TEST BED FOR DEMONSTRATING

AND TEACHING SOFT

SENSOR CONCEPTS

By

JEREMY PAUL EVERT

Bachelor of Science in Mechanical

and Nuclear Engineering

Kansas State University

Manhattan, Kansas

2003

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2010

TEST BED FOR DEMONSTRATING AND TEACHING SOFT SENSOR CONCEPTS

Thesis Approved:

Dr. Martin Hagan

Thesis Adviser

Dr. Carl D. Latino

Dr. James E. Stine, Jr.

Dr. Mark E. Payton

Dean of the Graduate College

ACKNOWLEDGMENTS

Thank you Amey Phadke and Surpriya Tike for your support with Xilinx. It was always comforting to know I could come to you for help with the software. To my friend Amanuel Aseffa, many thanks for the long hours working through the A/D FSM code and learning Chipscope. We will need to drink some of your coffee and go over the new code together some time. Thank you Jarett Teel for finishing the motor control switch design and laying it out for milling. It was good to have the company of the NATCAR team down in the basement last summer. Thank you Megan Brady for your help with soldering the PIC boards and sensor boards. Thank you Kellen Butler for your help with the motor control switch and PIC board. You were very kind to come back to the project and help us learn to use the compiler. Thank you Max Murphy for your technical council and assistance with many aspects of the project. Thank you Jeff Henson for your continued support of the Smart Position Sensor GUI code. Thanks also to Taylor York and Daniel Nash for your work with the HDL code for the Smart Position Sensor project. Thank you Lory Ferguson for your help in purchasing the necessary components to make this project work.

.

I am thankful for the guidance and assistance of my committee. Thank you Dr. Stine for your help with Xilinx and the FPGA boards. Thank you Dr. Latino for the use of your lab and your time and guidance with the project. Thank you Dr. Hagan for all of your time and your help. I am very grateful for your help and patience.

Thank you to all my friends and family for your love and support throughout this process. I am especially grateful for my loving wife Amanda. Thank you for bearing with me while I completed this project.

TABLE OF CONTENTS

Chapter Page

Chapter Page

Chapter Page

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

The objective of this project was to develop a platform that could be used to demonstrate two concepts: 1) how neural networks can be implemented on FPGAs, and 2) how the FPGA neural network can be used as the fundamental component of a smart sensor system. This chapter provides an overview of this platform and an outline of the rest of this thesis.

1.1 Project Overview

The purpose of this project is to produce a test bed for demonstrating soft sensors, and how they can be implemented with neural networks on FPGAs. The test bed of choice is a smart sensor car. The smart sensor car follows a wire. The wire forms a track and produces a changing magnetic field. This magnetic field is translated into a position measurement using sensors, signal conditioning, analog to digital converter and a neural network. The position measurement is used by a PD controller that sends signals to the car motor and steering mechanism, and results in motion around the track. Figure 1-1 shows the main block diagram.

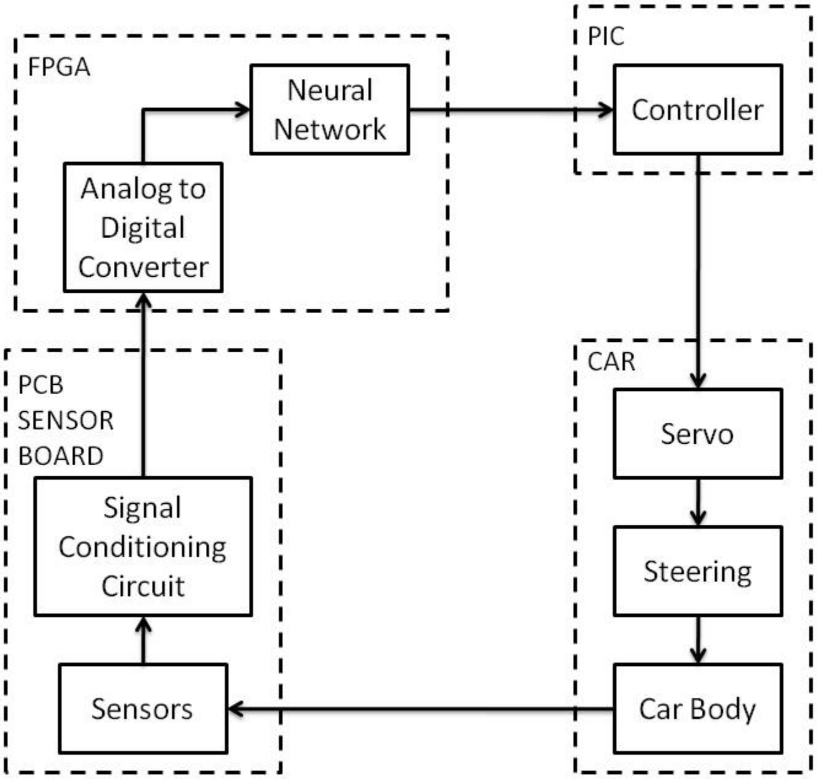


Figure 1-1: Main Block Diagram

Smart sensor concepts translate the magnetic field into a position measurement. First, four inductors produce signals in response to the strength of the magnetic field in their area. These signals go through a signal conditioning circuit and are then converted to digital numbers by the analog to digital (A/D) converters. The four digital numbers become the four inputs to the artificial neural network on the Field Programmable Gate Array (FPGA). The output of the neural network is position. The controller uses this position to generate motor speed and steering servo commands for the electric car. The electric car responds by moving around the track. When the car moves, the sensors enter a different magnetic field, which results in a new position calculation and new steering command. Figure 1-2 shows the system functional overview.

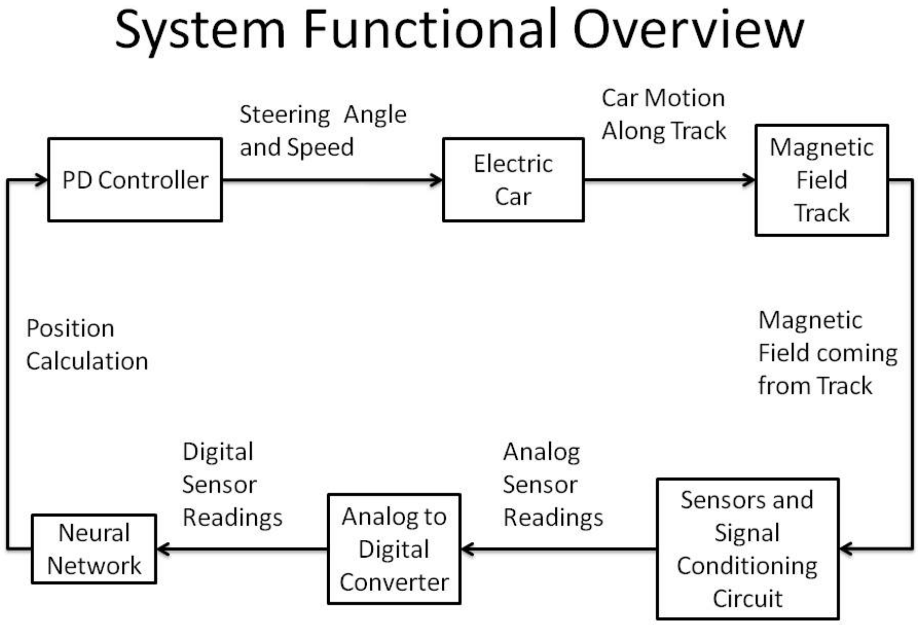


Figure 1-2: System Functional Overview

1.2 Project History and Current Status

The main component of the smart sensor is the artificial neural network. In the summer of 2008, Dr. Hagan, Dr. Latino and Dr. Moreno-Armendariz wrote code to implement the neural network on an FPGA. During the fall of 2008, Jeff Henson, Daniel Nash, Craig Noltensmeyer and Taylor York worked together to expand the code to work as a smart position sensor. This work was part of a capstone design class at Oklahoma State University. Their project used the output of two light sensors as inputs to the neural network. The neural network calculated the position of an object in front of the sensors blocking the light. Their project incorporated use of the on-board A/D converter and LCD display. During the fall of 2009, the project was given to another group in the capstone design class. The team included Amanuel Assefa, Kellen Butler and Stephanie Pickrel. The team was successful in building a microcontroller board to communicate with the FPGA and generate speed and steering commands for the electric car. The team also changed the code to read in four sensor readings by incorporating an off board multiplexer.

The author's contributions began during the fall of 2009 by assisting Amanuel Assefa with the Xilinx software to change the FPGA code to read in four sensors and control the multiplexer. In the spring of 2010, the author began work on the rest of the smart sensor car. This included development, fabrication and testing of the sensor board. The next contribution was the design, testing and fabrication of the motor control switch. After the individual components were ready, the author designed and fabricated the power system and board mounting fixtures. Five cars were prepared for an academy for high school students during summer 2010. During the summer academy, the author assisted students with modules and equipment in the lab. The author contributed to some of the writing for the summer academy documents. The author instructed students about how to use the smart sensor cars and assisted them as needed. After the summer academy, the author documented all system components.

The current status of the project is that the smart sensor car is able to go around a simple track slowly. The car has considerable steering oscillation. The project reached this point before the summer academy.

1.3 Thesis Outline

Chapter two covers the sensors and signal conditioning circuit. It describes the magnetic field and sensors to detect it. It also describes the signal conditioning circuit, defining the circuit components, how they were tested and the fabrication of the printed circuit board.

Chapter three focuses on the A/D converter. It describes the on-board hardware and off-board multiplexer operation, and the control of the A/D conversion. The chapter also describes how the A/D conversion process was tested.

Chapter four focuses on the artificial neural network. It begins by providing basic information about the artificial neural network implemented on the smart sensor car. Next, It describes the implementation of the neural network showing timing diagrams for the process. The chapter also discusses how the neural network was trained. It also includes discussion about the support equipment required to make the FPGA neural network functional.

Chapter five describes the PD controller. The chapter describes how the controller is implemented and explains how the controller works. The chapter also describes the modeling process and how the model was used to determine the controller parameters. It also includes discussion about the support equipment required to make the controller functional.

Chapter six describes the electric car. It describes the components that make up the electric car, including the motor control switch. The chapter also describes the board-mounting fixture and how the individual boards connect to each other.

Chapter seven describes the summer soft sensor academy. It provides an overview of the academy and states the academy objectives. The chapter also describes the student activities.

Chapter eight, Conclusions and Future Work, is the last chapter. It provides a summary of the project and reiterates the author’s contributions. The chapter ends with a description of possible future work.

CHAPTER II

SENSORS AND SIGNAL CONDITIONING CIRCUIT

This chapter discuses how the magnetic field coming from a wire is transformed into signals that can be sampled by analog to digital converters (A/D) and read into the FPGA neural network, as shown in Figure 2-1. The chapter begins with an introduction, followed by a brief discussion of sensors, magnetic fields and how they interact. Section two covers the basic design of the signal conditioning circuit. Section three describes the steps in proto-board testing. Section four is on the printed circuit board. Section five summarizes the chapter.

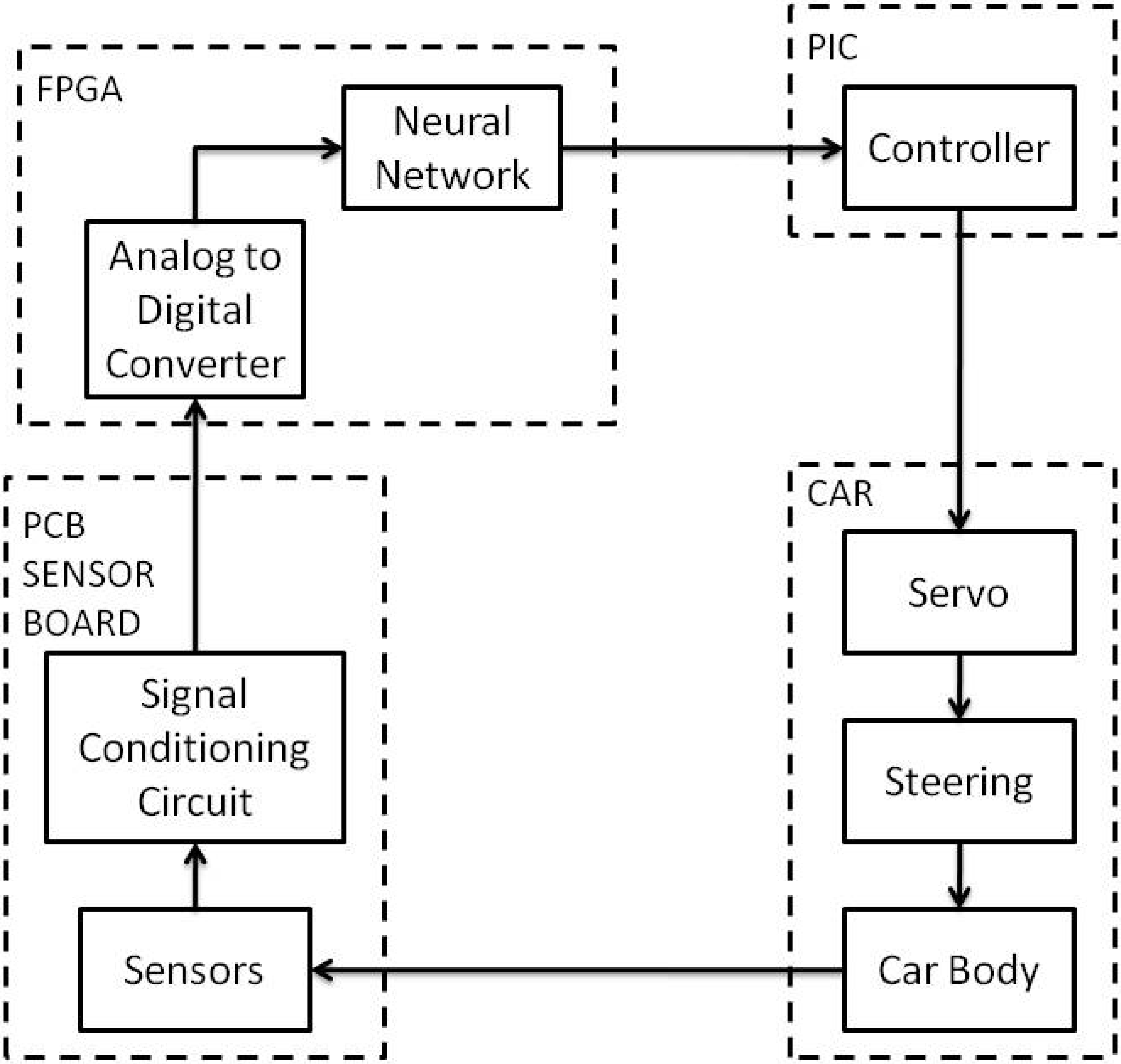


Figure 2-1: Main Block Diagram

2.1 Magnetic Fields and Inductors

As the car moves along the wire track, it encounters a magnetic field coming from current moving through the wire. Sensors detect the magnetic field and generate voltages that go to the signal conditioning circuit. Figure 2-2 shows how these functions fit into the rest of the system. This section describes the magnetic field and sensors that detect it.

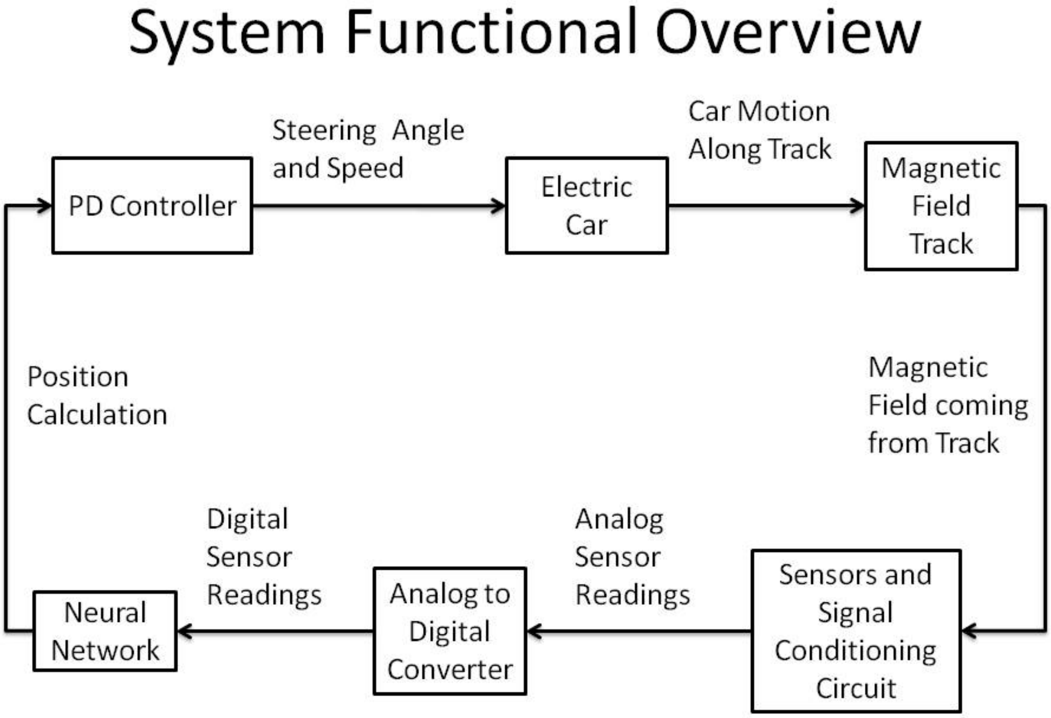


Figure 2-2: System Functional Overview

The magnetic field is generated by passing a sinusoidal wave through a wire. The generated field is a series of rings perpendicular to the path of the electrons moving through the wire, as shown in Figure 2-3.

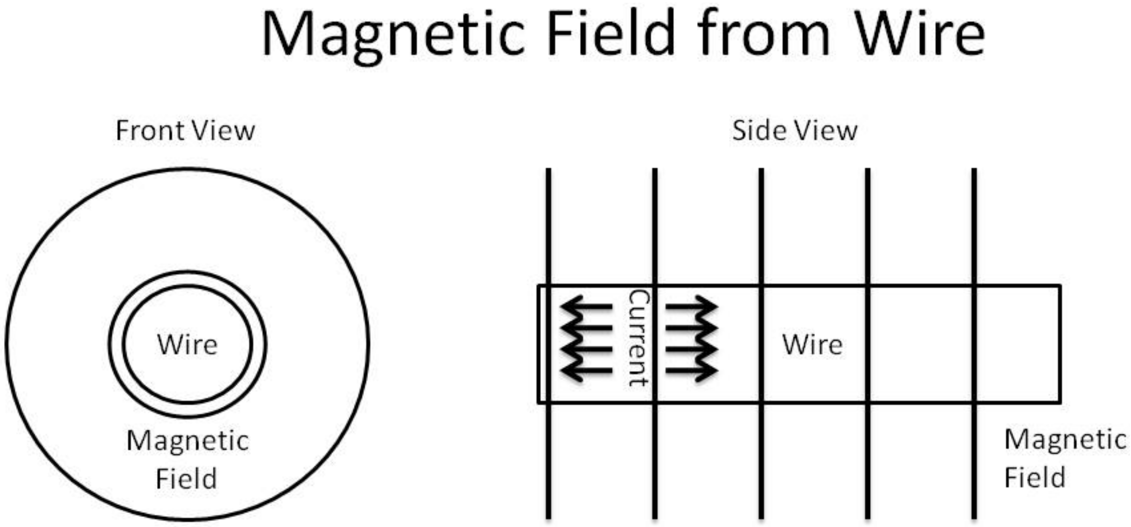


Figure 2-3: Magnetic Field from Wire

The Biot-Savart law describes the magnetic field generated by an electric current: $B = kq\frac{v \times x}{|x|^3}$, where B is the magnetic field, k is a constant, q is the charge moving through a wire, v is charge velocity and x is the distance from the wire to the point being observed. In parallel wires, a charge moving in the first wire will cause a magnetic field. The magnetic field will cause a charge to move in the second wire. The moving charge in the second wire can be measured as a voltage difference between the two ends of the second wire.

An experiment with a wire and inductor can help visualize the magnetic field. (see Figure 2-4) The first wire is connected to a function generator. This provides the moving charge in the first wire, which results in the magnetic field. The inductor is placed in the magnetic field. The voltage across the inductor is due to Faraday's law of induction. The voltage induced in a coil is proportional to the time rate of change of the magnetic flux through the coil. The voltage difference is measured across the two leads of the inductor. The field strength is proportional to the inverse square of the distance. The field strength is directional. As the angle between the two wires increases, the magnetic induction effects are reduced. Figure 2-4 shows an inductor on a wire. In Figure 2-4, distance between the center of the wire and center of the inductor is zero and the angle between wire and inductor is ninety degrees.

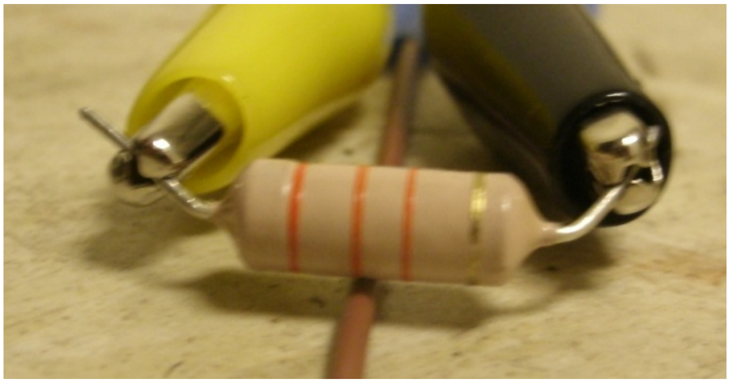


Figure 2-4: Inductor on a Wire

Figure 2-5 shows the voltage measured across an inductor as distance from the center of the inductor to the center of the wire is increased. This displays an inverse square relationship, as predicted by the Biot-Savart law.

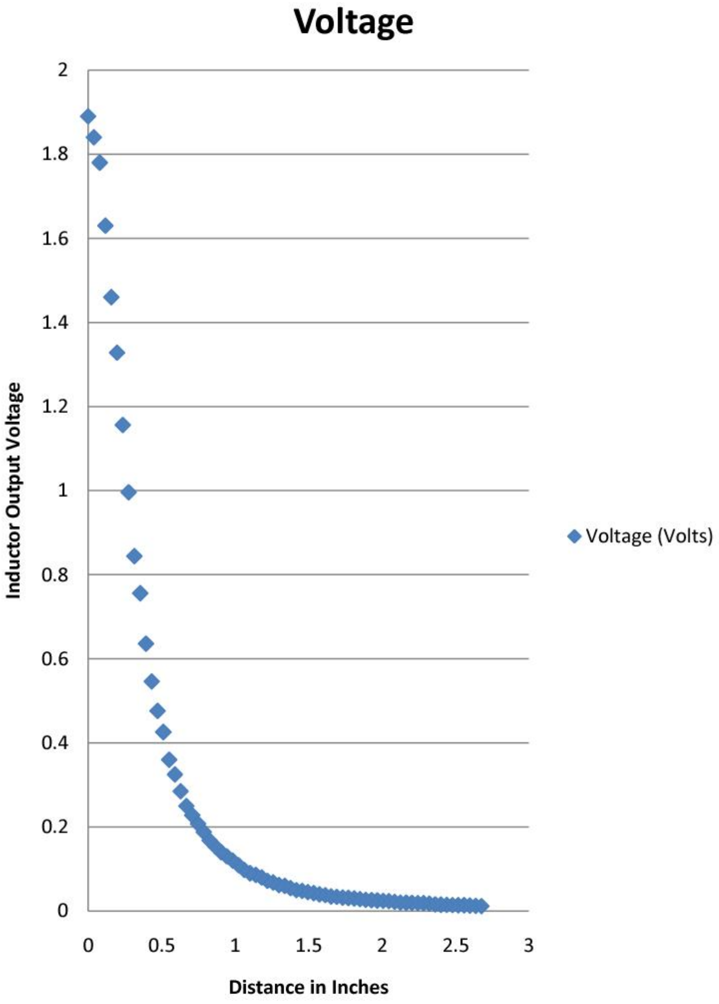


Figure 2-5: Voltage Across an Inductor versus Distance

Figure 2-6 shows the voltage across the inductor as the angle between the inductor and wire is increased. This shows that the field effect depends on the orientation of the wire, with parallel wires having the strongest inductive effects, as predicted by Faraday's law. If the changing magnetic field does not pass through the coils, no voltage will be induced.

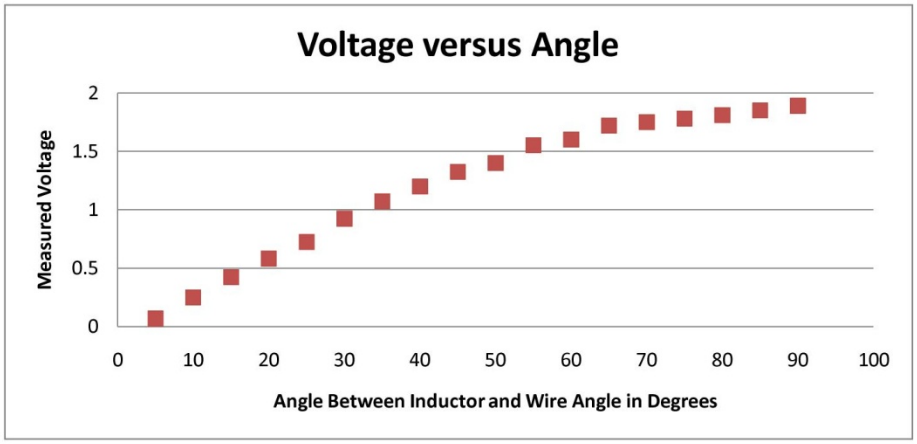


Figure 2-6: Voltage versus Angle

2.1.1 PNI Magneto Inductive Sensors

One possibility for sensing magnetic fields is the application of PNI Magneto-inductive position sensors, from PNI Sensor Corporation. This sensor is designed to be sensitive enough to detect the Earth's magnetic field. The maximum allowed voltage between sensor terminals is 2.5 volts, unlike the inductors that experienced differences of over 25 volts between the terminals without obvious signs of damage. Previous OSU students have used inductors for sensing magnetic fields. The PNI sensor performance was compared against inductors. The response of both sensors was normalized to unity. Figure 2-7 shows the percent of maximum signal strength versus distance in mm, measured from the center of the wire to the center of the sensor, for both sensors.

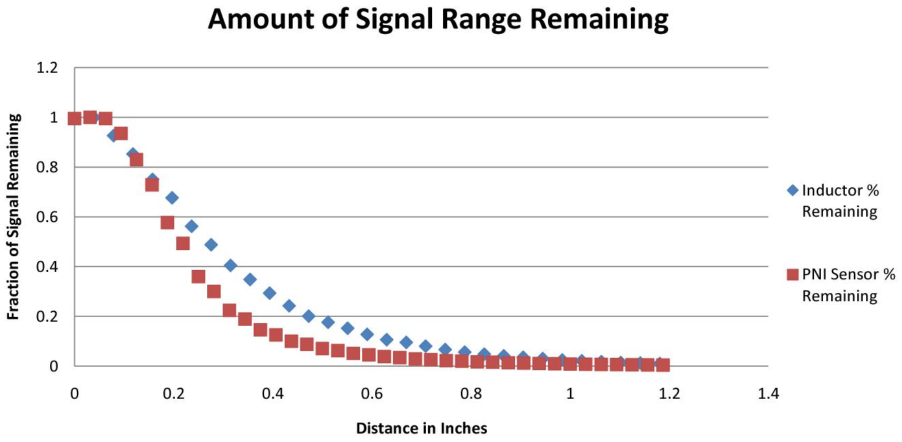


Figure 2-7: Normalized Circuit Output Response versus Distance from Center of Sensor to Center of Wire

The magneto-inductive sensors did not outperform the inductors. Because inductors have been used in the past, are cheaper, more readily available, and more robust, inductors are the sensor of choice for this project.

2.1.2 Inductors

A previous NATCAR team of OSU Tulsa students suggested using 33 millihenry inductors to sense a magnetic field from a wire carrying a 100 mA sine wave with a 75-kilohertz frequency. Experiments showed that these inductors have a slightly greater response at higher frequencies, with a maximum near 100 kHz. 33 millihenry inductors sensing a magnetic field from a wire carrying a 100-kilohertz sine wave are used for this project.

Figure 2-7 was created at a single sensor elevation above the wire. The sensor stayed at the same vertical height and moved horizontally away from the wire. To determine the most useful sensor elevation, multiple sets of data were collected at different elevations. The normalized results are shown in Figure 2-8. The sensors with greater elevation have a lower voltage output when directly over the wire, but this can be compensated by amplifying the signal. The advantage of the greater elevation is that the signal drop-off with distance is much slower. This gives the sensor a wider range of operation, and therefore fewer sensors will be needed. Extensive testing showed that the sensors could be raised to 2.75 in. This provided enough signal strength and a wide range of operation. The final design used a sensor elevation of 2.75 inches.

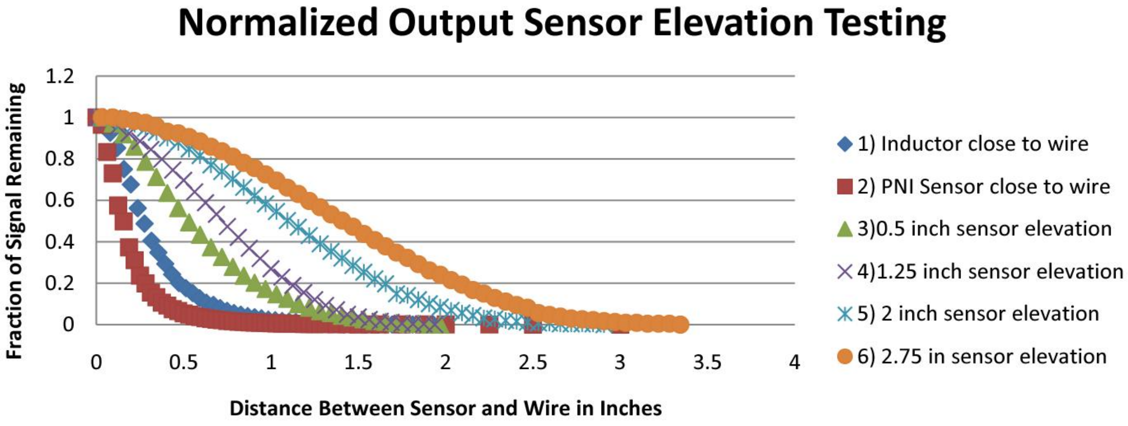


Figure 2-8: Sensor Elevation Testing

The peak-to-peak inductor voltage only indicates the distance from sensor to wire, but not the direction. Applying the smart sensor concept allows the neural network to use multiple sensors in concert to determine the position of the wire. Placing the inductors in a straight line increases the horizontal distance over which the smart sensor is effective.

Figure 2-9 shows four sensors laid out in a straight line. Figure 2-8 shows that with a 2.75-inch elevation, the sensor has a range of nearly three inches. A spacing of two inches between inductors ensures the wire will not fall into a flat spot between inductor response curves. Figure 2.10 shows four sensor responses at a spacing of two inches. The smart sensor car uses four sensors resulting in an effective measurement range of eight inches.

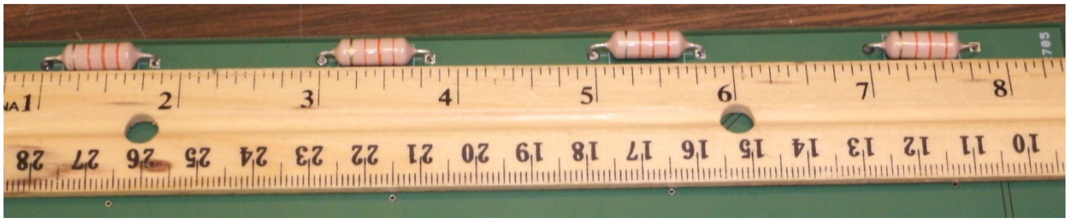


Figure 2-9: Multiple Sensor Layout

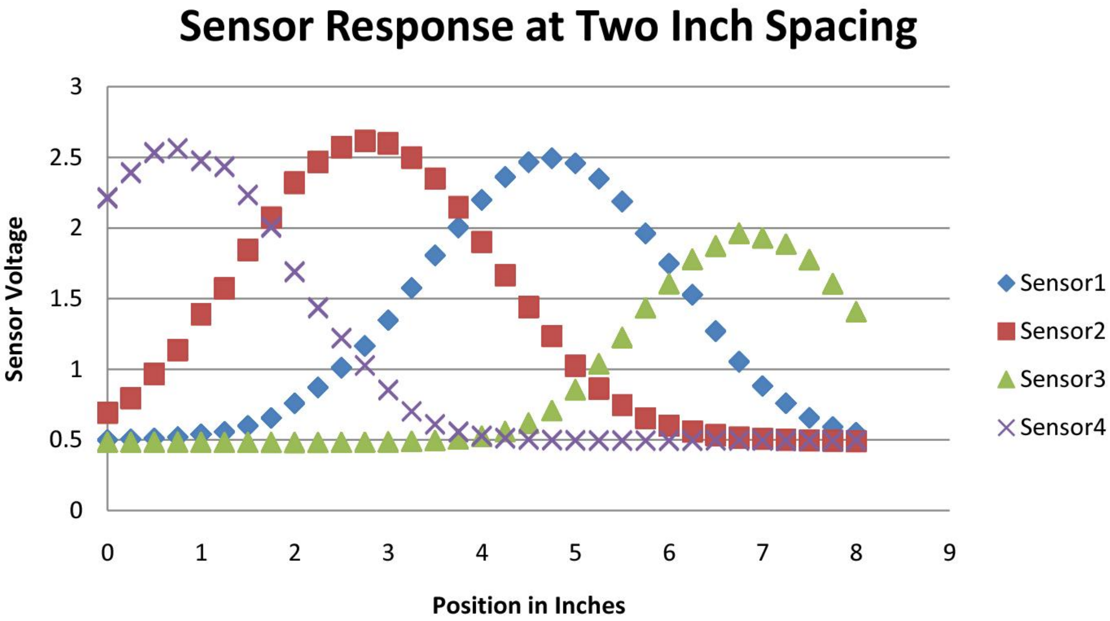


Figure 2-10: Sensor Responses

2.2 Signal Conditioning Circuit

Figure 2-11 shows how the sensors and signal conditioning circuit fit into the functional overview. The signal conditioning circuit makes the sensor output usable for the A/D converters.

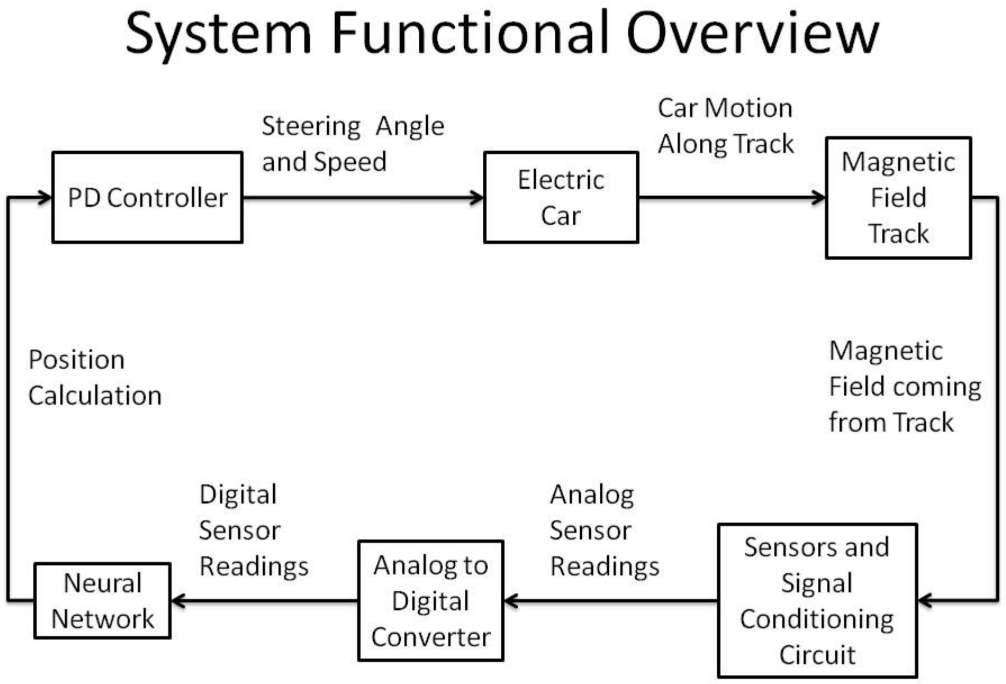


Figure 2-11: System Functional Overview

Figure 2-12 shows the major components on the sensor and signal conditioning board. The signal conditioning board has four sensors. There are only two A/D channels on the FPGA board. The output of the four signal conditioning circuits are passed through an analog four to two multiplexer to allow four sensors to be read by the two A/D channels.

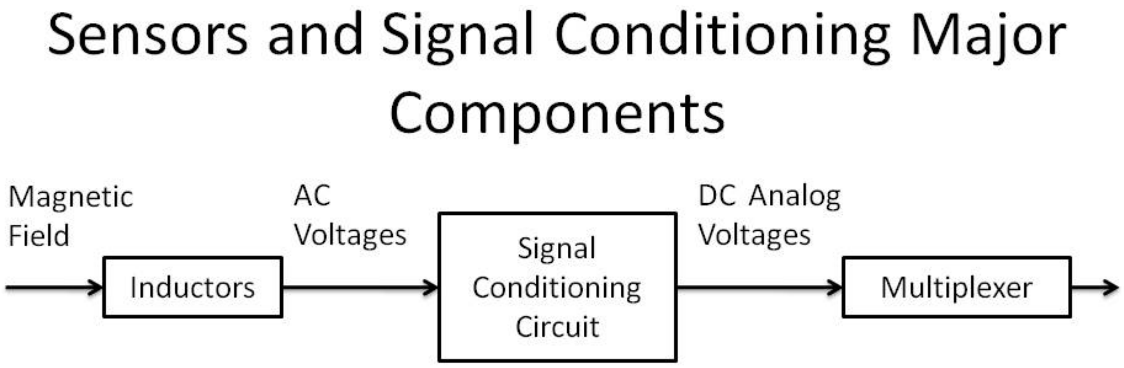

Sensors and Signal Conditioning Major Components
Magnetic Field
Inductors
AC Voltages
Signal Conditioning Circuit
DC Analog Voltages
Multiplexer

Figure 2-12: Sensors and Signal Conditioning Macro Components

The signal conditioning circuit accepts signals from the inductor and provides signals to the analog to digital converters. The inductor provides a sinusoid waveform voltage oscillating at 100 kilohertz with a magnitude that changes with proximity to the wire. The inductor output is never more than 100 millivolts peak to peak. The A/D converter accepts DC signals in the range of 0.4 to 2.9 volts. The signal conditioning circuit converts the sinusoidal voltage from the inductor to a DC value between 0.4 and 2.9 volts.

Figure 2-13 shows the six individual components of the signal conditioning circuit. The inductor output first goes through a voltage follower that prevents the sensor from being loaded by the rest of the circuit. The non-inverting amplifier makes the signal large enough to work with. The peak detector changes the 100 kHz sine wave into a DC signal. The second voltage follower prevents loading of the peak detector. The summing amplifier provides the appropriate gain and offset to match the input range of the A/D converters. A first order low pass filter prevents high frequency noise from going into the A/D converters.

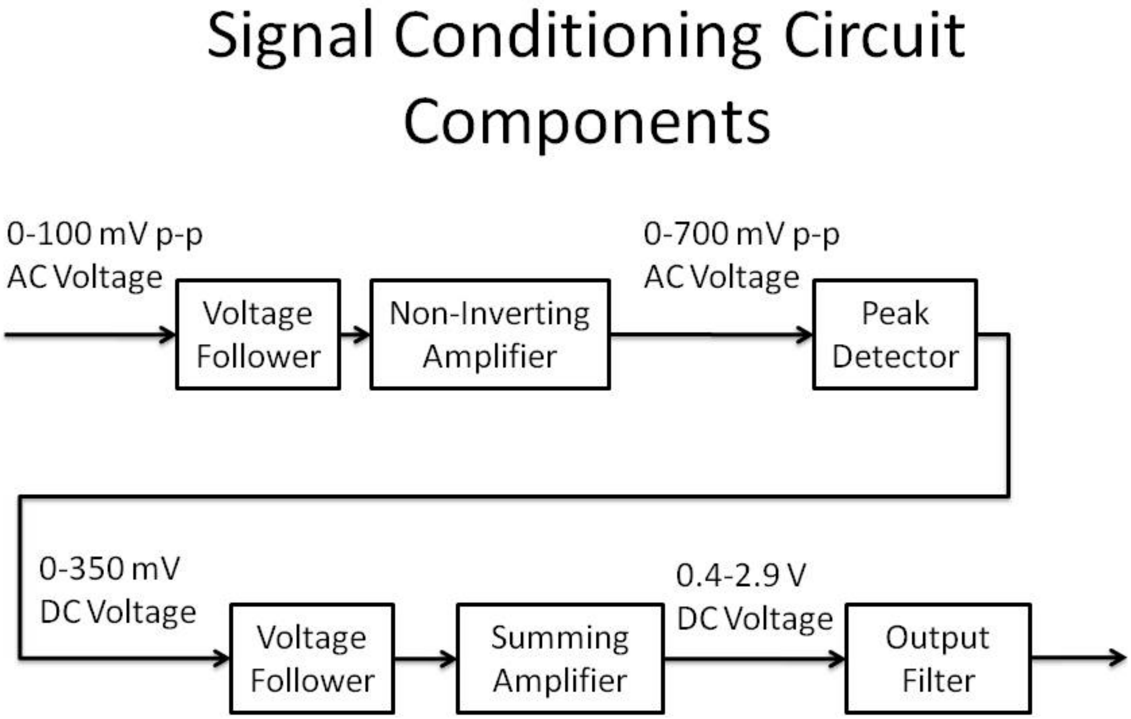


Figure 2-13: Signal Conditioning Circuit Components

The first component in the signal conditioning circuit is a voltage follower. The voltage follower goes between the inductor and remaining circuit to act as a buffer to prevent loading of the inductor. The equation for a voltage follower is:$V_{out} = V_{in}$. Figure 2-14 shows an op amp in a voltage follower configuration.

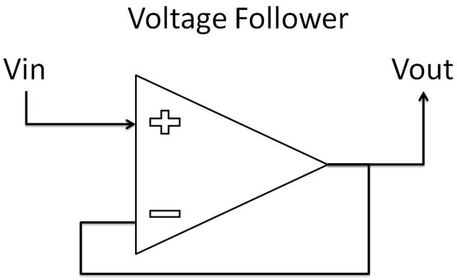


Figure 2-14: Voltage Follower

For the first voltage follower, the input is a 100 mV p-p sine wave at 100 kHz. Figure 2-15 shows a Multisim Circuit to simulate a 100 mV p-p sine wave at 100 kHz.

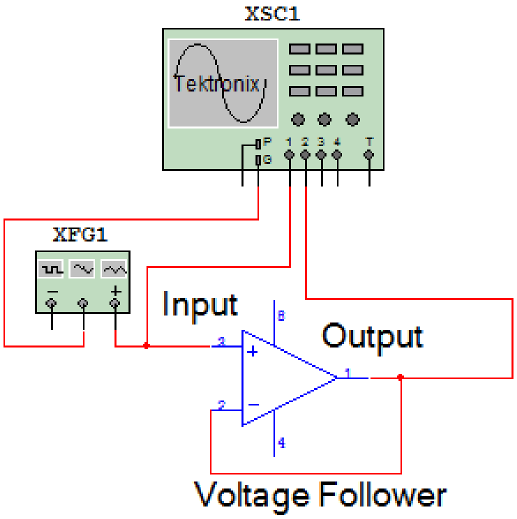
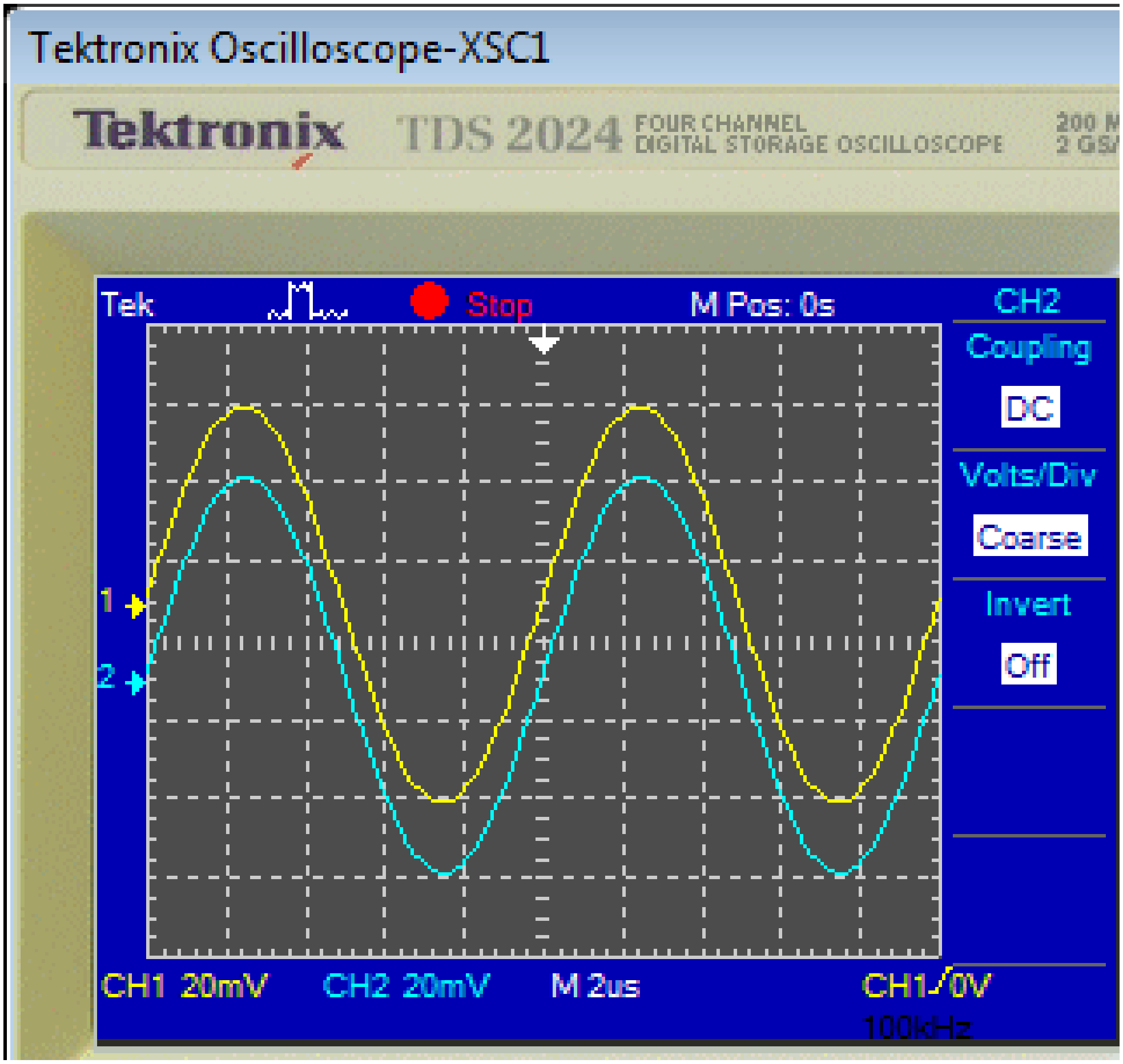


Figure 2-15: Multisim Simulation of a Voltage Follower

Figure 2-16 shows the oscilloscope reading from the voltage follower simulation. The input is channel 1, show on top. The output is channel 2, shown on bottom. Both channels are set to 20 mV per division, and the time scale is two microseconds per division. This shows that the input matches the output for the voltage follower.


Tektronix Oscilloscope-XSC1
Tektronix
TDS 2024 FOUR CHANNEL DIGITAL STORAGE OSCILLOSCOPE
200 M
2 GS/
Tek
Stop
1
2
CH2
Invert
Off

M Pos: 0s

Coupling DC

Volts/Div Coarse

CH1 20mV CH2 20mV M 2us CH1 ╱ 0V

100kHz

Figure 2-16: Voltage Follower Simulation Oscilloscope

Figure 2-17 is a plot of the oscilloscope reading from testing the completed circuit, where the output is nearly a perfect match to the input. Figures 2-16 and 2-17 show that the simulated and actual oscilloscope readings are similar.

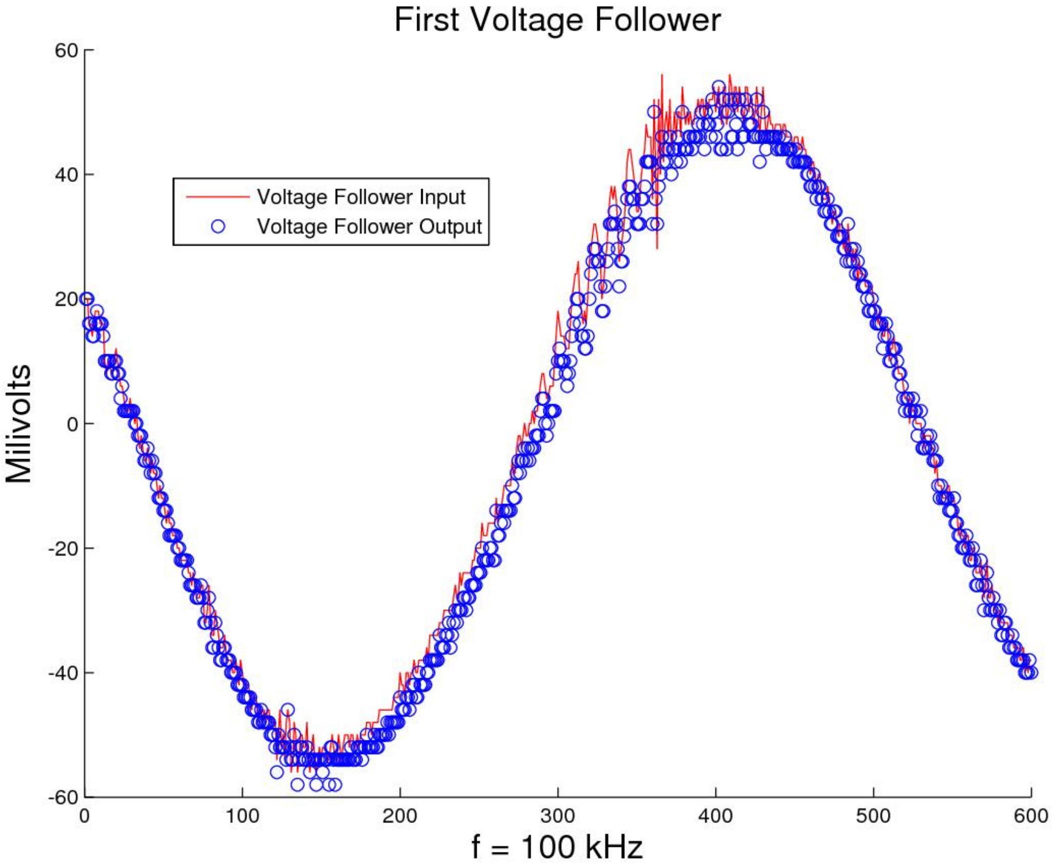


Figure 2-17: Voltage Follower Physical Oscilloscope Data

The voltage follower input is shown as a continuous signal to make its plot distinguishable from the output. Both the input and output were digitally sampled. The noise on the voltage follower input signal came from the physical connection between the function generator and oscilloscope.

The output of the voltage follower then goes to the input of a non-inverting amplifier. The non-inverting amplifier ensures the incoming signal is large enough for the peak detector to work with. Specifically, the non-inverting amplifier makes the signal large enough to overcome the turn-on voltage of the diode. Figure 2-18 shows an op amp in a non-inverting amplifier configuration.

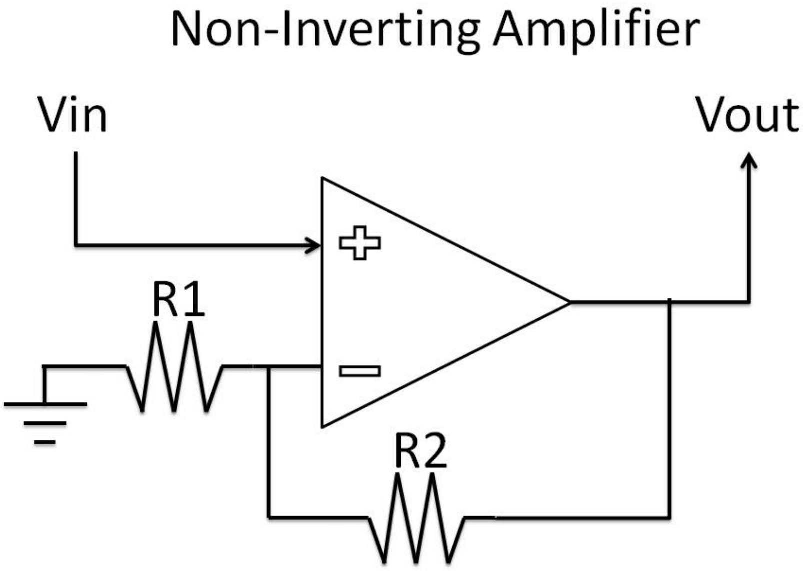
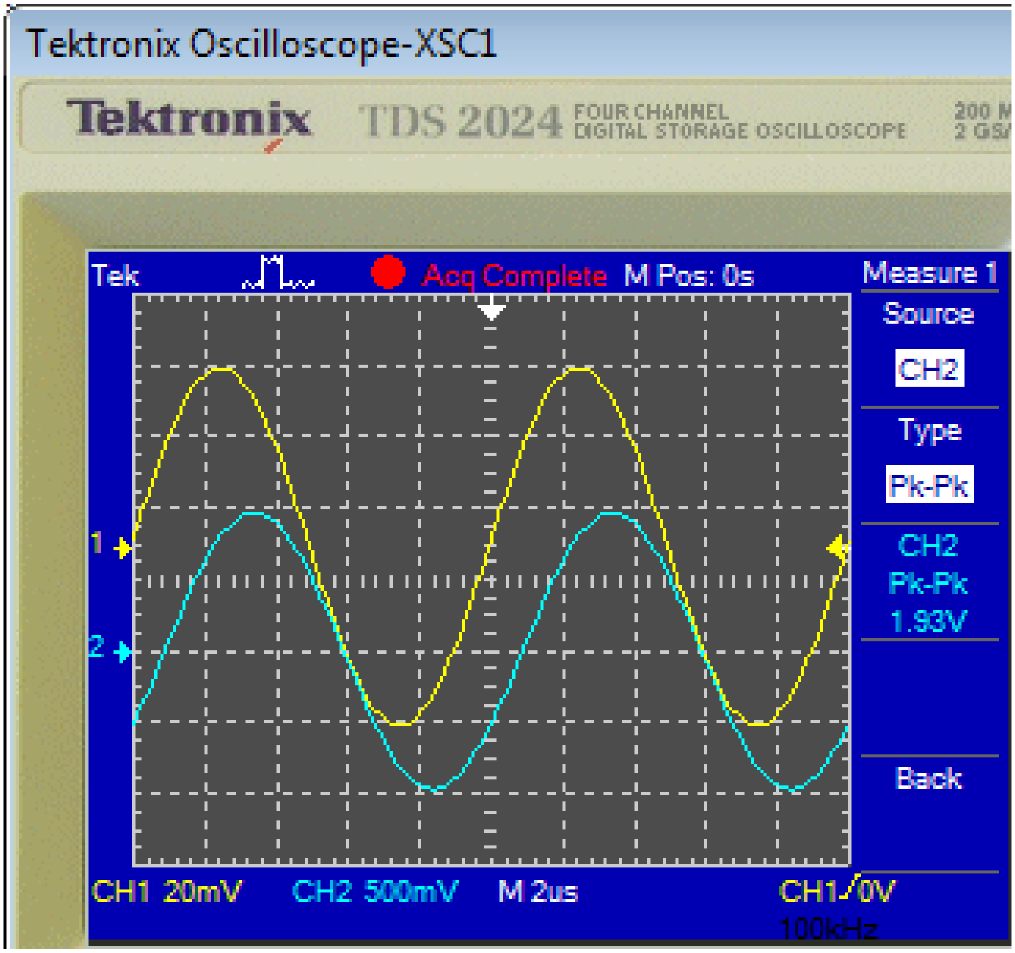


Figure 2-18: Non-Inverting Amplifier

The equation for a non-inverting amplifier is:$V_{out} = V_{in} * (1 + {}^{R2}/_{R1})$. For the sensor board, R2 is a 150-kOhm resistor and R1 is a 6.8 kOhm resistor, and the resulting gain for the circuit is 23.1. A non-inverting op amp with an input of 100 mV p-p and a gain of 23.1 should produce an output of about two volts peak to peak.

Figure 2-19 shows the oscilloscope output for simulating the non-inverting amplifier. Channel 1 is the input on top and is still 100 mV p-p. Channel 2 is on bottom. The volts per division for channel 2 has changed to 500 mV per division, which means the output signal is almost two volts peak to peak which is close to the expected value from the equation.



Acq Complete M Pos: 0s

Measure 1
Source
CH2
Type
Pk-Pk
CH2
Pk-Pk
1.93V
Back

CH1 20mV CH2 500mV M 2us CH1 ⁄ 0V
100kHz

Figure 2-19: Non-Inverting Amplifier Simulation Oscilloscope

Figure 2-20 shows the measured input and output of the physical circuit. Figures 2-19 and 2-20 show that the physical implementation matches the theoretical expectations from the simulation.

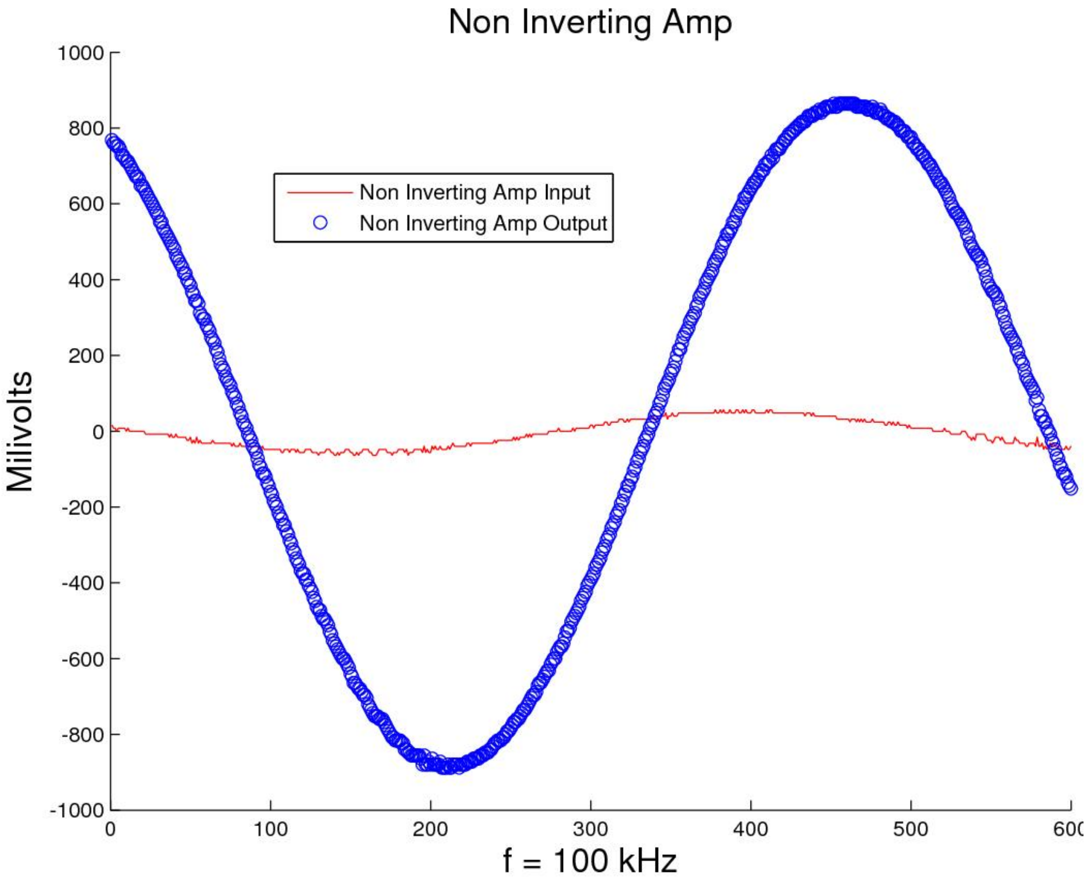


Figure 2-20: Non-Inverting Amp Physical Oscilloscope

Note that the voltage follower which has a gain of 1 had no phase shift, but the non-inverting amplifier with a gain over 20 had a noticeable phase shift. The simulation predicted this phase shift, and the physical response verified the model.

The output of the non-inverting amplifier is fed into a peak detector circuit. The peak detector circuit converts the sine wave signal into a DC signal. Figure 2-21 shows a negative peak detector.

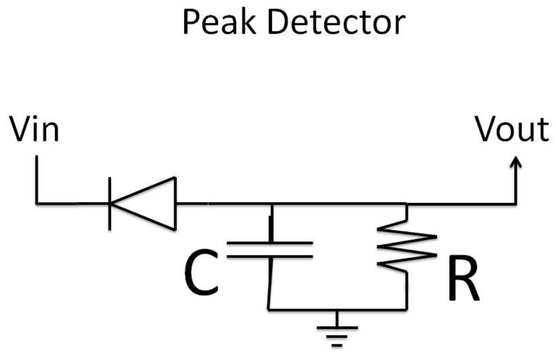
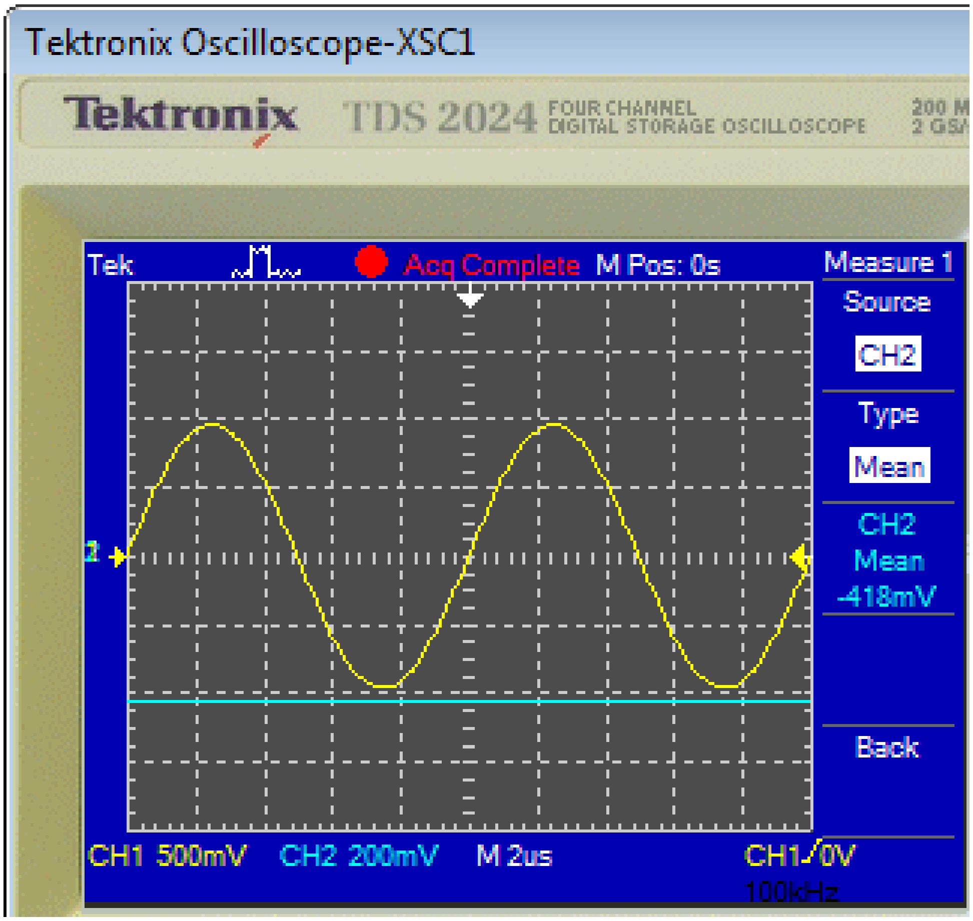


Figure 2-21: Negative Peak Detector

The time constant for the RC circuit is $\tau = R * C$. For the sensor board, R is 61.9 kOhms and C is 0.1 µFarads, so $tau\ =\ 0.1 * 10^{-6}\ X\ 61.9 * 10^{3} = 6.2 * 10^{3}$ or 6.2milleseconds. The input wave period is $0.1 * 10^{-5}$ seconds.

For tau values that are large with respect to the frequency of the wave they filter, the output of a peak detector circuit can appear as a DC signal. Figure 2-22 shows the oscilliscope output for the simulation.



Tektronix Oscilloscope-XSC1

Tek Acq Complete M Pos: 0s

Measure 1

Source CH2

Type Mean

CH2 Mean -418mV

CH1 500mV CH2 200mV M 2us CH1 ⁄ 0V 100kHz

Figure 2-22: Peak Detector Simulation Oscilloscope

Figure 2-23 shows the measured signals from the physical circuit. Figures 2-22 and 2-23 show that the simulated outcome and the measured outcome of the physical circuit are close.

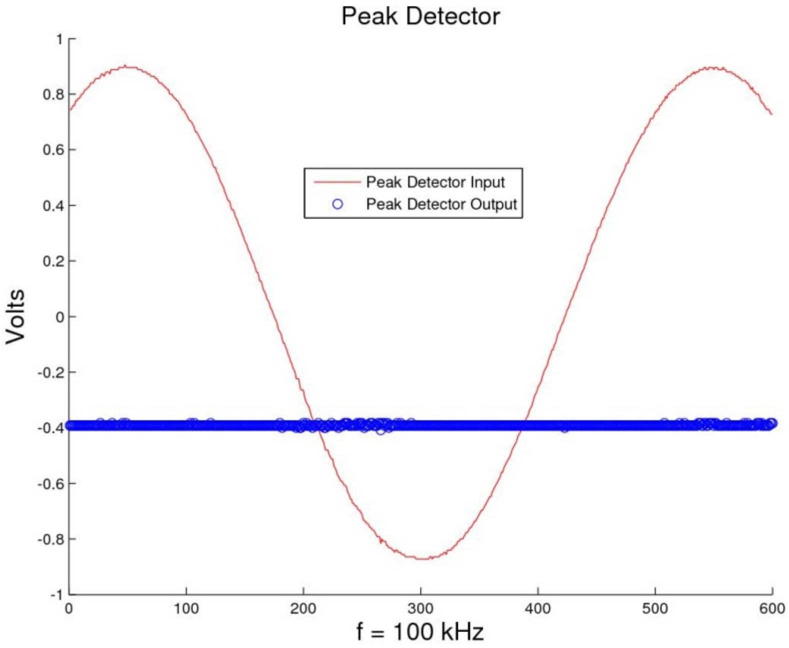


Figure 2-23: Peak Detector Physical Oscilloscope

The output of the peak detector is passed to the input of the second voltage follower. The second voltage follower is used to prevent loading of the peak detector circuit. The output of the second voltage follower is connected to one input of a summing amplifier. The summing amplifier is used to adjust the voltage to a range of 0.4 to 2.9 volts. A summing amplifier can add together many inputs and apply different gains to those inputs. The equation for a summing amplifier is

$$V_{out} = -1 * R_f * \left(\frac{V_1}{R_1} + \frac{V_2}{R_2} + \cdots + \frac{V_n}{R_n}\right)$$

The summing amplifier for the signal conditioning circuit has two inputs. Figure 2-24 shows the summing amplifier for the signal conditioning circuit.

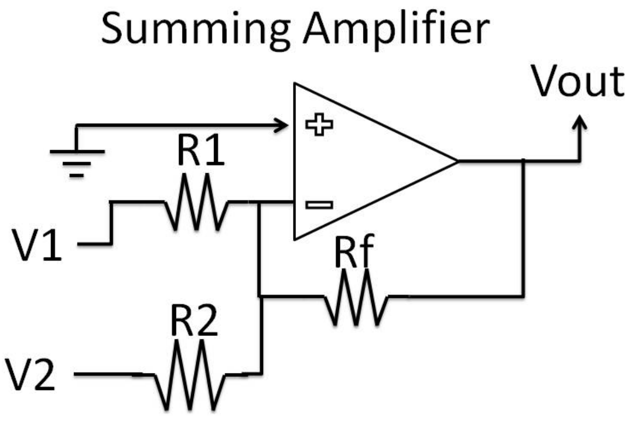


Figure 2-24: Summing Amplifier with Two Inputs

The first input, V1, comes from the second voltage follower. The second input, V2, comes from a voltage divider. The voltage divider input provides an offset to the signal. Figure 2-25 shows a voltage divider.

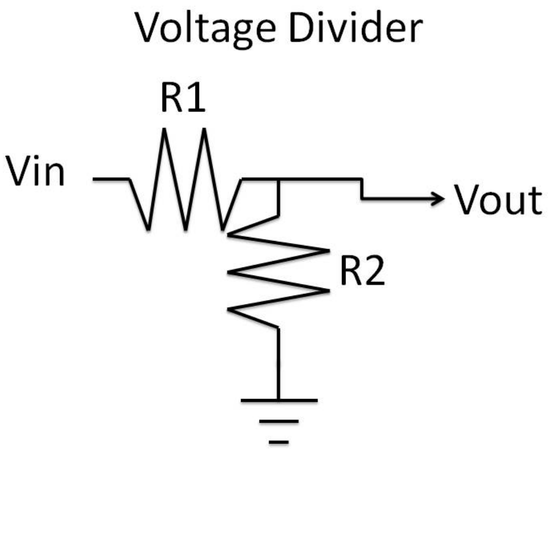


Figure 2-25: Voltage Divider

The equation for a voltage divider is:

$$V_2 = V_1 * \left(\frac{R_2}{R_1 + R_2}\right)$$

For the voltage divider, R2 is 1.8 kOhms and R1 is two kOhms. The input to the voltage divider is -5 volts, the output of the voltage divider is -2.37 volts. The gain for the V2 is -1*26.7k/130k or about -0.21. This means the voltage divider will contribute -0.21 * -2.37 volts or 0.4977 volts to the output. The summing amplifier signal gain is 26.7k/4.64k or 5.75. Figure 2-26 shows the summing amplifier simulation circuit.

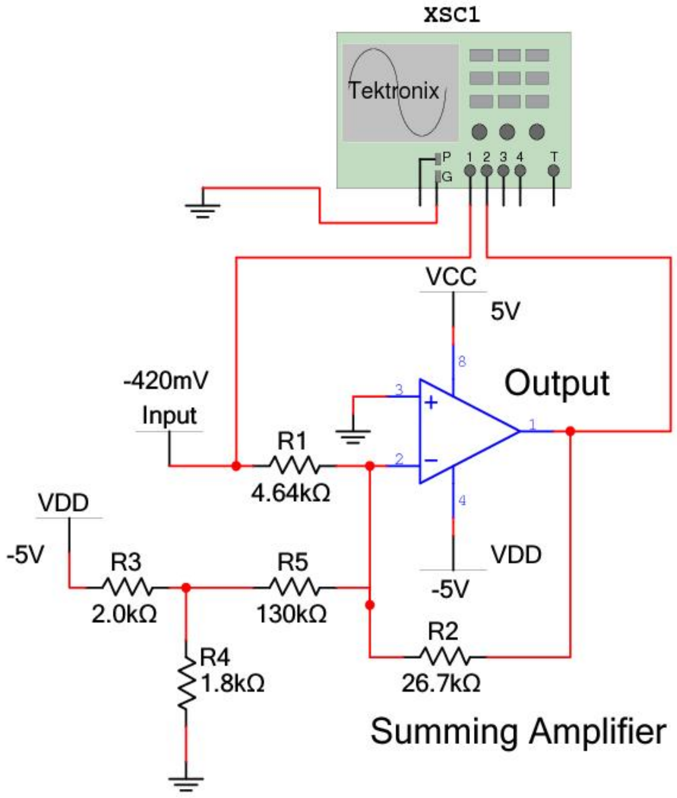
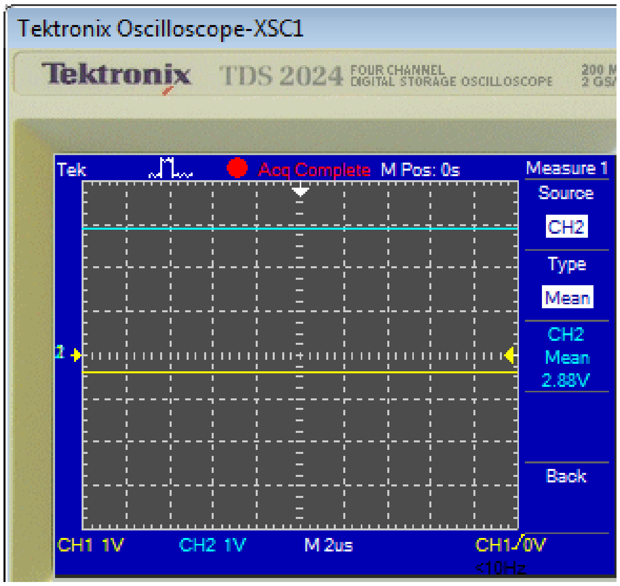


Figure 2-26: Summing Amplifier Simulation Circuit

Figure 2-27 shows the simulation oscilloscope for the summing amplifier. Figure 2-28 shows the physical oscilloscope readings for the summing amplifier.



Measure 1
Source
CH2
Type
Mean
CH2
Mean
2.88V

Figure 2-27: Summing Amplifier Simulation Oscilloscope

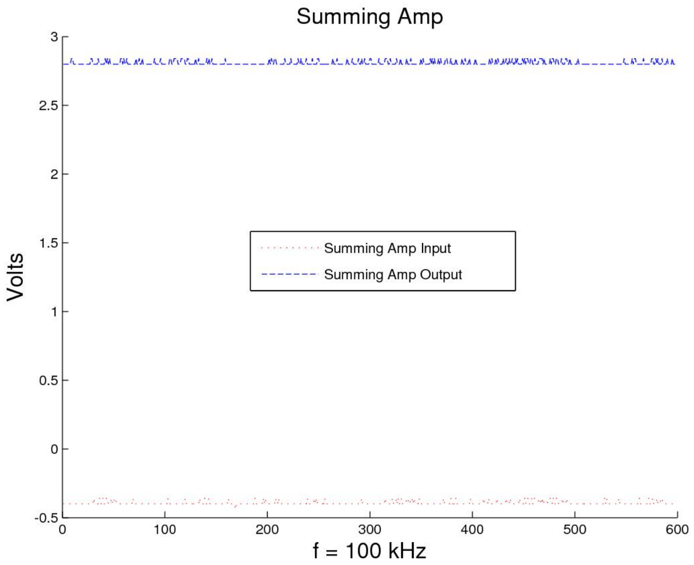


Figure 2-28: Summing Amplifier Physical Oscilloscope

The output of the summing amplifier is passed to the output filter. The purpose of this filter is to reduce high frequency noise on the circuit output, which is mostly a DC analog voltage. Figure 2-29 shows the signal conditioning circuit output filter. The cut off frequency in hertz is: $f_c = \frac{1}{2\pi RC}$, where R is a 15-kOhm resistor and C is a 3,300 pF capacitor, and $f_c = 3.2\ kHz$. Figure 2-29 shows an output filter and Figure 2-30 shows the frequency response.

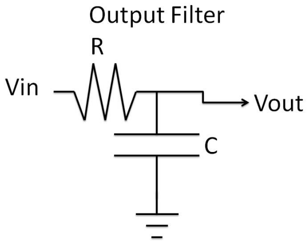


Figure 2-29: Output Filter

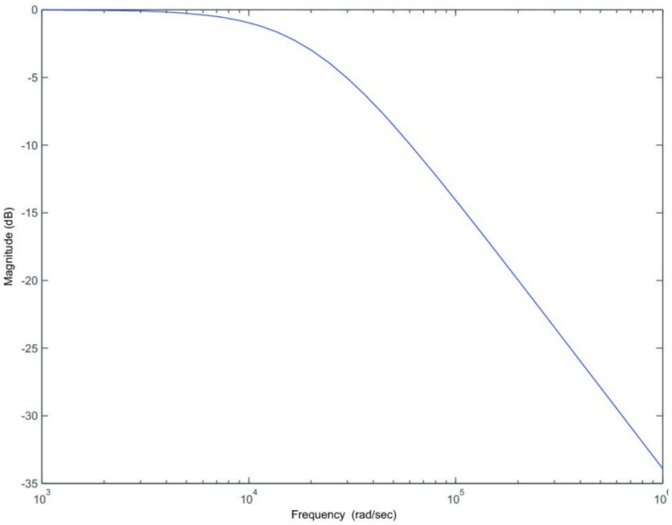


Figure 2-30: Frequency Response of Filter

In summary, the circuit has six main components. The first voltage follower prevents loading of the inductor. The non-inverting amp makes the signal large enough to work with. The peak detector smoothes the sine wave into a DC signal. The second voltage follower prevents loading of the peak detector. The inverting amp and voltage divider offset and amplify the signal to fill the range of 0.5 to 2.9 volts to match the input range of the A/D converters. The filter reduces noise.

