Autonomous Vehicle Control using Image Processing

Nikolai Schlegel

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

Master of Science
in
Electrical Engineering

Dr. John S. Bay, Chair

Dr. Pushkin Kachroo, Member

Dr. Charles E. Nunnally, Member

January 27, 1997
Blacksburg, Virginia

Keywords: autonomous vehicle, lateral control, image processing, telerobotic operation, H-Infinity control

Autonomous Vehicle Control using Image Processing

Nikolai Schlegel

(ABSTRACT)

This thesis describes the design of an inexpensive autonomous vehicle system using a small scaled model vehicle. The system is capable of operating in two different modes: telerobotic manual mode and automated driving mode.

In telerobotic manual mode, the model vehicle is controlled by a human driver at a stationary remote control station with full-scale steering wheel and gas pedal. The vehicle can either be an unmodified toy remote-control car or a vehicle equipped with wireless radio modem for communication and microcontroller for speed control. In both cases the vehicle also carries a video camera capable of transmitting video images back to the remote control station where they are displayed on a monitor.

In automated driving mode, the vehicle's lateral movement is controlled by a lateral control algorithm. The objective of this algorithm is to keep the vehicle in the center of a road. Position and orientation of the vehicle are determined by an image processing algorithm identifying a white middle marker on the road. Two different algorithm for image processing have been designed: one based on the pixel intensity profile and the other on vanishing points in the image plane. For the control algorithm itself, two designs are introduced as well: a simple classical P-control and a control scheme based on H_∞.

The design and testing of this autonomous vehicle system are performed in the Flexible Low-cost Automated Scaled Highway (FLASH) laboratory at Virginia Tech.

I would like to thank all members of my committee for their support and encouragement. Without the input and feedback from Dr. John Bay and Dr. Charles Nunnally, this thesis work would not have been possible. My special thanks go to Pushkin Kachroo for giving me the opportunity to work with him at the Center for Transportation Research and the FLASH-Lab and for always being available with help and advice when I needed it. Furthermore, I would like to thank Dr. Joe Ball for his valuable input and the discussions about H_∞ control design. Thanks also to Bill Green for making the FLASH-Lab look the way it looks today and for his help at numerous presentations.

Most importantly, I owe a lot of thanks to my parents Kristian and Birgit Schlegel. They have given me their love, support and encouragement no matter where I was and what I have been doing. This thesis work is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Motivation

In recent years, a lot of research has been done in the area of automated vehicles and automatic highway systems (AHS), see [4], [5], [6], [7], [8], [9], [10] and many others. All this research has a common goal: to make driving on today's highways safer and easier. The requirement of making driving easier especially comes into play in lengthy trips on highways or interstates when the driving process itself is not difficult, yet the drivers' full attention is necessary to keep the vehicle on the road. Such driving is very boring and can lead to accidents when the driver gets distracted for a longer time or even falls asleep.

One step towards a solution exists already in most of today's cars. A cruise control relieves the driver from constantly having to adjust the speed of the car. Instead, a feedback loop with a controller will keep the speed constant at all times and as a consequence, the driver can take his foot of the gas pedal.

The logical consequence would therefore be to have an "advanced" cruise control that also relieves our driver from the tiring burden of steering the vehicle. Analogous to today's cruise controls that keep constant speed, such a system could keep the vehicle in one lane of a highway at all times.

A possible scenario for such an advanced cruise control could look like this: a driver steers the vehicle manually from his point of origin onto the nearest highway. Once on the highway, he will bring the vehicle into a desired lane and cruising speed and then press a button that will allow the advanced cruise control to take over. The system will keep speed and

lane for the driver, therefore relieving him of the actual driving process. The driver's task is now only to monitor the system and possibly react in an emergency.[1] Once the driver comes close to his destination, he will turn off the advanced cruise control and manually steer off the highway to his final destination.

Of course, there are lots of possibilities to even improve this scenario: the system could assist the driver not only with lane keeping but also with lane changing, it could detect emergencies like obstacles by itself and when combined with some navigation device possibly even have the vehicle under complete automated control all the way from origin to destination.

1.2 Motivation for Thesis Work

The previous section outlines the general motivation and goal of this thesis work: To design an advanced cruise control that keeps both the speed and the lane for the driver.

Since there has been so much research going on in the area of automated vehicles and intelligent highways, a number of projects exist in which such an advanced cruise control has been implemented both on full-scale and model vehicles. These projects vary in the types of sensors used for the the automated driving (infrared [1, 2], ultrasonic [1, 2], magnetic [4, 5] or video image [8, 9]) and the type of controller used (classical controller [9], fuzzy logic [7], neural network [6], H_∞ [11] etc.)

In this thesis, the above outlined advanced cruise control will be implemented on a small scale model vehicle using a video image as sensory input and several different control designs for the automated driving. A video image as sensory input was chosen because of its similarity to the way a human drives a vehicle. However, the best type of input for a full-scale automated driving system is probably a combination of the sensors mentioned above (sensor fusion). The work also includes a way to let a human driver manually steer the model vehicle from a remote control station that emulates the controls of a full-scale vehicle. This was done to fully implement the scenario of Section 1.1 on a small scale.

This thesis work has to be seen in the context of the work and the idea of the FLASH-Laboratory at the Center for Transportation Research (CTR) [1, 2]. The acronym FLASH stands for *F*lexible *L*ow-cost *A*utomated *S*caled *H*ighway. It is a small scale (approximately 1/15th) instrumented model of an AHS system and is presently in development at the

[1] This scenario is similar to the way commercial planes are flown. Once in the air, the pilots engage the autopilot and only monitors its work.

CTR. In this laboratory, small scale instrumented vehicles are used for experimentation. The laboratory is being developed so that it will have a modular and portable highway vehicle system which can be easily modified to suit various experimentation needs. This laboratory can be used to test the effects of various alternative methods for the different aspects of AHS on the system.

Examples for vehicle systems that have already been implemented or are planned for the FLASH-Lab are: a vehicle with lateral control based on infrared sensors, a vehicle that follows a lead vehicle with an ultrasonic sensor, lateral control based on magnetic markers etc. (see [1] and [2]). It is therefore desirable to also to have a vehicle system that is based on video input and image processing as part of the FLASH-Lab vehicle park, since image processing is one of the possible future ways to implement such an automated driving.

1.3 Problem Analysis

Based on what is said in previous sections, the design work to be done can be broken down into two major parts:

1. Find a way to operate a model vehicle using telerobotic remote operation while at the same time simulating full-scale driving as closely as possible.
2. Design and implement a system that will keep a model vehicle's speed constant (longitudinal control) and at the same time keeping it on the model highway (lateral control)

In the next two subsections, the main aspects of telerobotic operation and automatic control are outlined.

1.3.1 Telerobotic Operation

When operating the model vehicle, the human driver cannot be in the vehicle, for obvious reasons. Therefore, the key issue for a simulation of full-scale driving with a model car is to provide the same environment to the human driver that he would encounter during real-life driving. Ideally this environment includes visual, audio and sensory clues. Since the visual reference is the most important for driving, a simulation is normally restricted to this type of sensory input. Note that this telerobotic operation of a small scale model that simulates a full-scale vehicle has to separated from a human driving a full computer simulation of a full-scale vehicle. It is assumed here that reactions of the human driver

will be different when he is driving a real vehicle than when he is just steering a computer simulation.

Part of the environment is further to simulate the controls of a full-scale vehicle as close as possible. This includes the steering wheel, the gas pedal, a brake, a gear shift, etc. The input of the human driver to these controls should have the same effect as they would have in a full-scale vehicle. This leads to a division of the system for telerobotic operation up into two separate units or subsystems (Figure 1.1). One is the mobile model vehicle that executes the driving commands and collects the visual input whereas the other one is a stationary remote control station with full-scale driving controls operated by a human driver. Placed before the controls is a video monitor to reproduce the visual environment in front of the mobile unit.

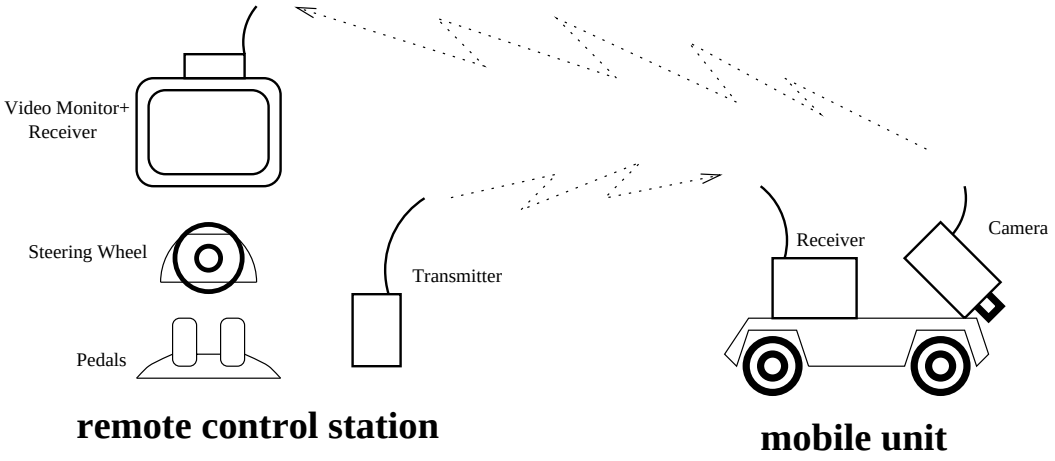

Figure 1.1: Separate subsystems for telerobotic operation

1.3.2 Automated Vehicle Control

In automated mode, the driving commands from a human driver are replaced by a controller that generates these commands from the information it gets as its input, in this case visual information from a camera mounted on the mobile vehicle.

Here, a separate stationary unit is no longer necessary, since the controller could be located on the vehicle itself. However, there are certain advantages to having the controller stationary. These include constraints in the available space as well as limited computing power on-board the vehicle. Therefore in this design, the image recorded by the camera on the vehicle is transmitted back to a computer at the remote control station. This computer has

the necessary image processing hardware and runs a program to do the automatic lateral control. The automatic lateral control based on image processing is actually a two-step process (Figure 1.2). The goal of the first step, the actual image processing, is to obtain the position of the vehicle with respect to the road from the video image. In the second step this information is used as the input to a control algorithm. The output of this algorithm will be a steering angle that will maintain the vehicle in a desired position on the road. The control algorithm can be based on classical controllers (P, PI, PID) [9], on modern robust control theory [11] or on any other method that is suitable to do the controlling (e.g. a neural net, [6]).

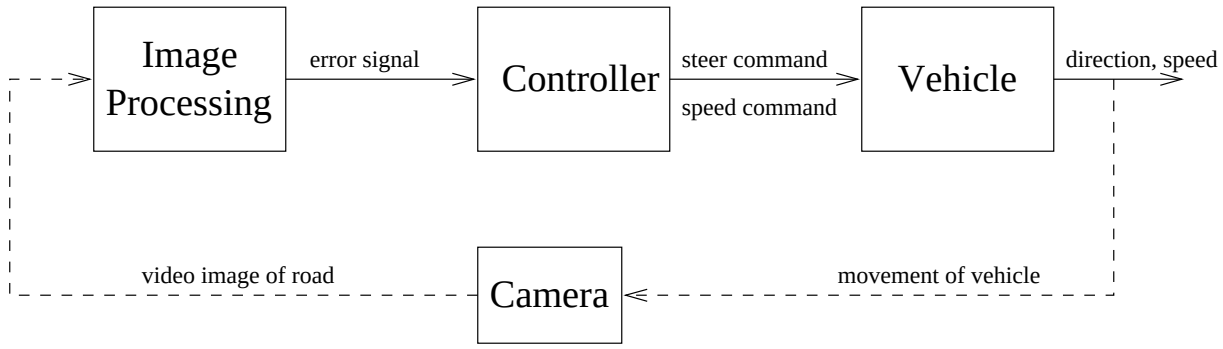

Figure 1.2: The two steps of automatic lateral control

1.4 Contributions of this Thesis

After some motivation for the problem and after giving some problem analysis, here is a quick overview about the contributions in this thesis work:

- Make one of the already existing FLASH-Lab vehicles suitable for telerobotic operation. This means mainly creating a software environment that accepts human driving commands, communicates them to the vehicle and executes them.
- Addition of the necessary hardware to the system for an automated lateral control mode based on image processing.
- Addition of modules to the existing software environment that implement the software part of the automatic lateral control based on image processing.
- Implementation and experimentation with two different methods to obtain position information from a video image.

- Design and implementation of a straightforward lateral controller based on classical control principles.
- Adaptation of an existing mathematical model for lateral vehicle control to the specific hardware in use. Use of parameter estimation for hard-to-measure parameters in the system.
- An attempt do design a modern robust controller based on the model gained by parameter estimation.

1.5 Outline of the Following Chapters

Chapter 2 goes into the details of the hardware of vehicles that are used in the FLASH-Lab for this thesis. All the hardware that is described here already existed by the time the author started working in the FLASH-Lab.

The following chapter, Chapter 3, is intended to outline the fundamentals of operating the telerobotic model vehicles. The main focus in this chapter is on the software for telerobotic operation. This includes both software running on the vehicle's computing unit (if the vehicle is equipped with one) and the software running on the control computer at the remote control station that accepts the driving commands from the human operator.

In Chapter 4, the existing telerobotic operation design is modified to include an automated control mode based on image processing. First, the additional hardware for this mode is described and then the necessary modifications to the software environment are explained.

Chapter 5 deals with different algorithms that can be used for image processing to obtain the position of the vehicle. Two algorithms are presented: one based on a pixel intensity profile, the other on edge-detection combined with an analysis of lines in the image.

The sixth chapter gives the details of the different control algorithms used for vehicle control. It starts with an explanation of a longitudinal speed controller used both in telerobotic and automated mode and then continues with the lateral steering control. For the design of the lateral control algorithm, a simple straightforward classical controller is described first. In a second approach, the chapter then outlines a way to come up with a mathematical model of the vehicle and its interaction with road by using parameter estimation. Based on this mathematical model, a robust H_∞ controller is designed for lateral control.

The thesis finishes with Chapter 7 in which some conclusions about the work done are drawn. It also gives some outlook on possible future work.

Chapter 2

Existing Hardware and Modifications

Unlike other projects in the area of telerobotic control or automated vehicles, the main focus in this project was not to make the vehicle and its supporting infrastructure as sophisticated as possible by using high-end technology like parallel computers, image processing chips and so on, but instead as it is stated in the acronym for FLASH, to make such a vehicle as cheap, easy to build, and easy to modify as possible [1, 2]. This approach allows experimentation with a number of similar model vehicles, each worth less than $1000 in hardware cost. The main issue as far as the hardware is concerned is therefore to use standard equipment like IBM compatible PCs, standard microcontrollers, a prefabricated chassis for the vehicle etc., and to try to avoid using specialized (and therefore expensive) hardware wherever it is feasible.

This chapter describes the details of the hardware used in this project. It starts with a short description of the model vehicle that was used as a basic chassis. Then the two different types of modified vehicles are described: the straightforward and easy-to-modify RC-vehicle (Type I) and a more sophisticated Type II with microcontroller and wireless serial communications link. However, even with the more sophisticated Type II vehicle, an inexpensive design was still the primary objective.

It should be mentioned at this point that most of the hardware modifications described in this chapter were not done by the author of this thesis and are not regarded part of the thesis work. Nevertheless a detailed description is given here both for the purpose of documentation and to explain the foundations on which this thesis work builds.

2.1 The Basic Frame for the Vehicle

The basic framework used for the vehicle is a model of an all-terrain (dune buggy) remote control car in the scale 1:15. In this case, a model from TAMIYA was used. Such cars are available in hobby stores like Radio Shack for less than $100.[1] Therefore, this toy model car provides an inexpensive chassis for the additions necessary in order to make it capable for telerobotic and automatic mode (see Table 2.1 and Figure 2.1).

Table 2.1: Technical data for vehicle

Name	Rookie Rabbit Off Road Racer
Manufacturer	TAMIYA
Scale	1 : 15
Motor	6 Vdc Type 540 with gear box
Drive battery	7.2V NiCd racing pack
Max. Speed	$\approx$ 15 km/H
Turning angle	$\pm 35^o$
Remote control	2 Channel transmitter / receiver operating at 27.255 MHz
Dimensions	30cm x 12cm x 6cm

In fact, any type of model vehicle could have been chosen for our purpose, since they all have similar features. Choosing an off-road all-terrain vehicle has the advantage of a very robust design.

The vehicle is equipped with a standard 6V DC motor that drives the rear wheels. With this motor, it is capable of speeds up to 10 MPH. Different speeds for the motor are achieved by using a pulse-width-modulation (PWM) of the motor voltage. The PWM signal is created in the motor-control unit (MCU) that is located on the same circuit board as the receiver electronics. The front wheels have a mechanical connection to a standard RC steering servo that is also controlled by PWM signals. This servo makes it possible to move the front wheels to an angle of about 35^o to both sides. By using two different PWM signals, it is therefore possible to control both the longitudinal and the lateral behavior of the model vehicle. One modification that was immediately applied to all model vehicles used in the FLASH-Lab is replacing the plastic off-road tires by low-profile rubber tires that are more suitable for indoor use.

The vehicle comes with a standard remote control set. It consists of a hand-held transmitter with potentiometer (variable resistor) type input devices for giving steering and speed commands. On the vehicle itself is a circuit board that contains the receiver and the motor-

[1] See the Appendix for an address list of hardware suppliers

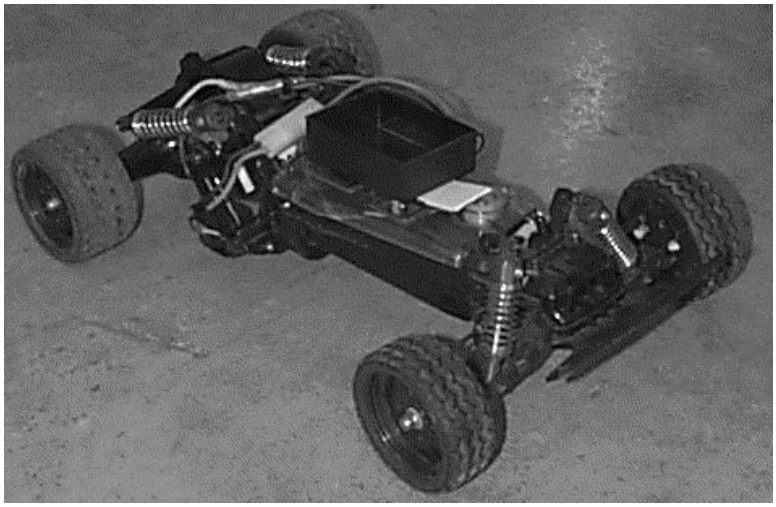

Figure 2.1: Picture of unmodified / Type I vehicle

control unit. In the MCU, drive commands from the receiver are converted to high-power PWM voltage cycles to drive the steering servo and the drive motor. All the electronics on the vehicle including the servo and the motor run of a single 7.2V NiCd battery.

2.2 Type I Vehicle: RC with Modified Transmitter

The original goal was to make as few modifications to the basic vehicle hardware as possible. The only significant alteration necessary for telerobotic operation is modifying the transmitter part of the remote control, so that it can get its input (the steering commands) from different sources. The original transmitter unit has two potentiometers (variable resistors) that are used to input the desired steering angle and speed. The actual input to the transmitter electronics is the variable resistance and therefore the voltage drop over these two potentiometers (see Figure 2.2).

Although this manual input for steering commands is still necessary for telerobotic operation, it alone is not sufficient, since it provides no way to modify, store or otherwise process the commands given by the human operator. By inserting a computer between the actual source of the driving commands and the transmitter unit, a way to process commands is made possible. In addition, the advantage of using a PC as an intermediate step is that the actual driving command input can now be selected from a number of sources. Examples are of course some type of manual input device like the previously used potentiometer.

One device that immediately comes to mind in connection with a computer is a joystick. But the driving commands could also originate from a table in the main memory of the computer, a data file or they could even be generated inside a control program (Figure 2.2).

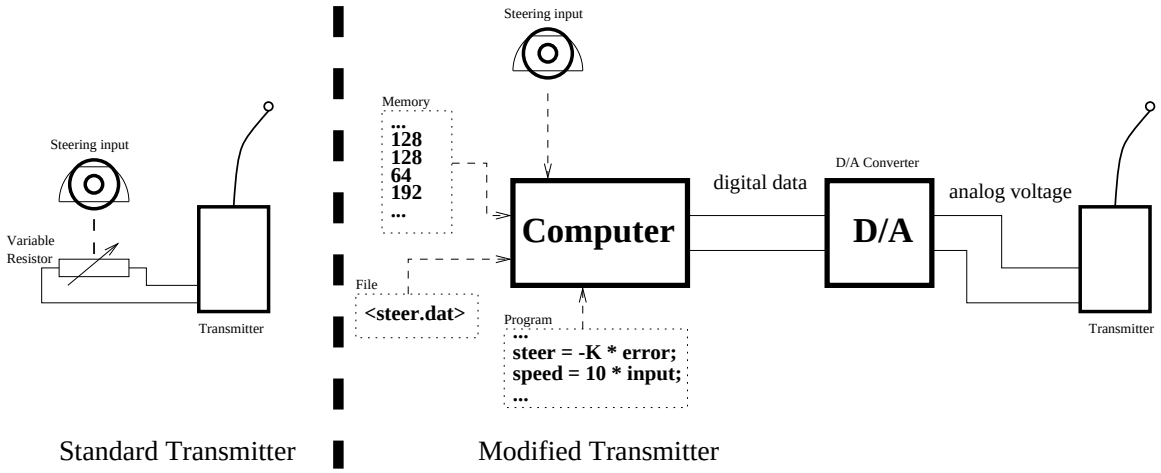

Figure 2.2: Conventional input of driving commands vs. modified input

The actual hardware modification to the transmitter unit is done by disconnecting the two potentiometers and routing their connections to the outside of the unit. The transmitter unit then has two ports that accept variable voltage levels as a measure for the desired speed or steering angle. These analog voltage levels are provided by a D/A board plugged into the computer. Passing a digital value to the board by calling one of its API functions will result in an analog voltage at one of the outputs of the board [14]. Table 2.2 shows some technical data of the D/A board in use. The control computer itself will be described in Section 3.1.2.2.

Table 2.2: Technical data for the D/A board

Name	CIO-DAS08/Jr-AO
Manufacturer	ComputerBoards, Inc.
D/A Channels	2
Resolution	12 Bit
Voltage Range	-5 V ...+5V Bipolar
Interface	Plugs into standard AT-Bus slot
(Note: Board also comes with 8-channel AD converter)	

Although this design is very simple and is useful for quick testing of the hardware, it has a number of disadvantages. The most serious one is that the control over speed and steering angle is very inaccurate due to the way the transmitter/receiver unit is designed: the steering and speed commands are transmitted as analog values in the form of varying frequencies. This poor design comes into play especially at low speeds that are therefore very difficult to achieve. Of course this limitation comes from the original design of the vehicle as a high-speed robust off-road racing car. The next section will outline how this disadvantage can be overcome with a different type of vehicle design.

2.3 Type II Vehicle: Serial Link with Microcontroller

The Type II vehicle is an improvement over the Type I design in the sense that it overcomes some of the limitations of the earlier design. This is done by applying the same general type of modifications on the receiver side as were done on the transmitter side on the first design: introducing more hardware in the path of the driving commands to make the execution of these commands more accurate and more flexible.

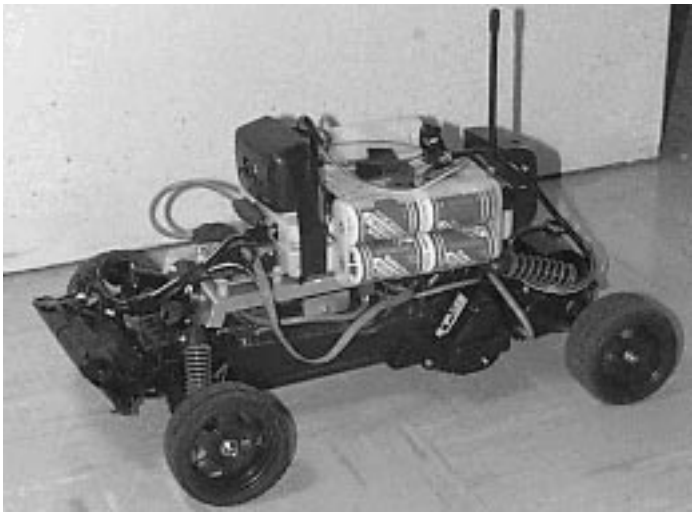

Figure 2.3: Picture of Type II vehicle

One part of this hardware modification is to use a microcontroller to interface with the motor-control unit instead of directly connecting it to the receiver. This ensures not only a more precise execution of the commands by generating well-defined signals in the microcontroller, but provides also the ability to make sure that the commands are really executed the way they are should to be (using some form of feedback).

Another part of the modification is to replace the analog transmitter/receiver pair with a wireless digital communication link. This improves the signal quality in communications and makes a two-way communication possible.

2.3.1 The Microcontroller On-Board the Vehicle

For the on-board computing device, we decided to use the 68HC11 microcontroller from Motorola [16]. The HC11 has been chosen for its simplicity of operation and availability of software. It is used here on a special board from CoActive Aesthetics called the GCB11 (see Table 2.3). It has its own embedded C code functions to simplify the task of programming the system [15].

Table 2.3: Technical Data for GCB11 board

Name	GCB11
Manufacturer	CoActive Aesthetics
Microcontroller	68HC11F1 from Motorola
Operating frequency	14.764 MHz
RAM	8K
EPROM	32K
Interface	All ports A - G of HC11, RS-232, RS-485
Software API	Standard Serial I/O (GIO), Port I/O (GAPP), multi-node Network I/O (GNET), Operating system monitor / debugger (GBUG)

These C functions, located in a 32K EPROM, are grouped into different modules. There is the GIO module used for standard I/O going over the serial RS-232 interface. The GAPP module provides functions for everything that has to do with the output-compare registers (OCx) and the input-capture registers (ICx) as well as their respective pins on the HC11. The HC11 has a number of 8-bit ports (port A, B etc.) that are used for input and output with the outside world [16, 19]. Some of these ports can be configured to perform special functions. The different bits of port A for example can be configured to interface with the above mentioned output-compare and input capture-registers (see Figure 2.4 and the explanations below and in Section 3.2.1 on how these registers can be used).

Specifically, GAPP includes functions for counting digital pulses and performing pulse-width-modulation of the supply voltage for a DC-Motor. Both types of functions are used extensively on the vehicle and will be described in more detail later. The ROM also contains the GNET module for network I/O using the on-board RS-485 connector and GBUG, a monitor/debugger to operate the GCB11 board [15].

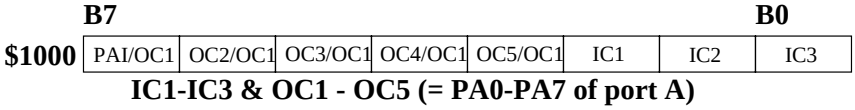

Figure 2.4: OCx and ICx register of HC11

For programming, the board also comes with 8K of RAM. The development and compilation of programs is done on a separate host computer after which the executable code is transferred to the microcontroller via the serial port of the board. The microcontroller as it is used on the model vehicle interfaces with the motor control unit that is part of the original vehicle's drive electronics. This interface consists of three of the HC11's output-compare pins (see Figure 2.5). The unit gets its input in the form of PWM voltages. In the unit itself, theses low power signals are converted to a high power output without changing the shape of the signal. This is done by utilizing two H-bridge circuits with four power transistors for the drive motor and the steering servo. See [3] to where exactly on the circuit board the H-bridges of the MCU are connected to the HC11 ports. In order two protect the sensitive ports of the HC11 from the MCU, a 7404 Inverter-IC is used for buffering.

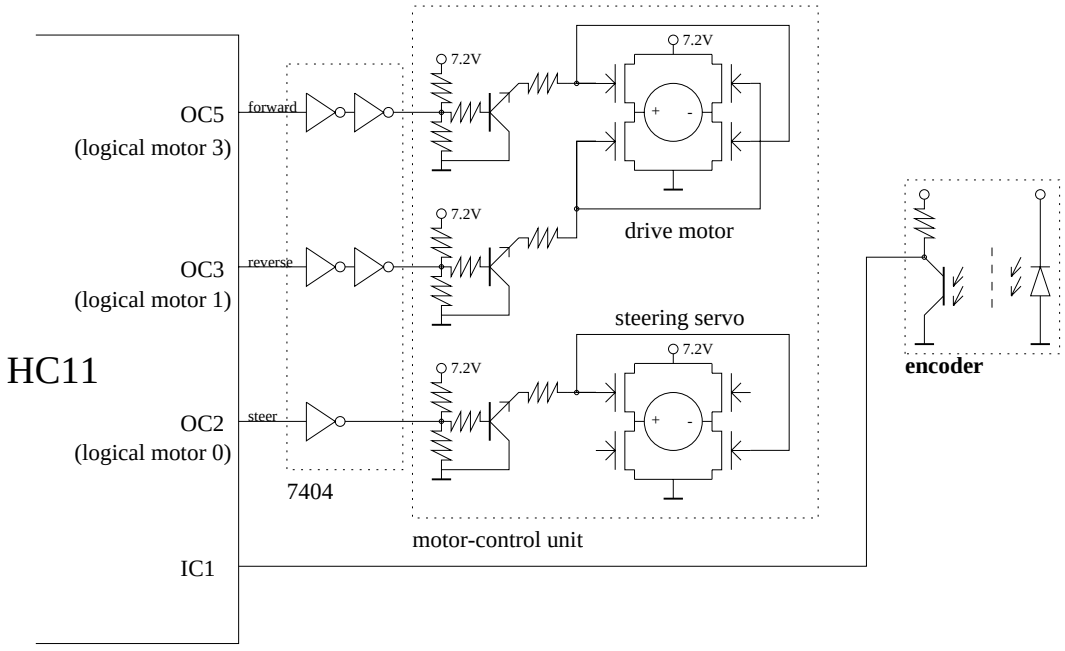

Figure 2.5: Interface to motor-control unit

On the HC11 side, the GAPP module provides all necessary functions to create varying PWM cycles on the output-compare pins. It does this by regarding each output-compare register and its corresponding pin as a logical motor. All the user has to do in software is to set the base frequency and the duration of the high period for each logical motor. The base frequency for all logical motors used is set to 15 ms. In order to move the steering servo assigned to logical motor 0 (corresponding to OC2), the PWM cycles have to be in the range given in Table 2.4 (see also [27]). Note that the steering signal is only inverted once; the servo therefore actually gets the inverse of the signal on OC2.

