Automatic Ultrasonic Headway Control for a Scaled Robotic Car

By

Richard D. Henry

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
COMPUTER ENGINEERING

APPROVED

____________________
Dr. Pushkin Kachroo, Chair

____________________    _____________________
Ray Pethtel                  Dr. Hugh VanLandingham

December 18, 2001

Blacksburg, Virginia

Keywords: Automatic, Control, Headway, Robotic, Ultrasound
(ABSTRACT)

Intelligent Transportation Systems and supporting technologies have been an active area of research for some time. Human drivers exhibit slower response times and errors in judgment that can have serious adverse affects on traffic flow. These types of errors can be reduced or eliminated from the driving experience by introducing computer control systems into the automotive arena.

The purpose of this research was to develop a scale model platform for the rapid prototyping and testing of ITS systems and technologies. Specifically, this body of work was concerned with the development of an automatic headway control system that utilized ultrasonic sensors. This control system was intended to automatically maintain headway distance in an effort to create an adaptive cruise control system for this scale model vehicle. Implementation of such systems could conceivably reduce driver fatigue by removing the burden of maintaining safe following distance from the driver.
System dynamics of car-like robots with nonholonomic constraints were employed in this research to create a controller for an autonomous path following vehicle. The application of a working kinematic model describing car-like robotic systems allowed the development of a simple first order controller, as well as a sliding mode controller.

Following the development and simulation of these two control laws, the system was applied to the FLASH project scale model vehicle to assess the practical use of the system on a mock highway. A satisfactory result is produced after testing was completed, and the application of such systems to scale model platforms is feasible.
# TABLE OF CONTENTS

Chapter 1.0 Introduction .................................................................................................. 1

Chapter 2.0 Motivation ................................................................................................. 5

Chapter 3.0 Hardware Environment .............................................................................. 9
  3.1 Model Vehicle Platform ...................................................................................... 10
  3.2 Digital Signal Processor Platform ...................................................................... 13
  3.3 PIC Processor Platform .................................................................................... 17
  3.4 Infrared Sensor Arrays ..................................................................................... 24
  3.5 Hall Effect Sensor Arrays ................................................................................. 26
  3.6 Visual System .................................................................................................. 28
  3.7 Ultrasonic Sensors ........................................................................................... 30

Chapter 4.0 Software Platform .................................................................................... 33
  4.1 Programming in C for the TMS320C31 ............................................................. 33
  4.2 C Program Code Description .......................................................................... 41
  4.3 The PIC16F874 Microcontroller ...................................................................... 43
  4.4 PIC Assembly Program Description .................................................................. 46

Chapter 5.0 Mathematical Modeling and Simulation Results .................................... 52
  5.1 The Kinematic Model ....................................................................................... 52
  5.2 Headway controller algorithm development .................................................... 57
  5.3 Matlab Simulation Results ............................................................................. 60

Chapter 6.0 Application of the Control Laws .............................................................. 84
  6.1 Application of the first order control law ......................................................... 84
  6.2 Application of the sliding mode control law .................................................... 87

Chapter 7.0 Conclusions and Future Work ................................................................. 89
  7.1 Conclusions ................................................................................................... 89
  7.2 Future work .................................................................................................. 90

Appendix A C Control Program Source Code Listing .............................................. 93
Appendix B PIC Microcontroller source program listing ........................................ 100
Appendix C Matlab simulation source code ......................................................... 110
Appendix D Hardware Sources ............................................................................ 117
Appendix E Images of the FLASH Project Model Vehicle ..................................... 120

References: ................................................................................................................. 125
LIST OF FIGURES
Figure 1.1 Block diagram of a cruise control system [17] ................................................ 3
Figure 2.1 Sample museum exhibit for the FLASH project [24] ........................................... 6
Figure 2.2 Safety poster illustrations from the FLASH project [24] ........................................ 7
Figure 3.0.1 High-level architecture block diagram of the vehicle hardware platform .......... 9
Figure 3.1.1 PWM control signal format for steering and velocity ................................... 12
Figure 3.2.1 Program Flow Control ................................................................................ 15
Figure 3.3.1 Microchip PIC Processor Pin Diagram .............................................................. 18
Figure 3.3.2a PIC Processor Circuit Board Schematic .......................................................... 19
Figure 3.3.2b PIC Processor Circuit Board Schematic .......................................................... 20
Figure 3.3.3 Encoder specifications [27] ............................................................................. 23
Figure 3.4.1a Infrared Control Circuit Schematic ............................................................... 24
Figure 3.4.1b Infrared Control Circuit Schematic ............................................................... 25
Figure 3.5.1 Signal Characteristics for the HAL506UA-E Hall Effect Sensor .................... 27
Figure 3.6.1 Camera processing block diagram ................................................................. 30
Figure 3.7.1 Block Diagram for the Ultrasound Module ....................................................... 31
Figure 4.1.1 dsk3d Program development environment ...................................................... 37
Figure 4.1.2 Source data stream structure for an external boot table [25] ......................... 39
Figure 5.1.1 (a)Point-to-point stabilization (b)Path following [26] ...................................... 53
Figure 5.1.2 Generalized coordinates of a car-like robot [26] ............................................. 54
Figure 5.1.3 Coordinate definition for a path following task [26] ........................................ 55
Figure 5.3.1 Leader velocity input and transformed velocity with headway of 0.5m .......... 61
Figure 5.3.2 Follower velocity input and transformed velocity with headway of 0.5m ...... 62
Figure 5.3.3 Error in headway distance with headway of 0.5m ........................................ 62
Figure 5.3.4 Headway distance between two vehicles with headway of 0.5m .................... 63
Figure 5.3.5 Leader velocity input and transformed velocity with headway of 0.9m .......... 65
Figure 5.3.6 Follower velocity input and transformed velocity with headway of 0.9m ...... 65
Figure 5.3.7 Error in headway distance with headway of 0.9m ........................................ 66
Figure 5.3.8 Headway distance between two vehicles with headway of 0.9m ................... 66
Figure 5.3.9 Leader velocity input and transformed velocity ............................................. 67
Figure 5.3.10 Follower velocity input and transformed velocity ........................................ 67
Figure 5.3.11 Error in headway distance with headway of 0.9m ......................................... 68
Figure 5.3.12 Headway distance between two vehicles with headway of 0.9m ................. 68
Figure 5.3.13 Leader velocity input and transformed velocity ............................................ 70
Figure 5.3.14 Follower velocity input and transformed velocity ........................................ 70
Figure 5.3.15 Error in headway distance .......................................................................... 71
Figure 5.3.16 Headway distance between two vehicles ...................................................... 71
Figure 5.3.17 Leader velocity input and transformed velocity ............................................ 73
Figure 5.3.18 Follower velocity input and transformed velocity ........................................ 73
Figure 5.3.19 Error in headway distance .......................................................................... 74
Figure 5.3.20 Headway distance between two vehicles ...................................................... 74
Figure 5.3.21 Leader velocity input and transformed velocity ............................................ 76
Figure 5.3.22 Follower velocity input and transformed velocity ........................................ 76
Figure 5.3.23 Error in headway distance ................................................................. 77
Figure 5.3.24 Headway distance between two vehicles ........................................ 77
Figure 5.3.25 Leader velocity input and transformed velocity ............................ 78
Figure 5.3.26 Follower velocity input and transformed velocity ....................... 79
Figure 5.3.27 Error in headway distance ............................................................... 79
Figure 5.3.28 Headway distance between two vehicles ....................................... 80
Figure 5.3.29 Leader velocity input and transformed velocity ............................ 81
Figure 5.3.30 Follower velocity input and transformed velocity ....................... 81
Figure 5.3.31 Error in headway distance ............................................................... 82
Figure 5.3.32 Headway distance between two vehicles ....................................... 82
Figure E1: Top view of the FLASH Project Model vehicle ................................. 120
Figure E2: Top view of the FLASH Project Model vehicle with PIC/DSP assembly removed ................................................................. 120
Figure E3: Front view of the FLASH Project model vehicle .............................. 121
Figure E4: Bottom view of the FLASH Project model vehicle .......................... 121
Figure E5: Ultrasound Owl Sensor kit ................................................................. 122
Figure E6: Left: Infrared(bottom)/HED(top) sensor arrays Right: Infrared/HED control interface PCB ................................................................. 122
Figure E7: PIC/DSP Assembly ......................................................................... 123
Figure E8: Left: DSP development module Right: PIC processor controller circuit board ................................................................. 123
Figure E9: RC car components and encoder location ................................ ...... 124
Figure E10: Optical encoder unit and encoder wheel ....................................... 124

LIST OF TABLES
Table 3.1.1 Model vehicle component list ....................................................... 10
Table 3.2.1 Texas Instruments Model TMS230C31 DSP ................................ 14
Table 3.6.1 Camera Specifications ................................................................. 29
Table 4.1.1 Command line options used to compile the FLASH project code .... 33
Table 4.1.2 Linker options used to link the FLASH project control code ......... 35
Table 4.1.3 Hex30 conversion utility command file options ....................... 38
Table 4.3.1 PIC microcontroller configuration bits ....................................... 44
ACKNOWLEDGEMENTS

The author wishes to acknowledge his advisor, Dr. Pushkin Kachroo, for his advice, his encouragement, and his enthusiasm. Without the combination of these things, the quest for knowledge would be empty.

The author wishes to thank Patricia Mellodge for her support and inspiration as they worked together on the FLASH Project. Without colleagues such as her, the quest for knowledge would be lonely.

To my friend and comrade in arms, Eric Moret, who ate all of his caps.

This work is dedicated to Kristin, who is my reason for trying to be the best that I can be, and who lights my way when it becomes dark. I love you.
Chapter 1.0 Introduction

When the internal combustion engine, and later the automobile, were first introduced to the public, no one could have foreseen the extent to which they would influence daily life. Today, with the information age in full swing, it is still hard to believe the way that computers and other information technology have permeated people’s lives. Now it seems only natural to expect information technologies to enhance the way we view automobiles.

People now take for granted automotive systems like emission control and fuel injection. In fact, many people do not realize how many systems inside their automobile are already monitored and controlled by computers [7]. Fuel delivery, ignition, emissions, air-conditioning, and automatic transmission systems are examples of the systems used daily by a car that are computer controlled or assisted.

Now, in the information age, people have come to rely on other driver assistance technologies, such as mobile phones and in-vehicle navigation systems. The goal of these technologies is to make the experience of driving less burdensome, especially on long trips. In-vehicle navigation systems have become more complex as they become more accepted, and instead of simply providing up to date maps and directions, some systems can now let drivers know about points of interest, restaurants, and fuel stations, for example.

With the advent of new automotive and computer technologies, new models of automobile behavior, and robust control algorithms and technologies, we have begun to
increase driver safety and comfort by adding systems like air bags, anti-lock braking systems, and cruise control that are all controlled electronically [3][5][6].

Research groups like the Program on Advanced Technology for the Highway (PATH) at the University of California, Berkeley and the European PROMETHEUS project have been studying systems for automobiles that increase a vehicle’s overall traction, such as anti-skid braking, alternatives to the internal combustion engine, and methods for making the highway itself part of the driving experience [1][2]. Programs like PATH have been researching methods for centralized control of the highway system, integrating roadside technologies that would allow the United States’ already overburdened highway system to be used more efficiently [11][4][12][13].

Government organizations like the United States Department of Transportation and the Intelligent Vehicle Initiative under the Federal Highway Administration, hope to realize such systems in the near future as the problems concerning overcrowding on highway systems reach critical mass. Proposed systems often consist of roadside sensors that obtain information about current traffic conditions, and relay them to receivers in the automobiles on the road. In this manner, the traffic systems and the automobiles work together to bring passengers safely and quickly to their destinations. The ultimate goal of such technology is to enable cars to perform what has been termed “platooning” [1]. In platooning, automobiles can be grouped together at highway speeds, 65-70 MPH, no more than a few feet apart which makes better use of the available roadways [1][2][4][13].

For years people have been awaiting such technologies with a mixture of anticipation and fear. As we continue our research into ITS, and into Automated
Highway Systems (AHS) in general, the automotive and academic communities hope to provide safer, more reliable, and less expensive methods of transportation for the general public. However, as this area of research is still in its infancy, people must accept that current systems are still intended as conveniences provided to drivers to help make their driving experiences more relaxing. In fact, many reported accidents are the result of driver fatigue [4][11]. This is exactly where new technologies that allow the implementation of ideas like cruise control, adaptive cruise control, and platooning come into play.

Most people today are familiar with cruise control. One simply presses a button to set the speed they wish the automobile to maintain, and then the driver only need be concerned with steering the vehicle [17]. Figure 1.1 illustrates a basic cruise control system as it appears in most vehicles today.

![Figure 1.1 Block diagram of a cruise control system [17]]
When the driver approaches a vehicle in front of their own, he must decide whether to go around the obstacle, set the cruise control to match the speed of the blocking vehicle, or disengage the cruise control entirely.

In an effort to make cruise control more advanced, engineers and scientists posed the idea that forward looking systems, like ultrasound, radar, and laser range finding technologies could be used to determine distance from vehicles and obstacles in front of the automobile, and then maintain a set distance from these obstacles.

Automatic headway control, also known as adaptive cruise control, is an area that has been under research for some time [3][4][5][6][10][12][13]. Adaptive cruise control reduces driver strain by reducing the need to constantly monitor the vehicle’s speed. It can also be coupled with other driver convenience systems to provide control for lane changing, obstacle detection, and collision avoidance to further reduce driver fatigue when traveling long distances.

Automatic headway control is one of the topics that readers would be likely to find in literature concerning ITS today [18][19][20][22][23]. Cruise control, anti-lock braking systems, collision warning, air bags, and other driver safety systems all fall into this area of research [1][2]. ITS research is also concerned with the development of systems that monitor and help to guide vehicles as well. Traffic signals, traffic cameras, road sensors, central control towers, inter-vehicle communication systems, and even the composition of road surface and lane marking materials are also in the domain of ITS [1][2][11]. Major ITS goals are to provide safer and more convenient transportation to the general public, and to utilize transportation resources more effectively in order to maximize transportation infrastructure dollars.
Chapter 2.0 Motivation

This thesis is concerned with research work on small-scale model vehicles. It has been shown in other similar studies that such work is often directly applicable to full scale transportation systems. The primary focus of this study is to provide automatic headway control for a 1/10 scale autonomous vehicle by using closed loop control systems and ultrasonic sensors to increase the vehicle’s ability to automatically maneuver around a scaled mock highway system.

Adaptive cruise control allows a user to input a desired vehicle velocity, the same way one would set cruise control in a modern vehicle. Adaptive cruise control also allows the driver to set a desired following distance for objects that may be in front of the vehicle. As long as the adaptive cruise control system does not detect any obstacles in the vehicle’s path, the maximum velocity as entered by the driver will be maintained. When an object is in the vehicle’s path, the adaptive cruise control system will adjust the vehicle’s velocity accordingly to maintain the desired, safe, distance from the obstacle.

It is easy to see how adaptive cruise control can be further enhanced to provide lane changing mechanisms for automatic passing of slower traffic, and to provide collision avoidance mechanisms that allow vehicles to not only maintain a specific distance, but to go around, or even halt in the presence of an accident or other non-moving obstacle in the road.

Development of control systems for scaled, as well as full sized, vehicles should be concerned at all times with the safety of the vehicle’s occupants, and with the stability of the system as a whole. Automatic headway control allows a driver another degree of freedom in controlling a vehicle as mentioned earlier.
Other goals of this research are to provide functional systems for incorporation into a series of educational museum displays for the Science Museum of Virginia and the Roanoke Transportation Museum as part of the Flexible Low-cost Automated Scaled Highway (FLASH) Project. The goals for this project are to provide an educational experience for museum attendees that enlighten the general public concerning the development and operation of ITS products [14][15][16][24].

Figure 2.1 Sample museum exhibit for the FLASH project [24]

The project is also concerned with the development of future technologies, and with helping the public adapt to the rapid changes that are being made in the automotive industry. Figure 2.1 shows a sample exhibit for the museum that is designed to highlight the individual technologies involved in creating an autonomous vehicle. People are naturally reluctant to allow technology become so prevalent in automobiles as they feel a loss of control in their daily activities. As such, a major responsibility of the project is to provide interaction with these new and emerging technologies and to provide as much information as possible so that the public can make their own informed decisions as to
whether this use of technology is appropriate. In dealing with other aspects of ITS, such as road surface materials and current traffic management technologies, the project is concerned with providing a look into what the public might expect to become common place in the future. Overall, the FLASH project museum exhibits are committed to showing museum patrons the development of automobile and driver safety, and how new automotive technologies can further enhance the driving experience. Figure 2.2 is a sample document from the FLASH literature that is intended to highlight the different components of an automobile that are designed to increase the safety of the passengers.

![Figure 2.2 Safety poster illustrations from the FLASH project [24]](image)

Additional goals include the creation of a scaled platform that can be used to design and test potential new technologies in the area of ITS at lower costs than that of their full scale counter parts. The sections of this thesis are developed as follows:

I. Introduction

The significance of this research is explained.

II. Motivation
Reasons for the need for this research are detailed

III. Hardware Environment

The hardware platform for the experimental research is described.

IV. Software Environment

The programming and control logic responsible for controlling the scale model is described.

V. Mathematical Modeling and Simulation Results

The development of the control algorithm and the mathematical modeling of the algorithm are detailed. The control algorithm is based on the kinematic model posed in [26]. Development of the first order control law, and a possible sliding mode controller are presented, and Matlab simulations results illustrate the effectiveness of each controller.

