EMBEDDED SYSTEM

CLASS PROJECT

[PARKING MAID]

Author: [Po-yen Chou, Ruoqing Fu, Hao-Yu Hsieh]

Project Advisor： Stephen Edwards

[05/14/2014]

Table of Contents

Introduction

The project idea of automatic parking robot was inspired by the movie “Knight Rider”. Amazed by the storyline made in the movie, we aim to make a smart car just like “KITT” fitted with artificial Intelligence. The first step we took to actualize a smart robot is to train him how to park by himself.

This project presents a smart parking robot implemented on Altera Cyclone FPGA. The car is designed to detect empty parking space and park into the area automatically with FPGA control. Our implementations include hardware modules to detect the parking space, and software to control the parking procedure. Users can start the system remotely on computers/mobile phones when they need to park. The parking robot will then be woken up, start finding the space and complete his parking task. The robot is designed to processes equivalent parking ability to humans’ and creates an image of artificial intelligence. In detail, the robot can move and turn with different speed, directions and angles, and perform parallel parking, garage parking, parking space detection and automatic trace adjustment functions.

Design Overview

The following is a high-level overview of space detection and automatic parking using Altera Cyclone FPGA daughter card and hardware components.

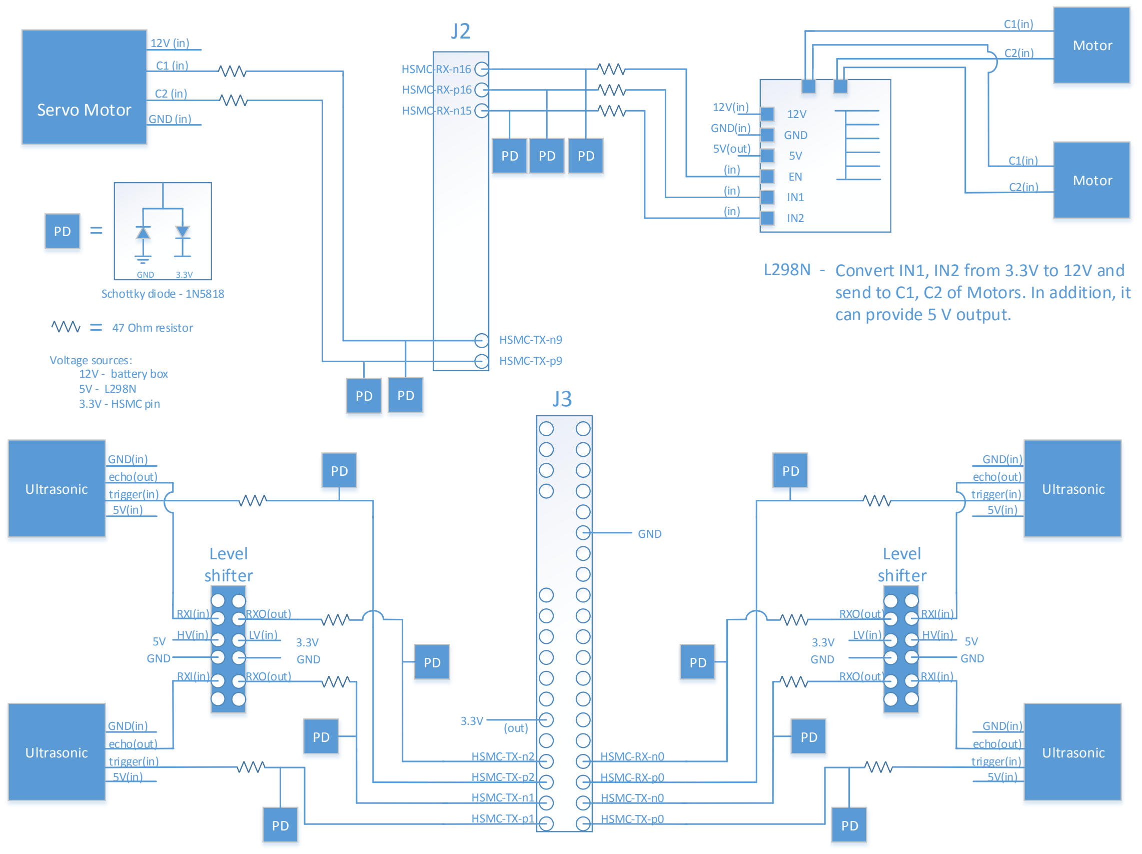


Figure 1-Top Level Design Schematic 1

As seen in the circuitry shown above, FPGA daughter card (labeled as J2 and J3) is primarily connected to Servo Motor, L298N converter, two level shifters and two regular motors and four ultrasonic detectors. The three motors came with the LEGO car we purchased. Servo motor is used to make turns, and regular motors help the car move straight in either forward or backward direction. Four Ultrasonic modules are also placed on the car with two on the right side, one in front and the other in the back. They are used to track the dynamic distance between the current position of the car and the obstacles nearby for parking calculations. L298N and level shifters are merely voltage converters to support various power needs.

Implementation

