Optimization of Data Acquisition System for Novel DNA Sequencing Instrument

by

Jamie Song

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by. .....
Paul T. Matsudaira
Professor
Thesis Supervisor

Accepted by .. ..
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Optimization of Data Acquisition System for Novel DNA Sequencing Instrument

by

Jamie Song

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

For this thesis, I designed and implemented a program that controls data acquisition in a novel DNA sequencing technology and coordinates several peripheral components crucial to the operation of the sequencing machine. The module for data acquisition control is separate from the peripheral device control. Both software modules were designed with an object-oriented model to allow for modules providing additional processing capabilities and new peripheral instrument control to be added. All of the control software and parts of the data acquisition software were written with Visual C++. Other portions of the data acquisition software were written in assembly language.

Thesis Supervisor: Paul T. Matsudaira
Title: Professor

Acknowledgments

I would like to thank the entire Ehrlich/Matsudaira lab for the opportunity work with such a great group of people. Thank you, Paul and Dan, for providing me with the opportunity. Thank you, Aram, for all of the countless ways you were a mentor to me. Thank you, Sameh, for your patience and graciousness. I'd also like to thank all of the people who helped me persevere through these long, five years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 Human Genome Project

Improving methods of DNA sequencing has become a significant focus in biotechnology, especially with the effort to sequence the human genome with the Human Genome Project (HGP). The HGP is jointly coordinated by the National Institutes of Health (NIH) and the Department of Energy (DOE). The DOE and NIH initially presented a plan of attack in 1990 and have since revised and updated their goals. The original goals included sequencing all the base pairs of the human genome by the year 2005 or earlier. The HGP is currently on schedule to meet its goals. However, the rate at which DNA can be sequenced has often been cited as a limiting factor. The challenge, now, of sequencing the genome is "largely one of doing the job cheaper and faster" [2]. Added to the existing pressure of reaching the established goals, biologist Craig Venter announced in 1998 his plan to finish sequencing the entire genome in 3 years, in 2001, 4 years ahead of the HGP schedule. [3]

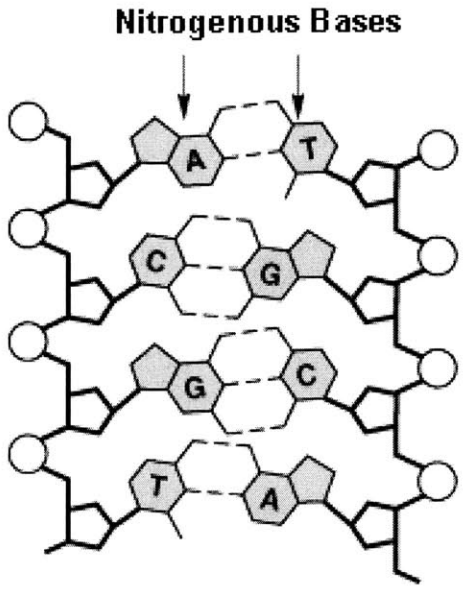

Figure 1-1: Structure of DNA. Dotted lines represent weak bonds that form between two complimentary bases. Pentagons and circles on the outer edges represent the sugar and phosphate backbone of the DNA ladder.

1.1.2 The Science

Basic DNA Structure

The human genome consists of all the genetic material in a human cell. It is structurally divided into 23 chromosomes. Each chromosome can contain hundreds of genes, the functional units of heredity. Genes are segments of genetic material that prescribe the construction of a single type of protein. The basic building block of these genes are nucleotides. Nucleotides are molecules consisting of a sugar, a phosphate, and a nitrogenous base. There are 4 types of nucleotide bases: adenine (A), thymine (T), cytosine (C), and guanine (G). These nucleotide bases are pieced together linearly to form deoxyribonucleic acid (DNA) strands.

The two DNA strands are bonded together in a form of a ladder, as shown in Figure 1-1. Each nucleotide on one strand makes a weak chemical bond with a complimentary nucleotide on the second strand. In essence, one strand is simply the compliment of the other and holds no extra genetic information. The chemical bonds

that form are very specific: A only bonds with T and C only bonds with G. These bonded bases constitute a base pair. The human genome contains about 3 billion of these base pairs. The two DNA strands twist to form a double helix. [1]

Sequencing

The current technique for sequencing DNA is to shear many copies of the same DNA segment into fragments of various lengths and to tag the ends of each fragment with fluorescent dyes (a different dye for each base: A, T, C, and G). The result is a sample containing numerous DNA segments, beginning at the same location and ending at different locations. The sample of DNA fragments is loaded onto one end of a gel (generally agarose or acrylamide). The sample undergoes a process called electrophoresis, in which voltage is applied across the gel and, due to the negative electrical charge of the DNA, the DNA fragments migrate toward the positively charged end of the gel. Smaller fragments in the sample travel faster and the longer fragments travel more slowly. This results in a separation of the fragments in the gel based on size, with the shorter fragments reaching the end first.

A high degree of separation must be achieved from the electrophoresis to obtain sequence data. Electrophoresis can be performed at various voltages and for various time durations. Controlling these two factors determines the resolution of the DNA fragment separation. To obtain sequence from the electrophoresis, we must achieve a resolution that allows us to distinguish fragment lengths that differ by a single base.

The migrating DNA fragments are detected at the positively charged end of the gel. The DNA fragments pass through the detection area, which is continuously scanned by a laser beam. The fluorescent dyes attached to each fragment are excited by the laser and the resulting fluorescence is detected and read. Identifying these base tags in succession, under the proper resolution, allows us to reconstruct the location of each base and, consequently, construct the sequence.

1.1.3 Sequencing Instrumentation

Existing Instrumentation

The current types of sequencing instruments are slab gels, ultra-thin slab gels, and capillary machines. Until the mid to late 1990's, slab gels were the predominant type of instrument used in sequencing. Some of the major sequencing centers in the world for the HGP, Washington University, The Sanger Centre and The Whitehead Institute Center for Genome Research all have used the slab gel instrument of choice, the Perkin Elmer ABI (Applied Biosystems) 377 slab gel apparatus. The current slab gels can hold at maximum 96 lanes in one gel and take 4 to 6 hours to complete one run.

Ultra-thin slab gels are also available, which are less than 0.1 mm thick. Electrophoresis can be performed at higher temperatures on thinner slab gels and therefore, result in faster fragment length separations. This technology however, did not gain popularity.

One of the drawbacks of slab gels is that the lanes are not clearly delineated. Since samples migrate down one continuous piece of gel, it is the job of the software to determine where one sample begins and the other ends.

The capillary machine is the most popular of the existing types of instruments. Instead of a gel, each sample is run through a quartz tubule of approximately 80 μm in diameter. Each sample is loaded into an individual capillary and eliminates the need for software to ascertain lane delineation and sample delineation. Due to the small diameter of the capillary, less sample volume is required for electrophoresis to attain sequence results. The decreased sample volume allows electrophoresis to be performed at higher voltages resulting in faster run times. There are commercial machines currently available which can process 96 capillaries at a time. [4]

Microfabricated Instruments

A novel type of instrument, using glass microfabricated devices for DNA sequencing, promises throughput of up to 30 times that of the current slab gel instruments. This

	slab gel	capillary machine	microfabricated
# of lanes	96	96	384
run time	5 hrs	3hrs	80 mins
average read (bases)	800	600	700
throughput (million bases/day)	0.4	0.5	4.8

Table 1.1: Comparison of Types of Sequencing Instruments. This comparison shows the tremendous throughput increases of microfabricated instruments due to their greater number of lanes and significantly faster run times. (Sinclair, 1999).

form of sequencing runs each DNA sample through an isolated microchannel of 50 to 100 μm in diameter. Unlike capillary technology, which uses quartz to make flexible capillary strands, one can etch many of these capillaries into a glass plate, using microfabrication techniques similar to those used in the semiconductor industry. This process allows packing a larger number of capillaries into a compact space.

A single glass plate can hold up to 384 lanes. Due to the physical properties of the plate and the machine design itself, one is able to complete a run in a little over an hour and maintain a high level of data integrity. This allows achievement of a substantially higher sequencing throughput. Table 1.1 shows representative statistics for each type of instrument.

1.2 Objectives of Thesis

The laboratory of Paul Matsudaira at the Whitehead Institute and MIT is currently pursuing this new microfabricated sequencing instrumentation. In the midst of the collaborative effort of chemists, biologists, and mechanical engineers to create and refine a prototype instrument, the objectives of my thesis work were to design and construct data acquisition and control system software to operate the prototype instrument.

1.3 Organization of Thesis

In this introduction, I have presented basic background information for the science and instrumentation behind DNA sequencing which will aid in understanding this thesis. In Chapter 2, I present a description and specification of the instrument our laboratory has developed as well as the objectives of the software development. The rest of this thesis is focused on the three software modules developed: data acquisition, instrument control, and peripheral component control. In Chapters, 3, 4, and 5, I will discuss in depth the software requirements, design, and implementation for each of these software modules. Chapter 6 concludes this thesis with discussion of potential future work.

Chapter 2

Instrument Specifications

2.1 Hardware Specifications

2.1.1 Mechanical

Shown in Figure 2-1 is a schematic of the sequencing instrument and all of its separate components. The largest portion of the instrument is the element which holds the microfabricated chip. The microfabricated chip is a glass plate, of dimensions 0.11 cm thick by 25 cm wide by 50 cm long. The chip contains 384 microchannels, each about 40 μm deep and 90 μm wide. High voltage is applied across the length these channels to perform electrophoresis. Under the detection area of the chip is the spinner motor, with fluorescence detection optics, which is connected to the laser beam source. There is also a position encoder attached to the spinner motor which allows the spinner location to be read. (See Figure 2-2)

2.1.2 Optics

The optical portion of this instrument is an epifluorescent system. The laser beam projected through the spinner optics returns a fluorescence signal through a set of optical filters. These optical filters are arranged to separate the fluorescence signal into its 4 constituent colors, each corresponding to a different base dye. After the signal passes through the filters, it passes through a set of photomultiplier tubes

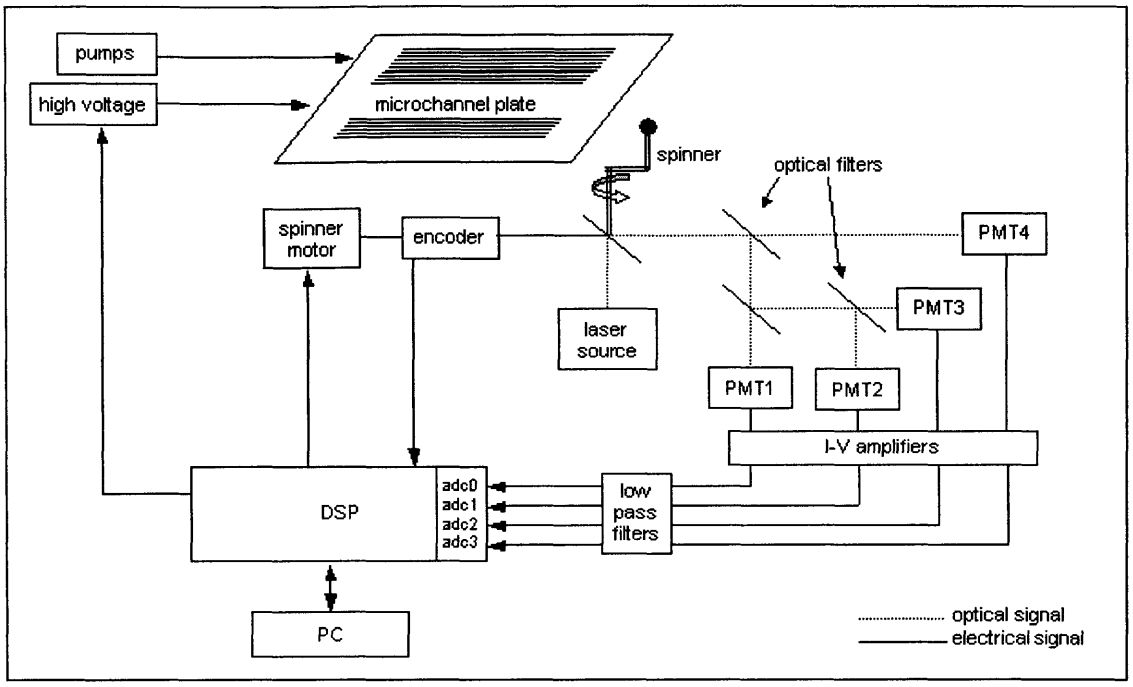

Figure 2-1: Instrument Diagram. The data is acquired through the spinner optics and passed through the optical filters, PMTs, amplifiers, and electronic filters as shown. The data is finally collected by the DSP through the ADCs, which all reside on the data acquisition board.

