beam Scan) */
```

```
extern char **list2[5];
extern int   npts2[5];

/* 3rd level */

extern char r1d[81];
extern char r2d[81];
extern char tomo[81];
extern char *engy[3];
extern char *chan[3];
extern char mfb[81];
extern int axis_status[4][8];

/* 4th level */

extern char *item4a[9];
```

mover.h header file:

```
/* File: mover.h; Header file for PC23 indexer board  */
/* Contains numerous definitions for the indexer board */
#include <math.h>

#define byte unsigned char
#define FAIL_MASK   0x20  /* Board Failure mask used to read SB5          */
#define START_MASK  0x7F  /* Recovery mask used to test for PC23 recovery  */
#define INTCLR      0x20  /* Control Byte clears the interrupt output latch */
#define RESTART     0x17  /* Used to verify presence/recovery             */
#define STOP        0x64  /* Control byte used to stop timer, set CB2     */
#define START       0x40  /* Control byte used to restart, clear CB5      */
#define CONTROL_BYTE 0x60  /* Normal control byte, bits 5 and 6 set       */
#define IDB_MASK    0x10  /* Mask used to read SB4                        */
#define IDB         0x70  /* Control byte used to set CB4                 */
#define CB          0x60  /* Normal control byte config.. bits 5 & 6 set  */
#define ODB_MASK    0x08  /* Mask used to read CB3                        */
#define ACK         0xE0  /* Control byte used to set CB7                 */
#define ERROR_CONDITION (int)(-1)  /* Indicates an error condition         */
#define MAXINT 32767
#ifndef NULL
  #define NULL 0
#endif
#define DISPLAY_WIDTH 80         /* Width of display screen in characters */

#define CURSOR_UP 0x48
#define CURSOR_DOWN 0x50
#define CURSOR_LEFT 0x4b
#define CURSOR_RIGHT 0x4d
#define PAGE_UP 73
#define HOME 71
#define END 79
#define PAGE_DOWN 81
#define DELETE 83
#define F1_KEY 59
#define F2_KEY 60
#define F3_KEY 61
#define F4_KEY 62
#define F5_KEY 63
#define F6_KEY 64
#define F7_KEY 65
#define F8_KEY 66
#define F9_KEY 67
#define F10_KEY 68
#define BACKSPACE 8

#define NORMAL  "\x1B[0m"
```

```
#define BOLD   "\x1B[1m"
#define UNDER  "\x1B[4m"
#define REVERSE "\x1B[7m"
#define CLS printf("%c[2J",ESC);
#define SPACE  32
#define CR     13
#define POS_CURSOR "\x1B["

struct Board_struct  {
        long base;
        long Command;
        long Control;
        long Status;
        long Data;
        };

extern struct Board_struct *board;

char *report_pos(struct Board_struct *,short);
int  Initialize(void);
int get_key();
char ReadChar(void);
void WriteCmd(char *);
int ReadAnswer(char *);
```