VI. Results

Results of research and experimentation are discussed. Assessments of the control algorithms usage in this experimental test bed are made.

VII. Conclusions and Future Work

Possible extensions and other applications of this research are detailed.
Chapter 3.0 Hardware Environment

The test bed for this thesis is a scale vehicle that is being prototyped for the Flexible Low Cost Automatic Scaled Highway (FLASH) project. As stated in the previous chapter, one of the goals of this project is to provide an inexpensive modeling platform for use in research of emerging ITS technologies. As technology has progressed in recent years, the amount of processing power that can be obtained for a relatively small investment of capital has grown incredibly. In this chapter we discuss the hardware used to create the test platform.

The hardware environment for this experiment is shown in the block diagram in Figure 3.0.1. Major components of this architecture include the model vehicle platform, the Digital Signal Processor (DSP), the PIC processor, and the sensor arrays including ultrasound, infrared, Hall Effect, and vision control systems.

![High-level architecture block diagram of the vehicle hardware platform](image)

Figure 3.0.1 High-level architecture block diagram of the vehicle hardware platform
3.1 Model Vehicle Platform

The model vehicle used is manufactured for remote control vehicle kits produced by Bolink. The kit contains a 1/10th scale carbon fiber car frame with associated mounting hardware to assemble the frame, and a Futaba steering servo rated for 90 degrees of lateral travel. The lightweight frame allows easy modification for adding ITS technologies to the design of the car. The motor, not included in the kit, is a Trinity Paradox rebuildable stock motor. This motor is rated for scaled speeds of 300MPH, which is clearly too fast for our project. The goal of the project is to show how a vehicle would react in urban or suburban settings at speeds around 20-30MPH. Therefore, the DC motor was rewound with 100 turns of 30 AWG wire in order to limit the top speed of the vehicle, and to increase the amount of resolution available for commanding the motor. A Novak Super Rooster Reversible electronic speed control is used to help limit the current drawn by the drive motor from the battery, to smooth the effect of sudden changes in velocity on the power system, and to reduce the dead time between control signals as they are sent from the PIC to the motor. Table 3.1.1 shows the specifications for the standard model car components used.

<table>
<thead>
<tr>
<th>Component</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legends 1/10 scale model car kit</td>
<td>Bolink</td>
</tr>
<tr>
<td>Steering servo</td>
<td>Futaba</td>
</tr>
<tr>
<td>Paradox rebuildable Electric Motor</td>
<td>Trinity</td>
</tr>
<tr>
<td>7.2V NiMh Rechargeable Battery</td>
<td>Radio Shack</td>
</tr>
<tr>
<td>Super Rooster Reversible Electronic speed control</td>
<td>Novak</td>
</tr>
</tbody>
</table>
Plans for the power delivery system included the use of nickel metal hydride (NiMh) laptop batteries. These batteries have microcontrollers embedded in them, and are capable of telling the control system that they are running low and it is time to recharge. However, complications in creating an interface for these batteries to the DSP arose in the early stages of development. Combined with the fact that laptop batteries also have the potential to be very dangerous if operated outside of the specifications, and the fact that three hours of run time could be achieved with standard NiMh remote control car batteries, development of this technology was temporarily abandoned.

These off the shelf components are easily obtainable, and provide consistent, reliable performance that is in-line with the FLASH project design goals. These are, of course, the reasons for choosing remote control cars as the platform for this vehicle. The flexibility, and relatively inexpensive cost of these components, affords the project the most latitude in making design decisions, and in testing multiple sensor and control technology platforms.

The steering servo is a standard component and is controlled with a pulse width modulated (PWM) signal. The period of the signal should be in the range of 10ms to 30ms, while the high period of the signal ranges nominally from 1.0ms to 2.0ms. In the FLASH project implementation, the servo is being controlled with a 14ms period and 1.0ms of high time corresponds to full left steering, 1.5ms corresponds to center, and 2.0ms to full right steering. Figure 3.1.1 illustrates the format of the PWM signal expected by the steering servo.
Commands from the DSP to the PIC have a resolution of approximately 3 degrees of steering per unit in the DSP. This gives the vehicle smooth, robust control over the actuation of the steering.

In addition to the responsibilities mentioned earlier, the ESC is also used to provide PWM control for the motor, similar to the signal used to control the steering servo. Instructions provided with the speed control detail the simple one touch process used to calibrate the speed control. However, as the project does not use a typical transmitter that would normally be used to control a regular RC car, the motor is calibrated using a PIC processor with special code that only applies three pulse width signals as described below and a wireless link operating at 400MHz that communicates with the PIC processor serially. Using this link, the ESC is programmed to set full reverse at a 1ms high time, neutral at 1.2ms high time, and full forward at 2.0ms. This greatly increases resolution in the forward direction, giving the best control over the speed of the vehicle, while still yielding enough range in reverse to slow the vehicle down when descending a grade. As there are no brakes on the vehicle, decelerating requires that the vehicle be placed in reverse to counteract the effects of gravity.

The vehicle has speed control that allows it to maintain a constant speed when climbing or descending a grade. This is, in effect, cruise control. Using a HEDS-90041
optical encoder from U.S. Digital as the rear axle position sensor, the car is able to maintain constant speed using the following algorithm. The encoder disc with 512 counts per revolution is mounted on the rear axle of the vehicle. This allows the actual speed of the vehicle to be measured and compared with the desired speed that the car is commanded to maintain. The digital signal processor, discussed in the next section, commands the PIC processor by sending a desired speed in terms of encoder ticks per sample period. The PIC processor then counts the number of encoder ticks and compares this value to the value received from the DSP. If the actual number of encoder ticks is less than the desired number, then the vehicle is moving too slowly, and the velocity must be increased. If the actual number of encoder ticks is greater than the desired number, then the vehicle is moving too quickly, and the velocity command must be decreased. This control will be discussed further in the software platform chapter under the PIC processor programming section.

A simple conversion allows the calculation of velocity from the number of ticks sent to the PIC as the velocity command. The number of encoder ticks per 8ms is simply divided by 10.266 to achieve the actual velocity of the vehicle in meters per second. Full-scale speed can then be calculated by multiplying this velocity by 10, as the vehicle is 1/10 scale.

3.2 Digital Signal Processor Platform

The choice of a digital signal processor as the high level processor for the automated vehicle was made primarily for the ability to perform image processing. However, due to the inexpensive nature of today’s DSP platforms, the FLASH project was able to incorporate a DSP that could meet the project’s needs for image processing.
and that still had enough power to process the rest of the sensor data for the control of the vehicle. Table 3.2.1 lists the statistics for the Texas Instruments TMS320C31 DSP used by the FLASH project.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>TMS320C31-50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>50</td>
</tr>
<tr>
<td>MIPS</td>
<td>25</td>
</tr>
<tr>
<td>MOPS</td>
<td>275</td>
</tr>
<tr>
<td>Cycle Time (ns)</td>
<td>40</td>
</tr>
<tr>
<td>Data/Program Memory (Words)</td>
<td>16M</td>
</tr>
<tr>
<td>RAM (Words)</td>
<td>2K</td>
</tr>
<tr>
<td>Cache</td>
<td>64</td>
</tr>
<tr>
<td>DMA</td>
<td>1</td>
</tr>
<tr>
<td>Timers</td>
<td>2</td>
</tr>
<tr>
<td>Total Serial Ports</td>
<td>1</td>
</tr>
<tr>
<td>Serial Ports</td>
<td>1</td>
</tr>
<tr>
<td>Parallel Ports</td>
<td>16Mx32</td>
</tr>
</tbody>
</table>

The DSP is seated in a development kit that grants easy access to control signals, and external interface hardware that is critical to the development of the vehicle’s control structure. Functioning as the driver of the vehicle, the DSP performs most of the data processing required to interpret the data received from the individual sensor arrays. The exception to this rule is the optical encoder used to determine the vehicle’s actual speed, which is interfaced to the PIC processor.

Each sensor is treated as a slave device on the memory bus, and during a sensor processing cycle, each device is read, and the data from that sensor is processed to determine what adjustments are to be made to the steering and velocity controls. This is accomplished by making ample use of the 24 bit addressable 32 bit data bus. The infrared and Hall Effect sensors are thus located in memory-mapped space in the DSP while the ultrasound data is read separately via the analog-to-digital conversion unit.
provided on the DSP development board. The PIC processor is also treated as a slave device by the DSP, and once the sensor data has been processed and the appropriate drive commands have been determined, the commands are sent to the PIC processor, which is the direct interface to the system’s velocity and steering control. The program flow is shown in Figure 3.2.1.

Another feature of the DSP that the vehicle makes heavy use of is the boot loader capability. Upon resetting the vehicle, the processor is triggered by an external interrupt that indicates that the vehicle’s control code is loaded from one of three addresses in memory. In the case of the FLASH vehicle, external interrupt 1, INT1, is de-asserted to indicate that the processor should begin loading code from address 0x400000. At this
address resides an EEPROM that contains the operating program for the vehicle. The EEPROM is an ATMEML 24HC64B 8-bit wide memory device, and so the 32-bit DSP must read in the program code four bytes at a time, and reassemble by placing them correctly into one word in the internal RAM of the processor. Once the code has been reassembled in memory, the boot loader program transfers control of the processor to a pre-determined address, at which point the control code is executed.

Difficulties in converting the program to boot table format arose early in the development of the vehicle. While the control program could easily be compiled and loaded onto the DSP via a parallel interface with a host computer, the same code could not successfully be translated into a boot table and loaded from the EEPROM. As the code could be loaded and run from a host computer, the development of the EEPROM was tabled until such time as it was necessary to further the development of the vehicle.

The infrared and Hall Effect sensors are each addressed via the addressing structure shown in figure 3.3.2. Each of these two sensors use tri-state buffers, which keeps them from transmitting data to the bus when they are not active. As each device is addressed, they supply their data to the 32-bit data bus. Two infrared sensors, each with 12 bits, are read in during one cycle with the front sensor supplying the lower 12 bits on the data bus, and the rear sensor supplying the upper 12 bits on the data bus. The Hall Effect sensors each supply 8 bits on the data bus, with the front sensor array supplying the lower nibble, and the rear sensor supplying the upper nibble.

The ultrasound data is read differently, as it is captured using A/D conversion. The A/D conversion unit is software controlled, and so whenever a reading is required, the DSP simply triggers one conversion event, and the A/D interrupt is signaled when the
conversion is complete. The ultrasound sensor analog output is connected to the A/D capture port on the DSP development board. When the interrupt is triggered, the DSP services the interrupt and thus generates the relative distance between two vehicles on the scaled highway.

Finally, the camera is interfaced with external memory that is mapped into the DSP memory. The camera downloads data to external memory when frames are captured for processing, and the DSP reads each required row of a single frame from this external memory, performing the required operations upon the image data. Thus the camera data is fed directly into memory without having to traverse the data bus. This allows more efficient use of the data bus, as the vehicle will take approximately thirty sensor cycles to transfer one frame’s worth of data into memory. Direct Memory Access (DMA) facilities incorporated with the DSP allow this scheme to operate rapidly and to maintain control over the vehicle.

The camera requires approximately 300KB to store one frame of image data in memory. As the DSP currently in use has only 64KB of memory for the program and the data, use of the camera is not feasible at the time of this writing. The choices present for remedying this situation are to interface external RAM to the system via the external data and address bus, or to research the possibility of using a different DSP platform with more internal RAM. Each choice has its technical merits, but a decision has yet to be made as to how development will proceed in this case.

Once the control code has begun execution, the DSP reads in data from each of its sensors, processes the data to determine a steering angle and velocity, and sends these commands to the PIC processor, which will be discussed next.
3.3 PIC Processor Platform

A PIC16F874-20P is responsible for the direct control of the scale vehicle, translating the high level commands coming from the DSP into electrical signals that can be applied directly to the servos controlling the model car. Figure 3.3.1 shows a pin diagram of the PIC processor. This PIC is a low power RISC processor that operates at 20 MHz. There are only 35 instructions in PIC16F874 repertoire, making it a simple, but powerful microcontroller. The PIC processor is marketed as a versatile development platform, allowing for rapid design and development of custom microcontroller solutions. The circuit schematic for the PIC processor board is shown in Figure 3.3.2.
Figure 3.3.2a PIC Processor Circuit Board Schematic
Figure 3.3.2b PIC Processor Circuit Board Schematic

<table>
<thead>
<tr>
<th>Component Number</th>
<th>Quantity</th>
<th>Reference</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>C1,C2</td>
<td>10μF</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>C4,C3</td>
<td>0.01μF</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>D1,D2</td>
<td>LED</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>E1</td>
<td>ANTENNA</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>JP1,JP2</td>
<td>HEADER 3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>R1,R2</td>
<td>150</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>R3</td>
<td>10K</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>U1</td>
<td>PIC16F874</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>U2</td>
<td>RXM-433-RM</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>U3</td>
<td>LM340</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>U4</td>
<td>LM7805</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>Y1</td>
<td>CRYSTAL, 20MHz</td>
</tr>
</tbody>
</table>
The PIC receives its driving commands from the DSP via a built-in Parallel Slave Port (PSP). The DSP treats the PIC as a slave device on the interconnection bus and simply sends commands to the PIC that are interpreted as mode changes, steering, or velocity commands based on a proprietary protocol. The command format is a simple one-byte format.

<table>
<thead>
<tr>
<th>E/D</th>
<th>S/V</th>
<th>Data</th>
<th>Data</th>
<th>Data</th>
<th>Data</th>
<th>Data</th>
</tr>
</thead>
</table>

Bit 7 E/D: 1 – The vehicle is enabled. 0 – The vehicle is disabled.
Bit 6 S/V: 1 – Velocity command. 0 – Steering command.
Bits 5:0 Data: These bits are the value of the steering or velocity command. For a steering command, values from 0-63 sent to the PIC and offset by 85 to determine the steering angle pulse width. The value 32 sent from the DSP is center. For a velocity command, values from 0-63 are sent to the PIC where 0 and 1 are slow reverse, 2 is stopped, and 3 to 63 are increasing velocities in the forward direction.

Upon receiving a command, the PIC decodes the operation and makes corresponding changes in either the steering or velocity pulse width generators, or enables or disables operation of the vehicle. The PIC uses two internal timers, TMR0 and TMR2, to maintain the Pulse Width Modulation (PWM) signals that control the vehicle. The servo control signals are then simply output to two pins of PORTA. TMR0 is used for the steering pulse and is configured with a prescaler value of 64. This allows the pulse width to have a resolution of 12.8us and meet the range requirements for full throw of the servo. TMR2 is used for the velocity pulse width. This timer is configured with a prescaler value of 16 and a postscaler value of 2, effectively scaling the timer by 32. This gives a resolution of 6.4us. Greater resolution is required by the velocity pulse
width in order to smoothly control the velocity using the encoder. Coarser resolution
results in jerky movement and eventual loss of control as the lateral control algorithm
depends on a smoothly changing velocity to maintain stability.

Originally the PIC code used a single timer to manipulate both of the pulse widths
by determining which pulse width would go low first, and loading the appropriate value
in the timer match register. Then, when the timer expires, the corresponding output pin
would be driven low and the difference in the high times of the two pulses would be
placed in the match register. This amounts to putting the remaining high time for the
longer of the two pulses into the match register. When the timer expires for the second
time, the corresponding output pin is then driven low. However, under operating
conditions, it was noted that the vehicle would speed up when making left turns, and slow
down when making right turns. This phenomenon was then observed under bench top
test conditions and after some research into the causes of this behavior, it was determined
that when the steering and velocity high times were required to be too close together or
too far apart that the delay in processing these times in addition to the extra time required
to determine the interrupt that occurred caused the timer’s match register to be updated
too early or too late. This produced the behavior noted, and is the reason that two timers
are required for operation of the vehicle.

The third timer on the PIC is used as a counter to incorporate the encoder data
collected as the car is running. The encoder attached to the drive wheel that produces
256 pulses, after being filtered by a D flip-flop, per revolution of the drive wheel is
shown in Figure 3.3.3.
TMR1 is configured to act as a counter, and the encoder pulse train is used as an input to RC0, the external timer input. Thus the timer increments every time there is a rising edge on the encoder pulse train. The PIC implements a one second sample period that is read 8 times during the 10ms required to operate the velocity servo. This allows us some flexibility in the resolution of the speed controller as we are averaging the encoder data over 8 sample periods. While the timer is being read, the timer is disabled. The timer value is read and added to the total over encoder count over 8 sample periods, which is then compared with the desired number of encoder ticks as input from the DSP. If the actual number of encoder ticks is less than the desired value, then the vehicle is moving too slowly, and the speed is increased, or the vehicle speed is decreased if the actual encoder tick count is greater than the desired number of ticks. This is the basis for the closed loop velocity control that allows the vehicle to maintain a nearly constant velocity under changing grades and road conditions.
In order for the DSP to issue other commands to the PIC processor, the DSP must obtain data from the sensor arrays. These sensor devices are discussed in the following sections.

3.4 Infrared Sensor Arrays

The simplest of the sensor technologies is the infrared sensor array. Using these types of sensors, the FLASH vehicle can follow a white line on a black road surface by detecting reflected infrared light.

Two arrays of infrared sensors are mounted on the vehicle as the front and rear bumpers. Figure 3.4.1 is a circuit schematic of the control circuit board and diagrams of the infrared sensor arrays.

