Feedback Control for a Path Following Robotic Car

Patricia Mellodge

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

April 2, 2002
Blacksburg, Virginia

Keywords: autonomous vehicle, intelligent transportation system, lateral control, nonholonomic, path following, curvature estimation

Feedback Control for a Path Following Robotic Car

Patricia Mellodge

(ABSTRACT)

This thesis describes the current state of development of the Flexible Low-cost Automated Scaled Highway (FLASH) laboratory at the Virginia Tech Transportation Institute (VTTI). The FLASH lab and the scale model cars contained therein provide a testbed for the small scale development stage of intelligent transportation systems (ITS). In addition, the FLASH lab serves as a home to the prototype display being developed for an educational museum exhibit.

This thesis also gives details of the path following lateral controller implemented on the FLASH car. The controller was developed using the kinematic model for a wheeled robot. The global kinematic model was derived using the nonholonomic contraints of the system. This global model is converted into the path coordinate model so that only local variables are needed. Then the path coordinate model is converted into chained form and a controller is given to perform path following.

The path coordinate model introduces a new parameter to the system: the curvature of the path. Thus, it is necessary to provide the path's curvature value to the controller. Because of the environment in which the car is operating, the curvature values are known a priori. Several online methods for determining the curvature are developed.

A MATLAB simulation environment was created with which to test the above algorithms. The simulation uses the kinematic model to show the car's behavior and implements the sensors and controller as closely as possible to the actual system.

The implementation of the lateral controller in hardware is discussed. The vehicle platform is described and the hardware and software architecture detailed. The car described is capable of operating manually and autonomously. In autonomous mode, several sensors are utilized including: infrared, magnetic, ultrasound, and image based technology. The operation of each sensor type is described and the information received by the processor from each is discussed.

Acknowledgments

The author would like to thank Dr. Pushkin Kachroo for all his help, support, and seemingly endless enthusiasm. Being within his ”radius of understanding” allowed the author to gain much insight into the field of control as well as many other fields of research and life. Without him this research would not have been possible.

In addition, the author would like to thank Dr. Abbott and Dr. VanLandingham for serving on the advisory committee for this thesis.

The author would also like to thank the members of the FLASH team, in particular Ricky Henry and Eric Moret. It was their dedication and hardwork that have made the FLASH lab what it is today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The automation of the driving task has been the subject of much research recently. Automobile manufacturers have developed and are continuing to develop systems for cars that alleviate the driver's need to monitor and control all aspects of the vehicle. Such systems include antilock braking systems, traction control, cruise control and others in development that will surely alleviate the driving burden in years to come.

1.1 Motvation

Why automate the driving task? One of the major reasons is safety. In 2000, there were approximately 6,394,000 police reported motor vehicle traffic crashes, resulting in 3,189,000 people being injured and 41,821 lives lost [1]. Accidents on our roadways not only cause injuries and fatalities, but they also have a huge economic impact [2]. Many accidents are caused by human error and eliminating this error will reduce the number of injuries and fatalities on our roadways.

Human driving error may be caused by a number of factors including fatigue and distraction. During long drives on the highway, the driver must constantly monitor the road conditions and react to them over an extended period of time. Such constant attentiveness is tiring and the resulting fatigue may reduce the driver's reaction time. Additionally, the driver may be distracted from the task of driving by conversations with other passengers, tuning the radio, using a cell phone, etc. Such distractions may also lead to accidents. According to [3], driver distraction was a factor in 11% of fatal crashes and 25-30% of injury and property-damage-only crashes in 1999. Viewed from another perspective however, a car capable of driving itself can allow the occupants to perfom non-driving tasks safely while traveling to their destination.

Another reason to automate cars is to alleviate congestion on the highways. A method called "platooning" would allow cars to drive at highway speed while only a few feet apart. Since the electronics on the car can respond faster than a human, cars would be able to drive much closer together. This would allow much more efficient use of the existing highways in a safe manner.

1.2 Autonomous Vehicles

The inventions of the integrated circuit (IC) and later, the microcomputer, were major factors in the development of electronic control in automobiles. The importance of the microcomputer cannot be overemphasized as it is the "brain" that controls many systems in today's cars. For example, in a cruise control system, the driver sets the desired speed and enables the system by pushing a button. A microcomputer then monitors the actual speed of the vehicle using data from velocity sensors. The actual speed is compared to the desired speed and the controller adjusts the throttle as necessary. See [4] for a complete overview of electronic control systems used in cars today.

The U.S. government has also played a role in encouraging the technological advancement of automobiles and development of intelligent transportation systems (ITS). In the early 1990s, the Center for Transportation Research (now known as the Virginia Tech Transportation Institute (VTTI)) received funding to build the Smart Road, a 6 mile highway connecting Blacksburg, VA to Interstate 81. The road was built to be "intelligent", with sensors embedded to alert a traveling vehicle to road conditions. Advanced automotive technologies such as lane detection, obstacle detection, adaptive cruise control, collision avoidance, and lateral control were intended to be developed using the Smart Road as a testing ground [5].

A completely autonomous vehicle is one in which a computer performs all the tasks that the human driver normally would. Ultimately, this would mean getting in a car, entering the destination into a computer, and enabling the system. From there, the car would take over and drive to the destination with no human input. The car would be able to sense its environment and make steering and speed changes as necessary.

This scenario would require all of the automotive technologies mentioned above: lane detection to aid in passing slower vehicles or exiting a highway; obstacle detection to locate other cars, pedestrians, animals, etc.; adaptive cruise control to maintain a safe speed; collision avoidance to avoid hitting obstacles in the roadway; and lateral control to maintain the car's position on the roadway. In addition, sensors would be needed to alert the car to road or weather conditions to ensure safe traveling speeds. For example, the car would need to slow down in snowy or icy conditions.

We perform many tasks while driving without even thinking about it. Completely automating the car is a challenging task and is a long way off. However, advances have been made in the individual systems. Cruise control is common in cars today. Adaptive cruise control, in which the car slows if it detects a slower moving vehicle in front of it, is starting to be-

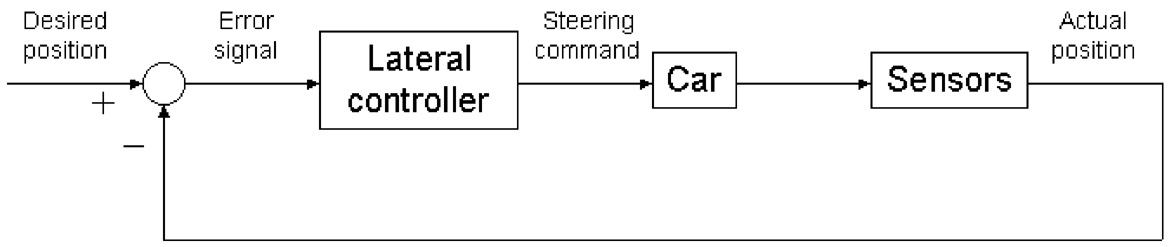

Figure 1.1: Block diagram of the lateral controller.

come available on higher-end models. In addition, some cars come equiped with sensors to determine if an obstacle is near and sounds an audible warning to the driver when it is too close.

The focus of this work is lateral control. With this type of vehicle control, the driver would be able to remove his hands from the steering wheel and let the car steer itself. Here, the idea is that the car has some desired path to follow. Sensors on the car must be able to detect the location of the desired path. The error between the desired path and the car is calculated and the microcomputer acting as the controller determines how to turn the steering wheels to follow the correct path. Fig. 1.1 shows the feedback control system for lateral control.

The lateral controller's purpose is to follow the desired path. It does not determine what the desired path is. A higher level planner is responsible for that task. This planner may take into account data from other sensors so as to avoid collisions or arrive at its ultimate destination. The lateral controller does not know or need to know such high level information. It only needs to know the car's location with respect to the desired path.

1.3 Previous Research

1.3.1 Modeling

Designing a lateral controller requires a model of the vehicle's behavior. There have been two approaches to this modeling: dynamic and kinematic.

Dynamic modeling takes into account such factors as the vehicle's weight, center of gravity, cornering stiffness, wheel slippage, and others. The resulting equations, as used in [6], are very complex and difficult to work with. In addition, it may be difficult to measure parameters such as cornering stiffness. However, they give a highly accurate portrayal of the vehicle's behavior and the controllers designed with them are robust to those dynamics.

A simpler approach to modeling (and the one used here) is to ignore the *dynamics* of the system and only use its *kinematics*. The effects of weight, inertia, etc. are ignored and the model is derived using only the nonholonomic contraints of the system as in [7]. The advantage of this model is that it is much simpler than the dynamic one. However, it is a much less accurate depiction of the actual system as a result. Details of this model are given in Chapter 3.

1.3.2 Controllers

There has been much research in the area of control theory and many modern controllers have been developed as a result. The most widely used controller is still the PID (proportional, integral, derivative) controller because of it simplicity and ease of implementation. However, with the increasing power of computers and microprocessors, more robust and more powerful controllers are able to be implemented in many systems.

Among the classes of controllers that have been implemented are: fuzzy controllers, neural networks, and adaptive controllers. In addition, specific controllers are developed for individual applications. In this work, an input scaling controller is implemented and is described in Chapter 3.

1.3.3 Sensors

The controller must know where the path is located with respect to the vehicle. This location information is provided by sensors on the vehicle. Various sensors are available to perform this task and their accuracy and ease of implementation vary. Also, certain types of sensors require changes to the roads themselves while others can be used on existing roads.

Cameras

Much research has been devoted to the use of cameras in autonomous vehicles. The camera is used to take images of the roadway in front of the vehicle. Image processing is then performed to extract information from the image about the car's location on the road. This type of sensing is most like that used by human drivers. The camera sees ahead and the controller can make steering adjustments based on how the road is curving up ahead.

Infrared Sensors

Infrared sensors have been used to detect white lines on dark pavement. Infrared light is emitted from LEDs under the car. The light is reflected by the white line and absorbed by the dark pavement. Sensors detect the light that is reflected back and so the location of the white line is known. This method assumes that the car is to follow the white line.

Magnetic Sensors

Magnetic sensors work by detecting the presence of a magnetic field. Sensors under the car detect magnets embedded in the roadway. This method is very similar to the use of infrared sensors but requires major changes to the infrastructure since most roads do not have magnets embedded in them.

Radar

The use of radar follows the same principle as infrared technology except a different kind of energy is used. An RF signal is emitted towards the road and it may be redirected back by a reflector stripe. Thus position information is provided to the controller.

Each of the sensor types has advantages and disadvantages. Chapter 6 gives specific descriptions of the sensors used in this project. In that chapter, the relative merits are discussed in more detail.

1.4 Contributions of this Thesis

The following contributions were made during this thesis work:

- A simulation environment was developed to test the various algorithms used on the vehicle.
- Hardware implementation of a lateral controller was done on a 1/10 scale model car using infrared and magnetic sensors.
- Several methods of curvature estimation were developed and tested for use with the controller.
- This thesis provides full documentation of the project's hardware and software as it existed at the time of this writing.

1.5 Organization of this Thesis

This thesis is organized as follows:

- Chapter 2 describes the FLASH project for which the 1/10 scale car has been developed.
- Chapter 3 gives the derivation of the car's mathematical model and control law.
- Chapter 4 provides several methods of curvature estimation for use with the controller.
- Chapter 5 fully describes the simulation environment and gives simulation results.
- Chapter 6 provides complete documentation of the hardware implementation of the car.
- Chapter 7 gives conclusions and possibilities for future work.

Chapter 2

Project Background

This chapter describes the FLASH (Flexible Low-cost Automated Scaled Highway) project at the Virginia Tech Transportation Institute (VTTI). Previous work done on this project is discussed as well as the current status of the lab's development.

2.1 Purpose

2.1.1 Scale Model Testing

The FLASH laboratory was created at VTTI as one stage in the four-stage development of automated highway systems. Each of the stages is shown in Fig. 2.1 [8]. The first stage is software, during which simulations are run to ensure the viability of designs. Next, scale modeling is done and designs are tested in hardware. After scale modeling comes full scale testing, as is done on the Smart Road. Finally, the systems are deployed and made available commercially.

The second stage, scale modeling, allows for the safe and inexpensive implementation of

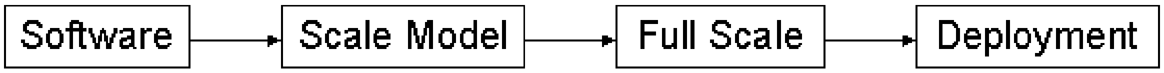

Figure 2.1: The four stages of ITS development.

protoype designs. It is more cost effective and safer to use a scale model car rather than a full scale car for initial testing. Testing (and repairing after the inevitable crashes!) is also easier on a scale model vehicle. Additionally, a full scale protoype requires a full scale roadway on which to test, rather than the relatively small area needed for scale model testing.

The FLASH laboratory fulfills this need for scale modeling. The lab itself is located in a 1600 square foot trailer at VTTI. It contains a scale roadway and several 1/10 scale cars. Each car is capable of operating manually or autonomously. The cars are described in detail in Chapter 6.

2.1.2 Eduational Exhibit

Currently, an educational exhibit is being developed to educate the public about vehicle technology and ITS. The exhibit, to be displayed in the Science Museum of Virginia in Richmond and the Virginia Museum of Transportation in Roanoke, is intended to help the public understand the technology that is currently available and what will likely be available in the near future.

The exhibit will include displays with which the public can interact to understand the technologies being used. A concept of the exhibit layout and track is shown in Fig. 2.2. Additionally, there will be several *working*, fully autonomous 1/10 scale cars driving around the track. These cars will actively demonstrate the technology that is presented by the interactive displays. These technologies include:

- Infrared and magnetic sensors for lateral control
- Image processing for lateral and longitudinal control
- Ultrasound for adaptive cruise control and obstacle detection
- In-vehicle navigation and traveler information

The FLASH lab also fulfills this need for developing the museum displays. In addition to the scale roadway and cars, the FLASH lab is home to the prototype interactive displays.

2.2 Previous FLASH Development

Previous versions of the FLASH vehicles were capable of manual and autonomous driving in a laboratory setting. The vehicles were regular remote control (RC) cars like those available from many hobby shops. These RC cars were then modified to include the various sensors and controllers needed for autonomous driving. Complete details of this work are given in [8].

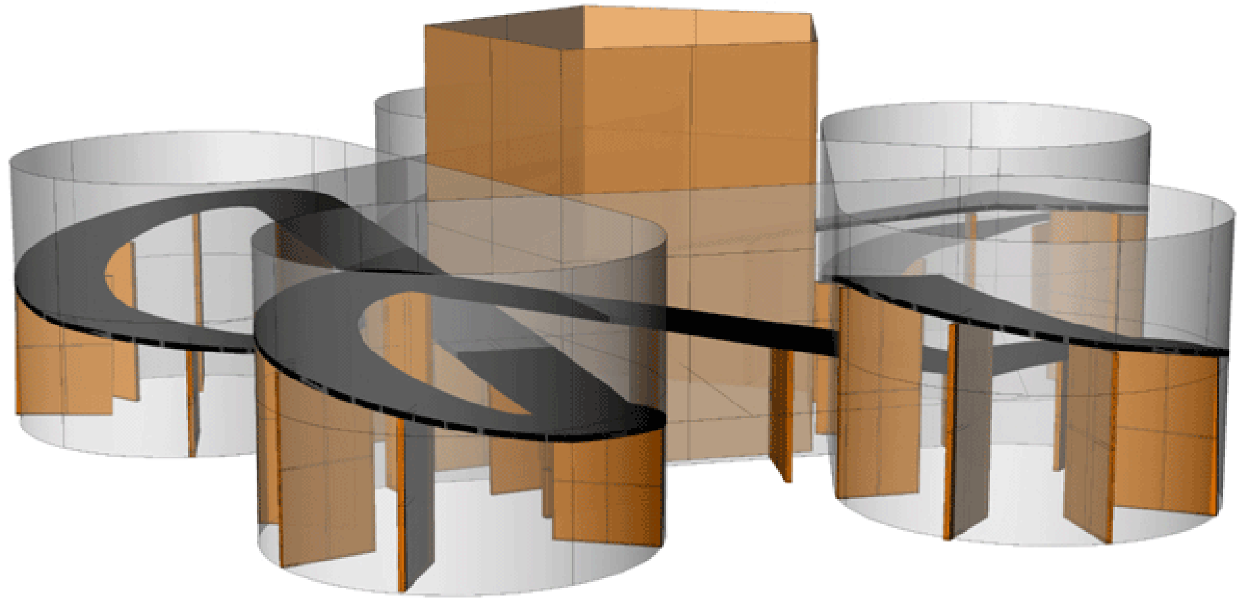

Figure 2.2: Layout concept for the museum exhibit.

Several modfications were made to the standard RC cars to improve their performance. The standard handheld controller was replaced by a steering console interfaced with a PC to allow for a more real-life driving position. On board the car, the steering and velocity commands were sent from the wireless receiver to a 68HC11 microcontoller. This microcontroller interpreted the received commands into PWM signals for the steering servo and motor. Additionally, the microcontroller implemented speed control by using feedback from an optical encoder. The 68HC11 allowed for very precise control of the velocity and steering.

For autonomous driving, infrared sensors or a camera were used. The data from the infrared sensors was used by the 68HC11 to perform lateral control. Signals from the camera were sent via wireless link to a frame grabber housed in a PC. The PC then processed the image and used the information to determine steering commands. The infrared sensors and camera were not used simultaneously.

The cars were powered by a single standard 7.2V NiCd RC car battery. With these batteries and all of the addtional electronics, about 15 minutes of drive time was provided.

2.3 Current FLASH Development

While the cars currently being developed in the FLASH lab are similar to the previous versions in concept, the implementation is very different. Because the cars will be part of a museum display, the emphasis is now on reliability. The cars must be capable of running most of the day with little intervention from the museum staff. Also, the cars must be robust to little kids throwing stuff at them. Current development is done with these issues in mind.

These concerns are addressed in the following ways:

- Low power components are used wherever possible to minimize power consumption.
- The cars are capable of automatic recharging and the display itself houses recharging bays.
- Manual driving is diabled to prevent museum visitors from controlling the vehicle.
- All control processing is done on board the car (rather than sent via wireless link to a PC) to preserve the integrity of the data.
- The display is enclosed and the cars have bodies for asthetics and to prevent access to the circuitry.

While this section provides an overview and highlights of the current FLASH car development, complete details of the hardware and software are given in Chapter 6.

Chapter 3

Mathematical Modeling and Control Algorithm

In this chapter, the kinematic model for the rear wheel drive, front wheel steered robotic car is derived. Using this model, a controller is given to perform path following.

3.1 Mathematical Modeling

The model used throughout this work is a kinematic model. This type of model allows for the decoupling of vehicle dynamics from its movement. Therefore, the vehicle's dynamic properties, such as mass, center of gravity, etc. do not enter into the equations. To derive this model, the nonholonomic constraints of the system are utilized.

3.1.1 Nonholonomic Contraints

If a system has restrictions in its velocity, but those restrictions do not cause restrictions in its positioning, the system is said to be nonholonomically constrained. Viewed another way, the system's local movement is restricted, but not its global movement. Mathematically, this means that the velocity constraints cannot be integrated to position constraints.

The most familiar example of a nonholonomic system is demonstrated by a parallel parking maneuver. When a driver arrives next to a parking space, he cannot simply slide his car sideways into the spot. The car is not capable of sliding sideways and this is the velocity restriction. However, by moving the car forwards and backwards and turning the wheels, the car can be placed in the parking space. Ignoring the restrictions caused by external objects, the car can be located at any position with any orientation, despite lack of sideways movement.

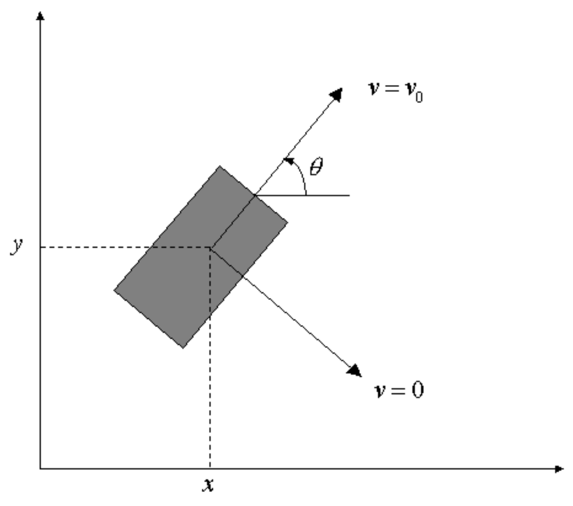

Figure 3.1: The velocity constraints on a rolling wheel with no slippage.

The nonholonomic constraints of each wheel of the mobile robot are shown in Fig. 3.1. The wheel's velocity is in the direction of rolling. There is no velocity in the perpendicular direction. This model assumes that there is no wheel slippage.

3.1.2 Global Coordinate Model

The exact position and orientation of the car in some global coordinate system can be described by four variables. Fig. 3.2 shows each of the variables. The (x, y) coordinates give the location of the center of the rear axle. The car's angle with respect to the x-axis is given by θ. The steering wheel's angle with respect to the car's longitudinal axis is given by ϕ.

From the constraints shown in Fig. 3.1, the velocity of the car in the x and y directions is given as

$$\dot{x} = v_1 \cos\theta \tag{3.1}$$
$$\dot{y} = v_1 \sin\theta \tag{3.2}$$

where v_1 is the linear velocity of the rear wheels.

The location of the center of the front axle (x_1, y_1) is given by

$$x_1 = x + l\cos\theta \tag{3.3}$$
$$y_1 = y + l\sin\theta \tag{3.4}$$

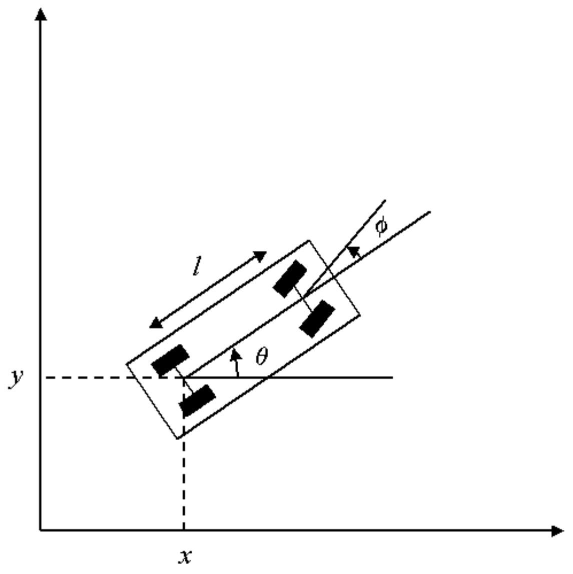

Figure 3.2: The global coordinate system for the car.

and the velocity is given by

$$\dot{x_1} = \dot{x} - l\dot{\theta}\sin\theta \tag{3.5}$$
$$\dot{y_1} = \dot{y} + l\dot{\theta}\cos\theta \tag{3.6}$$

Applying the no-slippage constraint to the front wheels gives

$$\dot{x_1}\sin(\theta + \phi) = \dot{y_1}\cos(\theta + \phi) \tag{3.7}$$

Inserting (3.5) and (3.6) into (3.7) and solving for $\dot{\theta}$ yields

$$\dot{\theta} = \frac{\tan\phi}{l} v_1 \tag{3.8}$$

The complete kinematic model is then given as

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{\tan\phi}{l} \\ 0 \end{bmatrix} v_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} v_2 \tag{3.9}$$

where v_1 is the linear velocity of the rear wheels and v_2 is the angular velocity of the steering wheels.

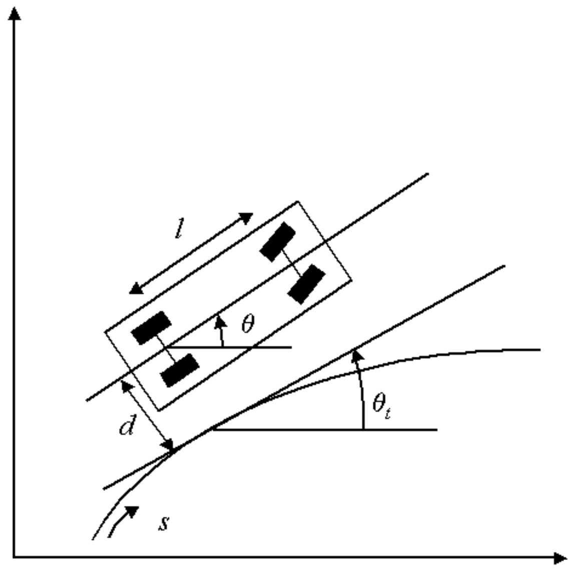

Figure 3.3: The path coordinates for the car.

3.1.3 Path Coordinate Model

The global model is useful for performing simulations and its use is described in Chapter 5. However, on the hardware implementation, the sensors cannot detect the car's location with respect to some global coordinates. The sensors can only detect the car's location with respect to the desired path. Therefore, a more useful model is one that describes the car's behavior in terms of the path coordinates.

The path coordinates are shown in Fig. 3.3. The perpendicular distance between the rear axle and the path is given by d. The angle between the car and the tangent to the path is $\theta_p = \theta - \theta_t$. The distance traveled along the path starting at some arbitrary initial position is given by s, the arc lengh.

The car's kinematic model in terms of the path coordinates is given by [9]

$$\begin{bmatrix} \dot{s} \\ \dot{d} \\ \dot{\theta}_p \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \frac{\cos\theta_p}{1-dc(s)} \\ \sin\theta_p \\ \frac{\tan\phi}{l} - \frac{c(s)\cos\theta_p}{1-dc(s)} \\ 0 \end{bmatrix} v_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} v_2 \tag{3.10}$$

where $c(s)$ is the path's curvature and is defined as

$$c(s) = \frac{d\theta_t}{ds}$$

3.2 Control Law

3.2.1 Path Following

There are three possible tasks that the car could perform: point-to-point stabilization, path following, and trajectory tracking. Point-to-point stabilization requires that the car move from point A to point B with no restrictions on its movement between those two points. With path following, the car must move along a geometric path. Trajectory tracking is similar to path following, except the car must follow a path at a given speed.

In this project, the goal for the car is path following. The car must sense its position with respect to the path and return to the path if it is off course. The track in the lab contains a white line on a black surface which the car is to follow. In addition, there are magnets beneath the track. The car can sense both of these types of lines and both provide a path for following. A higher level planner, independent of the controller discussed here, is responsible for determining which type of line to follow. The hardware and software that performs the sensing and planning are discussed in Chapter 6.

3.2.2 Chained Form

Before developing the controller for the model given in (3.10), the system must be converted into chained form. The (2,n) single-chain form has the following structure [10]:

$$\begin{aligned} \dot{x_1} &= u_1 \\ \dot{x_2} &= u_2 \\ \dot{x_3} &= x_2 u_1 \\ &\vdots \\ \dot{x_n} &= x_{n-1} u_1 \end{aligned} \tag{3.11}$$

Although the system has two inputs, u_1 and u_2, this model can be considered single input if u_1 is known a priori.

For the car model with four states, the (2,4) chained form becomes

$$\begin{aligned} \dot{x_1} &= u_1 \\ \dot{x_2} &= u_2 \\ \dot{x_3} &= x_2 u_1 \\ \dot{x_4} &= x_3 u_1 \end{aligned} \tag{3.12}$$

The states are given as

$$x_1 = s \tag{3.13}$$

$$\begin{aligned}
x_2 &= -c'(s)d\tan\theta_p - c(s)(1-dc(s))\frac{1+\sin^2\theta_p}{\cos^2\theta_p} + \frac{(1-dc(s))^2\tan\phi}{l\cos^3\theta_p} && (3.14)\\
x_3 &= (1-dc(s))\tan\theta_p && (3.15)\\
x_4 &= d && (3.16)
\end{aligned}$$

where the variables are defined in Fig. 3.3, $c(s)$ is the path's curvature, and $c'(s)$ denotes the derivative of c with respect to s.