Table 2.4: PWM cycles for steering servo

Time	Position of front wheels
0.38 ms	far left
0.61 ms	straight ahead
0.85 ms	far right

Although physically connected to the same motor, rotating the drive motor forward and reverse is done by two different logical motors (logical motors 3 and 1, see Figures 2.5 and 2.6). The PWM signals from these two logical motors drive two different power transistors of the same H-bridge. Care must be taken that both transistors are not driven at the same time, because this will result in a short circuit in the H-bridge. The high times for the logical motors connected to the drive motor can be in the full range between 0 milliseconds to 15 milliseconds.

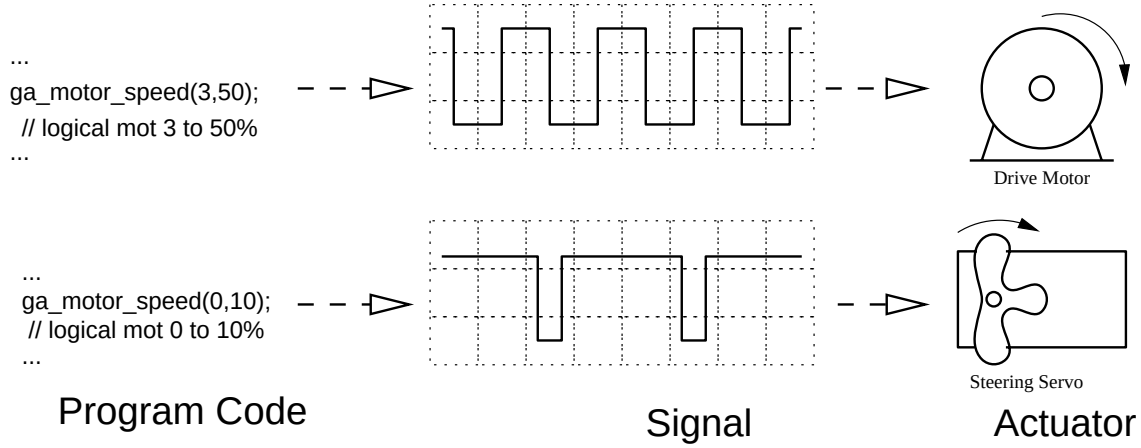

Figure 2.6: Program code → signal → movement of actuator.

As has been mentioned before, a microcontroller can also be used to ensure that driving commands are executed properly. A good example is a speed command. A desired speed is sent to the vehicle. The microcontroller on-board the vehicle measures the actual speed and compares it to the desired value. Using some kind of controller, the output of the microcontroller to the drive motor can then be altered to match desired and actual speed.

For this case it is necessary to determine the actual speed of the wheels. This can be done with an encoder that is mounted on the axle of the drive motor. When the motor is turning, this encoder will send out pulses (square waves) with the number of pulses per time period being proportional to the rotation speed of the axle. These pulses are feed into one of the input-capture registers (here IC1) where they can be counted using functions in the GAPP module (see Figure 2.7).

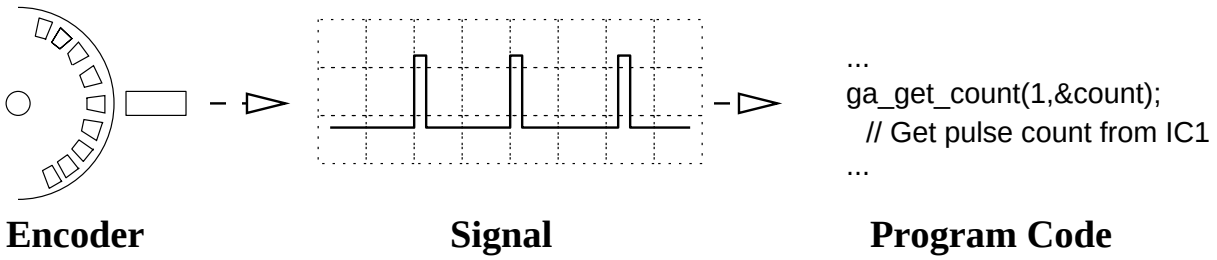

Figure 2.7: Encoder → signal → program code

2.3.2 The Wireless Serial Link

The second major change to the Type I vehicle is replacing the standard analog transmitter/receiver pair by a digital communication link. A digital communication channel is by its very nature less sensitive to noise than an analog channel. In addition to that, it allows transmission error detection and possibly correction.

For a wireless digital communication system we considered cellular, infrared, and radio technologies. All are available technologies; the cheapest is the infrared but it restricts us with limited space and range. The cellular technology right now is booming, yet it is very expensive at this time. Therefore, we chose a wireless radio modem to be the communication link. We selected the COMRAD (CCL901-DP) wireless data link radio modem (Table 2.5). This package comes with two CCL901 transceiver units, compatible software, two power adapters, two serial cables and user manuals.

Table 2.5: Technical data for wireless modem

Name	CCL901-DP wireless data link radio modem
Manufacturer	COMRAD
Frequency	902 - 928 MHz
Range	over 200 Ft indoors
Data rate	1200 - 38400 BPS (used at 19200 BPS)
Interface	RS232 (one-way or two-way)
Channels	2 out of 40 factory fixed channels
Power	6.25 - 10 Vdc
Dimensions	18cm x 10 cm x 4 cm

COMRAD assures compatibility between hardware and software by featuring a standard RS-232 connection with both full duplex and half duplex. Therefore the use of this wireless serial link does not differ from the use of a standard two-way serial null-modem cable to connect two computers. The fact that the link is wireless is completely transparent to the computers at both ends.

Chapter 3

Telerobotic Operation

3.1 Hardware for Telerobotic Operation

3.1.1 Modifications for the Vehicles

On the hardware side of the vehicles, no further modifications have to be made. Both types of vehicles described in Chapter 2 can be used in their present form for telerobotic operation.

3.1.2 The Remote Control Station

So far, the mobile part of the system has been described. Telerobotic operation however also requires a stationary unit that allows the input of driving commands. In the current layout of the system, a steering console with pedals is used. In addition to the actual input device (sensor), a subsystem to further process this input might be required. This is done by using a standard PC. A second computer might be added to supervise the work of this control computer (see Section 3.1.2.4). Finally, a communication link to the mobile unit is required, in this case the other half of the wireless serial link described in Section 2.3.2. These components together with a video monitor form the remote control station, which is the stationary part of the system. Figure 3.1 gives a complete overview on all components of the system. The modifications for automated driving mode will be discussed in Section 4.1.

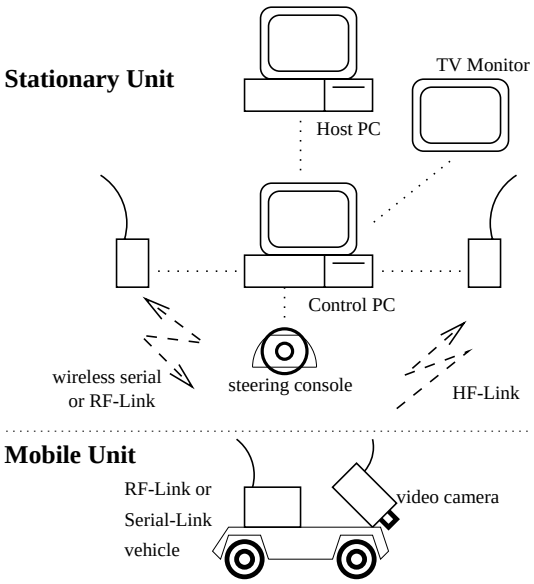

Figure 3.1: The different hardware components of the system

3.1.2.1 The Steering Console

In order to simulate the controls in a real vehicle as closely as possible in telerobotic operation, the remote control station is equipped with a steering console. This console consists of a steering wheel, a digital joystick that can be used as a gear shift, several buttons as well as a gas- and a brake pedal. The steering console in use is a Thrustmaster Formula T2, originally intended for computer games (refer to Table 3.1 and Appendix for manufacturer address). The whole console is in fact nothing but two standard analog PC joysticks in a slightly different shape. With such a standard analog PC joystick, two analog signals are created by moving it in X- and Y-directions.

Table 3.1: Technical data for steering console

Name	Formula T2
Manufacturer	Thrustmaster
Inputs	4 analog (wheel, pedals), 4 digital (gear shift, buttons)
Connection	15 pin game port

Turning the steering wheel corresponds to moving joystick 1 in the X- or Y- direction (either signal can be used). Pressing the gas pedal is equivalent to moving joystick 2 in the X-direction whereas pressing the brake pedal is the same as moving it in Y-direction. In general, a PC joystick creates a signal with a value corresponding to the proportion of the applied movement although it is not necessarily linear (moving e.g. the gas pedal down two times a distance does not necessarily create a signal twice as large). If linearization is desired, this can be achieved using a look-up table for the joystick. In this case, a linearization of the steering console inputs was not done. The steering wheel of the console is used for lateral commands to the vehicle (moving the steering servo), while the gas pedal provides an analog signal to control longitudinal movement. In the current setup, the brake pedal is not used, since the model vehicle doesn't have active brakes. Figure 3.2 shows the movement of the joysticks and their corresponding movements on the console.

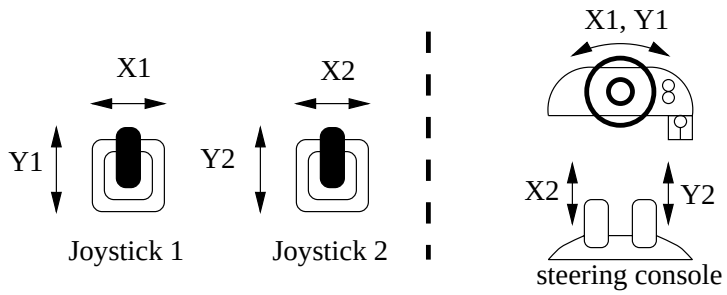

Figure 3.2: Corresponding movements on joysticks / steering console

Moving the digital joystick on the steering console back and forth corresponds to pressing the two fire buttons of joystick 1. This digital joystick is used as an "gear shift" to change driving direction from forward to reverse.

Finally the two buttons on the console, that correspond to the fire buttons on joystick 2 are used to switch from telerobotic to automated mode (see Section 4.3.2) and to turn the system off.

3.1.2.2 The Control Computer

Even though it is located outside the vehicle, the control computer is in effect "the brain" of the remotely operated vehicle, because all high-level processing is done on it. For example all driving commands go through this computer whether they originate from a human operator in telerobotic mode or from a controller in automatic mode (see 4.3.2)).

The control computer is a standard IBM 386 PC. No modifications on this PC were necessary apart from inserting the cards needed for telerobotic and automated operation (frame grabber, joystick interface and data acquisition board). Although this PC does have a monitor and keyboard for testing purposes, they will not be necessary once everything is done. The computer is running standard MS-DOS as the basic operating system. On top of DOS sits a software-kernel providing the PC with multitasking and real-time interrupt servicing capabilities (see Table 3.2).

Table 3.2: Technical data o the control computer

Name	386 - 20
Manufacturer	IBM
Operating Frequency	20 MHz
RAM	4 MB
Hard-disk	100 MB
Interface	6 standard AT-bus slots
Cards in slots	Multi-IO-card (2x serial port, 1x parallel port) Soundblaster-card (used for joystick port) CIO-DAS08/Jr-AO D/A board FF1 DSP frame grabber
Operating System	MS-DOS 5.0 + RTKernel V4.5 (see Section 3.2.2)

Connected to the control computer on the joystick port is the steering console described earlier. The COM1 serial port is used for the wireless serial link to the vehicle whereas COM2 connects the control computer to a second computer called the host computer (Section 3.1.2.4).

3.1.2.3 The Wireless Radio Modem

The same type of wireless data link radio modem described in section 2.3.2 is used at the remote control station. It is directly connected to the COM1 port of the control computer.

3.1.2.4 The Host Computer

The host computer is used to display status messages from the control program running on the control computer as well as changing the program's mode of operation. For example, the command to switch from telerobotic to automatic mode can be given both from the

steering console and the host computer. On the host computer, any terminal program (e.g. Q-Modem or the terminal program of Windows) could be used.

The reason for having a separate host computer supervising the control computer are future plans to turn the control computer into an embedded PC or single board computer (SBC). Such a SBC would be much smaller in size since it contains only the microprocessor, some memory, and the standard AT-bus (possible is also a PC104 type bus). Specifically, the SBC would not have a keyboard nor a monitor or a hard-disk connected to it. Therefore the serial connection would be the only way to communicate with this SBC (see Figure 3.3). Although it is too big for the vehicle at hand, such an arrangement could be placed on the model of a truck that is also used in the FLASH-Lab.

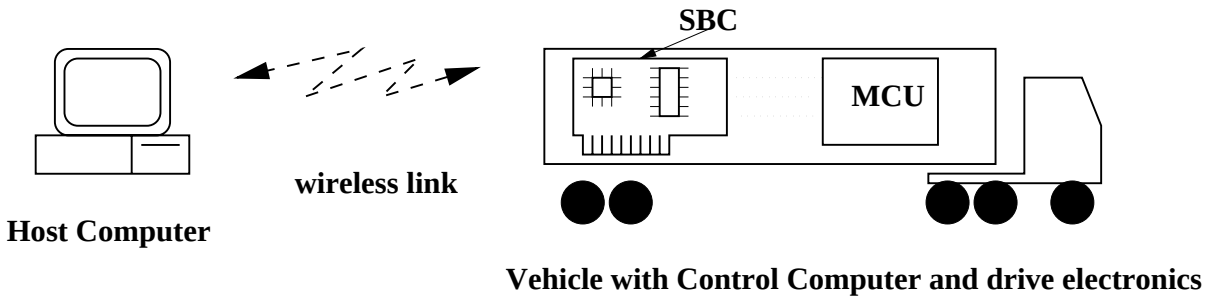

Figure 3.3: Use of Host PC in combination with an SBC

Apart from the use described above, the host computer, in this case a Pentium 90 running Windows 3.1 is also used for the software development in this project. Compiled programs are then transferred to the control computer and the microcontroller using the serial null-modem connection.

The link between host and control computer is implemented using a (possibly wireless) serial connection. However the concept is not restricted to this type of connection. Should a broader communication bandwidth be necessary, other links like Ethernet might become interesting. One example for a higher bandwidth need would be if the host computer acts as a web server at the same time, implementing a web site that shows real-time pictures from the on-board camera. [1]

[1]Such a project has been implemented with a mobile robot at Carnegie Mellon

3.2 Software for Telerobotic Operation

The same rules apply to software development as what was stated earlier for the hardware: specialized products should be avoided, instead making use of widely available tools like the Borland C compiler or even public domain software like the GNU C compiler.

3.2.1 Program on HC11 Microcontroller

When the Type II vehicle is used for telerobotic operation, the on-board microcontroller has to be loaded with a program. The purpose of this program is mainly to convert high-level commands from the control-computer into the pulse-width modulated voltages that are capable of driving either the drive motor or the steering servo. In addition to that, the program can also read an axle encoder to determine the actual speed of the motor. If desired, it is also possible to establish a closed-loop system that will try to maintain a constant axle rotation speed with changing loads on the axle.

As mentioned earlier, the GCB11 board provides a library of preprogrammed C-functions for all kinds of I/O related to microcontrollers. In order to keep the program both easily maintainable and portable at the same time, it was coded in C, with only very few inline assembler instructions. With 8 K of RAM and a (relatively) simple program, the increased code size resulting from the use of a compiler was not an issue. Assembly code was only used when there was no other implementation choice, e.g. the proper return from an interrupt service routine (ISR).

The program was developed and compiled on a MS-DOS host computer and then transferred to the microcontroller in the Motorola *.S19* object-code format. On the microcontroller, the object-code is written into memory and executed using the GBUG monitor. There are numerous C-compilers available MS-DOS Computers that are capable of producing code for the 68HC11. The one used in this project was the HC11-version of the GNU-C/C++ compiler. The GNU-C Compiler is a freeware program developed by the Free Software Foundation. Since this compiler is freeware and is distributed with all the sources, it has been adapted to produce code for almost any microprocessor in existence. Another possible freeware compiler to use would be Micro-C, a C compiler designed specifically for microcontrollers.

3.2.1.1 Interpretation of High-Level Drive Commands

In order to perform its task, the program running on the HC11 consists of two functions: a time-measuring ISR and a main function with the purpose of the time-measuring ISR being to synchronize the main loop in the main function (see below for more details).

After it has been initialized properly, the Timer-ISR is triggered every millisecond. This is done by using the HC11's free-running counter register (TCNT) and one of the four output-compare registers (OC4 in this case). The counter register is incremented every E-clock cycle (0.5 μs corresponding to 2 MHz in this case). Whenever the counter register matches one of the output- compare registers, the respective OC-interrupt will be triggered [16, 19]. Setting the output-compare register to a value of 2000 E-clock cycles ahead of the current counter value each time the ISR is called ensures that the next interrupt will occur exactly one millisecond later. All the Timer-ISR does is increment another counter each time it is called until it has counted up to 25. Then every 25 milliseconds a flag will be set. This flag is then polled in the main loop, ensuring that it will be executed once every 25 ms. See the pseudo-code for this function in Figure 3.4.

```
function Timer-ISR (called when TCNT equals value in TOC4):
    if count = 25 then
        reset count
        set flag time_passed
    end if
    clear interrupt flag for OC4
    Add # of e-clocks equivalent to one millisecond to value in TOC4
    return from interrupt
```

Figure 3.4: Pseudo-code for Timer-ISR

The main function starts by initializing the logical motors for the PWM using the functions of the GAPP-module (refer to Figure 3.5 for the pseudo-code). Although physically the same motor, the forward and reverse mode of the drive motor are treated as two logical motors by the program (see Section 2.3.1). Once the logical motors are initialized and the Timer-ISR has been set up, the main loop is started. At the beginning of the loop, the flag set in the Timer-ISR is polled to ensure that the loop is only executed every 25 milliseconds. Inside the loop, the program checks for input (steering and speed commands) coming from the control computer.

```
function main (called at program start):
    set base PWM interval for logical motors to 15 milliseconds
    initialize logical motors 0, 1 and 3
    bring all logical motors in starting position
    disable interrupts
    fill interrupt jump table with jump to Timer-ISR
    activate OC4 interrupt
    enable interrupts
    do forever
        wait until time_passed flag is set
        if new drive command arrived
            get drive command
            if STEER command change PWM of logical motor 0
            if SPEED command change PWM of logical motor motor_num
            if FORWARD command set motor_num to 3
            if REVERSE command set motor_num to 1
        end if
    end do
```

Figure 3.5: Pseudo-code for main function

The high-level commands from the control computer have the form of a sequence of one or two bytes. The first byte is the actual command while the optional second byte is a parameter for this command. Table 3.3 shows the commands that have been implemented.

Table 3.3: Driving commands for the vehicle's microcontroller

Command	Explanation
STEER (= '1')	Set new steering angle, parameter: 0=left, 127=straight, 255=right
SPEED (= '2')	Set new speed, parameter: 0=idle, 255=full speed
FORWARD (= '3')	set drive motor to forward mode
REVERSE (= '4')	set drive motor to reverse mode

Setting a new steering angle is done by changing the duration of the high period (based on the parameter with the STEER command) of the PWM for the logical motor that corresponds to the servo motor. This is done using a preprogrammed function from the GAPP module (refer to Section 2.3.1 and Figure 2.6). If no feedback for the longitudinal velocity is desired, setting the speed works in a similar fashion by varying the PWM cycles for the respective logical motors according to the parameter of the SPEED-command.

3.2.1.2 Speed Feedback

Since the Type II vehicle is already equipped with an axle encoder to measure the actual speed of the vehicle, an optional addition to the microcontroller program is to have a speed feedback control instead of the simpler feed forward control in the previous section. Such a feedback control ensures a (approximately) constant speed of the vehicle at all time.

The algorithm for the speed control could either be implemented on the microcontroller or on the control computer. In the second case, the measured speed has to be transmitted back to the control computer and a SPEED command containing the speed necessary to reach the desired speed has to be transmitted back. In order to reduce the amount of communication between the control computer and the microcontroller, the control algorithm was implemented on the microcontroller itself. This means that the SPEED command sent to the vehicle contains the desired speed that the controller tries to achieve. The actual speed can be measured by counting pulses from the axle encoder over the period of 25 ms using one of the input-capture registers of the HC11. Again, the GAPP module provides the necessary functions to do this (refer to Section 2.3.1 and Figure 2.7). The error between the desired and the actual speed is then fed into a software implementation of the controller. The actual design of the controller will be discussed in Section 6.1 in Chapter 6.4.6. The output of the controller is then used to vary the PWM cycle of the drive motor using a function of the GAPP module. Figure 3.6 shows the pseudo-code for the necessary modifications of the main loop to incorporate speed feedback.

```
function main (called at program start):
        ...
    (in init part of function main)
    Initialize COUNTER1 on IC1 to count rising edges
        ...
        (in loop part of function main)
        get current value of COUNTER1 in cnt
        reset COUNTER1 to zero
        calculate new PWM for drive motor based on last value from
          SPEED command (desired speed) and cnt (actual speed)
        change PWM of logical motor motor_num
            ...
```

Figure 3.6: Pseudo-code for speed feedback

3.2.2 Program Running on Control Computer

As indicated in Table 3.2 and in Section 3.1.2.2, the operating system running on the control computer consists of MS-DOS enhanced with a real-time capable multitasking kernel together forming a real-time operating system (RTOS). Such a RTOS is necessary when several things have to be done at the same time and an immediate response to certain events is necessary. An example in this application would be the continuous polling of the steering console, while at the same time the control program has to send commands to the microcontroller and be ready to respond to commands from the host computer. A RTOS provides all the functions and structures necessary to manage several tasks performing different duties at the same time. It will schedule the tasks in the order of their importance, assign resources and service interrupts due to external or internal events. [2]

The RTOS in use is the RTKernel 4.5 software package from On Time Informatik GmbH (See the manual [18]). This kernel normally sits on top of the normal MS-DOS, still using its functionality for things like Disk I/O etc. However, it can also be used as a stand-alone OS on an embedded PC [18]. With this, replacing the current hardware of the control computer with a SBC at a later stage is still an option (see Section 3.1.2.4 for a discussion about this).

Writing a program for the RTOS can be done with any compiler capable of producing MS-DOS executable code. The RTOS kernel is linked to the compiled program in form of a library and can be interfaced using normal C-functions. In this case, the control program was developed using the Borland C/C++ V4.5 compiler and it was written in ANSI C. Tasks in this RTOS have the form of C-functions without any parameters or return values (*void* functions). Instead of being called by the main program, these functions are registered with the kernel to become tasks (see Figure 3.7 for the tasks in the control program). This registration provides each function with a separate stack and other important data structures for the task management. Once a function/task is completely set up, it can be called by a scheduler based on its priority and certain external conditions like interrupts.

3.2.2.1 Implementation of Manual Mode

The control program for the vehicle contains a number of tasks that run all the time or are waiting to be activated by an external condition. In this section, the tasks that are necessary for the telerobotic operation in manual mode are discussed while the task specific for automated driving are discussed in Section 4.3.2.

[2] Note that it is possible to do all these things using MS-DOS alone. The real-time kernel add-on however provides an easy-to-use interface for task management and similar jobs

Following is a brief description of the tasks used in manual mode. Figure 3.7 shows how the different tasks interact with each other.

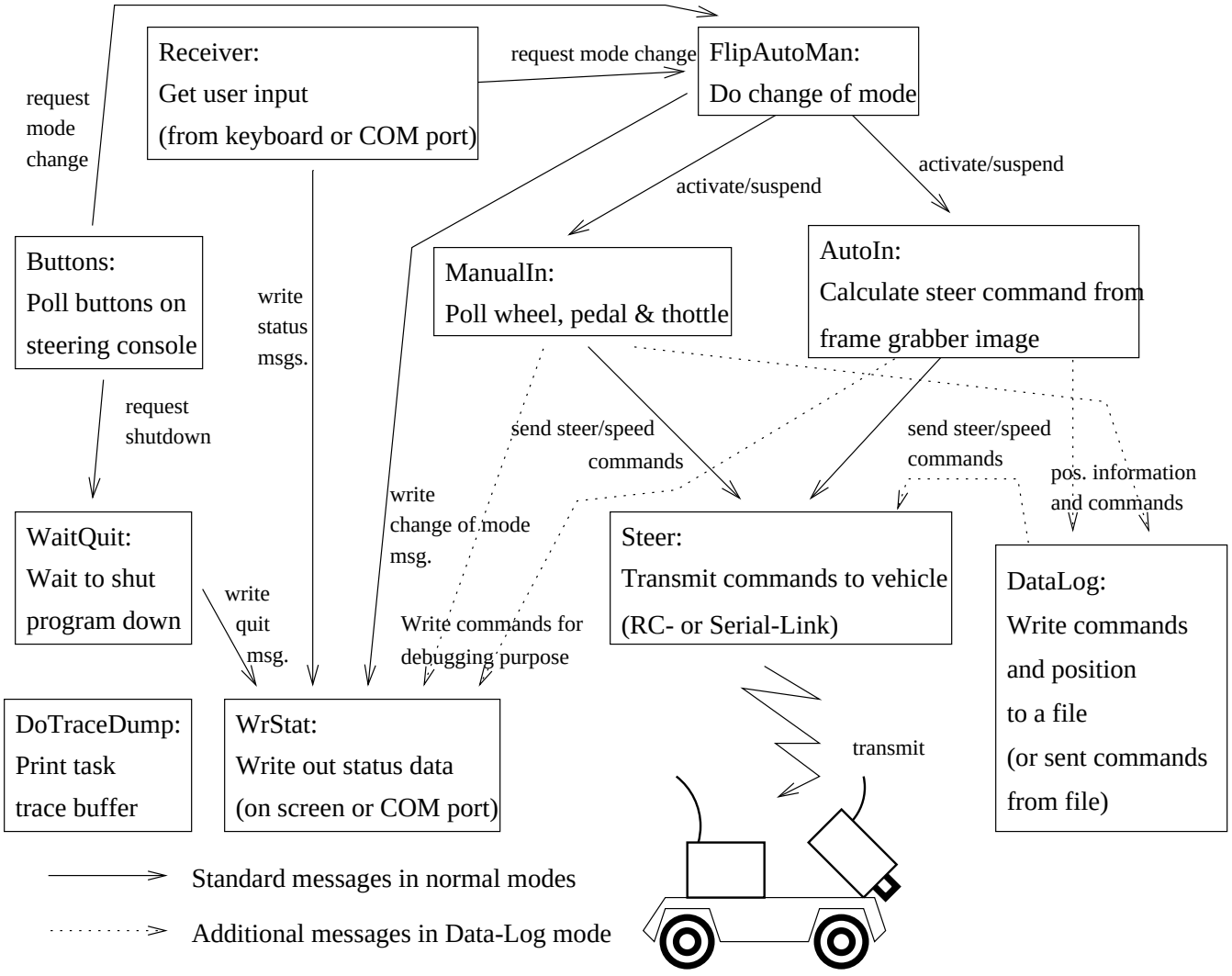

Figure 3.7: Task interaction in the control program

ManualIn: In this task, the position of the steering wheel, the gas pedal and the throttle is polled. Since the steering console is in fact two analog joysticks (see Section 3.1.2.1), this is done by polling the joystick port of the control computer. This task is called approximately once every 1/10 second. When a change in the position of one of the input devices is detected, a message is sent to the *Steer* task.

The following list summarizes the effect of the different input devices:

- The steering wheel serves as the input device for the steering angle of the front wheels.
- With the gas pedal, the PWM voltage at the drive motor is controlled, and therefore the speed of the drive motor. This control of the voltage happens either directly or indirectly using a speed controller (refer to the discussion on speed feedback in Section 3.2.1).
- The "throttle" of the steering console acts as a gear shift. Moving the throttle forward will make the car go forward when the gas pedal is pressed. Moving it backward will make the car go backwards.

At a later stage it might be useful to also use the second pedal to have an active brake for the vehicle (instead of just stepping off the gas and letting the car roll out).

Steer: This task is activated by messages from the *ManualIn* task (in the case of human operation) or from the *AutoIn* task (in the case of automatic driving). The reason for having a separate *Steer* task is that the back end of the vehicle control is the same in both cases. The term "back end" specifies the actual sending of STEER or SPEED commands to the vehicle (writing to the COM port or D/A Board).

Depending on the type of vehicle used, the following will happen to the received driving command:

- It is converted to an analog voltage between 0 and $5V$ and written to the ANALOG OUT port of the data acquisition board (see Table 2.2). The D/A board is connected to a transmitter that sends this position to the receiver on the Type I vehicle.
- It is send to the microcontroller of the Type II vehicle using the wireless serial data link described in Section 2.3.2.

When using different types of vehicles, the front end (polling the steering console or determining a steering command with the lateral control algorithm) for manual and automatic mode can stay the same, but the back end (sending the command to the vehicle) has to be changed accordingly.

Buttons: This task polls the two buttons on the steering console. The top button is used to transfer to automatic driving. The other button can be used to shut the control program down.

Receiver: This task is responsible for receiving commands either from the keyboard or the serial port (from the host computer). Depending on the commands received, it will send messages to other tasks (for example to shut down the program). It will get activated whenever a key on the control computer's keyboard is pressed or a byte is received on COM2.

WrStat: This task writes status information such as control algorithm parameters to the serial port or the screen. The task can be activated by a number of other tasks.

WaitQuit: The only job of this task is to wait for an event to shut down the control program. Such an event can either be pressing 'Q' on the keyboard of the control computer, pressing the lower button on the steering console or a message from the host computer.

DoTraceDump: This task dumps the last 64 trace buffer entries in a file. The trace buffer is used to keep track of the sequence the different tasks that were called and what event led to their activation. This is a feature of RTKernel [18] that was included in the control program for debugging purpose.

