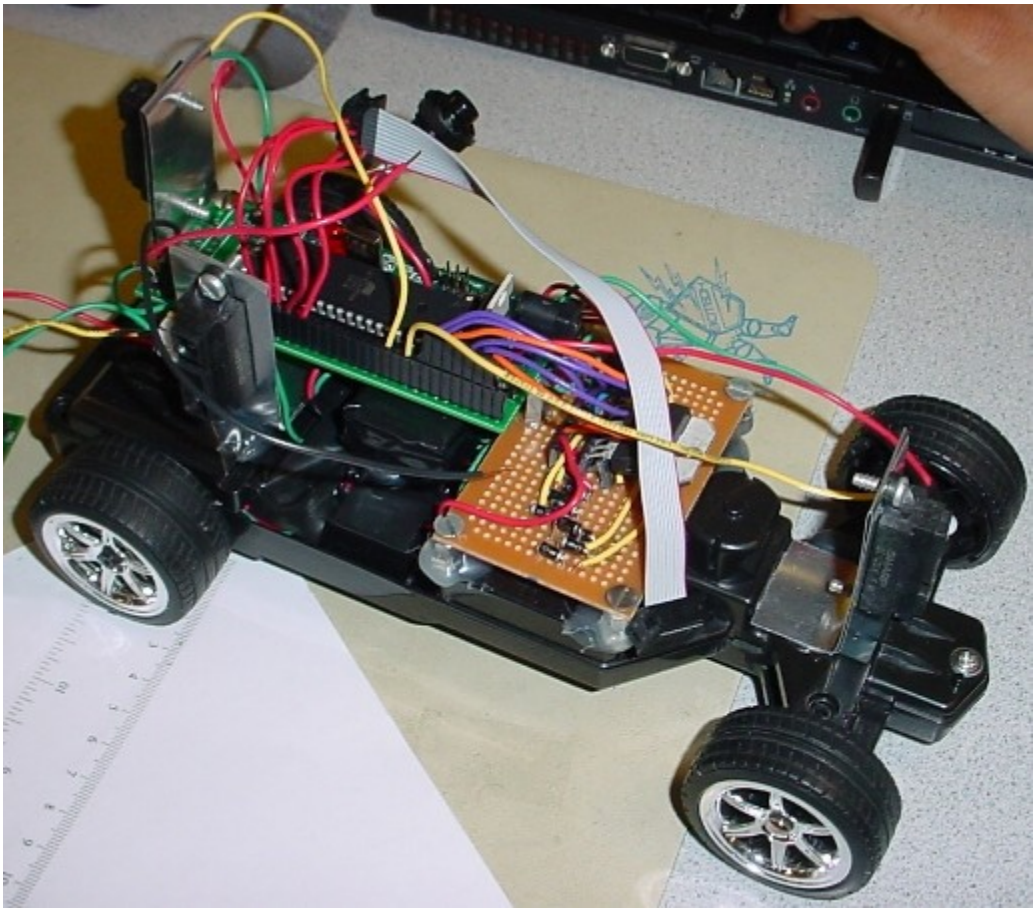
Autonomous Parallel Parking RC Car

- High Level Design
- Hardware
- Software
- Results
- Conclusion
- Appendix



Side View of RC Car

Arjun Nagappan (asn28)

Arjun Prakash (asp36)

Introduction

We created an RC Car that can identify a parking space and parallel park by itself. The RC Car drives down a street searching for a parking space to its right using a distance sensor. When the car has identified a space, the car checks to see whether that space is large enough to park in. If it determines that there is sufficient space, the car will begin parallel parking into that space. It uses information from sensors placed on the front, right, and rear of the car to direct the car into the parking space. Once the car has parked, it will remain in that position until it is reset.

High Level Design

Rationale | Logical Structure | Hardware/Software Tradeoffs

Rationale

After discussing various project ideas, we eventually stumbled onto the subject of cars. So we started brainstorming possible projects related driving. When brainstorming, we saw something in the ECE lounge that reminded us of a garage. This led us to parking. Parallel parking is something that many drivers struggle with, yet there are very few tools available to help with parallel parking. Though a few auto manufacturers have developed systems that can parallel park

cars autonomously, these solutions are very expensive. We thought this would be both a fun and interesting problem to tackle using an RC Car as a proxy for a real car.

Logical Structure

Our project is broken down into two major components: the control system and the move car algorithm. The move car algorithm directs the car and the control system implements the directions of the move car algorithm.

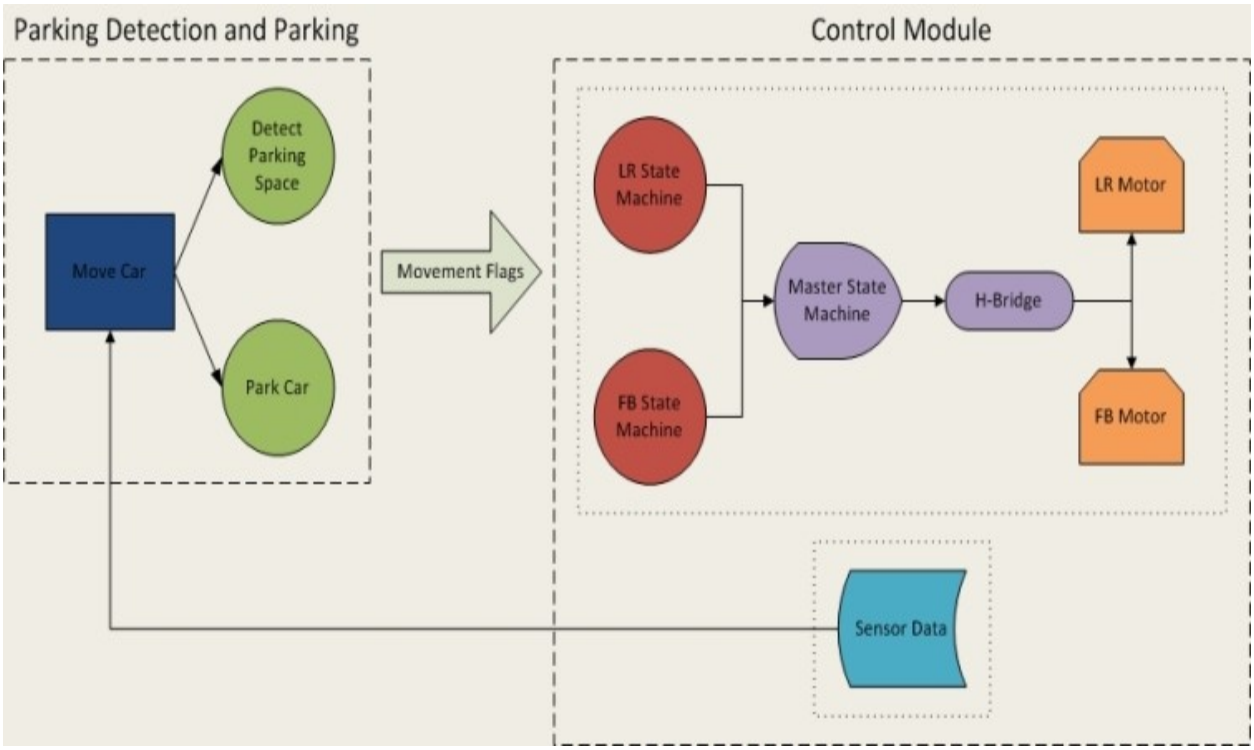


Figure 1: Logical Structure of High Level Design

Control System

The control system contains all the hardware and its associated software. It allows the parking and parking detection algorithms to interface with the car. The software in this module is broken up into three major sections: the Left-Right/Front-Back (LR/FB) state machines, master state machine, and distance calculations. The LR/FB state machines determines which direction to move the car based on flags set by the detect parking space and park car algorithms. Once the LR/FB state machines decides which direction to move the car, the master state machine implements this movement by sending the correct input and enable signals to the H-Bridge. The distance calculations implemented independently every millisecond.

Move Car

Move car contains the detect parking space and parallel parking algorithms. All functions in move car interface with the control module by setting movement flags. The parking space detection and parking algorithms use information from the distance sensors to set these movement flags and guide the car.

Move car works by initializing the movement flags of the car. It sets the car on a default trajectory and then calls detect parking space. Once a parking space has been detected, the parking algorithm is called. After the car has successfully parked, it idles until it is reset.

