

Autonomous Crash Avoidance System

Kristen Anderson
Kat Kononov
6.111 Fall 2010
Final Project Report

Abstract (Kat/Kristen)

Our project is a proof-of-concept model of a crash avoidance system for road vehicles. In order to accomplish this task, the system uses a radio controlled toy car equipped with a X-Bee transmitter and four distance sensors. When the car senses that it is going to crash, it avoids the crash by either stopping or swerving out of the way. The transmitter sends sensor data to the FPGA, and the FPGA processes that data, decides whether it needs to intervene, and generates a signal to control the car.

This project allows for avoidance of different types of crashes. The system can avoid head on collisions, swerving into objects, and if something gets too close to an object, it avoids the object. In addition, the user has the option to turn the avoidance system on or off, always allowing the driver to override the automatic crash avoidance. Finally, this setup allows subroutines to be programmed into the car. This project implemented the framework for autonomous parallel parking.

Table of Contents

1	Introduction.....	1
1.1	Motivation (Kristen).....	1
1.2	Overview (Kat/Kristen).....	1
1.3	High-level Architecture (Kat/Kristen)	1
2	FPGA (Kat/Kristen).....	2
2.1	Controller FSM (Kristen).....	3
2.1.1	Functionality.....	3
2.1.1.1	General Driving.....	3
2.1.1.2	Static Avoidance	4
2.1.1.3	Dynamic Avoidance.....	4
2.1.1.4	Front and Back Avoidance.....	4
2.1.1.5	Override.....	4
2.1.1.6	Correctional Override.....	4
2.1.1.7	Parallel Park.....	4
2.1.2	Implementation.....	5
2.1.2.1	Stop.....	6
2.1.2.2	Speed_Passive.....	6
2.1.2.3	Avoidance_Turn.....	6
2.1.2.4	Continue_Turning.....	7
2.1.2.5	Avoidance_Straight.....	7
2.1.2.6	Return_Turn.....	8
2.1.2.7	No_Turn.....	8
2.1.2.8	No_Right_Turn.....	8
2.1.2.9	No_Left_Turn.....	8
2.1.2.10	Parallel_Search.....	8
2.1.2.11	Parallel_Back.....	9
2.1.2.12	Parallel_Back_Other.....	9
2.1.2.13	Parallel_Forward.....	9
2.2	Sensor Input Module (Kat).....	9
2.2.1	Divider.....	10
2.2.2	Synchronizer.....	10
2.2.3	Sampler.....	10
2.2.4	Timer.....	11
2.2.5	Helper FSM.....	11
2.2.6	Minor FSM.....	12
2.2.7	Major FSM.....	13
2.2.8	Digit Buffer.....	14
2.2.9	Distance Buffer.....	14
2.2.10	Data Buffer.....	15
2.2.11	Data Latch.....	15
2.3	Driver Input Module (Kat).....	15
2.3.1	Synchronizer.....	15
2.3.2	Divider.....	16
2.3.3	Sampler.....	16

2.3.4 Receive FSM.....	16
2.3.5 Bit Timer.....	17
2.3.6 Buffer.....	17
2.3.7 Idle Timer.....	18
2.3.8 Enable FSM.....	18
2.3.9 Data Decoder.....	18
3 External Hardware (Kat).....	19
3.1 Sensor Infrastructure.....	19
3.1.1 Sensors.....	19
3.1.2 Microcontroller.....	20
3.1.3 Radio.....	21
3.2 Car Control Infrastructure.....	21
3.2.1 R/C Transmitter.....	21
3.2.2 FPGA Interface Circuit.....	22
3.2.3 TV Remote and IR Receiver Chip.....	22
4 Testing.....	23
4.1 Controller FSM (Kristen).....	23
4.1.1 Switch and Button Testing.....	23
4.1.2 Sensor Testing.....	23
4.1.2 System Testing.....	24
4.2 Sensor Input Module (Kat).....	24
4.3 Driver Input Module (Kat).....	25
4.4 External Hardware (Kat).....	25
4.4.1 Sensors, Microcontroller, and Radio.....	25
4.4.2 Car Control.....	25
5 Conclusion (Kristen).....	25
Appendix A: Works Cited	27

List of Figures

Figure 1: RC car equipped with sensors and X-Bee.....	1
Figure 2: System overview.....	2
Figure 3: High-level diagram of FPGA modules.....	3
Figure 4: Controller FSM state machine diagram.....	5
Figure 5: Block diagram of Sensor Input module.....	10
Figure 6: Helper FSM state transition diagram.....	12
Figure 7: Minor FSM state transition diagram.....	13
Figure 8: Major FSM state transition diagram.....	14
Figure 9: Driver Input module block diagram.....	15
Figure 10: Receive FSM state transition diagram.....	17
Figure 11: Enable FSM state transition diagram.....	18
Figure 12: (a) Sharp GP2Y0A21YK sensor.[1] (b) Output voltage characteristic for the sensor[2].....	20
Figure 13: High-level diagram of sensor hardware infrastructure.....	21
Figure 14: (a) Relay circuit diagram. (b) High-level diagram of FPGA to R/C interface.....	22
Figure 15: IR receiver chip diagram [3].....	23

1 Introduction

1.1 Motivation (Kristen)

Cars crash. From 2000 to 2007, there were over 10 million accidents a year in the United States, according to the National Safety Council [1]. Most of these accidents are caused by human error. This project focused on a way to decrease the number of crashes by designing a crash avoidance system. This system was designed for use on a car, and this project implemented a prototype on an RC car.

1.2 Overview (Kat/Kristen)

This project created a proof-of-concept model for a semi- autonomous car that avoids crashing into obstacles. The RC car prototype was equipped with distance sensors and a X-Bee radio to send the signals to the FPGA. The system avoids crashing by disallowing the driver to turn into an object, and if an object is in its path, the car avoids the object and regains its original direction.



Figure 1: RC car equipped with sensors and X-Bee

1.3 High-level Architecture (Kat/Kristen)

This project involves several pieces of hardware in addition to the labkit, including a TV remote, an infrared receiver chip, a RC car and the car's controller, four distance sensors, a microcontroller, and a pair of X-Bee radios. The driver uses the TV remote to send signals to the labkit. The infrared receiver chip receives the signals and passes

them to the FPGA. The Driver Input module on the FPGA decodes the received signal into a command.

On the car, each of the distance sensors are mounted on the four sides of the vehicle: front, back, left, and right. The distance sensors' data is serialized by the microcontroller and sent to the FPGA using the X-Bee radio. The corresponding X-Bee receiver is connected to the labkit, allowing for the sensor data to be input to the FPGA. The Sensor Input module on the FPGA decodes the received signal and outputs the four distances. The Controller module of the FPGA processes the driver commands and the sensor data, and creates commands for the car based on those inputs. Those commands are transmitted to the car through the labkit in/outs which are connected to the remote control that comes with the car. See the System Overview Figure 2.

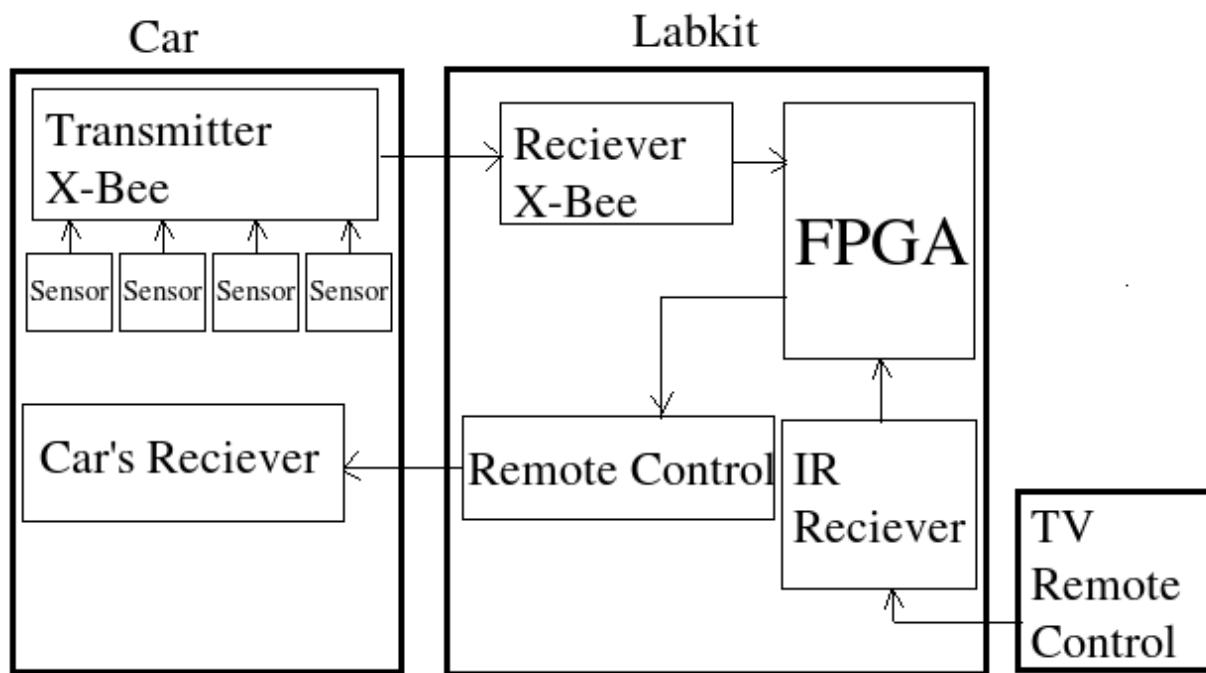


Figure 2: System overview

2 FPGA (Kat/Kristen)

The FPGA contains three modules: the Controller Finite State Machine (FSM), the Driver Input Module, and the Sensor Input Module. The Controller FSM is responsible for making the car command decisions based on the sensor and user inputs. The Sensor Input module processes serial signals received from the car's distance sensors and outputs the distances to the Controller FSM in an easily utilized form. Likewise, the Driver Input module processes the IR signal that the driver sends from the TV remote and outputs the command for the Controller FSM to use. Figure 3 shows a high-level diagram of the modules contained in the FPGA.

