PINS: A Haptic Computer Interface System

by

Bradley C. Kaanta

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 13, 2004 [June 2004]

© 2004 Bradley C. Kaanta. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author [illegible]
Department of Electrical Engineering and Computer Science
May 13, 2004

Certified by [illegible]
Hiroshi Ishii
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 0 2004
LIBRARIES

ARCHIVES

PINS: A Haptic Computer Interface System
by
Bradley C. Kaanta

Submitted to the
Department of Electrical Engineering and Computer Science

May, 13 2004

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The research goal was to develop a dense array of discreet vertical actuators as an input and output device with haptic feedback for Human Computer Interaction (HCI). This expands upon the current research of table surfaces as medium for HCI by adding a third dimension that both a user and a computer can control. The use of vertical actuation makes possible new kinds of physical interactions with virtual objects and allows a computer to maintain constancy with the physical representation and the digital information. This requires the design and constructions of an elegant, reliable, and economically reasonable actuator array. Each array element requires autonomy to quickly and accurately move to a precise height. As an array, combined elements must provide enough resolution so that the user perceives the array as a continuously morphing, three-dimensional surface. Shape transformations are accomplished either indirectly by digital means or directly by user touch. The proposed research will focus on development of a real-time haptic actuation arrays supporting technology. The process includes working on the design, function, appearance, response, and implementation.

Thesis Supervisor: Hiroshi Ishii
Title: Professor of Media Arts and Sciences, MIT Media Laboratory

Acknowledgements

I would like to thank Jason Alonso for all the time he spent helping with the computer side software, his help developing the computer interface for PINS, and his obsessive mono-phobia. I also like to thank Chad Dyner for his effort in helping to design some of the mechanical system and its implementation. Most of all, I would like to thank Professor Hiroshi Ishii for his support and encouragement.

Table of Contents

Table of Figures

1 Introduction

Human Computer Interaction (HCI) is how people interact with computers. The overwhelmingly predominate mode of this interaction is with the Graphical User Interface (GUI). Despite the dominance of the GUI, it is generally agreed that the GUI is not always the most efficient way to work with a computer. The development of the Tangible User Interface (TUI) attempts to build upon existing human skills with everyday physical objects to make the use of computer applications intuitive. This research goal was begun by a project "Tangible Bits" [Ishii 1997] was to extend HCI beyond the traditional Graphical User Interface (GUI) and instead use physical objects to represent digital information.

1.1 A Need for Movement

The desire to touch and feel a virtual world has lead researchers to continually try to develop haptic interfaces. A haptic interface is a mechanical device that attempts to physically represent a virtual object with a physical-reaction-force. Haptic interfaces allow computer-modeled constraints to be enforced in a way people can experience, beyond just graphics and sound.

1.2 Missing Movement in Tangible Interfaces

In general, physical outputs have been underutilized in Tangible User Interfaces. Most computer interfaces respond asymmetrically to the caress of a human hand with only images and sounds.

This incongruity of input and output modes arises naturally from the fact that visual and auditory sensations are relatively easy to synthesize while tactical feedback is complicated. Complexity is greater because a sense of force is inseparable from real physical contact. The missing ability of the digital system to affect the physical world can make the tangible objects feel more like pointers to data and less like physical manifestations of the data itself. Some systems like Illuminating Clay [Piper 2002] use passive objects, like moldable clay, do link the input and the output creating physical and digital consistency, but the computational system cannot control the physical output. Other projects such as inTouch [Brave 1998] and Actuated Workbench [Pangaro 2003] map physical input to a physical output, but do so in a limited manner. These systems are restricted to only one and two degrees of freedom (DOF) respectively.

To be fair the DOF of these system were limited on purpose to reduce complication while exploring their research goals.

1.3 Unnatural Haptic Feedback

Generally, haptic systems that try to give force feedback from virtual information do not build from natural, everyday, human handling skills. The commercially available haptic feedback device PHANToM [Massie 1994] requires the user to make the mental leap between the image on the screen and the movements they are making with the device. This is not a true Tangible User Interface. Instead, the PHANToM-like interfaces actually add another level of abstraction between computer and user because a lot of practice is required to lean to use these devices.

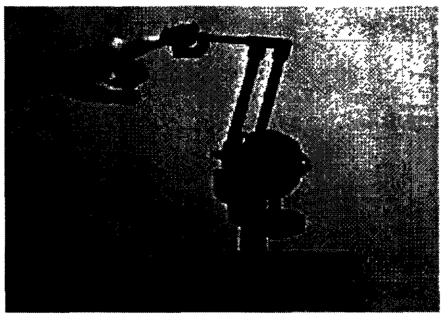


Figure 1.1: When this PHANToM haptic interface is connected to a desktop computer, it lets the user feel the physical characteristics of the virtual objects seen on the monitor[1].

1.4 Natural tangible interface

An example of a natural tangible interface is the Illuminated Design Environment [Piper 2002], which uses modeling clay and a laser scanner to upload the shape of a surface into a computer. This allows designers who normally work with clay to use their professional skill working in their normal medium while simultaneously using the power of computers to analyze their designs. However, there is no mechanism for the computer to do anything but passively observe the interface.

1.5 Best of Both Worlds

[1] ***Photo courtesy of SensAble Technologies, Inc***

The goal of PINS (haPtic Intuitive N-scalable System) is to build the framework for a more ideal interface, which will be a haptic feedback system that is both physically intuitive and provides tangible access to the virtual world.

Figure 1.2: Concept Picture of a top view of PINS system by Chad Dyner

The display component of PINS comprises of a densely packed array of discrete nodes or elements operating as both a physically and computationally deformable surface. The geometric representation allows visualization of both scalar and relational changes of complex form over time. A tangible overlaid input component allows users to manipulate and mold surfaces using their hands directly on PINS' surface. Similarly, the surface can be responsive to solid shapes (i.e. object placement on the PINS surface) thereby serving as a sensor field to capture ambient and physical variations.

This chapter will describe the conceptual framework behind PINS along with some related work. Chapter 2 will introduce several versions of the mechanical and computational forms PINS has grown through. Chapter 3 will examine the hardware development of PINS and will be followed by a discussion of the microcontroller firmware in Chapter 4. Chapter 5 describes the assembly of the PINS system. Chapter 6 discusses the accessories PINS uses. Chapter 7 outlines several areas of future work. Finally, Chapter 8 contains the conclusions from the research.

1.6 Related Work With Haptic Interface Systems

There are five main categories for methods of implementing a haptic interface: The tool-handling system, the exoskeleton force display, two dimension tabletop systems, skin haptics and the deformation display.

1.6.1 Tool-Handling Systems

Haptic tool-handling systems are by far the most common and commercially available. This haptic implementation allows a user to hold a physical tool and explore a digital realm. The tool often has a force feedback system built in giving tactile data to the user about a virtual world. The PHANToM pantograph style device is perhaps the most famous with 3-degrees of freedom (DOF). The user places a fingertip in the grip of a thimble at the end of a gimble, allowing for the 3 DOF force feedback to be applied. [Massie 1994].

A more mainstream version of tool-handling haptic systems is the force feedback joystick. Examples are the Microsoft Sidewinder Force Feedback 2 Joystick and the Saitek J45 Cyborg 3D Force Stick. Both of these systems help pass information and a sense of realism about the digital environment to the user. The joysticks, however, do not allow for exploration of a 3 dimensional space. They only allows the user to probe the space directly adjacent to and avatar.

1.6.2 Exoskeleton Force Display

The exoskeleton force display consists of a set of actuators attached to the hand or around the body. These powered exoskeletons are designed to follow, guide and constrict a users movements. These systems have traditionally required extensive amounts of hardware, and are often used in robotics as the master controller of teleoperations. However, due to their size and complexity, these systems can be extremely expensive and are often specialized to a specific task. These systems also have to be fitted and attached to the user. A few groups have been able to make small lightweight exoskeletons for the fingers or hand. One such commercially available product is Cyber Grasp, which is a glove system that uses cables to transmit force [Immersion Corp, 2003].

1.6.3 2D Tabletop System

The movement of objects on a flat surface is a problem that has been studied for both HCI applications and industrial use, such as shipping and sorting packages. Early systems such as Seek [Negroponte, 1970], used robotic arms to arrange parts on a table. In a strict sense, this is not haptic feedback, however, it was the first step towards HCI that allowed the computer to effect the physical world. More recently, tabletop systems have been developed that attempt to move objects without robotic arms. The psyBench [Brave 1998] was a prototype from the MIT Media Lab that moved magnetic, computerized, chess pieces using an under the table x-y plotter and electromagnets. This led to the development of the Actuated Work Bench, also developed at the MIT Media Lab [Pangaro 2002]. This system was composed of an array of electro-magnets which could push and pull magnetic object around a flat surface. This system was a vast improvement over psyBench in performance because it could manipulate more than one object simultaneously. However, the user can easily overpower the force applied to the puck objects that the system manipulates. This meant that there is no firm constraint enforcing the virtual boundaries. Also, the Actuated Work Bench can only move small magnetic objects on a 2-D surface and has no way of implementing a third dimension.

1.6.4 Skin Haptics

Tactile displays that stimulate skin sensation are also relatively well explored. Angela Chang, along with the MIT Media Lab, developed a system call ComTouch, which uses the vibro-tactile effect to transmit the feeling of touch between two physically separated users [Chang 2002]. In addition, micro pin arrays have been used to help blind people read and communicate by providing a slightly raised surface [G. Moy, 2000]. The micro pin array has the ability create the texture of a 2-D surface, but the length of the pins is not long enough to create the feeling of a 3-D space.

1.6.5 Deformation Displays

The deformation display is a radical departure from the other methods of haptic displays. Instead of being able to probe and explore a point of the virtual object, as with PHANToM, or moving a finite object over the surface, as with Actuated Workbench, the deformation display actually recreates a physical model of the data being represented. The user can then be in physical contact with this model. One example of this approach was called "Robotic Graphics" [McNeely, 1993] This method involves measuring the position of finger tips of the user and then having a shape approximation prop mounted on a manipulator move to provide a contact point for what represents the virtual object. In this way the system creates a moving barrier that keeps the users hand from entering what is considered solid in the virtual space. Another example is FEELEX which is a true attempt to make a haptic surface [Iwata 1997]. FEELEX uses a 6 by 6 array of vertical actuators to deform a rubber membrane with graphics projected onto this surface. The system's major limitation is the coarseness of the 6x6 actuator

array. There is also the limitation associated with having a membrane smoothing the output surface of the device. The membrane limits the maximum delta between the adjacent actuators, thus limiting the steepness of displayed slopes.
The MATRIX from the MIT Media Lab is another vertical pin device, but this one is passive relying only on force sensors [Overholt 2003]. This passive device is another approach to the deformation display, allowing for deformation of the input surface, but since the device has no actuators it cannot represent the shape of a virtual object. Also the goal of the MATRIX is to explore the use of a linear array as a musical instrument rather than as a generalized I/O device.

2 Evolution of the Design

2.1 Mechanical System Approaches

Several different design approaches were used in the development of PINS. These all varied mainly in the mechanical elements of individual actuators. Different approaches tried included, analog motorized potentiometer, repulsion coil, screw actuator, and rack and pinion system. The goal was to develop a simple, reliable single pin design that could be easily reproduced. From this point forward, an *actuator* will refer to the combined electrical and mechanical subsystem required to move one pin, and represents an abstracted self-contained interaction unit.

2.1.1 Analog Motorized Potentiometer

This was the first mock up of a pin element, and helped greatly with figuring out what the important design requirements were. The system was simply a motorized linear potentiometer. The voltage on the wiper of the pot was used to provide feedback to control the height of the system.

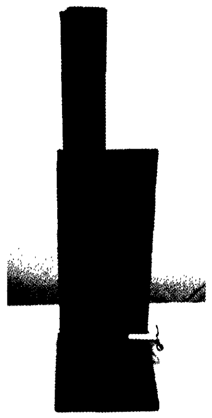


Figure 2.1: Analog Motorized Potentiometer PIN

2.1.1.1 Analog Hardware Implementation

First the system was implemented using an analog hardware PID controller. The input was set by a digital to analog converter from a PIC microprocessor. The PIC does not have a true analog output but by using the chips pulse width modulated output and an appropriate low pass filter (an example of an active low pass filter is shown below), a very good D/A converter can be created. The analog output is approximately the duty cycle of the input times the supply voltage, assuming you are using a rail-to-rail operational amplifier.

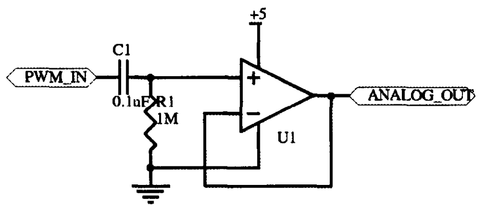


Figure 2.2: Generating an analog voltage from the PWM signal

The voltage follower is added to the analoge output to ensure that the output has low impedance and does not load circuits connected to it. The time constant of the ciruit is set to be much slower the the PWM input signal but much faster then the mechanical timeconstant of the system.

The mechanically driven wiper on the pot is moved to try to match the analog output from the controller. Unfortunately, this system required several operational amplifiers and a power driver output. The power driver alone was fairly complex. The transistor amplifier built to drive the motor required a positive and negative voltage supply.

Complimentary bipolar junction transistors act as voltage controlled current sources to push and pull current though the motor as required. There is an operational amplifier feedback path tied around the transistors to correct for diode drops, and resistive loss in the load. The gain for the whole amplifier is controlled by a potentiometer, in the op-amp feedback path.

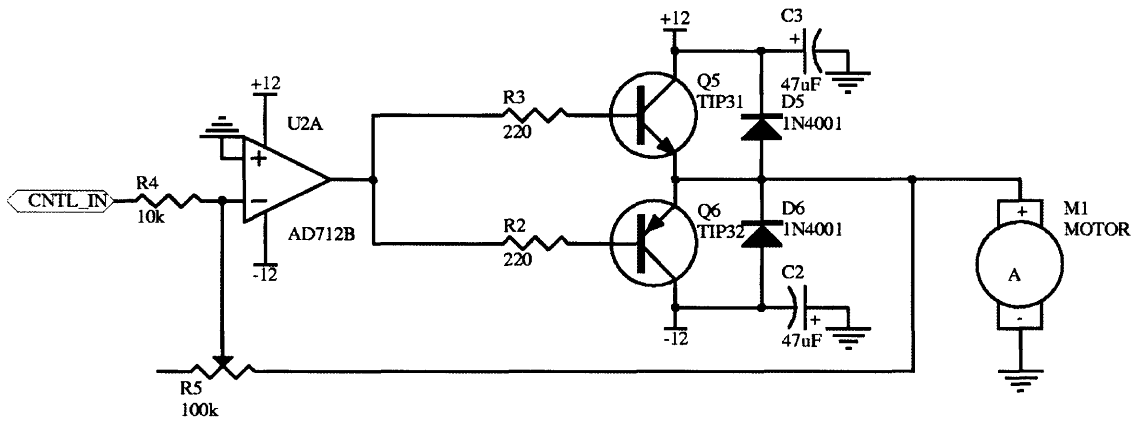


Figure 2.3: Power amplifier circuit

This is a lot of hardware for each pin. Another alternative would be to use a monolithic high power operational amplifier designed to drive high current loads. However, these op-amps are about $10 each putting them out of the price range that would be acceptable for the construction of hundreds of pins. Also the construction and the settings for the PID controller were not adjustable with out physically changing components, this was definitely not acceptable when hundreds of pins were going to be built. Also the cost of the individual analog components with out a power op-amp (low power op-amps, power transistors, diode all about 50 cents each) would add up to make the cost of mass production prohibitive.

2.1.1.2 Digital Firmware Implementation [2]

In an effort to reduce overall part count and simplify design, the controller was implemented in firmware on a Microchip PIC16F876 microcontroller. This component sampled the motorized potentiometers wiper with a 10 bit analog to digital converter, a built in component of the PIC16F876. The value from the A/D was compared to a desired value set by the user, and fed though a PID routine. This routine set a 10-bit pulse width modulated signal, which was used to diver a dual H-bridge motor diver,

[2] components and concepts mention in this section are explained thoroughly in Hardware Implementation section

which drove the motor attached to the potentiometer. This required far fewer components, and the control loop was adjustable by changing firmware variables. This was preferable then having to physically replace capacitor and resistor values to change performance.

2.1.1.3 Key Learnings from Motorized Potentiometer Actuator

This actuator design had many nice features. Most importantly it was satisfying to touch. If you pushed it from its desired position it would "spring" back to place quickly and smoothly. You could also feel the pin pushing against your hand if it was out of place. The ability to back drive the system, and thus deform its shape was one of the original design requirements and was met well by this actuator.

However, the footprint of a single motorized potentiometer was much too large for the final system, almost 2 inches by 1 inch. Also, the motor was not designed to push any load, and would quickly heat up. Under long-term uses the motor would most likely burnout. When maintaining a set height the motor would sometimes continually draw a current, causing the motor to overheat, this is because it was designed for uses on a flat surface. Also the mechanical carbon contacts of the resistor would wear out fairly quickly. All of these factors made the motorized pot unsuitable for use in the final design.

2.1.2 Linear Screw Actuator

The linear screw actuator was developed in several iterations. The goal of the screw actuator was to solve some of the problems of the motorized potentiometer, like the constant power consumption, inconsistency and wear on the position sensing elements. The screw actuator made it so that a platform was moved up and down when the rod turned but was locked in place when no power was applied to the actuator. So, it took zero power to maintain a position. The position sensing was done with an optical encoder. Because the optical encoder had no physical contacts this height measurement system would not wear out with use.

Figure 2.4: First Screw Actuated PIN

A problem with this system was that a user could not back drive the system. Pressing on the top of the actuator will not move it. To address this problem a sensor was added to the top of the pin. At first a push button was installed, so when the button was pushed the pin moved down. This push button was replaced with a capacitive sensor (discussed in section 3.4.1) this was much more astatically and tactilely pleasing then a button. A method of "pulling" the actuator up was never satisfactory solved. Another problem was that the screw actuator could be noisy if the threaded rod was out of alignment with the actuator tube.

2.1.2.1 Multiple Pin Setups

The first multiple pin setup was a large (two square inches per actuator) wooden monstrosity, but the circuitry used to drive this system was put onto a PCB to make the first small-scale functional version of pins. This original system was accurate to one thirty-second of an inch. Later smaller versions were accurate to a fifty-sixth of an inch and used optical encoders for position control.

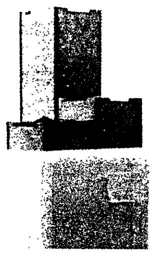


Figure 2.5: First multiple pin mockup of PINS

Using these multiple pin setups the first communication scheme involving several microcontrollers was setup. Placing the control circuitry on a PCB board made it so boards with one microcontroller could easily be plugged into a connecting strip. This modularity allowed by the PCB board made testing and assembly fairly easy. This version of PINS used one microcontroller to control two mechanical actuators.

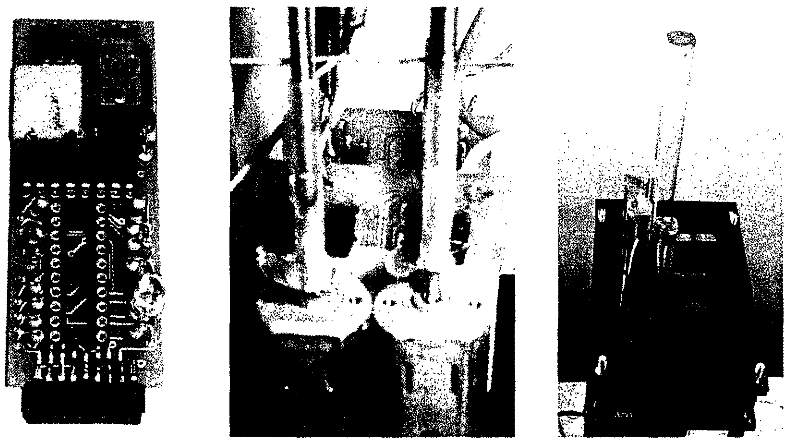


Figure 2.6: First PCB mockup of PINS

2.1.2.2 Key Learnings from Screw Actuator Pin

The screw actuator pins were very accurate and had relatively low power consumption. However, recreating the tactile experience of a material that deformed under the pressure of your hand was hard to simulate. This was mostly because of the limited speed at which the actuator could move. People describe it as "unnatural" and "very mechanical" feeling. There was also no way to pull the actuator up.

2.1.3 Linear Induction Actuator

One of the main problems with any of the actuators examined so far is that they involved moving parts, gears, ect. These parts could get jammed, wear out and break. The goal with testing developing the linear induction actuator was to reduce the moving parts to one, the actuator, and to make it have no physical contact except with a guide rod. The principle behind the linear Induction Actuator is similar to that behind a Thompson Ring [Mak, 1986]. A changing field passing through the center of a ring induces a current in the ring. This current now flowing through the ring creates it own field but 180 degrees out of phase with the original field. These two fields repel each other. By running alternating current of various magnitudes through the field inducting coil the aluminum rod floats guided by the ferrite rod.