![Figure 3.4.1a Infrared Control Circuit Schematic](image-url)
Figure 3.4.1b Infrared Control Circuit Schematic

<table>
<thead>
<tr>
<th>Component Number</th>
<th>Quantity</th>
<th>Reference</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>C1</td>
<td>10uF</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>C2, C3, C4, C5</td>
<td>0.01uF</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>D1</td>
<td>DIODE ZENER1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>JP1</td>
<td>HEADER 2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>JP2</td>
<td>HEADER 9x2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>JP3</td>
<td>HEADER 5x2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>JP5, JP4</td>
<td>HEADER 12</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Q1, Q2, Q3, Q4</td>
<td>2N2222A</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>Q05, Q06, Q07, Q08, Q09, Q10, Q11, Q12</td>
<td>HED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>R01</td>
<td>33</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>R02</td>
<td>10K</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>R03, R04, R05, R06, R09, R10, R11, R12, R13, R14, R15, R16</td>
<td>1K</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>R07, R08</td>
<td>220SIP, SIP-10</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>U1</td>
<td>LM7805</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>U2</td>
<td>LM358N</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>U3, U4, U5</td>
<td>74HCT244N</td>
</tr>
</tbody>
</table>
These arrays serve to provide lateral control for the vehicle by sensing the centerline of the road. The control circuit is simple. It contains an operational amplifier, a potentiometer, and four transistors that are used to drive four banks of three infrared emitters that operate at a peak wavelength of 900nm. The 10k? potentiometer serves to vary the strength of the emitters to tune them to the specific characteristics of the operating environment and road surface.

Two octal buffer/line drivers, 74HC244N, serve to buffer the signals from the detectors, and operate in high impedance mode when the DSP is not requesting data from the sensors. This serves to keep the data bus free of unnecessary traffic. The outputs from the detectors are pulled to logic high by a set of 10k? SIP resistors. When active, the detectors provide a path to ground, and the bit for the corresponding detector is driven low. The detectors are active at a peak of 850nm. The emitter detector pairs were chosen for their precision in emission characteristics, which allow them to be packed tightly in a sensor bank, allowing for high resolution in the detection of the centerline.

Two 12-bit numbers are returned from the control circuit to the DSP, one bit per detector, indicating the position of the centerline to the vehicle. The control circuit allows the DSP to request data when it needs it by driving the buffer enables low, activating the buffer chips. Based on the data received from front and rear sensor arrays, the DSP uses a simple PID control algorithm to determine how much steering must be applied to keep the centerline of the road under the center of the vehicle.

3.5 Hall Effect Sensor Arrays

Hall effect devices work by detecting magnetic fields. In full sized vehicles, such devices are used in conjunction with magnets embedded in the road at regular intervals.
The vehicle can then follow the centerline of the road in much the same manner as it would by using infrared sensors, but using a different sensing medium.

Mounted as a separate module on the infrared circuit board are banks of Hall Effect devices. These devices are connected to the data bus by octal buffers 74HC244N as well. The individual magnetic sensors used are Micronas HAL506UA-E Hall Effect devices. These devices are 3 pin digital output unipolar sensors that respond to south polar magnetic fields. They are sensitive to fields stronger than 7mT in strength. Figure 3.5.1 shows the characteristics for the sensor. It is important to note that the input and output pins are shorted together by a 1kΩ resistor to provide the correct signal biasing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>On point B&lt;sub&gt;ON&lt;/sub&gt;</th>
<th>Off point B&lt;sub&gt;OFF&lt;/sub&gt;</th>
<th>Hysteresis B&lt;sub&gt;HYS&lt;/sub&gt;</th>
<th>Magnetic Offset</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>-40 °C</td>
<td>4.3 5.9 7.7</td>
<td>2.1 3.8 5.4</td>
<td>1.6 2.1 2.8</td>
<td>4.8</td>
<td>mT</td>
</tr>
<tr>
<td>25 °C</td>
<td>5.8 5.5 7.2</td>
<td>2 3.5 5</td>
<td>1.5 2 2.7</td>
<td>3.8 4.5 6.2</td>
<td>mT</td>
</tr>
<tr>
<td>100 °C</td>
<td>3.6 5.1 7</td>
<td>1.9 3.3 4.9</td>
<td>1.2 1.8 2.6</td>
<td>4.2</td>
<td>mT</td>
</tr>
<tr>
<td>140 °C</td>
<td>3.4 4.8 6.9</td>
<td>1.8 3.1 5.1</td>
<td>1 1.7 2.6</td>
<td>4</td>
<td>mT</td>
</tr>
<tr>
<td>170 °C</td>
<td>3.2 4.6 6.8</td>
<td>1.7 3 5.2</td>
<td>0.9 1.6 2.6</td>
<td>3.8</td>
<td>mT</td>
</tr>
</tbody>
</table>

Figure 3.5.1 Signal Characteristics for the HAL506UA-E Hall Effect Sensor

These sensors turn on in the presence of a magnetic field, and turn off in the absence of any field. The sensors were simple to integrate into the FLASH framework as their small package allows them to be placed close together for increased resolution in determining the distance the FLASH vehicle is off of the centerline, but unlike the infrared emitter/detector pairs, the Hall Effect devices can not be placed directly adjacent to one another because their area of sensitivity is not as tightly focused as that of the infrared devices.
The magnets used in the mock highway system are simple 2in by ½in by ¼in household bipolar magnets obtained from Radio Shack. Placed at regular intervals of approximately one half an inch in the track bed under the centerline used by the infrared sensors, the vehicle’s magnetic sensors can follow the magnetic centerline. Placed accordingly under the roadway, the magnetic field appears continuous to the vehicle and thus the same lateral controller applied to the data from the infrared sensors can be applied to the data from the Hall Effect sensors.

In terms of practical applications, it is more likely that full sized automobiles will use magnetic sensors in this type of highway system because of cost and reliability factors involved in the infrared systems. In terms of the FLASH project, both sensor types are used to show contrast of device type and sensing media in the museum setting.

### 3.6 Visual System

The primary reason for choosing a DSP platform was for the processing power required to rapidly process visual data. The visual system is perhaps the most intuitively easy system to understand because it so closely resembles the way that humans drive automobiles today.

A digital color camera in conjunction with the DSP comprises the vision system utilized in this project. The DSP interfaces with the camera via a DMA channel and the interconnection bus to capture video frames of the road ahead. The camera is an Omnivision model OV7630 color CMOS camera. Table 3.6.1 shows the statistics for the camera.
Table 3.6.1 Camera Specifications

<table>
<thead>
<tr>
<th>Key Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Element (VGA) (QVGA)</td>
<td>640x480</td>
</tr>
<tr>
<td></td>
<td>(320x240)</td>
</tr>
<tr>
<td>Pixel Size</td>
<td>5.6μm x 5.6μm</td>
</tr>
<tr>
<td>Image Area</td>
<td>2.7mm x 2.18mm</td>
</tr>
<tr>
<td>Max Frames/Sec</td>
<td>Up to 60 FPS for QVGA</td>
</tr>
<tr>
<td>Electronics Exposure</td>
<td>Up to 648.1 (for selected FPS)</td>
</tr>
<tr>
<td>Scan Mode</td>
<td>Progressive or Interlace</td>
</tr>
<tr>
<td>Gamma Correction</td>
<td>0.45/0.55/1.0</td>
</tr>
<tr>
<td>Min. Illumination (3000K)</td>
<td>OV7630 &lt; 5 lux @ f1.2</td>
</tr>
<tr>
<td></td>
<td>OV7130 &lt; 0.8 lux @ f1.2</td>
</tr>
<tr>
<td>S/N Ratio</td>
<td>&gt; 48 dB (AGC off, Gamma=1)</td>
</tr>
<tr>
<td>FPN</td>
<td>&lt; 0.03% Vpp</td>
</tr>
<tr>
<td>Dark Current</td>
<td>&lt; 1.9nA/cm²</td>
</tr>
<tr>
<td>Dynamic Range</td>
<td>&gt; 72 dB</td>
</tr>
<tr>
<td>Power Supply</td>
<td>3.0-3.6VDC</td>
</tr>
<tr>
<td>Power Requirements</td>
<td>&lt; 25mA Active</td>
</tr>
<tr>
<td></td>
<td>&lt; 10μA Standby</td>
</tr>
<tr>
<td>Package</td>
<td>28pin LCC</td>
</tr>
</tbody>
</table>

Although it is a color camera, it can be used as a black and white camera, which is more than sufficient for use in the FLASH project. Future enhancements to the vision system may include the incorporation of color analysis to recognize special or hazardous situations, such as recognizing orange construction cones, or special hazard signs.

Figure 3.6.1 shows a block diagram of the camera operation. The camera is interfaced to the DSP via external RAM as discussed in earlier sections. Using the DMA facilities of the DSP, the camera can communicate with the processor in the background, giving the DSP the time needed to process other data while waiting for the pixels from a single frame to upload into working memory. It takes approximately 30ms to transfer one frame from the camera to the DSP, and so the DSP must be able to work with the other devices and process their data in order to keep the vehicle on the road while it waits for the image data.
Once the camera has uploaded a frame of image data to the DSP, the processor uses edge detection and prediction algorithms to determine the curvature of the upcoming road. Thus the camera data can be incorporated with the other sensor data to allow the vehicle to adjust its speed to travel more efficiently down straight stretches of road and through turns.

### 3.7 Ultrasonic Sensors

The last sensor technology to be described in this chapter is the ultrasonic sensor. Polaroid ultrasonic sensors provide the means to determine the vehicle’s relative distance from obstacles in the vehicle’s path. Ultrasound technology is an inexpensive means to model more complicated laser based radar and microwave technologies that are currently in use around the world in several types of production vehicles.

The Polaroid sensors can be used to detect objects in front of the vehicle, at a range of 150mm to 2.67m with 10mm resolution. The ultrasound sensor is in fact a kit
(Figure 3.7.1) designed for use in laboratory settings for experiments involving range finding.

**Board Layout and Connections**

![Board Layout and Connections](image)

**Figure 3.7.1 Block Diagram for the Ultrasound Module**

Operating at 40kHz, the Polaroid transducer emits an ultrasonic pulse, which is then detected by the transducer when it is reflected off of obstacles in front of the vehicle. The Polaroid sensor is connected to the circuit board shown above, which controls the timing of the pulses and calculates the perceived distance of objects from the sensor.

The sensor outputs a voltage that ranges from 0V to 5V, with 0V indicating that an object is 150mm or closer to the sensor, and 5V indicating an object is 2.67m or further from the sensor. This voltage is in turn captured using the DSP’s analog to digital conversion facility. The DSP captures the analog voltage at the A/D conversion input, and interprets the voltage level to indicate a distance of an obstacle, and according to the control program adjusts the speed of the vehicle to match that of the detected obstacle. Samples are taken from the ultrasound sensor at 200us intervals.

The ultrasound unit has an approximate resolution of 30mV per inch, which is more than sufficient for the needs of the project. In addition, the DSP has a resolution of 80 units per inch when the analog signal from the ultrasound unit is converted into a
digital number. Again, this is more than adequate, providing the vehicle with enough resolution to comfortably detect small changes in distance at operating ranges of 3 to 10 inches in front of the vehicle.

This concludes the discussion of the hardware platform used in the FLASH project. In the following section the details of the software platform including language specifications and development environments are discussed.
There are several software platforms involved in the development of control algorithms for this project. C, Texas Instruments TMS320C31 assembly, and PIC assembly are the three languages that are employed in developing code for the FLASH vehicle. Beginning with a discussion on how to write and build a program for the TMS320C31 DSP, we will then move on to how to program for the PIC16F874 microcontroller.

4.1 Programming in C for the TMS320C31

The C31 DSP is programmed largely in standard C. Packaged cross compilers, linkers, and hex conversion utilities allow rapid development of new code revisions for the target environment. After a C program is written, it is then compiled, assembled, and linked to form a program executable by the C31 DSP. An extra step, the hex conversion utility, translates the program into a format that can be placed on an EEPROM and used by the bootloader to boot the DSP upon restart.

The high level code is first written in C, a programming language well known for its power and flexibility. Texas Instruments provides a compiler, cl30, that translates the high level C code into the Texas Instruments assembly language and places code modules into object files. The command line to run the compiler is `cl30 -v31 -k -g control.c`

Several command line options specify how the program is to be assembled.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-k</td>
<td>Keep intermediate files</td>
</tr>
<tr>
<td>-v31</td>
<td>Produce assembly for the C31 platform</td>
</tr>
<tr>
<td>-g</td>
<td>Generate debugger information</td>
</tr>
</tbody>
</table>
The –k option keeps intermediate object files, rather than removing them after processing. This allows for faster building times if changes are made in only one or two modules of a large program. The –v31 option builds code specifically for the C31 platform, as the compiler is compatible with several DSP platforms. Finally, the –g option allows the debugger to display information about the current state of the program when it is being run in the debugger environment. Developers use this information to determine the correctness of the programs as they are being designed.

Once this has been completed, the linker program, l30, which is also packaged with the DSK is invoked from the command line with `lnk30 control.cmd` to link together all of the assembled modules for the program including the runtime libraries required to execute the code on the DSP host, and a special object file called boot.asm that defines several global options that are needed when the program is to loaded by the bootloader at reset. The command file is given as follows:

```bash
control.obj
putmem.obj
getmem.obj
init_aic.obj
-l boot.obj
-l rts30.lib
-cr
-o control.out
-heap 128
-stack 560
-m control.map

MEMORY
{
  RAM0 : org=0x00809800, len=1424 /* INTERNAL BLK 0 */
  RAM1 : org=0x00809D90, len=560 /* INTERNAL BLK 1 */
  VECS : org=0x809FC5, len=0x0002
}

SECTIONS
{
  .text : {} > RAM0
  .cinit : {} > RAM0
```
.data : {} > RAM0  
.bss : {} > RAM0  
.sysmem: {} > RAM0  
.const : {} > RAM0  
.stack : {} > RAM1  
SP0VECTS : {} > VECS

Table 4.1.2 Linker options used to link the FLASH project control code

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cr</td>
<td>Link the modules using the RAM model</td>
</tr>
<tr>
<td>-e _cint00</td>
<td>Tells the linker that the program entry point is the symbol _cint00</td>
</tr>
<tr>
<td>-heap16</td>
<td>Sets the size of the 16-bit heap</td>
</tr>
<tr>
<td>-l rts30.lib</td>
<td>Specifies the run time library to link with</td>
</tr>
<tr>
<td>-o control.out</td>
<td>Names the output file</td>
</tr>
<tr>
<td>-l boot.asm</td>
<td>Links the control code with the boot.asm file</td>
</tr>
<tr>
<td>-m control.map</td>
<td>Names the map file</td>
</tr>
</tbody>
</table>

Each object file listed is required by the linker to provide symbolic information during link time. Following these files, the linker requires several options to correctly link the program for the target environment. The –cr option produces executable code that is created under the RAM model, meaning that it is intended to be executed from working memory, and not from an external ROM device.

The –e option specifies the entry point for the program. This in effect tells the DSP where to start execution of the control program once it has completed its initialization. In the case of the FLASH project, the entry point is defined as the symbol _e_int00. This is also important information when creating a boot table to load the control program from external memory when the processor is reset. The entry point is associated with an address in memory, and this address can be obtained from the map file, specified with the –m option. Once the address for the entry point is known, the code can be translated into a bootable format.

The –l option specifies libraries or other external modules that must be linked with the code to produce executable modules. The rts30.lib file is needed for several
runtime function calls, including some basic arithmetic operations and memory access functions. The boot.asm file is required because, as mentioned earlier, it provides symbols required for the correct initialization of the program once it has been translated into executable form. The most important of these symbols is of course the definition of the _cint00.

Heap, or runtime memory, size is defined by the –heap16 option. Using this option, the programmer can define the size of the 16-bit heap that is to be used by the processor. This is another way in which the programmer has excellent control over all operation of the program. Lastly, the –o option specifies the name of the output file from the linker. This file is the executable module that is loadable via the debugger.

The MEMORY directive tells the linker how the address space is being divided between sections when the program is linked, thus defining the amount of space that is allocated to each section. The SECTIONS directive uses the blocks defined by the MEMORY directive to locate modules of code in RAM in the DSP. The stack is given its own memory block, and the interrupt vector table is located in the section required by the memory map defined by Texas Instruments for user-defined interrupt service routines.

The executable module can be run by the target platform. The debugger program connects with the target platform via a parallel cable that provides several control signals, as well as a path for compiled programs to the host memory. Upon initialization, the debugger asserts control signals that essentially allow the DSP to treat the host PC as an external memory unit, from where the DSP will download its program code. The debugger provides a disassembly window, a command interface, a register watch
window, and a working memory watch window. In order to load the control program via the debugger, the debugger must be invoked from the directory on the host computer where the executable module resides. Once the debugger has started, one simply types load <executable module name>, and then executes the run command. At this point, the program has been loaded onto the target system, and is running.

Using these windows, it is possible to watch program execution via the use of breakpoints in the code. Stepping through the assembly code one line at a time, it is easy to see what registers have been modified by a single statement, and how main memory is modified by the execution of the program. Figure 4.1.1 shows a screen shot of the debugger interface.

