// IJ’T Engineering

1234 N Laboratory Way
EE Building
Seattle, WA 987654

Designing a Holonomic Test Platform

March 17, 2004

Ivar Thorson
Jeremy Wu
Tyrel Newton

Table of Contents

ABSTRACT .... *pg. 4*
INTRODUCTION .... *pg. 5*
1.1 What is holonomic? .... ***pg. 5***
1.2 Project Overview .... ***pg. 5***
LIST OF TABLES AND FIGURES .... *pg. 6*
MATEHMATICAL THEORY .... *pg. 7*
2.1 Math of Movement .... ***pg. 7***
2.2 Math of Feedback .... ***pg. 9***
2.3 Math of Paths .... ***pg. 10***
SPECIFICATIONS .... *pg. 11*
PHOTO GALLERY .... *pg. 13*
SYSTEM DESIGN .... *pg. 15*
4.1 System Overview .... ***pg. 15***
4.1.1 High Level Design .... pg. 15
4.1.2 Electrical Resources .... pg. 16
4.2 User Interface .... ***pg. 16***
4.2.1 Nintendo Controller .... pg. 16
4.2.2 PC Application and Serial Communication .... pg. 18
4.3 Path Engine .... ***pg. 20***
4.3.1 Necessary Approximations .... pg. 20
4.3.2 Paths Array .... pg. 21
4.3.3 Movement Calculations .... pg. 21
4.3.4 Filtering and Feed-Forward Control .... pg. 22
4.4 3-Phase AC Motor Control .... ***pg. 22***
4.4.1 Background Narrative .... pg. 22
4.4.2 Reverse Engineering .... pg. 23
4.4.3 VVVF Control .... pg. 24
4.4.4 PID Control .... pg. 25
4.4.5 Electrical Design .... pg. 27
4.4.6 Motor Control PIC Software .... pg. 28
4.4.7 FPGA .... pg. 28
4.4.8 Motor Amplification .... pg. 29
4.4.9 Optoisolators .... pg. 30
4.4.10 Motor Power Source .... pg. 30
TESTING AND RESULTS ANALYSIS .... *pg. 31*
5.1 System Architecture .... ***pg. 31***
5.2 User Interfaces .... ***pg. 31***
5.2.1 Remote Control .... pg. 31
5.2.2 Application to Robot Communication .... pg. 32
5.3 Path Engine .... ***pg. 32***
5.3.1 Simulation .... pg. 32
5.3.2 Tests with Servos .... pg. 34
5.4 Motor Control Module .... ***pg. 34***
5.4.1 FPGA .... pg. 34

5.4.2 Motor Control Software .... *pg. 35*
5.4.3 Motor Drivers .... *pg. 35*
5.4.4 Optoisolators .... *pg. 36*
STATUS .... ***pg. 37***
6.1 Motor Drivers .... ***pg. 37***
6.2 GUI Application and RS-232 Serial Communication .... ***pg. 37***
6.3 Path Engine .... ***pg. 37***
6.4 IR Controller .... ***pg. 38***
SUMMARY .... ***pg. 39***
APPENDIX A: PARTS COST .... ***pg. 40***
APPENDIX B: MOTOR DRIVER SCHEMATICS .... ***pg. 41***
APPENDIX C: SOURCE CODE .... ***pg. 43***

Abstract

In many industries, the ability of a robot or mechanical platform to perform precision movements in tight spaces is often a limiting factor in the efficiency and productivity of the system. In such environments, the best choice of platforms are the ones that have the ability to move omni-directionally. This project will explore the detailed design of an omni-directional robot that uses a holonomic drive system. A holonomic drive system has the advantage of being very simple mechanically, allowing for robust and efficient design. Though the controlling mathematics governing the movements of this type of drive system are rather complex, we will simplify it enough to make the implementation feasible on low-cost PIC microprocessors. An attempt will also be made to use and control 3-phase induction AC motors, common actuators in powerful industrial drive systems. The signals required for driving AC motors from a DC power supply are by no means simple. As such, an inexpensive FPGA is used to keep the system cost and complexity at reasonable levels. This paper also shows through simulation that the PIC microprocessors can calculate the variables required for performing holonomic movements, although the full hardware implementation is at this time incomplete. Our self-guided foray into AC motor control will prove to be too difficult an obstacle to surmount, and an attempt will be made to build a DC servo-based platform. While the control of servos is considerably less complex than the control of 3-phase AC induction motors, time constraints end up being the final factor in preventing us from testing our theory and design using real hardware in real time. At the time of this writing, we are considering pursuing our interest in a holonomic robot further and completing this project at a later date.

INTRODUCTION

1.1 What is Holonomic?

Holonomic literally means freedom of movement. More generally, a holonomic robot is a way to build an omni-directional robot (where omni-directional means that the robot has the freedom to move in any direction). Specifically, an omni-directional robot has three degrees of freedom and can make any type of movement in a 2-D plane. These three degrees of freedom are achieved by using special wheels and mounting them on the robot in a very particular manner. Each wheel must be able to be driven in the normal manner (like a car's wheels) as well as have the ability to slide laterally. Figure 1.1 shows an image of the wheels that we used in the construction of our robot. Notice the multiple rollers spread out around the wheel that allow it to slide laterally. Because these wheels can slide laterally, there is no constraint on their orientation with respect to the robot's body. Figure 1.2 shows the 120 degree orientation that we used for each wheel. Obviously, controlling the motion of this strange apparatus needs a special set of mathematical equations (refer to section 2). However, once the motion can be controlled, any type of movement in the 2-D plane can be made.

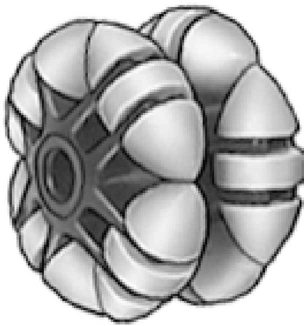


Figure 1.1. Bidirectional wheel.

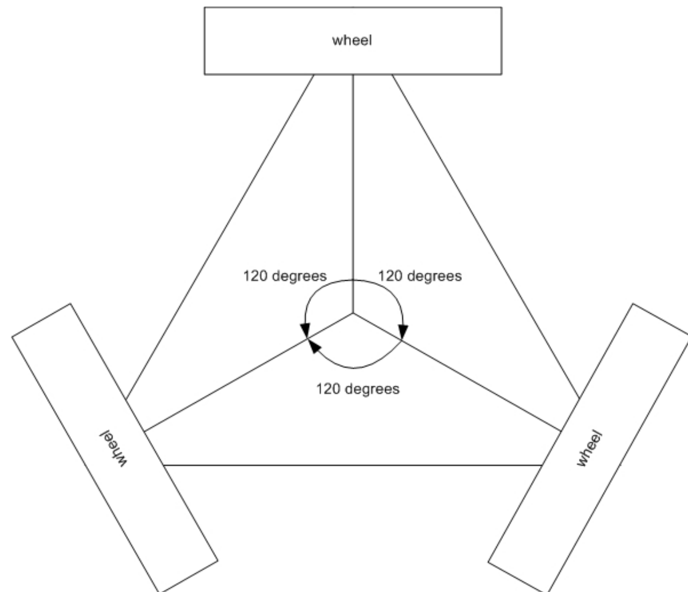


Figure 1.2. Wheel orientation for a Holonomic robot.

1.1 Project Overview

The question now becomes, what is a holonomic robot good for? Because of its freedom to make any movement in a 2-D plane, a holonomic robot has the potential to be useful in any industry where the ability to maneuver well is valuable. For example, forklifts operating in tight warehouses would greatly benefit from the ability to move sideways down a narrow isle. Therefore, this project aims to create a platform that demonstrates the practicality of using holonomic drive systems.

The goal of the project then becomes to build a mechanically simple robot that does a lot of complex trigonometry. Electronically, the robot will consist mainly of a few PIC microprocessors that perform a variety of tasks, ranging from controlling the motors to communicating with a simple driver application on a PC. Aside from downloading preprogrammed paths into the robot from the PC application, we would also like to control the robot using a basic Nintendo controller.

LIST OF TABLES AND FIGURES

Figures

- **Figure 1.1.** Bidirectional wheel.
- **Figure 1.2.** Wheel orientation for a holonomic robot.
- **Figure 2.1.** Equations for movement.
- **Figure 2.2.** Drive and slip vectors.
- **Figure 2.3.** Equations for feedback control.
- **Figure 2.4.** Interface for creating specific paths.
- **Figure 4.1.** High level block diagram.
- **Figure 4.2.** Basic SPI operation.
- **Figure 4.3.** Schematic for modulating signal for IR communication.
- **Figure 4.4.** IR Packet Protocol.
- **Figure 4.5.** Main window of PC application.
- **Figure 4.6.** Packet protocol for robot communication.
- **Figure 4.7.** PID control loop block diagram.
- **Figure 5.1.** Text file simulation produced by MS Visual Studio (linear accelerating path).

Tables

- **Table 4.1.** Motor encoder pin-outs.
- **Table 4.2.** FPGA Register Map.

Mathematical Theory

2.1 Math of Movement

To create holonomic movement, each motor of the robot must be driven independently, but in a coordinated manner. In other words, it is possible to decompose the desired motion of the robot into components for each motor. For the vast majority of the holonomic drive operation, all three wheels will be spinning in a concerted effort to move the robot from point A to point B in the manner prescribed. However, it was found that the calculation for each motor's desired movement could be calculated on an independent basis. The mathematical theory involved in calculating the desired motor spin for each control cycle is outlined in this section.

We began with devising a system of variables to describe the actual path of the robot itself. These variables are identified as the Variables of Motion and are composed of three unique quantities. The first is the Linear Heading which is the direction we would like the robot to move. It is defined as the polar angle between the front of the robot and the direction of movement. This is important because unlike most vehicles the holonomic drive can move in any direction, regardless of which direction it is actually facing. The front of the robot itself is defined as the line originating from the center of the robot and passing through the first motor. Each motor was given an arbitrary numerical id (1, 2, and 3) to assist in the calculations for each motor.

The second variable of motion is the Linear Velocity, represented in ticks per control cycle. Ticks is a unit derived from the motors itself. The smallest increment of the motor that can be measured by the quadrature encoders (refer to section 4.x). For the 3-phase AC motors this number was found to be 1/2000th of a revolution. The control cycle was defined as the period between which we refreshed the desired motor movement to the motor controller. We arbitrarily decided to run this cycle at 100 Hz because we believe that to be much faster than the motors can respond. Therefore, the Linear Velocity variable was essentially how far we wanted the robot to move per 10 ms.

The final variable of motion is the Rotational Velocity, measured in degrees per control cycle. For the sake of the integer calculations required by the path engine microprocessor we devised our own units for degrees. In our system, 90 degrees was represented by the integer value 16,000, 180 degrees by 32,000 and so on. This gave us the ability to divide by reasonably large numbers without losing too much accuracy. The degrees represent the change in degrees of the robot's heading per control cycle. Essentially, this variable represented the rate of spin of the robot as the holonomic robot has the ability to spin around the axis defined by it's center of mass.

$$m = v_L \cos(\theta_H - (\phi_{off} + 90°)) + v_r$$
$$v_L = linear_velocity$$
$$v_r = rotational_velocity$$
$$\theta_H = relative_heading$$
$$\phi_{off} = motor_offset$$

Figure 2.1. Equations for movement.

The three variables of motion were then used to calculate two separate terms of movement that would later be used to calculate the actual desired motor spin of each wheel. The first term is called the Linear Term.

This term is calculated by the equation shown in Figure 2.1. It is derived from two of the Variables of Motion: the Linear Heading, and the Linear Velocity. The basic principle used in calculating the motor spin is based on the idea that the actual movement of the wheel itself in reference to an absolute plane is composed of two orthogonal vectors (see Figure 2.2). The first is the distance traveled parallel to the orientation of the wheel, called the Driving Vector. This is the actual distance covered by the rotational force on the wheel by the motor which is then transferred to the ground. The second vector is composed of the distance traveled sideways by the wheel in what is called the Slip Vector. As discussed previously, holonomic robots have special wheels that can slide perpendicular to the driving force. This allows it to have the movement associated with a holonomic drive. The first of these vectors, the Driving Vector, is the actual velocity we wish to calculate as it is the only one we control from the individual wheels perspective. Using the equation shown we can see that it is essentially a basic trigonometric equation using a cosine of the offset Linear Heading. Each motor faces a different direction in reference to the front of the robot so each motor must be driven in a unique way, despite the fact that they are all referencing the same type of movement from a larger point of view.