2.3 Proto-Board Testing

The circuit was tested in proto-board form before finalizing the printed circuit board design. This was the first hardware testing to validate the circuit models, and helped ensure proper component selection.

2.3.1 General Concerns for Proto-Board Testing

One primary goal of the proto-board testing phase was to test physical responses for the actual components intended for the final design. Printed circuit board layout, ordering, manufacturing, shipping, populating and testing take a considerable amount of time. A misplaced wire or wrong component can make this process take even longer. Proto-Board testing helps reduce the likelihood of these mistakes.

2.3.2 Fixture Development

The test bench fixture holds the sensors and wire for taking readings. The prototype-testing fixture as constructed to allow precision consistent movements for the sensors. The sensors are moved laterally with respect to the wire while maintaining a set elevation. The fixture was built to allow testing of different sensor elevations and different wire angles with respect to the sensors.

Figures 2-31 and 2-32 show the test fixture. The test fixture helped generate very consistent results. Consistent results are important in choosing the correct parameters in the signal conditioning circuit. Consistent results also help in debugging other problems from the rest of the smart sensor car system.

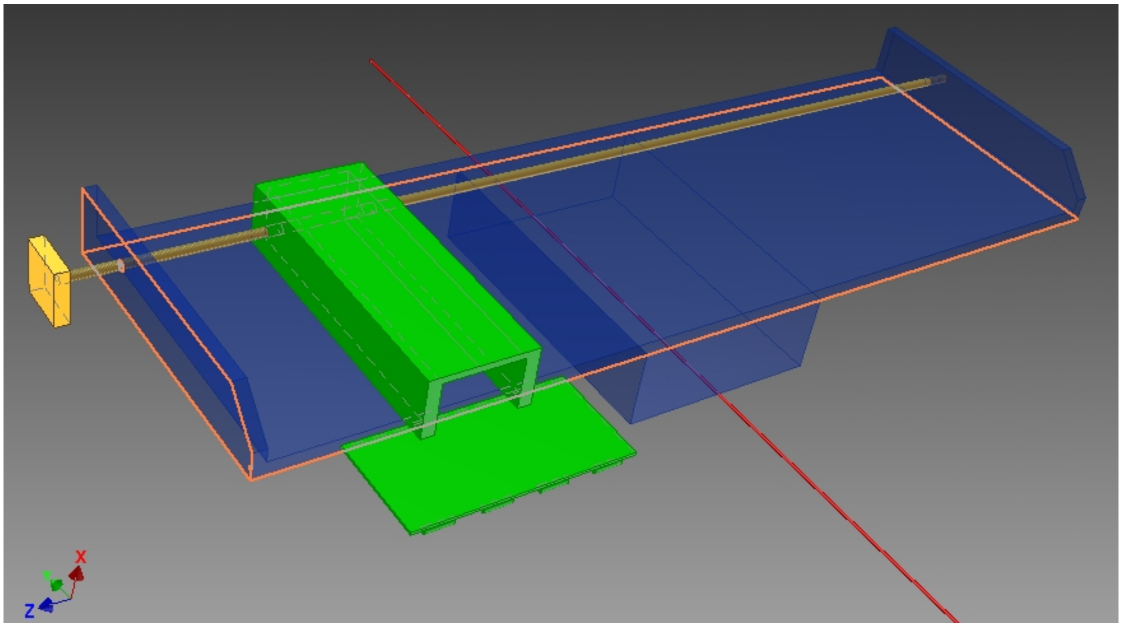


Figure 2-31: CAD Drawing of Test Fixture

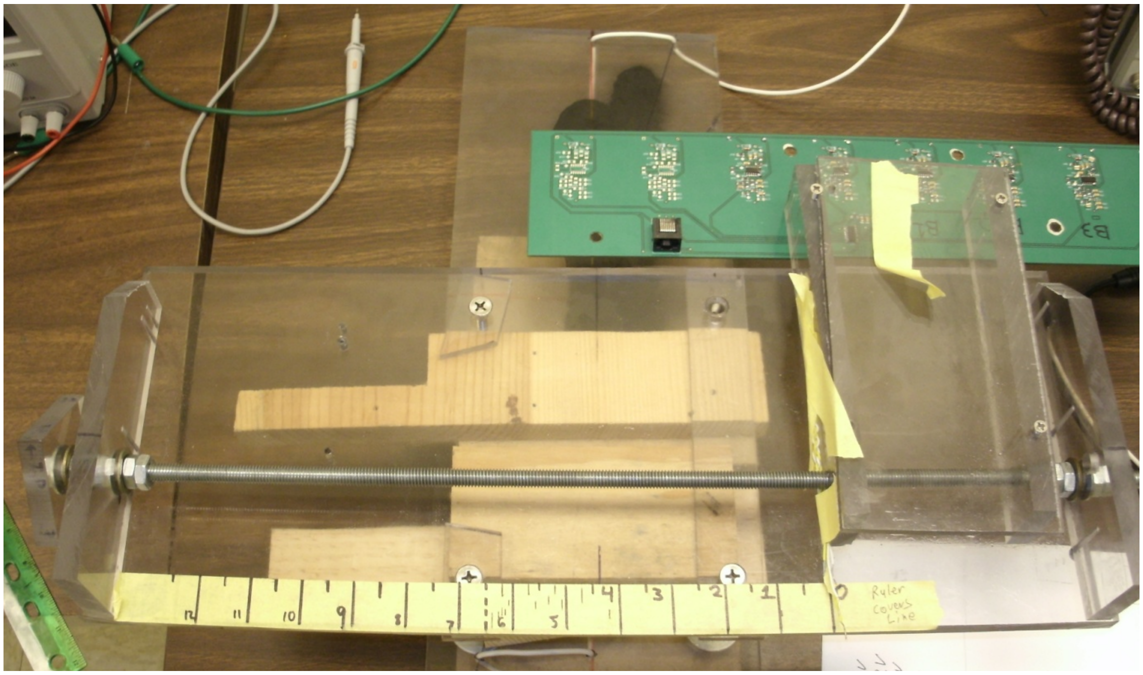


Figure 2-32: Photo of Test Fixture

2.3.3 Proto-Board Testing Results

The next important experiment helped select an appropriate op amp. The op amp is the basis of most components in the signal conditioning circuit. The 741 chip is the traditional op amp of choice. Slew rate is the amount the output voltage can change in set amount of time. The 741 op amp has a slew rate of about half a volt per microsecond. For some of the experiments, the 741 op amp slew rate was not sufficient, which resulted in degraded performance. Figure 2-33 shows the results of an insufficient slew rate. A 2-volt peak-to-peak 75 kHz sine wave is the input signal for a voltage follower circuit using a 741 op amp. The output of the voltage follower was a sharp saw toothed wave form. The output waveform is about four volts peak-to-peak. The saw tooth appearance is because the op amp is responding as fast as it can to the input, which is not fast enough to show the actual sine wave.

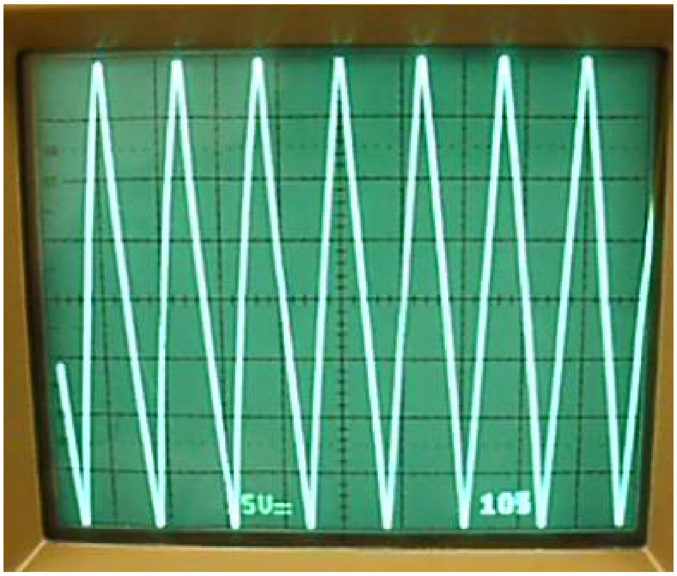


Figure 2-33: Scope Reading for Amplifier Circuit, gain of 2.99

Going to the device with a higher slew rate resolves this issue. The TL082 op amp has a slew rate of 16 volts per microsecond. Figure 2-34 is from an input waveform over ten volts peak-to-peak and has the same period. The voltage follower was implemented using a TL 082 op amp. Note the difference in the shape of the waveforms.

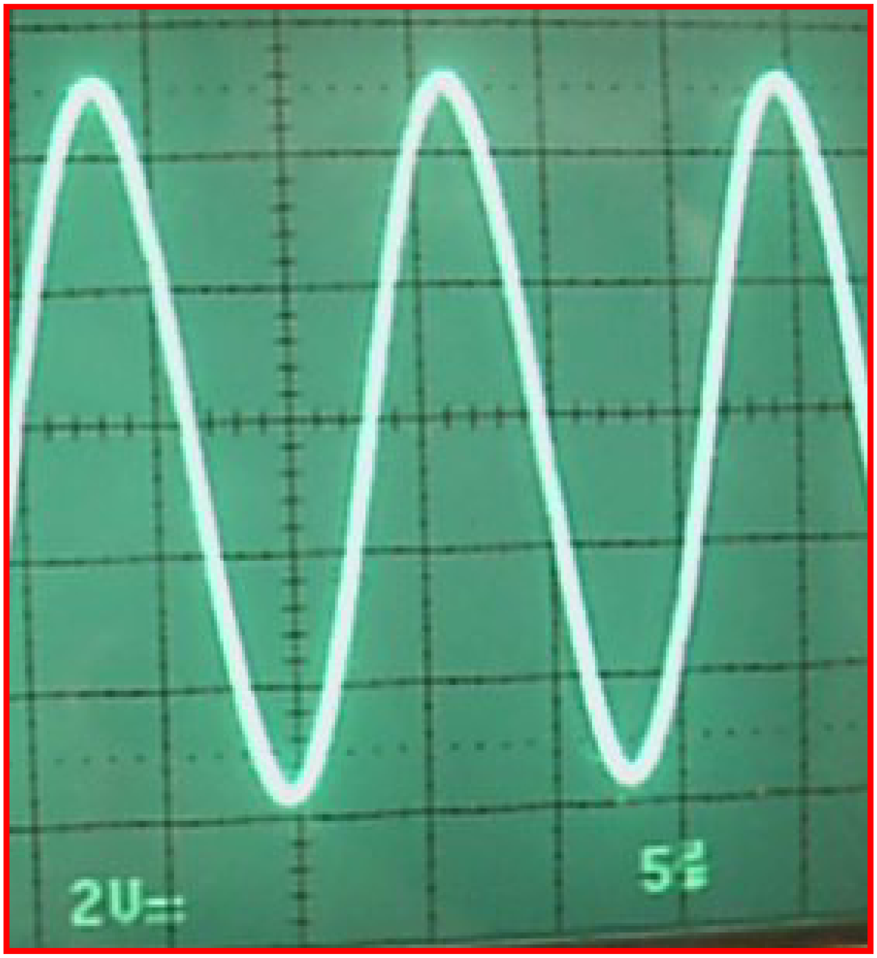


Figure 2-34: Output of TL082 Op-Amp in Voltage Follower Configurations

In order to accommodate the input of the first voltage follower and the output of the summing amplifier, both positive and negative rails are required for each op amp chip. This will require positive and negative voltage supplies for the sensor board.

2.4 Printed Circuit Board Development

Printed circuit boards increase circuit density, reduce noise and look professional. National Instruments' provides two programs to do circuit modeling and printed circuit board design. First, a circuit is laid out and simulated in Multisim, and then Ultiboard is used for printed circuit board design.

Printed wiring boards behave differently than simulations and protoboards. This can be due to the differences between surface mount and through hole components, as well as electromagnetic interference between traces on the printed wiring boards. After manufacture, testing must be performed to verify that design specifications are still met.

During prototype testing, a single signal conditioning circuit would cover an entire bread board. Using a printed circuit board allowed testing of four signal conditioning circuits and the multiplexer could be placed on a single small board. The A/D converter did not load the output of the signal conditioning circuit during bread board testing, but it did load the output of the multiplexer during printed wiring board testing. This was resolved by passing the multiplexer outputs through an off board voltage follower before going to the A/D converters. The second board design incorporated the voltage followers.

Initial car designs called for a separate board that would regulate the battery output down to the needed voltage for the sensor board. The first board design showed that there was enough un-used board space to incorporate the separate power board onto the sensor board. The second board design incorporated linear voltage regulators to power the sensor board and FPGA board, as they both required ground and five volts power. Testing of the second board showed that the FPGA current load caused excessive noise on the sensor board circuits. Large decoupling capacitors and decade decoupling capacitors were not successful in reducing the noise to an acceptable level. Isolating the FPGA power by installing a separate voltage regulator away from the sensor board did reduce the noise to an acceptable level. The third sensor board design only powered the sensor board from the on board regulators.

Figure 2-35 shows the sensor board layout. This image does not include the top and bottom ground planes. This shows the four identical circuits used to condition the output of four inductors. The output of each circuit is fed into a multiplexer.

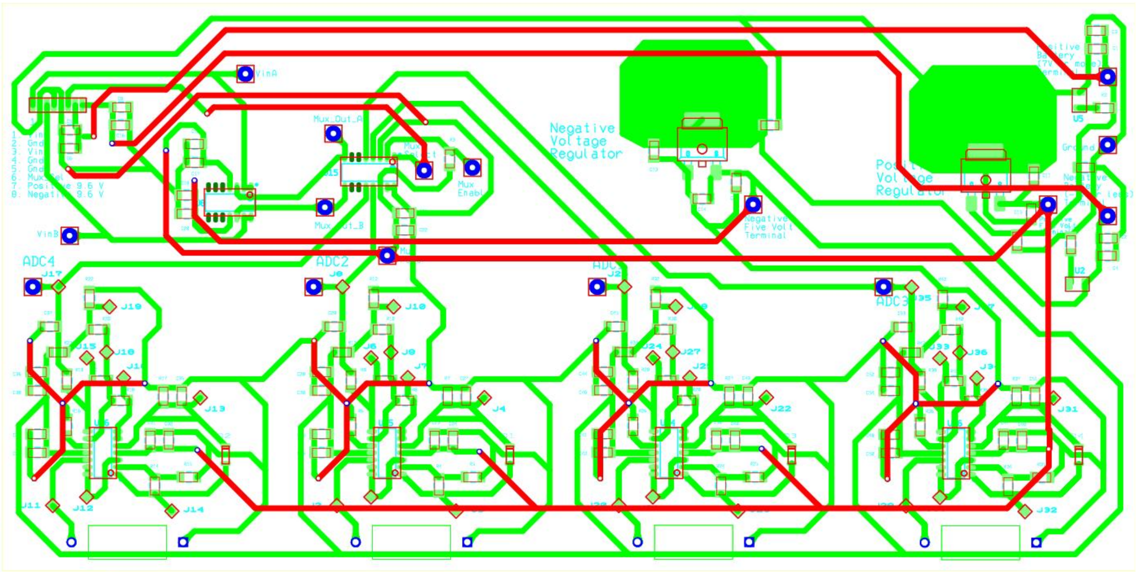


Figure 2-35: Sensor Board Layout

Figure 2-36 shows a photograph of the completed sensor board. The only connection to the printed circuit board is the RJ45 header in the upper left corner. Positive power, negative power, A/D grounds, the output signals and mux select signals go through the RJ45 connector.

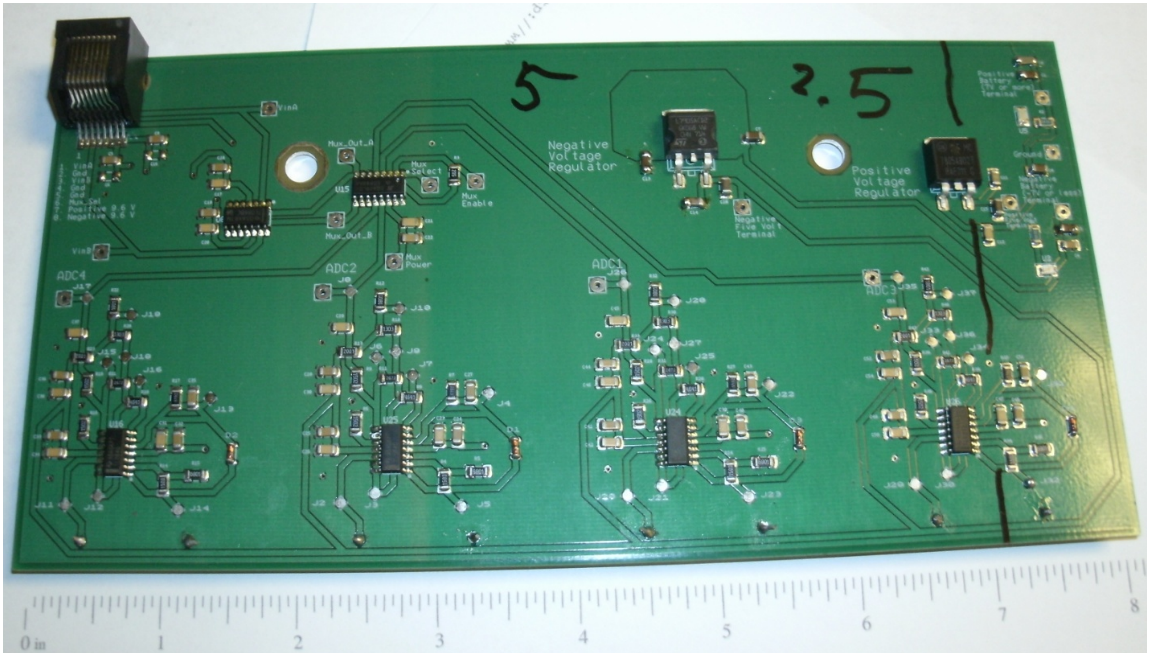


Figure 2-36: Sensor Board Photo

One way to ensure consistent results was to adjust the current flowing in the track wire to give a specific voltage from one of the circuit outputs. The right hand side of the sensor board has a black mark and the number 2.5, as shown in Figure 2.36. The black mark shows where the wire should pass under the board, and 2.5 is the voltage that circuit should output. Tuning the amplitude of the sine wave on the signal generator that drives the track wire so that the output of the signal conditioning circuit produces 2.5 volts produces consistent sensor board results.

Figure 2-37 shows a photograph of the completed sensor board with labels.

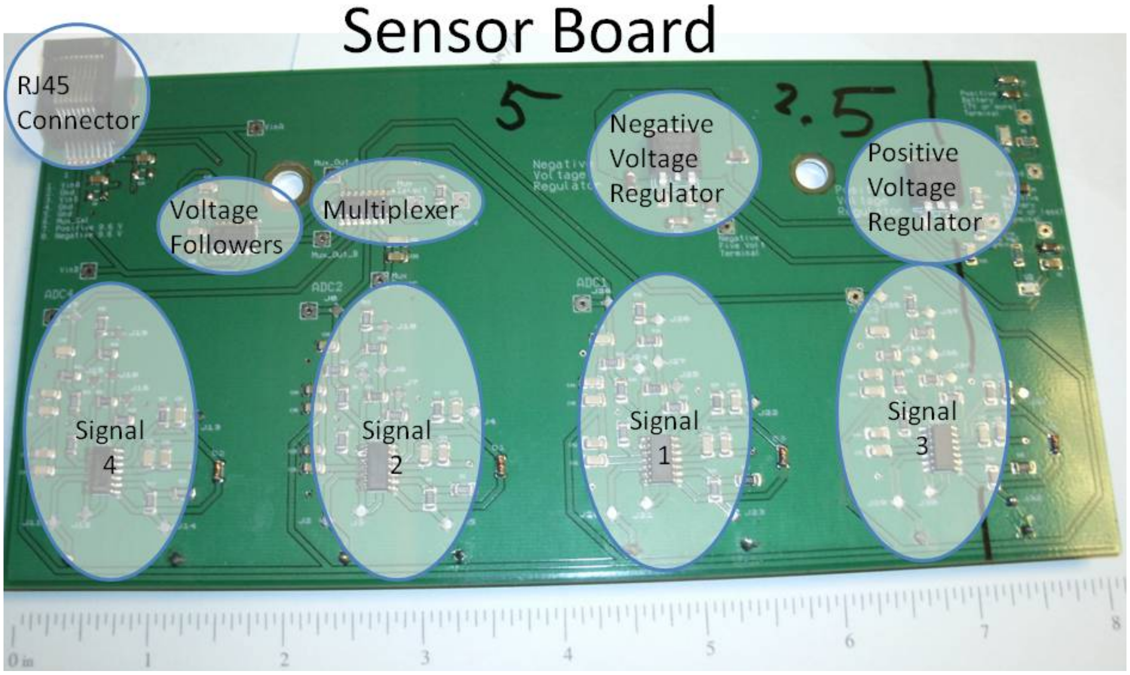


Figure 2-37: Labeled Sensor Board Photo

Figure 2-38 shows the ADC4 layout. This layout shows connections on top and bottom, as well as the silk screen text to help identify components. This layout design focused on allowing ample area between components to make soldering easier. This layout avoids traces making right angles, which could cause noise. The signal traces never pass from the top copper layer to the bottom copper layer. This helps reduce noise on the signal.

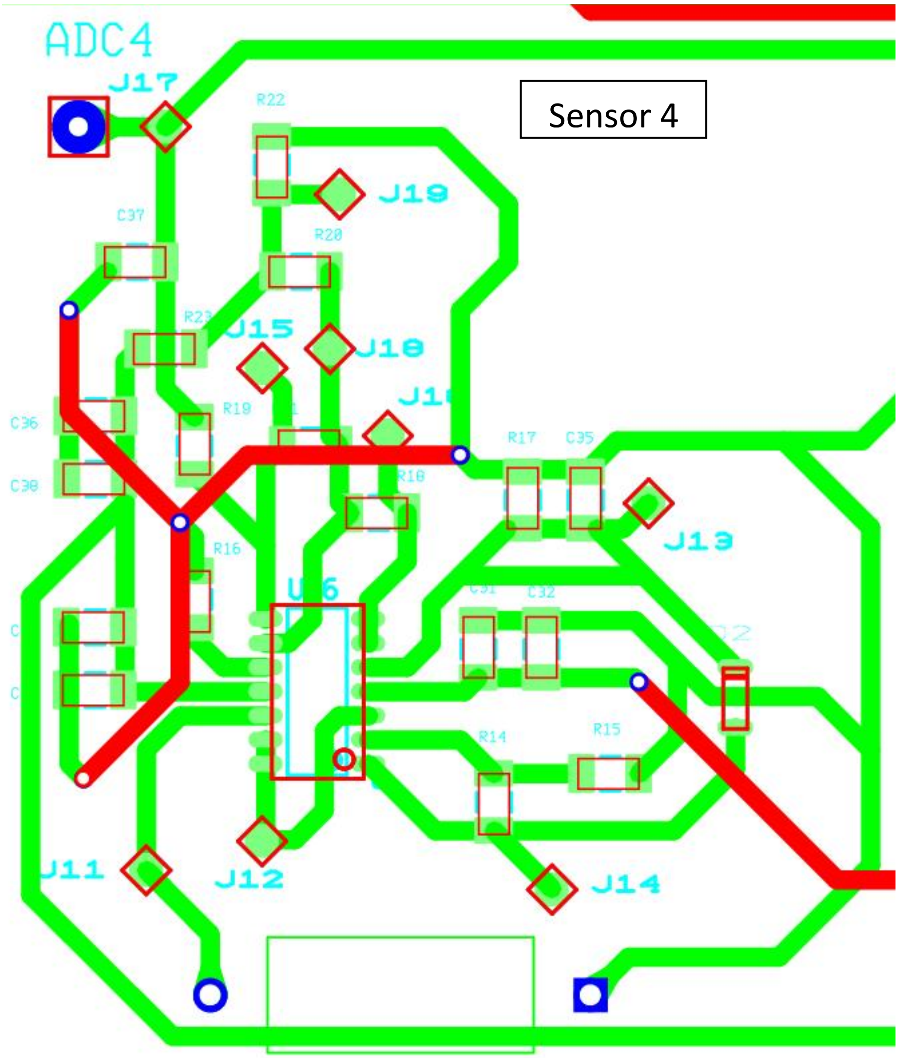


Figure 2-38: ADC4 Layout

Figure 2-39 shows where the individual components of the signal conditioning circuit fall onto the board layout.

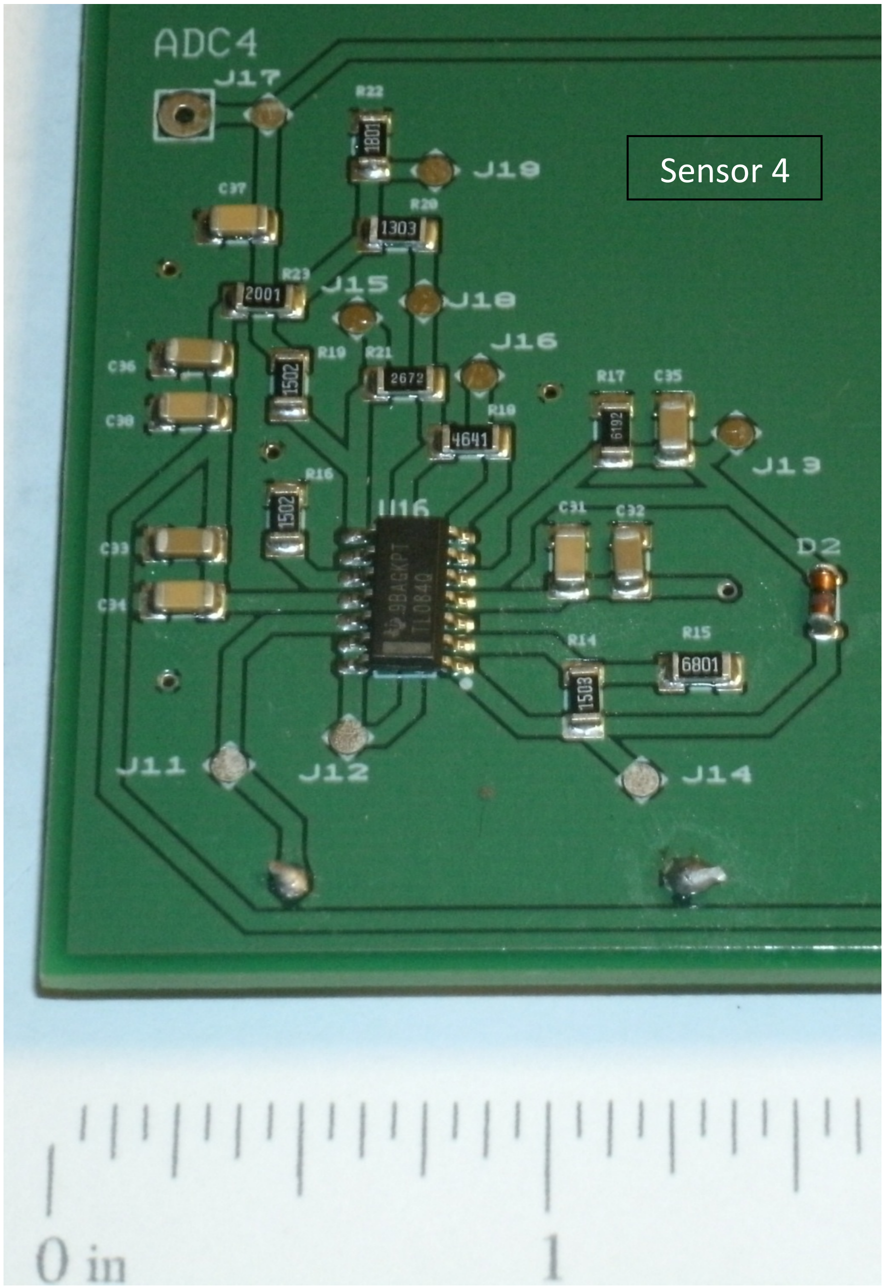


Figure 2-39: Sensor 4 Photo

Figure 2-40 shows the a photograph of where the individual components of the signal conditioning circuit fall onto the board layout with the components labeled.

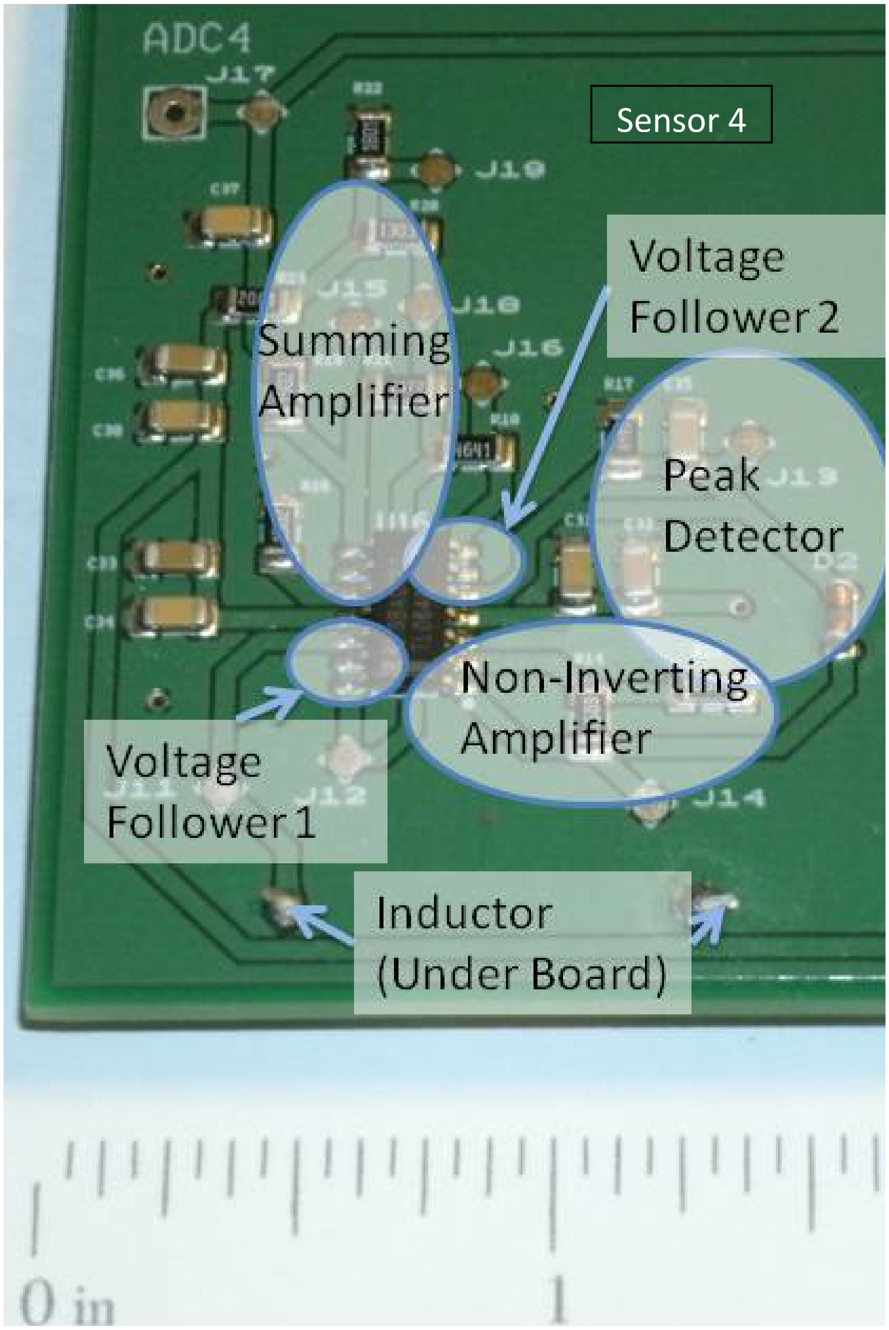


Figure 2-40: Sensor 4 Photo with Labels

2.5 Summary of Input Signal Capture Circuit

The sensors and signal conditioning circuit provide a path for information about the magnetic field to travel to the A/D converter. The sensors and components for the signal conditioning board are common and readily available from local vendors and can be used with prototyping boards. Inductors are a good choice for sensing a magnetic field because they are readily available, have been used for this application before, and their output can be measured as a voltage. Proto-board testing reduced the likelihood of problems during printed wiring board development. Printed wiring boards are the method of choice for the final implementation of the signal conditioning circuit. Printed wiring boards presented unique challenges there were not seen during earlier design phases.

CHAPTER III

ANALOG TO DIGITAL CONVERSION

As the Car moves along the magnetic field track, sensors detect the magnetic field. The sensors signals go through a signal conditioning circuit which prepares analog sensor readings for the Analog To Digital (A/D) converters on the FPGA board. The A/D converters change the analog signals into digital numbers for the neural network to use as inputs for the position calculation. The position calculation is fed to the PD controller, which generates steering angle and speed commands for the car so it can continue moving along the track. This chapter focuses on the A/D converter. Figure 3-1 shows the main block diagram.

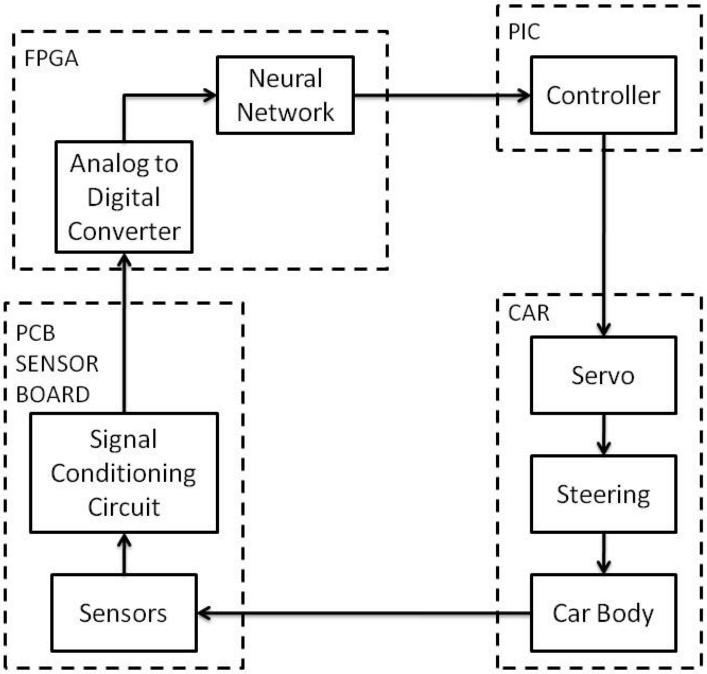


Figure 3-1: Main Block Diagram

Figure 3-2 shows how the A/D converter fits into the system functional overview.

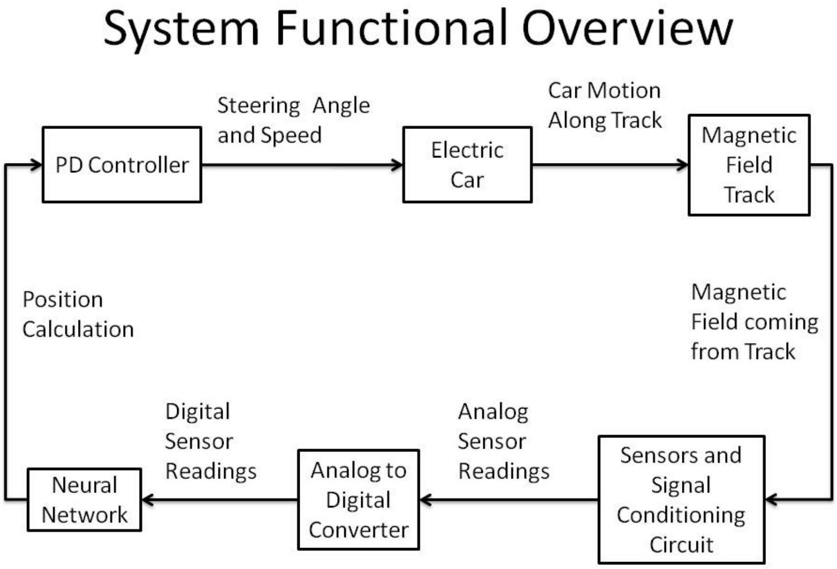


Figure 3-2: System Functional Overview

3.1 Introduction, Purpose and Overview of Analog to Digital Hardware and Software Interface

The A/D converter converts analog signals into digital numbers. The sensor board outputs analog signals. The neural network on the FPGA accepts digital numbers. The A/D conversion changes the available output of the sensor board into the acceptable inputs of the neural network. The rest of this chapter describes the hardware required for this process and the intelligence that controls it.

3.2 Hardware

Figure 3-3 shows the five major pieces of hardware required for the A/D conversion. The FPGA is the one piece of hardware that will hold all the intelligence to control everything else. The FPGA drives the multiplexer on the signal conditioning circuit with a single channel. The FPGA communicates with the pre-amplifier and A/D converter chip with a data bus. The signal conditioning circuit is the hardware that produces the input voltages which are key signals for the operation of the smart sensor car. This section describes the major pieces of the hardware.

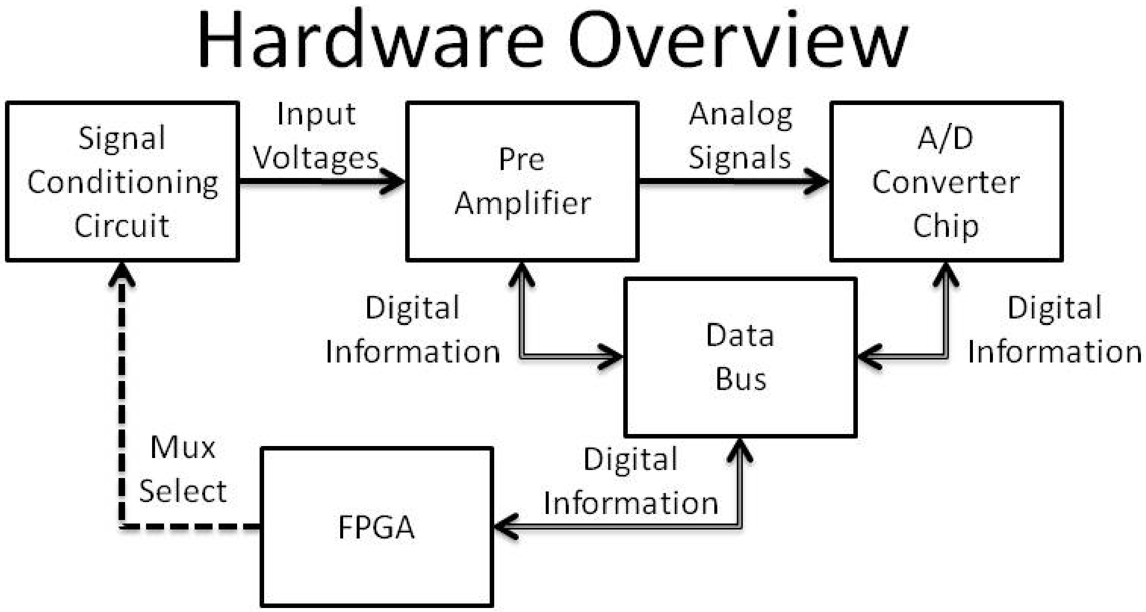


Figure 3-3: Hardware Overview

3.2.1 Input Voltage

The input voltages come from the signal conditioning circuit, pass through a multiplexer and go to the pre-amplifier. The input voltages tell the rest of the system about the magnetic field the sensors are experiencing. These voltages are translated into the car position and used to steer the car. Figure 3-4 shows the major components that produce the input voltages.

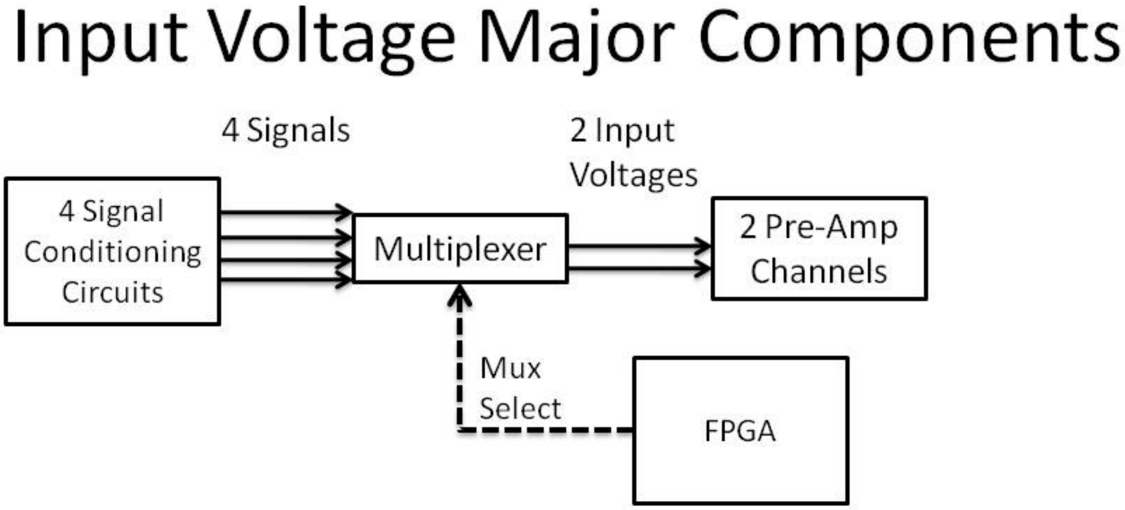


Figure 3-4: Input Voltage Major Components

Four signal conditioning circuits produced four signals. A multiplexer with two outputs allows the FPGA to choose which two signals are input voltages to the two pre-amp channels at any time. The FPGA chooses with the *MUX_SELECT* channel. Figure 3-5 shows the relationship between the pre-amplifier channels, input voltages and *MUX_SELECT* channel with a timing diagram.

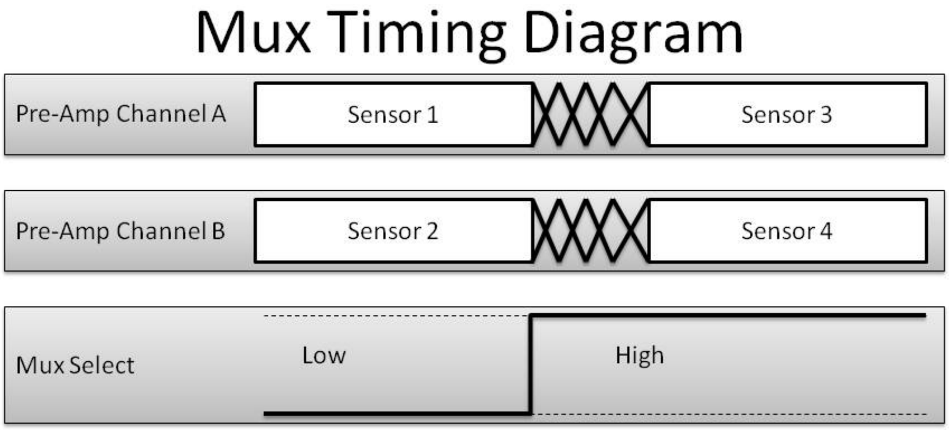


Figure 3-5: Mux Timing Diagram

Note that when *MUX_SELECT* goes from low to high, there is a bit of transition time before the input voltages have settled to sensors three and four.

3.2.2 Pre-Amplifier

Figure 3-6 shows how the pre-amplifier ties to the other hardware components. The LTC6912-1 programmable inverting pre-amplifier by Linear Technologies comes on the FPGA board. The purpose of the pre-amplifier is to ensure the input signals completely fill the input range of the A/D converter chip.

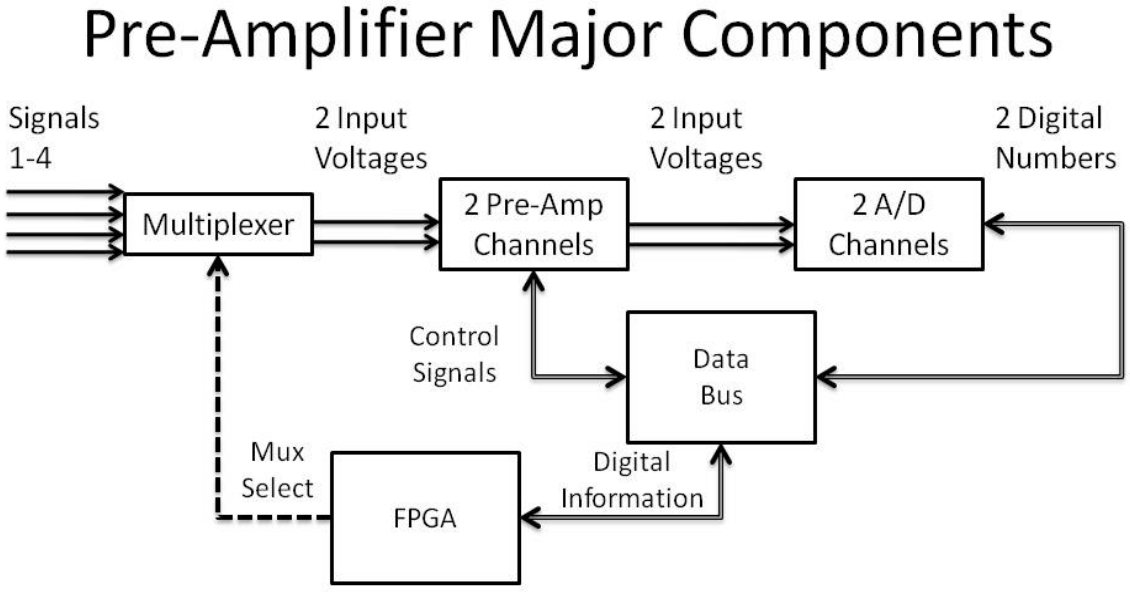


Figure 3-6: Pre-Amplifier Major Components

The pre-amplifier communicates with the FPGA through the data bus using five different channels. The inverting pre-amplifier has eight possible gain settings, ranging from 0 to -100. The A/D converter chip can only accept signals between -0.4 and -2.9 volts. The pre-amplifier is in place to magnify signals if needed. Excessive signals that exceed this range saturate the A/D converter. This is why increased gain is only needed for smaller input signals For the smart sensor car project, the input voltages are sufficiently large to fill the input range of the A/D chip, and a the gain of -1 is used. This means that the pre-amplifier inverts the voltage without changing the magnitude before passing the signals on to the A/D chip. By making the input voltages coming from the signal conditioning circuit as large as possible, effects of noise on the input signals are minimized.

3.2.3 Converter Chip

Figure 3-7 shows how the A/D converter chip connects with the other hardware components. The LTC1407A-1 dual channel A/D converter by Linear Technologies comes on the FPGA board. The A/D converter converts the two Input voltages into two 14 bit two's complement numbers. The A/D converter chip uses the data bus to communicate with the FPGA using three channels.

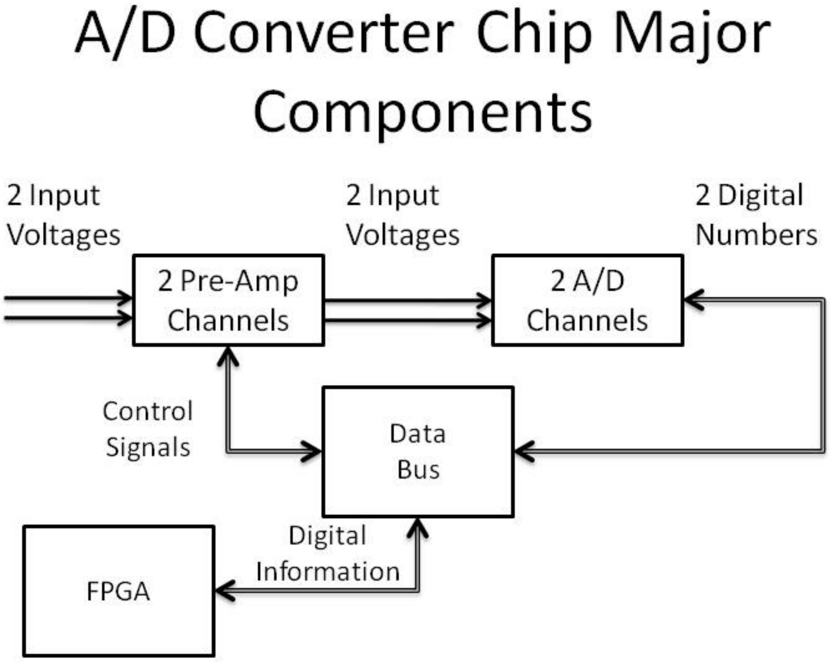


Figure 3-7: A/D Converter Chip Major Components

The A/D conversion takes time. The serial data transmission also takes time. When the A/D converter receives the command, it takes a sample of the current input voltages, then serially transmits the results of the previous data conversion. The current sample is converted while the previous sample is transmitted.

3.2.4 Data Bus

The FPGA uses the data bus to get information to and from the other devices. The FPGA has intelligence to control the other devices on the board. The FPGA has a limited number of pins it can use to communicate with other devices. The FPGA selects one device to communicate with and de-selects the remaining devices. Then the FPGA uses a shared set of communication lines to send information to all the devices, but only the selected device responds to the communication. The hardware used for device selection and communication is the data bus. The data bus has multiple channels. The FPGA uses chip select commands to identify which device it wants to work with, while turning off other devices. This allows the FPGA to use the same pins to communicate with a device without interference or contention from other devices. To further reduce the number of channels required for communication, serial communication is used when possible.

Figure 3-8 shows the data bus connections for the pre-amplifier. The pre-amplifier and FPGA communicate using the data bus using five different signals. The first four are inputs to the pre-amplifier, the last one is an output from the pre-amplifier. *Pre-amp enable* tells the pre-amplifier when to load new gains for two channels. *Master out slave in* tells the pre-amplifier what the gains for each channel should be. *Peripheral clock* provides the timing for the pre-amplifier to read in the gains. *Pre-amp shut down* is used shut down or reset the pre-amplifier. *Pre-amp gain echo* is the channel the pre-amplifier uses to echo back the gain settings to the FPGA. *Pre-amp shut down* and *pre-amp enable load gain* only have two states. *Pre-amp gain echo* is a serial transmission of eight bits. Pre-amplifier specific channels connect the pre-amplifier to the FPGA. The pre-amplifier specific channels are `pre-amp shut down`, `pre-amp enable load gain` and `pre-amp gain echo`. Signals that are shared with other devices include the `peripheral clock` and `master out slave in`. The `peripheral clock` is used by peripheral devices uses for timing their serial communications. `Master out slave in` is the channel the FPGA uses to serially communicate the data to peripheral devices.

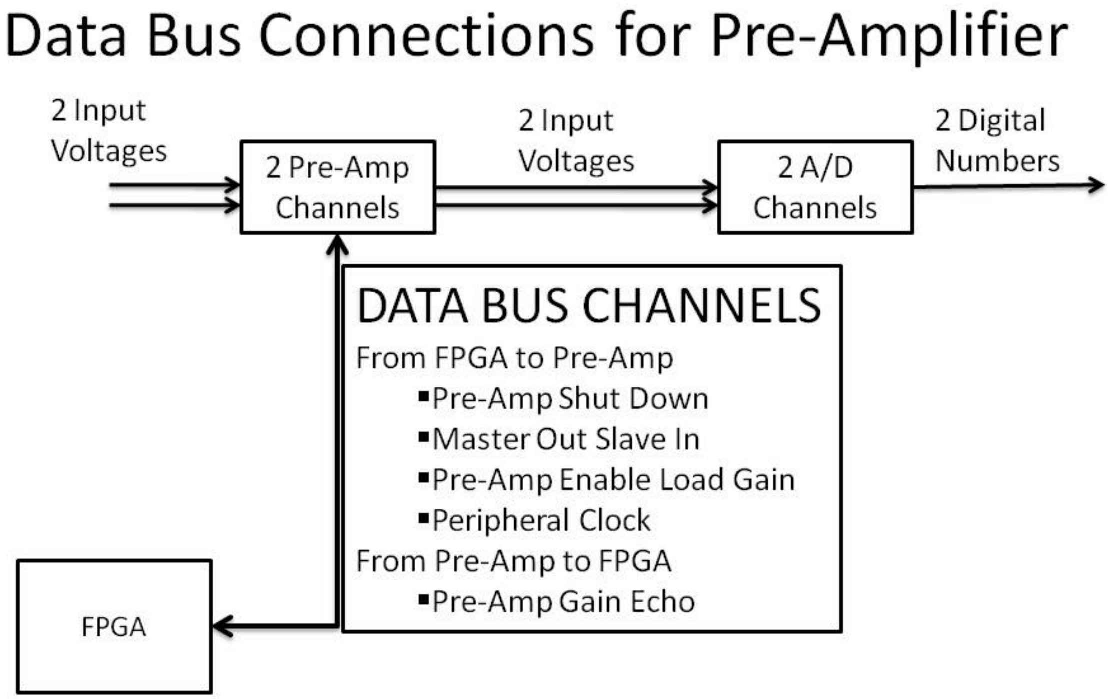


Figure 3-8: Data Bus Connections for Pre-Amplifier

Figure 3-9 shows the data bus connections used with the A/D converter. The A/D converter chip uses the data bus to communicate with the FPGA using three channels. The only A/D converter chip specific channel is `A2D converter start`, which tells the

A/D converter when to capture a sample and read off the previous conversion. The two shared channels are the `peripheral clock` and `master in slave out`. `Master in slave out` is the channel the FPGA uses to serially receive information from the peripheral devices. After the A/D chip receives the A/D converter start command, it captures a sample and reads off the previous conversion. The previous conversion consists of two 14 bit two's complement numbers. These numbers are transmitted one bit at a time from the A/D converter to the FPGA over the master in slave out channel on the data bus.

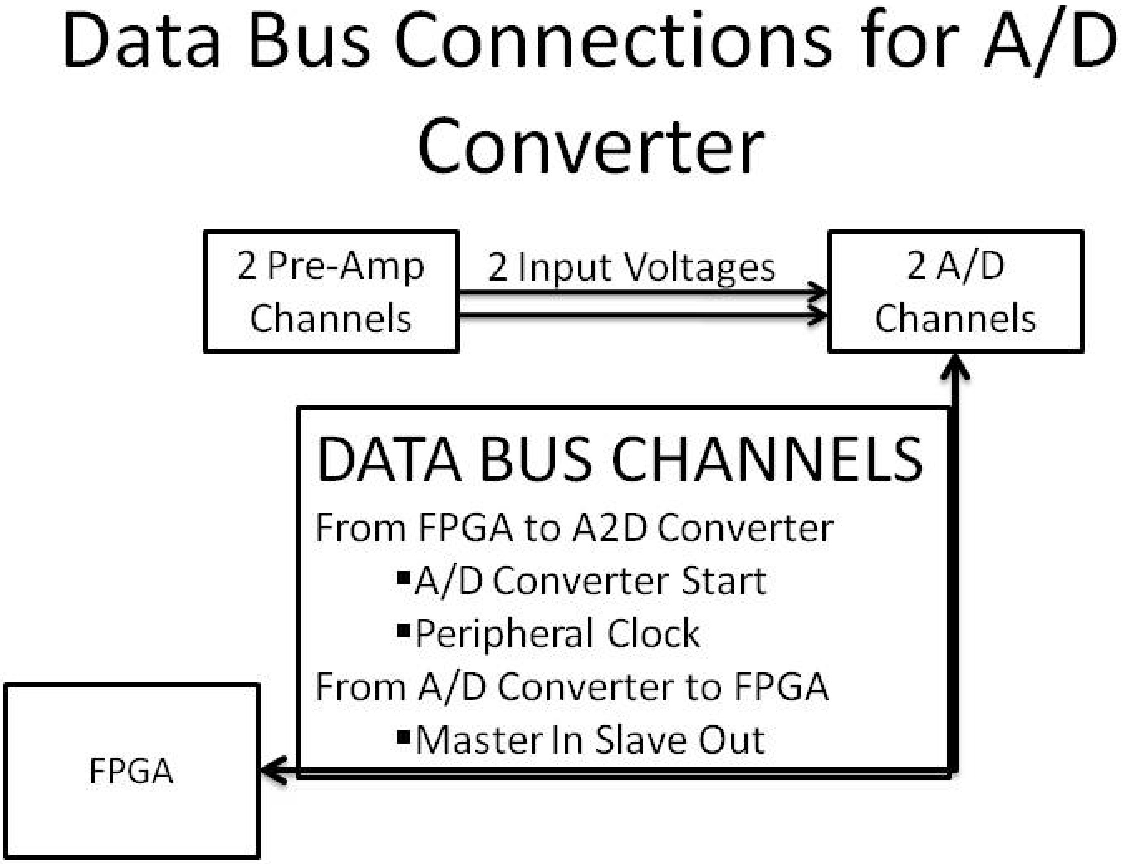


Figure 3-9: Data Bus Connections for A/D Converter

3.2.5 FPGA

Field Programmable Gate Arrays (FPGA) are re-programmable hardware. The FPGA board used for this project came with the Xilinx XC3S500E Spartan-3E FPGA chip. The FPGA has three main tasks. First, it controls the A/D conversion. Second it uses a neural network to calculate the car position. Third, it transmits the position to the PD controller. The FPGA was a good choice for this project for two reasons. First, it can do the needed tasks at processing speeds that allow stable control of the smart sensor car. Second, the FPGA circuitry is designed using code and can be quickly re-programmed. This helped reduce development time. Figure 3-10 shows the FPGA circuit design cycle.

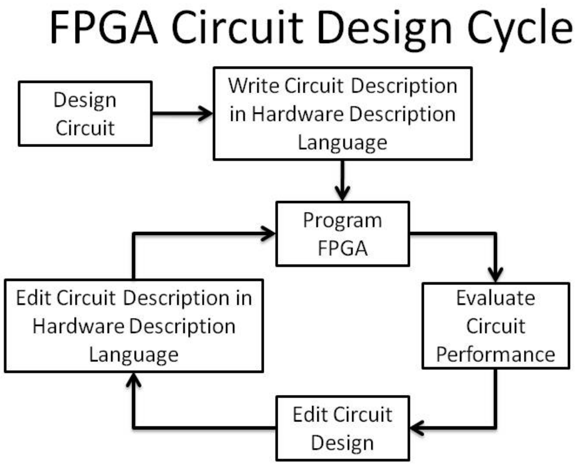

FPGA Circuit Design Cycle
Design Circuit
Write Circuit Description in Hardware Description Language
Program FPGA
Evaluate Circuit Performance
Edit Circuit Design
Edit Circuit Description in Hardware Description Language

Figure 3-10: FPGA Circuit Design Cycle

The time between finishing changes to the code and evaluating circuit performance can be as little as a few minutes. This is much preferred to manually changing circuit wiring by hand or sending away for a newly fabricated chip. This allowed the smart sensor car to be developed in time for use with the summer academy.

3.3 Finite State Machine

Figure 3-11 is the system functional overview that shows the required process to steer the car around the track. The PD controller gives the car the steering command. The PD controller needs the position calculation from the neural network to know how to steer the car. The neural network needs the digital sensor readings from the A/D converter to calculate the position. The A/D converter needs analog sensor signals from the signal conditioning circuit to concert to digital numbers. The signal conditioning circuit needs the voltages from the sensors to produce the analog signals. The sensors need the magnetic field from the track to produce a voltage. The magnetic field strength will depend on the motion of the car, which was determined by the previous steering command from the PD controller.

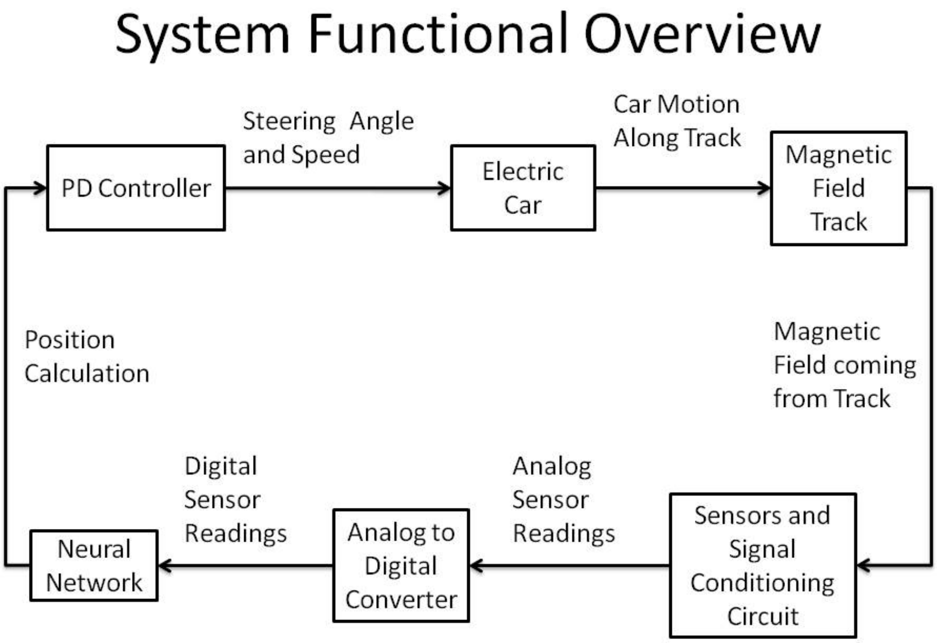


Figure 3-11: System Functional Overview

For the car to follow the wire, the process needs happen in a sequence. The PD Controller initiates the sequence by sending a request for a new position to the master Finite State Machine (FSM), then waits for a response. When the master FSM responds with a position, the PD controller calculates the steering command and requests another position. Figure 3-12 shows the PD controller process.

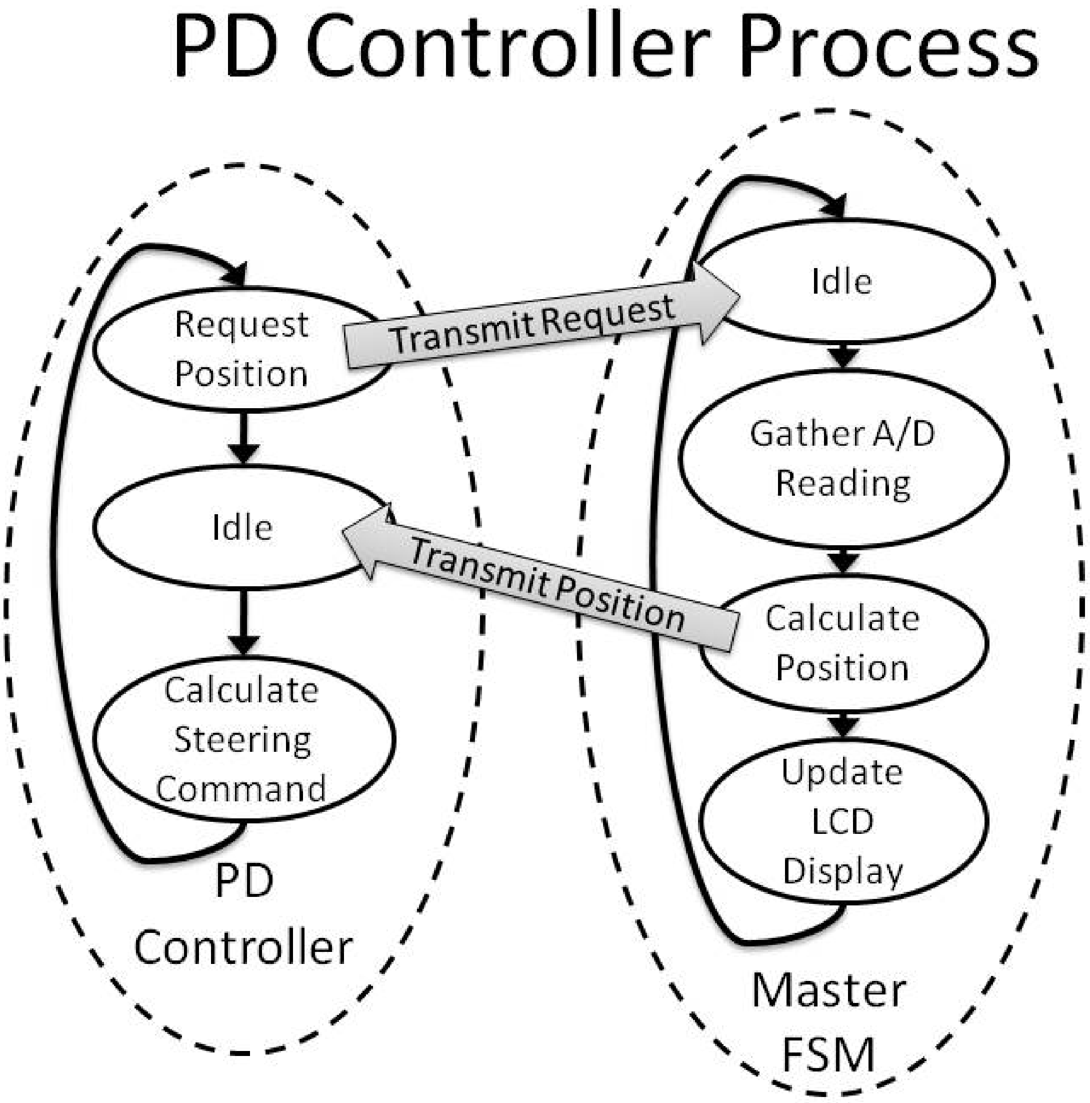


Figure 3-12: PD Controller Process

When the master FSM receives the request from the PD controller, it goes through a series of tasks. The first task is to command the A/D FSM to start a conversion. The master FSM waits for a response from the A/D FSM and then completes its other tasks. Figure 3-13 shows the master FSM process and gives an overview of the A/D FSM.

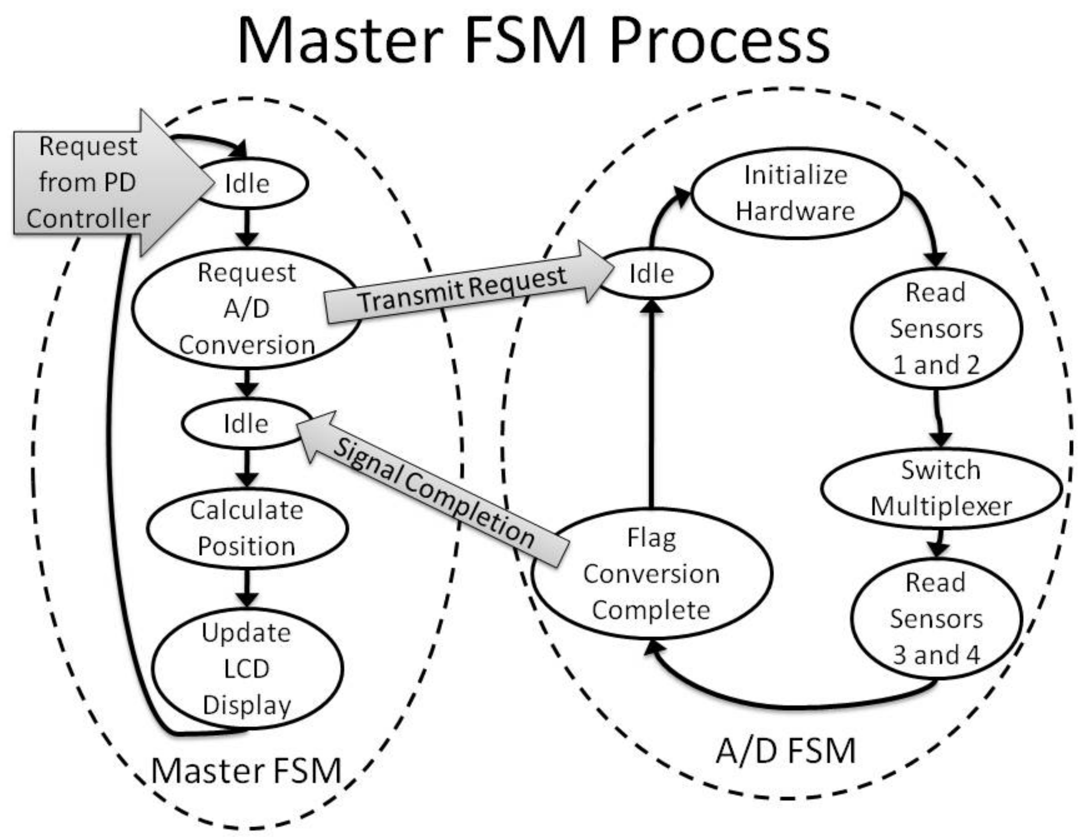


Figure 3-13: Master FSM Process Overview

After receiving the start command, the A/D FSM will initialize the hardware, take a reading from the first two sensors, switch the multiplexer, take a reading from the second two sensors, flag the master FSM that the conversion is complete and return to the state `IDLE`. The remainder of this section focuses on the A/D FSM.

3.3.1 Master FSM Initiates A/D Conversion

The code from the master FSM to initiate the A/D conversion is below. The code is written in VHDL which is a hardware description language. The *rs232_receiver_stb* bit is driven by the transmission from the PD controller and lets the master FSM know that the request transmission is complete. Two of the signals drive the A/D FSM. The first is *ce_amp* that lets the A/D FSM initialize the hardware. The second is *START_ADC* which allows the A/D FSM to read the four sensors. *ADC_DONE* comes from the A/D FSM and lets the master FSM know when the conversion is complete. The code for the master FSM is in Appendix A, and below is a copy of the first four states that start the A/D FSM.

```
//////////////////////////////////////////////////////////////////////
MASTER FINITE STATE MACHINE CODE START
begin
      when IDLE =>
            next_state <= WAIT_RECEIVE;

      when WAIT_RECEIVE => -- wait for RS232 data
            if (rs232_receiver_stb = '1') then -- data done
                  next_state <= START_ADC;
            else
                  next_state <= WAIT_RECEIVE;
            end if;

      when START_ADC =>
            ce_amp <= '1'; --active high
            start_conv <= '1';
            next_state <= ADC;

      when ADC =>
            if ADC_DONE = '1' then
                  ce_amp <= '0'; --active low
                  start_conv <= '0';
                  next_state <= ADC2FIXED;
            else
                  next_state <= ADC;
            end if;
MASTER FINITE STATE MACHINE CODE STOP
//////////////////////////////////////////////////////////////////////
```

Figure 3-14 shows the initial states of the master FSM. The master FSM will loop in the state `WAIT_RECEIVE` until the PD controller has completed its request. After that, it will pass through the state `START_ADC` which sends the command to the A/D FSM to begin the conversion. Then the master FSM will loop in the state `ADC` until the A/D FSM completes the conversion.

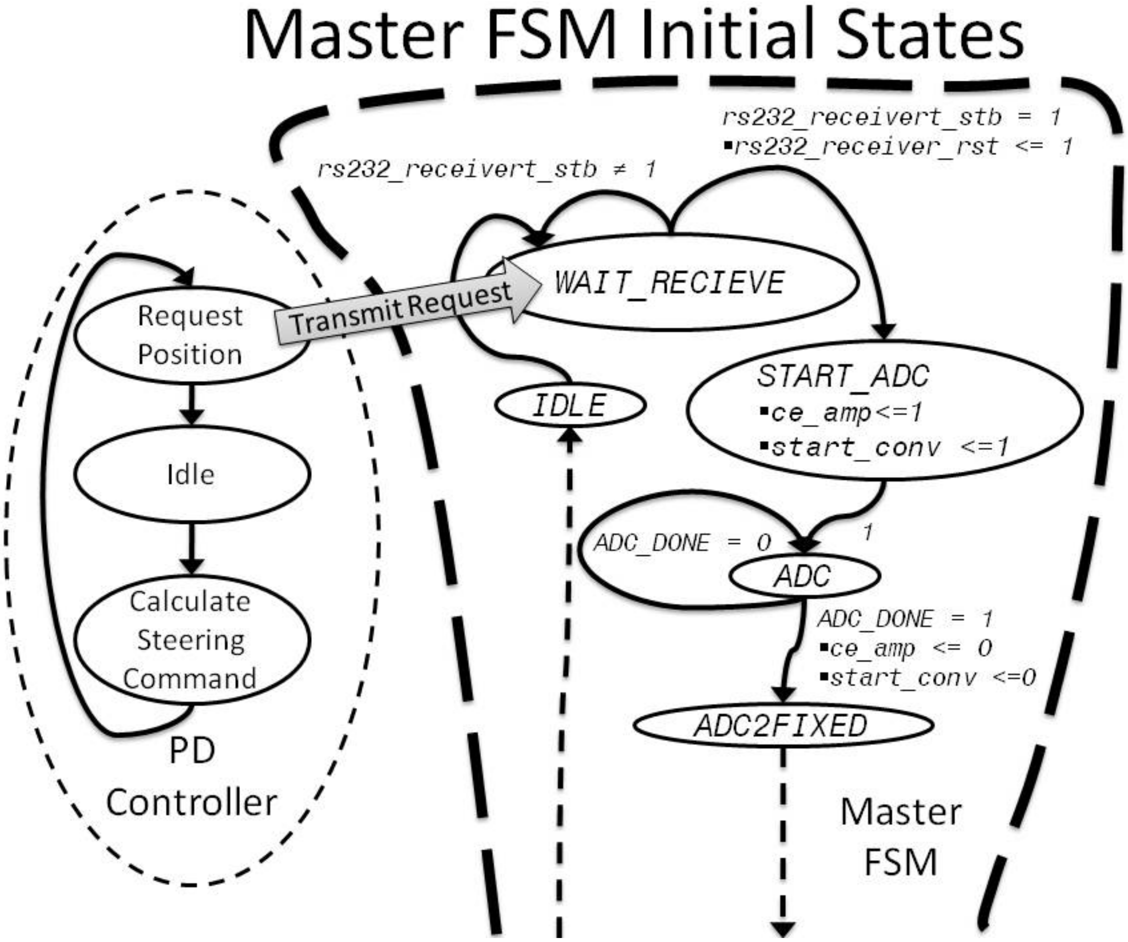


Figure 3-14: Master FSM Initial States

3.3.2 A/D FSM Initializes Hardware

Figure 3-13 shows that after the A/D FSM receives the start command from the master FSM, it leaves the state `IDLE`. Figure 3-6 shows the pieces of hardware the A/D FSM initializes. The multiplexer is set to read the first two sensors, the pre amp and converter chip are turned on. Next, the gain settings are loaded into the pre-amplifier. The variable `MUX_SELECT` is tied to the multiplexer address. `MUX_SELECT = 0` will read sensors 1 and 2. `AMP_CS` is tied to `pre-amp enable load gain` channel of the data bus. When `AMP_CS` is high, the pre-amplifier and A/D converter chips are sleeping and will not accept changes to the gain settings. `MOSI` is connected to the `master out slave in` channel of the data bus. `SCK` drives the `peripheral clock`. In this code, `pre-amp shut down and pre-amp gain echo` are not shown. `Pre-amp shut down` is set to a constant and `pre-amplifier gain echo` back to the FPGA is not recorded. Figure 3-15 is the A/D FSM state map corresponding to the code to initialize the hardware. The A/D FSM code is in Appendix B. The first nine states load the gains for the pre-amplifier.

```
//////////////////////////////////////////////////////////////////////
ANALOG TO DIGITAL CONVERTER FINITE STATE MACHINE CODE TO INITILIZE
HARDWARE START

begin
      when IDLE =>
            MUX_SELECT <= '0';
            AMP_CS <= '1';
            counter <=0;
            if ce_amp ='1' then
                  next_state <= START;
            else
                  next_state <= IDLE;
            end if;

      when START =>
            AMP_CS <= '0'; --turn amp on
            next_state <= START2;
            index1 <= 7; -- 8 bit value

      when START2 =>
            MOSI <= gain(index1);
            next_state <= HI;
            bit_count <= 0;

      when HI =>
            SCK <= '1';
            counter <= counter +1;
            if counter = 2 then
                  next_state <= HI_DUMMY;
            else
                  next_state <= HI;
            end if;

      when HI_DUMMY =>
            counter <=0;
            bit_count <= bit_count + 1;
            index1 <= index1-1;
            next_state <= LO;

      when LO =>
            SCK <= '0';
            counter <= counter +1;
            if counter = 2 then
                  next_state <= LO_DUMMY;
            else
                  next_state <= LO;
            end if;

      when LO_DUMMY =>
            counter <=0;
            if bit_count = 8 then
                  next_state <= FINISH;
            else
                  MOSI <= gain(index1);
                  next_state <= HI;
            end if;

      when FINISH =>
            next_state <= IDLE_AD;
            AMP_CS <='1';
            SCK <= '0';
            MOSI <= '0';

      when IDLE_AD =>
            if start_conv ='1' then
                  next_state <= START_AD;
            else
                  next_state <= IDLE_AD;
            end if;

ANALOG TO DIGITAL CONVERTER FINITE STATE MACHINE CODE TO INITILIZE
HARDWARE STOP
//////////////////////////////////////////////////////////////////////
```

3.3.2.1 Set Multiplexer to Read the First Two Sensors. When the device is powered on, it will begin in the *IDLE* state. In the *IDLE* state the multiplexer is set to read the first two sensors and the pre-amp and A/D converter chip are put to sleep with the line of code, *AMP_CS <= '1'*. This means that the pre-amplifier and A/D converter chip are dormant and will not respond to commands. The A/D FSM will loop in the state *IDLE* until *ce_amp* is set to 1 by the master FSM. Figure 3-14 shows that the master FSM sets *ce_amp* 1 in the state *START_ADC*. The dashed line on the left signifies that after all the other states in the A/D FSM are complete, the machine returns to *IDLE*.