This actuator was back drivable and very quick to respond to changing inputs. However, the power consumption increased exponentially as the floating aluminum tube was pushed higher. This would make holding a whole array of actuators at a constant height very difficult on the power supply. Also, because the test system was run using mains and a variable transformer, you could feel a 120 Hz vibration on the aluminum rod when you pressed down on it. The frequency was 120 Hz instead of 60 Hz, which is the frequency of mains power lines in the USA, because the direction of the changing current does not matter only the delta over time.

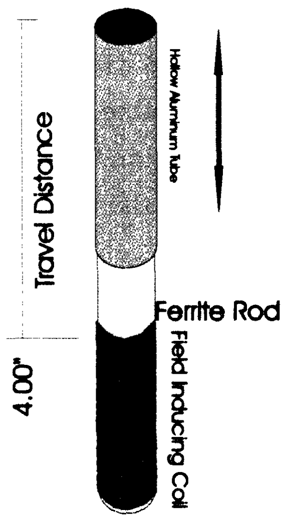


Figure 2.7: Linear Induction Actuator

Although I feel like this approach had the best chance of being successful in the long term as a durable actuator for PINS and because the footprint could be miniaturized increasing the resolutions of PINS. However, the immediate problems of working out the electro dynamics, optimization for minimal power consumption, the interactions between closely packed pins and methods of feedback made this problem seem like it was suitable as a thesis topic of its own. So, this approach was unfortunately abandoned for the current development of PINS.

2.1.4 Dynamic Alloys

Another method of actuation that was looked into were dynamic alloys such as Mondo Tronics "Muscle Wires" or Dynalloy Inc's "Flexinol." These wires are made of nickel titanium and dynamically change their internal crystal structure at certain temperatures. The heating of these alloys can be achieved by running current through them allowing you to electrically control the deformation of the metal. This change in size happens smoothly and quietly. Unfortunately, this wires only contracts at most 4 percent, so would require it either 5 feet of wire per actuator or a complicated mechanical system to harness the power to get the amount of movement required. There may be a simple elegant way of using these materials but it was not pursued as part of this research.

2.1.5 Rack and Pinion System

The system used with the final version of PINS was a rack and pinion setup. The mechanical portion of this system was constructed with the help of Chad Dyner. A small gear was place on the axel of the motor, and it meshed with a photo etched rack gear. This translated the rotational energy of the motor to linear movement.

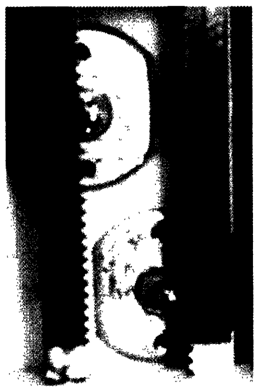


Figure 2.8: Rack & Pinion Linear Actuator Setup

The small size of the motor's gear harness the motor's power and keeps it from hitting its stall torque. Stall torque is the amount of power the motor can apply when it is not moving or just starting to move. This system moves quickly, and can be back driven by the user. It has less accuracy in reaching a set height then the screw actuator but is still accurate within a 16th of an inch. The control for this system was also very easy. The system was naturally critically damped, so the microcontrollers where set up to run a bang-bang control scheme, this allowed the processor to run with less overhead allowing it to better handle dynamic changes in the system.

A system of stacking the motors and routing the racks made this by far the densest number of pins per inch of any of the earlier versions developed.

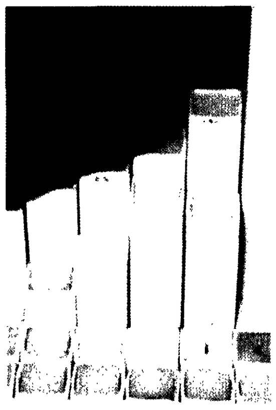


Figure 2.9: Densely packed pins from rack & pinion actuator

Unfortunately, during extensive testing we also discover that the racks destroy the motor gears if there is any slop in the connection. However, while working this was the best version of pins developed.

2.2 System Communication Approaches

Three different approaches were attempted to facilitate communication between the PINS modules and the control computer. These involved different methods of serial communication, and the development of firmware and software libraries.

2.2.1 Single Actuator Communication

The first single actuators tested, used the RS-232 serial communication standard. Since there were many pre-developed libraries for computers and for PIC microprocessors this was the fastest way to set up communication between the microprocessors and a computer.

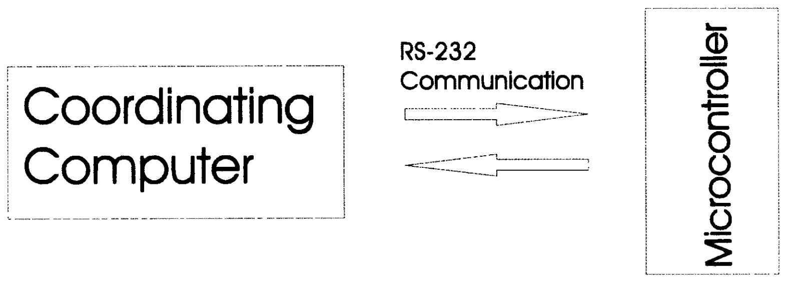


Figure 2.10: Single Actuator Communication

2.2.2 First Stage Multiple Actuator Communication

The first system that allowed for multiple micro controllers to communicate with one computer involved using a combination of RS-232 and 3 wire SPI communication.

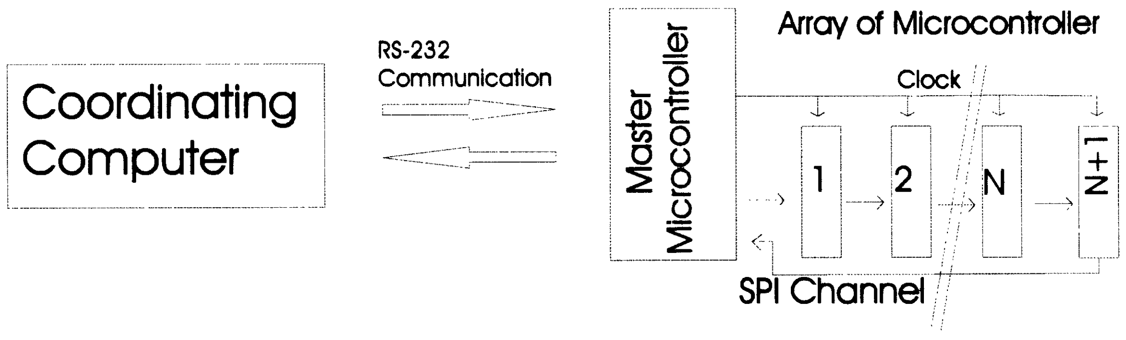


Figure 2.11: Multiple Actuator Communication

In this system the Master microcontroller would receive an array of numbers, the array size equaling the number of microcontrollers in the system. This array would be shifted into the shift registers in the microcontrollers. Once the data was in the microcontrollers internal shift registers the data could be latched and processed.

This implementation was used because there were existing libraries for RS-232 communication between the computer and master microcontroller, and built in functions for SPI between the master microcontroller and slave components.

2.2.3 Improved Multiple Actuator Communication

To simplify the system libraries were developed that allowed for the parallel port of a PC to be used to communicate directly with the array of microcontrollers. This simplified the system making it faster and more reliable.

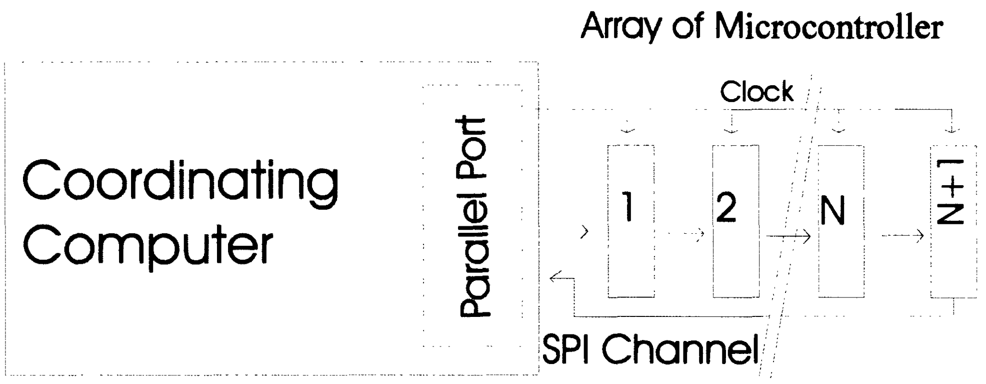


Figure 2.12: Improved Multiple Actuator Communication

Since more of this system was implemented in software, changes and improvements in the system became easier.

2.2.4 AVR X 16: Enhanced Multiple Actuator Communication & Programming

The AVR X 16 is a control block that allows for the coordinating computer to dynamically change the configuration of the connections of the microcontrollers. It can connect them all in a long serial string or select out a signal microcontroller. When a single microcontroller is selected out, it can have its configuration bits set and can be programmed.

Earlier systems required physically removing the microcontrollers; placing them in a programmer, installing the new program, and then replacing the chip into the system. This would take at least 2 minutes. This would be completely infeasible for an array of hundreds let alone thousands of microcontrollers, which is a possible future goal for the PINS project.

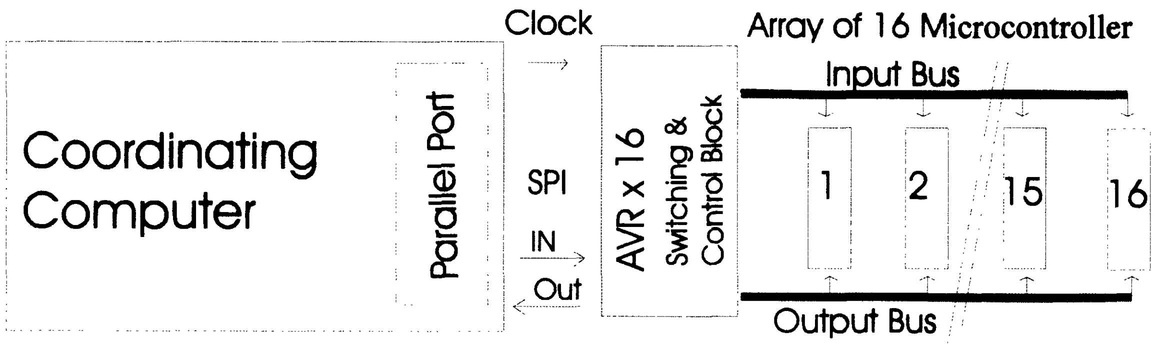


Figure 2.13: Enhanced Multiple Actuator Communication & Programming

With the AVR X 16 system it takes about 1 second to program a microcontroller and once the system starts running it does not require any human attention to process all the chips in the device. This system would not only be useful in the PINS project but any project requiring distributed microcontroller networks.

3 Hardware Implementation

3.1 Overview

The PINS system can be represented with a simple block diagram. There were actually 2 different PINS systems developed during the course of the project. These were the screw actuator system and the pinion actuator system. The difference in the block diagrams of theses systems comes from how the users physical input effects the system.

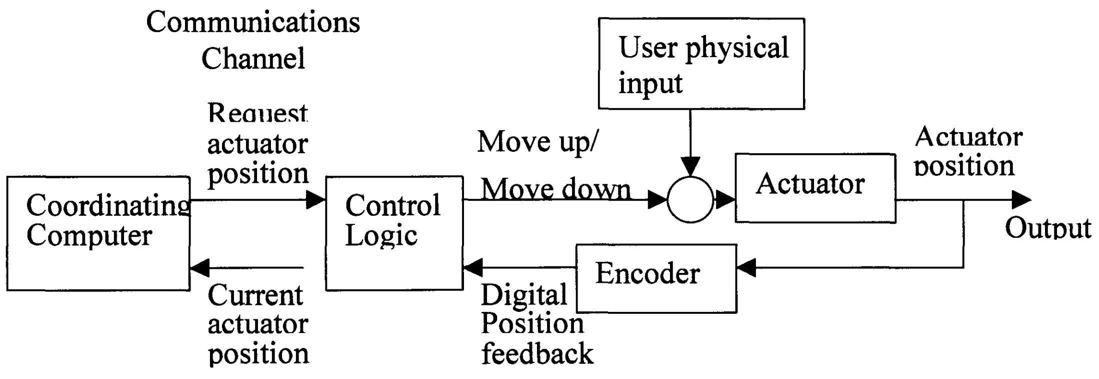


Figure 3.1: Functional diagram of final screw actuator PINS system

With the screw actuator system the user cannot physically move the actuator, but has to activate a sensor that detects the users touch. This means the user is affecting the move signals that travel to the actuator causing the actuator to change position, but not actually pushing the actuator themselves.

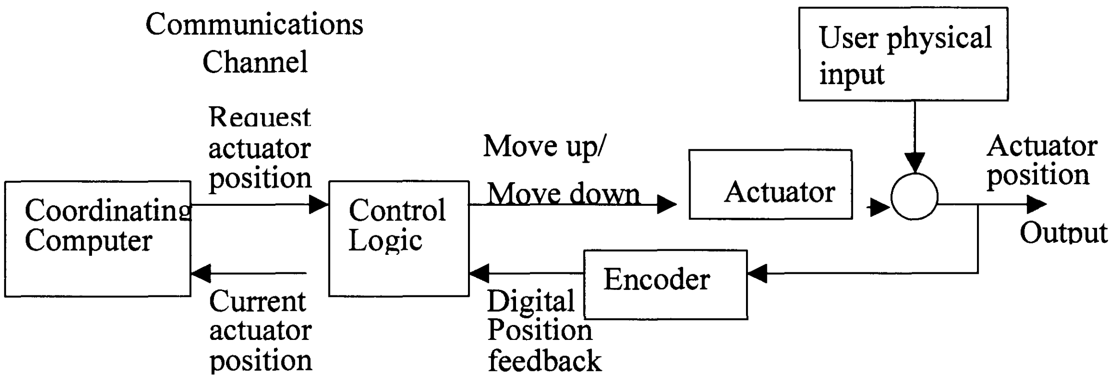


Figure 3.2: Functional diagram of final pinion gear PINS system

With the pinion gear system the user can back-drive the actuator allowing them to actually move the pins. The system then detects that the actuator is moving and knows it is the user's input.

The other blocks are the same for both systems and will be discussed in the following sections. These blocks are the control logic that translates between the actuators and the computer, telling the actuators what the computers requirements are. There is also the encoder, which detects movement of the actuators so the system status can be kept up to date. The Communication channel between the control logic and the computer, and of course the computer that runs the software PINS interfaces with, and coordinates the commands sent to all of the actuator. And as already discussed there is the user physical input, which represents the users' manipulation of the mechanical system.

3.2 Control Logic

The control logic in the final system was black boxed into units named AVR X 16. This functional block contains everything required to plug together any number of AVR X 16's in series and plug both ends of this string into the coordinating computer.

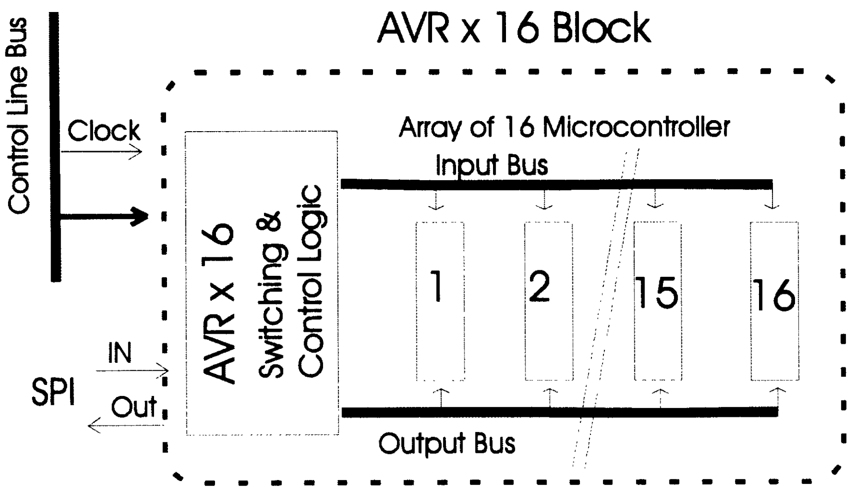


Figure 3.3: Control logic implemented in AVR X 16 unit

Each AVR block contains 16 microcontrollers. The X 16 allows for this microcontrollers to be run in normal operation or to be selected out of the string and run or programmed individually.

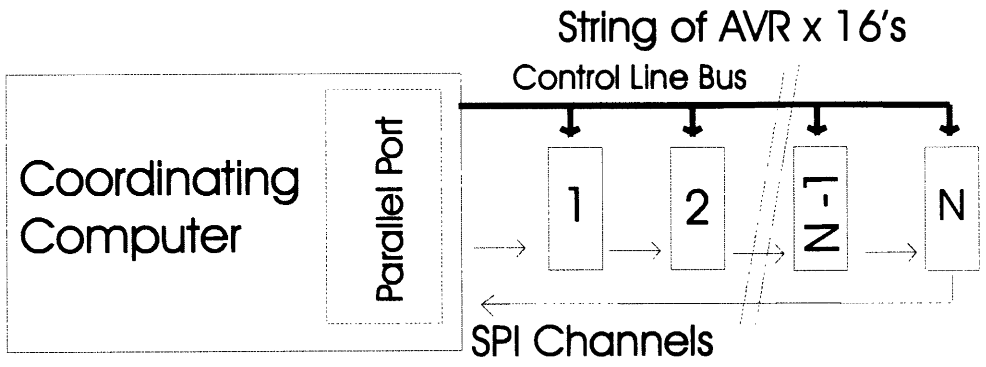


Figure 3.4: Complete system showing N AVR X 16 units in series

Using the AVR X 16 as discrete design blocks, which can be created on a single PCB board, makes construction and expansion of the units extremely easy. The boards simply need to be plugged together. Also if something goes seriously wrong on any one of the X 16 boards, it can simply be replaced.

3.2.1 Microcontroller Selection

Several microprocessors were used during the development of the PINS system. However, the Atmel Attiny26 microcontroller was selected as the distributed microprocessor element for the most advanced version of PINS. There were several reasons for this selection. This microcontroller can operate at fairly high speed (16

MHz), and can thus provide ample processing power to implement the PINS control algorithm.

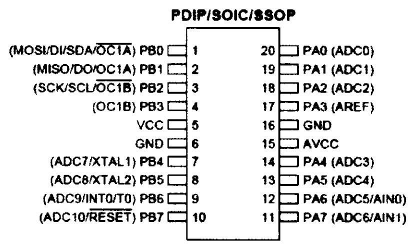


Figure 3.5: Pin out of Atmel ATTINY 26

Moreover, the chip is very inexpensive (about $1.50 when bought in bulk). It could also be programmed with true C code which could be complied using a free GCC compiler. PIC microprocessors use something called PIC C, which does not all ways behaved as expected. The ATTINY 26 microcontroller also offers several attractive features such as built in hardware shift registers, and a built in RC oscillator whith frees up two pins as I/O ports and pin change interrupts. With an external oscillator two I/O pins have to be dedicated to listening to its clock signal. All around the ATTINY 26 is an amazingly useful and powerful IC component.

The built in oscillator has internal fuse bits that can be programmed to set the microprocessor system clock to 1, 2, 4, or 8 MHz. At these frequencies the chip can operate with a supply voltage between 3 and 5 volts.

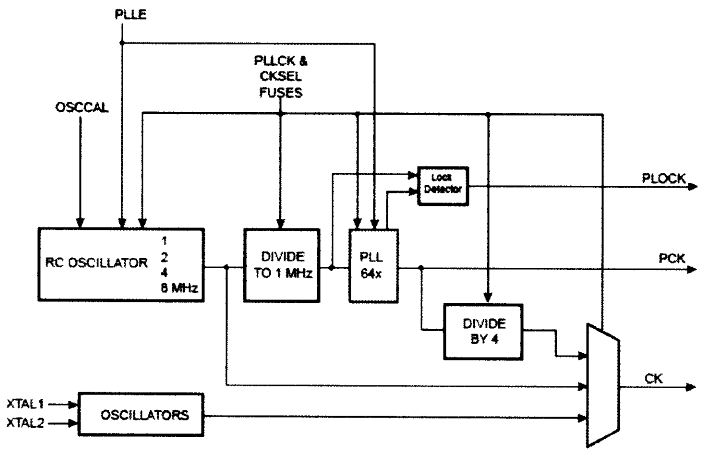


Figure 3.6: Fuse bits can be set so that and Phase Lock Loop is used in combination with a divide by 4 counter to generate a 16 MHz clock signal with no external components[3]

[3] Diagram from ATMEL ATTiny26 complete data sheet

The chips maximum frequency is however 16 MHz. To achieve a 16 MHz frequency from the internal clock the system fuse bits can be set to route a 1 MHz oscillation from the RC oscillator to the built in Phase Lock Loop (PLL). The PLL effectively multiplies the 1 MHz signal to 64 MHz, these is intended for use with locking a signal or to clock fast external devices. However, this 64 MHz signal can also be shunted into a divide by four register and then used as the system clock. This effectively creates an internal 16 MHz clock signal. The only disadvantage is that the supply voltage for the system has to be maintained between 4.5 and 5 volts to operate in this mode. This power supply requirement can be easily met by off the shelf switching power supplies and the use of bypass capacitors distributed through out the system.