Chapter 4

Automated Vehicle Operation

4.1 Modifications for the Vehicle Hardware

It is feasible to have only the Type II vehicle run in automated driving mode. The reason for not choosing the Type I vehicle is the poor accuracy in controlling this vehicle type, especially at relatively low speeds (refer to Section 2.1). Therefore the modifications described in the following sections are applied to only the Type II vehicle.

As stated earlier, in automatic mode, the vehicle will operate without any interaction from a human operator, instead extracting all necessary steering commands from an image of the road ahead. Therefore, the vehicle described in the Section 2.3 has to be enhanced to collect the necessary information.

Note however that the video equipment described in the following sections is not only necessary for automatic driving mode. In telerobotic manual mode, the same equipment is used to record and transmit the image of road ahead of the vehicle to a TV monitor at the remote control station where it is used as visual reference for the human driver.

4.1.1 The Video Camera

The camera that is carried on the vehicle is a standard CCD (charge coupled device) video camera with a resolution of 512x492 pixels and a image frequency of 25 Hz (50 half images per second). Refer to Table 4.1 for further technical data. The camera electronics create the same signals as a normal video camera (NTSC (CCIR), with PAL being also possible),

so the signals from the camera can be displayed with a normal TV or recorded on a VCR. Modern surface mounted device techniques make it possible to enclose the CCD sensor together with all necessary electronics in a small case not larger than 2.5" x 2.5" x 2". Therefore the camera can be mounted almost anywhere on the model vehicle. The actual mounting point of the camera is on top of the wireless serial modem, in the front part of the vehicle (refer to Figure 2.3). This position ensures a good field of view over the road ahead. The camera does not face directly forward but instead is tilted downward about 5^o to make use of more area in the image plane (only the parts of the image plane showing the road are interesting for image processing).

Table 4.1: Technical data for the video camera

Name	929WS portable wireless CCD camera
Manufacturer	Goldbeam (Supplied by SuperCircuits)
Image sensor	1/3" B/W/ CCD image
Picture elements	512(H) x 492(V)
Scanning system	2:1 interscale
Scanning frequency	15.734 kHz (H) 59.94 Hz (V)
Resolution	380 TV lines (H) x 420 TV lines (V)
Lens	78^o wide-angle lens (60^o measured)
Video Output	1.0V P-P / 75Ω synchronous negative polarity
Min. illumination	less than 1 Lux
Power	12 Vdc $\approx 200mA^a$
Dimensions	5 3/4" x 1 7/8" x 2 3/4" ^{b}

[a] when used together with transmitter
[b] case includes transmitter

4.1.2 The Video Link

In addition to the serial link between the microcontroller and the control computer for the telerobotic operation, there has to be a second link to send the video signal from the camera to the remote control station. The output of the camera is connected to a transmitter (see Table 4.2) operating in the amateur TV band (around 925 MHz). In order to increase the operation range and improve signal quality, the HF output of the transmitter first goes through an amplifier (see Table 4.3) before going to the antenna. Both the transmitter and the amplifier as well as the receiver were bought from SuperCircuits.[1]

[1] Again refer to Appendix A for information about the different manufacturers

Table 4.2: Technical data for the video transmitter

Name	929WS portable wireless CCD camera
Manufacturer	Goldbeam (Supplied by SuperCircuits)
Operating frequency	916 MHz ($+-$ 1.5 MHz)
RF out level	50 mV / meter
Frequency stability	$+-$ 100kHz
Harmonic radiation	50 dB below

However, even with the amplifier, the video link is very sensitive to noise affecting the transmission. While this is not a big problem when the image is just displayed on the monitor for the human driver as a reference (the human brain can make sense of the image even if it is disturbed occasionally), a clear picture is crucial for the image processing done in automated driving mode. Since the place where the system is operated (the FLASH-Lab) seems to have a lot of disturbance in the high frequency band, an option was included in the system to bypass the transmitter and directly send the video signal through a wire to the remote control station. Of course this option requires that the vehicle drags a long wire behind while driving, but it does ensure a clear picture for the image processing.

Table 4.3: Technical data for the amplifier

Name	MODEL 3310PAK
Manufacturer	SuperCircuits (Supplier)
Active device	Hitachi PF0011
Operating frequency	902 - 928 MHz
Power in	10mW (12VDC)
Power out	9W linear, 10W saturated
Supply voltage	12 - 14 VDC (max 2A)

4.2 New Hardware at Remote Control Station

It has been mentioned in Section 3.1.2.2 that most of the high level processing in telerobotic manual mode is done not on the vehicle's microcontroller itself, but on a stationary control computer. This is even more true in automatic driving mode. In automatic mode, the system relies on image processing to provide the driving commands. This image processing requires special hardware that has to be packed on the vehicle and connected to the microcontroller. Both the size of the image processing hardware board and an incompatible interface between board and microcontroller (standard AT-Bus on one side and the HC11

ports on the other) make this difficult to realize. Since the video image is sent to the remote control station anyway as a reference for the human driver, it is much more convenient to do the image processing with the more powerful control computer. This computer has both the necessary interface and the space to house the image processing hardware.

4.2.1 Video Link Receiver

The receiver at the remote control station is a small device that can be connected to any TV or VCR. For optimal quality, the receiver can be manually tuned to the right frequency band. The output of this receiver then goes to the frame grabber board described in Section 4.2.2. In case of a wire video link, the cable from the vehicle is directly connected to the frame grabber.

4.2.2 Frame Grabber with DSP

At the remote control station, the video signal is received and routed to a frame grabber board residing in one of the AT-Bus slots of the control computer. This board is a FF1 DSP Frame grabber from Current Technologies that comes with an on-board digital signal processor (DSP) for image processing (see the manual[17] for further information).

Table 4.4: Technical data for the DSP Frame Grabber

Name	FF1 DSP Frame Grabber
Manufacturer	Current Technologies
Input	A/D converter digitizing 8bit at 10MHz accepts RS-170 and CCIR signals
DSP	Analog Devices ADSP 210x Operating frequency 10 MHz Programmable in C or 210x assembler 512 16-bit word of data memory 1024 24-bit words of code memory
Memory	512*512 16 bit words of image memory (accessible with zero wait states)
Output	D/A converter for RS-170 or CCIR analog by-pass for direct camera-monitor connection
API	FF1 software in C library (for Microsoft C or Borland C)

The manufacturer includes a C-library for the DSP to perform various functions from the simple capturing of images over all kinds of image manipulation up to such high-level functions as object-detection in the captured image. Most of the code in these functions runs on the on-board DSP, therefore not using the CPU resources of the control computer. In fact it is possible to have the CPU and the DSP run in parallel. The C-library provides the necessary functions to synchronize the two processors and to allow data exchange [17].

The functions used for the image processing in the automatic mode are the capture of an image followed possibly by either the calculation of an intensity profile or a series of edge-detections to determine the location of the middle marker of the road in the captured image. Because Chapter 5 contains a detailed discussion about the algorithms used for image processing, only a brief description of how to do the edge-detection is given here.

The edge-detection algorithm makes use of a predefined template containing a number of steep gradients in the pixel intensity corresponding to sharp changes from dark to white pixels. Figure 4.1 shows an example for such a template and also how it is created using functions of the FF1 Frame Grabber (see [17]). Once a template is created, it can be saved and later reloaded. The edge-detection algorithm then tries to match this template with pixel gradients found in the captured image. The matching is done by calculating a matrix of the gradients in pixel intensity at each pixel in the captured image and then trying to find the template (also a matrix) in this gradient matrix (see [17] for more details). The result of this matching is a certain integer quality value. If this value is greater than a certain threshold, the algorithm will return the location (the x- and y-coordinates) where the template could be successfully matched. This whole algorithm is embedded in a function of the FF1 library (function *ff_search_for_template*). This function is called with the template to look for and a search area as parameters and returns the location of the best match and the corresponding quality value.

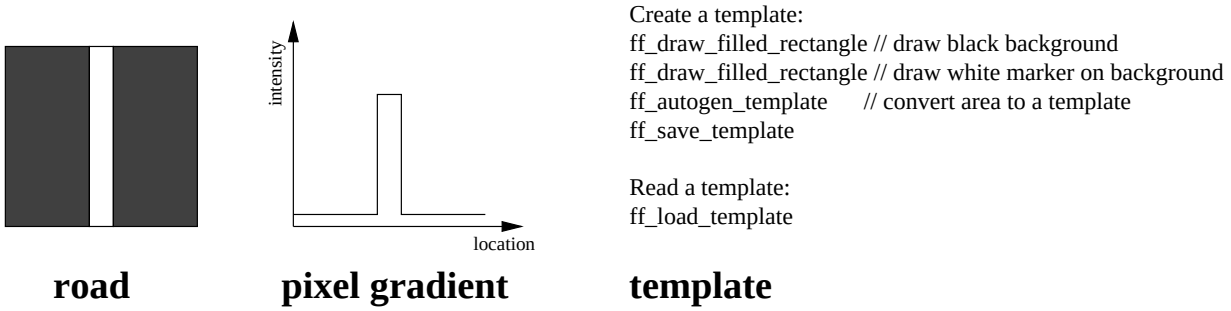

Figure 4.1: An edge-detection template and its representation

The execution time of this edge-detection function varies with the size of the area where the search is performed. In order to have fast execution time, the image is captured using only a 128x128 pixel matrix instead of the full resolution of 512x512. In addition to that, the edge-detections are performed only in a specific area of the captured image (refer to Section 5.2.2). Note that this method of edge-detection is not the only possible one. However, since it was already implemented in the function library of the FF1 Frame Grabber, it was just used without modifications.

4.3 Software for Automated Vehicle Operation

The only modifications in software have to be done in the program running on the control computer. It is necessary to include a task to do the image processing and controller simulation and another to switch between manual and automatic mode. The program running on the microcontroller of the vehicle can be used without any modifications, since the same functionality is required in both modes.

4.3.1 The FlipAutoMan Task

The *FlipAutoMan* task is activated whenever the control program should change the operation mode from telerobotic operation to automatic driving mode or vice versa. This request to change the mode can be issued either through a command from the host computer or by pressing the top button on the steering console. The *FlipAutoMan* task then suspends the *AutoIn* (see Section 4.3.2) or *ManualIn* task (see Section 3.2.2) and resumes the corresponding other task, depending on the previous mode of the control program.

4.3.2 The AutoIn Task

The job of the *AutoIn* task in automated driving mode is to do the following two subtasks:

- perform the necessary image processing to determine the position of the vehicle with respect to the road.
- feed this position information to a software implementation of a lateral controller with the control objective being to keep the vehicle on a fixed position relative to the road (e.g. in the center)

As Figure 4.2 illustrates, the DSP and the CPU of the control computer work in parallel in the *AutoIn* task [2] to achieve these two subtasks. The whole process starts with the DSP capturing an image from stream of video input from the camera. After an image is captured, the DSP does whatever image processing is necessary to identify features of the road that give a clue about the vehicle's relative position (see Chapter 5 for the details of the algorithms used). This can include calling the edge-detection function discussed in Section 4.2.2 or other functions of the FF1 library. Once this is done, the results are transferred to the *AutoIn* task in the control program. The CPU then does additional calculations that result in one or more error values that are a measure of how far the vehicle is from the desired position. These errors then go into a software implementation of a lateral control algorithm (see Chapter 6.4.6 for possible controller types). The output of such a controller is a steering angle for the front wheels of the vehicle that will bring it closer to the desired position. This steering angle is converted to a message that is then sent to the *Steer* task (see Section 3.2.2). From there on, the procedure is the same as in telerobotic manual mode: the message gets converted to a STEER command and is transmitted to the vehicle where it is executed. Of course, while the CPU is doing all these calculations, the DSP is free to capture a new image and also perform the other image processing functions. Ideally, no processor has to wait for the other to do its job. In practice however the DSP is normally slower than the control computer CPU. But having the processors run in parallel still means some gain in performance of the overall system.

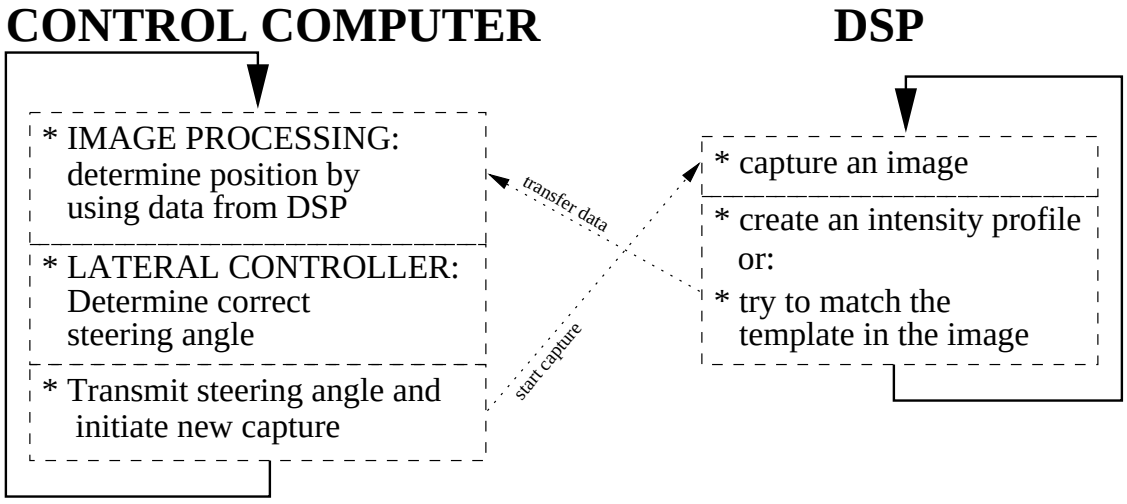

Figure 4.2: DSP and CPU working in parallel

[2] Of course the DSP continues to work even when the CPU is not executing the *AutoIn* task. This task is just the only place where the two processors communicate with each other.

4.3.3 The Data Log Option

Beside the two modes for telerobotic manual operation and automated driving, a third mode of operation is implemented in the control program. In this mode called "data-log" mode, important data like driving commands and position information is stored ("logged") for later use. In this mode, the driving commands for the vehicle come from a human driver but in addition the image processing algorithm is used to determine the position of the vehicle on the road. The driving commands (from the human driver) and the position information (from the image processing algorithm) are stored in two files. A variation of this form of data-log mode is that the driving commands are read from a file (instead of a human driver moving the controls at the steering console) while the position of the vehicle is stored in another file.

Although the data-log mode is described as a separate mode, the code for it is embedded in the *ManualIn*, *AutoIn* and *Steer* task. In normal telerobotic operation, the *ManualIn* task just polls the position of the steering wheel and pedal and sends their positions as messages to the *Steer* task. In data-log mode, these messages are also written in a file. Also, the *AutoIn* task is executed, but just the part of the task that contains the image processing. Once the position of the vehicle is determined, this information is written in another file and the controller part of the *AutoIn* task is omitted. When the drive commands do not come from the human operator but from a file, the *Steer* task reads messages from this file instead of receiving them from the *ManualIn* task. Figure 3.7 in Section 3.2.2 shows how theses tasks interact with each other in data-log mode.

There were two major reasons for implementing a separate data-log mode in addition the two normal operating modes:

- Records of the driving commands from a human driver together with the vehicle's response to these commands allows "human factor" studies. An example could be measuring the reaction of human drivers to a suddenly appearing danger on the road. Doing these kinds of studies on a small scale vehicle system ensures safety for the driver but also provides a good amount of realism. The level of realism in a small scale system is higher than in a full computer simulation, since a wrong reaction might lead to real damage to the vehicle. But due to the inexpensive layout of the system, even in a worst-case scenario (vehicle damaged beyond repair) all that has to be done is to replace the damaged hardware, which is possible at a very low cost.
- The fact that it is possible to record the vehicle's behavior in response to certain pre-recorded driving commands can be used to gain more knowledge about the dynamic system involving vehicle and road. In particular, the recorded data can be used to

estimate some unknown parameters when trying to find a mathematical model for the dynamic system. In this case, the estimated parameters are plugged in the mathematical model. Then the same input (driving commands) is applied to both the model and the actual system. By comparing the different outputs, the parameters that generate the "best match" (the most similarity in the two outputs) can be found by iterating the parameters. This parameter estimation technique is actually used for obtaining a mathematical representation of the dynamic vehicle-road interaction found in the small scale system. Details about this parameter estimation are given in Chapter 6.4.6.

Figure 4.3 shows an example of recorded human drive commands and the response of the vehicle.

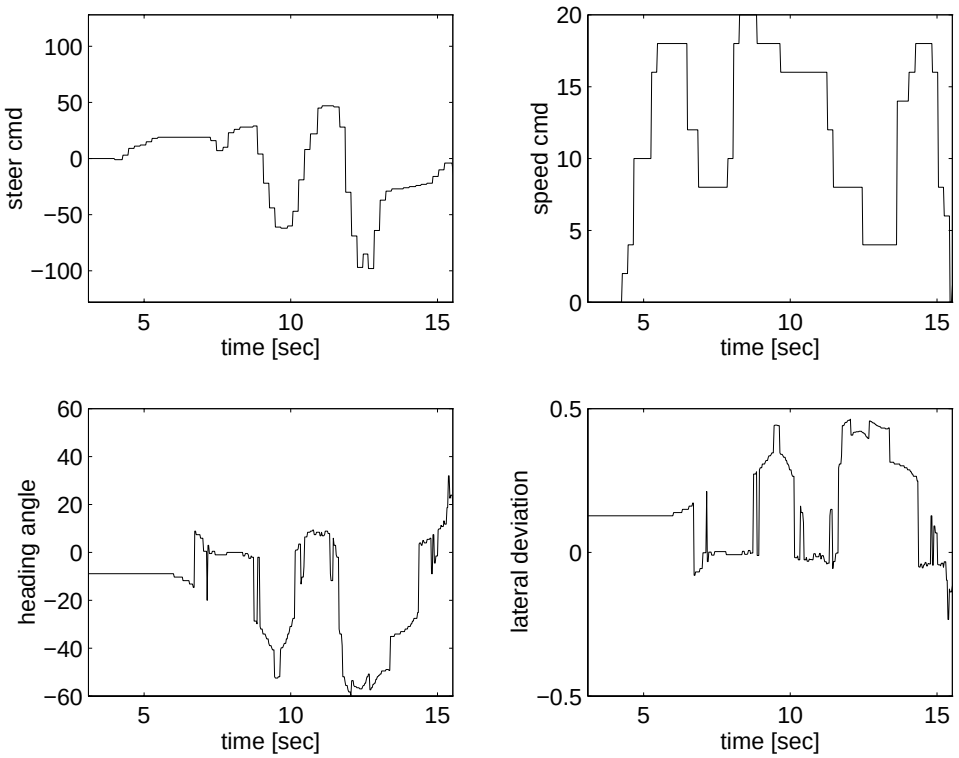

Figure 4.3: Example of data log

Chapter 5

Methods for Image Processing

The goal for the image processing has already been defined in previous sections: extract information about the position of the vehicle from an image of the road. This information is then used as feedback to generate steering commands that will keep the vehicle on the correct path on the road. Two different approaches for the image processing part have been investigated and will be explained in detail in this chapter:

- A straight forward approach based on the intensity of pixels in the captured image. This approach involves calculating an intensity profile of the image (see Section 5.1 for details).
- An approach that takes into account the correspondence between the two-dimensional image plane and the three-dimensional real world. The edge detection functions described in Section 4.2.2 are used in this approach (see Section 5.2.2).

The reason for having two different approaches for the image processing is that the approach described in Section 5.1 was implemented first due to its simplicity. However it was later found to be insufficient and lacking the necessary performance for automated driving and therefore the second approach was implemented.

In both approaches it is assumed that there exists a visual reference on the road to guide the vehicle. In this case, the reference has the form of a continuous middle marker. This marker has to appear brighter on the video image than the surrounding road to make it easily identifiable. It is further assumed that the road is flat (no hills), has only small curvatures and that there are no obstacles or other moving objects on the road. Note that image processing for automated driving is not restricted to a reference in the form of a

middle marker. Other possible references are markers on one or both sides of the road or the boundaries of the road itself. The only criterion is that the reference has to be visually identifiable.

5.1 Calculating the Intensity Profile

The original algorithm used to extract steering information from the image tackles this problem in a very simple way. All it does is sum up all pixels in a certain area in the lower part of the captured image in the vertical direction, therefore getting a one-dimensional horizontal pixel intensity profile of the image. Figure 5.1 shows an example of an image of the road and the resulting intensity profile.

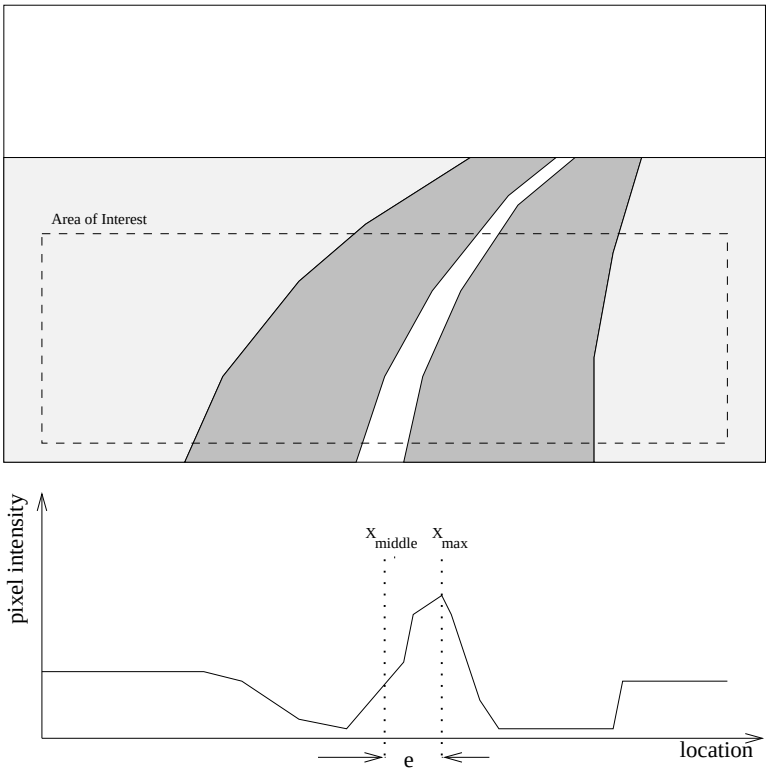

Figure 5.1: The pixel intensity profile of an image

Since the line in the middle of the road is very bright due to the white middle marker, it will result in a peak in the computed horizontal intensity profile. Now all that has to be done is find the peak in the intensity profile (with the x-coordinate x_{max}) and measure

its distance from the center of the image (denoted by x_{middle}). This distance can then be used as an error signal for a lateral controller that will produce the commands to move the steering angle of the front wheels. The goal of this controller has to be to reduce this error to zero (refer to Chapter 6.4.6 for the different controllers).

$$e = x_{max} - x_{middle} \qquad \text{objective of lateral controller:} \quad e \longrightarrow 0 \tag{5.1}$$

The pixel intensity profile can be calculated using the function *ff_profile_x* utilizing the DSP on the FF1 Frame Grabber. This function is executed directly on the DSP while the CPU of the control computer can do other jobs like calculating the response of the lateral controller (see Figure 4.2). Therefore, this algorithm used for detecting the middle marker has a good performance and is very fast, therefore potentially allowing high speeds of the vehicle.

Before the algorithm mentioned above was included in the control program, a small test program under MS-Windows was written that implemented the algorithm and displayed the the resulting intensity profile and the error (see Figure 5.2 for the dialog of this program). The purpose of this small program was first to test the capabilities of the FF1 Frame Grabber and second to act as an easy to understand demonstration of the intensity profile algorithm.

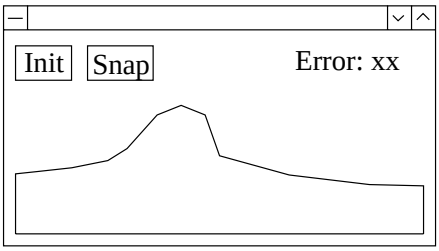

Figure 5.2: Dialog of test program for intensity profile algorithm

Although the above algorithm has actually been tested successfully with the vehicle, it has a number of disadvantages. First of all, is very sensitive to misinterpretations since it looks for peaks in the profile. If there is something next to the road with a higher intensity than the white line, the algorithm won't be able to distinguish the two. Another disadvantage is that the distance that is returned as an error is not very precise. It is actually a combination of the lateral deviation of the vehicle's center to the white middle marker and the angle that the vehicle has to the marker (see Section 5.2.1 for a discussion about this). With the above method there is no way of getting these two parameters separately. One consequence is that in a curve, an error is be detected although the vehicle is still in the center of the road and on a straight road, no error is detected even if the vehicle has a non-zero lateral deviation to the road center.

5.2 Vanishing Point Analysis

Because of the inherent weaknesses of the intensity profile algorithm, a more powerful algorithm to obtain position information was needed. A literature search revealed an algorithm based on the analysis of vanishing points (see [13]). This concept is explained in the next sections after an explanation of what is meant by "position information" is given.

5.2.1 Vehicle Motion Variables

In the previous section the position of the vehicle on the road was described only by an error variable that represents the distance between the white middle marker in the image plane and the center of the image. However, to accurately describe the position of the vehicle on the road, two motion variables are needed.[1] These variables are the lateral distance Δy of the vehicle's center of gravity to the middle of the road and the heading or yaw angle between the vehicle's orientation and the orientation of the road, denoted by φ_h. The lateral distance Δy is a measure of the distance of the vehicle in the direction orthogonal to the road or a measure of how far "off the track" the vehicle is. If the vehicle's center of gravity has a position on the road's middle marker, then the lateral distance is zero. The heading angle φ_h describes the direction in which the vehicle is pointed, relative to the direction of the road. This is a measure of whether or not the vehicle is heading in the "right" direction. If the orientation of the vehicle is parallel to the direction of the road, the heading angle is zero. See Figure 5.3 for a graphical explanation of these two variables.

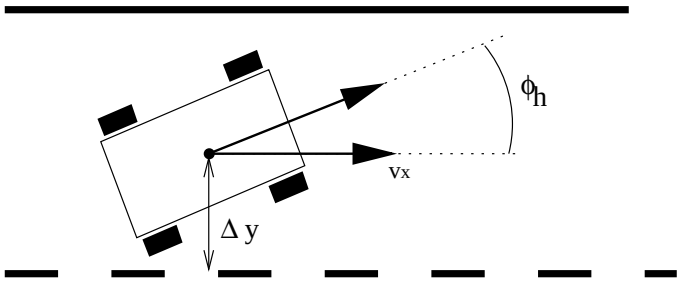

Figure 5.3: Variables for the position of the vehicle

[1] There is in fact a third important variable: the position x or equivalently the speed of the vehicle v_x in the x- or longitudinal direction. This is the direction parallel to a straight road or tangent to a curved road. However since the main objective is to come up with a controller for the *lateral* movement, this variable can be omitted

5.2.2 Vanishing Lines and Vanishing Points

The following concept about vanishing point analysis was taken from a paper dealing with the dynamic measurement of vehicle parameters [13]. The basic concept described in this paper is to obtain the orientation of the vehicle based on the location of some vanishing points in the image plane. A vanishing point is defined to be the virtual intersection of two straight parallel lines at a great distance under perspective projection.[2] The set of all vanishing points form the vanishing line (the horizon). Vanishing point analysis is based on the fact that there is a correspondence between the distance of two different vanishing points on the vanishing line and the angle between the set of parallel lines for the respective vanishing points. Figure 5.4 shows this correspondence. The vanishing point O is formed by all straight lines that are parallel to the principal axis of view (the direction the camera is facing). The vanishing point P is formed by another set of parallel lines. From Figure 5.4 it is intuitively clear that there exists a correspondence between the angle η between the two sets of parallel lines and the distance $\overline{OP}$ between O and P.

$$\eta \sim \overline{OP} \tag{5.2}$$

This correspondence can be rewritten as an equation by introducing the vanishing point P_2. This point on the vanishing line represents the edge of the area visible in the image plane. It is formed by lines that are parallel to the right border of the visible area. The angle θ is the angle between this set of lines and the principal camera axis. Two triangles are formed by the camera location C, the vanishing point O and the vanishing points P or P_2. By introducing θ and $\overline{OP_2}$ and using what is known about the triangles $C-O-P$ and $C-O-P_2$, the correspondence in Equation 5.2 can be rewritten as

$$\frac{\tan\eta}{\tan\theta} = \frac{\overline{OP}}{\overline{OC}} * \frac{\overline{OC}}{\overline{OP_2}} = \frac{\overline{OP}}{\overline{OP_2}} \tag{5.3}$$

[2] A typical real world example for a vanishing point are two tracks of a railroad that seem to converge at the horizon

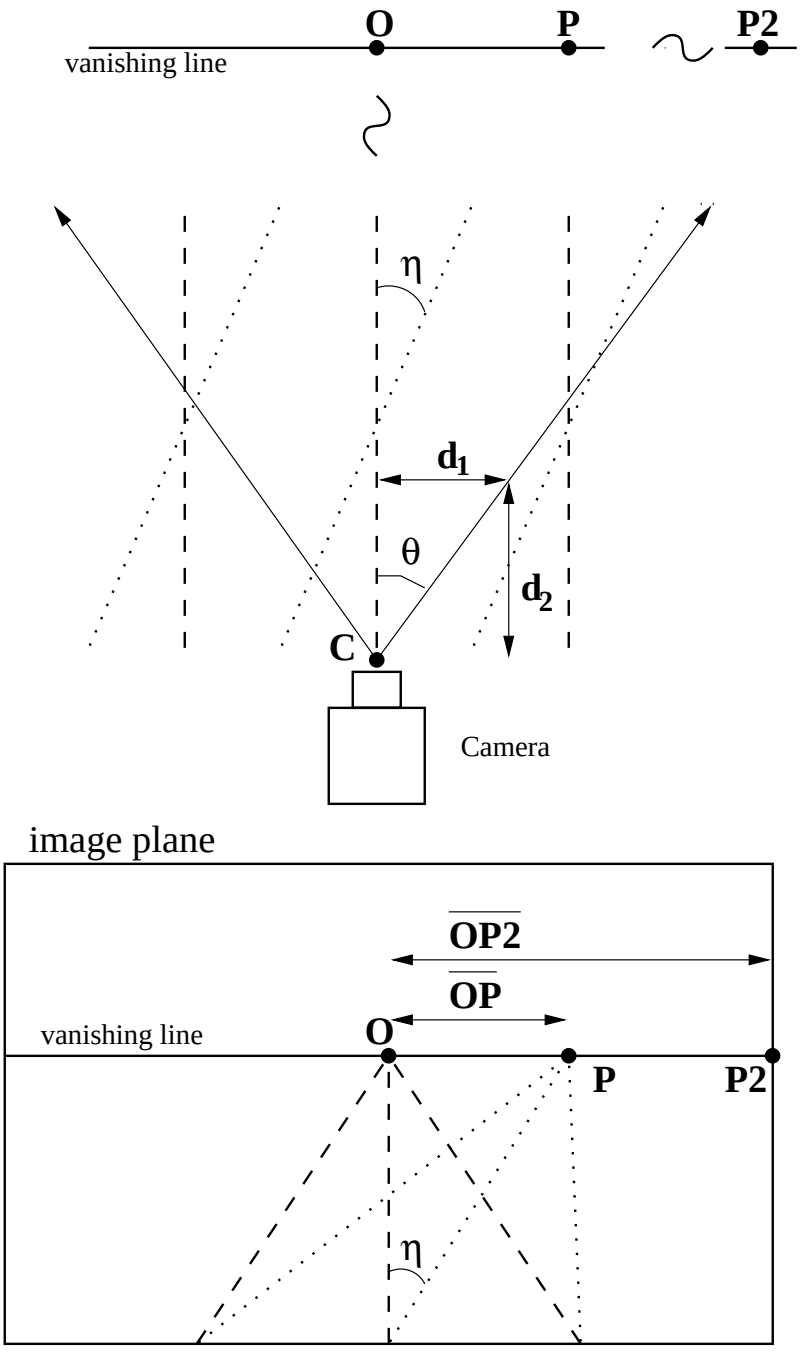

Figure 5.4: Parallel lines and vanishing points

Assuming that both the distance $\overline{OP_2}$ on the vanishing line and the angle θ are known or can be measured, the angle η can be expressed as a function of the distance $\overline{OP}$.

$$\eta = f(\overline{OP}) = \tan^{-1}\left(\frac{\overline{OP} * \tan\theta}{\overline{OP_2}}\right) \tag{5.4}$$

While it is very easy to measure the distance $\overline{OP_2}$ in the image plane (measured in pixels of the captured image) by simply counting pixels, measuring the angle θ is a little bit more difficult. It can be derived by measuring the distances d_1 and d_2 (see Figure 5.4). Note that the measurement is done in the real world (using a ruler etc.), not in the image plane! The angle θ can then be expressed as

$$\theta = \tan^{-1}\left(\frac{d_1}{d_2}\right) \tag{5.5}$$

5.2.2.1 Measuring the Heading Angle

Now that the theoretical concept of the vanishing point analysis has been laid out, the algorithm of obtaining the heading angle φ_h and the lateral deviation Δy can be explained. It is assumed here that the distance $\overline{OP_2}$ and the angle θ (refer to Section 5.2.2) are known. These two variables can be measured in a calibration step and will stay the same as long as the same camera is used. The algorithm for obtaining the heading angle consists of four steps:

Step 1: Use the *ff_snap* function of the FF1 Frame Grabber to capture an image from the camera. Note that even though the Frame Grabber is capable of snapping images with a resolution up to 512*512 pixel, in this case only a resolution of 128*128 pixel is used. The advantage of having this reduced resolution is that fewer pixels have to be processed in the next step.