The inputs are defined as follows

$$\begin{aligned}
v_1 &= \frac{1-dc(s)}{\cos\theta_p}u_1 && (3.17)\\
v_2 &= \alpha_2(u_2 - \alpha_1 u_1) && (3.18)
\end{aligned}$$

where v_1 is the linear velocity of the rear wheels, v_2 is the angular velocity of the steering wheels, and

$$\begin{aligned}
\alpha_1 &= \frac{\partial x_2}{\partial s} + \frac{\partial x_2}{\partial d}(1-dc(s))\tan\theta_p + \frac{\partial x_2}{\partial \theta_p}\left[\frac{\tan\phi(1-dc(s))}{l\cos\theta_p} - c(s)\right]\\
\alpha_2 &= \frac{l\cos^3\theta_p\cos^2\phi}{(1-dc(s))^2}
\end{aligned}$$

3.2.3 Input-Scaling Controller

With the system in chained form, the controller to perform path following can be developed. In this form, path following equates to stabilizing x_2, x_3, x_4 from (3.12) to zero. The input scaling controller from [9] is given here.

First the variables are redefined as follows

$$\chi = (\chi_1, \chi_2, \chi_3, \chi_4) = (x_1, x_4, x_3, x_2)$$

so the chained form system is then

$$\begin{aligned}
\dot{\chi_1} &= u_1\\
\dot{\chi_2} &= \chi_3 u_1\\
\dot{\chi_3} &= \chi_4 u_1\\
\dot{\chi_4} &= u_2 && (3.19)
\end{aligned}$$

As stated in [9], the system (3.19) is controllable if $u_1(t)$ is a "piecewise continuous, bounded, and strictly positive (or negative) function". With u_1 known a priori, u_2 is left as the only input to the system. The controller for u_2 (with the appropriate restrictions on u_1) becomes

$$u_2 = -k_1|u_1(t)|\chi_2 - k_2 u_1(t)\chi_3 - k_3|u_1(t)|\chi_4 \tag{3.20}$$

Chapter 4

Curvature Estimation

The model for the car and the resulting controller given in the previous chapter require knowledge of the path's curvature. This chapter describes several methods for estimating the path's curvature.

Except for $c(s)$, all of the variables in (3.10) are known or can be measured by sensors on the car. The feedback control algorithm based on this model must know the curvature to calculate the desired inputs v_1 and v_2. The problem then is to determine the curvature of the path based on the known or measured variables.

In the FLASH lab, a two-foot wide track circuit has been built for prototype development. Several constraints have been placed on the path configuration. One is that the path be continuous. Another is that the path be either straight or a curve of known constant radius. A sample path showing these constraints is shown in Fig. 4.1. This sample path is made up of straight sections and curves of two different radii. The resulting curvature profile is shown in Fig 4.2.

From the previous chapter, $c(s)$ is defined as

$$c(s) = \frac{d\theta_t}{ds}$$

Therefore, if the path is turning left, $c(s)$ is positive and if the path is turning right, $c(s)$ is negative. The magnitude of $c(s)$ is $\frac{1}{R}$, where R is the radius of the circle describing the curve.

As a result the constraints, the curvature of the path as a function of distance is discontinuous and piecewise constant. The derivative of $c(s)$ with respect to distance is zero except for those locations where the curvature changes. There, the derivative is infinite. Therefore, the following assuption is made

$$c'(s) \quad = \quad 0$$

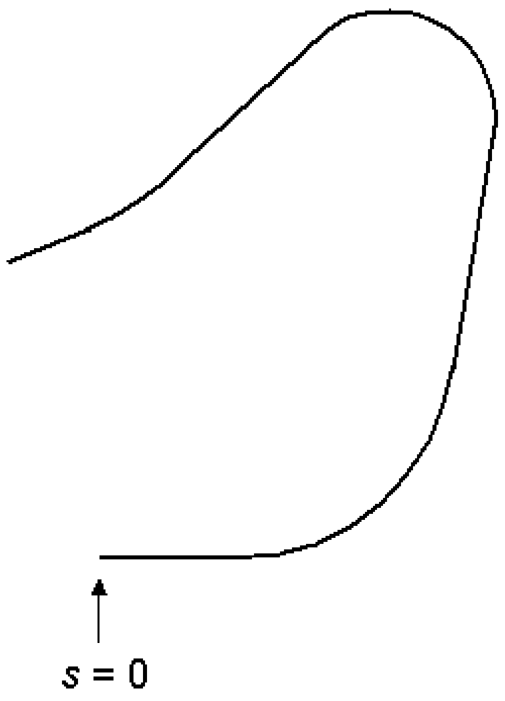

Figure 4.1: A sample path showing the constraints.

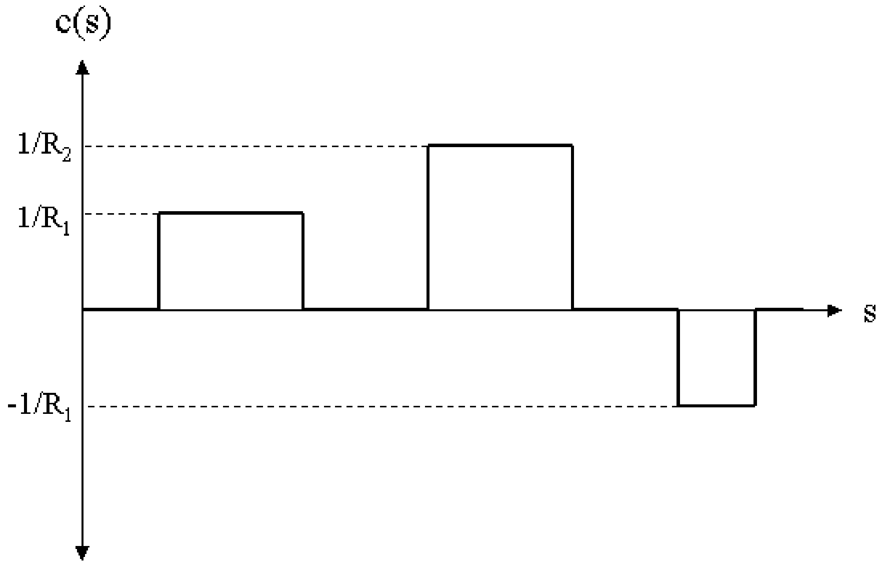

Figure 4.2: The curvature of the path in Fig. 4.1 with respect to the path length, s.

$$\begin{aligned} c''(s) &= 0 \\ &\vdots \end{aligned}$$

with $c'(s)$ denoting the derivative of c with respect to s. The derivatives are taken to be zero with disturbances where the curvature changes.

The curvature of the path is known a priori; the track is built using pieces with known radii. Therefore, the estimation result can be used to select the actual value. In other words, the calculated curvature need not be used for $c(s)$ in the state equations and controller. Rather, the actual curvature value can be selected based on the outcome of the estimation.

With the path configuration constraints defined, three methods of curvature estimation are now presented.

4.1 Estimation Methods

4.1.1 Estimation Based on the Steering Angle ϕ

The first method of estimating $c(s)$ is based solely on the steering angle, ϕ. At steady state, the car's steering wheels turn with the curves of the path. This method simply estimates the curvature using the steering wheels' angle.

If the front wheels are fixed at a certain angle, the car will describe a circle of a certain radius. Using (3.9), a MATLAB simulation was used to find the radius, R, described for several values of ϕ. It was found that the relationship between the circle's curvature, $c(s) = \frac{1}{R}$, and ϕ was nearly a straight line. So the relationship between $c(s)$ and ϕ was approximated to be

$$c(s) = \alpha + \beta\phi \tag{4.1}$$

where α and β were determined using the method of least squares to fit a line to the data. The sign of $c(s)$ is the same as ϕ.

To make this method more robust to noise, the value of ϕ used in (4.1) can be averaged over several sample periods. By averaging ϕ, this method provides a good estimate even if the car is oscillating about the desired path. However, (4.1) will work only if the car is generally following the desired path.

4.1.2 Estimation Based on the Vehicle Kinematics

The second method of estimating the curvature is based on the vehicle kinematics. If all the variables in (3.10) are known or can be measured, the equation can be solved for $c(s)$.

The third equation in (3.10) is

$$\dot{\theta}_p = \frac{v_1 \tan\phi}{l} - \frac{v_1 c(s) \cos\theta_p}{1 - dc(s)} \tag{4.2}$$

This equation can be rearranged as

$$c(s)\left[v_1 \cos\theta_p + \frac{v_1 d \tan\phi}{l} - \dot{\theta}_p d\right] = \frac{v_1 \tan\phi}{l} - \dot{\theta}_p \tag{4.3}$$

which is linearly parameterizable in $c(s)$. This can be rewritten in the following form:

$$y = wa \tag{4.4}$$

where

$$\begin{aligned} y &= \frac{v_1 \tan\phi}{l} - \dot{\theta}_p && (4.5)\\ w &= v_1 \cos\theta_p + \frac{v_1 d \tan\phi}{l} - \dot{\theta}_p d && (4.6)\\ a &= c(s) && (4.7) \end{aligned}$$

Knowing w and y, a can be obtained using a least squares estimator. We want to find the $\hat{a}$ that minimizes J where

$$J = \int_0^t (y - w\hat{a})^2 dr \tag{4.8}$$

Making $\frac{\partial J}{\partial \hat{a}} = 0$ gives

$$\left[\int_0^t w^2 dr\right] \hat{a} = \int_0^t wy dr \tag{4.9}$$

Differentiating gives an update equation for $\hat{a}$:

$$\dot{\hat{a}} = -Pwe \tag{4.10}$$

where

$$\begin{aligned} P &= \frac{1}{\int_0^t w^2 dr}\\ e &= w\hat{a} - y \end{aligned}$$

and w is defined in (4.6).

We can make the equation for P iterative by using the following update equation.

$$\dot{P} = -P^2 w^2 \tag{4.11}$$

where P is initialized to some large value.

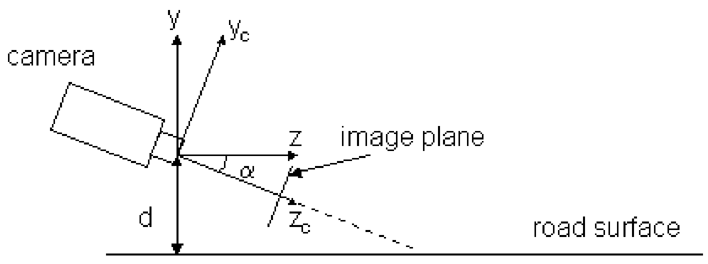

Figure 4.3: Side view of the car's camera configuration.

4.1.3 Estimation Using Image Processing

In addition to the constraints on the path's curvature, the track itself consists of a black surface with a white line. The white line is the path that the car is to follow. This scheme allows for fairly easy processing to be performed on images of roadway. This section describes the image processing methods used to estimate the curvature

On the FLASH car, a camera is mounted in such a way as to capture an image of the road directly in front of the car. The configuration of the camera is shown in Fig. 4.3. The car's frame is given by (x, y, z) and the camera's frame by (x_c, y_c, z_c). One sample image is shown in Fig. 4.4. It is assumed that in the camera's field of view, the track is a plane and perpendicular to the car's y-axis. The problem is to determine the radius of the curve in the car's frame of reference. Then, the curvature is found by taking the reciprocal of the radius. Based on this value, the actual curvature can selected to be used in the control algorithm as the curvature of the path at that point.

Edge Detection

The first task is to find the location of the road's centerline in the image plane. This section describes the method used to accomplish this task.

The images obtained from the camera contain a white curve against a black background. The roadway was designed so that the transition between the black background and the white centerline gives the highest contrast possible. Also, the camera is oriented on the car so that the image plane consists of mostly the roadway (as opposed to the scenery on the side of the road).

To locate the white curve in the image, the vertical Sobel operator shown in Fig. 4.5 was used. Only vertical edges were found because it is assumed that in all the images, the white centerline is moving away from the car rather than perpendicular to it. Fig. 4.6 shows

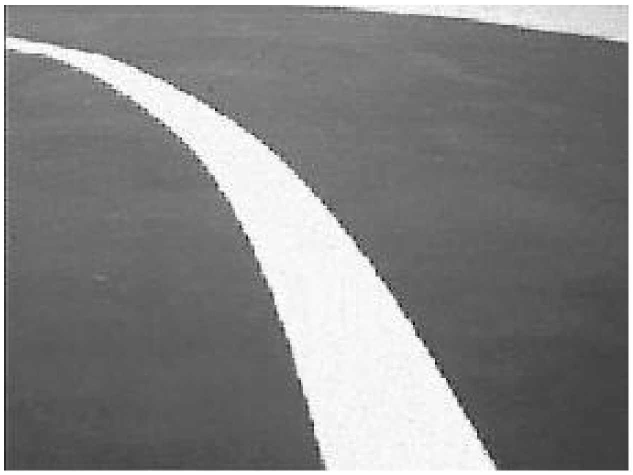
Figure 4.4: A sample image obtained from a camera mounted on the car.

-1	0	1
-2	0	2
-1	0	1

Figure 4.5: The vertical Sobel mask applied to the roadway images to find the location of the white centerline.

the result of this Sobel operator applied to the middle row of the sample image. The Sobel operator gives a positive result when the image transitions from dark to light, and a negative result when the road transition from light to dark. So the most positive result for a given row is assumed to be the left edge of the white centerline and the most negative result the right edge. The location of the white centerline in a row of the image is taken as the midpoint between these two edges.

The above edge detection algorithm is applied to the entire image. The result is that the column location of the white line is known for each row in the image. However, the equations of transformation given in the next section require that the locations in the image plane, (x', y'), be in real world units such as inches, rather than pixels. So the row and column locations, r and c, must be converted as follows:

$$x' = (c - \tfrac{columns}{2})k \tag{4.12}$$
$$y' = (r - \tfrac{rows}{2})k \tag{4.13}$$

where *columns* is the total number of columns in the image, *rows* is the total number of rows, and k is the pixel size as given in the camera's specifications.

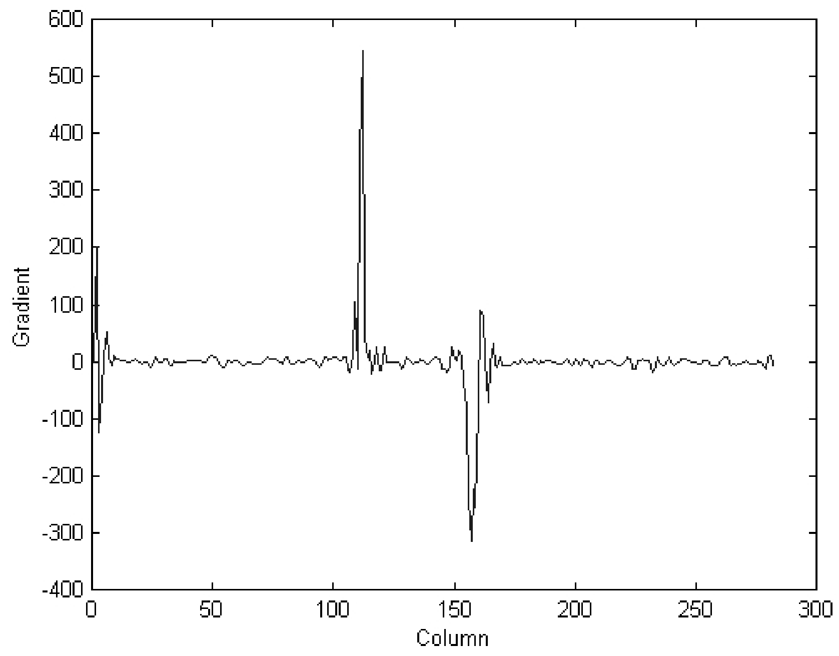

Figure 4.6: The result of the Sobel operator applied to the middle row of Fig. 4.4.

Coordinate Transformations

The relationship between the car's frame of reference and the camera's is shown in Fig. 4.3. The height of the camera, d, and its tilt angle, α, are known. The tranformation from the car's frame to the camera's frame is simply a rotation about the x-axis and is given as follows:

$$x_c = x \tag{4.14}$$
$$y_c = y\text{cos}\alpha + z\text{sin}\alpha \tag{4.15}$$
$$z_c = -y\text{sin}\alpha + z\text{cos}\alpha \tag{4.16}$$

However, y is fixed at $-d$.

The coordinates in the image plane are then given by:

$$x' = f\frac{x_c}{z_c} \tag{4.17}$$
$$y' = f\frac{y_c}{z_c} \tag{4.18}$$

where f is the focal length of the camera.

Combining the two tranformations gives a point in the image plane in terms of the car's coordinates.

$$x' = f\frac{x}{d\sin\alpha + z\cos\alpha} \tag{4.19}$$

$$y' = f\frac{-d\cos\alpha + z\sin\alpha}{d\sin\alpha + z\cos\alpha} \tag{4.20}$$

Now given a point (x', y') in the image plane, (4.19) and (4.20) can be solved for x and z. Using (4.20), z can be found as

$$z = \frac{-d(f\cos\alpha + y'\sin\alpha)}{y'\cos\alpha - f\sin\alpha} \tag{4.21}$$

Substituting (4.21) into (4.19), x becomes

$$x = \frac{x'}{f}(d\sin\alpha + z\cos\alpha) \tag{4.22}$$

So, given a point in the image plane, its location in the car's frame of reference can be recovered if d, α, and f are known.

Calculation of the Radius

With the points known in the car's frame of reference and working under the assumption that the real world curves are of constant radius, the task now is to calculate that radius.

If three points on the circumference of a circle are known as shown in Fig. 4.7, the radius of that circle is given by the following:

$$R = \frac{a}{2\sin A} \tag{4.23}$$

By choosing three different rows in the image and finding the location of the white centerline in those rows, the (x, z) coordinates in the car's frame of reference can be found using the transformation described above. With three (x, z) points known, the Euclidean distances between them, a, b, and c, can be found. The law of cosines is then used to find the angle A.

The problem then is to find three sample points to use. Two different approaches were tried and they are described below. The performance of these two methods is described in the Results section.

Fixed Row Method With this method, three rows in the image were chosen a priori to be the location of the points on the circumference of the circle. Different rows were tried on several images and the ones that gave the best results overall were the ones used for the final implementation.

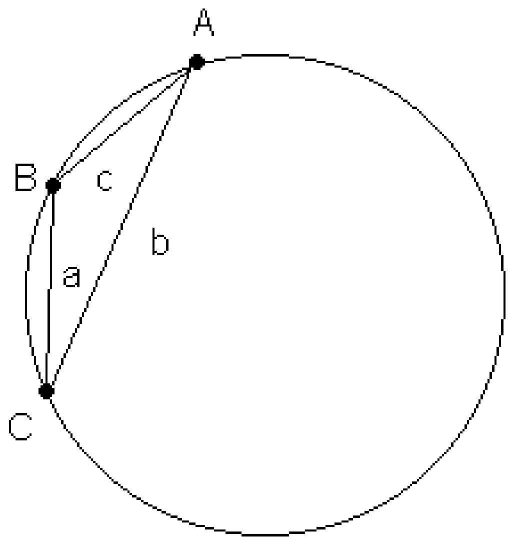

Figure 4.7: A triangle circumscribed by a circle of radius R. The triangle can be described by angles A, B, and C and side lengths a, b, c.

Variable Row Method In many of the images, the white centerline curves away and out of the image before reaching the top row. See Fig 4.4. The variable row method tries to take advantage of the entire useful picture.

Once edge detection has been performed on the entire image and the location of the centerline found for each row, these locations are checked for large changes from row to row. If one centerline location differs from the one in the previous row by more than some threshold (5 pixels, for example), it is assumed that the centerline has been lost in noise. The pixel threshold value was chosen based upon the curvature of the actual roadway geometry. It is known that the real roadway will not produce a change of more than 5 pixels per row in the image.

4.2 Simulation Results

Each of the above estimation methods was simulated using MATLAB. This section describes the performance of each method in simulation. A MATLAB program environment has been created to simulate the car using the kinematic model given in (3.9). The simulation was run using the controller as given in (3.20). The simulation environment is detailed in Chapter 5.

A path was created in MATLAB to simulate the actual track in the FLASH lab. This path

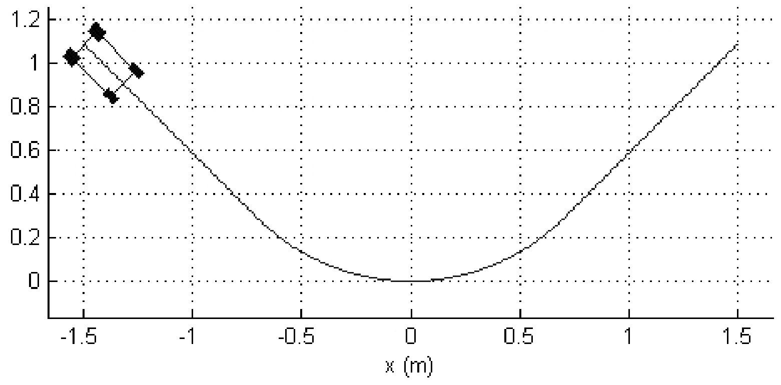

Figure 4.8: The path generated using MATLAB.

consists of a straight section, a curve of radius 1m, followed by another straight section. See Fig. 4.8. The curvature profile is shown in Fig. 4.9. A simulated car was run on the track using the result of the curvature estimate algorithm. Because the curvature of the path was known to be 0 or 1, these actual values were used in the controller. The output of the estimator was utilized to determine which curvature value to use.

4.2.1 Steering Angle Estimator

First, the curvature estimate based on ϕ was tried. The curvature was calculated using (4.1) with $\alpha = -0.1599$ and $\beta = 4.8975$. For filtering, ϕ was averaged over 10 sample periods. A threshold of 0.5 was used so that if the calculated curvature was less than 0.5, a $c(s)$ value of 0 was used. If the calculated curvature was greater than this threshold, $c(s)$ was set to 1.

The car was initially placed so that it was starting on the straight section of the path and oriented so that d and θ_p were both zero. Because of this starting location, there were no transients while the car corrected itself. Fig. 4.10a shows the estimated curvature and the actual curvature plotted together. The actual curvature is shown by a dotted line. Fig. 4.10b shows the thresholded estimate together with the actual curvature. The thresholded value is slightly delayed with respect to the actual curvature.

Next, the car was placed on the path so that θ_p was initially nonzero. This resulted in some transients while the car centered itself on the path. The estimate of the curvature is shown in Fig. 4.11a. The value used for $c(s)$ is shown as the solid line in Fig. 4.11b. Because of the transients, this situation caused $c(s)$ to erroneously have a value of 1 well before the car reached the curve. This method gave a more accurate $c(s)$ during steady-state, showing only a slight delay as before.

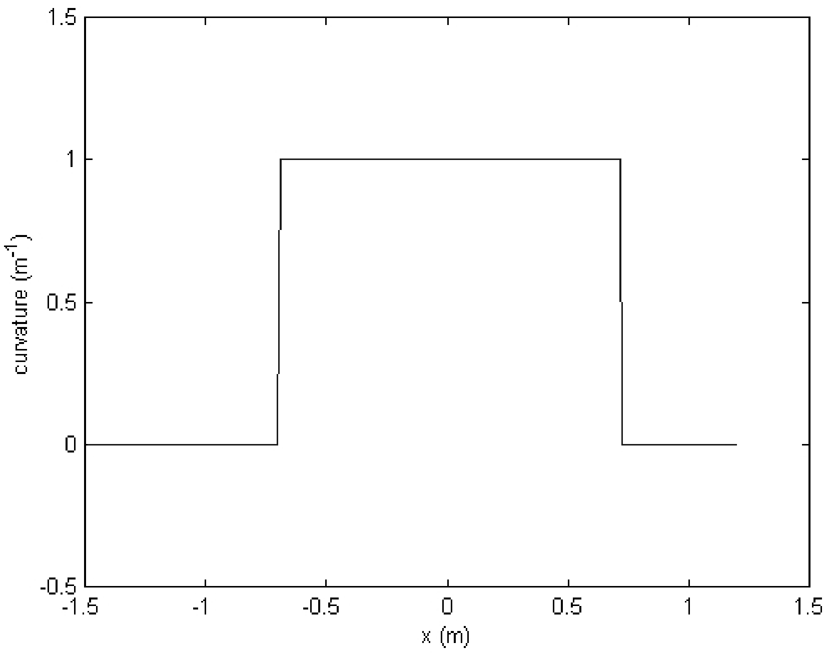

Figure 4.9: The curvature profile of the path in Fig. 4.8.

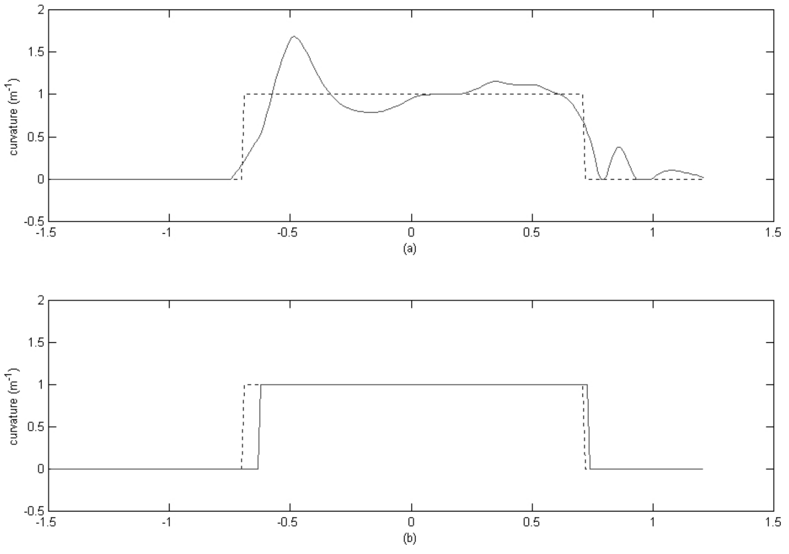

Figure 4.10: The curvature estimated using only the steering angle, ϕ, with θ_p initially zero.

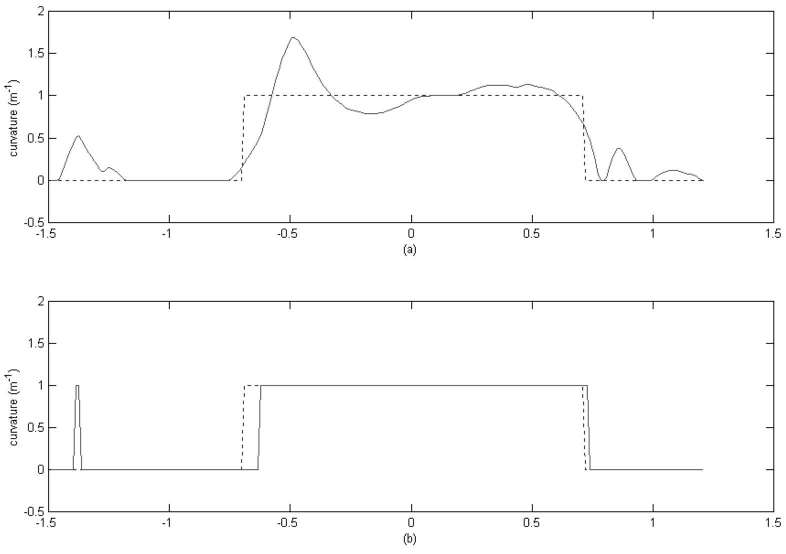

Figure 4.11: The curvature estimated using only the steering angle, ϕ, with θ_p initially nonzero.