(PMTs) which process the signal. The PMTs convert the fluorescence to current, which is subsequently amplified to a voltage and passed through low pass electronic filters.

2.1.3 Digital Processing

The digital processing portion of the instrument includes the data acquisition board with a digital signal processor (DSP). The specific DSP used in this system is an Analog Devices ADSP-2181 processor, clocked at 32MHz. The data acquisition board also has analog to digital converters (ADCs) on board which sample the voltage that is produced by the low pass filters. The data acquisition board also has an EPP parallel port interface which allows it to send and receive information from a PC. The software to control the data acquisition board and collect the sequence data

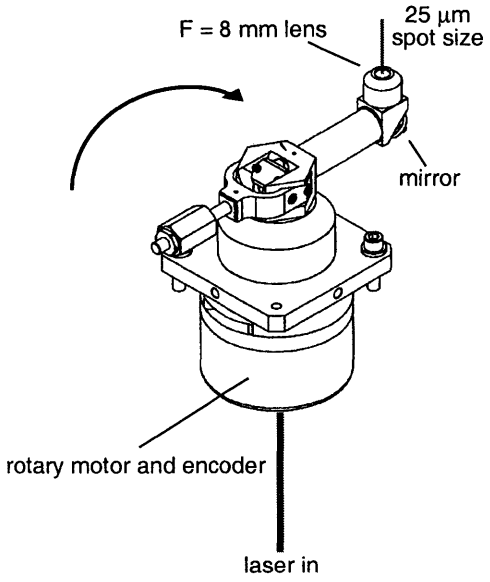

Figure 2-2: Spinner Motor and Encoder. This schematic of the spinner motor shows the mechanical design of the motor. The shaft which the laser enters houses the spinner optics. Various lenses and mirrors are used to retrieve the optical signal.

must reside on the PC.

2.1.4 Peripheral Components

The remaining peripheral components of the instrument are the high voltage supplies, pumps, valves, and the robotic arm. The custom built high voltage supplies control a voltage of up to 10 kV. A variety of peristaltic and syringe pumps supply buffer and gel to the chip. Valves control the direction of flow of buffers and gel and also the refilling of the pumps. The robotic arm is mobile in 3 orthogonal, linear axes and carries a custom built 96-tip pipetter.

2.1.5 Overall Function

The peripheral components are coordinated to prepare the instrument for an electrophoretic run. The DNA sample is loaded into a special injection area at one end of the glass plate by the robotic arm. The various pumps and valves coordinate to ensure that the gel is pumped into the channels and that there is enough buffer present

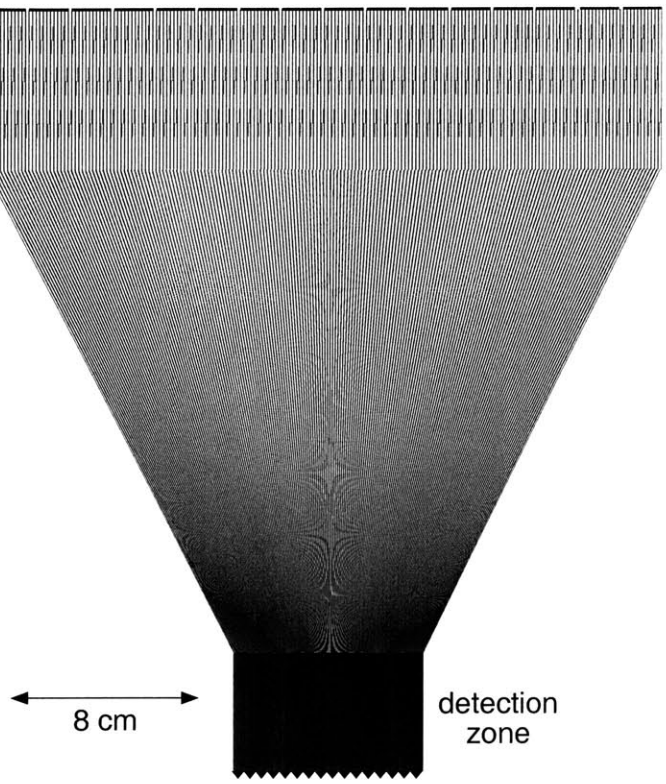

Figure 2-3: Schematic of 384-lane microfabricated glass plate (top view). The sample is loaded onto the plate at the upper portion of the plate. It migrates down the channels toward the detection area on the bottom.

in the microchannels such that the voltage can be applied across it. A voltage in the range of 6kV - 10kV is then applied across each channel to begin electrophoresis.

Once electrophoresis begins, the laser beam is activated and the spinner motor begins rotation. As the spinner motor rotates 360°, the laser beam sweeps across the detection area of the chip (See Figure 2-3). The DNA sample migrates towards the detection area as the laser continuously scans the area. As the laser intersects each of the channels, the fluorescent tagged DNA passing through the detection area is excited by the laser. The resulting signal passes through the optics as described in Section 2.1.2, and produces a voltage signal for each dye color.

The separation of the signal into different colors is in effect identifying the tagged base. Each base signal is separately sampled by the ADCs on the data acquisition board. The DSP collects and pre-processes the incoming data and stores it in DSP

memory. When the memory buffers are full on the DSP, it signals the PC to begin uploading data. The PC uploads and processes the raw data into a suitable format for basecalling software to decode the DNA sequence.

2.2 Software Specifications

2.2.1 Software Objectives

The ultimate goal of building this software system is to create a production machine for DNA sequencing that provides an increased throughput, measured in million bases per day (Mb/day). The specific goals for software development are to automate instrument operation, data collection, and processing of data to produce sequence.

Many of the tasks concerning the operation of the instrument can be automated and controlled by software. These tasks include sample loading and labeling, instrument status feedback (gel volume, laser life, voltage, etc.), instrument operation, peripheral component control, data acquisition, data pre-processing and basecalling. This thesis specifically deals with the data acquisition, instrument operation, and peripheral component control tasks in this system.

2.2.2 Software Modules

There are three software modules discussed in this thesis: the DSP software, the instrument control software, and the peripheral component control software. The DSP software is the software that resides inside the processor on the data acquisition board. This software provides the PC with access to the hardware of the instrument. Without this software, the application that runs on the PC could not control the instrument. The DSP software provides lower level coordination of the data acquisition and instrument control.

The instrument control software is the primary piece of software that handles all user interaction. It is the application that provides functionality to the user and allows them to control and manipulate the instrument. This application coordinates

data acquisition on the highest level.

The peripheral component software is the software that coordinates the actions of all the peripheral devices in the system. These are devices such as valves, pumps, high voltage supplies, and robot arms, which aid in the automation of the sequencing process. The peripheral component control software coordinates automation of all of these devices from a central application.

Chapter 3

DSP Software

3.1 Software Requirements

The software requirements for the digital signal processor (DSP) software include managing all the components that reside on the data acquisition board and managing data acquisition from the instrument. The components on the board include the high voltage supplies, spinner motor, position encoder, and analog to digital converters (ADCs).

3.1.1 Component Control

The DSP software needs to provide a variety of features to operate and control the instrument and its components. For the high voltage supplies, the software must be able to activate them, deactivate them, and set the voltage values. For the spinner motor, the software must control rotation, speed, and spinner location. For the position encoder, the software must be able to read the encoder and recalibrate it. For the ADCs, it must provide control of the sample rate and access to the data that is sampled.

3.1.2 Data Acquisition

The DSP software also needs to manage the data acquisition from the instrument. This specifically includes implementing the collection method, data pre-processing, and storage of data prior to upload.

A design consideration in creating the data acquisition module in the DSP software is the minimum data resolution require to sequence DNA. To sequence with an acceptable level of confidence, a minimum number of data points per base is required. The two variables involved in determining this minimum number of data points are the number of points per lane and the number of points per peak in the electropherogram.

A peak is an increase in signal that corresponds to the fluorescence detected from the DNA sample. When the sample runs down the length of the channels during electrophoresis, the fragments separate by length. The many copies of fragments of the same length, referred to as a band of DNA, migrate toward the detection area with approximately the same velocity. When a band of DNA passes through the detection area, the strength of the fluorescence emitted corresponds to the number of fragments excited by the laser (i.e. more fragments give rise to a stronger signal). An example of the resulting fluorescence signal is shown in Figure 3-1. In this figure, rise in signal strength indicates a distinct fragment length passing the detection area, and when the signal falls, it corresponds to little or no DNA fragments at the detection area at that time.

As the laser sweeps through the detection area, it intersects each channel once in each rotation, but may collect several data points within a channel. Ideally, we would only need one point per channel per revolution. However, elements of noise and low signal require us to collect more than one point per channel and then process those points to achieve one representative point per channel per revolution. In terms of points per peak, we need a minimum number of points to define each peak and build the electropherogram. The minimum number of points that current basecalling applications use to determine a peak are approximately 8 to 10 points.

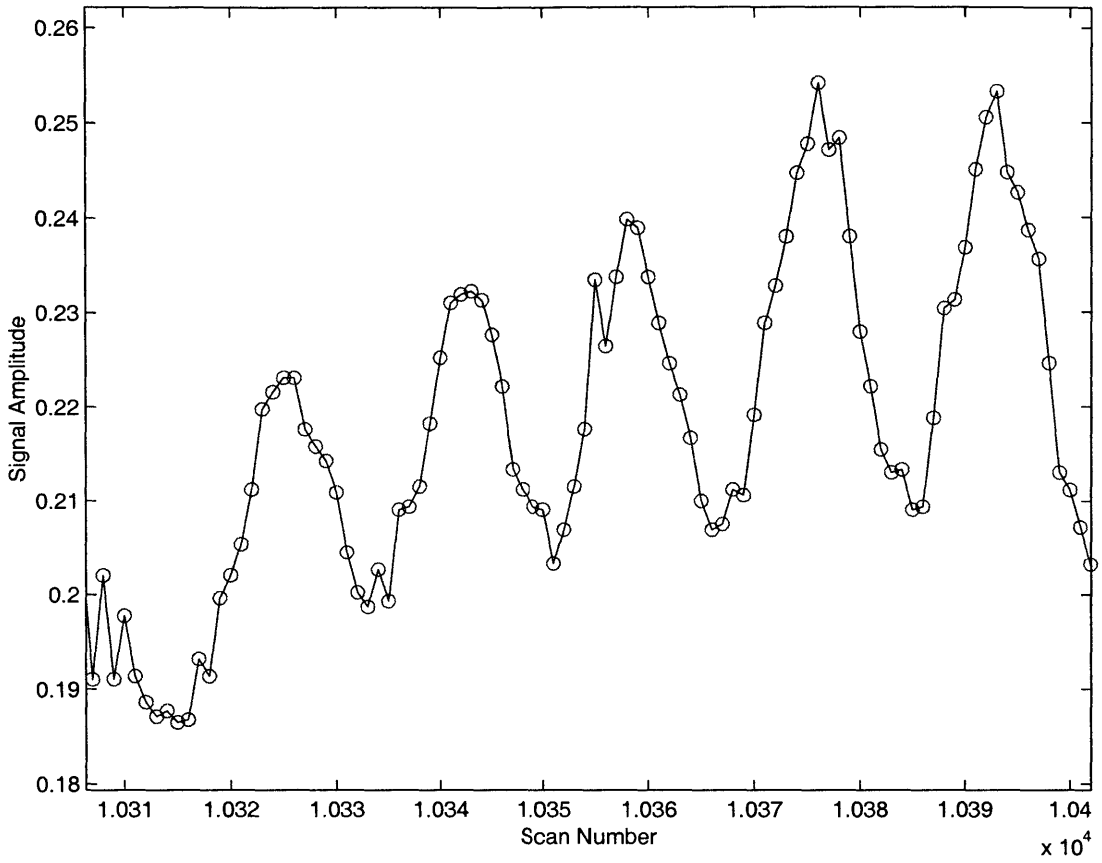

Figure 3-1: PMT signal in a single channel. The X axis shows the number of scans which is correlated to the time elapsed. The open circles indicate data points that were sampled.