The chip's built in SPI port was also a critical feature for its final selection. This port can operate asynchronously to the internal clock of the microcontroller. The SPI port can also be clocked at speeds far exceeding 40 MHz when data is just being passed though and at 12 MHz if every bit is being read out of the SPI registers [ATMEL Attiny 2]. We did not have the computer to send data at anything approaching these speeds.

Another convent features of the ATtiny 26 is its 16 I/O ports. This allows for easy interface with the encoders, motor drivers, and other peripherals of on the actuators.

3.2.2 Function of AVR X 16's

The AVR X 16, which was the idea of Jason Alonso, was critical in making the project possible. The flexibility of the AVR X 16 is due to its ability to dynamically reconfigure the flow of the ones and zeros from the computer from passing through all the chips to flow through just one microcontroller, addressing it directly. Or the system can be set up to selectively pass the data to any number of microcontrollers that are linked in series.

3.2.3 Components of the AVR X 16

The parts selected for the AVR X 16 were chosen based on availability, price and package. A lot of work was put into designing the system is carefully as possible to have the minimum part count required for functionality. Almost all the parts selected were available in surface mount and DIP packages allowing them to be tested in proto-boards and then built on PCBs without having to worry about part substitution. The following parts were selected and their uses in the system are explained below.

3.2.3.1 Three-State 8-Input Multiplexer -- 74HC251

This component selects one of eight binary inputs to pass to the output. The three state output (high, low and high impedance) capability is essential because the chips output is connected to a common bus line. Feeding a low logic level to the OE pin activates the high impedance output letting the line it is connected to float. Three data select lines select the input channel that is fed to the output.

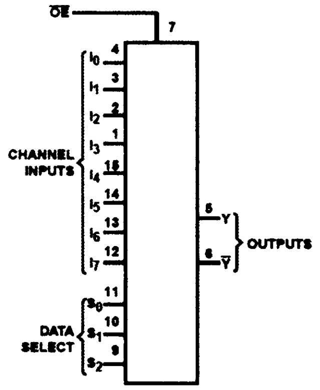


Figure 3.7: 74HC251 functional diagram of eight input multiplexer [4]

Two of these chips are used in parallel to select the output line from one of the 16 microcontrollers SPI ports. This data is then piped directly back to the controlling computer. This is necessary for programming of the microcontroller, testing and debugging of individual parts of the system.

3.2.3.2 4-to-16 Line Decoder with Latch - MM74HC4514

This chip can be used to route data from one input pin to any of 16 outputs. There are four data lines that select the output pin that is active. After the output pin is selected the *inhibit* input can be changed causing an inverted but corresponding change in the selected output.

[4] Diagram from Texas instruments 74HC251 data sheet

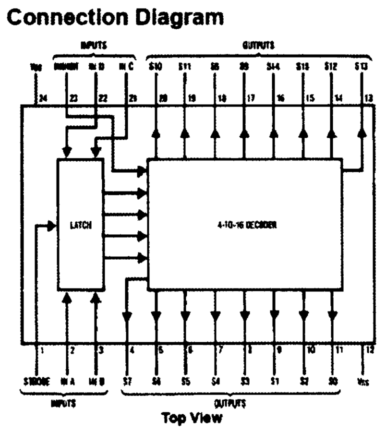


Figure 3.8: MM74HC4514 connection diagram of 1-16 data router [5]

This allows the control computer to select one of the 16 microcontrollers to send data directly to with out passing it through other components. This router is used on both the SPI data input and latch line inputs for the microcontrollers when they are being addressed directly.

3.2.3.3 32-Bit Two Port Bus Switch - PI5C34X245

The Pericom PI5C34X245 is designed as a bus switch. When activated it simply connects the wire at the input to the wire on the output. When active there is only a 5-ohm impedance between these ports and no appreciable propagation delay. This chip has 4 enable pins that allow you to control 8 connections at a time effectively shorting the input to the output, connecting or disconnecting a byte data line to a bus with one control bit.

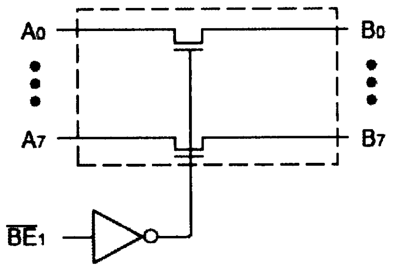


Figure 3.9: Logic diagram of one of the 8 bit blocks that connect the A pins to the B pin with only a 5 ohm impedance [6]

[5] Diagram from Pericom PI5C34X245data sheet

Two of these chips would be used in a complimentary fashion, connecting and disconnecting bus lines to prevent bus contention. Bus contention is where two or more sources try to drive the same logic line. This chip would control if the microcontrollers are in direct selection mode or in serial communication mode.

3.2.3.4 2 to 1 Bus Data Selector - SN74HC157

This 4 bit data MUX switches between 1A-4A and 1B-4B depending on that *not A B* input. It also has an inhibit pin that makes all the outputs logic zeros.

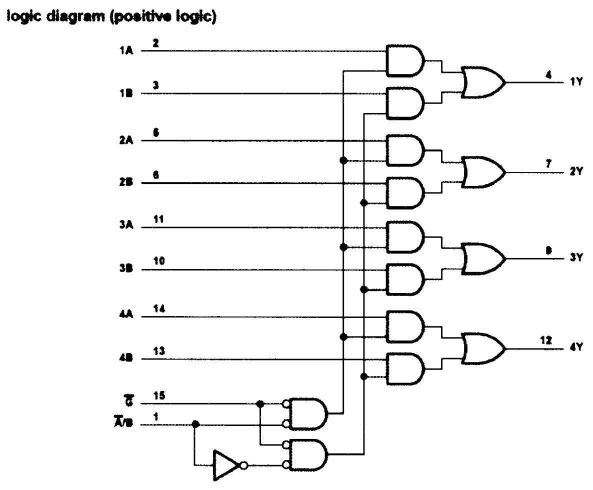


Figure 3.10: SN74HC157 Quadruple 2-1 Data Selector [7]

This chip was used in the prototype AVR Board to switch between the having the board in direct chip selection mode, where the computer select and talk to one microcontroller, and serial communication mode, where the chips are hooked in series. This configuration change is cased but selecting one of two 16-bit buses. This is done simply by toggling the *not A B* pin. Eight of these MUX chips are required for this operation. This shows why 2 of the very small 32 bit bus connects is a preferable solution.

[6] Diagram from Texas Instruments SN74HC157 data sheet
[7] Diagram from Texas Instruments SN74HC157 data sheet

3.2.3.5 8-Bit Shift Register

The 74HC595 is a very venerable, and powerful logic chip. It allows for a stream of input bits to be clocked in and the received bits values to be applied on 8 separate output lines. This is an invaluable tool in serial to parallel data conversion.

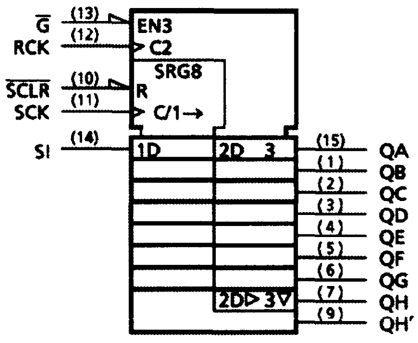


Figure 3.11: Toshiba TC74HC595 diagram of eight bit shift register[8]

In the AVR X 16 this shift register can be hooked up to the serial output of the computer and is used to set the address bits, and control lines for all the other logic devices on the board.

3.2.4 Control Signals

There are two levels of control signals in the PINS project. There are the global system level control signals set by the computer and there are the X 16 broad level signals that are local to each AVR X 16 components. Through out this paper a / will denoted inverted logic.

3.2.4.1 System Wide Control Signals

The Computer parallel port sets the logic signals that control the overall system configuration. These signals change wither the unit is in set-up or running mode and several other key operations. The signal names and their functions follow.

[8] Diagram from Toshiba TC74HC595 data sheet

- **Latch**: Is the global signal to tell all the microcontrollers to grab the data in there shift registers and replace it with the byte they need to send back to the computer.

- **/Reset**: This command shuts off all the microcontrollers. When Reset is unasserted the microcontrollers startup again in their default state.

- **SCK**: This is the Clock signal generated by the computer. It is used by the shift registers in the system to make them run synchronously.

- **ADDR_SEL**: The ADDR_SEL (Addressing Select) line controls wither the serial line data flows to the control line shift registers built into the AVR boards or directly to the microcontrollers. If ADDR_SEL is high any change on the Serial In line carries to the Control Serial Line. If ADDR_SEL is low changes on Serial In show up on the Serial Comm line while the other line stays low.

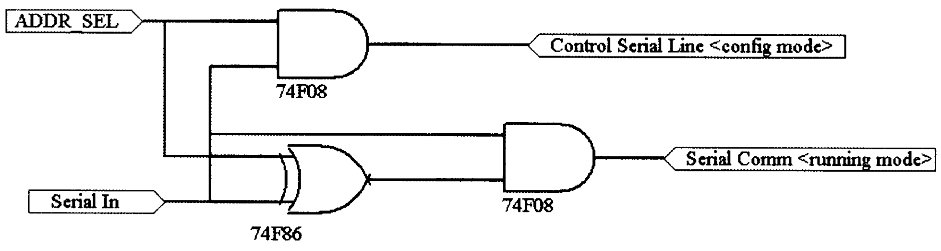


Figure 3.12: Xor and And gates that control the flow of serial date to the control setup line and running mode serial line.

- **SI**: This stands for Serial Input and is the Serial data sent to the X 16 boards. The path this data follows depends on the ADDR_SEL line, and how the data is processed in the unit depends on the current configuration of the X 16 boards, which is controlled in large part by the DIRECT_SELECT line.

- **/RUN**: RUN controls whither the microcontrollers are hooked in series or if the computer is talking to one directly. It is name because when it is low the system is in the normal running mode. It does this buy toggling the bus buffers that connect the microcontrollers to themselves in series or directly to the 16 output multiplexers.

- **GND**: This can been attached to pins 18-25 on the parallel port all of which are ground. They provide a shared ground so that the logic signals from the computer do not float in comparison to the logic on the AVR Boards. Not providing a common ground reference could result in invalid data signals.

- **SO**: This is the only input to the computer. Serial Out is the return path of the SIP data loop after it has passed through all the microcontroller or 595 shift registers, depending on the current state of ADDR_SEL. With the 595's this return path serves to check the continuity of the data path. When the data stream is flowing trough the microcontrollers it allows the chips to return their state data to the computer.

3.2.4.2 Common Bus and AVR Board Input and Output Signals

The common bus for the AVR Boards is 8 bits wide. Several of the signals on this bus are identical to the outputs from the computer. These signals are RESET and DIRECT_SELECT. However, several of these signals are created from a combination of the computer outputs, depending on the desired operations.

- **/RESET**: As mentioned exactly the same as computer output.

- **/RUN**: As mentioned exactly the same as computer output.

- **ASCK & SCK**: These signals are created from SCK and DIRECT_SELECT using a 1 to 2 multiplexer. ASCK is the addressing clock and is the clock signal for the 595 shift registers. When DIRECT_SELECT is set to 1 ASCK receives the computer clock signal and SCK does not change. When DIRECT_SELECT is 0 SCK on the common bus which is the clock input for the microcontroller's shift registers receives the clock signal and ASCK does not change.

- **LATCH & ALATCH**: These signals are created by the ADDR_SEL and LATCH lines. When ADDR_SEL is 1 the ALATCH line follows the computer output LATCH signal. This causes the discrete 595 shift registers to take the values they are holding and assert them on their output lines. When ADDR_SEL is zero the common bus LATCH line follows the computer LATCH line. This signal goes to the microcontrollers.

- **SI & ASI**: The serial input (SI) and the addressing serial input (ASI) are created using a 1 input to 2 output multiplexers to the first X 16 board in the series. For the following boards in the chain, these signals come from the SO and ASO outputs of the previous board. These are the serial data inputs for the boards discrete shift registers the communication path for the microcontrollers.

- **SO & ASO**: These signals are also different between each control board so they are not on the common bus. These are the serial outputs from an X 16 board and feed into the SI and ASI inputs of the next board. Or in the case of the last board in the chain the data is passed to the computers return data path, the SO line.

3.2.4.3 AVR Board Control and Data Lines

The control lines on the individual AVR broads are all set by the 595 shift register. The control computer could send a different configuration to each AVR Board but this ability is seldom used. The control lines used on the AVR Boards are:

- **addr [A0..A3]**: These 4 local address lines control which chip is being directly selected, normally used for programming purposes. These data lines are created by the 595 shift register from serial input data that has been latched to the outputs by Alatch.

- **Microcontroller Slave Input Bus [DI0..DI15]**: Each microcontroller is connected to one line of a 16 bit slave input bus. For example, microcontroller 5 's shift register is connected to DI5. Slave simply means the device is following a clock signal from another source.

- **Microcontroller Slave Output Bus [DO1..DO16]**: Each output of the built in microcontroller shift register is connected to one of the lines on a 16 bit output bus. These bus lines are driven by and a 16 bit output bus. Again, by hooking the microcontrollers to a data bus it is possible easily routed and change the connections.

- **/haen**: Is the High byte Addressed output Enable. It selects the output to the computer as microcontroller output lines D0 8 through DO 16. One of these outputs is then selected by address lines and the signal is passed to the to the global DSO line.

- **/laen**: Is the Low byte Addressed output Enable. It selects the output to the computer as one of microcontroller output lines D0 1 though DO 8. One of these outputs is then selected by the address lines to pass the signal to the global DSO line.

- **/smen**: This stands or Serial Mode Enable. When this inverted logic line is asserted to microcontrollers are hooked up in serial data mode. So the serial output of one chip is routed to the serial input of the next chip.

- **/dmen**: This stands for Direct Mode Enable. When this inverted logic line is asserted the microcontrollers are hooked up in direct mode. This means that the address line value sets which chip on the AVR Board is directly talking to the computer.

3.2.5 How the Components Fit Together

There are two different states that the PINS system can be in. One is setup mode; the second is running mode. Setup mode is when the computer defines the way the PINS system will operate during running mode. There are also two main ways the PINS system can configured during setup mode. These are in Serial operation and in Direct operation. There are also wide ranges of other running operational modes that the system can be configured to but these are not generally used.

3.2.5.1 Setup Mode

To go into Setup Mode the computer puts the /RUN line high. This disables normal running operations by disabling the outputs of the 595 shift register, allowing the logic lines it drives to be controlled by resistive pull ups. Also, ADDR_SEL is set high routing the computer control lines through the multiplexers crating the Alatch, ASCK and ASI signals.

At this point the computer starts pumping bits into the discrete shift registers by pulsing the clock line. When the computer thinks all the registers have the correct bits it pulses Alatch. To start using the newly setup configuration, /RUN is put low asserting the newly latched data to the board controlling outputs. Then ADDR_SEL is set to low routing the data back to the microcontrollers and restoring the SCK, LATCH and SI lines as the active signals on the X 16 boards.

3.2.5.2 Serial Running Mode

For normal running operation mode, the local control lines are set during setup with /smen low. Both /haem and /laem are high to avoid bus contention, and the address does not matter. The flow of the data is as follows. The control computer generates a clock signal which is passed to SCK on the common bus which the microcontroller shift registers are monitoring. SI from the computer is being passed to the SI input on the first X 16 board.

On the first X 16 board the microcontrollers are configured so that the DI0 input is coming from the computer. All the boards have the DO1 output line is shorted with the DI1 input line. This pattern is repeated through all 16 chips. The output of this last AVR is passed to the next X 16 boards SI input. Only on the last X 16 board in the chain /haem is set low and the address has to be set to 16 so that the last chip on the last board is connected to the computer's return SO path.

When the computer has loaded the bits it wanted into the shift registers the latch line is pulsed and the microcontrollers code is interrupted to swap bytes in the register. The computer then collects and processes the data sent from the microcontrollers. This process is run continually creating closed loop communication between the actuators and the computer.

3.2.5.3 Direct Running Mode

In direct running mode the setup process is the same except we only want to select one chip on one board, for example that's select chip five on a given board. To do this that /dmen is set low on the desired board and the address is set to five (0101), selecting the requested chip. On all the other board /haem and /laem are set high disabling the outputs of the 8-Input Multiplexers, again this is to avoid bus contention. On the selected board /laem is set low because 5 is in the low byte of the microcontroller bus. Now chip five's shift register input and output are hooked directly to the control computer's input and output. In this state the chip's flash memory can be programmed and it configuration bits set.

3.2.6 Propagation Delays

The PINS system was design to insure that propagation delay would not be a problem. In the running mode we have to make sure that valid data is returned to the computer before the next clock cycle. The longest propagation path of the logic signals is for the clock signal to reach the last microcontroller in the string and that microcontroller to send it bit back to the computer. This is the beauty of a system based on serial data, the system can be run at high speeds because all the logic is isolated in latched stages. The path for the worst case is as follows:

Line Buffer -> 2 to 1 multiplexer -> Bus Driver Buffer ->microcontroller -> 8-Input Multiplexers -> Line Buffer -> Computer to be sampled on next clock cycle

This is a total delay of 120 ns for the 3 buffers, 12 ns for the 2 to 1 multiplexer, 50 ns for the 8-Input Multiplexers and a max of 62.5 ns at the microcontroller. This is a total of 0.485 us. Since the clock cycle is 270KHz, which has a period of 3.7 us, the there is no problem with propagation delay. All other modes have a similar delay varying by just a few nanoseconds.

3.3 Encoder / Decoder

The digital encoder is critical for creating the position feedback signal required to make a closed loop allowing the movement of the actuator rods to be controlled.

3.3.1 Encoding Scheme

The output of the encoder is a quadrature logic signal as shown in the figure below. Note that the Phase A and Phase B waveforms differ in phase by 90 degrees.

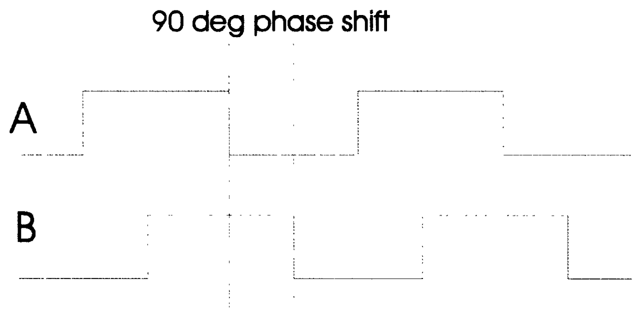


Figure 3.13: Quadrature waveform of encoder

When the outputs of encoder are used as digital inputs, a sequence {00, 01, 11, 10, 00...} is created. This sequence is known as a gray code. The important feature of a gray code is that only one bit changes between each of the code words no matter how big the code word is. In this case the words are two bits long. This prevents errors from occurring from what is call logic race conditions during the transition between code words. The two sequences shown below illustrate the output from the encoder lines in each direction of rotation or movement. The sequences should be read from top to bottom.

CW Rotation / Up			CCW Rotation/ Down	
Bit A	**Bit B**		**Bit A**	**Bit B**
0	0		0	0
0	1		1	0
1	1		1	1
1	0		0	1
0	0		0	0

Figure 3.14: Code sequences of quadrature encoding for each direction of rotation

By tracking the changes of the code words, a record can be made of wither there is up or down movement of the system

The direction sensing may be implemented in software as firmware on the microcontroller or as hardware with the connection of two flip-flops. Both methods provide the ability to remember previous states of the system. In the interest of reducing overall parts count, and system cost, the firmware was the final choice for the pinion motors; however, both systems were built and tested.

3.3.2 Encoder Hardware

The encoders used were built out of infrared photointerrupters. These sensors consist of an infrared light emitting diode (LED) and a photosensitive transistor. The LED is mounted in the package so that its light shines across a gap to activate the phototransistor. If, however, the gap is blocked the phototransistor would stop conducting current and the output would rise. When the IR light excites the transistor it pulls the output voltage towards zero.