4.2.2 Model Estimator

Next, the curvature estimate based on the kinematic model as described in Section 4.1.2 was simulated. This method used the same initial conditions as the ϕ estimate method.

First the car was placed on the path so that d and θ_p were both zero initially. The resulting estimate of the curvature is shown in Fig. 4.12a. This estimate was thresholded as before to determine the value for $c(s)$ as 0 or 1. However, to give better performance, hysteresis was used. On the rising edge, the threshold was 0.9; while on the falling edge, the threshold was 0.1. The resulting value for $c(s)$ is shown in Fig. 4.12b. This method seemed to anticipate the curve and thus performed better than the ϕ estimate method.

As with the ϕ estimate method, this method was also tested with a nonzero θ_p. The resulting estimate is shown in Fig. 4.13a. The same hysteresis thresholding was applied in this case and the resulting values for $c(s)$ are shown in Fig. 4.13b. This method did not give erroneous results while the car corrected itself on the path.

Another approach was tried with the dynamic curve estimate. After applying the update equation, (4.10), $\hat{a}$ was thresholded. If it was greater than 0.5, it was set to 1. If it was less than 0.5, $\hat{a}$ was set to 0. The curvature value for $c(s)$ was then $\hat{a}$. The resulting curvature for both initial conditions is given in Fig. 4.14 and Fig. 4.15. This method performed very well. The estimated curvature matched the actual curvature going from the straightaway to the curve. Coming out of the curve, there was only a slight delay before the estimator

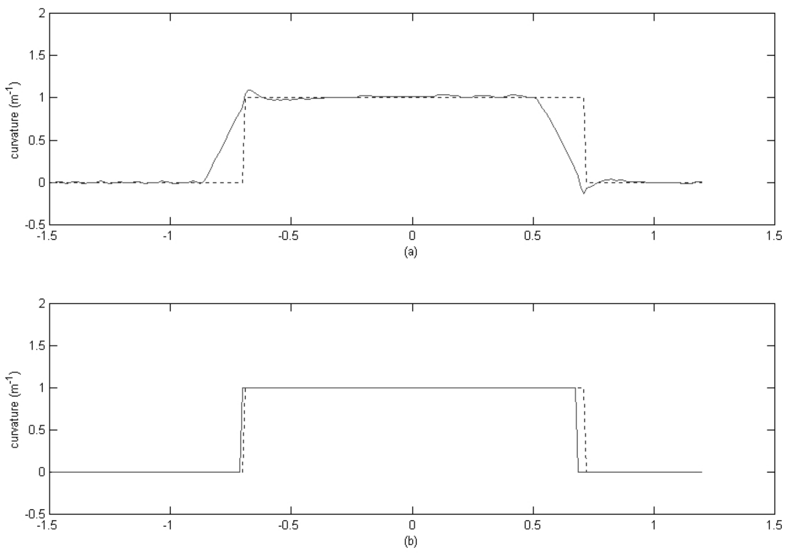

Figure 4.12: The curvature determined by using the model estimator with θ_p initially zero.

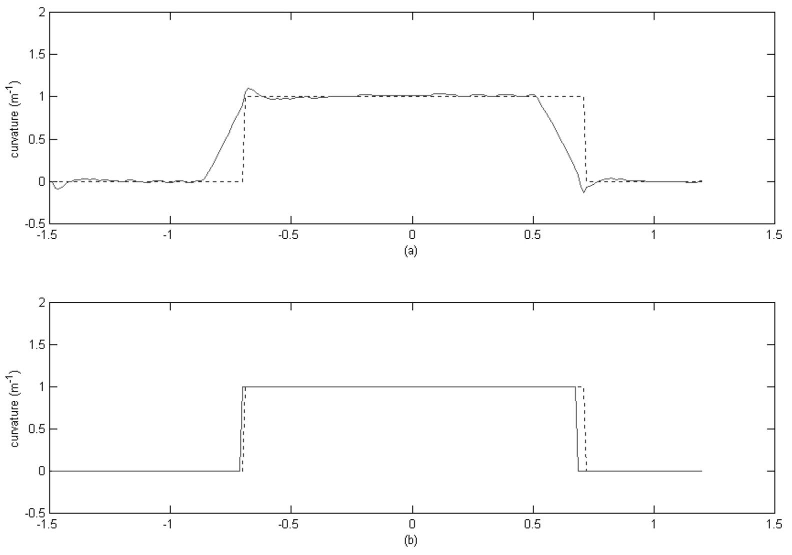

Figure 4.13: The curvature determined by using the model estimator with θ_p initially nonzero.

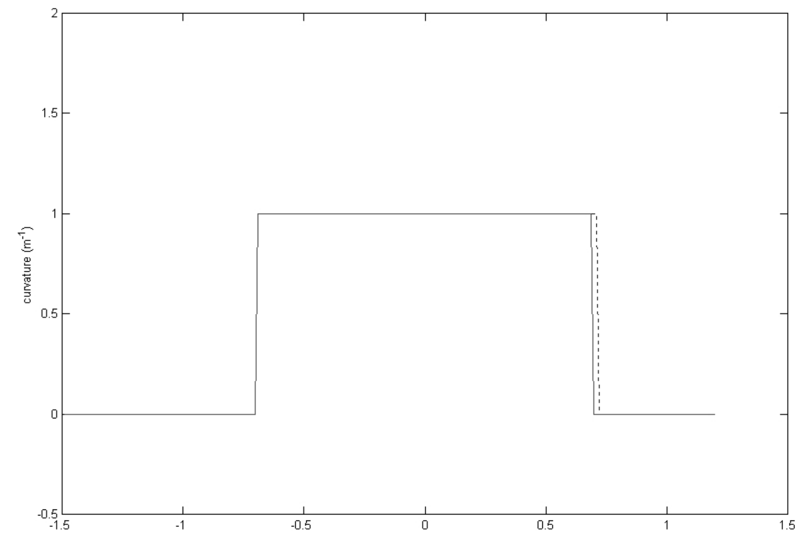

Figure 4.14: The curvature determined by thresholding $\hat{a}$ with θ_p initially zero.

determined the correct value for $c(s)$.

4.2.3 Image Estimator

The simulation environment used for the above simulation results is based on the kinematic model for the car only and does not allow for simulation of the image processing algorithm to be embedded into the controller. Thus, a separate MATLAB environment was created to determine the curvature of actual images of the FLASH roadway. The image processing algorithms described in Section 4.1.3 were successfully applied to several sample images with curves of known radii. Additionally, there were some images for which the algorithms failed.

First, the image shown in Fig. 4.4 is discussed. This image contains a curve with a radius of 35 inches. The curve transformed into the car's (x,z) frame is shown in Fig. 4.16. The circles on the plot indicate the points selected using the fixed row method. The rows used were 50, 200, and 250 (as measured from the bottom of the image). The radius resulting from these points was 34.1509 inches. The sample points chosen by the variable row method are illustrated in Fig. 4.17. The radius was calculated to be 36.0629 inches in this case.

Next, the image of Fig. 4.18 was used. This image also has a 35 inch radius. The fixed row method failed with this image because the centerline could not be detected in one of the rows as shown in Fig. 4.19. The radius was calculated as 5.2670 inches. The variable row method chose sample points as in Fig. 4.20. Using these rows, the resulting radius was

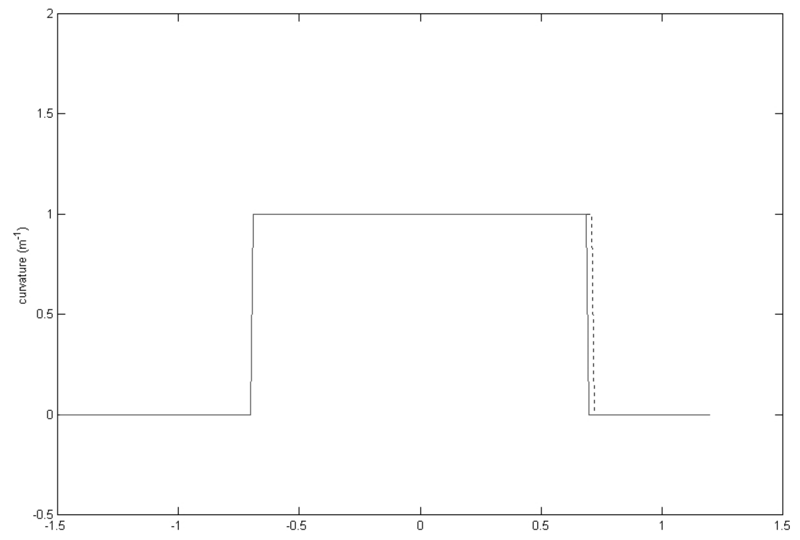

Figure 4.15: The curvature determined by thresholding $\hat{a}$ with θ_p initially nonzero.

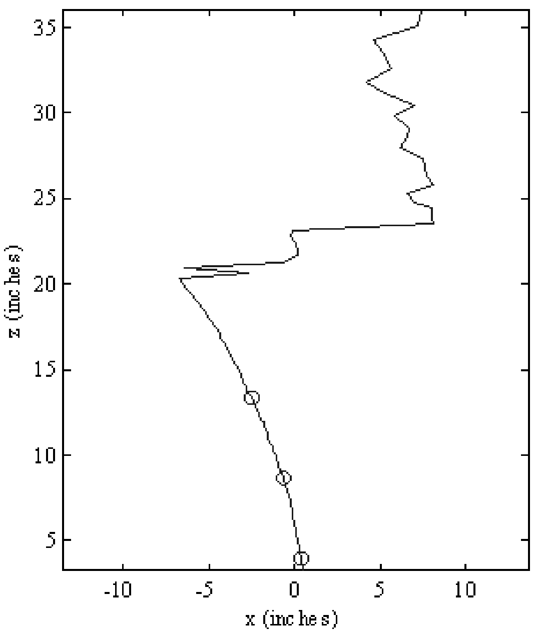

Figure 4.16: The curve of Fig. 4.4 tranformed into the car's (x,z) coordinates using the fixed row method.

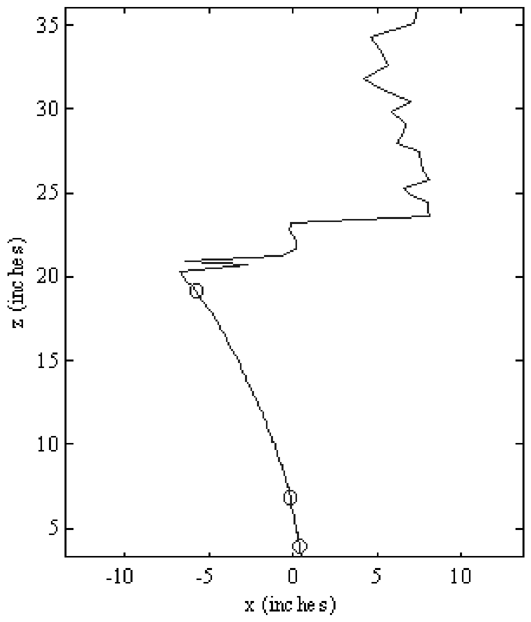

Figure 4.17: The curve of Fig. 4.4 tranformed into the car's (x,z) coordinates using the variable row method.

26.0307 inches.

Finally, two images were tested that contained a straight section of roadway. These two images were taken from slightly different points of view and are shown in Fig 4.21. These images are almost ideal in that the centerline is very prominent in every row of the image. The radius of curvature in this case is approaching infinity. Both methods calculated the radius of curvature to be more than 1000 inches. This easily identifies the images as being from the straight section of the path.

4.2.4 Method Comparison

All methods were able to successfully determine the curvature of the path. The least robust method was the ϕ estimator in the presence of transients. The robustness to these transients can be improved if the steering angle, ϕ, is averaged over a greater number of sample periods. More averaging will, however, degrade the overall performance of this method. This is because the estimator would react more slowly to changes in curvature. This method also requires more memory than the model estimator because the previous steering angles must be stored.

The model estimator, in both its forms, performed very well. It was not susceptible to

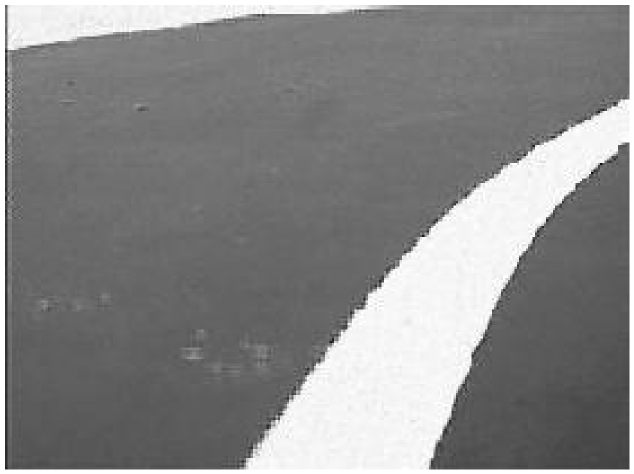
Figure 4.18: Another sample image to which the algorithm was applied.

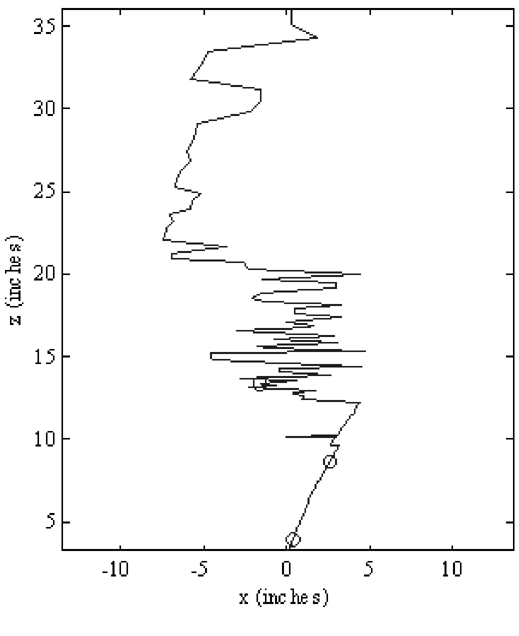

Figure 4.19: The transformation of Fig. 4.18 using the fixed row method.

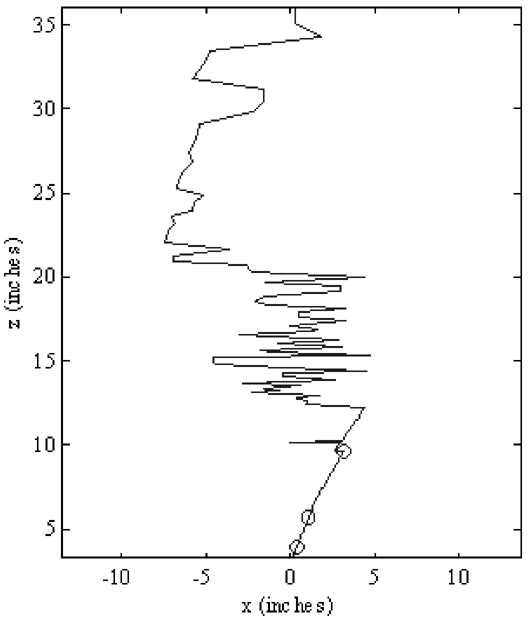

Figure 4.20: The transformation of Fig. 4.18 using the variable row method.

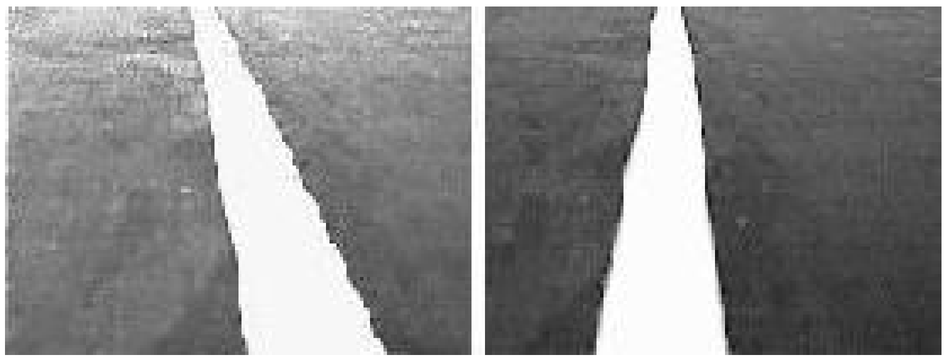

Figure 4.21: Two sample images of a straight section of the path, taken from different viewpoints.

transients as the ϕ estimator was. Also, it was able to anticipate the upcoming change in curvature due to the dynamics of the system. Thus, there was very little delay in the curvature transitions. This method also requires less memory. Only $\dot{\theta_p}$ and P must be numerically computed. In this implementation, a first order approximation was used so only one previous value was needed for both θ_p and P.

There was very little difference in the performance of the car between these two methods in simulation. The car stayed on the path, even with an erroneous $c(s)$ provided by the ϕ estimator in one case. However, in the hardware implementation, these methods performed very differently. Chapter 6 discusses their performance on the FLASH car.

Tests with several sample images indicate that the image processing algorithm works well for images in which the centerline is prominent and located near the center of the image. The algorithm was less accurate for images in which the centerline came in at an angle or was located in one small part of the image away from the center. Poorer results were obtained when calculations were made using pixels located away from the center of the image. This may be due to distortions in the camera's lens or error in the measurement of camera's height and angle.

The edge detection algorithm was able to locate the centerline in most of the image area, until the topmost part of the image in most cases. Near the top of the image, the centerline was less prominent and was lost in the background.

The two methods of selecting sample rows gave similar results in images that were "nice". That is, images in which the sample rows contained the actual centerline. If one of the preselected sample rows was in an area of the image where the centerline could not be detected, the algorithm failed. The variable row method, however, could detect where the centerline was lost and adjust the sample rows to take advantage of the full useful image area.

There is a tradeoff between the two methods of sample row selection. The fixed row method is much faster because it only requires that the Sobel operator be applied to three rows of the image. However, it requires that the algorithm be tried several times with different rows to determine which rows give the best results. The major drawback to this method is that it is not robust. If the image contains a curve which cannot be detected in those sample rows, the algorithm will provide inaccurate results.

The variable row method is much more computationally expensive. It requires that the Sobel operator be applied to the entire image. In addition, it must check the results of the edge detection to determine where the centerline has been lost. However, this method is much more robust and can produce good results over a greater variety of images.

The algorithms applied here did not give exact results. In some cases the error was quite high. However, in the implemetation on the FLASH vehicle, the estimated curvature will not be used in the controller. Because the track has been built specifically for this application, the radii of the curves are known a priori. So the calculated curvature can be used to determine on which of the curves the car is located. The result of this algorithm need only help determine which curvature to use in the controller.

4.3 Implementation Issues

The methods described above differ greatly in the amount of hardware and programming required on the car. Those issues are important because all processing is done on the car and power consumption must be minimized to allow for longer runtime.

The ϕ estimator and the model estimator require no additional hardware. The ϕ estimator only requires knowledge of the steering angle for use in (4.1) as α and β are constants determined offline. Some additional memory is necessary if ϕ is to be averaged over several sample periods. The model estimator requires the implementation of only (4.10) for which v_1, θ_p, ϕ, and d are already known for use by the controller.

The image processing method of estimation is by far the most hardware and software intensive. It requires the use of an on-board camera. This camera must be interfaced with the processor to store images. Edge detection must be performed on the image and several more calculations done to recover the radius of curvature. In addition, this estimation method is very sensitive to errors in measurement of the tilt angle, α, and camera height, d. Errors are also introduced because this method assumes the roadway is a plane perpendicular to the car's vertical axis. The actual track however, is not flat but contains hills for the car to navigate.

Because of all these issues, the image processing algorithm is the most challenging of the estimators to implement on the FLASH car. The hardware and software implementation of these algorithms is discussed in Chapter 6.

Chapter 5

Simulation Environment

5.1 Simulation Overview

This chapter describes the MATLAB simulation environment used for developing and testing the control algorithms used on the FLASH vehicle. This simulation provides, as closely as possible, a program environment similar to that used by the FLASH vehicle. The car's methods of measurement and calculation are the same in the simulation as in the hardware. However, in the simulation, an ideal path is created for the car to follow, the car's movement is given by the kinematic model derived in Chapter 3, and the car's movement is shown using the MATLAB animation toolbox.

A flowchart of the program is shown in Fig. 5.1. First, the initialization involves creating the car and path for animation and placing the car on the path. Next, the car's position on the path is determined and the values needed by the controller are calculated. With these values known, the controller then calculates the necessary velocity and steering inputs to make the car follow the path. These inputs are used in the kinematic model to update the car's position and the animation is then updated to show the car's new location. These steps are repeated until the end of the simulation is reached.

The next section gives the details of how each step is performed.

5.2 The Simulation Program

5.2.1 Path Creation

As stated in the previous chapter, there are several constraints on the construction of the track. Because of these constraints, the path is assumed to be continuous and the curvature

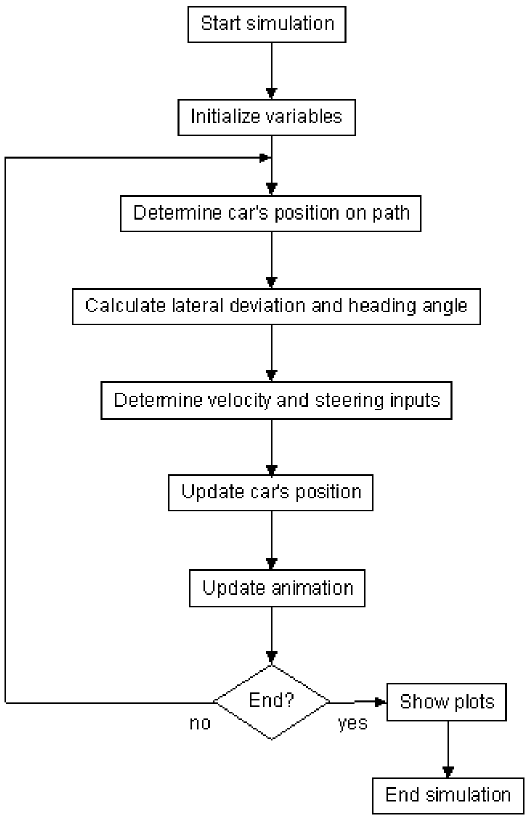
Start simulation
Initialize variables
Determine car's position on path
Calculate lateral deviation and heading angle
Determine velocity and steering inputs
Update car's position
Update animation
End?
no
yes
Show plots
End simulation

Figure 5.1: Flowchart for the MATLAB simulation program.

is assumed to be piecewise constant. In addition, it is known a priori that the track is made up of straight sections and curves of constant radius.

The simulation has been set up to create a path that contains a straight segment, followed by a curve, followed by another straight segment. The path is defined in the (x, y) global coordinates and its length and the radius of curvature can be defined by the user in the initialization file. The path can be defined using the following equation.

$$y = \begin{cases} -x + r(1 - \frac{2}{\sqrt{2}}), & x < -\frac{r}{\sqrt{2}} \\ -\sqrt{r^2 - x^2} + r, & -\frac{r}{\sqrt{2}} \le x \le \frac{r}{\sqrt{2}} \\ x + r(1 - \frac{2}{\sqrt{2}}), & x > \frac{r}{\sqrt{2}} \end{cases} \tag{5.1}$$

Here, r is the radius of the curved section of the path. Using $r = 1$ creates the path shown in Fig. 4.8.

5.2.2 Error Calculation

With the car on the path, the controller must know where the car is located and how it is oriented. On the FLASH car, there are sensors on the front and rear which detect the presence of the line beneath the car. In the simulation, the distance between the path and the car is found. Then this value is converted to the same representation as on the actual vehicle. Finally, the sensor data is converted to an actual distance.

Calculating the Actual Error

The vehicle's position is known, (x_0, y_0), as well as its orientation θ and steering angle ϕ. From this, the position of the front sensor can be found as follows.

$$x_1 = x_0 + l\cos\theta \tag{5.2}$$

$$y_1 = y_0 + l\sin\theta \tag{5.3}$$

Knowing two points along the center axis of the vehicle, (x_0, y_0) and (x_1, y_1), the slope of LINE 1 in Fig. 5.2 can be found as $\frac{y_1 - y_0}{x_1 - x_0}$. Since LINE 1 and LINE 2 are perpendicular, the slope of LINE 2 is $\frac{-(x_1 - x_0)}{y_1 - y_0}$. Now the slope of LINE 2 and a point on it are known, so its equation is

$$y = -m(x - x_1) + y_1 \tag{5.4}$$

where $m = \frac{x_1 - x_0}{y_1 - y_0}$.

Next, the point (x_2, y_2) must be determined by finding the intercept of LINE 2 and the path. Setting the right side of (5.1) equal to the right side of (5.4) yields the following.

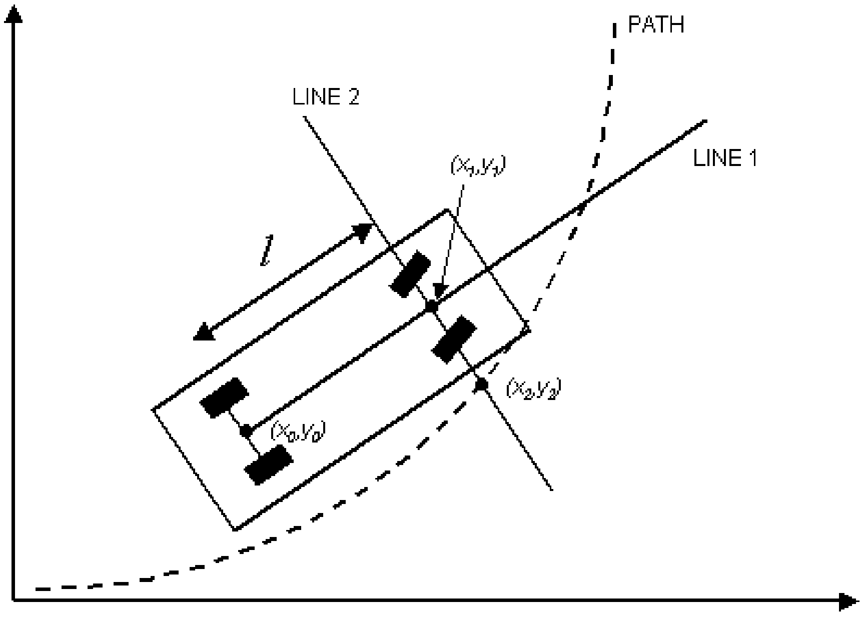

Figure 5.2: Errors of the path following vehicle.

For $x_1 < \frac{-r}{\sqrt{2}}$,

$$x_2 = \frac{mx_1 + y_1 - r(1 - \frac{2}{\sqrt{2}})}{m-1} \tag{5.5}$$

$$y_2 = -x_2 + r(1 - \frac{2}{\sqrt{2}}) \tag{5.6}$$

For $\frac{-r}{\sqrt{2}} \le x_1 \le \frac{r}{\sqrt{2}}$,

$$x_2 = \frac{-b^2 \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.7}$$

$$y_2 = -\sqrt{r^2 - x_2^2} + r \tag{5.8}$$

where the sign of the square root in (5.7) is the same as the sign of m and

$$\begin{aligned} a &= m^2 + 1 \\ b &= -2m^2x_1 - 2my_1 + 2mr \\ c &= 2mx_1y_1 - 2mx_1r + y_1^2 - 2y_1r \end{aligned}$$

For $x_1 > \frac{r}{\sqrt{2}}$,

$$x_2 = \frac{mx_1 + y_1 - r(1 - \frac{2}{\sqrt{2}})}{m-1} \tag{5.9}$$

$$y_2 = x_2 + r(1 - \frac{2}{\sqrt{2}}) \tag{5.10}$$

Now that the points (x_1, y_1) and (x_2, y_2) are known, the error is the difference between them.

$$e_f = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{5.11}$$

Either the positive or negative square root is used depending on whether the path is to the left or right of the car's center. The convention used here is if path is to the right, the positive value is taken.