In this section, we will discuss the implementations in both hardware and software aspects. In the hardware implementation, we will provide details of how connections are physically made and how each component is specified. In the software implementation, we will describe the abstract parking process and present its physics model. We will also provide explanations for our software strategy of accomplishing the parking task.

Hardware Implementation

Ultrasonic and Level Shifters

There are 4 pins on each Ultrasonic module--Vcc, Gnd, Trigger and Echo. Vcc and Gnd are tied to 5V power rails converted by L298N. Trigger signal is an input sent by FPGA and connected to 4 pins on the J3 column of the FPGA daughter card. When ultrasonic finishes detecting, the Echo pin will send a responding signal to FPGA to notify the ready state. FPGA will compare the time interval between sending Trigger and receiving Echo, characterized by clock cycles to calculate the distance that ultrasonic just detected. Level Shifter is used to step down the 5V pulse generated by Echo to 3.3V that the FPGA pins can tolerate. The following is the scope capture of Trigger and Echo working waveforms:

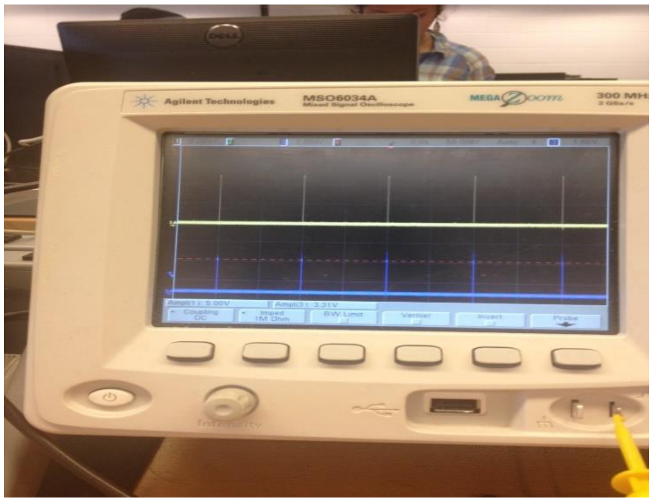


Figure2- Echo and Trigger Waveforms (yellow->Echo Blue->Trigger)

Servo Motors

There are 4 pins on Servo Motor--Vcc, Gnd, C1 and C2. Vcc and Gnd are hooked up with 12V power rails. C1 and C2 are to control directions and angles when car make turns. They are connected to J2 column of FPGA daughter card.

Regular Motors and L298N

Two regular motors are installed to move the car forward and backward. Each motor only has two pins C1 and C2 that are connected through L298N to FPGA. Enable, In1 and In2 on L298N are tied to 3 pins on FPGA daughter card to convert the input signal sent by FPGA to 12Vs that motors can receive. Enable is used to control the speed and direction. L298N also provides a voltage conversion from 12V to 5V for ultrasonic use.

Software Implementation

Parallel parking algorithm

For this part, the position and direction of our car is assumed to be known at any time. We first attempt to develop a parking method with the shortest length lot needed and implement the model in Matlab.

First we assume the car is a rectangular rigid body with length L and width W, and is expressed in the two dimensional Cartesian coordinate system. X and Y are the center locations of the car. Θ represents the angle between the vertical direction of the car and the y-axis.