Once fully debugged, source programs can then be further processed to produce boot tables, used to boot strap the DSP when it is reset. The provided utility, hex30, uses a command file that specifies the options for translation. The options in the command
file allow the translator to produce bootable code that can be placed in an external EEPROM that is accessed at reset. The command file invoked by the hex30 executable is as follows:

```
control.out
-a
-map control.mxp
-boot
-bootorg 400000h
-memwidth 8
-romwidth 8
-e 809948h /* need to change this to entry point of program */
-cg 10E8h
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Produce ASCII hex output</td>
</tr>
<tr>
<td>-cr</td>
<td>Use the RAM model</td>
</tr>
<tr>
<td>-boot</td>
<td>Translate all sections into boot table entries</td>
</tr>
<tr>
<td>-e</td>
<td>Specify the entry point for the program</td>
</tr>
<tr>
<td>-bootorg</td>
<td>Specify the access point in memory for the boot table</td>
</tr>
<tr>
<td>-romwidth</td>
<td>Specifies the data width of the boot ROM device</td>
</tr>
</tbody>
</table>

Table 4.1.3 Hex30 conversion utility command file options

The first option, -a, specifies that the output should be in ASCII hex format. The translation utility can produce several types of hex files, from Intel format, to Motorola S-Records. The -cr option specifies the RAM model has been used to assemble the program. This tells the processor that when it starts, it is going to take the code stored at these external memory locations and assemble the program in working memory, rather than executing the instructions in place from the slower EEPROM.

The –boot option specifies that any sections that can be made bootable are to be translated into boot table format. Un-initialized sections of code are not translated into the boot table, so the conversion utility must make sure that any initializations that are
required are moved into sections that will be translated. For more information on which program sections are converted, refer to [25].

The width of the external memory is defined with the –romwidth option. The EEPROM has 8-bit wide data words, and so the romwidth is specified as 8. This option plays a large role in determining how the program is translated and how it is stored in the EEPROM. The boot table format species, among other things, how the code resides in external memory.

<table>
<thead>
<tr>
<th>Word</th>
<th>Content</th>
<th>Valid Data Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Memory width (8, 16, or 32 bits) where source program resides</td>
<td>8h, 10h, or 20h, respectively</td>
</tr>
<tr>
<td>2</td>
<td>Value to set the STRB control register</td>
<td>See subsection 10.7</td>
</tr>
<tr>
<td>3</td>
<td>Size of first data block. The blocksize is the number of 32-bit words in the data block. A 0 in this entry signifies the end of the source data stream</td>
<td>$0 \leq \text{size} \leq 2^{24}$</td>
</tr>
<tr>
<td>4</td>
<td>Destination address to load the first block</td>
<td>A valid C31 24-bit address</td>
</tr>
<tr>
<td>5</td>
<td>First word of first block</td>
<td>A C31 valid instruction or any 32-bit wide data value (LSB first)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>Last word of first block</td>
<td>A C31 valid instruction or any 32-bit wide data value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>m</td>
<td>Size of last data block. The blocksize is the number of 32-bit words in the data block. If the next word following this block is not 0, another block is loaded.</td>
<td>$0 \leq \text{size} \leq 2^{24}$</td>
</tr>
<tr>
<td>m + 1</td>
<td>Destination address to load the last block</td>
<td>A valid C31 24-bit address</td>
</tr>
<tr>
<td>m + 2</td>
<td>First word of last block</td>
<td>A C31 valid instruction or any 32-bit wide data value (LSB first)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>j</td>
<td>Last word of last source block</td>
<td></td>
</tr>
<tr>
<td>j + 1</td>
<td>Zero word. If more than one source block was read, word $j + 1$ would be the last word of the last block. Each block consists of header and data portions. The block's header is shaded darker than the block's data section.</td>
<td></td>
</tr>
</tbody>
</table>

Table 11–2. Source Data Stream Structure

As can be seen from the Figure 4.1.2, the first word in the boot table specifies the width of the data words as they are stored in the EEPROM. This is how the processor
determines how many words are to be assembled into one instruction in working memory. In the case of the FLASH project, the DSP memory width is 32-bits, and the EEPROM stores 8-bit words, and so the processor must read four bytes and assemble them into a single instruction before that instruction can be placed in memory. The second byte defines the strobe control register word. This word configures the strobe port in its use of wait states and definable control signals.

The third and fourth bytes in the EEPROM are the size of a data block and the starting address in memory that the data should be placed at after assembly. The bootloader continues to read in data, assemble instructions, and place them in memory until a zero block size is read. This is the terminator for the boot table.

Following this translation, the code is then loaded into the BK Precision 844 programming unit software. The hex file is read by the program, interpreted as an ASCII hex file, and written to an EEPROM by the programming module. The boot table is then accessed on startup by the DSP. Control circuitry shown in Figure 3.3.3 asserts the INT1 signal, which informs the processor that the boot table begins at 0x400000 in memory. The DSP then reads the control stream from the boot table as defined above, and once the boot load process completes, control of the processor transfers to the program at the address specified by the –e option in the hex command file.

At the time of this writing, the EEPROM development was not complete. Difficulties arose in attempting to produce a boot table from the C code written. Debugging the boot process proved to be difficult because of erratic behavior exhibited by the target system. Since development of the code was not dependent upon being able
to boot from EEPROM, the development of this feature was put on hold until time warranted further scrutiny.

4.2 C Program Code Description

The C program that controls the vehicle’s operation on the large scale is quite simple. What follows in this section is a description of the program flow through the control code. Refer to Appendix A for the code referenced in this section.

Upon initialization, the C program begins by executing the GetMem() function. This call is an assembly function that reads data from the data bus into an input buffer where it can be manipulated in RAM. The GetMem function reads in data from the infrared sensors. Once the data has been collected, the sensor values are processed by the main program and passed to the FindError() function.

Having read the infrared sensors, the vehicle then sets the vehicle’s speed by using the command protocol discussed in Chapter 3 and the function PutMem(). PutMem is another assembly level function that accesses a specific location in the DSP memory. The device that will be receiving the data sent by the DSP is determined by the offset parameter sent in the PutMem() function call. The data is then sent over the data bus, and the slave device responsible for reading the data is enabled to receive the data. Determined by the transforms in the code, the velocity is set and sent to the PIC as determined by the DSP’s control algorithm.

After the velocity command has been sent for the current iteration of the control loop, FindError() takes the error read by the infrared sensors and converts it to an error term in meters from the centerline of the vehicle. This allows the controller to determine the correct steering angle to minimize the error between the angle of curvature of the road
and the vehicle’s orientation to the road. This lateral deviation in inches can then be used in the function LateralController().

LateralController() is called to determine the steering angle that minimizes the error between the vehicle’s orientation to the road and the angle of the line tangent to the curve being followed, keeping the vehicle on the centerline of the road. Once the steering angle is determined, LateralController() returns this angle that is further manipulated by the control program. The steering angle is limited to between plus and minus 45 degrees, as that is the extent of travel for the servo in use. Then, as the angle is in floating point format, it must be rounded in order to command the PIC, which must have integer data. The integer angle is then offset to create a positive number, which the PIC then interprets as a steering command as detailed in the next section. After the angle has been converted into a format that the PIC can read, it is sent using the command format described earlier and the PutMem() function.

One should note that the PutMem() function is always called twice when addressing the PIC processor. The first time PutMem() is called, it is called with the offset 1, which effectively causes the PIC processor to allow data to be asserted at the parallel slave port pins. Then, PutMem() is called with the offset 3, which causes the PIC to actually latch the data at the parallel slave port into an internal register, and to trigger the PSP interrupt to interpret the command data it has just received. PutMem() is called twice in this manner because the PIC responds to a rising edge on the read and chip select lines that control the PSP operation.

The function _c_int06() is the serial receive interrupt that is triggered when an analog to digital conversion has taken place, and the digital data is ready to be read into RAM.
The AIC unit on the DSK board is programmable to determine the sampling rate for the A/D conversions. In this case, the sampling time is set to approximately 200us in the init_aic() function. The init_aic() assembly function is responsible for initializing the AIC unit, programming the sampling times, and informing the DSP of the location of the interrupt service routine that will handle the remainder of the conversion process after the data has been converted into digital format.

More information on the PIC programming, including some more details on how the PIC interfaces with the DSP is presented in the next section.

4.3 The PIC16F874 Microcontroller

As stated in previous sections, the PIC16F874 Microcontroller is a powerful RISC processor with only 35 instructions. This section details the use of MPLAB, Microchip’s Integrated Development Environment, in developing and testing PIC code.

The PIC is programmed at the assembly level. Thus memory management and configuration of the microcontroller is left entirely to the programmer. While this may seem burdensome at times, it allows the programmer total control over the execution of the program flow.

The program is developed in the text editor provided by MPLAB. First, the device type is set to the PIC16F874. This allows MPLAB to do some checking to determine whether or not selected options are available, and to correctly assemble programs that are under development. The assembler is then invoked from the IDE, and the assembly language code is assembled into machine format that the PIC can execute directly. Once the code has been assembled, the Register Control window allows easy access to the configuration bits for the PIC. The configuration bits are used to configure
the features of the PIC. Table 4.3.1 shows the configuration bits as they are programmed for the FLASH project.

<table>
<thead>
<tr>
<th>Configuration Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchdog Timer</td>
<td>Off</td>
</tr>
<tr>
<td>Oscillator Frequency</td>
<td>HS</td>
</tr>
<tr>
<td>Low Voltage Program</td>
<td>Disabled</td>
</tr>
<tr>
<td>Brown Out Detect</td>
<td>Disabled</td>
</tr>
<tr>
<td>Memory Protect</td>
<td>Disabled</td>
</tr>
</tbody>
</table>

The Watchdog timer is disabled, as the FLASH project is currently not using this feature. In the future, the watchdog timer may be used to control device failure during runtime. Watchdog timers are used to detect failures in the field by counting down from a specified value to zero. If the timer reaches zero, it resets the processor in an attempt to recover from an error. Thus, an interrupt is required to reset the timer during normal operation to keep the timer from reaching zero.

PIC processors can be used at several operational frequencies, and so a configuration bit allows the selection of the desired operating frequency. HS specifies high-speed operation at 20MHz. The remaining three configuration bits allow the PIC processor to manage the contents of its re-programmable memory. The low voltage program option allows the chip to be reprogrammed serially while still in the circuit. Since reprogramming the PIC in the circuit is not required for operation in the FLASH project, this option is disabled to keep the PIC from being reprogrammed during normal operation. Brown out detection enables the processor to watch for power surges that may cause the PIC to reset. This option consumes extra power, and the functionality is unnecessary for this application, and so is disabled. Memory protection allows the PIC to
make the memory write only, adding an extra safeguard to keep the PIC from being reprogrammed accidentally. This is again a feature that is unused in this application.

Once the configuration bits are set, and the code has been assembled, the PIC is programmed with the PIC Start+ programming device. This programmer is attached to the host PC via a serial cable, and MPLAB programs the PIC with the code that has been developed. After the chip has been programmed and inserted into the circuit it is ready for operation. When the chip powers up, it consults the start vector to determine where the interrupt vector is located, and where program execution begins.

Using interrupt driven operations, the PIC efficiently utilizes its resources while it interfaces with the FLASH vehicle, receives commands from the DSP, and receives data from sensors that are connected directly to the PIC’s input ports. MPLAB makes the development of the program quite simple. In addition to the development environment, MPLAB also provides developers with the facilities to simulate operation of the code in the target environment.

MPLAB has facilities for managing program memory and to generate pin stimuli, which can be used to ensure developers that the program will behave under all operating conditions before the code is actually placed on the PIC chip. This can prove invaluable in developing programs for the PIC platform. However, there are operating conditions and several features of the PIC devices that MPLAB cannot simulate. For these situations, developers must find their own way to test and debug the interface circuitry and operating program for the PIC device. These conditions and caveats are listed in the MPLAB help files, freely available online from Microchip. In the next section, the
details of the assembly file developed for the PIC processor for the FLASH project are discussed.

4.4 PIC Assembly Program Description

At the start of the program, many of the PIC’s facilities are disabled, and so the program code must enable all of the features used in the program’s operation. See Appendix B for the source code listing of the PIC program for specifics.

First the PIC initializes all data variables to appropriate values, and then begins the initialization of all the facilities that will be enabled to control the FLASH vehicle. First, TMR0 is configured with a prescaler value of 64 and the global and TMR0 interrupts are then enabled.

Next, TMR1 is configured to count encoder ticks as the drive shaft turns. The timer is configured with a 1:1 prescaler, and is set to increment once for each rising edge detected on pin RC0.

TMR2 is then configured for use in generating the velocity control PWM signal. The timer is configured with a 1:16 prescaler and a 1:2 postscaler, which causes the timer to increment once every 6.4us. An interrupt is generated every time the 8-bit timer register matches the 8-bit match register, PR2, which is used to set the period of the interrupt. This interrupt is then used to generate the PWM signal as detailed later in this section.

The Capture/Compare modules are then disabled so that the pins normally assigned to this facility can be used as ordinary digital input pins for other devices, such as TMR2. Once this is complete, RC0 is made an input for TMR2.
Next, the parallel slave port is configured. The slave port is configured by setting the pins of PORTE to be inputs, as these three pins will serve as the three control signals for the PSP. The slave port is then enabled, and the PORTA pins are made outputs that will assert a high impedance signal when the system comes online. The final step in this process is to configure the analog-to-digital conversion facility to make the PORTA pins digital I/O. Otherwise, the pins are automatically assigned to the A/D unit and are unusable as input pins to the PSP.

After this has been completed, the serial port is configured for 9600 Baud and placed into continuous reception mode. Thus, whenever the serial input buffer is filled, an interrupt is triggered that serves to process any serial data that may be coming from a manual driving console.

Lastly, all the interrupt flags that will be used are cleared, and the interrupt facility is turned on, officially enabling all of the interrupts. At this point, the program simply loops waiting for serial data to be processed. The main program simply waits for commands to come from the DSP while maintaining the last known velocity and steering commands. At reset, the PIC is programmed to set the steering angle and velocity to zero while waiting for updated driving commands.

The PIC uses interrupt driven communications to receive commands and to interface with the FLASH vehicle. Several interrupt service routines (ISRs) that perform the PIC’s many duties have already been mentioned, and will now be described in more detail. When an interrupt occurs, the PIC vectors to the ISR table and immediately saves the context of the processor so that when the ISR completes, the processor can resume operations where it left off.
Once the processor context has been saved, the ISR determines the interrupt type that occurred by checking all of the interrupt flags for the interrupts that were enabled by the program. After determining which of the PIC devices interrupted the processor, the ISR clears the interrupt flag for the interrupt that occurred, and performs the necessary operations as defined by the program.