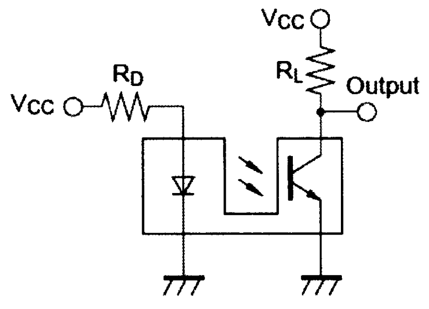


Figure 3.15: Diagram of photointerrupter and external pull up resistor. Rd limits current though the IR LED and Rl determines how quickly the output responds to the photo transistor changing states9

The problem with the encoder having open-collector outputs is that it requires pull-up resistors to produce valid logic levels. Also open-collector interfaces do not work well over long wire connections that are subject to environmental noise and signal reflections. The solution was to give the wiring for these logic signals priority on the PCB board layout. The pull-up resistor on the collector was sized as small as possible to give the maximum amount of noise immunity to the line. The Rl resistor could not be two small though because the on resistance of the phototransistor can vary wildly and could be as high as 500 Ω. This would crate a voltage divider preventing the voltage from ever dropping into the logical zero data range (this was discovered the hard way with the system not working and had to be debugged). So Rl was set at 5 KΩ allowing it to always fall into the logic zero range. Reflections were not an issue because distances traveled by the signal were to short when the frequencies involved where taken into account. At higher data rates reflections could have been a more serious problem. These considerations minimized the exposure the data lines would have to the extremely noisy environment created on the board.

[9] Diagram from SHARP GP1S093HCZ data sheet

The photointerrupter packages used were the SHARP GP1S093HCZ and duel photo interrupter package OMRON EE-SX1131. Both of these are miniature packages, but the OMRON, because it is a duel package only requires one component to make the gray code signal, while the sharp component requires two photointerrupters set up side by side to generate the required signals.

3.3.2.1 Screw-Actuator Mechanical Mechanism

The SHARP package is used with the screw actuator version of PINS. To create the gray code two SHARP photo interrupters are placed on the radius of a circle. A semi-circle was attached to the threaded rod of the actuator so that as the rod spins the semi-circle interrupts the IR light of one of the SHARP packages and then the other. As the disk continues to spin through the photointerrupters it creates the second half of the code. Fifty-six revolutions in one direction relates to one inch of movement.

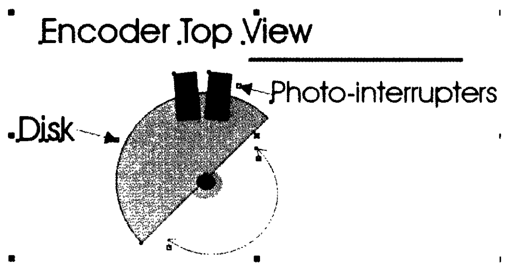


Figure 3.16: Close up of actual encoder on motor on left. Diagram of the configuration of disk encoder on right.

Changing the shape of the interrupter disk so the photointerrupters were blocked and uncovered two or three times per revolution could easily have increased the resolution of the encoder. However, this increased resolution was not necessary for the system to work, and would have required more processing time for the microprocessor.

3.3.2.2 Pinion Drive Mechanical Mechanism

The encoder for the pinion actuator system was designed around the convenient OMRON duel encoder package. For the design used with the pinion drive system the use of two SHARP single photointerrupters would not have created a high enough resolution. The OMRON package shares a collector for the two built in phototransistors so emitter pull-downs have to be used instead of the collector pull-up used with the

Sharp package, but this only inverts the logic and does not matter for the encoding scheme used, because it is a symmetric code.

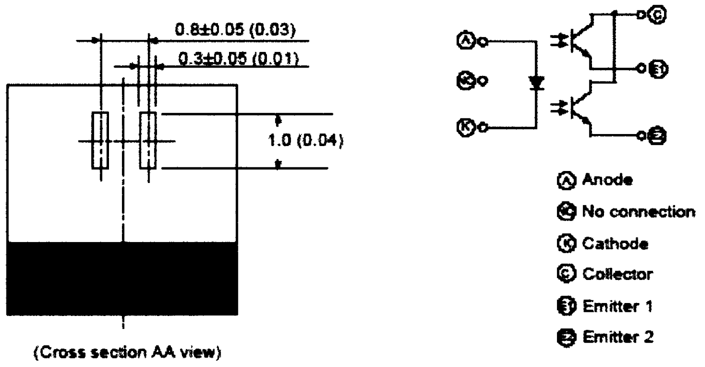


Figure 3.17: The figure on the left shows the spacing of the slits that house the phototransistors in the Omron duel photointerrupter package. The Right figure shows the circuit diagram of the duel package with the single IR diode, and shared collector. [10]

The space of the holes in the encoder strip a set up so that both the slits for the photo interrupters can be covered and then as the strip moves they will both be exposed creating the gray code with the up/down motion of the actuator. The slots in the encoder strip are 1/16th of an inch wide, so a complete cycle of the code is created every 1/8th of and inch.

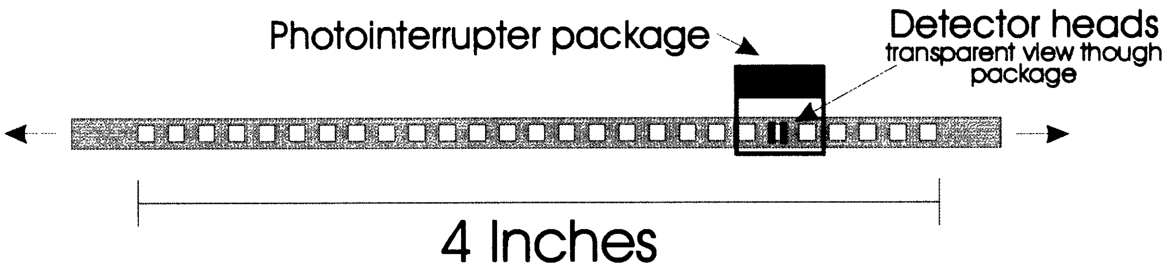


Figure 3.18: Picture of encoding strip. As the strip moves though the photointerrupter package the full quadrature gray code is produce.

These encoding strips were attached to the bottom of the actuators creating the digital position feedback required to make the PINS project work.

3.3.3 Software Decoding

[10] Diagrams from OMRON EE-SX1131 Data Sheet

The firmware to decode the quadrature encoding can be implemented as a finite state machine.

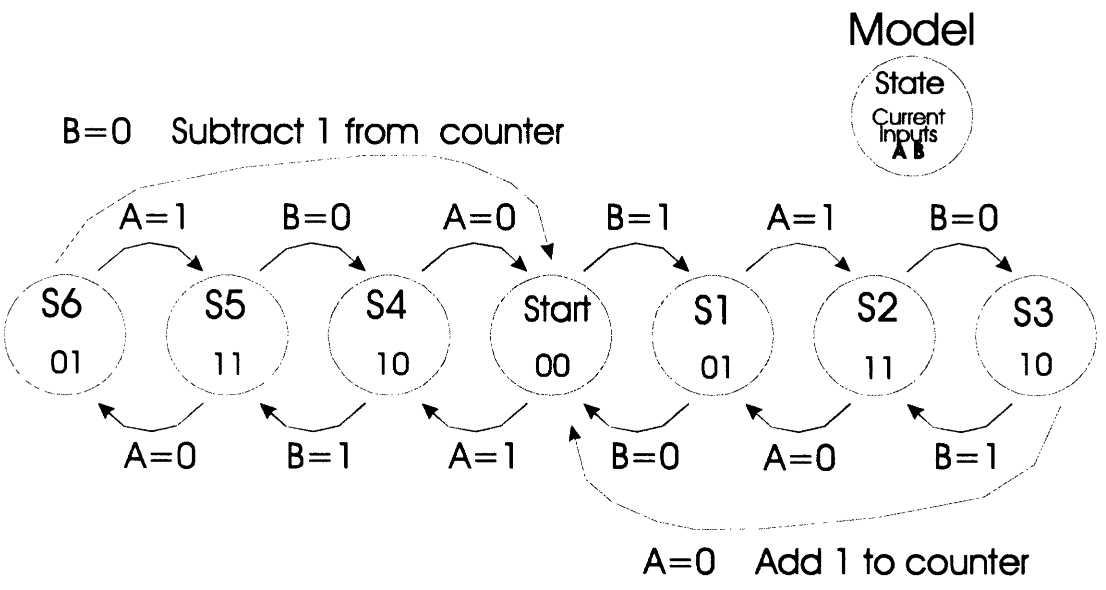


Figure 3.19: Diagram of FSM implemented in firmware to track movement of the actuator.

This 7 state FSM tracks every step of the 2 bit gray code, and adjusts the value of a software counter when the required conditions have been met. The use of this method requires constant polling of the input values on two pins of the microcontroller. If the values are the same as last time they were checked, the state of the FSM does not change. If either the A or B input changes then the state is adjusted accordingly. To make this system work you have to make sure that the outputs from the encoder to the microcontroller do not change faster then they are polled.

With the pinion actuator system it was found the fastest the physical mechanism could be moved was by a user pressing it down sharply. In this case it took 0.5 seconds to move the actuator its 4 inches of travel. This system had 32 points of resolution along its complete length. Each point of resolution requires 4 gray code values. This means there are 32 x 4 point the actuator needed to be polled while it moved to catch all the required states. This is a total of 128 states a half second that have to be captured to make sure there are no errors in the position tracking. So, if the input signal is perfectly synchronized with the polling of the microcontroller inputs then pins have to be tested a minimum of 256 times a second. However, there is no guarantee that there is synchronization between the input signal and the measurement timing. So, it is possible that aliasing could occur causing errors in the height data and failure of the system. To avoid this problem we must observe the Nyquest sampling criterion, this require sampling at twice the frequency of the fastest input signal. (Oppenheim, 1997) The fastest input signal was determined to have up to 256 values a second. This means that their needs to be 512 measurements per second to guarantee valid measured data with no aliasing.

By measuring an output pin set to go high and then low every other cycle of the microcontrollers final main code loop required for complete operation, it was found that that the input pins were sampled about 50,000 times per second. This is almost 100 times faster then the minimum required working condition. The software FSM implementation for direction sensing and tracking works very reliably and requires only the sensor and the microcontroller, keeping total part count and total cost low.

3.3.4 Hardware Decoding

With the screw actuator system, which was built, a hardware decoder was required helping in the decoding of the signal from the sensors. This was because the physical set up of the sensor made the change between the states 00 to 11 and vice versa very quickly.

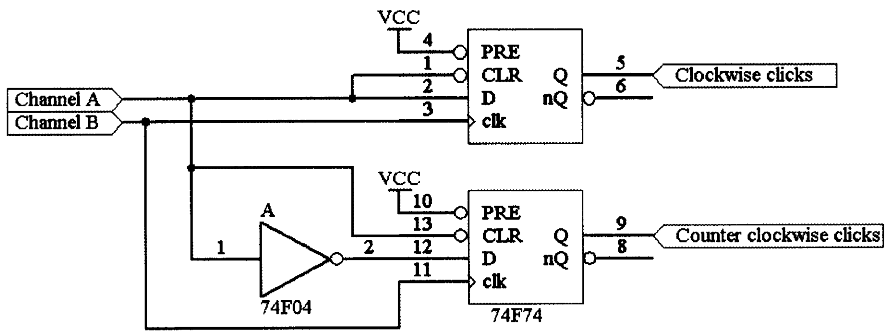


Figure 3.20: Schematic of decoder and direction sensing circuit

The direction sensing circuit output is a square wave signal with one rising edge for every completion of the gray code cycle on the channel A and B inputs. This circuit was originally designed to be used with a MAXON motor that had a built in encoder. The attached encoder made this motor very easy to use for early testing of the screw actuator design, even though the motor was ways to big and far to expensive. This MAXON encoder's outputs go through 2000 complete cycles of a 2 bit gray code per revolution. There was no way a microcontroller could poll its inputs quickly enough to accurately follow the rotational position of the motor. This decoder circuit was hooked up to a PIC microcontroller, which had two built in rising edge triggered counters. These counters tracked every click from the decoder circuit. When the software needed to know the current position of the motor, the counter would update a 32-bit motor position variable. The clockwise counter added to the value and the counter-clockwise counter subtracted from the value.

Luckily the pins system did not require the level of accuracy the MAXON motor system was capable of. This system cost about 500 dollars per motor. The screw actuator system with its hand built encoders was only capable of 1 gray code cycle per revolution. Even this low level of resolution creates changes in the logic levels faster then the microcontroller can pole it, so the hardware decoder continued to be useful with several of the screw actuated versions of PINS.

3.3.5 Cost of implementation

The encoders used in this system were designed to be cheaply and easily constructed. It is possible to buy encoders with much higher tolerances and better resolution, but this would cost tens or hundreds of dollars each blowing the budget. Both the encoders for the screw actuator system and the pinion system cost less that a dollar each, making them effective, reliable and economical.

3.4 User Physical Input

3.4.1 Capacitive Sensor

In the screw actuator multiple pin mock up it was not possible to back drive the system. However, it was necessary that there be some way to effect change in the system. Simple buttons were tested but it was difficult to get the mechanics of these buttons to work reliably. A capacitive sensor was approached as very reliable, very cheap solution.

It is very reliable because there are no moving parts. Also there is no one contact point, in this implementation. The top of the actuator is covered with a piece of copper, and touching the copper pad anywhere is detected.

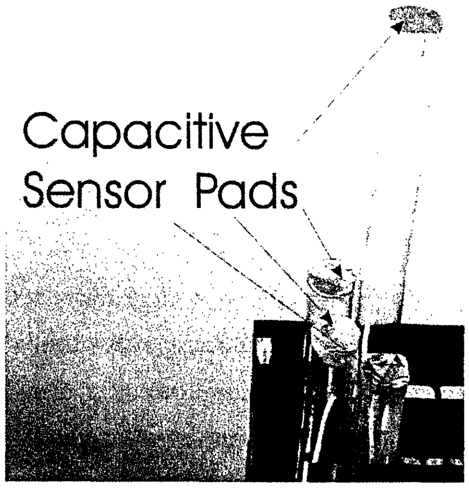


Figure 3.21: Copper pads topped all the actuator pins as part of the capacitive sensors

The sensors are very cheap because they require only the copper pad and a resistor and a free pin on I/O pin on a microcontroller. The cost of these parts is almost nothing since we already have the microcontroller. The I/O pin is used to reset and test the voltage on the copper pad, and the microprocessor, and using very few instructions, handles timing.

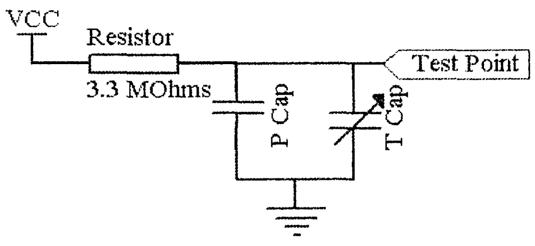


Figure 3.22: Circuit model of capsitive sensor. The Test Point is connected to the microcontroller, which test the voltage on the capacitors and then reset the capacitor voltage to zero. When the circuit is being touched the touch capacitor (T Cap) dominates the system. If the circuit is not being touched the T cap goes to zero the parasitic capacitor (P Cap) dominates the system.

The principle behind the sensor is that the human hand acts like a charged plate. When you put your hand close to or touch the copper pad you are increasing the capacitance of the system, thus increasing its RC charge time.

$$C = \varepsilon \frac{A}{d}$$

Where C is the capacitance, and ε is the electrical permittivity of the material between the two plates of the capacitor, in this case air and skin. A is the area of the two charged surfaces and d is the distance between the surfaces. As the users hand, which is one of the plates, is far away, the capacitance is almost zero. As the users hand moves closer the capacitance increases dramatically. Without a hand touching the system only the parasitic capacitance of the system effects the charge time. The voltage of the capacitor with respect to time is controlled by.

$$v = VCC(1 - e^{\frac{-t}{\tau}})$$

VCC is the supply voltage of the system. Small v is the time varying voltage. Small t is the time and τ is the time constant, which is defined as resistance times capacitance, RC. Therefore the bigger the RC time constant the slower the voltage at the test point will rise after being reset. Below are measurements from tests of the voltage on the PINS systems capacitive sensors.

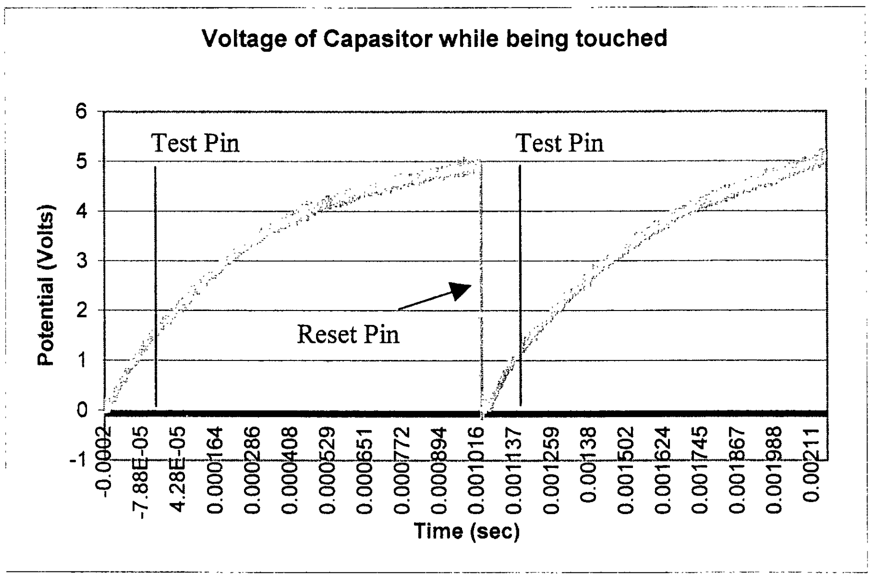


Figure 3.23: When the voltage is tested it is below 2 volts which is the threshold for detecting a "true" logical 1 value[11]

[11] Data measurements collected using Tektronics digital oscilloscope with a person touching capacitive sensor of multiple pin screw actuator system

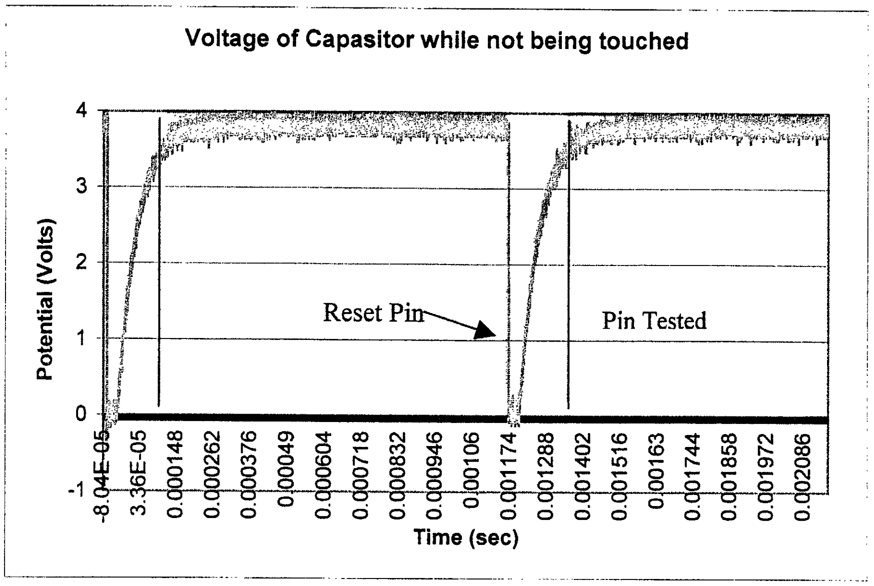


Figure 3.24: When the voltage is tested it is well above 2 volt which is the threshold for detecting a "true" value[12]

The capacitive sensor works very reliably when a person is touching it, and can inform the controlling computer exactly which actuators are being touched, even if the user is not pressing the actuator down. However, it does require a lot of processor time to continually monitor the sensor pad. Also, this system has the disadvantage of not being able to tell when a non-conducting material was present on the actuators, such as a plastic model.

3.4.1.1 Capacitive Sensor Mechanical End Stop Detector

The capsitive sensor is also used to detect when the actuator rod has been pushed all the way down to the bottom of its range. This is done by making a contact that connects the sensor with a ground line when it reaches the bottom of its physical movement range. This contact shorts the capacitive sensor pad to ground. In this case the RC charge time is infinite. This is detected by testing the pad just before the reset. If the pad voltage is still zero just before reset, several things happen. First, the motor is disabled from moving down any further. This keeps the mechanical system from hurting itself. Next the height counter is reset to zero, just incase any of the encoder

[12] Data measurements collected using Tektronics digital oscilloscope without a person touching capacitive sensor of multiple pin screw actuator system

height clicks have been missed. The condition of the pad being grounded overrides any other user input so the system does not get damaged.