Step 2: The function *ff_search_for_template* is used to find a template that matches a horizontal transition in pixel intensity from dark to bright and back to dark corresponding to a white line on a dark background. The function *ff_search_for_template* is called twice to look for this template in two different areas of the captured image. The first area is a narrow horizontal stripe at the lower edge of the image, the second area a similar stripe close to the middle of the image (see Figure 5.5). Calling *ff_search_for_template* twice returns two coordinate pairs (x_1, y_1) and (x_2, y_2) representing points in the captured image where the template could be matched best.

Step 3: The two points from the previous step are used to form a linear approximation of the location of the white middle marker in the captured image. By extending this line to the horizon (the vanishing line), the location of the vanishing point P can be determined (see Figure 5.4 and 5.5). Now a second line is introduced in the captured image. This line starts at the lower edge in the middle of the captured image and goes straight up to the vanishing line to form the vanishing point O. This line in the captured image corresponds to the principal axis of the camera in the three-dimensional real world.

Step 4: Since both the vanishing points P and O are now known, the distance $\overline{OP}$ (in pixels) between these points can be calculated. Assuming that $\overline{OP_2}$ and θ are known (these have to be measured only once in the calibration step), the correspondence in Equation 5.4 can now be used to calculate the angle η. In the three-dimensional real world, this η corresponds to the angle between the principal axis of the camera and the linear approximation of the white middle marker. Assuming that the middle marker is a straight line in the part of the road that is currently in view, the angle η is exactly the heading angle φ_h as it is defined in Figure 5.3. Even when the middle marker is not a straight line (because the vehicle is in a curve), the angle η can still be used as an approximation for φ_h.

5.2.2.2 Measuring the Lateral Deviation

In the previous section, the four steps to obtain the heading angle φ_h were outlined. In order to get the lateral deviation Δy, an additional parameter d has to measured beforehand. This d is the distance from the center of gravity of the vehicle to the line that marks the lower edge of the image plane (see Figure 5.5). After this parameter has been measured once, a fifth step has to be added to the algorithm above to obtain the lateral deviation Δy:

Step 5: Once the template is matched in the two areas of the captured image and a linear approximation of the white middle marker is found, this line is also extended to the lower edge of the image plane. The distance from the point where this line hits the lower edge of the image plane to the center of the image on the lower edge is denoted as a (see Figure 5.5). Using this a and the heading angle φ_h the lateral deviation can be calculated as

$$\Delta y = a \cos \varphi_h - d \sin \varphi_h \tag{5.6}$$

Note that both d and a have to be measured in real-world units (inches, centimeters etc.) and not in pixels. Therefore it is necessary to first convert a to one of these

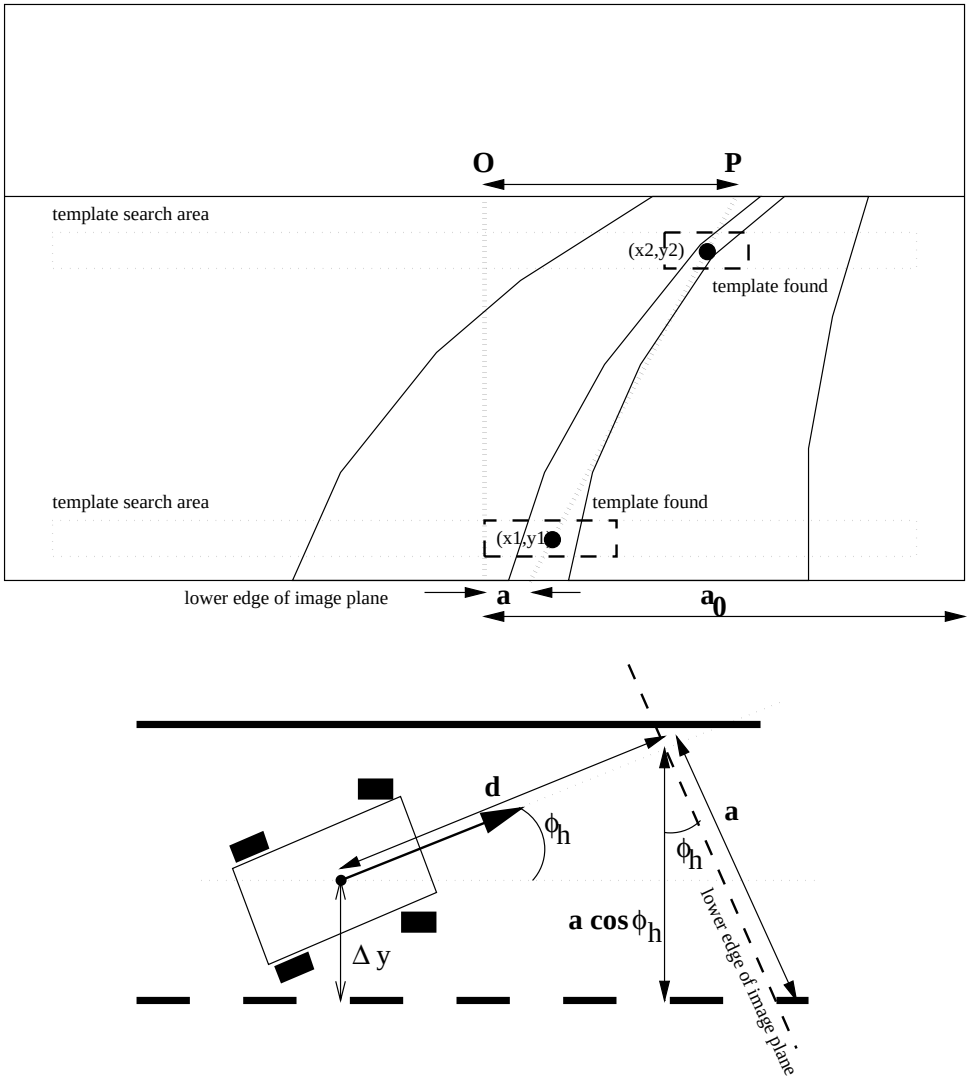

Figure 5.5: Linear approximation of white middle marker

units. But since d was measured before and the camera angle θ is known, the distance a_0 can be calculated in real-world units. As the distance a_0 is also known in pixels, the real-world a can be calculated as

$$a_{real} = \frac{a_{0_{real}}}{a_{0_{pixel}}} * a_{pixel} \tag{5.7}$$

As with the heading angle $\varphi_h.$, the lateral deviation Δy is only measured correctly when the middle marker is a straight line. But even when the road is curved, the Equation 5.6 can still be used as a good approximation for Δy.

The paper [13] explains how even more dynamic motion variables can be calculated using vanishing points. In particular, the yaw rate and side-slip angle are measured using the method above. Since these variables are not needed for this implementation of automated driving, they are not calculated here. Note again that a prerequisite for the algorithm presented here is that the road (almost) flat. If hills or descents in the road are too steep, the algorithm will return incorrect data.

The algorithm described here can be seen as the solution to a parameter-estimation problem. A mathematical model of the vehicle and the road is derived where two parameters are unknown. These parameters, heading angle and lateral deviation, are then estimated for each captured image using the described image processing. Note that there is only one iteration per captured image (memoryless identification). The estimated parameters after the first iteration are then used as inputs for the controller and a new image is captured to determine the changes in the vehicle's position.

5.2.3 Demonstration of Vanishing Point Analysis

As with the intensity profile algorithm explained earlier, the algorithm using vanishing point analysis was first tested before being implemented in the control program. In this case, two test programs were written. The first test program called *edge* captures an image every 2 seconds and then searches for the template in the two above mentioned areas. When the template could be matched twice, the location of the match is highlighted and displayed on the video monitor (see Figure 5.6 for an example).

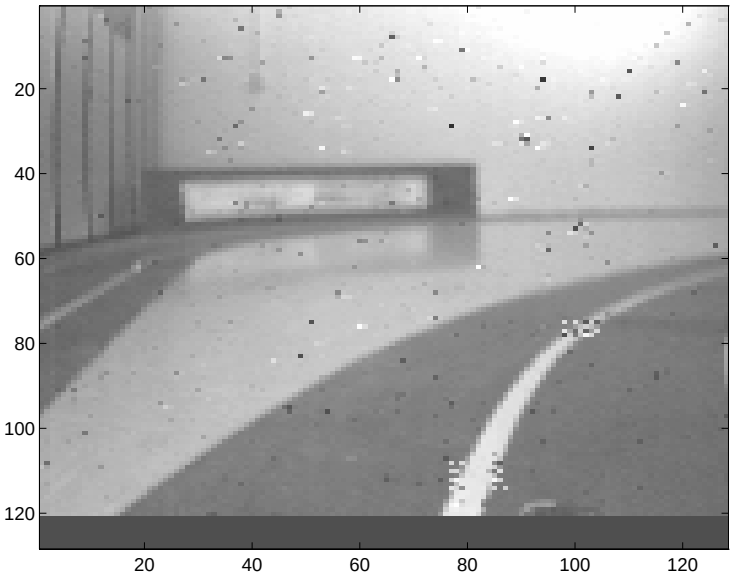

Figure 5.6: Template matching in the captured image

The second test program *edge2* captures the images in real-time (once every $50ms$), matches the template and does all the calculations outlined above to obtain heading angle and lateral deviation. The value for these two motion variables are then displayed on the monitor of the control computer, together with a line that represents the linear approximation of the middle marker in the video image. The output also includes the time necessary to do the image processing (see Figure 5.7).

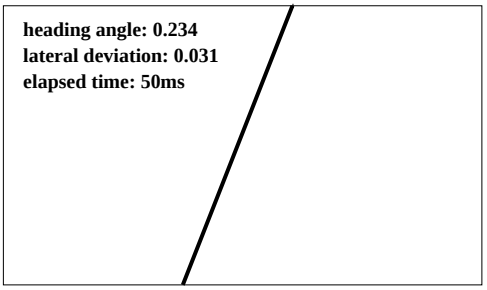

Figure 5.7: Output of second demo program

Chapter 6

Designing the Controllers

In this chapter, the different methods used for the vehicle control are presented. First an algorithm dealing with the vehicle's longitudinal movement (movement parallel to the road) is introduced. For an algorithm controlling the lateral movement (movement perpendicular to the road) there are numerous possibilities to choose from. These include controllers based on classical control theory (PI, PID), modern LQG type control, neural networks and fuzzy logic (see [4, 9, 10] for some examples). The upcoming of H_∞ control opened the possibility to also apply this theory to the design. H_∞ control design specifically has some advantages for a control problem like this due to it's inherent robust stabilization effect [11] [21]. Two types of lateral controllers were implemented for the vehicle. This first one introduced in Section 6.2 is a simple classical P-control algorithm were the gain is picked by trial and error. The second type of lateral control algorithm is more sophisticated and makes uses of the H_∞ control theory (Section 6.4). Since this controller requires a mathematical model of the vehicle dynamics, Section 6.3 outlines how to derive such a model. The chapter ends with a comparison between the two types of control algorithms.

6.1 A Speed Feedback Control

The objective of a longitudinal or speed feedback controller is to control the speed of the vehicle. While it is possible to have very sophisticated longitudinal controllers, e.g. to match the vehicle's own speed with that of another vehicle, the only requirement on a longitudinal controller in this system is that it keeps the speed at a desired value, even if the vehicle enters a curve or goes up a hill. Note that the algorithms for lateral control presented in the following sections also require a constant speed of the vehicle.

The hardware requirements for a speed feedback control were already outlined in Section 2.3.1. Required is a way of measuring the actual turning rate of the wheels and therefore the actual speed of the vehicle. On the model vehicle, the speed is measured by counting the number of pulses received from an encoder in a certain amount of time. This measurement is done using the microcontroller on the vehicle (see Section 3.2.1 for the pseudo-code of this part of the microcontroller program). The number of pulses is scaled to a value from $0 \ldots 255$ and represents the *actual* speed of the vehicle. It serves as one input for the control algorithm. The other input for the control algorithm is a value sent as a parameter of a SPEED command (also in the range $0 \ldots 255$, see Table 3.3). The SPEED command is received from the control computer and its parameter represents the *desired* speed for the vehicle.

Since measuring the speed of the vehicle requires a certain amount of time (the interval between measurement is in this case 25 ms long, see Section 3.2.1), the measured actual speed v_{actual} is not available as a continuous function $v_{actual}\ (t)$ but rather as a sequence of values $v_{actual}\ (0)$, $v_{actual}\ (1)$, $\ldots v_{actual}\ (n)$. Therefore the control algorithm has to be executed only whenever a new value for v_{actual} is available, in this case every 25 ms. Such an algorithm is called a *sampled data controller* or a *discrete controller* because the output is not a continuous function but rather a series of discrete values that only change at certain steps in time.

The control algorithm introduced here is iterative. That means that the output of the control algorithm at step n is based both on the inputs at step n and the results from step $n-1$. The algorithm starts by calculating the difference between desired and actual speed at step n (the speed error $e(n)$:

$$e\ (n) = v_{desired}\ (n) - v_{actual}\ (n) \tag{6.1}$$

This speed error is used to calculate two more variables called $u_p(n)$ and $u_i(n)$. Note that $u_i(n)$ depends not only on the speed error but also on the previous value $u_i(n-1)$. This make the algorithm iterative.

$$\begin{aligned} u_p\ (n) &= k_p * e\ (n) \\ u_i\ (n) &= u_i\ (n-1) + k_i * e\ (n) \end{aligned} \tag{6.2}$$

The actual output of the control algorithm is formed by

$$u\ (n) = u_p\ (n) + u_i\ (n) \tag{6.3}$$

This variable $u\ (n)$ is the new PWM cycle for the drive motor necessary to have the vehicle move at the desired speed.

Note that the control algorithm in Equation 6.1 - 6.3 is a discrete digital equivalent to an continuous analog PI controller (see [23]). The behavior of the closed loop formed by the control algorithm and the controlled plant (the vehicle) can be altered by modifying k_p and k_i. The factor k_p is equivalent to the gain in a PI controller. It determines how fast the closed loop system responds to changes in the desired value. In this case, k_p determines for example how fast the vehicle will accelerate when the gas pedal is suddenly pushed down (a step input). The factor k_i is similar to the integration time in an continuous PI controller. The integration time is a measure of how long it takes for the control error (here, the speed error) to converge to zero (no difference between desired and actual speed). A PI controller (and its discrete equivalent) always tries to reach a control error of zero. This feature is also called a *zero steady-state error*. Other types of controllers like the P-controller might have a nonzero steady state error.

The actual values for k_p and k_i were determined by trial and error, due to a lack of a mathematical model for the longitudinal dynamics of the vehicle. Note that it is not possible to have both a quick response to input changes and a fast convergence to zero in the control error since these objectives conflict with each other. The goal of the trial and error process was to find values of k_p and k_i that achieve a zero steady state error in a short time but still have a good response for sudden changes in the input. Figure 6.1 shows the performance of the closed-loop system with the values $k_i = 1$ and $k_p = 4$. Only integer values can be chosen for the control algorithm since the GROM library for the GCB11 does not contain floating-point support.

6.2 Using P-Control for Lateral Control

While the objective of a longitudinal controller is to control the speed of the vehicle, a lateral controller is responsible for the steering of a vehicle operating in automated driving mode. The goal for such a controller has already been defined in previous chapters: to steer the vehicle in such a way that it follows a road or more precisely a white middle marker in the center of a road. All controllers require some kind of control error that is a measurement of how well the control objective (following the road) is met. As outlined in Chapter 5, such a control error can be obtained from a video image of the road ahead of the vehicle using one of the two introduced image processing algorithms. In case of the intensity profile algorithm, this error value is the distance $x_{max} - x_{middle}$ between the peak of the intensity profile and the middle of the image (see Section 5.1). In the case of the vanishing point analysis (see Section 5.2.2) there are two control errors, the heading angle φ_h and the lateral deviation Δy.

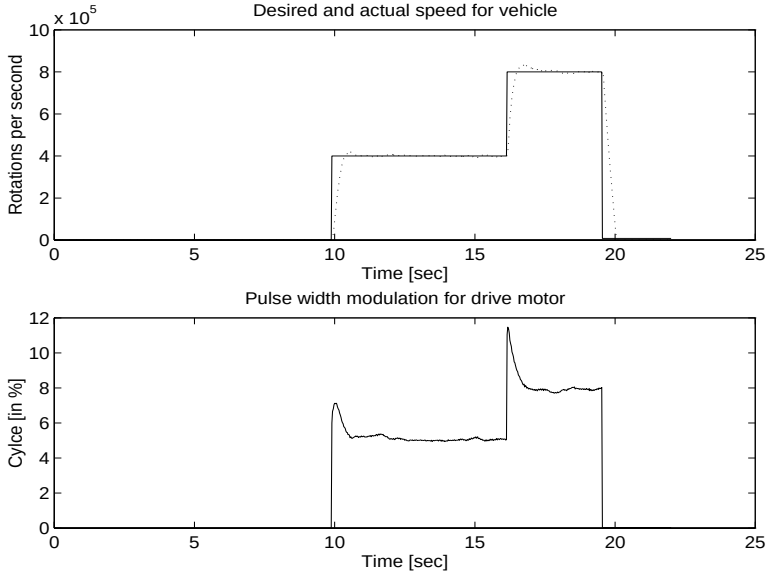

Figure 6.1: Performance of the longitudinal control algorithm

The whole idea of a P-controller is to multiply the control error value e by a constant gain k_p. As with the measuring of the actual speed in Section 6.1, it takes time to measure the control error. This is the time for the image processing algorithm to execute ($50ms$ for the intensity profile algorithm and $75ms$ for the vanishing point algorithm). Therefore the control error is changing only at certain steps in time and the output of the P-control algorithm is discrete rather than continuous. The output $u(n)$ of the control algorithm is a steering angle that can be converted to a STEER command and sent to the vehicle.

$$u\ (n) = -k_p * e\ (n) \tag{6.4}$$

Note that the gain k_p has to be multiplied by -1 in order for the control error to decrease. Since the P-controller is like all classical controllers a SISO system (a single input and a single output), the two control errors of the vanishing point analysis have to be combined to one value. This can be done by making $e(n)$ a linear combination of φ_h and Δy.

$$e\ (n) = k_1 * \varphi_h\ (n) + k_2 * \Delta y\ (n) \tag{6.5}$$

The optimal value for the constant gain k_p can either be derived from a mathematical analysis of the closed loop consisting of control algorithm and controlled plant (e.g. root-locus) or by trial and error. In this case the later approach was chosen, since initially, a mathematical model of the lateral vehicle dynamics (see Section 6.3) was not available. Therefore, a random value for k_p was picked and it was changed accordingly to increase the

performance of the closed loop. The performance criteria for the closed loop is of course how well the vehicle with lateral controller is able to follow a (curved) road. The values for the weighting factors k_1 and k_2 when using the vanishing point algorithm can be found in a similar fashion.

Of course, combining the two independent variables φ_h and Δy to one control error e means a loss of information. Therefore the combination of the P-control algorithm with the vanishing point analysis is not a good idea because part of the advantage of this image processing algorithm (having two independent variables) is given up here.

When testing the P-control algorithm in automatic driving mode, its performance is surprisingly good, considering it simplicity. When the model vehicle is operating in automatic driving mode with the lateral P-control algorithm it is capable of following the model highway laid out in the FLASH Lab without any problems. Figure 6.2 shows a data-log output of how the vehicle is following part of a race-track-shaped road in automatic driving mode. Note how the vehicle reacts (steer command) whenever a curve is detected (change in heading angle and lateral deviation). In this case the vanishing point algorithm was chosen for doing the image processing

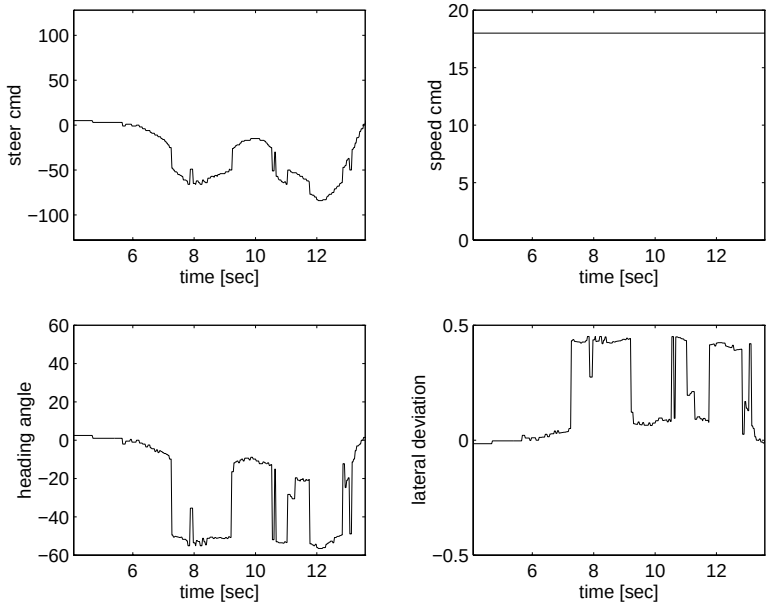

Figure 6.2: Performance of the lateral P-control algorithm

The only restriction for the algorithm is a speed limitation. The P-control will only operate for speeds of up to about $1\frac{m}{sec}$ which is approximately walking speed. [1] A reason for this limitation is the time it takes to capture and process a video image: if the vehicle moves too fast, it might have already driven off the road in this sampling time. The control algorithm works with both of the image processing algorithms. Choosing the vanishing point algorithm instead of the intensity profile algorithm has the additional advantage of making the system less sensitive to bright spots in the image (this justifies the use of this algorithm in spite of the disadvantage mentioned above). Such spots usually occur due to reflections of the sun of ceiling lights on the floor.

6.3 Model of the Lateral Vehicle Dynamics

So far, parameters of the control algorithms like the gain k_p or the integration constant k_i have always been determined by a trial and error method on the actual plant (the vehicle). This is a valid approach for a system like the one in use, because in case wrong parameters are chosen, the worst thing that can happen is a damage to inexpensive hardware. But for a full-scale vehicle, such an approach is unacceptable. Instead, a way has to be found to simulate the behavior of the vehicle on the road on a computer using a mathematical language like MATLAB. This involves creating a mathematical model of the vehicle dynamics (delays, integrators etc.) Once such a model is established, the control algorithms can be tested on the simulation first before implementing them in the code for automated driving. A mathematical model of the plant is also a great help for finding the values for parameters like k_p since they can be derived by analyzing the closed loop consisting of control algorithm and controlled plant. [2] For the analysis, methods like root-locus and gain-/phase-margins in Nyquist or Bode plots can be used [22].

6.3.1 Linearized model of vehicle dynamics

In order to design a controller or to find values for control parameters, the response of the plant to certain inputs has to be known. In this case, with the steering angle being the controller output and the plant input, the response of the plant is a change in heading angle φ_h and lateral deviation Δy. One way to model these dynamics is the classical single-track

[1] Converted to a full-scale vehicle, this would be a speed of about 35 *mph*

[2] Note that a computer simulation is always discrete even though the simulated plant is continuous in nature. This has to be taken into account when creating a software implementation

model. [3] This model has been used in various lateral controller designs (see for example [10, 12])

This model is actually a linearization of some of the nonlinear effects like adhesion, wheel slip etc. (see [12] for some details on the nonlinearities involved). Note that it is not necessary for a simulation of a plant to be linear. In fact, since real world plants are almost always non-linear in nature, a non-linear simulation might even be closer to the original plant. However, the process of finding values for parameters or designing a controller based on the simulated representation is greatly simplified (and sometimes only made possible) by linearizing the mathematical model of the plant.

The single-track model is a four state, one input (steering angle δ_f), two output (heading angle φ_h and lateral deviation Δy) system normally represented in state-space form. Figure 6.3 shows the state and output variables of the model. The model is written as

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \\ x &= \begin{bmatrix} \beta & r & \varphi_h & \Delta y \end{bmatrix}' \quad u = [\delta_f] \end{aligned} \tag{6.6}$$

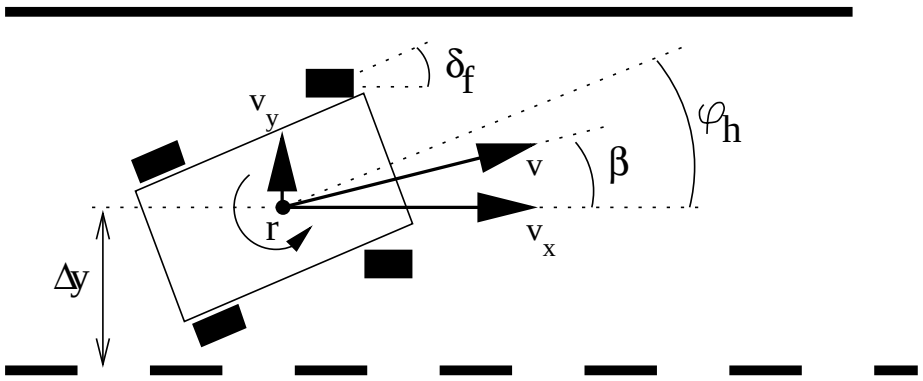

Figure 6.3: Variables used for lateral control

Apart from φ_h and Δy, which are both states and outputs, the systems also has the side-slip angle β and the yaw rate r as states. The side-slip angle is proportional to the speed of the vehicle perpendicular to the direction of travel on the road (denoted as v_y in Figure 6.3) and is the angle between the speed of the vehicle v and it's component v_x parallel to the road. Note that it is not identical to the heading angle φ_h. The yaw rate is a measure

[3] It is called single-track because the two front and the two rear wheels are lumped together to one wheel on the center line of the vehicle

for the turning rate of the vehicle around the center of gravity and is the derivative of the heading angle φ_h. [4] The four matrices of the state-space system for the single-track model have the following form:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ v & 0 & v & 0 \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ 0 \\ 0 \end{bmatrix} \quad C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{6.7}$$

with

$$\begin{aligned} a_{11} &= -\frac{C_r + C_f}{mv} \\ a_{12} &= -1 + \frac{C_r l_r - C_f l_f}{mv^2} \\ a_{21} &= \frac{C_r l_r - C_f l_f}{J} \\ a_{22} &= \frac{-(C_r l_r^2 - C_f l_f^2)}{Jv} \\ b_1 &= \frac{C_f}{mv} \\ b_2 &= \frac{C_f l_f}{J} \end{aligned} \tag{6.8}$$

Note that due to the integrative nature of the plant, it has a double pole at zero (a constant step input for the steering angle δ_f results in a linear change of φ_h and a parabolic change of Δy). The parameter C_f and C_r are called *cornering stiffnesses*. All the parameters in the equations above will be discussed in more detail in Section 6.3.2.

One important modification has been made to the single-track model in this work. It has been explained in Section 3.2.1 that the desired steering angle is sent as a command to the vehicle's microcontroller where it is converted to a signal to move a steering servo. Since this servo is a mechanical component, it cannot react instantly to a changing input. Also the transmission of the command takes a small amount of time. Therefore the dynamics of the servo and the transmission delay also have to be modeled and added to the system. In this case, they are modeled as a delay using a first order PADE approximation (a PDT_1 system, see [26]). The overall system therefore has one input, two outputs and five states, the additional state being the delayed steering angle.