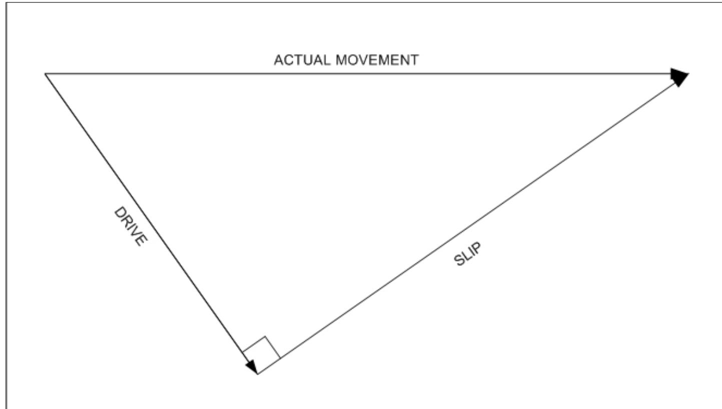


Figure 2.2. Drive and slip vectors.

The second term, used to calculate the actual motor spin required, is called the Rotational Term. This term is derived only from the Rotational Velocity variable. To understand how this term equates into motor spin of each wheel, we must first look at the geometry of the entire robot as a whole. If we take the radius from the center of the robot to any of it's wheels, we are able to construct a circle which will be traveled by all three wheels while the robot is spinning in place. The circumference of this circle is calculated in centimeters and then converted to motor ticks. Distance in centimeters can be directly converted into motor ticks because of the known circumference of the robot's wheels. The calculation returns a conversion rate of approximately 78 ticks per cm. From there, we must decide what percentage of the circle should be traveled per control cycle based on Rotational Velocity itself. A Rotational Velocity of 6 degrees per control cycle equates to an arc that covers 1/60th of the entire circumference or 1.6%. From this ratio we can calculate how many ticks of the motor are required to cover the desired spin of the robot.

The beauty of the final calculation comes from the assumption of superposition. If we can assume that the motion of the holonomic robot can be separated into the two terms, linear and rotational, we can then simply add the two components together to calculate the final motor ticks for each individual motor. This is how the equations seen in Figure 2.1 are created. These values are the actual numbers sent to each motor which describe that motor's movement for the next control cycle.

2.2 Math of Feedback

Using the quadrature encoders built into the 3-phase AC motors allowed us to create a feedback loop used to more accurately control the robot's movement. After every control cycle the motor control system was able to send back a report of how many ticks each motor actually moved and in what direction. Using this information we are able to reverse the equations of motion described in the section above to find the actual movement traveled by the robot. However, this was not a simple process. The calculation begins with the set of three equations seen at the top of Figure 2.3 in the previous section. Given m1, m2, and m3, we must now calculate the unknown variables Linear Velocity, Linear Heading, and Rotational Velocity. The only way to do this is to simultaneously solve the system of three equations. The result of this derivation is shown in Figure 2.3, with the eventual result being an iterative calculation of the Linear Heading. From the result of that calculation we are able to easily find the values for both Linear Velocity and Rotational Velocity. It should be noted that the process for iterative calculation requires the ability to divide integers as well as take the sine and arccosine with extreme accuracy. This was not an easy feat using non-floating point math and while limited by 16 bit variables. However, when simulating the calculations while replicating these limitations, we were able to accurately converge to the correct value about 75-80% of the time. Unfortunately, this is insufficient for a precision robot. However we believe that by using a larger processor with 32-bit floating point math, the accuracy desired could be achieved. There is a separate issue involving the calculation of an arccosine in iteration. For any value given to an arccosine (except -1 and 1), there are two valid angles that can be returned that fall between the range of 0 to 360 degrees. This effect was mitigated by a check against the original guess used in the iteration to see which half of the cosine wave we should be calculating in, however if the desired angle falls too close to 0 or 180 degrees this could become an issue. A full solution for this problem is not currently known.

$$m_1 = v_l \cos(\theta_H - 90°) + v_r$$
$$m_2 = v_l \cos(\theta_H - 210°) + v_r$$
$$m_3 = v_l \cos(\theta_H - 330°) + v_r$$

$$\theta_H = \cos^{-1}(\sin(\theta_H + 30°) * \frac{m3 - m1}{m1 - m2}) = \cos^{-1}(\cos(\theta_H + 60°) * \frac{m3 - m2}{m1 - m2}) = \cos^{-1}(\sin(\theta_H + 30°) * \frac{m3 - m2}{m3 - m1}) - 60°$$
$$v_l = \frac{m1 - m2}{\sqrt{3}\sin(\theta_H + 30°)} = \frac{m2 - m3}{-\sqrt{3}\cos(\theta_H)} = \frac{m1 - m3}{-\sqrt{3}\sin(\theta_H + 60°)}$$
$$v_r = m1 - v_l \cos(\theta_H - 90°)$$

Figure 2.3. Equations for feedback control.

Once we have calculated the actual motion defined by the actual variables of motion traveled, we can compute the error vector. The error vector is composed of the difference between the actual variables of motion and the desired ones initially sent to the motors at the beginning of the control cycle. This error vector is also defined by the three variables of motion. By subtracting the error vector off of the desired vector for the next control cycle, we are able to correct the path with a one control cycle delay.

2.3 Math of Paths

While the holonomic drive can be driven in any type of 2-dimensional movement imaginable, there are a few programmable paths that we decided to implement. The first was the simplest; a linear path. This was done by setting the linear heading to a single desired angle. For the Linear Velocity, the option to set an initial velocity, a final velocity, and a rate of acceleration were also given. As mentioned before, the principle of superposition was applied in reference to the rotational and linear terms so for any type of movement, rotational spin can be introduced by setting the rotational velocity to the desired rate. The second path that we created was a circular path. This was simply done by changing the Linear Heading by a constant rate. This rate is given by the user through the PC interface. Additionally, a rate of change in Linear Velocity can also be set so that the circle turns into a spiral of movement. Again, rotational spin may be introduced through the Rotational Velocity variable. Our final, and most interesting path of movement was the sinusoidal path. This path was created by varying the Linear Heading of the robot in a sinusoidal manner. The linear velocity was also adjusted sinusoidally because as can be seen when graphing on a calculator, the path of the robot should slow as it reaches the peaks of the sinusoidal wave. Additionally, an Absolute Linear Heading is available to set which direction the x axis of the sinusoid pattern should point in reference to the front of the robot. The period of the sine wave can be modified by setting the control cycles per period variable which represent the number of control cycles needed to complete one period of the sinusoid.

Our method of programming these paths into the robot was done through a Windows GUI (described in detail later) which can be seen in Figure 2.4.

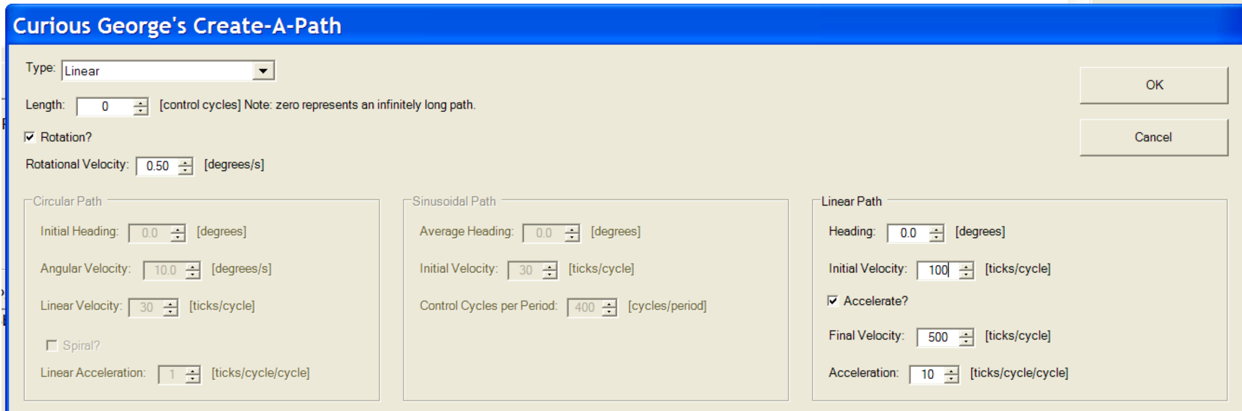


Figure 2.4. Interface used for creating specific paths.

SPECIFICATIONS

The goal of this project was to design a holonomic drive platform that could be built upon by later research projects. Additional features such as IR range finders, ultrasonic mapping, robotic appendages and a variety of sensors could be mounted upon the platform for a myriad of research and commercial uses. The beauty of the holonomic drive is its extreme agility. Unlike most wheeled vehicles its movement is not restricted by the direction in which the wheels are pointed, so the angular change in heading is only limited by the robots inertia and it's individual motor response. This application could be especially useful to wheelchair confined persons who may have trouble negotiating tight corners. Also, in research environments where a highly mobile robot is required to take the place of a human, a holonomic drive could be used as an ideal format.

Operation of the platform is performed in two distinct modes. The first mode is via a wireless remote control which gives the user real time manual control of the robot. The remote control is implemented over infrared serial communication and is integrated with an original Nintendo controller for easy input. The second method of operation is via RS-232 for the downloading of preset paths to the robot from a user PC. In this method, a type of path with specific path variables is entered into the robot, which is subsequently disconnected from the serial line. Upon manually or remotely (with IR controller) starting the robot, the robot should proceed to move along the programmed while calculating the necessary motor ticks at each step of the control cycle (see Theory section). The robot should be able to move a variety of paths including straight line, circle, spiral, and sinusoid. In addition, regardless of which path the robot is traveling in, it should be able to spin about its central point while following the desired path. Programming of paths is made easier by a user friendly GUI application (see application section). Packet protocol is used to communicate with the robot serially in both IR and RS-232 mode. This ensures that the robot receives the correct information at the appropriate time.

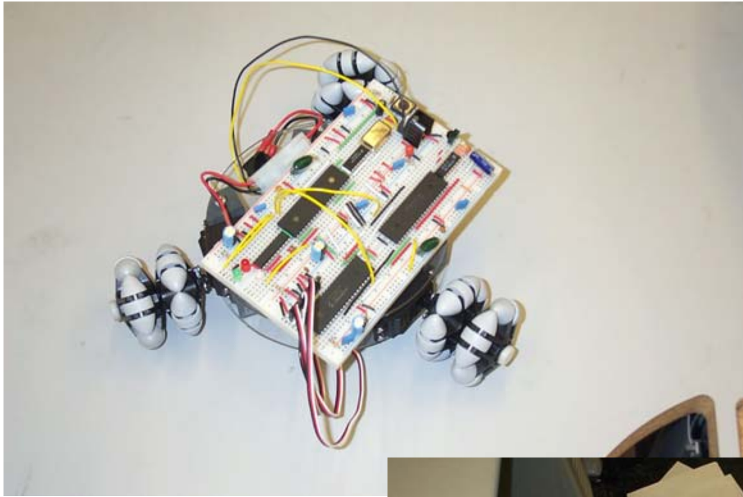
Propulsion of the robot is accomplished via three 24V 3-phase AC motors oriented 120 degrees apart. The motors drive omni directional skate wheels that allow for both parallel and lateral movement of the wheel itself. A picture of the wheel can be seen in Figure 1.1. Quadrature encoders mounted within the motors are used to create an error correcting control loop which creates accurate path movement for the robot. In order to create a responsive loop, we must run at a higher frequency of 1 KHz with respect to the 100 Hz path update cycle. This control loop also creates a low pass filtering effect which smoothes out and interpolates between the motor ticks from back to back control cycles.

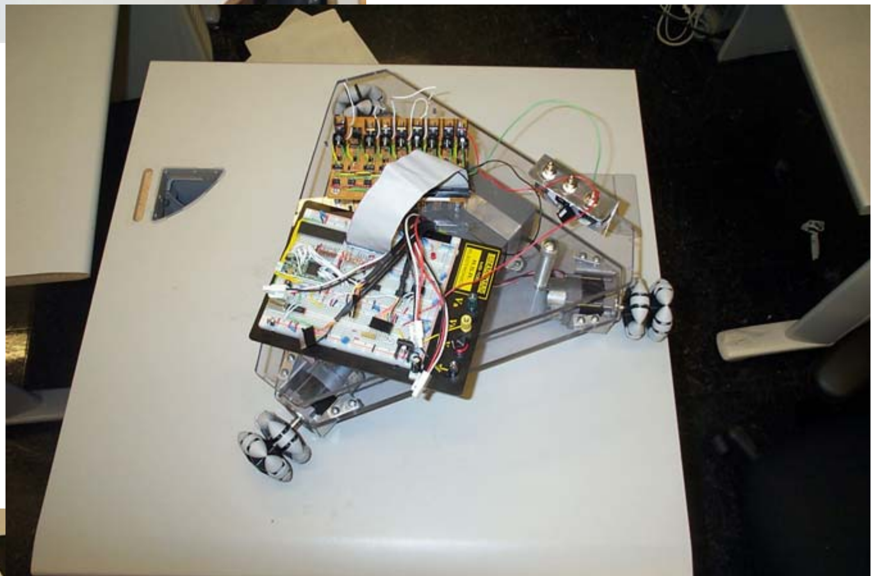
The motors themselves must be driven by individual motor drivers. They must be able to create a total of 9 phased AC sine waves to drive the motors (3 AC waves for each motor). It creates the AC wave by simulating with a pulse width modulated DC signal operating at 10 KHz (see motor control section).