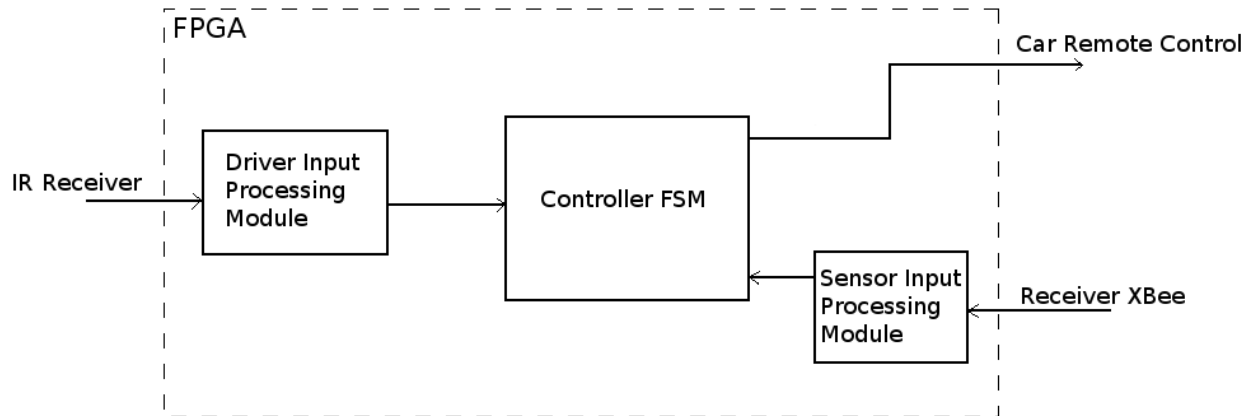


Figure 3: High-level diagram of FPGA modules

2.1 Controller FSM (Kristen)

2.1.1 Functionality

The Controller FSM's inputs are the driver's commands and the distances from the sensors. From this information, the Controller FSM makes a decision on what commands to send to the car.

2.1.1.1 General Driving

One of the fundamental differences between driving the car with its original remote and the TV remote is the inability to send two commands at once. The car remote is made with two joysticks: one that turns and one that controls the speed. Unfortunately with the TV remote, while the driver can push two buttons at once, due to the serial transmission, only one of the commands is sent at a time. To handle this discrepancy in user inputs, when the remote up button is pushed once, the car drives forward until the back button is pushed once. This same setup was utilized for backing as well. When the back button is pushed once, the car continues to back until the forward button is pushed once. The turn commands remain the same, where the car turns for as long as the left or right button is pressed.

This functionality was implemented throughout the Controller FSM. In the Stop state, when the up or down button was pressed (when the previous value of the button was 0 and the following value 1) the FSM entered the Speed_Passive state. The same was true in the Speed_Passive state. If the car was going forward and the back button was pressed, or the car was going backward and the forward button was pressed, the FSM entered the Stop state. As the car needed to retain the ability to stop in any situation, a statement to allow stopping was entered into all states.

2.1.1.2 Static Avoidance

The purpose of the system going through the FPGA is to implement a crash avoidance system. One of the ways this system is implemented is to constrain the car to not go forward unless there is enough distance in front of it to turn to avoid the object. The same is true for going backwards. This functionality was implemented in the Stop state, by not allowing the FSM to go to the Speed_Passive state and move, without the front or back respective distances being smaller than the minimum distance.

The other static avoidance is the car won't turn if there is an object on that side. This functionality was implemented for the car going both backwards and forwards. If there are objects on either side the car, the car is simply not allowed to turn.

2.1.1.3 Dynamic Avoidance

While the car is stopped, if an object gets too close from any side, the car moves to avoid it. For instance, if the car is approached from the rear, it moves forward. If an object is too close to the side of the car, it moves forward. If there is an object in front of the car and an object too close to the side of the car, it moves backwards.

2.1.1.4 Front and Back Avoidance

If the car is going forwards and it approaches an object, it turns to avoid the object. The car then attempts to return to going in the same direction, however, it only turns if there isn't an object in the way, otherwise it continues to go straight until an opening is found.

2.1.1.5 Override

While the default state is to have the crash avoidance on, there is also the option to turn off all of the avoidance and just retain the functionality described in the General Driving section. This button can be pressed at any time and the avoidance turns off. Every state contains a statement checking whether the override is on, and if override is on then the state transfers to the Speed_Passive state and can only change to the Stop state.

2.1.1.6 Correctional Override

Another setting on the system, is the ability to turn off the car avoidance correction. This leaves all the avoidance maneuvers on, except for the Front and Back avoidance.

2.1.1.7 Parallel Park

The number of buttons on the TV remote allows subroutines to be programmed into the car. This project implemented a parallel parking subroutine that is initiated by pressing 2 on the TV remote.

2.1.2 Implementation

The Controller FSM has 13 states: Stop (0) , Speed_Passive (1), Avoidance_Turn (2), Avoidance_Straight (3), Return_Turn (4), No_Right_Turn (6), No_Turn (7), No_Left_Turn (8), Continue_Turning (9), Parallel_Search (10), Parallel_Back (11), Parallel_Forward (12), Parallel_Back_Other (13). The Stop and Speed_Passive states are all that are needed to drive the car, without any avoidance maneuvers. The front and back avoidance turn and return turn require the Avoidance_Turn, Avoidance_Straight, Continue_Turning and Return_Turn states. The No_Right_Turn, No_Turn, and No_Left_Turn state ensure that the driver doesn't turn the car into a wall. The final four states, Parallel_Search, Parallel_Back, Parallel_Forward, and Parallel_Back_Other allow the user to push a single button and the car parks itself. Figure 4 shows the state diagram of the Controller FSM.

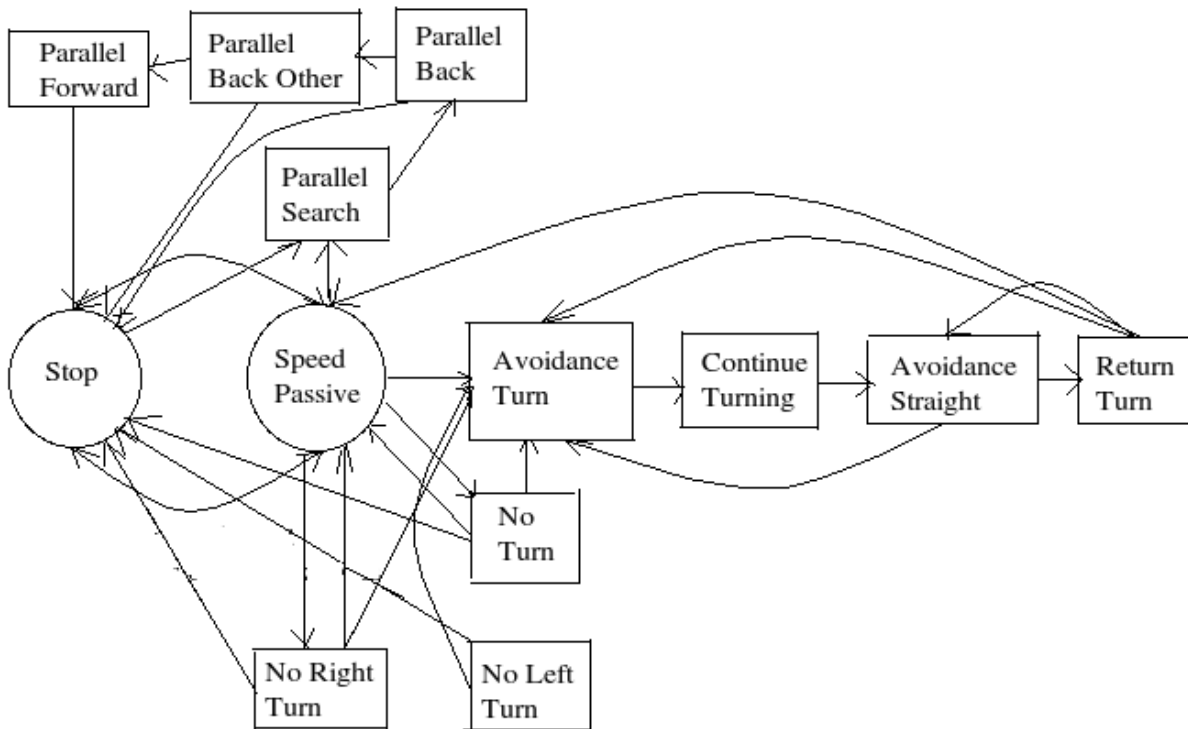


Figure 4: Controller FSM state machine diagram

Every state assigns nine registers, the four controls (control_front, control_back, control_left, control_right), the speed (speed), how much the car has turned (turned), which direction the car turned in (turn_dir), the car's previously turned amount

(prev_turned), and the state (state). The FSM also contains 14 different parameters. These parameters store the various minimum distances and times used in the various states.