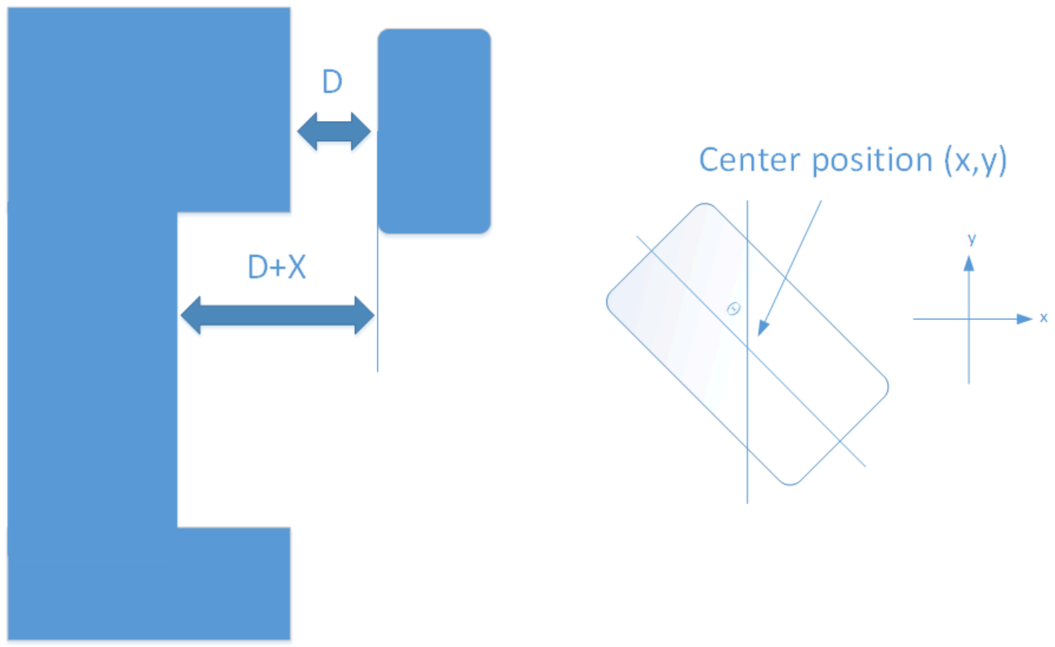
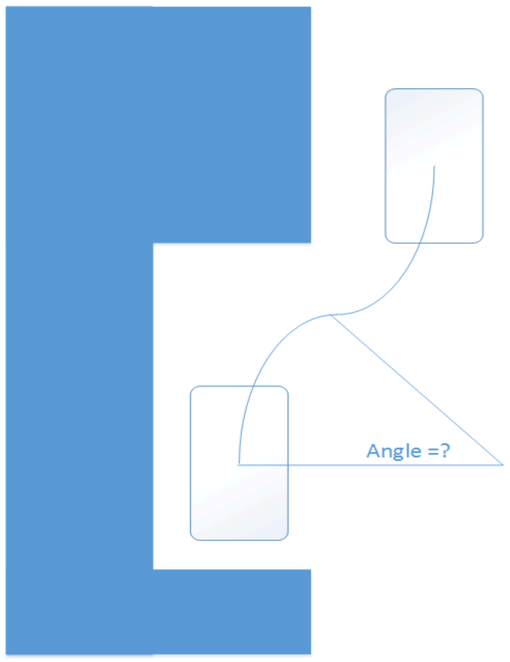


Figure 3 car parameter specification 1

As for the algorithm, we are going to implement a four wheel steering method as opposed to the Ackermann steering algorithm (two wheels in the rear never turn), which is mostly used by the production cars. This means that the trajectory of all points of the car is an arc but with different radius.

The car is initially sitting in parallel with the wall where the back of the car lines up with the upper side of the parking lot. The distance between the car and the wall is D, and the width of the parking lot is X. We plan to move the car along two identical arcs in the opposite directions with the closest distance to the wall. What angle should we turn?