The serial interrupt is triggered whenever serial data is present in the input buffer. When a byte has been received, the ISR moves the byte to a storage location, and processing is performed to convert the data into a command for the PIC. The PIC checks for one letter commands ‘e’ to enable the vehicle, ‘d’ to disable the vehicle, ‘a’ to place the vehicle in automatic mode, allowing the DSP control, and ‘m’ to place the vehicle in manual mode, allowing control of the vehicle from a remote driving console. Three byte commands tell the vehicle that it must either steer or change the velocity. ‘V’ followed by a two digit number is a velocity command, and the two digit number is processed to convert the command data back into a binary number from 0-99. ‘S’ followed by a two-digit number is a steering command, and the two-digit number is again processed to convert the command data back into a binary number from 0-99. All commands must be terminated by a carriage return, else the command data is dropped by the PIC, which assumes that the data was corrupted in transmission.

The TMR0 interrupt is relatively simple. The timer interrupt is asserted each time the 8-bit timer register overflows from 255 to 0. The timer register is incremented once every 12.8us. Upon the triggering of the interrupt, a flag is checked to determine whether the interrupt should be setting the corresponding output pin high or low. If the interrupt is setting the pin high, the steering command value, which ranges from 85 to 148, is
subtracted from 255 and the result is placed in the timer period register. The output pin is then set high, and a flag is set signaling that the pin is currently set high. Note that 85*12.8us is 1.0ms, and 148*12.8us is 2.0ms, which covers the entire range of the steering servo.

When the timer register is incremented and overflows the next time, the flag set during the last trip through the interrupt indicates that the output pin should be set low this time. In order to maintain a constant period even though the high time of the pulse is variable, a simple subtraction is performed. The steering command value is subtracted from the value 139, which is equivalent to subtracting the high time from the total period duration of 14.6ms. Thus, the low time is variable, but the period is a constant, yielding more effective control over the servo. This low time is then subtracted from 255 and the result is placed in the period register. Since the period time of 14.6ms is out of range with a prescaler value of 64, the low time is actually applied four times. A loop count variable is decremented each time the interrupt is triggered during the low period of the pulse, and when the loop count reaches zero, the next period of the PWM signal begins.

The parallel slave port interrupt is also fairly simple in design. When data has been latched into the PSP by the DSP asserting the correct control signals, the PSP interrupt is generated. This interrupt then quickly determines whether the DSP is attempting to disable the vehicle or not. If so, then the steering is centered, the velocity is set to zero, and further commands are ignored until the vehicle is re-enabled. Assuming that the command received from the DSP enables the vehicle, the PSP interrupt then decodes the command word by checking bit 6 of the byte to determine whether the current byte is a steering or a velocity command. Once this is done, the lower 6 bits of
the command word are processed into command data for a steering or a velocity command. These values are then made available to the rest of the program, and the next time the timer interrupts for the steering and velocity signals occur, any change in the state of the vehicle’s control signals will be applied.

The TMR2 interrupt is more complicated. In addition to applying the velocity command in the same way that the steering command value is applied by the TRM0 interrupt, the TMR2 interrupt also serves to perform closed loop velocity control in order to maintain a constant velocity during any changes in grade on the scaled highway. The timer is configured to trigger once every 1ms. Thus the velocity pulse width, which is controlled over ten of these periods, operates at 100Hz. During the first of the ten sample periods, the output pin corresponding to the velocity control signal is set high, resulting in a minimum signal high time of 1ms. During the next sample period, the remainder of the signal is applied resulting in a pulse width high time of up to 2ms.

The velocity command actually takes the form of a desired speed in terms of encoder ticks per eight 1ms sample periods. A running total of the encoder ticks generated by TMR1 is kept for eight sample periods. Then, during the ninth sample period, the total is compared to the desired number of encoder ticks last received from the DSP. If the actual number of encoder ticks is greater than the desired number, then the vehicle is traveling too fast and must slow down. If the actual number of encoder ticks is less than the desired number, then the vehicle is moving too slow, and must speed up. The difference between the actual number of encoder ticks and the desired number of encoder ticks is then converted to a proportional error by dividing by eight. This error term is then applied during the closed loop velocity control.
The total number of encoder ticks over eight sample periods is saved, and compared with the previous total. Taking the difference between two consecutive totals, and dividing it by eight we calculate the derivative error. This derivative error is added to the proportional error, which is then added into the speed control variable. In this manner, P-D control is used to maintain a constant speed in much the same way that cruise control on a full-scale vehicle is implemented. Thus, given a constant desired velocity from the DSP, the PIC processor works to maintain that velocity through changes in grade on the scale highway system.

The development of this control procedure was problematic at first, because the PIC processor uses two’s complement signed arithmetic during subtraction operations, but not during addition operations. Thus, a conscious decision was made to use two’s complement numbers at all times during the closed loop speed control. This allowed the designers to implement over and under flow checks simply and uniformly to make sure that the minimum and maximum values for the speed control variable were not exceeded, and that the addition of error terms that could be positive or negative did not cause a positive speed to become negative, or vice versa. Once this decision was made, development of the P-D control loop proceeded quickly and smooth speed control was achieved.

This concludes the description of the development environment for the TMS320C31 and the PIC16F874. The next chapter details the theoretical framework for this thesis, and provides the test data obtained under laboratory conditions using the platforms as described in Chapters 3 and 4.
Chapter 5.0 Mathematical Modeling and Simulation Results

In this chapter, the mathematical modeling of the adaptive cruise control algorithm is discussed in detail. First the kinematic model adapted from [26] is presented and the implications in using this model are discussed. Then the development of the control law used in this project is detailed, followed by an analysis of the control law. Finally, the modeling and simulations completed in Matlab for control of the first order system, and for a sliding mode control implementation are shown and discussed.

5.1 The Kinematic Model

The kinematic model presented here was adapted from [26] and was developed for the control of nonholonomically constrained car-like robot systems. Nonholonomic systems are systems in which the instantaneous velocities of system components are restricted, thereby limiting the local movement of the system. However, by executing a series of local movements, it is still possible to make a global change in the system that seems impossible given the local velocity restraints.

For example, take the parallel parking of a car. It is immediately obvious that the car cannot move instantaneously sideways to pull into a parking space, but by executing a series of forward and backward maneuvers, it is possible to park the car in such a space. Thus it is possible to create such movement in the system globally, even though the velocity of the vehicle is constrained such that it cannot move instantaneously sideways. This assumes the fact zero wheel slippage, and important assumption to note. However, in practice, wheel slippage in both scale and full sized vehicles can be shown to be small enough to be negligible.
Taken from [26], this thesis is concerned with the problem of path following, which is in turn a subset of the point-to-point stabilization problem in which a vehicle “must reach a desired goal configuration starting from a given initial condition. [26]” Obviously, the path following problem is a relaxed form of the point-to-point stabilization problem as the vehicle must only be able to follow a path, and is not required to have a specific goal orientation that it must satisfy by the end of any applied procedure.

Figure 5.1.1(a) illustrated the typical point-to-point stabilization problem while 5.1.1(b) shows path following problem. Figure 5.1.2 illustrates the model for the generalized coordinates of a car-like robot. In this figure $l$ is the length of the car body measured from the rear axle to the front axel. $\dot{\theta}$ is the orientation of the vehicle to the x-axis, and $\dot{\phi}$ is the angle of the front wheels relative to the car body. It is easy to see that for a rear-wheel drive system the kinematic model of the vehicle’s motion can be represented by

$$
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{\theta} \\
\dot{\phi}
\end{bmatrix} =
\begin{bmatrix}
cos\theta \\
\sin\theta \\
tan\phi/l \\
0
\end{bmatrix}
\begin{bmatrix}
v_1 \\
v_2
\end{bmatrix}
+ 
\begin{bmatrix}
0 \\
0 \\
0 \\
1
\end{bmatrix}
$$
In this set of equations, \( v_1 \) and \( v_2 \) are the driving and steering velocities, respectively. Given this format for the kinematic model, we then convert the system to a canonical form called chained form. A (2,n) chained form of a two-input driftless control system has the following format.
\[ x_1 = u_1 \]
\[ x_2 = u_2 \]
\[ x_3 = x_2 u_1 \]
\[ \vdots \]
\[ x_n = x_{n-1} u_1 \]

Note that although the chained form is nonlinear, it maintains a linear structure that is clearly visible when \( u_I \) is made a function of time, and is no longer regarded as a control variable. At this point, this equation becomes a single-input timer-varying linear system [26].

Figure 5.1.3 Coordinate definition for a path following task [26]

Figure 5.1.3 illustrates the coordinate system for the path following task performed for this thesis. The definitions of \( \dot{e}, \ddot{e}, \) and \( l \) remain as before. The additional variables \( d \) and \( \dot{e}_r \) represent the distance of the rear axle from the path and the angle of the tangent line to the curve being followed. \((x, y)\) is the Cartesian location of the midpoint of the rear axle. Note that without loss of generality, the car-like robot can be viewed using
the bicycle model shown in figure 5.1.3, and in this case \((x,y)\) becomes simply the position of the rear wheel of the condensed model.

Finally, the parameter \(s\) is introduced. Given a path to follow, this path can be treated as a manifold, and thus we can follow the path as it is parameterized by its arc-length, \(s\). [26] also defines \(\dot{\theta}_p\) as \(\dot{\theta}_p = \dot{\theta} - \dot{\epsilon}_t\). Thus, the curvature of the path can be represented by

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

which after some manipulation can be shown to be equivalent to

\[
\dot{\theta}_p = c(s) \dot{s}
\]

Thus, it can easily be seen that

\[
\begin{align*}
\dot{s} &= v_1 \cos \theta_p + \dot{\theta}_p d \\
\dot{\theta}_p &= v_1 \sin \theta_p
\end{align*}
\]

By combining these equations with the kinematic model for a rear-wheel drive vehicle as given earlier, the model in terms of path coordinates \(q_p = (s, d, \dot{\theta}_p, \dot{\epsilon})\) is seen as

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

Then, by applying the following transformation, this model can be placed into (2,4) chained form.
\[ x_1 = s \]

\[ x_2 = c'(s)d \tan \theta_p - c(s)(1 - dc(s)) \frac{1 + \sin^2 \theta_p}{\cos^2 \theta_p} + \frac{(1 - dc(s))^2 \tan \phi}{l \cos^3 \theta_p} \]

\[ x_3 = (1 - dc(s)) \tan \theta_p \]

\[ x_4 = d \]

with the input transformation

\[ v_1 = \frac{1 - dc(s)}{\cos \theta_p} u_i \]

\[ v_2 = \alpha_2(q_p)(u_2 - \alpha_1(q_p)u_i) \]

where \( c'(s) \) is the derivative of the curvature with respect to \( s \) and

\[ \alpha_1 = \frac{\partial x_2}{\partial s} + \frac{\partial x_2}{\partial d}(1 - dc(s)) \tan \theta_p + \frac{\partial x_2}{\partial \theta_p} (\tan \phi (1 - dc(s))) - c(s) \]

\[ \alpha_2 = \frac{l \cos^3 \theta_p \cos^2 \phi}{(1 - dc(s))^2} \]

Note that this form is locally defined in open connected domains, because

\[ \theta_p = \frac{\pi}{2} \pm k \pi, k \in \mathbb{N} \]

must be excluded because of the physical constraints of the system

and that when \( c(s) = 0 \) we have obtained the Euclidian path coordinate system again.

Once we have achieved this form, the problems of lateral control and longitudinal control are completely decoupled, and may be solved independently. In this thesis, the lateral controller developed in [26] was implemented by Patricia Mellodge and developed in tandem with the longitudinal controller presented next.

### 5.2 Headway controller algorithm development

Given the framework developed above for transforming the problem space into arc length parameters, developing a controller for maintaining a specific distance from objects in front of a vehicle is relatively simple.
First, the error term that must be controlled to maintain a specific distance from objects in front of the vehicle is defined to be

\[ e_h = s_1 - s_2 - h \]

Where \( e_h \) is the error in the headway, \( s_1 - s_2 \) is the distance between the two vehicles in terms of the arc length parameter, and \( h \) is the desired headway distance to maintain between the two vehicles. The dynamics of this equation are then given by

\[ \dot{e}_h = \dot{s}_1 - \dot{s}_2 = \omega(t) - f(c)v_1 \]

where \( \dot{u}(t) \) is the estimated velocity of the lead vehicle in terms of arc length at the current sample time, and \( f(c) \) represents the dynamics of the follower given by the transformation from the kinematic model in terms arc length

\[ f(c) = \frac{\cos \theta_p}{1 - dc(s)} \]

Clearly \( f \) is a function of the curvature of the path to be followed, and thus the lateral controller is decoupled from the headway controller, as mentioned previously. \( v_1 \) is the driving velocity of the input and can be chosen to linearize this system, as the following input

\[ v_1 = \frac{1}{f(c)} [\omega(t) + ke_h] \]

When modeled theoretically, \( \dot{u}(t) \) is simply \( \dot{s}_1 - \dot{s}_2 \). However, it is obvious that when applied practically, only an estimation of the dynamics of the leading vehicle can be obtained.

In this case, the estimate of the distance between the two vehicles is obtained from the ultrasonic sensor, and is represented by \( \omega(t) \). The difference in terms of arc
length between the two vehicles can be estimated simply from the formula for arc length given by

\[ L = \left| \int_a^b \sqrt{\left( \frac{dx}{dt} \right)^2 + \left( \frac{dy}{dt} \right)^2} \, dt \right| \]

The controller implemented uses the trapezoidal rule to approximate the integral. Given a sampling interval, the difference in arc length between the two vehicles can be found simply by approximating the velocity as the distance between the two vehicles divided by the sampling time. Knowing this value for the current sampling time and for the previous sampling time, the amount of arc length between the two vehicles is estimated by

\[ \text{diff}_s = \frac{T}{2} \left( \sqrt{\left( \frac{\hat{dx}}{T} \right)^2 + \left( \frac{\hat{dy}}{T} \right)^2} + \sqrt{\left( \frac{\hat{dx}_p}{T} \right)^2 + \left( \frac{\hat{dy}_p}{T} \right)^2} \right) \]

Where \( T \) is the known sampling time, \( \hat{dx} \), \( \hat{dx}_p \), \( \hat{dy} \), and \( \hat{dy}_p \) are the estimates in the difference between the positions of the two vehicles given by the ultrasound sensor for two consecutive readings. Thus we see that the velocity of the follower vehicle is given by

\[ v_1 = \frac{(1 - dc(s))(ds_1/dt + ke_h)}{\cos \theta_p} \]

\[ e_h = \text{diff}_s - h \]

In this application, \( ds_1/dt \) is the estimated velocity of the leader vehicle given by using two consecutive readings from the ultrasound sensor to estimate the rate of closing and taking into account the known velocity of the follower vehicle. Thus, as the vehicles move around the path, assuming that the follower vehicle initially has a velocity greater
than the leader vehicle, the error term, $e_h$ will go to zero, and the input velocity of the follower vehicle, $v_f$, will go to the input velocity of the leader vehicle over time.

Practically speaking, under operating conditions it is clear that the follower vehicle would have a maximum velocity under which it is operating, as in cruise control. Thus the follower vehicle will match the speed of the leader vehicle, maintaining the correct headway distance, until such time as the leader vehicle exceeds the follower vehicle’s maximum speed. Another way to see this is the situation where there is no vehicle in front of the vehicle being controlled. In this case, there would be no change in the velocity of the follower vehicle, as there are no obstacles to impede the vehicle’s velocity.

Also, it is obvious that one would not want their vehicle to be thrown into reverse should an obstacle suddenly appear within the headway distance, and so the velocity has a lower limit of 0m/s. Thus, if for any reason an obstacle appears within the headway distance, the vehicle will simply stop as quickly as possible, and allow enough headway distance between the two objects to accumulate before matching the obstacle’s velocity.

5.3 Matlab Simulation Results

In this section, the results from the Matlab simulations run on the first order controller and the sliding mode controller are presented. Several cases are illustrated, and practical problems and solutions in implementing this controller are discussed. In all cases presented, the vehicles are placed upon a parabolic path. A parabolic path was chosen to test this algorithm as the curvature of a parabola is a well-defined function, and also corresponds well to the types of paths that were developed under laboratory conditions. Appendix C contains the Matlab code used to create these simulations. Patricia Mellodge
wrote the lateral controller and the code for running the simulated vehicle animations in Appendix C. Darren Redfern and Colin Camp he created the animation toolbox used to display the animations in Matlab.

The first case presented is an example of the situation for which the controller was designed. In this case, the lead vehicle has a constant velocity of 1.0m/s, while the follower vehicle begins with a velocity of 1.5m/s. As the follower vehicle begins to approach the lead vehicle, the follower’s velocity begins to decrease. As the distance between the two vehicles approaches the desired headway distance, the velocity of follower matches the leader’s velocity. This can be seen in figures 5.3.1 and 5.3.2, which illustrate the input velocity, as well as the transformed chained form velocity for both the leader and the follower, respectively.