2.1.2.1 Stop

The stop state contains the dynamic avoidance functionality and the static front and back avoidance functionality. This state also specifically accounts for the override function, by having two different methods to allow the car to go forward and switch into the Speed_Passive state. With the override on, the car moves forward on the rising edge of the user's input. With the override off, the car can only move when the direction that the user wants to go in has a greater distance than the minimum specified distance. When the car moves, the speed register changes to indicate if the car is going forwards or backwards. This feature allowed the same states to be used to implement the avoidance maneuvers, regardless if the car is going backwards or forwards.

The dynamic avoidance is implemented by checking each of the sensors and seeing if any of them are less than the minimum distance for this phase. If a sensor indicated that something is too close, the FSM applied a control to move the car forward or backward accordingly. If either of the side sensors indicate a too close of object, the state checks to see if there is an object in front of the car. If not, the car moved forward. If so, the system checked to see if an object is in back of it. If not, the car moves backwards. If the car finds itself boxed in, it stops, as it has no other options to avoid getting hit.

Another statement in the Stop state checks if a parallel park command has been issued. If so, the state transitions to the Parallel_Search.

2.1.2.2 Speed_Passive

The front and back avoidance and the side object avoidance initiate in this state. This state first checks to see if the user has asked the car to stop, and if so switches to the Stop state. Otherwise, if the override isn't initiated, the FSM checks to see if the distance in the direction the car is traveling is less than the minimum. If the distance is less, the state transitions to Avoidance_Turn. Next, FSM checks to see if the distance of either or both of the sides are too close. If one side is too close the FSM transitions to either No_Left_Turn, or No_Right_Turn, respectively. If both sides are too close the FSM transitions to No_Turn. In this state the left and right controls are set to whatever the driver specifies.

As in the Stop state, the Speed_Passive state sees if the driver has input a parallel park command changes states accordingly.

2.1.2.3 Avoidance_Turn

During the Avoidance_Turn state the car continues to turn until there is no longer an object in front of it. This state originally chooses a direction to turn by checking to

see which side distance is greater. If neither side is large enough to allow for turning, the car stops and the FSM changes back to the Stop state. Otherwise, the car keeps turning in the same direction, by initially setting the turn direction to whichever direction it decides, where right is 1 and left is 0. Before the car has decided which direction to go, the turn direction register is set to 2. Whenever the car reenters the Speed_Passive state, this register is reset to 2 to ensure that the car chooses a direction based off of closest object every time.

This state keeps track of how long it has turned, by the use of the turned register. This register counts on the enable of the 600 microsecond divider. The divider was used to make the count smaller and any timing parameters able to fit on the 32-bit computer. This count was book-kept so the car could return turn for the same amount of time that it originally turned.

This state also checks if the user has input a stop and transitions to the Stop state. Additionally, if the user has issued an override or correction override the system transitions to the Speed_Passive state.

2.1.2.4 Continue_Turning

In the Continue_Turning state, the car continues to turn in the direction that it was previously going until the count reaches the necessary additional turn. This state needs to be included due the difference between the sensor's measurement of the front or back distance when the car is at an angle and the actual distance between the car's corners and the object. During the Avoidance_Turn state, the prev_turned register is set to equal the turned register. This assignment is done to allow the Continue_Turning state to continue to increment the count of the turned register on the enable pulse, until it reaches the additional turn plus the previously turned amount. This method allows the FSM to keep track of how much the car has turned, as well as turn this additional amount.

If the user inputs a stop command, or the car can no longer continue turning, the car stops and the FSM changes to the Stop state. If the user inputs the override or correction override command the car stops turning and transitions to the Speed_Passive state.

2.1.2.5 Avoidance_Straight

During the Avoidance_Straight state, the car goes straight until the opposite direction's distance is large enough to allow for the car to turn in that direction. When distance is large enough, the car changes to the Return_Turn state. If the car encounters another object while in this state, it again enters the Avoidance_Turn state. The amount it has already turned is stored as it can't turn in the direction that it needs to go the car continues to turn in the same direction. Again, all the user's override inputs and stop inputs are executed.

2.1.2.6 Return_Turn

The car returns going it's original direction in the Return_Turn state. In this state, the car has found an opening to perform a return turn and turns until the turned register that counted up the other direction, counts down to zero. When the count gets to zero the FSM transitions to Speed_Passive. In this state, if the front distance becomes less the the minimum, the car again reenters the Avoidance_Turn state, but the FSM clears the amount turned and direction. This method means that the system will then attempt to return to going the direction that it was in when it reentered the Avoidance_Turn state.

If the car can't continue turning because that side distance is too small it returns to going straight in the Straight_Avoidance state. Like the previous avoidance state, if the user has asked to stop the state transitions to stop. If the user turns on the override or correction override, the state transitions to Speed_Passive.

2.1.2.7 No_Turn

The No_Turn state is similar to the Speed_Passive state, except that it won't allow the user to turn, enter the Parallel_Search state and lacks the transition to any of the no turn states. The control_left and control_right are always assigned to 0. If either or both of the side distances are more than the minimum the state goes the Speed_Passive, which will then transition to No_Right_Turn or No_Left_Turn if warranted. If the front distance becomes less the minimum, the state transitions to Avoidance_Turn.

2.1.2.8 No_Right_Turn

The No_Right_Turn state is very similar to the No_Turn state, except that it will allow the user to turn left. Again it transitions to the Speed_Passive state when the right distance is larger than the minimum. This state, unlike the No_Turn state, does have to take into account the speed of the car. If the car is going forwards it is not allowed to turn right, if the car is going backwards, it is not allowed to turn left.

2.1.2.9 No_Left_Turn

In this state, the car is not allowed to turn left. In exactly the same manner as in the No_Right_Turn state, if the car is going forward it can't turn left and if it is going backwards it can't turn right.

2.1.2.10 Parallel_Search

The Parallel_Search state sends the car forward until it has passed the length of the car. If either the left or the right side distances have a greater than the specified distances for the specified length of time, the car will enter the next phase, Parallel_Back. Otherwise, the car will continue searching. The state is using a similar method as employed in the previous states to count the amount of time. In fact, the turned register is reused, as it is not being used in any of the parallel parking states. As

described above, the turned register increments on the positive edge of the divider's enable. In this phase, only the user's override or stop will have any affect. Otherwise, the car is going forward and not turning.

2.1.2.11 Parallel_Back

The Parallel_Back state sends the car back and in the direction that the system found the gap. It will continue in this state for another specified amount of time before transferring to the Parallel_Back_Other state. In this state, the FSM is again counting using the turned register and the divider's enable. If the car approaches something too close, it will transition into the Stop state. Again, the user can only stop or override the vehicle.

2.1.2.12 Parallel_Back_Other

The other portion in parallel parking is the straightening out. In order to accomplish this, the driver usually backs in the other direction. In the Parallel_Back_Other state, the car is continuing the back, but this time turning in the other direction. Again, the state is counting the time using the divider's enable and the turned register. After another set amount of time the FSM transfers to the Parallel_Forward State.

2.1.2.13 Parallel_Forward

The final portion of parallel parking is to pull forward a little. The Parallel_Forward state sends the car forward for a specified amount of time. After this amount of time the car enters the Stop state, presumably successfully parked.

2.2 Sensor Input Module (Kat)

The role of the Sensor Input module is to decode sensor data received by the radio in serial form, and to present to the Controller Module four stable values representing the distance each sensor sees. A block diagram of this module is shown in Figure 5. The Sensor Input module uses three FSMs because the sensor data consists of four distances, each distance is made of three digits, and each digit has four bits.