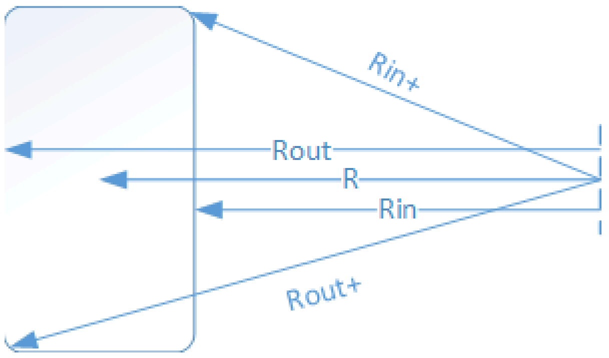


Figure 4 angle to turn specification

Figure 5 radius specification

If the rotation radius of the center of the car is R, then the radius of other corners of the car can be expressed as the following:

$$R_{in} = R - \frac{W}{2}, R_{out} = R + W/2$$

$$R_{in+} = \sqrt{{R_{in}}^2 + (\tfrac{L}{2})^2} \;, \quad R_{out+} = \sqrt{{R_{out}}^2 + (\tfrac{L}{2})^2}$$

If the car moves along an arc, then the closest safe distance that our car can be away from the wall is $R_{out+} - R_{out}$, in order to prevent the lower left corner of the car from hitting the wall.

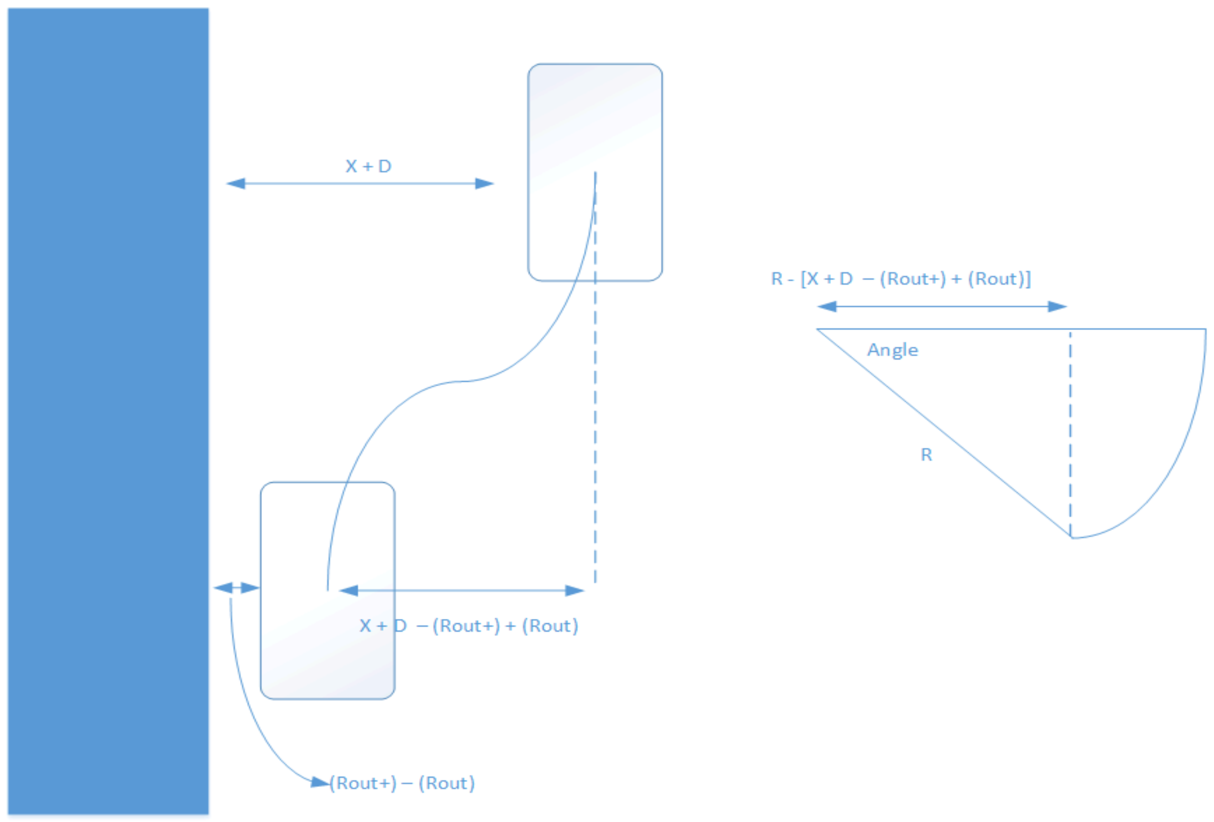


Figure 6 angle to turn calculation

As shown on the diagram above, the x-axis displacement of the arc is

$$X + D \ - R_{out+} + \ R_{out}$$

Hence we can calculate the Angle:

$$Angle = cos^{-1}\{(R \ - [X + D \ - R_{out+} + \ R_{out}])/R\}$$

This is the angle we and to turn with a specific X + D.