3.4.2 Back Drive

Back Drive is simply where the user pressing on the system surface overpowers the actuator components. Ideally if the user is apposing the system it has a springy feel, giving tangible feed back to the user that the system is trying to do something else. This requires the power train linkage to have a very smooth mechanical mechanism.

3.4.2.1 Back Drive End Stop Detector

The back drive system also requires and end stop detector. As with the screw actuator system, it is necessary the system does not try to drive itself passed its physical end. If for some reason the counter is inaccurate, when it hit the bottom stop it is reset to zero. During startup the actuators are told to move down to hit the bottom stops. This is necessary because at startup the height counter is initialized to zero. So when the system is starting all the actuators are driven down to make sure their position sensors are calibrated correctly.

There are several options, for detecting when the system has reached its physical end. One option is to use a button or switch. This, however, requires very precisely placing the button and mounting it so it is not bent out of the way after it is hit a few times (this happened with early tests). The other option with the pinion gear system is to use another photointerrupter. When the encoder strip moves all the way down the end of the strip activates the photointerrupter. This requires no physical contact, making it very reliable.

3.5 Coordinating Computer

The coordinating computer is responsible for insuring that valid signals are sent to the microcontrollers and the X 16 boards. The computer configures the boards and then sends instructions and receives feedback from the actuators. The computer acts to make the individual actuators behave as a unified system.

The computer communicates using a parallel printer port. The outputs of the port are used as data and control lines.

It should be kept in perspective that the purpose of pins is as a device to aid human computer interactions. So, the main purpose is to have software running on the computer that utilizes the PINS interface, not to just have a computer that controls the actuators.

3.6 Communications

As discussed, two different standards were used to transmit bytes between the computer and the control logic. These two schemes were RS-232 and SPI. In general RS-232 is better/more reliable at transmitting data along longer cables then SPI. However, the longest cable used in this project was three feet. RS-232 is also, very well developed and it is easy to interface with the serial port of a computer because there are numerous libraries for it, while we ended up having to develop libraries for the SPI communications scheme used. However, most of the testing of microcontroller code, and the development of the firmware that runs PINS was done using RS-232 for communication with the computer making it worth of discussion. The two schemes work as follows.

3.6.1 RS-232 Data Communications Interface

The RS-232C standard specifies electrical parameters as well as the data format. The RS-232C standard uses what is known as non-return-to-zero (NRZ) bipolar voltage signaling. A logical HIGH is indicated by a voltage of –5 to –15 volts, while a logical LOW is indicated by a voltage of +5 to +15 volts. The driver must be able to drive loads of 3000 to 7000 ohms with a slew rate less than 30V/μs. It also must not burnout when short-circuited. The most common data format consists of a start bit, a logical HIGH to LOW transition, followed by 8 data bits, followed by a stop bit. The stop bit is created with a LOW to HIGH transition. This format is known as 8N1, which represents eight data bits, no parity, and one stop bit. The number of data bits can vary from 5 to 9 and more than one stop bit may be used. Another important parameter of RS-232 is the baud rate, which is the number of clock periods per second.

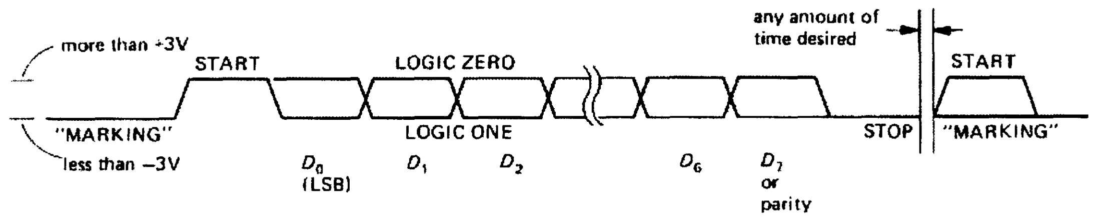


Figure 3.25: RS-232 serial byte waveform [13]

The receiver must drive a load of 3000 to 7000 ohms and convert an input of +3 to +15V to logical LOW and -3 to −15 volts to logical HIGH [Horowitz, 1990]. Early PINS systems uses a baud rate of 115200 and a 8N1 data format. This means transmitting a byte takes approximately 87μs.

RS-232 signals transmitted from a PC serial port is plus and minus 9 volts. These logic levels have to be converted to logic level the control logic understands. To remedy this problem, a MAX233 serial line driver was used. The MAX233 acts as a translator for the two different logic types. Compared to other serial line drivers, which require positive and negative voltage supplies greater then 9 volts, the MAX233 chip only requires one 0-5 volt supply. The MAX233 internally steps the DC voltage up and down, using an integrated charge pump, to make the positive and negative voltages required for serial communication (positive and negative 9 volts).

The data format is handled by the microcontroller in firmware or internal hardware. The PIC 16F876, originally used with PINS, contains a module known as a universal synchronous/asynchronous receiver transmitter (USART), which handles generating the baud clock, serializing data for transmission, and shifting incoming data into a receive data register. For microcontrollers that do not possess internal USART's, like the PIC16F628 and ATMEL ATTiny26, the RS-232 data format may be implemented in firmware. The generation of the baud clock and data registers in firmware requires a huge data overhead. This processing overhead of RS-232 on the AVR ATTiny26 along with the fact that it is difficult to communicate with more the one peripheral using this standard caused it to be abandoned.

3.6.2 SPI Data Communications

[13] Horowitz, 1990

SPI stands for Serial Peripheral Interface and is often used to describe a synchronous serial port. It operates on 5 volt TTL (Transistor to Transistor Logic) like most standard logic chips. Data is sent one bit at a time, with bits distinguished by a clock pulse from the computer. The system is like a buck brigade, with each clock tick shifting the buckets one bit further down the line.

Each microcontroller has a built in 8-bit shift register that requires no processor cycles to pass bits along the chain. The chips only have to process the bytes in there shift registers when the computer sends a "*Latch*" command. The *Latch* is a data line on the common bus and is connected to a pin on the microcontrollers that detects rising edges. When a rising voltage edge is detected it causes a software interrupt. This interrupt makes the microcontroller grab the byte in the shift register and swap it for a byte to send back to the computer.

Using the parallel port of the computer the clock line can be run a 270 KHz. This is a measured value from a standard PC parallel port. This .27 MHz is way below the maximum speed the shift registers can pass along bits. The addressing 8-bit shift registers can shuffle bits along at 55 MHz (HC595 data sheet). However, 270 KHz is plenty fast for any current or near future implementation of pins. At 270 KHz 100 pins can be updated at over 120 Hz, and 1000 pins at 12 Hz, which is also above the mechanical time constant. This data rate could be increased easily with improved hardware, but currently the existing computer parallel port has sufficient throughput.

For any long wires such as between the computer and the AVR X 16 control board digital buffers where place a both ends so that as long as there was not overly excessive noise the signals came out of the second buffer cleanly. To decrease what little delay is cause by the buffers inverting buffers were used at both the input and the output, cause no net change in signal polarity.

3.7 Actuator Motion and Control

3.7.1 Motor Driver

All the versions of the PINS project require bidirectional motor drive. To change the direction of a motor's movement the polarities at the motor ports have to be reversed. This can either be achieved by having two separate power supplies, one positive and one negative or by dynamically switching which leads are connected to the positive and negative terminals of the supply.

To keep costs and overall bulk down it was decided to go with one supply and find a way of dynamically changing the connections. The microcontroller had to be able to control the connections, which means that the switching circuit has to accept TTL logic control. Clearly the best solution involved using MOSFETs that can be driven with logic line voltages.

The solution was to use a MOSFETs circuit design known as a H-Bridge (Scherz 2000). With this circuit, by activating two opposing sets of MOSFETs either lead of the motor can be connected to power or ground simply by changing the values of the logic inputs. Fly back diodes are included to help protect the circuit from the voltage transients that occur when switching an inductive load.

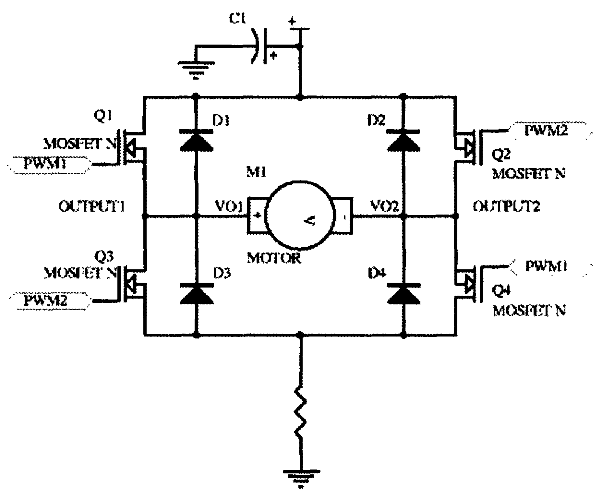
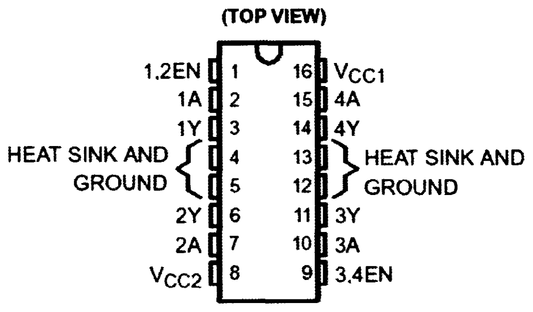


Figure 3.26: schematic diagram of a H. Bridge motor driver circuit

This part was found in a dual in line package (DIP package). The Texas Instruments SN754410 is a quadruple half H driver but can be easily configured as two full H. Bridge drives. This part is especially useful in keeping part count low since two motors can be driven off of one driver chip.



Quadruple Half-H Driver Package pins of SN754410

INPUTS†		OUTPUT
A	EN	Y
H	H	H
L	H	L
X	L	Z

H = high-level, L = low-level
X = irrelevant
Z = high-impedance (off)
Logic Controls ofSN754410

Figure 3.27: Pin out and logic diagram of SN754410 package

This chip is designed provide bidirectional drive to inductive loads such as a DC motor. To turn on the outputs to the motor first the Enable (EN in figure) corresponding to the output must be set high. If the Enable is high then the output Y follows the input A. So, both EN1 and A1 must be high for output Y1 to be high. The SN754410 Quadruple Half-H Driver has two separate power supply inputs. VCC1 is for the CMOS logic built into the chip. The second supply input is for driving the inductive load. This dual supply setup allows for reduced power consumption in the device for logic operations and allows for the motor to be driven with a supply isolated from the 5-volt supply used to power the rest of the circuit.

3.7.2 Pulse Width Modulation Drive

Using an H Bridge motor driver there's no middle ground, either the motor is connected to full power or it is not. Pulse Width Modulation (PWM) is a method of switching the power on and off at a set frequency while varying the duty cycle. The duty cycle is the percent of the time the signal is a high. 100% is a constant high value and 0% is always off.

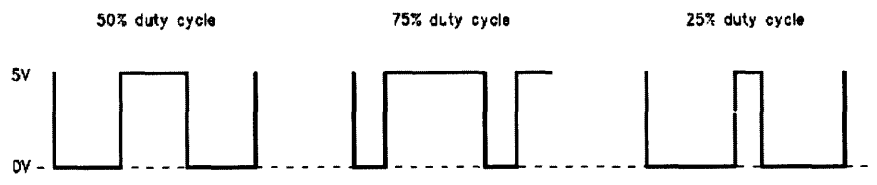


Figure 3.28: PWM waveforms for varying duty cycles

With a microcontroller this signal is created using a software counter and comparator. When the counters value is less then the value in the comparator we will call this value X, the output is set low. When the counter value is greater then X the output goes high. X can be defined as how high your counter can count times the desired duty cycle. For an 8-bit counter, the max value is 256. For a 50% duty cycle you want X to be 256 times 0.5, which equals 128. With an 8-bit counter you have 256 different power levels available. By changing the number of bits in the counter you can change the resolution of your PWM drive. Using a 10-bit counter you have 1024 possibilities with in the same voltage range. If more or less resolution is required depends on the application. With PINS there was no need for any resolution grater then 8-bits.

If the PWM waveform is hooked to the enable pin of the H-bridge motor drive, the motor can be turned on and off very quickly; limiting the current, and making the actuator move very slowly. Also, as apposed to limiting voltage to control the motor speed, which severely limits the torque due to the weaker electromagnetic field, use of PWM varies the period that full voltage is supplied, thereby maintaining high torque, for a very short period of time.

3.7.3 PID For Screw Actuator System

A PID controller takes as inputs the desired position and the actual position. The encoder digitizes the position of the motor, which is represented by counting the clicks of the shaft encoder. This value is taken as the actual motor position. The desired, position is retrieved from the central control computer through the serial interface. The difference between the desired position and the actual position is the error. The output of the controller is computed by multiplying the error by a constant. This number is then added it to a multiple of the derivative of the error and the integral of the error. The derivative of the error is approximated in real-time as the first difference of the error. The integral is approximated using an error accumulator whose initial state is set as zero to be zero. The Equation repressing this can be found in appendix C. Using PID control the screw actuators are setup to have a first order response to step changes in requested height (Gould, 1997).

3.8 Power Supply

Most of the power consumed by the PINS system happens when the motors are stalled. The motors get stalled when they're trying to move to location and the user is not letting them. Under normal operation with the load of moving a pinion actuator the motors used pull about 120 mA. When stalled, the motors pull 270 mA (Mabuchi Motor). There are 16 motors in the pinion actuator version of PINS. This is a max current of 4.32 amps. The motors are powered with a surplus 22-amp computer power supply. The supply has enough power to drive 80 PINS. This is 110 watts of power. A normal AC power outlet can supply about 15 amps of current, at 120 volts. This is 1800 watts of power. This means by simply using spare computer power supplies (a box of these was collected using the MIT reuse mailing list), over 1200 pins could be driven, all at stall current, which is a very unlikely condition. If and interface with more the 1200 pins is built in the future this power problem can be fixed using more advanced solutions.

The digital logic of the system operates on a five-volt supply with low current. To provide protection from a high current of the motor supplies this is provided with a simple switching wall power transformer.

3.9 Fault Recovery and Troubleshooting

Several features were included both in hardware and firmware to make PINS more robust and easier to troubleshoot. In hardware, several status LED's were included to aid in diagnosing problems. For example, LED's were added to verify operation of the five-volt logic supply, operation of the encoders, serial data reception and transmission, and a general flashing status LED to indicate that the firmware was running normally.

Several features of the AVR microcontroller code facilitate fault isolation and recovery. Also included in the firmware are several initialization checks to verify proper operation of the SPI communication. The control computer is able to cycle through all of the chips in the unit and test each for functionality. If a faulty microcontroller is found the computer can identify exactly where it is. This drastically narrows how hard it is to find and fix problems.

Also the modularity of the design of PINS makes it very easy to test and replace single units, in what otherwise would be a horrendously complex system.

4 Microcontroller Firmware

The software for the microcontrollers had to be written very carefully, and conservatively. The microcontroller only has a few kilobytes of flash memory that has to be allocated sparingly. The code had to be very reliable because errors not only can cause software crashes but also can damage the physical system. Also an effort was made so that if any mechanical parts broke the code would default to operations that would not damage the system more. The code for the PINS system has four main sections, pre-compiler definitions, startup, main loop, and interrupt functions.

4.1 Pre-Compiler Definitions

The Pins.h file contained a list of pre-compiler definitions. This file was very useful to make coding clean and easy to understand. It also was useful in correcting wiring problems in the hardware. This file allowed for any pin to be assigned to any of the functional blocks by name.

```
#define ENCODER1_1  bit_is_set(PINA, 0)
#define ENCODER1_2  bit_is_set(PINA, 1)
```

For example the encoder input for a channel was defined as ENCODER1_1 which when compiled became a test to see if that encoder input was high or low. However, to reassign the encoder input to a different pin only the Pins.h file had to be change while all the instances in the rest of the code refer to the same ENCODER1_1 variable. This was extremely useful when the first prototypes were being built and it was discover that several of the pins were hooked up to the wrong data lines.

This method of naming functions for pins was also used for the control lines for the motor drivers.

```
#define MOTOR1_ALOW    cbi(PORTA, 6) // set pin A6's output low
#define MOTOR1_AHIGH   sbi(PORTA, 6) // set pin A6's output high
```

Changing the number 6 in the example above to any number 0 through 7, inclusive, can change the motor outputs to any pin on Port A.

The other set of precompiled definitions that was vital for SPI communication was the *spi_commands.h* file. This file defines the byte codes for all the commands the microcontrollers can send and receive. Byte codes were design in such a way that they can be easily parsed on the microcontrollers side using bit masks and *if()* statements. This saves considerable processor overhead as apposed to running compare operations on 8-bit numbers.

4.2 Startup Code

The startup section runs once every time the chip is powered up. These commands make sure the chip is configured properly to run the operational code loop. Configuring the chip consist mainly of setting the correct configuration bits. The parts that have to be configured in the microcontrollers on startup include the state of the I/O pins. All the pins can be set up as inputs, outputs or as inputs with resistive pull-ups. Also during startup the pins that have external interrupts are setup to perform this function.

4.2.1 Setting Tri-State Values of I/O Ports

Setting whether each pin is an input or output on the microcontroller is critical for correct operation. With the AVR microcontrollers this is done by loading a byte into the I/O registers at startup. Because the chip has 16 I/O pins it requires two bytes to set all the pins. These bytes are loaded into the I/O registers using the commands DDRA for the port A pins and DDRB for the port B pins.

```
DDRA = PORTA_IO;
DDRB = PORTB_IO;
```

PORTA_IO and PORTB_IO actually just stand in for the bite values. For the PINS system PORTA_IO and PORTB_IO are set as follows in the software's pins.h configuration file.

```
#define PORTA_IO 0xc0       //set pin 7, 8 to outputs
#define PORTB_IO 0x0a       //0x0a  0000 1010 sets pins 1, 4 to output
```

For Port A, 0xC0 is the hex code for 1100 0000 showing that pins 6 and 7 are used as outputs while all the other pins are passive inputs, even if they are not used. These outputs are used to control the motor driver chip, while the inputs used come from the photo interrupters. For Port B, 0x0A relates to the binary of 0000 1010, making pins 1 and 3 outputs. The output pins on the sport come from SPI transmit and the output that controls the LED. The input are SPI receive, the SPI clock signal, and the microcontroller reset and latch signals.

There is one other command that is initialized at startup. This commands turns on the internal resistor pull-ups for the desired input pins.

```
PORTA |= 0x0f;
```

This sets the control register for Port A that turns on resistor pull-ups for pins 0 through 3. These are the pins the photointerrupters connect to, using the internal resistor pull-ups should decrease the time it takes for the encoder outputs to rise towards the supply voltage. If the encoders outputs are unable to go low enough to register as logic zeros the resistive pull-ups can be turned off, but this was not necessary in the final PINS system.

4.2.2 Setting Up the SPI Port

Having the SPI Port configured correctly every time the microcontrollers reset is vital for correct system operation. Without the SPI port there is no communication and no interface. During start up *setup_spi();* is run and calls for a sequence declared in SPI.C.

```
void setup_spi(){
  // Initialize the USI control register
  USICR = SPI_CONTROL_SETTINGS;  //SPI_CONTROL_SETTINGS defined as 0x1C

  // Initialize buffer indices
  rx_i.head = rx_i.tail = 0;
  tx_i.head = tx_i.tail = 0;

  // Initialize spi status flags
  spi_s.rx_overflow = 0;
  spi_s.tx_underflow = 0;
  spi_s.rx_empty = 1;
  spi_s.tx_full = 0;

}  // Operation Complete!
```

The *USICR =SPI_CONTROL_SETTINGS* sets the byte value 0x1c as the control bits for the built-in shift register. These control bits set whether the shift register uses a rising or falling edge as the clock. To prevent sampling error's for PINS we always define the clock on the rising edge, which means setting bits 3 and 2 both to 1. The control bits also set whether the SPI unit's pins are disabled or set in two or three wire mode. Setting bits 5 and 4 to 0 and 1 respectively sets the system in three-wire mode.[14]

The rest of `setup_spi()` initializes the send and receive buffers that are run in software allowing for the creation of an abstraction layer between the communications and the normal operation code. It also initializes the status flags that aid in the operation of the communications layer.

4.2.3 Enabling External Interrupts

The last section of the startup routine enables external interrupts on pin B-6. This pin receives a LATCH command from the controlling computer telling the microcontrollers that they should immediately process the data currently in their shift register. This command must be executed promptly so that the data remains valid, hence the need for an external interrupt.

```
enable_external_int( _BV(INT0));  // enable pin change interrupt pin B6
MCUCR = ( _BV(1) | _BV(0));
sei ();  //enable global interrupts
```

This code sets pin B6 to accept external interrupts. The two lowest order bits of the MCU control register, MCUCR, are both set to 1 making the interrupt occur on a rising edge. The system can also be set to interrupt on low voltage levels or falling edges. Finally the global interrupts have to be enabled, this allows for all enabled interrupts to be processed. Without enabling global interrupts or if they are disabled, no change can happen due to external influences on the flow of the microcontroller code.