[4] The yaw rate is NOT a derivative of β as it might seem at first. An example might illustrate this: A vehicle might drive over an icy road spinning around itself. In this case, the side-slip angle is zero (vehicle moving parallel to the road) but the yaw rate is nonzero (spinning around the c.g.)

Once the numerical values for all the matrix elements have been found (see Section 6.3.2), the A, B and C matrix can be used to simulate the dynamics of the vehicle using MATLAB (or a similar tool). It has been mentioned before that a software simulation of a plant is always discrete in nature. Therefore the matrices of the system have to be converted from the continuous state space to the discrete state space. This operation is analogous to doing the z-transformation instead of the s-transformation in the frequency domain (see [29] for an explanation of s- and z-transforms). When simulating the plant with MATLAB, this matrix conversion can either be done internally by using the *lsim* command or by hand with the *c2d* command (see the reference manual [26]). The *c2d* command accepts the continuous state space matrices A and B and a time interval Δt as parameters and returns the discrete state space matrices A_d and B_d (the C matrix stays the same in this conversion). Using A_d and B_d the state and output variables at step n can be calculated as

$$\begin{aligned} x\ (n+1, 1:5) &= A_d\ x\ (n, 1:5) + B_d u\ (n, 1) \\ y\ (n, 1:2) &= C\ x\ (n, 1:5) \end{aligned} \tag{6.9}$$

The vector x (n,1:5) is a five element vector containing the five states at step n, the input $u\ (n, 1)$ is the steering angle δ_f and $y\ (n, 1:2)$ contains the value of φ_h and Δy at step n. Note that the values for the states at step $n+1$ depend both on the input and the states at step n. The time interval Δt between the steps can be chosen arbitrarily small depending on how precisely the continuous plant should be simulated and how fast the control algorithm is. However, care has to be taken, that Δt is significantly smaller than the dynamics of the simulated system or otherwise certain dynamic behavior wouldn't be simulated. [5] In this case Δt was chosen to be 25 ms. The dynamics (the reactions of the vehicle to changing input) lie in the order of several tenths of a second to several seconds, therefore no problem arises.

6.3.2 Parameters of the Model

6.3.2.1 Known Parameters

To fill the elements of the matrices of the mathematical model used above, some parameters of the vehicle like speed, mass etc. have to be known. Most of the required parameters for the model can either be measured, calculated or reasonably approximated (see Table 6.1).

[5] In communications, this requirement is known as Shannon's Theorem

Table 6.1: Known parameters of vehicle

parameter	description	value
$m[kg]$	vehicle mass	1.5
$l_f[m]$	distance c.g. to front axle	0.15
$l_r[m]$	distance c.g. to rear axle	0.14
$b[m]$	width of vehicle	0.21
$J[kg\ m^2]$	momentum of inertia [a]	0.09
$v[m/sec]$	speed of vehicle	0.2
$T_s[sec]$	servo delay	0.6

[a] Approximated by $J = (l_f + l_r) * b * m$

6.3.2.2 Obtaining the Unknown Parameters

However, some parameters, namely C_f and C_r which denote the cornering stiffness of the front and rear wheels cannot be measured easily and are therefore unknown. These two parameters are a measure of the interaction of the wheels and the road surface. They are in fact highly non-linear functions of several variables, but are normally approximated as constants for modeling purpose [12].

While there exist tables containing these parameters for different makes of full-scale cars on different types of roads [12, 11], there is absolutely no data available for a small scale model vehicles like the one used in this case. The problem therefore is that the structure of the plant is known (from the single-track model), but the values for some of the elements in the matrices are unknown, since C_f and C_r are not known. It is in fact possible to get some information about the relation of the elements to each other since the other parameters are known (mass, dimensions, speed etc.).

The goal therefore is to somehow get the values for these unknown parameters. There are two different approaches to deal with this problem:

1. A control algorithm could be designed just on the known structure of the plant. The parameters for the control algorithm (gain, etc.) are modified on-line while the algorithm is controlling the plant. The parameters have to be changed in a fashion to achieve the best control performance (see below how this can be done). Such a control algorithm is also called an *adaptive controller*. This approach however addresses only the controller design itself and is not of any use for obtaining a complete software simulation of the plant. On the other hand, it has an advantage if the unknown parameters change over time, since the control algorithm can adapt to these changes.

2. An initial estimate for the unknown C_f and C_r is chosen. The complete mathematical model is then created using these estimates. Next, an input signal (in this case several steering commands) is applied to the mathematical simulation of the plant and the actual plant itself. The output of both systems is recorded and compared. Then, the initial estimate for the parameters is modified so that the output of the simulated plant better matches the output of the actual plant. Once the two outputs of the systems are the same, the values for the unknown parameters are found.

For this work, the second approach was chosen. Even though the cornering stiffness does change over time (when the vehicle drives over different types of ground), it was assumed here that the cornering stiffness is a constant. This is a valid assumption, since the objective for the lateral control algorithm is to keep the small scale vehicle on the model highway used in the FLASH Lab at all times. Once the values for C_f and C_r are found, the mathematical model of the vehicle dynamics is complete and can be used to design the control algorithm itself.

Note that it is possible to combine the two approaches outlined above: A control algorithm is designed on the initial estimates for C_f and C_r and used to control the actual plant. While the vehicle is driving along the road, the input obtained from the image processing algorithm is fed into the control algorithm. The output of the controller (the steering commands) is sent to the vehicle and applied to the mathematical model of the plant. The output of the model and the actual plant output can be compared and new estimates for C_f and C_r are calculated. These new estimates are finally used to design a new control algorithm that will then take over the the control of the vehicle. This combination of on-line parameter estimation and adaptive controller promises to be very sophisticated but it requires a considerable amount of computing resources. The lack of these resources is the reason why this combination was not implemented in the system at hand.

6.3.2.3 Implementing a Parameter Estimation Algorithm

For the implementation of a parameter estimation algorithm outlined in the previous section, three things are needed:

1. An initial estimate for C_f and C_r.
2. A way to compare the output of the actual plant with the output of the mathematical model. In other words, a function is needed that returns a nonnegative value based on the similarity of the two outputs. The more similar the two outputs are, the smaller the returned value. Two identical outputs should return 0.

3. A method that indicates how to modify the estimated parameters to make the output of the mathematical model more similar to the actual output.

An initial estimate for C_f and C_r can be chosen almost arbitrarily. However, to avoid a large number of iterations in the parameter estimation algorithm, the initial estimates should be in the general area of the final values. This is especially important if the parameter space has local minima (see below). Therefore the initial estimate should come from a table of cornering stiffnesses or from previous calculations.

One way to compare the two outputs is to have a function $f(C_{f_{est}}, C_{r_{est}})$ that calculates the sum of the squared difference between the two outputs at each time step. Assuming that there exist outputs from both systems for the time steps $1 \dots n$, the nonnegative value returned by the function can be calculated as

$$f(C_{f_{est}}, C_{r_{est}}) = \sum_{i=1}^{n} (\vec{y}_{actual}\ (i) - \vec{y}_{model}\ (i))^2 \quad \text{with } \vec{y}_{model} \text{ using } C_{f_{est}},\ C_{r_{est}} \tag{6.10}$$

In this case, $\vec{y}$ is a two-element vector. The square of a vector is calculated by multiplying the vector with its transpose (the inner product: $||\vec{x}^2|| = \vec{x}' * \vec{x}$). Using this function, the goal of the parameter estimation algorithm can be written as a minimization problem:

$$\text{Find } C_{f_{opt}},\ C_{r_{opt}} \text{ with } f(C_{f_{opt}}, C_{r_{opt}}) = min\, ||f(C_{f_{est}}, C_{r_{est}})|| \tag{6.11}$$

This is also called the Minimum-Least-Square method. Note that this is not the only way to compare to the two outputs. An alternate way would be to look at the squared difference of maxima of the two outputs.

$$f(C_{f_{est}}, C_{r_{est}}) = (max\, ||\vec{y}_{actual}|| - max\, ||\vec{y}_{model}||)^2 \quad \text{with } \vec{y}_{model} \text{ using } C_{f_{est}},\ C_{r_{est}} \tag{6.12}$$

However, the Minimum-Least-Square method is the most common algorithm for parameter estimation.

As mentioned before, the goal of the parameter estimation algorithm is to find the two values $C_{f_{opt}}$ and $C_{r_{opt}}$ that minimize the function $f(C_{f_{est}}, C_{r_{est}})$ or finding the global minimum in the parameter space of $C_{f_{est}}$ and $C_{r_{est}}$. This space can be viewed as a "terrain" over the $C_{f_{est}}, C_{r_{est}}$-plane where the value $f(C_{f_{est}}, C_{r_{est}})$ denotes the height at a certain point $(C_{f_{est}}, C_{r_{est}})$ in this plane (see Figure 6.4).

In general, ($C_{f_{opt}}$,$C_{r_{opt}}$) can be found by starting at a point (C_{f_0}, C_{r_0}) and going in the direction of the steepest descent. The simplest method to update the estimates for C_f and C_r is therefore to use the negative gradient:

$$\begin{pmatrix} C_{f_{new}} \\ C_{r_{new}} \end{pmatrix} = \begin{pmatrix} C_{f_{old}} \\ C_{r_{old}} \end{pmatrix} - \nabla f(C_{f_{old}}, C_{r_{old}}) \tag{6.13}$$

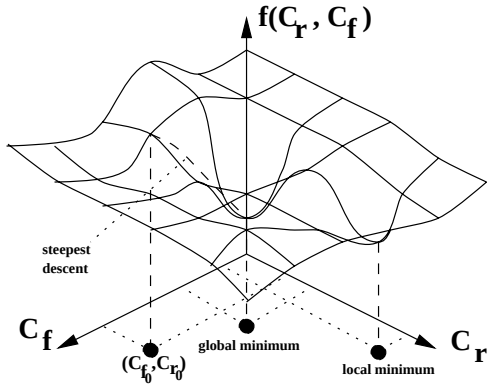

Figure 6.4: The parameter space for C_f and C_r

The gradient ∇f can be calculated by:

$$\nabla f(C_{f_{old}}, C_{r_{old}}) = \begin{pmatrix} \frac{\partial f(C_{f_{old}}, C_{r_{old}})}{\partial C_{f_{old}}} \\ \\ \frac{\partial f(C_{f_{old}}, C_{r_{old}})}{\partial C_{r_{old}}} \end{pmatrix} \tag{6.14}$$

There are however a number of problems with this method. First, in order to calculate the gradient of f, the derivatives

$$\frac{\partial \vec{y}_{model}}{\partial C_f} \quad \text{and} \quad \frac{\partial \vec{y}_{model}}{\partial C_r} \tag{6.15}$$

have to be calculated. In the frequency domain, $\vec{y}_{model}$ can be written as

$$\vec{y}_{model} = \begin{pmatrix} \varphi_h \\ \Delta y \end{pmatrix} = \begin{pmatrix} T_{\varphi_h} * \delta f \\ T_{\Delta y} * \delta f \end{pmatrix} \tag{6.16}$$

where T_{xxx} are the respective transfer functions. These transfer functions can be obtained by transforming the continuous state space matrices A, B and C (Laplace transform). In the discrete representation, the matrices A_d, B_d and C are transformed to the discrete transfer functions (z-transform). The problem that arises when calculating the derivative of Equation 6.15 is that the parameters C_f and C_r are not linear in the transfer functions (both continuous and discrete). In other words the transfer functions cannot be written as

$$T_{xxx} = C_f * T_1 + C_r * T_2 \tag{6.17}$$

since there are also factors with $C_f * C_r$ and C_f^2. This is called *non-linear-in-parameter* and makes the analytic calculation of the gradient very difficult or even impossible. A solution

is to approximate the differential $\frac{\partial}{\partial C_x}$ by a difference as shown here for C_f.

$$\frac{\partial f(C_f, C_r)}{\partial C_f} \approx \frac{f(C_f + \Delta C_f, C_r) - f(C_f, C_r)}{\Delta C_f} \tag{6.18}$$

This makes it possible to calculate the gradient in Equation 6.13.

Another problem is local minima in the parameter space (see Figure 6.4). The method of the steepest descent might end up in a local minimum instead of the global minimum. As explained above, this can be prevented by choosing the initial estimates close enough to the global minimum.

Although the steepest descent method will eventually converge to a minimum, it might take many iterations to do so. To speed up the process, more sophisticated methods called quasi-Newton methods can be used. The methods are implemented in the MATLAB Optimization Toolbox and are explained in the accompanying manual [25].

After all the necessary methods and algorithms are explained, the parameter estimation algorithm can be summed up as follows:

Step 1: Come up with an initial estimate for C_f and C_r.

Step 2: Calculate the A,B and C matrix of the mathematical model using these parameters

Step 3: Apply some input (steering commands) to the model and record the output.

Step 4: Apply the same input to the actual plant and record the output.

Step 5: Measure the difference between model output and actual output (e.g. by using Minimum-Least-Square).

Step 6: Change the parameters in a direction that minimizes the result of **Step 5** (steepest descent or other method).

Step 7: Go back to **Step 2** and iterate until the result of **Step 5** is less than a predefined bound (close enough to zero).

A final remark on parameter estimation in general. In order to be able to estimate any parameters, the states of the plant that are affected by these parameters have to be observable. In other words the changing of a parameter has to have an effect on the the output of the modeled plant since otherwise no optimization of the parameter is possible. In this case, all states of the modeled vehicle dynamics are observable. In control theory, this is written as $\text{rank}(C'\ C' * A \ldots) \neq 0$, with A and C being the matrices from the state space model. Therefore a parameter estimation is possible with this plant.

6.3.2.4 Results and Problems with Parameter Estimation

The parameter estimation algorithm outlined above has been performed for the two cornering stiffnesses C_f and C_r. Unfortunately, the results of this algorithm were not very encouraging. Here is a list of some of the problems encountered:

- It was in no case possible to get exactly the same outputs when applying the same input to both the mathematical model and the actual plant, no matter what values were assumed for the cornering stiffnesses (see Figure 6.5). This problem gets worse with a more complicated input (see Figure 6.6). It can take such an extent that the parameter estimation returns a 0 as the best estimate for one of the the cornering stiffnesses. While a zero (which results in constant zero output for the model) might be the best value to approximate the actual output, it is not a valid value for the modeled vehicle dynamics.

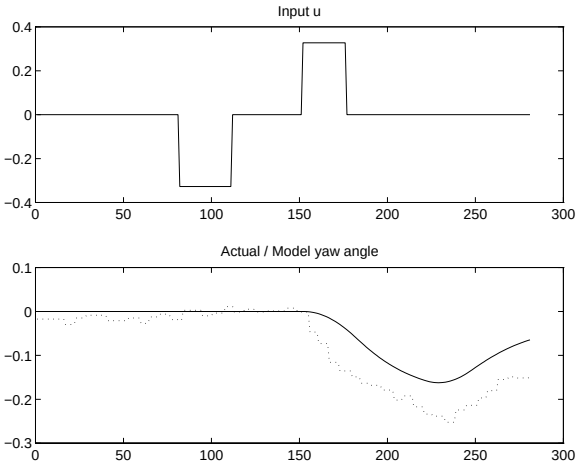

Figure 6.5: Result of parameter estimation (note delay between input and output)

- When the parameter estimation returns values for C_f and C_r, they do not stay the same when different input signals are applied. The values for the cornering stiffness of the rear wheel C_r vary over an especially large interval. A reason for this could be that the rear wheels are not steerable, therefore their effect on the output is not so much noticeable when the only input to the system is a steering angle for the front wheels.

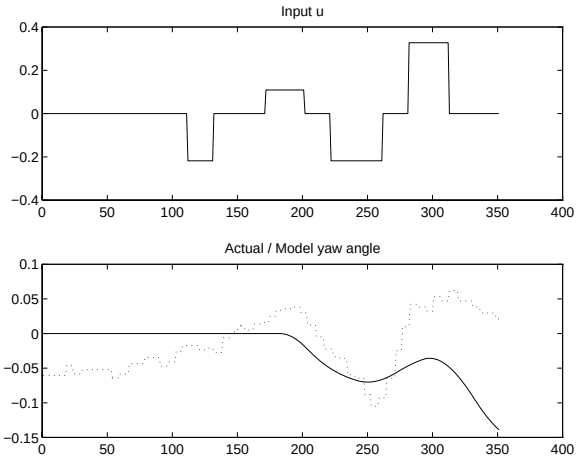

Figure 6.6: Another parameter estimation result (here with vehicle not pointing exactly in the direction of the road initially

- The actual output of the plant (heading angle φ_h and lateral deviation Δy) is measured by using the vanishing point analysis outlined in Chapter 5. The measured variables are very noisy. In fact the measured lateral deviation is so disturbed that it is unusable, at least for parameter estimation purposes. Therefore the estimation had to be performed on the output of the heading angle alone.

Despite all these problems, some results could be obtained. After a number of parameter estimations with different input signals was done (about 5 different input signals returned usable results), the values seemed to concentrate in the intervals shown in Table 6.2.

Table 6.2: Variation in cornering stiffnesses

parameter	description	range	nominal value
$C_f[N/rad]$	cornering stiffness of front wheels	0.25 - 1.0	0.5
$C_r[N/rad]$	cornering stiffness of rear wheels	500 - 1500	1000

It has to be stressed that the real values for C_f and C_r are probably not the nominal values of Table 6.2. The small number of parameter estimations done does not even justify the statement that the real values lie in the specified range. The parameters however do make some physical sense in that they are about 100 to 1000 times smaller than cornering stiffnesses of a full-scale vehicle [12], roughly the same factor by which the vehicle mass is scaled down. The main use of the nominal values and ranges of Table 6.2 is to have a starting point for the control algorithm design introduced in Section 6.4

The following is an attempt to explain the bad performance of parameter estimation algorithm and some possible improvements.

- As previously mentioned, measuring the actual output of the plant with an image processing algorithm introduces a lot of noise in the measured variables. Better results might be obtained by filtering the actual output before comparing it with the model output (e.g. a low-pass filter to smoothen the output). A filter could also make the lateral deviation output usable for parameter estimation. However, the results of using filtered actual outputs have to be part of future work.

- The measurement is also not very accurate. The error for the heading angle is about $\pm 5^o$ and about $\pm 0.01m$ for the lateral deviation. A possible solution here could be not to rely on image processing at all. While it is good enough for operating the lateral controller, the image processing algorithm might be too inaccurate for parameter estimation. Instead, a device similar to a computer mouse could be used to detect movement and orientation of the vehicle.

- The steering servo that translates the steering commands (inputs) to a movement of the front wheel angle is not very precise. When the same steering command is sent to the vehicle at different times, the actual position of the front wheel might differ by several degrees. A solution is of course to use a better servo. This however violates the premise that the hardware in use should be as inexpensive as possible.

- Finally the fact that it is not possible to create an output with the mathematical model that is exactly the same as the actual output indicates that the model used is incomplete. There might by dynamics in the system that were not modeled at all or the linearization of the non-linear original model was an over-simplification. Possible solutions that address this problem are a better linear model (more states modeling missing dynamics) or even a non-linear model. Increasing the complexity of the model however also makes the control algorithm design more complicated, especially for a non-linear model.

6.4 Using H_∞ Control for Lateral Control

In this section, a different way to design a control algorithm for lateral control is presented. Here, the framework of H_∞ control theory is used (see [21] for an introduction to H_∞ control). Other than the control algorithm design presented in Section 6.2, a mathematical model of the vehicle dynamics is absolutely required for this design. Even though the mathematical model derived in Section 6.3 is inaccurate, it can still be used for the controller design in this section. The reason for this is that a properly designed H_∞ control algorithm guarantees stability [6] for the plant it controls, even if the plant contains some uncertainties such as unknown parameters. The only assumption that has to be made is that these uncertainties have to be bounded, or more precisely, the worst-case deviation of the plant due to the uncertainties has to be known. This property of H_∞ control is also called robust stabilization.

The nominal values and ranges in Table 6.2 can therefore be taken as a starting point for the H_∞ controller design. The ranges in which the actual values lie can then be treated as a bounded uncertainty, making the true plant a combination of the nominal model and the (unknown) uncertainties due to not exactly known parameters. This can be written as

$$G = G_0 + \Delta_M G_0 \tag{6.19}$$

where G is the transfer function of the true plant, G_0 is the transfer function of the plant using nominal values for the unknown parameters and the factor Δ_M represents the bounded uncertainty. In this case, G and G_0 are actually two-element transfer function matrices, mapping the input δ_f to the outputs φ_h and Δy. Note that throughout this section, the plant will be referred to both in the representation as a transfer function matrix (frequency domain) and in the representation using the state space matrices (time domain). The correspondence between the two representations is

$$G\ (s) = C_G * (s * I - A_G)^{-1} B_G \tag{6.20}$$

with I being the Identity matrix or

$$G\ (z) = C_G * (z * I - A_{G_d})^{-1} B_{G_d} \tag{6.21}$$

in the discrete case. The matrices A_G, B_G and C_G are the matrices of Equation 6.7, the subscript G has been added to make clear that these matrices are used to form the plant G.

[6] Stability of the plant means in this case that the vehicle stays on the road

6.4.1 The Objective

As mentioned earlier, the objective for the controller design described here is to obtain a lateral controller that creates steering commands for the vehicle based on its present location with the goal of keeping the vehicle on the road. The plant however, is exposed to several disturbances that make the job for the control algorithm more difficult (see Figure 6.7):

1. The model for the vehicle dynamics in Section 6.3 assumes that the road is always straight. Curves in the road can therefore be modeled as external disturbances to the plant (Δ_{ext} in Figure 6.7).
2. The variations in the cornering stiffness C_f and C_r can be modeled with a multiplicative uncertainty. The real plant G is modeled as $G = G_0(I + \Delta_M)$ (see Figure 6.7, equation 6.19). This multiplicative uncertainty can also take the form of a disturbance.

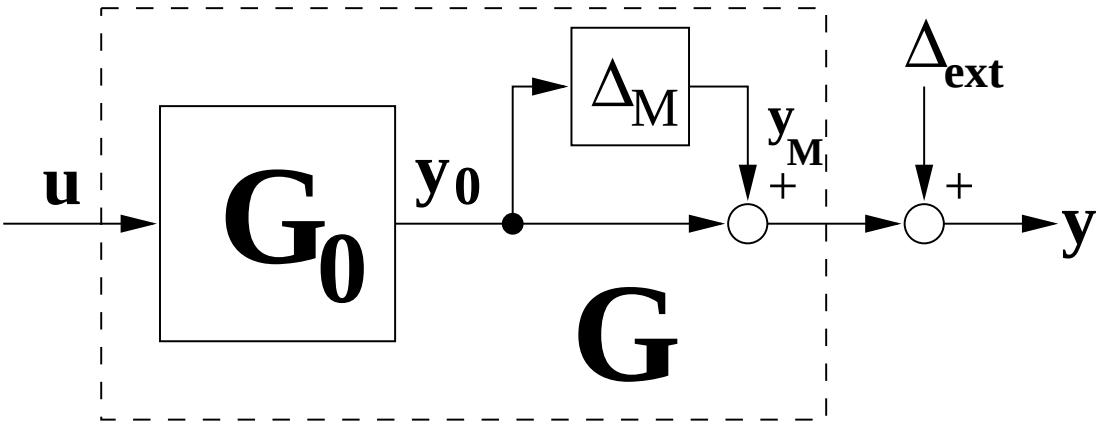

Figure 6.7: Disturbances acting on the plant

Since there are two different disturbances due to two different effects in the plant, there are also two different design goals for the lateral control algorithm. These design goals can be specified as following:

Design for robust stability: The uncertainties in the plant due to parameter variations should have a minimal effect on the control signal u (the output of the controller). In other words, the complementary sensitivity function T with

$$T = KG(I + GK)^{-1} \tag{6.22}$$

should be minimized. This is the transfer function of the closed loop that maps $y_M \rightarrow u$ (see Figure 6.8, see also [33]).

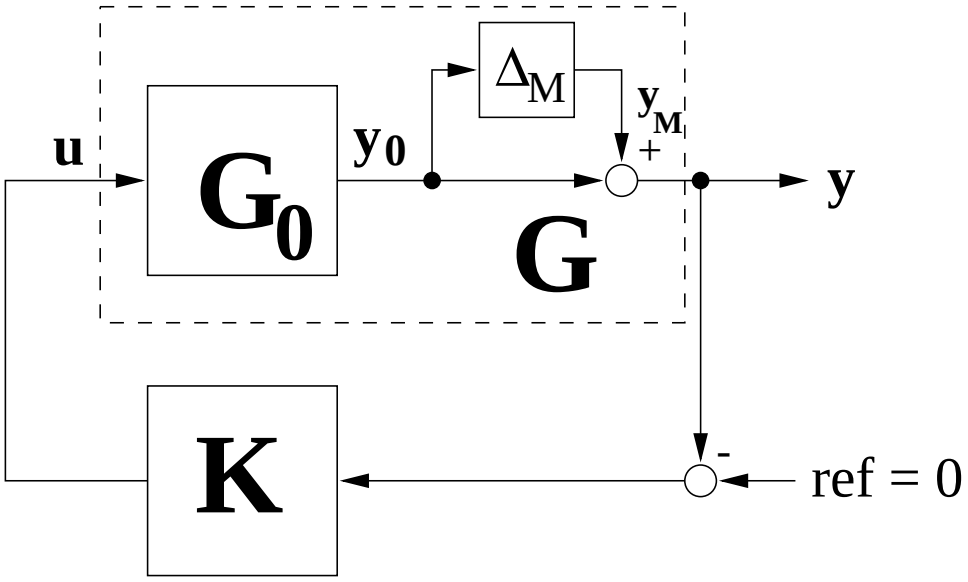

Figure 6.8: Effect of plant uncertainties on controller output

Design for performance: The external disturbances (curves in the road) should have a frequency dependent effect on the output of the plant. For low frequencies, the effect of the disturbances should be maximized (the vehicle should follow the curve) whereas for high frequencies, the effect of the disturbances should be minimized (high frequency disturbances are most likely measurement errors). This design goal can be expressed as a frequency dependent weighting of the sensitivity function S with

$$S = (I + GK)^{-1} = I - T \tag{6.23}$$

As shown in Figure 6.9, the sensitivity function is the closed loop transfer function that maps $\Delta_{ext} \rightarrow y$ (see [33]).

For the H_∞ control design described here, the mixed sensitivity approach is used [21, 24]. In H_∞ terms, the goal of the mixed sensitivity approach can be specified as follows:

$$\left\| \begin{array}{c} W_1 S \\ W_3 T \end{array} \right\|_\infty < \gamma \tag{6.24}$$

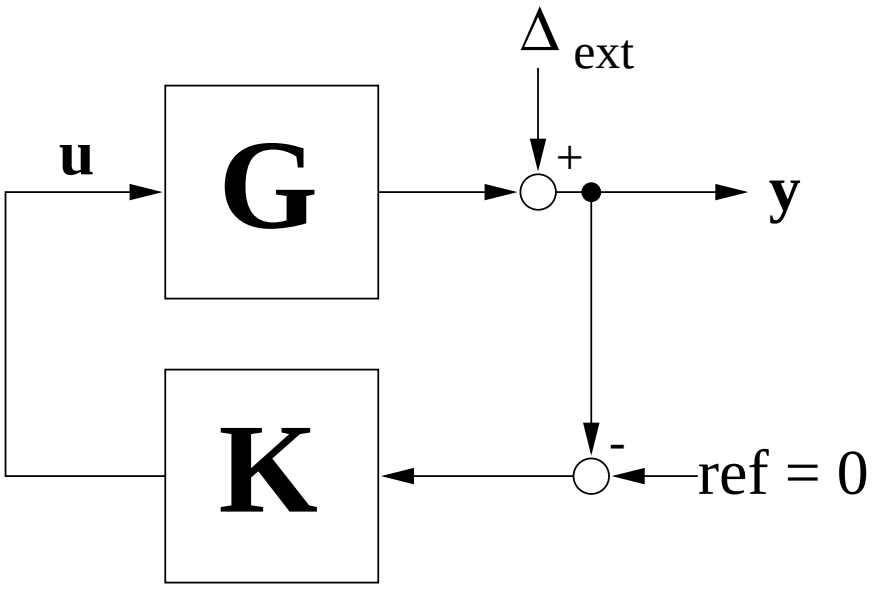

Figure 6.9: Effect of external disturbances on plant output

The H_∞ norm of W_1S and W_3T has to be smaller than some upper bound γ at all times and at all frequencies (normally the upper bound γ is scaled to be 1). Here W_1 and W_3 are some frequency dependent weighting functions that can be used to further specify a design goal for the two transfer functions (see Section 6.4.3 for details).

6.4.2 The H_∞ generalized regulator problem

A standard way of designing an H_∞-controller is by solving the generalized regulator problem [21, Section 4.2 and 8]. In this problem, there exists a system P known as the generalized plant with the following state space representation (these matrices are not identical to the ones in Equation 6.7, see below):

$$\begin{aligned} \dot{x} &= Ax + B_1 w + B_2 u \\ z &= C_1 x + D_{11} w + D_{12} u \\ y &= C_2 x + D_{21} w + D_{22} u \end{aligned} \tag{6.25}$$

This system P contains four signals w, u, z and y. The signal w of dimension l that contains all exogenous inputs and model-error outputs, the signal u of dimension m that is the controller output (and therefore a plant input) and the signal y of dimension q as the

controller input (and therefore a plant output). The signal z of dimension p is called the objective signal.

The goal for the generalized regulator problem is to find a controller K for the generalized plant P so that the transfer function R_{zw} mapping the input w to the objective z (see Figure 6.10) is smaller than the upper bound γ for all frequencies. An additional constraint is that the closed loop of P and K has to be stable. In H_∞ terms, this can be expressed as

$$||R_{zw}||_\infty < \gamma \text{ and } K \text{ stabilizes } P \tag{6.26}$$

In this case, the input signal w consists of a 2-dimensional exogenous input that lumps together the effect of the external disturbance Δ_{ext} and the effect of the multiplicative uncertainty Δ_M on the 2-dimensional plant output y consisting of the heading angle φ_h and the lateral deviation Δy. The control signal u is the 1-dimensional steering angle δ_f. The objective signal z is

$$z = \begin{bmatrix} W_1 S \\ W_3 T \end{bmatrix} \tag{6.27}$$

as defined in Section 6.4.1. This is a 3-dimensional signal in this case.