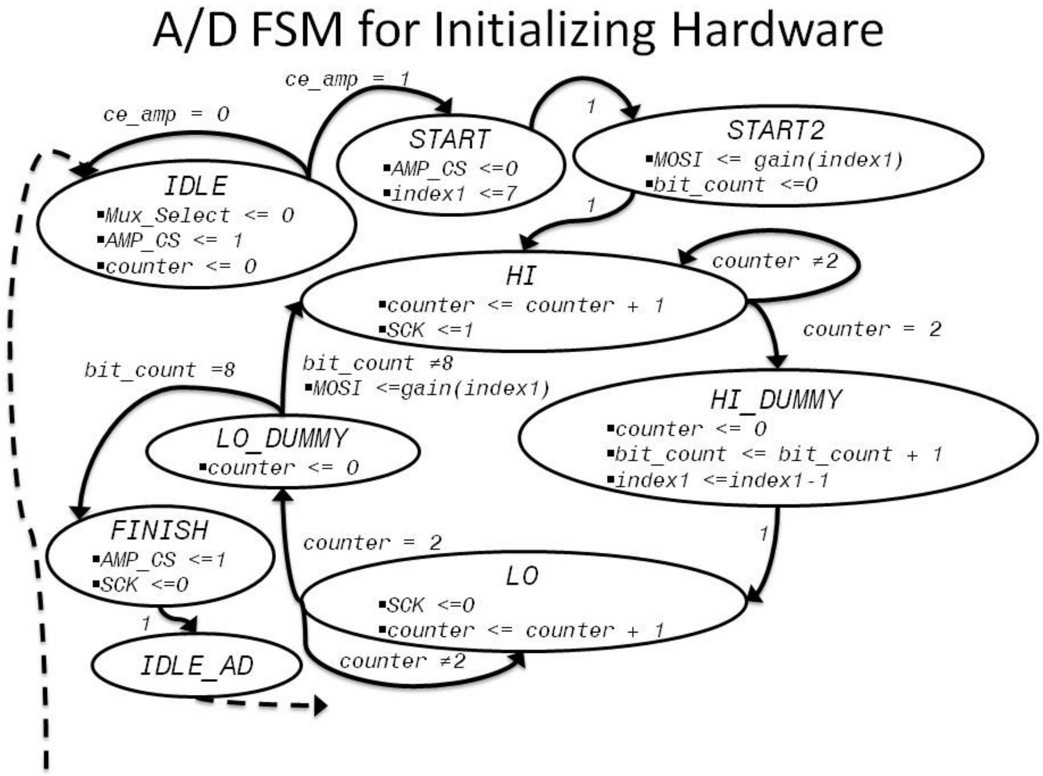


Figure 3-15: A/D FSM State Map for Initializing Hardware

3.3.2.2 Turn on Pre-Amp and Converter Chip. During the state *START*, the A/D FSM wakes up the pre-amplifier and A/D chip and with the command, *AMP_CS <= '1'*.

3.3.2.3 Load Gain Into Pre-Amp. Loading the gain into the pre-amplifier is a serial operation. This requires the A/D FSM to load the gain bits one at a time onto the *master out slave in* channel of the data bus, and then cycle the peripheral clock. The FPGA clock is too fast for the peripheral devices, so the A/D FSM goes through extra states to increase the peripheral clock period. The pre-amplifier gain is an eight bit variable. The gain transmission starts with the most significant bit. The states, *IDLE* and *START* are used in setting the multiplexer and turning on the pre-amplifier and A/D converter chip. These states are also used to initialize the two variables *counter*, *index1* and *bit_count*. These variables are integers and help control the timing for the serial communication between the FPGA and pre-amplifier.

The A/D FSM waits in the state *IDLE* until the master FSM sets *ce_amp* high. During *IDLE*, the multiplexer is set to read the first two signals, the pre-amplifier and A/D chip are put to sleep, and *counter* is reset to 0. After *ce_amp* goes high, the A/D FSM transitions to *START* where it wakes up the pre-amplifier and A/D chip and initializes *index1* to 7. The A/D FSM automatically goes to the next state, *START2*, where *MOSI* is set to the most significant bit of the gain. The variable *bit_count* is set to zero as well. The A/D FSM automatically goes to the next state, *HI*. In this state, it will set the peripheral clock high and loop back into itself until the variable *counter* has been indexed to two. Then the A/D FSM will transition to *HI_DUMMY* where *counter* is reset to 0, *bit_count* is incremented, and *index1* is decremented. The A/D FSM will automatically go to the next state *LO*. In *LO*, the peripheral clock is set low and the A/D FSM loops back into *LO* until counter is incremented to two. Next, the A/D FSM goes to the state *LO_DUMMY*, where *counter* is reset to 0. The variable *bit_count* drives A/D FSM to either set *MOSI* to the next bit of gain and make another cycle through the states *HI*, *HI_DUMMY*, *LO* and *LO_DUMMY*, or go to *FINISH*. If *bit_count* is eight, the A/D FSM will go to *FINISH*, put the pre-amplifier back into a dormant state and set the peripheral clock low. The A/D FSM will automatically go from *FINISH* to *IDLE_AD*.

3.3.3 Read Sensors 1 and 2

After the A/D FSM has initialized the hardware, it passes through the state *FINISH* and waits in the state *IDLE_AD* for the master FSM to set the variable *start_conv* high. Figure 3-14 shows that this happens in the master FSM state *START_ADC*, so the A/D FSM can continue. Figure 3-13 shows an overview of the A/D FSM. The code for the remainder of the A/D process makes a large loop. That loop includes steps to read the first two sensors, switch the multiplexer, read the second two sensors and set the flag *ADC_DONE* high. The variable *MUX_SELECT* shows if the A/D conversion is recording the first or second pair of sensors. Explanations of the code are in the following sub sections. As mentioned in section 3.2.3, the A/D chip will transmit the previous reading while it converts the current sample. The variable *prime* is high if the current transmission being clocked in is old data or low if the desired reading is being recorded. Figure 3-16 is an overview of the relationship of *prime*, *MUX_Select* and sensor readings.

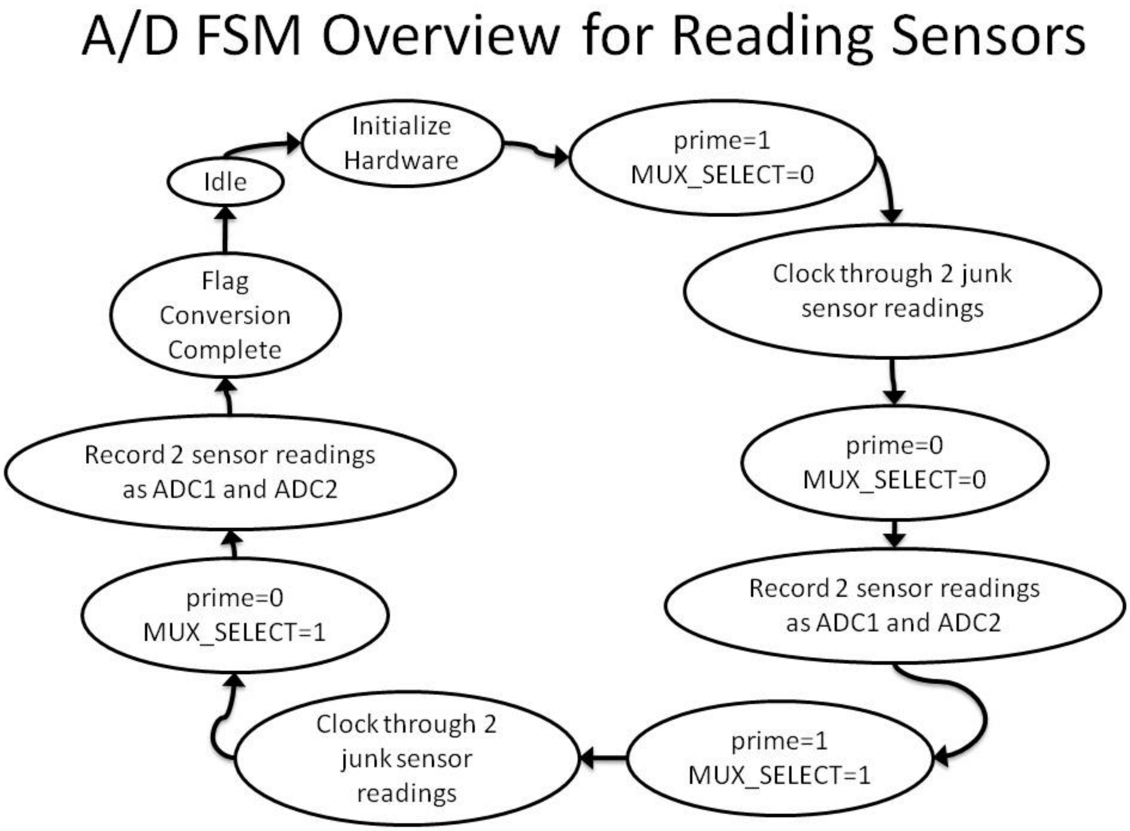

A/D FSM Overview for Reading Sensors
Idle
Initialize Hardware
prime=1 MUX_SELECT=0
Clock through 2 junk sensor readings
prime=0 MUX_SELECT=0
Record 2 sensor readings as ADC1 and ADC2
prime=1 MUX_SELECT=1
Clock through 2 junk sensor readings
prime=0 MUX_SELECT=1
Record 2 sensor readings as ADC1 and ADC2
Flag Conversion Complete

Figure 3-16: A/D FSM Overview for Reading Sensors

The states next eight states in the A/D FSM complete the tasks listed in Figure 3-16. Figures 3-17 through 3-19 show corresponding the state transition maps.

```
//////////////////////////////////////////////////////////////////////
ANALOG TO DIGITAL CONVERTER FINITE STATE MACHINE CODE TO READ SENSORS
AND SIGNAL COMPLETION START

when FINISH =>
      next_state <= IDLE_AD;
      AMP_CS <='1';
      SCK <= '0';
      MOSI <= '0';

when IDLE_AD =>
      if start_conv ='1' then
            next_state <= START_AD;
      else
            next_state <= IDLE_AD;
      end if;
      CONV <= '0';
      prime <= '1';
      MUX_SELECT <= '0';

when START_AD =>
      SCK <= '0';
      CONV <= '1';
      counter <= -1;
      index1 <= 13; -- 14 bit value
      index2 <= 13; -- 14 bit value
      next_state <= HI_AD;

when HI_AD =>
      SCK <= '1';
      CONV <= '0';
      counter <= counter +1;
      next_state <= LO_AD;

when LO_AD =>
      SCK <= '0';
      if prime = '0'  and mux_sel = '0' then    -- REAL ADC data
            if(counter > 2 and counter < 17) then
                  ADC1(index1)  <= SPI_MISO;
                  index1 <= index1 -1;
            elsif(counter > 18 and counter < 33) then
                  ADC2(index2)  <= SPI_MISO;
                  index2 <= index2 -1;
            end if;

            if counter = 34 then -- DONE
                  next_state <= FINISH_AD;
            else
                  next_state <= HI_AD;
            end if;

      elsif prime = '1' and mux_sel = '0' then    -- prime = 1
            if counter = 34 then -- done priming
                  prime <= '0';
                  next_state <= START_AD; -- start real data
            else
                  next_state <= HI_AD;
            end if;

      elsif prime = '0' and mux_sel = '1' then        -- REAL ADC data
            if(counter > 2 and counter < 17) then
                  ADC3(index1)  <= SPI_MISO;
                  index1 <= index1 -1;
            elsif(counter > 18 and counter < 33) then
                  ADC4(index2)  <= SPI_MISO;
                  index2 <= index2 -1;
            end if;

            if counter = 34 then -- DONE
                  ADC_DONE <= '1';
                  next_state <= FINISH_AD;
            else
                  next_state <= HI_AD;
            end if;

      elsif prime = '1' and mux_sel = '1' then    -- prime = 1
            if counter = 34 then -- done priming
                  prime <= '0';
                  next_state <= START_AD; -- start real data
            else
                  next_state <= HI_AD;
            end if;
      end if;

when FINISH_AD =>
      counter <= 0;
      MUX_counter <= 0;
      SCK <= '0';
      CONV <= '0';
      if MUX_SELECT = '0' then
            prime <= '1';
            MUX_SELECT <='1';
            next_state <= MUX_SWITCH_1;
      else
            next_state <= IDLE;
      end if;

when MUX_SWITCH_1 =>
      MUX_counter <= MUX_counter + 1;
      next_state <= MUX_SWITCH_2;

when MUX_SWITCH_2 =>
      if MUX_counter = 100 then
            next_state <= START_AD;
      else
            next_state <= MUX_SWITCH_1;
      end if;

when others =>
      MUX_SELECT <= '0';
      mux_sel <= '0';
      SCK <= '0';
      CONV <= '0';
      AMP_CS <= '1';
      MOSI <='0';
      next_state <= IDLE;
end case;

end process;

ANALOG TO DIGITAL CONVERTER FINITE STATE MACHINE CODE TO READ SENSORS
AND SIGNAL COMPLETION STOP
//////////////////////////////////////////////////////////////////////
```

Figure 3-16 gives an overview of the A/D FSM state maps. Figures 3-17 and 3-18 show the state maps corresponding to the A/D FSM code for input voltage conversion. The key state is `LO_AD`. In `LO_AD`, the A/D FSM will decide if the incoming data should be recorded and how it should be stored.

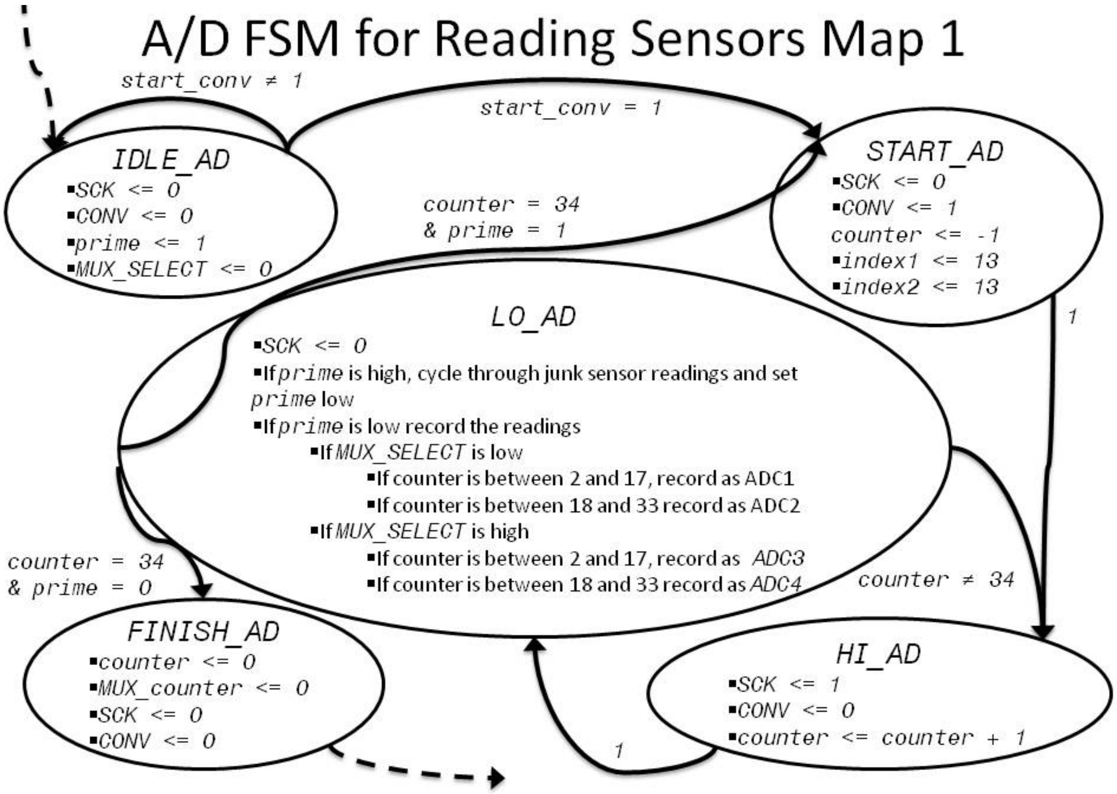


Figure 3-17: A/D FSM for Reading Sensors Map 1

<u>3.3.3.1 Cycle Through one set of Old Conversions.</u> Section 3.3.2.3 concluded after the A/D FSM passed through the state `FINISH` and into the state `IDLE_AD`. The A/D FSM begins the process of the data conversion when it leaves that state `IDLE_AD`. This happens when the variable `conv` is high. The master FSM sets this variable high in the state `START_ADC`.

In the A/D FSM state `IDLE_AD`, the peripheral clock is set low, the variable `prime` is set high, the multiplexer is set to read the first two sensors, and the variable `CONV` is set low. `CONV` is connected to the data bus channel `A2D Converter Start`. When it is low, the A/D chip will transmit its previous conversion, changing bits in time with the peripheral clock. When `CONV` is set high, the A/D chip captures a new sample and will begin transmitting the previous sample conversion.

After the A/D FSM leaves `IDLE_AD`, it goes to the state `START_AD`. In `START_AD`, the peripheral clock stays low, the variable `CONV` goes high, causing the A/D chip to take a new sample. The integer variables `counter`, `index1` and `index2` are initialized. Reading two samples from the A/D chip requires 34 peripheral clock cycles. The state `HI_AD` will index the variable `counter`, every time the A/D FSM enters that state. Because the peripheral clock has not been cycled yet, `counter` is set to -1 in the state `START_AD` so that it will be 0 after leaving `HI_AD` the first time and go to 1 only after the peripheral clock has been cycled once. The A/D chip generates a pair of 14 bit two's complement numbers, one for each sample. When appropriate, these samples will be stored as one of four `ADC` variables. The ADC variables store the 13th bit first and bit 0 last.This is why `index1` and `index2` are set to 13.

After the A/D FSM leaves `START_AD` it automatically goes to `HI_AD`. In `HI_AD`, the peripheral clock is set high, the variable `CONV` is set low and the variable `counter` is incremented.

The A/D FSM enters the state *LO_AD* automatically after being in the state *HI_AD*. In *LO_AD*, the peripheral clock is set low. During the first pass through *LO_AD*, the variable *prime* is high and *MUX_SELECT* is low, and the variable *counter* will be 0. The logic will then put the A/D FSM back into the state *HI_AD*, where the peripheral clock will be set high again and the variable *counter* will be incremented to 1, signifying that the peripheral clock cycled once. The A/D FSM will cycle between *LO_AD* and *HI_AD* until *counter* has incremented to 34. At that time, the variable *prime* will be set low and the A/D FSM will go back to the state *START_AD* and where it begins the cycle to record a good set of conversions from sensors 1 and 2.

3.3.3.2 Cycle and Record Good Set of Conversions. After cycling through the first set of readings, the A/D FSM is ready to record the first two sensor readings as the two 14 bit variables *ADC1* and *ADC2*. This process starts in the state *START_AD*, where the peripheral clock is set low, the variables *counter*, *index1* and *index2* are reset and the variable *CONV* commands the A/D chip to take a sample and transmit the previous conversion.

The A/D FSM will automatically go from *START_AD* to *HI_AD*. In *HI_AD*, the peripheral clock is set high, *counter* is incremented and the variable *CONV* is set low. The A/D FSM will automatically go from *HI_AD* to *LO_AD*.

In *LO_AD*, the peripheral clock is set low. The previous section explains that the last cycle through *LO_AD* sets the variable *prime* to 0. Because *prime* is now 0, the state *LO_AD* will complete a different process than before, and record the two sensor readings coming from the A/D chip. The sample being transmitted now was collected the first time the A/D FSM was in the state *START_AD*, when *MUX_SELECT* was set to 0. The timing diagram for the A/D chip is available from Linear Technologies. For the first pass through *LO_AD* for this cycle, *counter* is not yet greater than 2, so the A/D FSM will go back and fourth between *LO_AD* and *HI_AD* to run the peripheral clock until the variable *counter* has been indexed to 3. At that time, when the A/D FSM is in *LO_AD*, the variable *ADC1* will store the information from the data bus channel Master In Slave Out in its 13th bit. When the A/D FSM leaves *LO_AD*, the variable *index1* is decremented and the FSM goes to *HI_AD*. This process is completed until all 14 bits of *ADC1* have been recorded. At that point, *index1* is 0, *counter* is 16, and *index2* is 13. The peripheral clock is run by looping between *LO_AD* and *HI_AD* until *counter* has been incremented to 19. At this point, the variable *ADC2* will store the information from the data bus channel *Master In Slave Out* in its 13th bit. When the A/D FSM leaves *LO_AD*, the variable of *index2* is decremented and the FSM goes to *HI_AD*. This process repeats until all 14 bits of *ADC2* have been recorded. At this point, *index1* and *index2* are both 0 and *counter* is 32. The peripheral clock is run some more by looping between *HI_AD* and *LO_AD* until *counter* reaches 34, at which time the A/D FSM goes to the state *FINISH_AD*. This completes the cycle to record the first two sensors as digital numbers.

3.3.4 Switch Multiplexer to Read Sensors 3 and 4

After the first two sensor readings are recorded, the A/D FSM is ready to change the multiplexer to read and convert the second two sensors. The A/D FSM was in the state `FINISH_AD` at the end of the last process. Figure 3-18 shows how the A/D FSM changes the multiplexer and resets the variable `prime`.

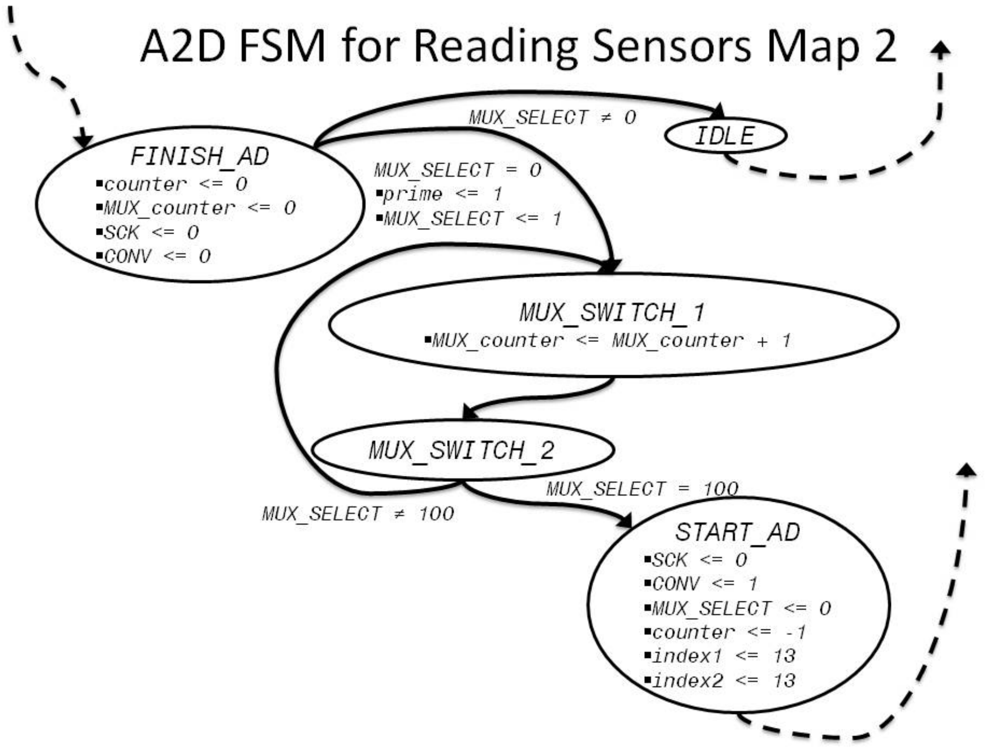


Figure 3-18: A/D FSM for Reading Sensors Map 2

This process begins when the A/D FSM is in the state `FINISH_AD` and the multiplexer is set to read the first two sensors. In the state `FINISH_AD`, the variables `counter` and `MUX_counter` are set to 0. The peripheral clock and the variable `CONV` are set low. Because *MUX_SELECT* is 0, the variable *prime* will be set to 1 and *MUX_SELECT* will be set to 1. This causes the multiplexer to switch the input voltages from the first two sensors to the second two sensors. The variables *prime* and *MUX_SELECT* are changed during the A/D FSM state transition from *FINISH_AD* to *MUX_SWITCH_1*. In the state *MUX_SWITCH_1*, the variable *MUX_counter* is incremented. The A/D FSM will automatically transition to the state *MUX_SWITCH_2*. The A/D FSM will go back to *MUX_SWITCH_1* and increment the variable *MUX_counter* until it reaches 100. This was done to allow enough delay between switching the multiplexer and sampling the channels for the input voltages to settle. After *MUX_counter* has reached 100, the A/D FSM will go back to the state *START_AD*. This time though, the multiplexer is set to read the second two voltages. This completes the process to switch the multiplexer to read the second set of sensors.

3.3.5 Read Sensors 3 and 4

Section 3.3.3 describes how the A/D FSM records the first two sensor readings as `ADC1` and `ADC2`. Section 3.3.4 describes the process to switch the multiplexer and restart the A/D FSM. This section describes how the A/D FSM records the second two sensor readings as `ADC3` and `ADC4`. Figure 3-19 shows the A/D FSM state map to record sensor readings. This is the same process used to record the first two sensor readings, except that now the readings will be recorded as `ADC3` and `ADC4`.

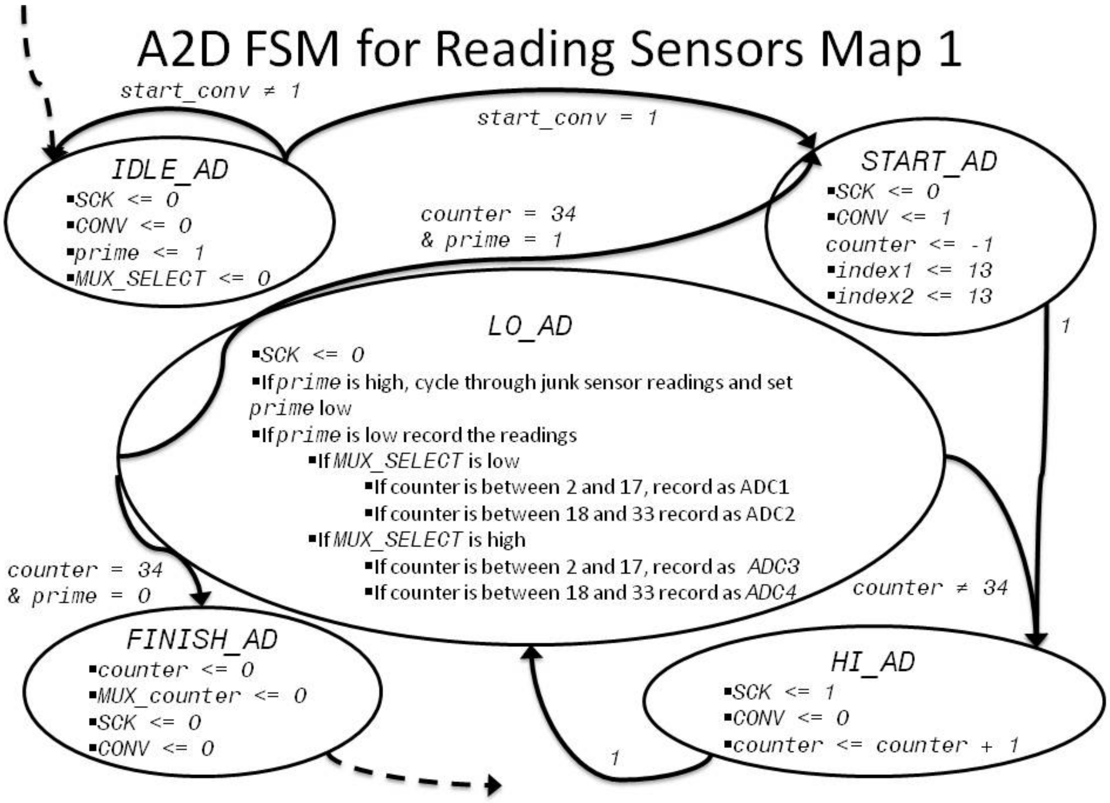


Figure 3-19: A/D FSM for Reading Sensor Map 1

3.3.5.1 Cycle Through One Set of Old Conversions. The A/D FSM is in the state `START_AD` after switching the multiplexer. In `START_AD`, the peripheral clock stays low, the variable `CONV` goes high, causing the A/D chip to take a new sample. The integer variables `counter`, `index1` and `index2` are initialized. From `START_AD`, the A/D FSM will automatically go to the state `HI_AD`. In `HI_AD`, the peripheral clock is set high, the variable `CONV` is set low and the variable `counter` is incremented.

The A/D FSM enters the state `LO_AD` automatically after being in the state `HI_AD`. In `LO_AD`, the peripheral clock is set low. During this pass through `LO_AD`, the variable `prime` is high and `MUX_SELECT` is high, and the variable `counter` will be 0. The logic will then put the A/D FSM back into the state `HI_AD`, where the peripheral clock will be set high again and the variable `counter` will be incremented to 1, signifying that the clock cycled once. The A/D FSM will cycle between `LO_AD` and `HI_AD` until `counter` has incremented to 34. At that time, the variable `prime` will be set low and the A/D FSM will go back to the state `START_AD` and where it begins the cycle to record a good set of conversions from sensors 3 and 4. This completes the phase to cycle through an old reading.

3.3.5.2 Cycle and Record Good Set of Conversions. After cycling through the old reading, the A/D FSM is ready to record the second two sensor readings as the two 14 bit variables `ADC3` and `ADC4`. This process starts in the state `START_AD`, where the peripheral clock is set low, the variables `counter`, `index1` and `index2` are reset and the variable `CONV` commands the A/D chip to take a sample and transmit the previous conversion.

The A/D FSM will automatically go from `START_AD` to `HI_AD`. In `HI_AD`, the peripheral clock is set high, `counter` is incremented and the variable `CONV` is set low. The A/D FSM will automatically go from `HI_AD` to `LO_AD`.

In `LO_AD`, the peripheral clock is set low. Because `prime` is now 0, the state `LO_AD` will complete a different process than before, and record the two sensor readings coming from the A/D chip. The sample being transmitted now was collected the last time the A/D FSM was in the state `START_AD`, but this time the multiplexer was set to read the second two channels. For the first pass through `LO_AD` for this cycle, `counter` is not yet greater than 2, so the A/D FSM will go back and forth between *LO_AD* and `HI_AD` to run the peripheral clock until the variable `counter` has been indexed to 3. At that time, when the A/D FSM is in `LO_AD`, the variable `ADC3` will store the information from the data bus channel `Master In Slave Out` in its 13th bit. When the A/D FSM leaves `LO_AD`, the variable `index1` is decremented and the FSM goes to `HI_AD`. This process is repeated until all 14 bits of `ADC3` have been recorded. At that point, `index1` is 0, `counter` is 16, and `index2` is 13. The peripheral clock is run by looping between `LO_AD` and `HI_AD` until `counter` has been incremented to 19. At this point, the variable `ADC4` will store the information from the data bus channel Master In Slave Out in its 13th bit. When the A/D FSM leaves `LO_AD`, the variable of `index2` is decremented and the FSM goes to `HI_AD`. This process repeats until all 14 bits of `ADC4` have been recorded. At this point, `index1` and `index2` are both 0 and `counter` is 32. This completes the process to convert the four sensor readings to digital numbers.

3.3.6 Set Flag "ADC_DONE" High so Master FSM Can Continue and Use A/D Conversion Results

The last process ended with the A/D FSM in the state *LO_AD* and the variable *counter* at 32. The peripheral clock is run some more by looping between *HI_AD* and *LO_AD* until *counter* reaches 34. At this time, the variable *ADC_DONE* is set high. This signals the master FSM that the conversion is complete, and it can continue to its next task.

After completing the four conversions and signaling the master FSM to continue, the A/D FSM resets itself and prepares for the next reading. From the state *LO_AD*, the A/D FSM transitions to the state *FINISH_AD*. In the state *FINISH_AD*, the variables *counter* and *MUX_counter* are set to 0. The peripheral clock and the variable *CONV* are set low. Because *MUX_SELECT* is 1, the A/D FSM will transition to the state *IDLE*, and will wait there until the master FSM requests the next sensor reading. This completes the process to tell the master FSM the conversion is complete.

3.4 Testing the A/D Converter

This section describes the validation process for the A/D conversion hardware. The multiplexer is examined to ensure it has ample settling time and will not be a source of noise on the system. The A/D conversion consistently has noise on the readings. Testing helped rule out possible sources of the noise.

3.4.1 Testing the Multiplexer Timing

The multiplexer is a VISHAY DG409L Precision Dual 4 Channel Low Voltage Analog Multiplexer. The multiplexer is supplied with ground and 5 volts. In this configuration, the maximum transition time should be 138 ns. The FPGA clock has a 20 ns period. The delay allowed in the A/D FSM is 200 clock cycles. This results in an allowed delay of 4000 ns to ensure that the signal has no leftover transition effects.

To test the multiplexer timing, the first channel was connected to a signal conditioning circuit outputting 0.4 volts. The second channel was connected to a different signal conditioning circuit tuned to output about 3 volts. An oscilloscope captured the output of the multiplexer triggering off the multiplexer control line. According to the data sheet for the multiplexer, a digital control logic high input voltage is 2.4 volts. In Figure 3-20, the switching threshold is after the 253 ns point. At 353 ns, the multiplexer appears to be past any transient unique to the switching condition.

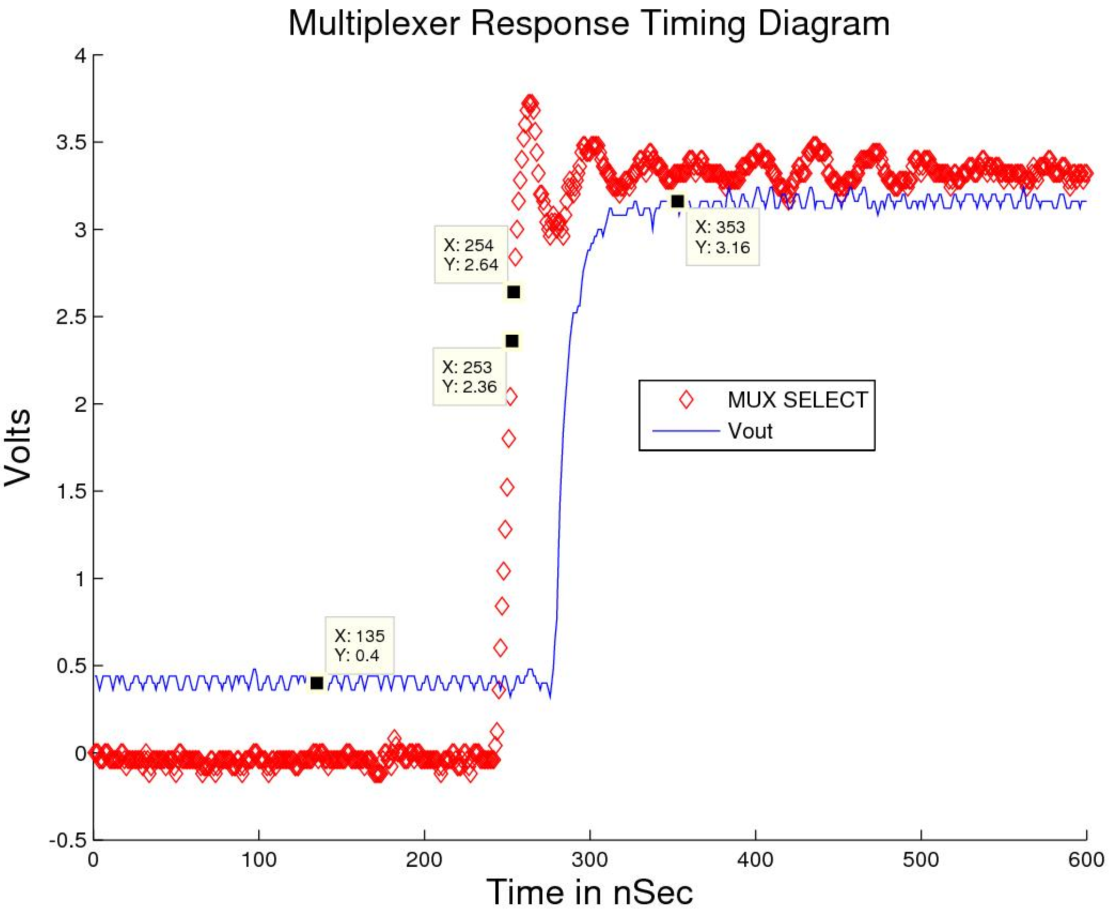


Figure 3-20: Multiplexer Response Timing Diagram

3.4.2 Testing for Noise in the A/D system

Isolating the A/D system allowed testing for noise. To isolate the system, an FPGA was powered from the wall socket transformer and connected directly to a power supply. Table 1 was collected at 0.8 volts and Table 2 was collected at 2.4 volts.

Table 1: Voltage Supply Readings at 0.8 Volts

Reading	ADC1	ADC2	ADC3	ADC4
Max	1418	1414	1419	1417
Min	1404	1393	1404	1392
Spread	14	21	15	25
Average	1411.3	1404.51	1411.93	1403.68
Standard Deviation	2.17	5.88	2.48	5.45

Table 2: Voltage Supply Feadings at 2.4 Volts

Reading	ADC1	ADC2	ADC3	ADC4
Max	-1290	-1289	-1290	-1289
Min	-1307	-1315	-1307	-1316
Spread	17	26	17	27
Average	-1297.52	-1302.33	-1297.96	-1302.69
Standard Deviation	2.30	5.62	2.72	6.13

Both sets of results for tables 1 and 2 are from 190 samples. Both channels of the A/D converter were connected to the same line from the voltage supply. This is similar to results found when running the boards from battery supplies while taking readings with the sensor boards.

Table 3 and Table 4 gathered using battery power and a test fixture to move the board. These two position readings were chosen because they show that even though the sensor readings may change in sign or magnitude, the noise pattern stays the same.

Table 3: Sensor Readings from Data Collection at 12.5 Inches

Reading	ADC1	ADC2	ADC3	ADC4
Max	2040	1975	2015	1613
Min	2033	1958	2007	1599
Spread	7	17	8	14
Average	2035.74	1965.22	2010.54	1606.04
Standard Deviation	1.04	5.00	1.14	5.20

Table 4: Sensor Readings from Data Collection at 6.0 Inches

Reading	ADC1	ADC2	ADC3	ADC4
Max	-892	207	1026	1869
Min	-907	183	1016	1851
Spread	15	24	10	18
Average	-899.19	194.71	1021.68	1861.52
Standard Deviation	2.96	5.58	1.83	5.29

It appears that values *ADC1* and *ADC3* will have less noise than values *ADC2* and *ADC4*, but the noise is not because of the sensor board, multiplexer or power supply.

Data was collected using a slower clock speed for the FPGA, which in turn slowed down the A/D FSM. Results were very similar to those above and the original clock speed was kept for the FPGA.

3.5 Credit for Previous Help on Writing the A/D FSM Code

Taylor York and Daniel Nash, former Oklahoma State University students enrolled in a capstone design course, submitted most of the A/D code. Amanuel Assefa, another Oklahoma State University student, contributed to changing the A/D code to incorporate the multiplexer and doubled the number of sensor readings.

3.6 Summary of the A/D Conversion Process

The A/D conversion provides the neural network with sensor readings by converting the input voltages to digital numbers. This process requires hardware and software. Most of the hardware came with the FPGA board. Intelligence and a process to control the hardware enables the A/D FSM to do its job which contributes to the smart sensor car being able to follow a wire.

CHAPTER IV

ARTIFICIAL NEURAL NETWORK

The neural network is a function that maps four sensor readings to the car position. The A/D converter produces four digital sensor readings. The PD controller accepts the car position. The neural network changes the output of the A/D converter into the desired input for the PD controller. Figure 4-1 shows the main block diagram.

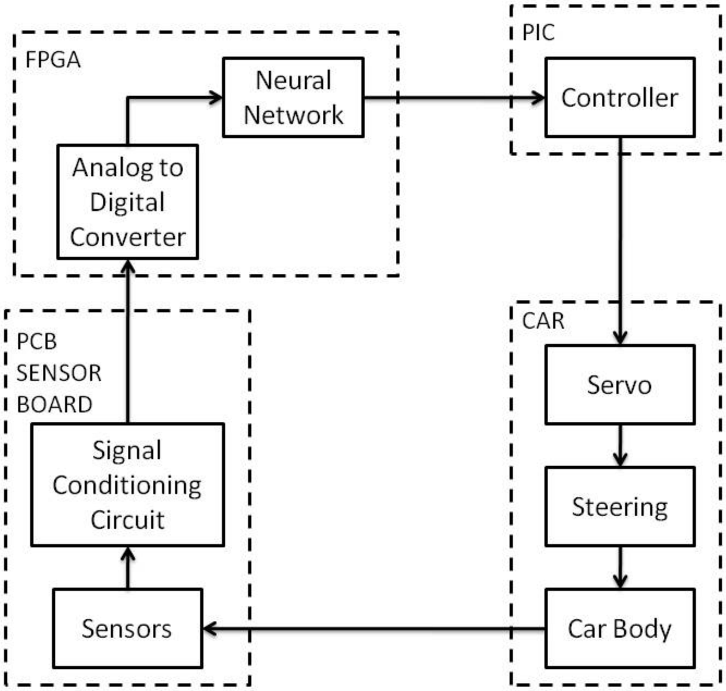


Figure 4-1: Main Block Diagram

4.1 Introduction to Neural Networks

The PD controller gives the car the steering command. The PD controller needs the position calculation from the neural network to know how to steer the car. The neural network needs the digital sensor readings from the A/D converter to calculate the position. The A/D converter needs analog signals from the signal conditioning circuit to produce the digital numbers. The signal conditioning circuit needs the voltages from the sensors to produce the analog signals. The sensors need the magnetic field from the track to produce a voltage. The magnetic field strength will depend on the position of the car, which was determined by the previous steering command from the PD controller. Figure 4-2 is the system functional overview that shows the required process to steer the car around the track.

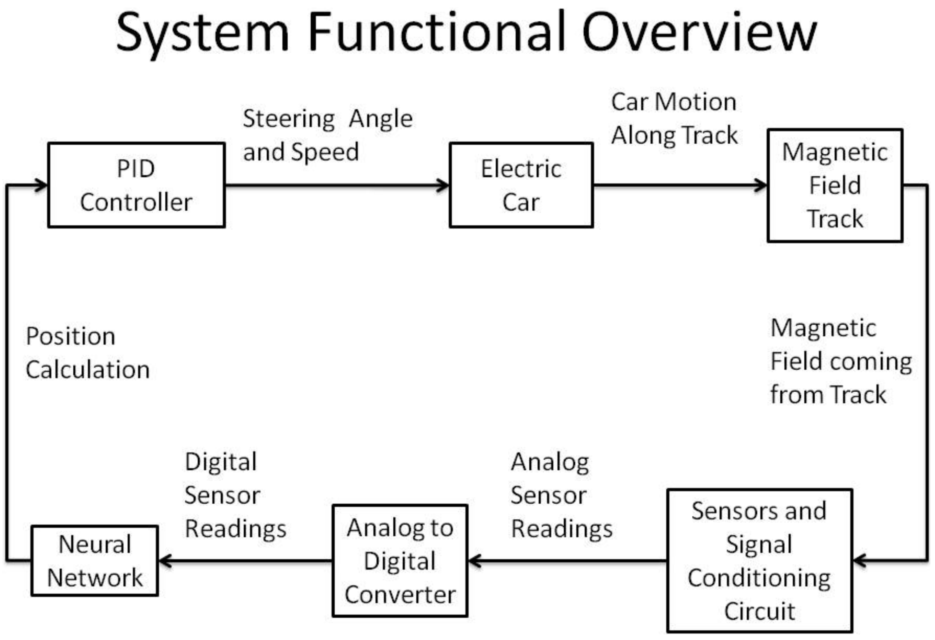


Figure 4-2: System Functional Overview

For the car to follow the wire, the process needs happen in a sequence. The PD Controller initiates the sequence by sending a request for a new position to the master Finite State Machine (FSM), and then the PD controller waits for a response. When the master FSM responds with a position, the PD controller calculates the steering command and requests another position. Figure 4-3 shows the PD controller process.

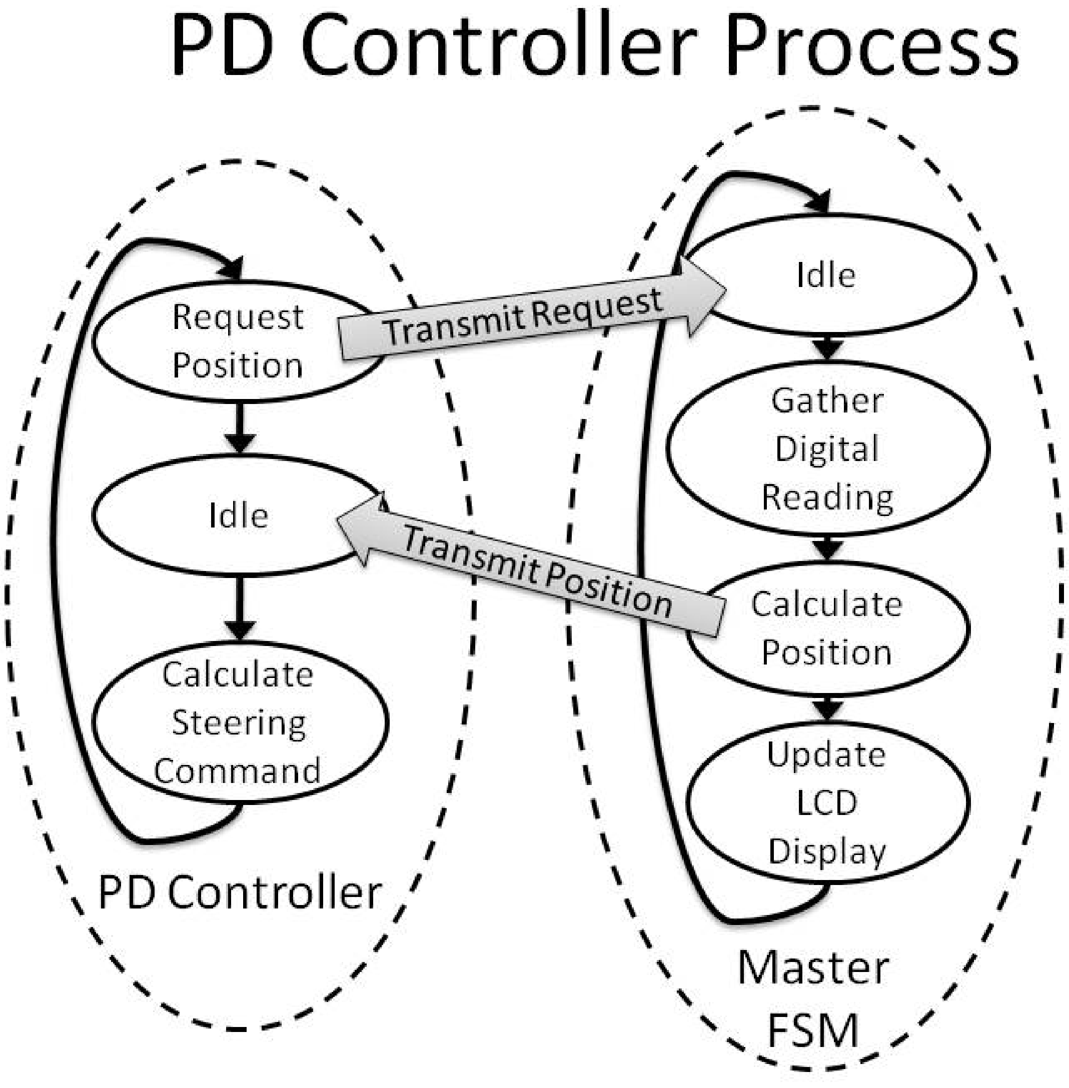


Figure 4-3: PD Controller Process

When the master FSM receives the request from the PD controller, it goes through a series of tasks. As covered in chapter 3, one of these tasks is to cycle the A/D converter FSM. Figure 4-4 shows an overview of how the master FSM interacts with the A/D FSM.

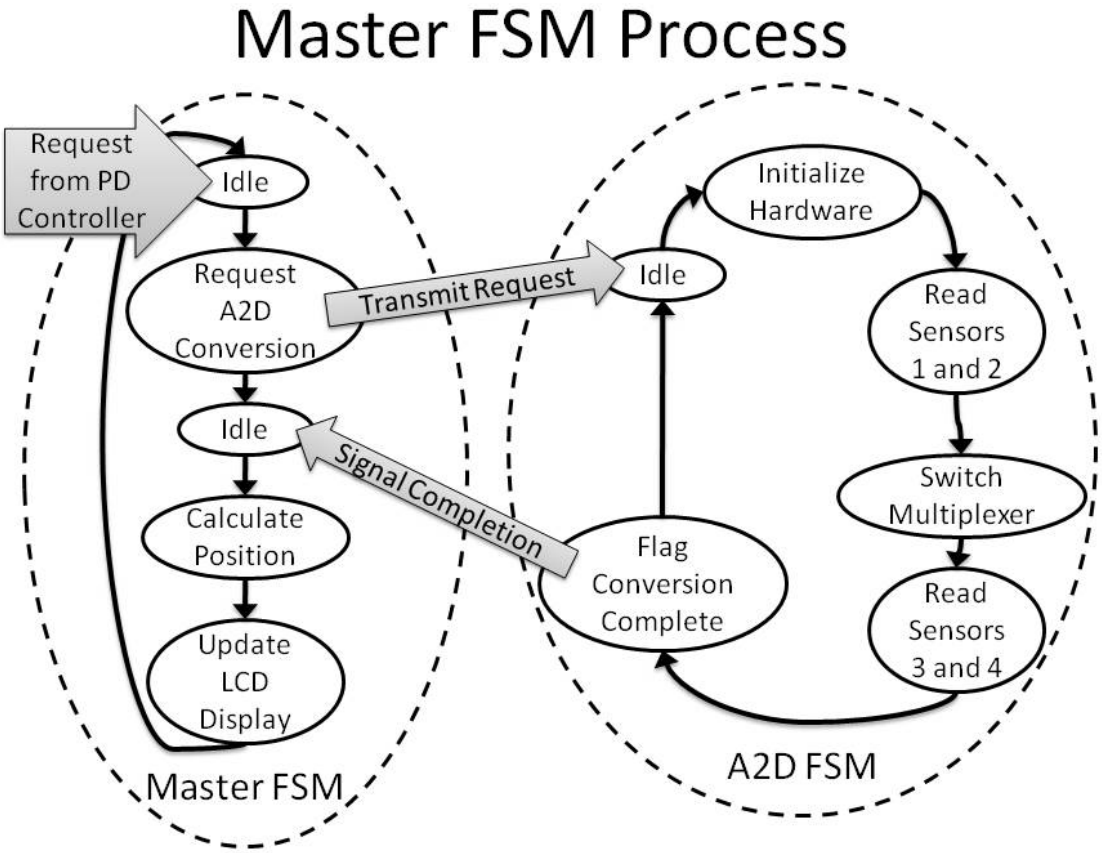


Figure 4-4: Master FSM Process Overview

After starting the A/D FSM, the master FSM waits for the A/D FSM to signal completion, then the master FSM continues with its tasks. One of those tasks is calculating position. The neural network calculates the car position using the four digital sensor readings as inputs. Like the A/D FSM, the neural network is implemented in hardware on the FPGA.

The neural network calculates the car position. This calculation can be thought of as a function or equation that has four inputs, the four digital sensor readings. That function combines the four inputs and generates a single answer, the car position. Training the neural network ensures that this equation is correct. Training has three major components. The first step is collection of training data. The four sensor readings are training inputs. The car position that corresponds to that set of sensor inputs will serve as the training target. Together, the inputs and target form the training data. The car will operate over a range of different positions. The training data set needs to have samples over that range. After the training data is collected, the neural network parameters are adjusted to properly map the sensor inputs to the car position. This process is called training. After the network is trained, its performance needs validation. This process is covered in section 4.4. Section 4.5 covers neural network supporting components. Supporting components do not fit in the main block diagram as the car goes around the track, but are an important part of making the smart car work. These components help with debugging, compiling and keeping all the other pieces of code working together. Another supporting component is custom software to help automate data collection.

4.2 Multilayered Perceptron

The neural network does a mapping from four sensor readings into a single car position. A neural network is a group of individual components called neurons. A neuron is also made of components. Figure 4-5 shows the neural network in decreasing levels of abstraction, down to the component level in line 3. Most of the information about the multilayered perceptron is from [1].

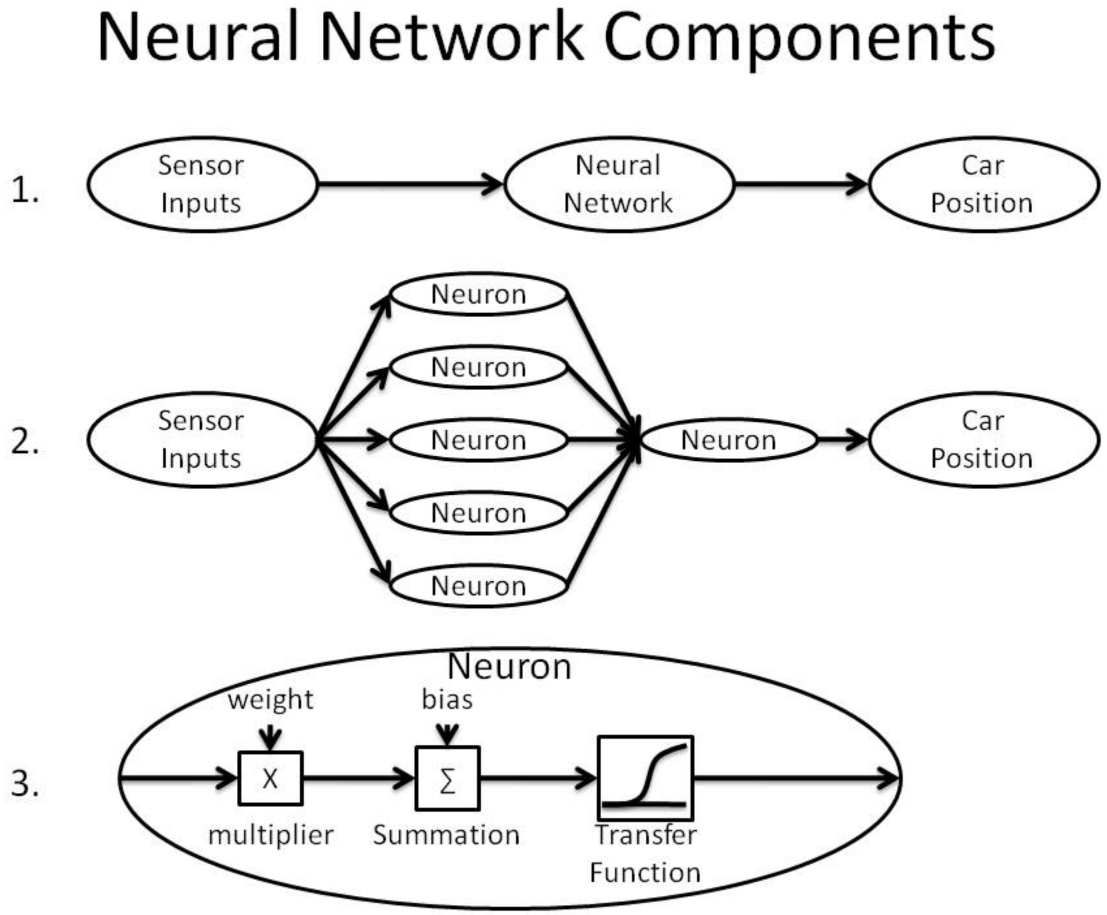


Figure 4-5: Neuron Components

4.2.1 Log Sigmoid Transfer Function

One of the components in the neuron is the transfer function. The smart car uses the log sigmoid transfer function in most of its neurons. The equation for the log sigmoid transfer function is:

$$Output = \frac{1}{1 + e^{-input}}$$

This transfer function is a squashing function because over the full range of inputs, the output will only vary between zero and one. Figure 4-6 shows a plot of the log sigmoid function.

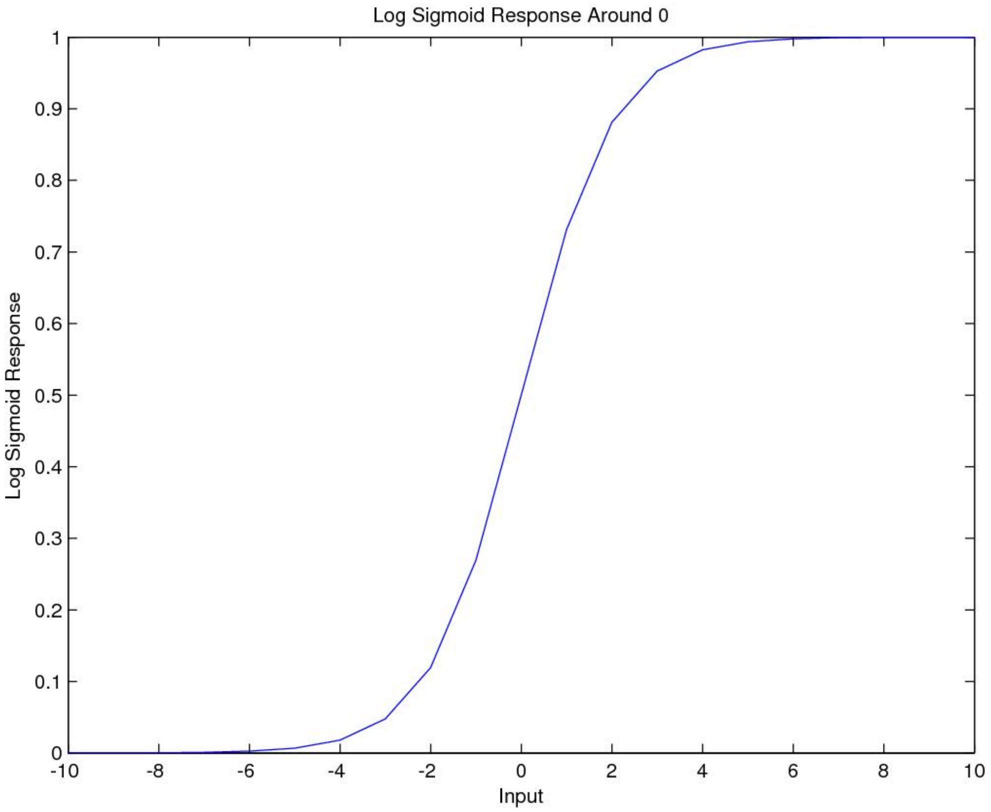


Figure 4-6: Log Sigmoid Response

4.2.2 Single Input Neuron

The single input neuron is made up of several components. The single input (p) is multiplied by a weight (w). The weight determines the neuron sensitivity to the input. The product of the input and weight is passed through a summation. The summation adds together the product and a bias (b) and produces the net input (n). The bias helps set the threshold for the neuron response. The equation for the net input is $n = w * p + b$. The net input is passed into the transfer function $\left(f(\quad)\right)$. The output of the transfer function is the output of the neuron (a). The equation for the single neuron is $a = f(n)$ which can be equivalently written:

$$a = f(w * p + b)$$

The perceptron neuron uses the log sigmoid transfer function, so it has the equation:

$$a = 1/(1 + e\char`^(-n)\,)$$

This equation can be equivalently written:

$$a = \frac{1}{1 + e^{-(w*p+b)}}$$

Figure 4-7 shows a single input neuron.

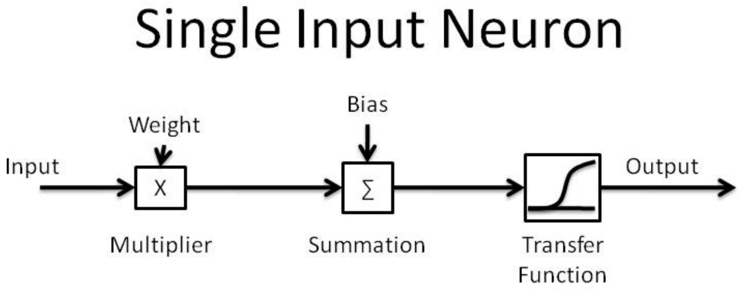


Figure 4-7: Single Input Neuron

Figure 4-8 shows the response of a single input log sigmoid neuron.

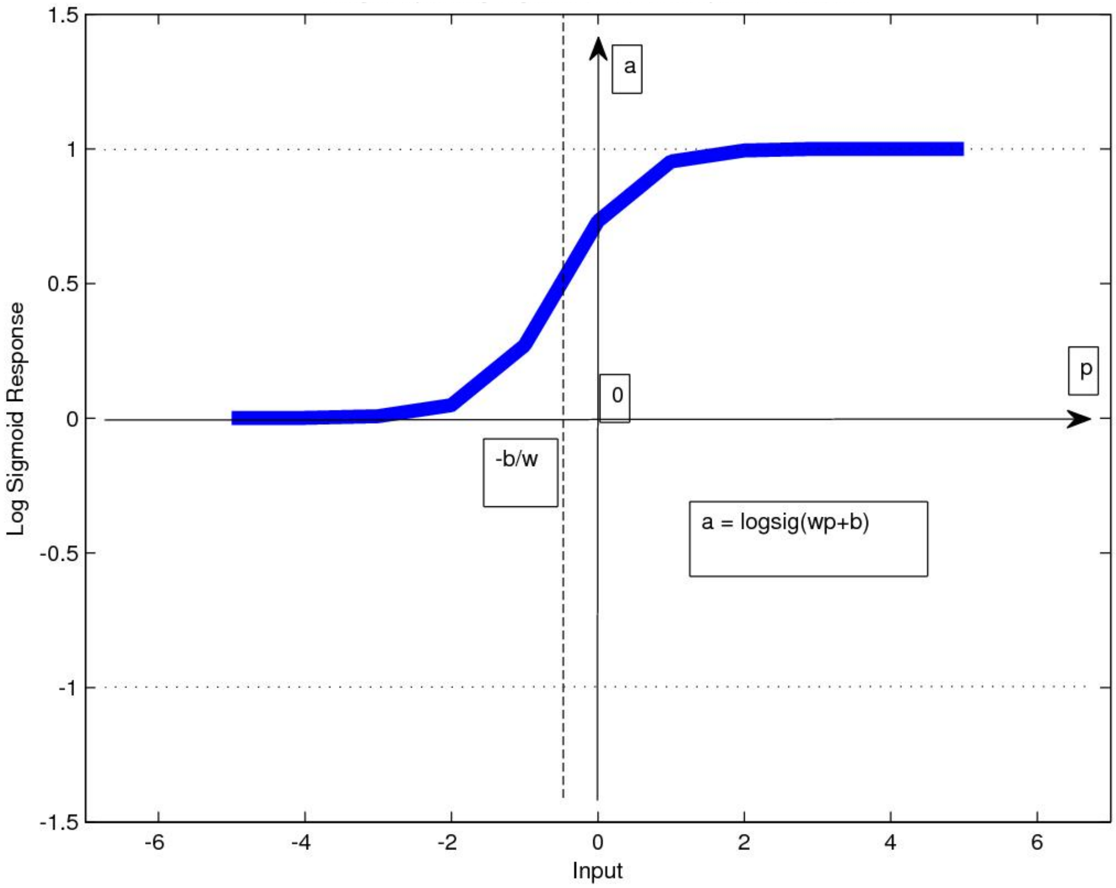


Figure 4-8: Single Input Log Sigmoid Neuron Response

4.2.3 Multiple Input Neuron

Multiple input neurons are similar to single input neurons. Each input (p_1), is multiplied by its corresponding weight (w_1), so that each input has a unique sensitivity. The product of each input and weight is added together with the products of the other inputs and weights and is then added to the bias. The result of the summation is the net input. The equation for the net input with R neuron inputs is $n = w_1 * p_1 + w_2 * p_2 + \cdots + w_R * p_R + b$. The net input goes through the transfer function to be the neuron output. Figure 4-9 shows a multiple input neuron

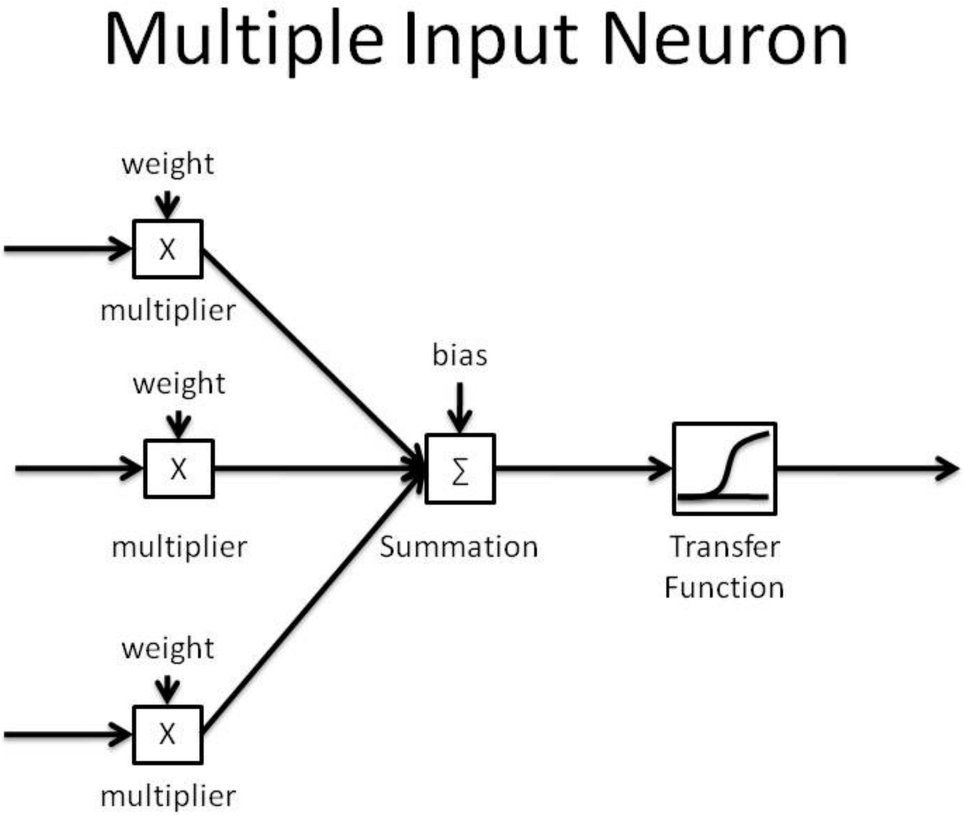


Figure 4-9: Multiple Input Neuron

The equation for a neuron with R inputs can be written:

$$a = f(w_1 * p_1 + w_2 * p_2 + \cdots + w_R * p_R + b)$$

or equivalently:

$$a = f(n)$$

For a multiple input perceptron neuron, the equation is

$$a = \frac{1}{1 + e^{-n}}$$

which can also be written as:

$$a = \frac{1}{1 + e^{(w_1 * p_1 + w_2 * p_2 + \cdots + w_R * p_R + b)}}$$

4.2.4 Multilayer Perceptron Network

Several neurons can be used together at once. When the neurons get the same inputs at the same time, they are said to be in the same layer. If there is only one input to the layer, then each neuron in that layer will have a single weight for that input. If there are multiple inputs to a layer, each neuron has a weight for every input. Figure 4-10 shows one layer of neurons.

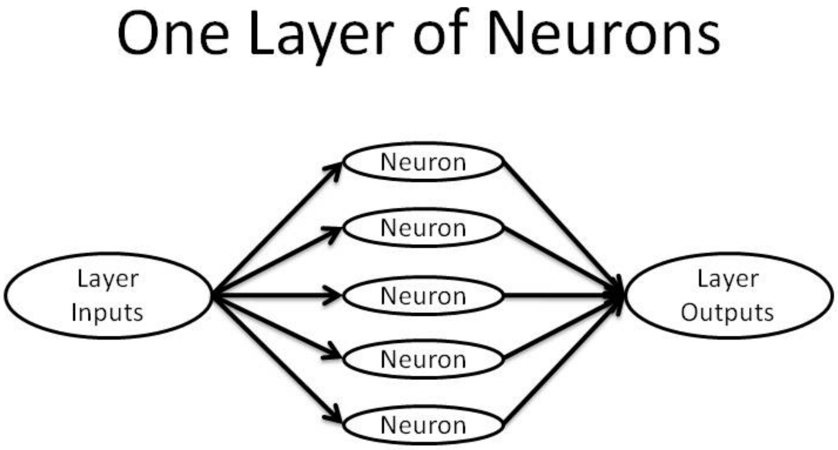


Figure 4-10: One Layer of Neurons

Layers of neurons can be linked together to form Multilayer Perceptron Networks (MLPs). The network inputs are the inputs to the first layer. The outputs of the first layer become the inputs to the second layer. The output of the last layer is the output of the network. The number of neurons in the last layer determines the number of outputs for the neural network. A superscript can show which layer a variable is associated with, and subscripts can be used to identify the place of the neuron in a layer and which input it is associated with. For example, the weight for the second neuron in the first layer for the third input is $w_{2,3}^{1}$. The typical MLP network used in smart sensors has two layers. The first layer of neurons uses the log sigmoid transfer function, and the second layer uses a linear transfer function. It can be shown that a two-layer MLP network is a universal approximator. This means it can approximate almost any data set with few limitations. The smart car uses an MLP network with five neurons in the first layer and one neuron in the second layer. The one network output is car position. The network has four inputs from the four sensors. Each neuron in the first layer has a weight for each input. The neuron in the second layer has a weight for each neuron in the first layer. Figure 4-11 shows the smart car MLP network focusing on the first neuron in the first layer.

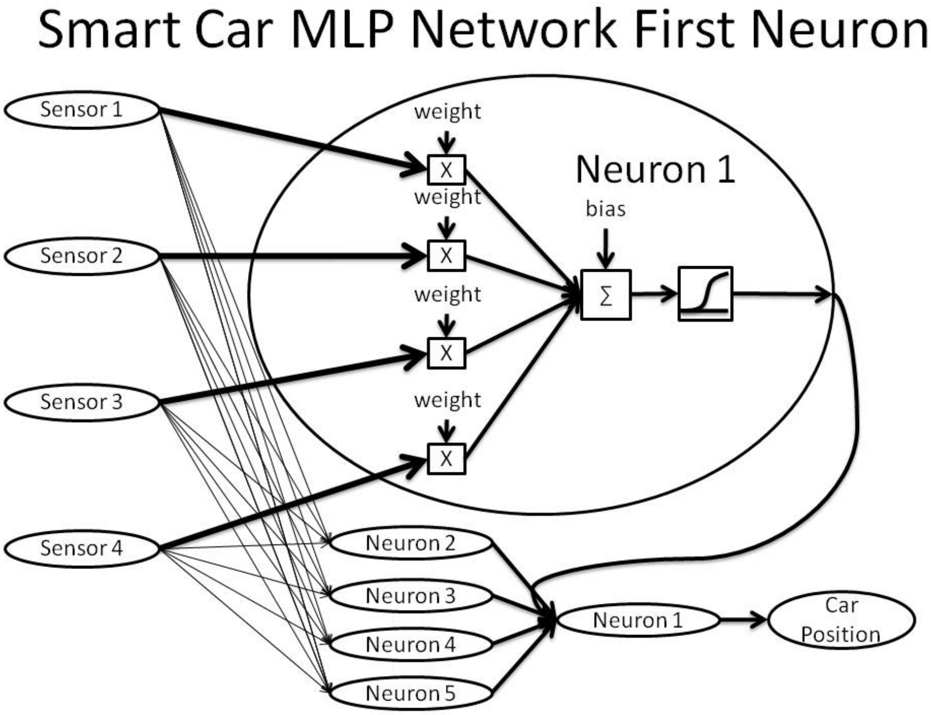


Figure 4-11: Smart Car MLP Network with Detailed Connections for First Neuron in the First Layer

Figure 4-12 shows the smart car MLP network focusing on the neuron in the output layer.

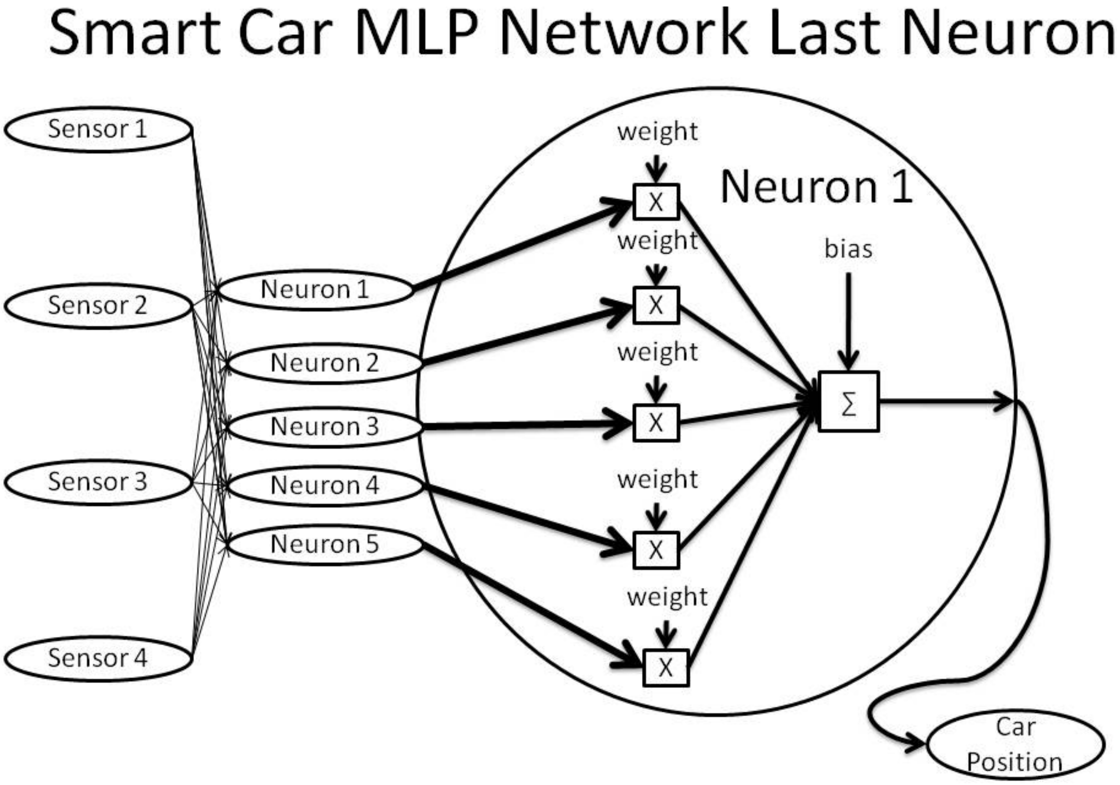


Figure 4-12: Smart Car MLP Network Connections for the Output Layer

The output equations for the five neurons in the first layer are as follows:

$$a_1^1 = {}^{1}\!/_{1 + e^{-(w_{1,1}^1 * p_1 + w_{1,2}^1 * p_2 + w_{1,3}^1 * p_3 + w_{1,4}^1 * p_4 + b_1^1)}}$$

$$a_2^1 = {}^{1}\!/_{1 + e^{-(w_{2,1}^1 * p_1 + w_{2,2}^1 * p_2 + w_{2,3}^1 * p_3 + w_{2,4}^1 * p_4 + b_1^1)}}$$

$$a_3^1 = {}^{1}\!/_{1 + e^{-(w_{3,1}^1 * p_1 + w_{3,2}^1 * p_2 + w_{3,3}^1 * p_3 + w_{3,4}^1 * p_4 + b_1^1)}}$$

$$a_4^1 = {}^{1}\!/_{1 + e^{-(w_{4,1}^1 * p_1 + w_{4,2}^1 * p_2 + w_{4,3}^1 * p_3 + w_{4,4}^1 * p_4 + b_1^1)}}$$

$$a_5^1 = {}^{1}\!/_{1 + e^{-(w_{5,1}^1 * p_1 + w_{5,2}^1 * p_2 + w_{5,3}^1 * p_3 + w_{5,4}^1 * p_4 + b_1^1)}}$$

The function for car position is as follows:

$$Car\ Position = {}^{1}\!/_{1 + e^{-(w_{1,1}^2 * a_1^1 + w_{2,1}^2 * a_2^1 + w_{3,1}^2 * a_3^1 + w_{4,1}^2 * a_4^1 + w_{5,1}^2 * a_5^1 + b_1^2)}}$$

4.3 Artificial Neural Network State Machine

The neural network provides the PD controller with the car position. The PD controller uses the car position in calculations that will help steer the car around the track. This process happens in a sequence driven by the PD controller. Figure 4-13 shows the system functional overview.