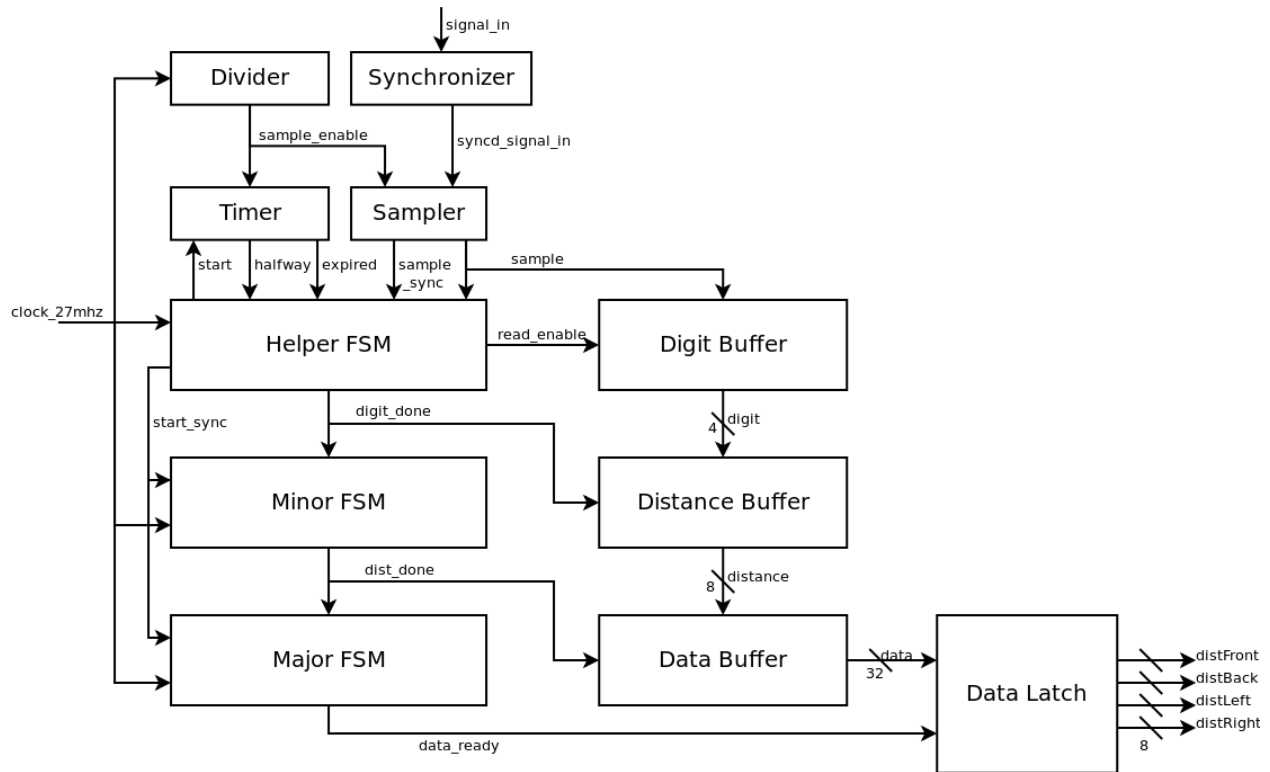


Figure 5: Block diagram of Sensor Input module

2.2.1 Divider

The Divider uses the FPGA's 27MHz system clock to generate a 76.8kHz (13 μ s) `sample_enable` signal which sets the sampling frequency to oversample the incoming signal by eight times. The Divider uses a register to count up to the appropriate number of system clock ticks and when it reaches that number, it asserts the `sample_enable` signal and resets the count to zero to start over.

2.2.2 Synchronizer

The Synchronizer is used to synchronize the incoming asynchronous signal with the 27MHz system clock.

2.2.3 Sampler

The Sampler block in the Sensor Input module does two tasks. First, it samples the incoming signal at the sampling frequency provided by the Divider, and it holds that value until the next sampling cycle. Second, the Sampler generates a `sample_sync` signal, which is used by the Helper FSM to reset to a waiting state when there is a pause in transmission. The `sample_sync` signal is asserted when 48 consecutive samples have been HIGH, which is longer than the longest possible string of 1s in the

data stream. The `sample_sync` signal is necessary to ensure that the FSMs in the Sensor Input module all start when a new set of data begins being received and not in the middle of a set of data.

2.2.4 Timer

Since the data received is using a fixed-bit time protocol where each bit lasts for $104\mu\text{s}$, the Timer counts out the duration of each bit and signals the Helper FSM to move on the next bit. When the Helper FSM starts a new bit, it signals the Timer to start over. The Timer also generates a halfway signal that is asserted halfway through a bit. This signal is used by the Helper FSM to generate a signal for the Digit Buffer to store the sample value at the middle of a bit for the appropriate bits.

2.2.5 Helper FSM

The Helper FSM keeps track of the state while receiving each digit of each distance. It keeps track of each bit that is received and signals when a value should be stored. This process is complicated by the fact the microcontroller and radios are configured to send each digit in ASCII encoding, because they are meant to display on a computer screen. Fortunately, only the ASCII encodings for the digits 0-9 are being received. Of the 9 bits received for each digit, only 4 bits change, and those 4 bits correspond to the digit being transmitted. Therefore, the Helper FSM is designed to skip the 5 bits that don't matter and only assert the `read_enable` signal during one of the four bits that needed to be stored in the Digit Buffer. However, the FSM waits until all bits of the ASCII encoding have been received before beginning to wait for the next digit, as shown in the state transition diagram in Figure 6. After all the bits of a digit have been received, the Helper FSM generates a `digit_done` signal which signals the Minor FSM to move on to the next digit, and the Distance Buffer to store the current digit contained in the Digit Buffer.

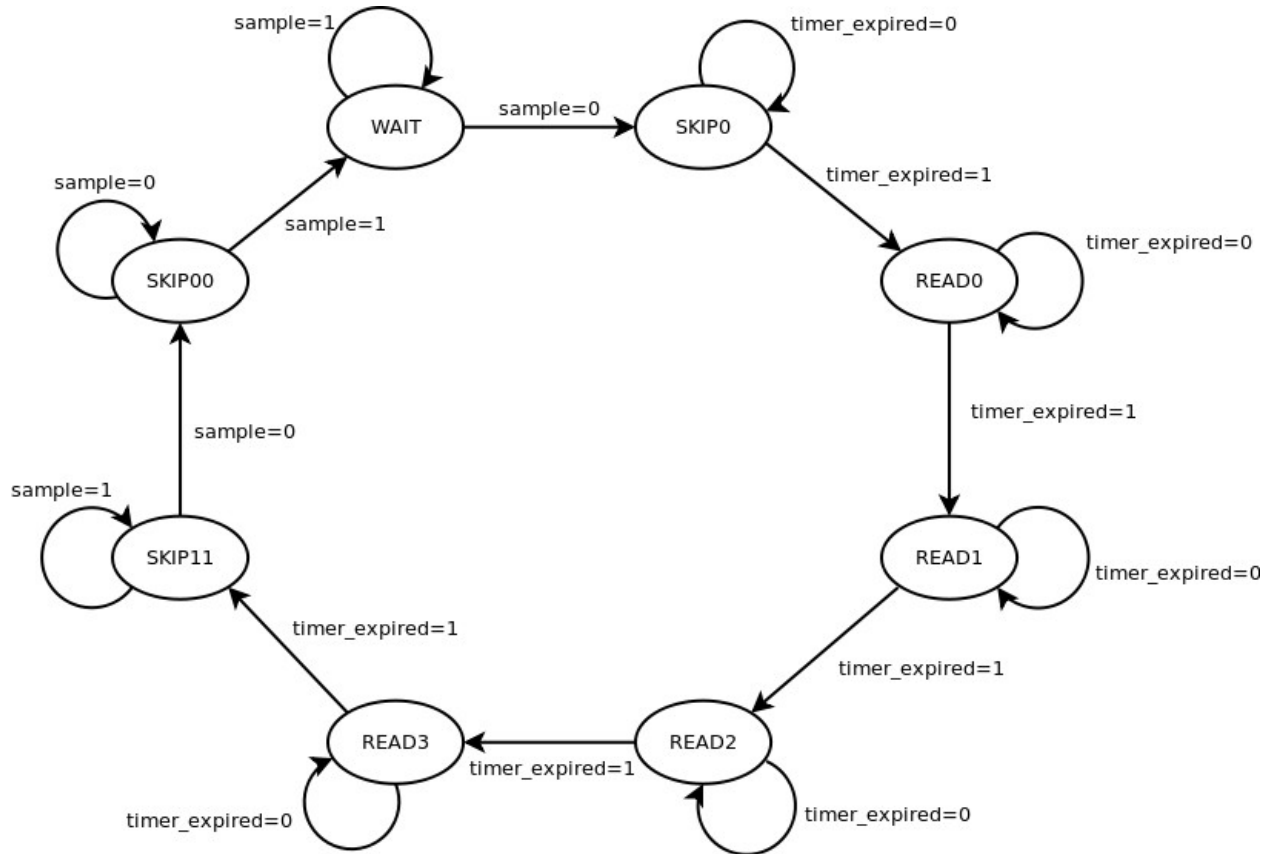


Figure 6: Helper FSM state transition diagram

2.2.6 Minor FSM

The Minor FSM keeps track of which of the three digits are being received per distance. The state transition diagram is shown in Figure 7. The Minor FSM starts in the WAIT state and waits until the transmission of a new digit begins. As all three digits of a distance are received, with the highest order digit first, the Minor FSM generates a `dist_done` signal to signal the Major FSM to move on to the next distance. The `dist_done` signal also signals the Data Buffer to store the current distance contained in the Distance Buffer.