The error at the rear of the car, e_b, can be found using the above method but LINE 2 in Fig. 5.2 must go through (x_0, y_0).

Conversion to Sensor Representation

On the FLASH car, the line beneath the car is detected using an array of sensors. The sensors are turned on or off depending on whether the line is detected. The result is a binary representation of the line's location such as

1110011111111111

where the zeros indicate the location of the line.

The error distance calculated in the previous section must be converted to this binary representation. This is done as follows.

```
array = ones(s,1);

for k = -0.5*s:0.5*s-1
   if (d >= k*sp) & (d <= (k+1)*sp)
      array(s/2-k) = 0;
   end
end
```

Here, *s* is the number of sensors, *d* is the error distance as calculated in the previous section, and *sp* is the spacing between the sensors. If the line is outside the width of the sensors, the array is all ones.

Conversion to Distance

The binary representation must now be converted back into an actual distance for use by the controller. The following code performs the conversion.

```
error = 0;
num = 0;
val = (s+1)/2;

for k = 1:s
   if array(k) == 0
      num = num+1;
      error = error+(val-k)*sp;
   end
end

if num ~= 0
   error = error/num;
else
   error = w/2*sign(p);
end
```

where s is the number of sensors, sp is the spacing between the sensors, w is the width of the sensor array, and p is the previously calculated error value. If the array is all ones, the line is outside the range of the sensors and the error is saturated to its maximum value. The sign of the error is then assumed to be the same as p.

Because the rear sensor array is placed directly below the rear axle, the error obtained from that sensor is taken to be d, the car's lateral displacement from the path.

5.2.3 Heading Angle Calculation

The controller must know the heading angle, θ_p, the angle between the car and the path. The value can be calculated using the displacement errors at the front and rear of the car as determined above, e_f and e_b and the distance between them, l. Assuming the path directly underneath the car is straight, the heading angle is

$$\theta_p = \tan^{-1}\left(\frac{e_f - e_b}{l}\right) \tag{5.12}$$

In this equation, either the actual errors or the discretized errors can be used. However, to simulate the actual car, the discretized errors should be used.

5.2.4 Control Input Calculation

The next step in the program is to determine the steering and velocity inputs to move the car along the path. The controller must know the values for d, θ_p, and c. The values for d and θ_p are known from the previous sections. The simulation has been set up to implement the curvature estimation methods discussed in Sections 4.1.1 and 4.1.2.

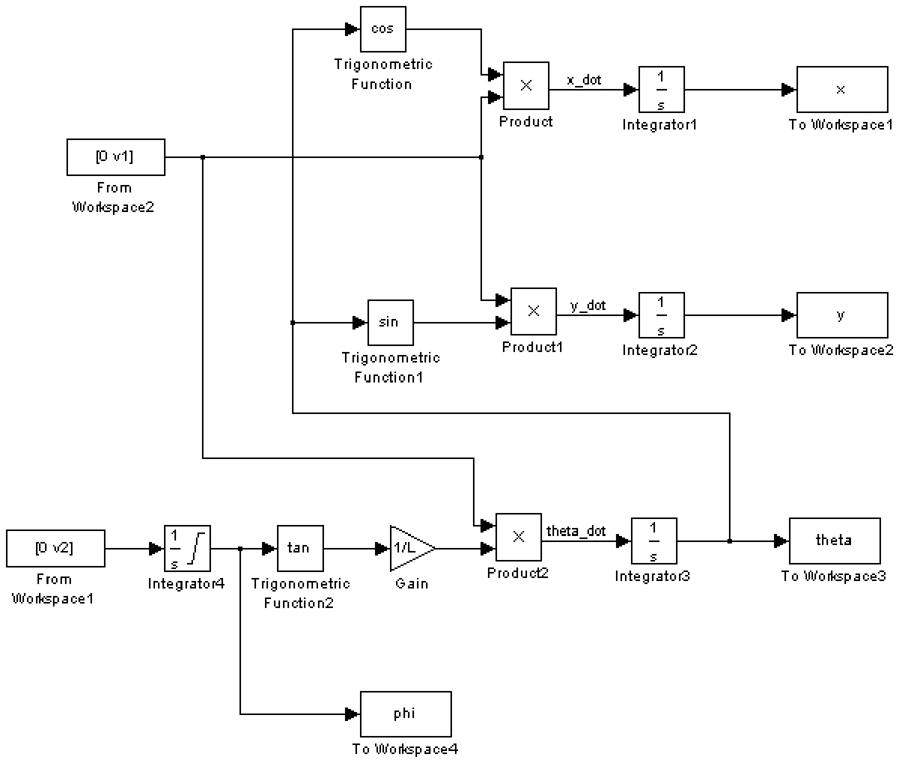

Figure 5.3: The simulink representation of the car's kinematic model.

With these values known, the states x_2, x_3, and x_4 can be calculated using (3.14)-(3.16). The controller is given by (3.20) and its output is transformed into steering and velocity inputs by (3.17) and (3.18). The most challenging aspect of the controller implementation was typing in the equations without errors.

5.2.5 Car Model

Next the movement of the car is determined over the sampling period, T. The kinematic model given by (3.9) was implemented in Simulink and is shown in Fig. 5.3. The model uses the current position (x, y, θ, ϕ) as the initial conditions and integrates to determine the car's new position after the inputs are applied for time T. It is assumed that the inputs are constant over the sampling time, as they are on the actual car.

5.2.6 Animation

Finally, the movement is animated to provide a means of viewing the car's behavior. The animation toolbox for MATLAB was used for this purpose. The animation toolbox utilizes Handle Graphics, MATLAB's object-oriented graphics system. This toolbox allows for animation of any object created in MATLAB. Complete details of this toolbox can be found in [11].

The first step in using animation is to create the car. The car is simply a rectangle with four wheels attached. The car body is defined using the *patch* command with the appropriate vertices. The wheels are defined as cylinders of necessary height and radius and rotated ninety degrees. Once the individual pieces of the car are created, they must be placed in the proper orientation using the *locate* and *rotate* commands. Finally, they are joined together using the *attach* command so that the entire car can be moved as one piece. In addition, each of the components can be moved individually.

Now with the car fully defined, it can be located and oriented anywhere on a MATLAB plot. So given the car's position from the kinematic model, the updated position is obtained by using the following commands.

```
locate(car,[x0,y0 0]);
turn(car,'z',(theta0-theta0_prev)*180/pi);
turn(car.tire_fl,'z',(phi0-phi0_prev)*180/pi);
turn(car.tire_fr,'z',(phi0-phi0_prev)*180/pi);
```

This code positions the car at (x_0, y_0) with an orientation of θ_0 and turns the front wheels by an additional ϕ_0.

5.3 Simulation Results

This section provides simulation results for the controller in varying conditions. The performance of the controllers are discussed and comparisons between them are made. The path used is the same as shown in Fig. 4.8. This controller was tested using both the actual and the discretized errors.

The input scaling controller in (3.20) was inplemented using three different forms. The first form used the actual curvature of the path. The second used the ϕ estimation method. The last used the model estimator. These methods were described in detail in Chapter 4. In that chapter, the performance of the estimation methods were discussed with respect to their accuracy in determining the curvature. In this section, the performance of the controller using these methods is discussed.

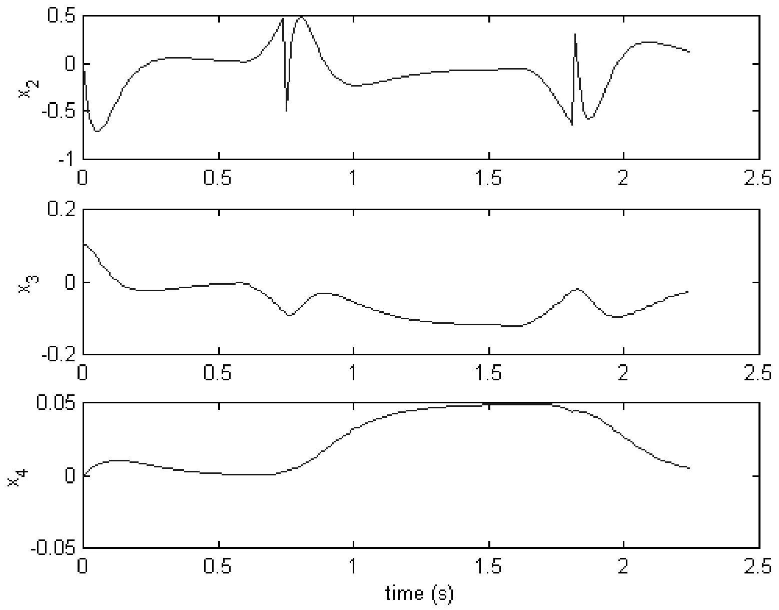

Figure 5.4: The states, x_2, x_3, and x_4, resulting from using the actual errors and curvature.

5.3.1 Control Using the Actual Curvature

First, the actual error distances as given in (5.11) and the actual curvature was used in the controller. This was possible because the path was created using (5.1) and the car's location is known. In addition, it is known in which direction along the path the car is traveling. Therefore, the curvature can be determined to be $\pm\frac{1}{R}$ or 0. Figs. 5.4-5.6 show the results of applying this controller. The car's path speed, u_1, was held constant at 1.5 m/s. The gains used were: $k_1 = \lambda^3$, $k_2 = 3\lambda^2$, and $k_3 = 3\lambda$ with $\lambda = 8$.

There are two important things to note about the performance of this controller. The first is that even though u_1 is constant, v_1 does not remain constant. u_1 is transformed into v_1 by taking into account the car's state and also the curvature. The result is that the car's slows down in the curve.

The second thing to note is that there are spikes in the steering control input, u_2. These result from the spikes that are present in x_2. These spikes occur exactly where the path changes curvature. At these points, the derivative of the curvature is infinite. However, in the implementation, the derivatives of curvature are set to zero. The discrepancy is seen here as a disturbance in the system.

Next, the same controller was used with the discretized errors. It was assumed that there were twelve sensors spaced 0.2 inches apart. The same gains and initial conditions were used

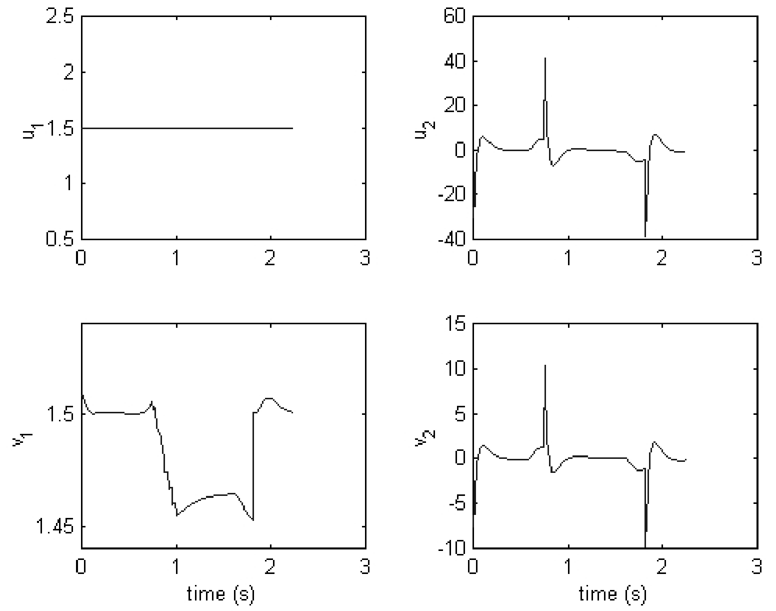

Figure 5.5: The control inputs, v_1 and v_2, resulting from using the actual errors and curvature.

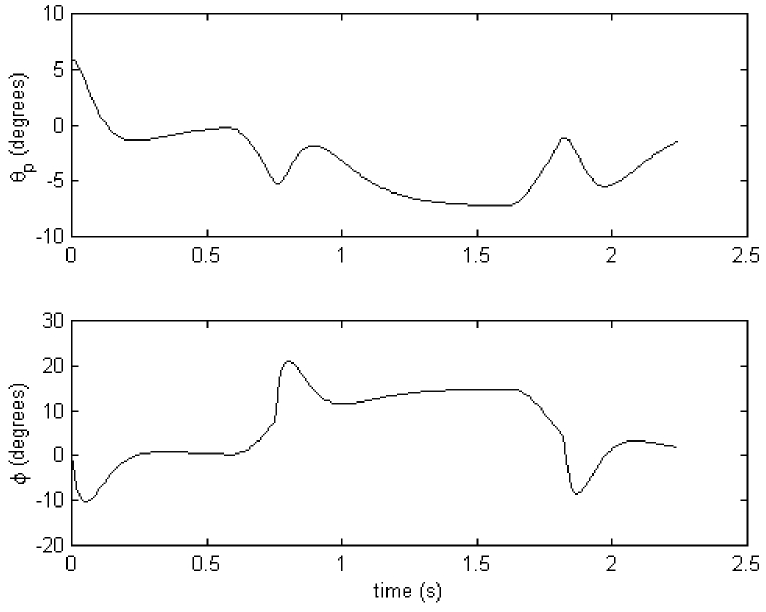

Figure 5.6: The heading angle, θ_p, and steering angle, ϕ, resulting from using the actual errors and curvature.

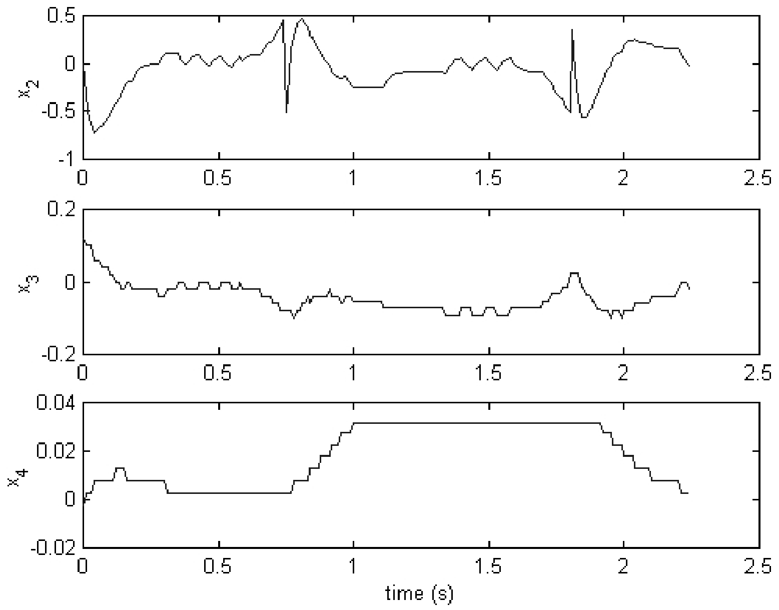

Figure 5.7: The states, x_2, x_3, and x_4 resulting from using the discretized errors.

as above. Figs. 5.7-5.9 show the results of discretization. Again the actual curvature is used.

As is to be expected, using the discretized errors caused the control input, and thus the steering angle ϕ, to become much choppier. This resulted in a less smoothe trajectory being traversed by the car.

5.3.2 Control Using the ϕ estimator

Next, the simulation was run using the ϕ estimator as described in Section 4.1.1. The results of this algorithm are shown in Figs. 5.10-5.12.

In Fig. 5.11, spikes are seen where the path's curvature does not transition between 0 and $\frac{1}{R}$. Comparing this result with Fig. 4.11 provides an explanation for this. As seen before, the ϕ estimator produced a false curvature while it was on the straightaway due to transients while the car corrected itself. This can also be seen in the controller in Fig. 5.11. The two spikes occur in u_1 as it is starting out because the curvature is incorrectly estimated to be $\frac{1}{R}$. The first spike occurs at the 0 to $\frac{1}{R}$ transition, while the second occurs at the $\frac{1}{R}$ to 0 transition. After the initial transients, the controller performed similar to the above controller with which the actual curvature was used.

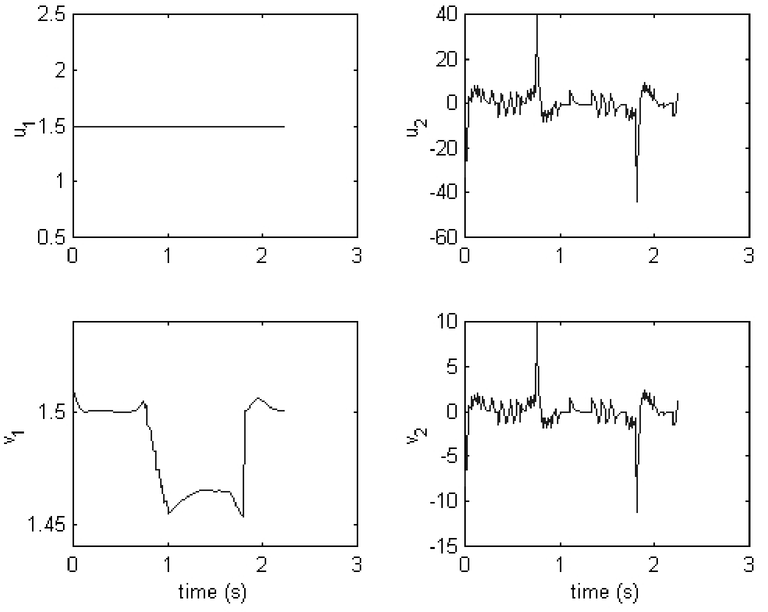

Figure 5.8: The control inputs, v_1 and v_2, resulting from the discretized errors.

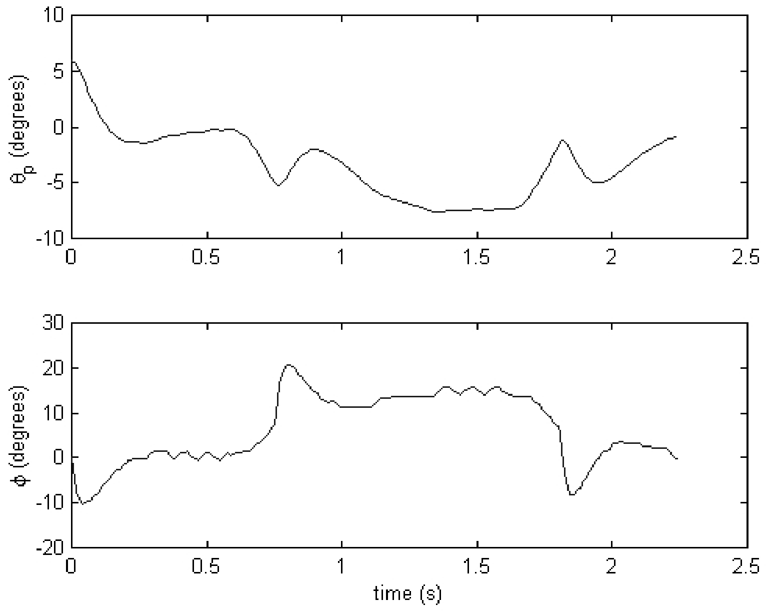

Figure 5.9: The heading angle, θ_p, and the steering angle, ϕ, resulting from using the discretized errors.

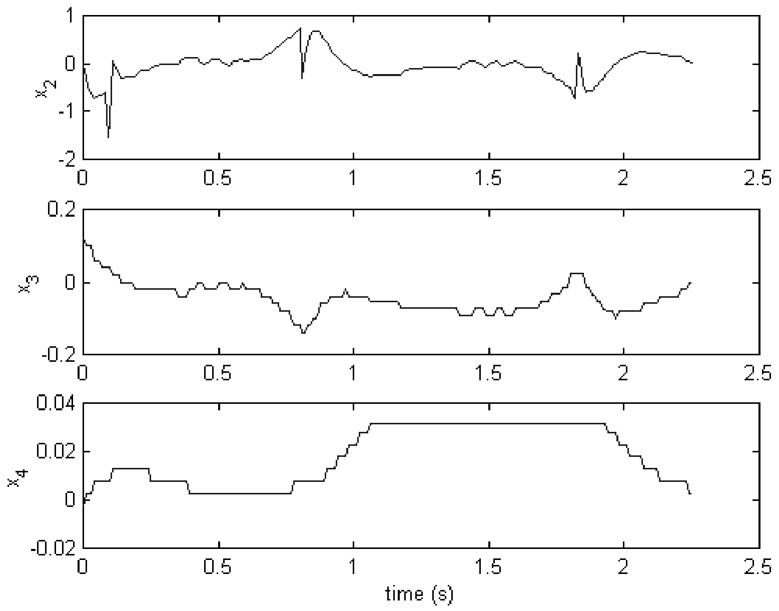

Figure 5.10: The car's states resulting from the using the ϕ estimator.

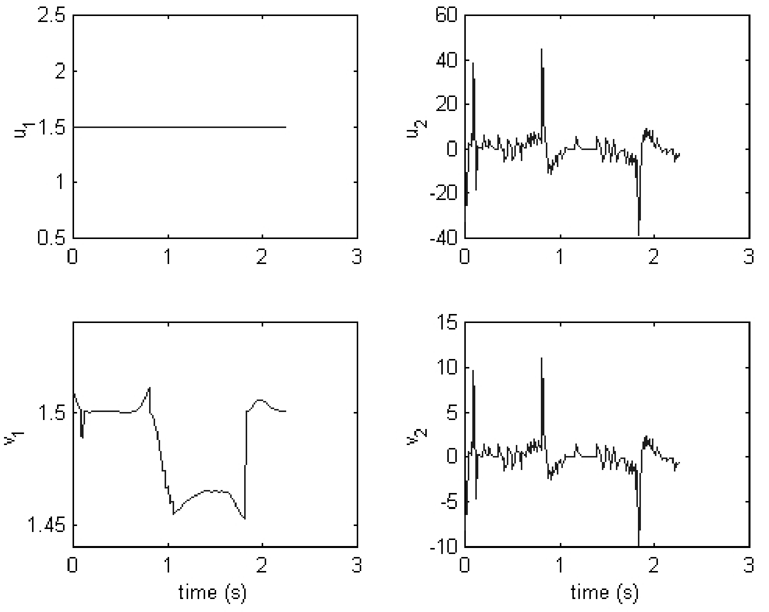

Figure 5.11: The control resulting from the using the ϕ estimator.

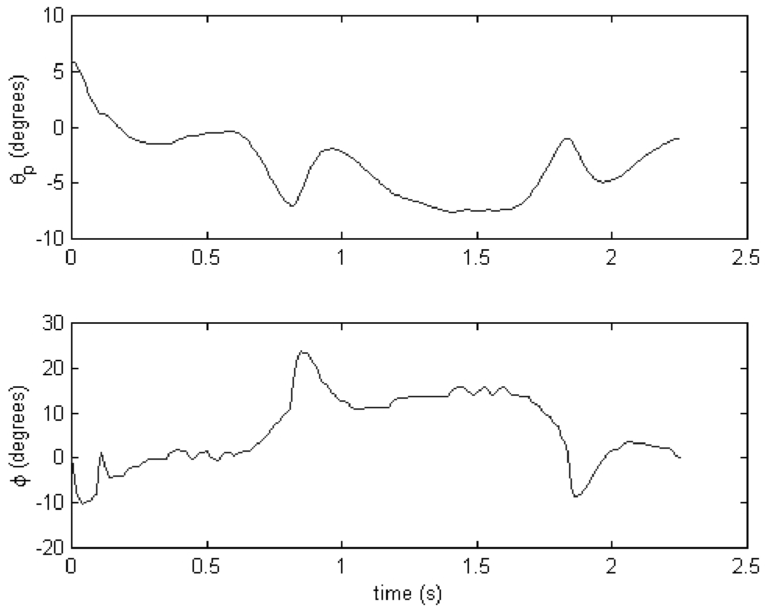

Figure 5.12: The heading angle, θ_p and steering angle, ϕ, resulting from using the ϕ estimator.

5.3.3 Control Using the Model Estimator

Next, the simulation was run using the model estimate method described in Section 4.1.2. The performance of this estimator as shown in the previous chapter was very good. As a result, there is very little difference between the controller's performance using the actual curvature or the estimated curvature. The results are shown in Figs. 5.13-5.15. This estimator in both it's forms were tested. The form in which $\hat{a}$ is thresholded to obtain c is shown here. The form in which $\hat{a}$ itself was thesholded was also tested and produced identical results in the controller performance.

This concludes the discussion of the algorithm's performance in the MATLAB simulation environment. The next chapter describes the FLASH car and the hardware implementation of the controller.

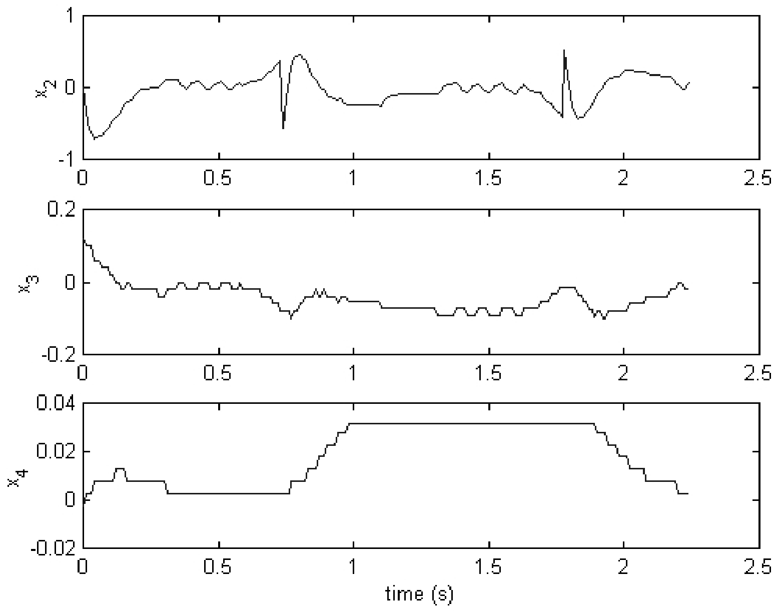

Figure 5.13: The car's states resulting from the using the model estimator.

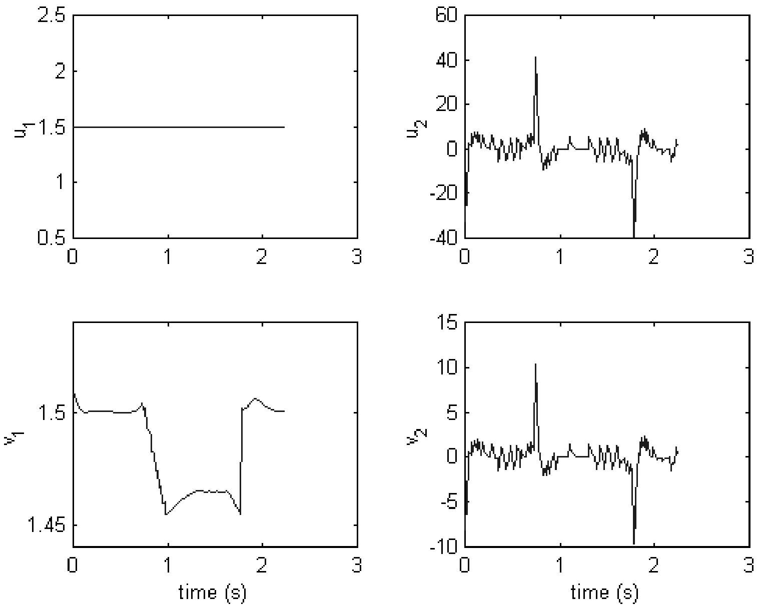

Figure 5.14: The control resulting from the using the model estimator.