3.2 Data Acquisition System Description

In order to implement the data acquisition software for this system, we need to identify the hardware constraints in the system and understand the software aspect of data acquisition in detail.

3.2.1 Hardware Attributes

The spinner motor is the means through which data is collected from the plate. The spinner rotates at a rotational speed measured in encoder counts per millisecond (cts/ms). One full revolution of the spinner is 2^{16} encoder counts. The spinner

can move in a clockwise and counterclockwise motion corresponding to positive and negative speeds, respectively.

The data sample rate is the rate at which information is available to the DSP. The data sample rate is an attribute of the ADCs, which are located on the data acquisition board. The voltage signal that is output from the PMTs is continuously provided to the ADCs. The ADCs sample the input at a fixed rate. The maximum rate for the ADCs is to sample every 5 μs.

The rate at which data is uploaded from the DSP to the PC is currently fixed by hardware and protocols at 6.6 points per millisecond. There is also an overhead of 6.5 μs for each upload that takes place. For this reason, it is more desirable to upload a large set of points at once, than to upload smaller sets of points more often.

3.2.2 Collection Method

The collection method is the method that the DSP uses to decide when to collect data. The ADCs are continuously sampling data that is delivered through the optics and PMTs. The DSP, however, does not need to collect all those points. Figure 3-1 shows a plot of data that streams into one of the ADCs and the points that are actually sampled and collected. Ideally, the DSP should only collect the data that correspond to the channels containing DNA and ignore the rest of the data points (e.g. data from channels without DNA, glass plate, and empty space).

As the collection method becomes more complex, the overhead involved in implementing it on the DSP increases. The overhead costs processing cycles, which might be valuable for other processes. Using more complex collection mechanisms, however, can increase the quality of the data by reducing the amount of useless data collected.

3.2.3 Pre-processing

DSP pre-processing refers to any manipulation of the data that occurs after it is retrieved from the ADCs and before it is uploaded onto the PC. More specifically, this pre-processing reduces the set of data points to a single representative data point.

An example would be to take the mode of all the data points collected in a single channel.

The main reason for performing any data pre-processing is to reduce total upload time to the PC. Once the DSP memory buffer is full, it can no longer accept any data and must empty its buffers before continuing data collection. While the PC uploads the data, data collection on the DSP has ceased. If the DSP is uploading data while the spinner is passing through the detection region, data is lost. Pre-processing the data to reduce the number of points stored in DSP memory allows more data to be collected before upload takes place.

3.2.4 Data Storage

Data storage refers to how data is arranged and stored in memory on the DSP. The data memory of the 2181 DSP chip has a capacity of 16 Kbytes. The storage implementation affects the rate at which the data buffers can be uploaded to the PC. The storage implementation should maximize the rate at which data can be uploaded with the current memory limitations.

3.3 Initial Software Design

The DSP software was initially designed and written by an outside consultant (Craig Simpson, Simpson Research, Danville, VT). We have evaluated and modified this software to reflect the requirements and functionalities needed. The data acquisition module of the initial version of DSP software implemented the collection method, data pre-processing, and storage according to the design specifications of the hardware.

In the original implementation of the collection method, the DSP only collected data points from the ADCs while the spinner was intersecting a channel. This was implemented by triggering data collection when the laser hit the beginning of a channel. When the laser beam hits the edge of the channel, the edge of the channel scatters the beam. This is a unique effect which can be detected. This scatter signal from the optics was used to trigger the DSP to collect data.

The DSP was programmed to collect 8 data points after the scatter signal was detected. After the 8 points were collected, the DSP stops data collection and pre-processes the data. It averages the 8 points to produce one representative data point for that channel. After saving that point to the data buffer, the DSP waits for the next scatter signal to arrive.

The data storage in the initial design was comprised of five buffers, each holding a maximum of 384 points (one point per channel). The first four buffers were for each of the ADC signals and the fifth buffer was to hold the encoder position.

This initial design fulfilled the data acquisition requirements and the hardware specifications. However, the scatter signal hardware did not function according to the specifications. This required us to modify the initial design to achieve data acquisition.

3.4 Software Modifications

Many modifications have been made to the initial version of DSP software over the past several months. Most of the changes to the software pertain to the variables and methods discussed in Section 3.2. The most current version of our DSP code is provided in Appendix A.

3.4.1 Collection Method

The collection method (described in Section 3.2.2) can be implemented in various modes. While determining the best implementation, two independent variables were considered: spinner motion and collection mechanism. The spinner motion refers to the uniformity or non-uniformity of the spinner velocity. The collection mechanism refers to the frequency and timing of data collection.

The most simple, but inefficient, method is to collect data continuously as the spinner rotated 360°. This mode was inefficient because all of the data points are not needed. The DSP only needs to collect data from the 60° arc corresponding to the detection zone on the plate (See Figure 2-3). This mode was implemented to

provide the necessary data acquisition capabilities at the initial stages of software development.

Spinner Motion

The problem with full 360° rotation of the spinner at a constant velocity was that we could not meet the data resolution requirement (described in Section 3.1.2). The spinner velocity is limited by the resolution requirement. If the spinner rotates too quickly, the data resolution is insufficient. However, even at the fastest spinner speed, the time that elapsed between leaving the detection region and returning to the beginning of the region was too long. We were unable to collect enough points per peak.

We explored methods to narrow the region that the spinner scanned. The region for the detection zone could be defined on the DSP by a start encoder position and end encoder position. If the spinner only moved between the start and end positions in the region, we would only collect data from that region.

This idea seemed good in design, however, it did not prove successful. Moving the spinner in a zig-zag motion, from beginning to end of the collection region solved the problem of peak resolution. However, due to the mechanics of the spinner motor and encoder, we experienced a feedback effect of the spinner motor.

After the spinner reaches the end of the detection region, it is supposed to move back to the start location of the region to scan again. However, the spinner was not able to move to a precise location. The spinner moved beyond the target location and mechanically oscillated until it settled to the correct position. The time that it took to settle to its new position added to the time it took to accelerate to sweep through the region.

After unsuccessful attempts with spinner motion, we decided to implement a custom speed profile for the spinner motor that causes it to rotate continuously for 360° at varying speeds. The spinner would move at a constant velocity of 60 cts/ms, low enough for the required data resolution, and then accelerate and decelerate until it returns to the beginning of the collection region. This method proved successful.

Collection Mechanism

Although we solved the problem of timing, we were still collecting data under the minimum rate necessary. The initial experiments only used 1-30 channels and therefore only needed limited data acquisition capabilities. As the experiments expanded to use all of the 384 channels on the plate, the data collection mechanism was insufficient.

One of the problems with continuously collecting data is that data was collected inside and outside of the detection zone. To narrow the region in which data was collected, we implemented a collection mechanism that triggered data collection when the spinner passed the start location of the detection region. The DSP continued to collect data until it reached the end location.

Although this reduced the number of points collected, it was still insufficient. Data only needs to be collected *within* the channels of the plate. Each channel is 100 μm wide and the amount of glass between two neighboring channels is also 100 μm. This means that we can reduce the number of points collected by 50% by collecting data only in the channels.

The original collection method was one that collected only within the channels or lanes. We had to find an alternative implementation to accomplish data collection within the lanes. We knew that could successfully trigger data collection based on an encoder position. So, we decided to trigger data collection at the beginning location of each lane.

This method would require us to ascertain the position of all of the lanes, download those positions to the DSP, and collect only within the width of that lane. In essence, we are implementing data collection over 384 mini-collection regions.

One concern is that the glass plate and the 384 channels will not always be in the same position. Once a glass plate is installed onto the instrument, however, the locations will be fixed. But the glass plate is removable and will be replaced between runs. This might cause a minute change in lane locations, but when dealing with widths of microns, it is a valid concern.

The solution is that the detection of the lane edges must be done prior to every

run, after the plate is installed. This can be accomplished by taking approximately 100 preliminary scans of the detection region. The properties of the glass and the gel are different and their fluorescence signals will also differ. We can detect the location of these variances and thus locate the beginning of each lane.

Once the location of each lane is determined, we can download the lane locations to the DSP and instruct the DSP to trigger data collection only after it passes one of these locations. We programmed the DSP to collect 8 points after being triggered. Taking into account a constant velocity of 60 cts/ms, 8 points (one point per encoder position) corresponds to region slightly wider than the width of one channel.

3.4.2 Pre-processing

Data upload to the PC takes on the order of 50-100ms. This is a significant portion of time, relative to data collection, during which we cannot collect any additional data. If the spinner rotates at 60 cts/ms, then the spinner will have moved between 3000 to 6000 encoder counts by the time data upload is complete. Considering that the detection area is slightly less than 6000 counts wide, this poses a challenge if data upload begins anywhere near the vicinity of the detection area.

We would like to minimize the number of points uploaded to the PC, so that we can maximize the time the DSP has to collect data. Pre-processing data sampled from the ADCs allows us to reduce the number of data points uploaded to the PC. The alternatives for pre-processing are simply to upload all the points collected or to compute the minimum, maximum, or average of a set of points to reduce the total number of data points.

The simplest option here is to do no pre-processing at all and simply upload all the raw data. This would require no overhead on the part of the DSP. However, due to the fact that the DSP can execute instructions at the rate of 33 million instructions per second, it would be to our advantage to offload some of this processing to the DSP.

We noticed that our data set included multiple data points corresponding to a single encoder position. The lowest distance resolution is a single encoder position,

so it did not make sense to upload multiple points corresponding to the same location. Only one representative point is necessary. Since a single channel spans the width of approximately 15 encoder counts, it is unnecessary to store multiple points per encoder position. It is inefficient to store such excess data. So we decided to reduce the multiple set of data points corresponding to one encoder position to a single representative point.

To produce a representative point out of a set of data points, it seems most logical to take the average of the set of points. However, there were a variable number of data points that needed processing at each encoder position and this was hard to accommodate on the DSP, especially with the increased overhead of performing division on the DSP. Division requires more cycles on a DSP than addition or subtraction.

The other alternatives to pre-process the data were to take the minimum or the maximum of the set of data points. Our concern with using this method was that any anamolous point in the set would cause a false data point to be recorded. An anamolous point could very likely be the result of noise in the system. Due to the properties of the amplifiers, we found that taking the minimum of the set of points reduced the effect of noise and allowed us to collected the best quality data.

3.4.3 Data Storage

The data storage implementation was only slightly modified from the original implementation. The original implementation was to use five separate buffers, each holding a maximum of 384 points. The significance of the data storage size is that it limits the maximum contiguous region scanned during data collection. When the storage is full, the DSP can no longer collect data. The faster the storage buffer reaches its limit, the faster data collection must occur.

To allow the DSP more time to collect data, we expanded the buffer sizes to the maximum capacity of the DSP memory buffer. As mentioned in Section 3.2.4, the DSP has 16 Kbytes of data memory, which allowed us to expand the buffer size to 3225 data points.