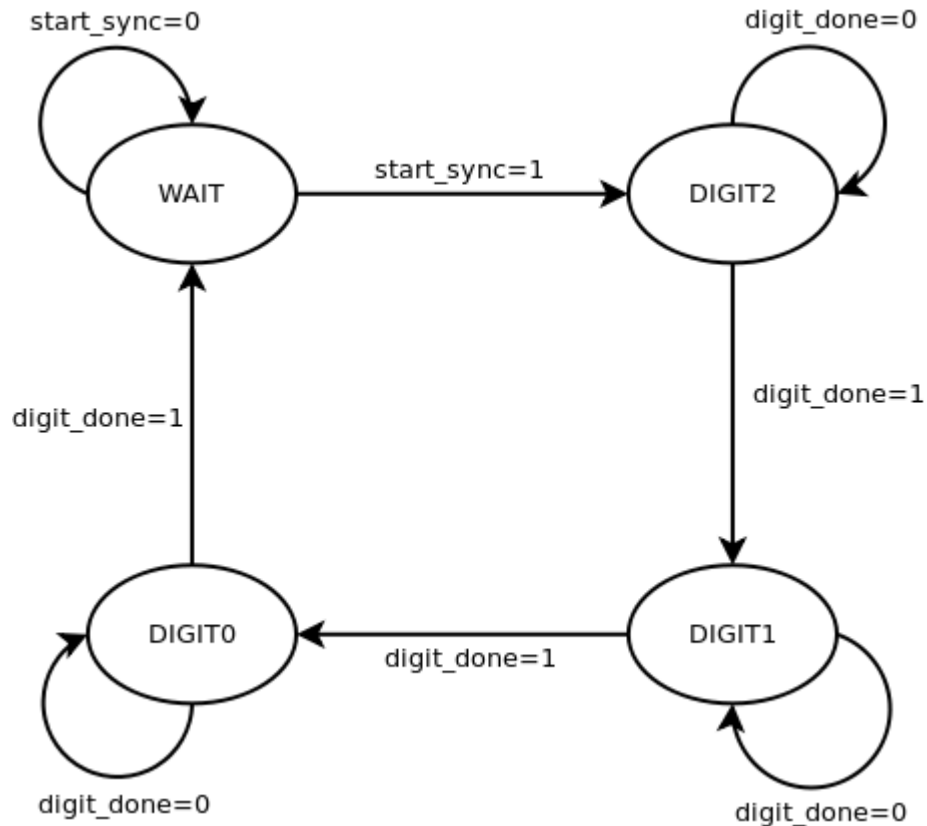


Figure 7: Minor FSM state transition diagram

2.2.7 Major FSM

The Major FSM keeps track of which of four distances are being received. It starts in the WAIT state and waits for the first distance to begin. It uses the `dist_done` signal from the Minor FSM to move to the next distance. After the Major FSM finishes receiving the fourth distance, it generates a `data_ready` signal that tells the Data Latch that a complete set of data has been received and stored, and it is ready to be presented to the Controller module. The state transition diagram for the Major FSM is shown in Figure 8.

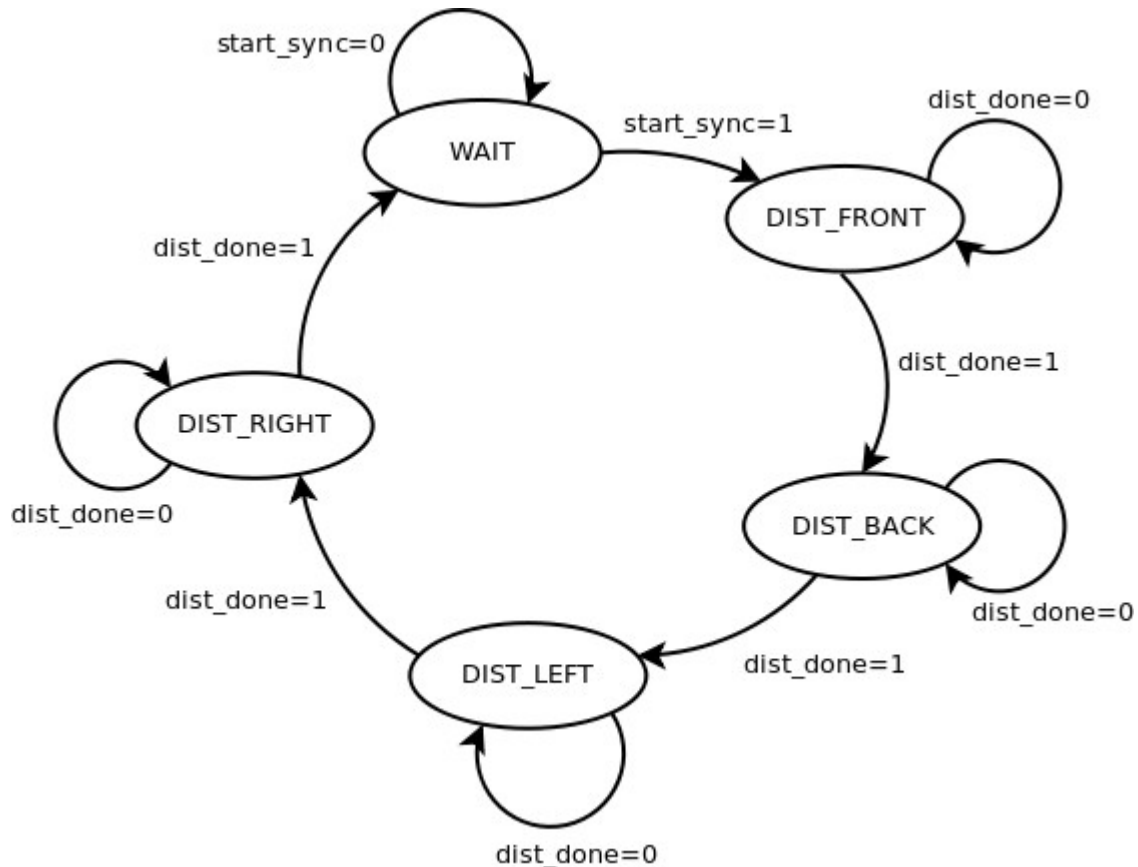


Figure 8: Major FSM state transition diagram

2.2.8 Digit Buffer

The Digit Buffer stores the four bits of each digit as the bits are received. It uses the read_enable signal from the Helper FSM to read the current value of the sample and store it as a bit in the current digit. The Digit Buffer passes the last four stored bits to the Distance Buffer.

2.2.9 Distance Buffer

The Distance Buffer stores the three digits of each distance as they are received. It uses the digit_done signal from the Helper FSM to know when to read and store the four bits it gets from the Digit Buffer. The Distance buffer stores the last three digits that were received, and it computes the distance from those digits. The highest order digit multiplied by 100 is added to the middle digit multiplied by 10 and added to the lowest order digit to produce the actual distance. This eight-bit distance is passed to the Data Buffer.

2.2.10 Data Buffer

The Data Buffer stores the four distances that comprise a full set of sensor data. It uses the `dist_done` signal generated by the Minor FSM to know when to read the distance presented to it by the Distance Buffer. The Data Buffer passes the last four distances received to the Data Latch.

2.2.11 Data Latch

The Data Latch is the output block of the Sensor Input module. It reads the four distances from the Data Buffer and uses the `data_ready` signal from the Major FSM to update the output after each complete set of distances has been received. The set of distances is held constant until another set of distances is ready to replace it.

2.3 Driver Input Module (Kat)

The Driver Input module is responsible for receiving the driver's commands from the TV remote and decoding them to present to the Controller module in an easily usable way. A block diagram of the Driver Input module is shown in Figure 9.

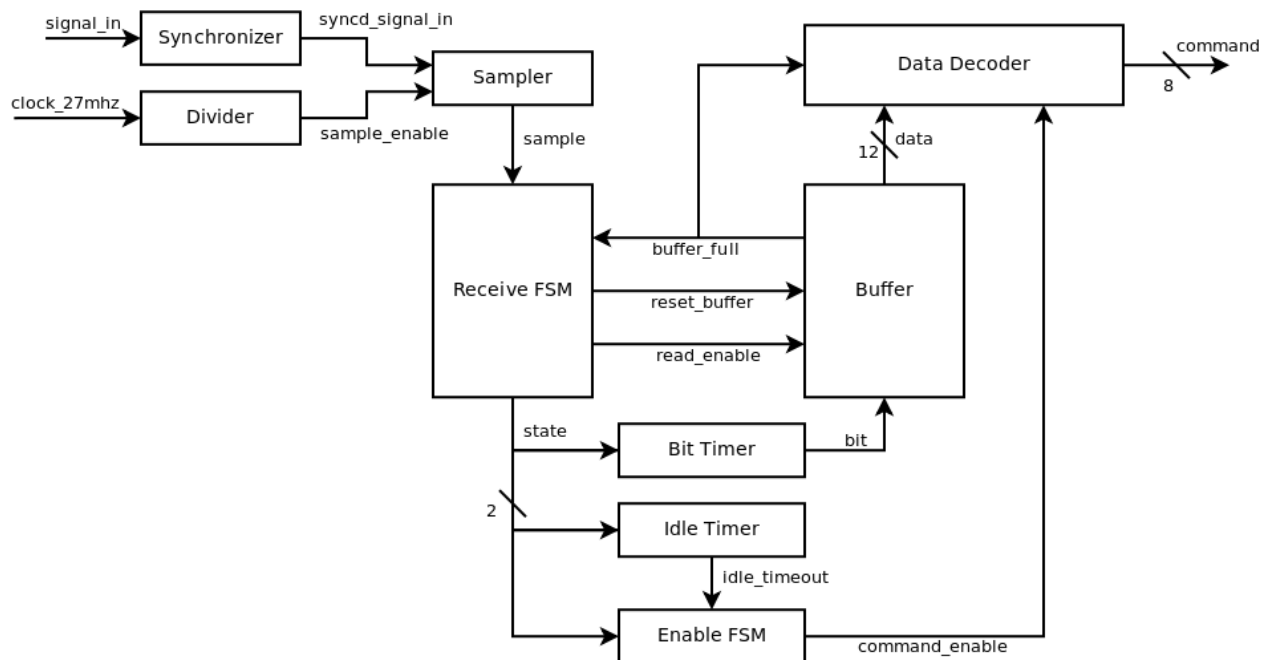


Figure 9: Driver Input module block diagram

2.3.1 Synchronizer

The Synchronizer is used to synchronize the incoming asynchronous signal with the 27MHz system clock.