![Figure 5.3.1 Leader velocity input and transformed velocity with headway of 0.5m](image)

Figure 5.3.1 Leader velocity input and transformed velocity with headway of 0.5m
Figure 5.3.2 Follower velocity input and transformed velocity with headway of 0.5m

Figure 5.3.3 Error in headway distance with headway of 0.5m
As shown in figure 5.3.3, the error in the headway distance approaches 0. Note that at approximately 3 seconds into the simulation, the error briefly increases, and then decreases below 0, and finally returns to 0. This corresponds to the increasing curvature of the path. It can be seen from Figure 5.3.1 that as the leader approaches the point of maximum curvature in the path, the vehicle’s transformed velocity as a function of arc length remains constant, but as a function of $x$ and $y$ position, it briefly decreases as the vehicle’s rear axle is pivoting about a point. Thus, as the leader’s velocity fluctuates as it corners, the follower’s velocity changes to maintain constant headway. There is some ringing about the follower’s input velocity that is introduced by the estimation of both the distance between the vehicles from the ultrasound sensor, and of the lead vehicle’s velocity. The extents of this ringing are theorized to be minimal, in practice, as the mechanical dynamics of the system cannot react fast enough to maintain the changes in
velocity shown in the simulations. Thus, the system itself may introduce some natural filtering, helping to stabilize the vehicle’s velocity.

Figure 5.3.4 shows the actual headway as perceived by the follower vehicle. As the simulation starts, the two vehicles are at a fixed distance from each other, but as the simulation progresses, the faster moving follower vehicle slows to match speed with the leader until approximately 2 seconds into the simulation, when the two vehicles are moving at equivalent velocities, and the desired headway of 0.5m has been achieved.

Note that the model indicates headway in terms of the distance between rear axles of the two vehicles. Thus, when deciding upon an appropriate headway distance, one must consider the length of the follower vehicle, as well as the distance of the ultrasound sensor from the rear axle.

In this second example, the headway distance was set to 0.9m. Notice that the vehicle’s match velocity at 1s, which is faster than the time required in the first example. This is, of course, because the desired headway distance is achieved sooner. The same ringing is present in the changes in the velocity of the follower vehicle, which in turn induces the ringing present in the error term. Again, at the time of development it was theorized that the mechanical systems would help to smooth these effects out. The practical results of this controller as implemented on the FLASH vehicle are presented in Chapter 6.
Figure 5.3.5 Leader velocity input and transformed velocity with headway of 0.9m

Figure 5.3.6 Follower velocity input and transformed velocity with headway of 0.9m
Figure 5.3.7 Error in headway distance with headway of 0.9m

Figure 5.3.8 Headway distance between two vehicles with headway of 0.9m
Illustrated in this third example is the condition where the two vehicles have equal velocities of 1.0m/s when they start outside of the headway distance.

Figure 5.3.9 Leader velocity input and transformed velocity with headway of 0.9m

Figure 5.3.10 Follower velocity input and transformed velocity with headway of 0.9m
Figure 5.3.11 Error in headway distance with headway of 0.9m

Figure 5.3.12 Headway distance between two vehicles with headway of 0.9m

In this situation, the follower’s velocity input remains at its maximum as the two vehicles never come close enough to affect the velocity of the follower. Notice in Figure 5.3.12 that the headway distance does indeed decrease considerably as the vehicles
approach the point of maximum curvature at approximately 3.5s. This is perhaps a subtle point in the development of the controller. The model is based upon the parameterized arc length, and while the distance in terms of arc length remains constant, it should be observed that the Euclidean distance between the vehicles is changing. This is why we see the drop in the headway distance in Figure 5.3.12, and yet the follower vehicle’s velocity remains constant. As the vehicles approach the curve, the perceived distance between the vehicles does decrease, although the arc length between the two vehicles remains nearly constant.

In the next simulation, the two vehicles were initialized to be inside the desired headway distance with equal velocities. It can be seen in Figures 5.3.13 and 5.3.14 that the velocity of the leader remains as it has through all the simulations, while the velocity of the follower is forced down from its initial 1.0m/s to nearly 0m/s, where it rises slowly to allow the leader time to outdistance the follower. Note that the transformed velocity does spike to a high value in order to maintain the headway distance once the vehicles are far enough apart. Figure 5.3.15 shows that the error term is effectively kept at 0 while Figure 5.3.16 shows the headway distance maintained during the simulation. As the two vehicles approach the point of maximum curvature, the distance again becomes smaller, and then increases back to the desired headway. In this case, the leader vehicle actually increases the distance between the two vehicles to the point where it is greater than the desired headway as the two vehicles round the corner, and so the follower vehicle is limited to its maximum velocity, and continues to maintain the new distance of 0.53m between the two vehicles as the maximum velocities are in fact equal at 1.0m/s.
Figure 5.3.13 Leader velocity input and transformed velocity with headway of 0.5m

Figure 5.3.14 Follower velocity input and transformed velocity with headway of 0.5m
After developing this simple control for the vehicle, sliding mode controller was developed and simulated. Sliding mode controllers are stable and simple to develop, but
because of chattering across the sliding surface, can be seen to be hard on the physical actuators in the system. This will become evident as the graphs of the velocities show that the follower vehicle’s velocity fluctuates rapidly between extremes.

The development of the sliding mode control is as follows

\[ x = s_1 - s_2 - h \]
\[ s = x \]
\[ s = x = s_1 - s_2 = \omega(t) - f(c)v_1 \]
\[ v_1 = f(c)^{-1}[\hat{u} + k\text{sign}(s)] \]
\[ \hat{u} = \omega(t) \]

Where

\[ k \geq \beta(F + \eta) + (\beta - 1)\hat{u} \]
\[ \beta = \sqrt{\frac{f_{\text{max}}(c)}{f_{\text{min}}(c)}} \]
\[ \left| \hat{\omega}(t) - \omega(t) \right| \leq F \]
\[ \eta > 0 \]

In this case, it is assumed that the estimate of the position and velocity of the follower is sufficient such that \( f(c) \) can be cancelled exactly, making \( \beta = 1 \). On the other hand, \( \hat{\omega}(t) \) is the estimate of the information concerning the lead vehicle as given by the ultrasound sensor. This controller was designed allowing the range \( 0.5 \text{m/s} \leq \omega(t) \leq 1.5 \text{m/s} \) as these are the known bounds on the lead vehicle in simulation as well as the bounds placed on the test vehicle’s velocity in laboratory exercises. In the following simulations \( k \) was chosen experimentally to be 3.5.

In this first simulation, the headway was 0.9m, and the follower vehicle’s initial velocity was 1.5m/s while the leader’s constant velocity was 1.0m/s.
Figure 5.3.17 Leader velocity input and transformed velocity with headway of 0.9m

Figure 5.3.18 Follower velocity input and transformed velocity with headway of 0.9m
Figure 5.3.19 Error in headway distance with headway of 0.9m

Figure 5.3.20 Headway distance between two vehicles with headway of 0.9m
It is immediately obvious from Figure 5.3.18 what the physical effects of sliding mode control look like. Chattering about the sliding surface, which can be seen as the oscillation about 0.9m in the headway distance in Figure 5.3.20, results in the vehicle velocity oscillating rapidly between the extremes of 0m/s and 1.5m/s. This results in the follower vehicle starting and moving at maximum velocity until it reaches the headway distance behind the lead vehicle, at which point it stops moving until the lead vehicle moves outside the headway distance again. This type of erratic motion would be uncomfortable for a human driver. It was surmised that the dynamics of the velocity in the physical implementation are such that these changes in velocity are not physically possible, and that this would result in smoother changes in velocity, and hence smoother motion overall.

In the second case, the headway was set to 0.5m while the velocities of the leader and the follower were 1.0m/s and 1.5m/s, respectively. Again, note the oscillations in the velocity of the follower in Figure 5.3.22 and how that would physically impact the system. It is interesting to see that the system introduces these oscillations in favor of allowing more positive error in the system (Figure 5.3.23). In other words, the system favors allowing more headway room between the two vehicles, which is safer than allowing the system to spend more time just inside the boundary set by the desired headway distance. It is easy to see from Figure 5.3.24 how well the system behaves in terms of maintaining the desired headway distance. In all the examples shown here, the headway is maintained steadily with minimum variation due to chattering across the sliding surface.
Figure 5.3.21 Leader velocity input and transformed velocity with headway of 0.5m

Figure 5.3.22 Follower velocity input and transformed velocity with headway of 0.5m
Figure 5.3.23 Error in headway distance with headway of 0.5m

In this next example, it is interesting to note the lack of effect from chattering. Here the two vehicles began with equal velocities of 1.0m/s and were initially farther apart than the 0.5m headway distance. As the controller implemented for the follower
vehicle does not reach the sliding surface, this controller behaves exactly as the simple first order system developed early does.

Figures 5.3.26 through 5.3.28 illustrate the smooth functions of the velocity, as well as the error term and the actual measured headway. This behavior was expected, and is another reason that it was believed that the scale model vehicle would behave well when the sliding mode controller was applied in the laboratory.

Figure 5.3.25 Leader velocity input and transformed velocity with headway of 0.5m
Figure 5.3.26 Follower velocity input and transformed velocity with headway of 0.5m

Figure 5.3.27 Error in headway distance with headway of 0.5m
Finally, the last example shows the sliding mode control with headway of 0.5m where each vehicle has an equal velocity of 1.0m/s. Initially the follower is inside the desired headway distance. As in the non-sliding mode control example, once the vehicles pass the point of maximum curvature, the lead vehicle pulls outside of the desired following distance, and since the two vehicles have equal velocities, the follower remains at its maximum velocity at this distance.
Figure 5.3.29 Leader velocity input and transformed velocity with headway of 0.5m

Figure 5.3.30 Follower velocity input and transformed velocity with headway of 0.5m
Figure 5.3.31 Error in headway distance with headway of 0.5m

Figure 5.3.32 Headway distance between two vehicles with headway of 0.5m
The next chapter details the results of these control laws being applied to the FLASH vehicle in a laboratory setting. Simulation results indicate that the vehicle should behave as expected, and the range and sensitivity of the ultrasound sensor are sufficient to provide the vehicle with the control inputs needed to ensure stable operation.
Chapter 6.0 Application of the Control Laws

to the FLASH Project Vehicle

In this chapter, the practical implementation of the control laws developed in the
previous chapter are presented and discussed. In general, the application of these laws to
the FLASH vehicle was successful, although discrepancies between the modeled system
and the physical system exist.

6.1 Application of the first order control law

Application of the first order control of the error term in the headway distance
was successful. The FLASH vehicle, outfitted with the control law operating on the DSP
and the ultrasound sensor kit, was able to maintain a given headway distance from
moving obstacles impeding the vehicle’s motion.

In simulation, the lead vehicle’s velocity was approximated by using the
trapezoidal rule for integrations, and by approximating the difference in Cartesian
coordinates based upon the reading from the ultrasound sensor, and the vehicle’s
measured angle with respect to the path being followed. In practice, these
approximations worked remarkably well.

First we noted that the ultrasound sensor was extremely sensitive to obstacles.
While the sensor provided excellent resolution and response time, it was sensitive enough
to detect particles of saw dust present in the air in the laboratory, which cause brief, but
disturbingly large, fluctuations in the readings from the sensor. However, as suspected,
the vehicle’s mechanical dynamics were such that the vehicle could not react quickly
enough to such disturbances to affect the velocity of the vehicle noticeably.
In addition, it was also discovered that the controller had trouble maintaining the headway distance when the lead vehicle exhibited non-smooth changes in its velocity. The cruise control algorithm that allows the follower vehicle to assume that it is moving at the velocity commanded by the DSP had a coarse resolution, limited by the counter resolution available on the PIC processor. This, in turn, resulted in fluctuations in the perceived velocity of the lead vehicle, as the acceleration times of the follower vehicle were not taken into account in the model.

The fluctuations in the velocity of the follower vehicle occurred because of the following problem in the closed loop velocity controller. The vehicle detected a distance between the lead vehicle via the ultrasound sensor, and the DSP commanded the follower vehicle to slow down to match speed with the leader. However, the follower vehicle, lacking active breaking, must simply coast down to match speed. During this deceleration period, the DSP has already completed another sensor reading, and sends another command to the PIC processor to slow down again, even though the vehicle has not yet settled to the speed commanded in the previous sensor cycle. Thus we see fluctuations and ringing about the matched velocity, which in turn creates ringing about the desired headway distance.

In simulation, the trend for the vehicle was to maintain a distance outside of the desired headway distance under these circumstances. In practice, this behavior resulted in maintaining a distance closer than the desired headway because of the lag between the commanding of a velocity, and the achievement of that steady state velocity by the PIC processor. This trend to maintain a distance inside the headway distance can easily be compensated for by adding a buffer range to the desired headway distance.
It should also be noted that the adaptive cruise control law was developed assuming a specific later controller that was also derived from [26]. At the time of the development of this longitudinal controller, the lateral controller simulated was not functional, and a PID lateral controller was substituted. In practice, the PID controller implemented was shown to be stable and capable of following the path created in the FLASH laboratory, and so the substitution of this algorithm was successful under test conditions. However, the PID controller did not take into account the velocity of the vehicle when making a steering correction, as the proposed lateral controller did. Under the test conditions in the laboratory, the vehicle was given a maximum velocity of 1m/s, and was able to maintain the path. It is expected that as the maximum velocity of the vehicle increases, its ability to maintain the path without considering the velocity of the vehicle will decrease.

The test vehicle also had difficulty when presented with a stationary object in its path. Again, the resolution of the system responsible for maintaining a constant velocity as commanded by the DSP directly affects the stability of the adaptive cruise control law. In practice, the test vehicle was not designed to come to a complete stop for the museum exhibit it was created for. Thus, when approaching a stationary object, the vehicle could not stop for more than a few seconds. The cruise control system, dependent upon readings from the encoder attached to the drive wheel, would attempt to maintain a zero velocity as commanded by the DSP. However, to accomplish this, the vehicle would speed up or slow down in order to attempt to match the correct number of encoder ticks as commanded, and thus was unable to stabilize its velocity at 0m/s before encountering stationary objects directly in its path.
An unexpected result from this test was also found when it was noticed that the ultrasound sensor was capable of detecting the guardrails placed on the mock highway. As the vehicle approached a corner, the ultrasound sensor detected the guardrail as an obstacle, and because the headway distance was chosen accidentally to be close to the distance between the edge of the track and the centerline that the vehicle was following, the vehicle began to slow down on approach to the corner. The vehicle does not stop, but merely slows down to a controllable speed for cornering. When following a vehicle, the following distance would be short enough that assuming the lead vehicle was safely negotiating the turn, the follower vehicle would also be forced to do so to maintain proper headway distance. Thus, for single car operation, the ultrasound sensor is capable of providing some forward looking information to the system, allowing the system to operate more smoothly than it was operating without such information.

6.2 Application of the sliding mode control law

Application of the sliding mode control law required a simple change to the calculation of the following vehicle’s velocity, and so was also implemented on the FLASH vehicle to compare its operation with the operation of the first order law.

As in the previous section, the vehicle dynamics provide the system with an amount of natural filtering, which does help to stabilize the system, keeping it from becoming a bang-bang system. However, the sliding mode controller still exhibited an amount of wild oscillation that made its implementation on this platform infeasible at this time.

The sliding mode controller, though simple to develop and implement, introduces a large amount of oscillation in the calculation of the following vehicle’s velocity. This,