4.3 Main Code Loop

The main control loop is called from the actu.C file, and is run in a *while(1)* command. This just means that once the code entered this loop it only leaves the loop for interrupt commands.

[14] ATTiny26 Datasheet

The commands executed in this loop occur in the following order.

4.3.1.1 Flash LED

```
flash_led();    // just to let user know program is running
```

This is a simple command called from functions.C file, which turns a LED on every time a 16-bit number overflow's and then turns it off the next time the number overflows. Every time `flash_led()` is called it adds 4 to the 16-bit number. This means the LED is on for 16,383 loops of the processor code, and it takes 32,766 loops of the main processor code to flash the LED on and off once. This results in a blink rate of about 2 Hz or once every half-second.

4.3.1.2 Capacitive Sensing **(for screw actuated PINS system only)**

```
if (cap_result = cap_sensor( CAP1_PIN, &CAP_PORT, &CAP1_IN, &CAP_DDR)){
      cap_sense_dispactch(cap_result);
    }
```

This line of code is a call to cap_sensor which is located in *capsense.C*. This code controls the pin connected to the capacitive sensor plate. This code counts how many times it has been executed for timing. This is effectively the same as counting main code loops which are about 1/50000th of a second each. After resetting the voltage on the capacitive sensor and the counter clock, the system counts until the number of loops equals the defined value `touch-constant`. At this point the capacitive sensing pad is tested to see if it has a logic higher or logic low. The local variable `result` is set to equal *`cap_sense_no`* `or` *`cap_sense_yes`* but does not return the value to the main code loop. The counter continues to tally main system loops until the counter value equals the defined value *`end_stop_constant`*. At this point the microcontroller test the voltage on the pin again. If the logic level is still 0 it means the actuator has hit the mechanical stop and the pin is grounded. If the pin is grounded `result` is set to equal *`end_detected`* `and` this value is return to the main program loop. If the end was not detected than the original *`cap_sense_no`* `or` *`cap_sense_yes`* is returned to the main program loop.

For all the times *cap_sensor()* is called and the test cycles are not finished testing the capacitive sensor pin, it returned *`CAP_SENSE_NOT_READY`*. *`cap_sense_not_ready`* is defined as a logical zero, so returning *CAP_SENSE_NOT_READY* means returning a zero. Because the value from ***cap_sensor()*** was returned in an *if* statement the result is only past to `cap_sense_dispactch` when it was *end_detected, cap_sense_no* or *cap_sense_yes*.

The function *cap_sense_dispactch int()* is located in functions.C and determines what the response of the microcontroller will be to the state of the button. If the mechanical end stop is detected than the system turns off the downloader and reset the counter to zero. If a user's hand is detected than the actuator moves down away from the pressure and if nothing is detected than the system stays the same.

4.3.1.3 Tracking Encoder

```
track_encoder(ENCODER1_1, ENCODER1_2);
```

This function implements the software encoder, using a finite state machine. *ENCODER1_1, ENCODER1_2* return bit values present currently on the assigned pins. These values are past to the program ***track_encoder(value1, value2)*** in the ***encoder.C*** file. ***Track_encoder*** has seven cases; it is initialized in case 0. These cases make up the elements of the finite state machine discussed in the hardware implementation section 3.3.3. When a proper conditions are met, one is either added or subtracted from the *shaft position* which is a 16 bit system global variable representing the height of the actuator shaft.

4.3.1.4 Data Transmission

```
if (!spi_s.rx_empty)  message_handler(spi_rx_byte());
```

This message test to see if the message-received buffer is empty. If it is not empty then the message byte is passed to the message handler located in *fuctions.C.* This message handler finds the operation the received byte calls for and executes that function. It does this by using byte masks and case statements.

4.3.1.5 Setting the Movement Control Flags

```
posistion_control();  // control for actuator shaft1
pwm_flag = pwm();
```

When the actuator unit has received a command from the controlling computer `posistion_control()` is the module that tells the motor how to move to get the actuator to the desired height. This program is also called from the *functions.C* folder. If the actuator has received a command to go to a set height the *go_to* flag will be high. Position control first checks this flag, if it is high the control algorithm is launched otherwise it does nothing. The system LED is set so it stops flashing and is constantly on, this allows the user to see that the actuator is operating in position control mode. The algorithm also checks to see if the actuator is at its maximum height (defined in software to keep the actuator from damaging itself) in which case it will not be allowed to move up any further. There's also a check to see if the actuator is at height 0 in which case will not be allowed to move down any further. After the safety checks have been performed the system goes through a routine of defining whether the current position as above or below the desired position and setting the motor masks accordingly.

The pulse width modulation algorithm as mentioned before, was used to make the screw actuator system behave with a first-order response. Varying the duty cycle of the PWM signal depended on the error between the desired an actual position and how quickly the error changed. However, the pinion gear system is naturally first-order so processor overhead that would be needed for the control algorithms is not required. A PWM signal is used to decrease the time averaged current given to the motors. This is because the motor driver chips can only operate with a minimum output voltage of 4 1/2 volts. This generated too much torque in the motors causing the gears to strip. So a PWM signal is generated to reduce the average voltage motors are exposed to, reducing their output power.

The PWM control code operates by letting the position control algorithm decide which movement should happened (up, down, stop) but then selectively masking or unmasking the motor drive flags, creating PWM.

4.3.1.6 Moving the Actuator

```
    if(motor1flag & MOTOR_UP_MASK) move1_up();
    if(motor1flag & MOTOR_DOWN_MASK) move1_down();
    if(!motor1flag) move1_stop();
```

This is the section of code were the action actually happens. Here the motor masks are applied to the motor flag variable. If the flags is bit wise ANDed (&) with the motor mask and returns a non-zero value, move1_up, move1_down or move1_stop is run.

Bitwise & operation

Possible motor1flag	1110010
MOTOR_UP_MASK	0000010
Result	0000010

A bitwise AND compares each bit of two different bytes, and returns the ANDed values. Only one of the movement commands should run in any given code cycle. Flags are used instead of setting the outputs to the motor directly because that way, only the desired output is given to the motor drivers. It is possible that the output values could change several times because the Capacitive Sensing code wants the motor to do one thing and the Movement Control wants something else. The output would just flicker unless dominance is set and the output is applied once per cycle. This is also why applying the values on the motor is the last operation preformed in the main code loop.

4.4 Interrupt Process Code

```
SIGNAL( SIG_INTERRUPT0 ){
  spi_latch();
}
```

This code is declared outside of the main program loop. *SIGNAL* is the process that handles the interrupt and *SIG_INTERRUPT0* is the microcontrollers definition for external interrupt on pin B 6. All the code in the brackets of the *SIGNAL* are run then the system branches back to were it left the main code loop. The system cannot branch to an interrupt while it is processing an interrupt. There is a control flag that is set when the system branches to the interrupt with is only reset when the interrupt is over. For this reason the code contained in the interrupt process is as short as possible, so no future signals will be missed.

The function *spi_latch*, grabs the 8-bit value in the shift registers and loads it on the spi received buffer so it can be processed. It then checks to see if there is any data to transmit in the spi transmit buffer. If there is a byte in the buffer it is popped from the stack and the byte value is loaded into the shift register.

5 Production

Two different versions of PINS were produced, the screw actuated and the pinion gear version. They had very similar circuit designs and only differed in their hardware. Both were designed on PCB, and had stands built. However, since the pinion gear version of PINS was the most advanced iteration its PCB layout, assembly, and mounting will be discussed in this section.

5.1 PCB Layout and Manufacturing

The process of generating a PCB design was first to draw the circuit schematic in Protel Client's schematic editor. The schematic can then be used to generate a file known as a netlist, which contains all the components of the circuit. The netlist also contains references to their PCB footprints, and their electrical interconnections.

With the completed netlist the layout of the PCB can begin. The first step is designing the size of the board. The size of the board for the PINS system is very important because these boards need to be very tightly packed to give the density of actuators required. This led to a rectangular designed for the boards, making them very tall and narrow. (See appendix B) The number of actuators that could be controlled by a single board decided the width of the board. After the width of the board was set the height was adjusted to fit the required components. The next step required placement of components on the board. The PINS system has requirements for where the photointerrupters and connectors have to be. The photointerrupters have to be very precisely place so that the mechanical mechanism, depending on whether it was the screw actuator or pinion drive system, would interact with the beams of light. The placement of these components had about a millimeter of tolerance. The connectors also had to be carefully placed so that they can easily be accessed and would not get in the way of the moving components. The microcontroller and LEDs were placed so that it is easy to tell which microcontroller corresponded with which LED, making testing and debugging easier. From this point the other components were placed to try to minimize the trace length, and to try and reduce the complexity of routing the traces to connect the parts.

The layout of the traces was done by Protel's auto-router. This router runs an algorithm to try to minimize the length of all the traces in the system. The user however, can give design directives, and specifically can give higher priority to certain traces. As mentioned earlier, the photointerrupters outputs are given priority to try to make sure that the signal is exposed to as little noise as possible. Having shorter traces for the photointerrupters also decreases the capacitive load that has to be driven every time the signal changes, increasing their response time. Another set of traces that are given priority are the power and ground lines for the motor drivers and the power lines running from the motor drivers to the motors. Because of the high currents going to the motor drivers they have the most potential to cause noise on other signal lines. The PWM on the motor power lines has even more potential to cause problems since these lines will also be exposed to the noise and spikes created by the motor's inductors.

To get the smallest board size, surface mount components were used whenever possible. The motor drivers were the only component that could not be replaced with a surface mount version; however, DIP-16 surface mount sockets are available. Using the surface mount dip sockets was an excellent solution since the motor drivers can blow up if over driven and if they could not be easily replace the entire board would be useless. Some other components that the system was originally designed to use were on backorder or no longer available, but often it was not very hard to find a similar component from a different manufacture. So luckily there were no long delays in waiting for backorder parts.

The AVR X 16 could not be fit onto a two-layer board that fit only under the footprints of the actuators that it controlled. There are too many traces and surface mount components in to small an area to make the connections required. So for the first prototype the board had to be made much wider. The later version of the AVR board was design for a four layer PCB allowing for much denser packing of the components and traces.

After the printed circuit boards were laid out and finalized, they were output to an industry standard format known as the Gerber format. Gerber files are used by computer-controlled machinery to manufacture the printed circuit boards. The actual PCB manufacturing was carried out by PCBexpress.com, who can ship next day on two layer boards and require only a three-day lead-time on four layer boards.

The original boards were ordered without a solder mask, partially because solder mask is expensive, because there's a faster turnaround time without it, and because it makes it easier to make physical corrections to the circuit, cutting and splicing traces, if it hasn't been solder masked.

5.2 PCB Assembly

Assembling printed circuit boards can be an extremely tedious task and many of the problems that can be found with a final design happened due to mistakes during a step. If a large number of PINS boards where going to be developed they would be outsourced to companies are specialized machines and testing equipment to do this process.

However, due to mechanical reliability issues with the system we never reach the stage and the prototypes built were hand assembled with the help of several UROP students.

5.3 Mounting Racks

The PCB boards were designed to be placed on mounting racks. These racks were specially designed for the PINS system and they were manufactured in the MIT media lab using a laser cutter. Designs were created in Corel Draw 10, which uses vector graphics. The laser cutter takes these vectors, and follows the path they draw with an infrared laser cutting the plexiglass. These cut out plexiglass sheets are given depth using board spacers. These Plexiglas mounting racks were also used to guide the motion of the actuators, and hold the PCB boards.

5.4 Cabling

The cabling and connectors used were based on 16 pin DIP (dual in line socket) chips. The choice to use 16 pin DIP sockets as connectors had several reasons. The first reason was that the dimensions of the 16 pin did socket is very well known and highly standardized. The second reason was because of the price of this versus other connectors was extremely low, many specialized connectors were several dollars each while 16 pin DIP sockets were just a few cents.

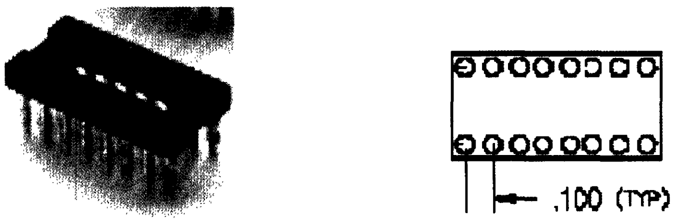


Figure 5.1: 16 Pin DIP socket image and schematic diagram

This is also a very mechanically stable connector, with solid electrical connections and it can be trusted to stay connected. Furthermore, during the debugging process it was very easy to plug these connectors into proto-boards where the logic levels can be tested in problems can be isolated

5.4.1 AVR board to AVR board connection

The common bus and the SPI data signals are passed between the AVR boards using the 16 pin socket connectors. Each board has two 16 pin did socket connectors; one is the input from the previous board and one outputs to the next board. The top two pins on one side are the input pins where the SPI data for the previous boards is received. On the same side of the dual in line package logic power and ground are also connecting between all the boards. On the other side of the package the eight common bus PINS are shorted between all the boards.

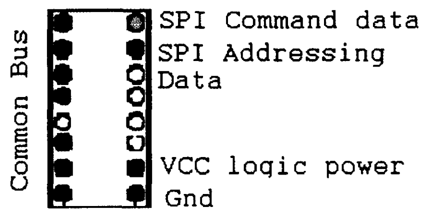


Figure 5.2: Pin Assignments for AVR board to AVR board connection

The output 16 pin connector on any the AVR boards in the series can be connected back to the computer completing the data loop.

5.4.2 Computer To Control Logic Connection

A standard 25-pin parallel port connector was used to plug into the computers parallel port. The other end of this cable was cut off and required wires were plug into a dip socket which could be easily plug into the PCB or proto-board with the routing logic that connected to the X 16 boards.

6 Other Accessories

6.1 Separate Power Supplies

The motor drive system, by its very nature, generates a great deal of electrical noise through PWM switching of high currents. This electrical noise could be extremely hazardous to logic on the controller board, and could or may cause improper operation at the very least. This problem was avoided by supping two completely separate power supplies for the five-volt logic systems and for the motors and motor drivers. There is also the extensive use of bypass capacitors throughout the system making the power lines less susceptible to noise.

7 Future Work

There are four main areas open for future work on PINS. These areas are, increasing the reliability and durability of the mechanical systems of the actuators. Adding enhanced tactile information to the individual actuators. Third is further development of the software programs that interface and used the three dimensional input/output features of PINS. Forth is the development of synchronized PINS platforms. Two or more coordinating computers could be networked to synchronize the movements of several PINS interfaces.

7.1 Increased Mechanical Reliability

The most important improvement is the development of a more reliable system for the mechanical actuators. As discussed in this thesis many approaches were taken to developing a reliable actuator for mechanical interface. Although there were some successful attempts none these reached a satisfactory level to allow the desired functionality of PINS.

7.2 Tactually Enhanced Actuators

The second area for future work is in adding enhancements to the pins. Each microcontroller currently has available ports to handle such expansions.

- A tactile enhancement could include electrically activated polymers. These polymers would allow for the feel of the pins to become soft or hard trading a more complex tactile sensation for the user. This also allows for the sense of having texture instead of just hard plastic rods.

- Another enhancement would be to play small Peltier junctions on the top of the pins. These junctions could then be powered to create hot and cold areas over the PINS input surface, adding yet more tactile sensation and another dimension to the data represented.

- A very simple enhancement would be to place LEDs inside of the actuator pins. Multicolored LEDs can be used to create simple displays and highlight certain areas of the interface.

- Another simple enhancement would be to take advantage of the vibrational modes in the actuators and used these for vibro-tactile communication. There exists a growing body of research on the perception of touch. Experiments on the use of vibration for human-computer interaction have shown that humans can distinguish vibration well enough to use it as a code for communication [Geldard, 1960]. Vibration could be yet another dimension to the PINS interface.

- Perhaps most challenging and intricately tied with creation of a reliable actuator, is the tactile enhancement of a much finer actuator array.

7.3 Software Interface Development

The third area for expansion has to do with the software that uses the PINS interface. There is currently a very robust API developed for the PINS platform, and a couple of simple applications and simulations designed to run with it. As discussed, these include a constant volume mode where PINS integrates the volume contained within each pin and adjusts the whole system to keep this volume the same as the system is deformed. Also there is a water simulation, which causes PINS to act like a waterbed, where the user can perturbed the system. Finally there is also the recorded and play

back mode, which simply records movements of the actuators and plays them back on request.
With a little creativity there are hundreds of different and useful applications that can be developed. Sensetable [Tangible Media Group, 2004], developed in the Tangible Media Group is an example of how once a reliable in with a platform is developed numerous advanced software projects, some of which using the interface in ways never conceived by the creators, can spring off of that work.

7.4 Synchronized Movement

Two or more coordinating computers could be networked synchronizing the movements of several PINS platform. This could be used for remote group collaboration allowing everyone to see and manipulate the same physical display simultaneously. Some work with synchronized haptic interfaces has been done with projects such as In-Touch [Brave, 1998] and actuated workbench [Pangaro, 2002], but remote collaboration with PINS could bring this work to a new level.

8 Conclusions

This thesis discussed the design and implementation of the electrical hardware, firmware and mechanical systems of a novel computer interface. The hardware developed allows for communication between an easily scalable number of discrete actuator units and a coordinating computer.

The use of many different types of physical actuators was explored and along with the positive and negative aspects of each as a tangible interface. It is clear that the ideal design for an actuator has not yet been reached, and I believe work in this area should continue.

This thesis also explored systems for communications between a complex peripheral, and a control computer, and how data processing can be distributed. It was found that by using a distributed processor network that a high level of feedback and control can be achieved using a relatively low speed communications path.

PINS was and continues to be an extremely complex design challenge. Creating a smooth and intuitive interface between a users' physical world and digital information is going to continue to be one of the most challenging areas of research in the years to come. PINS was an interesting experiment into this field, but due to issues with mechanical reliability the physical system never reached a satisfactory level of operation. To make a large-scale model of PINS, a mechanical actuator that is simpler to construct and more reliable in operation will have to be developed. However, despite the problems with PINS' mechanical system the distributed microcontroller platform designed in this thesis could have many uses in other haptic interfaces. This is especially true in systems with a large number of similar units requiring continual monitoring to create an active interface.

References

1. Brave, S., Ishii, H. and Dahley, A. Tangible Interfaces for Remote Collaboration and Communication. In Proceedings of CSCW '98, ACM Press, pp. 169-178.
2. Brave. S., Nass, C., Sirinian, E., Force Feedback in Computer Mediated Communication. Proceedings at HCI International 2001 Conference (New Orleans, LA).
3. Chang, Angela. ComTouch: A touch-based remote communication device, MS Thesis, MIT, June 2002.
4. Fairchild Semiconductor DM74150, DM74151A datasheet
5. Fairchild Semiconductor MM74HC4514 datasheet
6. Fogg, B.J., Cutler, L., Arnold, P., and Eisback, C. HandJive: A Device for Interpersonal Haptic Entertainment, to appear in Proceedings of CHI '98 (Los Angeles, April 1998), ACM Press.
7. G. Moy, C. Wagner, and R.S. Fearing. Compliant Tactile display for Teletaction, IEEE Int. Conf. On Robotics and Automation, April 2000
8. Geldard, F.A. Some Neglected Possibilities of Communication Science 1960 May 27; 131, (3413): 1583-1588.
9. Gould, L.A., Markey, W.R., Roberge, J.K., Trumper, D.L. Controls Systems Theory. Not published. 2nd revision, 1997.
10. Horowitz, P. and Hill, W. The Art of Electronics. 2nd ed. New York: Cambridge University Press, 1990.
11. Immersion Corp. Cyber Grasp 1.2 http://www.immersion.com/3d/docs/CyberGrasp_030619.pdf , Dec 2003
12. Ishii, H., Kobayashi, M. and Arita, K. Iterative Design of Seamless Collaboration Media. Communications of the ACM (CACM) (1994). ACM Press, 37, 8, 83-97.
13. Iwata, H.,. "Project FEELEX: Adding Haptic Surface to Graphics". Institute of Engineering Mechanics and systems, University of Tsukuba, 2001
14. Mabuchi Motor FF-N20PA/PN datasheet
15. Mak, S. Y., and K. Young. Floating Metal Ring in an Alternating Magnetic Field. American Journal of Physics 54 (1986): 8008-811.
16. Massie,T., and Salisbury, K., The PHANToM Haptic Interface: A Device for Probing Virtual Objects, ASME Winter Annual Meeting , DSC-Vol.55-1 , 1994
17. McNeely, W., Robotic Graphics: A New Approach to Force Feedback for Virtual Reality, Proc, of IEEE VRAIS'93 1993
18. Microchip Technologies PIC16F876 datasheet.