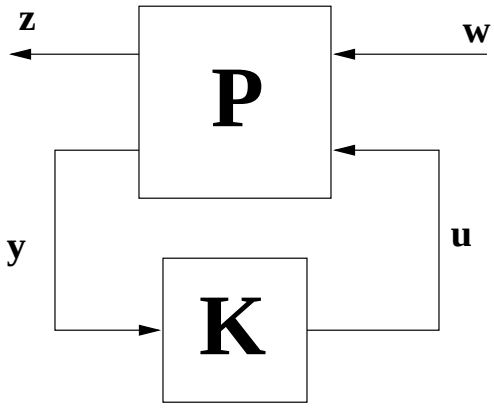

Figure 6.10: The generalized regulator problem

For the matrices of the generalized plant, the following standard assumptions are made [21, Section 4.2]:

1. (A, B_2, C_2) is stabilizable and detectable
2. $\text{rank}(D_{12}) = m$ and $\text{rank}(D_{21}) = q$ (here $m = 1$ and $q = 2$)

3. rank $\begin{bmatrix} j\omega I - A & -B_2 \\ C_1 & D_{12} \end{bmatrix} = m + n$ for all real ω (here $m + n = 6$). [7]

4. rank $\begin{bmatrix} j\omega I - A & -B_1 \\ C_2 & D_{21} \end{bmatrix} = q + n$ for all real ω (here $q + n = 7$).

Assumption 1 is necessary and sufficient for the existence of a stabilizing controller. Assumption 2 makes sure that there are at least as many objectives as controls and as many exogenous inputs as plant outputs. Assumptions 3 and 4 say that the system P has to have full rank on the imaginary axis. This means that no poles or zeros of P can be on the imaginary axis. This is necessary for the solution of the Riccati equation (see section 6.4.6).

6.4.3 The weighting functions

The choice of the weighting functions W_1 and W_3 has to reflect the design goals specified in Section 6.4.1. The design goal for the sensitivity function S was that low-frequency disturbances have to pass through to the output without any change while high-frequency disturbances should be damped. Such a design goal can be achieved by using the following frequency domain representation for W_1 (see Figure 6.11 showing W_1^{-1}):

$$W_1 = \frac{s + 0.1}{s + 0.02} \tag{6.28}$$

When choosing the bound for the H_∞ norm to be $\gamma = 1$, this weighting function ensures that

$$||S||_\infty \leq \left|\left|W_1^{-1}\right|\right|_\infty \tag{6.29}$$

Since W_1^{-1} is large for high frequencies and small for low frequencies, this equation enforces the specified design goal of damping high-frequency disturbances and passing low frequencies (curves in the road).

The weighting function W_3 can be chosen by looking at the variation of the plant output when picking different values for the plant parameters C_f and C_r. The upper plot in Figure 6.12 shows how the output of the plant changes when varying C_f and C_r (only the output of the heading angle is shown). The solid line is the heading angle output for the nominal values ($C_f = 0.5$, $C_r = 1000$), the dotted line is the output for $C_f = 1.0$, $C_r = 1500$ and

[7] n is the number of states in P

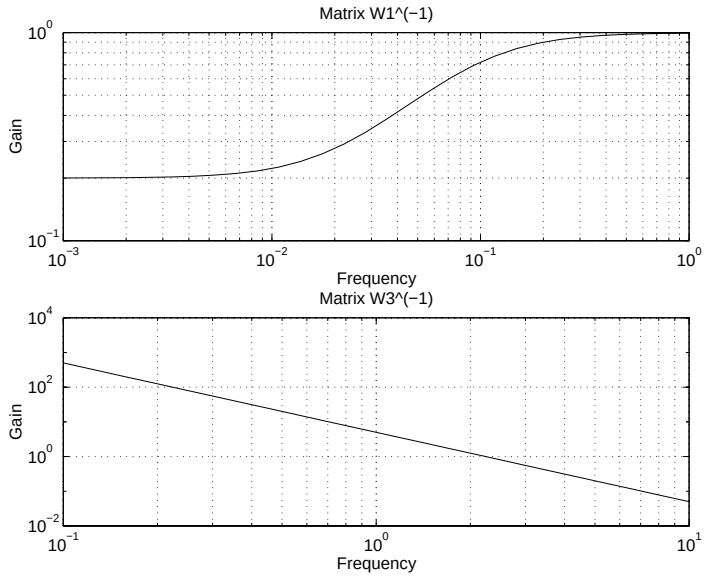

Figure 6.11: The two weighting functions

the dashed line for $C_f = 0.25$, $C_r = 500$ (see Table 6.2). The lower plot in Figure 6.12 shows the maximum deviation of the displayed outputs compared to the nominal output according to

$$\Delta_{M_{Maximum}}(\omega) = \max\left(y_x(\omega)\right) \tag{6.30}$$

where y_x is the output of the plant with different C_f, C_r. It can be observed that the change of the output compared to the output of the nominal plant is always smaller than 5 dB over all frequencies. Since the change of the output due to parameter variation has been modeled by a multiplicative uncertainty (see Figure 6.7), an upper bound for this uncertainty is $\Delta_{M_{max}} = 5\ dB$.

Assumption 3 and 4 for the generalized regulator problem stated that the generalized plant P has to have full rank for all ω. The plant G however has two poles at zero (the double integrator mentioned in Section 6.3), therefore it does not have full rank at $\omega \to \infty$. Since the transfer function T is formed using G (Equation 6.22) and T is used to form the objective signal z of the generalized plant P (Equation 6.27), the generalized plant also does not have full rank at ∞. However by introducing a double differentiator in W_3 the generalized plant P can be made full rank at ∞ [24].

$$G = \frac{G'}{s^2} \implies T = \frac{T'}{s^s} \implies W_3 T = s^2 {W_3}' \frac{T'}{s^2} = {W_3}' T' \tag{6.31}$$

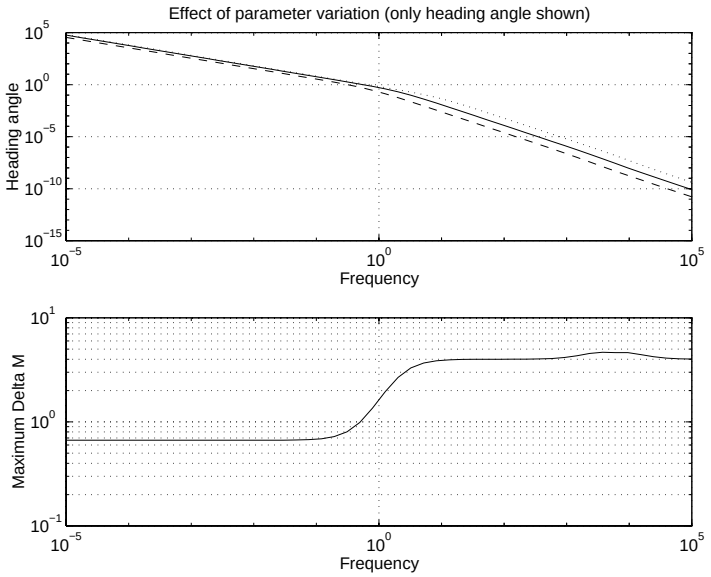

Figure 6.12: Changing output of plant for variation of parameters

With these two premises in mind, the following frequency domain W_3 was chosen (see Figure 6.11 for a plot of W_3^{-1}).

$$W_3 = \frac{1}{\Delta_{M_{max}}} s^2 \doteq P_2 s^2 \tag{6.32}$$

This weighting function enforces the design goal of robust stabilization for the transfer function T that maps $y_M \rightarrow u$.

$$||T||_\infty \leq \left|\left|W_3^{-1}\right|\right|_\infty \tag{6.33}$$

6.4.4 Formulating the generalized plant

After the generalized regulator problem has been adapted for the controller design at hand and some weighting functions reflecting the design goals have been chosen, it is now time to formulate the generalized plant P in terms of the state space matrices of the plant model. The equations in 6.25 can be written as

$$\begin{bmatrix} \dot{x} \\ z \\ y \end{bmatrix} = \begin{bmatrix} A & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} x \\ w \\ u \end{bmatrix} \tag{6.34}$$

with the state space representation of the generalized plant being

$$P = \begin{bmatrix} A & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{bmatrix} \tag{6.35}$$

For the problem at hand, the state space representations of the plant model G and the two weighting functions W_1 and W_3 have to be incorporated in the matrix for P. Note that W_3 is non-proper, therefore no state-space representation exists for it. It is however possible to come up with a state space representation of W_3T since it is proper [24]. According to [21, Section 4], the generalized plant P of Figure 6.10 can be written in the following matrix form (A_G, B_G and C_G are the state space representation of the plant model of Equation 6.7 and A_{W_x}, B_{W_x}, C_{W_x} and D_{W_x} is the state space representation of W_1 and W_3T):

$$P = \begin{bmatrix} A_G & 0 & 0 & 0 & B_G \\ -B_{W_1} & A_{W_1} & 0 & B_{W_1} & 0 \\ B_{W_3}C_G & 0 & A_{W_3} & 0 & 0 \\ -D_{W_1}C_G & C_{W_1} & 0 & D_{W_1} & 0 \\ \tilde{C_G} + D_{W_3}C_G & 0 & C_{W_3} & 0 & \tilde{D_G} \\ C_G & 0 & 0 & I & 0 \end{bmatrix} \tag{6.36}$$

with

$$\begin{aligned} \tilde{C_G} &= P_2 C_G A_G^2 \\ \tilde{D_G} &= P_2 C_G A_G B_G \end{aligned} \tag{6.37}$$

However, as explained in [21, Section 4.2.1], the process of synthesizing the controller is simplified if the generalized plant has the following form:

$$P = \begin{bmatrix} \hat{A} & \hat{B_1} & \hat{B_2} \\ \hat{C_1} & 0 & \hat{D_{12}} \\ \hat{C_2} & \hat{D_{21}} & 0 \end{bmatrix} \tag{6.38}$$

$$\hat{D_{12}'}\hat{D_{12}} = I_m \quad \text{and} \quad \hat{D_{21}}\hat{D_{21}'} = I_q \tag{6.39}$$

which is equivalent to

$$\begin{aligned} \dot{x} &= \hat{A}x + \hat{B_1}w + \hat{B_2}u \\ z &= \hat{C_1}x + \hat{D_{12}}w \\ y &= \hat{C_2}x + \hat{D_{21}}u \end{aligned} \tag{6.40}$$

Therefore the D_{11} term of P matrix has to be removed and D_{12} has to be scaled to be I_3. As explained in [21, Section 4.6], this can be done using loop shifting and scaling transformations. The loop-shifting is used to minimize the D_{11} term while the scaling transformations convert D_{11} to [0] and D_{12} to I_3. Note that by scaling the elements of the D matrix, the bound for the H_∞ norm (assumed to be 1) is also scaled.

Once a controller has been designed for this transformed plant, the inverse of the above transformations have to be applied to the controller to make it suitable for the untransformed model plant.

6.4.5 Bilinear transform

With the generalized plant having the form in Equation 6.36, the generalized regulator program is almost ready to be solved. However there is one problem remaining: Assumption 3 and 4 for the generalized regulator problem states that P must not have any poles or zeros on the imaginary axis. The plant model as it is described in Section 6.3.1 however has a double integrator, corresponding to a double pole at zero. Introducing a double differentiator in W_3 gave P full rank at ∞, but the plant G still has two poles at zero.

This problem can be solved by using a bilinear transform, as shown in [24] for the stabilization of an double integrator. By using the bilinear transform

$$s = \frac{\tilde{s} + p_1}{\frac{\tilde{s}}{p_2} + 1} \qquad p_1 < 0 \quad p_2 = \infty \tag{6.41}$$

in the frequency domain, the imaginary axis in the s-domain gets transformed into a circle with infinite radius located p_1 units to the right of the imaginary axis in the $\tilde{s}$-domain. This is equivalent to just shifting the imaginary axis by p_1 units to the left.

In state space, this transform is done by changing the state matrix of the model plant:

$$\tilde{A_G} \longleftarrow A_G - p_1 * I \tag{6.42}$$

This shifted state matrix $\tilde{A_G}$ can then be used in the generalized plant P. As in the previous section, once a controller has been designed for the shifted matrix, the matrix $\tilde{A_K}$ of the controller (see Equation 6.49) has to be shifted back using

$$A_K \longleftarrow \tilde{A_K} + p_1 * I \tag{6.43}$$

6.4.6 Controller Synthesis

Obtaining a controller that solves the generalized regulator problem normally requires solving two algebraic Riccati equations. Similar to controller design using LQG (see [20]), one solution forms a full state feedback controller (also called full-information controller) while the second solution is used to create and observer that estimates the states of the plant from the actual output.

In [21, Section 8.2] such a controller is designed based on the solution of two Riccati equations. However, the authors also mention several special cases in which solving only one Riccati equation is sufficient. Specifically, if the output of the plant can be written as

$$y = C_2 x + w \tag{6.44}$$

the following observer can be created which perfectly reconstructs all the states $\hat{x}$ and exogenous inputs $\hat{w}$ from the output y

$$\begin{aligned} \dot{\hat{x}} &= A\hat{x} + B_2 u + B_1(y - C_2\hat{x}) \\ \hat{w} &= y - C_2\hat{x} \end{aligned} \tag{6.45}$$

Therefore it is only necessary to solve the first algebraic Riccati equation

$$0 = X_\infty \tilde{A} + X_\infty \tilde{A}' + \tilde{C}'\tilde{C} - X_\infty(B_2 B_2' - \gamma^{-2} B_1 B_1')X_\infty \tag{6.46}$$

with the solution X_∞ being used to form the full-information controller. Combining the solution of the Riccati equation with the observer of Equation 6.45 yields the matrix of the so called central H_∞ controller.

$$\begin{bmatrix} \dot{\hat{x}} \\ u \\ \hat{w} \end{bmatrix} = \begin{bmatrix} A - B_1 C_2 - B_2 F_\infty & B_1 & B_2 \\ -F_\infty & 0 & I \\ -(C_2 + \gamma^{-2} B_1' X_\infty) & I & 0 \end{bmatrix} \begin{bmatrix} \hat{x} \\ y \\ r \end{bmatrix} \tag{6.47}$$

with

$$\begin{aligned} r &= Uw \\ F_\infty &= D_{12}' C_1 + B_2' X_\infty \end{aligned} \tag{6.48}$$

F_∞ is the actual control matrix mapping the observed states $\hat{x}$ to a control output u. The input r is used for reconstructing the exogenous inputs w. Since this is not necessary for the control algorithm designed here, the last column and row of Equation 6.47 can be omitted. This central controller can be used to form all possible controllers that solve the specified

generalized regulator problem by using different functions for U. Any controller solving the generalized regulator problem will have the form of Equation 6.47 only the U is a free parameter that can be varied (refer to [21, Section 6] for details). The most simple solution for a controller K stabilizing the generalized plant P is the central controller itself (setting $U = 0$, therefore $r = 0$ and $\hat{\omega} = 0$). Therefore, a possible controller K is given as

$$u = Ky \text{ with } K = \left[\begin{array}{cc} A - B_1C_2 - B_2F_\infty & B_1 \\ -F_\infty & 0 \end{array} \right] \tag{6.49}$$

Using the transformed generalized plant P developed in Sections 6.4.2 to 6.4.4 and taking care of the back-transformations for the controller, a H_∞ controller K is then synthesized for the plant meeting all the previously specified design goals.

6.4.7 Results of the H_∞ Design

The results presented in this section are divided into two parts. First the designed H_∞ control algorithm is used to control the software implementation of the mathematical model derived in Section 6.3. Plots are given that show the performance of the simulated closed loop. Then the control algorithm is implemented in the code for automated driving and its performance on the actual plant is evaluated.

6.4.7.1 Computer Simulation

The plot in Figure 6.13 shows the open-loop performance of the nominal plant without the control algorithm. It shows the Bode plot for the two transfer functions mapping the steering angle to heading angle and to lateral deviation. Note that the phase of the lateral deviation output is always equal or less than -180^o, the open-loop transfer function is therefore unstable (gain- and phase-margin in a Bode plot, see [22]). This corresponds to experience with the actual plant: without any controller, the model vehicle will eventually run off the road.

When comparing this plot with the Bode plot of the nominal plant with feedback (see Figure 6.14), one can see that the gain goes below 0 dB before the phase crosses the -180^o line, both transfer functions in the closed-loop system are therefore stable.

The plot in Figure 6.14 shows the stability for the nominal plant, that is it shows that the controller stabilizes the plant for which it has been designed. Figure 6.15 however shows that the controller is also capable of stabilizing a plant with a multiplicative uncertainty.

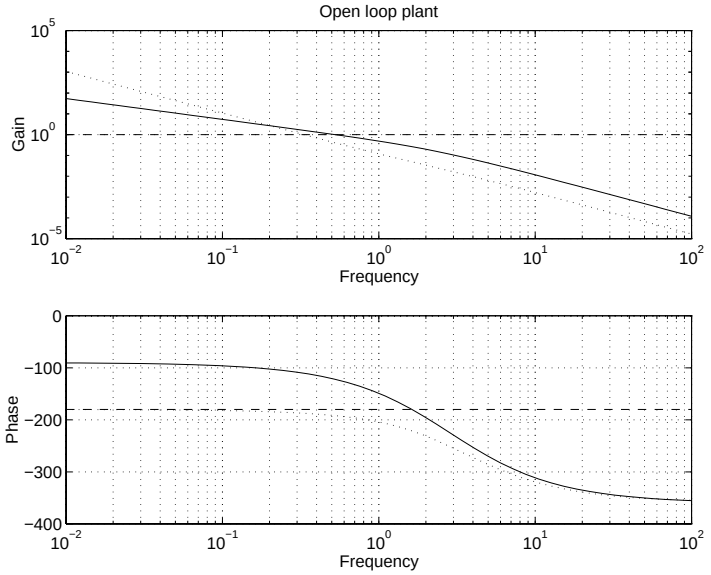

Figure 6.13: The open loop response of the nominal plant

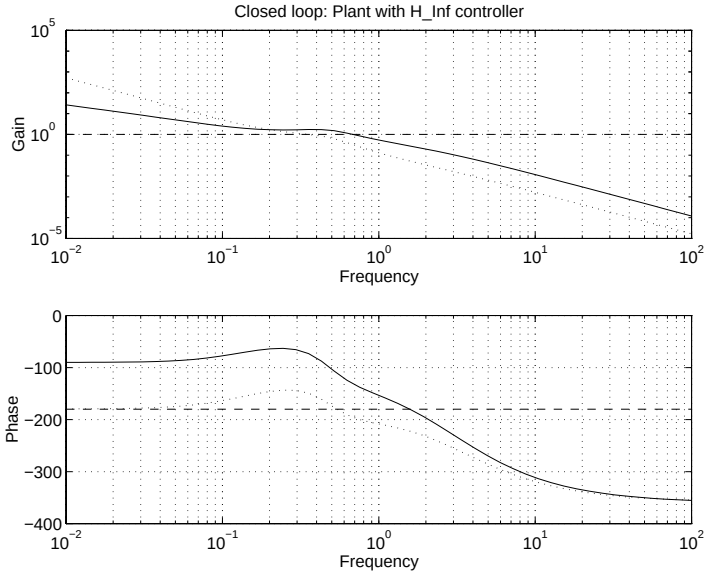

Figure 6.14: The closed loop response with nominal plant and controller

In this case, values from the edges of the ranges for C_f and C_r have been chosen ($C_f = 1.0$ and $C_r = 1500$, see Table 6.2).

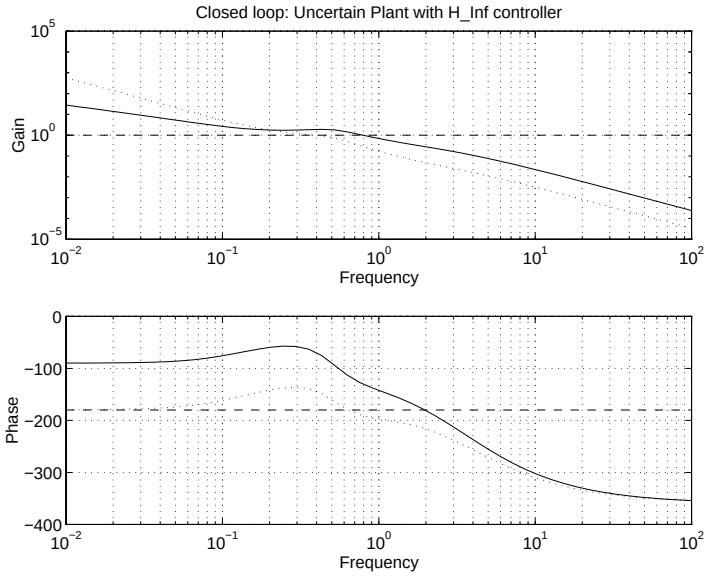

Figure 6.15: The closed loop with a plant containing uncertainties

Figure 6.16 shows the performance due to the weighting function W_1. Here, disturbances of different frequencies were applied to the output (heading angle and lateral deviation)

$$\begin{aligned} \varphi_{h_{dist}} &= \sin(\omega_{dist}\ t) \\ \Delta y_{dist} &= \cos(\omega_{dist}\ t) \end{aligned} \tag{6.50}$$

with ω_{dist} being 0.1 Hz and 5 Hz. The effect on the controller output u is observed. The plot clearly shows that low-frequency disturbances pass through the controller, allowing the vehicle to follow curves in the road (magnitude of output is about 0.5). High-frequency disturbances due to measurement errors however are filtered out by the controller (magnitude only 0.005).

In Figure 6.17, some "real life" disturbances are applied to the outputs of the plant. Such disturbances for heading angle and lateral deviation are typical when the vehicle enters a curve on the road. The plot shows that the controller reacts in the right way: after some initial oscillations it produces a steady state non-zero steering angle that allows the vehicle to follow the curve.

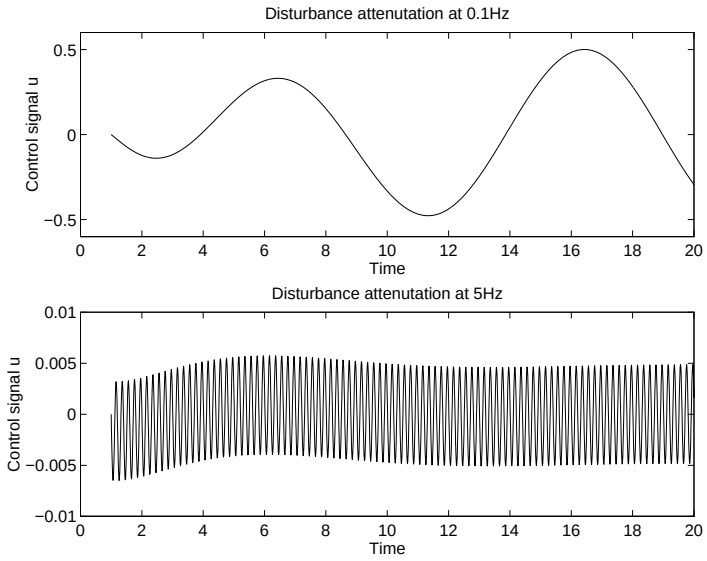

Figure 6.16: Controller output at different frequencies

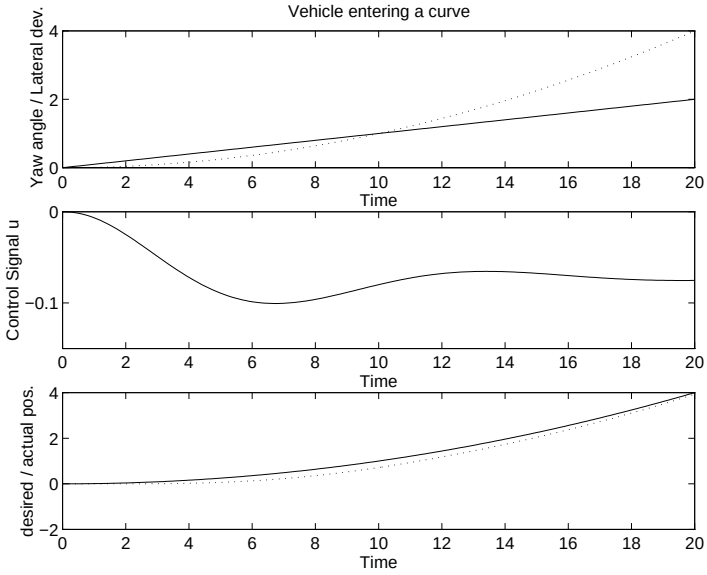

Figure 6.17: Vehicle entering a curve

6.4.7.2 The Actual Plant

In order to implement the the control algorithm in the *AutoIn* task, the A_k, B_k and C_k matrix calculated in Section 6.4.6 have to be converted to to their discrete equivalents using the MATLAB *c2d* function. Since the task is executed every 75 ms, the time interval δt has to be set to this value for the conversion. After this is done, the discrete matrices can be implemented as 2-dimensional floating point arrays in the program code. Since the C language does not support matrix addition or multiplication, the actual calculation of the output steer angle from the input heading angle and lateral deviation has to be done element-wise. Once a steer angle is calculated it can be converted to a steer command and sent to the vehicle.

However, when the automated driving mode was tested with this H_∞ control algorithm, the performance was once again not satisfactory. While the control algorithm does move the front wheels in the right direction when the vehicle strays off the road or enters a curve, the control algorithm either over-reacts (a too large steering angle) or it reacts to late so that the white middle marker has already disappeared off the image plane. Several different control algorithms have been tested using different longitudinal speeds and also different nominal values for C_f and C_r, but the results were similar for each new control algorithm. A possibility for future work could be to experiment with different W_1 and W_3.

This failure of the H_∞ control algorithm might have several reasons:

- As mentioned at the end of Section 6.3.2, the vanishing point analysis for measuring heading angle and lateral deviation is neither very accurate nor is it error-free. It is possible that while the performance of the algorithm is sufficient for a simple controller like the P control, it is not good enough for such a sophisticated control scheme as the H_∞ control.
- The results of the parameter estimation done in Section 6.3.2 did not bring the desired results. Even though the H_∞ design results in a robust control algorithm, it might not be robust enough to cope with this amount of uncertainty. One indication that seems to support this is the Bode plot in Figure 6.14. Even though the simulated plant is stabilized with the control algorithm, the actual gain and phase margin is not very large. So if the actual values for C_f and C_r lie outside of the specified range or even if some of the "known" parameters were not measured precise enough, this might turn the closed loop unstable.

6.5 Comparing the Control Algorithms

With the results of the previous sections, choosing between the P-control and the H_∞ control algorithm is not that difficult. In spite of being a very simple algorithm, the P-control is capable steering the vehicle around the model highway, although not at a very high speed. While the H_∞ control algorithm definitely has the greater potential, it is at this stage not developed enough to be useful. It is possible that precisely the simplicity of the P-control is it's greatest advantage. The simple design makes this control algorithm robust against plant noise in the measurement of the vehicle position and unknown dynamics in the system. Optimizing this algorithm by hand using trial and error provided a greater robustness than a design method that creates intrinsically robust control algorithms. Of course, this does not mean that an H_∞ control algorithm is totally unusable for the lateral control, it just needs a lot more fine tuning in order to work satisfactorily (see also Chapter 7 for this).

Chapter 7

Conclusion

7.1 Concluding Remarks

This thesis work showed the design of a small scale autonomous vehicle system for the FLASH-Lab model vehicle park. A major requirement for the design was that it had to be inexpensive. This goal has clearly been reached, since the total hardware/software cost for the presented system is only about $1000, depending on what type of vehicle is used. This includes components that only have to be bought once (like the real-time kernel software) or hardware that can be used in other vehicle designs (GCB11 board, wireless radio modem). Outfitting the FLASH Lab with a number of these vehicle types is therefore not expensive.

The designed autonomous vehicle system can be operated in two modes: a telerobotic manual mode and a fully automated driving mode. Both modes of operation make use of a decentralized architecture consisting of a mobile unit with sensors, actuators and possibly a microcontroller as well as a stationary unit with a control computer and the above mentioned remote control station. Software has been written for the microcontroller on the mobile unit and the stationary control computer. The C program on the microcontroller provides a high-level interface for driving commands an incorporates a simple longitudinal control algorithm. The program on the control computer was laid out as a multi-tasking system. Its tasks are responsible for managing automated and manual driving, establishing an maintaining communications with the mobile unit, interacting with the user etc. Both programs and the accessed hardware have been thoroughly tested and are fully operational.

In telerobotic mode, the vehicle is operated by a human driver sitting at a remote control station with full-scale vehicle controls (steering wheel, gas pedal) and a video monitor showing the view in front of the model vehicle. By itself, this operating mode can be used for the "human factor" studies outlined in Section 4.3.3. It can also be used as part of a demonstration about the capabilities of future smart highways. Finally this mode is useful to demonstrate the transition from manual to automated driving. The telerobotic mode is fully functional and the vehicle can operate with no wire connection to the stationary unit.

The automated driving mode uses a combination of image processing and a lateral control algorithm to generate steering commands that keep the vehicle on a small scale road or highway. All that is necessary for the automated driving mode to work is that a white middle marker is visible in the image plane of the video camera. The system is capable of operating with no wire connections in automated driving mode, it is however advisable to sent the video signal thru a wire in order to reduce the effect of noise in the HF channel.

For the image processing part, two algorithms have been tested. Both of them work well within their limitations (one of them being that the road surface is flat) [1] and provide information about the position and orientation of the vehicle with respect to the road. For future applications, the vanishing point analysis should be the first choice, since it offers some advantages (two independent variables, less sensitive to stray light) over the intensity profile algorithms.

The thesis work also included the testing of two different lateral control algorithms. The first one, relying on a simple classical control law is working reliably and can be used for automated driving at moderate speed. The second algorithm needed a mathematical model of the vehicle dynamics as a prerequisite. Deriving this model was done by using parameter estimation for a set of vehicle parameters that are difficult to measure. Due to a number of reasons, one of them being the inexpensive layout of the system, the results of the parameter estimation were not completely satisfying. Nevertheless, the results were used to design a robust control algorithm based on H_∞ control theory, hoping that the plant uncertainties would not be a problem due to the robust nature of H_∞. This algorithm was then implemented and tested in automated driving mode. However, the performance of this lateral controller could not compare with the other, more simple design. While definitely having the greater potential due to its robust stabilization and greater complexity, at this stage the H_∞ control algorithm lacks the necessary performance for automated driving mode and requires future fine-tuning. Still, the design procedure itself is valid, as the results of the computer simulations in Section 6.4.7 show. All that is necessary is a more accurate mathematical model and probably more sophisticated hardware.