Hardware/Software Tradeoffs:

Distance Sensors

1. When selecting infrared distance sensors there was always a tradeoff between the sensors ability to measure close range and long range. We tried to minimize this problem by using sensors designed for varying ranges.
2. Using accurate sensors cost significant time. Every measurement from our distance sensors is approximately 40ms delayed. This affected our ability to start and stop the motors of the car at the correct times.
3. We used integer calculations rather than floating point to calculate distances. We decided the increased accuracy would not significantly improve our ability to park the car because we cannot control the movement of the car with that degree of accuracy.
4. Each sensor draws a maximum of 50mA. To accommodate for this, we needed to use a 5v regulator that could source up to 1A.

Batteries

1. We decided to power our car using batteries rather than using a steady power source. This gave us increased mobility but was very inconsistent in the current it supplied to the motors. As the batteries wore out, they supplied less and less current to the motors. This made calibrating the velocity of the car very difficult.
2. In order to best utilize the mobile power resources we have, we power the motors using four AA batteries, which are stored in the battery compartment of the RC car. These batteries supply the Supply Voltage to the H-bridge, which in turn powers the motor. We use a 9V battery to power the PCB.

Software

1. 1. Our code requires the motor control software, parking algorithm software, and distance sensor software to run in parallel. However, this is not possible in the Atmega644. We got around this issue by making every important task a state machine. By breaking up each function into small pieces, we can execute one piece of function one, then one piece of function two, followed by one piece of function3, and then another piece of function one, etc. This enables us to emulate a multi-tasking architecture.

Hardware

RC Car | H-Bridge | Distance Sensors

Hardware consists of three main components:

- RC Car
- H-Bridge
- Distance Sensors

All hardware used the following color convention:

Color	Connected To
Red	Vss
Green	Ground
Purple	Input
Yellow	Output
Orange	Enable

Table 1: Wire Color Convention

RC Car

The first step of our hardware design involved fully understanding the mechanics our RC car. We took apart the car and rebuilt it multiple times to fully understand how it was built, what every part in the car is used for, and how those parts contribute to the control of the car.

After understanding the mechanics of the car, we decided the easiest way to control our car would be to directly control the inputs to the DC brush motors controlling the front and rear wheels, bypassing all of the car's internal circuitry. To do this, we scoped the control signals of the car. We found that the control signals were very simple. There is one motor for the reverse and forward movement of the rear wheels and one motor to turn the front wheels left and right. These motors are controlled by a simple 5V DC input. A +5V turns the rear wheels forward and the front wheel to the left. A -5V input turns the rear wheels backwards and turns the front wheels to the right. To more easily control the motors we soldered wires to their plus and minus terminals. This allows us to easily apply a +/-5V without opening up the car again.

H-Bridge

We use an ST Micro L298HN H-Bridge to control the motors of the RC Car. It allows us to switch between +/-5V across the motor. It also allows us to source the power from the batteries while using the processor to control the transistors in the H-Bridge. The control algorithm turns the appropriate transistors on/off, applying the proper voltage across the brush motor. The H-Bridge is connected using the following configuration:

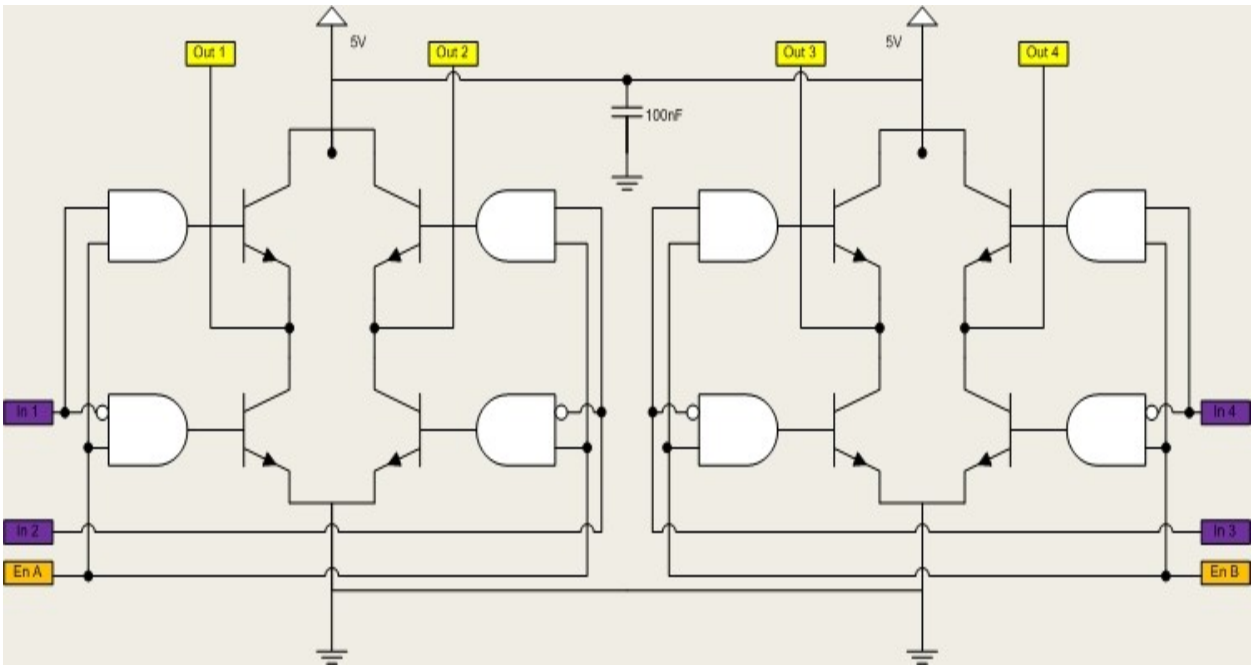


Figure 2: H-Bridge Schematic

The first H-Bridge (to the left) is used to control the front motor of the car. This motor turns the front wheels either left or right. The second H-Bridge (the the right) is used to control the rear motor, which is used for the forward and reverse functionality of the car. The inputs and enables of the H-Bridge are connected to port B.

Front Motor (Left/Right)	**Rear Motor (Forward/Reverse)**

Pin	**Connected To**	**Pin**	**Connected To**
In 1	Port B7	In 3	Port B3
In 2	Port B6	In 4	Port B2
En A	Port B5	En B	Port B1
Out 1	+ Motor Terminal	Out 3	+ Motor Terminal
Out 2	- Motor Terminal	Out 4	- Motor Terminal

Table 2: H-Bridge Pin Configuration

In addition configuring the H-Bridge to control the motors, we also had to protect the H-Bridge from inductive spikes caused by turning the DC brush motors on and off. We used diodes on the output to protect from these spikes. The H-Bridge was wired as follows:

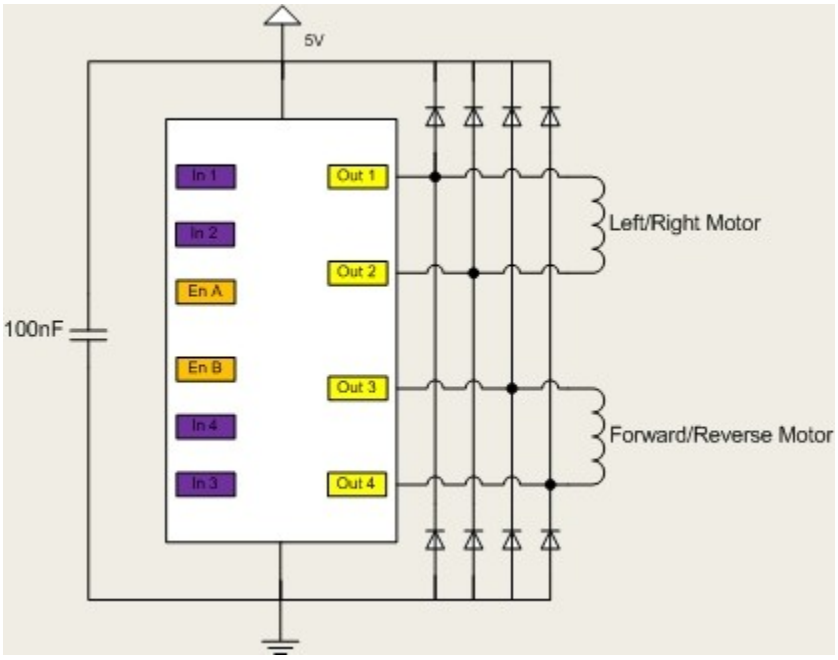


Figure 3: Inductive Current Protection on H-Bridge Outputs

Distance Sensors

We used three Sharp infrared distance sensors to determine the distance between our car and nearby objects. We placed a sensor on the front, the right side, and the rear of the car. For the front and rear, we used 4-30cm sensors. For the right side, we used we used a 10-80cm sensor. We decided to use a sensor with a larger range for the side so that we could more easily detect a parking space. However, this made aligning the parking the car more difficult, so we rely more heavily on the front and rear sensors to park the car. To slightly improve the short distance range of our sensors, we placed the sensors as far back on the car as possible.