Desired motor spin for each individual motor is calculated within an update path engine. This engine must calculate these values based on the variables of motion which describe overall robot movement. These variables are defined as the Linear Heading, the Linear Velocity, and the Rotational Velocity (see Theory Section). The engine should also calculate the error between

the traveled and desired paths and subsequently adjust the next desired path to correct for the error. In addition, the update path engine must filter the incoming desired path for large changes that cannot be realized by the motors. It will then filter these changes down to an acceptable rate. It must scan the inputs coming in via IR to determine if any corrections should be made to the motor movement (i.e. full emergency stop). During manual control mode, it must sample the remote control at a 10 Hz rate to give proper response of the robot. All these operations must be accomplished within a 50 Hz path update refresh rate.

PHOTO GALLERY





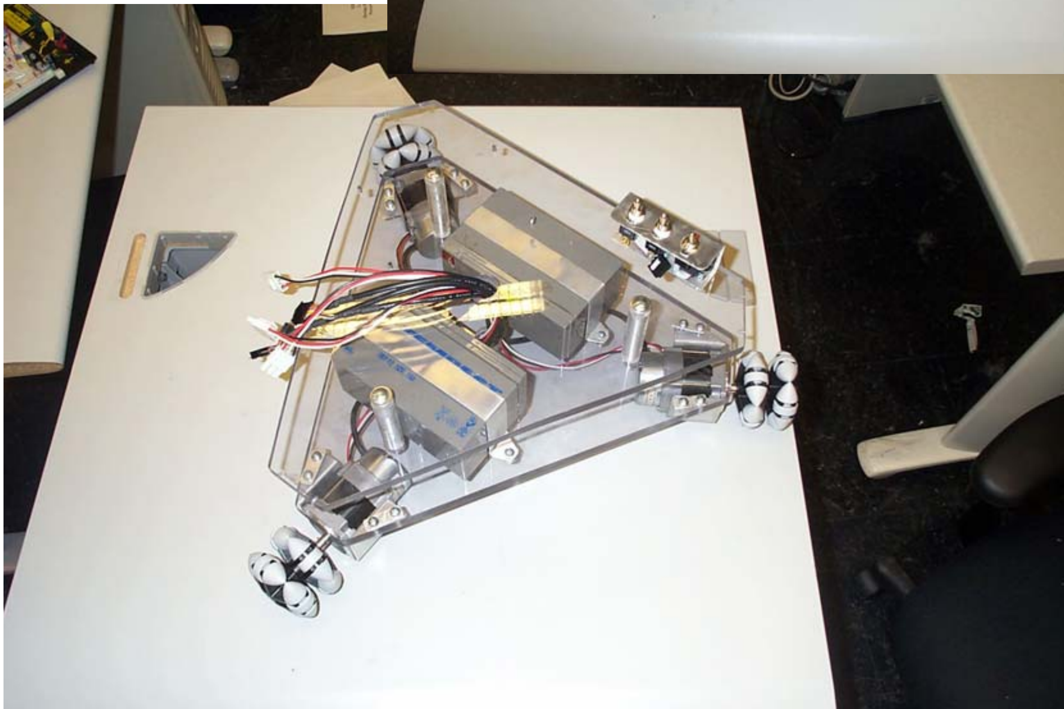
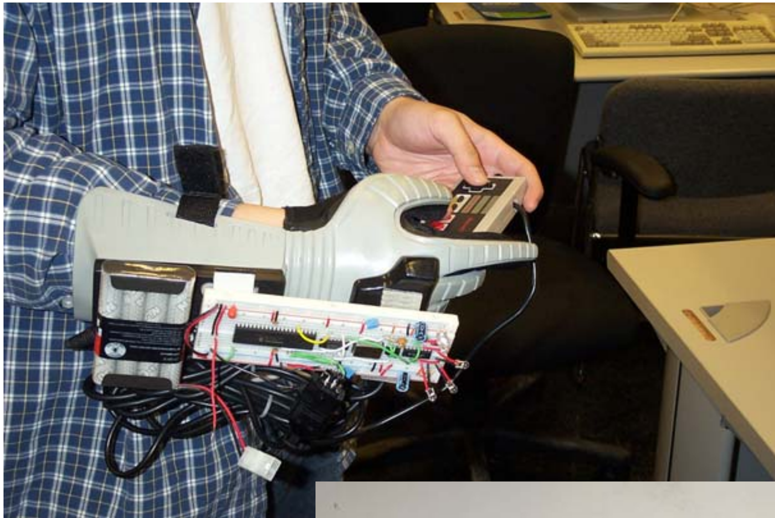
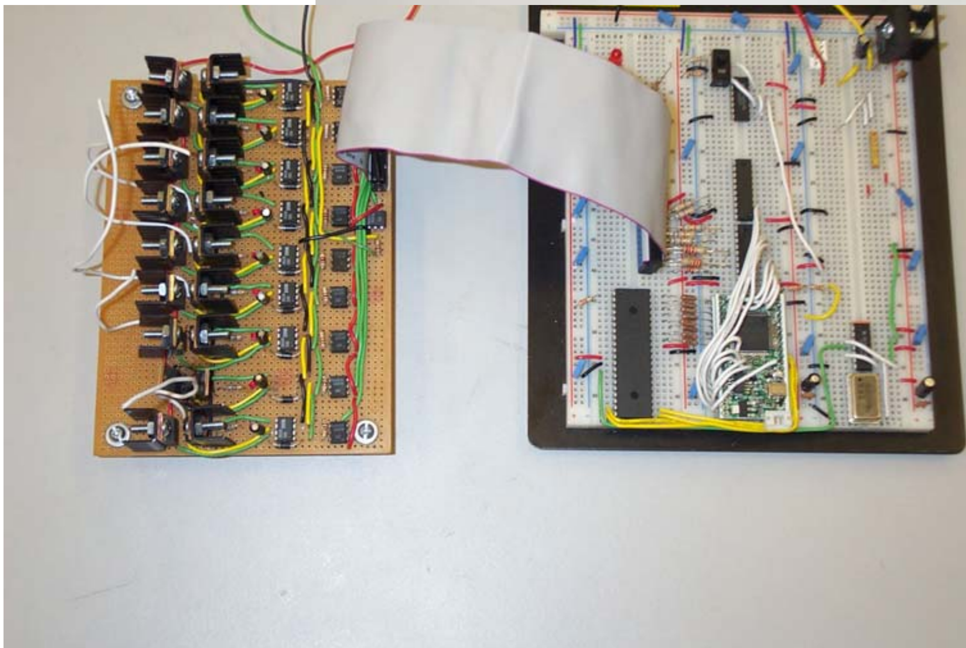


PHOTO GALLERY





4.1 System Overview

4.1.1 High Level Design

Figure 4.1 shows a block diagram for our robot at its highest level. At the core of the robot is a master control module that synchronizes all of the robot's actions. This module is responsible for communicating with the PC as well as calculating the required motor ticks while the robot is moving. Communication with the other modules in the system is done using a simple serial link, known as SPI (serial parallel interface). SPI consists of two shift registers connected in a ring that allow data to be exchanged between two devices in a synchronized manner. In general, and in our case, the loading and unloading of the shift registers seen in Figure 4.2 are done in software. In SPI, there is always a master device that controls the operation of both shift registers. We chose to use SPI both for its speed and because it was integrated into the PIC microcontrollers that we used almost exclusively throughout this project.

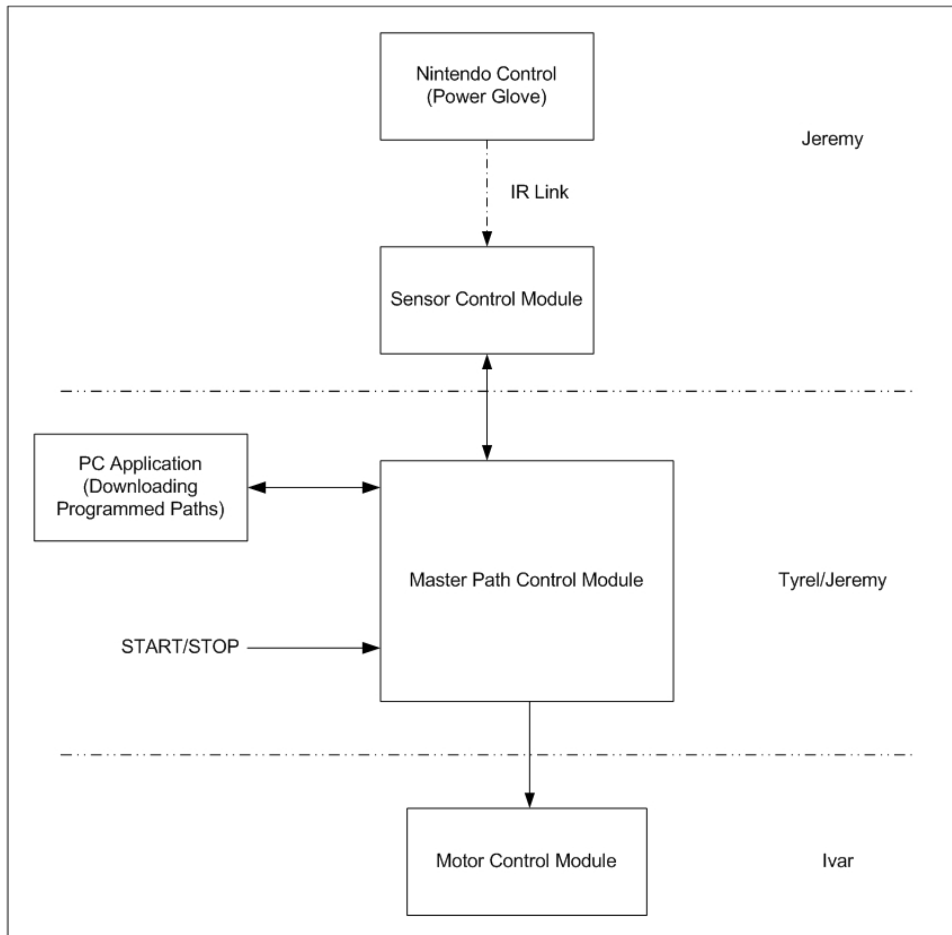


Figure 4.1. High level block diagram.

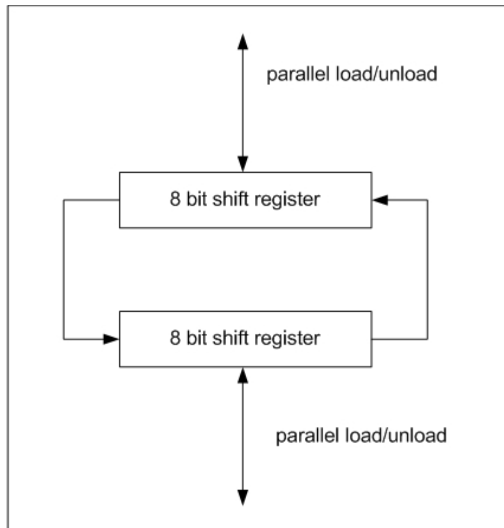


Figure 4.2. Basic SPI operation.

In our design, the path control module has two primary jobs, interact with the user either through the Nintendo controller or the PC application and to calculate the number of motor ticks that each motor should cover in the next control cycle. Therefore, there are two primary ways to control the robot, use the Nintendo controller or download a specific path from the PC application.

Connected to the master control module are the sensor control module and the motor control module. The primary job of the sensor control module is to communicate with a remote controller over a simple IR link. For remote control, we chose the ubiquitous Nintendo controller both for its simplicity and retro feel. The buttons pressed on the Nintendo controller are then translated into changes in movement inside the path control module.

The motor control module had the difficult job of controlling the speed of three 3-phase induction AC motors. At every step in the control cycle, the path control module would send a desired number of ticks over the SPI bus to the motor control module. The motor control module then did simple PID control to attempt to drive the motors at the desired number of ticks.

4.1.2 Electrical Resources

Other than PIC microprocessors, a motor driver board, and an FPGA, we used a limited amount of actual hardware. For all of the electronic components other than the FPGA, we used a 20 MHz oscillator to drive all of the hardware. The only exception to this 20 MHz is that the FPGA has an internal clock that ran at 25 MHz. The switch between clock domains was not a problem because the PICs were effectively running at 20 MHz/4 = 5 MHz. This makes the FPGA fast enough to catch anything that the PIC throws at it.