combined with the coarse resolution of the cruise control system on the PIC, and the lack of active braking, resulted in the test vehicle frequently coming too close to and crashing into the lead vehicle. Settling times to reach a steady state velocity when following a vehicle were simply too large to be practical under these conditions.

Adjusting the gains provided some help, but gains that were large enough to provide reasonable stability, around 200, on a straightaway tended to react by leaving the path rapidly if the obstacle in front of the vehicle was removed too quickly. Lower gains, around 10, were unable to provide reasonable control, resulting in many crashes by the test vehicle.

However, when the vehicle was tested alone on the track, the sliding mode controller still exhibited the same tendency to slow down the vehicle in turns. Thus, the sliding mode controller, with some applied chattering reduction techniques may still be a valid solution to the problem of adaptive cruise control.
Chapter 7.0 Conclusions and Future Work

In this chapter the final conclusions drawn from the simulations and the practical application of the control laws to the FLASH project vehicle test bed are presented. In addition, topics of interest and future research are proposed that may further the work presented here.

7.1 Conclusions

From the computer simulations and the application of the developed adaptive cruise control laws to the FLASH vehicle we can conclude that automatic ultrasonic adaptive cruise control for a scaled robotic car is realizable.

Practically speaking, this controller was designed around operation of the FLASH scale model vehicle under road conditions that simulated suburban highway systems meant for use at speeds between 20MPH and 30MPH. The test vehicle was able to handle these conditions satisfactorily.

The following conclusions concerning the system developed and implemented are as follows:

1. The adaptive cruise control velocity is a function of the relative velocity between two vehicles and the desired headway distance, but in addition, should also take into account desired stopping criteria, including criteria for emergency situations. The system implemented was also dependent upon the acceleration of the lead vehicle, and this dependence must be removed to achieve stable operation.
2. Active braking may be required to eliminate oscillations present in maintaining the desired headway in order to remove the dependence upon the acceleration of the lead vehicle, and to handle emergency situations in which the vehicle must come to a complete stop.

3. The sliding mode controller, while simple to develop and implement, was not sufficient to create a stable operating environment for the test vehicle. The addition of acceleration components to the model, and active braking to the test vehicle may allow the sliding mode controller to be a viable option in the future.

4. In the absence of other forward looking devices that may help the FLASH vehicle to attain more stable operation, the ultrasonic sensors can also be used to detect upcoming changes in the roadway when combined with the correct elements in the track design.

7.2 Future work

While the system described in this research work is functional, there is much room for improvement. However, this system with a minimal amount of further development can also be incorporated into other systems as a basic building block.

The vehicle was capable of operating at moderate speeds and maintaining proper following distance while maintaining a path, but more research is needed to determine what, if any, dependence exists upon the velocity of the vehicle in maintaining the quantities.
Research into creating more robust control systems, and more efficient ways of estimating the parameters required in executing this system can be completed, allowing this adaptive cruise control system to be used to accomplish automatic lane changing in multi-lane highway systems. Coupled with more advanced parameter estimation systems, like vision control, which was not completed at the time of this work, more robust control can be developed as the vehicle can gather advanced information concerning conditions on the road that the vehicle otherwise wouldn’t be aware of until it had physically encountered them.

The use of sliding mode control in this situation also bears more scrutiny. Development of an active braking system for the FLASH vehicle would undoubtedly help to drastically improve the performance of the vehicle. In addition, the incorporation of acceleration information into the model of the system would also provide new insight into this system.

Also, experiments involving more than two vehicles will show how adaptable this system is in allowing platooning on highways. The system implemented is meant for use by individual vehicles on current highway systems, but its impact in platoon situations is not fully understood yet.

Finally, the relationship between the lateral and longitudinal controllers implemented on the FLASH vehicle has yet to be explored. As stated earlier, the lateral controller that was part of the simulations presented was not implemented on the test vehicle. Thus, more stable control may be realized by allowing the lateral controller to incorporate velocity information into the calculation of steering angles, which in turn can
be incorporated into estimates of maximum safe speeds with respect to the road conditions and parameters, as well as to other vehicles on the road.

There is no shortage of available research topics in ITS. It is certainly true that ITS in general is in its infancy, and leaps and bounds are sure to be made in controller technologies in the near future, along with the development of faster hardware that is capable of making more precise measurements. The design of the FLASH project museum exhibit expressly incorporates this hope as it continues its development, and along with the work presented here, should help to allow the acceptance of such technologies by the public in the future.
Appendix A
C Control Program Source Code Listing

/*******************************
* 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>

/*** FUNCTION PROTOTYPES ***/
extern void PutMem(long mem,long output,long offset);
extern void GetMem(long mem,long input,long offset);
extern void init_aic(void);
float FindError(long array, long bits, float spacing);
float LateralController(float f_error, float d_f_error, float i_f_error,
float b_error, float d_b_error, float i_b_error,long v);
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               // Transformed velocity (m/s)
#define T 0.000184         // Sampling time (seconds) (need to measure!!!)

/*** 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 d_front_error;
float i_front_error;
float back_error;    // Error determined by the rear bumper (meters)
float p_back_error;
float d_back_error;
float i_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
long counter;        // Loop counter
long dummy;          // Dummy variable
float theta_p;       // Heading angle (radians)
float c;             // Curvature (1/meters)
float v1;          // Actual control inputs (rad/s,m/s)  
float d;           // Lateral distance from line (meters)  
long dist;         // Distance as read by ultrasound sensor

double rel_distx;  
double rel_disty;  
double rel_distx_prev;  
double rel_disty_prev;  
double distancex;  
double distancey;  
double distancex_prev;  
double distancey_prev;  
double distancex_prev2;  
double distancey_prev2;  
double diff_s;  
double err_h;  
double err_h_prev;  
double headway;  
double k_h;  
double k_slider;  
double distance;

/*** MAIN PROGRAM ***/
void main(void)  
{
  init_aic();     //enable the A/D conversion

  /*** enable interrupts ***/
  asm(" or 2000h,ST");
  asm(" ldi 34h,IE");

  /*** variable initialization ***/
  d_front_error = 0;
  i_front_error = 0;
  p_front_error = 0;
  d_back_error = 0;
  i_back_error = 0;
  p_back_error = 0;
  angle = 10;
  dummy = 0;

  v1 = 1;
  k_h = 100;
  k_slider = 35;
  err_h = 0;
  err_h_prev = 0;
  headway = 0.5;
  dist = 0;
  distancex = distancey = 0;
  distancex_prev = distancey_prev = 0;

  /*** enable car for driving ***/
  //   asm(" LDI 6,IOF");   // set XF0 high to indicate program is running

  /*** 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;

/**************** *****************/
distance = (float)dist / 3150.0;  //convert from DSP units
//to meters
distancex_prev2 = distancex_prev;
distancey_prev2 = distancey_prev;
distancex_prev = distancex;
distancey_prev = distancey;
distancex = distance*cos(theta_p);
distancey = distance*sin(theta_p);

time = T + T/2.0;
rel_distx = distancex - distancex_prev + v1*cos(theta_p)*T;
rel_distx_prev = distancex_prev - distancex_prev2 + v1*cos(theta_p)*T;
rel_disty = distancey - distancey_prev + v1*sin(theta_p)*T;
rel_disty_prev = distancey_prev - distancey_prev2 + v1*sin(theta_p)*T;

//approximate the arclength between the two vehicles with the trapezoidal
//rule
diff_s = (T/2.0)*(sqrt( (distancex/T)*(distancex/T) +
           (distancey/T)*(distancey/T) ) +
           sqrt( (distancex_prev/T)*(distancex_prev/T) +
           (distancey_prev/T)*(distancey_prev/T) ) ) );

//error term to force to zero
err_h = err_h + (T/2.0)*(sqrt( (rel_distx/T)*(rel_distx/T) +
           (rel_disty/T)*(rel_disty/T) ) +
           sqrt( (rel_distx_prev/T)*(rel_distx_prev/T) +
           (rel_disty_prev/T)*(rel_disty_prev/T) ) ) );

// transform the velocity from chained form representation to a control
//input
v1 = (1 - d*c)*( (err_h - err_h_prev)/T +
          k_h*( diff_s - headway)))/cos(theta_p)/10.0;

// sliding mode transformation; used in place of above for sliding mode
//v1 = (1 - d*c)*( (err_h - err_h_prev)/T +
          k_slider*Sign( (diff_s - headway) ) )/cos(theta_p)/1.0;

err_h_prev = err_h;
/** car control **/
/* velocity info */
if( v1 < 0 )
{
    v1 = 0;
}
velocity = Round(v1*10.2666);
    if( velocity < 4 )
    {
      velocity = 4;
    }
else if( velocity > 16 )
    {
      velocity = 16;
    }
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);

/* steering info */
if (front_array == 0xFFFF)
{
  front_error = Sign(p_front_error)*IR_max;
  for (counter = 0; counter < 20; ++counter)
    {
    dummy = dummy+1;
    }
}
else
{
  front_error = FindError(front_array,IR_bits,IR_spacing);
}
d_front_error = front_error-p_front_error;
i_front_error = (p_front_error+front_error)/2;
if (back_array == 0xFFFF)
{
  back_error = Sign(p_back_error)*IR_max;
  for (counter = 0; counter < 20; ++counter)
    {
    dummy = dummy+1;
    }
}
else
{
  back_error = FindError(back_array,IR_bits,IR_spacing);
}
d_back_error = back_error-p_back_error;
i_back_error = (p_back_error+back_error)/2;
p_front_error = front_error;
p_back_error = back_error;
d = back_error;

/* determine the car's angle */
theta_p = FindHeadingAngle(front_error,back_error,L);
if ((phi >= -2) && (phi <= 2))
{
  c = 0; // straightaway
}
else
{
c = 1.125;  // 35 inch radius

phi = LateralController(front_error,d_front_error,i_front_error,back_error,d_back_error,i_back_error,velocity);

    /* Limit phi to +/- 45 degrees */
    if (phi > 31)
    {
      phi = 31;
    }
    if (phi < -31)
    {
      phi = -31;
    }

    angle = Round(phi)+32;  // phi is -31 to +31, PIC needs 0 to 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);

}  /* end while loop */

while (1)
{
    /* do nothing, should never get here */
}

}  /* end main */

}  /* 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 inches)
* 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)
{
    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+i;
        }        }
/***/ */ num should never be 0 */ error = error/(float)(num); return(error); } } /* Function: LateralController */ /* Parameters: values of the front and back errors, their derivatives and integrals, the car's velocity */ /* Returns: phi, the steering angle (in terms of the PIC control word) */ float LateralController(float f_error, float d_f_error, float i_f_error, float b_error, float d_b_error, float i_b_error, long v) { static float phi_local; // steering angle static float kp,kd,ki; // gains for PID kp = 800; // for errors in terms of meters kd = 80; // rather than in inches ki = 8; b_error = d_b_error = i_b_error = 0; phi_local = kp*(f_error+b_error)+ kd*(d_f_error+d_b_error)+ki*(i_f_error+i_b_error); return(phi_local); } /* Function: Round */ /* Parameters: x - the (float) number to round */ /* Returns: temp - rounded integer value of x */ long Round(float x) { long temp; 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); }
long Sign(float x)
{
    if (x >= 0)
        return(1);
    else
        return(-1);
}

/* Serial port 0 transmit ISR */
void c_int05()
{
    return;
}

/* Serial port 0 receive ISR */
void c_int06()
{
    asm(" LDI 080804Ch,R7");
    asm(" STI R7, @_dist");
    dist = dist & 0xFFFC;
    dist = dist>>2;
    return;
}

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);
}
Appendix B
PIC microcontroller source program listing

: looprev3.asm

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

;Version 1.3
;
;Version 1.3 uses the following command format:
; 0-5 command value for steering or velocity
; 6 0 = steering
; 1 = velocity
; 7 0 = disable
; 1 = enable
;
;The car expects a value from 0 to 63 from the DSP for the steering, with 32
;being center. The velocity command expects a desired number of encoder ticks per 14.6ms
;for the motor. MIN = 0  MAX = ?. In order to get the desired resolution in terms of the velocity pulse
;width, we sacrificed range. Using a scale value of 32, we have a resolution of 6.4us
;which in turn creates a resolution of .3ft/s in velocity. However, since TMR2 is
;an 8bit register, we can only create a maximum high time value of 200ns*32*255 = 1.632ms.
;This translates to an approximate speed of 35MPH.

char1 equ 0x20 ; serial byte 1
char2 equ 0x21 ; serial byte 2
char3 equ 0x22 ; serial byte 3
char4 equ 0x23 ; serial byte 4
Flag equ 0x24 ; flags for serial conversion and commands
speed equ 0x25 ; TMR2 control value that determines high pulse
angle equ 0x26 ; temporary steering control variable
temp equ 0x27 ; temp data storage
temp2 equ 0x28 ; temp data storage
temp3 equ 0x2e ; temp data storage
temp4 equ 0x2f ; temp data storage
steercnt equ 0x29 ; TMR0 control value that determines high pulse
STATUS_TEMP equ 0x2a ; temp storage for context switch
W_TEMP equ 0x2b ; temp storage for context switch
hold equ 0x2c ; hold var to determine which edge of pulse we are
; commanding
spdtmp equ 0x2d ; temp storage for speed conversion
command equ 0x30 ; command data from the DSP
desspeed equ 0x32 ; desired speed from DSP (ticks/8 sample periods)
loop_0 equ 0x3A ; loop to get low time for TMR0
loopcnt equ 0x33 ; loop to get low time for TMR2
temp equ 0x34 ; temp storage for speed conversion
remain equ 0x35 ; remaining time left in encoder sample period
diff equ 0x36 ;
total equ 0x37 ; total ticks per 8 sample periods
ptotal equ 0x38 ; previous total
d_err equ 0x39 ; derivative error

; Reset and Interrupt Vectors

org 00000h ; Reset Vector
goto Start

100
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'  ; 117 is the value that produces 1.5ms
movwf steercnt ; pulse width for center steering
movlw d'31'  ; 31 gives 1.2ms pulse for neutral
movwf speed
movlw d'2'  ; set to use encoder data next period
movwf loopcnt
movlw d'4' ; set to generate correct low time
movwf loop_0

; Set up tmr0 for SCP
bsf STATUS,RP0
movlw 0xd5  ; set TMR0 for prescaler=64
movwf OPTION_REG
movlw 0xa0  ;enable global and TMR0 interrupt
movwf INTCON
bcf STATUS,RP0

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

movlw 0x07  ; prescaler = 1:1
movwf T1CON  ; no sync, external clock generate, timer 1 on

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

clr T2CON ; clear T2CON
clr TMR2 ; clear Timer2
movlw 0x0F ; Enable TMR2 and set prescaler= 16
movwf T2CON ; postscalar=2

clr CCP1CON ; CCP module is off
clr 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

101
; 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
clf TRISB ; Config PORTB as all outputs
bcf TRISC.6 ; Make RC6 an output
bsf TRISC.7 ; make RC7 an input
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
bcf Flag,1

Stop
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|y)<<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
btfss STATUS,Z ; is it speed or steering?
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'
sublw 'm'; else if it was manual
btfss STATUS,Z
goto CheckA
bsf PORTA,0
bcf PIE1,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 impedance
bcf STATUS,RP0
goto EndCheck
CheckA
movf char2.0 ; else it was auto mode
sublw 'a'
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 PIE1,PSPIF
bsf PIE1,PSPIE ; Enable PSP interrupt in auto
BadCom
; add error checking stuff later

EndCheck

NoCR
    bsf PORTB,0    ; heart beat
    nop
    nop
    nop
    nop
    nop
    bcf PORTB,0
    nop
    nop
    nop
    nop
    bifs PORTA,1    ; see if vehicle is enabled
    goto disabled
    bffs PORTA,0    ; see if vehicle is in auto mode
    goto auto
    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
    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
    clr 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
    bffs 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
    bffs 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
movf 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

; Steering PWM generation interrupt. Generates steering pulses at 70Hz
; with 1.0ms left, 1.5ms center, 2.0ms right. 12.8us Resolution in pulse
; width for extra smooth steering control

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

bcf INTCON,2 ; Clear the interrupt
bsf Flag,1 ; check for serial command when done with interrupts
decfsz loop_0,1 ; dec loop counter. If 0, then the low time has expired
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 PORTB,6 ; output a high
bcf hold,0 ; toggle the hold
incf loop_0,1 ; next time in we are on the falling edge
goto Ret

soff
movlw d'4' ; 4xlow time x64 x 200ns gives us 14ms period
movwf loop_0

movfw steercnt ; divide the steering pulse time by 8
movwf stemp
movlw 0xF8 ; this calculation results in a constant
andwf stemp,1 ; pulse width period for all pulse widths
bcf STATUS,C ; and thus gives more reliable control
rrf stemp,1 ; when the high time varies from 1ms to 2ms
rrf stemp,0
sublw  d'139' ; set TMRO to put the velocity pulse
sublw  d'255' ; low for the remainder of the
           ; period
movwf  TMRO

bcf  PORTB,6 ; output a low
bsf  hold,0  ; toggle the hold

goto  Ret

; Command the PIC via the DSP. The interrupt services the commands
; from the DSP via the parallel slave port. See command format at
; program header.

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

Steer
movfw  temp3  ; if it was steering, offset the
addlw  d'85' ; command from the DSP by 85
movwf  steercnt ; 0-31 from the DSP are left, 32 is center
goto  Ret ; 33-63 are right

; Velocity PWM generation interrupt. Generates steering pulses at 70Hz
; with 1.0ms full reverse, 1.2ms neutral, 2.0ms full forward. 6.4us Resolution in pulse
; width for extra smooth velocity control

TMR2Int
```assembly
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 hold, 2 ; see if we are rising or falling
goto toff