2.3.2 Divider

The Divider uses the FPGA's 27MHz system clock to generate a 75 μ s `sample_enable` signal, which sets the sampling frequency to oversample the incoming IR signal by eight times. The Divider uses a register to count up to the appropriate number of system clock ticks and when it reaches that number it asserts the `sample_enable` signal and resets the count to zero to start over.

2.3.3 Sampler

The Sampler samples the incoming signal at the sampling frequency provided by the Divider and holds that value until the next sampling cycle.

2.3.4 Receive FSM

The Receive FSM is responsible for keeping track of whether a bit or a pause is being received, while a transmission is occurring. The state transition diagram is shown in Figure 10. The FSM starts in the IDLE state and waits for a command to begin being received. When the sample value changes from 0 to 1, the FSM knows that the command has started. It waits for the start sequence of the command, and then alternates between bits and pauses until the full command has been received. The Receive FSM generates a `reset_buffer` signal and asserts it each time it is done receiving a command. It also generates a `read_enable` signal and asserts it when a bit is ready to be added to the Buffer.

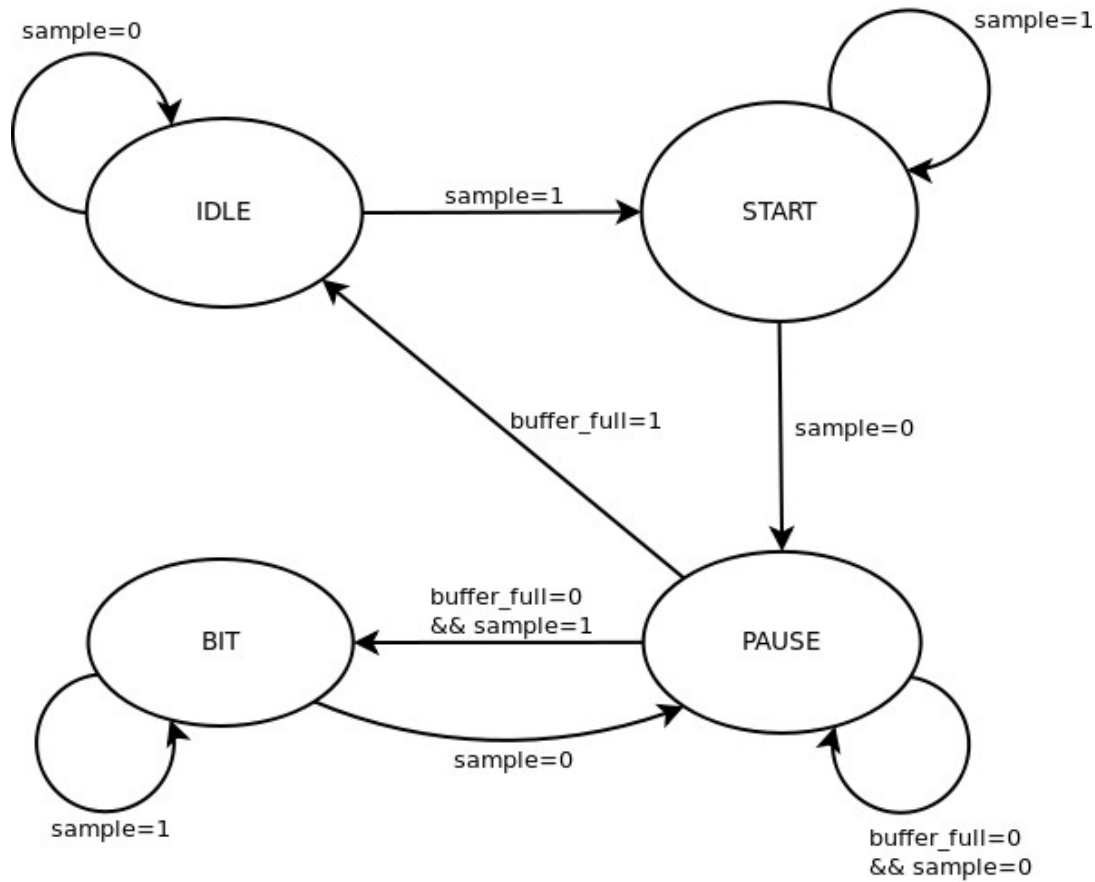


Figure 10: Receive FSM state transition diagram

2.3.5 Bit Timer

Since the TV remote uses a pulse width modulated protocol to send commands, meaning the length of each bit determines its value, each bit needs to be timed in order to tell what the value is. The Bit Timer starts when the Receive FSM enters the BIT state, and when the FSM exits the BIT state, the Bit Timer produces the value of the bit as 0 or 1 for the Buffer to read.

2.3.6 Buffer

The buffer stores the 20 bits of the command received from the TV remote and it generates the `buffer_full` signal that tells the Receive FSM and Data Decoder that the whole command has been received. However, only 5 bits of the 20 bits change for the eight commands used in this project. Therefore, the Buffer outputs the 5 bits as the command data for the Data Decoder. When storing bits from the Bit Timer, the Buffer acts as a shift register that shifts bits when prompted by the `read_enable` signal from the Receive FSM.

2.3.7 Idle Timer

The Idle Timer is a timer that determines how long the Receive FSM has been in the IDLE state. If the FSM has been in IDLE for longer than the pause between consecutive transmissions of the same command, then the driver has stopped sending a command. The longest pause between consecutive transmissions is 15.2ms. Therefore, after the FSM has been in IDLE for 15.3ms, the Idle Timer asserts the timeout signal which is used by the Enable FSM.

2.3.8 Enable FSM

The Enable FSM keeps track of whether or not the driver is sending a command. The state transition diagram is shown in Figure 11. The Enable FSM starts in NOTRCV state which indicates that no command is being received. When the Receive FSM moves out of IDLE state, the Enable FSM knows that the first transmission of a command has begun. After the first transmission is over, the Enable FSM stays in the RCV state for all subsequent transmissions, until the driver has stopped sending a command and the timeout signal is asserted by the Idle Timer. The Enable FSM generates a command_enable signal, which is asserted while the driver is sending a command. The command_enable signal is used by the Data Decoder to know when to output a command to the Controller module.

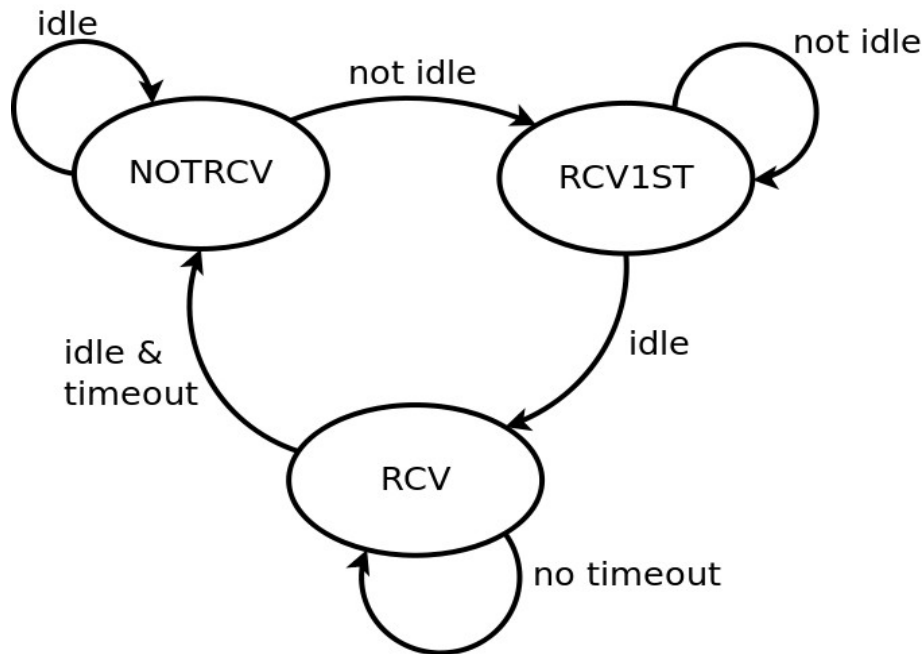


Figure 11: Enable FSM state transition diagram

2.3.9 Data Decoder

The Data Decoder reads the command data from the Buffer and uses the buffer_full command to update the data as a new command is received. However, it

only outputs the command when the `command_enable` signal from the Enable FSM is asserted. This ensures that the command is presented to the Controller module only when the driver is actually pressing a button on the TV remote, and otherwise the output is all 0s to indicate that no command is being given.