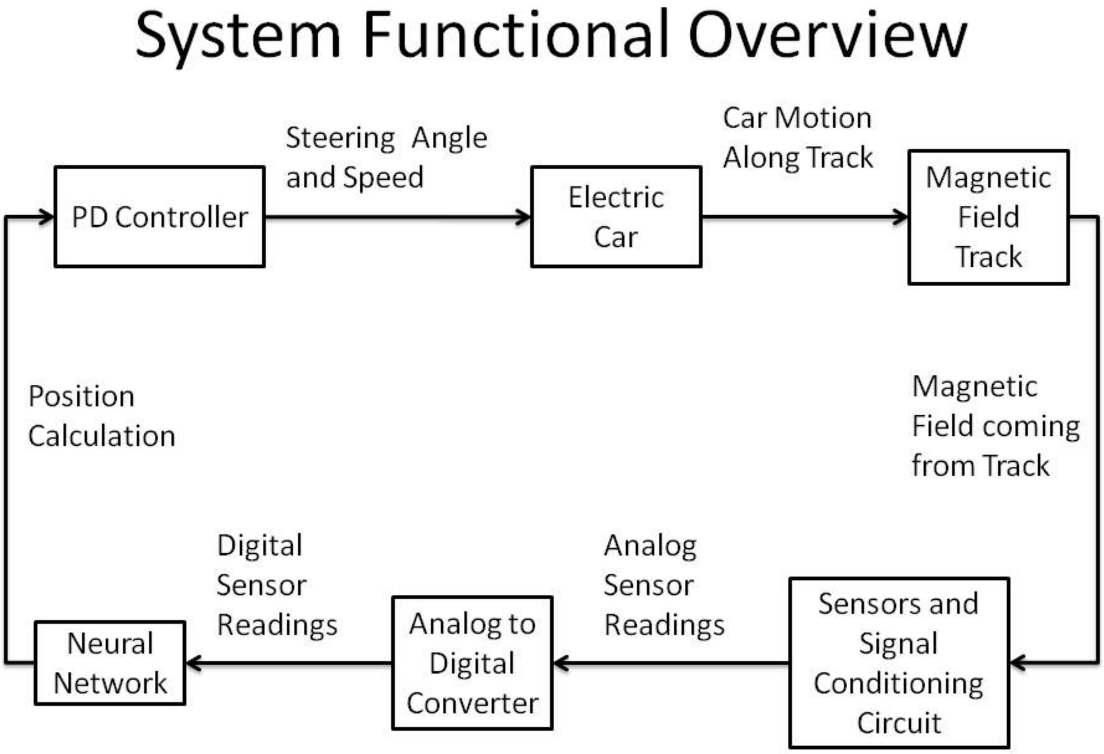


Figure 4-13: System Functional Overview

Every time the PD controller sends the command to the master FSM, the master FSM generates a new position. This section focuses on the processes inside the master FSM that allows it to generate a new car position. Figure 4-14 shows how the PD controller process interacts with the master FSM.

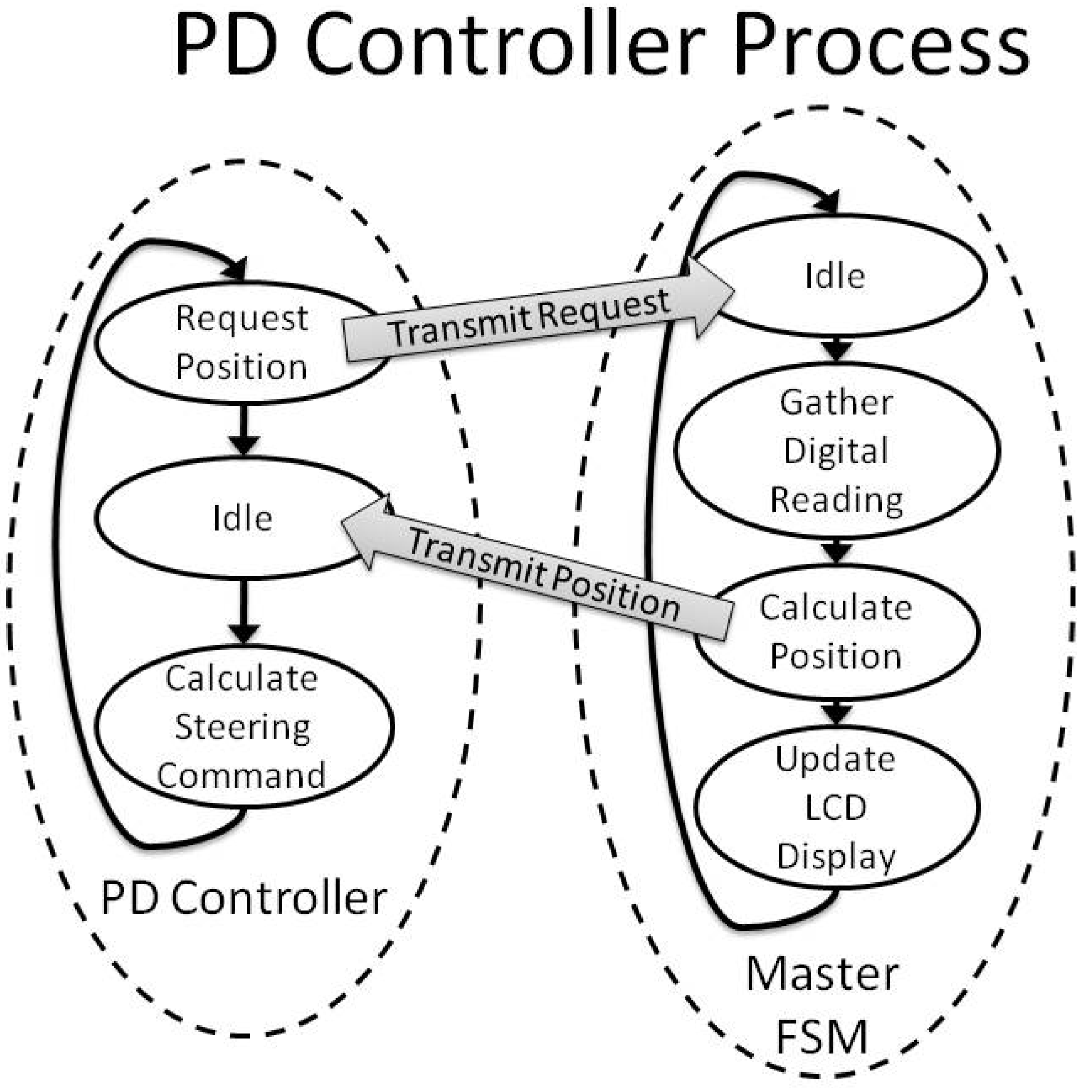


Figure 4-14: PD Controller Process

4.3.1 Introduction and Purpose: Position Calculation and the Master FSM

The master FSM has several pieces that work together to provide the PD controller with a new car position. Chapter 3 detailed the A/D conversion. Figure 4-15 shows the initial states the master FSM completed in order to cycle the A/D conversion process.

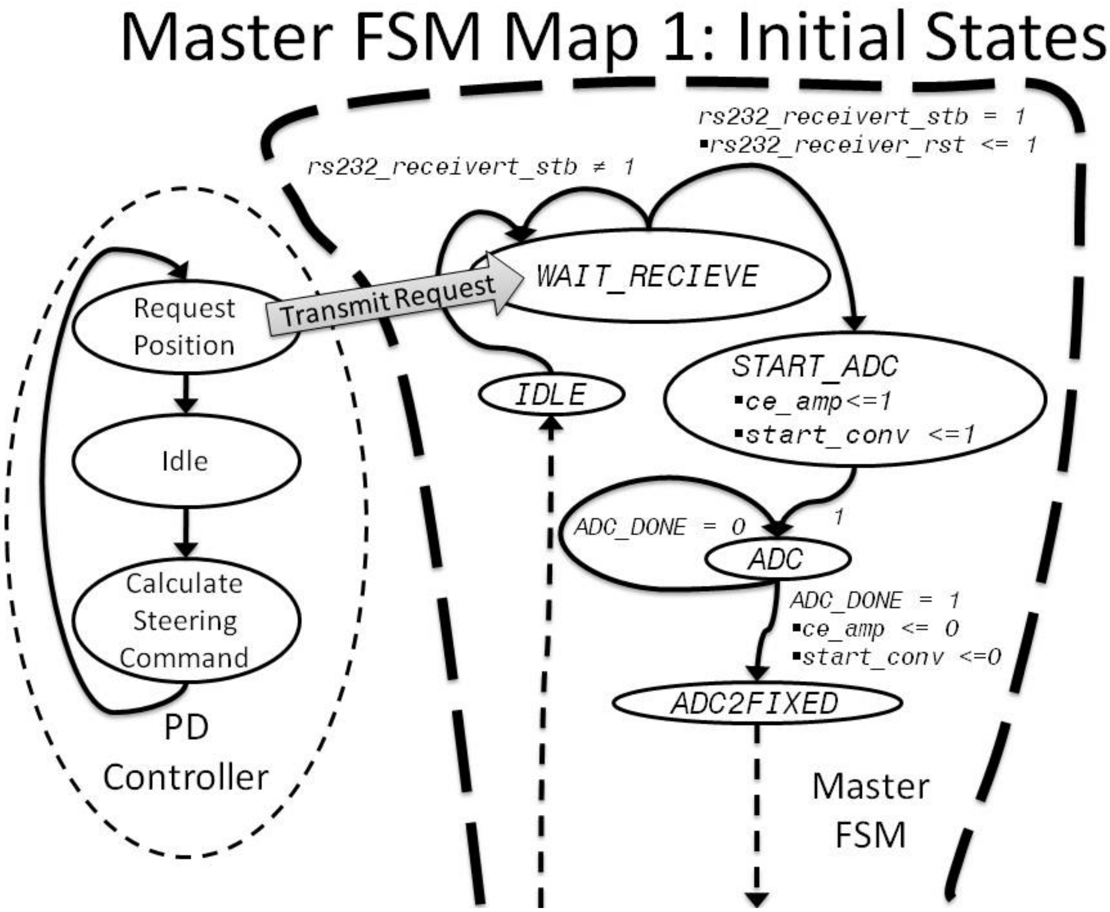


Figure 4-15: Master FSM Initial States

After the A/D conversion process, the master FSM has access to four digital numbers corresponding to the four sensor readings. In order to produce a new position calculation, the master FSM will put the numbers into the correct number format and store them into RAM. After storing them into RAM, the master FSM starts the neural network FSM. When the neural network FSM is complete, it sets a flag high and the master FSM can continue. The master FSM will convert the neural network output into the correct format, transmit the data to the PD controller, update the LCD display, then go back to the state `IDLE` and wait for the next request. Figure 4-16 shows an overview of the master FSM process.

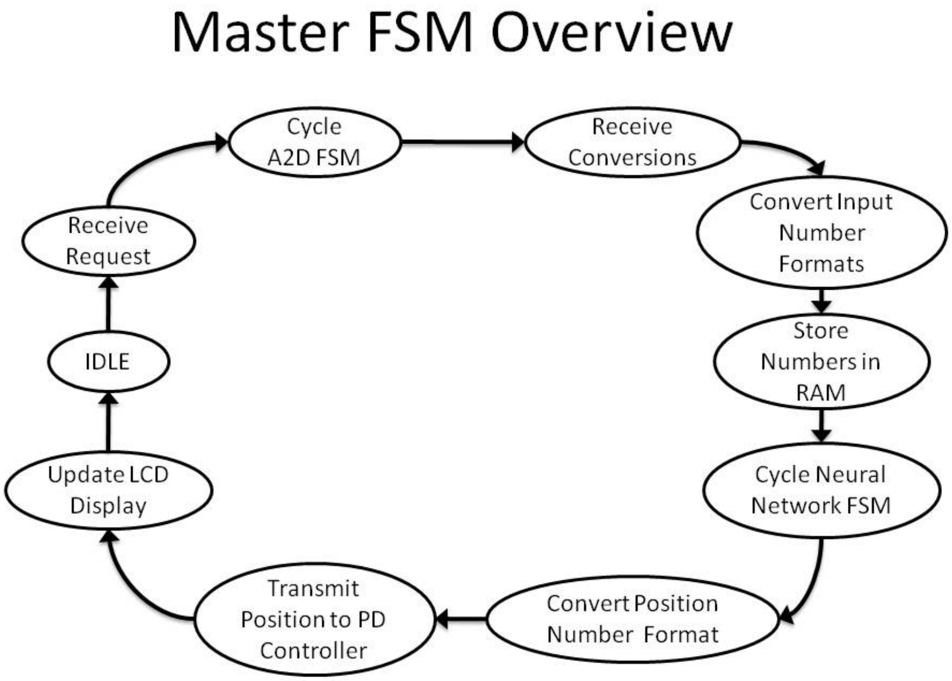


Figure 4-16: Master FSM Overview

The code for the master FSM to calculate a position is appendix A. In the code, many of the master FSM states interact with other FSMs and have bits or flags to drive transitions only after an outside FSM has completed its required task. The master FSM uses other FSMs to convert the fixed-point numbers that come from the A/D FSM into floating point numbers for use in the neural network FSM. After the neural network FSM calculates a new position, it gives the calculation result as a floating-point number. The master FSM drives another FSM to convert the floating-point output into a fixed-point number to transmit to the PD controller. In the states that drive conversions, many have a variable with *nd* in the name. These variables refer to the inputs of other FSMs and signify when the available data to convert is valid, meaning there is new valid data and the operation can begin. These variables are set high in the state prior to the conversion and set low once the conversion process begins. The conversion FSMs also have a variable with *rdy* in the name. This bit is set high by the conversion FSMs when the conversion result is valid.

When the master FSM is ready to store data into the RAM for later use by the neural network, it toggles the variable *ce_RAM* high. This variable drives the port in the neural network FSM with the variable name *WRITE_INPUT*. This variable needs to be high for the RAM to accept new information. The variables *RAM_DATA* and *RAM_ADDR* tell the neural network FSM the sensor reading conversion and the address to store it in. The neural network FSM uses the variables *Enable_Network* and *rdy_NN* to interface with the master FSM. When the variable *Enable_Network* is high, the network FSM can begin the calculation. When the variable *rdy_NN* goes high, the neural network FSM is done with its conversion and the master FSM can continue. When the variable *display_data* is set, the master FSM updates the LCD display. After the master FSM completes the transmission to the PD controller, it returns to the state `IDLE` and awaits the next request from the PD controller.

The process to cycle the A/D converter FSM begins with the state `WAIT_RECEIVE` where the master FSM waits on the PD controller to request a new signal and ends when the master FSM leaves the state ADC because the A/D FSM has signaled completion. Chapter 3 covers these states in greater detail. Figure 4-17 shows the first finite state machine map and first five states of the master FSM.

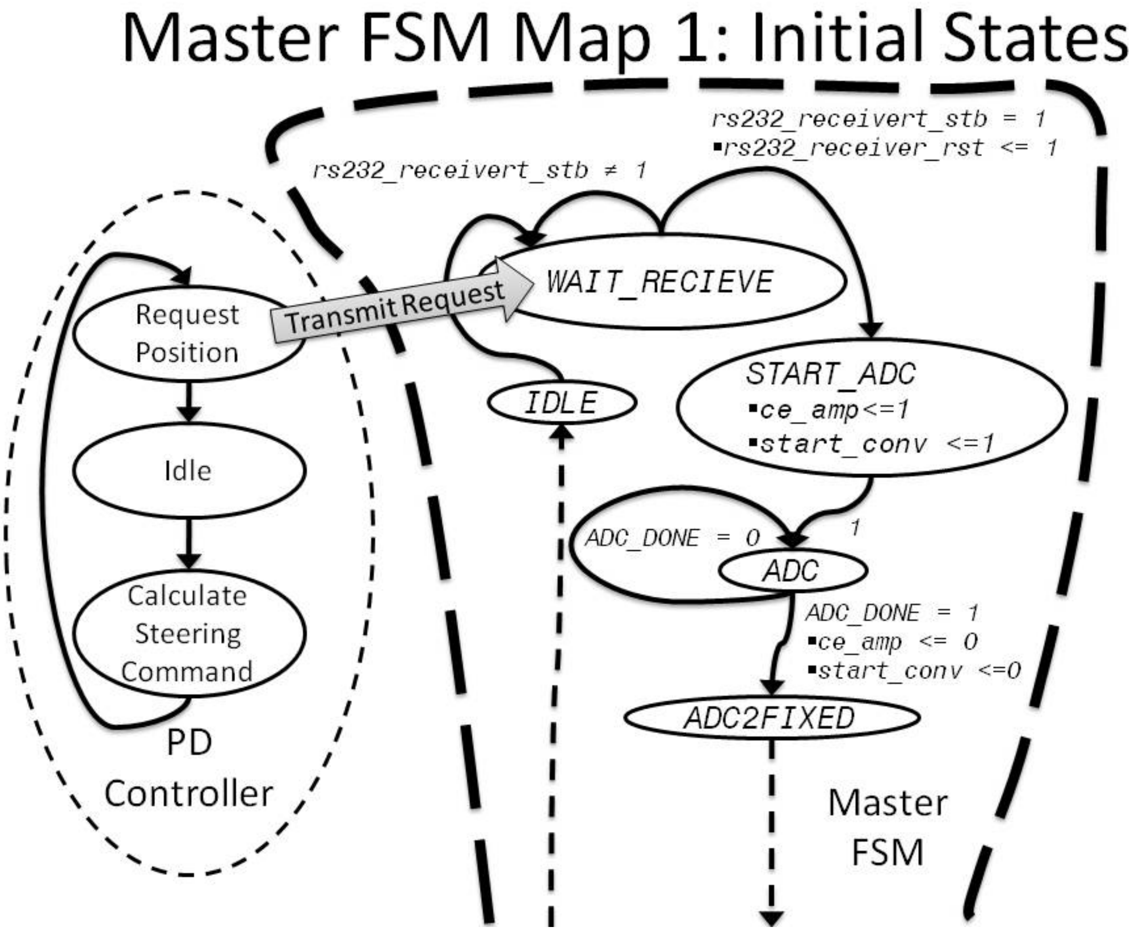


Figure 4-17: Master FSM Map 1: Initial States

When the master FSM leaves the state *ADC*, it automatically goes to the state *ADC2FIXED*. In this state, the variable *nd_fixed2float1* is set high, indicating that there is valid new data available and ready to be converted. The master FSM will automatically go from *ADC2FIXED* to the state *FIXED2FLOAT1*. In the state *FIXED2FLOAT1*, the variable *nd_fixed2float1* is set low. This signifies that the new data is now being converted. When the conversion is complete, the FSM to convert the first sensor reading will set the bit *rdy_fixed2float1* high. This will allow the master FSM to transition from the state *FIXED2FLOAT1* to the state *FIXED2FLOAT2*. During the transition, the variable *P1* is set to the result of the first conversion. The variable *P1* will be stored into the RAM and be the first input to the neural network. Also during the transition, the variable *nd_fixed2float2* is set high indicating that there is valid new data available and ready to be converted for the FSM that converts the second sensor reading.

In the state *FIXED2FLOAT2*, the second sensor reading is converted from fixed point to floating point. In this state, the variable *nd_fixed2float2* is set low. When the conversion is complete, the FSM to convert the second sensor reading will set the bit *rdy_fixed2float2* high. This will allow the master FSM to transition from the state *FIXED2FLOAT2* to the state *FIXED2FLOAT3*. During the transition, the variable *P2* is set to the result of the second conversion. Also during the transition, the variable *nd_fixed2float3* is set high. In the state *FIXED2FLOAT3*, the third sensor reading is converted from fixed point to floating point. In this state, the variable *nd_fixed2float3* is set low. When the conversion is complete, the FSM to convert the third sensor reading will set the bit *rdy_fixed2float3* high. This will allow the master FSM to transition from the state *FIXED2FLOAT3* to the state *FIXED2FLOAT4*. During the transition, the variable *P3* is set to the result of the third conversion. Also during the transition, the variable *nd_fixed2float4* is set high. In the state *FIXED2FLOAT4*, the fourth sensor reading is converted from fixed point to floating point. In this state, the variable *nd_fixed2float4* is set low. When the conversion is complete, the FSM to convert the fourth sensor reading will set the bit *rdy_fixed2float4* high. This will allow the master FSM to transition from the state *FIXED2FLOAT4* to the state *WRITE_ADC_DATA1*. During the transition, the variable *P4* is set to the result of the fourth conversion. Figure 4-18 shows the second master FSM map.

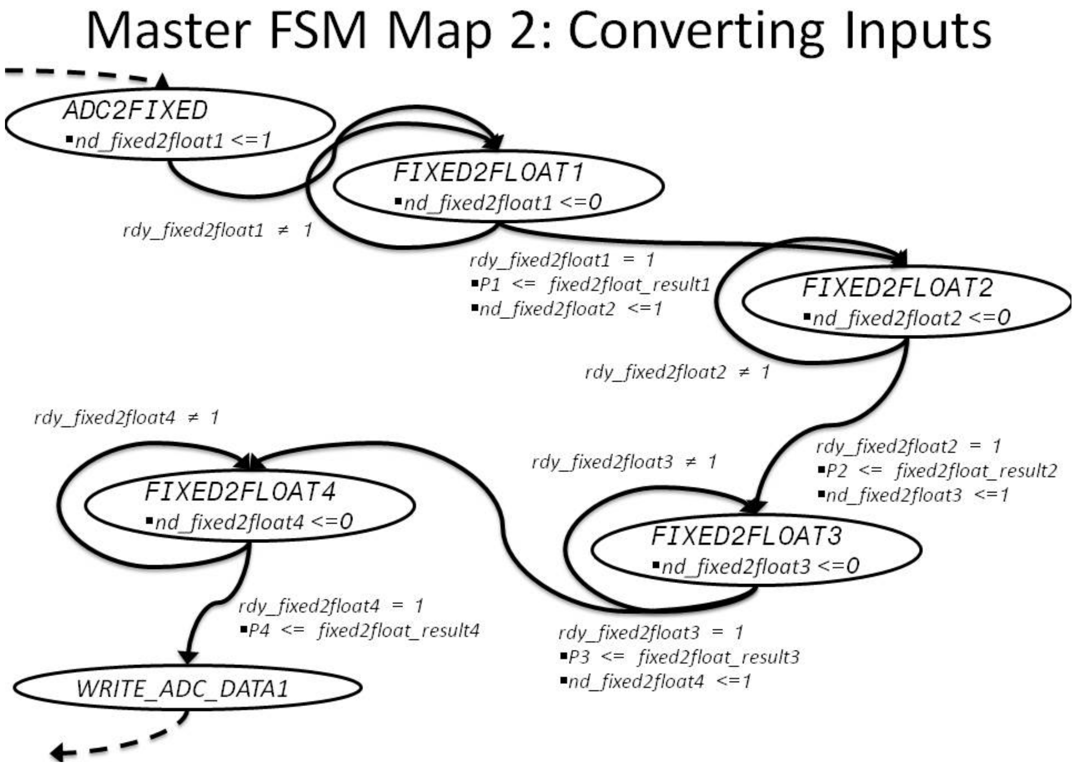


Figure 4-18: Master FSM Map 2: Input Conversion

The process to convert the four numbers takes 28 clock cycles at 20 nSec per clock cycle. Figure 4-19 shows the timing diagram for process to convert the number format for the four inputs. Figure 4-19 was generated using the Xilinx Chipscope software.

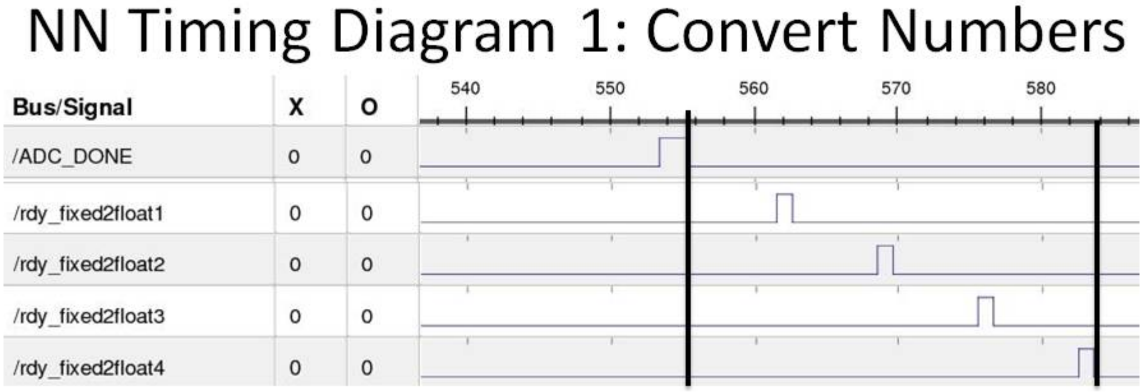


Figure 4-19: Timing Diagram for Number Conversion

The master FSM works together with the neural network FSM to write the four sensor readings to the RAM. In the state *WRITE_ADC_DATA1*, the first sensor reading is loaded into the RAM for later use with the neural network FSM. The variable *Enable_Network* is set low. This variable is tied to the neural network FSM that will be discussed in the next section. The variable *ce_RAM* is set high. This enables the RAM to load new values. The variable *RAM_DATA* is set to the variable *P1*. The variable *RAM_ADDR* is set to the value *"0000"*. This will set the first address in the RAM equal to the first converted sensor reading. The master FSM will automatically transition from the state *WRITE_ADC_DATA1* to the state *WRITE_ADC_DATA2*.

In the state *WRITE_ADC_DATA2*, the second sensor reading is loaded into the RAM. The variable *RAM_DATA* is set to the variable *P2*. The variable *RAM_ADDR* is set to the value *"0001"*. This will set the second address in the RAM equal to the second converted sensor reading. The master FSM will automatically transition from the state *WRITE_ADC_DATA2* to the state *WRITE_ADC_DATA3*.

In the state *WRITE_ADC_DATA3*, the variable *RAM_DATA* is set to the variable *P3*. The variable *RAM_ADDR* is set to the value *"0010"*. The master FSM will automatically transition from the state *WRITE_ADC_DATA3* to the state *WRITE_ADC_DATA4*.

In the state *WRITE_ADC_DATA4*, the variable *RAM_DATA* is set to the variable *P4*. The variable *RAM_ADDR* is set to the value *"0011"*. The master FSM will automatically transition from the state *WRITE_ADC_DATA4* to the state *WRITE_ADC_DONE*.

In the state *WRITE_ADC_DONE*, the variable *ce_RAM* is set low. The master FSM will automatically transition from the state *WRITE_ADC_DONE* to the state *START_NN*. In the state *START_NN* the variable *Enable_Network* is set high and the integer variable *counter* is set to zero. The master FSM will automatically transition from the state *START_NN* to the state *NN*.

In the state *NN*, the neural network FSM calculates the car position. Section 4.3.2 covers this calculation in greater detail. If the neural network FSM has not completed the calculation, the master FSM will loop back into the state *NN*. If the neural network FSM has completed its task, the variable *NN_Data* is set to *N*, the variable *Enable_Network* is set low, and the variable *nd_float2fixed* is set high in preparation for the next conversion. After the neural network FSM has completed the calculation, the master FSM will transition from the state *`NN`* to the state *`FLOAT2FIXED`*.

In the state *`FLOAT2FIXED`*, a FSM converts the floating-point car position from the neural network into a fixed-point number the master FSM can transmit to the PD Controller. If the converting FSM has completed its task, the variable *`nd_float2fixed`* is set low and the master FSM will transition from the state *`FLOAT2FIXED`* to the state *`START_RS232_HI`*.

Figure 4-20 shows the third master FSM state map and includes the states to write the four sensor readings to the RAM for the neural network FSM.

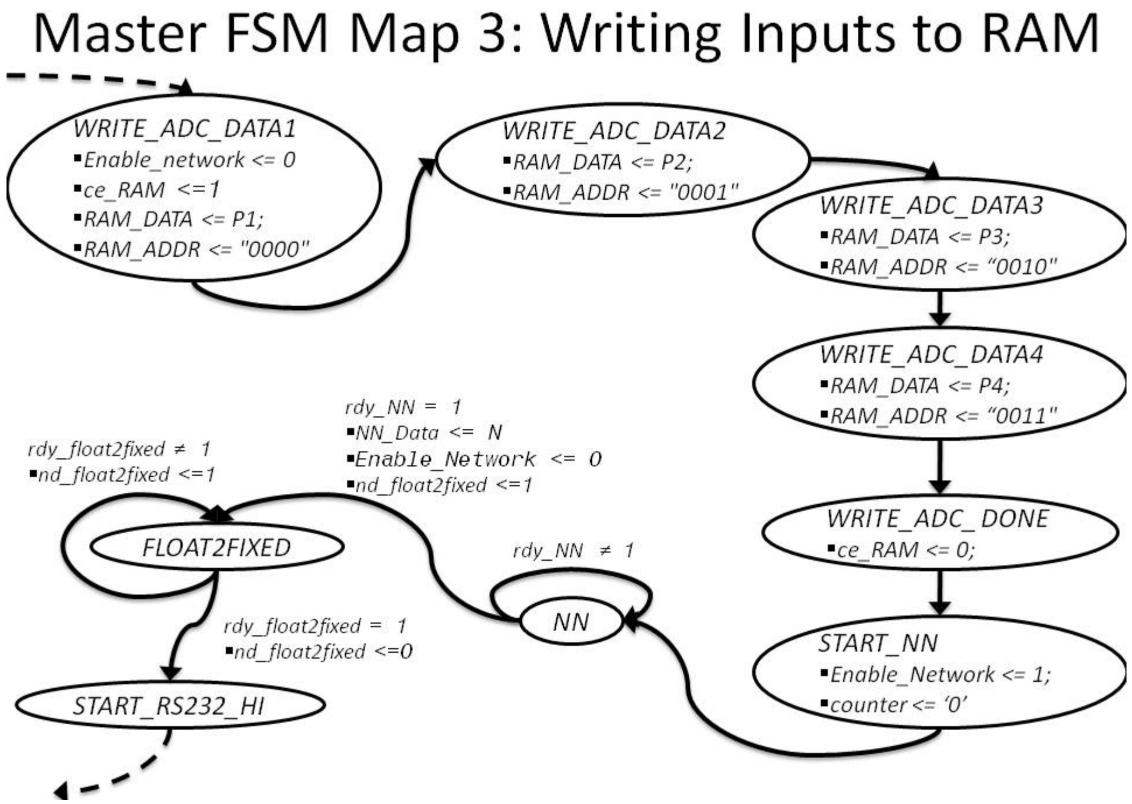
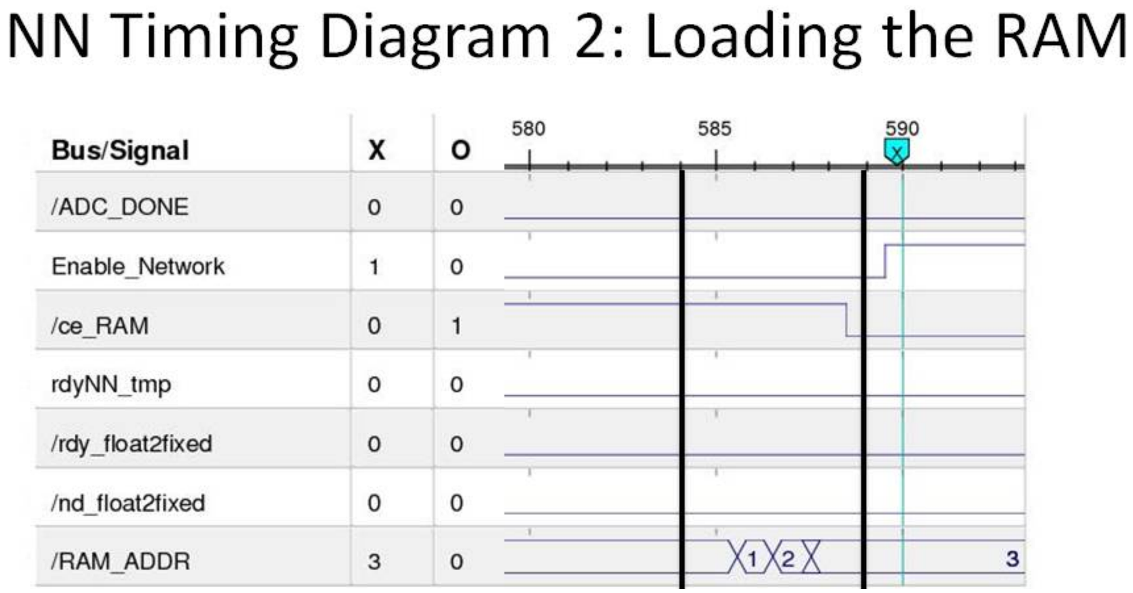


Figure 4-20: Master FSM Map 3: Writing Inputs to RAM

Loading the ram takes fewer than 10 clock cycles. Figure 4-19 shows that the last conversion is complete at clock cycle 584. Figure 4-21 is the timing diagram for loading the four inputs into the RAM and shows that by clock cycle 596 the RAM address has cycled through all four inputs.



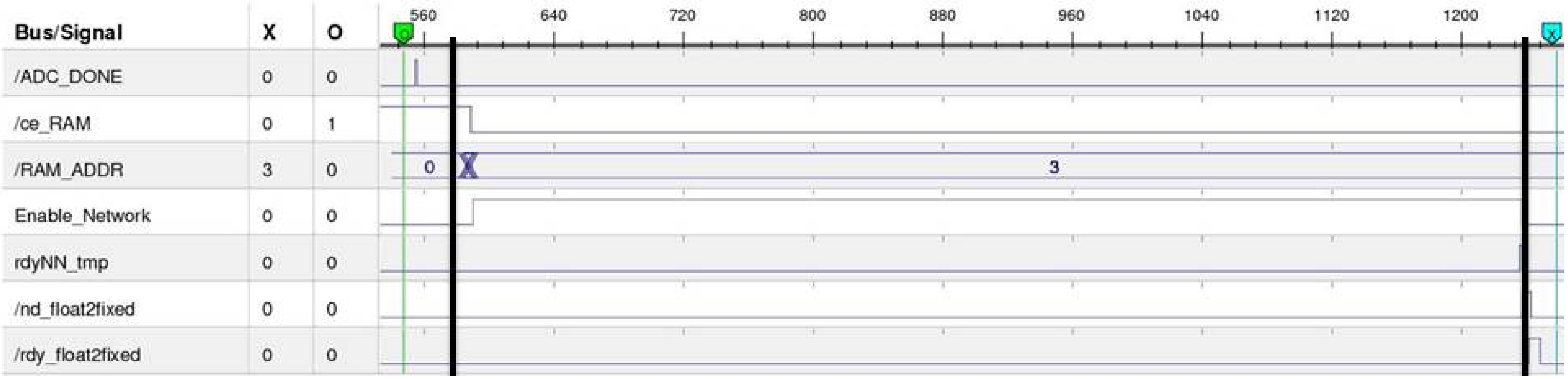
NN Timing Diagram 2: Loading the RAM

Bus/Signal	X	O
/ADC_DONE	0	0
Enable_Network	1	0
/ce_RAM	0	1
rdyNN_tmp	0	0
/rdy_float2fixed	0	0
/nd_float2fixed	0	0
/RAM_ADDR	3	0

Figure 4-21: Loading the RAM

The variable `Enable_Network` is high as long as the neural network FSM is calculating a position. The neural network takes less than 700 clock cycles from start to finish. Figure 4-22 shows the timing diagram encompassing the neural network calculation.

NN Timing Diagram 3: Network Calculation



Bus/Signal	X	O
/ADC_DONE	0	0
/ce_RAM	0	1
/RAM_ADDR	3	0
Enable_Network	0	0
rdyNN_tmp	0	0
/nd_float2fixed	0	0
/rdy_float2fixed	0	0

Figure 4-22: Neural Network Calculation Timing Diagram

After converting the car position from floating point to fixed point, the master FSM will transition from the state *FLOAT2FIXED* to the state *START_RS232_HI*. In the state *START_RS232_HI*, the variable *counter* is set to zero, the RS232 sender FSM loads three zeros and the first five bits of the car position into the variable *rs232_sender_dat*. The master FSM will automatically transition from the state *START_RS232_HI* to the state *RS232_HI*. In the state *RS232_HI*, the RS232 sender FSM broadcasts the variable *rs232_sender_dat*. When the variable *counter* reaches ten times the system speed divided by the baud rate, the transmission is complete and the master FSM goes from the state *RS232_HI* to the state *START_RS232_LO*. In the state *START_RS232_LO*, the variable *counter* is set to zero, the RS232 sender FSM loads the bottom byte of the car position into the variable *rs232_sender_dat*. The master FSM will automatically transition from the state *START_RS232_LO* to the state *RS232_LO*. In the state *RS232_LO*, the RS232 sender FSM broadcasts the variable *rs232_sender_dat*. When the variable *counter* reaches ten times the system speed divided by the baud rate, the transmission is complete and the master FSM goes from the state *RS232_LO* to the state *IDLE*.

The serial communication from the FPGA uses the 38400/8N1 parameter setting. This means the communication happens at 38400 pulses per second, there are eight data bits, no parity bit and one stop bit. The 8N1 setting is a common configuration for PC serial communications. The 38400-baud rate allows communication between the FPGA and PD controller to be quick enough to not interfere with steering the car. This configuration information allows other serial devices, such as a computer, to communicate with the master FSM.

Figure 4-23 shows the master FSM map that includes the states to transmit the car position to the PD controller.

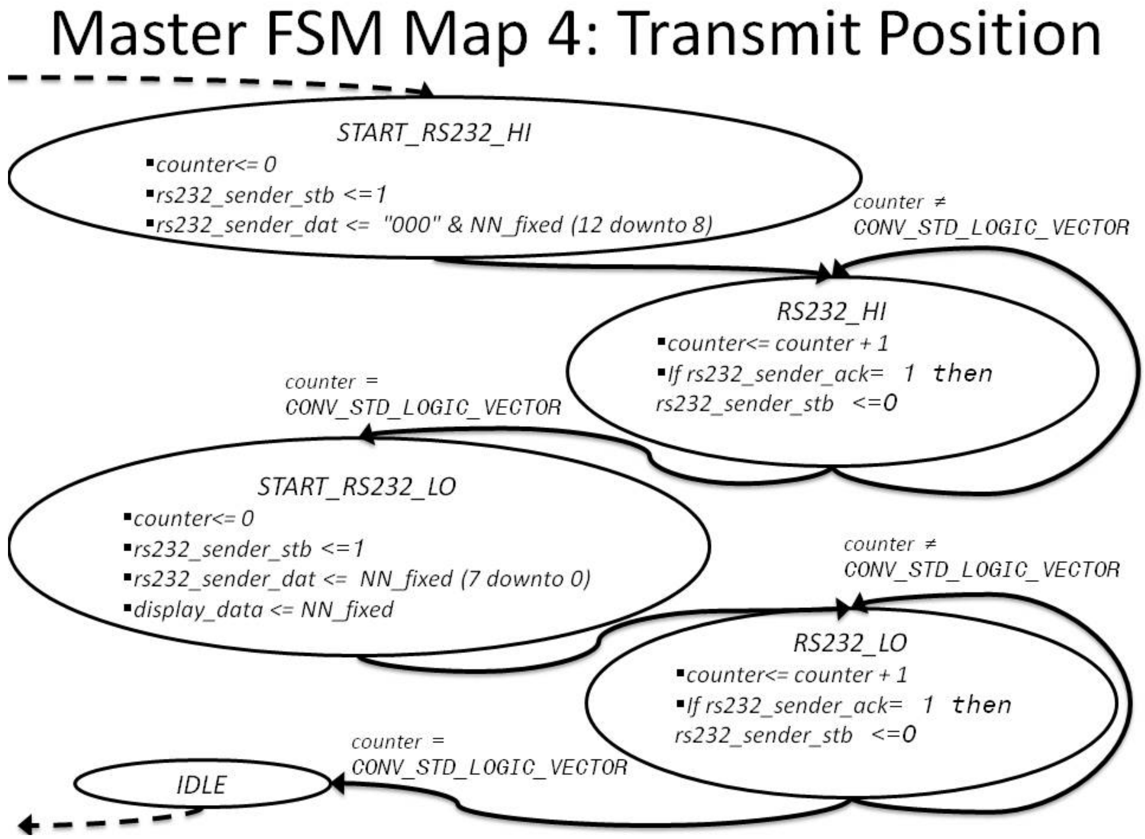


Figure 4-23: Master FSM Map 4: Transmit Position

4.3.2 Serial Implementation Description and Neural Network FSM Overview

The previous section described the master FSM. The master FSM drives many other FSMs to complete tasks, including the neural network FSM. The neural network and master FSM work together to load the sensor readings into the RAM. Then the neural network FSM calculates the car position. This calculation is a serial process, starting with calculating the output of the first layer, followed by calculating the output of the second layer. The network used here is described in [2]. Figure 4-24 shows an overview of this process.

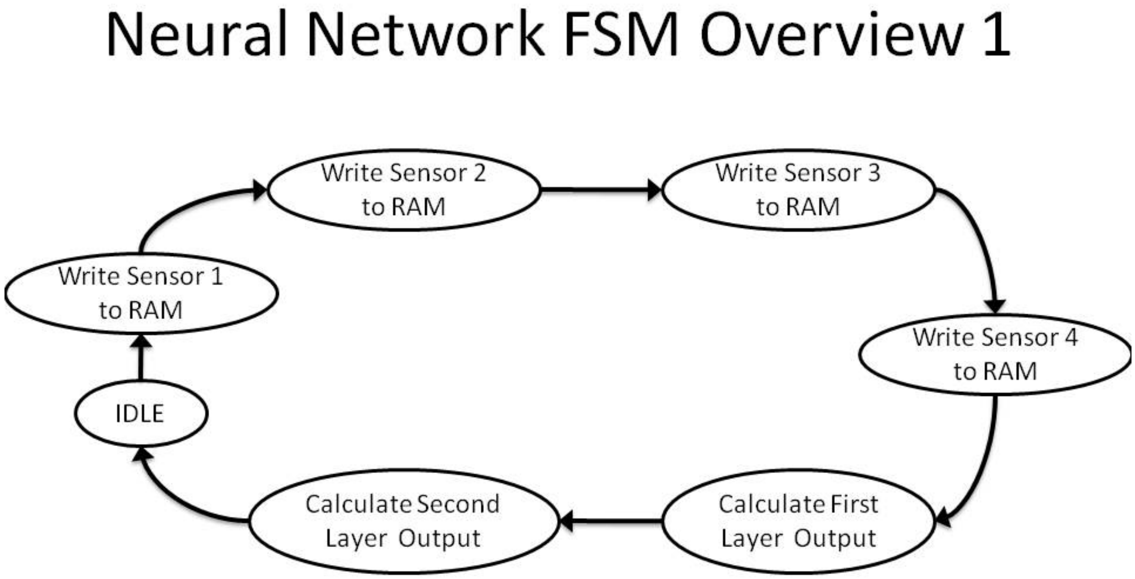


Figure 4-24: Neural Network FSM Overview 1

The process to calculate each layer output is also serial. Each layer output is found a single neuron at a time. Figure 4-25 shows the order the neuron outputs are calculated. This example is for five neurons in the first layer and one neuron in the second layer, but the NN FSM can be used with any MLP architecture. The number of neurons in each layer and the number of inputs is loaded into a structure ROM, as shown in Figure 4-27.

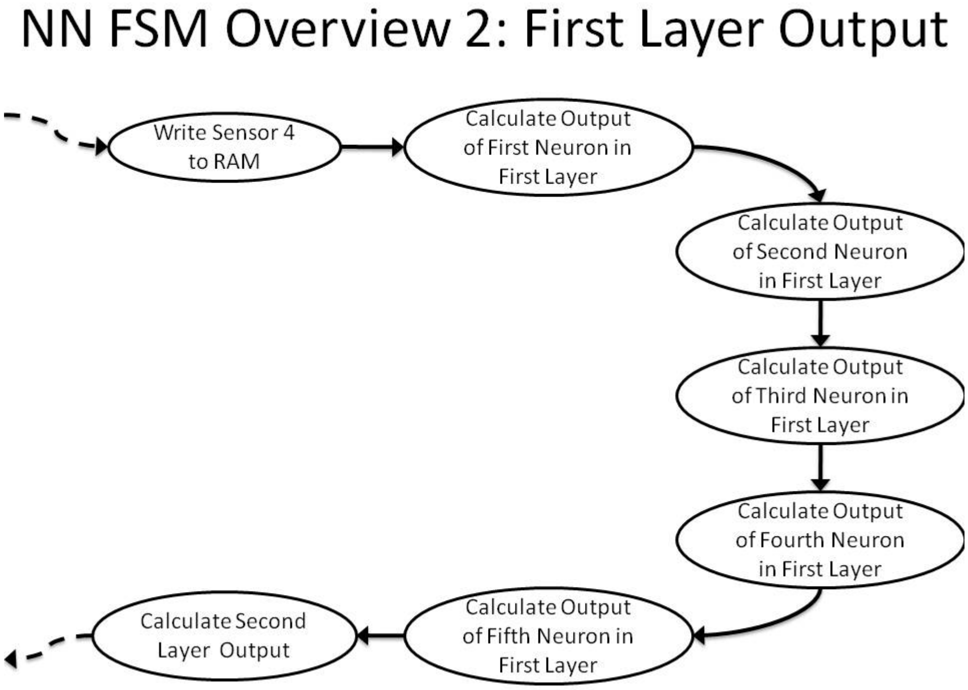

NN FSM Overview 2: First Layer Output
Write Sensor 4 to RAM
Calculate Output of First Neuron in First Layer
Calculate Output of Second Neuron in First Layer
Calculate Output of Third Neuron in First Layer
Calculate Output of Fourth Neuron in First Layer
Calculate Output of Fifth Neuron in First Layer
Calculate Second Layer Output

Figure 4-25: Neural Network Overview 2: First Layer Output

Each neuron output is also found serially. The first input is multiplied by its corresponding weight (which is loaded in a weight ROM, as show in Figure 4-27). That product is added to the bias, and stored as a temporary variable, (`n_tmp`, as shown in Figure 4-27). Then the second input is multiplied by its corresponding weight and that product is added to the previously stored temporary variable. The new sum is written over the old sum. Then the third input is multiplied by its corresponding weight, and that product is added to the previously stored temporary variable. The new sum is written over the old sum. This process continues until the stored temporary variable becomes the net input to the transfer function. When that happens, the net input goes through the transfer function and the output of that neuron,(*a_tmp*) is stored in dual-ported RAM as an input to the next layer (see Figure 4-27). The rest of the neurons in the first layer follow the same process. When all of the neuron outputs for the first layer are complete (and stored in dual-ported RAM), the neural network begins calculations for the second layer. Figure 4-26 shows this process for the first neuron. The output of the neuron in the last layer is stored as the car position.

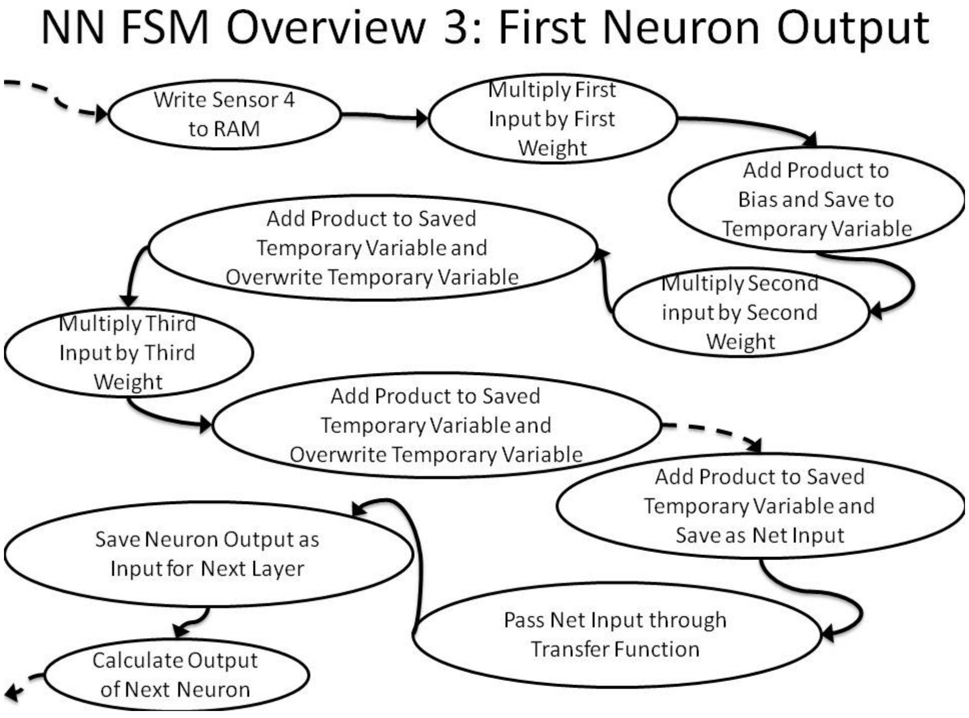

NN FSM Overview 3: First Neuron Output
Write Sensor 4 to RAM
Multiply First Input by First Weight
Add Product to Bias and Save to Temporary Variable
Multiply Second input by Second Weight
Add Product to Saved Temporary Variable and Overwrite Temporary Variable
Multiply Third Input by Third Weight
Add Product to Saved Temporary Variable and Overwrite Temporary Variable
Add Product to Saved Temporary Variable and Save as Net Input
Pass Net Input through Transfer Function
Save Neuron Output as Input for Next Layer
Calculate Output of Next Neuron

Figure 4-26: NN FSM Overview 3: First Neuron Output

The code for the neural network FSM is in appendix C. The master FSM drives the neural network code. The neural network code is a series of case statements, which works the same as a state machine, only with different syntax. Figure 4-27 was provided by Dr. Hagan and shows the architecture of the neural network implemented on the FPGA. The description of the operations illustrated in Figure 4-27 follow in the remainder of this section.

In Figure 4-27 there are several ROMs and a dual-ported RAM that control the operation of the FPGA neural network. The structure ROM contains data that define the network architecture: *R*, the number of inputs to the network (four for the car position application); *S1*, the number of neurons in the first layer; *S2*, the number of neurons in the second layer, etc. The final item in the structure ROM is a delimiter, which indicates that the final layer has been reached. The weight ROM contains all of the weights in the neural network. The bias ROM contains all of the neural network biases. The dual-ported memory is used to store the inputs to the network (the four sensor values, in this case) and the neuron outputs. Before the network calculations begin, the network inputs are loaded into the dual-ported memory. As the neuron outputs in the first layer are computed, they are stored in the dual-ported RAM at locations immediately following the network inputs. After the first layer neuron outputs have all been computed, these outputs act as inputs to the second layer, and the entire process is repeated. This process will be described in more detail in the following.

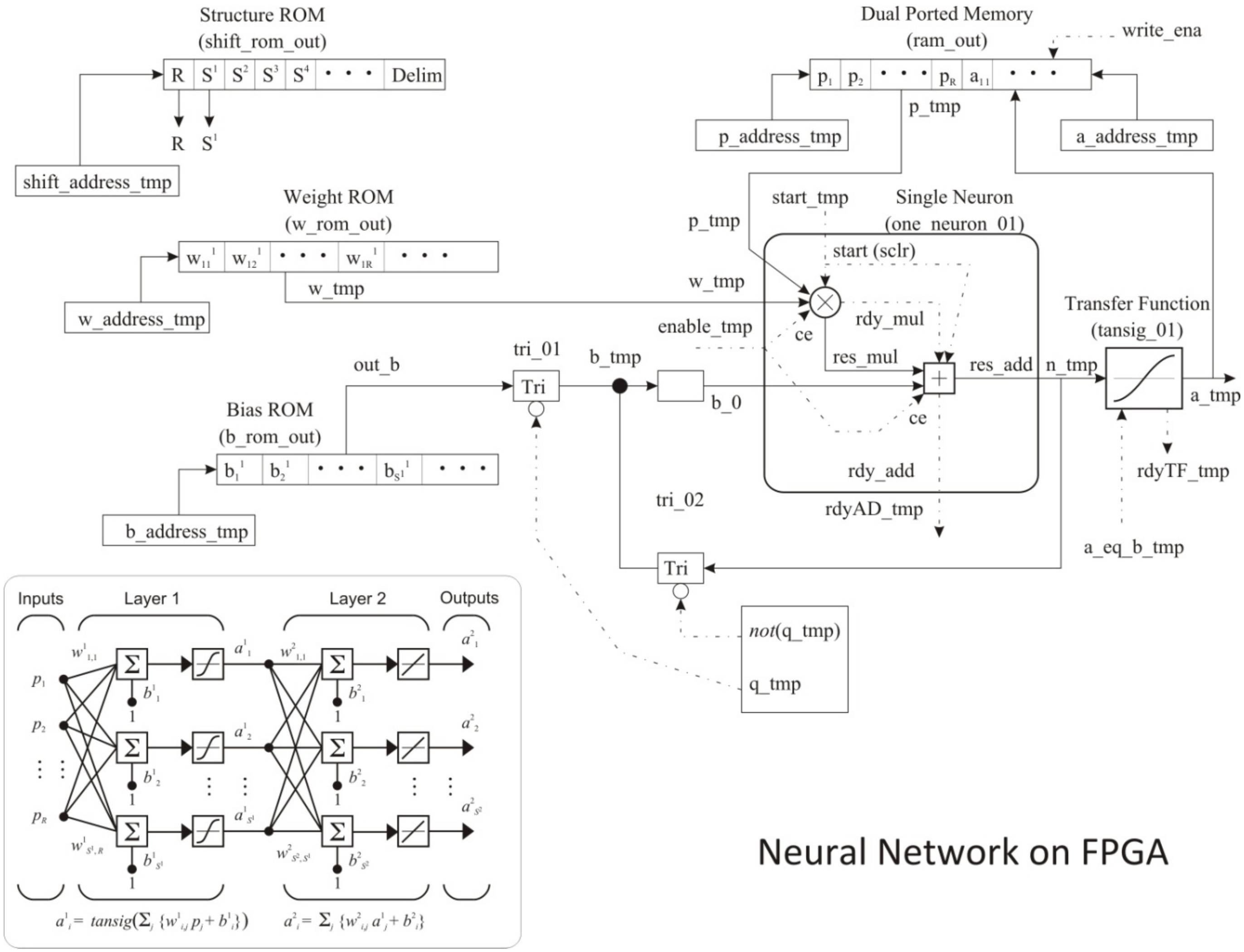
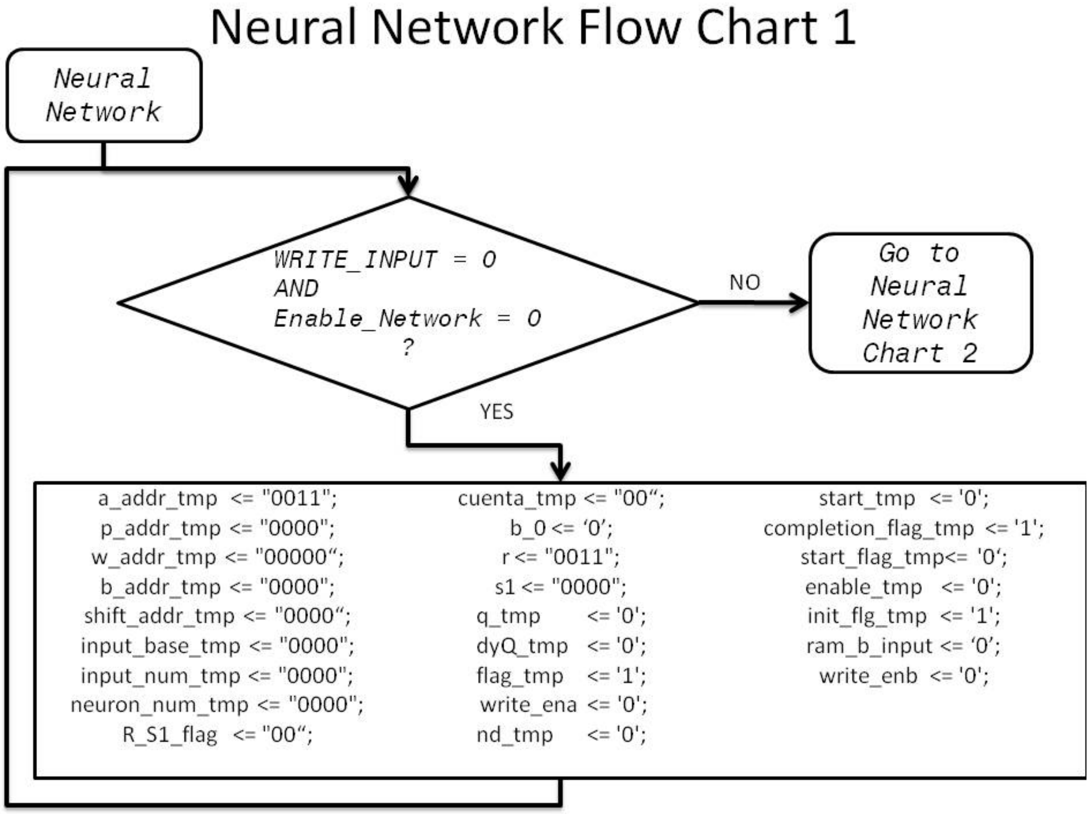


Figure 4-27: Neural Network on FPGA Schematic

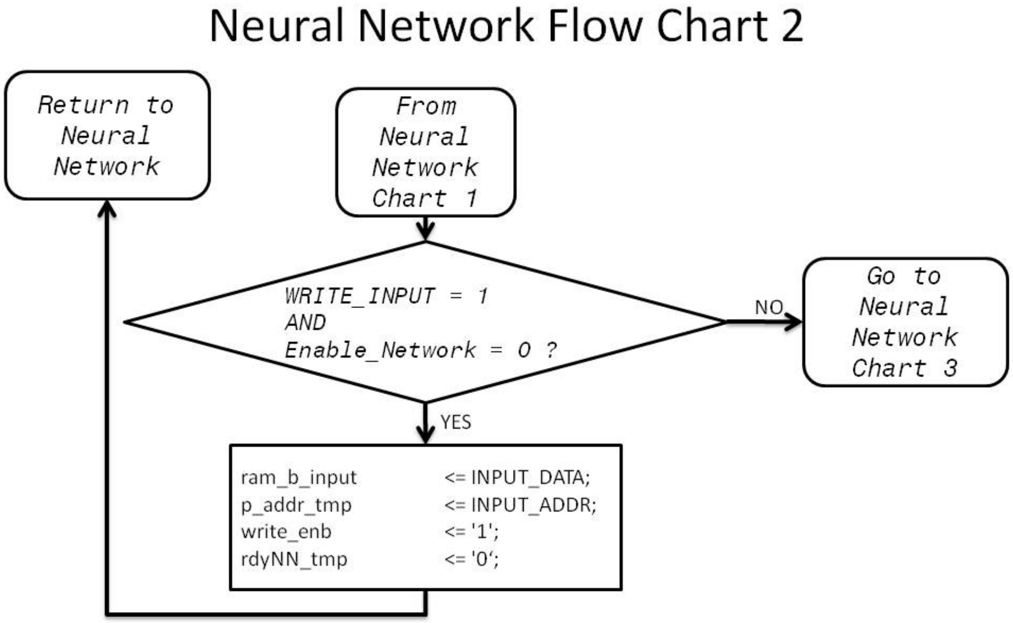
The different case statements in the neural network code can be described with a series of flow charts. The master FSM controls the variables `WRITE_INPUT` and `Enable_Network`. When the variables `WRITE_INPUT` and `Enable_Network` are low, the neural network code initializes a number of different variables in preparation for the calculation. Figure 4-28 is the first flow chart for the neural network code. Many of the variable names in Figure 4-28 are shown in Figure 4-27.



```
a_addr_tmp  <= "0011";
p_addr_tmp <= "0000";
w_addr_tmp <= "00000";
b_addr_tmp <= "0000";
shift_addr_tmp <= "0000";
input_base_tmp <= "0000";
input_num_tmp <= "0000";
neuron_num_tmp <= "0000";
R_S1_flag  <= "00";

cuenta_tmp <= "00";
b_0 <= '0';
r <= "0011";
s1 <= "0000";
q_tmp       <= '0';
dyQ_tmp   <= '0';
flag_tmp    <= '1';
write_ena <= '0';
nd_tmp      <= '0';

start_tmp  <= '0';
completion_flag_tmp  <= '1';
start_flag_tmp<=  '0';
enable_tmp   <= '0';
init_flg_tmp  <= '1';
ram_b_input <= '0';
write_enb  <= '0';
```

Figure 4-28: Neural Network Flow Chart 1

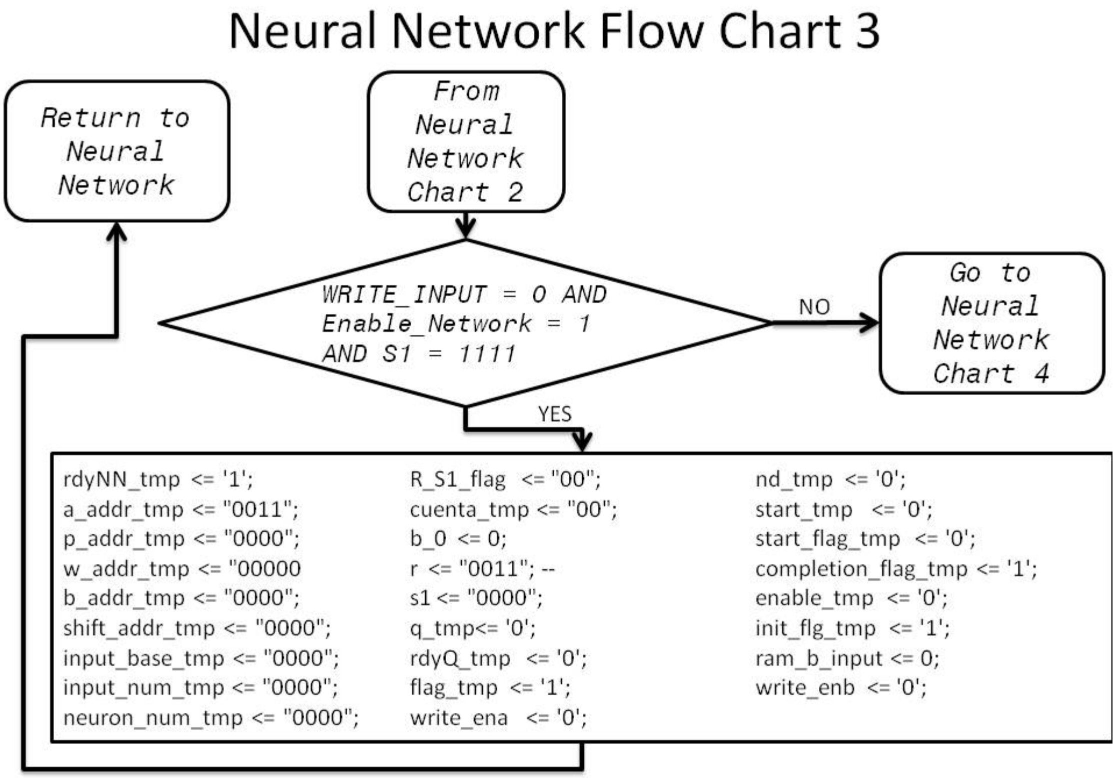
When the variables `WRITE_INPUT` and `ENABLE_Network` are not both low, the code will go into the second flow chart. In this state, when the variable `WRITE_INPUT` is high and the variable `ENABLE_Network` is low, the neural network code is going to record the sensor inputs into the dual ported memory. The variable *ram_b_input* will accept the sensor inputs from the master FSM and store them for the neural network. The variable *p_addr_tmp* tells the neural network which address to store the sensor inputs to in the dual-ported RAM (see Figure 4-27). The master FSM drives the variables INPUT_DATA and INPUT_ADDR. Each input will have a single value and single address. The neural network is in flow chart 2 while the master FSM goes through the states *WRITE_ADC_DATA1*, *WRITE_ADC_DATA2*, *WRITE_ADC_DATA3* and *WRITE_ADC_DATA4*. Figure 4-21 shows that loading each input happens in a single 20 nSec clock cycle, and the RAM loading process takes less than 10 clock cycles. The variable *write_enb* determines if the network inputs can be written to memory or not. The variable *rdyNN_tmp* is a flag that is set high when the network output is ready to be read. Figure 4-29 shows the second neural network flow chart.


Neural Network Flow Chart 2
Return to Neural Network
From Neural Network Chart 1
WRITE_INPUT = 1 AND Enable_Network = 0 ?
NO
Go to Neural Network Chart 3
YES

```
ram_b_input      <= INPUT_DATA;
p_addr_tmp       <= INPUT_ADDR;
write_enb        <= '1';
rdyNN_tmp        <= '0';
```

Figure 4-29: Neural Network Flow Chart 2

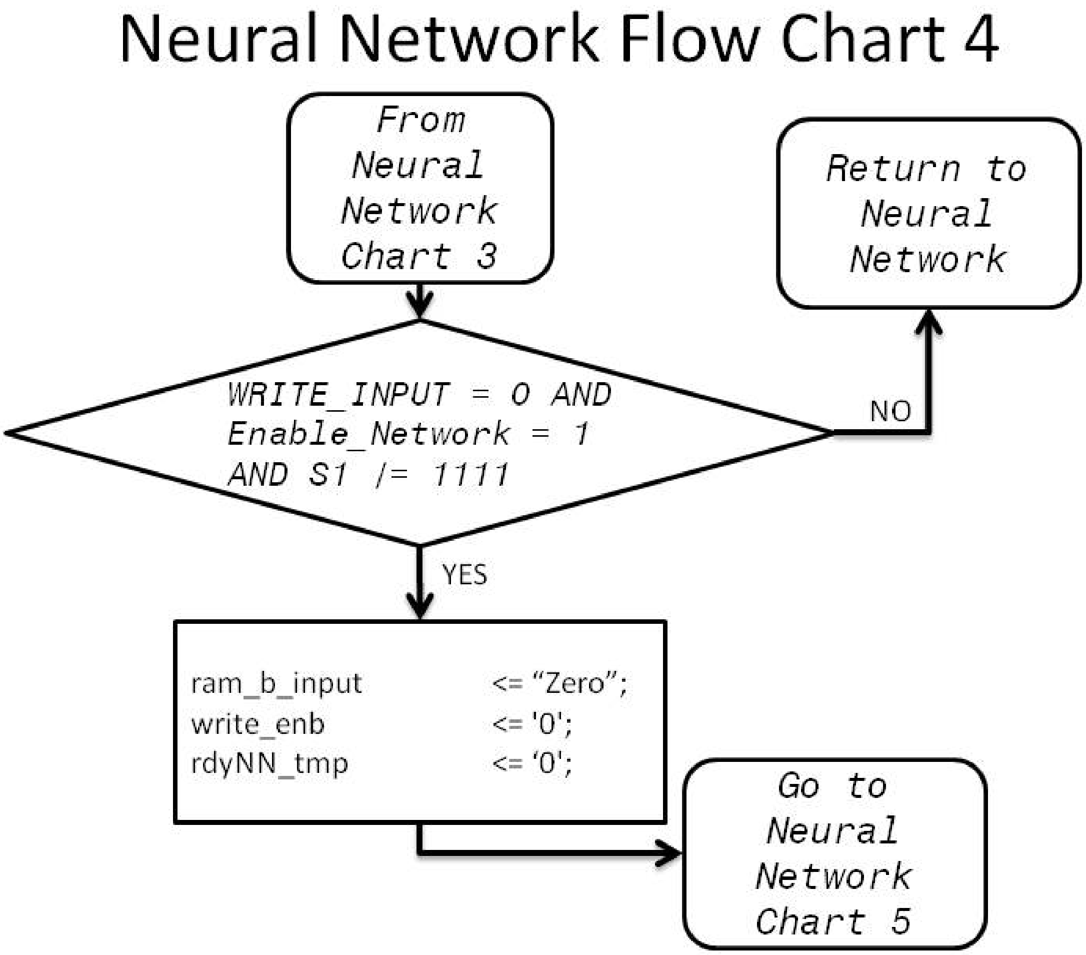
When the master FSM has loaded the inputs to the dual ported memory, it will change the variables `WRITE_INPUT` and `Enable_Network` to different values and put the neural network into a different state. The third flow chart shows a special state where the network has been enabled, and the number of neurons for the layer variable, `s1` (which was loaded from the structure ROM, as in Figure 4-27) is equal to the delimiter value. This is a special flag that the neural network code uses to signal the master FSM that the calculation is complete (the last layer has been reached). During this state, the neural network code resets the neural network variables for the next calculation. Figure 4-30 shows the third flow chart for the neural network.


Neural Network Flow Chart 3
Return to Neural Network
From Neural Network Chart 2
WRITE_INPUT = 0 AND Enable_Network = 1 AND S1 = 1111
NO
Go to Neural Network Chart 4
YES

```
rdyNN_tmp  <= '1';                R_S1_flag  <= "00";         nd_tmp  <= '0';
a_addr_tmp <= "0011";             cuenta_tmp <= "00";         start_tmp   <= '0';
p_addr_tmp <= "0000";             b_0  <= 0;                  start_flag_tmp  <= '0';
w_addr_tmp <= "00000              r  <= "0011"; --            completion_flag_tmp <= '1';
b_addr_tmp <= "0000";             s1 <= "0000";               enable_tmp  <= '0';
shift_addr_tmp <= "0000";         q_tmp<= '0';                init_flg_tmp  <= '1';
input_base_tmp <= "0000";         rdyQ_tmp  <= '0';           ram_b_input <= 0;
input_num_tmp <= "0000";          flag_tmp  <= '1';           write_enb  <= '0';
neuron_num_tmp <= "0000";         write_ena  <= '0';
```

Figure 4-30: Neural Network Flow Chart 3

If the number of neurons is not equal to the delimiter value, the code goes on to the fourth flow chart. In the fourth flow chart, when conditions are correct, the code will set the variable *write_enb* low and the variable *ram_b_input* is set to zero. This will keep the dual ported memory from loading new inputs. Figure 4-31 shows the fourth neural network flow chart.


Neural Network Flow Chart 4
From Neural Network Chart 3
WRITE_INPUT = 0 AND Enable_Network = 1 AND S1 /= 1111
NO
Return to Neural Network
YES
Go to Neural Network Chart 5

```
ram_b_input          <= “Zero”;
write_enb            <= '0';
rdyNN_tmp            <= ‘0';
```

Figure 4-31: Neural Network Flow Chart 4

The fifth flow chart is focused on the buffers. The fifth flow chart shows the code selecting between chart 6, which deals with the timing for switching neuron input buffers, or chart 7, which will decide if the neuron layer structure is loaded or if a neuron output is calculated. The variable *rdyAD_tmp* is driven by the adder in the summation junction and will go high when the current addition is complete and ready to be read. The variable *input_num_tmp* keeps track of when to add in the bias or partial sum for each neuron. When *input_num_tmp* is zero, the neuron will add in the bias. When it is not zero, the partial sum stored from the previous calculation will be stored. The variables *b_0* and *b_tmp* are used with the tri-state buffers, and *b_0* is only updated after the output of the adder is ready. (see Figure 4-27 to see the relationship between *b_0* and *b_tmp*) The variable *flag_tmp* is used with the tri-state buffer to help control timing. Figure 4-32 shows the fifth flow chart.

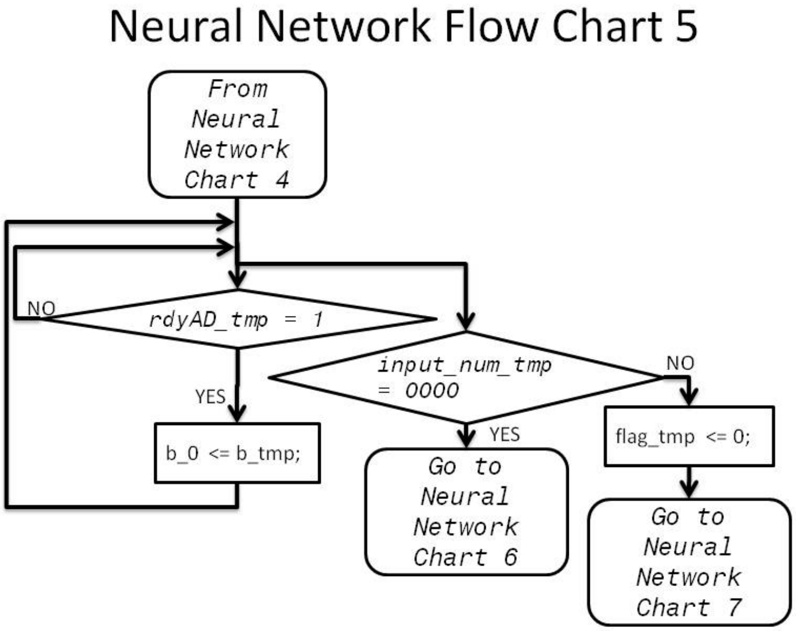

Neural Network Flow Chart 5
From Neural Network Chart 4
NO
rdyAD_tmp = 1
YES
b_0 <= b_tmp;
input_num_tmp = 0000
NO
YES
flag_tmp <= 0;
Go to Neural Network Chart 6
Go to Neural Network Chart 7

Figure 4-32: Neural Network Flow Chart 5

Depending on the condition of the variables, the code will go on to chart 6 or chart 7. The variable `q_tmp` is used with the variable `not_q_tmp` to drive the buffers that help load the variable `b_0`. The first input to the neuron summation is a bias. The second input to the neuron summation will be the partial net input resulting from the first calculation. (see Figure 4-27) To avoid contention, buffers are used to connect to both inputs, but only let one input be used at a time. The variable `rdyQ_tmp` controls the timing of the buffer switching. The variable `flag_tmp` controls the timing of `rdyQ_tmp` . Figure 4-33 shows the sixth neural network flow chart, which deals with the timing for switching neuron input buffers.

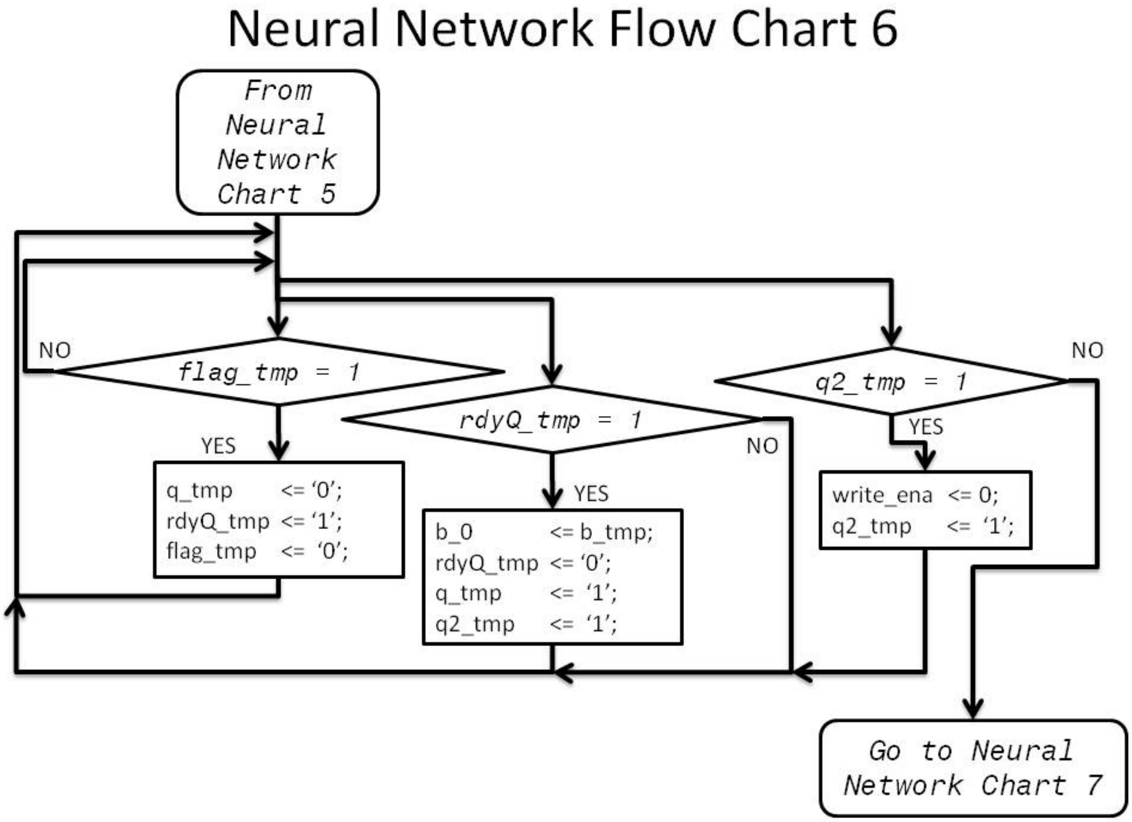


Figure 4-33: Neural Network Flow Chart 6

The seventh flow chart shows how the code selects between loading the variables `r` and `s1` from the structure ROM (see Figure 4-27) or calculating the neuron output. The variable `R_S1_flag` is 00 for loading the number of inputs for the layer (`R`), 01 for loading the number of neurons in the layer (`S1`), or 11 for calculating the network output. Figure 4-34 shows the seventh flow chart.