bsf hold, 2 ; pulse width is high for 1 1ms sample
movfw speed ; if it is time to set the remaining pulse
bfsc STATUS, Z ; width high time
goto arbit

bsf STATUS, RP0 ; then put speed in the
movwf PR2 ; period register.
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 PORTB, 7 ; start the high pulse
goto Ret

arbit
movlw 0xA ; wait for 10 sample periods
movwf loopcnt ; to command the speed again
bcf hold, 2 ; special case for 1.0ms high time
bcf PORTB, 7 ; set the port low
movlw d'156' ; and config for the entire period to be low
bsf STATUS, RP0 ; period register.
movwf PR2
bcf STATUS, RP0

goto Ret

toff
bcf hold, 2 ; 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
movfw loopcnt ; commanding the motor again
bcf PORTB, 7 ; 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
bfsc STATUS, Z ; get it ready now
```

bsf hold,3
btfss hold,3 ; if we are not commanding the motor
goto encoder ; then get a sample from the encoder
bsf PORTB,7 ; set the motor pulse high for 1ms
bcf hold,3
btfsc total,7 ; negative encoder ticks, bail
goto t2on
movfw total ; derivative error calculation
subwf ptotal,1 ; difference between 8 period encoder
movwf d_err ; tick totals
movfw total ; compare the desired number of ticks/8 sample
subwf desspeed,0 ; periods to the total sampled
movwf diff
btfsc diff,7 ; if the result is negative, then we need
goto dec_speed ; to slow down
goto inc_speed

dec_speed ; slowing down
comf diff,1 ; perform the 2's complement operation
incf diff,1
movlw 0xF0 ; divide by 8 to get the proportional error
andwf diff,1
bcf STATUS,C
rrf diff,1
rrf diff,1
comf diff,1 ; perform the 2's complement operation to
incf diff,1 ; get a negative number back
goto der_err

inc_speed ; speeding up
movlw 0xF0 ; divide by 8 to get the proportional error
andwf diff,1
bcf STATUS,C
rrf diff,1
rrf diff,1
rrf diff,1
comf diff,1 ; check the derivative error
goto der_err

der_err ; derivative error calculations
movfw total
movwf ptotal ; save the encoder ticks for the next d_err
btfsc d_err,7 ; calculation and check to see if the d_err
goto dec_d_err; is positive or negative
goto inc_d_err

dec_d_err ; d_err was negative
comf d_err,1 ; perform 2's compliment operation
movlw 0xF8 ; and divide by 8 to get derivative error
andwf d_err,1
bcf STATUS,C
rrf d_err,1
rrf d_err,1
rrf d_err,1
comf diff,1 ; perform the 2's complement operation to get
incf diff,1 ; negative number back
goto adj_speed; adjust the speed
inc_d_err : d_err was positive
movlw 0xF8
andwf d_err,1
bcf STATUS,C
rrf d_err,1
rrf d_err,1
rrf d_err,1 ; divide by 8 to get derivative error
goto adj_speed ; adjust the speed

adj_speed
clrf total ; clear out the total encoder ticks
movfw diff ; sum the proportional and derivative error
addwf d_err,1
movfw d_err ; save the sum
addwf speed,1 ; add the errors to the speed
btfss d_err,7 ; check for over or under flow
goto high_test ; by the sign of the error
goto low_test

low_test ; error was negative, see if we wrapped
btfsc speed,7 ; around from negative to positive
goto c2
goto t2on

high_test ; error was positive see if we wrapped
btfsc speed,7 ; around from positive to negative
goto clamp
goto t2on

clamp
movlw d'127' ; clamp the high speed at 1.63ms
movwf speed

goto t2on

c2
movlw d'0' ; clamp the speed at 1.0ms
movwf speed

goto t2on

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

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

end
% run1.m

clear all
close all

plot_me = 0;

global c c_2;
global theta_p theta_p_2;
global d d_2;
global v1 v1_2;
global i;

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

i = 0;
t = [0, .1];

S_model = 0;
s_dot_model = 0;
S_model_2 = 0;
s_dot_model_2 = 0;
s_dot_model0 = 0;
k_u1 = 10;

x0_prev = x0 - cos(theta0)*u1*T;
y0_prev = y0 - sin(theta0)*u1*T;
x0_prev2 = x0_prev - cos(theta0)*u1*T;
y0_prev2 = y0_prev - sin(theta0)*u1*T;

x0_2_prev = x0_2 - cos(theta0_2)*u1_2*T;
y0_2_prev = y0_2 - sin(theta0_2)*u1_2*T;
x0_2_prev2 = x0_2_prev - cos(theta0_2)*u1_2*T;
y0_2_prev2 = y0_2_prev - sin(theta0_2)*u1_2*T;

distancex = x0 - x0_2;
distancey = y0 - y0_2;
distancex_prev = x0_prev - x0_2_prev;
distancey_prev = y0_prev - y0_2_prev;
distancex_prev2 = x0_prev2 - x0_2_prev2;
distancey_prev2 = y0_prev2 - y0_2_prev2;

err_h = 0;
diff_s = (T/2)*(sqrt(((x0 - x0_2)/T)^2 + ((y0 - y0_2)/T)^2) + sqrt(((x0_prev - x0_2_prev)/T)^2 + ((y0_prev - y0_2_prev)/T)^2));
err_h = err_h + (T/2)*(sqrt(((x0 - x0_prev)/T)^2 + ((y0 - y0_prev)/T)^2) + sqrt(((x0_prev - x0_prev2)/T)^2 + ((y0_prev - y0_prev2)/T)^2));

err_h_prev = 0;
TEMP(1) = (err_h - err_h_prev)/T;

v1_2_max = u1_2;
v1_2 = u1_2;

F_slider = .5;
N_slider = 1.0;
slider = .1;
slider_dot = 0;
k_slider = 3.5;

while x0 <= 1.8
i = i+1;

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

ef_2(i) = FindError(x0_2,y0_2,theta0_2,phi0_2,a,L_2,L_2);
eb_2(i) = FindError(x0_2,y0_2,theta0_2,phi0_2,a,L_2,0);

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

front_2(i) = sensor(ef_2(i),B_w,prev_front_2,sensors);
back_2(i) = sensor(eb_2(i),B_w,prev_back_2,sensors);

% 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
theta_p_2 (i) = FindHeadingAngle(ef_2(i),eb_2(i),L_2);  % actual error
theta_p_hat_2(i) = FindHeadingAngle(front_2(i),back_2(i),L_2);  % discretized error

% assign the states
% curvature
curv(i) = 2*a/((1+(2*a*x0)^2)^(3/2));
curv1(i) = -24*a^3*x0/((1+4*a*a*x0^2)^5/2));
curv2(i) = (-24*a^3*(1+4*a^2*x0^2)^(5/2)
+480*a^5*x0^2*(1+4*a^2*x0^2)^3/2)/(1+4*a^2*x0^2)^5;

if abs(phi0) > 0.0326
  curv_hat(i) = -0.1599+4.8975*abs(phi0);
else
  curv_hat(i) = 0;
end

if i > 1
  curv_hat1(i) = (curv_hat(i)-curv_hat(i-1))/T;
  curv_hat2(i) = (curv_hat1(i)-curv_hat(i-1))/T;
else
  curv_hat1(i) = 0;
  curv_hat2(i) = 0;
end

c = curv(i);
c1 = curv1(i);
c2 = curv2(i);

curv_2(i) = 2*a/((1+(2*a*x0_2)^2)^(3/2));
curv1_2(i) = -24*a^3*x0_2/((1+4*a*a*x0_2^2)^5/2));
curv2_2(i) = (-24*a^3*(1+4*a^2*x0_2^2)^(5/2)
+480*a^5*x_0^2*(1+4*a^2*x0_2^2)^3/2)/(1+4*a^2*x0_2^2)^5;

if abs(phi0_2) > 0.0326
  curv_hat_2(i) = -0.1599+4.8975*abs(phi0_2);
else
  curv_hat_2(i) = 0;
end
if i > 1
    curv_hat1_2(i) = (curv_hat_2(i) - curv_hat_2(i-1))/T;
    curv_hat2_2(i) = (curv_hat1_2(i) - curv_hat_2(i-1))/T;
else
    curv_hat1_2(i) = 0;
    curv_hat2_2(i) = 0;
end

c_2 = curv_2(i);
c1_2 = curv1_2(i);
c2_2 = curv2_2(i);

S_model0 = S_model(length(S_model));
s_dot_model0 = s_dot_model(length(s_dot_model));
S_model0_2 = S_model_2(length(S_model_2));
s_dot_model0_2 = s_dot_model_2(length(s_dot_model_2));

% 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);
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;

th_2 = theta_p_2(i);
d_2 = eb_2(i);
x2_2 = -c1_2*d_2*tan(th_2) - c_2*(1-d_2*c_2)*(1+sin(th_2)^2)/(cos(th_2)^2)
    + (1-d_2*c_2)^2*tan(phi0_2)/L_2*cos(th_2);
x3_2 = (1-d_2*c_2)*tan(th_2);
x4_2 = d_2;
X2_2(i) = x2_2;
X3_2(i) = x3_2;
X4_2(i) = x4_2;

% discretized error
th_2 = theta_p_hat_2(i);
d_2 = back_2(i);
x2_hat_2 = -c1_2*d_2*tan(th_2) - c_2*(1-d_2*c_2)*(1+sin(th_2)^2)/(cos(th_2)^2)
    + (1-d_2*c_2)^2*tan(phi0_2)/L_2*cos(th_2);
x3_hat_2 = (1-d_2*c_2)*tan(th_2);
x4_hat_2 = d_2;
X2_hat_2(i) = x2_hat_2;
X3_hat_2(i) = x3_hat_2;
X4_hat_2(i) = x4_hat_2;
% determine plant input
u2 = lateral_controller(x2_hat,x3_hat,x4_hat,u1); % discretized error
u2_2 = lateral_controller(x2_hat_2,x3_hat_2,x4_hat_2,u1_2); % discretized error

U1(i) = u1;
U2(i) = u2;
U1_2(i) = u1_2;
U2_2(i) = u2_2;

% transform the variables
th = theta_p_hat(i); % discretized error
dxds = -c2*d*tan(th)-((1+sin(th)*sin(th))/(cos(th)*cos(th)))-
   (2*(1-d*c)*d*c1*tan(phi0))/(L*cos(th));
dxdd = -c1*d*sin(th)^2/(cos(th))^2 - 2*(1-d*c)*
c*tan(phi0)/(L*cos(th));
dxds_2 = -c2_2*d_2*tan(th_2)-((1+sin(th_2)*sin(th_2))/(cos(th_2)*cos(th_2)))-
   (2*(1-d_2*c_2)*d_2*c1_2*tan(phi0_2))/(L_2*cos(th_2));
dxdd_2 = -c1_2*d*sin(th_2)^2/(cos(th_2))^2 - 2*(1-d_2*c_2)*
c_2*tan(phi0_2)/(L_2*cos(th_2));
alpha1 = dxds+dxdd*(1-d_2*c_2)*tan(th_2)+
dxdtheta_2*(tan(phi0_2)*(1-d_2*c_2)/(L_2*cos(th_2))-c_2);
alpha2 = L_2*(cos(theta_p_2(i)))^3*(cos(phi0_2))^2/(1-d_2*c_2)^2;
v1 = (1-d_2*c_2)*u1_2/cos(theta_p_hat_2(i));
v2 = alpha2*(u2_2-alpha1*u1);
V1(i) = v1;
V2(i) = v2;

% discretized error

th_2 = theta_p_hat_2(i);
dxds_2 = -c2_2*d_2*tan(th_2)-((1+sin(th_2)*sin(th_2))/(cos(th_2)*cos(th_2)))-
   (2*(1-d_2*c_2)*d_2*c1_2*tan(phi0_2))/(L_2*cos(th_2));
dxdd_2 = -c1_2*d_2*sin(th_2)^2/(cos(th_2))^2 - 2*(1-d_2*c_2)*
c_2_2*tan(phi0_2)/(L_2*cos(th_2));
dxds_2 = -c2_2_2*d_2_2*tan(th_2)-((1+sin(th_2)*sin(th_2))/(cos(th_2)*cos(th_2)))-
   (2*(1+d_2*c_2_2)*d_2_2*c1_2_2*tan(phi0_2))/(L_2_2*cos(th_2));
dxdd_2 = -c1_2_2*d_2_2*sin(th_2)^2/(cos(th_2))^2 - 2*(1+d_2_2*c_2_2)*
c_2_2_2*tan(phi0_2)/(L_2_2*cos(th_2));
alpha1_2 = dxds_2+dxdd_2*(1-d_2_2*c_2_2)*tan(th_2)+
dxdtheta_2_2*(tan(phi0_2_2)*(1-d_2_2*c_2_2)/(L_2_2*cos(th_2))-c_2_2);
alpha2_2 = L_2_2*(cos(theta_p_2_2(i)))^3*(cos(phi0_2_2))^2/(1-d_2_2*c_2_2)^2;

distance_prev2 = distance_prev;
distancey_prev2 = distancey_prev;
distance_prev = distance;
distancey_prev = distancey;
distance = x0 - x0_2;
distancey = y0 - y0_2;

rel_distx = distance - distance_prev + v1_2*cos(theta_p_hat_2(i))*T;
rel_disty = distance - distance_prev + v1_2*sin(theta_p_hat_2(i))*T;
rel_distx_prev = distance_prev - distance_prev2 +
v1_2*cos(theta_p_hat_2(i))*T;
rel_disty_prev = distance_prev - distance_prev2 +
v1_2*sin(theta_p_hat_2(i))*T;

diff_s = (T/2)*(sqrt(((distance)/T)^2 + ((distance)/T)^2) +
               sqrt(((distance_prev)/T)^2 + ((distance_prev)/T)^2));
err_h = err_h + (T/2)*(sqrt(((rel_distx)/T)^2 + ((rel_disty)/T)^2) +
                 sqrt(((rel_distx_prev)/T)^2 + ((rel_disty_prev)/T)^2));

TEMP(i) = (err_h - err_h_prev)/T;
err_h_prev = err_h;

DIFFS(i) = diff_s;
slider0 = slider(length(slider));

v1_2 = (1 - d_2 * c_2)* ( TEMP(i) +
     k_u1* ( diff_s - headway))/cos(theta_p_hat_2(i));
%v1_2 = (1 - d_2 * c_2)* ( TEMP(i) +
     k_slider* sign(diff_s - headway))/cos(theta_p_hat_2(i));

if (v1_2 < 0)
    v1_2 = 0;
elseif (v1_2 > v1_2_max)
    v1_2 = v1_2_max;
end

u1_2 = cos(theta_p_hat(i)) / ( {1 - d_2 * c_2)* ( TEMP(i) +
     k_u1* ( diff_s - headway) });
%u1_2 = cos (theta_p_hat(i)) / ( {1 - d_2 * c_2)* ( TEMP(i) +
     k_slider* sign(diff_s - headway) });

if ( u1_2 < 0)
    u1_2 = 0;
end

v2_2 = alpha2_2*(u2_2-alpha1_2*u1_2);
V1_2(i) = v1_2;
V2_2(i) = v2_2;

% update car dynamics
[t_sim state output] = sim('dynamics',[0 T]); % returns vectors x,y,theta,phi
[t_sim_2 state_2 output_2] = sim('dynamics_2',[0 T]); % returns vectors x,y,theta,phi
[t_sim_v state_v output_v] = sim('velocity',[0 T]);
[t_sim_v_2 state_v_2 output_v_2] = sim('velocity_2',[0 T]);
[t_sim_vh state_vh output_vh] = sim('error_s',[0 T]);
x0_prev2 = x0_prev;
x0_prev = x0;
x0 = x(length(x));
y0_prev2 = y0_prev;
y0_prev = y0;
y0 = y(length(y));

theta0 = theta(length(theta));
phi0 = phi(length(phi));

phi_t = phi(length(phi));
phi_t_2 = phi_2(length(phi_2));

eh_0 = eh(length(eh));
EH(i) = eh_0;

SMODEL(i) = S_model0;
SDOTMODEL(i) = s_dot_model0;

SMODEL_2(i) = S_model0_2;
while theta0 > pi
    theta0 = theta0 - 2*pi;
end
while theta0 <= -pi
    theta0 = theta0 + 2*pi;
end
PHI = [PHI phi0];
mph(i) = v1/1000*0.62137*3600*10;

x0_2_prev2 = x0_2_prev;
x0_2_prev = x0_2;
x0_2 = x_2(length(x_2));
y0_2_prev2 = y0_2_prev;
y0_2_prev = y0_2;
y0_2 = y_2(length(y_2));

X(i) = x0;
Y(i) = y0;

X_2(i) = x0_2;
Y_2(i) = y0_2;

theta0_2 = theta_2(length(theta_2));
phi0_2 = phi_2(length(phi_2));
while theta0_2 > pi
    theta0_2 = theta0_2 - 2*pi;
end
while theta0_2 <= -pi
    theta0_2 = theta0_2 + 2*pi;
end
PHI_2 = [PHI_2 phi0_2];
mph_2(i) = v1_2/1000*0.62137*3600*10;
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;

M(i) = getframe;
locate(car_2,[x0_2,y0_2 0]);
turn(car_2,'z',(theta0_2-theta0_prev_2)*180/pi);
turn(car_2.tire_fl,'z',(phi0_2-phi0_prev_2)*180/pi);
turn(car_2.tire_fr,'z',(phi0_2-phi0_prev_2)*180/pi);
theta0_prev_2 = theta0_2;
phi0_prev_2 = phi0_2;

M_2(i) = getframe;
end
% plots

t = [0:T*T*(i-1)];

figure
plot(t,EH);
xlabel('Time (s)');
ylabel('Headway Error (m)');
title('Headway Error');

figure
subplot(2,1,1),plot(t,V1)
xlabel('Time (s)');
ylabel('Input Velocity, Leader (m/s)');
title('Leader Velocity');
subplot(2,1,2),plot(t,U1)
xlabel('Time (s)');
ylabel('Transformed Velocity (m/s)');

figure
subplot(2,1,1),plot(t,V1_2)
xlabel('Time (s)');
ylabel('Input Velocity, Follower (m/s)');
title('Follower Velocity');
subplot(2,1,2),plot(t,U1_2)
xlabel('Time (s)');
ylabel('Transformed Velocity (m/s)');

figure
plot(t, DIFFS);
xlabel('Time (s)');
ylabel('Headway (m)');
title('Measured Headway');
Appendix D
Hardware Sources

Company: Bolink
Product: Legends 1/10 scale RC model car kit
Telephone: 770-963-0252 FAX: 770-963-7334
Internet: http://www.bolink.com

Company: SuperCircuits
Product: 900MHz wireless video link
Address: One SuperCircuits Plaza
Leander, TX 78641
Telephone: (512) 260-0333 FAX 260-0444
Internet: http://www.supercircuits.com

Company: Digikey
Product: Discrete electronics components
Address: 701 Brooks Avenue South
Thief River Falls, MN 56701
Telephone: 800-344-4539 FAX: 219-681-3380
Internet: http://www.digikey.com

Company: Jameco
Product: Discrete electronics components
Address: 1355 Shoreway Road
Belmont, CA 94002
Telephone: 1-800-831-4242 Fax: 1-800-237-6948
Internet: http://www.jameco.com

Company: Microchip
Product: PIC16F874-20P
Address: 2355 West Chandler Blvd.
Chandler, Arizona 85224-6199
Telephone: 480-792-7200 FAX: 480-792-9210
Internet: http://www.microchip.com

Company: Texas Instruments
Product: TMS320C31 Digital Signal Processor, TMDS3200031 C3X DSP Starter Kit
Address: 12500 TI Boulevard
Dallas, TX 75243-4136
Telephone: 800-336-5236

Company: RC Alley
Product: Trinity Paradox RC car motors, Novak Electronic speed controller
Address: PO Box 971751
Ypsilanti, MI 48197-1751
Internet: http://www.rcalley.com
Company: Mondotronic’s Robot Store  
Product: Ultrasound Owl Kit  
Address: 4286 Redwood Hwy PMB-N  
San Rafael, CA 94903  
Telephone: 800-374-5764 Fax: 415-491-4696  
Internet: http://www.robotstore.com

Company: Omnivision  
Product: OV7630 CMOS camera  
Address: 930 Thompson Place  
Sunnyvale, CA, 94085, USA  
Telephone: 408-733-3030 Fax: 408-733-3061  
Internet: http://www.ovt.com

Company: Micronas  
Product: HAL560-UAE Hall Effect Sensors  
Address: Technopark  
Technoparkstrasse 1  
CH-8005 Zurich  
Switzerland  
Telephone: +41-1-445-3960 Fax: +41-1-445-3961  
Internet: http://www.micronas.com

Company: Atmel  
Product: AT28HC64B parallel EEPROM  
Address: 2325 Orchard Parkway  
San Jose, CA 95131  
Telephone: 408-441-0311  
Internet: http://www.atmel.com

Company: Radio Shack  
Product: 7.2V NiMH RC car batteries, discrete electronic components, prototype circuit board, magnets  
Address: 300 West Third Street, Suite 1400  
Fort Worth, Texas 76102  
Internet: http://www.radioshack.com

Company: ABC-RC Hobby  
Product: Bolink Legends 1/10 scale RC model car kit  
Address: 155 W. Main St.  
Forest City,NC 28043  
Internet: http://www.abc-rc-hobby.com
Appendix E
Images of the FLASH Project Model Vehicle

Figure E1: Top view of the FLASH Project model vehicle

Figure E2: Top view of the FLASH Project model vehicle with the PIC/DSP Assembly removed
Figure E3: Front view of the FLASH Project model vehicle

Figure E4: Bottom view of the FLASH Project model vehicle
Figure E5: Ultrasound Owl Sensor Kit

Figure E6: Left: Infrared(bottom)/HED(top) sensor arrays Right: Infrared/HED control interface PCB
Figure E7: PIC/DSP Assembly

Figure E8: Left: DSP development module Right: PIC processor controller circuit board
Figure E9: RC car components and encoder location

Figure E10: Encoder optical unit and encoder wheel
References:


Richard D. Henry

VITA

Mr. Henry was born in Annapolis, Maryland in June of 1978. In December of 2000, he graduated with honors from Virginia Tech with a Bachelors Degree in Computer Engineering, and a Bachelors Degree in Computer Science. He remained at Virginia Tech through December of 2001 in pursuit of his Masters Degree in Computer Engineering, which he also completed with honors. Areas of interest and research include communications, network architecture, parallel computing, robotics, and control.

In addition, he has been a systems engineer and software developer for Computer Sciences Corporation since 1997. His responsibilities included design and maintenance of dedicated signal processing hardware, software development and revisions, and general systems administration.

While at Virginia Tech, he also enjoyed several years with the Virginia Tech Fencing Club. He was a member of the Saber team, and enjoyed competing intercollegiately during his time with the club.