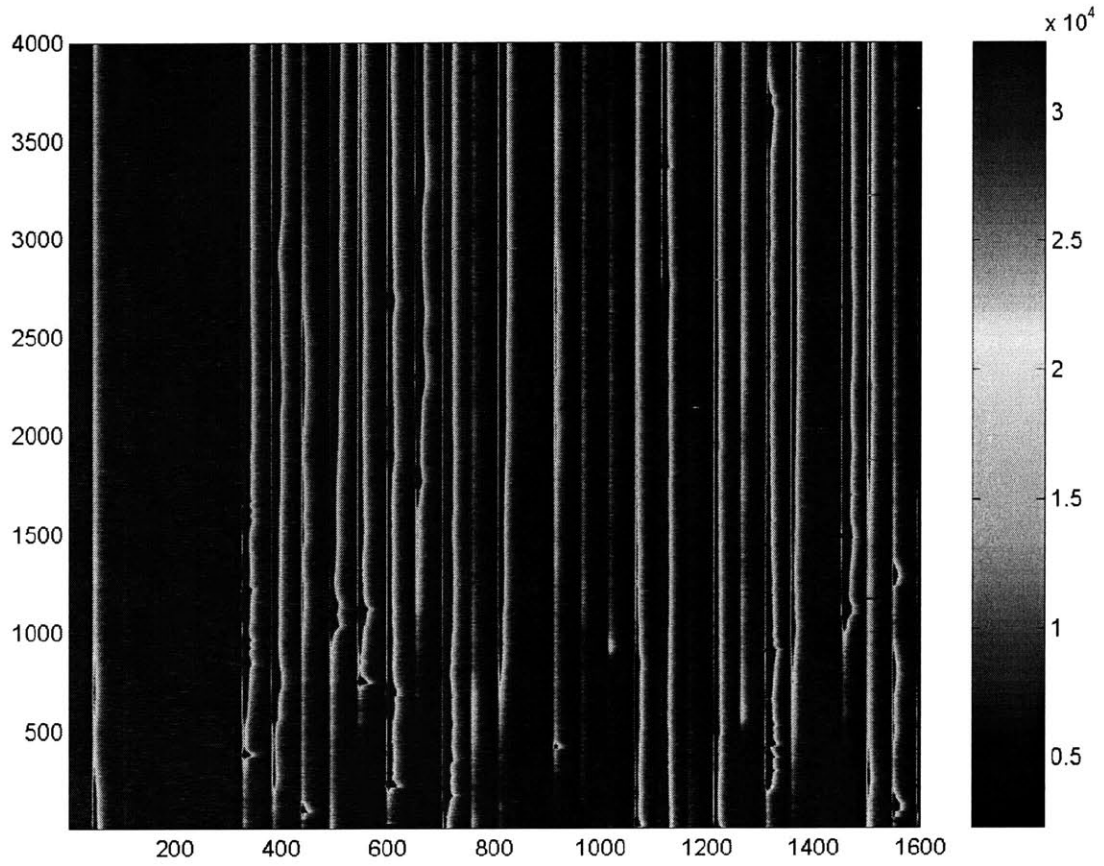

Figure 3-2: Raw data from electrophoresis. The X axis is the encoder position and the Y axis is the number of scanner sweeps. The color scale represents the intensity of signal.

Chapter 4

Instrument Control Software

4.1 Software Requirements

The software requirements for instrument control application are to establish communication with the data acquisition board, coordinate instrument components during electrophoresis, and upload and store acquired data. The function of this application can be summarized in this analogy: the DSP provides the tools and this software uses those tools to extract the desired data. This application also provides a graphical user interface (GUI) to the instrument and data collection routines.

The instrument control application must establish communication with the data acquisition board and DSP to control the various components. There are different protocols involved in establishing communication on the hardware and software level on both the PC and the data acquisition board. The control application translates commands from the user into commands that the data acquisition board can execute.

The control application must control the spinner motion, the data sampling, the high voltage supplies, and the data pre-processing. The control application must coordinate all these components and tasks to acquire data from the instrument.

The GUI aspect of this application requires consideration of a number of factors. This application will be used by biologists, chemists, and other laboratory technicians and should be written with their needs in mind. As is true with most software interfaces, the goal is to abstract away the protocols and underlying hardware and provide

a suitable, easy to use interface to the instrument. The user of the application should only be concerned with operating and manipulating the instrument, not learning the intricacies of using the software. As the work to date is the initial version of the application, the emphasis is to provide functionality and establish a solid basis for future work and improvements.

Because this version of the software will mainly be used in our laboratory for experiments, it should allow for manual control of the instrument when desired, in addition to automated control. For testing purposes, we would like to be able to break the abstraction and access lower level functions that are not provided in the GUI.

The DSP software provides the functionality for each component on the data acquisition board. This data acquisition control software must provide the overall coordination of all these components to perform electrophoresis and data acquisition. This software must set the high voltage to the necessary settings, begin spinner motion, home the spinner location, set the necessary parameters on the DSP, and then initiate data collection.

4.2 Software Design

For the instrument control software design, no formal specifications were made. Generally in software development, software requirements are evaluated to formulate design specifications. However, due to the immediate necessity for a functional interface to the instrument, for the purposes of testing and experimentation, an initial prototype that provided basic functionality was needed.

4.2.1 Communication

The major function of this control software is to serve as a user interface to the instrument. The first aspect in designing this application is to establish communication with the instrument. This is accomplished by creating library files to provide communication functions for the data acquisition board and the DSP. The library functions are

divided into three categories: lower level communication functions, DSP communication functions, and data acquisition board functions. The lower level communication library handles all the parallel port protocol and basic access to the board. This communication occurs at the byte level. The DSP communication library handles all the protocol necessary to execute DSP commands. The data acquisition board functions handle all the control and execution of the components that reside on the board, such as the high voltage switches.

4.2.2 Graphical User Interface

The graphical user interface must give the user access to various functionalities. This graphical interface must provide the ability for the user to activate and set the voltage for the high voltage supplies. The interface must also allow the user to control spinner motion, read spinner location, and recalibrate position encoder. In terms of data collection, this interface must allow the user to input DSP parameters for data collection, initiate data collection, and upload data from the DSP.

4.2.3 Data Collection

The instrument control software is responsible for initiating data collection on the DSP and uploading the resulting data from the DSP. The control software must also store the raw data in a format compatible with basecalling software. The software must work in conjunction with the DSP to implement data acquisition.

4.3 Implementation

The application, Sequencer.exe, was developed on the WindowsNT platform and uses Microsoft Visual C++ as the coding environment. The initial version of this application, shown in Figure 4-1, contained all the functionality deemed necessary at the outset. These functionalities included: controlling spinner location and speed, setting offset values for the analog to digital converters, controlling the high voltage supplies,

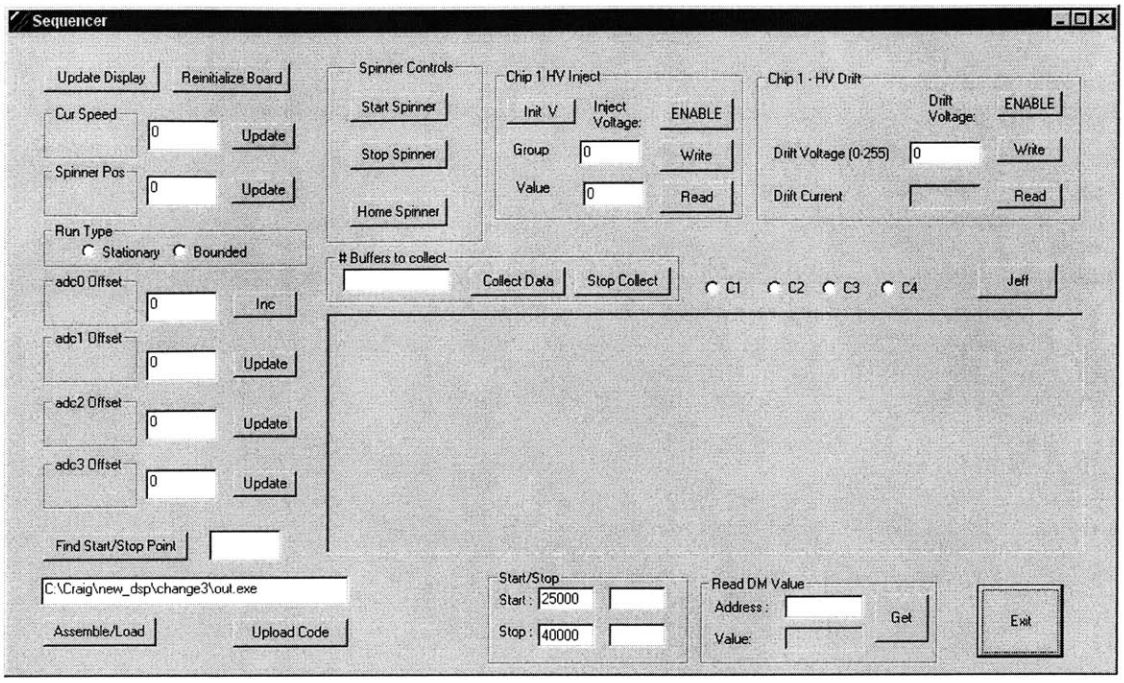

Figure 4-1: Initial Version of Sequencer.exe. This is the initial graphical user interface designed for the instrument control application.

uploading the appropriate version of DSP software, and starting and terminating data collection. Functionalities that were not necessary, but possibly useful for testing purposes were added. These included plotting incoming data so that data could be previewed, and accessing memory locations on the DSP.

During the course of the software development, there have been many revisions to the interface. These revisions include additional features, upgraded features, and removal of features. The successive versions were prompted by user responses to the application. Features that were not used or needed were removed. Features that were desired were provided either by modifying existing features to meet the requirements or implementing new features.

Some of the features removed from the initial version of the software were the plotting capabilities and setting offset values for the ADCs. These features were removed due to lack of use. Aside from the few features that were removed, many more were added. The interface was changed to accommodate the specification of

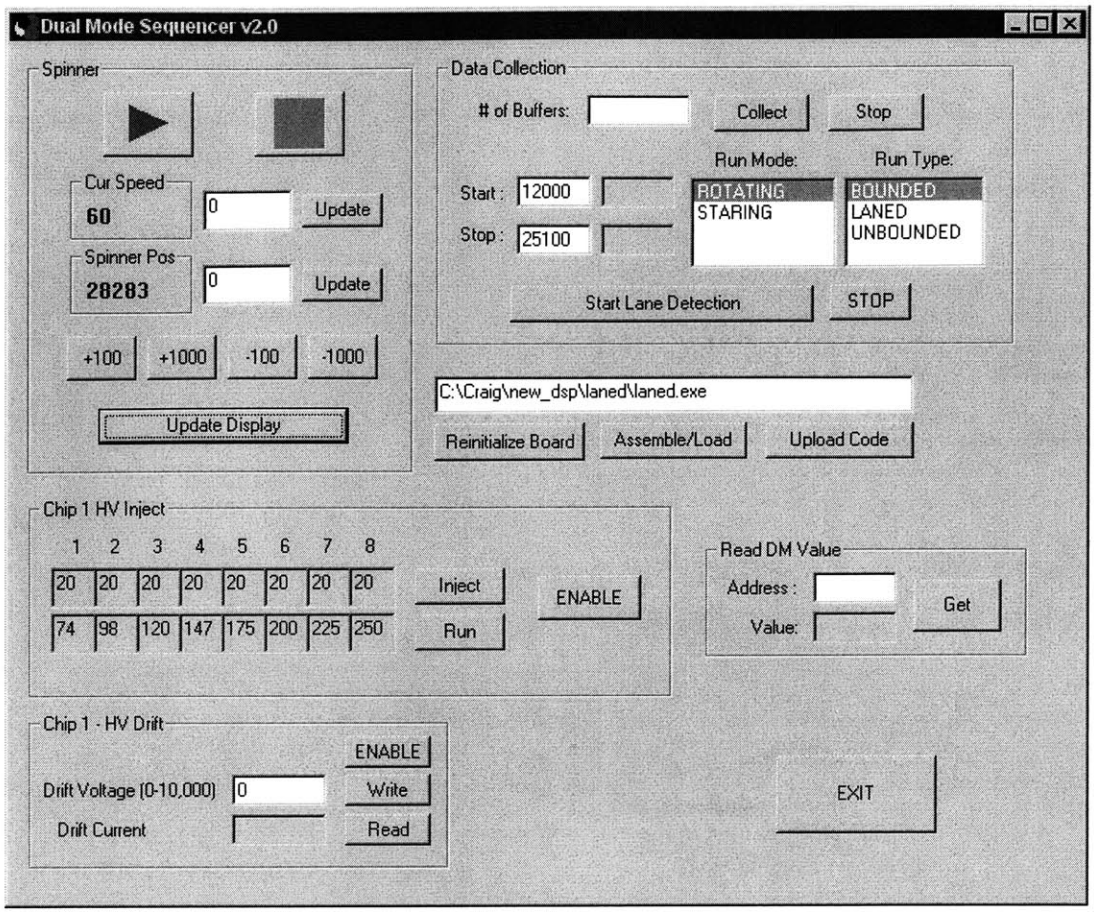

Figure 4-2: Latest Version of Sequencer.exe.

a "run type", "run mode", lane detection, and individual voltage controls. Some of the changes to the application were prompted by changes to the underlying DSP software. Other changes were prompted by the progression of instrument testing and experiments. As the instrument develops and begins to use more of the features originally intended, the interface must provide these functionalities to the user.