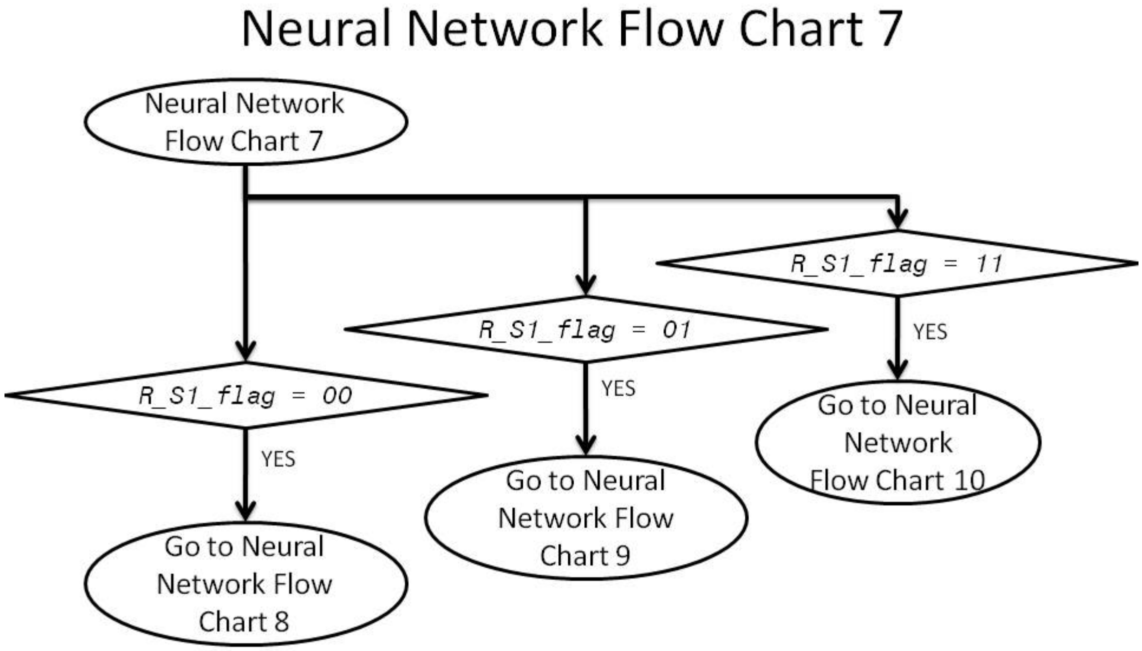

Neural Network Flow Chart 7
Neural Network Flow Chart 7
R_S1_flag = 11
YES
Go to Neural Network Flow Chart 10
R_S1_flag = 01
YES
Go to Neural Network Flow Chart 9
R_S1_flag = 00
YES
Go to Neural Network Flow Chart 8

Figure 4-34: Neural Network Flow Chart 7

In the eighth chart, the code is loading the number of inputs to the layer. The variable `nd_tmp` is used with the shifter ROM (structure ROM in Figure 4-27). It signals that there is valid new data available to be recorded. The variable `rdySH_tmp` identifies when the output of the shifter ROM is valid and ready to be read. The variable `r` is the number of inputs for a layer. For the first layer, `r` will be four inputs from the four sensors, but for the second layer, `r` will be five, indicating the number of neurons in the first layer. (The NN FPGA works for arbitrary numbers of inputs, numbers of neurons and numbers of layers, but for this particular application we are using *`R`* = 4, *`S1`* = 5 and *`S2`* = 1.) The variable *`memory_tmp`* is the output of the shifter ROM. Figure 4-35 shows the eighth flow chart.

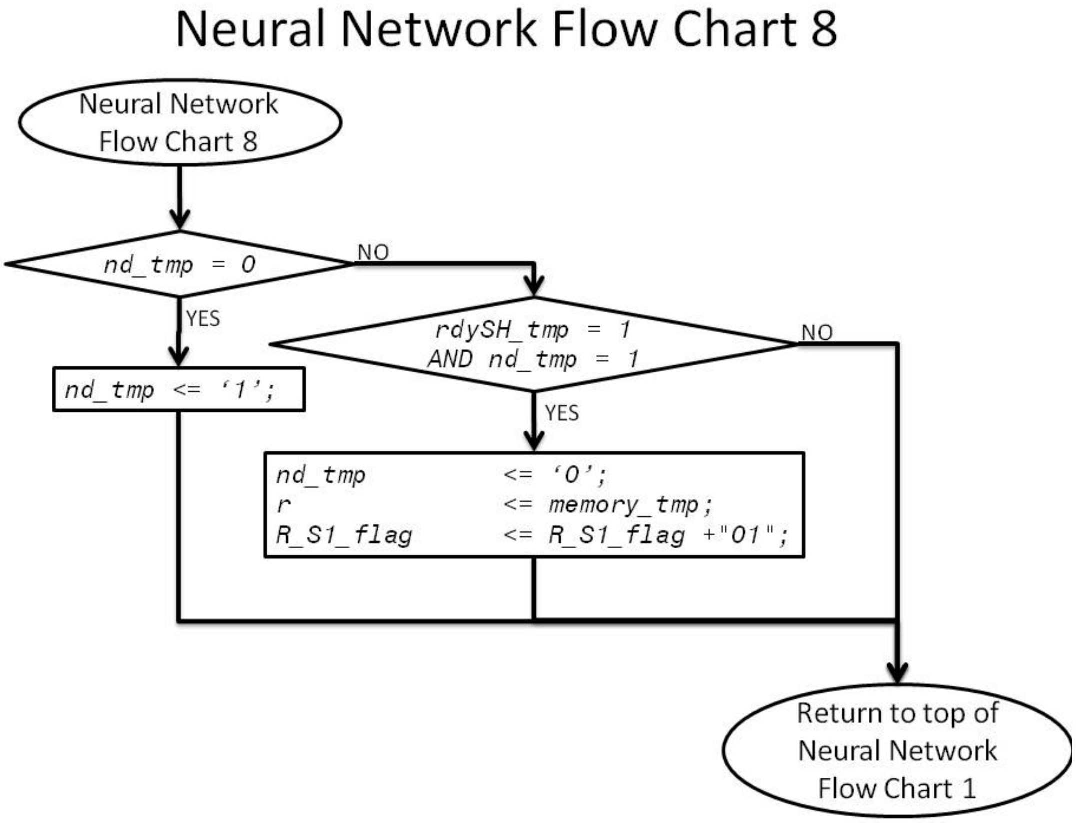

Neural Network Flow Chart 8
Neural Network Flow Chart 8
nd_tmp = 0
NO
YES
nd_tmp <= '1';
rdySH_tmp = 1 AND nd_tmp = 1
NO
YES
nd_tmp <= '0';
r <= memory_tmp;
R_S1_flag <= R_S1_flag +"01";
Return to top of Neural Network Flow Chart 1

Figure 4-35: Neural Network Flow Chart 8

In chart 9, the code is loading the number of neurons in the layer from the shifter ROM (structure ROM in Figure 4-27) into the variable *`s1`*. The variable *`shift_addr_tmp`* keeps track of the shift ROM memory that stores the neural network structure. Inside the shift ROM, for this application, the first address stores a “0011”. Zero corresponds to the first number in the counting sequence, so three in binary corresponds to having four inputs. The second address stores “0100” which is for the five hidden neurons. The third address stores “0000” which corresponds to the one output neuron, and the fourth address stores “1111” which is the delimiter, identifying that the network has only two layers. The variable `R_S1_flag` is “11” so the code will begin to calculate the layer output. Figure 4-36 shows neural network flow chart 9.

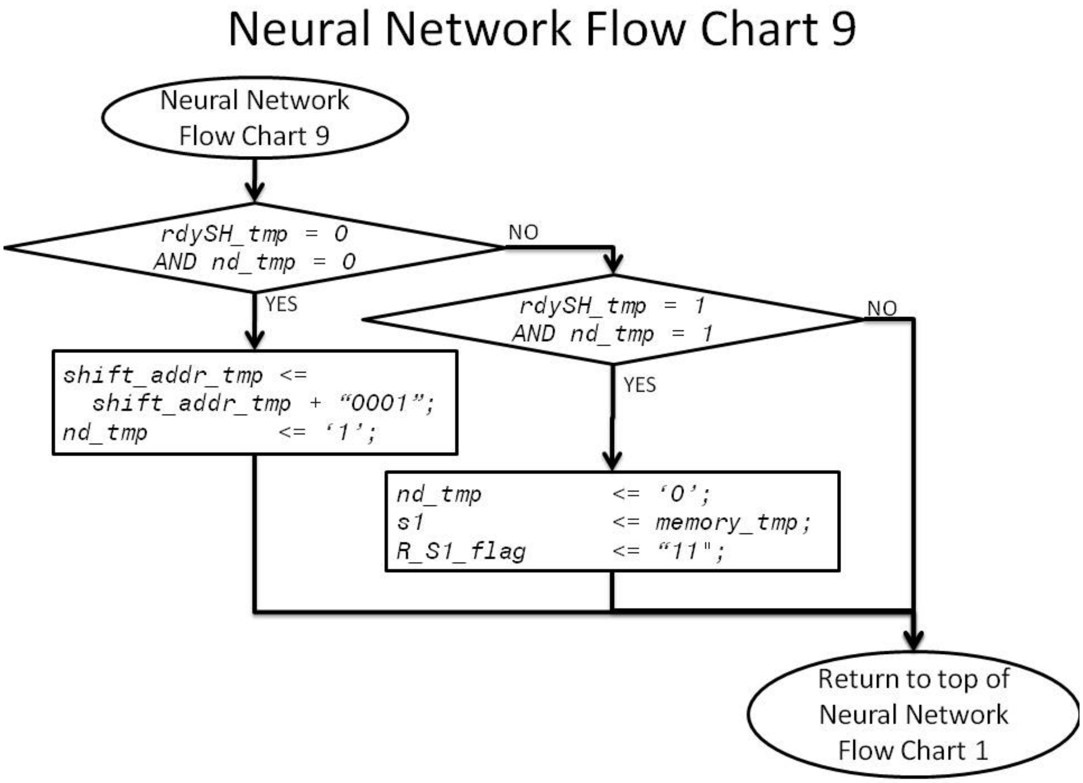

Neural Network Flow Chart 9
Neural Network Flow Chart 9
rdySH_tmp = 0 AND nd_tmp = 0
NO
YES
rdySH_tmp = 1 AND nd_tmp = 1
NO
YES
shift_addr_tmp <= shift_addr_tmp + “0001”;
nd_tmp <= ‘1’;
nd_tmp <= ‘0’;
s1 <= memory_tmp;
R_S1_flag <= “11";
Return to top of Neural Network Flow Chart 1

Figure 4-36: Neural Network Flow Chart 9

Flow charts ten through fourteen show how the code calculates the output of the neuron. If *rdyAD_tmp* is high, the result of the summation in the neuron is ready, so the next calculation can begin. This will cause the variable *start_tmp* to go high which will start the calculation of the next neuron and set the variable *enable_tmp* low which signifies that the output of the neuron is not valid. This gives time after setting the variable *start_flag_tmp* high for all the addresses to settle before the neuron calculation begins. The variable *completion_flag_tmp* will go high when an addition has been completed or if the first calculation is being performed. The variable *enable_tmp* goes high when a neuron output calculation is complete. The variable *cuenta_tmp* is a counter used to add delays, and is counted in binary. Figure 4-37 shows the tenth neural network flow chart.

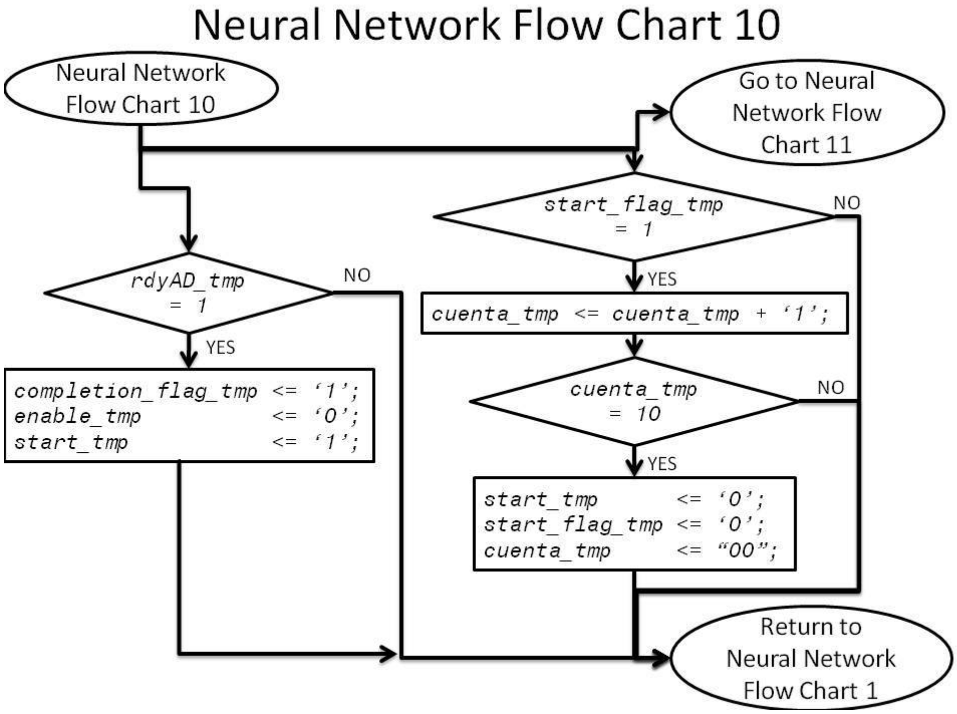

Neural Network Flow Chart 10
Neural Network Flow Chart 10
Go to Neural Network Flow Chart 11
start_flag_tmp = 1
NO
YES
cuenta_tmp <= cuenta_tmp + '1';
rdyAD_tmp = 1
NO
YES
completion_flag_tmp <= '1';
enable_tmp <= '0';
start_tmp <= '1';
cuenta_tmp = 10
NO
YES
start_tmp <= '0';
start_flag_tmp <= '0';
cuenta_tmp <= "00";
Return to Neural Network Flow Chart 1

Figure 4-37: Neural Network Flow Chart 10

Neural network flow chart eleven hinges on the variable `completion_flag_tmp`. This variable will be high if the summation output is available and valid or if the first calculation for a neuron is being performed. The variable `init_flag_tmp` identifies if the first calculation for a neuron is being performed. If the first calculation for a neuron is being performed, variables are set such that the neuron bias is loaded into the summation junction (`b_tmp` is set to `out_b`, as shown in Figure 4-27). Else, the variable `init_flag_tmp` is low and the partial net input is fed into the summation(`n_tmp` in Figure 4-27). Figure 4-38 shows neural network flow chart eleven.

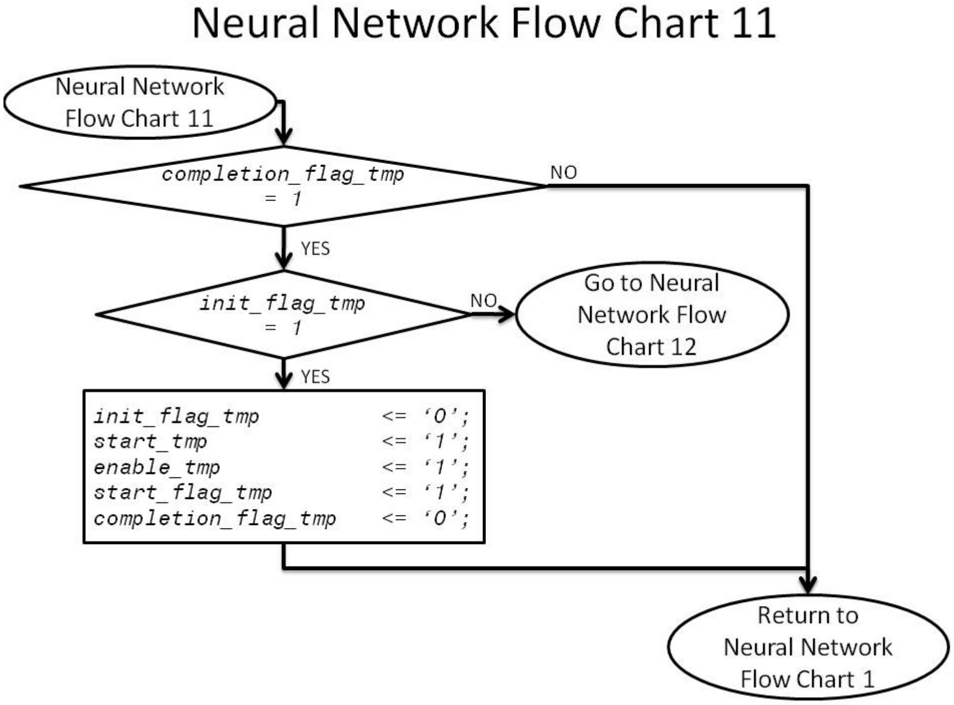

Neural Network Flow Chart 11
Neural Network Flow Chart 11
completion_flag_tmp = 1
NO
YES
init_flag_tmp = 1
NO
Go to Neural Network Flow Chart 12
YES
init_flag_tmp <= '0';
start_tmp <= '1';
enable_tmp <= '1';
start_flag_tmp <= '1';
completion_flag_tmp <= '0';
Return to Neural Network Flow Chart 1

Figure 4-38: Neural Network Flow Chart 11

The twelfth neural network flow chart shows decisions based on two variables. The variable `input_num_tmp` tells the code to add in the bias on the first input for a neuron, and when it equals the variable `r`, the last input for that neuron has been loaded. The variable `rdyTF_tmp` is a flag to identify when the transfer function output is available. The transfer function is implemented with a look up table. The input to the transfer function serves as the address for the look up table. At the corresponding address, the output of the transfer function is stored. The variable `a_eq_b_tmp` is a flag that triggers the transfer function look up table. When it is high, the look up table input is a valid address. When the variable `rdyTF_tmp` is high, the neuron output is ready. The variable `w_addr_tmp` is the weight address, which is incremented every cycle through flow chart twelve. Figure 4-39 shows neural network flow chart 12.

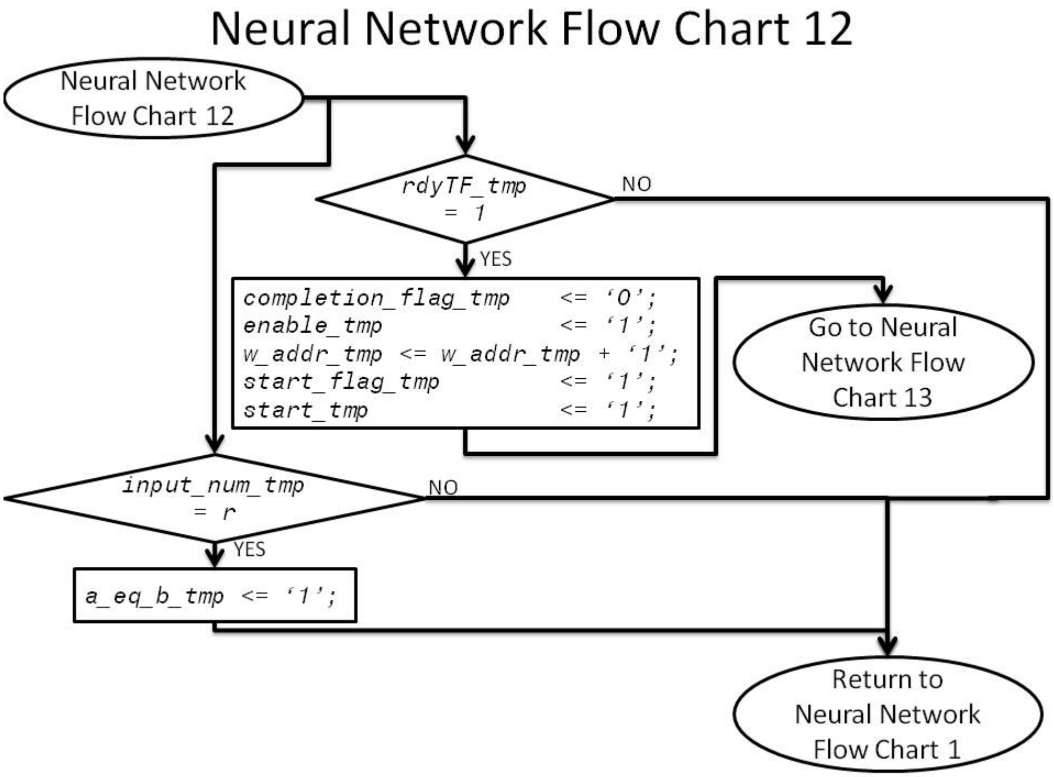
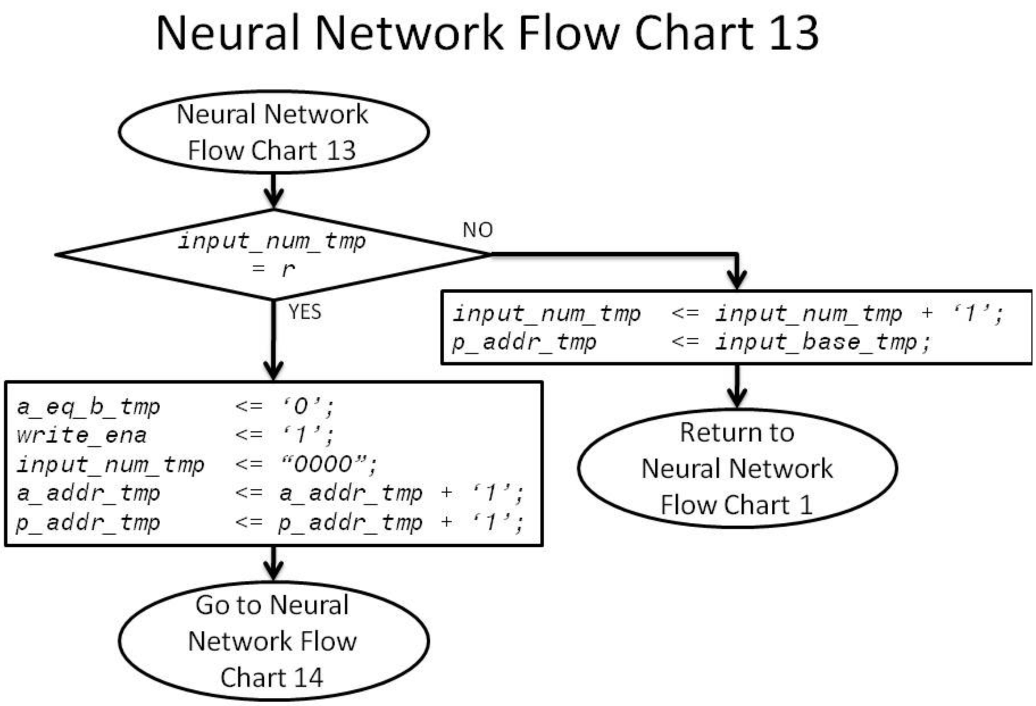

Neural Network Flow Chart 12
Neural Network Flow Chart 12
rdyTF_tmp = 1
NO
YES
completion_flag_tmp <= '0';
enable_tmp <= '1';
w_addr_tmp <= w_addr_tmp + '1';
start_flag_tmp <= '1';
start_tmp <= '1';
Go to Neural Network Flow Chart 13
input_num_tmp = r
NO
YES
a_eq_b_tmp <= '1';
Return to Neural Network Flow Chart 1

Figure 4-39: Neural Network Flow Chart 12

Given that conditions are correct, after the code completes setting the variables in chart 12, it will go on to chart 13. In flow chart 13, if the neuron is on the last input, the code will write the output to the dual ported memory (`a_tmp` in Figure 4-27), increment the addresses for the neuron input and output and set the input number counter variable to zero. The variable `a_addr_tmp` tells what address the neuron output will be stored in. The variable `p_addr_tmp` tells what address the neuron input will be retrieved from (see Figure 4-27). The variable `write_ena` determines if the dual ported memory will accept new neuron outputs. The variable `input_base_tmp` keeps track of which input is to be added next. If the neuron has not added all the inputs, the input number counter variable is incremented, and the input address is set back to its base value for that layer. Figure 4-40 shows neural network flow chart 13.

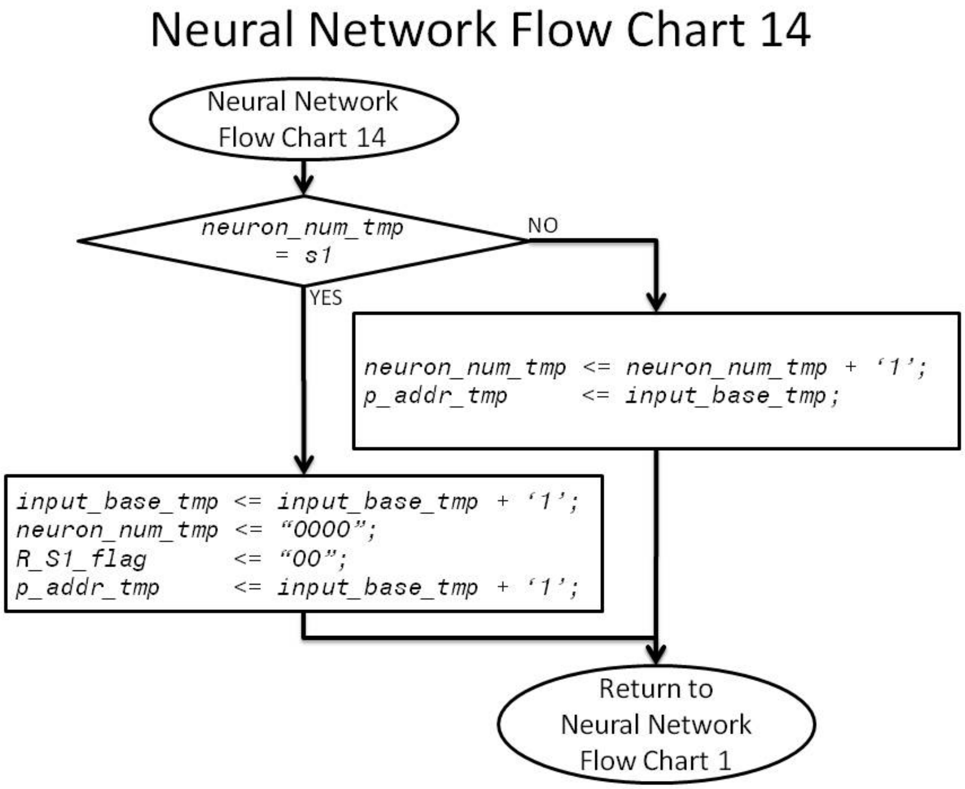

Neural Network Flow Chart 13
Neural Network Flow Chart 13
input_num_tmp = r
NO
YES
Return to Neural Network Flow Chart 1
Go to Neural Network Flow Chart 14

```
input_num_tmp   <= input_num_tmp + '1';
p_addr_tmp      <= input_base_tmp;
```

```
a_eq_b_tmp      <= '0';
write_ena       <= '1';
input_num_tmp   <= "0000";
a_addr_tmp      <= a_addr_tmp + '1';
p_addr_tmp      <= p_addr_tmp + '1';
```

Figure 4-40: Neural Network Flow Chart 13

If the neuron is on the last input in flow chart 13, then the code will move on to flow chart 14. The variable `neuron_num_tmp` keeps track of which neuron output is being calculated. In flow chart 14, the code checks to see if it has calculated the last neuron in the layer. If so, it will reset the variable *`R_S1_flag`* so the code can load the architecture for the next layer, and shifts over the input address so that the outputs of the finished layer become the inputs to the next layer. If the last neuron output for the layer has not been calculated, the input address is reset to the first input address for that layer and the neuron number counter is incremented to the next neuron. Figure 4-41 shows neural network flow chart 14.


Neural Network Flow Chart 14
Neural Network Flow Chart 14
neuron_num_tmp = s1
NO
YES
Return to Neural Network Flow Chart 1

```
neuron_num_tmp <= neuron_num_tmp + '1';
p_addr_tmp     <= input_base_tmp;
```

```
input_base_tmp <= input_base_tmp + '1';
neuron_num_tmp <= "0000";
R_S1_flag      <= "00";
p_addr_tmp     <= input_base_tmp + '1';
```

Figure 4-41: Neural Network Flow Chart 14

The entire neural network code is built around hardware for a single neuron. That hardware is going to multiply two numbers, a weight and input, and then add the product to a bias to produce a partial net input to the transfer function. Depending on how variables are set inside the neural network code, different numbers are loaded into the inputs, weights and biases. Figure 4-42 shows the timing diagram for the hardware. This diagram shows the delay between enabling the hardware, starting the calculation and the calculation being ready. In the timing diagram, the variables *START* and *CE* show when the calculation is started and when the multiplier and adder are enabled. The variables *B*, *W* and *P* are the weight, bias and input. The variable *N* is the output. The input, weight, bias and output are all in a 16-bit floating point format. The variable *RDY_N* is set high when the output is available.

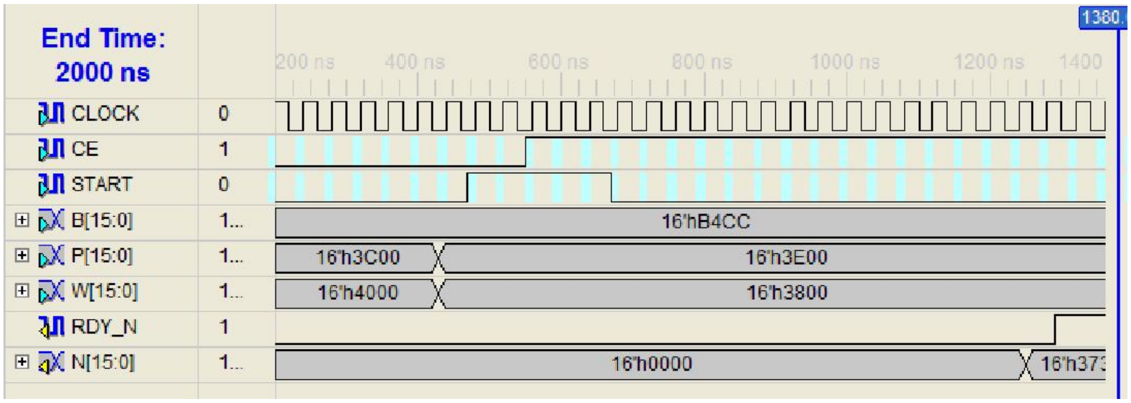


Figure 4-42: Example Timing Diagram for one Neuron

The neural network flow charts describe the process to calculate the position. The calculation is a serial process in which a single hardware neuron calculation is repeated as many times as needed to complete the full network calculation. Figure 4-43 shows a Chipscope waveform displaying most of the signals for this calculation. The black vertical lines signify the end of the calculations for the output of one part of a neural network and the beginning of the next calculation. Figure 4-44 shows the waveform for the calculation of the first neuron in the first layer. In Figure 4-44, the black lines separate calculations of the four partial sums for the four different inputs to the first neuron. Figure 4-45 shows the waveform for the first input to the first neuron of the first layer.

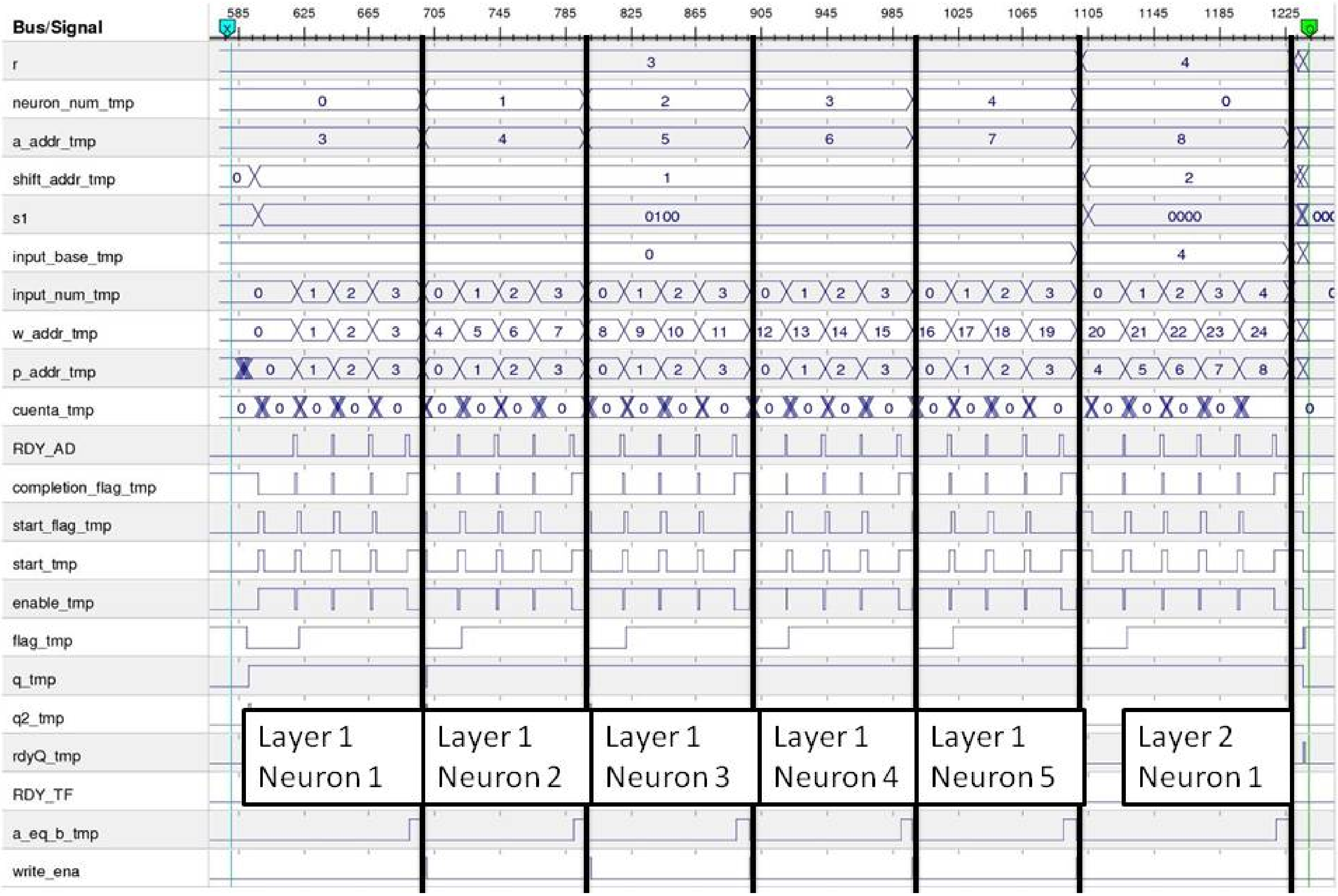


Figure 4-43: Overview of Neural Network Calculation

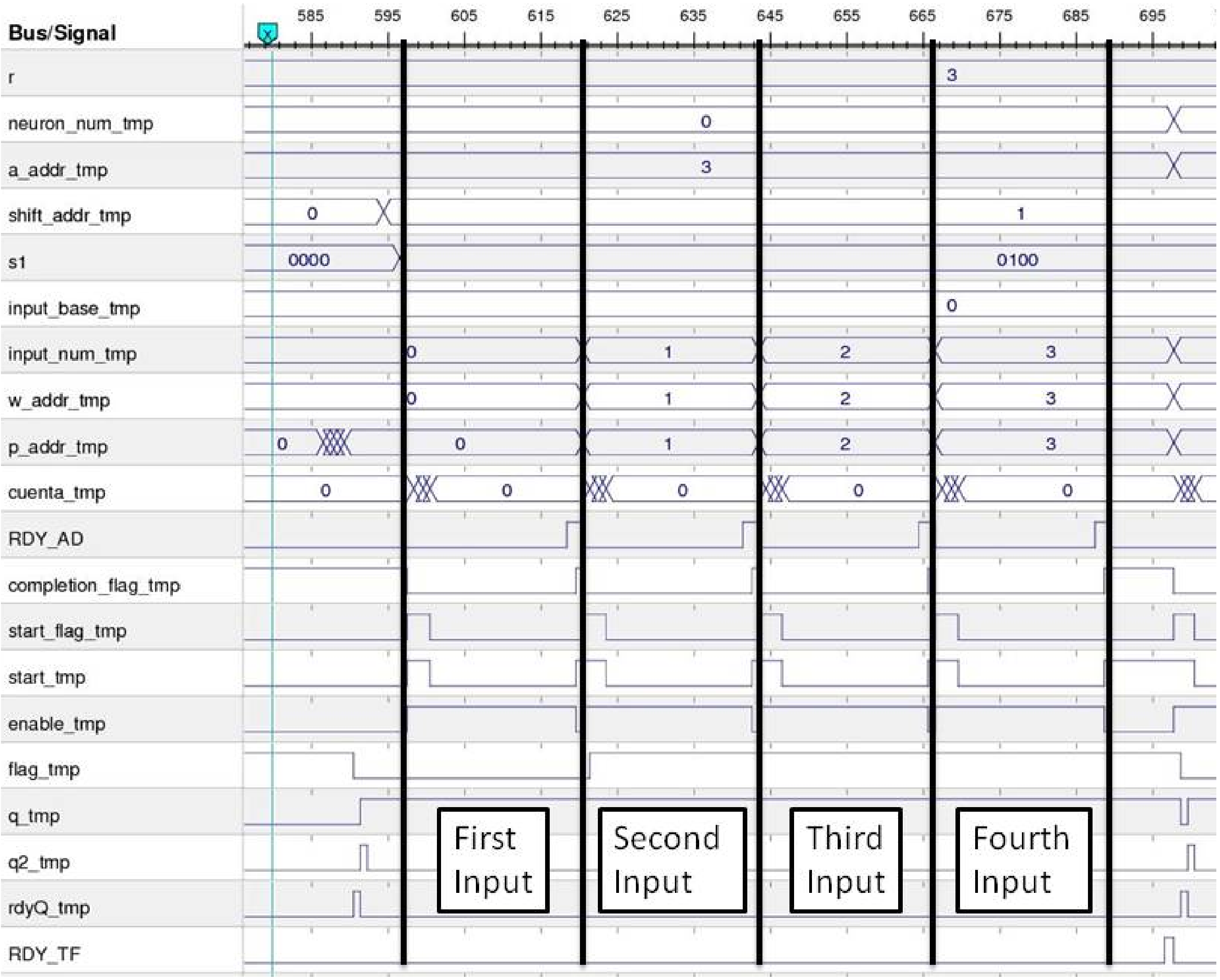


Figure 4-44: ChipScope Waveform for Calculating Output of First Neuron in First Layer

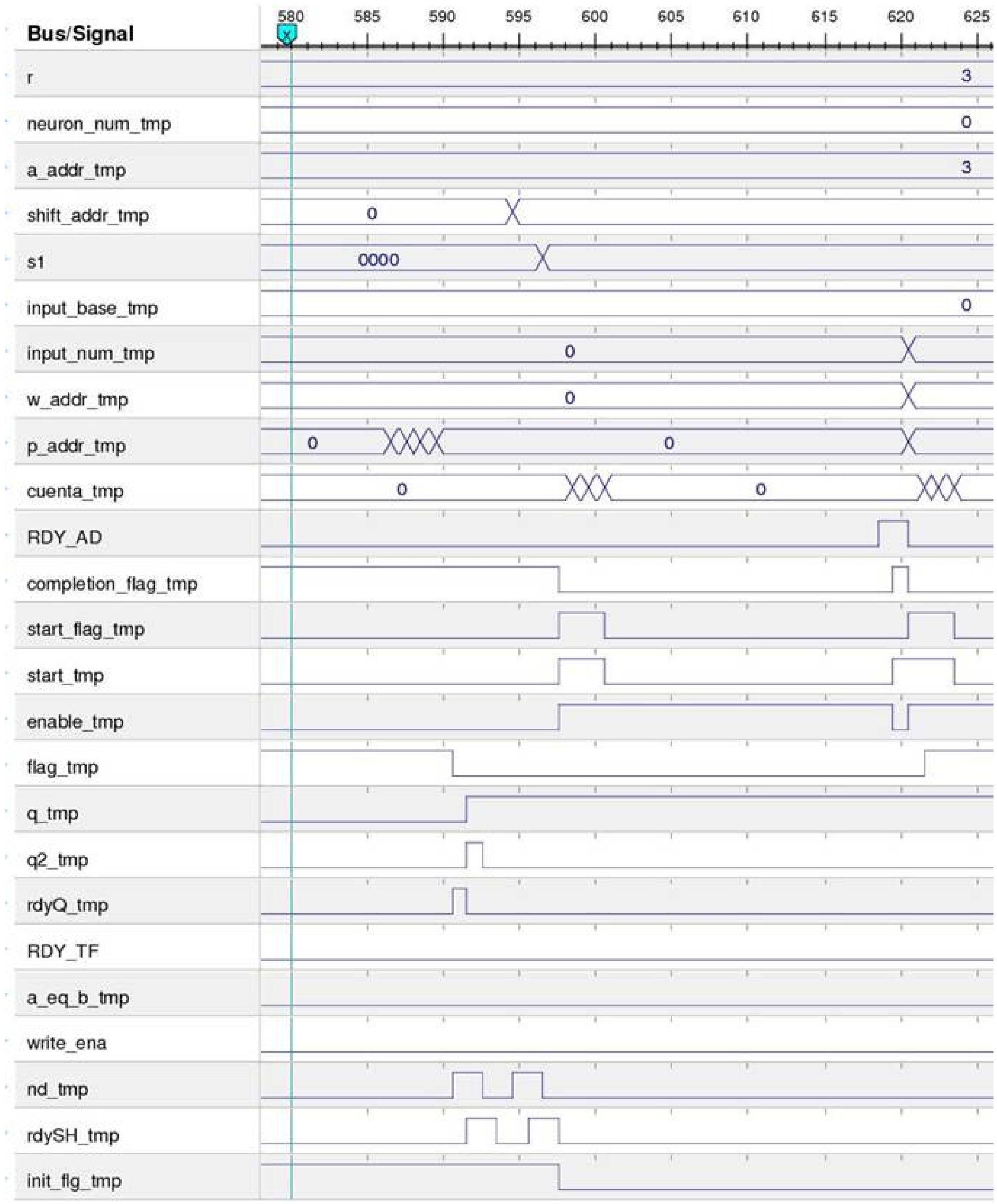


Figure 4-45: First Input to First Neuron in First Layer

In summary, the neural network completes the calculation of the car position using four inputs. Those inputs come from four electromagnetic sensors. The serial calculation is completed with one multiplier, one adder, two buffers and several memories. One layer is calculated at a time, and inside each layer, one neuron output is calculated at a time. To calculate the neuron output, one input is multiplied by its weight and added to the bias on the first calculation or the partial net input for the remaining calculations for that input. Once a complete net input is available for a neuron, it is passed through a look up table, which replaces the transfer function. The outputs of the first layer are stored as the inputs to the second layer. The output of the second layer is the car position.

4.3.3 Credit for Previous Work to Write the Neural Network Code

The original neural network code was a group effort between Professors Martin Hagan and Carl Latino from Oklahoma State University and Professor Marco A. Moreno-Armendariz from Instituto Politecnico Nacional, Mexico.

Taylor York, a former Oklahoma State University student, provided most of the code for the master FSM. Amanuel Assefa, another Oklahoma State University student, provided changes to the code to increase the number of sensors read to four.

4.4 Training the Neural Network

The neural network maps the four sensor readings into the car position. The neural network can be thought of as an equation. The sensor readings are the input to the equation and the car position is the output. The MLP network is a universal approximator, and given enough parameters or degrees of freedom, it can approximate nearly any relationship. To approximate a single set of data, mapping a set of inputs to outputs, requires a specific set of parameters. Training the neural network will find the correct parameters. The training cycle has three key phases. The first phase is data collection. The second phase is adjusting the function parameters. The third phase is validating the network performance. If the validation phase reveals the network performance is not acceptable, training will loop back to data collection or parameter adjustment as needed. This is followed by another phase of network performance validation.

4.4.1 Data Collection

Data collection gathers sensor inputs and corresponding car positions. One data point is four sensor readings and one car position measurement. The data collection process requires a fixture to allow for consistent readings. The smart car has foam holding a marker taped to the front of the sensor board for data collection. The marker indicates the center of the car, which is used as the car position. Tape holds the wire to the floor, and tape holds a ruler over the top of the wire. The wire goes under the ruler at the four-inch mark. The smart car position of four inches is directly on top of the wire. Figure 4-46 shows the smart car ready to collect a single data point.

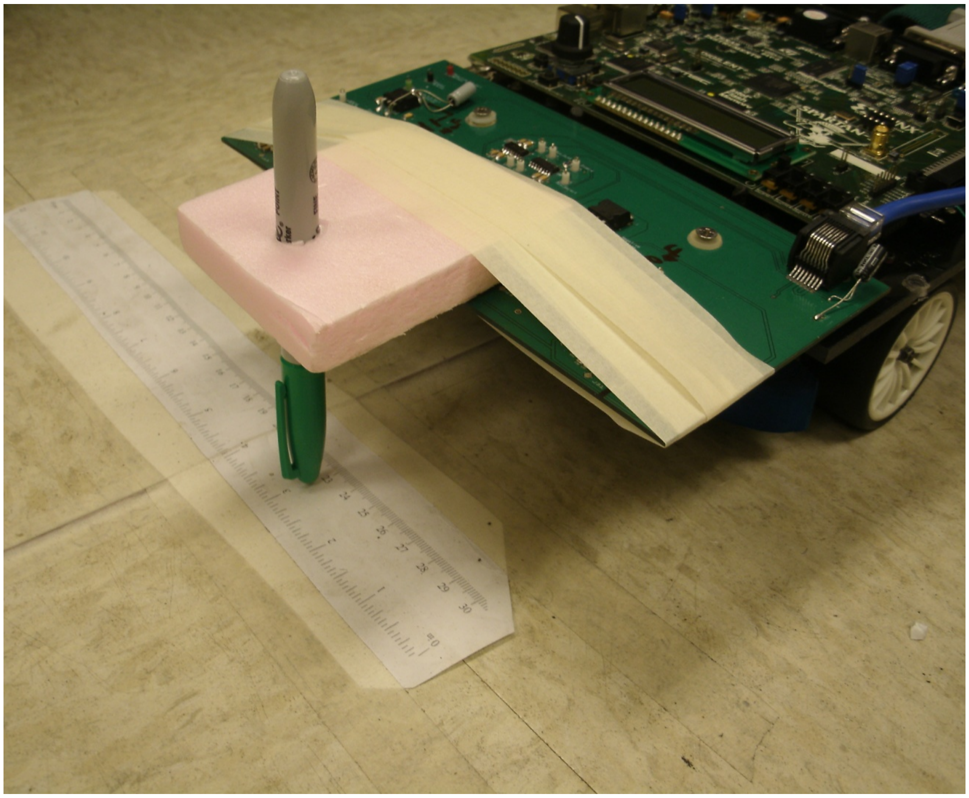


Figure 4-46: Smart Sensor Car for Data Collection

A training set for the smart sensor car spans eight inches, with measurements every quarter of an inch. This data set is large enough to ensure the smart car will have accurate position measurements, but small enough that the students in the summer academy can complete the task in a reasonable amount of time. Capturing several samples at each position reduces the effect of small disturbances on the sensor readings and makes for a more robust training set. At each measurement, 20 samples are taken. Figure 4-47 shows a complete set of training data. Sensor 4 has the most negative reading around 0.75 inches; sensor 2 has the bottom of its valley around 2.75 inches, sensor 1 at 4.75 inches and sensor 3 at 6.75 inches. The bottom of the valley occurs when the sensor is directly over the wire.

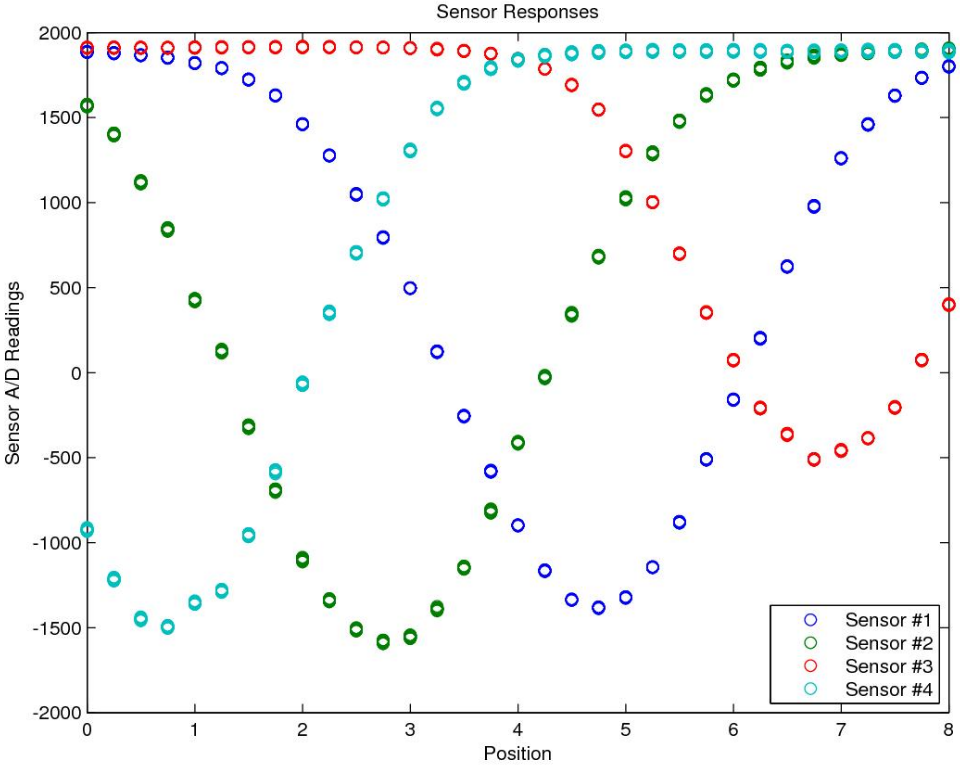


Figure 4-47: Sensor Responses versus Position

4.4.2 Training

One measure of neural network performance is the mean square position error over all data points. The goal of training the neural network is to adjust the parameters (weights and biases) in such a way as to reduce the mean square error. This is a standard unconstrained optimization problem, and any optimization method can be used. We used the Levenberg Marquardt optimization algorithm, as implemented in the Neural Network toolbox for MATLAB. Figure 4-48 shows how the mean square error improves during training. After 465 iterations (epochs), the neural network fits the training data as best it can. This means the network is trained.

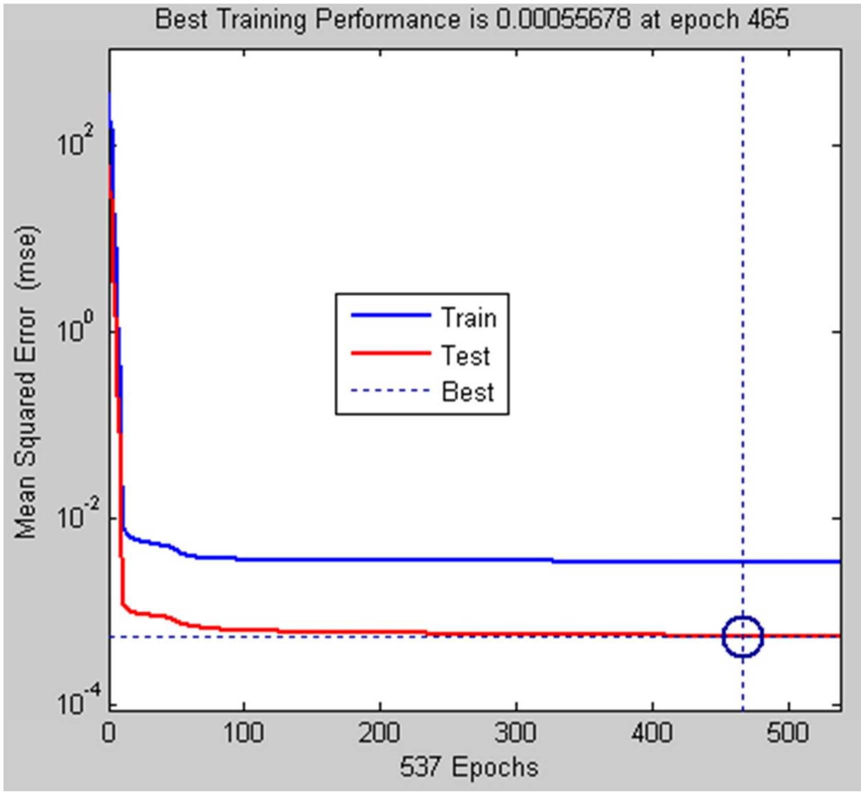


Figure 4-48: Mean Squared Error versus Training Epochs

4.4.3 Network Validation

A trained network produces calculated outcomes that closely match the true positions at the training points. Several steps go into the validation process. First, there needs to be training data over the full range of desired inputs. If the training data appears to be insufficient, additional data is collected, focusing on the problem areas. Figure 4-47 shows that training data is available for all areas the neural network is required to perform(4 inches on either side of the wire). Figure 4-49 shows neural network performance that appears acceptable. The network output and the true position are equal over the range from 0 to 8.

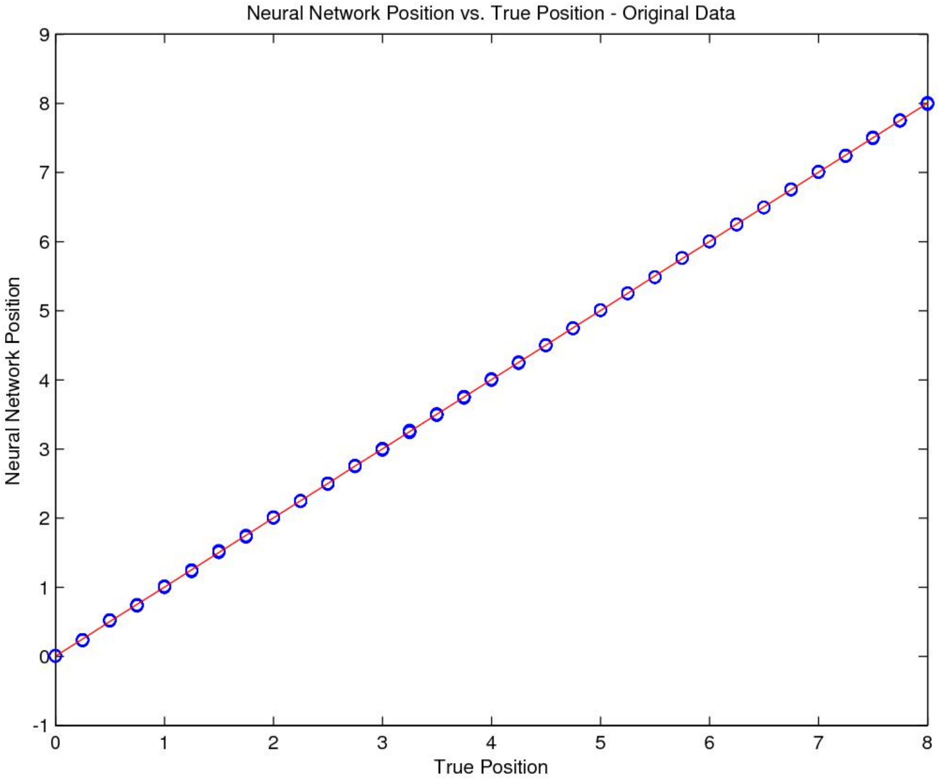


Figure 4-49: Neural Network Position versus True Position

Figure 4-50 shows the readings from sensor 2 at car positions of 2.75, 3 and 3.25 inches. Note the amount of noise on the sensor readings. The noise results in a vertical stack of points at each horizontal position.

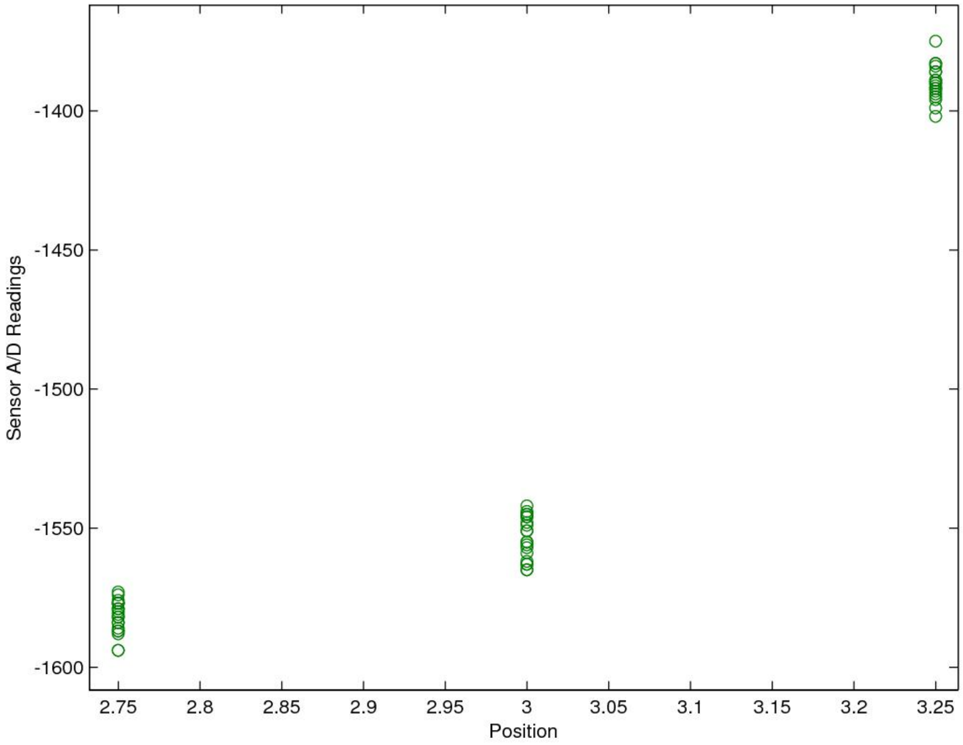


Figure 4-50: Sensor 2 Reading at 3 Inches

Because there is noise on the sensor readings, the neural network will have different inputs that should generate the same output. Noise on the system gives the training algorithm several points to match where readings were taken, and no information about what to do between measurements. In an effort to reach more points, a training algorithm can configure a neural network in an undesirable way. One way to check performance between training points is to use a model to estimate data points between the training points. Because the curves for the smart sensor car are mostly smooth, one option for modeling the data is interpolation. Linear interpolation is used to generate sensor inputs between points. The sensor inputs are averaged at each measurement. A line is fit to the averages of each point. Then additional inputs are evenly spaced on the line between points. The new interpolated sensor inputs are then passed through the neural network. Because the interpolated inputs are mostly on a straight line between measured inputs, the neural network outputs should also be near a straight line. Figure 4-51 shows the neural network response with the interpolated data. The response between measurements is still close to a straight line.

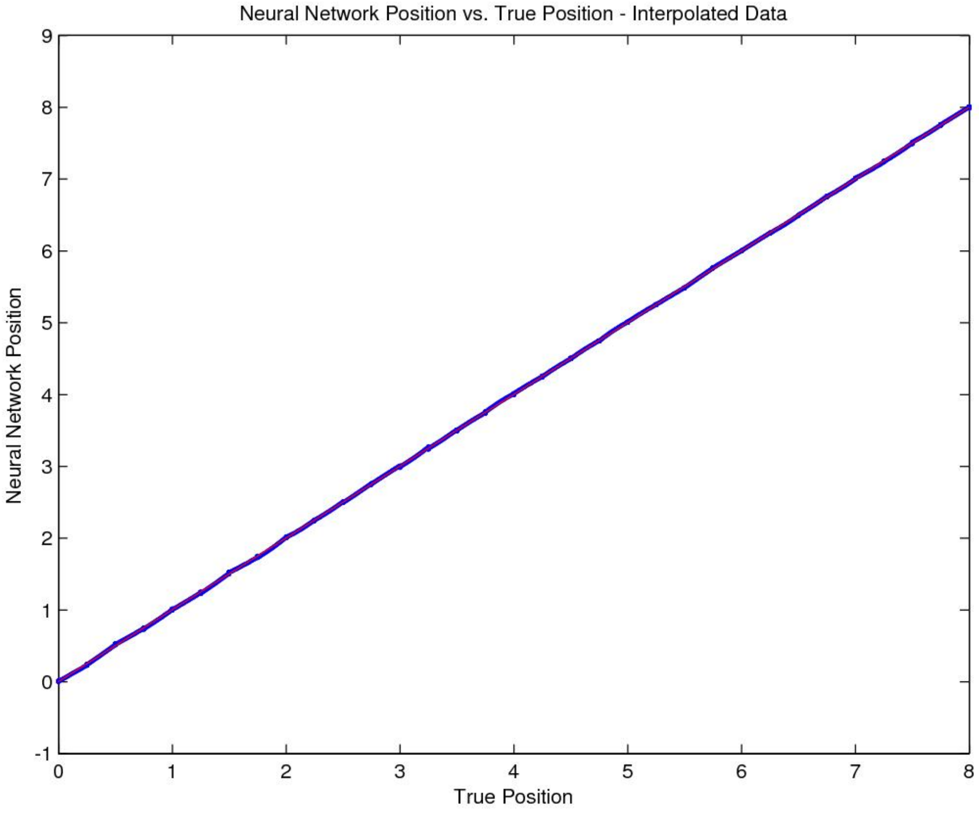


Figure 4-51: Neural Network Position versus True Position with Interpolated Data

Figure 4-52 is zoomed in to a small region of Figure 4-49. Note that the curve appears as a line passing through the averages of each measurement. This means the network is not over-fitting the data.

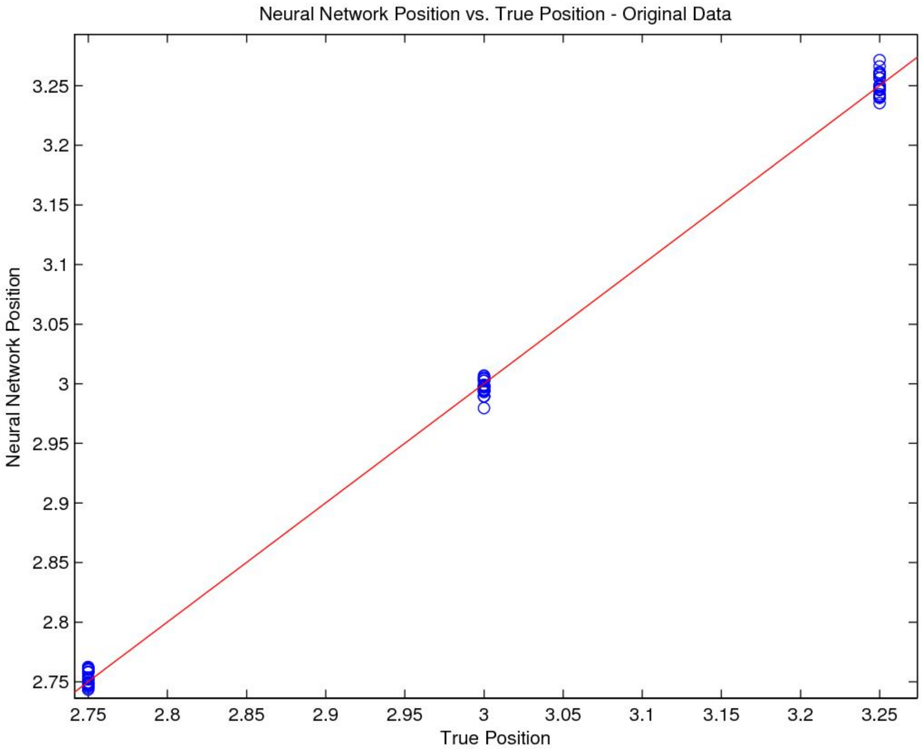


Figure 4-52: Neural Network Position versus True Position, Zoomed in at 3 Inches

4.5 Neural Network Supporting Components

Support equipment is not part of the system functional diagram when the car goes around the track, but is necessary for making some of the pieces work. The purpose of the training support equipment is to collect meaningful data using the same hardware the car uses to go around the track. In order to collect data with the master FSM and existing hardware, a computer replaces the PD controller. That computer runs custom data collection automation software. The code for the A/D FSM and master FSM were written, verified, compiled, and loaded onto the hardware using the Xilinx ISE design suite.

4.5.1 Master FSM Modifications for Training

Training the neural network requires training data. That training data includes the four digital sensor readings and the position. The position measurement comes from a ruler taped to the floor. Figure 4-53 shows the smart car at a position of three inches.

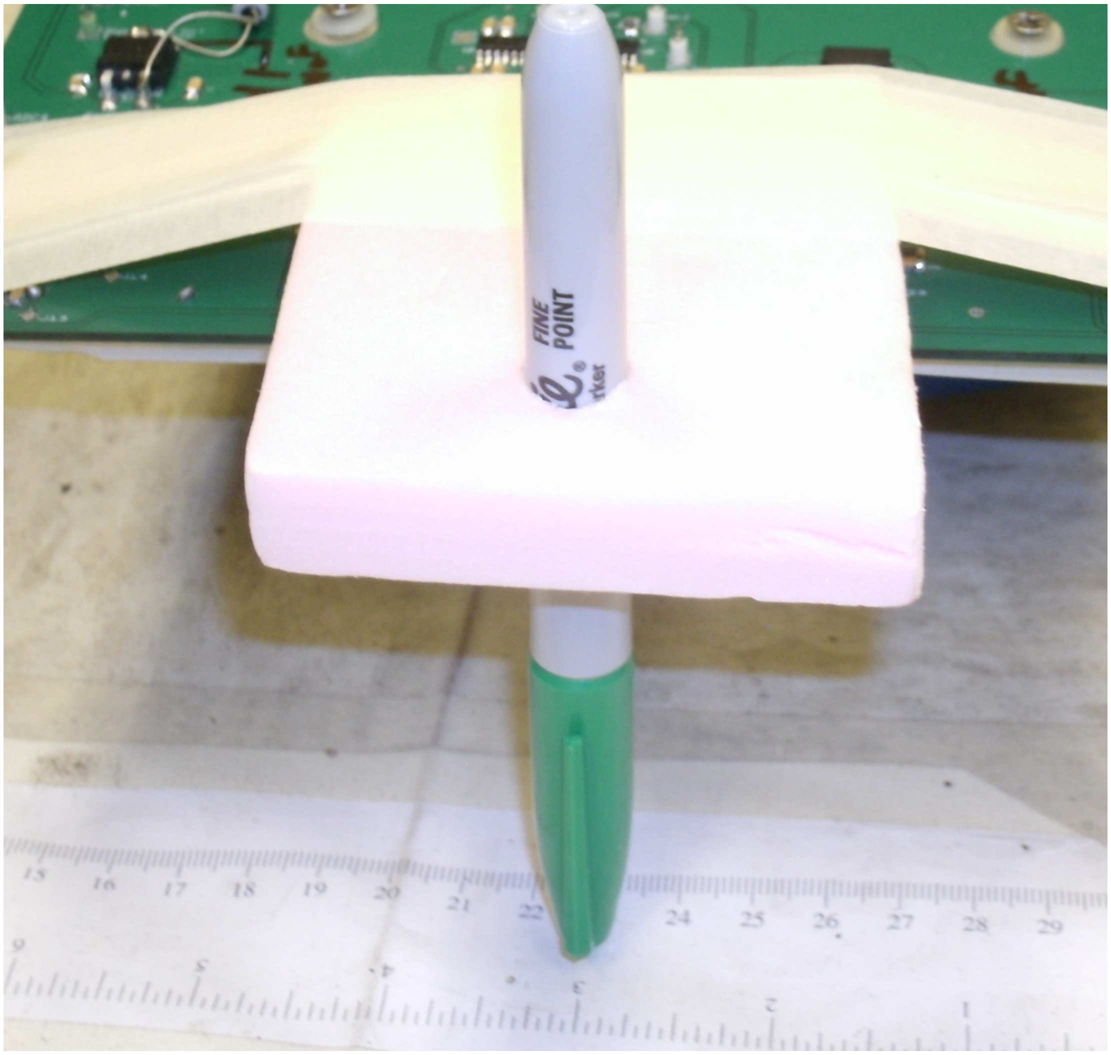


Figure 4-53: Car position of Three inches

The digital sensor readings come from the A/D FSM. The first modification to the master FSM lets it know if the request for new information is for a position calculation from the neural network or for training data from the A/D FSM. This modification happens in the state *WAIT_RECEIVE*. If the master FSM reads in symbols representing the letter "t" then it will send out the four sensor readings. If the letter "r" is read, it will transmit a position. The next step is the same for position calculation and data collection. The master FSM cycles the A/D converter to get four new digital sensor readings. When the digital sensor readings are complete, train mode will start to broadcast the positions over the RS232 communication. The code for this process is in Appendix D: Code for Master FSM to Transmit Training Data. Figure 4-54 is the data collection process overview showing how the computer and master FSM interact.

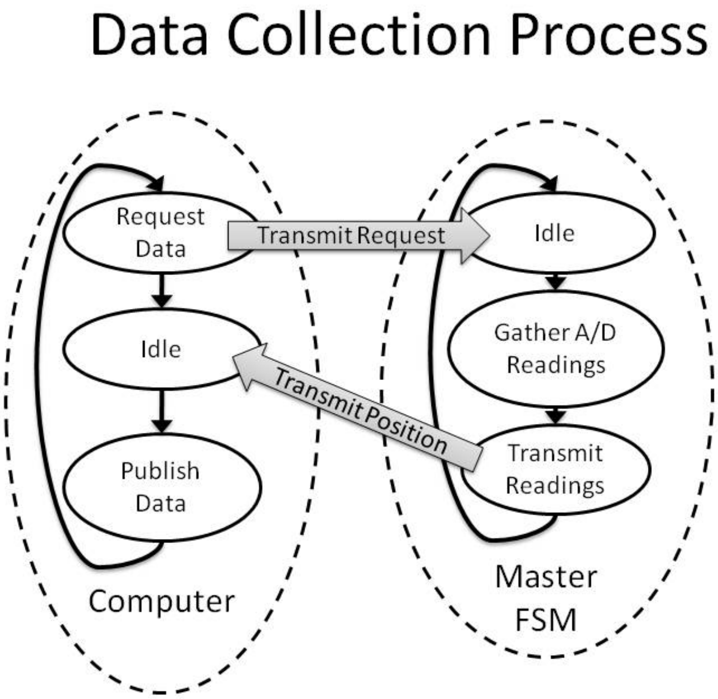


Figure 4-54: Data Collection Process Overview

The A/D FSM remains the same for both position calculation and data collection. Figure 4-55 shows how the master FSM drives the A/D FSM.

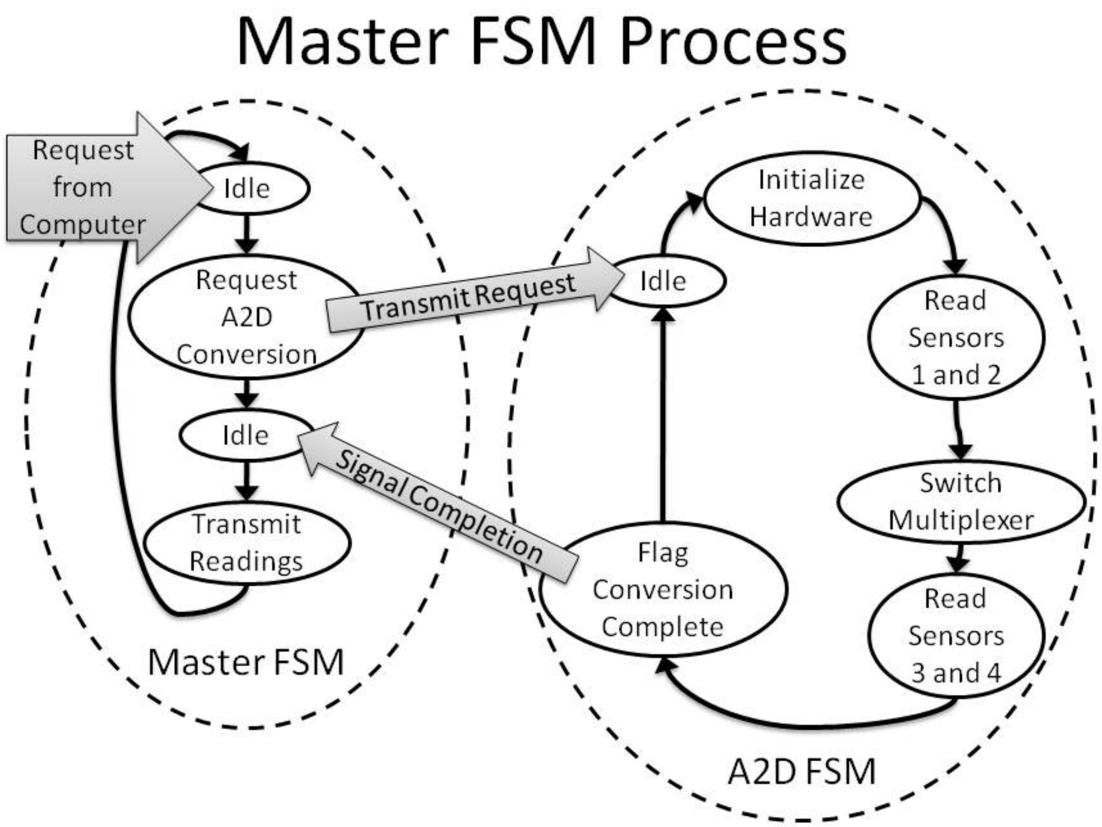


Figure 4-55: Master FSM Process Overview

The master FSM starts in the state *IDLE* and automatically goes to the state *WAIT_RECEIVE*. In the state *WAIT_RECEIVE* , the computer will transmit the letter "t" over serial communication. After the master FSM receives the command from the computer, it sets the variable *train_mode* high and moves on to the state *START_ADC*. In the state *START_ADC*, the master FSM sets variables to start the A/D FSM in motion. After the state *START_ADC*, the master FSM automatically moves to the state *ADC*. The master FSM will loop in this state until the A/D FSM signals completion. At that time, the master FSM will transition to the state *START_RS232_TRAIN1*. Figure 4-56 shows the master FSM state map for this initial process.

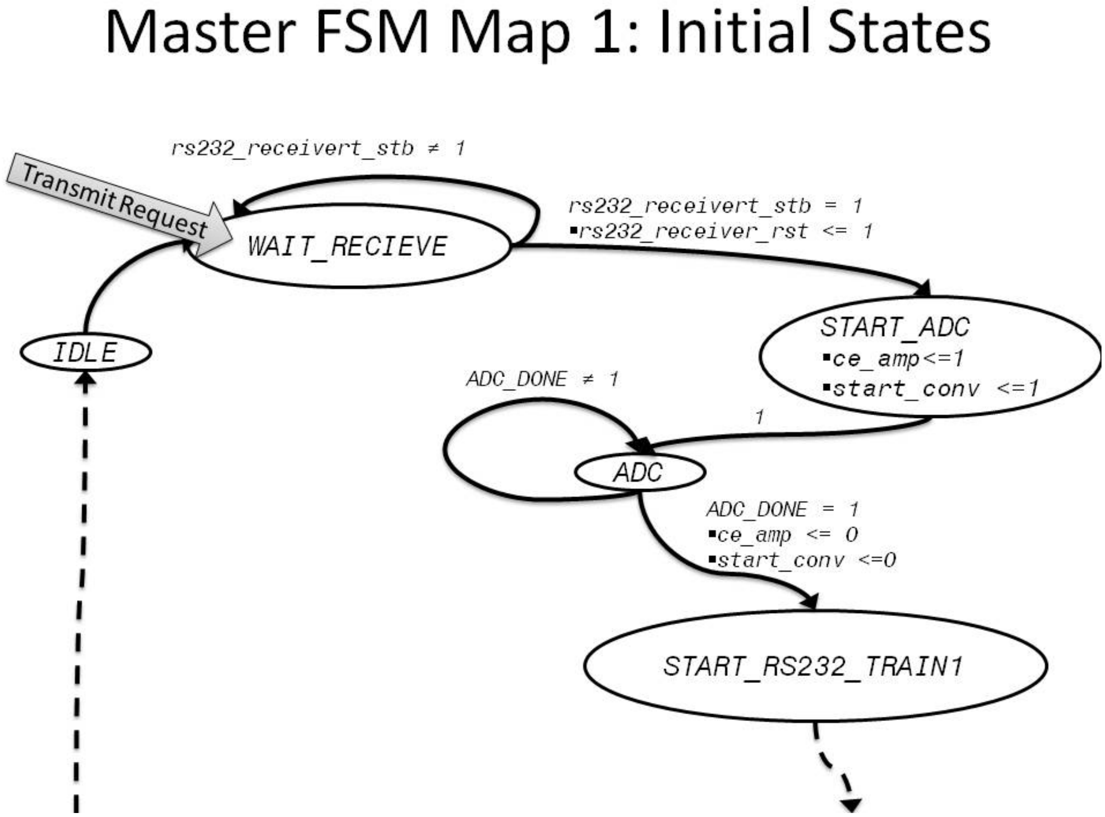


Figure 4-56: Master FSM Map 1: Initial States

When the master FSM enters the state `START_RS232_TRAIN1`, it begins the process to transmit the converted readings to the computer. The process to broadcast data is similar for a single number representing a position calculation or four numbers representing four sensor readings. Figure 4-57 shows the master FSM state map for transmitting the first sensor reading.