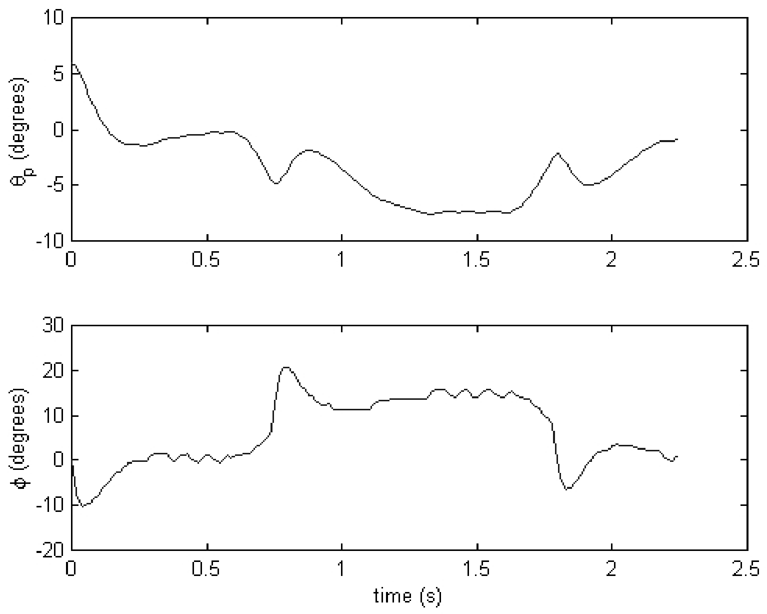

Figure 5.15: The heading angle, θ_p and steering angle, ϕ, resulting from using the model estimator.

Chapter 6

Hardware Implementation

This chapter describes the FLASH car hardware as it existed at the time of this writing. The car will continue development until the museum exhibit is completed. The car described here is the first fully operational prototype.

6.1 Overall System Structure

As the FLASH acronym indicates, one important feature of the car is that it be low-cost. To keep the cost down, the car is designed with as many low-cost, off-the-shelf components as possible. As such, the basic structure of the vehicle is standard RC model car equipment. The components that make up the basic structure of the car are given in Table 6.1.

The components in Table 6.1 are used "as is" except for the electric motor. Because the motor is manufactured for RC car racing, it is capable of traveling up to 35 mph. However, the FLASH car is being used to simulate real driving situations and need not exceed 10 mph. Therefore, the motor must be rewound with 100 turns of 30 AWG wire to reduce its top speed.

Table 6.1: Standard RC components used on the FLASH car.

Component	**Manufacturer**
Legends 1:10 scale model car kit	Bolink
Standard servo	Futaba
Paradox rebuildable electric motor	Trinity
Super Rooster electronic speed control	Novak
7.2V NiMH battery	Radio Shack

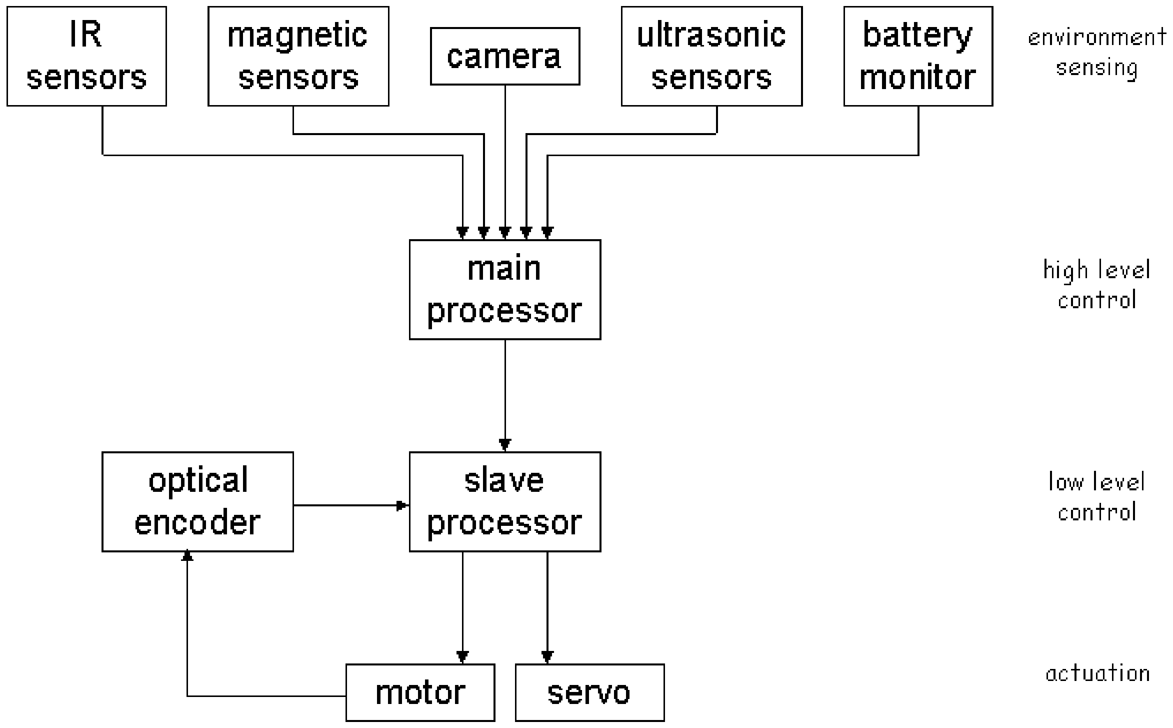

Figure 6.1: Overview of the car's hardware architecture.

Other than the components in Table 6.1, the rest of the electronics on the FLASH car were designed in-house specifically for use in this project. The only exception is the digital signal processor (DSP) which is bought as a development kit (described below).

The car's overall architecture is shown in Fig. 6.1. The system can be broken down into four hierarchical levels. First, there are several sensors which provide information about the car and its location in its environment. This information is sent to a high level processor which acts as the "brain" of the car by interpreting the information and deciding how to move the car. This main processor sends commands to a low level processor which translates the commands into drive signals. The drive signals are sent to the motor and servo which respectively drive and steer the car.

Each of the sensors shown in Fig. 6.1 provides different information to the main processor. The infrared and magnetic sensors indicate where the path is located directly beneath the car. The camera provides path information about the area in front of the car. The IR, magnetic, and camera sensors are used for lateral control, i.e. to determine how to steer the car. The ultrasonic sensor indicates how close an object is to the front of the car. The ultrasound is used for automatic cruise control or headway control. Finally, the battery monitor gives information about how much power remains. Knowing the power level helps

Table 6.2: Sensors used on the FLASH car.

Component	**Model**	**Manufacturer**
Reflective object sensor	QRD1114	Fairchild Semiconductor
Hall effect sensor	HAL506UA-E	Micronas
CMOS image sensor	OV7610	OmniVision
Ultrasonic scanner kit	3-705	Mondo-tronics

determine when the car needs recharging. Table 6.2 lists all of the sensors used on the vehicle.

In the following sections, each subsystem of the car is described in detail.

6.2 Actuator Control

Low level control is done with a microcontroller. The microcontroller receives commands from the main processor and interprets those commands to generate signals that directly control the steering servo and motor.

6.2.1 PIC16F874 Microcontroller

The microcontroller chosen was the PIC16F874 from Microchip. A microcontroller was chosen because it is ideal for performing the down-conversion from the DSP to the motor and servo. This architecture allows the DSP to perform high level control while the microcontoller interfaces directly with the actuators.

This PIC16F874 is a 40-pin, high performance RISC (reduced instruction set computer) processor that is capable of executing 35 instructions. It operates at 20 MHz and has 4K x 14 words of FLASH program memory, 192 x 8 bytes of data memory, and 128 x 8 bytes of EEPROM data memory. The PIC also has 2 8-bit timers and 1 16-bit timer. Communication with peripherals is done through a synchronous serial port and five parallel ports. The specifications for this microcontroller are summarized in Table 6.3. [12]

The PIC microcontroller is programmed using the MPLAB Integrated Development Environment (IDE) available from Microchip. This development tool allows for the assembly and simulation of code. The simulator is convenient for debugging and code verification. Once the code has been assembled and a HEX file created, the program can be downloaded to the chip using MPLAB and a PIC device programmer. After the chip has been programmed, it can be placed into a circuit and upon power-up, the code begins executing.

Table 6.3: Technical information for the PIC microcontroller.

Device name	PIC16F874-20/P
Manufacturer	Microchip
Operating voltage range	2.0 V - 5.5 V
Operating frequency	DC - 20 MHz
FLASH program memory	4 K x 14-bit words
Data memory	192 bytes
EEPROM data memory	128 bytes
Interrupts	14
I/O ports	5
Timers	3
Capture/compare modules	2
Instruction set	35 instructions
Package type	40-pin DIP

6.2.2 Program Flow

The PIC has two modes of operation. It can operate manually or autonomously. In manual mode, the PIC receives commands from the serial port. In the lab, a computer has been set up with a driving station so that the car can be driven by someone sitting at the computer. The computer sends commands through a wireless transmitter. A receiver is connected to the serial port on the PIC. This mode of operation is rarely used and will be disabled in the museum exhibit. Therefore, the details of its operation are not covered here. Autonomous mode utilizes the DSP to perform the driving task. Since this mode has received the most design attention and will be implemented in the museum exhibit, the rest of the section discusses the autonomous mode of operation.

As stated above, the PIC receives commands from the DSP and converts them into motor and servo drive signals. Fig. 6.2 shows the program flow for the PIC. The events shown in Fig. 6.2 are interrupt driven and the details of each process are given below.

6.2.3 Program Details

Interface with the DSP

The PIC is connected to the DSP through the parallel slave port (PSP) on the PIC. The PSP is one of the PIC's five parallel ports. The PSP can be read from or written to depending on the control signals sent to the $\overline{\text{CS}}$, $\overline{\text{RD}}$, and $\overline{\text{WR}}$ pins. In this application, the PSP is only written to by the DSP.

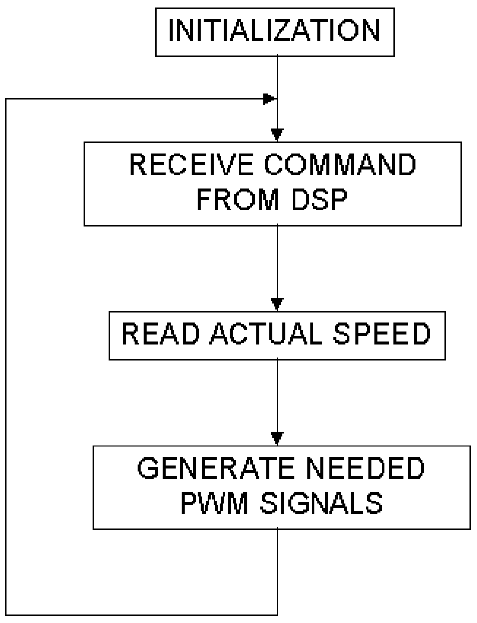

Figure 6.2: Program flow for the PIC microcontroller.

All of the control signals are generated by the DSP. When the $\overline{\mathrm{CS}}$ and $\overline{\mathrm{WR}}$ are both low, data is latched into the input buffer of the PSP. When these signals then become high, the input buffer full flag is set and an interrupt is triggered in the PIC. The interrupt service routine then interprets the data that was received from the DSP. The format of the command is covered in the next section on the DSP.

Speed Control

If the car encounters an incline, the power to the motor must be increased to maintain a constant speed due to the increased load. Similarly, if the car is traveling downhill, the power to the motor must be decreased. The process of adjusting the power to the motor to maintain constant velocity is called speed control. Since the PIC is responsible for generating the actual signals to drive the motor, it is also used to perform speed control. Thus, the DSP can assume that whatever speed it is commanding is being obtained.

An optical encoder is utilized to measure the speed of the wheels. An optical disk with slits like the one shown in Fig. 6.3a is connected to the car's rear axle. An optical emitter/detector pair is placed on either side of the disk. The slits in the disk allow light from the emitter to reach the detector causing the detector to turn on and off as the disk rotates. Thus, the output of the encoder is a square wave whose frequency is proportional to the angular

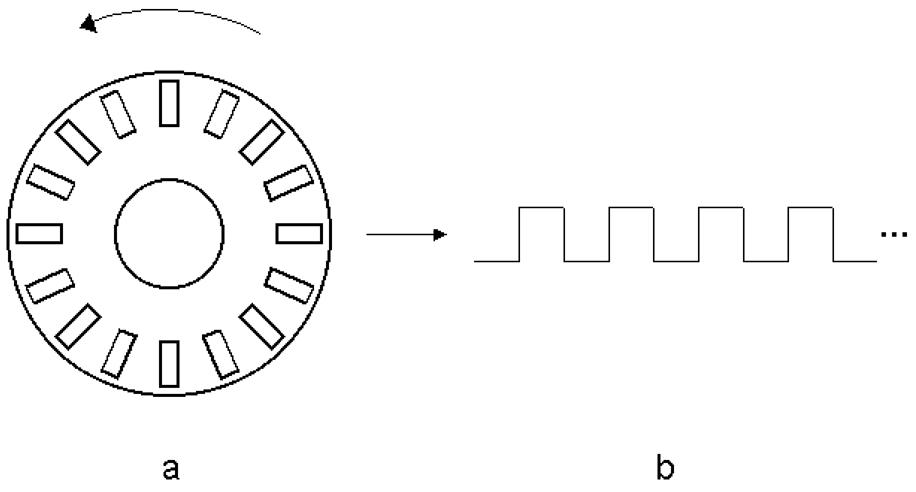

Figure 6.3: The optical disk placed on the rear axle of the car (a). The output signal generated by the encoder (b).

velocity of the rear wheels. See Fig. 6.3b.

The optical disk used on the FLASH car contains 512 slits. The output of the encoder is fed through a D flip-flop that divides the frequency by two. The resulting signal gives 256 rising edges per one revolution of the wheel. This signal is sent to one of the timers on the PIC, which is configured as a counter. The PIC counts the rising edges for a set period of time. The DSP is configured to send the desired number of encoder ticks in its velocity command. The PIC then compares the value received from the DSP to the value counted from the encoder. If the counter value is too small, the velocity must be increased and if it is too large, the velocity must be decreased. The velocity is varied by changing the pulse width of the control signal sent to the motor (see next seciton).

The amount that the velocity is changed by is determined by a proportional integral derivative (PID) controller. The PID controller has the following form.

$$\Delta v = k_i \int e + k_p e + k_d \dot{e}$$

where Δv is the change in velocity of the rear wheels, e is the error signal between the desired and actual number of encoder counts, and k_i, k_p, and k_d are gains that should be chosen for the best system performance and stability. Because the system is operating on discrete time intervals, $\int e$ becomes Σe and $\dot{e}$ becomes $e_k - e_{k-1}$.

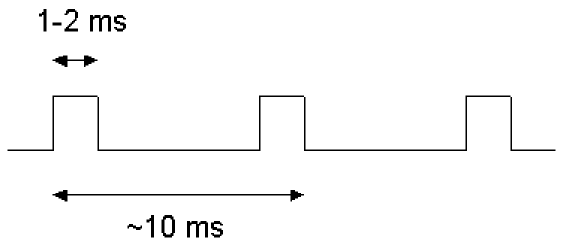

Figure 6.4: Format for the standard servo PWM control signal.

Generating PWM Signals

The servo and motor both operate using pulse width modulated (PWM) signals. A PWM signal has a fixed frequency and variable duty cycle. The standard PWM signal for servos is shown in Fig. 6.4. It has a pulse width between 1 ms and 2 ms and a frequency of about 100 Hz. A pulse width of 1.5 ms makes the wheels point straight ahead while 1 ms turns them 45° left and 2 ms turns them 45° right.

The motor control works in a similar way. The electronic speed control (ESC) from Novak receives a standard servo PWM signal and converts it to a motor drive signal using a logic circuit and an H-bridge. The ESC is programmable so that the pulse widths for neutral, full forward, and full reverse can be set by the user.

The PIC generates two independent PWM signals, one for the servo and one for the motor, using two of its timer modules. An output pin is set high and the timer module register is set to give the appropriate high time for the pulse width. When the timer expires an interrupt occurs and the interrupt service routine sets the output pin low and loads the necessary low time value into the timer register. When the timer expires again and the interrupt is set, the output pin is set high. Care is taken to make the sum of the high time and low time constant, so that the PWM frequency is constant.

6.3 Microprocessor Control

The high level control of the car is done by a microprocessor. The microprocessor is responsible for obtaining information from the various sensors. It then uses this data to determine steering and velocity control signals to be sent to the motor and servo control circuitry.

Table 6.4: Technical information for the C31 DSP.

Device name	TMS320C31
Manufacturer	Texas Instruments
Processor type	32-bit floating point
Operating voltage	5 V
Operating frequency	50 MHz
Cycle time	40 ns
On-chip RAM	2 Kwords
On-chip ROM	Boot loader
Off-chip addressable memory space	16M x 32
Serial ports	1
DMA channels	1
Timers	2
Package type	132 PQFP

6.3.1 TMS320C31 DSP

The microprocessor being used in this application is the TMS320C31 by Texas Instruments. This is a specific type of microprocessor known as digital signal processor (DSP). A DSP was chosen over a microcontroller for the car because DSPs are well suited for numerically intensive applications such as this one. Additionally, C compilers are available for the TI family of DSPs, thus eliminating the burden of writing assembly code.

The C31 DSP is a 32-bit floating-point device operating at 50 MHz. There is 2K of internal memory available as well as access to 64M x 32 of external RAM. The chip also has a built in boot loader so that programs can be stored and run on the DSP. Additional peripherals include a serial channel, a direct memory access (DMA) channel, and 2 timers. Table 6.4 summarizes the features of the C31 DSP. [13]

One advantage of the C31 is that it comes with a DSP Starter Kit (DSK). The features of this kit are given in Table 6.5. The DSK enables the user to connect the DSP to the parallel port on a PC and download code using a DOS interface. This interface allows the programmer to step through the code on the DSP and check the values of registers and memory locations while debugging. While appropriate for development, this is not practical in the final system as the program must be started using the PC and then disconnected. The final system will include boot memory, an electronically erasable programmable read-only memory (EEPROM) chip which contains the program code. The DSP can be set to load the program from this chip and begin execution.

Table 6.5: Technical information for the DSP Starter Kit.

Device name	TMS320Cx DSP Starter Kit
Manufacturer	Texas Instruments
On-board processor	TMS320C31
Operating voltage	6 V - 9 V
On-board oscillator frequency	50 MHz
Host PC interface	Standard or enhanced parallel printer port
Host interface logic	22V10Z PAL
Analog interface circuit	TLC32040
Analog interface	RCA plug connectors
Daughtercard interface	4 32-pin headers
User interface	Tri-color LED idiot light

6.3.2 Program Flow

The DSP is responsible for gathering information about the car and determining how fast to drive and in what direction to steer. The program flow is shown in Figure 6.5.

After boot-up, the main loop of the program sequentially gets data from each peripheral sensor, processes that data, determines control inputs, and sends the control inputs to the motor and servo control circuit. The motor and servo control are described in the previous section. The program continues in this loop until the car is turned off or a condition is met requiring the DSP to go into low power mode.

6.3.3 Program Details

Data Read

The DSP must collect data from various sensor devices to determine position information about the car. This section only describes how the DSP reads data from a given peripheral. The specific details of the individual sensor devices are covered in their respective sections. The processing done on this data is described in a the Data Processing section below.

There are two ways in which the DSP receives data from the peripherals: through the serial port bus and through the data bus.

Serial Port Operation The DSP can be configured to read in analog signals from a peripheral. The C31 development board contains TLC32040 analog interface circuit (AIC) which has built-in A/D and D/A converters. This device is connected to the serial port

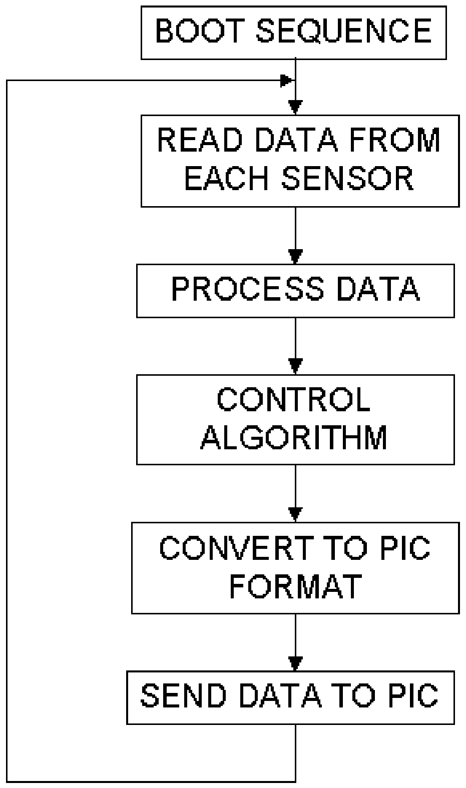

Figure 6.5: DSP program flow.

on the DSP and allows for data to be transmitted and received serially. Signals can be connected to a pin on a header or to the RCA jacks provided on the C31 circuit board. In this application, data is received serially from only one external device.

The DSP's timer can be configured to clock the AIC at rates between 75 kHz and 10 MHz, thus defining the sampling time for the A/D converter. The data gets received into an input buffer on the DSP. When the buffer is full an interrupt is generated and the data can be read into a variable. Full details of the AIC and DSP serial port interface are given in [14].

The A/D conversion rate is given by

$$f_{conv} = \frac{MCLK}{2\text{x}A\text{x}B}$$

where $MCLK$ is the frequency of the clock signal applied to the AIC by the DSP. The value can be either 6.25 MHz or 12.5 MHz and is selectable in software. A and B are the values loaded into the receive counters.

Once the serial port and AIC have been initialized, the global interrupt bit must be set and the serial port receive interrupt enabled by writing the appropriate values to the Interrupt-Enable register. When the serial port receive interrupt occurs, the program is diverted to the following C subroutine for handling.

```
/* Serial port 0 receive ISR */
void c_int06()
{
   asm(" LDI   @80804Ch,R7");
   asm(" STI   R7,@_dist");
   dist = dist & 0xFFFC;
   dist = dist>>2;
   return;
}
```

Reseting of the interrupt flags is done automatically and need not be included in the interrupt subroutine explicitly. The value that is received from the A/D converter is read into the global variable *dist*. The two LSBs of the received word are configuration bits and therefore must be masked out. The value in *dist* is now available to the main C program.

Data Bus Operation To read data from a peripheral, the program also utilizes the data and address buses of the DSP. The address bus is used to select one of the peripherals and the data bus is used to transfer information. Figure 6.6 shows how the data bus peripherals are connected to the DSP.

Each device is buffered before connecting to the DSP's data bus. Tri-state buffers are used so that each peripheral can be disconnected from the data bus. The tri-state buffers are enabled using a 3x8 decoder. This decoder takes the 3 LSBs of the address bus and enables

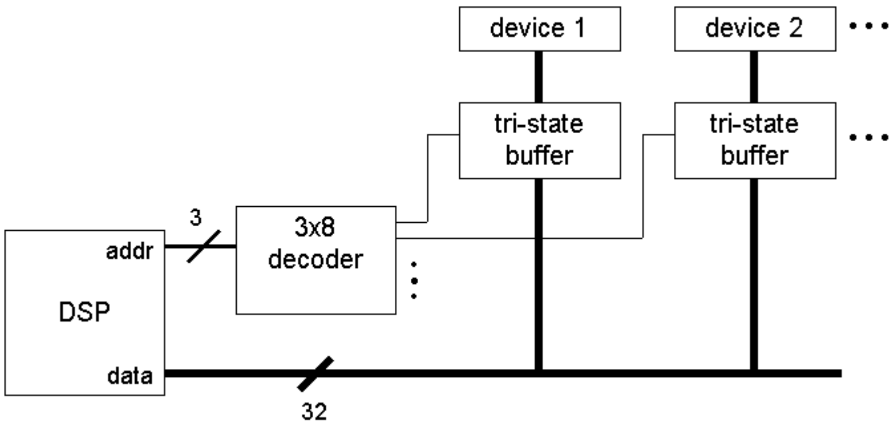

Figure 6.6: Interface between the DSP and the peripheral devices.

one peripheral for reading. Thus, only one device is connected to the data bus at a time. Because 3 address bits are being used, it is possible to select 8 different devices ($2^3 = 8$).

Once the appropriate data is on the data bus, the program must read it into a variable for use later. Access to the hardware is done using assembly language functions directly. The assembly language functions are called from C and the bus data gets put into a C variable.

The following assembly code function was written to read the data into a variable in C.

```
         .global _mem,_input,_offset
         .global _GetMem

_GetMem  LDP   800000h,DP     ;load 8 MSBs of address

         LDI   @_mem,AR0      ;load memory location into auxillary
         LSH   4,AR0          ;register.
         ADDI  @_offset,AR0

         PUSH  R0
         LDI   *AR0,R0        ;read stuff in from external memory
         STI   R0,@_input     ;store it in the variable "input"
         POP   R0

         RETS
```

The variables passed from C are prefixed by the "_". The global variables *mem*, *offset*, and *input* are used to select the particular device and store the data. First the data in the CPU registers are saved on the stack. The data is read from memory location *mem+offset* and stored in the variable *input*. The base external memory location is specified by *mem* and *offset* indicates which device is being accessed. Before leaving this function, the registers are restored to the values saved on the stack. The data is now available to C in the variable *input* for processing later.

Data Write

The DSP must send velocity and steering information to the PIC microcontroller to drive the motor and servo. It does this similar to the Data Read described above.

The function used to send information to the data bus is given now.

```
         .global _mem,_output,_offset
         .global _PutMem

_PutMem  LDP   800000h,DP      ;load 8 MSBs of address

         LDI   @_mem,AR0       ;load memory location into auxillary
         LSH   4,AR0           ;register.
         ADDI  @_offset,AR0

         PUSH  R0
         LDI   @_output,R0     ;send stuff to external
         STI   R0,*AR0         ;memory.
         POP   R0

         RETS
```

There is very little difference between the *PutMem()* function and the *GetMem()* function of the previous section. The *PutMem()* function puts the value stored in the global variable *output* onto the data bus using address *mem+offset*. However, there is a difference in the way these functions are used in the main program. The *PutMem()* function is used to send data to the PIC microcontroller parallel slave port (PSP). Data must be put on the slave port pins and then the port must be triggered. So the *PutMem()* function must be called twice: once to enable the PSP for collecting the data and once more with a different *offset* value to create a rising pulse to trigger the PSP.

Data Processing

Once the data from the various peripherals has been collected, the processor must interpret this information to determine the car's position with respect to the road. The following sections describe this process.

Infrared and Magnetic The infrared and magnetic sensors give data in an identical format. The algorithm for each sensor type is the same, so both are treated here.

Each bumper has N sensors so the DSP receives N bits of data from each bumper (front and rear). These bits are normally 5V (logic 1), but in the presence of the line become 0V (logic low). These bits are read in from each bumper at the same time, yielding an input of length $2N$ bits. The input is split into two variables, one for the front and one for the rear. The order of the bits is then corrected in each variable so that the leftmost bumper bit is the MSB and the rightmost bumper bit is the LSB.

Once the DSP has the correct data from each bumper, it must convert this information into a distance. This distance represents how far the bumper's center has deviated from the line in the road. To determine the distance, the width of the bumper and the spacing of the sensors must be known. This was done by measuring the bumper and recording the values in inches. These values are stored as constants in the C program.

To determine the actual distance in inches, the following algorithm was developed. Starting with the rightmost bumper sensor (the LSB of the variable), the value of each bit is checked. If it is a 1, no line was detected by that sensor and the bit is ignored. However, if the bit is a 0, a line was detected by that sensor and some calculation must be done. Each bit respresents some distance from the center of the bumper, depending on the bumper width and sensor spacing. Mathematically this distance can be represented as:

$$d = (k - i)\Delta x$$

where k is the number of spaces in half the bumper, i is the bit's position (0 to N-1), and Δx is the sensor spacing. This distance is summed over all i and the result is divided by the total number of turned-on sensors. The distance calculated is positive if the bumper is to the left of the line.