The DSP software added the feature of specifying a "run mode", or more specifically the spinner running mode. The spinner motion variable was discussed in Section 3.4.1. The DSP code accommodated uniform spinner velocity as well as a variable velocity profile. It was logical to implement a feature on the interface to specify the mode of spinner motion.

The "run type" feature was also added to the interface as a result of changes to the

DSP software. The DSP contained three run types: bounded, unbounded and laned. The unbounded run type corresponds to continuous collection of data as the spinner rotates. The bounded run type is the collection of data within a specific region. The interface provides input to specify the beginning and end location of that region. The laned run type is the collection method that collects only within the channels of the chip. This run type can only be used if lane detection has already occurred.

The addition of a new laned mode in the run mode category required the task of detecting all the lanes on the chip. The interface needed a feature that performed lane detection. Lane detection would scan the detection area of the chip, locate the starting location of each channel, and store that information on the DSP. This addition is, once again, a response to changes in the DSP software.

Finally, the last modification to the software was the addition of individual voltage controls for the high voltage supply. The voltage supply is subdivided into 8 individual voltage control groups. Since the voltage for each group can be independently set, the interface needed input features for each of those voltage groups.

Chapter 5

Peripheral Component Control Software

5.1 Software Requirements

The software requirement for the peripheral component control software is to automate the operation of the peripheral devices and coordinate the execution of device actions. There are various peripheral devices that are currently being employed alongside the sequencing apparatus, such as the Harvard syringe pump and Masterflex peristaltic pump. It is possible that the current devices in use will be replaced by other commercial products in the future, each having its own unique set of protocols and functionalities. The control software must therefore be able to handle such changeable components and provide for seamless integration of all components.

An immediate requirement for this machine control application is that it must interface with every device in the system. Each of these devices may have completely different software interfaces and this application must be able to communicate with all of them in order to coordinate them remotely. This application must implement automated device control, meaning that once the program is initiated, no further user interaction is necessary. The user should only need to specify a schedule for device execution at the application's initialization and allow the application to run to completion. Having an automated production sequencing instrument in mind, we

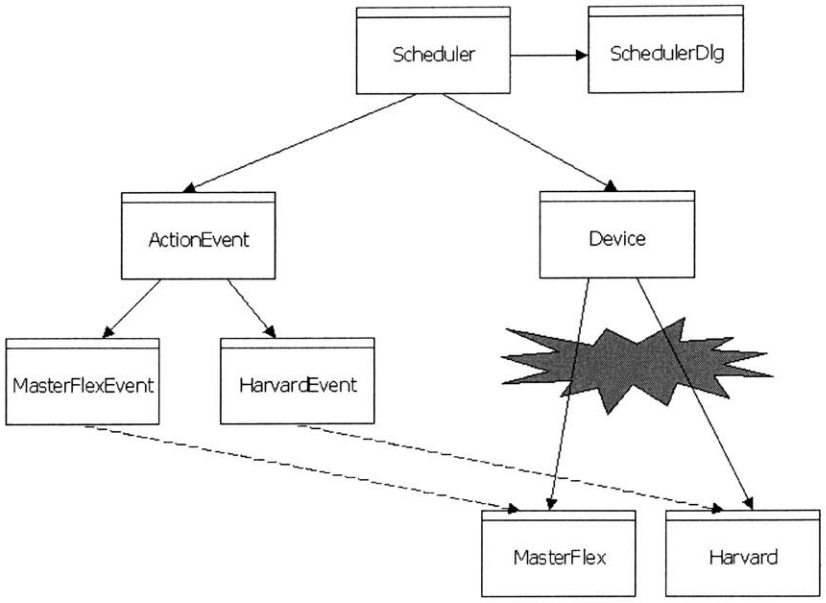

Figure 5-1: Housekeeper Code Model. The Scheduler class is at the top of this class hierarchy. It coordinates the interface through the SchedulerDlg class, and manages the ActionEvent and Device classes. The shaded region under the Device class is further elucidated in Figure 5-2.

need to implement automation. However, we would also like to have manual software controls for each device available, as the instrument is still in its testing phases and manual control during each experimental sequencing run is desirable.

5.2 Software Design

This application was named Housekeeper, because the essence of the application is to organize the different peripheral components and manage device execution efficiently. The general design of this application is centered around two basic concepts: devices and events. Devices are representations of the peripheral components of the instrument. Events are individual tasks that the devices must perform. For example, an MIT Pump would be a device and starting the pump would be an event.

The code model for the Housekeeper.exe application is an object-oriented one. The main application, HouseKeeper, maintains program control. It contains a Scheduler

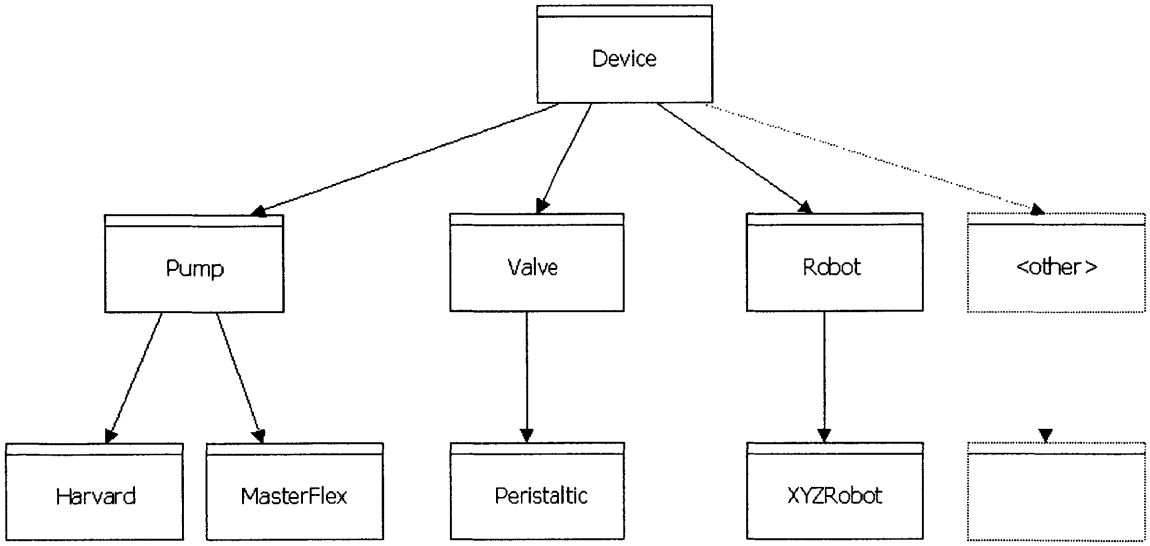

Figure 5-2: Device Code Model. The virtual Device object has different virtual subclasses, each pertaining to machine type. Each specific device, such as the Harvard Pump, is implemented as a subclass to its machine type class.

object, which is the brain of the program. The Scheduler initializes and manages all the devices in the system, all of the events to be executed, and the synchronization and execution of events. Figure 5-1 shows how each of the objects in this application are interconnected.

The various peripheral devices of the instrument are implemented by a virtual Device class. Each device has the attributes of a name and whether or not it has been initialized. Each device also knows its capabilities regarding what kinds of events it can execute. Within the Device class, there are virtual subclasses for each type of device in the system. The current subclasses are Pump, Valve, and Robot. For every device in the system, a new class must be implemented as a subclass of one of these types. For example, if we had an MIT pump, we would have to implement an MITPump class as a subclass of Pump (See Figure 5-2).

The events in the system are implemented by a virtual ActionEvent class. An ActionEvent knows what device is its target, what action to execute, and what time it should be executed. This virtual ActionEvent class must also be overridden by a subclass that is specific to the type of device used. For each device in the system,

there should be a corresponding ActionEvent subclass implemented (e.g. if we have a HarvardPump device, we should implement a HarvardEvent class). This ActionEvent subclass knows how to control its specific device and call the correct function from the Device class to execute its action.

The Scheduler class contains all the information that the user inputs to the program. It creates, initializes, and controls all the devices in the system. It is responsible for the event queue and correctly executing events according to an internal timer. The Scheduler also provides the functionality to pause execution of events.

Our design meets all of the requirements of the machine control software except manual control of the devices in the system. We decided in the design that this functionality could be provided in a separate application and need not be included in the final design. Manual control of the devices is only necessary during the testing phase of the instrument and could easily be implemented.

5.3 Implementation

The Housekeeper.exe application was developed on a WindowsNT platform and with the Microsoft Visual C++ coding environment. The application opens the initial window and begins the program (See Figure 5-3). It creates an instance of the Scheduler class, which as we mentioned before, is the brain of the application. All of the user input to the program concerning devices and events must be specified in an input file. This input file must contain the devices that are present in the system and the list of events that are to be executed. If any of the devices need extra parameters specified with a command, it should also be specified in this file. Exact specifications for this input file are given in Appendix B. The initial window allows the user to specify this file.

The Scheduler then takes over, by initializing all the devices specified in the input file. The Scheduler needs to establish communication with each device and perform any initialization protocol necessary. Since the Scheduler needs to perform the initialization for each device, it must have access to functions for all devices. Com-

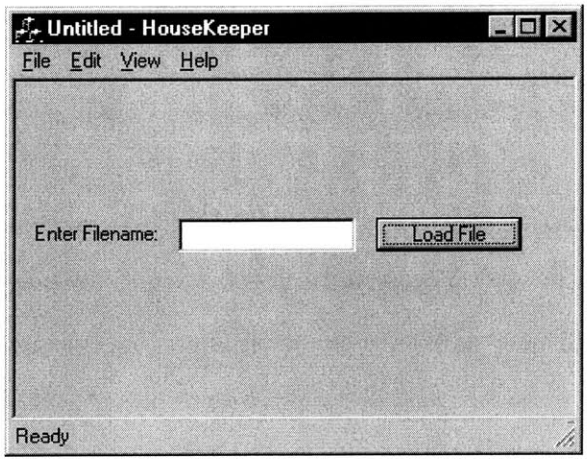

Figure 5-3: Housekeeper Initial Window. This window allows the user to input the file to specify the devices and events for the application.

munication library files needed to be created to provide this access to each device and allow the Scheduler to accomplish initialization. The current devices used in the system are the Harvard and Masterflex pumps previously mentioned, the pneumatic valves, and the XYZ Robot arm. Library files were created for each of these devices, along with device classes and event classes.

The devices must all be physically connected to the computer in order for this program to able to control them. Remote or network control of the devices is not implemented in this version. The Harvard Pump(s) should be connected via COM port 1, the Masterflex Pump(s) should be connected via COM port 2. If there are more than one of either pump type, they should be daisy-chained according to each individual pump's specifications.

The Scheduler then loads all the event information found in the file and creates an event queue and an internal timer thread. It displays the information in the Scheduler Dialog Box (See Figure 5-4) and is now ready for execution. Once the user clicks on the Start button, the timer thread begins counting from 00:00:00. When all of the events in the list are executed, the timer thread terminates. The user can also stop event execution by clicking on the Stop button. This suspends the thread and gives the user the option of resuming event execution.