19. Negroponte, Nicholas and the Architecture Machine Group, MIT. "Seek." Originally shown at Software, Information Technology: Its New Meaning for Art exhibition, Jewish Museum, New York, 1970

20. Omron EE-1108 datasheet

21. Omron EE-1131 datasheet

22. Oppenheim, A.V., Willsky, A.S., Young, I.T. Signals and Systems. Englewood Cliffs: Prentice-Hall, 1997.

23. Overholt, Dan. The MATRIX http://xenia.media.mit.edu/~dano/matrix/ DEC 2003.

24. Pangaro, Gian A.. The Actuated Workbench: Actuation Technology for Tabletop Tangible Interfaces, MS Thesis August 2003

25. Pangaro, Maynes-Aminzade, Hiroshi Ishii. The Actuated Workbench: Computer-Controlled Actuation in Tabletop Tangible Interfaces. UIST, October 2002.

26. Piper, Ben. The Illuminated Design Environment: a 3D Tangible Interface for Landscape Analysis. MAS Thesis May, 2002.

27. Pericom PI5C34X245 datasheet

28. Scherz, Paul. Practical Electronics for Inventors. McGraw-Hill, New York 2000.

29. Tangible Media Group PROJECTS: http://tangible.media.mit.edu/projects/Tangible_Bits/projects.htm May 4, 2004.

30. Texas Instruments CD54/74HC251 datasheet

31. Texas Instruments SN74HC157 datasheet

32. Texas Instruments SN754410 datasheet

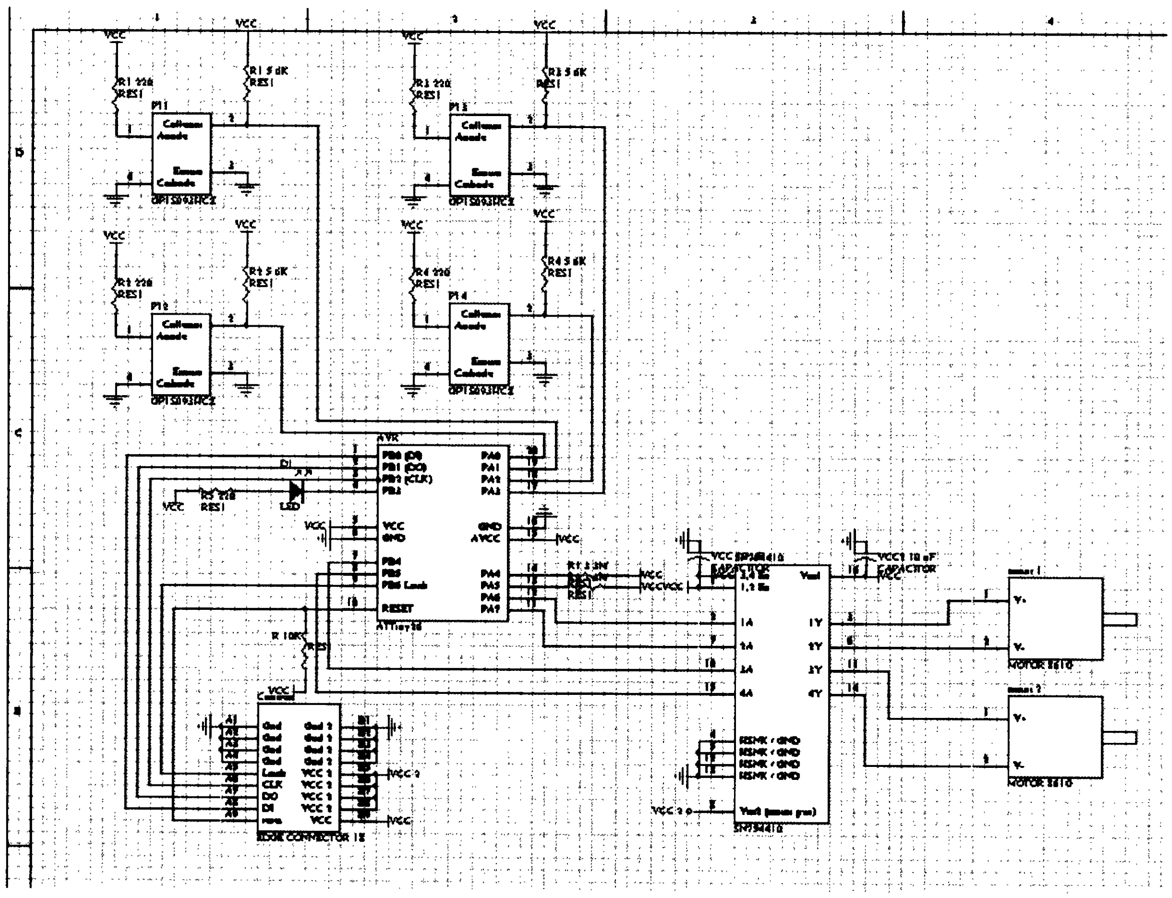
33. Toshiba TC74HC595 datasheet

34. Ullmer, B Ishii, H. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In Proceedings of Conference on Human Factors in Computing Systems (CHI '97), ACM Press, pp. 234-241.

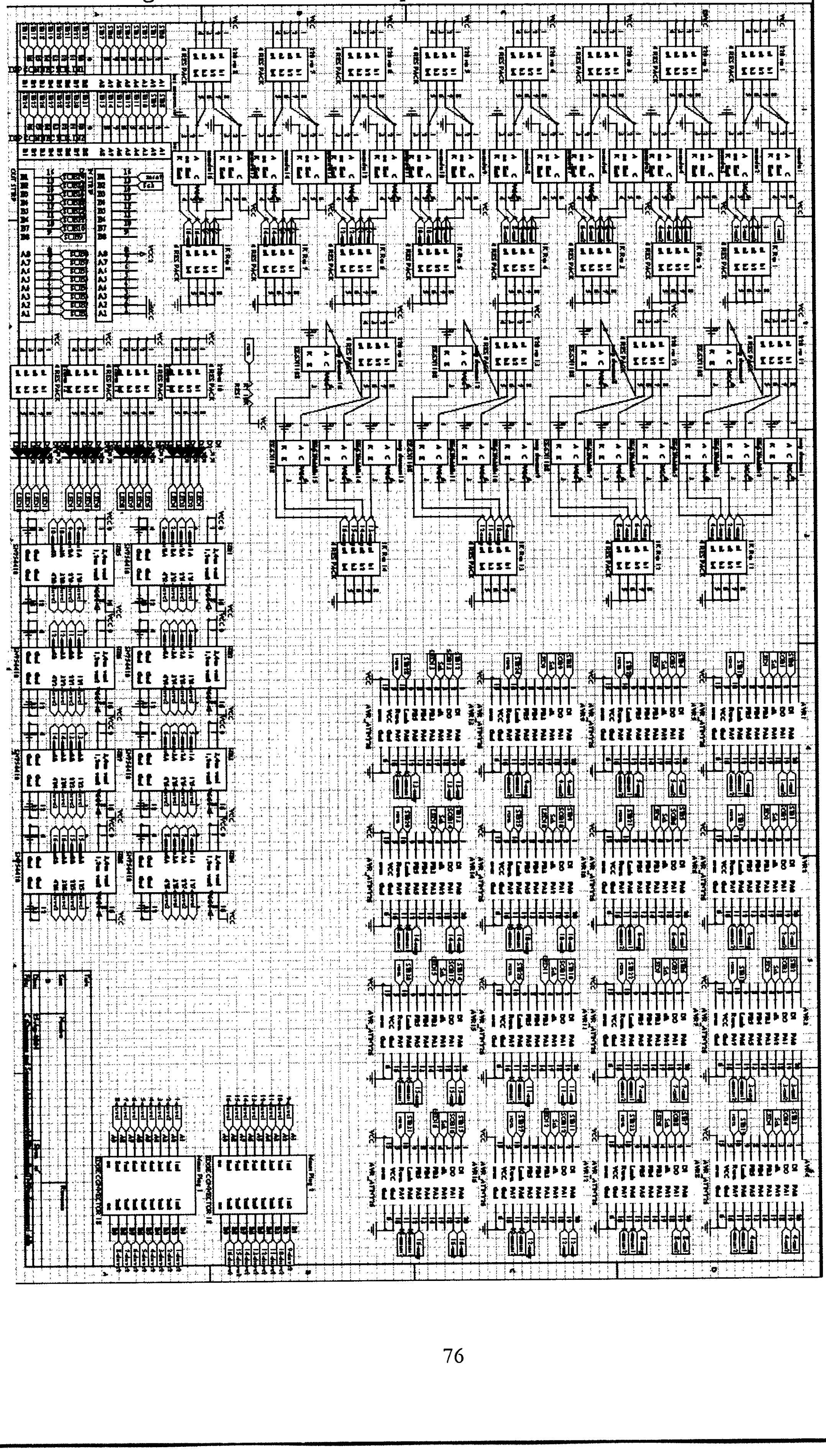
Appendix A: Circuit Schematics

This section contains circuit schematics for the PINS AVR X 16 controller board, the AVR X 16 boards and the earlier screw actuator boards.

Schematic Diagram for screw actuator system.

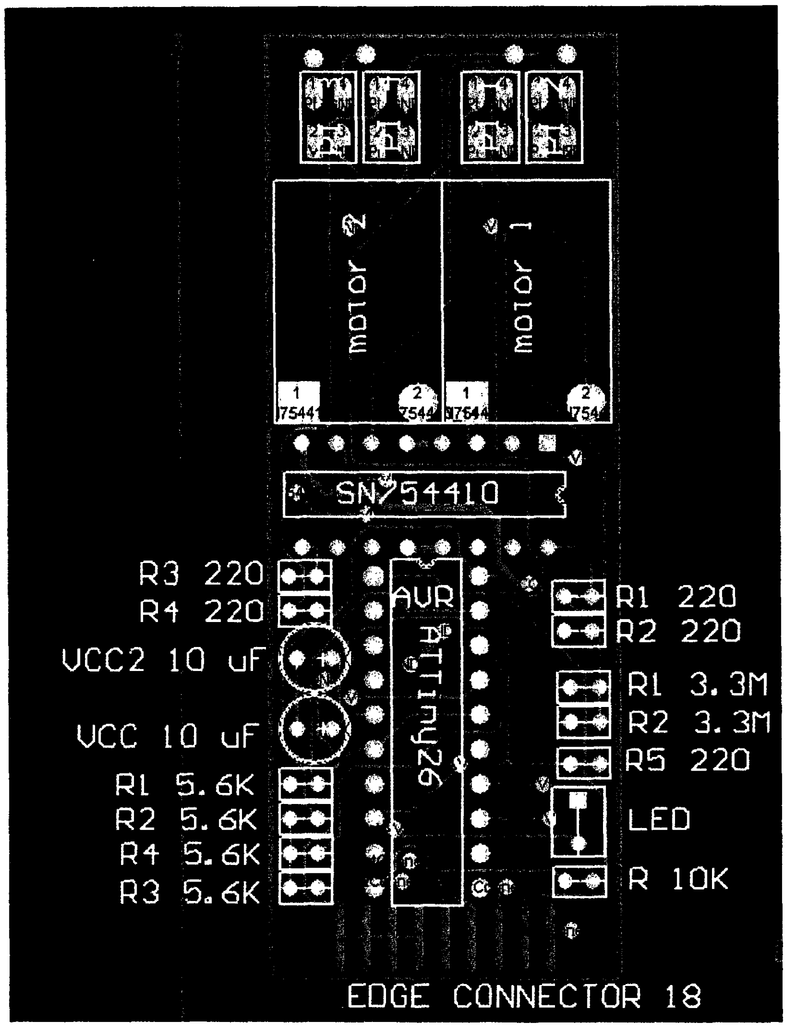


Schematic diagram for Micro controller portion of the AVR X 16 board.

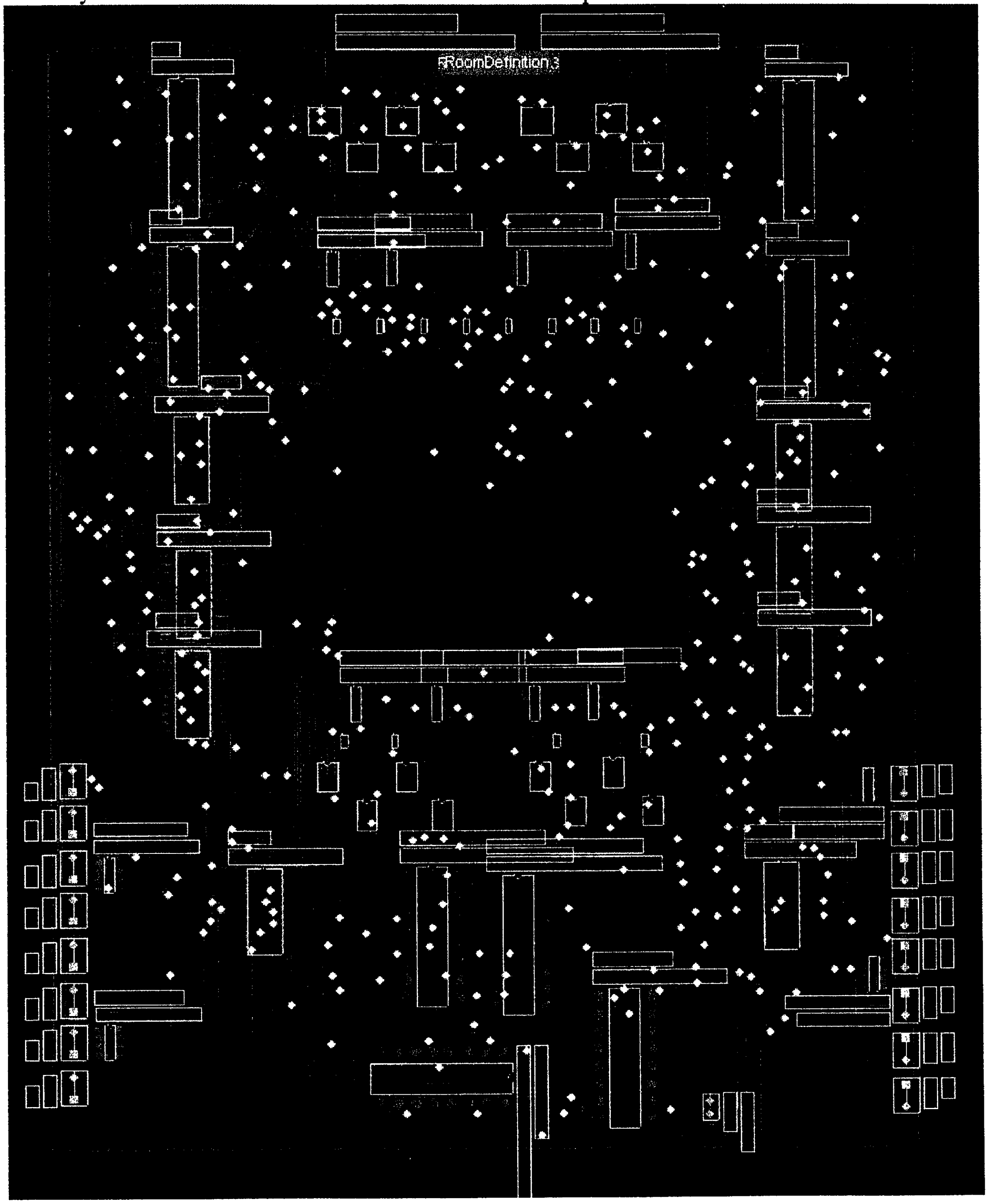


Appendix B: PCB Boards

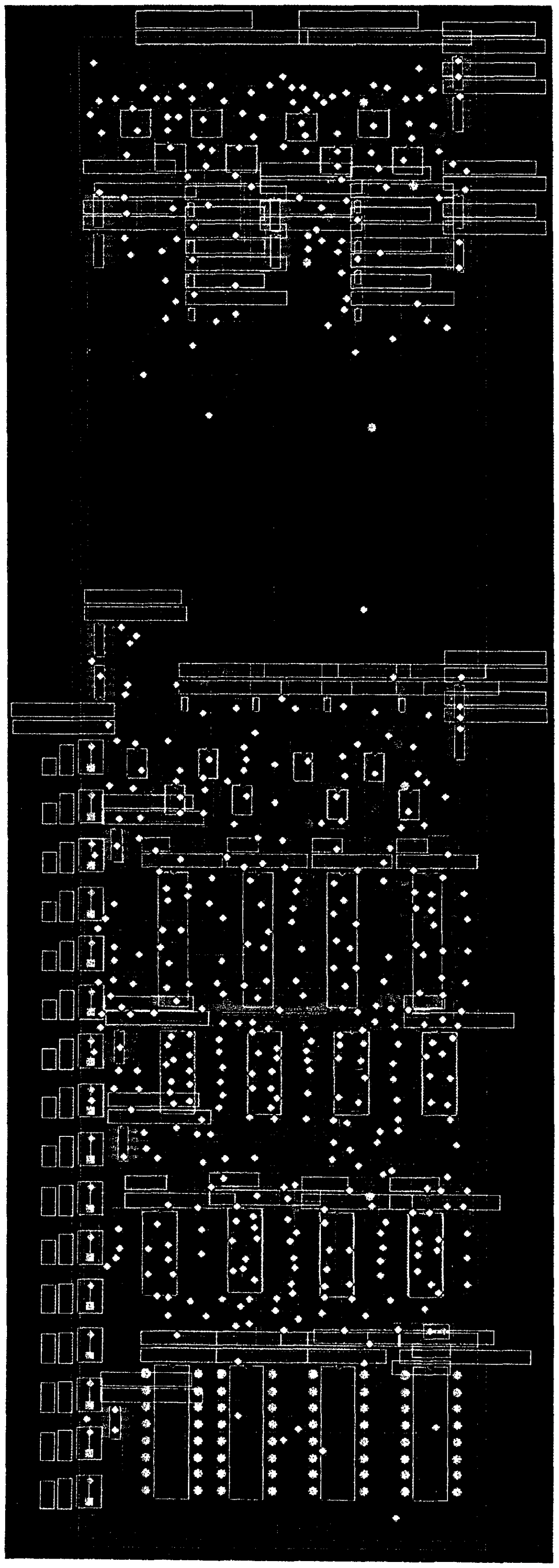
2 Layer screw actuator board



Two layer AVR X 16 board. The extra width was required to fit all the traces.



4 Layer AVR X 16 PCB board. Can fit the required form factors



Appendix C: Microcontroller Code

PINS Controller .