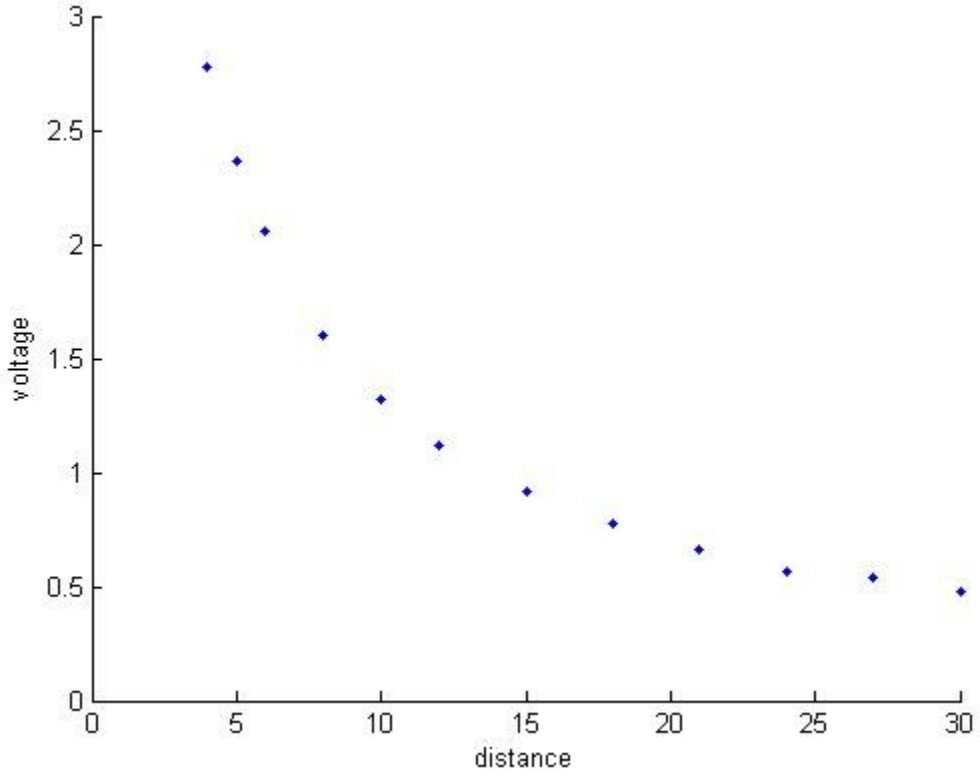
The challenge with using these sensors is that their voltage output is nonlinear (inverse function) and each sensor varies slightly. Therefore, we scoped the output of each sensor at various distance values, linearized the plot, curve fit the line, and implemented an analog to digital conversion so that we had reliable distance values.

Measurements and Linearization

Distance (cm)	Front Sensor Output (V)	Rear Sensor Output (V)
4	2.78	2.6
5	2.36	2.22
6	2.06	1.92
8	1.6	1.52
10	1.32	1.26
12	1.12	1.08
15	.92	.88

Distance (cm)	Front Sensor Output (V)	Rear Sensor Output (V)
18	.776	.76
21	.664	.656
24	.567	.576
27	.536	.52
30	.476	.48

Table 3: Front and Rear Sensor Measurements



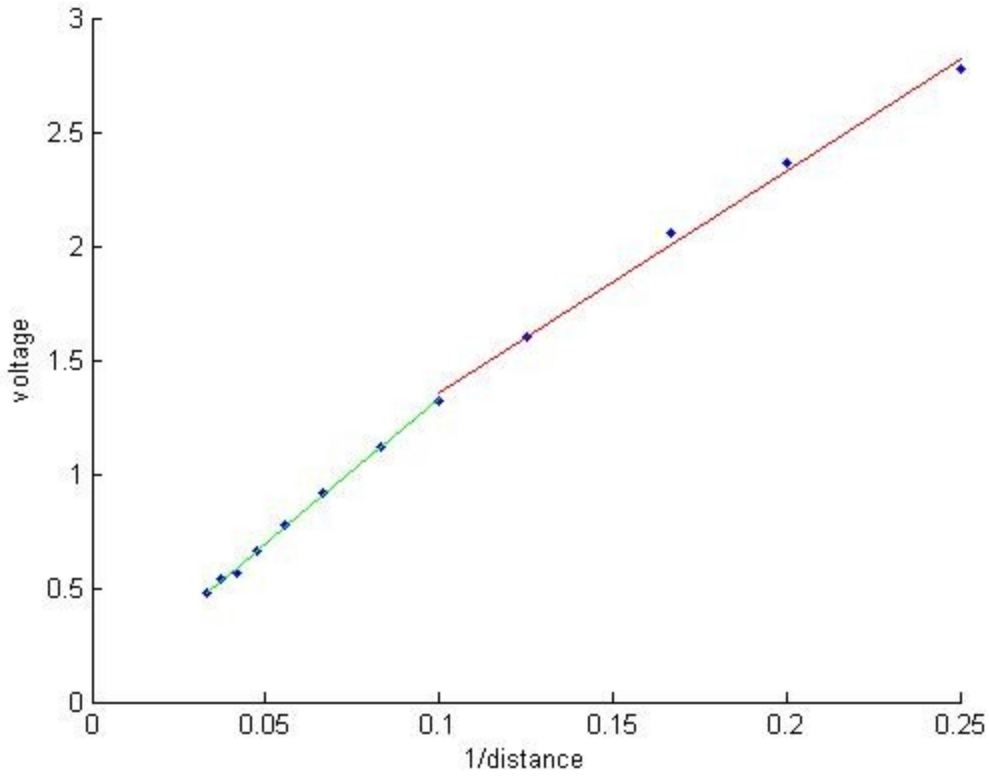
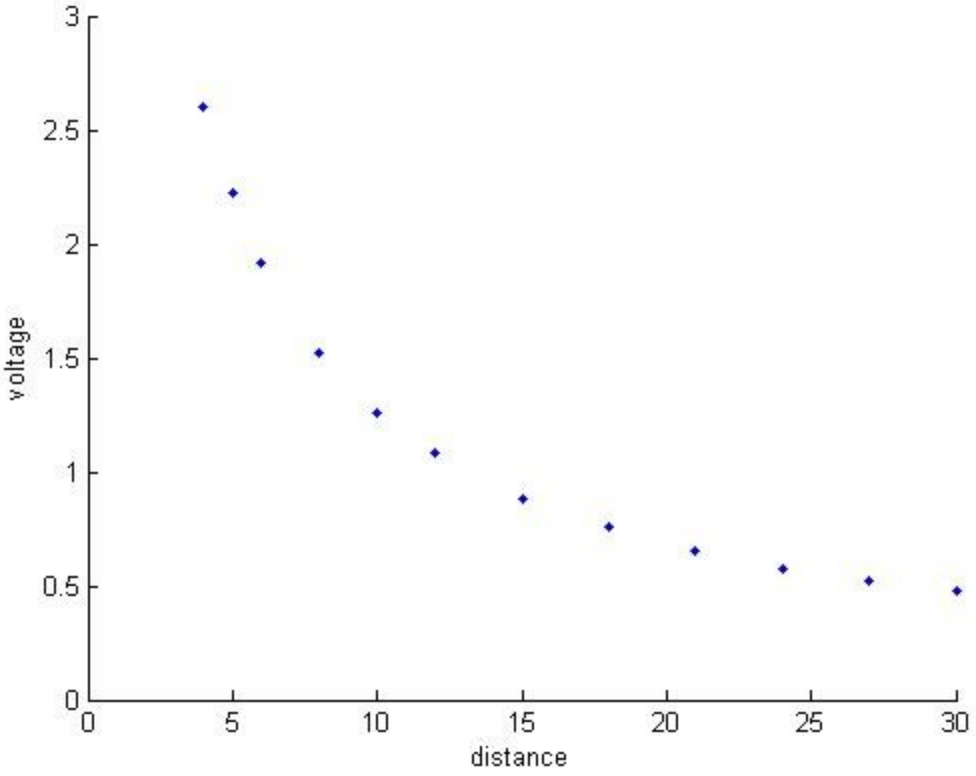