Image Processing The image processing algorithms have not been implemented on the FLASH car at the time of this writing. The C31 DSP does not have the capability to capture images from the camera. The details of the camera and its interface with the DSP are given in Section 6.5.

Ultrasound The ultrasound sensor outputs a voltage that is proportional to the distance that the object is located in front of the car. The analog signal from the ultrasound sensor is read into the A/D converter on the DSP circuit board. The digital signal is then read into the DSP through the serial port. When the serial input buffer is full, an interrupt is generated and the following interrupt service routine is executed as discribed in Section 6.3.3.

The data that is read into the variable *dist* is then converted into a number that represents the actual distance of the object in front of the car by

$$distance = \frac{dist}{3150}$$

where *dist* is the value received into the serial port buffer from the A/D converter.

Table 6.6: Format for the PIC control word

BIT	USAGE
0..5	velocity or steering value
6	0 is steering, 1 is velocity
7	0 is disable, 1 is enable

Control Algorithm

Using the data collected above, control algorithms can be implemented on the DSP to control the steering and velocity of the vehicle. For this research, the controller discussed in Chapter 3 and estimator discussed in Chapter 4 have been implemented. In addition, several others have implemented their own designs on the car including: PID, feedback linearization, sliding mode, and machine learning based controllers for lateral control. Automatic cruise control has also been implemented on the FLASH car independent of the lateral controller. See [15].

Control Word Format

The DSP communicates with the PIC microcontroller in the form of an 8-bit word sent on the data bus. Table 6.6 describes the format of the control word. The PIC microcontroller accepts velocity values that are between 0 and 32. The value 2 is neutral; 0 and 1 are reverse (high and low speed); 3 through 32 are forward (32 is the fastest speed). The program simply sets the value to the speed determined by the controller and it sends that value to the PIC each time through the main loop.

For steering, the PIC microcontroller accepts values between 0 and 25. The value 12 is a steering angle of zero degrees. The output of the lateral controller is an angular velocity. This is converted to an angle by integration and is limited to $\pm 45°$. This angle is then converted to give a value between 0 and 24. Before being sent to the data bus, the value for steering or velocity is ORed with the appropriate bits to indicate a steering or velocity command and enabled mode as given in Table 6.6.

6.4 Infrared and Magnetic Sensors

The FLASH car contains infrared and magnetic sensors to help determine where the car is located with respect to the desired path. These sensors are located on the front and rear of the car and look straight down at the roadway to determine where the desired path is underneath the car.

Table 6.7: Technical information for the infrared sensors.

Device name	QRD1114
Manufacturer	Fairchild Semiconductor
Emitter forward current	50 mA maximum
Emitter forward voltage	1.7 V
Peak emission wavelength	940 nm
Sensor dark current	100 nA
Collector emitter saturation voltage	0.4 V maximum
Collector current	1 mA minimum
Package size	0.173" x 0.183" x 0.240"

6.4.1 Infrared Sensor Operation

The FLASH car utilizes infrared emitter/detector pairs to locate the white line on the black road surface. The emitter sends out IR light with a wavelength of 940 nm. The detector is located next to the emitter in the same package. When the sensor is over the white line, the light is reflected back and seen by the detector. When the sensor is over the black line, no light is reflected and thus nothing is detected. The specifications for the IR sensors are given in Table 6.7.

The circuit for the IR sensor is shown in Fig. 6.7. The resistor value must be selected to send the appropriate amount of current through the light emitting diode. On the detector side, the output is open collector, so a pull up resistor must be used. The transistor is turned on in the presence of light, making $V_{out} = 0$. Otherwise, the transistor is off and V_{out} is pulled up to 5 V.

These sensors are very reliable in the presence of the high contrast line. However, they give erroneous signals if there is debris or sunlight on the track. Also, this configuration requires that the car straddle the white line while on roadways today, cars drive between the lines.

6.4.2 Magnetic Sensor Operation

The FLASH car also uses Hall effect sensors to locate the magnetic line which is beneath the roadway. These sensors detect the presence of a magnetic south pole. They turn on when 5.5 mT of magentic field strength is detected and turn of when the field strength falls below 3.5 mT. The circuit application is very similar to that of the IR sensor shown above. Only power, ground, and a pull up resistor are needed for the sensor to operate. Table 6.8 summarizes the specifications for these Hall effect devices.

Unlike the IR sensors, the Hall effect sensors do not detect a visible line and are not suscep-

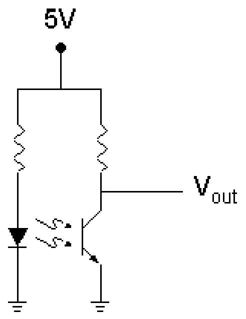

Figure 6.7: The circuit application for the IR sensor.

Table 6.8: Technical information for the Hall effect sensors.

Device name	HAL506UA-E
Manufacturer	Micronas
Operating voltage	3.8 V - 24 V
Supply current	3 mA
Switching type	unipolar
B_{on}	5.5 mT
B_{off}	3.5 mT
Package size	4.06 mm x 1.5 mm x 3.05 mm

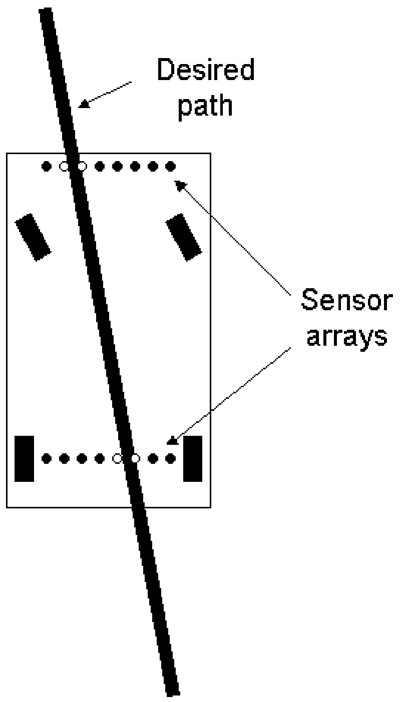

Figure 6.8: The location of the sensors on the vehicle.

tible to interference from ambient light. However, they require the presence of a magnetic field which is not embedded in most roadways.

6.4.3 Data Format

An array of IR sensors and magnetic sensors is placed on both the front and rear of the FLASH car. The thickness of the line and the spacing of the sensors is such that two are turned on at a time. The sensor configuration is shown in Fig. 6.8. Note that the rear sensors are directly between the rear wheels. This is done so that the error from the rear sensors gives the lateral placement, d, directly (see Fig. 3.3). At the time of this writing, 12 IR sensors and 8 magnetic sensors are used for each array and provide good resolution. However, the software is configured so that any number of sensors can be used. So the number of sensors is limited by the width of the data bus.

Table 6.9: Technical information for the digital camera.

Device name	OV7610
Manufacturer	OmniVision
Operating voltage	5 V
Operating current	40 mA
Array size	644 x 484 pixels
Pixel size	8.4 μm x 8.4 μm
Effective image area	5.4 mm x 4 mm
Communication interface	I^2C
Package size	40 mm x 28 mm

6.5 Vision System

The vision system includes the use of a digital camera mounted on the front of the vehicle facing forward. The use of the camera in the lateral control algorithm was described in Chapter 4. This camera provides the images of the roadway ahead of the car. The camera's specifications are given in Table 6.9.

At the time of this writing, the camera has not been incorporated into the FLASH vehicle. The C31 DSP does not have enough on-board memory to hold a frame of the image nor can it perform the necesssary processing on the image while controlling the vehicle at the same time. The camera captures images sized 640x480 pixels, thus requiring over 300 Kb of memory to store one image. The C31 DSP has only 8 Kb of internal memory and there is no memory located on the DSP circuit board.

To remedy this, the DSP on the car will be upgraded to a more powerful device with more memory on-board and more located off the chip on the evaluation circuit board. In addition, the processor itself will be faster. With these improvements, the image transfer can be performed over the data bus using direct memory access (DMA). DMA allows the processor to transfer data in the background while still performing other tasks. With the necessary upgrade, the DSP will be able to control the car and use the image information simultaneously.

6.6 Ultrasonic System

An ultrasonic sensor is mounted on the front of the car to detect the presence of another vehicle in front. This information is used to perform adaptive cruise control, a mechanism to maintain a minimum distance between cars. Although the ultrasonic sensor is not used

Table 6.10: Technical information for the ultrasound sensor.

Device name	Ultrasonic Owl Scanner Kit #3-705
Supplier	Mondo-tronics
Operating voltage	9 V - 12 V
Sensor	Polaroid transducer
Signal frequency	40 kHz
Communication interface	RS232 @ 9600 baud
Analog output	0 V - 5 V
Measured distances	150 mm - 2.6 m
Distance resolution	10 mm
Circuit board size	90 mm x 55 mm
Transducer size	1.513" diameter

by the controller described in this thesis, a brief description of its operation is given here. For complete details of the adaptive cruise controller on the FLASH car, see [15].

The specifications for the ultrasound kit are given in Table 6.10. The ultrasound kit consists of a Polaroid transducer and control module. The control module circuit provides 40 kHz signals to the Polaroid transducer to make it vibrate. If an object is in front of the transducer, the reflected signal is detected. The control module interprets the signal from the transducer and converts the distance into a voltage from 0-5 V. The transducer can detect objects between 150 mm and 2.6 m. 0 V indicates that the object is 150 mm away or closer. 5 V indicates that the object is at least 2.6 m away. This voltage is sent to the A/D converter on the DSP circuit board and gets read into the DSP as the distance from the front car (or some other object).

6.7 Power and Recharging System

With all of the electronics described above on the car, the nickel metal hydride (NiMH) battery provides about 1.5 hours of runtime. And because this car is part of a musuem exhibit, it is desirable to have the car run with as little intervention from the staff as possible. Thus the car should be capable of self-monitoring and automatic recharging. As of this writing, the automatic recharging system is still under development. An overview of this subsystem's operation is given in this section.

The flowchart for the recharging algorithm is shown in Fig. 6.9. While the car is operating normally, the DSP will monitor the battery voltage and current through a monitoring circuit. The DSP must know both the voltage and current so that the true battery voltage is known. After reading in this data, it must be filtered so that noise in the sensor does not falsely cause the car to go into recharge mode. If the battery voltage is sufficient, the car continues along the path. If the voltage is below some threshold, however, the car goes into recharge mode.

In normal running mode, the car uses the IR sensors as the primary input for path following. However, in recharge mode, the car switches to the magnetic sensors as the primary sensors. The magnets under the track deviate from the main path and lead the car into the recharging station. Once in the recharging station, the car can detect that the battery is receiving current and shuts down all unnecessary circuitry. When charging is done, the car re-enables itself and exits the recharging station, rejoining the main track and switches back to the IR sensors.

While the car can run for about 1.5 hours, recharging can take up to 5 hours. Thus there is a need for multiple cars in the exhibit, some driving while the others recharge, so there can be cars driving on the track during the entire day. Each car has its own station so that recharging can be done in parallel. To coordinate the multiple cars and multiple recharging stations, the track itself is intelligent. A central computer oversees the operation of the stations, knowing which have cars in them. Stations that have cars in them are disabled by turning off the magnetic field underneath to ensure that a car needing a recharge does not come crashing into the car that is already there.

6.8 Controller Performance

This section describes the implementation and performance of the input scaling controller on the hardware described in the previous sections.

6.8.1 Simulation vs. Hardware

As is to be expected, the actual car did not perform exactly the same as the simulated car. There are several reasons for this, the major one being differences between the modeling and true vehicle dynamics.

In MATLAB, the kinematic model given in (3.9) was used to simulate the movement of the car. This model does not account for slippage, inertia, or any other dynamic effects that may take place on the actual car. These effects are most evident in the turns at higher speeds where the forces on the car are greater and the tires may lose traction with the track. At lower speeds, the car did not experience these forces as much and performed more like the kinematic model. This is noticable in the performance of the controller.

The simulation is also unrealistic in that it does not take into account the dynamics of the actuators. Using the controller in the simulation, it was possible to instantaneously change the steering angle and velocity of the vehicle. This is not possible in the real world and this

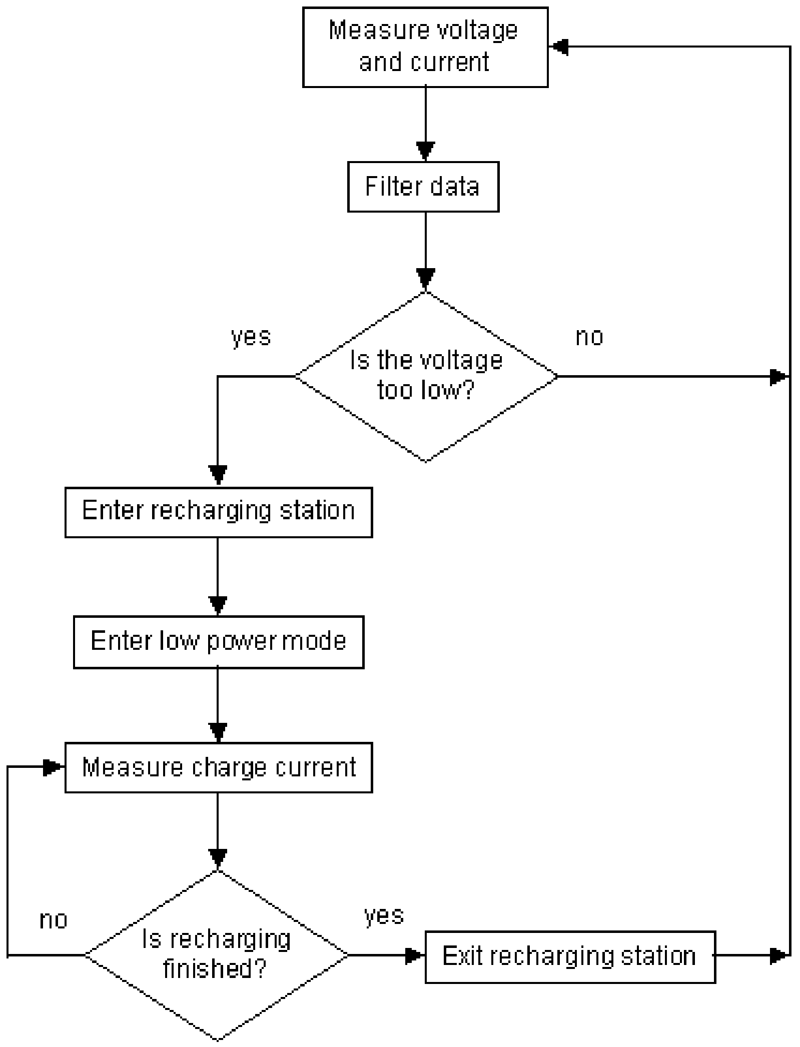

Figure 6.9: Flowchart for the recharging system.

affected the controller gains that could be used for stable operation. The dynamics of the vehicle's velocity were taken into account by utilizing speed feedback in the lateral controller.

As described above in the section on the car's hardware architecture, the DSP commands a velocity and the PIC microcontroller is responsible for reaching and maintaining that speed. The PIC implements a PID controller using feedback from an optical encoder attached to the rear axle. Because the PIC is a low level processor, an optimal PID controller is difficult to implement. Therefore the speed of the car does vary somewhat. However, to take this into account in the lateral controller, the PIC sends the actual speed (given by the encoder) back to the DSP. This actual speed is used by the lateral controller and the curvature estimator as u_1.

In addition to the motor dynamics, there are the dynamics of the steering servo. There is a lag of about 0.25 seconds between the servo control pulse being sent and the servo turning to the corresponding position. While this is very fast in terms of the car's time frame, it is very slow to the DSP operating at 50 MHz and could potentially cause instability.

Differences in the program code itself were few. Aside from the expected differences between the syntax of MATLAB and C, the programs were similar. As in the simulation, there was again difficulty typing equations (3.14)-(3.18) correctly.

6.8.2 Controller Performance

This section describes the performance of the controller under various conditions. Unfortunately, the car does not have any data logging and the variables that it calculates as it is driving are not available for analysis. As such, only a qualitative discussion of the car's performance can be given here.

Controller Implementation

One of the first problems that was noticed was that if the car lost the line completely, it would have trouble finding its way back. Initially this was puzzling because the car was programmed to "know" on which side the line was because it had the last known location stored in memory. Then it was realized that if the line was lost by each sensor array to the same side, the lateral controller used a value of zero for θ_p. With the given gains, a value of zero for θ_p cause the car to keep going straight rather than turn back towards the desired path. So the ratio of the gains k_1, k_2, and k_3 were modified so that the controller would react more to the lateral displacement, d, and return to the desired path. With this modification in gains, the car could find its way back.

However, with the change in gains, the controller had a different problem. When the car encountered a curve, it would wait until the rear axle was off the line ($d \neq 0$) before reacting to the change. When this happened, the car would suddenly jerk the wheels to get back to the desired path. The gains had been modified so that too much weight was given to d and not enough to θ_p.

Finally the proper gain ratio was found to remedy this problem. The gains where changed to $k_1 = 10\lambda^2$, $k_2 = 3\lambda^2$, and $k_3 = 3\lambda$. These ratios gave the proper weight to d and θ_p to make the car stay on the path. Using $\lambda = 20$ produced a good response time by the controller.

To aid in the debugging process for the curvature estimator, the qualitative performance of the car under various known conditions was necessary. This way, since the actual program variables were not known, certain ones (in particular the value being used for the curvature) could be deduced from the car's behavior.

The track in the lab consists of straightaways and turns with curvature values of $\pm 1.125\text{m}^2$. To get a feel for the controller's performance, each of these values was hardcoded and the car was allowed to travel around the track.

First the curvature was set to zero. The controller actually performed very well with $c = 0$ everywhere. The car traveled smoothly around the entire track and was able to navigate the turns without going too far off the line. With the curvature set to 1.125, the car stayed right on the line in the left turns. On straightaways, the performance was stable but there was a slight offset to one side. In the right turns, the car went completely off the line most like due to the large errors in x_2 and x_3 using the erroneous curvature. As is to be expected, the behavior of the controller with $c = -1.125$ was identical but opposite to the case where $c = 1.125$.

ϕ Estimator

The first estimator tried was the ϕ estimator described in Section 4.1.1. Although this method worked fairly well in simulation, it did not work at all on the car itself.

First, the algorithm was implemented exactly as it was in the simulation. However, the result was that the car oscillated on the straightaways, while performing well in the turns. The number of past ϕ samples to use was increased until there was no more memory on the DSP (over 1000 samples) but the performance did not improve.

Another approach was tried. The estimated value for c was required to be above or below a threshold for a certain number of consecutive sample times. But the results were similar, the car was unstable on the straightaways.

While the variables calculated by the estimator are not available, it is known that when the car oscillates, the value of c is oscillating. (If the value of c is fixed, even at the wrong value, the car follows the path in a stable fashion.) If the car's wheels began to oscillate, the estimated value for c would oscillate also because it was linearly related to ϕ. Averaging over several samples would not help because the average would still be biased to one side.

Model Estimator

Next the model estimator described in Section 4.1.2 was implemented. At first this estimator was implemented the same as in the simulation but the results were not good. In the simulation, the output of the estimator, $\hat{a}$ was thresholded to choose the actual curvature value, which was known a priori. However, it was found that $\hat{a}$ was very unstable and would oscillate back and forth across the threshold. This resulted in the value used for c to keep changing and the car's performance was unstable.

To try to fix this, the threshold values were modified from the original (0.9 on the rising edge and 0.1 on the falling edge). But no values seemed to improve the performance. So another approach was tried. It was required that the value of $\hat{a}$ be above or below the threshold for a certain number of sampling times before the curvature value was changed. This made the change in curvature be delayed a bit but the value used for c did not oscillate back and forth as before. It was found that requiring $\hat{a}$ to be above or below the threshold for 600 sample times produced the best results, reducing the transients to a minimum while not adding too much delay to the system.

The controller using the results of the estimator in this form performed very well. When the car was initially placed on the track, it needed about a second to right itself if it was displaced from the desired path. These transients were not severe however. There were also some transients as the car entered and exited a turn. This was due to the fact that there was a delay in c obtaining the correct curvature. It is interesting to note the performance of the car coming out of a turn. For the first foot or two of the straightaway, the car would be offset because it was still using the nonzero curvature value. Then when the output of the estimator indicated, the value of c would change to zero. At that point the car would jerk slightly and center itself on the road. The maximum speed that could be attained was about 1.2 m/s. If the car was set to travel faster than 1.2 m/s, it was unable to follow the path smoothly. The maximum speed was due to the controller itself rather than the curvature estimator.

To make the change between the curve and straighaway less abrupt and ease the transition, the value of c was changed in increments based on the output of the estimator. If $\hat{a}$ was greater than zero, the value for c was incremented by a certain step size. If $\hat{a}$ was less than zero, the value for c was decremented. So, c was never set to the actual values for the curvature, but it was never allowed to exceed ± 1.125.

The performance of the controller using this method was qualitatively different from the previous one. Exiting a turn, the car would be biased to one side and gradually shift itself to the center of the line as the value of c gradually changed. There was no abrupt shift as before. Using a step value of 0.0003 gave the best results. A value greater than this caused transients as the car entered or exited a curve. A value less than this cause the car to not turn quickly enough. As with the method described above, the maximum speed was again about 1.2 m/s.

While this method worked well in reducing transients, the implications in the controller need to be studied. Recall that the derivatives of $c(s)$ are assumed to be zero because the path's curvature is piecewise constant. However, with this estimation of c, the curvature being used by the controller is not piecewise constant but changing gradually.

Image Processing Estimator

One feature of the above estimators is that they are coupled to the performance of the controller. It was necessary to make them robust enough so that if the car went off the path, they could still provide an accurate curvature value. The method that is independent of the car's performance is the image processing method. Using a camera, the curvature estimation problem is decoupled from system performance to a certain extent (the road line must be in the camera's field of view). However this method requires more processing power and time. At this point, it is still an unanswered question as to how the car performs with the camera.

Overall the performance of the input scaling controller was very good. It performed well in spite of the differences between the kinematic model from which it was developed and the actual car. In addition it was not adversely affected by delays in the car's response time. It proved itself to be a controller robust to the errors and uncertainties in the system.

Chapter 7

Conclusions

7.1 Concluding Remarks

This thesis has described the current development of the FLASH lab at VTTI. Details of the car were given and the hardware and software implementation were detailed. The FLASH lab and the scale model cars contained therein provide a testbed for the small scale development stage of ITS. In addition, the FLASH lab serves as a home to the prototype display being developed for an educational museum exhibit.

This thesis also gave details of the path following lateral controller implemented on the FLASH car. The controller was developed using the kinematic model for a wheeled robot. The model was derived using the nonholonomic contraints of the system. The global model was then converted into the path coordinate model so that only local variables were needed. This was then converted into chained form and a controller was given for path following.

The path coordinate model introduced a new parameter to the system: the curvature of the path. Thus it was necessary to provide the path's curvature value to the controller. Because of the environment in which the car is operating, the curvature values are known a priori. Several online methods for determining the curvature were developed. One used the car's steering angle only to perform the estimation. The second linearly parameterized the path coordinate model and used a least square estimator. The third was based on images received from an on-board camera. For all these methods, the output of the estimator was used to choose the actual curvature value. In simulation, all of these methods were able to adequately determine the curvature of the path.

A MATLAB simulation environment was created in which to test the above algorithms. The simulation used the kinematic model to show the car's behavior and implemented the sensors and controller as closely as possible to the actual system. The details of the simulation program were given and the complete code is provided in the Appendix.

Finally, the lateral controller was implemented in hardware. The vehicle platform was described and the hardware and software architecture detailed. The code for implementing this controller is given in the Appendix. The car described is capable of operating manually and autonomously. In autonomous mode, several sensors are utilized including: infrared, magnetic, ultrasound, and image based technology. The operation of each sensor type was described and the information received by the processor from each was discussed. The possibility exists to implement many different types of controllers to perform path following or realize other control objectives.

The controller performance on the hardware was very good when the correct curvature value was used. It proved to be robust to inherent inaccuracies in the kinematic model. The curvature estimators implemented also performed well. They were able to reliably provide the correct curvature value to the controller under various conditions. The details of the hardware implementation were described as well as differences from the simulation. The image based curvature estimator was not implemented on the car due to hardware limitations with the available processor.

7.2 Future Work

7.2.1 Controller

The input scaling controller based on the kinematic model performed very well on the car itself. Major changes to the algorithm are not necessary. However, improvements can still be made in the smoothness of operation. Adjusting the algorithm so that driving comfort is the primary objective may result in smoother performance.

In addition, it is unknown how the controller will perform in conjunction with a longitudinal controller such as adaptive cruise control. The other controller may interfere with the lateral controller and cause instability. It is necessary to integrate the lateral contoller with others so that truely autonomous operation can be achieved.

7.2.2 Curvature Estimation

While two estimators have been implemented on the car itself, the image processing based one has not. The necessary hardware upgrades must be done before the camera can be used on the vehicle. Once this has been done, the image based curvature estimator can be tested. It is known through simulation how the algorithm performs on static images. However, the algorithm must be verified in a dynamic setting on the vehicle. The effects of the added processing power and time on the controller can then be assesssed.

7.2.3 Hardware

As stated above, the car must undergo a processor upgrade before the camera can be used. The new processor must be integrated into the existing hardware and the code developed on the original processor must be brought onto the new platform. Once the car is running with the new processor, the camera can then be integrated into the architecture.

For the car to be truly autonomous in the museum setting, the automatic recharging system must be implemented. The basic flow of the system has be decided and now the prototype must be built and tested. There are many details about the system that need to be worked out before the recharging system is fully operational.

Finally, the car's packaging needs to be completed. At the time of this writing, all of the circuit boards and hardware were in prototype form (i.e. lots of duct tape was used). To be robust and reliable in a museum setting, manufactured circuit boards must be made and the interconnection and mounting methods for all the components must be finalized.

There is no shortage of work to be done in the FLASH lab.

Bibliography

[1] "Traffic Safety Facts 2000: A Compilation of Motor Vehicle Crash Data from the Fatality Analysis Reporting System and the General Estimates System," DOT HS 809 337, U.S. Department of Transportation, National Highway Traffic Safety Administration, National Center for Statistics and Analysis, Washington, DC, December 2001.

[2] T. B. Reed, "Discussing Potential Improvements in Road Safety: A Comparison of Conditions in Japan and the United States to Guide Implementations of Intelligent Road Transportation Systems," *IVHS Issues and Technology*, SP-928, pp. 1-12, 1992.

[3] D. Utter, "Passenger Vehicle Driver Cell Phone Use Results from the Fall 2000 National Occupant Protection Use Survey," Research Note, DOT HS 809 293, U.S. Department of Transportation, National Highway Traffic Safety Administration, July 2001.

[4] S. Mizutani, *Car Electronics*, Nippondenso Co., Ltd., 1992.

[5] "Intelligent Vehicle Highway Systems Research at the Center for Transportation Research," Center for Transportation Research Report, 1994.

[6] P. Kachroo and M. Tomizuka, "Design and Analysis of Combined Longitudinal Traction and Lateral Vehicle Control for Automated Highway Systems Showing the Superiority of Traction Control in Providing Stability During Lateral Maneuvers," *1995 IEEE International Conference on Systems, Man, and Cybernetics.*

[7] J. C. Alexander and J. H. Brooks, "On the Kinematics of Wheeled Mobile Robots," *Int. J. of Robotics Research*, vol. 8, no.5, pp. 15-27, 1989.

[8] P. Kachroo, "Microprocessor-Controlled Small-Scale Vehicles for Experiments in Automated Highway Systems," *The Korean Transport Policy Review*, vol. 4, no. 3, pp. 145-178, 1997.

[9] J. Laumond, *Robot Motion Planning and Control*, Springer, 1998.