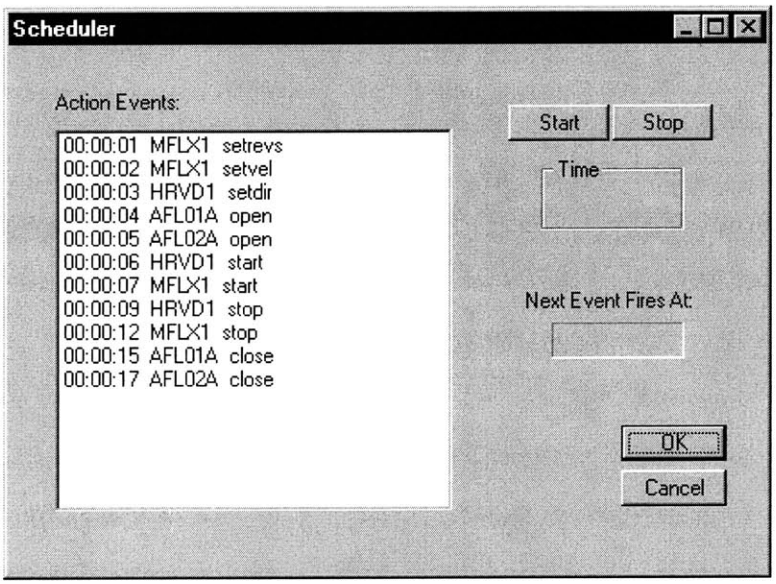

Figure 5-4: Scheduler Dialog Box. The box on the left displays the event queue. Each entry in the event queue displays the time to execute, device name, and action to be executed. The buttons on the right allow the user to start and pause the Scheduler timer.

Chapter 6

Conclusion

The goals of software development for this sequencing instrument were to automate instrument operation, data collection, and data processing to produce DNA sequence. Over the course of the past year, we have developed a functioning version of instrument control software, Sequencer.exe, and peripheral component control software, Housekeeper.exe. These tools have allowed us to being automating instrument operation and data collection.

Future work in the area of software development include refining both of these software pieces to develop a production piece of software. These two modules of software should be encapsulated by a larger application to manage instrument control as well as basecalling modules.

Immediate work that would be useful to the continuing software development for the instrument would be to create a manual control interface to the various peripheral devices and finish and test the lane detection mode for the DSP.