Figure 4: Front Sensor Plots of Distance vs. Voltage and Distance-1 vs. Voltage


3
2.5
2
1.5
1
0.5
0
voltage
0
5
10
15
20
25
30
distance

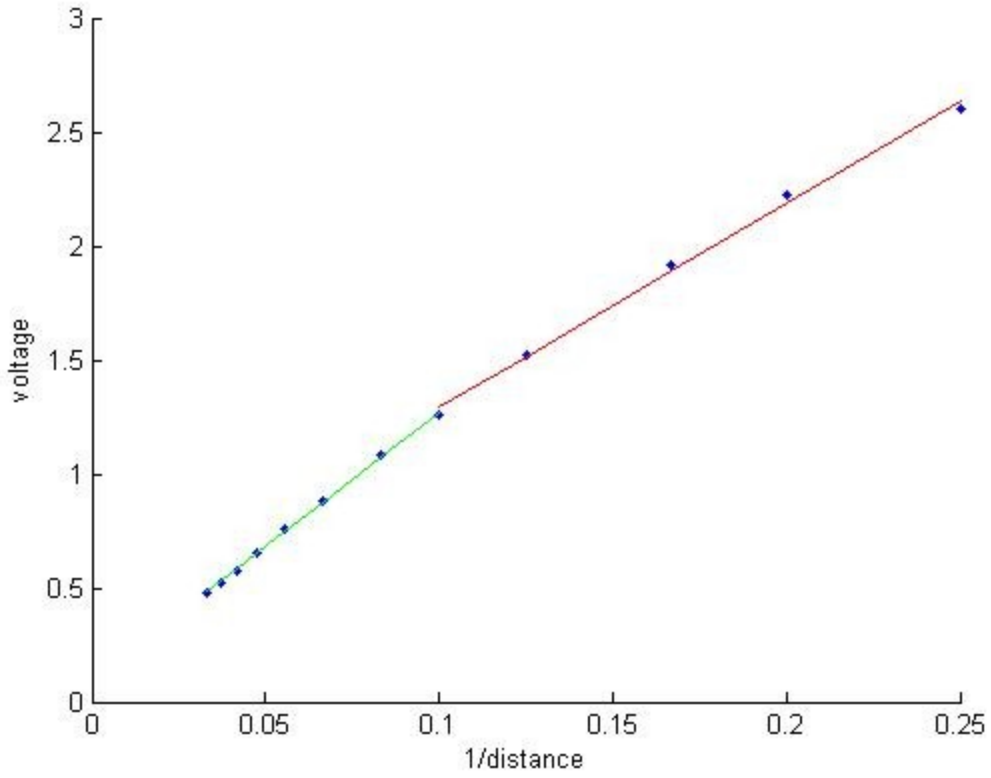


Figure 5: Rear Sensor Plots of Distance vs. Voltage and Distance-1 vs. Voltage

After taking the inverse of the distance versus voltage plots, we determined it would be best to fit the curve using two linear lines rather than one. The equations of the lines are:

Front Sensor	Rear Sensor
voltage=9.759*[1/distance]+0.381, >1.5V	voltage=8.963*[1/distance]+0.395, >1.25V
voltage=12.738*[1/distance]+0.057, <1.5V	voltage=11.806*[1/distance]+0.09, <1.25V

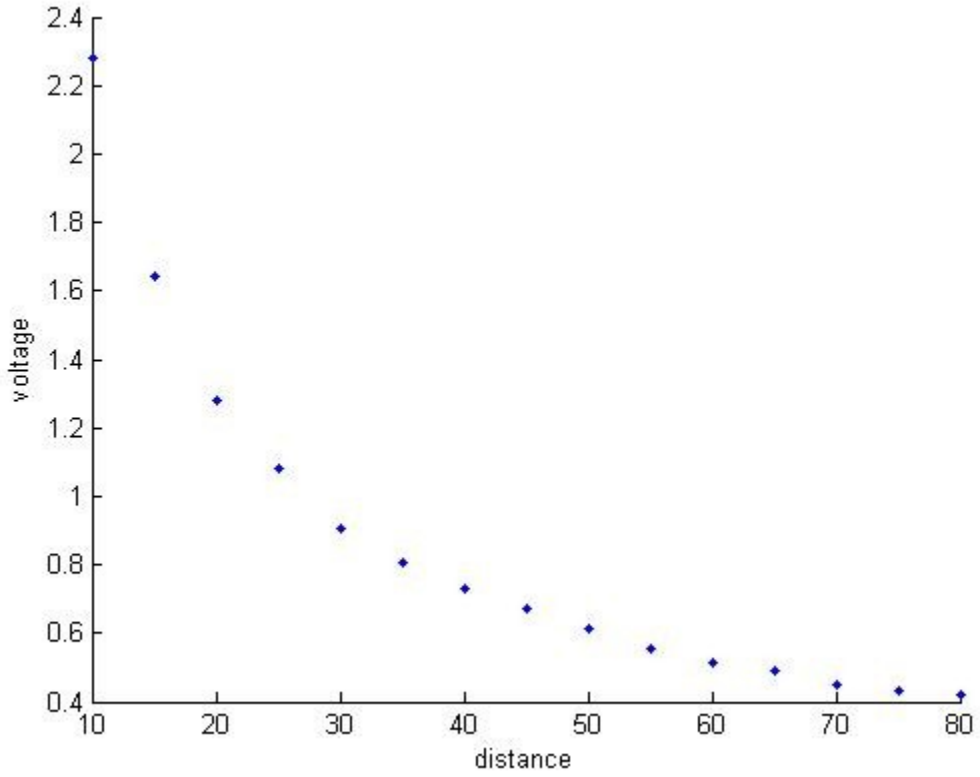
Table 4: Front and Rear Distance Sensor Equations

The analog to digital conversion uses a different equation depending on the value of the measure voltage. If the measured voltage is above a certain threshold, the ADC will use the equation in the top row. If it is below a certain threshold, the ADC will use the equation in the bottom row. The threshold for the front sensor is 1.5v and 1.25v for the rear sensor.

Distance (cm)	Side Sensor Output (V)
10	2.28
15	1.64
20	1.28
25	1.08

Distance (cm)	Side Sensor Output (V)
30	.808
35	.728
45	.672
50	.612
55	.556
60	.516
65	.488
70	.452
75	.432
80	.42

Table 5: Side Sensor Measurements



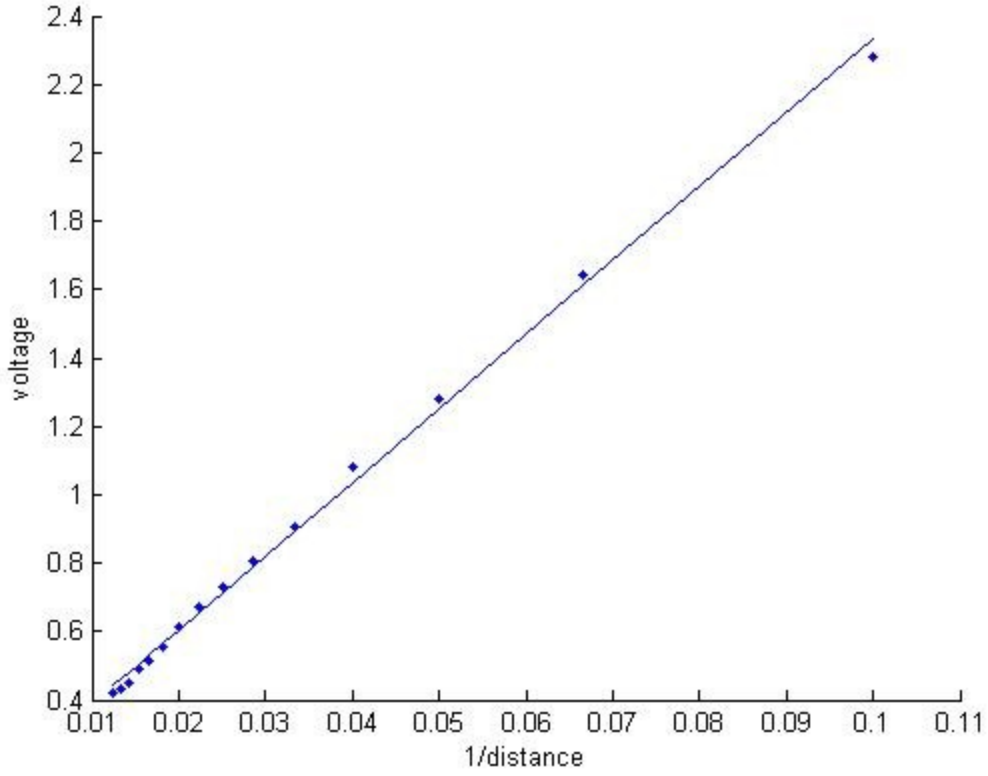


Figure 6: Rear Sensor Plots of Distance vs. Voltage and Distance-1 vs. Voltage

When looking at the data for the 10-80cm side sensor plot, we determined that one linear line would be sufficient to accurately capture the output voltage vs. distance characteristics of the sensor. The equation of this line is:

voltage=21.592*[distance]^(-1)+0.173

Analog to Digital Conversion

For the analog to digital conversion we used the built in ADC function of the MCU. Because we had three distance sensors, we had to rotate which sensor was connected to the ADC. This was simply done by changing the value of ADMUX. One of the three sensors is sampled every millisecond.

We take the value from the ADCH register and perform the appropriate calculation to convert this value to a distance. We decided to use only the ADCH register because the value represented by the bottom two bits of the ADC is on the same order of magnitude as noise. Therefore, these two bits are not important to our calculation. To do this, the ADC is left adjusted. By using the value in the ADCH register, we are using the upper eight bits of the ten ADC bits. We use the internal reference voltage of 2.56V for comparison, so our calculation is:

$$Distance^{-1} * slope + intercept = \frac{ADCH * V_{ref}}{\frac{1024}{4}}$$

$$Distance^{-1} * slope + intercept = \frac{ADCH * 2.56V}{\frac{1024}{4}} = \frac{ADCH}{100}$$

$$Distance^{-1} = \frac{\frac{ADCH}{100} - intercept}{slope}$$

$$Distance^{-1} = \frac{ADCH - intercept * 100}{slope * 100}$$

$$Distance = \frac{slope * 100}{ADCH - intercept * 100}$$

Software

Control Module | Algorithm Module

The software for this project has been partitioned into 2 files based on functionality. There are 2 files, ControlModule.c and AlgorithmModule.c.

The state machines in ControlModule control the motors, and are:

- fbStateMachine
- lrStateMachine
- masterStateMachine

The state machines in the AlgorithmModule use the sensor data and various algorithms to determine what should be the next movement the car must make. They assert flags which tell the ControlModule state machines to actually move the motors. The state machines in AlgorithmModule are:

- moveCar
- detectParking
- parkCar

Below we have explained the various state machines we have used in our project.

Control Module

fbStateMachine()

Function:

The fbStateMachine controls the motor for Forward-Backward operations. It is controlled by the isForward and isReverse flags. These flags serve as indicators as to whether the car should be traveling forward or reverse. In order to control the velocity of the forward-backward motion we anded the enable bit with a a PWM signal.

Working:

In State 0, the motor is at rest. The corresponding FB control bits are 00. When the algorithm requires the car to go forward or reverse, the corresponding flags (isForward and isReverse) are set, and the FB state machine switches states to 1 or 3 respectively.

In State 1, the motor rotates to drive the car forward. The state machine remains in this state while isForward is equal to 1. Once isForward is de-asserted, the state machine moves to a buffer state to stop the car from moving forward due to inertia.

After isForward is set to 0, leaving state 1 and stopping the motor isn't enough. The wheels might continue to rotate due to inertia, and so a buffer state, State 2, is required. It makes the motor go in Reverse for 1 cycle (50ms) of the FB State Machine, before going back to the rest state, State 0.

If isReverse is asserted, the state machine jumps to State 3. The state machine remains in this state while isReverse is equal to 1. Once isReverse is de-asserted, the state machine moves to a buffer state to stop the car from moving in reverse due to inertia.

After State 3, a buffer state, State 4, is needed to stop the wheels from continuing to rotate in reverse due to inertia. This is a 1 cycle Forward motion, similar in function to State 2's reverse functionality. Once done, the FB State Machine goes back to its rest state, State 0.

Timing:

The fbStateMachine is called upon every 50ms. This is enough time to evaluate the flags set in the AlgorithmModule, but at the same time fast enough to make the motor motion very accurate.

lrStateMachine()

The lrStateMachine() works the same way are the fbStatemachine. A forward corresponds to a left turn and a right corresponds to a reverse. The diagram for both are:

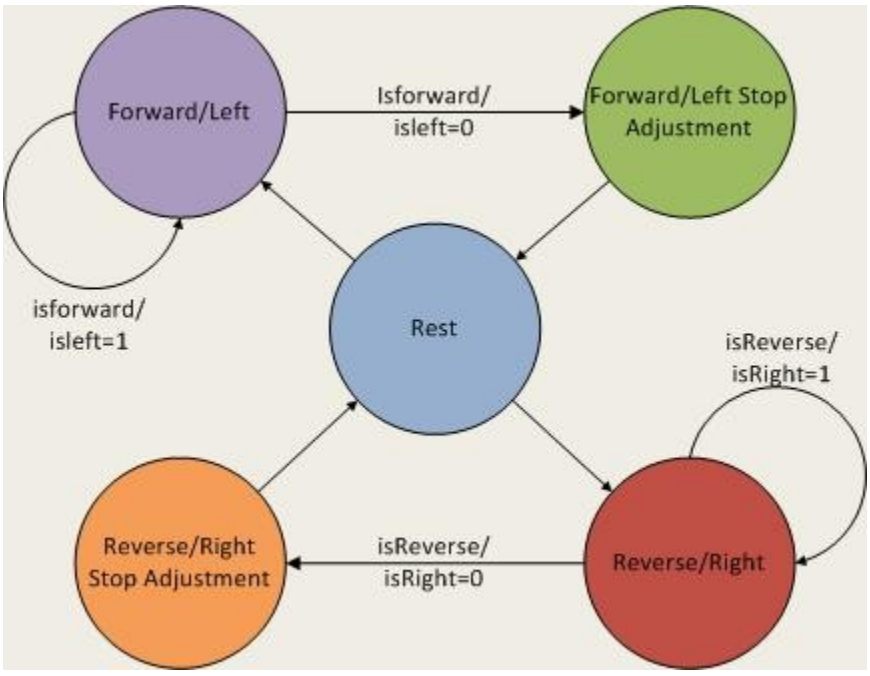


Figure 7: FB/LR Motor State Machine

masterStateMachine()

Function:

This uses the FB and LR control bits to call the required functions in order to send the appropriate input signals to the H-Bridge and make the motors rotate in the appropriate direction.

Working:

In this function, the 2 FB and LR control bits are combined to create 4 master control bits by left shifting the FW bits by 2 and adding it to the LR bits.

Therefore,
fbBits = fb.controlBits; // (FB FB)
lrBits = lr.controlBits; // (LR LR)
masterBits = (fbBits<<2) + (lrBits); // (FB FB)(LR LR)

As a result, each of the 7 car movements (stop, forward, forward-left, forward-right, reverse, reverse-left, reverse-right) have a unique masterBits combination associated with them. The master control bits are then used in the function to decide which motor control function is to be called.

Timing

This state machine is invoked in each iteration of the infinite while loop in main. In other words, it can be considered to be executing continuously and not at intervals. This is essential because parking requires a great deal of accuracy when controlling the motors. Therefore, we want to update the motors as often as possible, which would require us to call masterStateMachine as often as possible.

Algorithm Module

moveCar()

Function:

This is the master state machine of the algorithm module. It decides which mode the car is in, i.e., whether the car is moving forward to detect a parking spot, aligning itself once a parking spot has been detected, or actually going through the motion of parking.