And because the only state machines that we used were PICs, we did not need a reset switch as the PICs have built in power-on reset logic.

For power, we used surplus 12V 7Ah lead-acid batteries connected in series to create a powerful +12V and +24V supply. As explained in the motor driver section, the +24V was used drive the motors while the +12V supply was regulated down to +5V to power the rest of the electronics. One important thing to note here is how we achieved RS-232 level conversion using only +5V. This is where the very handy MAX2xx series of level converters comes into play (we specifically used the MAX233CPP chip).

4.2 User Interface

4.2.1 Nintendo Controller

For our design, the user is able to control the robot using one of two methods. The first is by programming a predetermined path into the robot via the PC. The second is by controlling the robot using a real-time remote control protocol (schematic is seen in Figure 4.3). We decided to implement this second feature by reverse engineering two types of Nintendo controllers: the original NES controller, and the Mattel Power Glove. The information gathered from these devices was then relayed to the robot via Infrared communication.

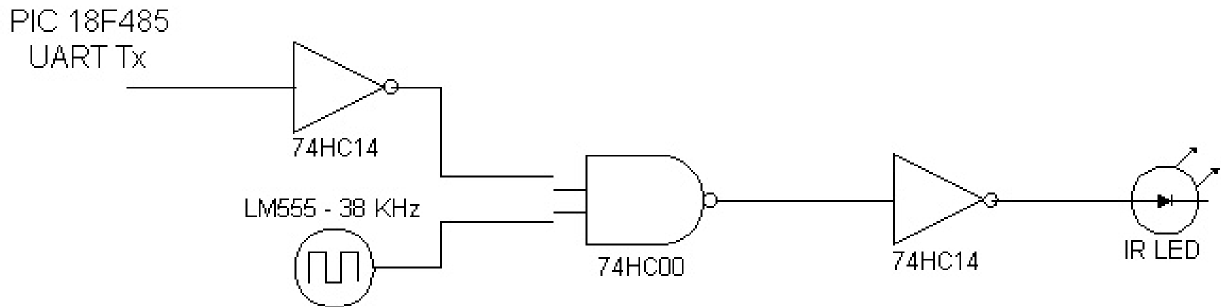


Figure 4.3. Schematic for modulating signal for IR communication.

Each button on the controller was used to control either a variable of motion or the starting/stopping function of the robot. A was used to increase the Rotational Velocity in the clockwise direction, B was used to increase it in the counterclockwise direction. UP was used to increase the robot's Linear Velocity, while DOWN was used to decrease it. LEFT indicated a counter clockwise change in Linear Heading, and RIGHT indicated a clockwise change.

The NES controller is a simple input device composed of 8 buttons and an internal shift register. In order to gather the information of which button is being pressed, a clock signal and reset latch signal must be provided. This process was done using a Microchip 18F485 PIC Microprocessor. We provided one reset latch pulse which shifted in the 8 bits of data (each bit corresponding to a single button) to the register, then clocked in seven pulses to serially shift out the data to the microprocessor via the Shift Out pin. Only seven clock pulses were required because as soon as the 8-bits are latched in, the first bit becomes available on the SO line.

Within the microprocessor we checked to see if the value of the buttons pressed had changed. If they had, we then placed the byte on the PIC's UART buffer to transmit out to the IR device. By only sending the signal when the buttons have changed, we were able to run the baud rate of the UART relatively slowly (1200 baud). This helped to create large clean square waves for the IR receiver on the robot to see. In order to send serial communications over IR, we had to first modulate the signal onto a 38 KHz carrier frequency. This 38 KHz frequency was created with an LM555 RC oscillator. To modulate the UART's signal onto the carrier frequency, we used a simple NAND gate with an inverter to produce the ANDing effect. To produce a powerful enough IR signal with sufficient range to create a continuous connection to the robot, we created an array of 5 LED's spaced 30 degrees apart.

In addition to the actual controller data byte, we created a small IR serial packet protocol to combat the inherent interference within the operating environment. The serial protocol can be viewed in Figure 4.4 below. It begins with a Start of Packet byte and is followed by a 2-byte

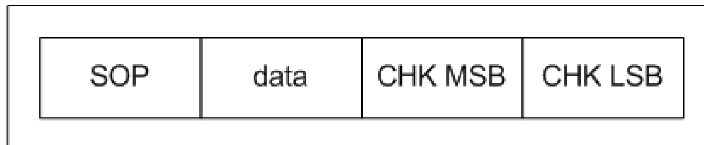


Figure 4.4. IR Packet Protocol

checksum that is calculated by adding the Nintendo data byte with the Start of Packet byte. On the other side of the signal is the receiving PIC which gathers the data on the UART then unpacks the packet and stores the new controller information.

One issue that we encountered was the loss of signal during transmission due to obstacle blocking. For instance, if the user were to hold down the UP-button and the signal to the robot was obstructed by an object (i.e. wall or another person), then when he releases the button that signal change would not be transmitted to the robot. Now, despite the fact that no buttons are being pushed on the controller, the robot would read a constant acceleration in Linear Velocity. This would result in the robot automatically increasing speed to it's maximum allowed velocity, despite the user's intent. In order to combat this effect, we implemented a watch-dog timer, essentially a recognizable byte that is sent out from the remote control every 500 ms. If the sensor receiver PIC misses this signal for two 500 ms periods in a row, it will automatically reset the stored Nintendo data byte to zero, or no buttons pressed.

The second phase of the remote control circuit was the use of the Mattel Power Glove. By reverse engineering and a fair amount of research, we were able to configure the Power Glove to simulate the NES controller functions but this time with the actual position of the hand itself. The Power Glove uses ultrasonic sensors to triangulate the position of the hand relative to a calibrated center point. The hand position then translates to the directional control on the original NES controller. It then uses bend sensors in the fingers to check if any of the four fingers (it does not monitor the pinky finger) are bent. The thumb bending represents the B button while the other three fingers represent the A button. In addition, the Power Glove can be initialized using a pattern of latching and clocking waveforms to transition into what is known as Hi-res (high resolution) format. In this format, the z position becomes available as well as the roll of the hand. Each finger is individually monitored to within 4 degrees of bend. All this is sent back over the Shift Out pin in a 12 byte packet with each byte representing one dimension of movement.

4.2.2 PC Application and Serial Communication

The primary job of the PC application was to communicate with the robot using a common RS-232 link (8 data bits, no parity, 1 stop bit, and 38400 baud to be exact). After plugging the robot into the serial link at the computer, the user can then use the application to interact with the robot. This interaction is limited to programming specific paths, such as circular and linear paths, into the robot. For every command the user gives the robot, the robot refreshes its cache of programmed paths with the application. In this way, the user can always know what paths he has programmed into the robot.

In terms of actually programming paths into the robot, each path is highly specific and needs to have some level of hard-coding done within the path control module. Therefore, we also needed to create a simple interface for creating these specific paths. Figure 4.4 shows the window that

we used for creating specific paths. The pull-down menu in the upper right allows the user to select between one of four paths: circular, sinusoidal, linear, and remote control. The remote control path is a special type of path that allows the robot to respond to controls from the Nintendo controller.

The PC application itself was built using Visual Studio .NET and C#. A quick search on the internet revealed a massive number of COM port drivers that help to insulate the application from the ugliness of the Win32 API. Figure 4.4 shows the main page of the PC application. Notice that there are many more windows than the few that would be required for simply programming paths into the robot. One of the primary difficulties of moving in a sinusoidal path is calculating the math for that path. Therefore, during testing, we uploaded the calculations that the path control module did to the PC application. The extra text boxes were used to display the calculations made during movement. During movement, the application also logged the path control module's calculations to a text file for later inspection.

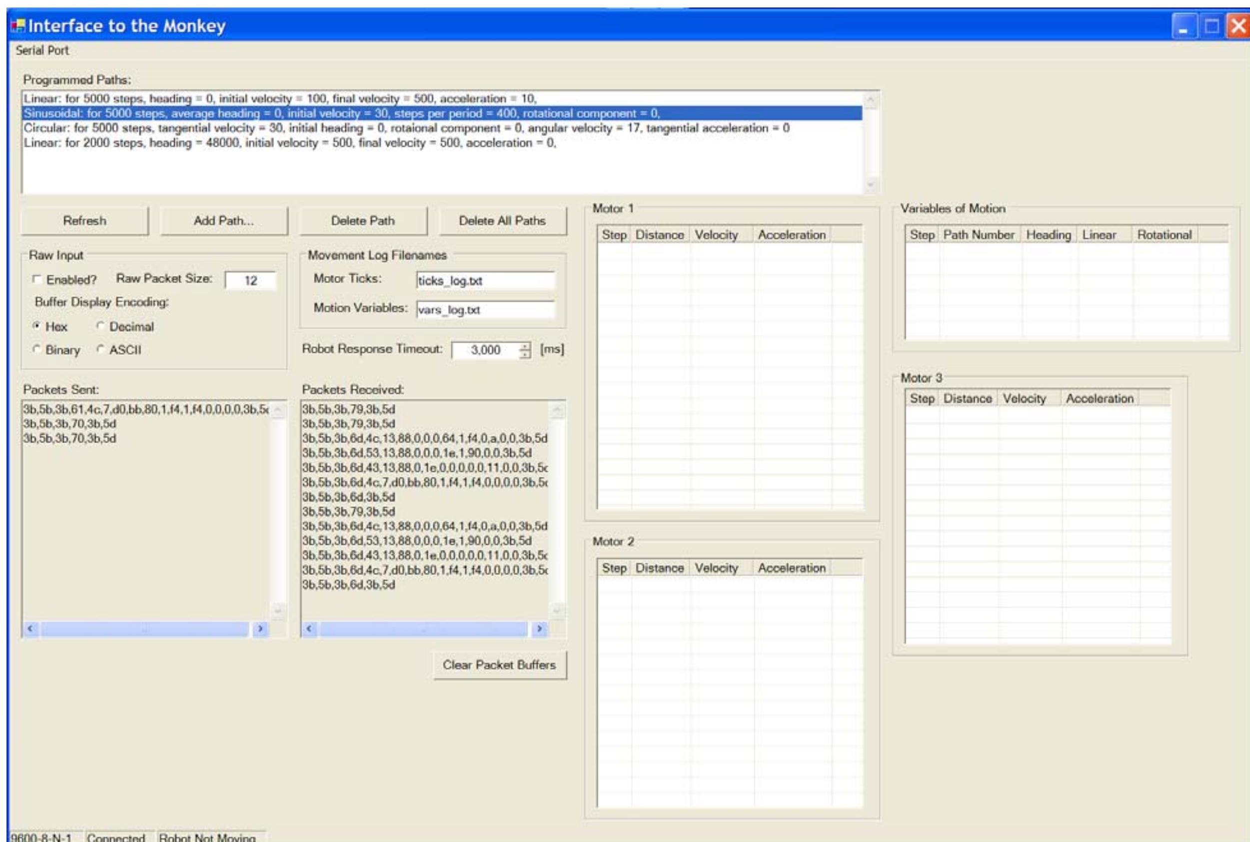


Figure 4.5. Main window of PC application.

Because the PC application and the robot are specific to each other, we did not attempt to build a very fancy serial packet protocol. To keep things simple, we only used escape characters, start-of-packets, end-of-packets, and a packet identification tag. Also, we left out any error checking because the link between the robot and the application is not important to the operation of the robot itself. Figure 4.5 shows the basic packet protocol that we used for communica-

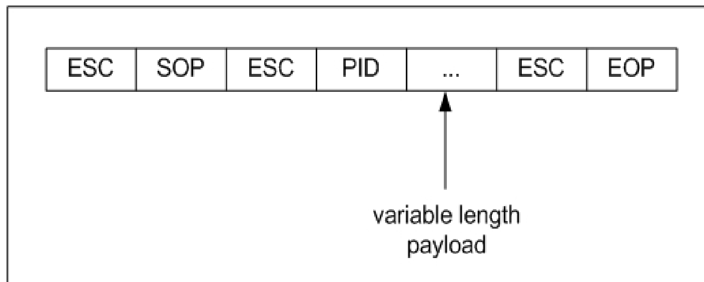


Figure 4.6. Packet protocol for robot communication.

tion between the robot and the PC application. In the figure ESC is an escape character, SOP is a start of packet, EOP is an end of packet, and PID is that packet identifier. In actuality, a packet protocol is not really necessary for this application. However, we found that would be useful to have a base for a packet protocol in case the system was ever extended.

In terms of the path control module on the robot, there are really only two exclusive modes. Either the robot is communicating with the PC application or it is moving along its programmed paths. Because the robot's job during communication is only restricted to communication, a simple polling algorithm while waiting for packets was adequate. In other words, the robot did not have any real-time constraints while communicating with the PC application. Therefore, the robot, and the application for that matter, can simply wait for an ESC followed by an EOP. There is the potential for getting a false ESC and EOP if they happened to appear in the payload. However, the communication that the robot was doing with the PC was not critical enough to warrant checking for that very small possibility.