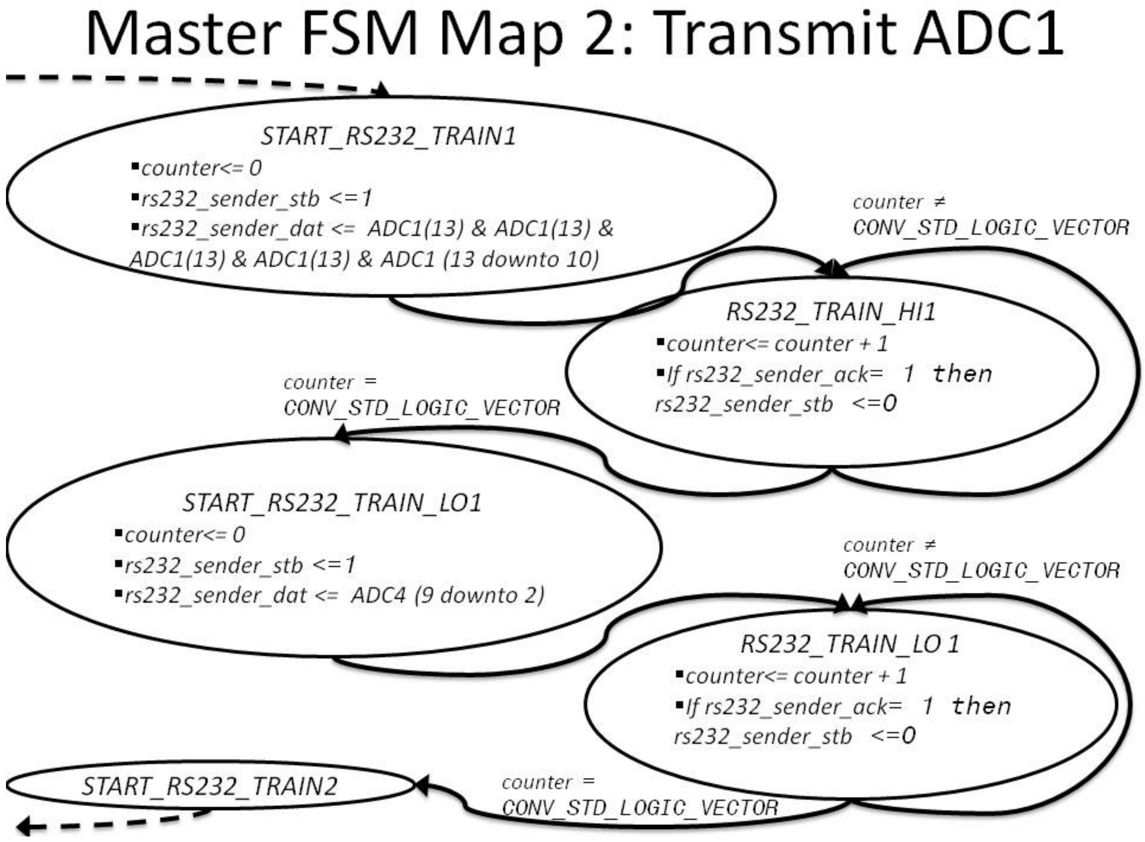

Master FSM Map 2: Transmit ADC1
START_RS232_TRAIN1
▪counter<= 0
▪rs232_sender_stb <=1
▪rs232_sender_dat <= ADC1(13) & ADC1(13) & ADC1(13) & ADC1(13) & ADC1 (13 downto 10)
counter ≠ CONV_STD_LOGIC_VECTOR
RS232_TRAIN_HI1
▪counter<= counter + 1
▪If rs232_sender_ack= 1 then rs232_sender_stb <=0
counter = CONV_STD_LOGIC_VECTOR
START_RS232_TRAIN_LO1
▪counter<= 0
▪rs232_sender_stb <=1
▪rs232_sender_dat <= ADC4 (9 downto 2)
counter ≠ CONV_STD_LOGIC_VECTOR
RS232_TRAIN_LO 1
▪counter<= counter + 1
▪If rs232_sender_ack= 1 then rs232_sender_stb <=0
START_RS232_TRAIN2
counter = CONV_STD_LOGIC_VECTOR

Figure 4-57: Master FSM Map 2: Transmit ADC1

Transmitting the next three sensor readings will require the same process with appropriate changes to the state names and variables to transmit. After transmitting all four sensor readings, the master FSM will automatically transition to the state `IDLE`. Figure 4-58 shows the last sensor reading transmission.

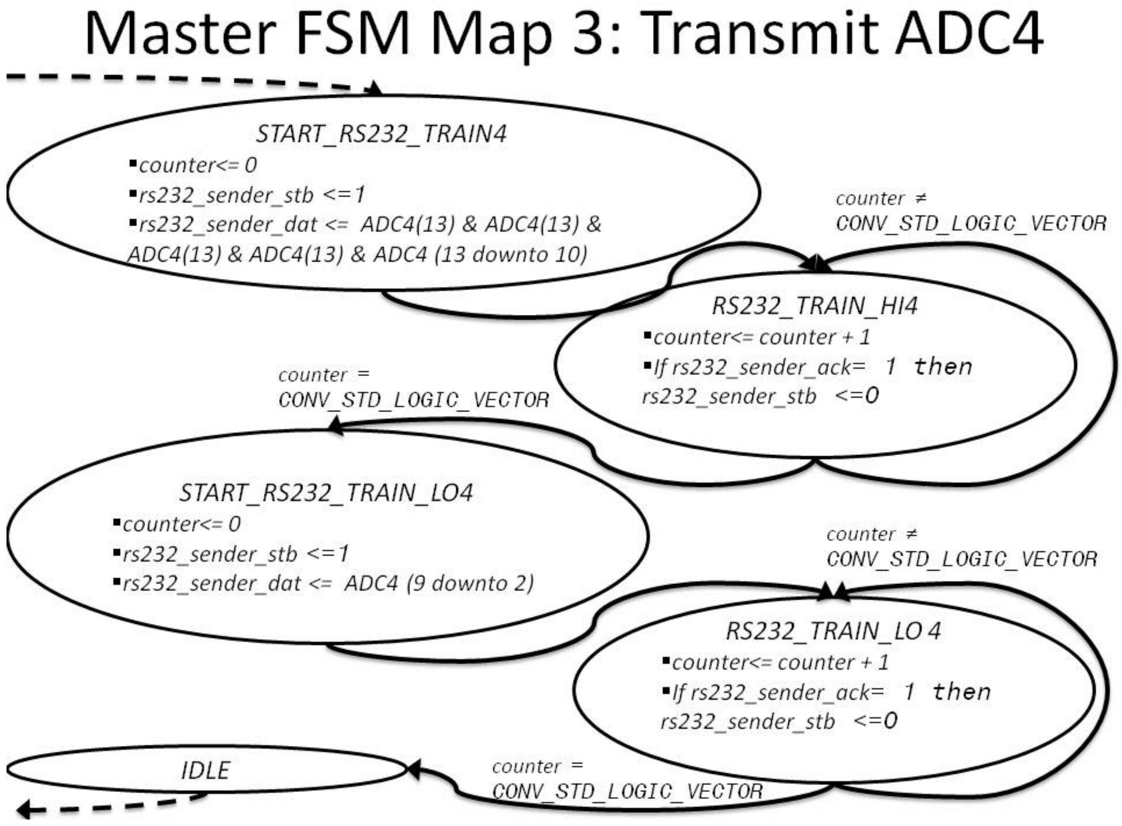
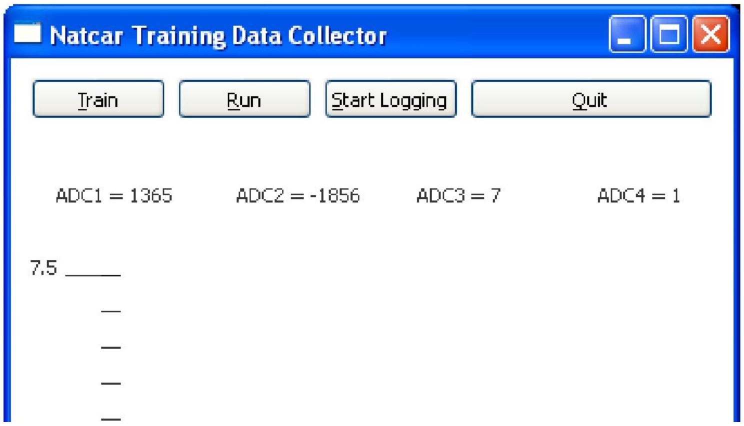

Master FSM Map 3: Transmit ADC4
START_RS232_TRAIN4
▪counter<= 0
▪rs232_sender_stb <=1
▪rs232_sender_dat <= ADC4(13) & ADC4(13) & ADC4(13) & ADC4(13) & ADC4 (13 downto 10)
counter ≠ CONV_STD_LOGIC_VECTOR
RS232_TRAIN_HI4
▪counter<= counter + 1
▪If rs232_sender_ack= 1 then rs232_sender_stb <=0
counter = CONV_STD_LOGIC_VECTOR
START_RS232_TRAIN_LO4
▪counter<= 0
▪rs232_sender_stb <=1
▪rs232_sender_dat <= ADC4 (9 downto 2)
counter ≠ CONV_STD_LOGIC_VECTOR
RS232_TRAIN_LO 4
▪counter<= counter + 1
▪If rs232_sender_ack= 1 then rs232_sender_stb <=0
IDLE
counter = CONV_STD_LOGIC_VECTOR

Figure 4-58: Master FSM Map 3: Transmit ADC4

4.5.2 Automated Data Collection Software

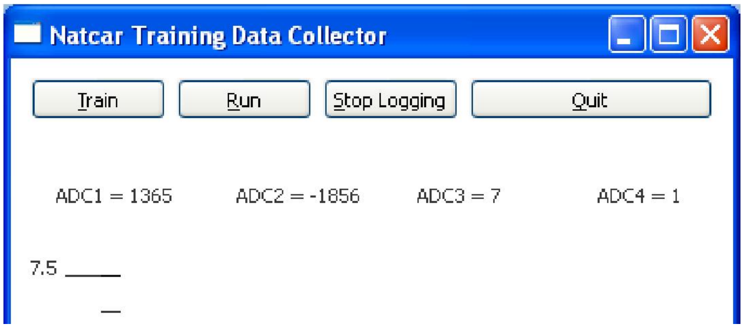
A computer replaces the PD controller for data collection. Figure 4-54 shows how the computer and master FSM interact. If the "Start Logging" button is pressed, the computer program writes the four sensor readings to the screen and records them in a comma-separated file. If "Start Logging" is not pressed or if "Stop Logging" is pressed, the computer program writes the four sensor readings to the screen. Figure 4-59 shows the computer program configured to write data to the screen only.



```
ADC1 = 1365     ADC2 = -1856     ADC3 = 7     ADC4 = 1
```

Figure 4-59: Computer Program Writing to Screen Only

Figure 4-60 shows the computer program writing both screen and comma separated file.



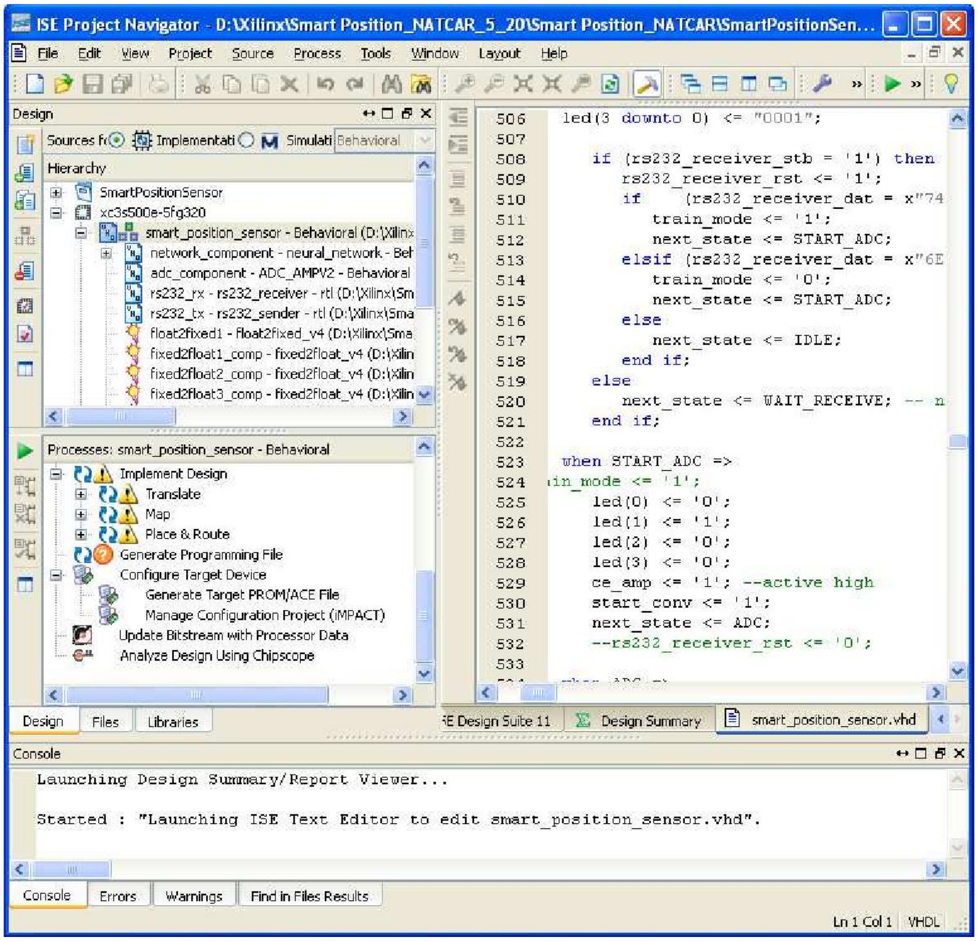
```
ADC1 = 1365     ADC2 = -1856     ADC3 = 7     ADC4 = 1

7.5
```

Figure 4-60: Computer Program Writing to Screen and Data File

Jeff Henson, a former Oklahoma State University Student, wrote the computer program for data collection. Amanuel Assefa, another Oklahoma State University student, contributed to changing the computer program to double the number of sensor readings. Dr. Hagan helped de-bug the code.

4.5.3 Other Support Software

The code for the master FSM and A/D FSM were written using the Xilinx ISE design suite. Xilinx ISE is a file management program. It makes sure that all the different codes for the different components work together. Figure 4-61 shows a screen capture of Xilinx ISE. After the codes are working, Xilinx ISE is used to compile the code and generate programming files. A different Xilinx program is used to configure the FPGA using the programming files.

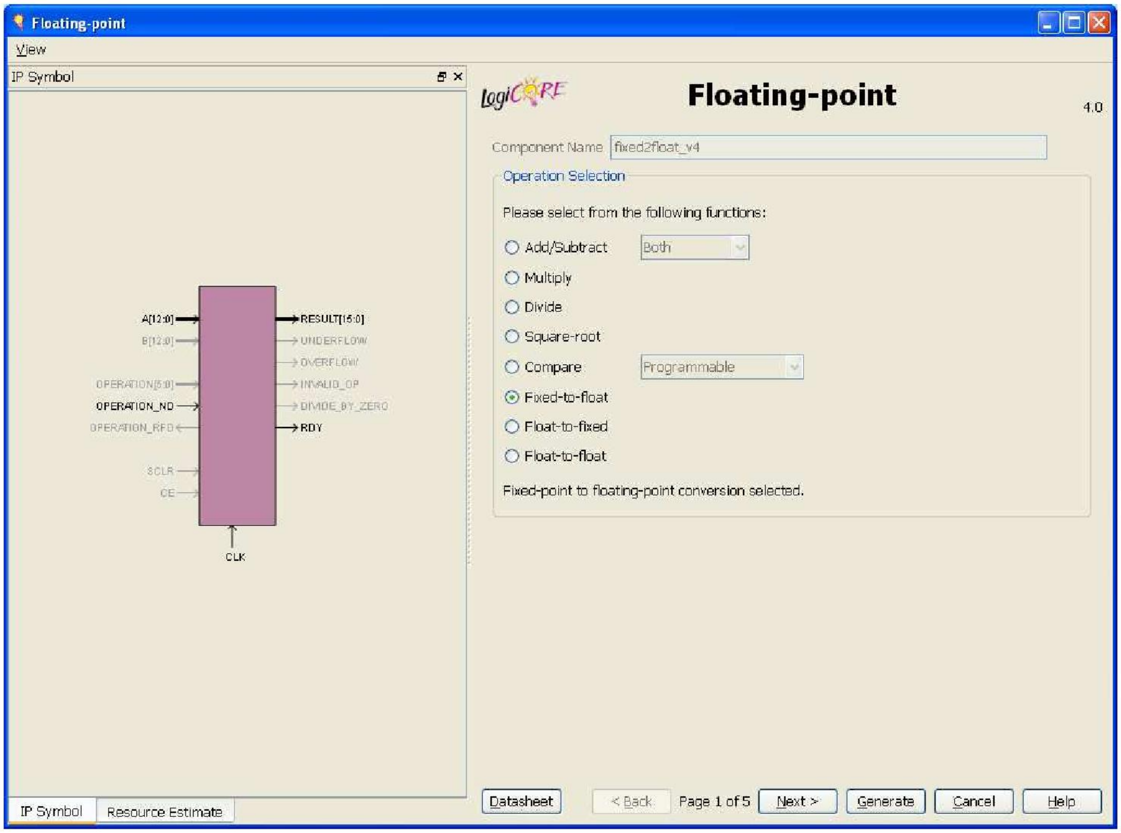

ISE Project Navigator - D:\Xilinx\Smart Position_NATCAR_5_20\Smart Position_NATCAR\SmartPositionSen...
File Edit View Project Source Process Tools Window Layout Help
Design
Sources f(Implementati Simulati Behavioral
Hierarchy
SmartPositionSensor
xc3s500e-5fg320
smart_position_sensor - Behavioral (D:\Xilinx
network_component - neural_network - Beh
adc_component - ADC_AMPV2 - Behavioral
rs232_rx - rs232_receiver - rtl (D:\Xilinx\Sm
rs232_tx - rs232_sender - rtl (D:\Xilinx\Sma
float2fixed1 - float2fixed_v4 (D:\Xilinx\Sma
fixed2float1_comp - fixed2float_v4 (D:\Xilin
fixed2float2_comp - fixed2float_v4 (D:\Xilin
fixed2float3_comp - fixed2float_v4 (D:\Xilin
Processes: smart_position_sensor - Behavioral
Implement Design
Translate
Map
Place & Route
Generate Programming File
Configure Target Device
Generate Target PROM/ACE File
Manage Configuration Project (iMPACT)
Update Bitstream with Processor Data
Analyze Design Using Chipscope
Design
Files
Libraries
E Design Suite 11
Design Summary
smart_position_sensor.vhd
Console
Console
Errors
Warnings
Find in Files Results
Ln 1 Col 1
VHDL

```
 led(3 downto 0) <= "0001";

    if (rs232_receiver_stb = '1') then
       rs232_receiver_rst <= '1';
       if    (rs232_receiver_dat = x"74
          train_mode <= '1';
          next_state <= START_ADC;
       elsif (rs232_receiver_dat = x"6E
          train_mode <= '0';
          next_state <= START_ADC;
       else
          next_state <= IDLE;
       end if;
    else
       next_state <= WAIT_RECEIVE; -- n
    end if;

 when START_ADC =>
in_mode <= '1';
    led(0) <= '0';
    led(1) <= '1';
    led(2) <= '0';
    led(3) <= '0';
    ce_amp <= '1'; --active high
    start_conv <= '1';
    next_state <= ADC;
    --rs232_receiver_rst <= '0';
```

```
Launching Design Summary/Report Viewer...

Started : "Launching ISE Text Editor to edit smart_position_sensor.vhd".
```

Figure 4-61: Xilinx ISE Screen Capture

Another program inside Xilinx ISE is the Xilinx Core Generator. This program generates VHDL code to do common tasks such as floating-point conversion or generating blocks of memory. Figure 4-62 shows a screen capture of the Xilinx CORE Generator software.



Component Name fixed2float_v4

Operation Selection

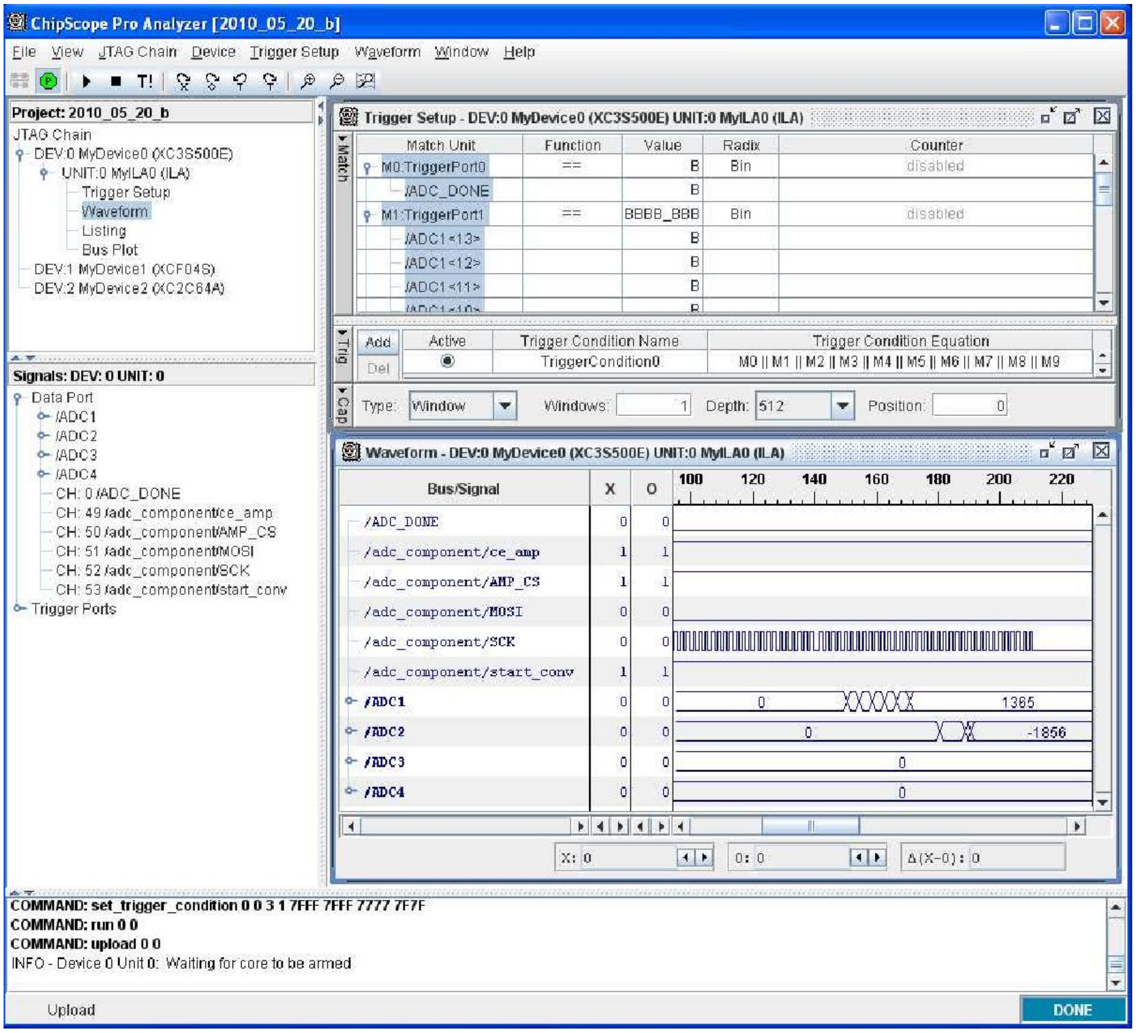
Please select from the following functions:

○ Add/Subtract Both
○ Multiply
○ Divide
○ Square-root
○ Compare Programmable
● Fixed-to-float
○ Float-to-fixed
○ Float-to-float

Fixed-point to floating-point conversion selected.

Figure 4-62: Xilinx CORE Generator Screen Capture

The third major component utilized from the Xilinx ISE design suite is ChipScope. This is software used to analyze designs while they run on the FPGA. This tool allows debugging in the same environment where the code will be deployed. Figure 4-63 is a screen capture of the ChipScope software.


ChipScope Pro Analyzer [2010_05_20_b]
File View JTAG Chain Device Trigger Setup Waveform Window Help
Project: 2010_05_20_b
JTAG Chain
DEV:0 MyDevice0 (XC3S500E)
UNIT:0 MyILA0 (ILA)
Trigger Setup
Waveform
Listing
Bus Plot
DEV:1 MyDevice1 (XCF04S)
DEV:2 MyDevice2 (XC2C64A)
Signals: DEV: 0 UNIT: 0
Data Port
/ADC1
/ADC2
/ADC3
/ADC4
CH: 0 /ADC_DONE
CH: 49 /adc_component/ce_amp
CH: 50 /adc_component/AMP_CS
CH: 51 /adc_component/MOSI
CH: 52 /adc_component/SCK
CH: 53 /adc_component/start_conv
Trigger Ports
Trigger Setup - DEV:0 MyDevice0 (XC3S500E) UNIT:0 MyILA0 (ILA)
Match
Trig
Cap
Add
Del
Waveform - DEV:0 MyDevice0 (XC3S500E) UNIT:0 MyILA0 (ILA)
100
120
140
160
180
200
220
0
1365
0
-1856
0
0
Upload
DONE

Match Unit	Function	Value	Radix	Counter
M0:TriggerPort0	==	B	Bin	disabled
/ADC_DONE		B		
M1:TriggerPort1	==	BBBB_BBB	Bin	disabled
/ADC1<13>		B		
/ADC1<12>		B		
/ADC1<11>		B		
/ADC1<10>		B		

Active	Trigger Condition Name	Trigger Condition Equation
●	TriggerCondition0	M0 || M1 || M2 || M3 || M4 || M5 || M6 || M7 || M8 || M9

Type: Window Windows: 1 Depth: 512 Position: 0

Bus/Signal	X	O
/ADC_DONE	0	0
/adc_component/ce_amp	1	1
/adc_component/AMP_CS	1	1
/adc_component/MOSI	0	0
/adc_component/SCK	0	0
/adc_component/start_conv	1	1
/ADC1	0	0
/ADC2	0	0
/ADC3	0	0
/ADC4	0	0

X: 0 O: 0 Δ(X-O): 0

```
COMMAND: set_trigger_condition 0 0 3 1 7FFF 7FFF 7777 7F7F
COMMAND: run 0 0
COMMAND: upload 0 0
INFO - Device 0 Unit 0:  Waiting for core to be armed
```

Figure 4-63: Xilinx ChipScope Screen Capture

4.6 Summary

The neural network maps four sensor readings into a car position calculation. The A/D converter produces four digital sensor readings. The PD controller accepts the car position. The neural network changes the output of the A/D converter into the acceptable input for the PD controller. The neural network calculates the car position serially, doing one arithmetic step at a time. Training helps ensure the neural network calculates the car position correctly. The master FSM has some modifications that allow it to interface with a computer to collect training data. In order to ensure the neural network, master FSM, A/D FSM and all other components work together properly, other support equipment is required. This support equipment is part of the Xilinx ISE design suite.

CHAPTER V

PD CONTROLLER

This chapter discuses the PD controller. The PD controller takes information from the neural network and generates commands for the car steering servo and motor to produce motion around the track. The neural network takes inputs from the analog to digital converters and calculates the car position. That position is sent to the PD controller. The PD controller uses the car position in a calculation to determine the command for the steering servo. The PD controller also controls the car motor. The car responds by producing motion around the track. Figure 5-1 is the main block diagram that shows how the components work together. This chapter describes the real-time executive program that implements the PD controller, as well as the associated hardware and support equipment. The chapter also describes the controller design.

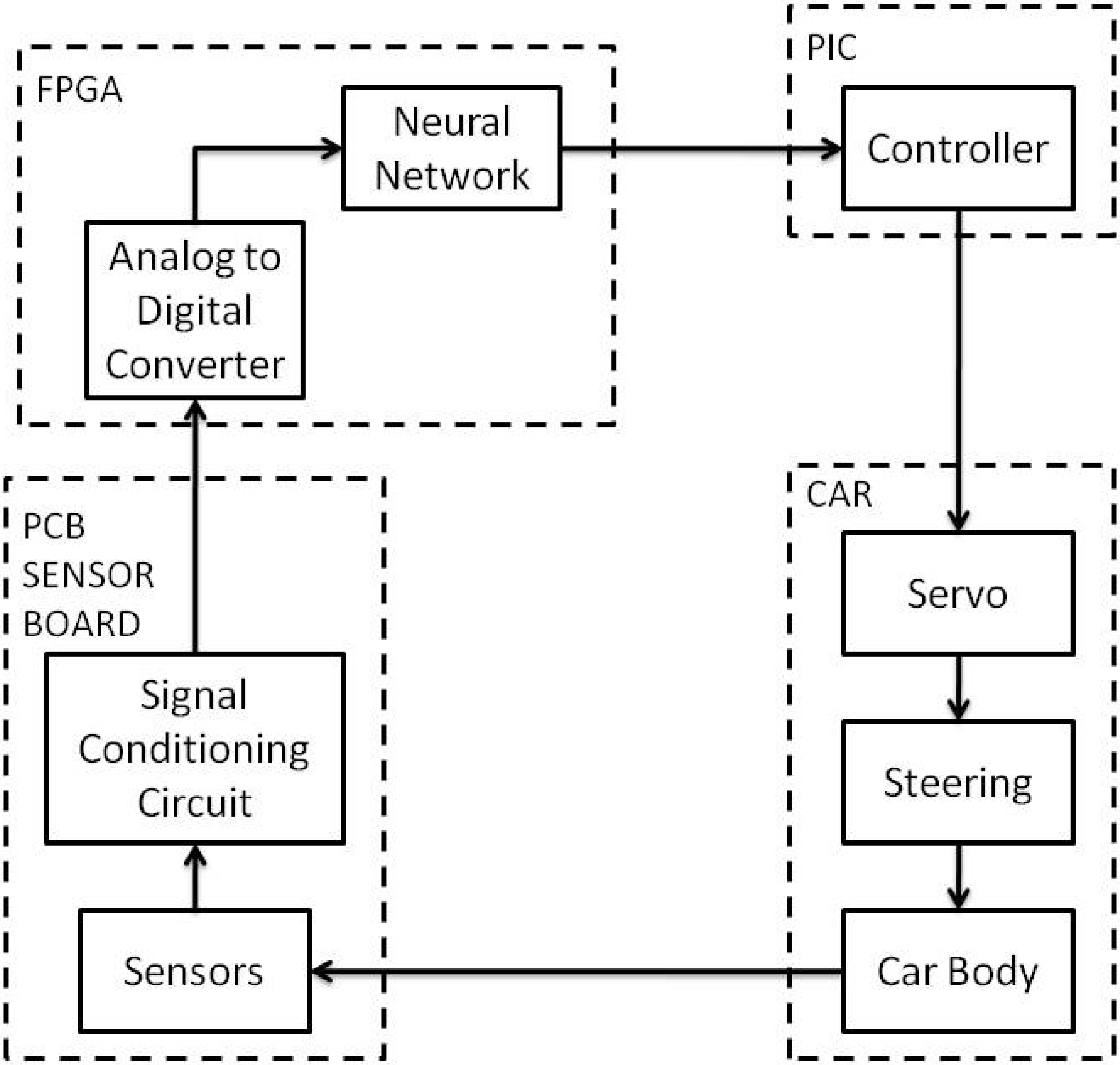


Figure 5-1: Main Block Diagram

5.1 Overview

The PD controller takes the distance of the car from the wire and uses it to determine the steering angle. As the Car moves along the track, sensors detect the magnetic field coming from the track. The sensor signals go through a signal conditioning circuit that prepares analog sensor readings for the Analog To Digital (A/D) converters on the FPGA board. The A/D converters change the analog signals into digital numbers for the neural network to use as inputs for the position calculation. The calculated position tells the PD controller the distance of the car from the wire. The PD controller then generates steering angle and speed commands for the car so it can continue moving along the track. This chapter focuses on the PD controller. Figure 5-2 shows how the PD controller fits into the system functional overview.

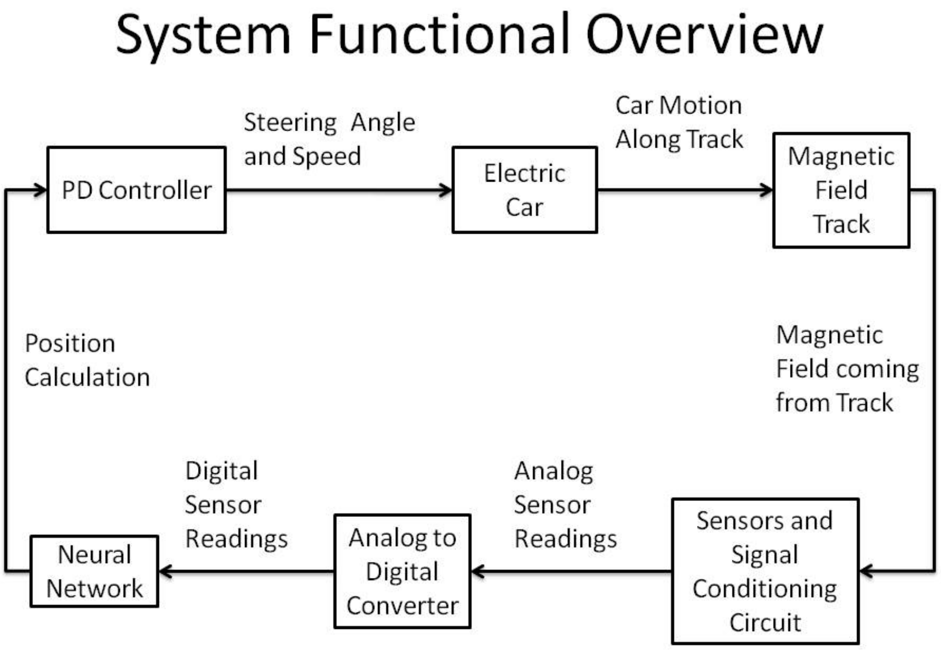


Figure 5-2: System Functional Overview

The PD controller algorithm controls the data flow and communication that make the timing of the process work. The PD controller starts the process by sending a request for a new distance. As discussed in chapter 3, the master FSM will then cycle the A/D converter to generate four digital sensor readings. Chapter 4 explains how the master FSM then cycles the neural network to calculate the distance of the car from the wire. After calculating the distance, the master FSM transmits the new distance to the PD controller. Figure 5-4 shows an overview of the interaction between the PD controller and master FSM.

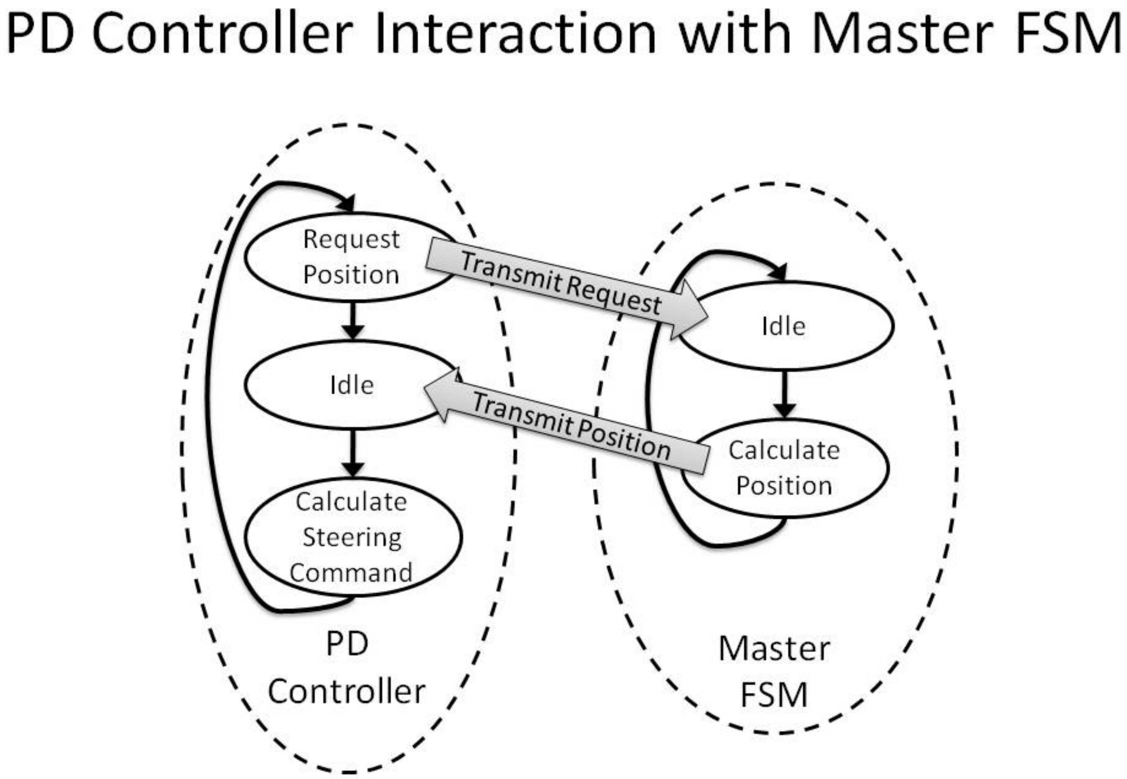


Figure 5-3: PD Controller Interaction with Master FSM

The PD controller repeats a process in order to keep the car moving around the track. The process begins with a request from the PD controller to the neural network for a new distance measurement. The PD controller waits until the neural network responds with the measurement. The PD controller then calculates how fast the car is moving toward or away from the wire. The PD controller uses the lateral velocity and distance measurement to calculate the necessary steering angle that will steer the car directly over the wire while maintaining stability. The PD controller then converts the steering angle into a pulse that the steering servo can accept as an input. The PD controller also generates a pulse for the motor control switch to set the car speed. The cycle will repeat as the car continues to circle around the track. Figure 5-4 shows the PD controller process overview.

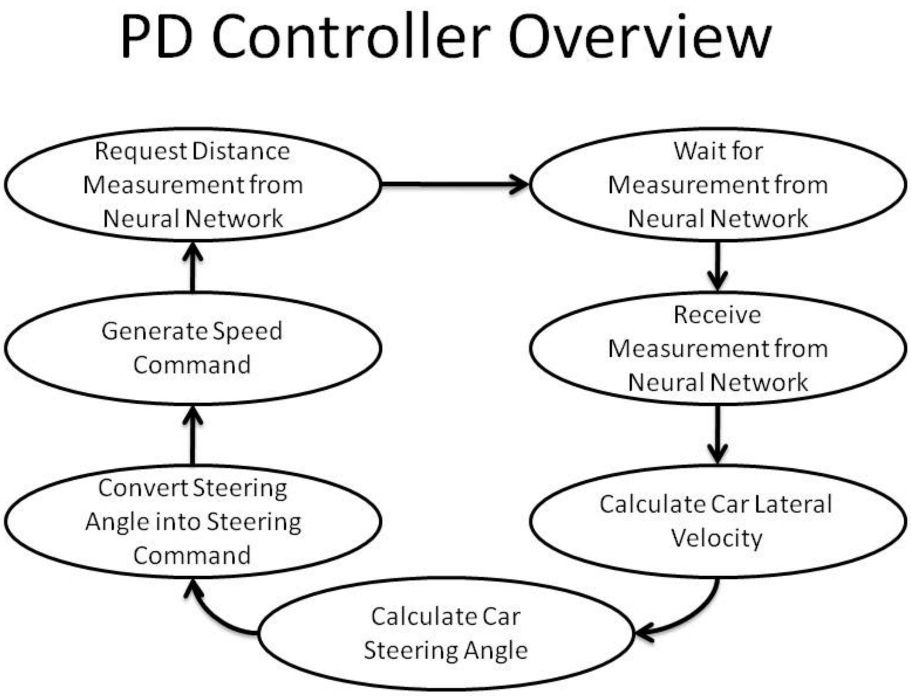

PD Controller Overview
Request Distance Measurement from Neural Network
Wait for Measurement from Neural Network
Receive Measurement from Neural Network
Calculate Car Lateral Velocity
Calculate Car Steering Angle
Convert Steering Angle into Steering Command
Generate Speed Command

Figure 5-4: PD Controller Overview

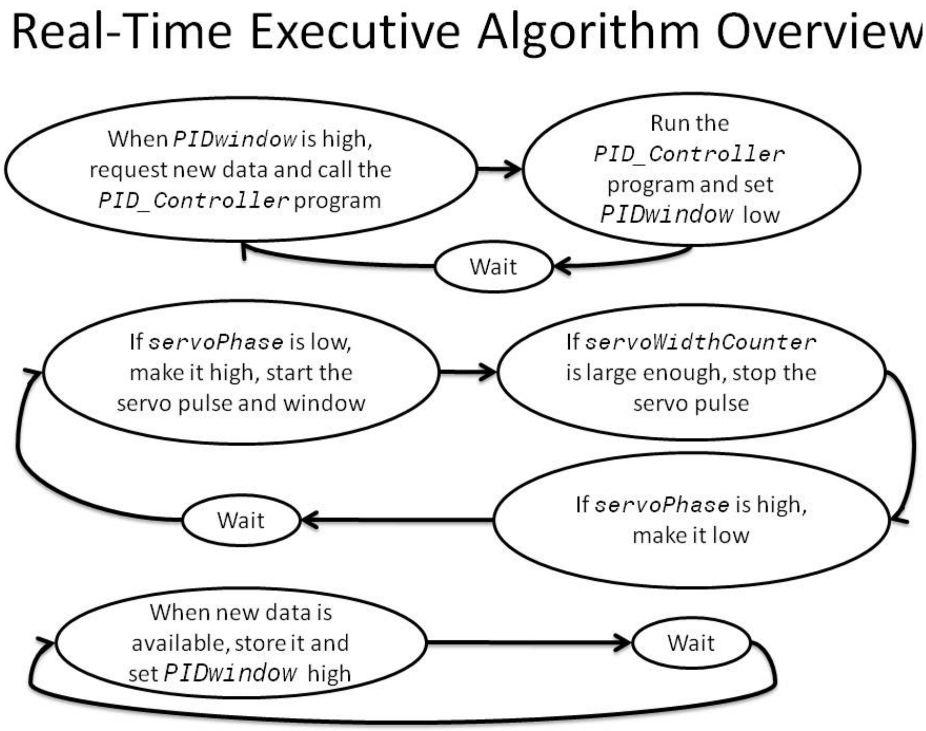
5.2 Implementation

The real-time executive algorithm controls the data flow and communications that make the timing of the processes work. The algorithm will first communicate with the master FSM to obtain a distance measurement from the neural network. The algorithm will then perform the calculation of the steering and speed commands.

The real-time executive algorithm is software. The software is supported by hardware. That hardware comes in the form of a printed circuit board with a Microchip PIC microcontroller. The PIC board provides power to the PIC and an interface to the rest of the boards and processes. The real-time executive algorithm is described in section 5.2.1. The PIC board circuit and hardware is described in section 5.2.2.

5.2.1 Real-Time Executive Algorithm

The code for the real time executive algorithm can be found in appendix E: PD Controller Real Time Executive Algorithm Code. Figure 5-4 described the basic tasks of the real-time executive algorithm. The algorithm has three separate loops for three separate tasks. The first task is calculating the steering angle. This is done by the program *`PID_Controller`* in the top loop. The second task is controlling the servo. The middle loop takes care of timing for the twenty millisecond window with the variable *`servoPhase`* and also handles the one to two millisecond pulse to control the servo with the variable *`servoWidthCounter`*. The bottom loop has the task of importing new data from the neural network. Figure 5-5 shows the real-time executive algorithm overview.



Real-Time Executive Algorithm Overview
When PIDwindow is high, request new data and call the PID_Controller program
Run the PID_Controller program and set PIDwindow low
Wait
If servoPhase is low, make it high, start the servo pulse and window
If servoWidthCounter is large enough, stop the servo pulse
If servoPhase is high, make it low
Wait
When new data is available, store it and set PIDwindow high
Wait

Figure 5-5: Real-Time Executive Algorithm Overview

The algorithm begins in the program `main`. This program includes the three interrupts, `RDA`, `Timer1` and `RTCC`. The program `main` calls the program `PID_Controller`. The variable `set_pwm1_duty` controls the motor speed, with 255 being the maximum allowed speed. The variable `PIDwindow` controls the algorithm timing related to reading in a new distance from the master FSM as well as calculating a new steering command. When a new position is read in, the variable `PIDwindow` is set high. This allows the program `main` to request a new position from the master FSM by broadcasting the letter "`n`." When the variable `PIDwindow` is high, the program `main` will also call the program `PID_Controller`. Figure 5-6 shows the first diagram for the real-time executive algorithm flow chart.

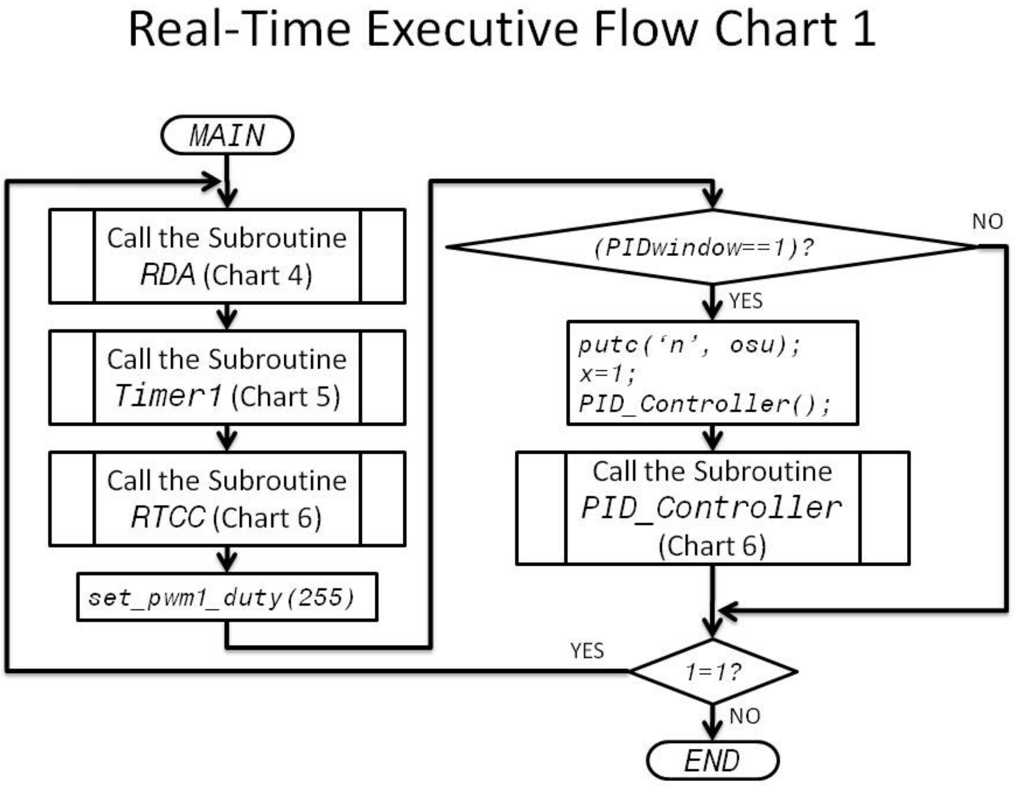

Real-Time Executive Flow Chart 1
MAIN
Call the Subroutine RDA (Chart 4)
Call the Subroutine Timer1 (Chart 5)
Call the Subroutine RTCC (Chart 6)
set_pwm1_duty(255)
(PIDwindow==1)?
NO
YES
putc('n', osu);
x=1;
PID_Controller();
Call the Subroutine PID_Controller (Chart 6)
YES
1=1?
NO
END

Figure 5-6: Real-Time Executive Algorithm Flow Chart 1

The PD controller subroutine performs the following tasks:

- Load the last raw position received from the neural network into the variable `fullpos`. The raw position is a number between 0 and 800, and has units of hundredths of an inch.

- Adjust for the center position and convert all measurements from hundredths of inches to meters. After this operation, the variable `position` represents the distance of the car to the right of the wire in meters.

- Save the old position into `prevPos`

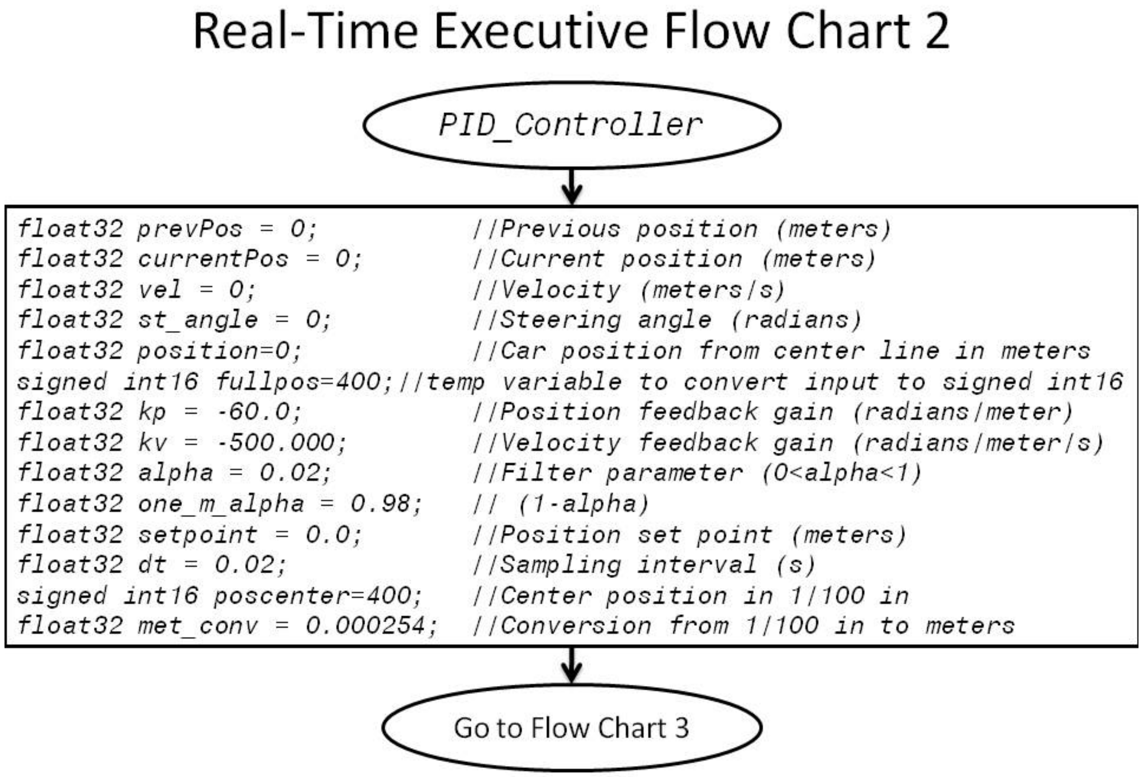
- Filter the position to remove some noise using the equation: `currentPos = alpha * position + (1 – alpha) * prevPos`. The variable *alpha* is the filter parameter that determines how many points are averaged together to find the position.

- Compute the velocity with the equation: `vel = (currentPos – prevPos)/dt`. The variable `dt` is the time between samples in seconds.

- Compute the steering angle in radians using a proportional-derivative control with the equation: `st_angle = (currentPos – setpoint) * kp + kv * vel`. The variable `kp` is the proportional gain and the variable `kv` is the derivative gain. Modeling to find these gain values is discussed in section 5.3.

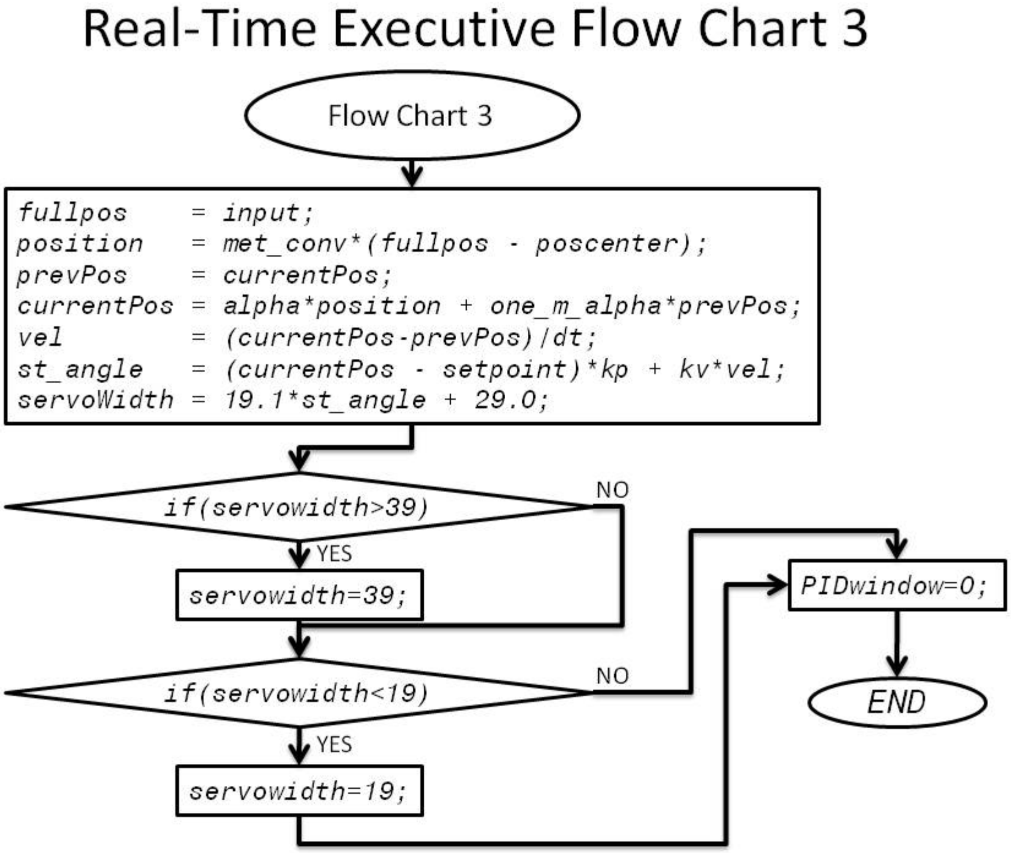
- Convert the steering angle into a pulse width count and save it as the variable `servoWidth` that can be sent to the servo. The count is a number that ranges from 19 to 39. The value 19 corresponds approximately to a 1 ms pulse and a steering angle of $-{}^{\pi}/_{6}$ radians. The value of 39 corresponds to approximately a 2 ms pulse and a steering angle of ${}^{\pi}/_{6}$ radians. The controller will adjust the calculated count to stay in the range of 19 to 39 if the calculation results in a count outside that range.

The flowchart for the PD controller routine is shown in Figures 5-7 and 5-8. At the end of the routine the variable `PIDwindow` is set to zero, which ensures that the PD controller does not run again until another position is read.



```
float32 prevPos = 0;          //Previous position (meters)
float32 currentPos = 0;       //Current position (meters)
float32 vel = 0;              //Velocity (meters/s)
float32 st_angle = 0;         //Steering angle (radians)
float32 position=0;           //Car position from center line in meters
signed int16 fullpos=400;//temp variable to convert input to signed int16
float32 kp = -60.0;           //Position feedback gain (radians/meter)
float32 kv = -500.000;        //Velocity feedback gain (radians/meter/s)
float32 alpha = 0.02;         //Filter parameter (0<alpha<1)
float32 one_m_alpha = 0.98;   // (1-alpha)
float32 setpoint = 0.0;       //Position set point (meters)
float32 dt = 0.02;            //Sampling interval (s)
signed int16 poscenter=400;   //Center position in 1/100 in
float32 met_conv = 0.000254;  //Conversion from 1/100 in to meters
```

Figure 5-7: Real-Time Executive Flow Chart 2



```
fullpos     = input;
position    = met_conv*(fullpos - poscenter);
prevPos     = currentPos;
currentPos  = alpha*position + one_m_alpha*prevPos;
vel         = (currentPos-prevPos)/dt;
st_angle    = (currentPos - setpoint)*kp + kv*vel;
servoWidth  = 19.1*st_angle + 29.0;
```

Figure 5-8: Real-Time Executive Flow Chart 3

The servo pulse width calculations are specific to the servo. For the servo used for the summer academy, the pulse was one to two milliseconds long in a twenty millisecond window. Setting the variable `servowidth` to 39 corresponded to a pulse width of two milliseconds and a steering angle of $\pi/6$ radians. When the variable `servowidth` was set to 29, the pulse was 1.5 milliseconds wide and the front tires were straight forward. When the variable `servowidth` was set to 19, the pulse was one millisecond wide and the steering angle was $-\pi/6$ radians.

There are several interrupt routines in the PIC software. The first interrupt is `RDA`, and it starts when the port connected to the RS232 communication lines has received a transmission from the FPGA and has new data available. The command `getc()` stores the eight bits read in through the port. In the interrupt `RDA`, if the variable `x` is one, the bits are stored to the variable `input1`, and the variable `x` is incremented. If the variable `x` is two, the bits are stored to the variable `input2`. When the interrupt is complete, it sets the variable `PIDWindow` high, sets the variable `x` to zero and combines the two eight bit numbers into a single 16-bit distance. Setting the variable `PIDWindow` high allows the program main to call the PD controller subroutine to calculate another steering command and ask for another distance from the neural network. Figure 5-9 shows the interrupt routine `RDA` in the fourth flow chart of the real-time executive algorithm.

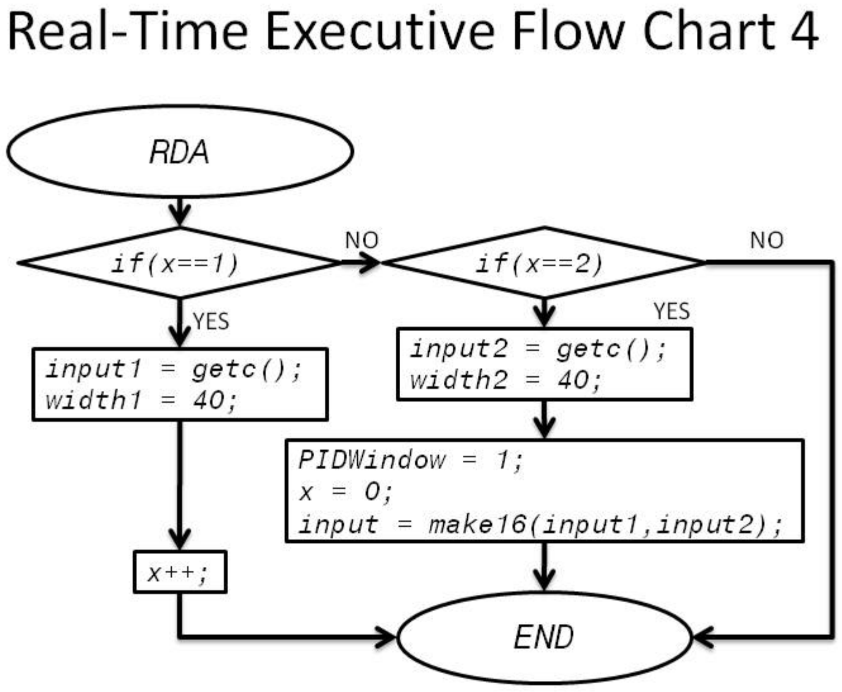


Figure 5-9: Real-Time Executive Flow Chart 4

The second interrupt routine is `Timer1`, which is executed every 13.1 milliseconds. This interrupt controls the timing for the 20 millisecond window for the steering servo command. If the variable `servoPhase` is low, the variable `servoWidthCounter` is set to zero, the variable `servoFlag` is set high, and the variable `servoPhase` is set high. In the event that the variable `servoPhase` is high, then the variable `servoPhase` is set low. The variables `servoFlag` and `servoWidthCounter` are used in the interrupt `RTCC`. The variable `servoWidthCounter` controls the pulse width for the steering servo command. The variable `servoFlag` ensures that the one to two millisecond pulse for the servo command happens only at the start of the twenty millisecond window. Figure 5-10 shows the fifth flow chart for the real-time executive algorithm that shows the interrupt `Timer1`.

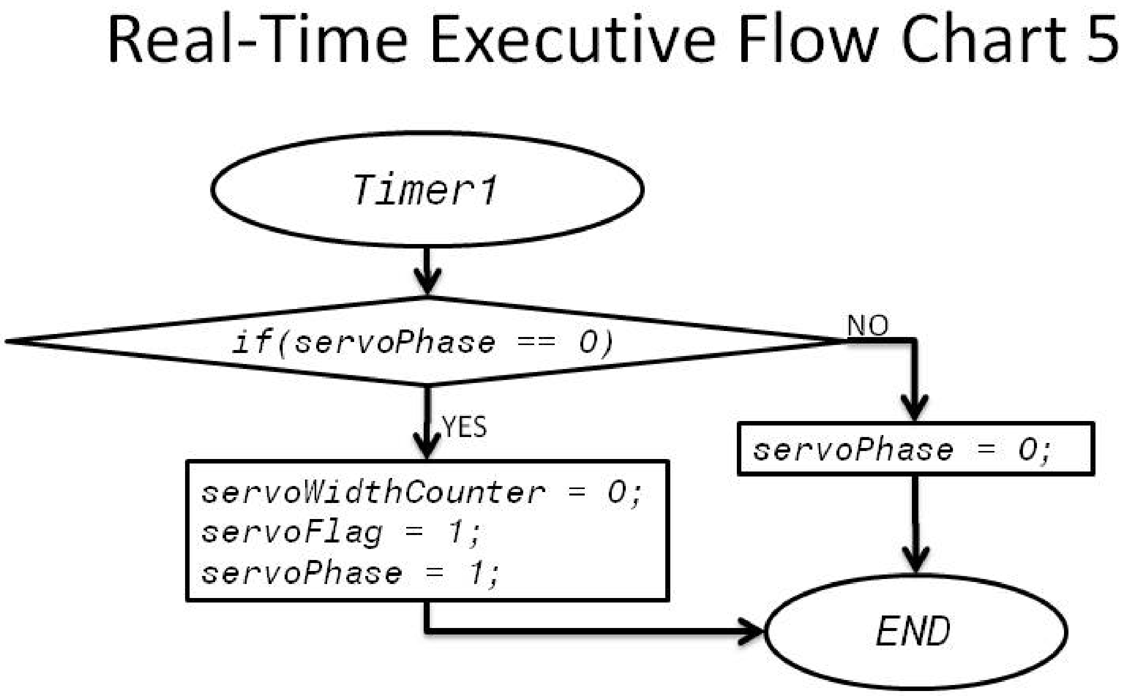


Figure 5-10: Real-Time Executive Flow Chart 5

The interrupt *Timer1* controls the timing of the interrupt routine *RTCC*, which is executed every 51.2 microseconds. The interrupt *RTCC* controls the variables that drive the car steering servo command. In the interrupt *RTCC*, if the variable *servoFlag* is high, the interrupt will compare the variable *servoWidthCounter* against the variable *servoWidth*. If the variable *servoWidthCounter* is less than the variable *servoWidth*, the pin connected to the steering servo control is set high and the variable *servoWidthCounter* is incremented. If the variable *servoWidthCounter* is less than the variable *servoWidth*, the pin connected to the steering servo control is set low and the variable *servoFlag* is set low. The variable *servoFlag* will be reset to high in the interrupt routine *RTCC* after the current twenty millisecond window has ended. Figure 5-11 shows the sixth flow chart for the real-time executive algorithm that describes the interrupt routine *RTCC*.

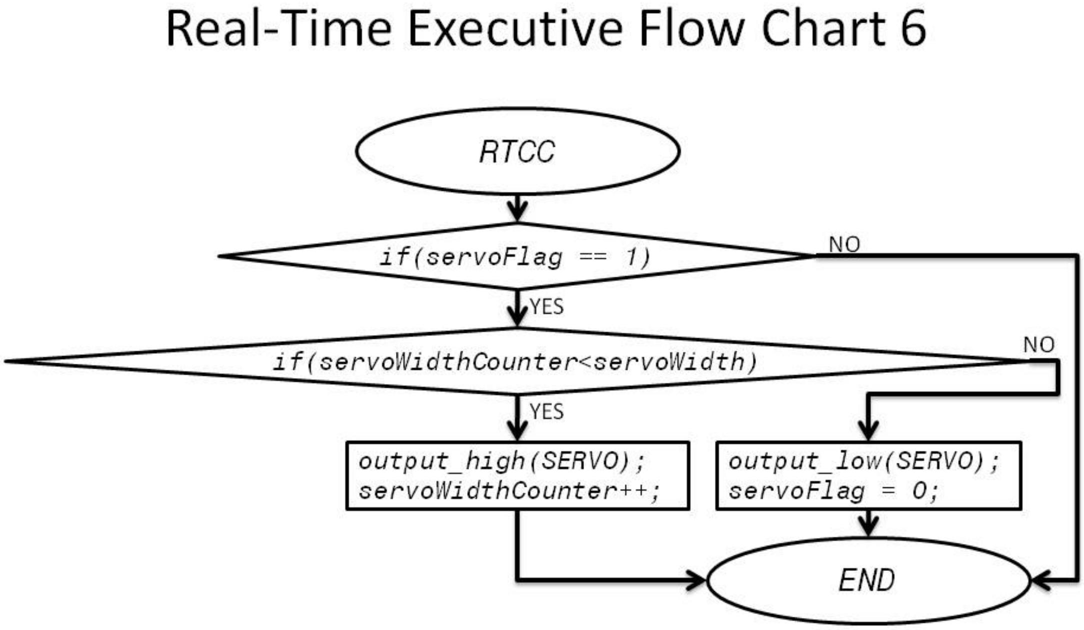


Figure 5-11: Real-Time Executive Flow Chart 6

5.2.2 Hardware

The real-time executive algorithm is implemented on a Microchip PIC 185F1220 microcontroller. The chip needs several connections and other components to support the real-time executive algorithm. A printed circuit board supports the chip and connects it to the other components. The PIC board receives power from a battery pack. The battery provides between nine and eleven volts. The PIC board connects to the battery through connector J3, shown in Figures 5-12 through 5-15, which is a two pin header. The power then goes through traces with decoupling capacitors into a pair of voltage regulators. Both regulators are Texas Instrument TLV1117050CDCYR linear voltage regulators that output five volts with a current load up to 0.8 amps. Component U6 provides power to the PIC and other on board components. Component U7 provides power to the center pin of header J1. J1 is a three pin header used to connect to the servo. The first pin is tied to ground, the second pin has power at five volts, and the third pin is controlled by the real-time executive algorithm with the variable *SERVO*. To operate the algorithm quickly, the PIC requires an external clock signal. That clock signal comes from component U4. The real-time executive algorithm controls the motor through the connection header J2, a four pin header. The first pin is connected to ground; the third pin is tied to the pulse width modulation port of the PIC chip. The second and fourth pins of header J2 are not connected in the current configuration. Loading the real-time executive algorithm onto the chip happens through the connection header U21. This is the standard connection header, built to conform to the requirements of the Microchip PICKit 3 device. RS232 communication goes through header J4, a DSUB9 receptacle. The PIC operates at 5 volts, but the RS232 communication can be between three and fifteen volts. The RS232 communication is passed through a Texas Instruments MAX232E dual RS-232 driver/receiver, which is component U5. Figure 5-12 shows the PIC board schematic which displays the connections between all the components.

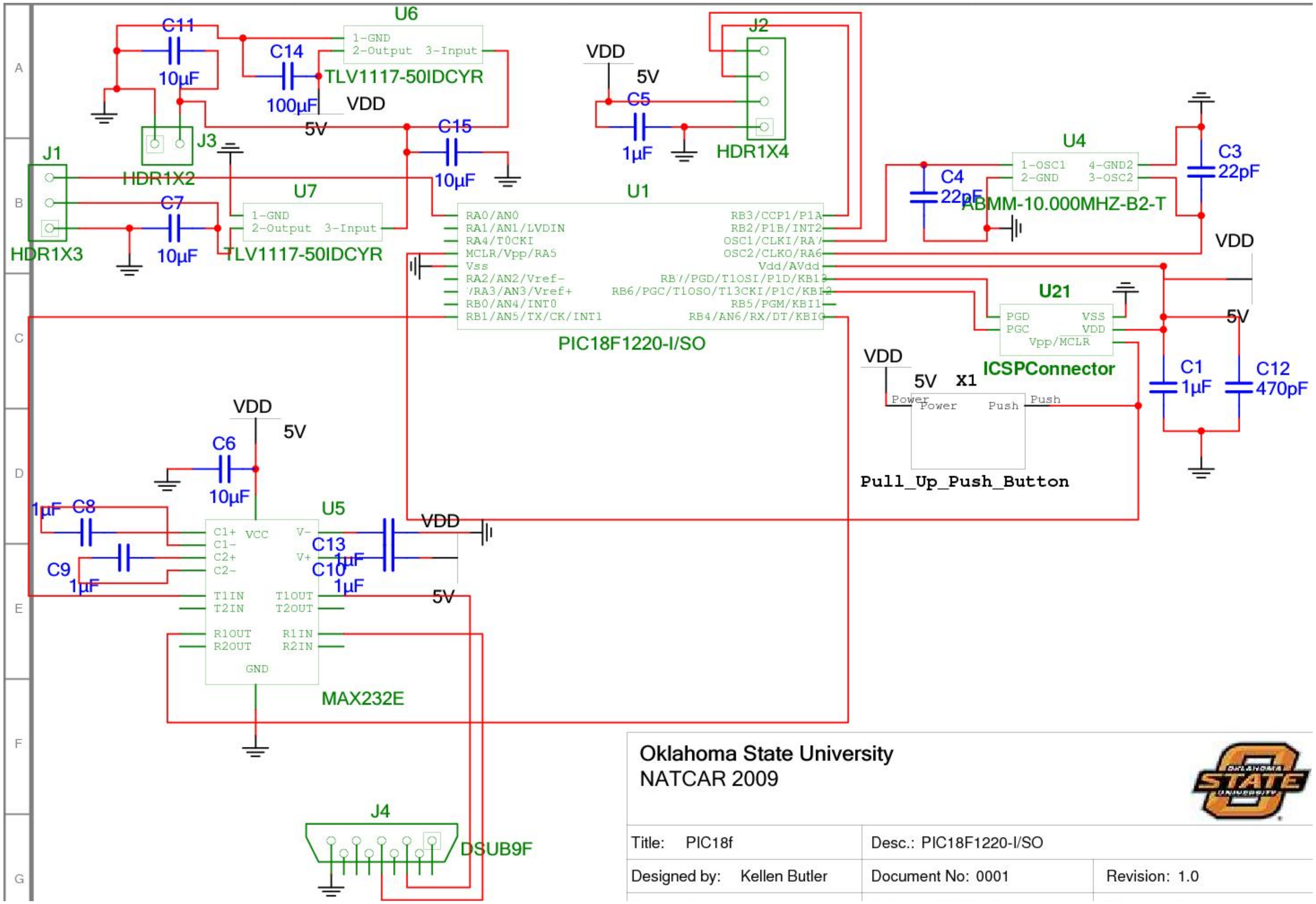


Figure 5-12: PIC Board Schematic

Figure 5-13 shows the printed circuit board layout for the PIC board.

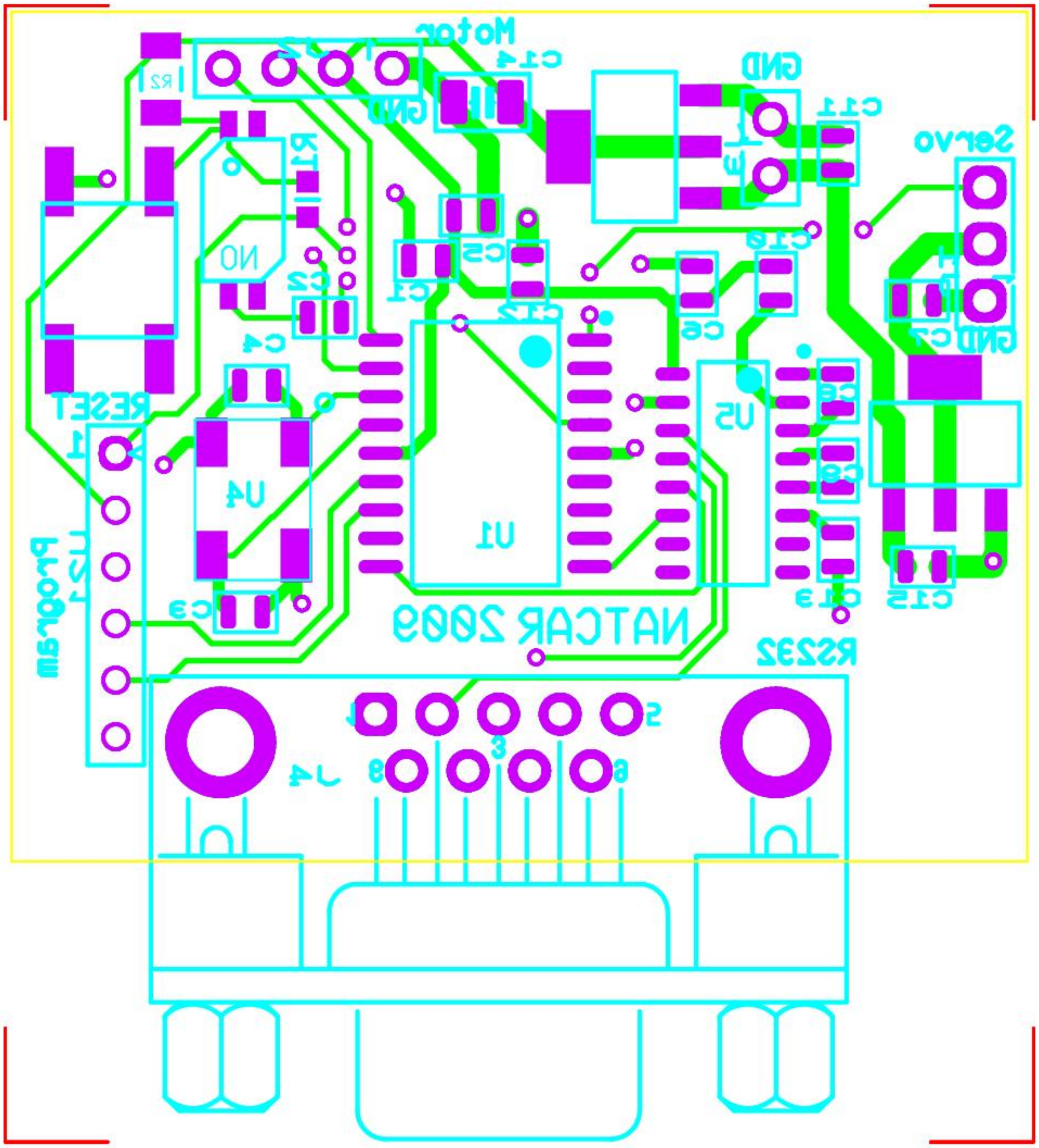


Figure 5-13: PIC Board PCB Layout

Figure 5-14 shows a photo of a populated printed circuit board.

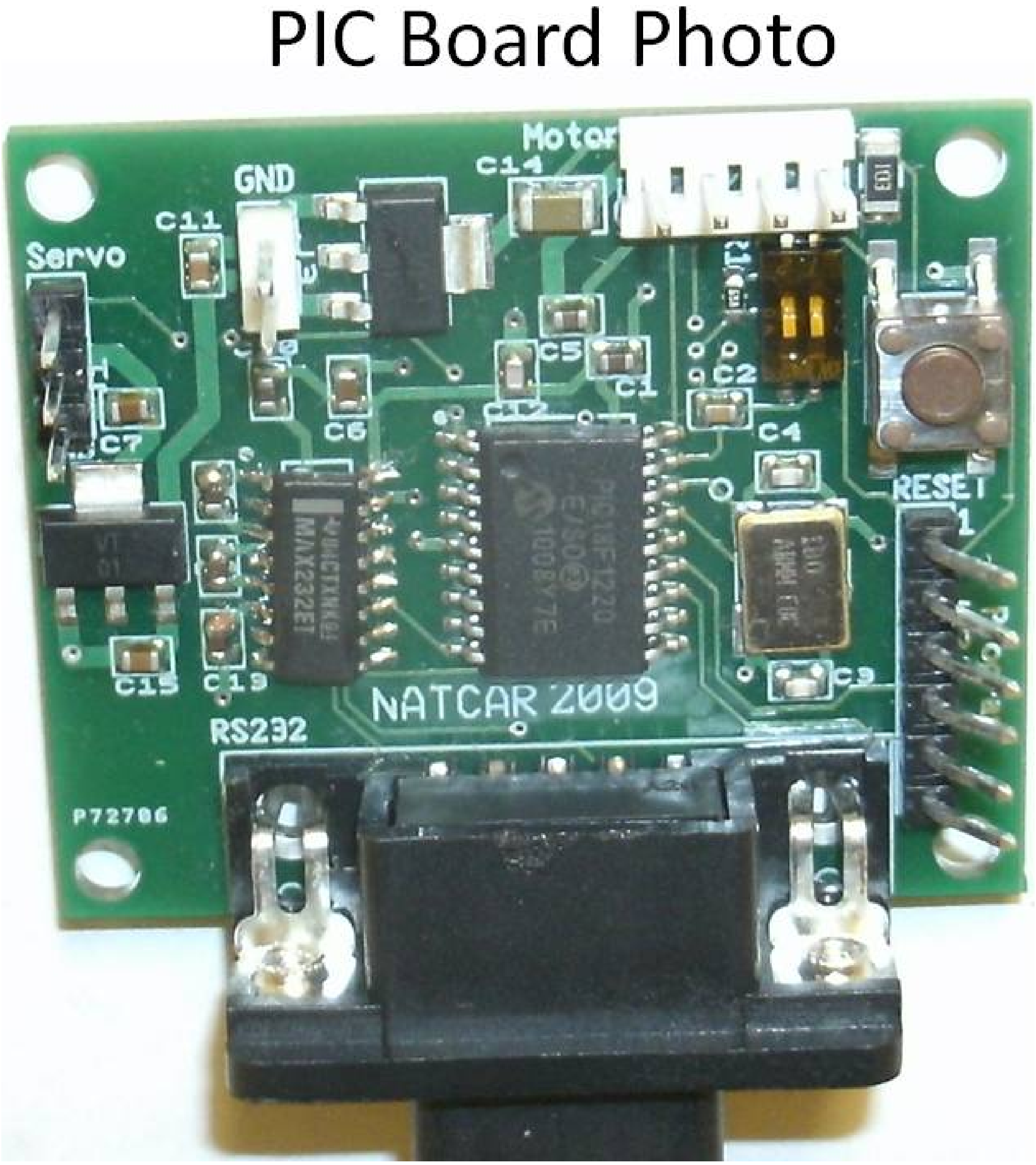


Figure 5-14: Photo of PIC Board

Figure 5-15 shows a picture of the PIC board with labels.