The following listing is the microcontroller code for the PINS controller. The code is written in C for and compiled using a GCC complier. The code is organized into libraries to facilitate reuse between different projects. Libraries are added using the compiler's *#include* directive as shown below. The very poor practice of having a "functions" library was done because there were very many small functions (move up, move down, flash led, ect) that did not make since to create separate libraries for them.

```
#include <avr/io.h>
#include "fuctions.h"
#include "pins.h"
#include "encoder.h"
#include "spi.h"

#include <avr/signal.h>
#include <avr/interrupt.h>
#include "spi_commands.h"


//Declare Global Variables
uint16_t old_shaft_posistion;
uint16_t shaft_posistion;
uint16_t disired_posistion;
//uint8_t pwm_flag;
extern uint8_t motor1flag;
//extern uint8_t motor2flag;

//Interrupts


SIGNAL( SIG_INTERRUPT0 ){
  //  YELLOW_LED_ON;
  spi_latch();
  //  YELLOW_LED_OFF;
}

//SIGNAL(__vector_default)
//{

//}

int  main() {


  // Declare local variables


  // Set Port A to output
  DDRA = PORTA_IO;
  DDRB = PORTB_IO;
  PORTA |= 0x0f;

  setup_spi();   // sets up SPI port

  enable_external_int( _BV(INT0));  // enable pin change interupt pin
B6
  MCUCR = ( _BV(1) | _BV(0));
  sei ();  //enable global interupts

  // USIDR = 0xaa;    //loads 0xaa into spi buffer at startup, just for
testing

  while(1) {

    flash_led();   // just to let user know program is running
//    if( END_STOP ) GREEN_LED_ON;
//    else GREEN_LED_OFF;

    // Shaft Posistion tracking
    track_encoder(ENCODER1_1, ENCODER1_2);

    if (!spi_s.rx_empty)  message_handler(spi_rx_byte());

    //pwm_flag = pwm();
    posistion_control();  // control for shaft1

    //if ( END_STOP) shaft_posistion = 0;
    pwm();  //pwm duty makes motor1flag = 0


    if(motor1flag & MOTOR_UP_MASK) move1_up();
    if(motor1flag & MOTOR_DOWN_MASK) move1_down();

    if(!motor1flag) move1_stop();

  }
  return;
}
```

Encoder FSM Unit

The following file implements a finite state machine that is used to decode the inputs from the photo encoders. This FSM tells the main program when the actuator has moved up or down a complete cycle of the gray code (see hardware section for decision of gray code).

```
#include "pins.h"
#include "encoder.h"
#include "fuctions.h"
```

```
void track_encoder(uint8_t ENCODER_1, uint8_t ENCODER_2){
  static uint8_t cur_state;

  switch(cur_state) {
  case 0:
    if (ENCODER_2) cur_state=1;
    if (ENCODER_1) cur_state=4;
    break;
  case 1:
    if (ENCODER_1) cur_state=2;
    if (!ENCODER_2) cur_state=0;
     break;
  case 2:
       if(!ENCODER_2) cur_state=3;
    if(!ENCODER_1) cur_state=1;
    break;
  case 3:
    if(!ENCODER_1) {
      cur_state=0;
      shaft_posistion++;
      //   flags[ encoder_number] .shaft_change = 1;
    }
    if(ENCODER_2) cur_state=2;
    break;
  case 4:

    if(ENCODER_2) cur_state=5;
    if (!ENCODER_1) cur_state=0;
    break;
  case 5:
    if(!ENCODER_1) cur_state=6;
    if(!ENCODER_2) cur_state=4;
    break;
  case 6:
    if(!ENCODER_2) {
      cur_state=0;
      if(shaft_posistion)
      shaft_posistion--;
      // flags[ encoder_number] .shaft_change = 1;

    }
    if(ENCODER_1) cur_state=5;
    break;
  }

  return;
}
```

Endcoder .h file

This file defines a number of encoders operating off of one microcontroller and the function definition for the software decoder.

```
#ifndef ENCODER_H
#define ENCODER_H
#include <inttypes.h>

#define NUM_ENCODERS 1      //the number of encoders running off of one
chip

void track_encoder_v2(uint8_t ENCODER_1,uint8_t ENCODER_2);

extern uint16_t shaft_posistion;

#endif  /* ENCODER */
```

Capacitive Sensor Control Code

The following is a code was used in the version of PINS that used capasitive sensors. It controlled the output state of pins connected to the sensor pad. It also controls the timing for when the voltages on the pad should be tested and tells the control program what the state of the pad is (being touched or not).

```
#include "capsense.h"
#include "pins.h"
#include <inttypes.h>

#define touch_constant 5 //set depending on hardware capapsidence and
number of sensors and software loop time 5
#define end_stop_constant 200 // stetting to test if end switch has
been hit.


uint8_t cap_sensor(uint8_t CAP_PIN, uint8_t* CAP_PORT, uint8_t* CAP_IN,
uint8_t* CAP_DDR){
  static uint8_t touch_counter;
  static uint8_t end_not_detected_result;
  //uint8_t i;
  uint8_t result;

  touch_counter++;
  cbi(*CAP_DDR, CAP_PIN);

  if (touch_counter == touch_constant){
  end_not_detected_result =
    bit_is_set(*CAP_IN, CAP_PIN) ? CAP_SENSE_NO : CAP_SENSE_YES;
//result =  test ? (if true) : (if false);
  // return result;
  }
  if (touch_counter == end_stop_constant){
    result = bit_is_set(*CAP_IN, CAP_PIN) ? end_not_detected_result :
END_DETECTED;
    touch_counter = 0;
    sbi(*CAP_DDR, CAP_PIN);
    return result;
  }
  return CAP_SENSE_NOT_READY;
}
```
Capacitive Sensor .h file

```
#ifndef CAPSENSE_H
#define CAPSENSE_H
#include <inttypes.h>

#define CAP_SENSE_YES        1
#define CAP_SENSE_NO         2
#define CAP_SENSE_NOT_READY  0
#define END_DETECTED         3
#define END_NOT_DETECTED     4

//#define SENSOR_NUM1          0
//#define SENSOR_NUM2          1

//#define NUM_CAPS 2  // number of capasitive sensors

uint8_t cap_sensor(uint8_t CAP_PIN, uint8_t* CAP_PORT, uint8_t* CAP_IN,
uint8_t* CAP_DDR);

/* //================================================== */
/* //end_stop_detect defineds and fuctions */

/* void end_stop_detect(uint8_t CAP_PIN, uint8_t* CAP_IN, uint8_t
cap_number){ */


#endif  /* CAPSENSE_H */
```

Fuctions

The following code contains many of the small but very important code sections used in the pinion implantation of PINS. These functions are:

Posistion_dispatch: sends a 16 bit number to the control computer
Message_handler: handles incoming 8-bit chunks to put together complete messages from the control computer.
Flash_led: Flashes the systems led as a default to show that the system is working.
Move1_up: Sets the microcontroller out puts to drive the actuator up.
Move1_down: Sets the microcontroller out puts to drive the actuator down.
Move1_stop: Sets the microcontroller out puts to stop the actuator .
PWM: Masks the flags that control the outputs to the motor depending on the control algorithm.

Posistion_control: Looks at the motor flags and implements the move_up, move_down, move_stop commands.

```
#include <inttypes.h>
#include "pins.h"
#include "spi.h"
#include "fuctions.h"
#include "spi_commands.h"  // bit codes for spi commands

// Public globals
uint8_t flags;
uint8_t motor1flag;

//uint8_t button_on =1;

// =================================
// Fuctions
//==================================

//returns a 16 bit number; the high order bit is 1 when the sring
contains data bit
//15 is high when the data is for actuator 2 and low for actuator 1

void posistion_dispatch(short actuator){
  spi_tx_byte(SENDING_ACTU1_POSTION);
  spi_tx_byte( (uint8_t)(shaft_posistion >> 8));  //send_high_byte
  spi_tx_byte( (uint8_t)(shaft_posistion & 0x00ff)); //send low byte
}

void message_handler(uint8_t msg){
  static uint16_t new_data;
  // static actu_data new_data;
  static msg_flags build_new_data;

  switch(build_new_data.count){
  case FIRST_BYTE:
    //new_data.high_byte = msg ;
    new_data = msg;
    new_data = (new_data << 8);
    build_new_data.count = SECOND_BYTE;
    break;
  case SECOND_BYTE:
    //new_data.low_byte = msg;
    new_data |= msg;
    build_new_data.count = COMMAND;
    disired_posistion = new_data;
    flags |= go_to1_mask;
    spi_tx_byte(POSITION_REQUEST_RECIVED);

    return;
    break;
  case COMMAND:

    switch(msg){
```

```
    case NO_OPERATION:
      break;
    case GO_TO_COMMAND1:
      build_new_data.count = FIRST_BYTE;
      build_new_data.actuator = ACTU_1;
      break;

    case DO_NOT_GO_TO_1:
      flags &= ~go_to1_mask;
      motor1flag =0;
      spi_tx_byte(DO_NOT_GO_TO_RECIVED);
      //move1_stop();
      break;
    case SEND_ACTU1_POSTION:
      posistion_dispatch(0);
      break;
    case SEND_BUTTON_STATES:
      spi_tx_byte(SEND_BUTTON_STATES | flags);
      break;
      //    case USER_INPUT_SWITCH:
      //     button_on = !button_on;
      //     break;
     case RESET_COMMAND:
      setup_spi();
      // button_on =1;
      flags = 0;
      break;
    }
  }
  return;
}

void flash_led(){
  static uint16_t counter;
  static uint8_t led_on;
  static uint8_t flash_on;

  counter += 4;
  if( counter < (counter -4)) {
    if(led_on){
      led_on = 0;
      GREEN_LED_ON;
    }
    else{
      led_on = 1;
      GREEN_LED_OFF;
    }
  }
}

void move1_up(){
  MOTOR1_AHIGH;
  MOTOR1_BLOW;
}

void move1_down(){
  MOTOR1_ALOW;
  MOTOR1_BHIGH;
}

void move1_stop(){
  MOTOR1_ALOW;
  MOTOR1_BLOW;
 }

/*
int pwm(){
  static uint8_t i;
  static uint8_t count;
  static int error;
  static int old_error1;
  //  static int old_error2;

  if (count){
    count--;
    if (i > count)
      return 1;
    else
      return 0;
  }else{
    count = PWM_RES;
    //   old_error2 = old_error1;
    old_error1 = error;
    error =abs(disired_posistion-shaft_posistion);
    if (error > 0){
      if (old_error1 > error){
      if(i) i--;
      //    else i=0;
      }else{
      if (i != PWM_RES) i++;
      }
    }else i = 0;

    //  i = (i + error);
    //i = i >> PWM_SCALE_FACTOR;
    //return 1;
  }
}
*/

void pwm(){
  static uint8_t i;

  if (i >= PWM_DUTY){
    motor1flag = 0;
  }
  if( i >= PWM_COUNT){
    i = 0;
  }
  i++;
}
```

```
void posistion_control(){
  if (flags & go_to1_mask){
    GREEN_LED_ON;
    if (disired_posistion > shaft_posistion){
      if(shaft_posistion <= MAX_HIGHT){
      if(pwm_flag){
      motor1flag = MOTOR_UP_MASK;
      }else motor1flag=0;
     }
    }
    if (disired_posistion < shaft_posistion){
     if(pwm_flag){
      if(END_STOP){
      motor1flag = MOTOR_DOWN_MASK;
      }      else{
      shaft_posistion = 0;
      }
       }else motor1flag = 0;
    }
    if(disired_posistion == shaft_posistion){
      motor1flag = 0;
    }
  }
}
```

functions .h definitions file

This file contains definitions for structures used in functions .c, preassembled definitions, and some deprecated code for PID controllers.

```
#ifndef FUCTIONS_H
#define FUCTIONS_H

#include <inttypes.h>

#define TRUE  1
#define FALSE 0


#define COMMAND       0
#define FIRST_BYTE    1
#define SECOND_BYTE   2

#define ACTU_1        0
#define ACTU_2        1

typedef struct {
  unsigned char count; //: 2;
  unsigned char actuator; //: 1;
}msg_flags;

/* typedef struct { */
/*   unsigned char go_to : 1; */
/*   unsigned char button_press : 1; */
/*   //  unsigned char shaft_change : 1; */
/* }global_flags; */


#define go_to1_mask 0x01

#define button1_mask 0x04

#define GO_TO1 1

#define STOP 3

#define PWM_DUTY  2
#define PWM_COUNT 5
void pwm();


#define MOTOR_UP_MASK 0x02
#define MOTOR_DOWN_MASK 0x01

#define MAX_HIGHT  ((uint8_t) 30) // max number of revolution the
actuator can make moveing up
//#define PWM_RES    ((uint8_t) 15)  // number of pwm values between 0
and 100 %
//#define PWM_SCALE_FACTOR 2        // how much difference between
shaft possition and disired shaft posistion is scaled by

//#define Kp //proportional gain for posisiton control PID sequence
//#define Ki //intrigal gain for Posistion control PID sequence
//#define Kd //dirivitive gain for Posistion control PID sequence

void message_handler(uint8_t msg);
inline void flash_led();
inline void move1_up();
inline void move1_down();
inline void move1_stop();

extern uint16_t old_shaft_posistion;
extern uint16_t disired_posistion;
extern uint16_t shaft_posistion;
//extern uint8_t pwm_flag;

#endif  /*fuctions*/
```

SPI Communications

This code keeps track of inputs received from the control computer and bytes to be sent to the control computer. It provided buffer so the computer and all the microprocessors can operate asynchronously, without stalling while waiting for each other.

The SPI code sets of the microcontroller as a slave device and enables communications with a master SPI unit. This code acts as a abstraction layer between the communications level and normal operation of the microcontroller. To transmit and receiver a byte from a master the user only has to call functions from this file and is not have to worry about the timing of the protocol being used. This file sets the size of the transmit and receive buffers, the functions to load bytes onto it off of these buffers, contains the routine for enabling the SPI hardware, and LATCH routine for receiving in loading data directly onto the shift register hardware.

```
#include <avr/io.h>
#include "spi.h"

#include <avr/signal.h>

// Implementation constants
#define SPI_RX_BUF_SIZE 8
#define SPI_TX_BUF_SIZE 8

// Declare compact types
typedef struct {
  unsigned char   head;
  unsigned char   tail;
} buf_index;

// Indices and buffers
uint8_t rx_buf[ SPI_RX_BUF_SIZE] ;
uint8_t tx_buf[ SPI_TX_BUF_SIZE] ;
buf_index rx_i;
buf_index tx_i;
buf_status spi_s;

void setup_spi(){
  // Initialize the USI control register
  USICR = SPI_CONTROL_SETTINGS;

  // Initialize buffer indices
  rx_i.head = rx_i.tail = 0;
  tx_i.head = tx_i.tail = 0;

  // Initialize spi status flags
  spi_s.rx_overflow = 0;
  spi_s.tx_underflow = 0;
  spi_s.rx_empty = 1;
  spi_s.tx_full = 0;

  // Operation Complete!
}

void spi_tx_byte(uint8_t usi_send_data){
  // Wait for an open buffer slot
  if( spi_s.tx_full ) return;
  // Write data into the buffer
  tx_buf[ tx_i.tail] = usi_send_data;

  // Advance index
  tx_i.tail = ( tx_i.tail + 1 ) % SPI_TX_BUF_SIZE;

  // Detect a full buffer
  if( tx_i.tail == tx_i.head ) spi_s.tx_full = 1;

  // Operation Complete!
}

uint8_t spi_rx_byte(){
  // Declare local variables
  uint8_t res;

  // Wait for data to be received
  while( spi_s.rx_empty ) asm volatile("nop");

  // Read data from buffer
  res = rx_buf[ rx_i.head] ;

  // Advance index
  rx_i.head = ( rx_i.head + 1 ) % SPI_RX_BUF_SIZE;

  // Detect an empty buffer
  if( rx_i.head == rx_i.tail ) spi_s.rx_empty = 1;

  // Operation Complete!
  return res;
}

void spi_latch() {
  // Reset overflow/underflow flags
  spi_s.rx_overflow = 0;
  spi_s.tx_underflow = 0;

  // Detect rx overflow
  if( !spi_s.rx_empty && ( rx_i.head == rx_i.tail ) )
  spi_s.rx_overflow = 1;

  // Detect tx underflow
  if( !spi_s.tx_full && ( tx_i.head == tx_i.tail ) )
  spi_s.tx_underflow = 1;

  // Receive and transmit data
  rx_buf[ rx_i.tail] = USIDR;
  USIDR = tx_buf[ tx_i.head] ;
  tx_buf[ tx_i.head] = 0;

  // Advance indices
  rx_i.tail = ( rx_i.tail + 1 ) % SPI_RX_BUF_SIZE;
  tx_i.head = ( tx_i.head + 1 ) % SPI_TX_BUF_SIZE;

  // Advance opposing indices on buffer overflow/underflow
  if( spi_s.rx_overflow )
  rx_i.head = ( rx_i.head + 1 ) % SPI_RX_BUF_SIZE;
  if( spi_s.tx_underflow )
  tx_i.tail = ( tx_i.tail + 1 ) % SPI_TX_BUF_SIZE;

  // Clear transient flags
  spi_s.tx_full = 0;
  spi_s.rx_empty = 0;

  // Operation Complete!
}

// $Log: spi.c,v $
```

SPI .h definitions file

```
// $Id: spi.h,v 1.5 2003/07/30 23:01:52 kolock Exp $
// spi.h -- AVR SPI slave i/o interface

/* This implementation of 3-wire SPI utilizes the USI with a software
latch.
 */

#ifndef SPI_H
#define SPI_H
#include <avr/io.h>
#include <inttypes.h>

/* Active options:
 * USISIE = Start condition Interrupt Enable
 * USIOIE = Overflow condition Interrupt Enable
 * USIWM[1,0] = Wire mode (3-wire = 0,1)
 * USICS[1,0] = Clock Source (1,1 = negative edge external)
 * USICLK = Clock select (0 = external, 1 = software)
 * USITC = strobe for software clock
 */
#define SPI_CONTROL_SETTINGS 0x1c  //0b0001 1100 //( _BV(USIWM1) |
_BV(USICS1) | _BV(USICS0) )

/* Call the setup_spi() initialization routine before using any SPI
facilities. */
void setup_spi();

/* Primary interface routines */
void spi_tx_byte(uint8_t usi_send_data);  /* Send a byte to TX buffer
*/
uint8_t spi_rx_byte();                    /* Fetch a byte from RX
buffer */
void spi_latch();                   /* Recognize SPI latch */

/* Status flags */
typedef struct {
      unsigned char     rx_overflow : 1;
      unsigned char     tx_underflow : 1;
      unsigned char     rx_empty : 1;
      unsigned char     tx_full : 1;
} buf_status;
extern buf_status spi_s;

#endif  /* SPI */
```

SPI Commands Definitions

The following file defined byte codes for the SPI communications. This file issues that both the control computer and the microcontroller's. It is basically a dictionary of all lay words they can save each other, such as to heiht x, or am at hight y.

```
#ifndef SPI_COMMANDS_H
#define SPI_COMMANDS_H

/*command next 1 and 2 are sent to tell the AVR's that a command will
be the next thing on
the line.  The AVR will then latch the command and wait for the the

*/

#define NUMBER_OF_AVRS 250

#define SHAFT1  0
#define BYTE1   0
#define SHAFT2  1
#define BYTE2   1

// commands to AVRS
#define NO_OPERATION  0x00

#define GO_TO_COMMAND1 0x02  //40
#define GO_TO_COMMAND2 0x03  //c0
#define DO_NOT_GO_TO_1 0x0A  //50
#define DO_NOT_GO_TO_2 0x0B  //d0
#define RESET_COMMAND 0xAA   //55
#define SEND_ACTU1_POSTION 0x0C //30
#define SEND_ACTU2_POSTION 0x0D //b0
#define SEND_MOTOR_STATES  0x0E
#define SEND_BUTTON_STATES 0x10 //08
#define USER_INPUT_SWITCH    0x0F


//depricated
/*
#define MOTOR1_UP   0x04
#define MOTOR2_UP   0x05
#define MOTOR1_DOWN 0x06
#define MOTOR2_DOWN 0x07
#define MOTOR1_STOP 0x08
#define MOTOR2_STOP 0x09
*/
```

```
// messages to computers

#define BUTTON1AND2_PRESSED 0x13
#define BUTTON1_PRESSED 0x11
#define BUTTON2_PRESSED 0x12
#define SENDING_ACTU1_POSTION 0x1A
#define SENDING_ACTU2_POSTION 0x1B
#define END_STOP1      0x17
#define END_STOP2      0x16

#define POSITION_REQUEST_RECIVED 0x47
#define DO_NOT_GO_TO_RECIVED   0x4A


#endif /*SPI_CONMMANDS_H */
```

PID screw actuator control method

```
output[n] = KP * error[n] + KI * integ_error[n] + KD * deriv_error[n]
error[n] = desired[n] - actual[n]
integ_error[n] = integ_error[n-1] + error[n]
deriv_error[n] = error[n] - error[n-1]
```

KP is the proportional gain, KI is integral gain, and KD is the derivative gain. These variables haves to be sets depending on the torque to compensate for both mechanical friction and inductive friction, the actual velocity of the motor is used, approximated again by the first difference of actual positions. The output of the system is thus computed as:

```
output[n] = KP * error[n] + KI * integ_error[n] + KD * deriv_error[n] +
            KA * (actual[n] - actual[n-1])
```

Where KA is the friction compensation (anti-damping) gain.

MAKEFILE

Blows the makefile where the microcontroller uses defined in all the options for the compiler are set. This project was compiled from C code using the free AVR dude software, and a GCC compiler.

```
# Configure AVR part here
# #######################
PART=t26
MCU=-mmcu=attiny26

# Project main file
# #################
```

```
all: actu.hex

# Project installation
# ####################
install: actu.hex
      avrdude -i actu.hex -p $(PART) -C /usr/local/avr/etc/avrdude.conf
-c AVRx16 -e

# Project dependencies (NOTE: mem => hex conversion is automatic)
# ####################

actu.mem: actu.o  encoder.o spi.o fuctions.o
      avr-gcc $(MCU) -g -Os -mcall-prologues -o $@ $^
actu.o: actu.c  pins.h fuctions.h spi.h
encoder.o: encoder.c encoder.h pins.h
spi.o: spi.c spi.h pins.h
fuctions.o: fuctions.c fuctions.h pins.h

# Linked projects
# ###############
rs232.o: rs232.c rs232.h
rs232.c:
      ln -s ../rs-232/rs232.c rs232.c
rs232.h:
      ln -s ../rs-232/rs232.h rs232.h

# DO NOT EDIT BELOW THIS LINE
# ##########################
clean:
      rm -f *.hex *.mem *.o

term:
      avrdude -p $(PART) -C /usr/local/avr/etc/avrdude.conf -c AVRx16 -
t

%.o : %.c
      avr-gcc $(MCU) -g -Os -mcall-prologues -c $<

%.hex : %.mem
      avr-objcopy -O ihex $< $@
```