4.3 Path Engine

As the computationally intensive part of the robot, the path engine has the job of calculating which direction the robot should move next. As explained in section 2, mathematical theory, the math is by no means simple. To make things even worse, we were using fairly limited 8 bit microprocessors (PIC18F452 to be exact). At every control cycle, which operated at about 100 Hz, the path engine had to recalculate the movement of the robot. Fortunately, this speed turned out to be slow enough to allow all of the calculations to be performed while still meeting the timing deadlines.

4.3.1 Necessary Approximations

Because we were using limited microprocessors to calculate complex trigonometric functions, we were forced to make many simplifications so that the control cycle could be run in a reasonable amount of time. The primary simplification that we made was using only integer math. Aside from being fast, integer math was also great for interacting with the motors because the motor controller computed everything in discrete motor ticks. Unfortunately though, we also had to multiply many numbers by cosines, which only have values less than one. To do these kinds of computations, we shifted the decimal point two places and truncated the remaining. As an example, the cosine of 45 degrees, which is actually 0.707106781, became 71. In our system then, 1 is represented as 100. One downside to doing only integer math is that division becomes very prone to errors. Although we could not make up for these errors, we tried to always multiply before dividing. This effectively made the numerator larger while keeping the denominator the same.

Another limitation imposed by the microprocessors was the bit length of calculations that we were able to do. Although it is possible to emulate 32 bit integer precision math in the PICs, it is not feasible when you have to do the calculations quickly. Therefore, we limited our integer calculations to 16 bits. The only problem this really imposed was overflow when multiplying large numbers. However, we were able to define our base unit of calculation, the motor tick, which allowed us to foresee, and thus compensate, for any overflow errors that we might have ran into.

The other major simplification, or rather optimization that we made was to use a cosine lookup table. In the microprocessor, we stored 8 bit numbers from 0 to 100 representing the cosine of 0 to 90 degrees. This offset of the decimal point also corresponded nicely to our offset used in integer approximations. Although the resolution of our cosine table was small, we found that under simulation, it effectively modeled a cosine function.

4.3.2 Paths Array

One of the goals that we wanted for the robot was to move in a sequence of predefined paths. Although we limited these paths to circular, linear, and sinusoidal types, combined with each other, they are enough to make a wide range of movements. In the actual path engine, a path is defined as a struct with an array of signed control variables and unsigned control variables. These control variables are used to update the variables of motion (heading, linear velocity, and rotational velocity) and are heavily dependent on the path being traveled. As an example, one of the control variables for circular movement is the angular velocity at which the robot moves. Some of these variables are constant while others depend on the number of control cycles traveled, such as the number of control cycles in the period of sinusoidal path.

Each path struct also holds a number of control cycles to travel which represents how long the robot will move along this path. As an example, consider a sinusoidal path with 100 control cycles per period. If the step length for this path was 10,000 control cycles, the robot would travel 10 periods before moving to the next path. When a path has completed the prescribed number of control cycles, the current index into the paths array is incremented to point at the next path. If there are no more paths, then the robot stops and returns to waiting for a restart command. We also created the ability for a path to be infinitely long by nominally assigning a zero step length as an infinite path.

4.3.3 Movement Calculations

During each control cycle, the job of the path engine (found in the path control module) is to recompute the desired number of ticks that each motor should perform in the next control cycle. Refer to section 2 for how these ticks are actually calculated. The number of calculations that must be done is rather long and best described in a list, as follows.

1. get button state from remote control and check for stop command, proceed only if stop command was not issued
2. store button state in path control variables array (if current path is a remote control path)
3. store previous distance motor ticks for low-pass filtering purposes

4. step motion variables for distance calculation
5. calculate motor ticks for next distance
6. increment path step without stopping robot
7. store motion variables, step length, and current path
8. step motion variables for velocity calculation
9. calculate motor ticks for next next distance ticks (velocity will be the first difference of motor ticks)
10. increment path step without stopping robot
11. step motion variables for acceleration calculation
12. calculate motor ticks for next next next distance ticks (acceleration will be second difference of motor ticks)
13. increment path step without stopping robot
14. restore path step, motion variables, and current path from first path step (only want one path step per control cycle)
15. filter each motor ticks against previous motor ticks
16. calculate distance, velocity, and acceleration on low-pass filtered motor ticks
17. check if path step made current path an empty path, which needs to then stop the robot

The basics of the loop is to advance the variables of motion followed by computing the motor ticks. Because we attempted to implement a feed forward control loop, the variables of motion had to be incremented three times (position, velocity, acceleration). However, we only really wanted to increment the variables once per control loop. So after the first increment and ticks calculations, we had to remember what the variables of motion were.

4.3.4 Filtering and Feed-forward Control

Because the motor control module had very tight timing deadlines, we offloaded a few of the calculations to the path control module. One such calculation was low-pass filtering the motor ticks so that we would not saturate the motors. Another calculation was to produce the velocity and acceleration terms for feed-forward control. These two numbers are most easily calculated along with the motor ticks since they are directly dependent on only the motor ticks. These two concepts are discussed further in the motor control section.

4.4. 3-Phase AC Motor Control

4.4.1 Background Narrative

To fully understand why the motor drivers turned out to be the limiting factor in the project, we must first have a short narrative describing the events that unfolded.

About four years ago, Ivar purchased three mystery motors from a surplus mail-order electronics catalog for $12.00 each, thinking that he had bought 12V DC brushless motors with encoders.

The mystery motors came with no documentation, datasheets, or even pin-outs. Physically, the motors are large, have three 18AWG wires coming out of the motor part, and nine thin wires

coming from the encoder. Some reverse engineering would obviously be required, but it seemed like a bargain to get powerful motors with accurate encoders for only 12 bucks each, and so we figured we would use them on this project because our robot requires great precision for its movements.

Initial tests of powering the motors revealed that the motors were NOT in fact 12VDC motors, as they locked in place when DC current was applied to 2 of the 3 wires. They were either AC motors or steppers, and a little searching online for the part number on the side of the encoder revealed that they are 75V AC servo motors. We confirmed that they ran on $\pm$12VAC using a transformer, so they were very likely not steppers.

Surprised and disappointed by this development, and worried about our ability to drive (what we thought were) single-phase AC motors, we purchased several hobby RC servo motors, thinking we would hack the servos for continuous rotation and add encoders to form a feedback loop. The scale of the robot would be decreased dramatically, but using hobby servos is an easy and known way of powering a robot for people with no mechanical engineering background.

In what would turn out to be one of the most significant decisions in the project, we decided that, what the hell, let's use the bigger motors and figure out how to drive them later. We were fast approaching our mechanical completion deadline, so the decision was made to build the robot using the large AC motors, and also a small robot using the DC servos, which we would use as a backup if we were unable to control the large robot.

With the mechanical construction complete, we set out in earnest to figure out how to drive the AC motors.

4.4.2 Reverse Engineering

As we were still unsure of exactly how to power the motors, we tested the resistances between each of the three heavy leads, and found about 1.2 Ω of resistance between each one. We found this strange, as we were expecting to have a power pin, a ground pin, and an earth pin of some sort, if it was a single phase induction motor.

Determining the pin-out of the quadrature encoder proved more informative. The nine wires coming from the encoder had 1 black wire, and 4 pairs of colored wires.

If you think like everyone else on this project, when you heard the words "lone black wire", you will immediately say "It's probably ground".

You would be wrong.

Pulling out our DMM, we measured the resistance and diode drops across all the 9 pins, looking for some sort of pattern that would reveal its internals.

There was a nice pattern during the diode drop testing between 5 of the pins and a single pin, which we correctly guessed were the CMOS output protection diodes and the positive supply.

Resistance testing confirmed this, and also hinted which pin was ground, as it had a different resistance to the positive supply than everything else.

Table 4.1. Motor encoder pin-outs.

C	Common
V	VCC
Z	Zero
A	Quadrature Encoder Channel A
B	Quadrature Encoder Channel B
P	Phase 1
Q	Phase 2
R	Phase 3

Since we could not afford to damage the encoders in this project (damaging one would require buying three new, very expensive motors), on this project, we carefully took apart the protective encoder housing to be sure that we had guessed right. Sure enough, there were silkscreen letters going to each of the 8 colored wires, as seen in Table 4.1.

The descriptions above came from looking at the output of each of the pins as we spun the motor by hand after applying power to C and V.

As near as we can tell, black is completely unconnected, and may just be shielding around the other 8 wires, so we grounded it just to be sure. Also, as it turns out, the pairs of colored wires were also meaningless and conveyed no useful information.

The two quadrature encoder channels match what we expected, providing two square waves 90 degrees out of phase with each other, with 500 lines per revolution. This gave us a decoded accuracy of 2000 lines per revolution plus the direction we are turning in.

The "Zero" pin would give a very short pulse out once per rotation, leading us to believe that it marks the zero degree point for the motor so that the phases can be calculated from this reference.

The three "phase" pins provided square waves were 120 degrees out of phase with each other as the motor was spun. We concluded that the motor is in fact a 3-phase AC induction motor, not single phase.

Also at this point, we found further documentation at the Japanese website for Shinano Kenshi, the motor manufacturer, and informs us that our motor was actually a 24V motor, and that the part number is the same as for the 75V motor. We were also able to get some useful specs on the motor, although still no pin-outs. Also confusingly, the motor is called both an AC servo motor and a DC servo motor.

We also learned that the motor is a 4-pole motor, so it requires two complete sine waves to complete one physical rotation.

4.4.3 VVVF Control

Freshly armed with a small amount of knowledge about the motors and a great deal of ignorance about how to power or control them, we bravely pushed on, trying to educate ourselves in the art of three-phase motor control.

Since we could neither find nor afford to purchase multiple commercial 3-phase motor DC-to-AC inverters, we were left with the option of trying to build one. Also, commercial inverters typically run off 120 V and are rated at hundreds of 100 W. In other words, there would be no way they would on any normal-sized robot.

The control of a DC motor is a topic commonly covered in introductory control theory classes, but AC motor control was not covered at all. The chapter on control of a 3-phase motors in our introductory control theory textbook was pitifully small. Furthermore, it only dealt with constant speed 3-phase motors and trying to keep the phase advance of the motors 90 degrees ahead of the rotor for maximum torque at a particular frequency.

A great deal of research later, we found that the name for the control scheme we were looking for was Variable Voltage Variable Frequency (VVVF) control, which is often implemented by modulating a low frequency sine wave onto a high-frequency PWM carrier to create the sine wave, and then multiplying the sine wave by a fraction to vary the effective amplitude of the voltage. If the PWM frequency is high enough, the motor only sees the sine wave and it appears as if it is receiving an AC wave of variable voltage and frequency. This allows for simple speed control of an AC induction motor.

Though the exact mathematics of the VVVF control was never revealed to us, we assumed that the phase advance of the signal should still always be 90 degrees ahead of rotor position for maximal torque, and that the effective voltage of the signal was the control variable used to adjust motor speed.

4.4.4 PID Control

Now that we had a control variable (variable voltage), the question became: how to decide what value to give it?

As described earlier, the two entities relevant to control of the motors are the main control PIC and the motor control PIC. The main processor sends control signals consisting of position information to the motor PIC, which is then expected to execute that command and move each motor by that distance before the time step is over.

Being a large focus of the only control theory class the author has ever taken, so we decided to use a traditional PID control scheme operating in the velocity domain.

We were also worried about excessively sharp commands from user or path control module, we decided to use a simple low-pass filter on the front of the control, to smooth out any abrupt commands that we receive.

Additionally, although it was never implemented, we included the concept of feed-forward into our PID control system, since during many movements the future behavior of the robot was predictable and using this knowledge could enhance the performance of the robot motor control system.