[1] Note that a flat road is not required for the longitudinal controller

7.2 Possible Future Work

The autonomous vehicle system presented in this thesis work was not designed as a fully developed system ready to be transfered to a full-scale vehicle. Instead, the main focus was on experimentation and research in different areas of autonomous vehicle technologies and algorithms. Possible future work based on this thesis therefore is further experimentation, research and optimization of the presented system. This work can be divided in three different areas.

7.2.1 Work on Hardware and Software

This is probably the area were the least work has to be done. While designing the autonomous vehicle control system, especially the software running on the control computer, great care has been taken to make it open for further extensions. Different types of image processing or control algorithms can be incorporated just by replacing the code in the *AutoIn* task, the rest of the program and the interaction of this task with other tasks can stay the same. The software framework and the hardware itself are reliable and should not require any major modifications. One area that could still be improved is the wireless transmission of the video signal to the control computer (stronger amplifier, signal filters etc.) Of course, the control software is not restricted to vehicle control based on image processing. It is possible to introduce new sensor hardware and adapt the software for it (see next section).

7.2.2 Image processing and General Data-Gathering Algorithms

The image processing algorithm presented here both work reliably and should not require modifications. However, they still have a number of limitations. First, neither algorithm addresses the problem that arises when the road is not flat. Possible work in this area might therefore be to design an algorithm that can cope with hills and descents on the road, although this is probably very difficult when only a video image as input is used. Another area of research is obstacle detection on the road. Both algorithms assume that there are no other vehicles or stationary objects on the road, an assumption that is hardly realistic in real world use.

As mentioned before, the system is not restricted to video as the sensory input. The image processing algorithm can be generalized as a data-gathering algorithm. The job of this algorithm is to collect sensory input and to determine the vehicle's position and

orientation based on this input. This can be generalized by incorporating other sensor types (ultrasonic, infrared, magnetic etc.) or even combining inputs from different sensors (sensor fusion). As long as the output of this algorithm can be used as input for the lateral control algorithm, any modification can be made.

7.2.3 Control Algorithms

This is obviously the area where most of the future work can be done. The most urgent area to work on is probably the H_∞ control algorithm and related to that, the mathematical model of the vehicle dynamics. With further work spent in these areas, a very sophisticated control scheme for lateral control is possible. Another possibility is replacing the lateral P-control algorithm with an improved classical control method like the PID controller. Implementing modern non-H_∞ control theory like sliding mode or LQG controllers is also possible.

A different area of work on the control algorithms could be the control objective. So far, the only objective was to keep the vehicle on the road. Additional objectives both for lateral and longitudinal control could be to driving comfort (smooth turns and accelerations), fuel efficiency or following a lead vehicle.

Yet another area that was not touched in this thesis work are scaling issues. It has to be determined how well control algorithms developed on a small scale vehicle can be applied to full-scale vehicles. This work should also address the question of whether it is valid to use a mathematical model for a full-scale vehicle on a smaller scale.

In summary one can say that the autonomous vehicle system presented here is working in most of the targeted areas and has room for further experimentation and research.

Bibliography

[1] Pushkin Kachroo, *Setup for advanced vehicle control systems experiments in the flexible low-cost automated scaled highway (FLASH) laboratory*, SPIE's Photonics East Symposium, Mobile Robots X, Philadelphia, PA, 1995

[2] P.Kachroo, K.Özbay, R.G.Leonard, C.Ünsal, *Flexible Low-cost Automated Scaled Highway (FLASH) Laboratory for Studies on Automated Highway Systems*, IEEE Intl. Conf. Systems, Man, & Cybernetics, Vancouver, Canada, 1995

[3] Brett Benham, *FLASH Notes*, Center for Transportation Research, 1995

[4] Steven E. Shladover et al., *Automatic Vehicle Control Developments in the PATH Program*, IEEE Transactions on Vehicular Technology, Vol 40, P. 114-130, IEEE, Piscataway, NJ, 1991

[5] Jesus Mena, *Finding the PATH to Automated Highways*, Berkeley Engineering - Forefront, UC Berkeley, Berkeley, CA 1990

[6] D. Pomerleau, *Neural Network Based Autonomous Naviagtion*, In: *Vision and Navigation: The Carnegie Mellon Navlab*, Kluwer, Norwall MA, 1990

[7] K. Liu, F.L. Lewis, *Fuzzy-Logic Based Navigation Controller for an Autonomous Mobile Robot*, Proceedings IEEE Intl. Conf. Systems, Man, & Cybernetics, P.1782-1789, Piscataway, NJ, 1994

[8] M. Nashman, H. Schneiderman, *Real-Time Visual Processing for Autonomous Driving*, Intelligent Vehicles Symposium Proceedings 1993, P.373-378, IEEE, Piscataway, NJ, 1993

[9] J. Manigel, W. Leonhard, *Vehicle Control by Computer Vision*, IEEE Transactions on Industrial Electronics, Vol 39, P.181-188, IEEE, Piscatawy, NJ, 1992

[10] J. Guldner, V Utkin, J.Ackermann, *A Sliding Mode Control Approach to Automatic Car Steering*, Proceedings of the American Control Conference, P.1969-1973, Baltimore, MY, 1994

[11] R.H. Byrne, C Abdallah, *Robust Lateral Control of Highway Vehicles*, Intelligent Vehicles Symposium Proceedings 1994, P.375-379, IEEE, Piscataway, NJ, 1994

[12] Pushkin Kachroo, *Nonlinear control strategies and vehicle traction control*, UMI, Ann Arbor, MI, 1993

[13] E.C. Yeh, J.-C. Hsu, R.H.Lin, *Image-Based Dynamic Measurement for Vehicle Steering Control*, Intelligent Vehicles Symposium Proceedings 1994, P.326-332, IEEE, Piscataway, NJ,1994

[14] ComputerBoards, *CIO-DASO8 User's Manual*, ComputerBoards, Inc., Mansfield, MA, 1994

[15] CoActive Aesthetics, *GCB1 Networked Microcontroller Reference Manual*, Version 2.0, CoActive Aesthetics, Inc. San Francisco, CA, 1995

[16] Motorola, HC11 Reference Manual, Revision 1, Motorola Inc., 1990

[17] Current Technology, *FF1 DSP Frame GRabber User's Manual*, Version 4.9, Current Technology, Inc., Durham, NH, 1995

[18] On Time Informatik, *RTKernel 4.5 User's Manual*, On Time Informatik GmbH, Hamburg, Germany, 1994

[19] Peter Spasov, *Microcontroller Technology, the 68HC11*, Prentice Hall, Englewood Cliffs, NJ, 1993

[20] P. Dorato, C. Abdallah, V.Cerone, *Linear-Quadratic Control - An Introduction*, Prentice Hall, Englewood Cliffs, NJ, 1995

[21] M. Green, D.J.N. Limebeer, *Linear Robust Control*, Prentice Hall, Englewood Cliffs, NJ, 1995

[22] W. Leonhard, *Einführung in die Regelungstechnik*, Vieweg, Braunschweig, Germany, 1992

[23] W. Leonhard, *Digitale Signalverarbeitung in der Meß- und Regelungstechnik*, Teubner Studienbücher, Stuttgart, Germany, 1989

[24] Richard Y. Ching, Michael G. Safonov, *Robust Control Toolbox*, The MathWorks Inc., 1992

[25] Andrew Grace, *Optimization Toolbox*, The MathWorks Inc, 1992

[26] Andrew Grace et. al.,*Control System Toolbox*, The MathWorks Inc., 1992

[27] J.L.Jones, A.M.Flynn, *Mobile Robots, Inspiration to Implementation*, A.K. Peters Ltd., Cambridge, MA, 1993

[28] K. Müller, *Regelungstheorie - Umdruck zur Vorlesung*, Institut für Regelungstechnik, TU Braunschweig, 1994

[29] A.C. Grove, *An Introduction to the Laplace-Transform and the z-Transform*, Prentice Hall UK, Hertfordshire, Great Britain, 1991

[30] Vladimir Strejc, *State Space Theory of Discrete Linear Control*, John Wiley &Sons, New York, 1981

[31] Pieter Eykhoff, *System Identification, Parameter and State Estimation*, John Wiley & Sons, London, 1974

[32] J.V. Beck, K.J. Arnold, *Parameter Estimation in Engineering and Science*, Wiley Series in Probability and Mathematical Statistics, John Wiley & Sons, New York, 1977

[33] J. Ball, . Rakowski, *Interpolation by Rational Matrix Functions and Stability of Feedback Systems: the 2-Block-Case*, Journal of Mathematical Systems, Estimation and Control, Vol 4, Birkhäuser, Boston, 1994

Appendix A

Sources for the Hardware

Company: America's Hobby Center
Product: TAMIYA Rookie Rabbit
Address: 146 West 22nd. Street
New York, NY 10011-2466
Telephone: 1-800-989-3989 FAX 1-800-323-2992

Company: TAMIYA Plastic Model CO.
Product: Rookie Rabbit
Address: 3-7, Ondawara
Shizuoka-City
Japan

Company: ThrustMaster
Product: Formula T2 (Steering Console)
Address: 10150 SW Nimbus
Portland, OR 97223-4337
Telephone: (530) 639-3200 FAX 620-8094
Internet: http://www.thrustmaster.com (WWW)

Company: ComputerBoards, Inc.
Product: CIO-DAS08/Jr-AO (D/A Board)
Address: 125 High Street, #6
Mansfield, MA 02048
Telephone: (508) 261-1123 FAX 261-1094
Internet: http://www.electricnet.com/cofolder/compbrds.htm (WWW)

Company: Communications Research and Development Corp. (COMRAD)
Product: Wireless Data Link
Address: 7210 Georgetowm Road #300
Indianapolis, IN 46209-8818
Telephone: (317) 290-9107

Company: CoActive Aesthetics, Inc
Product: GCB11 (HC11 Board)
Address: P.O. Box 425967
San Francisco, CA 94142
Telephone: (415) 626-5152 FAX 626-6320
Internet: gcb11@coactive.com (e-Mail)

Company: Motorola
Product: 68HC11
Address: P.O. Box 20912
Phoenix, AZ 85036
Internet: http://www.mcu.motsps.com/bu/amcu/home.html (WWW)

Company: On Time Informatik GmbH
Product: RTKernel 4.5 for C/C++ (Real-Time OS)
Address: 88 Christian Avenue
Setauket, NY 11733
Telephone: (516) 689-6654 FAX 689-1172
Internet: http://www.on-time.com:80/index.htm (WWW)
info@on-time.com (e-Mail)

Company: Goldbeam Electronics, Inc.
Product: 929WS Wireless CCD Camera & Receiver
Address: 2020 West 139th Street
Gardena, CA 90249

Company: SuperCircuits
Product: Goldbeam CCD Camera System, HF Amplifier
Address: One SuperCircuits Plaza
Leander, TX 78641
Telephone: (512) 260-0333 FAX 260-0444
Internet: http://www.scx.com/catalog.html (WWW)

Company: Current Technology, Inc.
Product: FF1 DSP Frame Grabber
Address: 97 Madbury Road
Durham, NH 03824
Telephone: (603) 868-2270 FAX 868-1352
Internet: http://curtech.com (WWW)
ff@curtech.com (e-Mail)

Appendix B

Microcontroller Program

```
/*****************************************************************
*
* FILE:
*       carctrl.c
*
* DESCRIPTION:
*       Routine to control the FLASH vehicle. Receives commands from
*       the Control Computer and translates them to motor speed and
*       steering angles. Also performs simple speed control.
*
* (c) 1996 by Nikolai Schlegel
*
*****************************************************************/

#include "stdlib.h"
#include "stdio.h"
#include "coactive.h"                  /* common equates for all coative
                                          systems */
#include "gcb11.h"                     /* header file for GCB11 equates */
#include "gios.h"                      /* header file for ROM libraries */
#include "gapp.h"                      /* motor header file */

#define PWM_BASE        30000          /* Base frequency for PWM ( = 15ms) */
#define ONE_MS          2000           /* one millisecond */

/* Prototype for timer interrupt service routine */
void timer_isr();

/* Global variables */
int count = 0;                         /* counter for timer isr */
int time_passed = 0;                   /* timer flag */

/*****************************************************************************
*
* FUNCTION:
*       main()
*
* DESCRIPTION:
```

```
*       Main Program. Receives commands from Control Computer and
*       and interprets them.
*
*************************************************************************/
int main()
{
   int cmd;
   int do_steer = 0, do_speed = 0;
   COUNT_REC cnt;
   int motor_num;                      /* logical motor number for foreward
                                          / reverse */
   int des_motor_speed = 0;            /* current desired motor speed */
   int act_motor_speed;                /* current actual motor speed */
   unsigned int motor_pwm;             /* motor PWM-frequency */
   int p_portion, i_portion = 0;       /* controller variables */

   /*  Set the base PWM frequency in E-cycles. 30000 E-cycles = 15ms */
   ga_setup_motors(PWM_BASE);

   /* Initialize logical motors 0, 1 and 3 */
   ga_init_motor(0, 0, 0, 0, 0, 0);    /* motor 0 = servo motor */
   ga_init_motor(1, 0, 0, 0, 0, 0);    /* motor 1 = reverse drive */
   ga_init_motor(3, 0, 0, 0, 0, 0);    /* motor 3 = foreward drive */

   motor_num = 3;                      /* foreward */
   ga_motor_speed(0,0);
   ga_motor_speed(1,0);
   ga_motor_speed(3,0);

   /* initialize interrupt service routine */

   gi_intr_disable();                  /* first disable interrupts */

   TFLG1 = 0x10;                       /* clear oc4 flag */

   (* (UINT16 *) (INTVTMROC4 + 1)) = (UINT16)timer_isr;
                                       /* fill interrupt jump table */

   TMSK1 |= 0x10;                      /* enable oc4 interrupt */

   TOC4 = TCNT + ONE_MS;               /* first interrupt after 1ms */

   gi_intr_enable();                   /* enable interrupts */

   ga_start_counter(COUNTER1, 0, 1, RISING_EDGE, ZERO_NONE, 0, 2500);
                                       /* start counting pulses on IC1,
                                          (rising edge, start at 2500
                                          counting down) */

   /* main loop */
   while (1)
     {
        while (!time_passed);          /* wait for flag from Timer ISR */
        time_passed = 0;               /* reset flag */

        ga_get_counter(COUNTER1,&cnt); /* get counted pulses on IC1 */
        ga_set_counter(COUNTER1,0,0);  /* and reset counter */
        if (cnt.count == 0)
          act_motor_speed = 0;         /* no pulses => vehicle does not move */
        else
          act_motor_speed = 2500-cnt.count;
```

```
                                /* pulses couunt down from 2500 */

        p_portion = 4 * (des_motor_speed - act_motor_speed);
                                /* P-protion of PI controller */
        i_portion = i_portion + (des_motor_speed - act_motor_speed) / 2;
                                /* I-Portion of PI controller */
        if ((p_portion + i_portion) < 0)
          motor_pwm = 10;       /* no negative PWM */
        else
          motor_pwm = p_portion + i_portion;
                                /* new PWM from PI controller */
        if (motor_pwm > PWM_BASE) motor_pwm = PWM_BASE;
                                /* PWM cannot be > 100% */
        ga_motor_speed(motor_num, motor_pwm);
                                /* set new PWM */

        /* process commands from control computer */
        if (gi_strm_size( STDIO, STRM_INPUT ))
        {                       /* command at COM port? */
             cmd = getchar();   /* read command */
             if (do_steer)      /* should we set a new steer angle? */
               {
                  ga_motor_speed(0,768+3*cmd);
                                /* convert parameter to PWM for
                                   steering servo and set it */
                  do_steer = 0; /* steer done */
               }
             else if (do_speed) /* should we set a new speed? */
               {
                  des_motor_speed = cmd * 8;
                                /* convert parameter to PWM for
                                   drive motor */
                  do_speed = 0; /* speed done */
               }
             else               /* otherwise parse cmd */
               {
                  switch (cmd)  /* Which Command? */
                    {
                     case '1':  /* STEER */
                       do_steer = 1;
                       break;
                     case '2':  /* SPEED */
                       do_speed = 1;
                       break;
                     case '3':  /* FORWARD */
                       ga_motor_speed(1, 0);
                                /* stop reverse motor */
                       motor_num = 3;  /* control foreward motor now */
                       break;
                     case '4':  /* REVERSE */
                       ga_motor_speed(3, 0);
                                /* stop foreward motor */
                       motor_num = 1;  /* control reverse motor now */
                       break;
                     default:
                       break;
                    }           /* switch */
               }                /* if */
          }                     /* if */
    }                           /* while */
}
```

```
/*****************************************************************************
*
* FUNCTION:
*       timer_isr()
*
* DESCRIPTION:
*       interrupt service routine.  gets called once every millisecond.
*       Counts up to 0.025s and then sets a flag. Its purpose is to keep
*       track of the time to synchronize the main loop.
*
*****************************************************************************/
void    timer_isr()
{
   if(++count >= 25)                    /* count up to 25 ms */
     {
        count = 0;                      /* reset counter */
        time_passed = 1;                /* set flag to start main loop */
     }

   TFLG1 = 0x10;                        /* clear interrupt flag */

   TOC4 = TOC4 + ONE_MS;                /* next interrupt after 1ms */

   /* this executes the RTI instruction */
   asm("   pulx");
   asm("   rti");
}
```