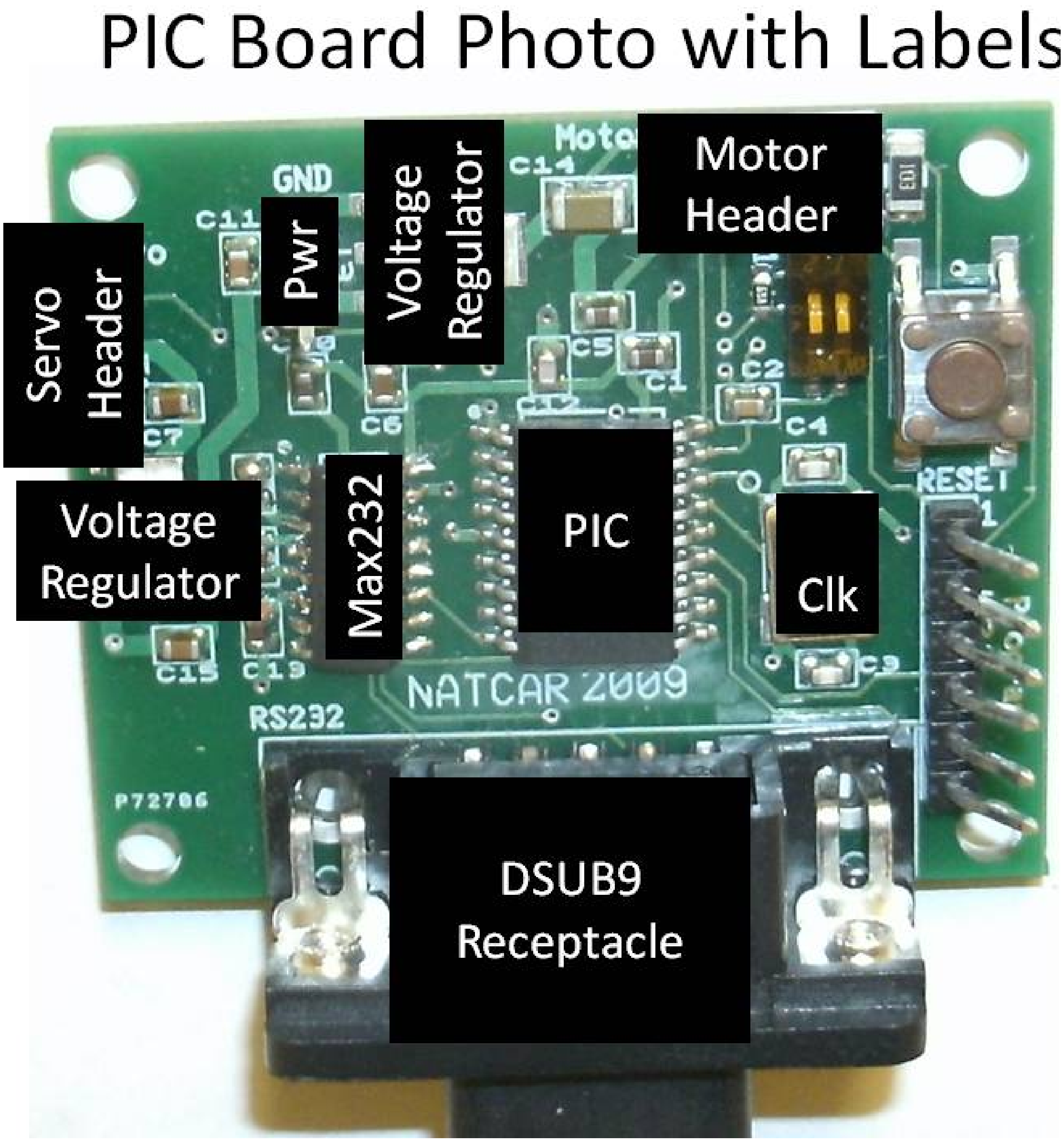


Figure 5-15: PIC Board Photo with Labels

Kellen Butler, an Oklahoma State University student designed the schematic and printed circuit board layout. Megan Brady, an Oklahoma State University student, fabricated the PIC boards.

5.3 Modeling and Controller Design

A system model needs developed to enable controller design. The system model will be part of the controller design loop. The first step in the controller design loop is building or updating a system model. This is a set of general equations that describe how the different parts of the system interact. The second design step is to define the system parameters. For the smart sensor car, this includes car speed, possible steering angles and car length. The third design step is to select the proportional and derivative gains. The fourth step in designing the controller is simulation. The closed loop system is simulated to verify proper response. If the response is not satisfactory, the control gain can be adjusted. The final control gains are loaded into the PIC software, which is then implemented with the entire physical system for testing. Based on the results of the physical testing, the model and controller are updated. This restarts the design process, and the process will continue until all the measures of performance are met. Figure 5-16 shows the controller design loop.

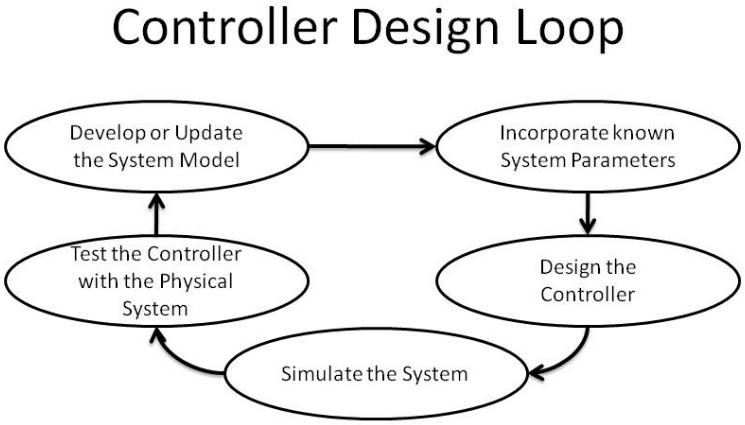

Controller Design Loop
Develop or Update the System Model
Incorporate known System Parameters
Design the Controller
Simulate the System
Test the Controller with the Physical System

Figure 5-16: Controller Design Loop

The smart sensor car can be modeled like the wheeled robot described in [3] and [4]. They describe the motion of the vehicle with equations 1-3 which correspond to Figure 5-17:

$$(1) \quad \dot{x} = v * \cos(\phi) * \cos(\theta)$$

$$(2) \quad \dot{y} = v * \cos(\phi) * \sin(\theta)$$

$$(3) \quad \dot{\theta} = \frac{v}{L} * \sin(\phi)$$

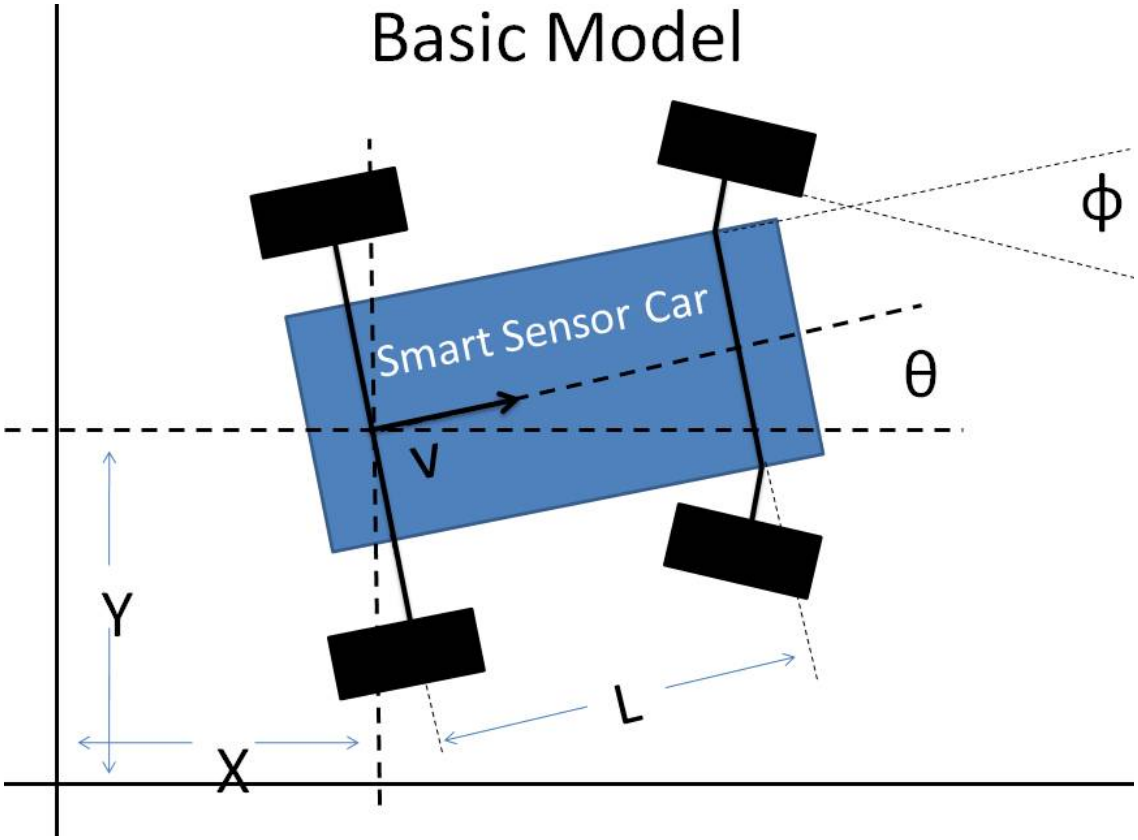


Figure 5-17: Basic Plant Model

The system model equations were implemented as a Simulink model. Figure 5-18 shows the Simulink model for the basic plant. The plant model has the car length modeled at one quarter of a meter and the car velocity is three meters per second. The plant input is the steering angle, φ. The input goes through trig function blocks, then gains, and other multiplication blocks. The blocks are arranged in such a way to represent the model equations. The integrator block outputs are the state variables.

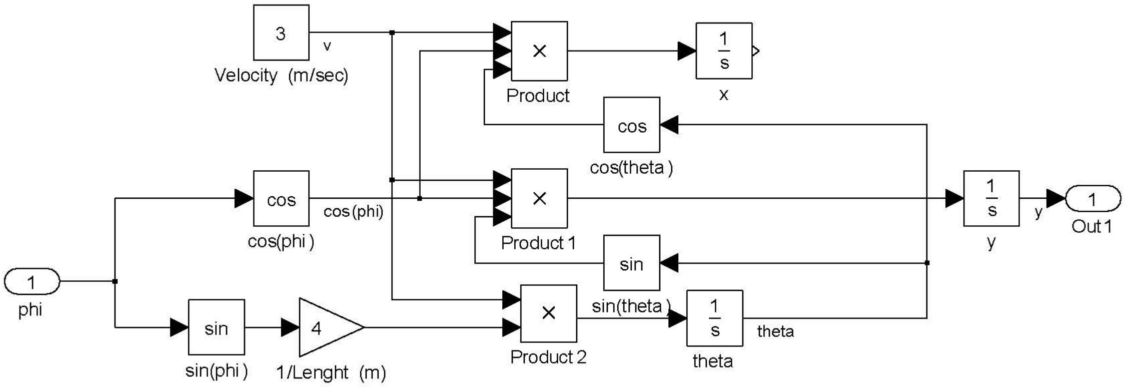


Figure 5-18: Simulink Model of Plant

The MATLAB software is able to take the Simulink nonlinear plant model and make a linearized set of equations to represent it. The software can take the linear system and form a transfer function to represent the original plant model. The Simulink model of the plant is saved as the file, “`CarModelDesign`.” Inside the MATLAB software, the commands to linearize the model and provide a transfer function are:

```
linsys = linearize ('CarModelDesign');
tf_model = tf(linsys);
```

The resulting state equations from this command are:

$$\begin{matrix}\dot{\theta}\\ \dot{y}\end{matrix} = \begin{bmatrix}0 & 0\\ 3 & 0\end{bmatrix}\begin{matrix}\theta\\ y\end{matrix} + \begin{bmatrix}12\\ 0\end{bmatrix}\phi$$

$$y = \begin{bmatrix}0 & 1\end{bmatrix}\begin{matrix}\theta\\ y\end{matrix} + [0]\phi$$

This model transfer function is:

$$\frac{y}{\phi} = \frac{36}{s^2}$$

The transfer function has two repeated roots at the origin. If only proportional feedback is used $(\phi = k_p * y)$, then the root locus is shown in Figure 5-19. It is not possible to produce a stable system with simple proportional control.

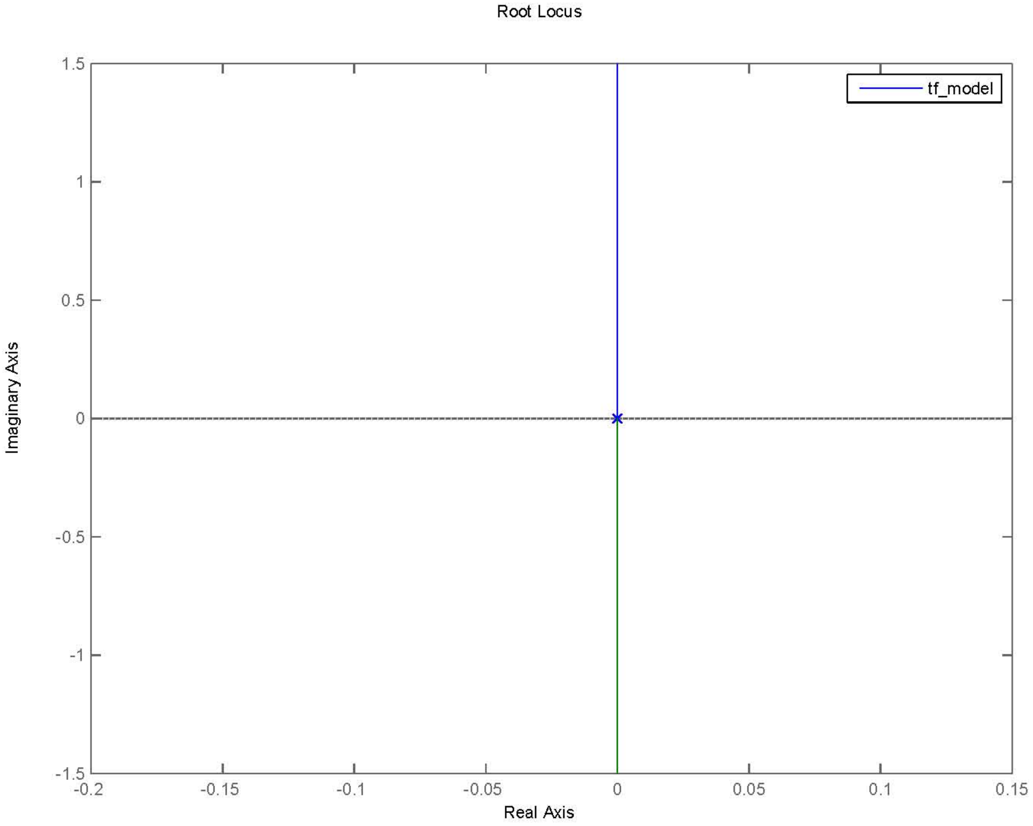


Figure 5-19: Root Locus of Linearized Car Model

To verify the model behaves as expected, a fixed steering angle of 7.5 degrees is input to the Simulink model, as shown in Figure 5-20.

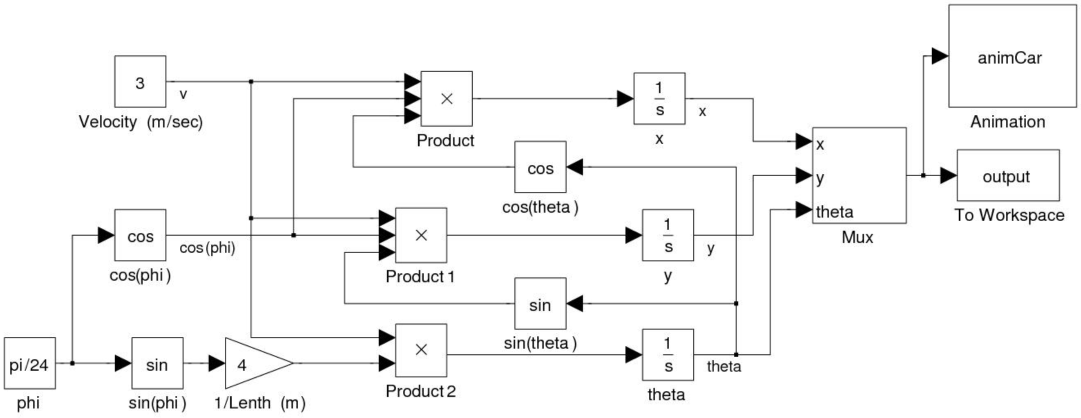


Figure 5-20: Open Loop Simulink Model with Fixed Steering Angle

The expected outcome from this model is the car going in a large circle. Figure 5-21 shows the results from the Simulink model.

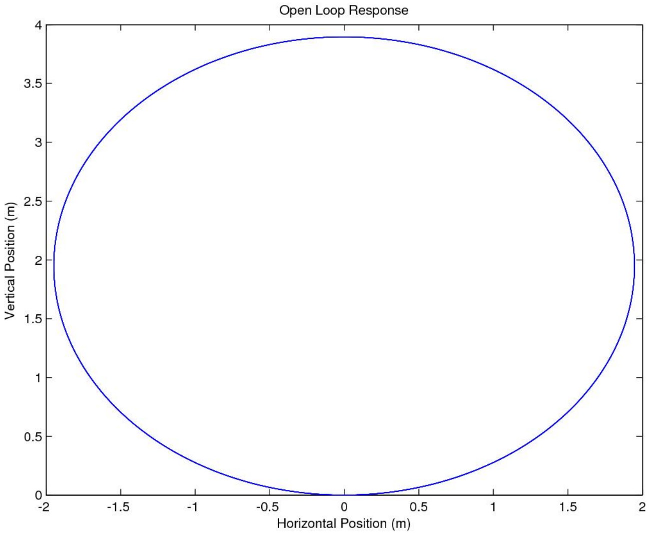


Figure 5-21: Open Loop Car Simulation Response for 7.5 degree Steering Angle

Figure 5-19 shows that the car model is marginally stable with two poles on the imaginary axis. A proportional controller alone will not move the roots of the system to the left of the imaginary axis. A proportional plus derivative controller can add damping which would allow the roots of the system to move left of the imaginary axis and improve the system stability. A proportional plus derivative controller can be represented by the following transfer function.

$$G_c = k_d s + k_p$$

The closed loop system transfer function is

$$\frac{y}{u} = \frac{36\left(k_d s + k_p\right)}{s^2 + 36\left(k_d s + k_p\right)}$$

Figure 5-22 shows the Simulink model for the closed loop system.

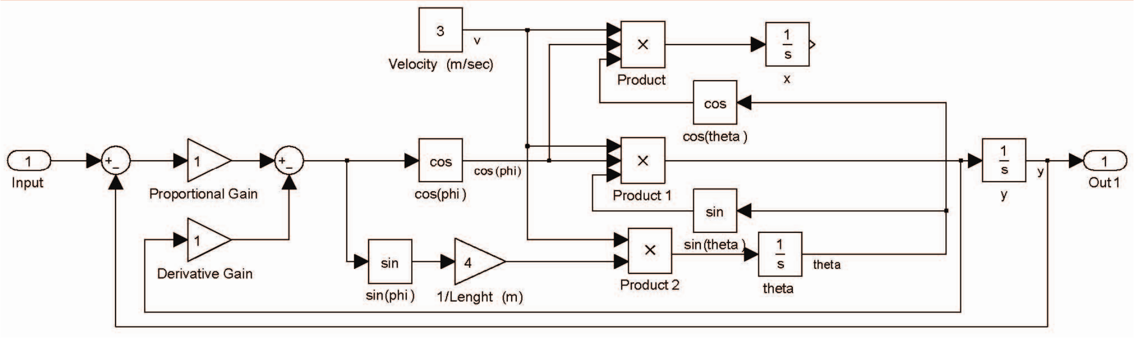


Figure 5-22: Closed Loop Simulink Model

The characteristic equation for this system is:

$$s^2 + 36k_d s + 36k_p$$

The characteristic equation matches the standard form:

$$s^2 + 2\zeta\omega_n + \omega_n^2$$

For this controller design, the settling time is set to one second and the output is set to be critically damped, or ζ is set to one. Assuming that the response should settle within a few percent of steady state after four time constants, the settling time can be estimated by:

$$T_s = \frac{4}{\zeta\omega_n}$$

So $\omega_n = 4$, $k_d = {}^{2}/_{9}$ and $k_p = {}^{4}/_{9}$.

To validate the controller design, the closed loop model was given an initial position for y of 0.1 m or ten centimeters. The model had no other input, so the system should settle close to zero after one second. Figure 5-23 shows the model response. Although the original model is nonlinear, the controller design based on the linearized model is satisfactory.

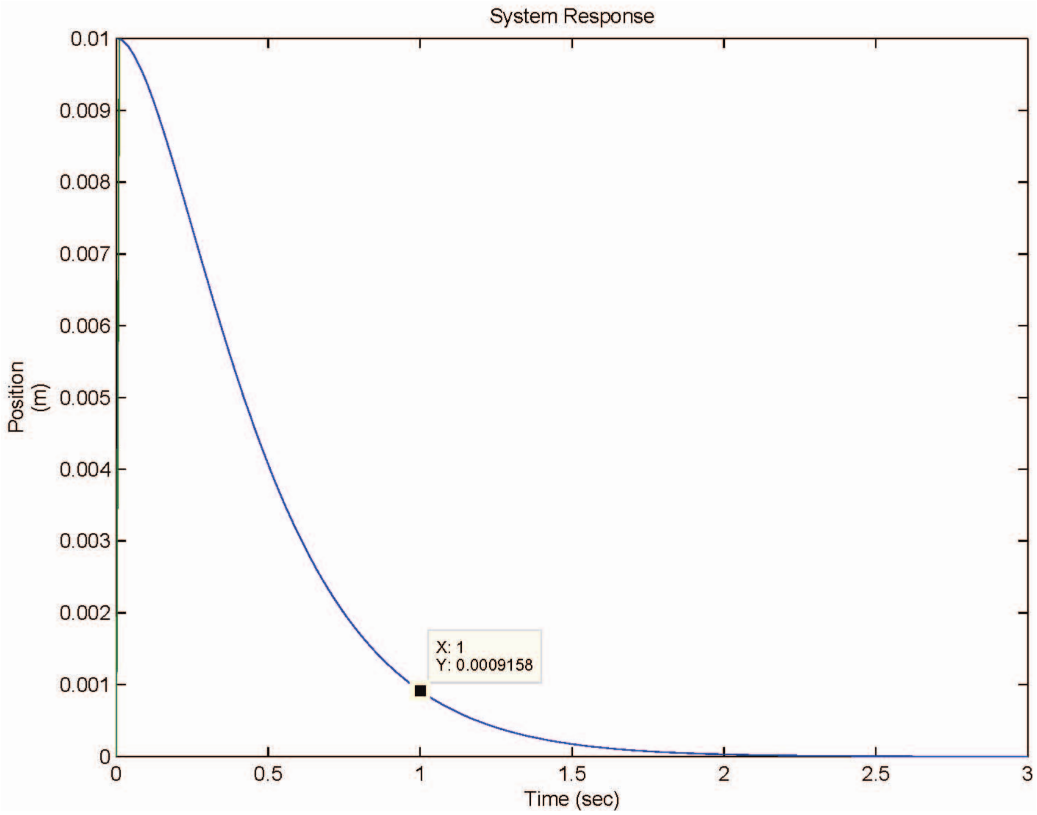


Figure 5-23: Validation of Model Response

The plant model and controller use units of meters. The FPGA provides positions in hundredths of an inch. This is why the real-time executive algorithm must convert the FPGA distance measurement from units of inches into units of meters.

There are three parameters of the car model that need to be set. The first parameter is length. The length between the front and rear axle of the car is 25 cm, or one quarter of a meter. The second parameter for the car is forward velocity. The fastest the smart sensor car will travel at the summer academy is three meters per second. The third parameter that will affect the model is maximum allowed steering angle. The steering mechanism can only turn the wheels so far before it runs out of travel. Figure 5-24 shows the steering mechanism turning the wheel as far as possible to the left. The steering mechanism runs out of travel after turning the wheels 30 degrees or $^{\pi}/_{6}$ radians.

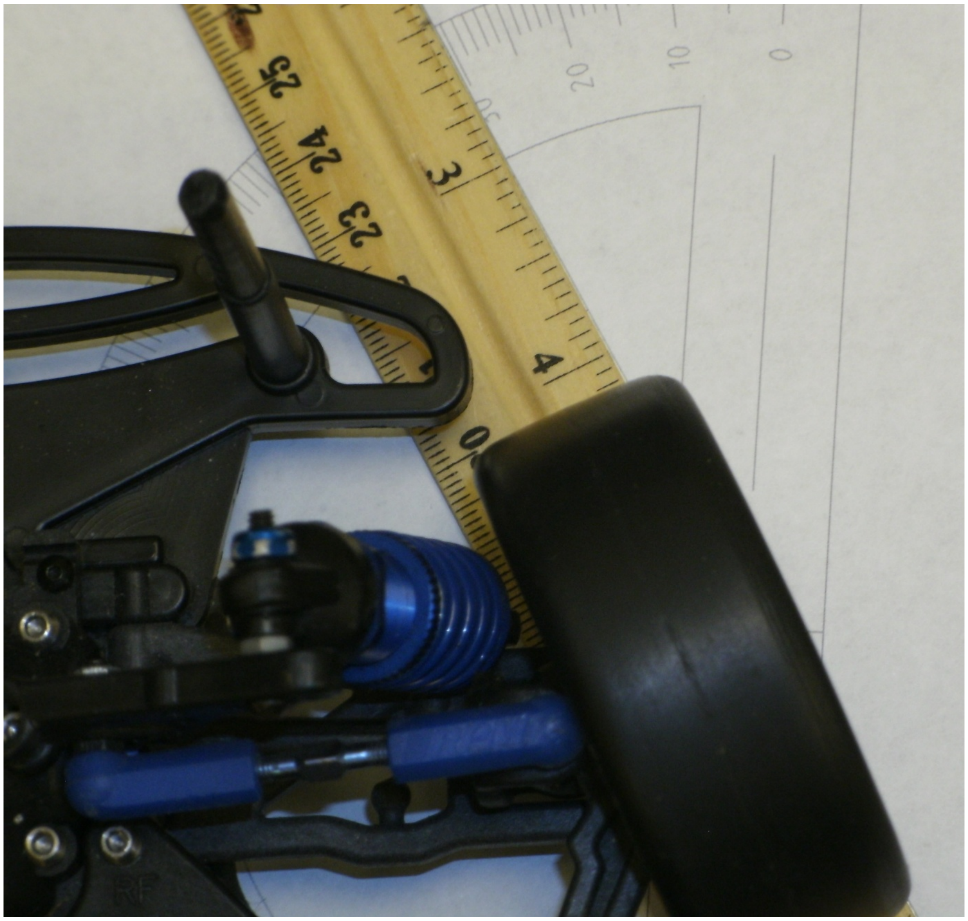


Figure 5-24: Maximum Possible Steering Angle

The system model should incorporate the same limitations. To include the steering limitation, a saturation block is inserted to limit the possible steering angle before it is fed into the rest of the plant model. Figure 5-25 shows the plant and controller with the saturation block.

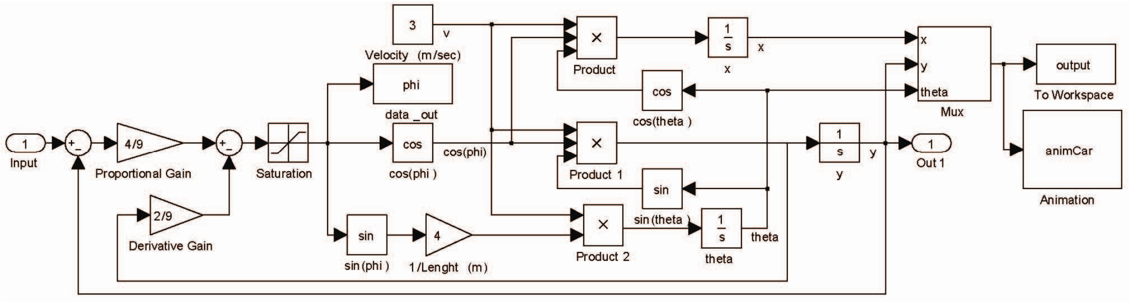


Figure 5-25: Plant and Controller Simulink Model with Steering Saturation Block

The gains are then adjusted to obtain the fastest response possible while not saturating the steering angle. Because the simulation begins with a position of 0.1 meters, and the maximum allowed steering angle is ${}^{\pi}/_{6}$ radians, the proportional gain should be less than $10 * {}^{\pi}/_{6}$. The simulation was ran with the fixed proportional gain. The derivative gain was varied in each run to find the fastest response time. Additional simulations with the final model suggest using gains of 0.749 for the derivative gain and 5.043 for the proportional gain. The resulting linearized closed-loop system transfer function is:

$$\frac{y}{u} = \frac{181.5}{s^2 + 26.95s + 181.5}$$

This transfer function has the pole zero map shown in Figure 5-26. Because there are two poles on the real axis, the step response should have no oscillation. In addition, because the real components of the roots are more than eight, the system response should settle in less than half a second.

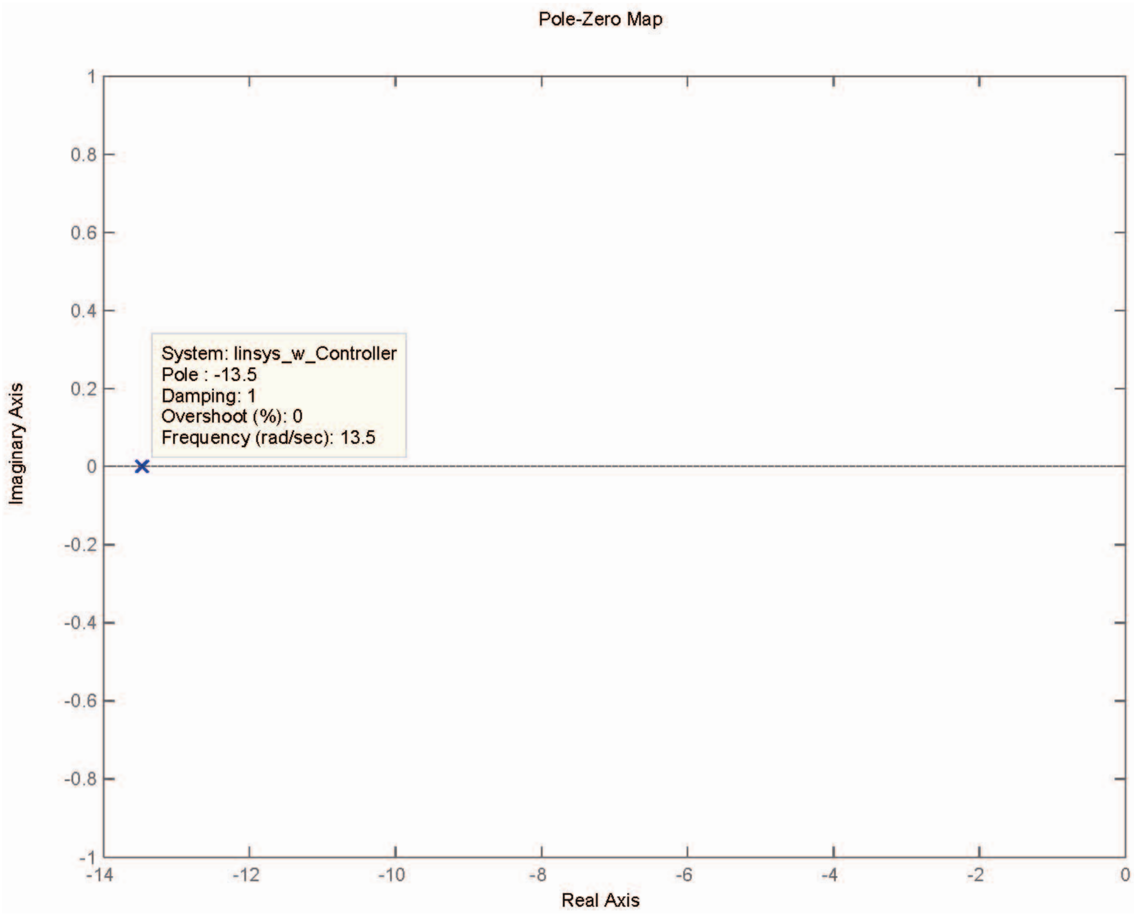


Figure 5-26: Pole Zero Map for the Controller and Plant Model

Figure 5-27 shows the model response to a ten-centimeter initial error. When time is about 0.3 seconds, position should be about (1-0.911)*-0.1 meters, or approximately -0.0089. The Figure shows that model response is close to the theoretical expectations.

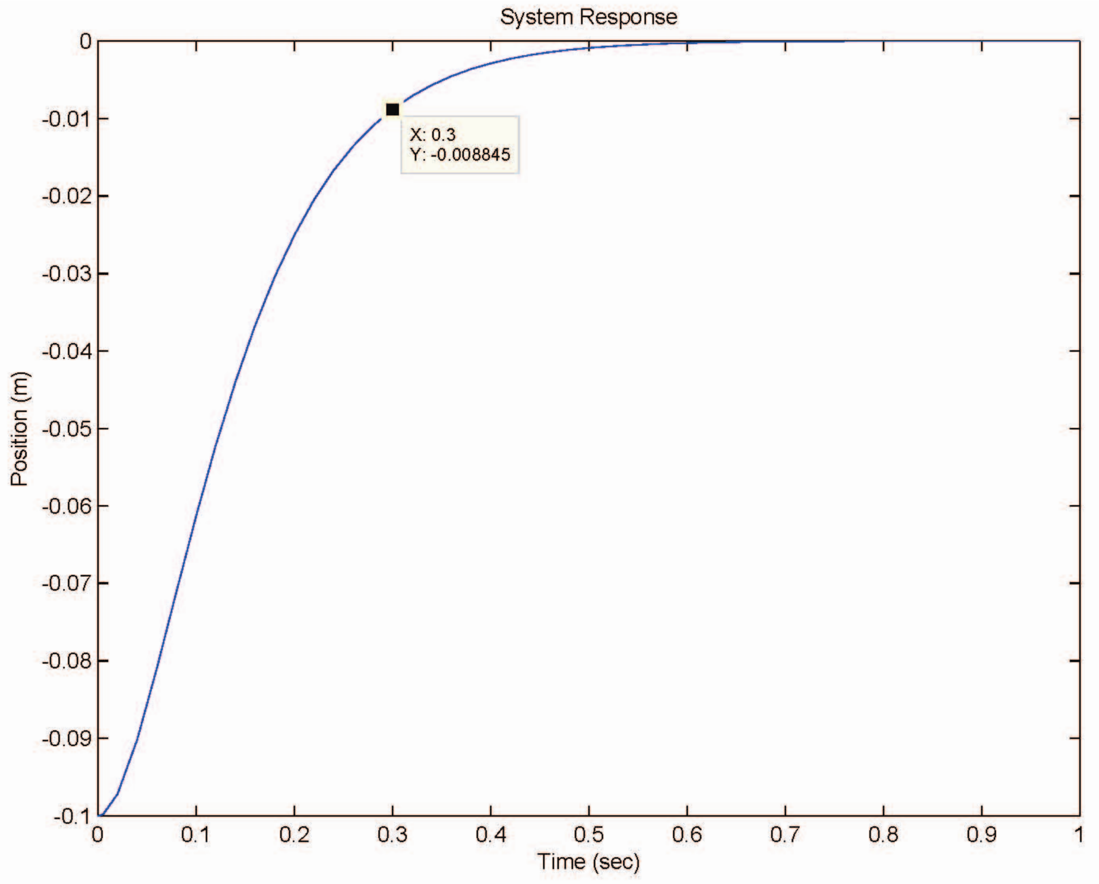


Figure 5-27: Model Response with Tuned Controller

The plots show that the model output and theoretical expected response are similar. The current gains should provide the best response for the car and not send commands to the servo that over extend the steering mechanism. This system model provides a safe starting point to begin testing the smart sensor car performance in following a straight wire.

During testing of the smart sensor car, the final gains were chosen by a heuristic optimization process. The proportional and derivative gains were adjusted one at a time and car performance in following a wire was measured. The gains that produced the best performance were used during the summer academy.

5.4 Support Equipment

Support equipment is not part of the process while the car is going around the track, but is a necessary part of the smart sensor car project. Separate support equipment is required for the software and hardware.

The PIC MCU C Compiler by Custom Computer Services, Inc. provided a platform to write the code for the real-time executive algorithm. The compiler came with a library of functions that simplified many of the processes, including the RS232 communication. The compiler converted the code into assembly language files for use with other software.

The Microchip MPLAB Integrated Development Environment provided an interface between the personal computer running the compiler and the hardware that could connect to the PIC board. The MPLAB IDE software would take the assembly language file and convert it into machine code for the microcontroller. The software then sent the machine code from the personal computer to the programmer.

The Microchip PICkit 3 programmer provides the hardware interface between the personal computer and the PIC board. The programmer takes the machine code from the personal computer and writes it to the memory of the microcontroller.

5.5 Summary

The PD controller takes a distance measurement from the neural network and calculates the steering angle to keep the car as close to the track as possible. The controller executive software controls the process flow. The control algorithm runs on a PIC microcontroller, which requires its own board.

CHAPTER VI

ELECTRIC CAR

This chapter discuses the electric car. The electric car takes inputs from the PD controller and provides motion around the track. The car provides the platform for motion while holding all other components together. Figure 6-1 is the main block diagram that shows how the components work together.

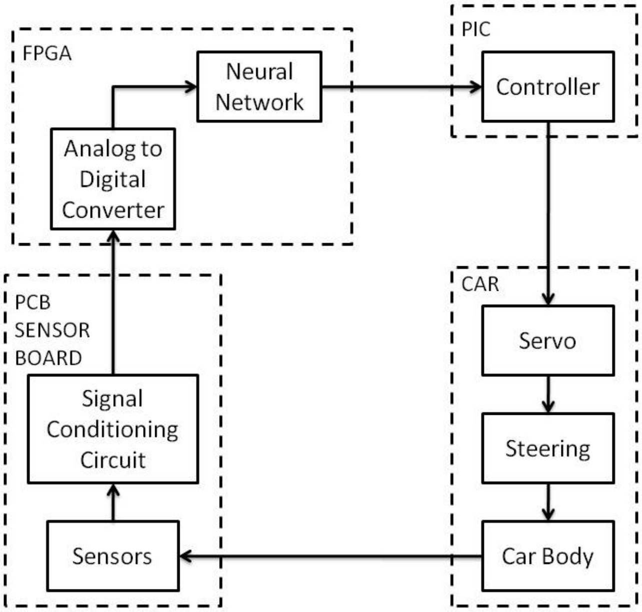


Figure 6-1: Main Block Diagram

6.1 Overview

The electric car provides the platform for motion while holding all other components together. The car is a standard one tenth-scale hobby car and uses the stock motor that came with the car kit. The PD controller sends commands to the car motor control switch and steering servo. The motor control switch provides current to the motor, which results in forward motion. The steering servo adjusts the car steering angle and the car moves along the track. The car motion moves the sensor board into a new location with a unique magnetic field. The field generates a new response from the signal conditioning circuit, which is fed through the Analog to Digital (A2D) converters and into the neural network. The neural network provides the PD controller with a new distance measurement, which the PD controller uses to calculate the next steering angle and speed commands. Figure 6-2 shows the system functional overview, which describes how the different system components relate to each other.

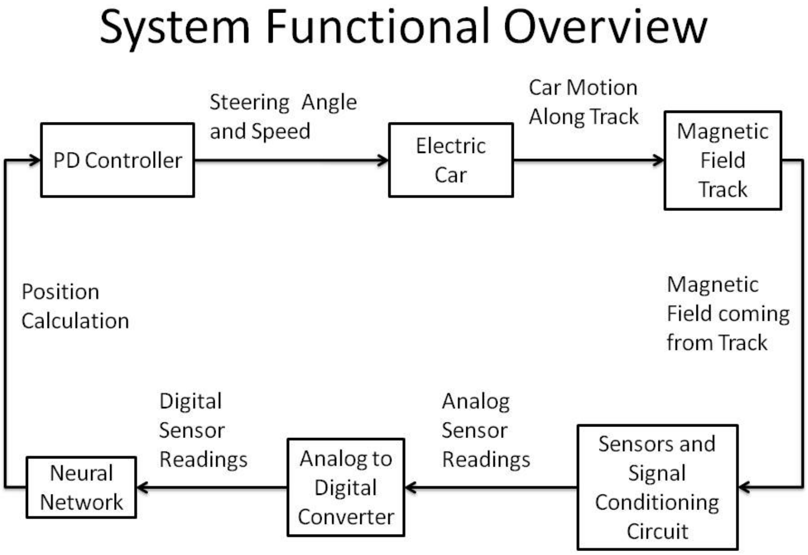


Figure 6-2: System Functional Overview

6.2 Components

Several components make up the electric car. The chassis is the basic piece that holds everything else together. The motor converts current into torque to spin the tires. The batteries provide power for the motor control switch, sensor board, FPGA board and PIC board. The motor control switch provides current to the motor. The servo steers the front tires. All of the boards are mounted to the car chassis with the board-mounting fixture. Figure 6-3 shows the electric car and components.

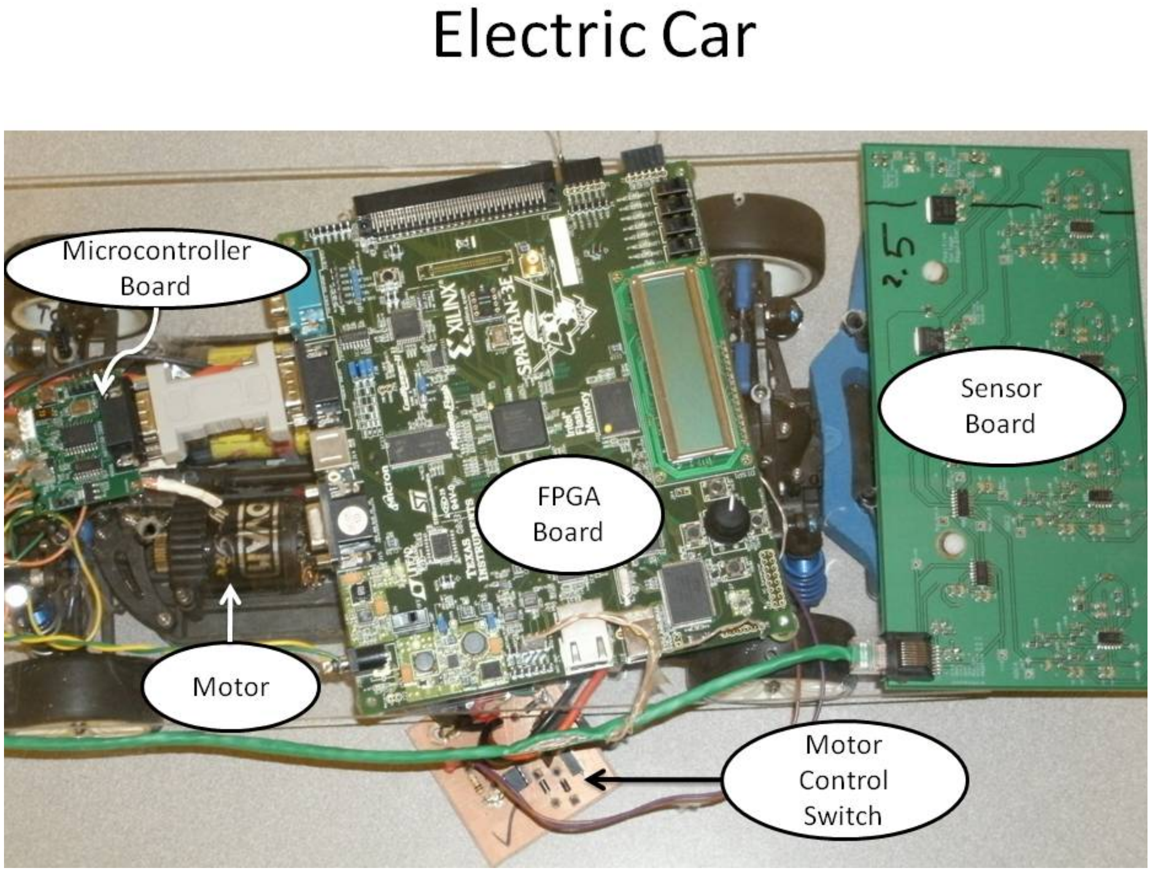


Figure 6-3: Electric Car

6.2.1 Chassis

The car chassis is a standard one tenth-scale hobby car. This chassis is a Tamya USA TT-01, on-road, shaft driven all wheel drive bathtub chassis. Figure 6-4 shows the car chassis with the battery, motor control switch, motor and steering servo.

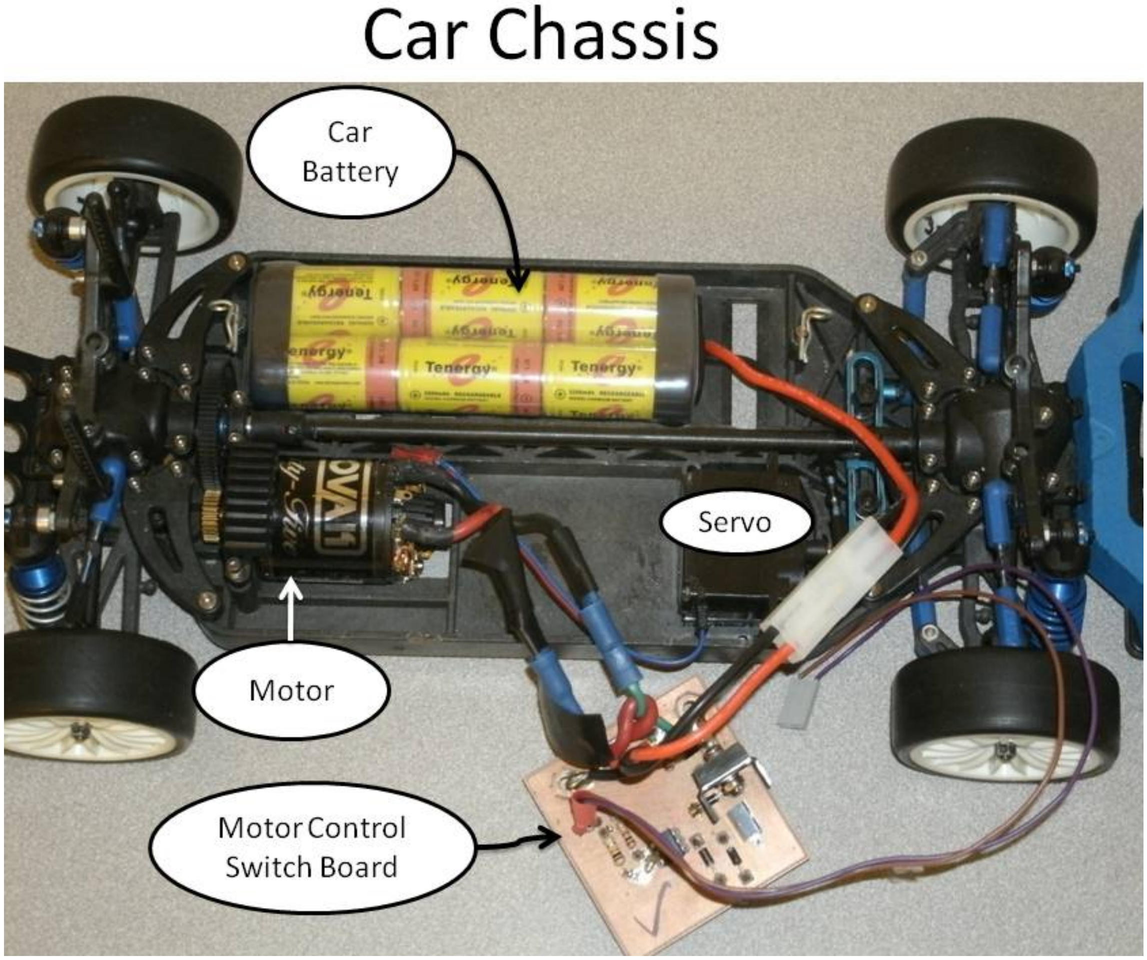


Figure 6-4: Car Chassis

The steering mechanism for the car chassis is a four bar linkage that forms a parallelogram. The frame provides the first bar, which can be thought of as the right side of the parallelogram. The two trailing arms form the top and bottom of the parallelogram. The two trailing arms stay parallel as the car steering angle changes. The trailing arms have three connection points. The leading edge of each arm is connected to the frame. The middle is connected to the tire. The third connection is to the steering linkage. The steering linkage is the fourth bar and fourth side of the parallelogram. The steering servo connects to the linkage to provide animation to the system. Figure 6-5 shows the steering mechanism parallelogram drawing.

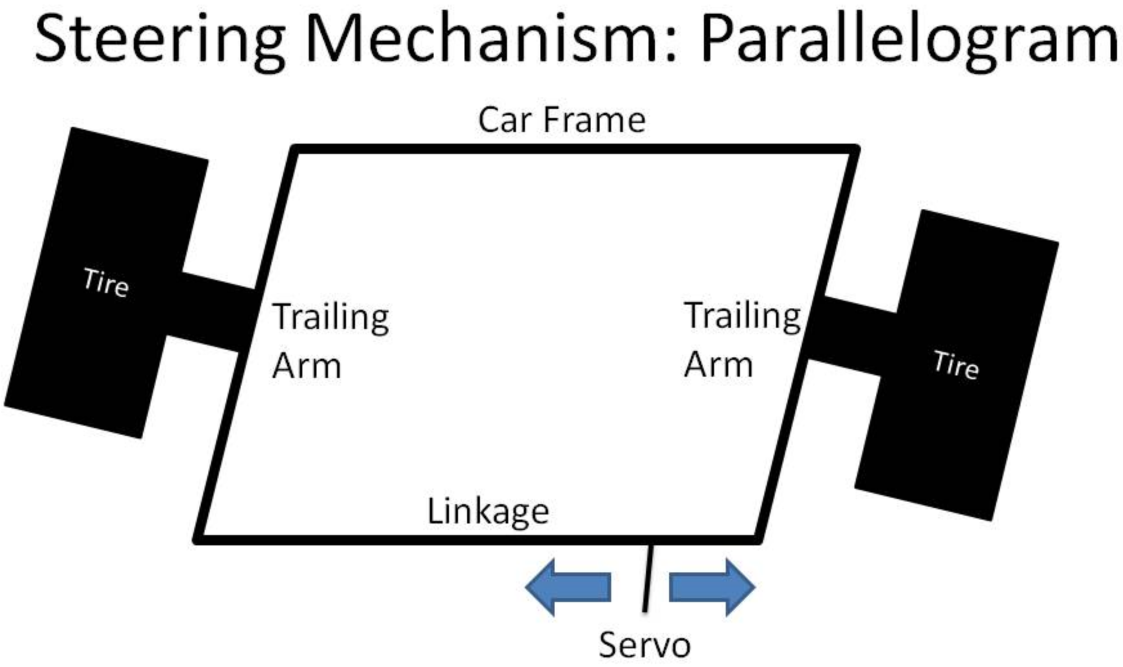
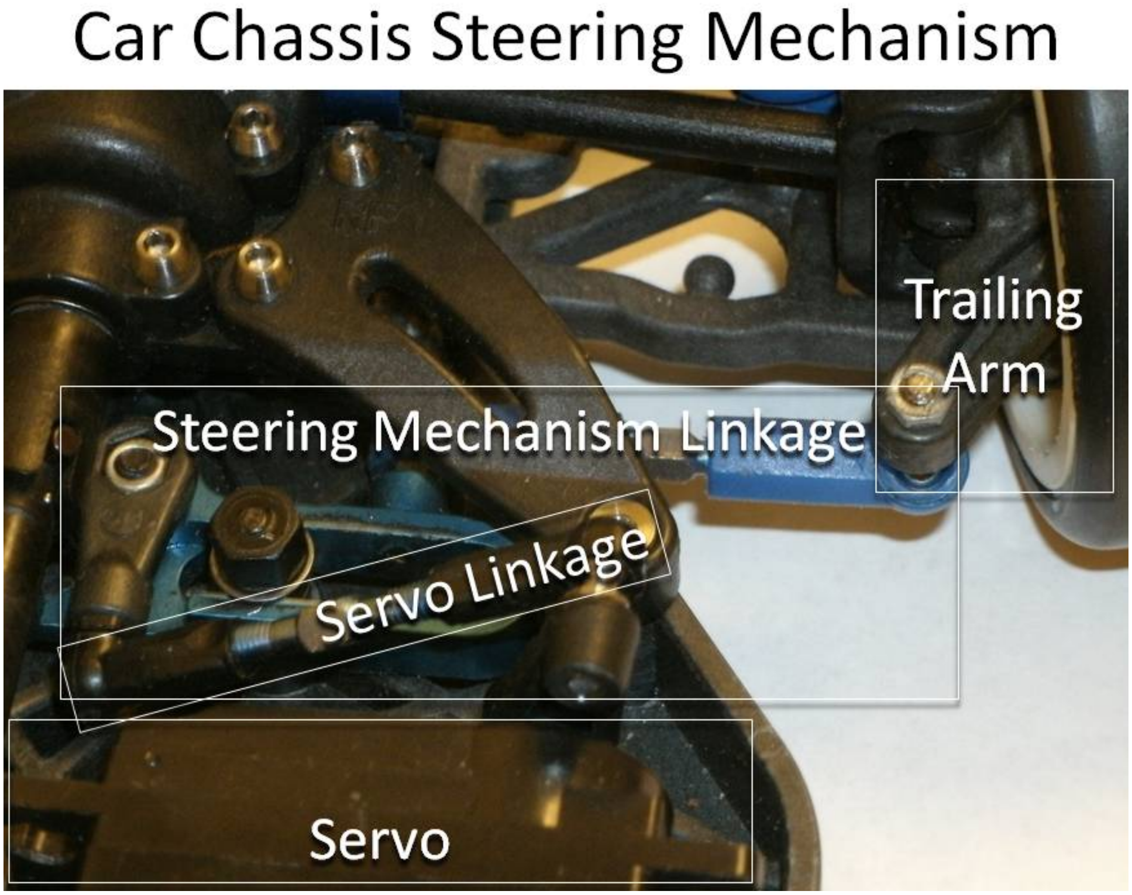


Figure 6-5: Steering Mechanism Parallelogram

Figure 6-6 shows a photograph of the car steering mechanism.



Car Chassis Steering Mechanism

Figure 6-6: Car Chassis Steering Mechanism

6.2.2 Motor

The car motor is a standard size 540 brushed electric motor that came with the chassis. The motor has two 14-gage wires with male bullet connector terminations to interface with the motor control switch board. The large wires adequately handle the considerable current passing through the motor. Other motors were used in testing. A rock crawler motor is a low speed high torque motor, which allows the car move slowly. The rock crawler motor is designed for long periods of use at high torque and low speed without requiring maintenance. This was useful during the initial stages of debugging. Another high-speed racing motor was considered for the summer academy. This motor required rebuilding per few hours of use, and was designed with an operating point much faster than the desired speed for the summer academy. The racing motor was not used. Figure 6-7 shows the testing motor in the electric car.

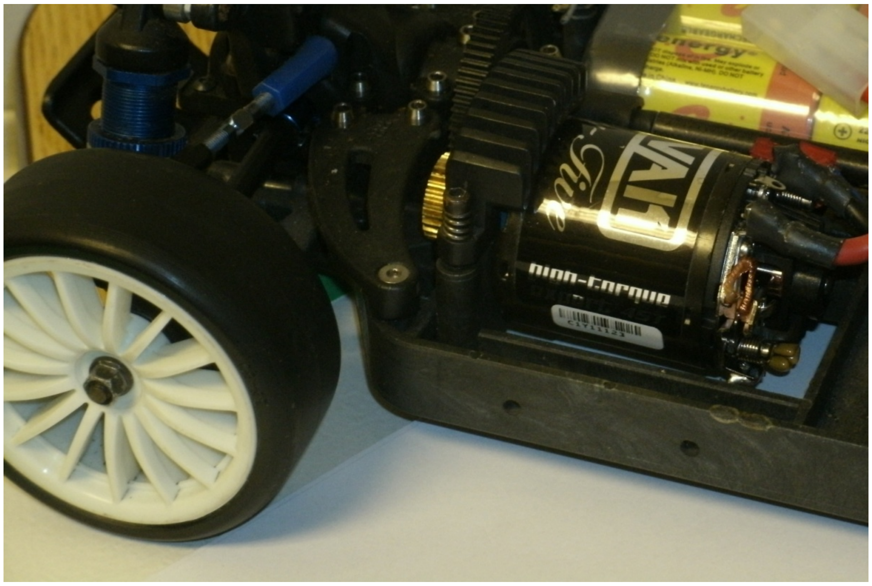


Figure 6-7: Car Motor

6.2.3 Batteries

The smart sensor car has three battery packs. The first two battery packs power the sensor board, FPGA board and PIC board. These are generic 9.6-volt Nickel-Metal Hydride 8 cell battery packs, rated to last 2000 milliamp hours. These two batteries connect to the main power switch on the board-mounting fixture. The third battery powers the motor control switch that provides power to the motor. This is a generic 6-cell 7.2-volt Nickel Cadmium battery, rated to last 2200 milliamp hours. All three batteries have male two conductor RC battery connector plugs. Figure 6-8 shows the three batteries for the smart sensor car.

Car Batteries

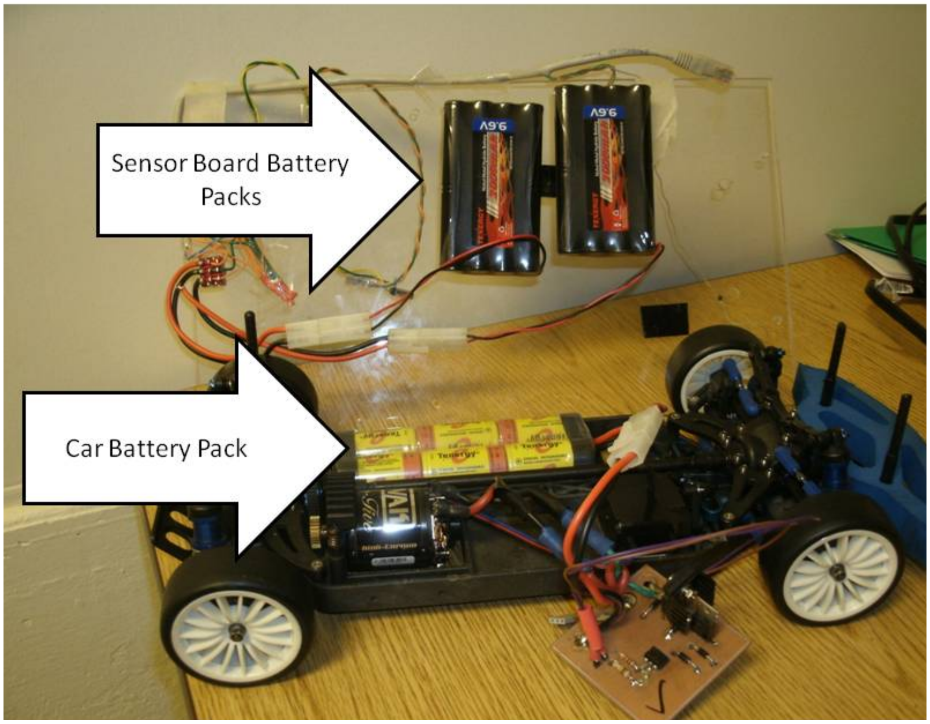


Figure 6-8: Car Batteries

6.2.4 Motor Control Switch

The purpose of the motor control switch is to convert the pulse-width modulated motor drive signal from the PIC microcontroller into current flow from the battery to the motor. The motor control switch has three connections. The first connection is a pair of female bullet connectors to connect to the car motor. These connectors are tied to the motor control switch board with fourteen-gage wire to adequately handle the large amount of current flowing through the motor. The second connector is a female two-conductor RC battery connector plug for use with the 7.2-volt battery pack. The last connector is a four pin female header to connect to the four pin male header of the PIC board. Pin one of the header provides the ground reference and pin three carries the control signal from the microcontroller to the motor control switch board. The control signal drives the LED side of a 4N33 opti-coupler. The opti-coupler isolates the high current motor circuit from the more sensitive microcontroller circuit. The output of the opti-coupler feeds a Darlington pair of bipolar junction transistors. The first transistor is a TIP41A. The second transistor is a TIP31C, rated to carry a load of 25 amps continuously. When conducting, the TIP31C provides a path for the current to flow from the positive car battery terminal through the motor to the negative car battery terminal. Protection diodes and current limiting resistors help protect circuit components. Figure 6-9 shows the motor control switch schematic.

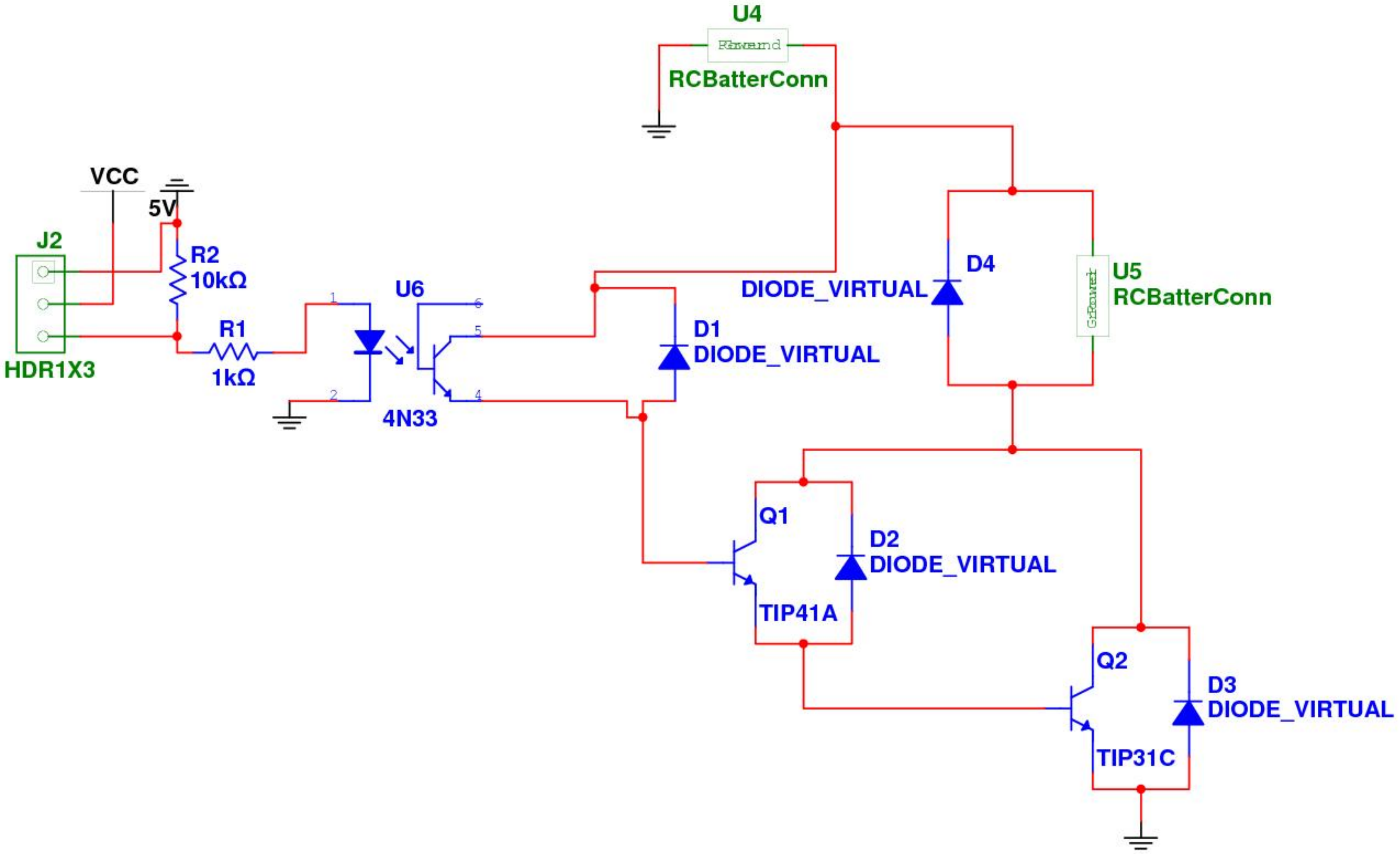


Figure 6-9: Motor Control Switch Schematic

The motor control switch board was designed for ease of fabrication while maintaining the capability of carrying significant currents. Large traces are easier to mill and solder. Figure 6-10 shows the motor control switch board layout.

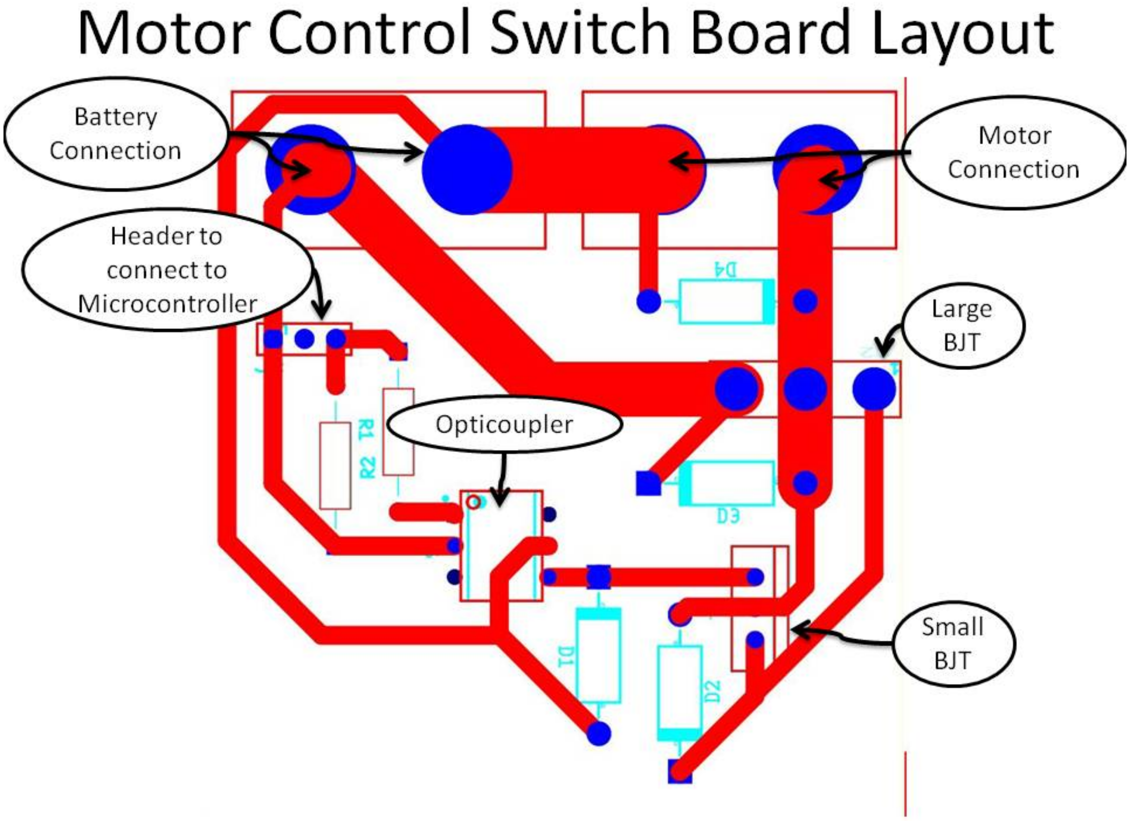


Figure 6-10: Motor Control Switch Board Layout

The large BJT required a heat sink to dissipate heat during long testing periods. Figure 6-11 shows the motor control switch board photograph with part labels.

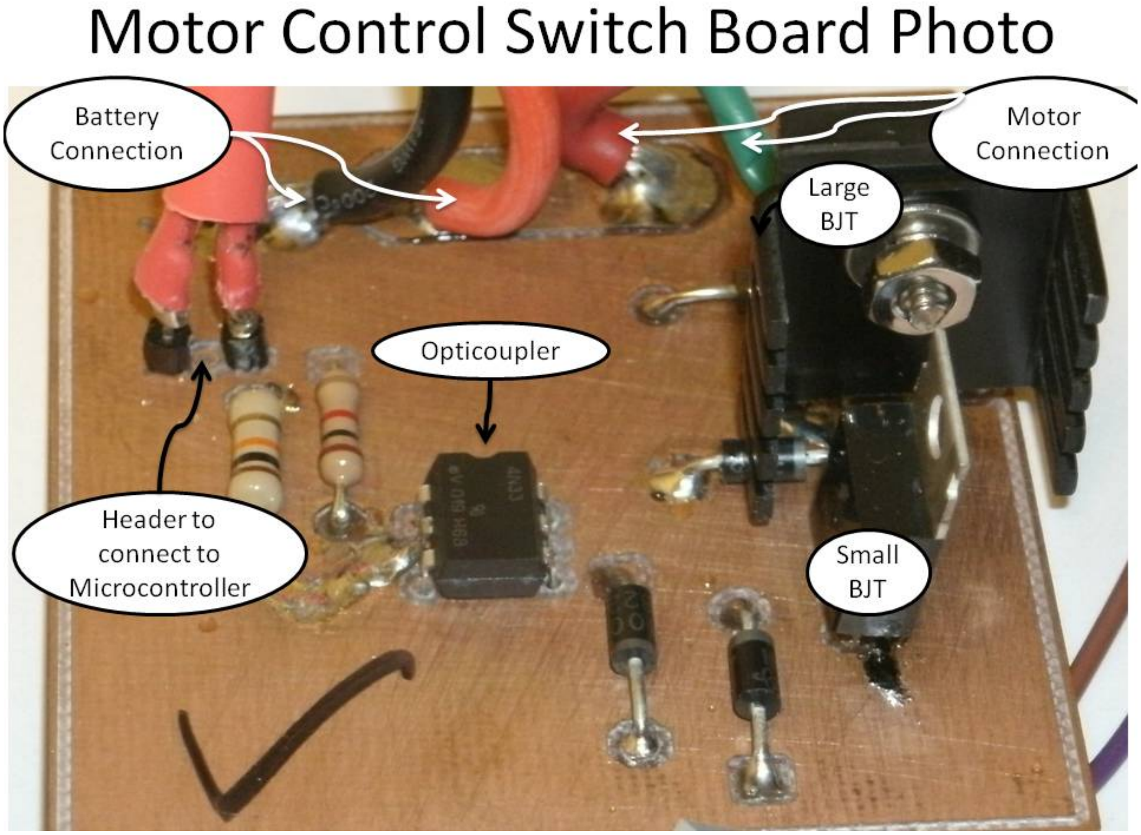

Motor Control Switch Board Photo
Battery Connection
Motor Connection
Large BJT
Opticoupler
Header to connect to Microcontroller
Small BJT

Figure 6-11: Motor Control Switch Board with Labels

The opti-coupler takes inputs from the microcontroller and sends current to the Darlington pair of BJTs. The purpose of the opti-coupler is to protect the more sensitive low current circuit of the microcontroller from the high current circuit of the motor. The opti-coupler is rated to turn on in five μs, and turn off in 100 μs. The TIP41C BJT is the first transistor in a Darlington pair and provides the current to activate the other power transistor. This device has a minimum current gain of 30, a saturation voltage of 0.8

volts, a bandwidth up to 3 MHz, and can supply up to six amps. The TIP35 is the main power transistor for the motor control switch and provides the current path from the battery, through the motor, to ground. It has a minimum current gain of 10, has a bandwidth up to 3 MHz, and can supply up to 25 amps continuously. The TIP35 has a collector-emitter saturation voltage of 1.8 volts.

The motor control switch board converts the pulse-width modulated motor drive signal from the PIC microcontroller into current flow from the positive side of the battery through the motor to the negative side of the battery. Figure 6-12 is a guide to where data was collected for the motor control switch plots.

Guide to Motor Control Switch Plots

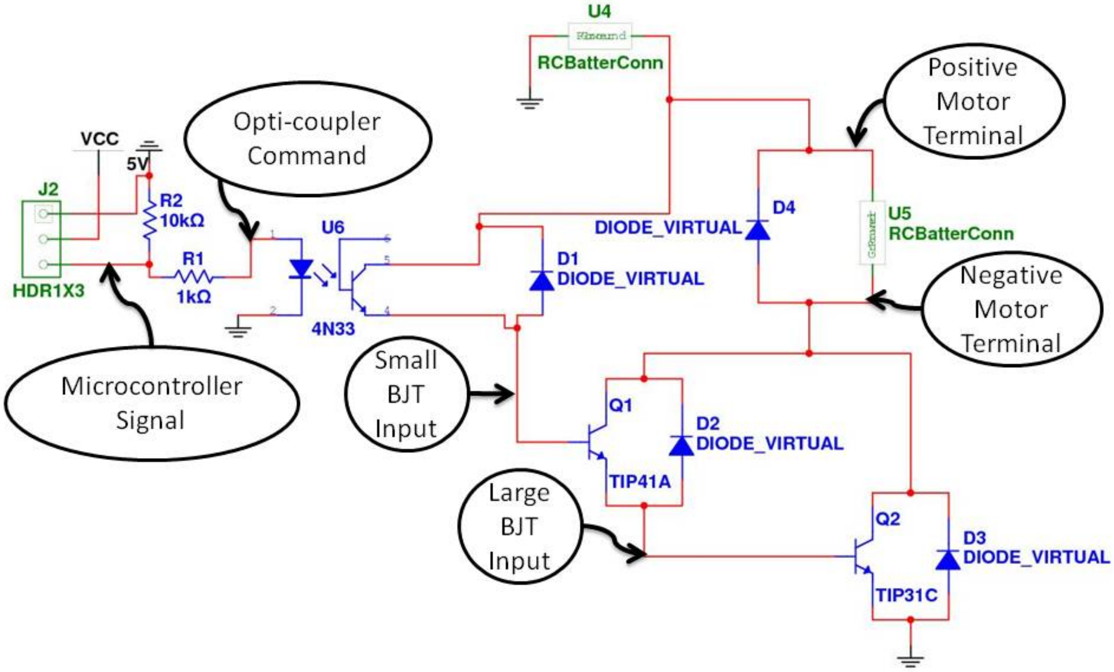


Figure 6-12: Guide to Motor Control Switch Plots

Figure 6-13 shows the microcontroller signal and opticoupler command. To adjust the current needed to trigger the opticoupler, the 10-kOhm resistor in the schematic was replaced with a 671-Ohm resistor, resulting in a much lower voltage coming into the top of the diode of the opticoupler.

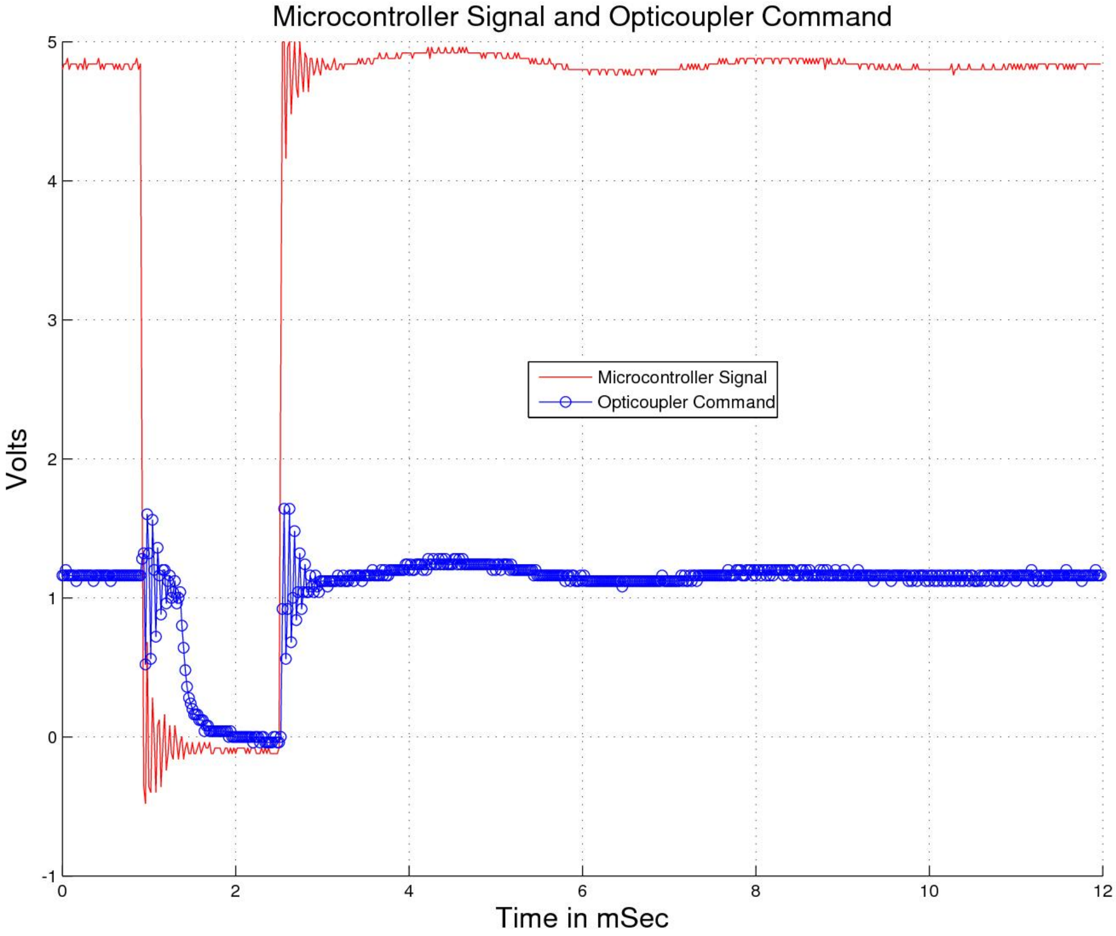


Figure 6-13: PIC Microcontroller Signal and Opticoupler Input

Because of the low speed needed for the operations of the smart sensor cars, neither of the BJTs were supplied enough current to saturate them. The result of operating in the transition phase was that both devices generated considerable heat. Figure 6-14 shows the two signals going into the base terminals of the two BJTs.