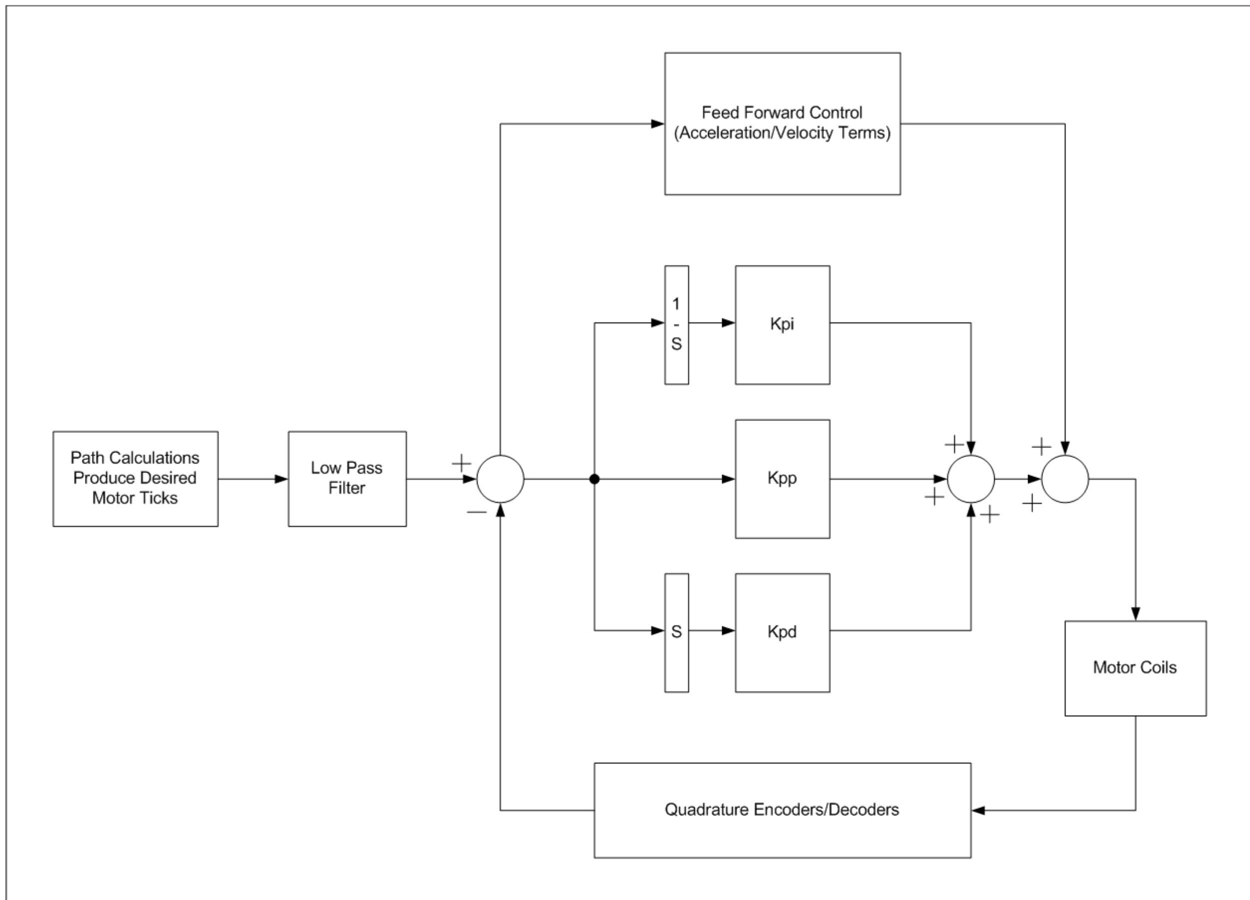


Figure 4.7. PID control loop block diagram.

A full block diagram of the PID control system is shown in Figure 4.x.

The integrator term incorporated anti-windup, which prevents the integrator from storing a value internally more than 120% of the output. This reduces ringing and oscillation dramatically in situations where the motor is saturated and unable to respond quickly enough. In such cases, the error integrator term typically balloons in size and the system will overshoot.

It should be noted that the filters, integrators, and differentiators are all shown in traditional z-domain notation, although for implementation on a computer they would naturally have to be converted into difference equations.

As a demonstration of this process, we will show the derivation of a simple first-order low pass filter.

The transfer function for a first order low pass filter in the s-domain looks like:

$$G(s) = \frac{\omega}{s + \omega}$$

where $\omega = 2\pi f$, the corner frequency of the filter in radians per second. In the discrete-time domain that is more appropriate for microcontrollers, this becomes:

$$G(z) = \frac{z(1-e^{-\omega T})}{z-e^{-\omega T}}$$

where T is the sampling period. To actually implement the filter, we need to convert the time domain. Remembering the transfer function is the output (written here as Y over the input X), we write:

$$G(z) = \frac{Y}{X}\frac{z(1-e^{-\omega T})}{z-e^{-\omega T}}$$

A little algebra can put this in terms of z-1, which in the discrete time domain corresponds to first differences:

$$Y(z-e^{-\omega T}) = Xz(1-e^{-\omega T})$$
$$Y(1-z^{-1}e^{-\omega T}) = X(1-e^{-\omega T})$$
$$Y = X(1-e^{-\omega T}) + Yz^{-1}e^{-\omega T}$$
$$Y[n] = X[n](1-e^{-\omega T}) + Y[n-1]e^{-\omega T}$$

which is our implementation equation for the time domain.

4.4.5 Electrical Design

With the theory out of the way, it's time for a description of the implementation. The first order of business is to figure out how in the world we would generate the 9 PWM signals (3 signals x 3 motors) which should all be modulated at 10 KHz or above. A commercial PWM DC-to-AC inverter from Motorola confirmed this ballpark figure as being close enough, as the Motorola inverter ran at 16 KHz.

Obviously, there are usually not 10 timers on a PIC microcontroller, even the complex ones, and emulating such fast counters in software was out of the question.

So, our basic strategy for generating these PWM signals is to have the microcontroller store a 1000 entry sine table in EEPROM, and index the table using the position feedback from the encoder plus whatever the control math requires.

The microcontroller would then load these values out into a bunch of loadable 8-bit down counters, which output a 1 while they are high, and 0 when they have expired. Assuming that the microcontroller refreshes all of the down counters at a rate around 10 KHz using a single interrupt, and that the clock speed of the down counters is around 2.5 MHz, then loading a value of 250 into the down counters would make the clock stay high for about 0.0001s. If you refresh at 10 KHz, then loading 250 repeatedly becomes 100% duty cycle, 125 would become 50% duty cycle, and 0 would become 0% duty cycle.

As should be obvious, implementing three 11-bit position counters, 9 self-resetting, loadable

PWM systems, three quadrature decoders, and synchronization circuitry in 7400 series logic, and making it fit on a small robot, is patently absurd. Besides, using only 7400 series logic is no longer a realistic solution to a problem in today's tightly integrated world of silicon.

Instead, we chose to move all of this logic to an FPGA, which would allow us a great deal of flexibility and speed. Specifically, we chose the Alterra ACEX 1K, a low-cost, low-cell count FPGA designed for cost-sensitive markets. Because the quantity of logic we have is rather minimal compared to most applications, we felt that with well-designed Verilog, we could easily fit all of our logic onto a single one of these low-cost FPGAs.

4.4.6 Motor Control PIC Software

The motor control software running on the PIC is extremely simple, due to the large amount of functionality which was placed on the FPGA hardware.

During normal operation, the internal timer 2 provides regular interrupts at a 1 KHz frequency, at which time the PWM values are refreshed according to the sine table, phase shift appropriate for that motor, and scaled according VVVF amplitude calculations gotten from the PID control. Periodically, at about 50-100 Hz, the PIC begins the handshaking process, requesting data from the master PIC and receiving it quickly over SPI.

During initialization and reset, the motors are enabled and there is a six-step motor drive function that is used to turn the motors once. This six-step drive function was used to combat the problem of initialization. For when the robot is started up, neither the exact position of the motors nor the required phase at which to drive each motor is known.

However, based on the three coarse phase position sensors, the initialization code can rotate the motors once, bringing the FPGA quadrature decoders and counters into a valid state after one complete rotation. At this point, the smoother sine wave control can be accurately calculated and used.

The three phase position sensors are called hall-effect sensors in the code; although they may or may not actually be hall-effect sensors they provide similar functionality.

4.4.7 FPGA

We purchased a cheap hobbyist kit with the SMT FPGA pre-soldered and tested. The board contains minimal logic for accepting downloads over serial, and also provides a 25 MHz clock for regular operation. As this clock was much faster than anything else in the circuit, synchronization DFFs were required whenever receiving from the PIC or the motor encoders to prevent any DFF meta-stability issues.

A very useful feature of coding on the FPGA is that it is possible to write detailed test cases to verify the operation of the circuit before trying it out for real. In fact, after writing detailed test cases for everything, the programmed FPGA worked reasonably well on the very first try, and after a few obvious and minor changes, worked flawlessly.

The Verilog implementation of the hardware was broken down into several modules:

1. Quadrature decoders, which take as inputs the unsynchronized A and B channels of the quadrature encoders, and output a synchronized 11 bit number corresponding to the rotational position (between 0 and 1999, the maximum resolvable number of counts per rotation).
2. 8-bit PWM modules that are synchronously loadable, and are clocked at 2.5 Mhz
3. Tri-State buffers with enables, used to select which of the quadrature encoder bytes appears on the data bus for use by the PIC.
4. An asynchronous 4-to-16 address decoder for selecting a PWM to write to or an encoder to read from.
5. A couple of counters to make the 2.5 MHz slow clock and a light that will blink at a few Hz so we can tell it is running.

The addressing and PWM loading method between the FPGA and PIC deserves some attention. Because the PIC only operates at 5 MHz (20 MHz clock divided by 4), and it has to perform sine lookups, scaling, and the control theory math, we could not afford to spend much time writing out the new values to the FPGA. Thus, all write lines and handshaking were removed from the circuit in the interests of speed.

Instead, what happens is the 4-to-16 address decoder selects a load signal on each of the nine PWM modules, and whenever the load signal is high the PWM will continually load in the value. This way, the PIC can just run through the addressing and put the values on the data bus, and the much faster FPGA will have time to load everything out without needing any write enables.

Also, the remaining address slots are used to select the 6 bytes required to read in the states of the three 11-bit counters indicating the positions of the motors.

The address map, along with who is driving the bus, is shown in Table 4.2.

Although there is a reset line that clears all the counters, initialization is not really an issue as the FPGA will automatically reset the motor position counters as the wheels spin.

The full Verilog source code is listed in Appendix C while a schematic is given in Appendix B.

Table 4.2. FPGA Register Map

Address	Object	Driver
0	PWM0	PIC
1	PWM1	PIC
2	PWM2	PIC
3	PWM3	PIC
4	PWM4	PIC
5	PWM5	PIC
6	PWM6	PIC
7	PWM7	PIC
8	PWM8	PIC
9	Enc0 (LSB)	FPGA
A	Enc0 (MSB)	FPGA
B	Enc1 (LSB)	FPGA
C	Enc1 (MSB)	FPGA
D	Enc2 (LSB)	FPGA
E	Enc2 (MSB)	FPGA
F	Unused	PIC

4.4.8 Motor Amplification

Many high-current motor drivers these days use IGBTs

(Insulated Gate Bipolar Transistors), which are basically just a CMOS transistor hooked up with a large BJT in a Darlington configuration. This provides the high-impedance benefit of a MOSFET with the current carrying capabilities of a BJT.

Our motor driver circuit is very similar to a common 3-phase motor driver circuit (see Appendix B), which uses 6 IGBTs in a H-bridge configuration. However, because of the large propagation delay when switching IGBTs, there must be delays inserted between the switching off of the pull-up transistor and the switching on of the pull-down transistor, to ensure that the circuit is break-before-make. Otherwise, there is a good chance that the circuit will short the positive rail to the negative rail for a short period of time during switching.

To attack this problem, we used IR2109 half-bridge drivers, which automatically ensure break-before-make switching and will be powerful enough to drive the IGBTs quickly, even as the IGBT gate capacitance changes because of the Miller effect.

The half-bridge drivers also incorporate a shutdown signal, which acts as a safety mechanism. If any of the PICs are not happy with their current status, or the user has pressed the stop button, the hardware will enable the motors and the machine will stay safely off. Safety is important in a machine that weighs 25 pounds and has powerful motors!

4.4.9 Optoisolators

To ensure that the high-current motor driver circuitry would not interfere with the FPGA, PICs, and communication systems, the motor drivers were electrically isolated using optoisolators. Specifically, we used inexpensive photo-Darlington optoisolators, which required 10mA of current on the transmission end (buffered signals from the FPGA) and a pull-up resistor on the receiving end.

4.4.10 Motor Power Source

As was mentioned earlier, we eventually discovered that the motors are in fact 24V motors. As a power supply, we used two 12V 7Ah sealed lead-acid batteries weighing at least 5 pounds each. Put in series to create a +12V and +24V power supply, these provided more than ample current to create big fat sparks and even start fires, as will be described in the testing section.

Internet research revealed that lead acid batteries are charged at a voltage of 13.8V, with two important rules of thumb:

1. Charge for 10-14 hours at 1/10th the rated Ah of the batteries
2. To keep the battery "topped up", you may supply a continuous current of 1/100th the Ah rating.

Due to the large capacity of the batteries and the relatively short amount of time we were able to test the motors, the batteries only required two charges throughout the project, which was accomplished using current limited lab power supplies.

5.1 System Architecture

The overall system architecture that we used on the robot was actually fairly simple, aside from the motor controller circuitry of course. Moving away from the motor control module, the reset of the system only consists of a few PICs communicating over SPI. Moreover, the actual exchange of information over the SPI bus was relatively small, the majority of the data relating to the actual motor ticks. As such, the robot itself was very modularized. While this made the merging of the modules very easy logically, it did not require us to do many tests on the actual communication between different devices. The few system wide tests that we were able to do did show that the SPI bus was working properly. In some ways, it's a moot point because the SPI protocol is so simple.