Appendix A

DSP Code

```
.MODULE/RAM/ABS=0  data_ctrl;

{------Definitions-------------}
.CONST  Sys_Ctrl_Reg  =0x3fff;
.CONST  Dm_Wait_Reg =0x3ffe;
.CONST  Tperiod_Reg =0x3ffd;
.CONST  Tcount_Reg =0x3ffc;
.CONST  Tscale_Reg =0x3ffb;
.CONST  Sport0_Ctrl_Reg =0x3ff6;
.CONST  Sport0_Sclkdiv =0x3ff5;
.CONST  Sport0_Rfsdiv =0x3ff4;
.CONST  Sport0_Autobuf_Ctrl =0x3ff3;
.CONST  Sport1_Ctrl_Reg =0x3ff2;
.CONST  Sport1_Sclkdiv =0x3ff1;
.CONST  Sport1_Rfsdiv =0x3ff0;
.CONST  Sport1_Autobuf_Ctrl =0x3fef;
.CONST pf_data =0x3fe5;
.CONST pf_control =0x3fe6;
.CONST adc0r =0x8; {adc0 read address}
.CONST adc1r =0x9; {adc1 read address}
.CONST adc2r =0xA; {adc2 read address}
.CONST adc3r =0xB; {adc3 read address}
.CONST Cdacw =0x1; {control dac write address}
.CONST Ddacw =0x2; {diag dac write address}
.CONST Posx =0x0; {spinner pos counter address}
.CONST start_conv =0x4; {start convert address}
.CONST  TotalSamples      =3072;
.CONST  divisor  =0;
.CONST rotating =0;
.CONST staring =1;
.CONST unbounded_run =1; {possible run_types}
.CONST  bounded_run =2;
.CONST lane_run =4;
.CONST detect_run =8;
.CONST Ld_buffer_size =6144; {lane detect buffer size}

.VAR/DM/ABS=0x00 control_reg;
.VAR/DM buff_full;
.VAR/DM kp;
.VAR/DM kd; {spinner servo params}
```

```
.VAR/DM ki;
.VAR/DM gain_shift; {system gain=2^gain_shift}
.VAR/DM p_range; {max position error}
.VAR/DM I_range; {max integral output}
.VAR/DM out_range; {maximum motor command}
.VAR/DM set_pos; {current position setpoint}
.VAR/DM set_vel; {spinner velocity in counts per loop}
.VAR/DM panic_err; {maximum error before shutdown}
.VAR/DM adc0;
.VAR/DM adc1;
.VAR/DM adc2;
.VAR/DM adc3;
.VAR/DM adc0_off;
.VAR/DM adc1_off;
.VAR/DM adc2_off;
.VAR/DM adc3_off;
.VAR/DM num_samples; {samples per lane}
.VAR/DM num_lanes;
.VAR/DM motor_val; {current motor command}
.VAR/DM in_lane;
.VAR/DM home_flag;
.VAR/DM run_mask;
.VAR/DM data_mask;
.VAR/DM home_mask;
.VAR/DM end_buff;
.VAR/DM cur_pos;
.VAR/DM cur_error;
.VAR/DM pst_pos;
.VAR/DM pst_error;
.VAR/DM cur_vel;
.VAR/DM home_pos;
.VAR/DM home_delta;
.VAR/DM I_out_hi;
.VAR/DM I_out_lo;
.VAR/DM spinner_init;
.VAR/DM first_pass;
.VAR/DM mtr_dis_mask;
.VAR/DM mtr_ena_mask;
.VAR/DM sample_rate;
.VAR/DM servo_rate;
.VAR/DM home_dis_mask;
.VAR/DM data_en;
.VAR/DM home_en;
.VAR/DM vel_err;
.VAR/DM low_position;
.VAR/DM high_position;
.VAR/DM run_type_flag; {detect/lane/bounded/unbounded}
.VAR/DM run_mode_flag; {rotating or stairing}
.VAR/DM new_read_pos;
.VAR/DM sample_counter;
.VAR/DM adc0_min;
.VAR/DM adc1_min;
.VAR/DM adc2_min;
.VAR/DM adc3_min;
.VAR/DM end_buff2; {this is for the end of buffer for lane_detection_mode}
.VAR/DM lane_mask;
.VAR/DM lane_dis_mask;

.VAR/DM/ABS=0x100 ch0_data[TotalSamples];
.VAR/DM/ABS=0xD00 ch1_data[TotalSamples];
.VAR/DM/ABS=0x1900 ch2_data[TotalSamples];
```

```
.VAR/DM/ABS=0x2500 ch3_data[TotalSamples];
.VAR/DM/ABS=0x3100 pos_data[TotalSamples];
.VAR/DM/ABS=0x3D00 lanes[384];

.VAR /PM lookuptable[4096];
.init lookuptable :<speed1r>;

.INIT control_reg: 0x0;
.INIT num_lanes: 384; {num lanes in chip}
.INIT num_samples: 8; {num samples per lane}
.INIT kp: 10000;  {10000}
.INIT kd: 26000;  {25000;}
.INIT ki: 1;  {10;} {200}
.INIT gain_shift: 0x6;
.INIT p_range: 100;
.INIT I_range: 25;
.INIT out_range: 2047; {dac clamp value}
.INIT set_pos: 0;
.INIT set_vel: -328; {should be -328 for 5 rps, cw rotation}
.INIT panic_err: 0x4000; {maximum error before shutdown...1/4 rev}
.INIT home_flag: 0;
.INIT run_mask: 1;
.INIT data_mask: 2;
.INIT home_mask: 4;
.INIT lane_mask: 8;
.INIT home_dis_mask: 0xfb;
.INIT lane_dis_mask: 0xf7;
.INIT adc0_off: 0;
.INIT adc1_off: 0;
.INIT adc2_off: 0;
.INIT adc3_off: 0;
.INIT spinner_init: 0;
.INIT mtr_dis_mask: 0x0E;
.INIT mtr_ena_mask: 0x01;
.INIT first_pass: 0;
.INIT buff_full: 0;
.INIT sample_rate:    79;   {5 uS interval}
.INIT servo_rate: 160000; {31999=1 K Hz}
.INIT data_en: 0;
.INIT home_en: 0;
.INIT   low_position:  12400;
.INIT high_position:  23800;
.INIT  sample_counter: divisor;  {48*2^16 = 3.2e6..number ticks that equal 5Hz}
.INIT  run_type_flag:  bounded_run;
.INIT   run_mode_flag:  rotating;
.INIT new_read_pos:  0;
.INIT adc0_min: 0;
.INIT adc1_min: 0;
.INIT adc2_min: 0;
.INIT adc3_min: 0;

{------Interrupt vector table------}

jump start; {jump over interrupt vectors}
rti; rti; rti; {do init routine at start}

rti; rti; rti; rti; {IRQ2 interupt}

rti; rti; rti; rti; {IRQL1 interrupt }

rti; rti; rti; rti; {IRQL0 interrupt }
```

```
rti; rti; rti; rti; {SPORT0 TX interrupt }

jump start_convert; {sport0 RX interupt}
rti; rti; rti;

jump home_int; {IRQE interrupt, enabled for homing}
rti; rti; rti;

     rti; rti; rti; rti; {BDMA interrupt }

rti; rti; rti; rti; {SPORT1 TX (IRQ1) interrupt }

    jump get_data; {SPORT1 RX (IRQ0) interrupt }
rti; rti; rti; {a/d done interrupt}

jump loop_t; {TIMER interupt}
rti; rti; rti; {run the pid algorithm in background}

rti; rti; rti; rti; {POWER DOWN interrupt }

{--------------SET UP SYSTEM AND MEMORY---------------}
start:
DIS ints;
ENA M_MODE;  {MSTAT=0x10;}

AX0=0x1000; {0 pgm wait states}
DM(Sys_Ctrl_Reg)=AX0; {sport 0 enabled}
{sport 1 disable,flags enable}

AX0=0x1FCD;
DM(Dm_Wait_Reg)=AX0;  {one wait state on}
{ext mem}
{5 wait on i/o at 0x0 to 0x1ff}
{1 waits on i/o at 0x200 to 0x3ff}

{---------------set up programable flags-------------------}

AX0=0x7f0f; {7 boot wait states}
DM(pf_control)=AX0; {cms true for all accesses}
          {pf bits 0-3 outputs}

{--------------SET UP TIMER----------------------------}

AX0=3199; {31999 32 MHz internal clock}
DM(Tperiod_Reg)=AX0;   {generate 1 kHz}
{interupt}
AX0=31999; {start after 1000 uS}
DM(Tcount_Reg)=AX0;
AX0=0x0000;
DM(Tscale_Reg)=AX0; {dec counter every cycle}

{--------------SET UP SPORT0--------------------------}
{0110 0000 0000 0010}
AX0=0x6002; {int rcv clk,ext rec frame sync}
DM(Sport0_Ctrl_Reg)=AX0; {3 bit data}
{sport 0 used to gen interupt}
AX0=0x0000;
DM(Sport0_Sclkdiv)=AX0; {generate (1/16) uS clock period}

AX0=DM(sample_rate);  {4f= 80 sclk, period of frame sync}
```

```
DM(Sport0_Rfsdiv)=AX0; {generate interupt every 5 uS}

AX0=0x0000; {enable sclk out}
DM(Sport0_Autobuf_Ctrl)=AX0; {autobuffer not used}

{--------------SET UP INTERUPTS-----------------------}

ICNTL=0x13; {level sensitive IRQ2}
{edge sensitive IRQ0, IRQ1,enable nesting}

{irq2 irql1 irql0 sprt0T sprtr0R irqE bdma irq1 irq0 timer}
{ 9 8 7 6 5 4 3 2 1 0}
{ 0 0 0 0 1 1 0 1 1 1}

IMASK=0x037; {enable timer,irq0,sport0r}

{------Wait for time to  start------------}

ENA g_mode;

inits:
M0=0; {increment value=0}
M1=1; {+1}
M2=2; {+2}
M3=4; {+4}

M4=1; {+1}
  M5=0; {+0}

L0=0;
L1=0;  {linear addressing mode}
L2=0;
L3=0;
L4=0;

I0=^ch0_data; {ch0_data buffer for adc0}
I1=^ch1_data; {ch1_data buffer for adc1}
I2=^ch2_data; {ch2_data buffer for adc2}
I3=^ch3_data; {ch3_data buffer for adc3}
I4=^pos_data; {spinner position of data}
I5=^lanes; {start position of each lane}

AX0=^ch0_data; {pointer to pos_data buffer}
AY0=\%ch0_data; {length of data buffer}
AR=AX0+AY0; {end of data buffer}
DM(end_buff)=AR;
AY0=Ld_buffer_size;
AR=AX0+AY0;
DM(end_buff2)=AR; {end_buff2 = start of ch0 + Ld_buffer_size}
AX0=0;
IO(Cdacw)=AX0; {clear the motor dac to 0}

IFC=0xff; {clear all pending interupts}
ENA timer;
ENA ints;
AX1=0x6102;
DM(Sport0_Ctrl_Reg)=AX1;

wait_t: {wait for timer}
IDLE;
JUMP wait_t;
```

```
{------Check the operating mode~------------}
loop_t:
AX0=DM(control_reg); {test to check the mode}
AY0=DM(run_mask);
AR=AX0 AND AY0; {bit set... run the spinner}
IF EQ JUMP loop_1; {bit not set...stop the spinner}

AX0=DM(spinner_init); {test to check the mode}
none=PASS AX0;
IF EQ CALL spin_init; {bit not set...initialize the  spinner}
jump run; {bit set... run the spinner}

loop_1: AX0=DM(mtr_dis_mask); {complement of enable mask (1110); 0xE }
AY0=DM(pf_data); {clear the motor driver enable PF0}
AR=AX0 AND AY0;
DM(pf_data)=AR;
AX0=0;
DM(spinner_init)=AX0; {clear spinner init flag. }
IO(Cdacw)=AX0; {clear the motor dac to 0}
AX0=DM(sample_rate);
DM(Sport0_Rfsdiv)=AX0; {set to 79 for 5 uS interval}
AX0=DM(servo_rate); {32 MHz internal clock}
DM(Tperiod_Reg)=AX0;   {generate 1 kHz, value=31999 interrupt}
RTI; { return}

{################################################}
{ RUN 13 cycles    }
{################################################}
run: AX0=IO(Posx); {get the current position}
DM(cur_pos)=AX0; {save it}

AX0=DM(control_reg); {test to check the mode}
AY0=DM(home_mask);
AR=AX0 AND AY0;
IF NE CALL home; {bit set... home the spinner}

AX0=DM(control_reg); {test to check for lane detect mode}
AY0=DM(lane_mask);
AR=AX0 AND AY0;
IF NE CALL lane_detect; {bit set... go to lane detection}

AX0=DM(control_reg); {get data collect enable flag}
AY0=DM(data_mask);
AR=AX0 AND AY0; {check flag status}
IF EQ jump run2; {if zero don't enable data collection}
{else...}
AX0=DM(buff_full); {get buffer full flag}
none=PASS AX0;
IF NE jump run1;  {if equal to 0 enable data collection}
{else...}
RESET fl1,RESET fl2;
DM(data_en)=M1; {enable data collection}
jump pid;

run1: SET fl1,SET fl2; {interupt the pc to get data}
run2: DM(data_en)=M0; {clear data enable}

{----------------------PID algorithm-------------------------}
{----------------------82 cycles max-------------------------}
```

```
pid:   AX0=DM(cur_pos); {current position}
AY0=DM(run_mode_flag); {get current mode}
AR=PASS AY0;
IF NE JUMP pid1; {if run_mode_flag != 0, staring mode, skip to pid1}
   CALL getspeedt;   {called with AX0=position, returns AX0=Speed}
   DM(set_vel)=AX0;
   AX0=DM(cur_pos); {current position}
pid1: AY0=DM(set_pos); {current setpoint position}
AR=AY0-AX0; {calculate pos error}
DM(cur_error)=AR; {current error}
SR=ASHIFT AR BY 3 (hi); {multiply by 8 to increase gain}
IO(Ddacw)=SR1; {write error to the diag dac}
AR=ABS AR;
AY0=DM(panic_err); {maximum error before panic stop}
NONE=AR-AY0;
IF GE JUMP panic; {do the run_mode_flag check in subroutine}
pid2: AY0=DM(pst_pos);
AR=AX0-AY0; {current velocity}
DM(cur_vel)=AR;
   AY0=DM(set_vel); {velocity setpoint}
AR=AR-AY0;
DM(vel_err)=AR; {velocity error =cur_vel - set_vel}
DM(pst_pos)=AX0;
AY0=DM(p_range); {max error clamp value}
MR1=DM(cur_error);
CALL clamp;
MX1=MR1;
CALL integ; {returns with ki * integral in MR}
MY0=DM(kp);
MR=MR + MX1*MY0 (SS); {kp*clamped error + integral}
IF MV SAT MR;
MX0=DM(vel_err);
MY0=DM(kd);
MR=MR - MX0 * MY0 (SS); {kp * error +ki * integ - kd * vel_err}
IF MV SAT MR;
SE=DM(gain_shift);
SR=ASHIFT MR1  (hi); {multiply by 2^ gain_shift}
SR=SR OR LSHIFT MR0(lo); { to increase gain}
MR1=SR1;
AY0=DM(out_range);
CALL clamp; {returns value in MR0, MR1}
AY0=DM(run_mode_flag); {get current mode}
AR=PASS AY0;
IF NE JUMP afclamp; {if run_mode_flag != 0, stairing mode, skip to afclamp}
next1:  AX0=DM(cur_pos);
     AY0=-30000;
AR=AX0+AY0;
     IF LE JUMP afclamp;
AY0=200;
call clampn;
afclamp:
IO(Cdacw)=MR1; {send the command to the motor DAC}
DM(motor_val)=MR1; {save value for diag purpose}
AY0=DM(set_vel); {increment the command (set) position}
AX0=DM(set_pos);
AR=AX0 + AY0;
DM(set_pos)=AR; {by the set_vel}
RTI;

{#############################################################}
{The PC must clear "buff_full"to 0 after uploading the current }
```

```
{data. buff_full will indicate the scanner position of the data }
{ 91 cycles max ~3 uS }
{this routine does not modify AX1 or AY1... }
{#################################################################}

get_data:
    ENA sec_reg; {primary regs used for PID}
AX1=DM(data_en); {equals 1 when data collect is enabled}
NONE=PASS AX1;
IF EQ jump return2; {data collect is not enabled, return}

AX0=DM(buff_full); {equals 1 when buffer is full}
NONE=PASS AX0;
IF NE JUMP return2; {if 1, buffer is full...return else...}

AX0=DM(run_type_flag); {checking for bounded mode}
AR=AX0 AND bounded_run;
IF NE JUMP bounded_mode;

AR=AX0 AND lane_run; {checking for lane mode}
IF NE JUMP lane_collect_mode;

AR=AX0 AND detect_run; {checking for lane detection mode}
IF NE JUMP detect;

AR=AX0 AND unbounded_run;
IF NE JUMP collect; {checking for unbounded mode}

JUMP bounded_mode; {if run_type isn't specified, go to bounded}

lane_collect_mode:
AR=DM(in_lane);
IF NE JUMP collect;
AX0=IO(posx);
AR=AX0;
    SR=LSHIFT AR BY -1 (lo);
    AX0=SR0;
AY0=DM(I5, M5);
AR=AY0-AX0;
IF NE JUMP return2; {we are not yet at the start position}
AY0=DM(I5, M4);
AY0=DM(num_samples);
DM(in_lane)=AY0;
JUMP collect;

bounded_mode:
AX0=IO(posx);
    AY0=DM(low_position);
    AR=AX0-AY0;
IF LE JUMP return2;
AY0=DM(high_position);
    AR=AX0-AY0;
IF GE jump return2;

collect: {called if we are collecting data at continuously}
AX0=IO(adc0r); {get a/d value}
AY0=DM(adc0_off); {get adc offset}
AR=AX0-AY0; {remove offset}
DM(adc0)=AR; {store value - offset}

AX0=IO(adc1r); {get a/d value}
```

```
AY0=DM(adc1_off); {get adc offset}
AR=AX0-AY0; {remove offset}
DM(adc1)=AR; {store value - offset}

AX0=IO(adc2r); {get a/d value}
AY0=DM(adc2_off); {get adc offset}
AR=AX0-AY0; {remove offset}
DM(adc2)=AR; {store value - offset}

AX0=IO(adc3r); {get a/d value}
AY0=DM(adc3_off); {get adc offset}
AR=AX0-AY0; {remove offset}
DM(adc3)=AX0; {store value - offset}

AX0=IO(posx);
AR=AX0;
    SR=LSHIFT AR BY -1 (lo);
    AX0=SR0;
AY0=DM(new_read_pos);
AR=AX0-AY0;
IF EQ JUMP min_compare;

{----------------------save data--------------------------}
DM(new_read_pos)=AX0;
DM(I4,M4)=AX0; {store pos in array "pos_data"}
  AX0=DM(adc0_min);
    DM(I0,M1)=AX0; {store min in array "ch0_data"}
AX0=DM(adc1_min);
DM(I1,M1)=AX0; {store min in array "ch1_data"}
AX0=DM(adc2_min);
DM(I2,M1)=AX0; {store min in array "ch2_data"}
AX0=DM(adc3_min);
DM(I3,M1)=AX0; {store min in array "ch3_data"}

AX0=DM(adc0);
DM(adc0_min) = AX0;
AX0=DM(adc1);
DM(adc1_min) = AX0;
AX0=DM(adc2);
DM(adc2_min) = AX0;
AX0=DM(adc3);
DM(adc3_min) = AX0;
AX0=DM(in_lane);
AR=AX0-1;
DM(in_lane)=AR;
JUMP prep_next;

min_compare:
AX0=DM(adc0);
AY0=DM(adc0_min);
AR=AY0-AX0;
IF LE JUMP min2;
DM(adc0_min)=AX0;
min2:
AX0=DM(adc1);
AY0=DM(adc1_min);
AR=AY0-AX0;
IF LE JUMP min3;
DM(adc1_min)=AX0;
min3:
AX0=DM(adc2);
```

```
AY0=DM(adc2_min);
AR=AY0-AX0;
IF LE JUMP min4;
DM(adc2_min)=AX0;
min4:
AX0=DM(adc3);
AY0=DM(adc3_min);
AR=AY0-AX0;
IF LE JUMP prep_next;
DM(adc3_min)=AX0;

prep_next:
DM(adc0)=M0; {clear for next iteration}
DM(adc1)=M0; {clear for next iteration}
DM(adc2)=M0; {clear for next iteration}
DM(adc3)=M0; {clear for next iteration}

AX0=I0;
AY0=DM(end_buff);
NONE=AX0-AY0; {test for end of buffer}
IF LT JUMP return2; {return if buffer not full else...}
DM(buff_full)=M1; {set buff_full flag to 1 ,M1=1}
    DM(data_en)=M0; {clear data enable flag}
I0=^ch0_data; {ch0_data buffer for adc0}
I1=^ch1_data; {ch1_data buffer for adc1}
I2=^ch2_data; {ch2_data buffer for adc2}
I3=^ch3_data; {ch3_data buffer for adc3}
I4=^pos_data; {pos_data buffer }
return2:
DIS sec_reg;
RTI;

detect:
AX0=I0(posx);
    AY0=DM(low_position);
    AR=AX0-AY0;
IF LE JUMP return2;
AY0=DM(high_position);
    AR=AX0-AY0;
IF GE jump return2;

collect_ld: {called if we are collecting data at continuously}
AX0=I0(adc2r); {get a/d value}
AY0=DM(adc2_off); {get adc offset}
AR=AX0-AY0; {remove offset}
DM(adc2)=AR; {store value - offset}

AX0=I0(posx);
AR=AX0; {lower resolution by factor of 2}
    SR=LSHIFT AR BY -1 (lo);
    AX0=SR0;
AY0=DM(new_read_pos);
AR=AX0-AY0;
IF EQ JUMP min_compare_ld;

{----------------------save data--------------------------}
DM(new_read_pos)=AX0;
DM(I3,M1)=AX0; {store pos in array "ch3_data"}
AX0=DM(adc2_min);
DM(I0,M1)=AX0; {store min in array "ch0_data"}
```

```
AX0=DM(adc2);
DM(adc2_min) = AX0;
JUMP prep_next_ld;

min_compare_ld:
AX0=DM(adc2);
AY0=DM(adc2_min);
AR=AY0-AX0;
IF LE JUMP prep_next_ld;
DM(adc2_min)=AX0;

prep_next_ld:
DM(adc2)=M0; {clear for next iteration}

AX0=I0;
AY0=DM(end_buff2);
NONE=AX0-AY0; {test for end of buffer}
IF LT JUMP return2; {return if buffer not full else...}
DM(buff_full)=M1; {set buff_full flag to 1 ,M1=1}
    DM(data_en)=M0; {clear data enable flag}
I0=^ch0_data; {ch0_data buffer for adc2}
I3=^ch3_data; {ch3_data buffer for pos}
JUMP return2;

{################################################}
{ HOME     25 cycles     }
{################################################}
home: AX0=DM(first_pass);
none=PASS AX0;
IF NE jump home1;
IFC=0x10; {clear pending irqe}
DM(home_en)=M1; {set home enable flag}
DM(first_pass)=M1; {set first_pass to 1}

home1: AX0=DM(home_flag); {test for home flag}
none=PASS AX0;
IF EQ RTS; {not set? continue looking}

AY0=DM(home_pos); {save home flag position}
AX0=DM(cur_pos); {cur_pos was reset at home _pos}
AR=AX0+AY0; {get home delta}
DM(home_delta)=AR;
AY0=DM(set_pos);
AR=AY0-AR;
DM(set_pos)=AR; {new set position for home=0}
AY0=DM(home_delta);
AX0=DM(pst_pos);
AR=AX0- AY0;
DM(pst_pos)=AR; {new past position for home=0}

AX0=DM(home_dis_mask); {finished homing}
AY0=DM(control_reg);
AR=AX0 AND AY0; {clear home bit}
AR=AR OR 0x10; {set homed flag}
DM(control_reg)=AR; {write to control reg }

DM(home_en)=M0; {clear home enable flag}
DM(first_pass)=M0; {clear first_pass to 0 for next home}
RTS;

{################################################}
```

```
{      INTEGRATE    }
{---------integrate and clamp subroutine ----------}
{-19 cycles + clamp( 9 cycles max) = 28 cycles max-}
{---------------the input is DM(I_in)--------------}
{---the routine...out=out+ki*(.5*in +.5*past in)---}
{################################################}

integ: {integrate the error}
MX0=DM(pst_error);
MY0=0x4000; {0.5 }
MR=MX0 * MY0 (SS); {.5 times past input}
MX0=DM(cur_error);
MR=MR +MX0 * MY0 (SS); {input + (past input * .5)}
DM(pst_error)=MX0; {save for past error}
MX0=MR1;
MY0=DM(ki); {and set gain. ki=ki(s) * t }
MR1=DM(I_out_hi); {get past output}
MR0=DM(I_out_lo);
MR=MR + MX0*MY0 (SS); {add ki * (error +pst error)/2 to output}
IF MV SAT MR;
AY0=DM(I_range);
CALL clamp; {clamp output to I_range}
DM(I_out_hi)=MR1; {save current output for next iteration}
DM(I_out_lo)=MR0;
RTS;

{*************************************************************}
{ CLAMP }
{*************************************************************}

clamp: {-----------------clamp the value to range--------------}
{9 or 5 or 7 cycles}
{--if |in|<= clamp val, out=in else out = +- clamp val -}
{Call with value to clamp in MR, and clamp value in AY0.}
{returns with Clamped value in MR. }

AR=PASS MR1;
IF GE JUMP testp;
AR=MR1 + AY0;
IF GE JUMP done;
AR= -AY0;
MR1=AR;
MR0=0;
JUMP done;
testp: AR=MR1 - AY0;
IF LT JUMP done;
MR1=AY0;
MR0=0;
done: RTS;

clampn: {-----------------clamp the value to range--------------}
{9 or 5 or 7 cycles}
{--if |in|<= clamp val, out=in else out = +- clamp val -}
{Call with value to clamp in MR, and clamp value in AY0.}
{returns with Clamped value in MR. }

AR=PASS MR1;
IF GE JUMP done2;
AR=MR1 + AY0;
IF GE JUMP done2;
AR= -AY0;
```

```
MR1=AR;
MR0=0;
done2: RTS;

{################ spinner initialization ######################}
spin_init:

AX0=IO(Posx); {get the current position}
DM(set_pos)=AX0; {set set_pos to current pos}
DM(pst_pos)=AX0;
DM(pst_error)=M0; {set to 0, M0=0}
DM(I_out_hi)=M0; {set to 0, M0=0}
DM(I_out_lo)=M0; {set to 0, M0=0}

AX0=DM(mtr_ena_mask); {enable mask (0001); 0x01 }
AY0=DM(pf_data); {enable the motor driver(set bit 0 of pf_data)}
AR=AX0 OR AY0;
DM(pf_data)=AR;
DM(spinner_init)=M1; {set spinner_init flag to 1, M1=1}
RTS;

{#########################################################}
start_convert: {SPORT0 RX interrupt}
reset FL0; {set read/convert line lo (convert)}
NOP; {32 nS wait}
IO(start_conv)=AX0; {start convert...write to a/d}
set FL0; {set read/convert line hi (read)}
RTI;

{################ home interrupt service ##########################}
home_int:
AX1=DM(home_en);
none=PASS AX1;
IF EQ RTI;

DM(home_flag)=M1; {IRQE interrupt, enabled for homing}
AX1=IO(Posx); {set home pos flag}
DM(home_pos)=AX1; {save home position}
AX1=0;
IO(Posx)=AX1; {set position to 0}
RTI;

{###################### panic ##########################}
panic:
AY0=DM(run_mode_flag); {get current mode}
AR=PASS AY0;
IF NE JUMP pid2; {if run_mode_flag != 0, stairing mode, skip to pid2}
AX0=DM(control_reg);
AY0=0xf8;
AR=AX0 AND AY0; {clear the run,data and home bits}
AR=AR OR 0x20; {set the panic stop bit}
DM(control_reg)=AR;
JUMP loop_t; {return to start of interupt}

{################### lane detection ######################}
lane_detect:
DM(buff_full)=M0; {clear buffer full}
AX0=DM(lane_dis_mask);
AY0=DM(control_reg);
AR=AX0 AND AY0; {clear lane detect bit}
AY0=DM(run_mask);
```

```
AR=AR OR AY0; {set run bit}
AY0=DM(data_mask);
AR=AR OR AY0; {set data bit}
DM(control_reg)=AR; {write to control reg}
AR=detect_run;
DM(run_type_flag)=AR; {set run_type to detection}
RTS;

{------------------ speed table lookup ---------------------- }
getspeedt:
MX0=AX0; { AX0 contains current position }
SI= MX0;
L5=0;
I5=^lookuptable;
SR= LSHIFT SI BY -4 (LO);

AY0=SR0;
M5=AY0;

{The following lines ensure it will not stall if speed=0}
AX0=20;
AY0=DM(cur_vel);
AR=AX0-AY0;
AX0=60;
IF GE RTS;

AX0=PM(I5,M5); {move pointer to location M5 }
AX0=PM(I5,M5);  {read value at I5[M5] }
AR=AX0-AY0;     {AR = Set speed - Actual speed}
AY0=100;
AR=AR-AY0; {AR = Speed Diff - 100}
IF LE RTS;   {Speed difference < 100}
AX0=DM(cur_vel);
AR=AX0+AY0;
AX0=AR;
RTS;

.ENDMOD;
```