Diagram:

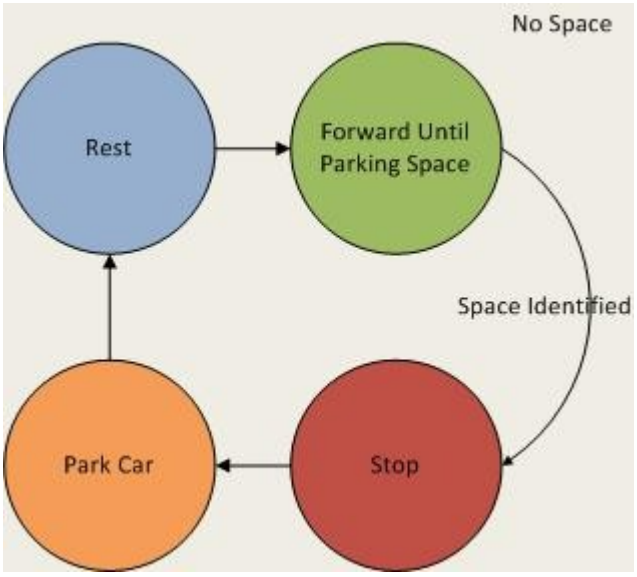


Figure 8: Move Car Motor State Machine

Working:

This is a 5 state linear state machine, as can be seen in the diagram above.

It starts off in State 0. In this state, the car is at rest. It gives enough time for all transients in the car to stabilize. Once everything is stable, it moves to State 1.

In State 1, car moves forward till it detects a parking spot. While in this state, the car invokes the detectParking state machine each time the moveCar state machine is called in the Control Module. Details of how the detectParking state machine works are explained in the next section.

Once a parking lot has been detected, the state machine moves into State 2. It remains in State 2 until the car has parked itself. The parkCar state machine is invoked for each cycle that the moveCar state machine is in State 2. Once the car has been parked by parkCar state machine, the isParked flag is asserted, and moveCar moves onto state 3.

When we reach State 3, the car parked itself. The car will eternally remain in this state hereafter, since the car has parked itself and is at rest.

In addition to serving as a state machine as described above, moveCar also makes available 2 values – rsDist and rrsDist – to its sub-state machines, detectParking and parkCar. rsDist stores the values of the side distance in the previous clock tick of the moveCar state machine, while rrsDist stores the value 2 clock cycles earlier.

Timing:

The moveCar state machine is invoked every 100ms. The moveCar state machine also serves as a clock for the detectParking and parkCar state machines. When in State 1, each clock tick of the moveCar state machine serves as a clock tick for the detectParking machine. When in State 3, each clock tick of the moveCar state machine serves as a clock tick for the parkCar machine.

detectParking

Function:

The function of detectParking state machine is, as its name suggests, to detect a parking space to park in. It accomplishes this by continuously polling the distance values from the side distance sensor.

Diagram:

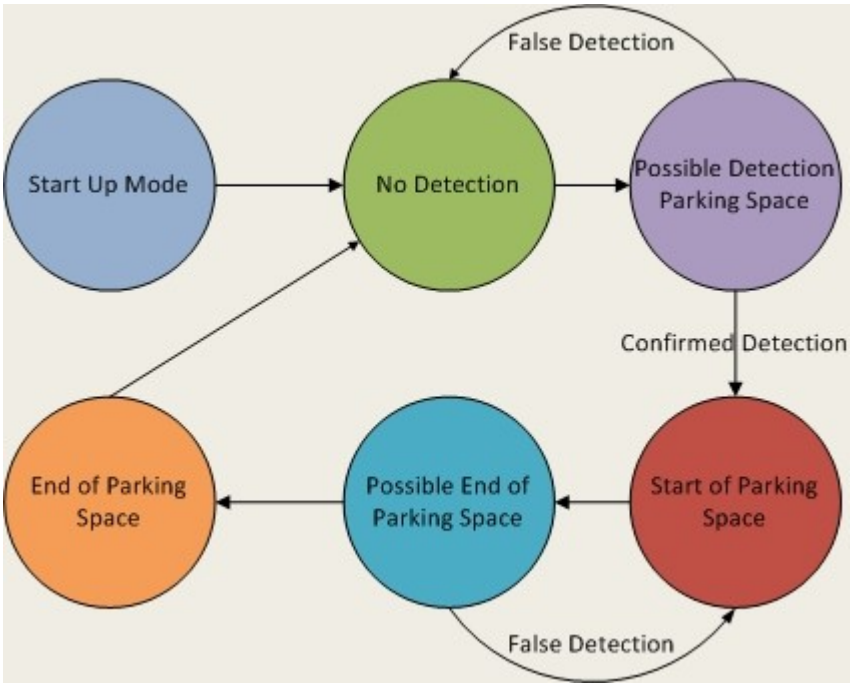


Figure 9: Detect Parking Space State Machine

Working:

detectParking is a 6 state state machine, as can be seen in the diagram above.

State 0 serves as a start-up. This is essential because the first few cycles of the detectParking take place while the side distance sensor is still calibrating itself. Once the wait state is done, the state machine enters state 1.

State 1, essentially, searches for a sudden increase in the side distance value. A sudden increase corresponds to the beginning of a parking space. It does this by checking the (sDistance – rsDist) value. If there is a sudden depression, sDistance will increase and so it's difference from its own previous value (rsDist) will be a large number. When this does occur, the state machine goes onto State 2.

In State 2 it attempts to confirm that it indeed is detecting a valid depression, by calculating (sDistance – rrsDist). Since State 2 is invoked 1 clock tick after the depression was last detected in State 1, rrsDist will store the value of the side distance before the depression began, i.e., from 2 clock cycles earlier. If (side distance – rrsDist) is still a large number, we can confirm that a depression has been detected, and we move to State 3.

In State 3, we keep track of how long the depression is. This is done by incrementing the detect.controlBits for each state machine clock tick that we are still in the depression. When there is a sudden decrease in the value of the side distance, we move to state 4, since it signals a probable end of the parking lot.

State 4 confirms that the possible end of the parking space, as detected in State 3, is indeed the end of the space. This is done in a manner similar to the confirmation done in State 2 using the rrsDist variable.

Once a parking space has been detected by the above states, the state machine moves into State 5 wherein it checks the control Bits (which kept track of how long the parking space was by

incrementing for each cock tick while in the depression) to make sure the parking space is large enough. If large enough, then the isParkingLot flag is asserted which would direct moveCar to stop and start the parking sequence.

Timing

Each tick of the detectParking state machine corresponds to a tick of the moveCar function. When moveCar is in State 1, it calls detectParking on each of its ticks. Therefore, detectParking is called every 100ms until a parking space has been located.

parkCar()

Function:

The function of the parkCar state machine is to park the car once a parking spot has been identified. The algorithm to park the car continuously interacts with its surroundings through the forward, side and rear sensors.

Diagram:

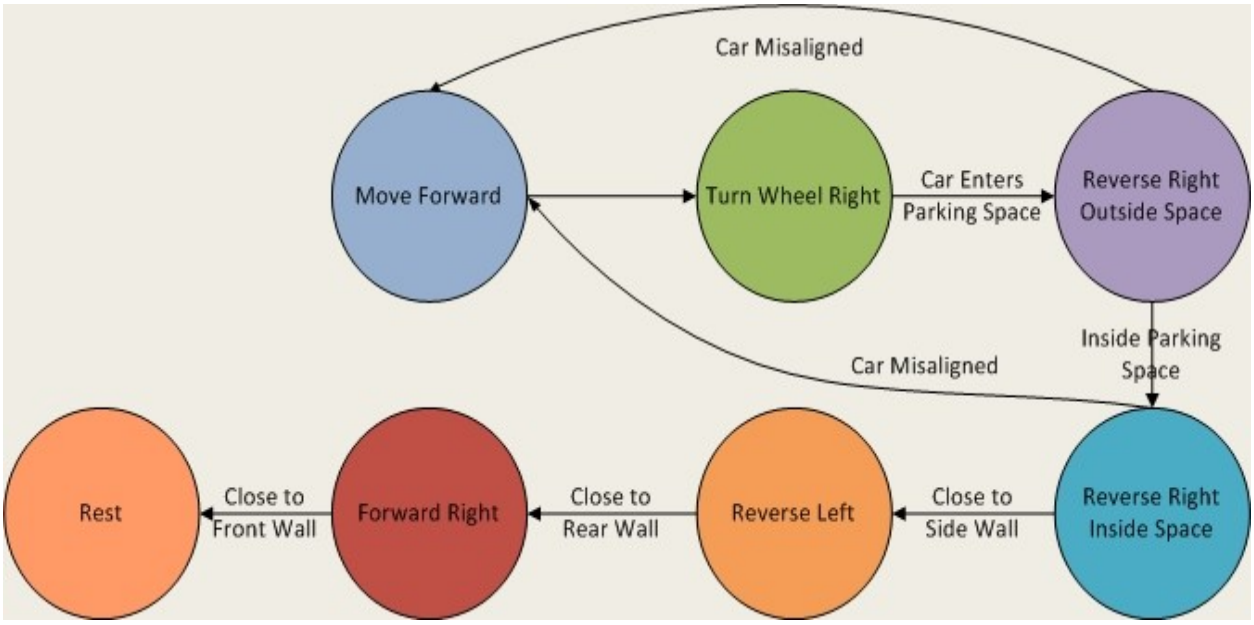


Figure 10: Park Car State Machine

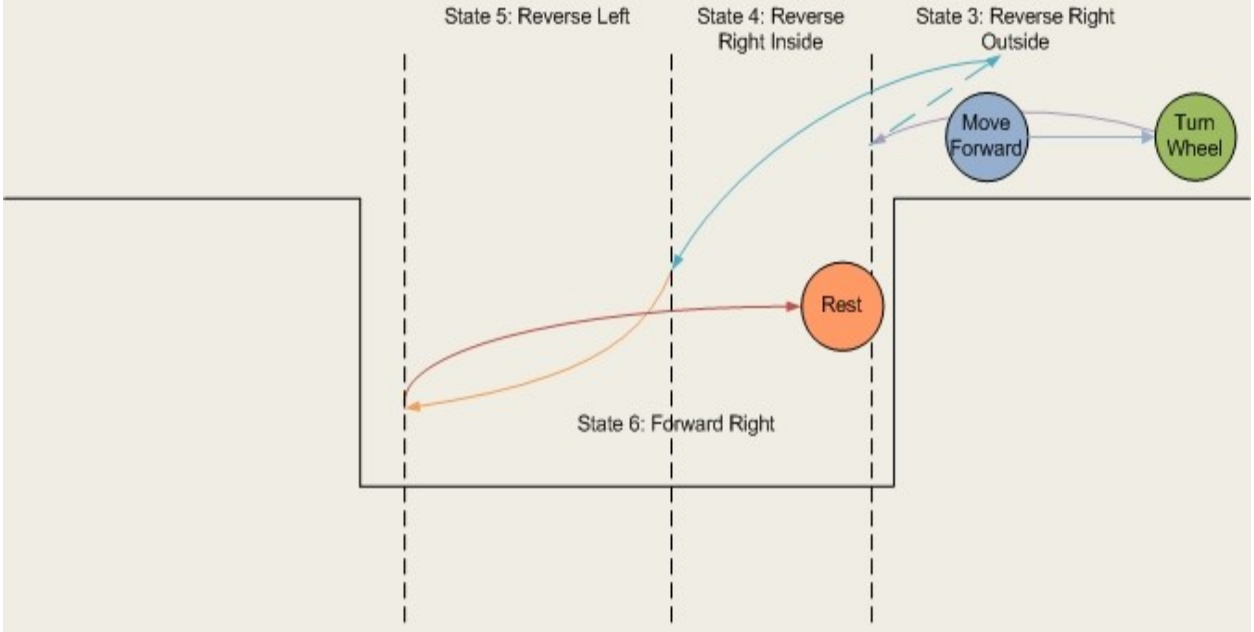


Figure 11: Parking Motion of Car, colors correspond to state of the Park Car State Machine

Working:

The parkCar function tries to simulate how a human would parallel park. It is, essentially, just the following 4 motions:

1. Reverse Right until you are inside the parking lot.
2. Go Forward and redo 1. if the car is not aligned.
3. Reverse Left until the car is fairly straight and close to the back wall.
4. Forward Right until the car is straight and close to the front wall.

The above routine is accomplished using a 7 state machine.

State 0 makes the car move forward by a certain amount. The idea is to give the car enough space to move and rotate into the parking space.

State 1 simply turns the front wheels to the right. We turn the wheel before reversing the car so as to not lose turning radius by turning as the car reverses. Once the wheel is turned, the state machine moves onto state 2.

State 2 commands the car to go reverse right for a specified amount of time until the car has passed the edge of the parking space. Once past the edge of the space, it moves to state 3.

In State 3, the car continues in reverse right until it is either a certain distance from inside of the parking space, or the rear distance is close to the edge. These conditions, as can be seen from the figure above, are checks to verify that the car is deep enough inside the parking lot to be able execute the reverse left maneuver. Once the conditions are met, the car stops and the state machine moves to state 4.

NOTE: If at any point in states 1, 2 or 3 the car's AI decides it is not in a position to go through with the parking, it will go back to State 0, and redo the whole procedure.

In State 4, the car moves reverse left. It does this until the rear of the car is close to the side wall of the parking space, which can be judged by the rear distance sensor value. Once close enough to the rear value, it stops and moves to state 5.

State 5 commands the car to go forward right. This attempts to straighten out the car completely and to align it nicely inside the spot. It goes forward right until it is close to the side wall of the parking space, as judged by the forward distance sensor. Once aligned, the car is parked and it moves to state 6.

State 6 is a 1 cycle stop before progressing back to state 0. Also, here the isParked variable is set so that the moveCar state machine can move out of parking mode to rest mode.

Timing:

Each tick of the parkCar state machine corresponds to a tick of the moveCar function. When moveCar is in State 3, it calls parkCar on each of its ticks. Therefore, parkCar is called very 100ms while the car is being parked.

Misalignment Detection

Our parking algorithm is equipped with a Misalignment Detector. Its role is to judge whether the car can park itself in the given space, and if it judges that parking is impossible, to correct the car's position to make it possible.

Our algorithm has a provision to keep track of the values of distance from the side sensor (sDIstance) from the earlier clock (rDist) and earlier 2 clock ticks (rrDist) of the state machine. Having these 2 values is extremely important to the successful working of the misalignment detector.

The detector works by checking how much sDIstance has changed over the last 2 clock cycles. If the change in sDistance is large, it means the car is not ideally positioned and it will set the park car state machine to the forward state. It will also define how long the car should remain in this state. This can be seen in the figure below:

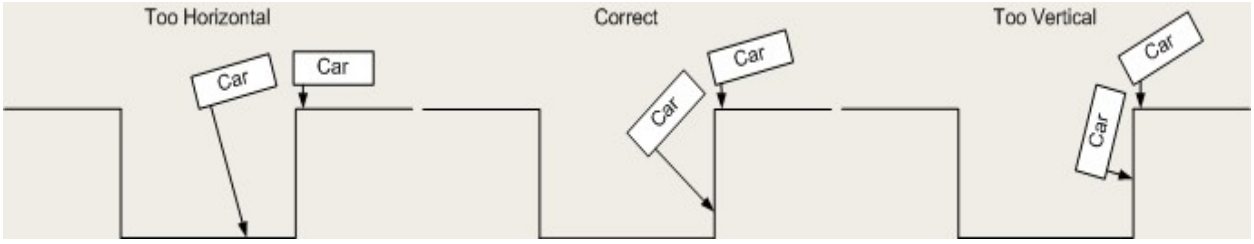


Figure 12: Example of Misalignment Detection

What is beautiful about this whole setup is that this exactly how a human would park the car! If the driver realizes that he is not aligned well enough, he will go forward and try again.

Putting it All Together

If you have taken a look at the high level design described earlier in the code, and read the description of the state machines in the earlier section, you have all the information you need to understand how the software of the car works.

Essentially, the AlgorithmModule state machines are what set the flags to control the movement of the car. These flags are interpreted by the ControlModule state machines, and translate it into actual motor control.

Results of Design

Speed of Execution | Accuracy | Safety and Interference | Usability

Speed of Execution

Speed wasn't a big issue for us. All components of the software were done as state machines.

The Motor Control state machines update at ticks every 50ms. This was ample time for the state machines to compute the necessary controlBits and assert the required inputs to the H-bridge. As a result, we were able to obtain highly accurate and sensitive responses from the motors to the control code.

The Algorithm Control state machines update at ticks every 100ms. This was enough time for the state machines to compute the necessary parameters, and to assert the necessary flags for the Control Module to interpret them and translate it into motor motion.

The response of the car to its surroundings is also very fast. The sensors have a response time of 20ms, which is quick enough for them to be processed in real time.

Accuracy

Distance Sensors

The sensors were very accurate within their specified range. Even with integer calculations, we were able to calculate distances with a +/- 1cm accuracy. Because we could not control the movement of the car with this degree of accuracy, the accuracy of our distance sensors are sufficient.