The one issue that we ran into with the SPI bus was too fast of a clock. We were forced to slow the clock down to allow RC transients to settle during each clock period. During initial tests we noticed that the system clock was capacitively coupling the data bits on the SPI bus. However, this coupling was small and easily fixed by running the SPI clock 16 times slower than the system clock. While this slowed down communication, it was not noticeable compared to the 100 Hz period that we were sending packets of data at.

5.2 User Interfaces

5.2.1 Remote Control

When testing the remote control, we initially created an isolated circuit composed of the sending and receiving IR modules. We were able to successfully clock in the correct data from the original NES controller and store it in the PIC. This was proven by a 10-segment LED display which we drove with the B-port on the PIC and subsequently the Nintendo data byte. It was then transmitted to the receiving module using the IR packet protocol and watchdog timer described in the previous remote control design section. We then tested the receiving PIC by again using the B-port to display the Nintendo byte on a 10-segment LED display. This operation was successful from up to 5 meters (15 feet) away and at a horizontal angle range of approximately 60 degrees in either direction away from the center line of the breadboard.

Unfortunately, upon moving the IR receiver circuit to the actual breadboard used in the final construction, the IR transmission began to fail. We were unable to repair the bug in the hardware before the demonstration. Given enough time, however, we are sure we would have been able to isolate the problem.

For the Power Glove, we were able to successfully reverse-engineer the Power Glove in the Lo-res mode (original NES controller emulation), however in our attempt to upgrade to Hi-res mode, We suspect that we over-voltaged the microprocessor inside the glove with a 6V power supply. This made the power glove essentially useless, but with a replacement glove we believe that we would be able to configure the Power Glove correctly and integrate it into our remote control subcircuit.

5.2.2 Application to Robot Communication

One of the major difficulties that we ran into in building the application was finding a suitable COM port driver for the .NET framework. There are numerous COM port drivers for C++ and MFC, but relatively few for .NET. Once we found a suitable driver package for .NET, the design process became a very small code, test, and debug loop. This is because there is a marked difference in designing a serial communication protocol in an object oriented language as opposed to a procedural language. It actually took a couple of attempts to get the flow of control through the different classes correct.

Once we were able to send and receive bytes from the PC, it became a matter of testing both the serial code on the PIC as well as the code on the PC. First cut tests were done using Hyperterm on another computer. However, it is very difficult to send bytes out of Hyperterm that are not common ASCII characters. This is where a custom serial application became a very useful debugger. It was a simple matter to display the received and transmitted bytes in a text box regardless of the byte's actual ASCII representation. This sped up the debugging process greatly and also gave us a small means of debugging the serial communication code in the PIC (since there is no real debugger for the PIC microprocessors).

One of the biggest tests we did with the serial communication was determining the maximum speed that we could run the baud rate at. This is because for useful debugging, the path control module needed to send each of the motor ticks and the variables of motion to the PC well within a 10 ms window (i.e. referring to the 100 Hz control cycle). When we were initially running the baud rate at 9600 baud, we found that the packets were getting corrupted en route to the PC. Only when we moved all the way up to 38400 baud did the path control module actually meeting its timing deadlines. We would have gone even faster with the baud rate, but the COM port driver that we were using would not allow a baud rate faster than 38400 baud.

After a thorough design, code, debug loop with the serial communication, we determined that the PC application and the path control module were communicating very well. In fact, the communication with the PC is one of the few aspects of this project that worked very well.

5.3 Path Engine

5.3.1 Simulation

Because of the inherent complexity of the movement calculations, we deemed it very important to thoroughly simulate the path control engine before we actually tested it with hardware. This was actually tricky because the compiler that we were using, PICC created by CCS, did not have the best debugger. Therefore, we wrote our path engine code for the PIC first, and then attempted to port it to MS Visual Studio for simulation. This is much easier than it sounds because of the simplicity of the PICC compiler and the complexity of C compiler in MS Visual Studio 6.0. The major trick for getting it to work was numerous compiler directives and specialty typedefs to get the data types the same. The biggest potential for flaws in the MS Visual Studio tests would be to not use the same data types (i.e. using 32 bit values for simulation instead of the 16 bit values that we were using in the PIC). This would have been a huge over-

sight because of our original concern for overflow during multiplication.

To simulate the path control engine, we emulated the action of the user by hard-coding specific paths into the engine. We then ran the engine while dumping lots of information to a text file for later processing. Although a bit complex, this process allowed us to catch many small math errors that we were making when calculating the motor ticks. Simulation also allowed us to find a few flaws in our original math equations for the movement.

The major drawback to simulating the path engine by dumping the calculations to a text file is that you can't actually see how the robot would move based on the numbers. Consider the complexity of the numbers when the robot is making a sinusoidal path while rotating. However, we did test linear paths and simple circular paths (i.e. no rotational component). These tests showed, after straining our eyes while examining the text files, that the math we were calculating was correct.

The easiest type of path to simulate by far was a linear path with no rotation. In this type of

```
variables = [phi_H,v_L,r_L]
motor x = array of ticks, current posi-
tion is index 1 (zero-based)
transmitted: position, velocity, accel-
eration for each motor

step: 2
---------------------------------------
variables: [0,30,0]
variables: [0,40,0]
variables: [0,50,0]
motor 1: [0,0,0,0,]
motor 2: [-17,-26,-34,-43,]
motor 3: [17,26,34,43,]
transmitted: [0,0,0,-26,-8,-1,26,8,1,]
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
step: 3
---------------------------------------
variables: [0,40,0]
variables: [0,50,0]
variables: [0,60,0]
motor 1: [0,0,0,0,]
motor 2: [-26,-34,-43,-52,]
motor 3: [26,34,43,52,]
transmitted: [0,0,0,-34,-9,0,34,9,0,]
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
step: 4
---------------------------------------
variables: [0,50,0]
variables: [0,60,0]
variables: [0,70,0]
motor 1: [0,0,0,0,]
motor 2: [-34,-43,-52,-60,]
motor 3: [34,43,52,60,]
transmitted: [0,0,0,-43,-9,1,43,9,-1,]
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
step: 5
---------------------------------------
variables: [0,60,0]
variables: [0,70,0]
variables: [0,80,0]
motor 1: [0,0,0,0,]
motor 2: [-43,-52,-60,-69,]
motor 3: [43,52,60,69,]
transmitted: [0,0,0,-52,-8,-1,52,8,1,]
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
step: 6
---------------------------------------
variables: [0,70,0]
variables: [0,80,0]
variables: [0,90,0]
motor 1: [0,0,0,0,]
motor 2: [-52,-60,-69,-78,]
motor 3: [52,60,69,78,]
transmitted: [0,0,0,-60,-9,0,60,9,0,]
```

Figure 5.1. Text file simulation produced by MS Visual Studio (linear accelerating path).

path, the motor ticks remain constant throughout the duration of the path. Our simulation of linear paths showed that our calculations were producing constant motor ticks. As an example of a simulation, Figure 5.1 shows a simulation of an accelerating linear path. While the motor ticks are not constant, you can see that they accelerate in accordance with the acceleration of the linear velocity term.

5.3.2 Tests with Servos

By far the biggest problem that we had with this project was successfully driving the AC motors. When we determined that we were not going to be able to drive the AC motors within the allotted timeframe, we decided to attempt to build a small test platform using DC servo motors. The idea was that although we were not going to be able to drive the AC motors, we would be able to test the theory that we developed for holonomic movement. Unfortunately, when we started the DC servo-based robot, we were already under severe time constraints.

Originally, we had planned to use the servo-based robot as a backup plan and therefore already constructed the hardware for the robot. The trickiest part about using servos is hacking them so that they will actually turn more than about 270 degrees. This hacking requires removing the internal stops and replacing the position sensing potentiometer with a normal resistor. This then forces the servo to keep spinning because it always thinks that it is returning to its neutral point (but never actually getting there). The ideal way to hack a servo is to replace the potentiometer with a precision resistor. This then keeps the neutral point of the servo constant. Unfortunately for us, we did not have any precision resistors and no time to buy them. The effect of this was that we could not accurately calibrate the servos, or in other words, find the neutral point and mark it down as a constant value. Once the neutral point is found, the servo is then driven by a small pulse of precise duration that happens approximately 20 ms apart. The width of this pulse is what defines the direction and speed in which the servo turns. The actual neutral point is a particular width of the pulse that causes the servo to not move.

Our biggest problem was that since we used imprecise resistors, the neutral point actually fluctuated a lot. Therefore, we were not able to accurately control the servos without a feedback loop. In fact, we were not even sure if we were finding the neutral point correctly. Ideally though, the servo-based robot would have been a nice easy test platform to prove the correctness of the path control math. Given a few more days, we would have been able to get the servos to work accurately allowing us to accurately simulate our path engine.

5.4 Motor Control Module

5.4.1 FPGA

Due to the fact that much of the Verilog code could be unit-tested before even programmed onto the FPGA at all, the FPGA actually worked on the first try, although there was an error with the addressing line pin-assignment which required a quick revision to fix.

Because we are currently using 91% of the FPGA, if more features are to be added on the FPGA on the few remaining pins, a good space-saving measure would be to redesign the PWM

modules to use only 10 counters (9 down counters plus 1 reset), rather than the current system with two counters per module.

By far the most difficult part of the FPGA design was handling the multiple clock frequencies. Most of the FPGA ran on a 25 MHz clock, the PWM devices ran on a 2.5 MHz clock, and the PIC runs at 5 MHz. The internal timing analysis tools provided by Alterra would block compilation of the system because the propagation delay would often be greater than the useable clock period. A considerable amount of time was spent playing with the timing analysis tools and learning about FPGA signal propagation issues.

The only unresolved issue with the FPGA at this time is providing it with an image on reset of the system, as the FPGA configuration is stored in volatile memory. At 20 Kbytes uncompressed, the FPGA image would barely fit in the program EEPROM of the motor control PIC, but we ran into a compiler bug with the PICC software which results in truncated tables although plenty of EEPROM still remains. We are still searching for a workaround to this problem.

5.4.2 Motor Control Software

The version of the motor control software provided in this software is incomplete, because much of the functionality was removed to isolate testing on the motor driver circuitry. However, it does refresh the FPGA motors at a constant frequencies, and if the math is correct presumably it would control the motors appropriately once the PID gains were tuned.

5.4.3 Motor Drivers

The lowly motor drivers caused us more testing grief than anything else in the entire project. Three driver boards were built in total, and each one would work perfectly up until a motor was connected, at which point it would get hot, smoke, short out, or even burst into flames.

The problem seems to be this: switching an inductive load like a motor at 10 KHz creates large inductive voltage spikes, which, despite fast acting diodes designed to counter this, would eventually propagate back to the half-bridge drivers (and even to the FPGA). After about 20 seconds, the half-bridge would fry, and sometimes when it fried it would turn on both the IGBT pull-up and pull-down at the same time.

The first time this happened, it smoked a couple of the IGBTs and all of the half-bridge drivers. The second time, one of the IGBTs burst into flames and glowed red-hot for quite a while, despite the heat sinks attached to every TO-220 package. This hilarious event prompted lab jokesters to place the fire extinguisher next to our lab bench whenever we were around.

Another simple problem was our simple ignorance of the fact that the collectors of each of the IGBTs are connected to the heat sink terminal, so the heat sinks must be electrically unconnected and not touch! Not knowing this, we initially made a heat sink that connected all the terminals to each other. During one test, they drew so much current that they heated and peeled back the copper traces on the PCB we had soldered them to! Later motor driver circuits were

done on wire wrapping breadboard, and soldered together with heavier wires to prevent such an event from occurring again.

A solution to this problem is to add more protection diodes, and probably an RC snubber circuit of some sort to damp out the high-voltage spikes from the inductance of the motor.

5.4.4 Optoisolators

The optoisolators were a Good Idea. They undoubtedly saved the life of the second FPGA.

The only downside was that there is a slight phase shift and distortion from the nonlinearity of the photo-Darlington that resulted in the PWM frequency usually being distorted by 1-5%. This was minimized by adjusting the values of the resistors driving the transmit diodes, and also the pull-ups on the photo-Darlington end.

More expensive optoisolators would provide more even timing, which would be useful if the PWM frequency increases beyond 10 KHz.

Status

At this time, the robot has not achieved all of the goals enumerated in the previous specifications section. Although most of the theory and software have been fully developed, hardware obstacles have prevented us from testing our design in a real world environment. Given more time and added expertise, especially in the motor driver circuit, we believe that we could complete the holonomic drive robot to specifications.