Appendix C

Control Computer Program

```
// ****************************************************************************
//
// PROJECT     : CONTROL
// FILE        : CONTROL.C
//
// DESCRIPTION :
// This file contains all neccessary tasks for the control of a scaled model
// vehicle used in the FLASH-Lab. These vehicles can either be controlled
// manual mode using steering wheel, gas pedal & throttle or in automatic mode
// using a video camera as an input and some sort of controller to maintain
// the vehicle in the middle of the road
//
// AUTHOR      : Nikolai Schlegel
// CREATED     :  3/ 7/96
// MODIFIED    :  1/20/96
//
// ****************************************************************************

//
// It is possible to create several versions of this program form this source.
// The following options are available :
//
// LOCAL_IO       : creates a version where input received from the keyboard
//                  and output is written to the console.
// SERIAL_IO      : creates a version where input and output go over the
//                  serial I/O port. This allows control of the program from
//                  a remote PC.
// RC_LINK        : creates a version of the program that can control the
//                  vehicle that is steered via a RC-Link and that has no
//                  micro-controller on board.
// SERIAL_LINK    : creates a version of the program that controls the vehicle
//                  with a wireless serial link and an onboard HC11 micro-
//                  controller.
// PROFILE        : Get an simple control error (linear combination of
//                  lateral deviation and heading angle) by finding maximum
//                  of image profile in y-direction.
// VANISH         : Explicitly calculate lateral deviation and heading angle
//                  by detecting edges and using vanishing line algorithm.
```

```
// DATA_LOG       : Record / playback important data. Two modes are possible
//                    * Record steering, speed commands and heading angle,
//                      lateral deviation over time (enabled with "record") on
//                      command line).
//                    * Play back steering ,speed commands, record heading angle,
//                      lateral deviation (enabled with "play" on the command
//                      line)
//                    Note: only works with VANISH
// HINF           : Use an H-Infinity controller instead of the simple
//                  P-control feedback. Requires calculating of 7 states at
//                  all times. Discrete state space matrices for controller
//                  have to be defined in global variable section.
//                  Note: only works with VANISH
//                  Note: Controller not restricted to HInf, any state space
//                        controller will do in principle
// FULL_INFO      : in this version, spefic parameters of the controller like
//                  the control error are continously written to the output
//                  device.
// TRACE_DUMP     : an input of "D" in this version causes an output of the
//                  trace buffer. The trace buffer is written in TRACE.DAT.
//                  This version also display the timing of important FF1
//                  subroutines.
//
// ***************************************************************************
//
// Startup instructions for the control program:
//
//       1. Make sure the wireless modem connected to control computer is on.
//       2. Make sure HC11 and wireless modem on vehicle are on (enough battery)
//       3. The camera should by connected to its battery pack
//       4a. Either connect camera via cable to control computer
//       4b. or connect control computer to HF  receiver and use wireless camera
//       5. Reset HC11 by pressing the button on the vehicle
//       6. The template files 'lower.tpl' and 'upper.tpl' should be in the
//          same directory as 'control.exe'
//       7a. for non-Data-Log operation, type 'control' on the control computer
//       7b. for Data-Log record, type 'control record <file1>.cmd <file2>.dta',
//           <file1> is the file to store steering and speed commands
//           and <file2> is the file to store heading angle and lat. deviation
//           NOTE: heading angle and lateral deviation are only stored in
//           automted driving mode
//       7c. for Data-Log play, type 'control play <file1>.cmd <file2>.dta',
//           where <file1> its the file to read the driving commands from and
//           <file2> is the file to store heading angle and lat. deviation
//           NOTE: start playback by switching to automated mode
//
#define LOCAL_IO
// #define SERIAL_IO
#define SERIAL_LINK
// #define RC_LINK
// #define PROFILE
#define VANISH
// #define HINF
// #define DATA_LOG
// #define FULL_INFO

// #define TRACE_DUMP

// INCLUDE & GLOBAL DEFINES
// ===========================================================================
```

```
// Including the Standard C Headers

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

// Headers for RTKernel

#include "RTKernel.H"         // Kernel functions
#include "ITimer.H"           // Timer Interupt
#include "RTKeybrd.H"         // Keyboard I/O using Interrupts

#if defined(SERIAL_IO) || defined(SERIAL_LINK)
   #include "RTCom.H"         // Serial Communication
#endif

// Headers for Plug-in Card used

extern "C"
{
#include "ff.h"               // Frame Grabber
};

#ifdef RC_LINK
   #undef OVERRUN
   #include "cb.h"            // DAQ Board
#endif

// GLOBAL DEFINES
// ---------------------------------------------------------------------------

// Defines for Task-handling
#define MAINPRIO        2     // priority of main()
#define OUTPUTPRIO      4     // priority of output-related tasks
#define INPUTPRIO       6     // priority of input-related tasks
#define STEERPRIO       8     // priority of tasks that do the actual steering
#define SWITCHPRIO      10    // priority for task that flips modes

#define DEFAULTSTACK    2000  // Default stack size for tasks

// Defines related to serial communication
#ifdef SERIAL_IO
  #define IOPORT        0     // COM-Port to handle program I/O
  #define IOBAUD        115200L // Baudrate for program I/O
#endif
#ifdef SERIAL_LINK
  #define LINKPORT      1     // COM-Port for serial link with vehicle
  #define LINKBAUD      19200 // Baudrate for serial link
#endif

// Constants for transmission over RC-link
#ifdef RC_LINK
   #define FOFFSET       2.5  // offset voltage for foreward speed
   #define FRANGE        2.0  // voltage range for foreward speed
   #define BOFFSET       2.0  // offset voltage for backward speed
   #define BRANGE        1.0  // voltage range for backward speed
   #define SOFFSET       2.5  // offset voltage for steering
   #define SRANGE        1.5  // voltage range for steering
#endif
```

```
// types of status messages WrStat gets
#define START           1     // start of of all controller tasks
#define STOP            2     // stop of all controller tasks
#define MANUAL          3     // switch to manual mode
#define AUTO            4     // switch to automatic mode
#define PINT            5     // output int var stored in WrStat_Int
#define PFLOAT          6     // output float var stored in WrStat_Float
#define ILIST           7     // output list of ints in WrStat_IList
#define INIT            8     // global initialize of system
#define CHECK           9     // check for HC11
#define BOOT           10     // bootup HC11

// Commands for Steer Mailbox
#define STEER           '1'      // set steer angle to value following
#define SPEED           '2'      // set speed to value following

// GLOBAL VARIABLES
// ============================================================================

TaskHandle  ManualTH;         // Task that deals with manual steer input
TaskHandle  AutoTH;           // Task that deals with automatic steer input

TaskHandle  SteerTH;          // Task that does steer output
TaskHandle  FlipAutoManTH;    // Task to switch between modes
TaskHandle  ButtonsTH,        // Other input tasks
            ReceiveTH,
            WaitQuitTH;
TaskHandle  WrStatTH;            // Output task

enum
   {Manual, Automatic} Mode;  // Flag for manual/automatic mode
enum
   {Forewards, Backwards} Direction; // Indicator for direction of travel

#ifdef SERIAL_IO
   bool DoWriteStatus;        // actually write status info or stay quiet
#endif
Semaphore FlipModeS;          // Semaphore to flip modes
Semaphore TerminateS;         // Semaphore to end program

Mailbox StatusMB;             // Mailxbox to write out status info
Mailbox SteerMB;              // Mailbox for steering commands

int      WrStat_Int;          // Integer value to be outputted by WrStat
char     WrStat_IntName[3];   // name of this value
float    WrStat_Float;        // Float value to be written by WrStat
char     WrStat_FloatName[3]; // name of this value
int      WrStat_IList[64];    // Integer list to be written by WrStat
short    WrStat_IListSize;    // size of this Integer list

#ifdef DATA_LOG
   bool doRecord = False;     // Are we in Record-Mode ?
   bool doPlay = False;       // Are we in Playback-Mode ?
   FILE *RecFileH;            // File for recording data
   FILE *PlayFileH;           // File for playing back data
#endif

#ifdef TRACE_DUMP
   char TraceBuffer[2048];    // Buffer to store Trace Data
#endif
```

```
#if defined(SERIAL_IO) || defined(SERIAL_LINK)
   char Buffer[8192];         // Buffer for Serial IO
#endif

#ifdef HINF
   // discrete state space matrices for controller (cut of below 1e-9)
   double Ak[7][7] = {{-6.0912e-4,-3.6732e0,-4.3998e0,-2.08438e0,-1.7932e-1,-8.51e-7,-7.02e-7},
                      {1.1227e-4,6.7685e-1,-3.9268e-1,-1.9002e-1,-1.6414e-2,-7.8e-8,-6.4e-8},
                      {1.0273e-5,6.2098e-2,9.7701e-1,-7.5537e-3,-6.5326e-4,-3.0e-9,-3.0e-9},
                      {4.0878e-7,2.4769e-3,7.4042e-2,9.9233e-1,-1.6796e-5,0.,0.},
                      {1.1e-8,6.3823e-5,2.7839e-3,7.4436e-2,9.9253e-1,0.,0.},
                      {3.9570e-3,2.3971e1,6.8162e2,-2.6945e3,2.6221e2,9.9104e-1,-6.54e-7},
                      {5.7866e-4,3.5056e0,1.007e2,-2.3947e2,-4.8805e2,-1.1531e-7,9.9104e-1}};

   double Bk[7][2] = {{5.52e-7,4.55e-7},
                      {2.2e-8,1.8e-8},
                      {0.,0.},
                      {0.,0.},
                      {0.,0.},
                      {-5.2664e-1,8.9e-8},
                      {1.6e-8,-5.2664e-1}};

   double Ck[7] = {-7.118e3,-2.9144e4,-5.6696e4,-1.0551e5,-5.7304e3,
                   -3.3297e-2,-2.7471e-2};
#endif
// FUNCTIONS (not seperate tasks)
// ============================================================================

// Function Prototypes
// ----------------------------------------------------------------------------
void Error(char *msg);
#ifdef SERIAL_IO
void InitIOPort();
bool ReceiveProg(char *fname);
#endif
#ifdef SERIAL_LINK
void InitLINKPort(void);
bool IsHC11There(void);
void InitHC11(void);
#endif

//
// ----------------------------------------------------------------------------
// NAME        :  Error
// DESCRIPTION :  Write an error message on the screen and sound bell. The
//                message always goes to the local IO device.
//
void Error(char *msg)
{
   printf("\b------------------------------------------------------------\n");
   printf("%s\n",msg);
   printf("------------------------------------------------------------\n\b");
}

#ifdef SERIAL_IO
//
// ----------------------------------------------------------------------------
// NAME        :  InitIOPort
// DESCRIPTION :  Set up everything for communication over IO-Port
//
void InitIOPort(void)
```

```
{
   // Initialze COM-Port 1
   InitPort(IOPORT, IOBAUD, PARITY_NONE, 1, 8);
   AllocateCOMBuffers(IOPORT, 1024, 1024);
   EnableCOMInterrupt(IOPORT, 1024);   // Enable COM-Interrupt

   DoWriteStatus = True;   // status info at first
}

//
// ----------------------------------------------------------------------------
// NAME        :  ReceiveProg
// DESCRIPTION :  Receive a program for download (either CONTROL.EXE or
//                CARCTRL.S19.  Program is tranfered in 2 KB chunks.
//                Transfer format: LenHI, LenLO, Byte0 ... Byte2047, ChkByte
bool ReceiveProg(char *fname)
{
   FILE *fd;
   unsigned  Data;
   int i,len;
   unsigned chk;

   fd = fopen(fname,"wb");

   while (True)
   {
        RTKGet(ReceiveBuffer[IOPORT], &Data); /* wait till a character comes in */
        len = 256 * (Data & 0x00FF);          // get the chunk length (HI-Byte)
        RTKGet(ReceiveBuffer[IOPORT], &Data); /* wait till a character comes in */
        len += (Data & 0x00FF);               // get the chunk length (LO-Byte)
        chk = 0;
        for (i=0; i<len; i++) {               // get the 2KB chunk
           RTKGet(ReceiveBuffer[IOPORT], &Data); /* wait till a character comes in */
           Buffer[i] = (Data & 0x00FF);
           chk += (Data & 0x00FF);            // and calculate checksum
        }
        RTKGet(ReceiveBuffer[IOPORT], &Data); /* wait till a character comes in */
        if ((Data & 0x00FF) != (chk % 256)) { // compare calculated with
           Error("Transmit Error !");         // transmitted checksum
           fclose(fd);
           return False;
        }
        fwrite(Buffer,len,1,fd);              // write the 2 KB chunk
        if (len < 2048) {                     // Read another chunk?
           fclose(fd);
           return True;
        }
   }
}
#endif

#ifdef SERIAL_LINK
//
// ----------------------------------------------------------------------------
// NAME        :  InitLINKPort
// DESCRIPTION :  Set up everything for communication over IO-Port
//
void InitLINKPort()
{
   // Initialze COM-Port
   InitPort(LINKPORT, LINKBAUD, PARITY_NONE, 1, 8);
```

```
   AllocateCOMBuffers(LINKPORT, 8, 8);
   EnableCOMInterrupt(LINKPORT, 8); // Enable COM-Interrupt
}

//
// ---------------------------------------------------------------------------
// NAME        :  IsHC11There
// DESCRIPTION :  Probes LINK-Port to check if there is a serial connection to the
//                HC11 MC onboard the vehicle.
//                !!! Locks the program if no HC11, therefore commented out
bool IsHC11There()
{
// int i;
// char c;
// SendChar(LINKPORT, 0x0D);        // send a CR

// for (i=0; i<3; i++) {            // should get back: '*** bad command ***
//    c =(ReceiveCharPolled(LINKPORT) & 0x00FF);
//    if (c != '*') return 0;
// }

// RTKClearMailbox(ReceiveBuffer);  // forget the rest of the message
   return 1;
}

//
// ---------------------------------------------------------------------------
// NAME        :  InitHC11
// DESCRIPTION :  Downloads the vehicle control program for the HC11 and starts
//                it.
//
void InitHC11()
{
   char cmd[20];
   FILE *fd;
   int flen;

   // load HC11 prog in buffer (it is small enough to load all at once)
   fd = fopen("carctrl.s19","rb");
   flen = fread(Buffer,1,8192,fd);
   fclose(fd);

   // send transfer command to HC11
   sprintf(cmd,"%s%c%c","td",0x0D,0x0A);  // 'td' is transfer cmd of GBUG
   SendBlock(LINKPORT,cmd,strlen(cmd));

   // send the program
   SendBlock(LINKPORT,Buffer,flen);       // Buffer contains S19 file

   // run it
   sprintf(cmd,"%s%c%c","r 2000",0x0D,0x0A);  // GBUG loads program at 0x2000H
   SendBlock(LINKPORT,cmd,strlen(cmd));

}
#endif


// TASKS
// ===========================================================================
```

```
// Task Prototypes
// ------------------------------------------------------------------------
void FlipAutoMan(void);
void AutoIn(void);
void ManualIn(void);
void Steer(void);
void Buttons(void);
void Receive(void);
void WaitQuit(void);
void WrStat(void);
#ifdef TRACE_DUMP
void DoTraceDump(void);
#endif

//
// ------------------------------------------------------------------------
// NAME        :  FlipAutoMan
// DESCRIPTION :  Starts tasks for automatic & manual mode and flips between
//                these modes when requested
//
void FlipAutoMan(void)
{
   unsigned AutoMsg = AUTO;
   unsigned ManMsg = MANUAL;

   ManualTH = RTKCreateTask(ManualIn, STEERPRIO,
                            DEFAULTSTACK, "Manual Steering");
   AutoTH   = RTKCreateTask(AutoIn, STEERPRIO,
                            4*DEFAULTSTACK, "Auto Steering"); // needs big stack!

   RTKSuspend(AutoTH);  // start up in manual mode
   Mode = Manual;

   while (True) {
         RTKWait(FlipModeS);     // wait for flip request

         RTKPut(StatusMB,&AutoMsg);
#ifdef DATA_LOG
         if (!doRecord) RTKSuspend(ManualTH);   // change to automatic
                                                // with (manual still active)
#endif
#ifndef DATA_LOG
         RTKSuspend(ManualTH);   // change to automatic
#endif
         RTKResume(AutoTH);

         RTKWait(FlipModeS);     // wait again for flip request

         RTKPut(StatusMB,&ManMsg);
         RTKSuspend(AutoTH);     // change to manual
         RTKResume(ManualTH);
   }
}

//
// ------------------------------------------------------------------------
// This task is responsible for the automatic steering using
// the data from the frame grabber
//
void AutoIn(void)
{
```

```
#ifdef PROFILE
   short fg_data[384];  // array to store the data from frame grabber
   int i,max;
   long error0 = 0,error1 = 0,error2 = 0,error3 = 0;
#endif
#ifdef VANISH
   FF_TEMPLATE lower,upper;
   float head,ye,theta;
   float h,f,X_Hor,X_Base,OP,OP2,x,a,d,m;
   int x0,y0;
   int x1,y1,x2,y2;
#endif

   int steer,speed;
   int old_steer = 0, old_speed = 0;
   unsigned char SteerMsg = STEER;
   unsigned char SpeedMsg = SPEED;

#ifdef DATA_LOG
   int time,oldtime,deltatime;
   long int lasttime;
   char line[40];
#endif
#ifdef PROFILE
   int error;

   int k_num = -8;         // best P-constants for intensity profile P-control
   int k_denom = 10;
#endif
#if defined(VANISH) && !defined(HINF)
   int k1 = 270;           // best P-constanst for vanishing point P-control
   int k2 = 430;           // modifiy at your own risk !!!!
#endif

#ifdef FULL_INFO
   unsigned char PINTMsg = PINT;
#endif

#if defined(HINF) && !defined(DATA_LOG)
   int n;
   double steerangle;
   // The states for the HInf controller
   double xs[7],xsn[7];
#endif

   // make sure that coprocessor is protected too!
   RTKProtect8087();

   // setup the frame grabber
   ff_initialize();

#ifdef PROFILE
   // clear data in buffer
   for (i=0; i<384; i++)
      fg_data[i] = -32767;
#endif

#ifdef VANISH
   ff_set_rs170_quarter_format();           // set 128x128 resolution

   ff_init_template(&lower);                // initialize template for
```

```
   ff_read_template(&lower,"lower.tpl");     // middle marker in lower part
                                             // of the image plane

   ff_init_template(&upper);                 // initialize template for
   ff_read_template(&upper,"upper.tpl");     // middle marker in upper part
                                             // of the image plane
   // define horizon on image
   x0 = 64; y0 = 60;
   h = 0.16; // (height of camera in meter)
   d = 0.43; // 14.75 (distance from baseline to C.G. of the vehicle in meter)
   theta = 0.5071; // 30 deg  (visual angle)

   f = 64./tan(theta); // (focal distance of camera in pixel)
   X_Hor = h*f; // (distance from camera to Horizon (sort of))
   X_Base = h*f/(128.-y0); // distance from camera to base line (lower edge of image)
   OP2 = X_Hor/f * x0; // half the length of vanishing line

   y1 = y0+40; y2 = 120;
#endif

#if defined(HINF) && !defined(DATA_LOG)
   // clear data in states
   for (n=0; n<7; n++)
      xs[n] = 0.;
   memcpy(xsn,xs,7*sizeof(double));
#endif

#ifdef DATA_LOG
   if (doPlay) {
      // read new line with time, steering and speed
      if (fgets(line,40,PlayFileH) == NULL)
         RTKSignal(TerminateS); // stop program when finished

      // interpret line
      sscanf(line,"%ld %d %d",&time,&steer,&speed);

      lasttime = RTKGetTime();
   }
#endif

   ff_snap_nowait(0,0);       // snap image without waiting

   while (True) {
      SteerMsg = STEER; SpeedMsg = SPEED;

      ff_wait_while_busy();   // wait for snap to finish

#ifdef PROFILE
      // calculate profile on frame grabbe while Windows does other things
      ff_profile_x_nowait(63,411,384,100);

      while (!ff_get_flag())
         RTKDelay(0);            // wait the nice way

      // Get the (hopefully) completed data from FG
      ff_profile_x_complete(fg_data);

      ff_snap_nowait(0,0);       // snap new image without waiting

      error3 = error2;           // store last 4 error values
      error2 = error1;
```

```
      error1 = error0;

      // find maximum
      max = -32767;
      for (i=0; i<384; i++)
         // if new maximum, calculate the error
         if (fg_data[i] > max) {
               max = fg_data[i];
               error0 = 256 - (i + 64);
         }

      error = (int)(error0+error1+error2+error3)/4;  // average of last 4 errors
#endif

#ifdef VANISH
      // find white line in upper part of road (middle of image)
      ff_search_for_template_nowait(&upper,9,y1,110,1,1,1,0,1,NULL);
      while (!ff_get_flag()) RTKDelay(0); // switch to other tasks
      ff_search_for_template_complete(&upper);

      // find white line in lower part of road
      ff_search_for_template_nowait(&lower,9,y2,110,1,1,1,0,1,NULL);
      while (!ff_get_flag()) RTKDelay(0); // switch to other tasks
      ff_search_for_template_complete(&lower);

      ff_snap_nowait(0,0);       // snap new image without waiting

      // calculate middle of white line
      x1 = (float)upper.result.x; y1 = upper.result.y;
      x2 = (float)lower.result.x; y2 = lower.result.y;

      // calculate incline?
      m = (float)(x1-x2)/(float)(y2-y1);

      // equation of line (x at vanishing line)
      x = m*(y2-y0)+x2;
      OP = X_Hor / f * (x-x0); // don't confuse the X's

      // calculate heading angle
      head = atan(OP * tan (theta) / OP2);

      // equation of line (x at base line)
      x = m*(y2-128.)+x2;
      a = X_Base / f * (x-x0); // don't confuse the X's

      // calculate lateral deviation
      ye = a*cos(head) - d*sin(head);
#endif

#ifdef DATA_LOG
      if ((doRecord) || (doPlay)) // record heading angle and lat. deviation
         fprintf(RecFileH,"%ld %f %f\n",RTKGetTime(),head,ye);
#endif

#ifndef DATA_LOG
      // The actual control law
#ifdef VANISH
#ifndef HINF
      // the simple P-control law
      steer = 128 + k1 * head+ k2*ye;
      if (steer > 255) steer = 255;
```

```
      if ( steer < 0) steer = 0;
#endif
#ifdef HINF
      // the incredible fancy HInf-control law
      // calculate new states from old states and inputs
      for (n=0; n<7; n++)
         xsn[n] = Ak[n][0]*xs[0]+Ak[n][1]*xs[1]+Ak[n][2]*xs[2]+Ak[n][3]*xs[3]+
                  Ak[n][4]*xs[4]+Ak[n][5]*xs[5]+Ak[n][6]*xs[6]+
                  Bk[n][0]*head+(-Bk[n][1]*ye);

      memcpy(xs,xsn,7*sizeof(double));    // make modified states new states

      // finally calculate the steer commands
      steerangle = (Ck[0]*xsn[0]+Ck[1]*xsn[1]+Ck[2]*xsn[2]+Ck[3]*xsn[3]
                   +Ck[4]*xsn[4]+Ck[5]*xsn[5]+Ck[6]*xsn[6])*10; // *20;

      if (steerangle > 0.5236) steerangle = 0.5236;   // angle > PI/6 = 30 deg
      if (steerangle < -0.5236) steerangle = -0.5236; // angle < -PI/6 = -30 deg
      steer = 128 + floor((steerangle/0.5236)*128.);
#endif // HINF
#endif // VANISH

#ifdef PROFILE
      // simple P-control law
      steer = 128 + k_num * error / k_denom;
#endif

      speed = 18;             // fixed speed in automated driving mode
      printf("steer: %u speed: %u\n",steer,speed);
#endif // !DATALOG

#ifdef FULL_INFO
      if (RTKPutCond(StatusMB, &PINTMsg)) {     // If space in Mailbox
         strcpy(WrStat_IntName,"ERR");          // write out the control error
         WrStat_Int = error;
      }
#endif

#ifdef DATA_LOG
      if (!doRecord) {        // in Data-Log record mode the driving commands
#endif                        // come from ManualIn task

         if ((steer < old_steer - 10) || (steer > old_steer + 10)) {
            if (RTKPutCond(SteerMB,&SteerMsg))   // Write steer command
               RTKPut(SteerMB,&steer);
            old_steer = steer;
         }
         if ((speed < old_speed - 5) || (speed > old_speed + 5)) {
            if (RTKPutCond(SteerMB,&SpeedMsg))     // Write speed command
               RTKPut(SteerMB,&speed);
            old_speed = speed;
         }
#ifdef DATA_LOG
      }
#endif
      RTKDelay(1);  // !!!!!!!!! CHECK DELAY HERE !!!!!!!!!!!!!!!!

#ifdef DATA_LOG
      if (doPlay) {  // read driving commands from <file1> ?
         // store previous point in time
         oldtime = time;
```

```
         // read new line with time, steering and speed
         if (fgets(line,40,PlayFileH) == NULL) {
            RTKSignal(TerminateS); // stop program when finished
            return;
         }
         // interpret line
         sscanf(line,"%ld %d %d",&time,&steer,&speed);

         deltatime = (int)(RTKGetTime() - lasttime);
         printf("steer: %d  speed: %d  delta: %d\n",steer,speed,time-oldtime-deltatime);
         if ((time - oldtime - deltatime) > 0)
            RTKDelay(time - oldtime - deltatime);
         else RTKDelay(0);

         lasttime = RTKGetTime();

      }
#endif
   }
}


//
// ---------------------------------------------------------------------
// NAME        : PollJoyStick
// DESCRIPTION : Poll one joystick movement (X, Y on joystick 1 & 2)
//                !!! NOTE : The waiting loop in this function for       !!!
//                !!!        the joystick port will change from computer !!!
//                !!!        to computer (trial & error)                 !!!
//
void PollJoyStick(int Bit, int *count)
{
   int i;

   *count = 0;
   RTKDisableInterrupts(); // don't let any interrupt take place during this

   outportb(0x201, 0x00);  // start counter of joystick ADC

   do {                    // Get joystick position (steering wheel)
      (*count)++;

      for (i=0;i<15;i++)   // this is very stupid way to waste some time
         asm { nop } ;     // but its the best in this case

   } while (inportb(0x201) & Bit);

   RTKEnableInterrupts();  // interrupts are okay again

   delay(5);               // wait to settle port down
}

//
// ---------------------------------------------------------------------------
// NAME        :  ManualIn
// DESCRIPTION :  This task handles the reading from the Joystick-port
//                (steering wheel, gas pedal & gear shift);
//
void ManualIn(void)
{
```

```
   int x1,x10,y1,y10,x2,x20;

   unsigned char SteerMsg = STEER;
   unsigned char SpeedMsg = SPEED;

   unsigned char steer;
   unsigned char speed;
   unsigned char old_speed = 0;
   unsigned char zero_speed = 0;
   unsigned char old_steer = 0;

#ifdef FULL_INFO
   unsigned char PINTMsg = PINT;
#endif

   x10=y10=x20=0;

   outportb(0x201,0x00);
   RTKDelay(2);            // let things settle down

   PollJoyStick(0x01,&x10);  // read the 'zero' postions of steering console
   RTKDelay(1);
   PollJoyStick(0x02,&y10);
   RTKDelay(1);
   PollJoyStick(0x04,&x20);
   RTKDelay(1);

   while (True) {

      x1=y1=x2=0;

      PollJoyStick(0x01,&x1); // poll steering wheel (clockswise)
      RTKDelay(1);
      PollJoyStick(0x02,&y1); // poll steering wheel (counter-clockwise)
      RTKDelay(1);
      PollJoyStick(0x04,&x2); // poll gas pedal
      RTKDelay(1);

      // calc steering angle (range -127 to 127 to 0...255)
      if (((x1-x10)+(y10-y1)) > 127)
         steer = 255;
      else if (((x1-x10)+(y10-y1)) < -127)
         steer = 0;
      else
         steer = 128 + (x1-x10)+(y10-y1);

#ifdef FULL_INFO
      if (RTKPutCond(StatusMB, &PINTMsg)) {     // If space in Mailbox
         strcpy(WrStat_IntName,"STR");          // write out the desired steer angle
         WrStat_Int = steer;
      }
#endif
      // if no significant change in steering angle, do not send message!
      if ((steer < old_steer - 10) || (steer > old_steer + 10)) {
         if (RTKPutCond(SteerMB,&SteerMsg))   // Write steer command
            RTKPut(SteerMB,&steer);
         old_steer = steer;
      }

      if (x2 > x20) // speed above zero threshold
         speed = 0;
```

```
      else
         speed = (x20 - x2)*2;            // calc speed ( 0 to 255)

      // if no significant change in speed, do not send message
      if ((speed < old_speed - 5) || (speed > old_speed + 5)) {
         if (RTKPutCond(SteerMB,&SpeedMsg))     // Write speed command
            if (speed <= 10)  // speed < 10 have no effect on vehicle
               RTKPut(SteerMB,&zero_speed);
            else
               RTKPut(SteerMB,&speed);
            old_speed = speed;
      }

#ifdef DATA_LOG
      // store steering and speed commands
      if (doRecord) fprintf(PlayFileH,"%ld %u %u\n",
                            RTKGetTime(),steer,speed);
#endif
      // poll the throttle (gear shift)
      if (!(inportb(0x0201) & 0x10)) Direction = Forewards;
      if (!(inportb(0x0201) & 0x20)) Direction = Backwards;

      RTKDelay(1);

   }
}

//
// ---------------------------------------------------------------------------
// NAME        :  Steer
// DESCRIPTION :  This task sends the speed & steering commands to the vehicle.
//                This either happens through a RC or a wireless serial link.
//                !!! NOTE : The actual values send over the RC-link depend !!!
//                !!!        heavily on the loading status of the battery   !!!
//                !!!        They are subject to change over time           !!!
void Steer(void)
{
   unsigned char cmd;
   unsigned char cspeed;
   unsigned char csteer;

#ifdef RC_LINK
   float gas, steer;
   unsigned igas, isteer;
#endif
#ifdef SERIAL_LINK
   bool is_forward;
#endif

   // Initialize steering and speed

#ifdef RC_LINK
   cbFromEngUnits(0,BIP5VOLTS,FOFFSET,&igas); // voltage to integer
   cbAOut(0, 0, BIP5VOLTS, igas);             // no gas

   cbFromEngUnits(0,BIP5VOLTS,SOFFSET-SRANGE/2.,&isteer); // voltage to integer
   cbAOut(0, 1, BIP5VOLTS, isteer);           // steer straight ahead
#endif
#ifdef SERIAL_LINK
   SendChar(LINKPORT,SPEED);  // set speed to zero
   SendChar(LINKPORT,0);
```

```
   SendChar(LINKPORT,STEER);  // go straight ahead
   SendChar(LINKPORT,0);
   is_forward = True;
#endif

   Direction = Forewards;  // go foreward first

   while (True) {
#ifdef SERIAL_LINK
      if ((is_forward) && (Direction == Backwards))  // change to forward?
         SendChar(LINKPORT,'4');
      if ((!is_forward) && (Direction == Forewards)) // change to backwards?
         SendChar(LINKPORT,'3');
      is_forward = (Direction == Forewards);
#endif
      RTKGet(SteerMB, &cmd);
      if (cmd == SPEED) {     // set desired speed
         RTKGet(SteerMB, &cspeed);
#ifdef RC_LINK
         if (Direction == Forewards)
            gas = FOFFSET + (float)(cspeed)*FRANGE/255.; // proper range foreward
         else
            gas = BOFFSET - (float)(cspeed)*BRANGE/255.; // proper range backward
         cbFromEngUnits(0,BIP5VOLTS,gas,&igas);  // voltage to integer
         cbAOut(0, 1, BIP5VOLTS, igas);          // set new speed
         RTKDelay(1);
#endif
#ifdef SERIAL_LINK
         // transmitting a 19 locks up the HC11
         if ((cspeed != 19) && (LineStatus(LINKPORT) & TX_SHIFT_EMPTY)) {
            SendChar(LINKPORT,SPEED);     // send speed Cmd
            SendChar(LINKPORT,cspeed);    // and the value
         }
#endif
      }

      if (cmd == STEER) {     // set desired steering angle
         RTKGet(SteerMB, &csteer);
#ifdef RC_LINK
         steer = SOFFSET - (float)(csteer)*SRANGE/127.; // proper range for steer
         cbFromEngUnits(0,BIP5VOLTS,steer,&isteer);     // voltage to integer
         cbAOut(0, 0, BIP5VOLTS, isteer);               // set new steer angle
         RTKDelay(1);
#endif
#ifdef SERIAL_LINK
         // transmitting a 19 locks up the HC11
         if ((csteer != 19) && (LineStatus(LINKPORT) & TX_SHIFT_EMPTY)) {
            SendChar(LINKPORT,STEER);     // send steer Cmd
            SendChar(LINKPORT,csteer);    // and the value
         }
#endif
      }
      RTKDelay(0);      // make task switch if necessary, otherwise go ahead
   }
}

//
// ----------------------------------------------------------------------------
// NAME        :  Buttons
// DESCRIPTION :  Polls the two buttons on the steering wheel console
//
```

```
void Buttons(void)
{
   bool button1 = False, button2 = False;
   while (True) {
      RTKDelay(5);
      if (!(inportb(0x0201) & 0x40)) {  // upper button
         if (!button1) {
            button1=True;
            RTKSignal(FlipModeS);         // if pressed change modes
         }
      }
      else
         button1=False;
      if (!(inportb(0x0201) & 0x80)) {   // lower button
         if (!button2) {
            button2=True;
            RTKSignal(TerminateS);        // if pressed end program
         }
      }
      else
         button2=False;
   }
}

//
// ---------------------------------------------------------------------------
// NAME        :  Receive
// DESCRIPTION :  This task receives user input either via serial IO or
//                via the keyboard
//
void Receive(void)
{
   char ch;

#ifdef SERIAL_IO
   unsigned data;
#endif

   while (True) {
#ifdef LOCAL_IO
      ch = (RTGetCh() & 0x00FF);       // Get input from keyboard
#endif
#ifdef SERIAL_IO
      RTKGet(ReceiveBuffer[IOPORT], &data); // Get input from COM port
      ch = (data & 0x00FF);
#endif
      switch (ch) {
         case 'a' :
         case 'A' :  if (Mode != Automatic)  // change to automatic mode
                        RTKSignal(FlipModeS);
                     break;
#ifdef TRACE_DUMP
         case 'd' :
         case 'D' :  DoTraceDump();
                     break;
#endif
#ifdef SERIAL_IO
         case 'e' :
         case 'E' :  DoWriteStatus = False;  // stop writing status info
                     break;
         case 'i' :
```

```
         case 'I' :  DoWriteStatus = True;   // start writing status info
                     break;
#endif
         case 'm' :
         case 'M' :  if (Mode != Manual)     // change to manual mode
                        RTKSignal(FlipModeS);
                     break;
         case 'r' :
         case 'R' :  RTKSignal(TerminateS);  // end (restart) program
                     break;
#ifdef SERIAL_IO
         case 't' :
         case 'T' :  // transfer new CONTROL.EXE from Host-PC
                     while (!ReceiveProg("CONTROL.EXE"));
                     break;
         case 'x' :
         case 'X' :  // transfer new carctrl.s19 from Host-PC
                     while (!ReceiveProg("CARCTRL.S19"));
                     break;
#endif
      }
   }
}

//
// ---------------------------------------------------------------------------
// NAME        :  WaitQuit
// DESCRIPTION :  This task polls the keyboard to wait for a 'q' to
//                terminate the program
//
void WaitQuit(void)
{
   char ch;
   while (True) {
      ch = (RTGetCh() & 0x00FF);
      if ((ch=='q') || (ch=='Q')) RTKSignal(TerminateS);
   }
}

//
// ---------------------------------------------------------------------------
// NAME        :  WrStat
// DESCRIPTION :  This task writes status information either on the screen
//                or to the serial port
//
void WrStat(void)
{
   unsigned msg;
   char s[130];

   while (True) {
      RTKGet(StatusMB, &msg); // read message typ from mailbox
      switch (msg) {          // branch to proper message type
         case INIT:     strcpy(s,"Initialising System...");
                        break;
#ifdef SERIAL_LINK
         case CHECK:    strcpy(s,"Checking for HC11...");
                        break;
         case BOOT:     strcpy(s,"Starting up HC11...");
                        break;
#endif
```

```
         case START:    strcpy(s,"Controller started...");
                        break;
         case STOP:     strcpy(s,"Controller stopped...");
                        break;
         case MANUAL:   strcpy(s,"Manual mode activated...");
                        break;
         case AUTO:     strcpy(s,"Automatic mode activated...");
                        break;
         case PINT:     sprintf(s,"P%s:%d",WrStat_IntName,WrStat_Int);
                        break;
      }
#ifdef LOCAL_IO
      printf(s);              // output message on console
      printf("\n");
#endif
#ifdef SERIAL_IO
      if (DoWriteStatus) {    // output message to COM port
         SendBlock(IOPORT,s,strlen(s));
         SendChar(IOPORT,0x0D); // send CR
         SendChar(IOPORT,0x0A); // send LF
      }
#endif
   }
}

#ifdef TRACE_DUMP
//
// ------------------------------------------------------------------------
// NAME        :  DoTraceDump
// DESCRIPTION :  Dump the last 64 trace buffer entries in a file
//                (Program must be compiled with Debug Version of RTKernel)
//
void DoTraceDump(void)
{
   FILE *f;
   int i;

   RTKStopTracing();                // stop the trace
   f = fopen("TRACE.DAT", "wb");
   RTKTraceHeader(TraceBuffer);     // write header
   fputs(TraceBuffer, f);
   for (i = 63; i >= 0; i--)
   {
      RTKDumpTrace(TraceBuffer, i); // write trace
      fputs(TraceBuffer, f);
   }
   fclose(f);
   RTKTraceAll();                   // resume trace
#ifdef LOCAL_IO
   RTKExec("C:\LIST.COM","TRACE.DAT");
#endif
}
#endif

//
// ------------------------------------------------------------------------
// NAME        :  MAIN
// DESCRIPTION :  main task, initalizes realtime OS, starts other tasks and
//                then waits until it gets a Terminate Signal
//
#ifdef DATA_LOG
```

```
void main(int argc, char *argv[])
#endif
#ifndef DATA_LOG
void main(void)
#endif
{
   unsigned InitMsg = INIT;
   unsigned StartMsg = START;
   unsigned StopMsg = STOP;
#ifdef SERIAL_LINK
   unsigned CheckMsg = CHECK;
   unsigned BootMsg = BOOT;
#endif

#ifdef DATA_LOG
   // interpret command line in Data-Log mode
   if (argc > 0) {
      if (strcmp(argv[1],"record") == 0) {  // enable record mode
         doRecord = True;
         PlayFileH = fopen(argv[2],"w");
         RecFileH = fopen(argv[3],"w");
      }
      if (strcmp(argv[1],"play") == 0) {    // enable playback mode
         doPlay = True;
         PlayFileH = fopen(argv[2],"r");
         RecFileH = fopen(argv[3],"w");
      }
   }
#endif

   // Initialize RTKernel and its seperate modules
   RTKernelInit(MAINPRIO);
   ITimerInit();
#if defined(SERIAL_IO) || defined(SERIAL_LINK)
   RTComInit();
#endif
   RTKeybrdInit();

#ifdef SERIAL_IO
   // Set up IO-Port
   InitIOPort();
#endif

   // Create semaphores and mailbox for task synchronization
   FlipModeS = RTKCreateSemaphore(Binary, 0, "FlipMode Semaphore");
   TerminateS = RTKCreateSemaphore(Binary, 0, "END Semaphore");
   StatusMB = RTKCreateMailbox(sizeof(int),1,"Status Info Mailbox");
   SteerMB = RTKCreateMailbox(sizeof(char),4,"Steer Command Mailbox");

   // Start the Status and Quit Task
   WaitQuitTH  = RTKCreateTask(WaitQuit, INPUTPRIO,
                               DEFAULTSTACK, "Quit key");
   WrStatTH    = RTKCreateTask(WrStat, OUTPUTPRIO,
                               DEFAULTSTACK, "Status info");

   RTKPut(StatusMB,&InitMsg);

#ifdef SERIAL_LINK
   // Set up and test LINK-Port

   while (1) {
```

```
      InitLINKPort();

      RTKPut(StatusMB,&CheckMsg);

      if (IsHC11There()) break;
      Error("Make sure the vehicle is turned on !!!");
   }

   RTKPut(StatusMB,&BootMsg);

   InitHC11();
#endif

   // Set interval for timer interrupt to 25ms
   SetTimerIntVal(25000);

   // Create all tasks with  their respective priority
   FlipAutoManTH = RTKCreateTask(FlipAutoMan, SWITCHPRIO,
                                 DEFAULTSTACK, "Mode Switch");
   SteerTH     = RTKCreateTask(Steer, STEERPRIO,
                                 DEFAULTSTACK, "Steering Output");
   ButtonsTH   = RTKCreateTask(Buttons, INPUTPRIO,
                               DEFAULTSTACK, "Poll console");
   ReceiveTH   = RTKCreateTask(Receive, INPUTPRIO,
                               DEFAULTSTACK, "Input Receiver");

   RTKPut(StatusMB,&StartMsg);

   RTKWait(TerminateS); // wait until everything is finished

   RTKPut(StatusMB,&StopMsg);

   // Terminate all tasks & everything else

   RTKDelay(20); // Give WrStat a chance to write out everything

   RTKTerminateTask(&FlipAutoManTH);
   RTKTerminateTask(&SteerTH);
   RTKTerminateTask(&ButtonsTH);
   RTKTerminateTask(&ReceiveTH);
   RTKTerminateTask(&WaitQuitTH);
   RTKTerminateTask(&WrStatTH);
   RTKDeleteMailbox(&StatusMB);

#ifdef DATA_LOG
   // Close all files for logging
   if (doRecord) fclose(RecFileH);
   if (doRecord) fclose(PlayFileH);
#endif
}
// ============================================================================
```

Vita

Name:	Nikolai Schlegel
Date of Birth:	September 30th 1970
Place of Birth:	Northeim, Germany
Parents:	Birgit and Kristian Schlegel
Current Address:	826 Orchard Street Blacksburg, VA 24060
Telephone:	(540) 953-0668
e-Mail:	nikolai@birch.ee.vt.edu
High School:	Gynmasium Corvinianum, Northeim, Germany. Graduated May 1990
Undergrad. Studies:	Technische Universität Braunschweig, Vordiplom in October 1993, Finished May 1995
Graduate Studies:	Virginia Polytechnic Institute & SU, Graduated with Master of Science in January 1997
Research Interests:	Modern Controls, Microcontrollers, VLSI Design
Published Papers:	Telerobotic Operation combined with Automated Vehicle Control, SPIE's Photonics East Symposium, Boston, November 1996