[10] C. Samson, "Control of chained systems. Application to path following and time-varying point-stabilization of mobile robots," *IEEE Trans. on Automatic Control*, vol. 40, no.1, pp. 64-77, 1995.

[11] D. Redfern and C. Campbell, *Matlab Handbook*, February 1, 2001.

[12] Microchip Technology Inc., *PIC16F87X Data Sheet*, Literature Number: DS30292C, 2001.

[13] Texas Instruments, *TMS320C3x User's Guide*, Literature Number: SPRU031E 2558539-9761 Revision L, July 1997.

[14] Texas Instruments, *TMS320C3x DSP Starter Kit User's Guide*, Literature Number: SPRU163A, 1996.

[15] R. D. Henry, *Automatic Ultrasonic Headway Control for a Scaled Robotic Car*, Thesis, Virginia Polytechnic Institute and State University, 2001.

Appendix A

Hardware Sources

Company:	Amitron Corporation
Product:	Printed circuit boards
Address:	2001 Landmeier Road
	Elk Grove Village, IL 60007
Telephone:	1-847-290-9800
Internet:	www.amitroncorp.com

Company:	Bolink
Product:	RC car chassis
Address:	420 Hosea Road
	Lawrenceville, GA 30245
Telephone:	1-770-963-0252
Internet:	www.bolink.com

Company:	Bourns, Inc.
Product:	Potentiometers
Address:	1200 Columbia Avenue
	Riverside, CA 92507-2114
Telephone:	10877-4-BOURNS
Internet:	www.bourns.com

Company:	Digi-Key
Product:	Electronic components
Address:	701 Brooks Avenue South
	Thief River Falls, MN 56701
Telephone:	1-800-DIGI-KEY
Internet:	www.digikey.com

Company:	Fairchild Semiconductor Corporation
Product:	Infrared sensors
Address:	82 Running Hill Road
	South Portland, ME
Telephone:	1-800-341-0392
Internet:	www.fairchildsemi.com
Company:	Futaba Corporation of America
Product:	Servos
Address:	2865 Wall Triana Highway
	Huntsville, AL 35824
Telephone:	1-256-461-7348
Internet:	www.futaba.com
Company:	Jameco Electronics
Product:	Electronic components
Address:	1355 Shoreway Road
	Belmont, CA 94002-4100
Telephone:	1-800-831-4242
Internet:	www.jameco.com
Company:	Microchip Technology Inc.
Product:	PIC microcontrollers
Address:	2355 West Chandler Boulevard
	Chandler, AZ 85224-6199
Telephone:	1-480-792-7200
Internet:	www.microchip.com
Company:	Micronas Semiconductor Holding AG
Product:	Hall effect sensors
Address:	Technopark
	Technoparkstrasse 1
	CH-8005 Zurich
	Switzerland
Telephone:	+41-1-445-3960
Internet:	www.micronas.com
Company:	Mondo-tronics, Inc.
Product:	Ultrasound kits
Address:	4286 Redwood Highway PMB-N
	San Rafael, CA 94903
Telephone:	1-800-374-5764
Internet:	www.robotstore.com

Company:	National Semiconductor
Product:	Discrete semiconductor components
Address:	2900 Semiconductor Drive
	P.O. Box 58090
	Santa Clara, CA 95052-8090
Telephone:	1-408-721-5000
Internet:	www.national.com

Company:	Novak Electronics, Inc.
Product:	Electronic speed control
Address:	18910 Teller Avenue
	Irvine, CA 92612
Telephone:	1-949-833-8873
Internet:	www.teamnovak.com

Company:	Radio Shack Corporation
Product:	NiMH RC car batteries
Address:	300 West Third Street, Suite 1400
	Fort Worth, TX 76102
Telephone:	1-800-THE SHACK
Internet:	www.radioshack.com

Company:	Symmetry Electronics Corporation
Product:	Hall effect sensors
Address:	5400 Rosecrans Avenue
	Hawthorne, CA 90250
Telephone:	1-310-536-6190
Internet:	www.symmetryla.com

Company:	Texas Instruments Incorporated
Product:	Digital signal processors and discrete logic
Address:	12500 TI Boulevard
	Dallas, TX 75243-4136
Telephone:	1-800-336-5236
Internet:	www.ti.com

Company:	Tower Hobbies
Product:	RC car components
Address:	PO Box 9078
	Champaign, IL 61826-9078
Telephone:	1-800-637-6050
Internet:	www.towerhobbies.com

Company:	Trinity
Product:	R/C electric motors
Address:	36 Meridian Road
	Edison, NJ 08820
Telephone:	1-732-635-1600
Internet:	www.teamtrinity.com

Company:	US Digital Corporation
Product:	Optical encoders
Address:	11100 NE 34th Circle
	Vancouver, WA 98682
Telephone:	1-800-736-0194
Internet:	www.usdigital.com

Company:	Vishay Americas, Inc.
Product:	Discrete semiconductor components
Address:	One Greenwich Place
	Shelton, CT 06484
Telephone:	1-402-563-6866
Internet:	www.vishay.com

Appendix B

MATLAB Source Code

B.1 run1.m

```
% run1.m

clear all
close all

% initialize car, position, speed, road, etc.
init;

i = 0;

while x0 <= 0.8*x_max
   i = i+1;

   % find error signal
   ef(i) = FindError(x0,y0,theta0,phi0,L,L,radius);
   eb(i) = FindError(x0,y0,theta0,phi0,L,0,radius);

   % determine array output based on car position
   front(i) = sensor(ef(i),FB_w,prev_front,sensors,spacing);
   back(i) = sensor(eb(i),RB_w,prev_back,sensors,spacing);

   % determine the car's angle
   theta_p(i) = FindHeadingAngle(ef(i),eb(i),L);            % actual error
   theta_p_hat(i) = FindHeadingAngle(front(i),back(i),L);   % discretized error

   % determine the curvature
   % actual curvature of path
   if x0 < -radius/sqrt(2)
      curv(i) = 0;
   else
      if x0 < radius/sqrt(2)
         curv(i) = sign(curv_sign)/radius;
      else
         curv(i) = 0;
      end
   end

   % estimate based on phi
   averaged_phi = sum(phi_s)/samples;
   if abs(averaged_phi) > 0.0326
      curv_hat(i) = -0.1599+4.8975*abs(averaged_phi);
   else
      curv_hat(i) = 0;
   end

   % estimate based on car dynamics
%   d = eb(i);    % actual error
   d = back(i);   % discrete error
   if i > 1
      theta_p_dot = (theta_p_hat(i)-theta_p_hat(i-1))/T;
   end
   THETA_P_DOT(i) = theta_p_dot;
   y = v1*tan(phi0)/L-theta_p_dot;
   w = v1*cos(theta_p(i))+v1*d*tan(phi0)/L-theta_p_dot*d;
   y_hat = w*a_hat;
   e = y_hat-y;
   E(i) = e;
   P = 1/(prev_P+w*w*T);
   prev_P = w*w*T;
   a_hat_dot = -w*e*P;
   a_hat = a_hat+a_hat_dot*T;

   A_hat(i) = a_hat;
   P_cum(i) = P;

   % real curvature
%   c = curv(i);

   % phi curvature estimate
%   if curv_hat(i) > 0.5
%      c = 1;
%   else
%      c = 0;
%   end

   % dynamic curve estimate
   if c == 0
      if abs(a_hat) > 0.9/radius
         c = sign(a_hat)/radius;
      end
   else
      if abs(a_hat) < 0.1/radius
         c = 0;
      end
   end

   C(i) = c
   c1 = 0;
   c2 = 0;

   % assign the states
   % actual error
   th = theta_p(i);
   d = eb(i);
   x2 = -c1*d*tan(th)-c*(1-d*c)*(1+sin(th)^2)/(cos(th)^2)+(1-d*c)^2*tan(phi0)/L*(cos(th)^3);
   x3 = (1-d*c)*tan(th);
   x4 = d;
   X2(i) = x2;
   X3(i) = x3;
   X4(i) = x4;

   % discretized error
   th = theta_p_hat(i);
   d = back(i);
   x2_hat = -c1*d*tan(th)-c*(1-d*c)*(1+sin(th)^2)/(cos(th)^2)+(1-d*c)^2*tan(phi0)/L*cos(th);
   x3_hat = (1-d*c)*tan(th);
   x4_hat = d;
   X2_hat(i) = x2_hat;
   X3_hat(i) = x3_hat;
   X4_hat(i) = x4_hat;

   % determine plant input
%  u2 = LateralController(x2,x3,x4,u1);              % actual error
   u2 = LateralController(x2_hat,x3_hat,x4_hat,u1);   % discretized error
   U1(i) = u1;
   U2(i) = u2;

   % transform the control inputs
%   th = theta_p(i);      % actual error
   th = theta_p_hat(i);   % discretized error
   dxds = -c2*d*tan(th)-(c1+2*d*c*c1)*((1+sin(th)*sin(th))/(cos(th)*cos(th)))-(2*(1-d*c)*d*c1*tan(phi0))/(L*(cos(th)^3));
   dxdd = c*c*(1+sin(th)*sin(th))/(cos(th)*cos(th))+2*c*(1-d*c)*tan(phi0)/(L*(cos(th))^3);
   dxdtheta = -c*(1-d*c)*4*tan(th)/(cos(th))^2+3*(1-d*c)^2*tan(phi0)*tan(th)/(L*(cos(th))^3);
   alpha1 = dxds+dxdd*(1-d*c)*tan(th)+dxdtheta*(tan(phi0)*(1-d*c)/(L*cos(th))-c);
   alpha2 = L*(cos(theta_p(i)))^3*(cos(phi0))^2/(1-d*c)^2;
   v1 = (1-d*c)*u1/cos(theta_p(i));
   v2 = alpha2*(u2-alpha1*u1);
   if v2 > pi/(2*T)
      v2 = pi/(2*T);
   end
   if v2 < -pi/(2*T)
      v2 = -pi/(2*T);
   end
   V1(i) = v1;
   V2(i) = v2;

   % update car dynamics
   [t_sim state output] = sim('dynamics',[0 T]);   % returns vectors x,y,theta,phi
   x0 = x(length(x));
   y0 = y(length(y));
   theta0 = theta(length(theta));
   phi0 = phi(length(phi));
   while theta0 > pi
      theta0 = theta0-2*pi;
   end
   while theta0 <= -pi
      theta0 = theta0+2*pi;
   end
   X(i) = x0;
   PHI = [PHI phi0];
   phi_s = phi_s(2:samples);
   phi_s = [phi_s phi0];
   mph(i) = v1/1000*0.62137*3600*10;

   % update animation
   locate(car,[x0,y0 0]);
   turn(car,'z',(theta0-theta0_prev)*180/pi);
   turn(car.tire_fl,'z',(phi0-phi0_prev)*180/pi);
   turn(car.tire_fr,'z',(phi0-phi0_prev)*180/pi);

   theta0_prev = theta0;
   phi0_prev = phi0;

   prev_front = front(i);
   prev_back = back(i);
   M(i) = getframe;
end
```

B.2 init.m

```
% init.m
% This subprogram initializes all car and road parameters.

% Constants
L = 10 *2.54/100;          % distance between rear wheel and front wheel (m)
W = 6.5 *2.54/100;         % space between wheels (m)
H = 2.5 *2.54/100/2;       % height (m)
D = 2.5 *2.54/100;         % diameter of wheels (m)
F = 1.25 *2.54/100;        % width of front wheels (m)
R = 2 *2.54/100;           % width of rear wheels (m)
FB_w = 2.5 *2.54/100;      % front bumper width (m)
RB_w = 2.5 *2.54/100;      % rear bumper width (m)
sensors = 12;              % number of sensors in a bumper
spacing = 0.2 *2.54/100;   % spacing between sensors (m)
T = 0.01;                  % sampling time (s)
radius = 1;                % radius of curvature of circular part (m)
x_max = 1.5;               % maximum x for path
samples = 10;              % number of samples used to filter phi
%*************************************************************

% Initial conditions (must be on the road!!!)
x0 = -x_max*radius;              % position of center of rear wheels along the x axis (m)
y0 = -x0+radius*(1-2/sqrt(2));   % position of center of rear wheels along the y axis (m)
theta0 = -39*pi/180;             % body angle (rad)
phi0 = 0*pi/180;                 % steering angle (limited -45 to 45 degrees) (rad)
u1 = 1.5;                        % v1 transformed (m/s)
mph = u1*0.62137*36
u2 = 0;                          % v2 transformed (rad/s)
phi_s = zeros(1,samples);        % used to average phi for curvature
a_hat = 0;                       % initial curvature estimate
theta_p_dot = 0;                 % initial angular velocity of the car
prev_P = 0;
v1 = u1;                         % assume this at the start (d=0,theta_p=0)
curv_sign = -sign(x0);

theta0_prev = theta0;
phi0_prev = phi0;

PHI = phi0;

c = 0;

prev_front = 0;
prev_back = 0;
%*************************************************************

hold on

% generate circular path
x_road = [-x_max*radius:T:x_max*radius]; % radius as defined above
for i = 1:length(x_road)
   if x_road(i) < -radius/sqrt(2)
      y_road(i) = -x_road(i)+radius*(1-2/sqrt(2));
   else
      if x_road(i) < radius/sqrt(2)
```

```
         y_road(i) = -sqrt(radius^2-x_road(i)^2)+radius;
      else
         y_road(i) = x_road(i)+radius*(1-2/sqrt(2));
      end
   end
end

plot(x_road,y_road)

dx = W;
dy = W;

view(2);
axis equal
axis([min(x_road)-dx max(x_road)+dx min(y_road)-dy max(y_road)+dy])
grid
%***************************************************************

% create car and locate it on the road
car = CreateCar(W,L,H,D,F,R);
locate(car,[x0 y0 0]);
aim(car,[x0+cos(theta0) y0+sin(theta0) 0]);
turn(car.tire_fl,'z',phi0*180/pi);
turn(car.tire_fr,'z',phi0*180/pi);

set(gcf,'renderer','OpenGL')
```

B.3 FindError.m

```
function error = FindError(x0,y0,theta0,phi0,l,dr,r)

x1 = x0+dr*cos(theta0);
y1 = y0+dr*sin(theta0);

xm = x0+cos(theta0);
ym = y0+sin(theta0);

m = (x1-xm)/(y1-ym);

% the path is circular
if x1 < -r/sqrt(2)
   x2 = (m*x1+y1-r*(1-2/sqrt(2)))/(m-1);
   y2 = -x2+r*(1-2/sqrt(2));
else
   if x1 < r/sqrt(2)
      if m < 0
         x2 = (2*m*(m*x1+y1-r)-sqrt(4*m^2*(m*x1+y1-r)^2-4*(m^2+1)*(m*x1*(m*x1+2*(y1-r))+y1*(y1-2*r))))/(2*(m^2+1));
      else
         x2 = (2*m*(m*x1+y1-r)+sqrt(4*m^2*(m*x1+y1-r)^2-4*(m^2+1)*(m*x1*(m*x1+2*(y1-r))+y1*(y1-2*r))))/(2*(m^2+1));
      end
      y2 = -sqrt(r^2-x2^2)+r;
   else
      x2 = (m*x1+y1-r*(1-2/sqrt(2)))/(m+1);
      y2 = x2+r*(1-2/sqrt(2));
   end
end

if (theta0 > -pi) & (theta0 <= -pi/2)
   if abs(y1-y2) >= 0.00001
      if y1 >= y2
```

```
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      end
   else
      if x1 >= x2
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      end
   end
end

if (theta0 > -pi/2) & (theta0 <= 0)
   if abs(y1-y2) >= 0.00001
      if y1 >= y2
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      end
   else
      if x1 >= x2
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      end
   end
end

if (theta0 > 0) & (theta0 <= pi/2)
   if abs(y1-y2) >= 0.00001
      if y1 >= y2
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      end
   else
      if x1 >= x2
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      end
   end
end

if (theta0 > pi/2) & (theta0 <= pi)
   if abs(y1-y2) >= 0.00001
      if y1 >= y2
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      end
   else
      if x1 >= x2
         error = -sqrt((x1-x2)^2+(y1-y2)^2);
      else
         error = sqrt((x1-x2)^2+(y1-y2)^2);
      end
   end
end
```

B.4 sensor.m

```
% sensor.m
function error = sensor(d,B_w,p,s,sp)

% IR sensors
prob = 1;
array = ones(s,1);

for k = -0.5*s:0.5*s-1
   if (d >= k*sp) & (d <= (k+1)*sp)
      array(s/2-k) = 0;
   end
   if rand > prob
      array(s/2-k) = 1-array(s/2-k);
   end
end

error = 0;
num = 0;
val = (s+1)/2;

for k = 1:s
   if array(k) == 0
      num = num+1;
    error = error+(val-k)*sp;
   end
end

if num ~= 0
   error = error/num;
else
   error = B_w/2*sign(p);
end
```

B.5 FindHeadingAngle.m

```
function angle = FindHeadingAngle(e_front,e_back,l)

angle = atan((e_front-e_back)/l); % radians
```

B.6 LateralController.m

```
function u2 = LateralController(x2,x3,x4,u1)
% Input scale controller using chained form
lambda = 8;
k1 = lambda^3;
k2 = 3*lambda^2;
k3 = 3*lambda;

u2 = -k1*abs(u1)*x4-k2*u1*x3-k3*abs(u1)*x2;
```

Appendix C

DSP Source Code

C.1 control.c

```
/************************************************
*  Format of the PIC control word:
*     bit      usage
*     ----     ---------------------------------
*     0..5  -  velocity or steering value
*     6     -  0 is steering, 1 is velocity
*     7     -  0 is disable, 1 is enable
************************************************/

#include <math.h>

//#define PHI_EST
#define MODEL_EST
//#define STEP_C


/*** FUNCTION PROTOTYPES ***/
extern void PutMem(long mem,long output,long offset);
extern void GetMem(long mem,long input,long offset);
float FindError(long array, long bits, float spacing, float max, float p_error);
float LateralController(float X2, float X3, float X4, float U1);
long Round(float x);
long Sign(float x);
float FindHeadingAngle(float f_error, float b_error, float length);


/*** CONSTANTS ***/
const long mem = 0x00A00;  // External memory location, must get shifted left by 4
#define s_bit 0x00         // To clear bit 5 of the control word
#define v_bit 0x40         // To set bit 5 of the control word
#define e_bit 0x80         // To set bit 6 of the control word
#define d_bit 0x00         // To clear bit 6 of the control word
#define IR_spacing 0.00508 // (meters)
#define IR_bits 12         // Number of IR sensors in one bumper
#define IR_max 0.03302     // Half the bumper width (meters)
#define L 0.3048           // Distance between front and rear sensors (meters)
#define u1 1.0             // Transformed velocity (m/s)
#define T 0.000190         // Sampling time (seconds)
```

```
#define curvature 1.125    // Actual curvature of track (1/m)
#define threshold 600      // Threshold for a_hat
#define c_step 0.0003      // Amount by which to step c


/*** VARIABLES ***/
long offset;         // Add to mem, cycles through all devices
long input;          // Stuff read in from external memory
long output;         // Stuff sent out to external memory
long left_bank;      // Leftmost sensors
long right_bank;     // Rightmost sensors
long front_array;    // Front bumper array
long back_array;     // Rear bumper array
float front_error;   // Error determined by the front bumper (meters)
float p_front_error;
float back_error;    // Error determined by the rear bumper (meters)
float p_back_error;
float phi;           // Calculated steering angle
long angle;          // Steering value sent to car (0-63), 32 is center
long velocity;       // Velocity value sent to car (0-31), 2 is neutral
float theta_p;       // Heading angle (radians)
float c;             // Curvature (1/meters)
float x2,x3,x4;      // Chained form variables
float u2;            // Transformed steering angular velocity
float v1,v2;         // Actual control inputs (rad/s,m/s)
float d;             // Lateral distance from line (meters)
float dxdd,dxdtheta;
float alpha1,alpha2;
float temp;
float curv_hat;      // Estimate of the curvature
long i;              // Loop counter
float a_hat;         // Used by model estimator
float a_hat_dot;
float theta_p_dot;   // Derivative of theta_p
float theta_p_prev;  // Previous value of theta_p
float w;
float y;
float y_hat;
float P;
float P_prev;
float e;
float v1_actual;
float u1_actual;
long high_count,mid_count,low_count;


/*** MAIN PROGRAM ***/
void main(void)
{
   /*** variable initialization ***/
   p_front_error = 0;
   p_back_error = 0;
   angle = 32;
   phi = 0;
   v1 = u1;
   c = 0;
   a_hat = 0;
   theta_p_prev = 0;
   P_prev = 0;
   high_count = 0;
   mid_count = 0;
   low_count = 0;

   /*** main loop ***/
```

```
while (1)
{
   /** IR data **/
   offset = 0;                // enable IR
   GetMem(mem,input,offset);  // read bumper data

   /* correct the order of the bits */
   left_bank = input&0x00FF;
   left_bank = left_bank<<8;
   right_bank = input&0xFF00;
   right_bank = right_bank>>8;
   front_array = left_bank|right_bank;
   front_array = front_array>>2;
   front_array = front_array&0xFFF;

   input = input>>16;
   left_bank = input&0x00FF;
   left_bank = left_bank<<8;
   right_bank = input&0xFF00;
   right_bank = right_bank>>8;
   back_array = left_bank|right_bank;
   back_array = back_array>>2;
   back_array = back_array&0xFFF;
   /*********************************/

   /** car control **/
   /* determine error in front and back */
   front_error = FindError(front_array,IR_bits,IR_spacing,IR_max,p_front_error);
   back_error = FindError(back_array,IR_bits,IR_spacing,IR_max,p_back_error);

   velocity = Round(v1*10.2666);
   if( velocity < 4 )
   {
      velocity = 4;
   }
   else if( velocity > 63 )
   {
      velocity = 63;
   }

   output = e_bit | v_bit | velocity;
   offset = 1;                // enable car
   PutMem(mem,output,offset); // load values to PIC PSP
   offset = 3;                // PIC interrupt triggers on rising edge
   PutMem(mem,output,offset);

   /* determine the car's angle */
   theta_p = FindHeadingAngle(front_error,back_error,L);

   d = back_error;

   offset = 2;                // enable PIC feedback
   GetMem(mem,input,offset);  // read velocity data
   v1_actual = input/10.2666;
   u1_actual = v1_actual*cos(theta_p)/(1-d*c);

   /* determine the curvature */
   /* model estimate */
   theta_p_dot = (theta_p-theta_p_prev)/T;
   y = v1_actual*tan(phi)/L-theta_p_dot;
   w = v1_actual*cos(theta_p)+v1_actual*d*tan(phi)/L-theta_p_dot*d;
   y_hat = w*a_hat;
   e = y_hat-y;
   P = 1/(P_prev+w*w*T);
```

```
P_prev = w*w*T;
a_hat_dot = -w*e*P;
a_hat = a_hat+a_hat_dot*T;

#ifdef PHI_EST
c = 5*phi;
if (c > 0.5*curvature)
{
   high_count = high_count+1;
   mid_count = 0;
   low_count = 0;
}
else
{
   if (c < -0.5*curvature)
   {
      high_count = 0;
      mid_count = 0;
      low_count = low_count+1;
   }
   else
   {
      high_count = 0;
      mid_count = mid_count+1;
      low_count = 0;
   }
}

if (high_count > threshold)
{
   c = curvature;
}
if (mid_count > threshold)
{
   c = 0;
}
if (low_count > threshold)
{
   c = -curvature;
}
#endif

#ifdef MODEL_EST
/* Part I */
if (c == 0)
{
   if (a_hat > 0.9*curvature)
   {
      high_count = high_count+1;
      mid_count = 0;
      low_count = 0;
   }
   if (a_hat < -0.9*curvature)
   {
      high_count = 0;
      mid_count = 0;
      low_count = low_count+1;
   }
}
else
{
   if ((a_hat < 0.1*curvature) && (a_hat > -0.1*curvature))
   {
      high_count = 0;
      mid_count = mid_count+1;
      low_count = 0;
   }
}

if (high_count > threshold)
{
   c = curvature;
}
if (mid_count > threshold)
{
   c = 0;
}
if (low_count > threshold)
{
   c = -curvature;
}
#endif

#ifdef STEP_C
/* Part I with c changing in steps */
if (a_hat > 0)
{
   c = c+c_step;
   if (c > curvature)
   {
      c = curvature;
   }
}
else
{
   c = c-c_step;
   if (c < -curvature)
   {
      c = -curvature;
   }
}
#endif

/* assign the states */
x2 = -c*(1-d*c)*(1+sin(theta_p)*sin(theta_p))/
      (cos(theta_p)*cos(theta_p))+
      (1-d*c)*(1-d*c)*tan(phi)/
      (L*cos(theta_p)*cos(theta_p)*cos(theta_p));
x3 = (1-d*c)*tan(theta_p);
x4 = d;

/* determine the plant input */
u2 = LateralController(x2,x3,x4,u1_actual);

/* transform the control inputs */
dxdd = c*c*(1+sin(theta_p)*sin(theta_p))/
      (cos(theta_p)*cos(theta_p))-
      2*(1-d*c)*c*tan(phi)/(L*cos(theta_p)*cos(theta_p)*cos(theta_p));
dxdtheta = -c*(1-d*c)*4*tan(theta_p)/(cos(theta_p)*cos(theta_p))+
           3*(1-d*c)*(1-d*c)*tan(phi)*tan(theta_p)/
           (L*cos(theta_p)*cos(theta_p)*cos(theta_p));

alpha1 = dxdd*(1-d*c)*tan(theta_p)+
         dxdtheta*(tan(phi)*(1-d*c)/(L*cos(theta_p))-c);
alpha2 = L*cos(theta_p)*cos(theta_p)*cos(theta_p)*cos(phi)*cos(phi)/
         ((1-d*c)*(1-d*c));

v1 = (1-d*c)*u1/cos(theta_p);
```

```
      v2 = alpha2*(u2-alpha1*u1_actual);

      phi = phi+v2*T;
      if (phi > 0.7854)
      {
         phi = 0.7854;
      }
      if (phi < -0.7854)
      {
         phi = -0.7854;
      }

      angle = Round(-40.107*phi+31.5);
      if (angle < 0)
      {
         angle = 0;
      }
      if (angle > 63)
      {
         angle = 63;
      }

      output = e_bit | s_bit | angle;
      offset = 1;                // enable car
      PutMem(mem,output,offset); // load values to PIC PSP
      offset = 3;                // PIC interrupt triggers on rising edge
      PutMem(mem,output,offset);

      p_front_error = front_error;
      p_back_error = back_error;
      theta_p_prev = theta_p;
      a_hat_prev = a_hat;

   }  /* end while loop */
}  /* end main */


/**************************************************************************
*  Function:   FindError
*  Parameters: array - the bits read from the bumper
*              bits - the number of individual sensors in the bumper
*              spacing - the distance between the sensors (in meters)
*  Returns:    error - the distance (in inches) that the line is off center
*              (if the line is to the right, error is positive)
**************************************************************************/
float FindError(long array, long bits, float spacing, float max, float p_error)
{
   static long i;
   static long mask;
   static long current;
   static long num;
   static float error;
   static float val;

   error = 0;
   num = 0;
   val = (bits-1)/2;

   for (i = 0; i < bits; ++i)
   {
      mask = 0x0001<<i;
      current = mask & array;
      if (current == 0)
      {
         error = error+(val-i)*spacing;
         num = num+1;
      }
   }

   if (num == 0)
   {
      error = Sign(p_error)*max;
   }
   else
   {
      error = error/(float)(num);
   }

   return(error);
}


/***************************************************************************
*  Function:   LateralController
*  Parameters: chained state variables x2,x3,x4 and the car's transformed
*              velocity u1
*  Returns:    U2 - the tranformed angular velocity
***************************************************************************/
float LateralController(float X2, float X3, float X4, float U1)
{
   static float U2;        // steering angular velocity and a very good rock band
   static float k1,k2,k3;  // gains
   #define lambda 20

   k1 = 10*lambda*lambda;
   k2 = 3*lambda*lambda;
   k3 = 3*lambda;

   U2 = -k1*U1*X4-k2*U1*X3-k3*U1*X2;

   return(U2);
}


/***************************************************************************
*  Function:   Round
*  Parameters: x - the (float) number to round
*  Returns:    temp - rounded integer value of x
***************************************************************************/
long Round(float x)
{
   static long temp;
   static float y;

   temp = (long)(x);
   if (x >=0)
   {
      y = x-temp;
      if (y >= 0.5)
      {
         temp = temp+1;
      }
   }
   else
   {
      y = temp-x;
      if (y >= 0.5)
      {
         temp = temp-1;
      }
   }

   return(temp);
}


/*************************************************************************
*  Function:   Sign
*  Parameters: x - the (float) number to take the sign of
*  Returns:    1 if x>0, -1 if x<0, zero otherwise
*************************************************************************/
long Sign(float x)
{
   if (x == 0)
   {
      return(0);
   }
   else
   {
      if (x > 0)
      {
         return(1);
      }
      else
      {
         return(-1);
      }
   }
}


/*************************************************************************
*  Function:   FindHeadingAngle
*  Parameters: f_error - error in meters from the front sensors
*              b_error - error in meters from the back sensors
*              length - the distance between the front and rear sensors (meters)
*  Returns:    theta_p - heading angle in radians
*              (if the line is to the right, angle is positive)
*************************************************************************/
float FindHeadingAngle(float f_error, float b_error, float length)
{
   static float theta_local;

   theta_local = atan( (f_error-b_error)/length );
   return(theta_local);
}
```