Parking Space Detection

The sequence to detect a parking space works very accurately. In the many trials that we performed, it always detected the parking space and stopped on detection.

Parking Algorithm

The parking algorithm we have written works very well when the car is close to a set distance from the side of the parking lot. It, however, becomes less accurate when the car is placed at larger distances from the parking space.

The parking algorithm we have written works very well when the car is close to a set distance from the side of the parking lot. It, however, becomes less accurate when the car is placed at larger distances from the parking space.

Safety and Interference

There were not many safety concerns with our project. In order to minimize disturbance to other project groups, and avoid the car colliding into students, we made a separate test area in the hallway. We used this test area for all testing purposes.

Also, since the car is completely autonomous, there was no human contact required (except for turning on the car). Therefore, there wasn't an issue of interference with the systems in the car.

Usability

In our opinion, this project has tremendous potential. With some more work on the parking algorithm, we feel that we can develop a system for the RC car to park irrespective of its orientation and distance from the parking lot. With enough research, this can be developed for real cars!

It can also be used as a learning tool for people who want to learn driving. By observing the motion of this car, students can learn how to parallel park better.

Lastly, this project could serve as a useful reference point for future projects dealing with R/C cars. The Control Module we have implemented to control the R/C car can be used universally for any 2-wheel drive R/C car.

Conclusion

Standards, IP, and Legal | Ethical Considerations

Overall, we feel the project met most of our expectations, as we were able to build an autonomous car which could detect a parking space, and park in it. When we started out, we intended the car to be able to locate a parking spot, and park irrespective of its distance from the parking space and its orientation. We were, however, unable to make it robust enough to accommodate parking from different orientations and distances. However, we feel the basic algorithm would remain the same, and this algorithm can be built upon to accommodate these features.

This was also a tremendous learning experience for us, especially with the hardware. We learn a tremendous amount about motor control systems, efficient circuit design, and hardware debugging. We also learned a lot about software. Through this project, we got valuable experience in developing efficient software using memory and run-time optimizations, something that cannot be gained through routine assignments.

If we had an opportunity to start this project over, there are a few things we would do differently.

1. Use a regulator to ensure a steady current is being supplied to the batteries despite the fluctuation in voltage across the batteries, particularly as they lose power
2. Consider implementing an optical sensor to track the velocity of the car
3. Build a feedback PWM loop to control the velocity of the car
4. Consider adding a fourth sensor on the side to calculate the orientation of the car
5. Consider building the car ourselves

Standards, Intellectual Property, and Legal Concerns

There were no standards or regulations that concerned our project because we decided not to control the car using radio signals. There are also no concerns with intellectual property because we purchased the RC car and are only using it for personal use. All the software and other hardware was designed by us.

Ethical Considerations

We adhered to the IEEE Code of Ethics throughout this project. We were very careful to consider the effects of our decisions on us, those around us, and the outcome of our project. We took significant time to plan our project before implementing it. We wanted to ensure our car was designed in the best way possible, in terms of performance, reliability, and safety. We worked cooperatively with other people, seeking and providing feedback and constructive criticism from the professor, TA's, and other groups to build the best possible design.

We also set up a safe testing environment. In an area with high traffic, people could have easily stepped on the car and been hurt. We tested our car in a highly controlled environment to ensure

everyone's safety. We did not make any decisions that would cause harm to others or the environment.

In no circumstances did we give or accept bribes. All of our parts were either bought or sampled in the appropriate manner. We have been honest regarding the results and limitations of our design. We have worked hard to create the best design possible in the given time frame, and it is safe to operate. All people involved with this project were treated fairly.

Appendix

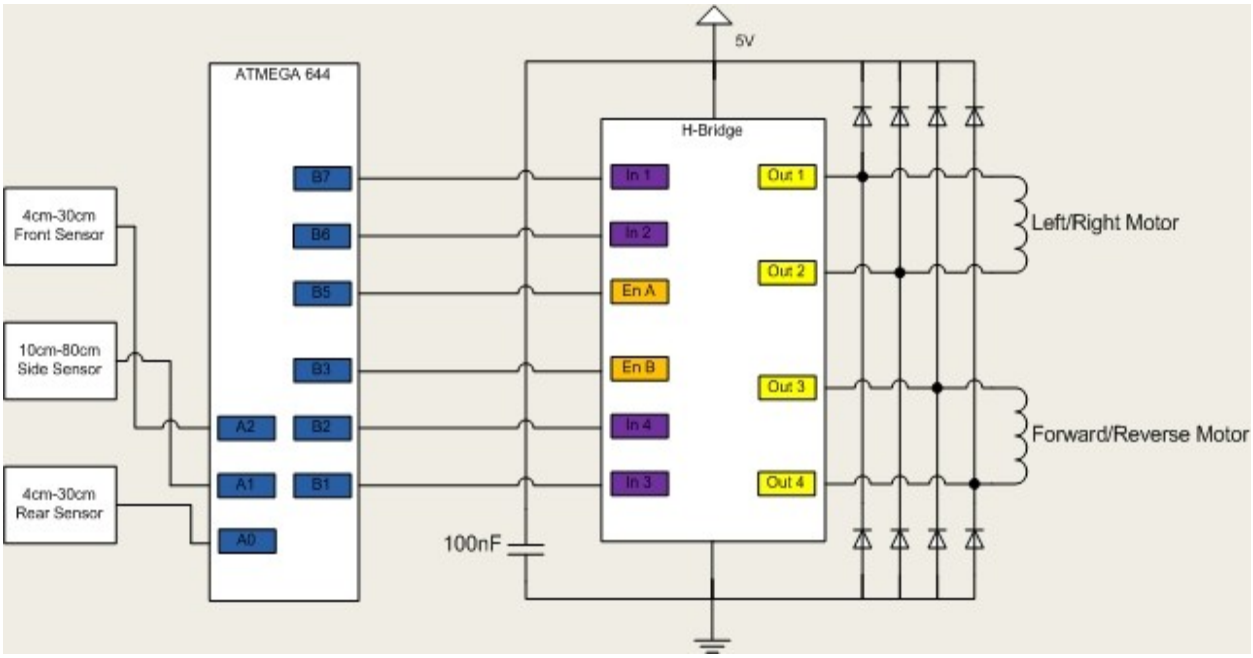
Commented Code | High Level Schematic | Parts List and Cost | Tasks | References

Appendix A: Commented Code

This link contains the source code for our project: Source Code

This link contain supplementary functions we wrote during the development of our project: Supplementary Code

Appendix B: High Level Schematic



Appendix C: Parts List and Cost

Part	Number	Cost
RC Car	1	$20
10cm-80cm Sharp IR Sensor (GP2Y0A21YK)	1	Free (Sampled From Sharp)
4cm-30cm Sharp IR Sensor (GP2D120XJ00F)	2	Free (Sampled From Sharp)
ST Micro H-Bridge (L298HN)	1	Free (Sampled from ST

Part	Number	Cost
		Microelectronics)
ATMEGA 644	1	Free (Sampled from Empire Technologies)
Custom PC Board	1	$4
Small Solder Board	1	$2
9V Battery	1	$2
AA Battery	4	$2
Headers	65	$3.25
Regulator (LM340T5)	1	Free (In Lab)
Screws and Spacers		Free (In Lab)
Total		$33.25

Appendix D: Tasks

Both of us contributed to all parts of the project, however we spent more time working on certain things.

- Design of Hardware: Nagappan
- Design of Control Algorithm: Both
- Design of Parking Algorithm: Prakash
- Testing of Hardware: Both
- Testing of Software: Both
- Compiling Report and Web Site: Both

Appendix E: References

Data Sheets

- http://document.sharpsma.com/files/gp2y0a21yk_e.pdf
- http://document.sharpsma.com/files/GP2D120XJ00F_SS.pdf
- http://www.st.com/stonline/products/literature/ds/1773/l298.pdf
- http://courses.cit.cornell.edu/courses/ee476/AtmelStuff/mega644full.pdf
- http://www.digchip.com/datasheets/download_datasheet.php?id=513599&part-number=LM340T5

Vendor Sites

- www.sharpmsa.com
- www.atmel.com
- www.st.com

Background Sites and Papers

- http://electronics.howstuffworks.com/rc-toy.htm

We would also like to thank Professor Land, Matt, and all the other TA's for the help they provided us during this project