6.1 Motor Drivers

Currently, the motor drivers are capable of outputting nine appropriately phased AC waves via DC PWM modulation. Proof of this operation is confirmed via oscilloscope readings and by visual inspection of the LED lights attached to the output of the motor drivers. However, the drivers are not able to safely drive the AC motors directly. To proceed with the motor construction, we will need to consult with a power electronics expert. With their assistance we should be able to modify the motor drivers so that they can drive the AC motors in the correct manner.

6.2 GUI Application and RS-232 Serial Communication

The GUI application has been a very useful tool thus far and has performed well under simulation and real world trials. Additionally, it is able to communicate directly to the robot using the RS-232 serial packet protocol. Back and forth communication has been confirmed as well as the downloading of the desired paths to be traveled by the robot. In fact, the use of the GUI simulator has become integral in testing the correct operation of the update path engine since we have been unable to drive the motors.

6.3 Path Engine

Using the contingency DC Servo motors the start and stop operations have been verified as working properly. Initial SPI communication to the motors has also been confirmed. The ability to start the robot, run autonomously on the DC servo motors and stop on it's own has been demonstrated. However, the desired paths are not followed by the current model. This is due to the mis-calibration of the DC servo motors. Unfortunately, due to the last minute change in design (going from 24V AC motors to 5V DC motors), we were unable to find the correct neutral point of the motors. This lead to an inability to vary the speed or direction of the motors based on the given desired motor ticks. With very little extra time, we should be able to reconfigure the DC servos to work properly, however they would be less responsive and less accurate than our initial AC motor design.

On a positive note, calculations required by the update path engine are being made at a faster rate than expected, allowing for potential upgradeability of the path engine model. The path control loop has not been implemented because of a lack of processing time needed to implement a high frequency control loop. Nevertheless, software code for such a control loop has been written. If we were to implement the control loop we would need to migrate to a faster processor. The processor would need to have either a Floating Point Unit to accurately calculate iterations or at least fast 32-bit operations. An example of such a processor would be some-

thing found on a PC-104 form factor board.

6.4 IR Controller

The Nintendo controller IR remote control was successfully implemented on isolated bread-boards. Correct operation of the IR serial packet protocol with the watchdog timer was also confirmed. However, integration into the final robot layout was not successful at this time. Continued work on wireless control would migrate to an RF implementation controlled solely by the master PC. Both predetermined path mode and manual control mode would be operated by the user from the master PC wirelessly over RF to the robot.

Given time, most if not all of these issues could be remedied. Aside from the inability to integrate the motor drivers to the actual AC motors, most issues were created from an insufficient amount of time. Unfortunately, most of the extra time needed would have been used for testing and debugging. With proper time for testing we would have been able to correct some of the minor issues that lead to system wide failure. In the end, we are confident that with continued effort, the holonomic drive platform will be successfully implemented.

SUMMARY

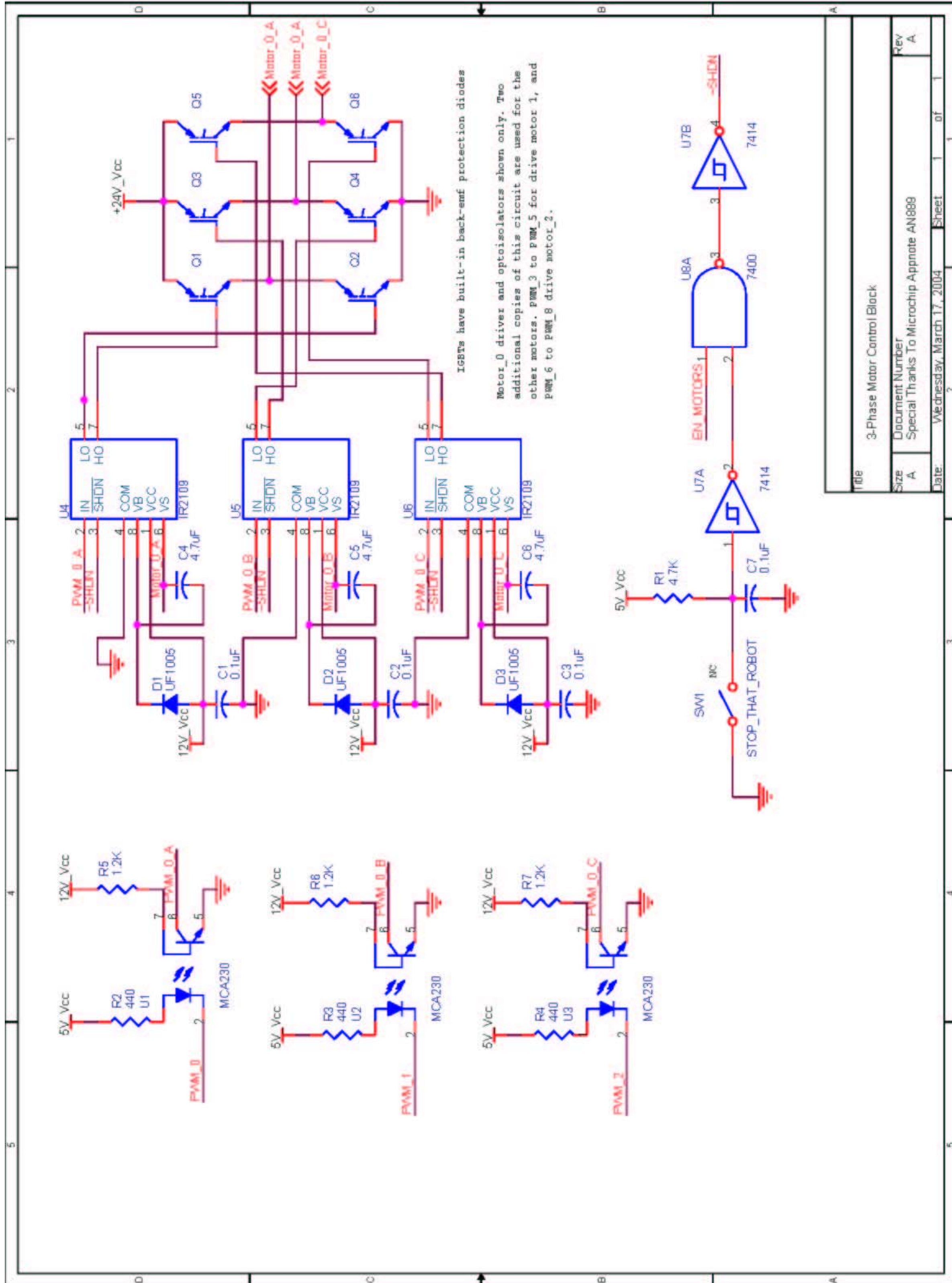
During the course of this project, we participated in a variety of complex, realistic engineering tasks that spanned the breadth of the field of electrical engineering. Although this project is technically classified as a digital systems capstone, the work on this project touched on computer science and software design for both desktop and embedded platforms, feedback control theory and robotic system design, FPGAs and digital signal processing, power electronics and amplification, communication protocols and implementation, reverse engineering, and electrical circuit design. Throughout this project, many difficult and unforseen problems arose, the majority of which were discovered, analyzed, and overcome. Despite not being fully completed, the level of complexity of this project, and the technical achievement that it represents, is evidence of the great amount of effort and learning that went into the production of this system.

And it was fun.

APPENDIX A: PARTS COST

Part Description	Part Number	Quantity	Source	Unit Price	Subtotal
3 phase ac motors (surplus) (Retail $100)	N/A	3	electronicsgoldmine	12.00	36.00
PIC Microprocessor	PIC18F458	3	Microchip	6.26	18.78
PIC Microprocessor	PIC18F452	1	Microchip	5.70	5.70
Omnidirectional acetal skate wheels	2289T3	3	mcmaster.com	18.15	54.45
IGBT's	IR351F	18	Digi-Key	1.38	24.84
Hex Inverter (Schmidt Trigger)	74HC14	3	EE Stores	0.60	1.80
RC Timer	LM555	1	EE Stores	0.40	0.40
12V Lead Acid Batteries (surplus) (Retail $20)	N/A	2	electronicsgoldmine	10.00	20.00
IR LED	QED233-ND	5	Digi-Key	0.37	1.85
IR Receiver IS1U621L	IS1U621L	1	Digi-Key	1.33	1.33
RS-232 level shifter	MAX233CPP	1	Digi-Key	4.32	4.32
20 MHz Oscillator	98 18 D	2	EE Stores	2.40	4.80
Surplus 1/2" Polycarbonate	N/A	1	ClearCut Plastics	10.00	10.00
Aluminum brackets	N/A	1	Newton's House of Hardware	10.00	10.00
Quad 2-input NAND	74HC00	2	EE Stores	0.50	1.00
1.5A 5V Voltage Regulator	LM1086CT-5.0	2	Digi-Key	0.70	1.40
25 Foot 22 AWG wire hookup	N/A	2	EE Stores	1.00	2.00
Large Breadboard	2420 Tie Points	1	Elexp.com	17.00	17.00
Small Breadboard	830 Tie Points	1	Elexp.com	2.40	2.40
PCB board	N/A	1	mutr.co.uk	1.25	1.25
Heat Sinks	HS104-2-ND	20	Digi-Key	0.33	6.60
Half-Bridge Drivers	IR2109-ND	9	Digi-Key	2.03	18.27
Assorted 1/4 W Resistors	N/A	36	EE Stores	0.10	3.60
Assorted 1/2 W Resistors	N/A	7	EE Stores	0.10	0.70
Assorted Ceramic Capacitors	N/A	32	Digi-Key	0.01	0.32
Assorted Electrolytic Capacitors	N/A	18	EE Stores	0.30	5.40
Assorted Polyester Capacitors	N/A	2	EE Stores	0.20	0.40
Photo-Darlington Optoisolator	MCA230	9	EE Stores	0.50	4.50
FPGA (with Dev Board $50)	EP1K10TC100	1	fpga4fun.com	10.00	10.00
Toggle Switches	N/A	3	Lowe's Hardware	7.00	21.00
High Speed Diodes	UF1005DICT-ND	9	Digi-Key	0.34	3.06
9.6V Nickel Hydride Batteries	N/A	1	RadioShack	20.00	20.00
Octal Bus Transciever	74LS245	2	EE Stores	0.60	1.20
Nintendo Controller	N/A	1	Ebay	3.00	3.00
80 Wire 40 Pin Ribbon Cable	N/A	1	Newegg	1.00	1.00
				Total	318.37

APPENDIX B: MOTOR DRIVER SCHEMATICS



APPENDIX B: MOTOR DRIVER SCHEMATICS

U9
PIN NUMBERS ARE INCORRECT
PIC18C458

100-Pin FPGA not shown. For the Pluto Dev Board, the following pins are used for each signal:

Pin Name/Usage	Location	Dir.	I/O Standard
slow_clk	5	output	
q0a	6	input	
LED	7	output	
pwm1c	8	output	
q1a	21	input	
q1b	22	input	
q2a	23	input	
q2b	26	input	
reset	29	input	
data[5]	45	bidir	
data[6]	46	bidir	
pwm2a	47	output	
pwm2b	48	output	
pwm2c	49	output	
data[7]	50	bidir	
data[4]	55	bidir	
data[3]	56	bidir	
data[2]	61	bidir	
data[1]	65	bidir	
data[0]	69	bidir	
z2	70	input	
z1	71	input	
pwm0a	77	output	
pwm0b	78	output	
pwm0c	79	output	
pwm1a	80	output	
pwm1b	81	output	
z0	86	input	
clk	91	input	
adx[0]	96	input	
adx[1]	97	input	
adx[3]	98	input	
adx[2]	99	input	

9-pin Mystery CMOS Encoder

Title: Motor Control PIC and FPGA
Size: A
Document Number: N/A
Rev: <RevCode>
Date: Wednesday, March 17, 2004
Sheet 1 of 1

APPENDIX C: SOURCE CODE

The numerous source code files that we wrote for this project can be found on the following pages. Each file is given a heading at the top which roughly corresponds to the file name given here. The files are also broken down into sections representing which major module in the system they belong to.

- Path Control Module
  - pic_defines.h
  - path_control_module.h
  - path_control_module.c
  - io.h
  - io.c
  - path_engine.h
  - path_engine.c
  - types.h
  - trig.h
  - trig.c
- Motor Control Module
  - fpga.v (motor control FPGA code with testbenches)
  - motor_pic.h
  - motor_pic.c
  - drivers.c
  - boot_fpga.c
  - rbf_code.c
- Sensor Control Module
  - sensor_control.h
  - sensor_control.c
  - serial.h
  - serial.c
- Nintendo Interface Module
  - nintendo_main.c
  - nintendo.h
  - serial.h
  - serial.c
- PC Application
  - serial_config.cs
  - serial_port.cs
  - monkey_splash.cs
  - add_path_ui.cs
  - toilet_splash.cs
  - monkey_interface.cs