Appendix B

Housekeeper Input File Specification

```
Input File specifications for HouseKeeper.exe

The format of the file is as such:
device: <devicetype>       <# of devices>       <init file name>
device: <devicetype>       <# of devices>       <init file name>
....
....

events:
<time>       <devicetype>       <device no.>       <action>      <params>
<time>       <devicetype>       <device no.>       <action>      <params>
<time>       <devicetype>       <device no.>       <action>      <params>
.....
.....
```

The words in bold are the keywords that should be written exactly as shown. The keywords in <...> are variables whose formats and possible values are listed below. The keywords and variables can be tab-delimited or they can simply be separated by whitespace. They must however be separated by at least one space.

Each device type should be specified on a separate line beginning with the ''device:'' keyword. It should then be followed by the number of such devices will be used. A separate line must be entered for each different device type that will be used. Once a device type has been listed, it should not be listed again. If there no instances of a particular type of device, the device should not be listed at all. Following the number of devices <init file name> should be specified if it exists. This is the name of the initialization file for the associated <devicetype>. The file name should be relative to the directory in which

the HouseKeeper class resides or absolute.

After all the devices are specified, the events should be listed. Once the ''events: '' keyword is specified, the program will expect the rest of the file to contain event specifications. Anything other than events listed after the ''events: '' keyword to the end of the file will cause the file to be parsed incorrectly and cause the program not to run properly (most likely by throwing an exception during execution).

The events should be listed in chronological order. The program looks to execute the events in the order that they are read in, so if an event is out of order, this could halt execution of all events after the out of order event. The device number refers to the unique id that is associated with each device. It will commonly be a consecutive numbering starting from the device directly attached to the last device connected in a chain, if the devices are daisy-chained to the computer. The action that is associated with an event should be a valid one for that particular device type. A listing of valid actions are given below. The parameters associated with an action should be listed after the action. If an action does not have a parameter, none should be listed. If the action needs one or more parameters, they should be listed after the action and separated by spaces. There should be a carriage return after the last parameter is entered.

Variables and Formats:

devicetype:
there are only 4 valid devices at this time: harvard and masterflex and xyzrobot and valve.

# of devices:
an integer value that reflects the number of devices of the type specified.

Init file name:
the name of any initialization file required by the device.

time:
a time for this event to be executed. the format is HH:MM:SS where HH is a two digit value for the hour, MM is the two digit value for the minute and SS is the two digit value for the second. If the value is a single digit, a leading zero should be inserted. The hour value ranges from 0-23, the minute value ranges from 0-59 and the second value ranges from 0-59.

action / params:
- for the harvard devicetype, the valid actions are:
start - no parameters

```
stop - no parameters
setdir - has 1 parameter, direction.  This parameter is
   just a string, ''infuse'' to set the pump to infusion
   and ''refill'' to set the pump to refill.
setinfrate - has 2 parameters, rate and units  The format
   for the rate is nnnnnn, a numerical value with 5
   digits and 1 decimal point (10.000 or 3.00000 e.g.).
   There are 4 possible units of measurement which are
   as follows:
ul/mn - microliters per minute ul/hr - microliters per hour
ml/mn - milliliters per minute ml/hr - milliliters per hour
setrefrate - has 2 parameters, rate and units.  The format
   is the same as detailed above.
changemode - has 1 parameter, mode.  This parameter is a
   string that can be set to either  ''pump'', ''volume''
   or ''program''.

- for the masterflex devicetype, the valid actions are:
start - no parameters
stop - no parameters
setrevs - has 1 parameter, revolutions.  This parameter has
   the format nnnnn.nn, a numerical value with 2 significant
   digits and a maximum value of 99999.99.
setvel - has 1 parameter, velocity.  This parameter has the
   format  [+/-]nnnn.n, a signed numerical value with 1 significant
           digit and a maximum value of 9999.9.  A positive value specifies
   a clockwise rotation and a negative value specifies a
   counterclockwise rotation.

- for the xyzrobot device type, the valid actions are:
write -  1 parameter.  Command to write to the 6K controller.  For
           command listing, see 6K Series Command Reference
start - no parameters

- for the valve devicetype, the valid actions are:
open  - no parameters
close - no parameters
```

Bibliography

[1] U.S. Department of Energy Human Genome Program. Primer on Molecular Genetics. Washington, D.C., 1992.

[2] U.S. Department of Energy Human Genome Program. To Know Ourselves, July 1996.

[3] Leon Jaroff. Venter's Bold Venture. *Time*, 151(25), June 1998.

[4] Bob Sinclair. Sequence or Die: Automated Instrumentation for the Genome Era. *The Scientist*, 13(8):18, April 12 1999.