The buttons on the remote were assigned to represent commands for the car to go forward, backward, steer left, steer right, and several other functions such as crash avoidance, override, and subroutines. The eight-bit command that the Data Decoder outputs has a bit that corresponds to each of the car actions. Only one bit at a time is 1 and rest are 0 because only one button at a time may be pressed on the TV remote. The Data Decoder uses a look-up table to translate the five bits of command data it gets from the Buffer to the eight-bit command that it passes to the Controller Module.

3 External Hardware (Kat)

The project consists of four distance sensors, a microcontroller, a radio, and a large battery mounted on a remote-controlled car. The car is controlled by its own R/C transmitter which was hacked to allow the FPGA to operate it. The driver communicates commands to the FPGA using a TV remote. The interface between all of the components involves hardware as well as software.

3.1 Sensor Infrastructure

3.1.1 Sensors

The four distance sensors used in this project are Sharp GP2Y0A21YK Optoelectronic distance sensors. Their operating range is between 10cm and 80cm. The analog output voltage of the sensor is approximately linear with the reciprocal of the distance within the operating range, as demonstrated in Figure 12(b). The sensor uses an IR signal to detect the distance and has a response time of 39ms.

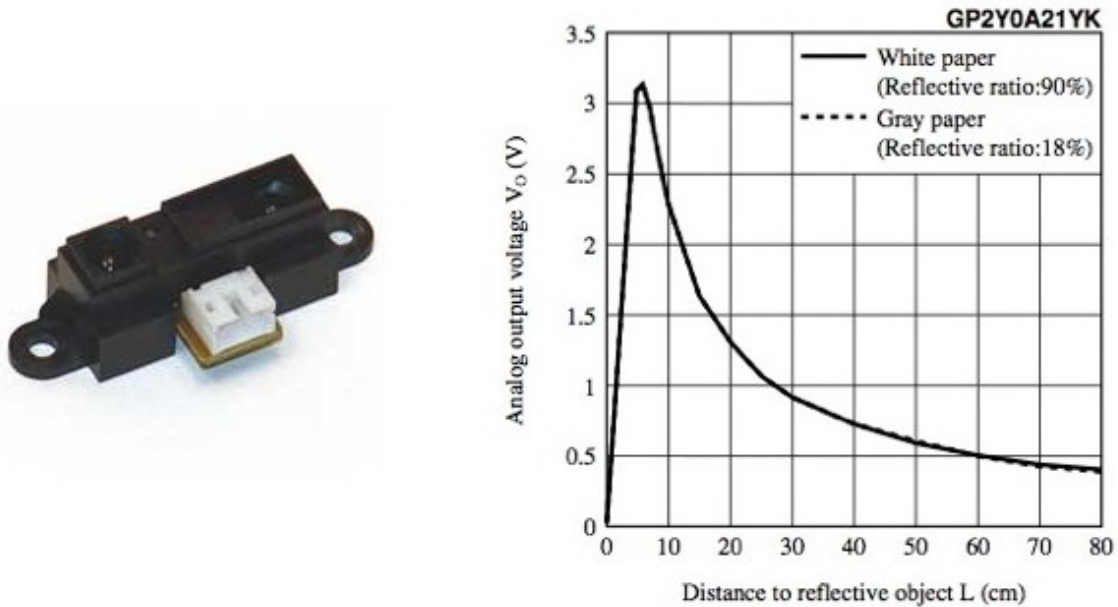


Figure 12: (a) Sharp GP2Y0A21YK sensor.[2] (b) Output voltage characteristic for the sensor[3].

3.1.2 Microcontroller

The microcontroller is a type of Arduino. It has several analog input ports that are connected to each of the distance sensors. The microcontroller is configured to convert each analog value into an eight-bit digital value. The digital values are then serialized and passed to the radio through the microcontroller's serial port at a rate of 9600 baud. The microcontroller loops through these steps continuously as the system is running. Figure 13 demonstrates the structure of the sensor, microcontroller, and radio assembly. All components are powered by a 9V battery.

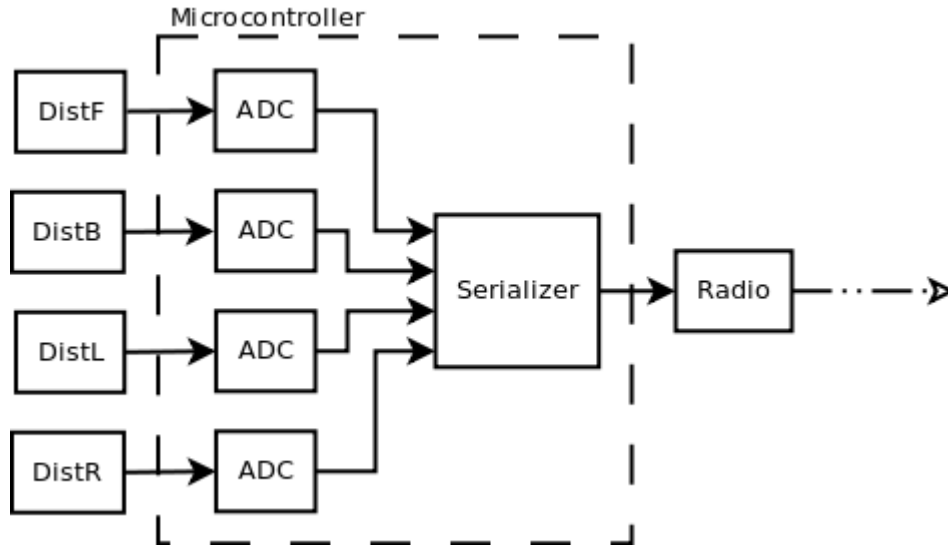


Figure 13: High-level diagram of sensor hardware infrastructure

3.1.3 Radio

The radio is a pair of identical X-Bee RF modules which transmit serial data over a 2.4GHz frequency. The radio on the car was configured to transmit sensor data to the radio on the labkit at a rate of 9600 baud, the same rate that it received from the microcontroller. Likewise, the radio on the labkit was configured to receive at 9600 baud rate. The serial protocol uses a fixed data time for each bit, so the transmission length is independent from the data. At this rate, a complete set of distance data is received by the FPGA every 12.5ms, getting three distance samples per sensor response cycle. Although the radios and microcontroller can be configured for a much higher baud rate than 9600, it would merely unnecessarily drain the battery because the limiting factor on data rate is the sensor response time of 38ms.

3.2 Car Control Infrastructure

3.2.1 R/C Transmitter

The car is controlled by its own R/C transmitter, which was hacked to allow the FPGA to interface with it. The circuit board of the transmitter was removed from its plastic casing. The joysticks on the R/C made electrical connections between corresponding contacts on the circuit board by a mechanical means. When these contacts were connected to a common ground, the R/C would transmit a signal that would instruct the car to go forward or backward and to steer left or right. Wires were soldered to these four contacts and connected to a set of relay circuits which simulated

the mechanical connection.

3.2.2 FPGA Interface Circuit

A relay circuit was used to simulate a mechanical connection between two electrical contacts. When a command needs to be sent to the car, the corresponding FPGA output port would assert a HIGH, turning on the transistor and allowing current to flow through the relay. The relay would close and make an electrical connection between two contacts on the R/C circuit board. A kickback diode is used to protect the relay when the FPGA signal goes LOW. The circuit, shown in Figure 14(a), is replicated four times as shown in Figure 14(b) to control each action of the car: forward, backward, steer left, and steer right.

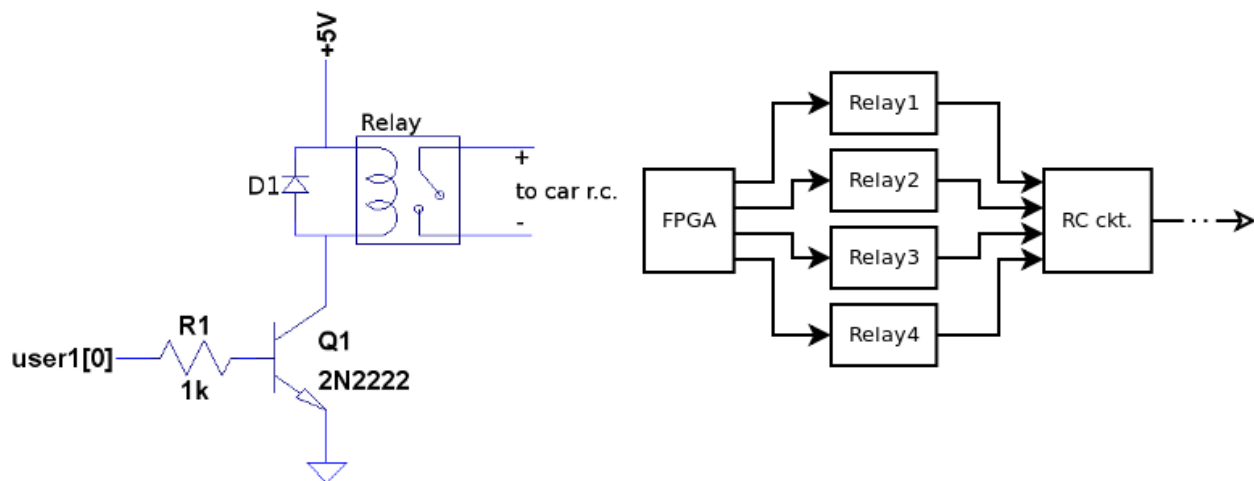


Figure 14: (a) Relay circuit diagram. (b) High-level diagram of FPGA to R/C interface.