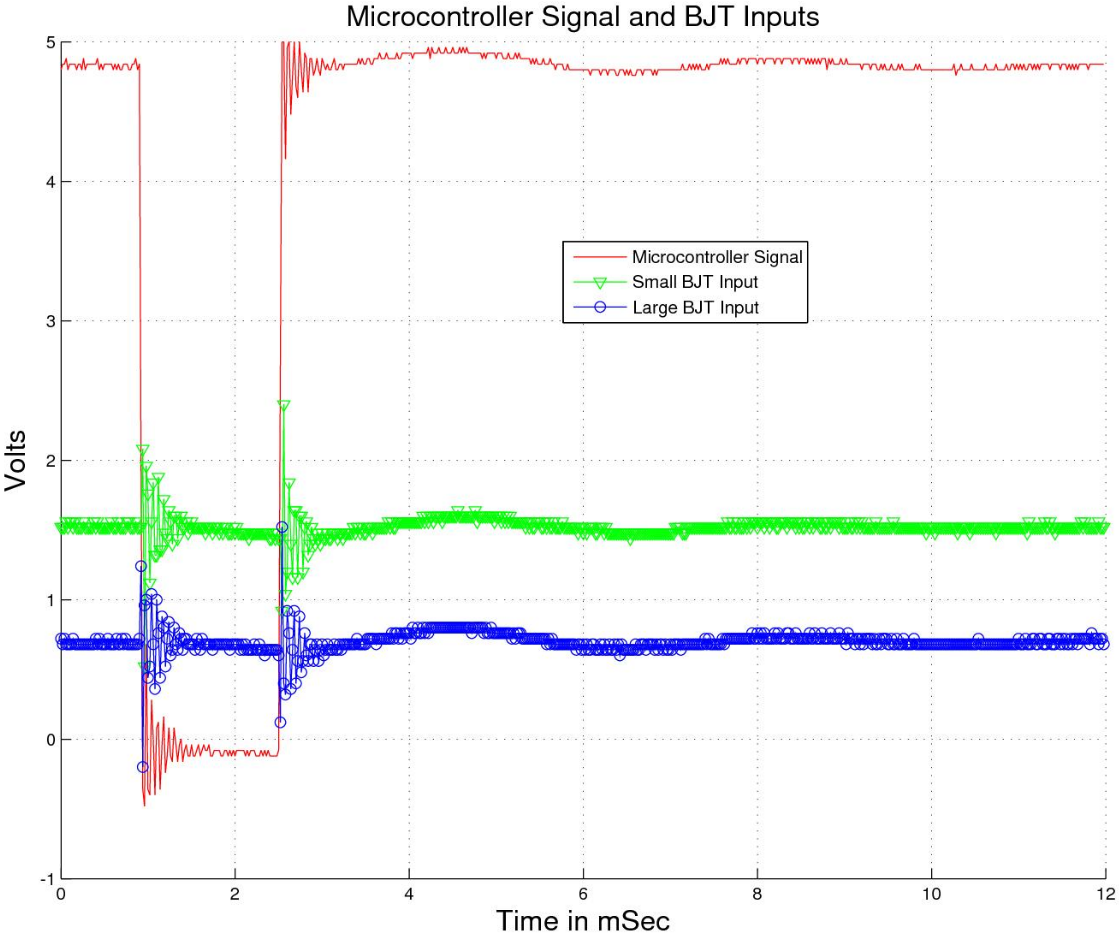


Figure 6-14: Microcontroller Signal and BJT Base Terminal Inputs

Figure 6-15 shows the voltages on the positive and negative terminals of the car motor. Note that when the microcontroller signal changes it has little impact on the voltage across the motor. This signifies that the control signal pulse window is short enough to allow the motor to run smoothly.

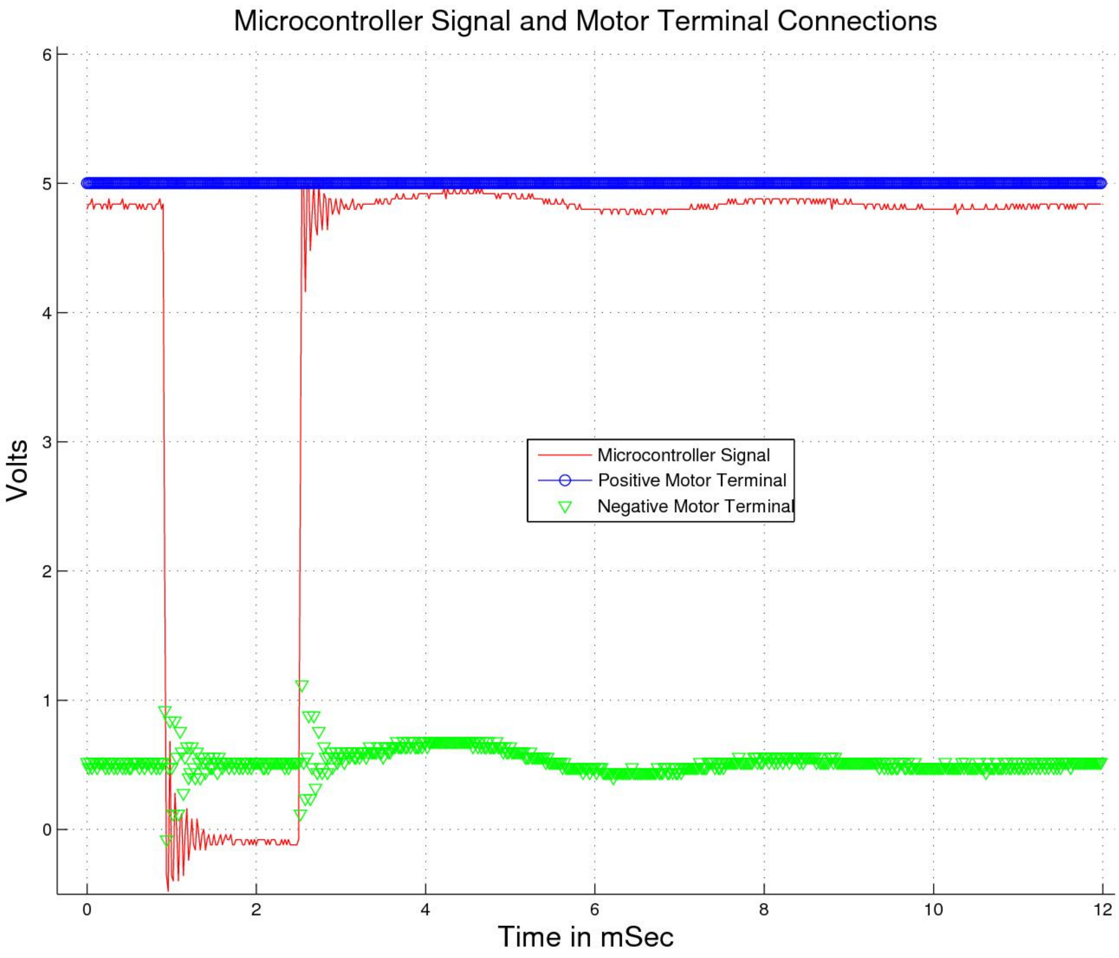


Figure 6-15: Microcontroller Signal and Motor Terminal Connections

Figure 6-16 shows a photo of the motor control switch connected to the battery and car motor. During operation, the motor control switch board is attached to the top of the servo.

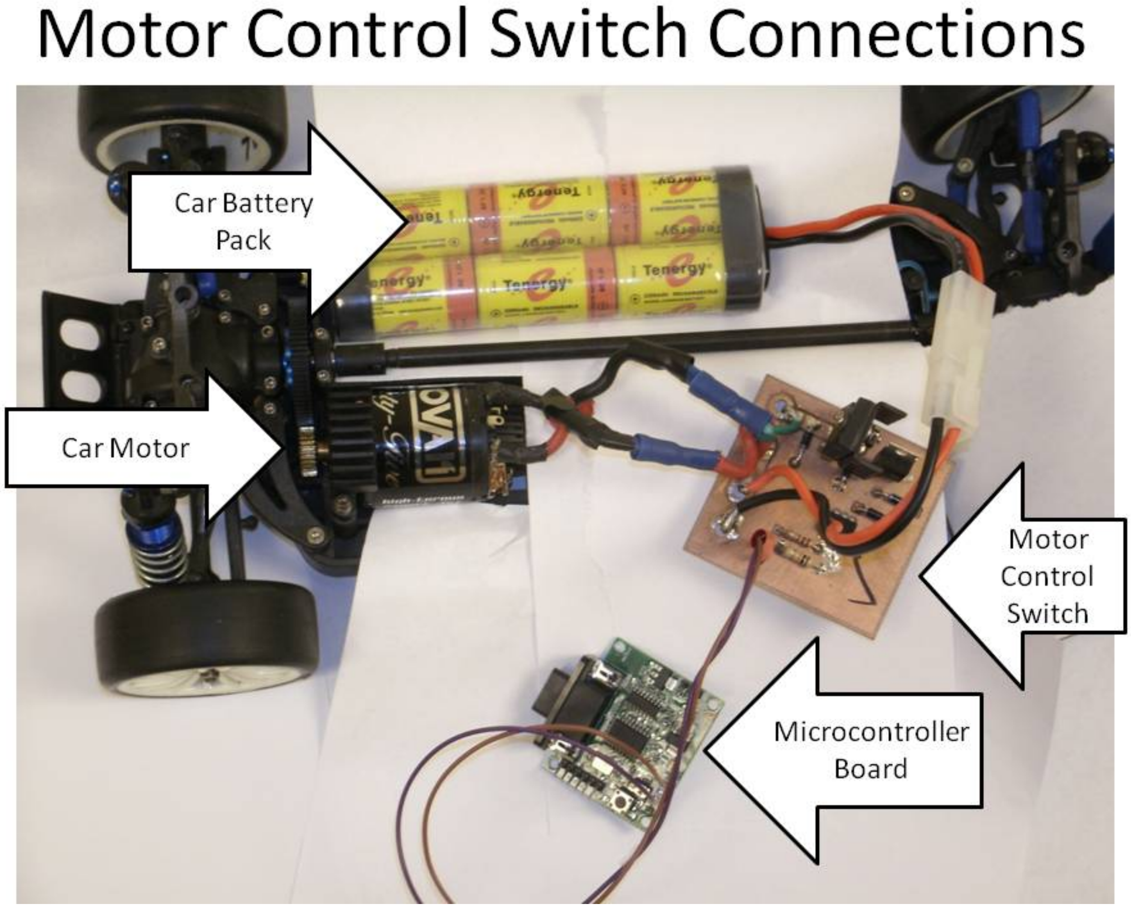


Figure 6-16: Motor Control Switch Connections

6.2.5 Servo

The car servo is a generic Tower Hobbies TS-53 standard servo. A one to two millisecond pulse with a twenty-millisecond window controls the servo. A pulse of 1.5 milliseconds will drive the servo to have the wheels straight forward. A 1 millisecond pulse will drive the servo to have the wheels turn left by 30 degrees. A 2 millisecond pulse will drive the servo to turn the wheels right 30 degrees. Figure 6-17 shows the connection of the servo to the steering mechanism.

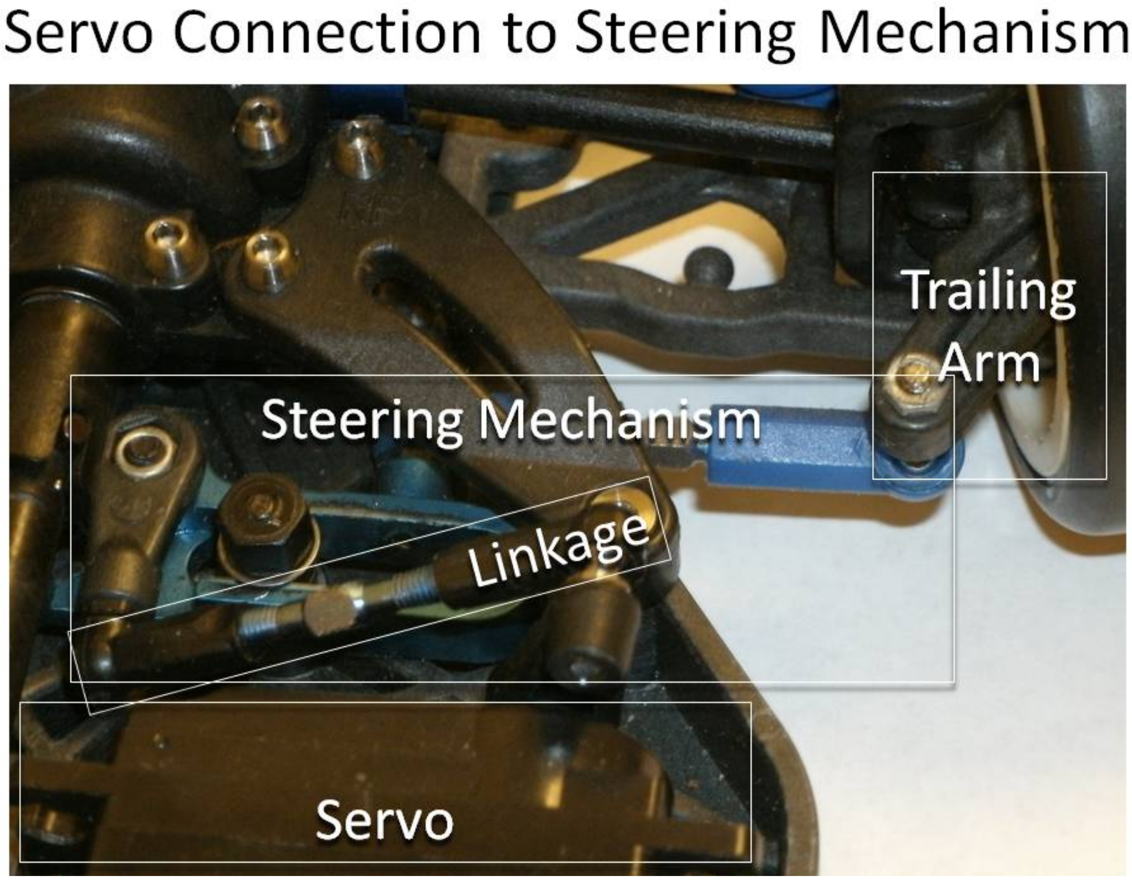


Figure 6-17: Car servo Connection to Steering Mechanism

6.3 Board Mounting

The main block diagram in Figure 6-18 shows the major components of the smart sensor car. The sensor board is attached with screws to the electric car using the board-mounting fixture. The signals from the sensor board go to the A/D converter using a RJ45 connector. The A/D chip and neural network are on the FPGA board and connected by traces. The neural network sends the position calculation to the microcontroller board over RS232 communication. The microcontroller transmits the steering command to the servo through a three-pin header.

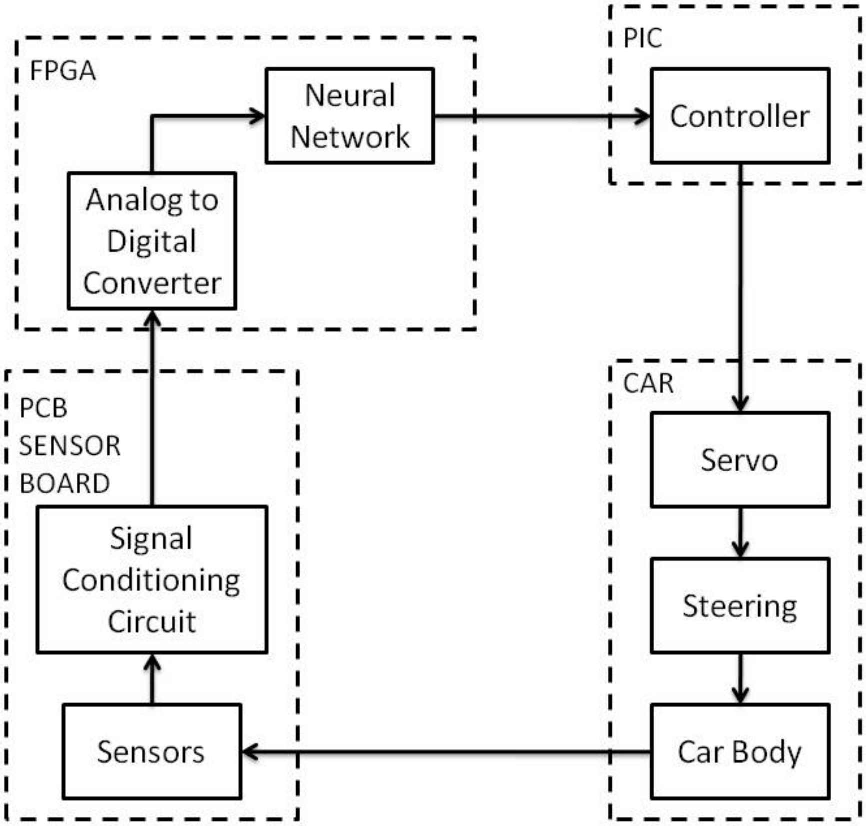


Figure 6-18: Main Block Diagram

Figure 6-19 shows the assembled car.

Assembled Car with Boards

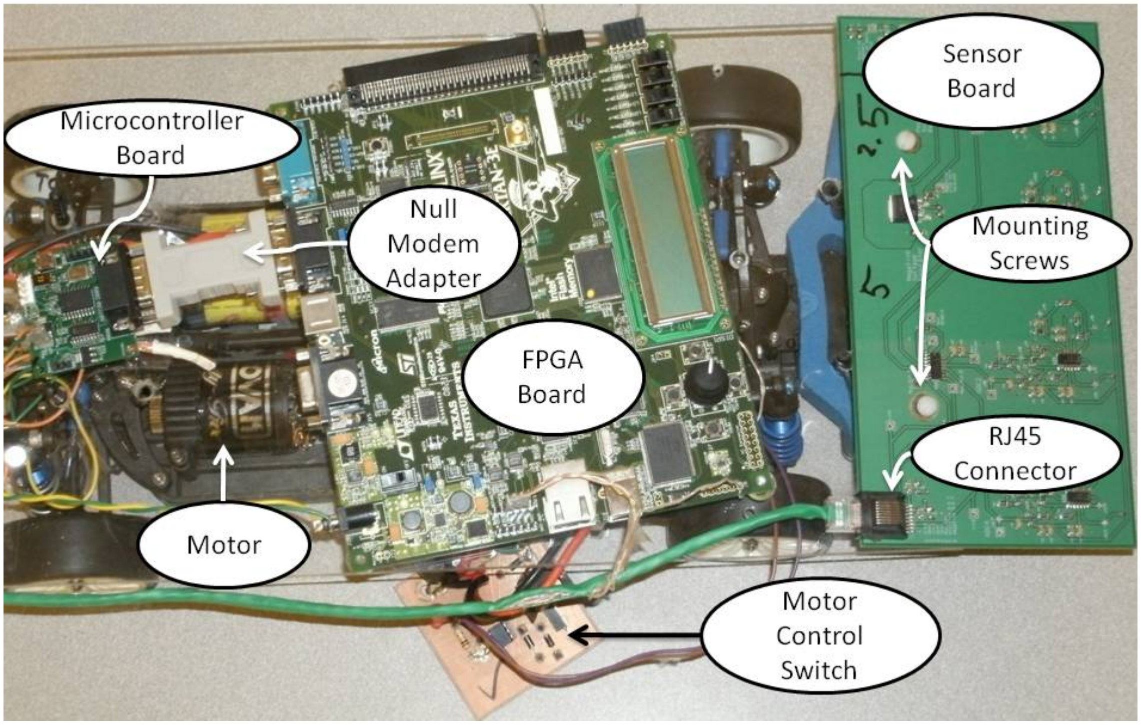


Figure 6-19: Assembled Car with Boards

The sensor board is discussed in detail in chapter 2. Figure 6-20 shows the connections for the sensor board. The sensor board is connected to the board-mounting fixture by screws. Signals and power for the sensor board go through an RJ45 eight place eight connect header. Pin 1 carries the first output of the multiplexer, which goes to the first channel of the A/D converter. Pin 3 carries the second output of the multiplexer, which goes to the second channel of the A/D converter. Pins 2, 4, and 5 are connected to the sensor board ground. Pin 6 carries the multiplexer select pin from the FPGA board to the multiplexer. Pin 7 connects to positive power and pin 8 connects to negative power.

Sensor Board Connections

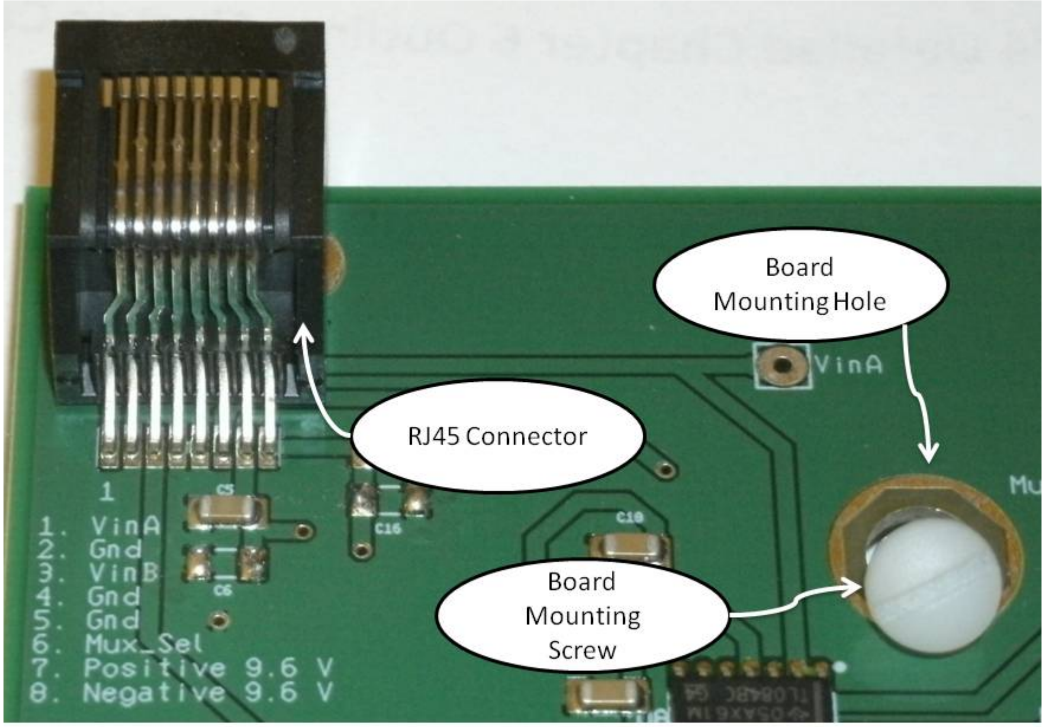


Figure 6-20: Sensor Board Connections

Figure 6-21 shows the FPGA board connections. The first connection is power and ground. This is a 2.1mm plug connection and comes from the board-mounting fixture. The next connection is a three-pin header that carries the two outputs from the multiplexer as well as a connection for ground. The third connection is a single pin header that the FPGA board uses to transmit the multiplexer selection signal. The fourth connection is the DSUB 9 connector that carries the RS232 communication to the microcontroller board. This communication is sent through a null modem adapter. The FPGA board is fastened to the board mounting fixture with a hook and pile fastener underneath the board.

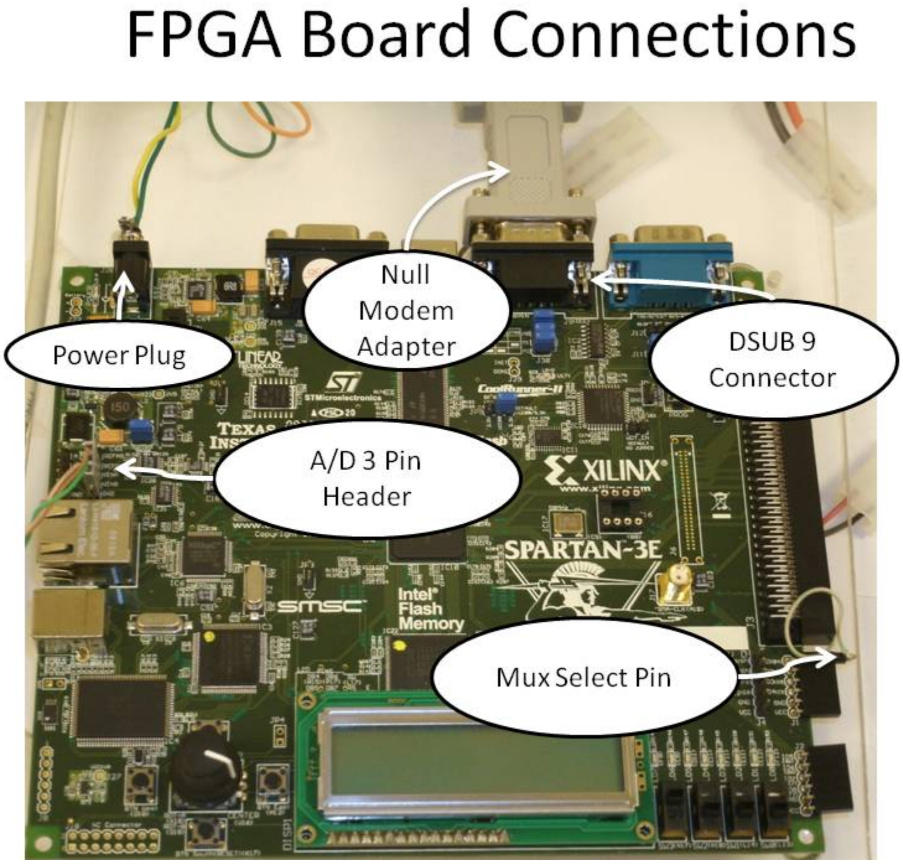

FPGA Board Connections

Figure 6-21: FPGA Board Connections

Figure 6-22 shows the four microcontroller board connections. It receives the RS232 position calculation through the null modem adapter. Power and ground come from a two-pin header from the board-mounting fixture. The microcontroller board controls the servo with a three-pin header. Pin 1 is ground, pin 2 carries five Volts, and pin 3 carries the steering servo signal pulse. The microcontroller communicates with the FPGA board using RS232 communication, which it transmits with the DSUB9 connection through a null modem adapter. The microcontroller sends the motor control signal to the motor control switch board using a four-pin header. Pin 1 is connected to ground and pin 3 carries the motor control signal pulse. Pins 2 and 4 are not connected. The microcontroller board is physically held in place by screws connecting the microcontroller board to the null modem adapter and screws connecting the null modem adapter to the FPGA board.

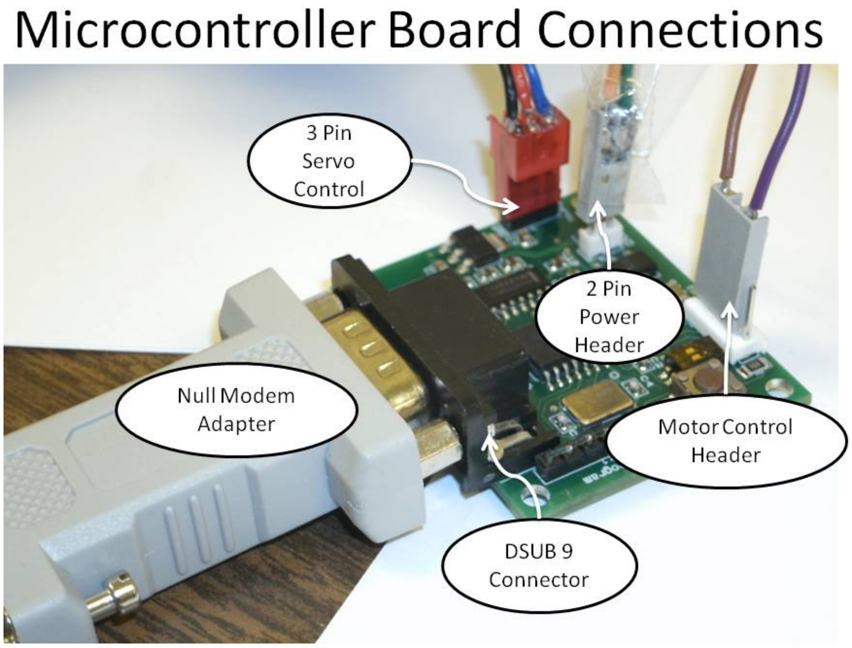


Figure 6-22: Microcontroller Board Connections

6.4 Summary

This chapter has described the hardware components of the electric car, which forms the platform for the neural network smart sensor demonstration system.

CHAPTER VII

SUMMER SOFT SENSOR ACADEMY

This chapter discuses the summer soft sensor academy. This summer academy for local high school students provides an opportunity for them to learn about science and engineering. Various activities demonstrate basic engineering concepts.

7.1 Overview and Academy Objectives

The summer soft sensor academy has two main objectives. The primary objective is to introduce students to the idea of smart sensors using neural networks. The students are shown that neural networks can be implemented in digital logic on a FPGA. The secondary objective is to show that science and mathematics are important to engineering.

7.2 Activities

The summer academy is divided into five training modules, each covering different concepts from engineering and science. The first module focuses on electromagnetic sensors and different ways to visualize a magnetic field. The second module covers data fitting and using data fitting software. The fourth module explains the basics of digital circuits. The third and fifth modules introduce the basic concepts of neural networks and how to train them.

7.2.1 Electromagnetic Sensors

During the electromagnetic sensors module, the students have a chance to work with different electromagnetic sensors and magnetic fields. The three sensors the students use are a compass, a coil of wire and a 33 milli-Henry inductor. The three magnetic field sources are a bar magnet, a steady moving current through a wire powered by a voltage source and the third source is a time varying magnetic field produced by a function generator driving a sine wave through a wire. In the first block, they use a compass to map the magnetic field lines coming from a permanent bar magnet. In the second block, they use a simple piece of wire rolled into a coil. They move the bar magnet through the wire coil. The time varying magnetic field induces a current in the coil, which they observe with an analog voltage meter and digital oscilloscope. In the third block, the students use a power supply to produce current flow inside a wire to generate a magnetic field. A compass next to the wire will have a change in needle directions as they toggled the power supply on and off. The students complete three experiments that use a function generator to provide a time varying current flow resulting in a time-varying magnetic field. In the fourth block, they use the coil of wire to observe that magnetic fields have directionality. They observe this by changing the orientation of their loop with respect to the source wire. The next experiment uses the coil orientation with the strongest response to observe the effects of distance between the coil and source wire. These experiments are repeated using an inductor to replace the hand coiled wire. Next, the inductor is used in data collection. The students put a block of wood with a ruler taped on top over the top of the wire. They use the ruler to measure horizontal distance between the center of the inductor and center of the source wire. They record the average peak-to-peak voltage across the inductor measured by the oscilloscope. With each voltage reading, they also record the inductor horizontal distance away from the wire.

7.2.2 Data Fitting

The second module focuses on data fitting activities. The first activity is basic function plotting. The next activity is fitting a line to data. The students then learn how to use software to fit a function to the data. Next, the students make a plot of their data from the electromagnetic sensors module and model the data with a curve. The first step of this process used distance as the independent variable and voltage as the dependent variable. The students are then asked to invert the relationship between voltage and distance. Finally, the students are asked to predict optimal sensor spacing based on their collected data and models.

7.2.3 Neural Networks

The third module introduces the students to the application of neural networks to data fitting. First, the students learn about the basic structure of neurons and similarities between the biological and artificial neurons. The students have a chance to work with single layer and multilayer networks to observe how changing network parameters drives the network output. Students then adjust the network parameters by hand to fit the network output to the data collected in the first module. Finally, the students learn how to use software to train the neural network to fit the data.

7.2.4 Digital Circuits

The fourth module provides an opportunity for students to learn about digital circuits. The module begins with an introduction to binary numbers. Next, the students learn the basics of Boolean algebra. The students apply this knowledge to design combinational circuits. The students are then given a basic circuit design and required hardware to assemble the circuit and test its functionality. The module concludes with an introduction to sequential logic basics, such as how a flip-flop works and how to read a timing diagram.

7.2.5 Training the Brain

During this module, the students train a neural network. The process begins with data collection. The students record car position and sensor readings at several locations. They use this data to train the neural network with the training software. The network learns to produce the car position from the four sensor readings. Finally, the students validate the network performance by comparing the trained network response to the collected data. The students also use interpolated data to check for over fitting, as discussed in chapter four.

7.2.6 Race Day

On the last day of the summer academy, the students have the opportunity to observe a performance evaluation of their smart sensor car systems. Performance is measured with three tests. The first test is a static measurement. A ruler is taped over the wire in the same configuration as data collection. The car is moved to several positions along the ruler. At each position, the error between the physical car location and the position calculated by the smart sensor car is recorded (as shown on the LCD screen of the FPGA board). The total squared error for each car is calculated and the car with the lowest total error wins the challenge. The second test is following a line. For this test, a ten-foot long piece of paper is placed over a straight run of the wire track. The caps are removed from the markers on the front of the cars to allow the cars to mark their paths. The students run the cars down the wire. The car that stays closest to the centerline wins the challenge. The third and final test is time to complete a lap. A simple wire track is laid out on the floor. Each team records three lap attempts. The shortest lap time of the three is kept for each team. The team with the shortest single lap time wins the challenge. This module focuses on performance evaluation. It displays how different tests show different performance characteristics

7.3 Summary

The summer soft sensor academy provides local high school students a chance to learn about engineering by working with soft sensors. The academy has modules focusing on five separate engineering concepts. The first module is about electromagnetic sensors and different ways to visualize a magnetic field. The second module is about data fitting and using data fitting software. The fourth module covers the basics of digital circuits. The third and fifth modules cover the basic concepts of neural networks and how to train them. The academy ended with a competition comparing system performance.

CHAPTER VIII

SUMMARY AND FUTURE WORK

This chapter is a summary of the thesis and project. The chapter begins with chapter summaries, followed by a description of the author's contributions to the project. The chapter ends with a discussion of some possible future work with the project components.

8.1 Summary of Thesis

The first chapter is an introduction to the project. The chapter begins with the system overview and description of the main block diagram. The chapter provides a brief project history and describes the current state of the project. The chapter ends with an outline of the rest of the thesis.

Chapter two covers the sensors and signal conditioning circuit. It describes the magnetic field and sensors to detect it. It also describes the signal conditioning circuit, defining the circuit components, how they were tested and the fabrication of the printed circuit board. The sensors and signal conditioning circuit provide a path for information about the magnetic field to travel to the A/D converter. The sensors and components for the signal conditioning board are common and readily available from local vendors and can be used with prototyping boards. Inductors are a good choice for sensing a magnetic field because they are readily available, have been used for this application before, and their output can be measured as a voltage. Proto-board testing reduced the likelihood of problems during printed circuit board development. Printed circuit boards are the method of choice for the final implementation of the signal conditioning circuit.

Chapter three focuses on the A/D converter. It describes the on-board hardware and off-board multiplexer operation, control of the A/D conversion and how the A/D conversion process was tested. The A/D conversion provides the neural network with sensor readings by converting the input voltages to digital numbers. This process requires hardware and software. Most of the hardware came with the FPGA board. Intelligence and a process to control the hardware enables the A/D FSM to do its job which contributes to the smart sensor car being able to follow a wire.

Chapter four is about the artificial neural network. It begins by providing basic information about the artificial neural network implemented on the smart sensor car. Next, It describes the FPGA implementation of the neural network, showing timing diagrams for the process. The neural network maps four sensor readings into a car position calculation. The neural network changes the outputs of the A/D converter into the acceptable input for the PD controller. The neural network calculates the car position serially, doing one arithmetic step at a time. Training helps ensure the neural network calculates the car position correctly. The master FSM has some modifications that allow it to interface with a computer to collect training data. In order to ensure the neural network, master FSM, A/D FSM and all other code components work together properly, software support equipment is required.

Chapter five describes the PD controller. The chapter describes how the controller is implemented and explains how the controller works. The chapter also describes the modeling process used to design controller parameters. The PD controller takes a distance measurement from the neural network and calculates the steering angle to keep the car as close to the track as possible. The controller executive software controls the process flow. The control algorithm runs on a PIC microcontroller, which requires its own board.

Chapter six describes the electric car. The electric car is made with many standard components. These components work together to provide motion around the track. The components share signals through various connections. The board-mounting fixture holds the components together.

Chapter seven describes the summer soft sensor academy. The summer soft sensor academy provides local high school students a chance to learn about engineering by working with soft sensors. The academy has modules focusing on five separate engineering concepts. The first module is about electromagnetic sensors and different ways to visualize a magnetic field. The second module is about data fitting and using data fitting software. The fourth module covers the basics of digital circuits. The third and fifth modules cover the basic concepts of neural networks and how to train them. The academy ended with a competition comparing system performance.

8.2 Author's Contributions to Project

The author's contributions began during the Fall of 2009 by assisting Amanuel Assefa with the Xilinx software to change the FPGA code to read in four sensors and control the multiplexer. This included mapping the states of the Master FSM and A/D FSM. The author learned Chipscope and used it to monitor code execution on the FPGA to verify the proper operation. The author also assisted with regeneration of cores using the CoreGen software modules.

In the Spring of 2010 the author began work on the rest of the smart sensor car. This began with collection of basic characteristics about how the inductor behaves with the magnetic field generated by the function generator. This provided an understanding of how to set up the different test fixtures. The next step was the introduction of op amp circuits. This allowed the development and tuning of a circuit to condition the output of a single sensor using a bread board. The experiments used different circuits using different components in different configurations. The author then developed a printed circuit board design and had it manufactured. The author populated the professionally fabricated board and tested it. The author integrated the FPGA board and sensor board together and verified the timing for the multiplexer. After the sensor board and FPGA were able to generate consistent results from bench power, the author developed a system to power the boards from batteries, which is the desired configuration to support the smart sensor car. The initial system suffered from noise issues on the power transmission lines. This was resolved by decoupling and isolation. The next component was the motor control switch. The author designed a circuit using a Darlington pair of BJTs driven by an opticoupler. After the individual components were ready, the author designed and fabricated the board mounting fixtures. Once all the system components were working together, the author modeled the smart sensor car to obtain PD controller gains. The author worked with Dr. Hagan to modify the microcontroller code to match the car steering and motor performance. A total of five cars were produced for the class. During the summer academy, the author assisted students with modules and equipment in the lab. The author contributed to some of the writing for the summer academy documents. The author instructed students about how to use the smart sensor cars and assisted them as needed. After the summer academy, the author documented all system components.

8.3 Future Work

The smart sensor car provides a flexible platform for future work with FPGAs, neural networks, control systems and other physical implementations. Future projects could include work with the sensor board, A/D converter, neural network, controller and electric car.

The sensor board provides a means to gather information about the outside world and provide feedback in signals that can be converted by A/D converters. The sensor board currently uses inductors as antennas. There may be more effective or more useful antenna designs than the basic coil. Another possible project would be to alter the number or placement of the sensors. This could be as simple as doubling the number of sensors and reducing the sensor spacing from two inches to one inch. The project could investigate using two rows of sensors to calculate the angle between the car and the wire. Another option would be to change the signal conditioning circuit. This project could involve experiments with different components that would allow manual or automatic tuning of individual sensors to provide a more consistent response. The project could also explore ways to build a conditioning circuit that could tune itself to adjust to automatically to changes in the track.

The A/D converter translates information from the outside world in the form of voltages into digital numbers the components on the FPGA can use. One future project could be to eliminate current redundant processes from the A/D FSM. Another project could add a filter to the A/D readings. Because the Spartan 3E starter kit has unused serial peripheral interface ports, it would be possible to incorporate more external serial A/D converters. A more complicated project could build a FPGA board with parallel A/D converters.

The current serial neural network design has been used in two projects and could be useful in many more. One possible change to the neural network code would be to change the implementation of the network. The current code is build around a single neuron with a single input. The three other possibilities are a single neuron with multiple inputs, multiple single input neurons and multiple neurons, each with multiple inputs. Each of these structures could provide the same functionality as the current structure. Another option that would maintain the functionality would be to implement different types of neural networks, such as a radial basis network. An added function to the neural network could be a filter on the calculation. Another option would be to incorporate the calculation of the angle between the car and wire. The neural network could also be used to calculate the steering angle or even servo command. This would reduce the need for extra components and calculations.

The controller takes the position calculation from the neural network and converts it into commands for the car. The current PD controller could be implemented on the FPGA. Once on the FPGA, more elaborate controllers could be experimented with. Another project could be to use the existing controller for steering while incorporating speed control. A follow on project to that could be to incorporate breaking. A different project could allow the controller to identify when the car is too far away from the track to sense the wire. The second part of this project could be to introduce a way for the car to begin a search for the lost wire.

The electric car takes inputs from the controller and provides motion to the system. The forward velocity source provides many opportunities for projects. The current motor control switch could be replaced by a half H bridge to allow breaking, or a full H bridge to allow the car to change directions. Another project could explore ways to increase efficiency of the system by using different chassis with different drive configurations. Also, different power sources such as fuel cells or solar panels could be investigated. A similar platform could also utilize an internal combustion motor allowing different experiments with bio-fuels. A different experiment could be to make the fixed wheels the front of the car and the articulated wheels the rear of the car, similar to most industrial forklifts. Another experiment could be to use a chassis with front and rear wheel articulation. A related project could be to explore a chassis that has fixed front and rear wheels and center articulation or the chassis. The smart sensor position measurement and steering control could also be used with different platforms. The system could be transferred to a tracked platform, a two or three wheel platform, or even a walking robot.

The current system is capable of making a circle around the track. One simple project could be to add a battery powered web cam to the car to monitor the system as it goes around the track. A follow on project could be to allow user feedback to shut off the car if they see it go off the track. A different project could be to broadcast other information such as sensor readings, position calculations and steering commands.

REFERENCES

[1] M. T. Hagan, H. B. Demuth, M. H. Beale, *Neural Network Design*. Boston, 1996.

[2] C. Latino, M. Moreno-Armendariz, "Realizing general MLP networks with minimal FPGA resources," in *2009 International Joint Conference on Neural Networks, IJCNN 2009, June 14, 2009 - June 19, 2009,* Atlanta, GA, United states, 2009, pp. 1722-1729.

[3] I. E. Paromtchik and C. Laugier, "Motion generation and control for parking an autonomous vehicle," vol. 4, ed, 1996, pp. 3122 vol.4-3122 vol.4.

[4] J. P. Laumond, P. E. Jacobs, M. Taix, R. M. Murray, "A motion planner for nonholonomic mobile robots," vol. 10, ed, 1994, pp. 577-93.

APPPENDICES

Appendix A: Code for Master FSM to Calculate a Position

```
begin
if rising_edge(CLOCK) then
      case state is

      when IDLE =>
            Enable_Network <= '0';
            nd_fixed2float1 <= '0';
            nd_fixed2float2 <= '0';
            nd_fixed2float3 <= '0';
            nd_fixed2float4 <= '0';
            NN_data (15 downto 0) <= (others => '0');
            rs232_receiver_rst <= '0';
            next_state <= WAIT_RECEIVE;

      when WAIT_RECEIVE => -- wait for RS232 data
            led(3 downto 0) <= "0001";
            if (rs232_receiver_stb = '1') then -- data done
                  rs232_receiver_rst <= '1';
                  train_mode <= '0';
                  next_state <= START_ADC;
            else
                  next_state <= WAIT_RECEIVE; -- receiving data
            end if;

      when START_ADC =>
            ce_amp <= '1'; --active high
            start_conv <= '1';
            next_state <= ADC;

      when ADC =>
            if ADC_DONE = '1' then
                  next_state <= ADC2FIXED;
                  ce_amp <= '0'; --active low
                  start_conv <= '0';
            else
                  next_state <= ADC;
            end if;
```

```
when ADC2FIXED =>
      nd_fixed2float1 <= '1';
      next_state <= FIXED2FLOAT1;

when FIXED2FLOAT1 =>     -- fixed to float 1
      nd_fixed2float1 <= '0';
      if (rdy_fixed2float1 = '1') then
            P1 <= fixed2float_result1;
            nd_fixed2float2 <= '1';
            next_state <= FIXED2FLOAT2;
      else
            next_state <= FIXED2FLOAT1;
      end if;

when FIXED2FLOAT2 =>     -- fixed to float 2
      nd_fixed2float2 <= '0';
      if (rdy_fixed2float2 = '1') then
            P2 <= fixed2float_result2;
            nd_fixed2float3 <= '1';
            next_state <= FIXED2FLOAT3;
      else
            next_state <= FIXED2FLOAT2;
      end if;

when FIXED2FLOAT3 =>     -- fixed to float 3
      nd_fixed2float3 <= '0';
      if (rdy_fixed2float3 = '1') then
            P3 <= fixed2float_result3;
            nd_fixed2float4 <= '1';
            next_state <= FIXED2FLOAT4;
      else
            next_state <= FIXED2FLOAT3;
      end if;

when FIXED2FLOAT4 =>     -- fixed to float 4
      nd_fixed2float4 <= '0';
      if (rdy_fixed2float4 = '1') then
            P4 <= fixed2float_result4;
            next_state <= WRITE_ADC_DATA1;
      else
            next_state <= FIXED2FLOAT4;
      end if;

when WRITE_ADC_DATA1 => -- 1 clock cycle every time
      Enable_network <= '0';
      ce_RAM <= '1';
      RAM_DATA <= P1;
      RAM_ADDR <= "0000"; -- P1 address
      next_state <= WRITE_ADC_DATA2;
```

```
when WRITE_ADC_DATA2 => -- 1 clock cycle every time
     RAM_DATA <= P2;
     RAM_ADDR <= "0001"; -- P2 address
     next_state <= WRITE_ADC_DATA3;

when WRITE_ADC_DATA3 => -- 1 clock cycle every time
     RAM_DATA <= P3;
     RAM_ADDR <= "0010"; -- P3 address
     next_state <= WRITE_ADC_DATA4;

when WRITE_ADC_DATA4 => -- 1 clock cycle every time
     RAM_DATA <= P4;
     RAM_ADDR <= "0011"; -- P4 address
     next_state <= WRITE_ADC_DONE;

when WRITE_ADC_DONE =>
     ce_RAM <= '0';
     next_state <= START_NN;

when START_NN =>
     Enable_Network <= '1';
     next_state <= NN;
     counter <= (others => '0');

when NN =>
     if rdy_NN = '1' then
          next_state <= FLOAT2FIXED;
          NN_Data <= N;
          Enable_Network <= '0';
          nd_float2fixed <= '1';
     else
          next_state <= NN; -- NN
     end if;

when FLOAT2FIXED =>
     if (rdy_float2fixed = '1') then
          nd_float2fixed <= '0';
          next_state <= START_RS232_HI;
     else
          nd_float2fixed <= '1';
          next_state <= FLOAT2FIXED;
     end if;

when START_RS232_HI =>
     counter <= (others => '0');
     rs232_sender_stb <= '1'; -- start pulse on
     rs232_sender_dat <= "000"&NN_fixed (12 downto 8);
     next_state <= RS232_HI;
```

```
           when RS232_HI =>
                counter <= counter + 1;
                if rs232_sender_ack = '1' then
                      rs232_sender_stb <= '0'; --start pulse off
                elsif counter (19 downto 0) =
CONV_STD_LOGIC_VECTOR (system_speed/baudrate*10, 20) then
                      next_state <= START_RS232_LO;
                else
                      next_state <= RS232_HI;
                end if;

           when START_RS232_LO =>
                counter <= (others => '0');
                rs232_sender_stb <= '1'; -- start pulse on
                rs232_sender_dat <=  NN_fixed (7 downto 0);
                display_data (12downto0)<=NN_fixed(12downto0);
                next_state <= RS232_LO;

           when RS232_LO =>
                counter <= counter + 1;
                if rs232_sender_ack = '1' then
                      rs232_sender_stb <= '0'; --start pulse off
                elsif counter (19 downto 0) =
CONV_STD_LOGIC_VECTOR (system_speed/baudrate*10, 20) then
                      next_state <= IDLE;
                else
                      next_state <= RS232_LO;
                end if;

           end case; -- state
     end if; -- clock

     end process sensor_state_machine;

     ADC1_fixed <= ADC1(13) & ADC1(13 downto 2);
     ADC2_fixed <= ADC2(13) & ADC2(13 downto 2);
     ADC3_fixed <= ADC3(13) & ADC3(13 downto 2);
     ADC4_fixed <= ADC4(13) & ADC4(13 downto 2);

     state <= next_state;

     RS232_DCE_TXD    <= RS232_TX_out;
     SPI_AMP_SHDN     <= '0';
     DAC_CS                <= '1'; -- could not find in chipscope
     SPI_SS_B              <= '1';

     gain <= "00010001";

     end Behavioral;
```

```
begin
      when IDLE =>
            MUX_SELECT <= '0';
            AMP_CS <= '1';
            counter <=0;
            if ce_amp ='1' then
                  next_state <= START;
            else
                  next_state <= IDLE;
            end if;

      when START =>
            AMP_CS <= '0'; --turn amp on
            next_state <= START2;
            index1 <= 7; -- 8 bit value

      when START2 =>
            MOSI <= gain(index1);
            next_state <= HI;
            bit_count <= 0;

      when HI =>
            SCK <= '1';
            counter <= counter +1;
            if counter = 2 then
                  next_state <= HI_DUMMY;
            else
                  next_state <= HI;
            end if;

      when HI_DUMMY =>
            counter <=0;
            bit_count <= bit_count + 1;
            index1 <= index1-1;
            next_state <= LO;

      when LO =>
            SCK <= '0';
            counter <= counter +1;
            if counter = 2 then
                  next_state <= LO_DUMMY;
            else
                  next_state <= LO;
            end if;
```

```
when LO_DUMMY =>
      counter <=0;
      if bit_count = 8 then
            next_state <= FINISH;
      else
            MOSI <= gain(index1);
            next_state <= HI;
      end if;

when FINISH =>
      next_state <= IDLE_AD;
      AMP_CS <='1';
      SCK <= '0';
      MOSI <= '0';

when IDLE_AD =>
      if start_conv ='1' then
            next_state <= START_AD;
      else
            next_state <= IDLE_AD;
      end if;

when FINISH =>
      next_state <= IDLE_AD;
      AMP_CS <='1';
      SCK <= '0';
      MOSI <= '0';

when IDLE_AD =>
      if start_conv ='1' then
            next_state <= START_AD;
      else
            next_state <= IDLE_AD;
      end if;
      CONV <= '0';
      prime <= '1';
      MUX_SELECT <= '0';

when START_AD =>
      SCK <= '0';
      CONV <= '1';
      counter <= -1;
      index1 <= 13; -- 14 bit value
      index2 <= 13; -- 14 bit value
      next_state <= HI_AD;

when HI_AD =>
      SCK <= '1';
      CONV <= '0';
      counter <= counter +1;
      next_state <= LO_AD;
```

```
when LO_AD =>
      SCK <= '0';
      if prime = '0'  and mux_sel = '0' then
            if(counter > 2 and counter < 17) then
                  ADC1(index1)  <= SPI_MISO;
                  index1 <= index1 -1;
            elsif(counter > 18 and counter < 33) then
                  ADC2(index2)  <= SPI_MISO;
                  index2 <= index2 -1;
            end if;

            if counter = 34 then -- DONE
                  next_state <= FINISH_AD;
            else
                  next_state <= HI_AD;
            end if;

      elsif prime = '1' and mux_sel = '0' then
            if counter = 34 then -- done priming
                  prime <= '0';
                  next_state <= START_AD;
            else
                  next_state <= HI_AD;
            end if;

      elsif prime = '0' and mux_sel = '1' then
            if(counter > 2 and counter < 17) then
                  ADC3(index1)  <= SPI_MISO;
                  index1 <= index1 -1;
            elsif(counter > 18 and counter < 33) then
                        ADC4(index2)  <= SPI_MISO;
            index2 <= index2 -1;
            end if;

            if counter = 34 then -- DONE
                  ADC_DONE <= '1';
                  next_state <= FINISH_AD;
            else
                  next_state <= HI_AD;
            end if;

      elsif prime = '1' and mux_sel = '1' then
            if counter = 34 then -- done priming
                  prime <= '0';
                  next_state <= START_AD;
            else
                  next_state <= HI_AD;
            end if;
      end if;
      when FINISH_AD =>
            counter <= 0;
            MUX_counter <= 0;
            SCK <= '0';
            CONV <= '0';
            if MUX_SELECT = '0' then
                  prime <= '1';
                  MUX_SELECT <='1';
                  next_state <= MUX_SWITCH_1;
            else
                  next_state <= IDLE;
            end if;

      when MUX_SWITCH_1 =>
            MUX_counter <= MUX_counter + 1;
            next_state <= MUX_SWITCH_2;

      when MUX_SWITCH_2 =>
            if MUX_counter = 100 then
                  next_state <= START_AD;
            else
                  next_state <= MUX_SWITCH_1;
            end if;

      when others =>
            MUX_SELECT <= '0';
            mux_sel <= '0';
            SCK <= '0';
            CONV <= '0';
            AMP_CS <= '1';
            MOSI <='0';
            next_state <= IDLE;
      end case;

end process;
```

Appendix C: Code for Neural Network FSM

```
process (CLOCK)
begin

   if CLOCK = '1' and CLOCK'event then
      if WRITE_INPUT = '0' and Enable_Network = '0' then
         a_addr_tmp           <= "0011";
         p_addr_tmp                 <= "0000";
         w_addr_tmp                 <= "00000";--
         b_addr_tmp                 <= "0000";
         shift_addr_tmp <= "0000";
         input_base_tmp <= "0000";
         input_num_tmp        <= "0000";
         neuron_num_tmp <= "0000";
         R_S1_flag                  <= "00";
         cuenta_tmp                 <= "00";
         b_0                        <= "0000000000000000";
         r                          <= "0011";
         s1                         <= "0000";
         q_tmp                      <= '0';
         rdyQ_tmp                   <= '0';
         flag_tmp                   <= '1';
         write_ena                  <= '0';
         nd_tmp                     <= '0';
         start_tmp                  <= '0';
         start_flag_tmp <= '0';
         completion_flag_tmp  <= '1';
         enable_tmp           <= '0';
         init_flg_tmp         <= '1';
         ram_b_input          <= "0000000000000000";
         write_enb                  <= '0';

      elsif WRITE_INPUT = '1' and Enable_Network = '0' then

         ram_b_input          <= INPUT_DATA;
         p_addr_tmp           <= INPUT_ADDR;
         write_enb                  <= '1';
         rdyNN_tmp                  <= '0';

      elsif WRITE_INPUT = '0'
      -and Enable_Network = '1' and S1 = "1111" then
         rdyNN_tmp                  <= '1';
         a_addr_tmp           <= "0011";
         p_addr_tmp                 <= "0000";
         w_addr_tmp                 <= "00000
         b_addr_tmp                 <= "0000";
         shift_addr_tmp <= "0000";
         input_base_tmp <= "0000";
         input_num_tmp        <= "0000";
         neuron_num_tmp <= "0000";
         R_S1_flag                  <= "00";
         cuenta_tmp                 <= "00";
         b_0                        <= "0000000000000000";
         r                          <= "0011";
         s1                         <= "0000";
         q_tmp                      <= '0';
         rdyQ_tmp                   <= '0';
         flag_tmp                   <= '1';
         write_ena                  <= '0';
         nd_tmp                     <= '0';
         start_tmp                  <= '0';
         start_flag_tmp       <= '0';
         completion_flag_tmp  <= '1';
         enable_tmp           <= '0';
         init_flg_tmp         <= '1';
         ram_b_input          <= "0000000000000000";
         write_enb                  <= '0';

      elsif WRITE_INPUT = '0'
      -and Enable_Network = '1' and S1 /= "1111" then
         ram_b_input    <= zero;
         write_enb            <= '0';
         rdyNN_tmp            <= '0';

-- The neural network should only run when told to, and must
-- tell the controling system it is done by setting rdy_nn to '1'
-- When we are at the end of the shiftrom (1111),
-- set rdy_nn to '1'
--if (r = "1111") then
--   rdyNN_tmp <= '1';
--end if;

-- When an addition is completed, update b_0.
-- We don't want to do this during an addition, because
-- intermediate values will be wrong.

         if (rdyAD_tmp = '1' ) then
            b_0               <= b_tmp;
         end if;

-- Select either bias or partial n to be added to w times p
-- When input_num is zero, bias is added.
-- When input_num is not zero, then partial n is added.

         if input_num_tmp = "0000" then
            if flag_tmp = '1' then
               q_tmp          <= '0';
               rdyQ_tmp       <= '1';
               flag_tmp       <= '0';
            end if;

            if (rdyQ_tmp = '1') then
               b_0                  <= b_tmp;
               rdyQ_tmp       <= '0';
               q_tmp                <= '1';
               q2_tmp         <= '1';
            end if;

            if (q2_tmp = '1') then
               write_ena      <= '0';
               q2_tmp         <= '0';
            end if;
         else
```

```
            flag_tmp          <= '1';
         end if;

-- When R_S1_flag is 00, then R is read from the shift_rom

         if R_S1_flag = "00" then
            if (nd_tmp = '0' and rdySH_tmp = '0') then
               nd_tmp         <= '1';
            else
               if (rdySH_tmp = '1' and nd_tmp = '1') then
                  nd_tmp <='0';
                  r           <= memory_tmp;
                  R_S1_flag <= R_S1_flag + "01";
               end if;
            end if;
         end if;

-- When R_S1_flag is 01, then S1 is read from the shift_rom

         if R_S1_flag = "01" then
            if (nd_tmp = '0' and rdySH_tmp = '0') then
               shift_addr_tmp       <= shift_addr_tmp + "0001";
               nd_tmp               <= '1';
            else
               if (rdySH_tmp = '1' and nd_tmp = '1') then
                  nd_tmp            <='0';
                  s1                <= memory_tmp;
                  R_S1_flag         <= "11";
               end if;
            end if;
         end if;

-- When R_S1_flag is 11, then we continue to compute partial
-- sums until all inputs have been applied, and then we compute
-- neuron outputs until all of the neurons in the current layer
-- have been computed. Then we set R_S1_flag back to 00 to
-- restart.

         if R_S1_flag = "11" then

-- When addition is complete, check that all other events are
-- completed - end of neuron, end of layer.  completion_flag
-- will be 1 until all events are complete, then it is set to 0.

            if (rdyAD_tmp= '1') then
               completion_flag_tmp <= '1';
               enable_tmp <= '0';
               start_tmp <= '1';
            end if;

-- Wait two clocks to be sure that the addresses have settled
-- before starting the neuron calculation.

            if start_flag_tmp = '1' then
               cuenta_tmp           <= cuenta_tmp + 1;
               if cuenta_tmp = "10" then
                  start_tmp         <= '0';
                  start_flag_tmp    <= '0';
                  cuenta_tmp  <= "00";
               end if;
            end if;

--completion_flag_tmp will be 1, if an addition has been
-- completed, or if we are on the initial pass.

            if (completion_flag_tmp = '1') then

-- The following if is for the first time. It is only done once.

               if init_flg_tmp = '1' then
                  init_flg_tmp            <= '0';
                  start_tmp               <= '1';
                  enable_tmp        <= '1';
                  start_flag_tmp          <= '1';
                  completion_flag_tmp     <= '0';
               else     --init_flg_tmp

-- When the input # eq r, start the tansig conversion.
-- The conversion starts when a_eq_b_temp is 1.

                  if input_num_tmp = r then
                        a_eq_b_tmp <= '1';
                  end if;

--We wait until rdyTF is 1, if we are at the last input.
--If we are not at the last input, we update the addresses
--and start the next input calculation.

                  if ((rdyTF_tmp = '1')
                  -or (not(input_num_tmp = r))) then
                        completion_flag_tmp     <= '0';
                        enable_tmp              <= '1';
                        w_addr_tmp        <= w_addr_tmp + 1;
                        start_flag_tmp          <= '1';
                        start_tmp               <= '1';

--If we are at the last input, start a new neuron

                        if (input_num_tmp = r) then
                              a_eq_b_tmp  <= '0';
                              write_ena         <= '1';
                              input_num_tmp     <= "0000";
                              a_addr_tmp  <= a_addr_tmp + 1;
                              b_addr_tmp  <= b_addr_tmp + 1;

--If we are at the last neuron, start a new layer.

                              if neuron_num_tmp = s1 then
                        input_base_tmp <= input_base_tmp+r+1;
                                    neuron_num_tmp <= "0000";
                                    R_S1_flag       <= "00";
                        p_addr_tmp     <= input_base_tmp+r+1;
                              else
                        neuron_num_tmp <= neuron_num_tmp + 1;
```

```
                              p_addr_tmp     <= input_base_tmp;
                              end if; --neuron_num_tmp

--If we are not at the last input, update the input #.

                        else ----input_num_tmp
                        input_num_tmp     <= input_num_tmp + 1;
                              p_addr_tmp  <= input_num_tmp +
                                             input_base_tmp +1;
                        end if; --input_num_tmp
                  end if; --rdyTF_tmp
               end if; --init_flg_tmp
            end if; --completion_flag_tmp
         end if; --R_S1_flag = '11'
      end if; -- write/enable_network
   end if; --CLOCK
end process;

not_q_tmp      <= NOT q_tmp;
N              <= n_tmp;
A              <= a_tmp;
P              <= p_tmp;
B              <= b_tmp;
B_OUT          <= b_0;
W              <= w_tmp;
OUT_B          <= out_b_tmp;
B_ADDR         <= a_addr_tmp;
SHIFT_ADDR     <= p_addr_tmp;       -- NOTE SWITCH
INPUT_BASE  <= input_base_tmp;
INPUT_NUM      <= input_num_tmp;
NEURON_NUM  <= neuron_num_tmp;
Q              <= q_tmp;
RDY_Q          <= rdyQ_tmp;
RDY_AD         <= rdyAD_tmp;
RDY_SH         <= init_flg_tmp;
RDY_TF         <= rdyTF_tmp;
A_EQ_B         <= a_eq_b_tmp;
FLAG           <= write_ena;
R_OUT          <= r;
S1_OUT         <= s1;
R_S1_FLG       <= R_S1_flag;
MEMORY         <= memory_tmp;
ND             <= start_flag_tmp;
WRITE_A        <= write_ena;
START          <= start_tmp;
CUENTA         <= cuenta_tmp;
COMPLETION  <= completion_flag_tmp;
ENABLE         <= enable_tmp;
RDY_NN         <= rdyNN_tmp;

end Behavioral;
```

```
      begin
      if rising_edge(CLOCK) then
            case state is

            when IDLE =>
                  Enable_Network <= '0';
                  next_state <= WAIT_RECEIVE;
                  rs232_receiver_rst <= '0';

            when WAIT_RECEIVE => -- wait for RS232 data
                  if (rs232_receiver_stb = '1') then -- data done
                        rs232_receiver_rst <= '1';
                        if    (rs232_receiver_dat = x"74" ) then -- 't'
                              train_mode <= '1';
                              next_state <= START_ADC;
                        elsif (rs232_receiver_dat = x"6E" ) then -- 'n'
                              train_mode <= '0';
                              next_state <= START_ADC;
                        else
                              next_state <= IDLE;
                        end if;
                  else
                        next_state <= WAIT_RECEIVE; --receiving data
                  end if;

            when START_ADC =>
                  ce_amp <= '1'; --active high
                  start_conv <= '1';
                  next_state <= ADC;

            when ADC =>
                  if ADC_DONE = '1' then
                        if (train_mode = '1') then
                              next_state <= START_RS232_TRAIN1;
                        else
                              next_state <= ADC2FIXED;
                        end if;
                        ce_amp <= '0'; --active low
                        start_conv <= '0';
                  else
                        next_state <= ADC;
                  end if;

            when START_RS232_TRAIN1 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <= ADC1(13) & ADC1(13) & ADC1(13) &
ADC1(13) & ADC1 (13 downto 10);
                  next_state <= RS232_TRAIN_HI1;
            when RS232_TRAIN_HI1 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1' then –
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) = 
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= START_RS232_TRAIN_LO1;
                  else
                        next_state <= RS232_TRAIN_HI1;
                  end if;

            when START_RS232_TRAIN_LO1 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <=  ADC1 (9 downto 2);
                  next_state <= RS232_TRAIN_LO1;

            when RS232_TRAIN_LO1 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1'
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) = 
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= START_RS232_TRAIN2;
                  else
                        next_state <= RS232_TRAIN_LO1;
                  end if;

            when START_RS232_TRAIN2 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <=  ADC2(13) & ADC2(13) & ADC2(13) & 
ADC2(13) & ADC2 (13 downto 10);
                  next_state <= RS232_TRAIN_HI2;

            when RS232_TRAIN_HI2 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1' then
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) = 
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= START_RS232_TRAIN_LO2;
                  else
                        next_state <= RS232_TRAIN_HI2;
                  end if;

            when START_RS232_TRAIN_LO2 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <=  ADC2 (9 downto 2);
                  next_state <= RS232_TRAIN_LO2;
            when RS232_TRAIN_LO2 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1' then
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) =
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= START_RS232_TRAIN3;
                  else
                        next_state <= RS232_TRAIN_LO2;
                  end if;

            when START_RS232_TRAIN3 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <= ADC3(13) & ADC3(13) & ADC3(13) &
ADC3(13) & ADC3 (13 downto 10);
                  next_state <= RS232_TRAIN_HI3;

            when RS232_TRAIN_HI3 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1' then
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) =
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= START_RS232_TRAIN_LO3;
                  else
                        next_state <= RS232_TRAIN_HI3;
                  end if;

            when START_RS232_TRAIN_LO3 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <=  ADC3 (9 downto 2);
                  next_state <= RS232_TRAIN_LO3;

            when RS232_TRAIN_LO3 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1' then
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) =
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= START_RS232_TRAIN4;
                  else
                        next_state <= RS232_TRAIN_LO3;
                  end if;

            when START_RS232_TRAIN4 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <=  ADC4(13) & ADC4(13) & ADC4(13) &
ADC4(13) & ADC4 (13 downto 10);
                  next_state <= RS232_TRAIN_HI4;
            when RS232_TRAIN_HI4 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1' then
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) = 
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= START_RS232_TRAIN_LO4;
                  else
                        next_state <= RS232_TRAIN_HI4;
                  end if;

            when START_RS232_TRAIN_LO4 =>
                  counter <= (others => '0');
                  rs232_sender_stb <= '1'; -- start pulse on
                  rs232_sender_dat <=  ADC4 (9 downto 2);
                  next_state <= RS232_TRAIN_LO4;

            when RS232_TRAIN_LO4 =>
                  counter <= counter + 1;
                  if rs232_sender_ack = '1' then –
                        rs232_sender_stb <= '0'; --start pulse off
                  elsif counter (19 downto 0) = 
CONV_STD_LOGIC_VECTOR(system_speed/baudrate*10, 20) then
                        next_state <= IDLE;
                  else
                        next_state <= RS232_TRAIN_LO4;
                  end if;

            when others =>
                  next_state <= IDLE;
            end case; -- state
      end if; -- clock

      end process sensor_state_machine;

            ADC1_fixed <= ADC1(13) & ADC1(13 downto 2);
            ADC2_fixed <= ADC2(13) & ADC2(13 downto 2);
            ADC3_fixed <= ADC3(13) & ADC3(13 downto 2);
            ADC4_fixed <= ADC4(13) & ADC4(13 downto 2);
            F8PIN  <= rdy_tf;
            E8PIN  <= '0';

            led(4) <= '0';
            led(5) <= '0';
            led(6) <= train_mode;
            led(7) <= not RS232_DCE_RXD;

            state <= next_state;

            RS232_DCE_TXD     <= RS232_TX_out;
            SPI_AMP_SHDN      <= '0';
            DAC_CS                  <= '1';
            SPI_SS_B                <= '1';
            gain <= "00010001";
            AMP_DOUT_OUT <= AMP_DOUT_IN;

      end Behavioral;
```

Appendix E: PD Controller Real Time Executive Code: CODE FROM main.c

```
      #include "main.h"
      #include <string.h>

      unsigned char input1=127;     //High byte of position from FPGA
      unsigned char width1=10;//Debugging pulse width in high byte read
      unsigned char width2=137;//Debugging pulse width in low byte rea
      unsigned char input2=127;     //Low byte of position from FPGA
      unsigned int16 input=400;     //Total position from FPGA
      unsigned char servoWidthCounter=127;//Counter used in servo loop
      short servoFlag = 1;          //Flag to indicate 20ms servo
window
      short servoPhase = 0;         //Flag to indicate servo restart
      int x;                        //Counter indicating which byte is
read
      signed int16 servoWidth=30;   //Width of servo pulse in counts

      #include "pid.h"

      // Received Data Available Interupt Subroutine
      // Reads 1 byte of data from our RS232 in
      // Stores the byte to the in global variable
      // Then write the letter 'n' to request for a new position

      #int_RDA
      void  RDA_isr(void)
      {
      if(x==1)
         {
         //Read high byte of position from FPGA

         input1 = getc();
         x++;
         }
      else if(x==2)
         {
         input2 = getc();
         PIDWindow = 1; //lets compute another sample3
         x=0;
         input = make16(input1,input2);
         }
      }

      // RTCC Interuput Subroutine
      // Interputed every 51.2us
      // Used to control the pulse width for the servo
      // Most servos should be between 1 and 2 ms pulses
      // window may be different for different servos
```

```
#int_RTCC
void  RTCC_isr(void)
{
if(servoFlag == 1) //Has our 20ms window passed?
   {
   if(servoWidthCounter<servoWidth)
      //Are we in the variable 0-2ms window?
      {
      output_high(SERVO);          //Set the servo pin high
      servoWidthCounter++;         //And increment our counter
      }
   else
      {
      output_low(SERVO);//outside of the pulse, turn off pulse
      servoFlag = 0;
      }
   }
}

// Timer1 Interuput Subroutine
// Interputed every 13.1ms
// Used to trigger step of the PID controller/Restart Servo PW

#int_TIMER1
void  TIMER1_isr(void)
{
//Servo can only be triggered every 20ms,
//    but we are sampling ever 13.1ms.
//A flag, ServoPhase is set to set
//    to determine if the servo can be updated.

if(servoPhase==0)//restart servo pulse
   {
   servoWidthCounter = 0;
   servoFlag = 1;
   servoPhase = 1;
   }
else //servo is resting
   {
   servoPhase = 0;
   }
}
```

```
void main(void)
{
init();
set_pwm1_duty(255);
while(1)
   {
   if(PIDwindow==1)
      {
      //request another position from the FPGA

      putc('n',osu);

      //Indicate that the first byte should be read

      x=1;

      //Call the PID controller

      PID_Controller();
      }
   }
}
```

Appendix F: PD Controller Real Time Executive Code: CODE FROM main.h

```
      #include <18F1220.h>
      #FUSES NOWDT                  //No Watch Dog Timer
      #FUSES WDT128                 //Watch Dog Timer uses 1:128 
Postscale
      #FUSES H4                     //High speed osc with HW enabled 4X 
PLL
      #FUSES FCMEN                  //Fail-safe clock monitor enabled
      #FUSES BROWNOUT         //Reset when brownout detected
      #FUSES BORV42                 //Brownout reset at 4.2V
      #FUSES NOPUT                  //No Power Up Timer
      #FUSES NOCPD                  //No EE protection
      #FUSES STVREN                 //Stack full/underflow will cause 
reset
      #FUSES NODEBUG          //No Debug mode for ICD
      #FUSES NOLVP            //No low voltage prgming,
                              //B3(PIC16) or B5(PIC18) used for I/O
      #FUSES NOWRT            //Program memory not write protected
      #FUSES NOWRTD                 //Data EEPROM not write protected
      #FUSES NOWRTC     //configuration not registers write protected
      #FUSES IESO       //Internal External Switch Over mode enabled
      #FUSES NOEBTR           //Memory not protected from table reads
      #FUSES NOEBTRB    //Boot block not protected from table reads
      #FUSES MCLR             //Master Clear pin enabled
      #FUSES NOPROTECT        //Code not protected from reading
      #FUSES NOCPB                  //No Boot Block code protection
      #FUSES NOWRTB                 //Boot block not write protected
      #use delay(clock=40M)

      #use 
rs232(baud=38400,parity=N,xmit=PIN_B1,rcv=PIN_B4,bits=8,stream=osu)

      #define SERVO           PIN_A0
      void init(void)
      {
      setup_adc_ports(NO_ANALOGS|VSS_VDD);
      setup_adc(ADC_CLOCK_DIV_2|ADC_TAD_MUL_0);
      setup_wdt(WDT_OFF);
      setup_timer_0(RTCC_INTERNAL|RTCC_DIV_2|RTCC_8_bit);
      setup_timer_1(T1_INTERNAL|T1_DIV_BY_2);
      setup_timer_2(T2_DIV_BY_16,255,1);
      setup_timer_3(T3_DISABLED|T3_DIV_BY_1);
      setup_ccp1(CCP_PWM);
      enable_interrupts(INT_RTCC);
      enable_interrupts(INT_TIMER1);
      enable_interrupts(INT_RDA);
      enable_interrupts(GLOBAL);
      set_pwm1_duty(0);
      output_high(PIN_B2);
      }
```

Appendix G: PD Controller Real Time Executive Code: CODE FROM PID.h

```
short PIDWindow = 1;          //Flag to indicate that control
                                    //    loop should execute
float32 prevPos = 0;          //Previous position (meters)
float32 currentPos = 0; //Current position (meters)
float32 vel = 0;              //Velocity (meters/s)
float32 st_angle = 0;   //Steering angle (radians)
float32 position=0;     //Car position from center line in meters
signed int16 fullpos=400;     //temp variable to convert input
                                    //    to signed int16
// Define parameters

float32 kp = -60.0;      //Position feedback gain (radians/meter)
float32 kv = -500.000; //Velocity feedback gain (radians/meter/s)
float32 alpha = 0.02;         //Filter parameter (0<alpha<1)
float32 one_m_alpha = 0.98;   // (1-alpha)
float32 setpoint = 0.0; //Position set point (meters)
float32 dt = 0.02;            //Sampling interval (s)
signed int16 poscenter=400;   //Center position in 1/100 in
float32 met_conv = 0.000254;//Conversion from 1/100 in to meters

void PID_Controller()
{
//Adjust for the center position and convert to meters

fullpos = input;
position = met_conv*(fullpos - poscenter);

// Increment time - save the old position

prevPos = currentPos;

// Filter the position

currentPos = alpha*position + one_m_alpha*prevPos;

// Compute the velocity

vel = (currentPos-prevPos)/dt;

// Compute the control signal (steering angle in radians)

st_angle = (currentPos - setpoint)*kp + kv*vel;

// Convert the control signal to pulse width count
servoWidth = 19.1*st_angle + 29.0;
if(servoWidth>39)
   {
   //servoWidth=42;
   servoWidth=39;
   }
else if(servoWidth<19)
   {
   //servoWidth=30;
   servoWidth=19;
   }
//clear the PIDwindow so we can wait for the next sample
PIDwindow=0;
return;
}
```

VITA

Jeremy Paul Evert

Candidate for the Degree of

Master of Science

Thesis: TEST BED FOR DEMONSTRATING AND TEACHING SOFT SENSOR CONCEPTS

Major Field: Electrical and Computer Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2010.

Completed the requirements for the Bachelor of Science in Mechanical and Nuclear Engineering at Kansas State University, Manhattan, Kansas in 2003.

Experience:

Active Duty Air Force Officer from April 2004 through April 2008.
Executive Officer, 448th Combat Sustainment Wing, Tinker Air Force Base, OK from November 2007 through March 2008.
Staff Officer, Oklahoma City Air Logistics Center/LR, Tinker Air Force Base, OK from April 2005 through November 2007.
Staff Officer, Oklahoma City Air Logistics Center Engineering Directorate, Tinker Air Force Base, OK from April 2004 through April 2005.

Professional Memberships:

Institute of Electrical and Electronics Engineers
International Society of Automation

Name: Jeremy Paul Evert Date of Degree: December, 2010

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: TEST BED FOR DEMONSTRATING AND TEACHING SOFT SENSOR CONCEPTS

Pages in Study: 277 Candidate for the Degree of Master of Science

Major Field: Electrical and Computer Engineering.

The smart sensor car is a test bed for demonstrating soft sensor concepts. The smart car follows the magnetic field coming from a wire track. Sensors on the smart car detect the magnetic field and generate signals. Those signals are conditioned and converted to digital numbers, which are used by the neural network as inputs. The neural network calculates the car position from these inputs. The car position is sent to a controller that calculates the car steering angle. The commands from the controller drive the smart sensor car around the track, where the sensors generate different signals resulting in different commands from the controller. The neural network is implemented on an Field Programmable Gate Array (FPGA) in a serial configuration.

ADVISER'S APPROVAL: Dr. Martin Hagan