C.2 PutMem.asm

```
   .global _mem,_output,_offset
   .global _PutMem

_PutMem  LDP   800000h,DP     ;load 8 MSBs of address

         LDI   @_mem,AR0      ;load memory location into auxillary
         LSH   4,AR0          ;register.
         ADDI  @_offset,AR0
```

```
        PUSH  R0
        LDI   @_output,R0    ;send stuff to external
        STI   R0,*AR0        ;memory.
        POP   R0

        RETS
```

C.3 GetMem.asm

```
   .global _mem,_input,_offset
   .global _GetMem

_GetMem  LDP   800000h,DP     ;load 8 MSBs of address

        LDI   @_mem,AR0      ;load memory location into auxillary
        LSH   4,AR0          ;register.
        ADDI  @_offset,AR0

        PUSH  R0
        LDI   *AR0,R0        ;read stuff in from external memory
        STI   R0,@_input     ;store it in the variable "input"
        POP   R0

        RETS
```

Appendix D

PIC Source Code

```
list p=16f874   ; set processor type
list n=0        ; supress page breaks in list file
include <P16f874.INC>

;Version 1.4
;
;Version 1.4 uses the following command format:
;   bits   meaning
;   0-5    command value for steering or velocity
;   6      0 = steering
;          1 = velocity
;   7      0 = disable
;          1 = enable

char1         equ   0x20
char2         equ   0x21
char3         equ   0x22
char4         equ   0x23
Flag          equ   0x24
speed         equ   0x25
angle         equ   0x26
temp          equ   0x27
temp2         equ   0x28
steercnt      equ   0x29
STATUS_TEMP   equ   0x2a
W_TEMP        equ   0x2b
hold          equ   0x2c
pass2         equ   0x2d
spdtmp        equ   0x2e
temp3         equ   0x2f
temp4         equ   0x30
command       equ   0x31
thold         equ   0x32
desspeed      equ   0x33
loopcnt       equ   0x34
stemp         equ   0x35
remain        equ   0x36
diff          equ   0x37
total         equ   0x38
ptotal        equ   0x39
d_err         equ   0x40
loop_0        equ   0x41
```

```
#DEFINE   MOTORPIN   PORTC,5
#DEFINE   SERVOPIN   PORTC,4
#DEFINE   HEART      PORTC,3

;************************************************************
; Current I/O Pinout
; PortA
;   1,2 - Status lights
;   0,3-5 - Not in use
; PortB
;   0-7 - Speed output to DSP
; PortC
;   0 - Encoder
;   1,2 - Not in use
;   3 - Heartbeat
;   4 - Servo Control
;   5 - Motor Control
;   6,7 - Serial Reciever
; PortD
;   0-7 - Speed & Steering input from DSP
; PortE
;   0 - 5V
;   1,2 - Car Enable
;************************************************************

;************************************************************
; Reset and Interrupt Vectors

   org 00000h ; Reset Vector
   goto Start

   org 00004h ; Interrupt vector
   goto IntVector

;************************************************************
; Program begins here

   org 00020h ; Beginning of program EPROM
Start
; Initialize variables
   movlw 0x00
   movwf Flag
   movwf hold
   movwf desspeed
   movwf diff
   movwf ptotal
   movwf total
   movlw d'117'; 28 is the value that produces 1.5ms
   movwf steercnt ; pulse width for centering servos
   movlw d'31'
   movwf speed
   clrf thold
   movlw d'2'
   movwf loopcnt
   movlw d'4'
   movwf loop_0

; Set up tmr0 for SCP
   bsf STATUS,RP0
   movlw 0xd5 ; set TMR0 for prescaler=256
   movwf OPTION_REG
   movlw 0xa0 ;enable global and TMR0 interrupt
   movwf INTCON
   bcf STATUS,RP0
```

```
; Set up timer 1 to count encoder "Up" pulses
   clrf TMR1L
   clrf TMR1H

   movlw 0x07
   movwf T1CON

; Set up TMR2 for use as the PWM generator
; for the velocity servo
   movlw d'156'; Set PWM frequency
   bsf STATUS,RP0 ; to 76Hz
   movwf PR2
   bcf STATUS,RP0

   clrf T2CON ; clear T2CON
   clrf TMR2 ; clear Timer2
   movlw 0x0F ; Enable TMR2 and set prescaler= 16
   movwf T2CON ; postscalar=4

   clrf CCP1CON ; CCP module is off
   clrf CCP2CON ; CCP module is off
; Modules must be off to enable
; PORTC 1,2 as outputs

   bsf STATUS,RP0
   bsf TRISC,0
   bcf STATUS,RP0

; Set up the parallel slave port to allow the DSP to communicate with
; the PIC. This segment also configures the serial port for 9600 Baud
; for use in manual driving.  Manual mode has been left in for debug
; purposes and will soon be removed
   bsf STATUS,RP0
   movlw 0x17 ;enable the PSP and configure
   movwf TRISE ;port e as inputs
   movlw 0x00 ;set port a to output
   movwf TRISA
   movlw 0x06
   movwf ADCON1 ; configure port a as digital i/o
   bcf STATUS,RP0

   clrf PORTB ; Clear PORTB output latches

   bsf STATUS,RP0
   clrf TRISB ; Config PORTB as all outputs
   bcf TRISC,3 ; Make RC3,4, and 5 an outputs
   bcf TRISC,4
   bcf TRISC,5
   bsf TRISC,7
   movlw 81h ; 9600 baud @20MHz
   movwf SPBRG
   bsf TXSTA,TXEN ; Enable transmit
   bsf TXSTA,BRGH ; Select high baud rate
   bcf STATUS,RP0

   bsf RCSTA,SPEN ; Enable Serial Port
   bsf RCSTA,CREN ; Enable continuous reception

   bcf PIR1,RCIF ; Clear RCIF Interrupt Flag
   bcf PIR1,PSPIF ; Clear PSP Interrupt Flag
   bcf PIR1,TMR2IF ; Clear the TMRP2 Interrupt

   bsf STATUS,RP0
   bsf PIE1,RCIE ; Set RCIE Interrupt Enable
   bsf PIE1,PSPIE ; Set PSP Interrupt Enable
   bsf PIE1,TMR2IE ; Set TMR2 Interrupt Enable
   bcf STATUS,RP0

   bsf INTCON,PEIE ; Enable peripheral interrupts
   bsf INTCON,GIE ; Enable global interrupts

   bcf PORTA,0

MainLp
   nop
   btfss Flag,1 ; Until serial data has been received
   goto MainLp ; Loop here
Stop
   bcf Flag,1
   btfss Flag,0 ; if char1 was a Carriage Return
   goto NoCR
   bcf Flag,0
; decode what was sent
   movf char2,0 ; is char2 a letter or a number?
   andlw 0xf0
   xorlw 0x30
   btfss STATUS,Z
   goto OneLet
   movlw 0x30 ;tens digit first
   subwf char3,0
   movwf temp
   movwf temp2
   bcf STATUS,C
   rlf temp2,1 ; (x<<2|x)<<1 = x*10
   rlf temp2,0
   addwf temp,1
   bcf STATUS,C
   rlf temp,1
   movlw 0x30
   subwf char2,0 ; add ones digit
   addwf temp,1 ; temp has 0 to 99 number
   movf char4,0
   sublw 'v'; we've got a number
; is it speed or steering?
   btfss STATUS,Z
   goto CheckS
   movf temp,0 ; if it was velocity, put the value
   movwf spdtmp ; in the control
   goto EndCheck
CheckS
   movf char4,0
   sublw 's'; else if it was steering, put the value
   btfss STATUS,Z
   goto BadCom
   movf temp,0
   movwf angle ; in the steering control
   goto EndCheck
OneLet ; else we have a one letter control
   movf char2,0 ; word
   sublw 'e'; if it was enable
   btfss STATUS,Z
   goto CheckD
   bcf PORTA,1 ; enable the vehicle
   goto EndCheck
CheckD movf char2,0 ;else if it was disable
   sublw 'd'
   btfss STATUS,Z
   goto CheckM
   bsf PORTA,1 ; disable the vehicle
   goto EndCheck
CheckM movf char2,0
   sublw 'm';else if it was manual
   btfss STATUS,Z
   goto CheckA
   bsf PORTA,0
   bcf PIR1,PSPIF ; enable manual mode by
   bcf PIE1,PSPIE ; Disabling PSP Interrupt
   bsf STATUS,RP0
   movlw 0x00
   movwf TRISE ; and set the PORTE pins to high impedence
   bcf STATUS,RP0
   goto EndCheck
CheckA movf char2,0
   sublw 'a'; else it was auto mode
   btfss STATUS,Z
   goto EndCheck
   bcf PORTA,0 ; put the vehicle in auto mode
   bsf STATUS,RP0
   movlw 0x17 ; make PORTE inputs
   movwf TRISE
   bcf STATUS,RP0
   bcf PIR1,PSPIF
   bsf PIE1,PSPIE ; Enable PSP interrupt in auto

EndCheck

NoCR
   bsf HEART ; heart beat
   nop
   nop
   nop
   nop
   nop
   bcf HEART
   nop
   nop
   nop
   nop
   nop
   btfsc PORTA,1 ; see if vehicle is enabled
   goto disabled
   btfss PORTA,0
   goto auto ; see if vehicle is in auto mode

   movf spdtmp,0
   addlw d'31'
   movwf speed ; set velocity PWM
   movf angle,0
   addlw d'18'
   movwf steercnt ; set steering PWM
   goto MainLp
disabled
   movlw d'31'; if vehicle is disabled
   movwf speed ; stop moving
   movlw d'117'; and center steering
   movwf steercnt

auto ; else if we are in auto,
NoFlag goto MainLp ; let the DSP direct the vehicle

IntVector
; save Status and W registers
```

```
   movwf W_TEMP  ;Copy W to TEMP register
   swapf STATUS,W ;Swap status to be saved into W
   clrf STATUS  ;bank 0, regardless of current bank, Clears IRP,RP1,RP0
   movwf STATUS_TEMP ;Save status to bank zero STATUS_TEMP register
   bcf INTCON,2 ; clear interrupt
; determine which interrupt occurred

   btfss PIR1,RCIF ; Did USART cause interrupt?
   goto TMR0Int ; No, some other interrupt

SERInt
   movlw 0x06 ; Mask out unwanted bits
   andwf RCSTA,W ; Check for errors
   btfss STATUS,Z ; Was either error status bit set?
   goto RcvError ; Found error, flag it

   movf char3,0 ; wait for four bytes
   movwf char4
   movf char2,0
   movwf char3
   movf char1,0
   movwf char2
   movf RCREG,W ; Get input data
   movwf TXREG ; Echo character back
   movwf char1
   sublw 0x0d
   btfss STATUS,Z
   goto Ret
   bsf Flag,0
   movlw 0x0a
   movwf TXREG
   movfw char1
   sublw 'm'; if we are not in manual
   btfss STATUS,Z
   goto Ret
   bsf PORTA,0 ; then put the vehicle in auto mode
   goto Ret ; go to end of ISR, restore context, return

RcvError
   movf RCREG,0
   bcf RCSTA,CREN ; Clear receiver status
   bsf RCSTA,CREN
   goto Ret ; go to end of ISR, restore context, return


TMR0Int
   btfss INTCON,T0IF   ; see if the interrupt was TMR0
   goto PSPInt

   bcf INTCON,2 ; Clear the interrupt
   bsf Flag,1

   decfsz loop_0
   goto Ret

   btfss hold,0 ; See if we are rising or falling
   goto soff

   movfw steercnt ; set TMR0 to put the steering
   sublw d'255'; pulse high for 255-steercnt ticks
   movwf TMR0

   bsf SERVOPIN ; output a high
   bcf hold,0 ; toggle the hold
   incf loop_0
   goto Ret

soff
   movlw d'4'
   movwf loop_0

   movfw steercnt
   movwf stemp
   movlw 0xF8
   andwf stemp
   bcf STATUS,C
   rrf stemp,1
   rrf stemp,1
   rrf stemp,0
   sublw d'139'
   sublw d'255'; set TMR0 to put the velocity pulse
   movwf TMR0 ; low for the remainder of the
; period

   bcf SERVOPIN ; output a low
   bsf hold,0 ; toggle the hold

   goto Ret

PSPInt
   btfss PIR1,PSPIF ; see if the interrupt was the PSP
   goto TMR2Int

   bcf PIR1,PSPIF ; Clear the interrupt

   movf PORTD,0 ; Read in the data from PSP
   movwf temp3

   nop
   btfss temp3,7 ; check to see if DSP is enabling or
   goto DSPdisable ; disabling car
   bcf PORTA,1 ; enable the vehicle
   goto  GetCmd

DSPdisable
   bsf PORTA,1 ; disable the vehicle
   movlw d'31'; center steering and
   movwf speed ; set velocity to 0
   movlw d'117'
   movwf steercnt
   goto Ret

GetCmd
   movwf temp3 ; put the value in temp3
   movwf temp4 ; Mask the command data value
   movlw 0x3f ; and hold the result in temp3
   andwf  temp3,1

   btfss temp4,6 ; see if it was steering or velocity
   goto Steer

   movf temp3,0
   andlw 0x3f ; extract lower 6 bits (velocity)
   movwf desspeed
   goto Ret

Steer
   movfw temp3 ; if it was steering, offset the
   addlw d'85'; command from the DSP by 18
   movwf steercnt ; 0-9 from the DSP are left, 10 is center
   goto Ret ; 11-20 are right

TMR2Int
   btfss PIR1,TMR2IF ; see if it was the TMR2 interrupt
   goto Ret

   bcf PIR1,TMR2IF ; Clear the interrupt

   decfsz loopcnt,1 ; We are sampling at 1KHz, and commanding
   goto sample ; the motor at 100Hz, so we must wait
; for 9 sample periods before we can
; command the motor

   btfsc thold,0 ; see if we are rising or falling
   goto toff

ton
   bsf thold,0 ; pulse width is high for 1 1ms sample
   movfw speed
   btfsc STATUS,Z
   goto arbit

   bsf STATUS,RP0 ; period register.
   movwf PR2
   bcf STATUS,RP0

   sublw d'156'; keep the remainder of the 156 = 1ms
   movwf remain ; pulse where the motor signal is low
   incf loopcnt,1
   bsf MOTORPIN ; start the high pulse
   goto Ret

arbit
   movlw 0x0A
   movwf loopcnt
   bcf thold,0
   bcf MOTORPIN
   movlw d'156'
   bsf STATUS,RP0 ; period register.
   movwf PR2
   bcf STATUS,RP0
   goto Ret

toff
   bcf thold,0 ; put the remainder of the 1ms in the
   movfw remain ; period register
   bsf STATUS,RP0
   movwf PR2
   bcf STATUS,RP0

   movlw d'10'; wait for 10 sample periods before
   movwf loopcnt ; commanding the motor again
   bcf MOTORPIN ; turn the motor pulse off
   goto Ret

sample
   movlw d'156'; set the timer for 1ms sample period
   bsf STATUS,RP0 ; 1ms = 156 * 200ns * 32(prescale)
   movwf PR2
   bcf STATUS,RP0

   movfw loopcnt ; if we are going to set the speed in
   sublw 0x01 ; in the next sample period, then
   btfsc STATUS,Z ; get it ready now
   bsf thold,1

   btfss thold,1 ; if we are not commanding the motor
   goto encoder ; then get a sample from the encoder

   bsf MOTORPIN ; set the motor pulse high for 1ms
   bcf thold,1

   btfsc total,7 ; negative encoder ticks, bail
   goto t2on

   movfw total
   subwf ptotal,1
   movwf d_err
   movfw total
   movwf PORTB ; sends the true speed to PORTB and compares it
   subwf desspeed,0 ; with the desired speed
   movwf diff
   btfsc diff,7
   goto dec_speed
   goto inc_speed

dec_speed
   comf diff,1
   incf diff,1
   movlw 0xFC
   andwf diff,1
   bcf STATUS,C
   rrf diff,1
   rrf diff,1
   comf diff,1
   incf diff,1
   goto der_err

inc_speed
   movlw 0xFC
   andwf diff,1
   bcf STATUS,C
   rrf diff,1
   rrf diff,1
   goto der_err

der_err
   movfw total
   movwf ptotal
   btfsc d_err,7
   goto dec_d_err
   goto inc_d_err

dec_d_err
   comf d_err,1
   incf d_err,1
   movlw 0xF8
   andwf d_err,1
   bcf STATUS,C
   rrf d_err,1
   rrf d_err,1
   rrf d_err,1
   comf diff,1
   incf diff,1
   goto adj_speed
```

```
inc_d_err
   movlw 0xF8
   andwf d_err,1
   bcf STATUS,C
   rrf d_err,1
   rrf d_err,1
   rrf d_err,1
   goto adj_speed

adj_speed
   clrf total
   movfw diff
   addwf d_err,1
   movfw d_err
   addwf speed,1
   btfss d_err,7
   goto high_test
   goto low_test

low_test
   btfsc speed,7
   goto c2
   goto t2on

high_test
   btfsc speed,7
   goto clamp
   goto t2on

encoder
   bcf T1CON,TMR1ON ; turn the timer off
   movfw TMR1L ; load the encoder count
   addwf total,1
   goto t2on

clamp
   movlw d'127'; clamp the high speed at 1.1ms
   movwf speed
   goto t2on

c2
   movlw d'0'; clamp the low speed at 1.5ms
   movwf speed
   goto t2on

t2on
   clrf TMR1L ; reset
   clrf TMR1H
   bsf T1CON,TMR1ON ; turn the timer back on

Ret
   swapf STATUS_TEMP,0 ;Swap STATUS_TEMP register into W
;(sets bank to original state)
   movwf STATUS  ;Move W into STATUS register
   swapf W_TEMP,1  ;Swap W_TEMP
   swapf W_TEMP,0  ;Swap W_TEMP into W
   retfie

   end
```

Appendix E

FLASH Images

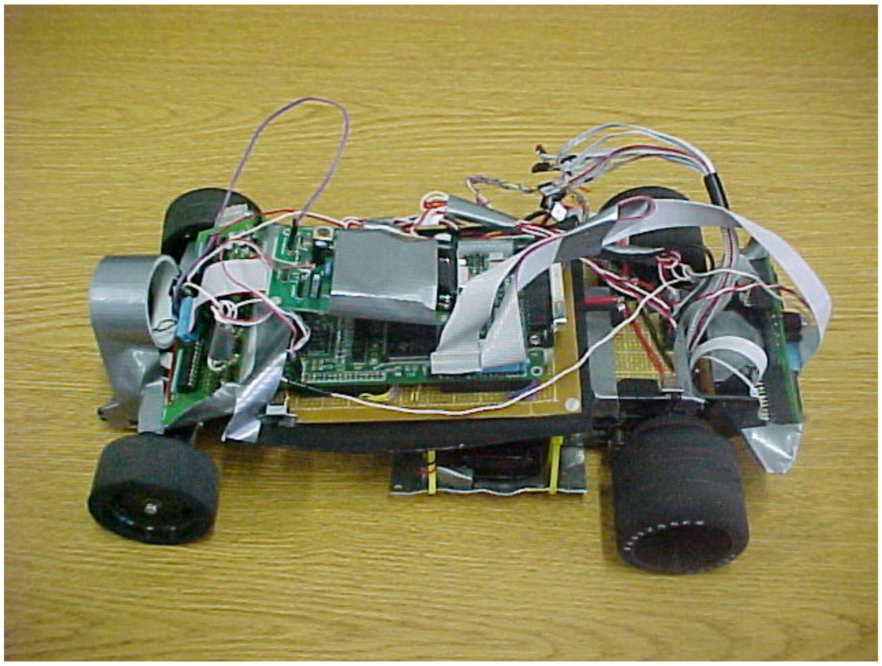

Figure E.1: FLASH vehicle prototype number 1.

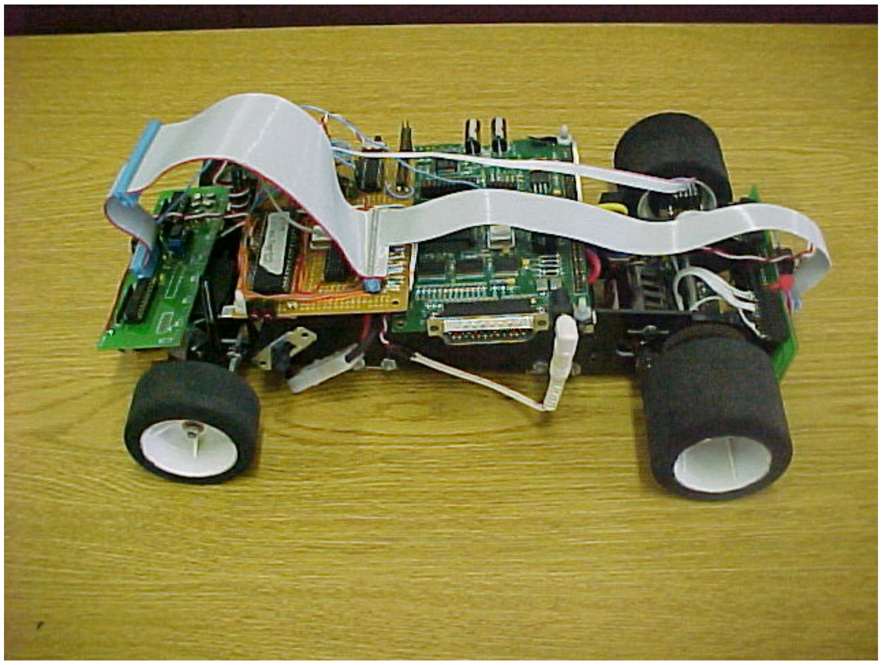

Figure E.2: FLASH vehicle prototype number 2.

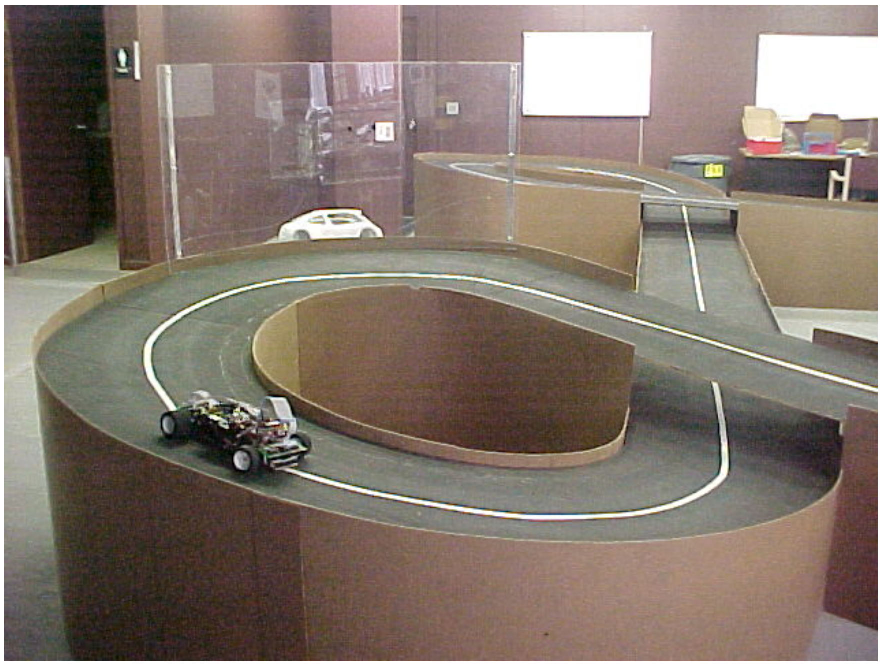

Figure E.3: The FLASH lab.

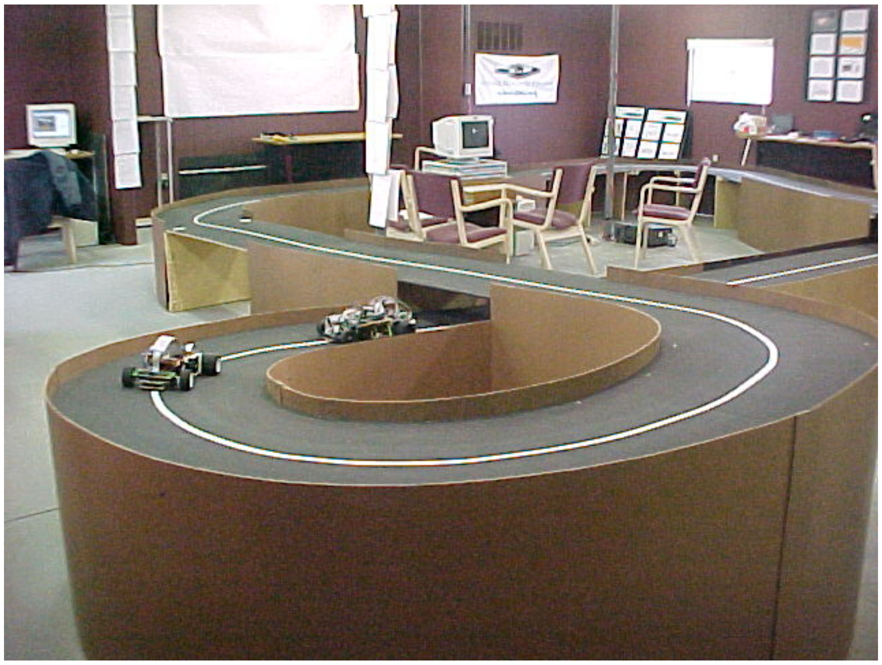

Figure E.4: Another view of the FLASH lab.

Vita

Patricia Mellodge is a native of Barrington, RI and graduated from Barrington High School in 1994. In 1998, she received her Bachelor's degree in Electrical Engineering from the University of Rhode Island in Kingston, RI. Upon graduating, Patricia took a job with Optigain, Inc. in Peacedale, RI where she designed analog circuits for use in the company's fiber optic amplifiers. In the fall of 2000, Patricia moved to Blacksburg, VA to join the Bradley Department of Electrical Engineering at Virginia Tech to pursue a Master's degree. She received her Master's degree in May 2002. Her areas of research interest include feedback control and signal processing and in addition she thinks that Rush is the greatest rock band of all time.