3.2.3 TV Remote and IR Receiver Chip

Since the car's native R/C is used by the FPGA to control the car, a TV remote is used to allow the driver to input commands to the car. An advantage of using the TV remote over the car's native controller is that more functionality can be added using software since the TV remote has more buttons than the car's R/C. These additional buttons allow complex subroutines, such as parallel parking, to be programmed for the car to perform, based on a single command from the driver.

When a button is pressed on the TV remote, it sends an IR signal to an IR receiver chip, shown in Figure 15. The IR receiver chip receives the signal and passes it to an I/O port on the FPGA. The IR signal contains a command encoded in serial form. The TV remote uses a 20-bit pulse width modulated protocol, which is different from the fixed bit-time serial protocol used by the microcontroller.

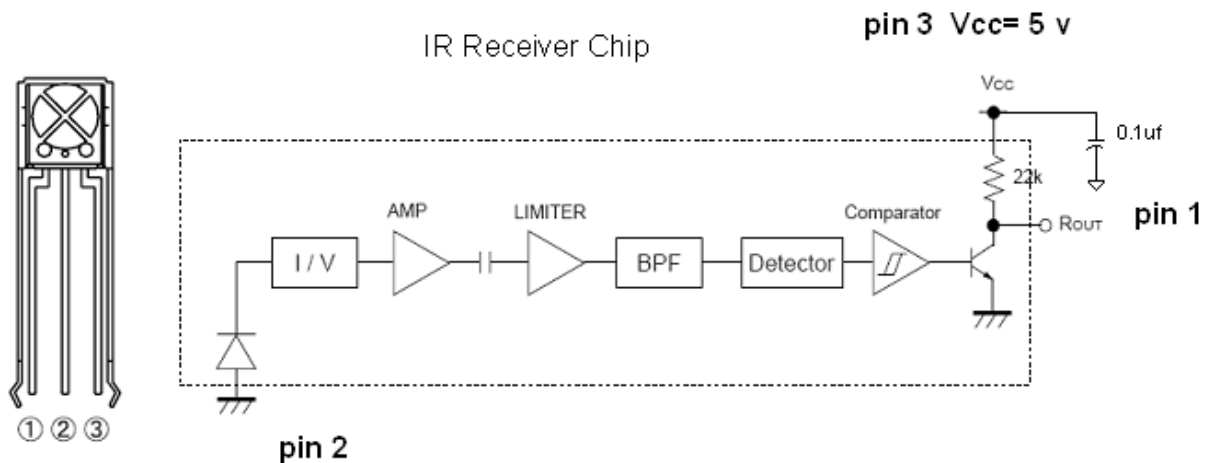


Figure 15: IR receiver chip diagram [4].

4 Testing

4.1 Controller FSM (Kristen)

For the Controller FSM, three major types of testing were performed. For the first bit of testing, the FSM was tested on the labkit using the switches as the distance inputs and buttons as the driver's inputs. For the second set of tests, the actual distance sensors were used, but the car turned off, and for the final round of tests everything was used and the car was turned on. This final round of testing was very important as the 14 distance and timing parameters had to be determined through this type of testing.

4.1.1 Switch and Button Testing

This type of testing was done to ensure that the FSM was transitioning through states correctly and the correct steps were being taken. This testing was done using the switches and buttons, as a substitute for the not yet ready Driver Input and Sensor Input modules. No major problems were found in this testing.

4.1.2 Sensor Testing

Testing with the Distance Sensor revealed a couple of different difficulties. The first one was that the distance sensors read a smaller number the closer the object is to the sensor. Unfortunately, the Controller FSM was written with the assumption that the closer the object, the smaller the distance the smaller the number. Another minor problem, was when the battery gets too low, the distance sensors still return a number, but the number is simply incorrect. The distance sensors are only useful from 10 cm to 80 cm. However, if the sensor is too close the sensor reads a larger distance. The

distance sensors are also quite sensitive to the mounting angle. Some were initially mounted pointing down and the FPGA received a value for the distance from the floor.

4.1.2 System Testing

System testing needed to be done to determine the values of the 14 parameters. Luckily, the code was fairly lightweight to allow relatively fast reprogramming of the FPGA. This testing not only revealed parameter values, but it also checked the dynamics of the Controller FSM. Originally the Continue_Turning state did not exist, but this testing revealed that there needed to be another state to continue the avoidance turn for a little longer. This test also revealed that the RC car has a poor turning radius, which resulted in a very large minimum front and back distance.

Over the course of the parameter determination for the parallel parking function, it became clear that the car has a delay in the turn, when it reverses turns. This fact is why the time spent in the two backing phases of the parallel park are significantly different, regardless of the similar dynamics. Unfortunately, this was realized a little late to fix the Controller FSM's Return_Turn state. Throughout the testing, it was unclear why the car continued to turn past the original direction, but it seems to be due to this lag in turn changes.

Another realization, that would improve the Controller FSM, was that due to the poor turning radius, the car would attempt to turn before hitting the wall, but simply not have time and crash. To correct this a useful statement to include in the Controller FSM would be to simply stop if the car was too close the wall in front of it.

4.2 Sensor Input Module (Kat)

The Sensor Input module was tested mainly the logic analyzer. Each state of each FSM was brought out to the logic analyzer and observed to verify correct transitioning. The biggest problem that was discovered while testing was that while the data transmission from the sensors would run continuously, the system would sometimes initialize into the waiting states in the middle of a data transmission. When this happened, the states would transition correctly, but they would be offset by however far the transmission was when the system initialized. To resolve this problem, logic was added to synchronize the three FSMs with the (relatively) long pause between transmissions. This way, although the system may still initialize in the middle of a transmission, it will get back into sync by the time the next transmission starts, and the behavior would be correct in the steady state.

4.3 Driver Input Module (Kat)

The Driver Input module was tested by setting up an IR receiver chip on another labkit and verifying that it was able to receive and decode commands from the TV remote. Correct transitioning between states was verified using the logic analyzer. After verifying that the states transitioned correctly, further testing revealed that the buffer_full

signal was being asserted one clock cycle too soon and the Data Decoder was latching the value of the buffer before the last bit was added. To resolve this in the easiest way, a delay register was added to the buffer_full signal. After the logic analyzer indicated that the command was being received correctly, the command decoding portion of the module was tested using the row of LEDs on the labkit to verify that the correct bit corresponding to the given command would light up.

4.4 External Hardware (Kat)

4.4.1 Sensors, Microcontroller, and Radio

The testing of the hardware mounted on the car was done incrementally. First, only one value was transmitted through the microcontroller and radio to verify that it was received at the other radio using the oscilloscope. A potentiometer was used instead of a distance sensor to produce an analog input value that is easily controlled. After that step was successful, four potentiometers were used to verify that four values are being received at the labkit. Then the potentiometers were replaced with distance sensors and a final verification was done.

4.4.2 Car Control

To test the relay circuit that interfaced the FPGA with the R/C controller, first each relay block was tested by applying a HIGH to the block input and verifying that the relay clicked. When the whole circuit was built correctly, each car function (forward, back, left, right) was assigned to a button on the labkit (button_up, button_down, etc.) to verify that the car would respond to these commands with the appropriate action (ie going forward when button_up was pressed).

5 Conclusion (Kristen)

The purpose of this project was to make a scalable crash avoidance system prototyped on a RC car. The RC car successfully avoided several types of crashes. Specifically, it avoided head-on collisions, swerving into walls, and other cars crashing into it while crash avoidance car was stationary.

The system avoided collisions by having four distance sensors, one in each direction attached to the car. These sensors were routed through an X-Bee, which sent the signal to the labkit and FPGA. The car's remote was disassembled and directly connected to the FPGA, to allow the FPGA to send commands to the car. To allow user inputs through a TV remote, the FPGA was wired with an IR receiver. The FPGA decrypted the distance signals as well as the user's inputs to decide what commands should be sent to the car to avoid crashing.

The FPGA had three main modules, the Controller FSM, the Sensor Input Module and the Driver Input Module. The Sensor Input module, deciphered the distances from the distance sensors. The Driver Input Module determined the

commands sent from the driver via the remote control. The Controller FSM decided what commands to send the car, from the sensor's and driver's inputs.

The main issues encountered were dealing with the hardware and system interface issues. In addition, hardware constraints and schedule deadlines prevented the system from working perfectly. However, while there were some difficulties, the system was successful in avoiding several types of crashes.

Appendix A: Works Cited

[1] National Safety Council. (2010, December 8). *Motor Vehicle Accidents- Number and Deaths*. [Online]. Available: <http://www.census.gov/compendia/>

[2] Dr. Robot, Inc (2010, December 8). Copyright 2001- 2010 [Online]. Available: www.drrobot.com.

[3] Sharp GP2Y0A21YK datasheet.

[4] 6.111 staff. (2010, December 8). *6.111 Lab #4* [Online]. Available: web.mit.edu/6.111/www/f2010/