Next we are going to estimate where should we start turning so that it will not hit the wall,

The two possible crashes may happen are indicated below:

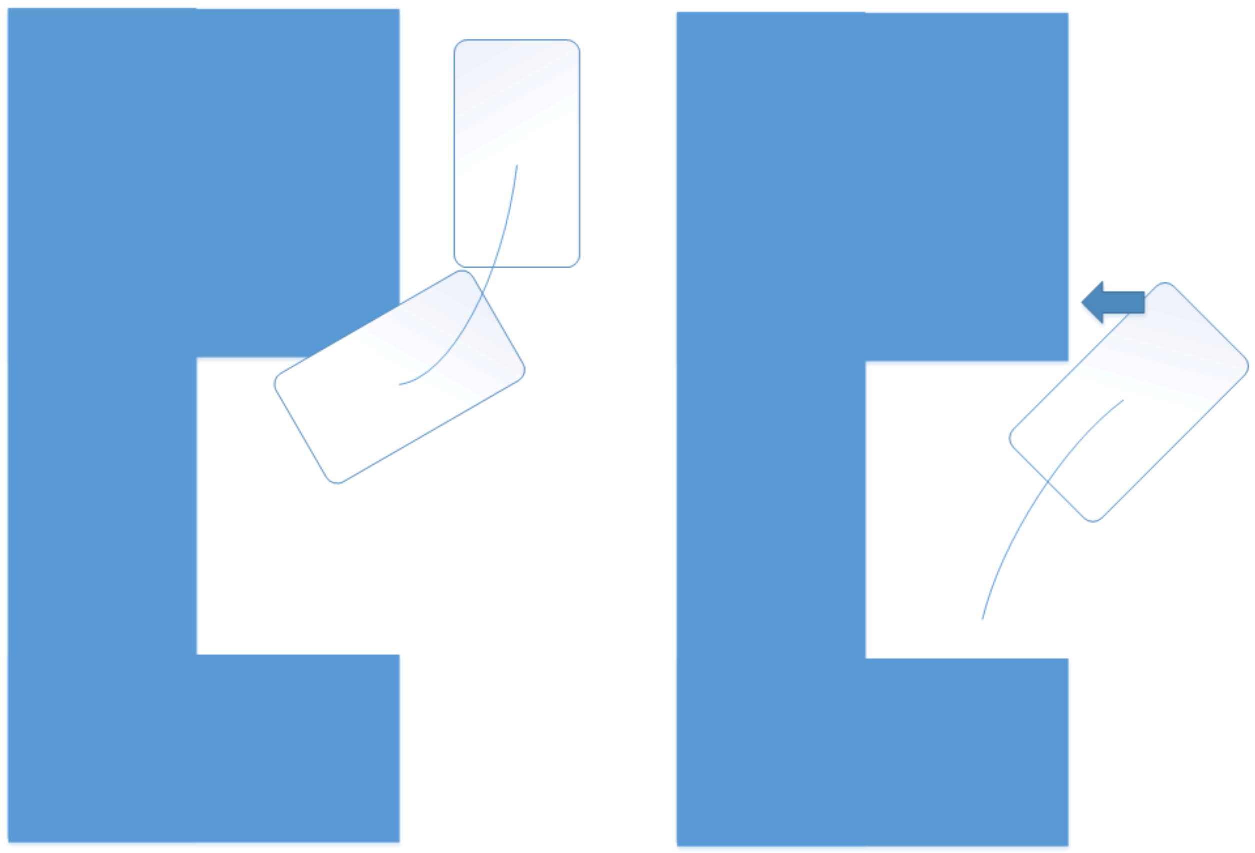


Figure 7 crashing case diagram

The left side of the car may hit on the wall when it is moving along the first and second half of the curve.

So how do we avoid the crashes?

For the first condition, the left center point of the car is most likely to hit with rotation radius Rin+. The following figures provide mathematical details on the crashing limits:

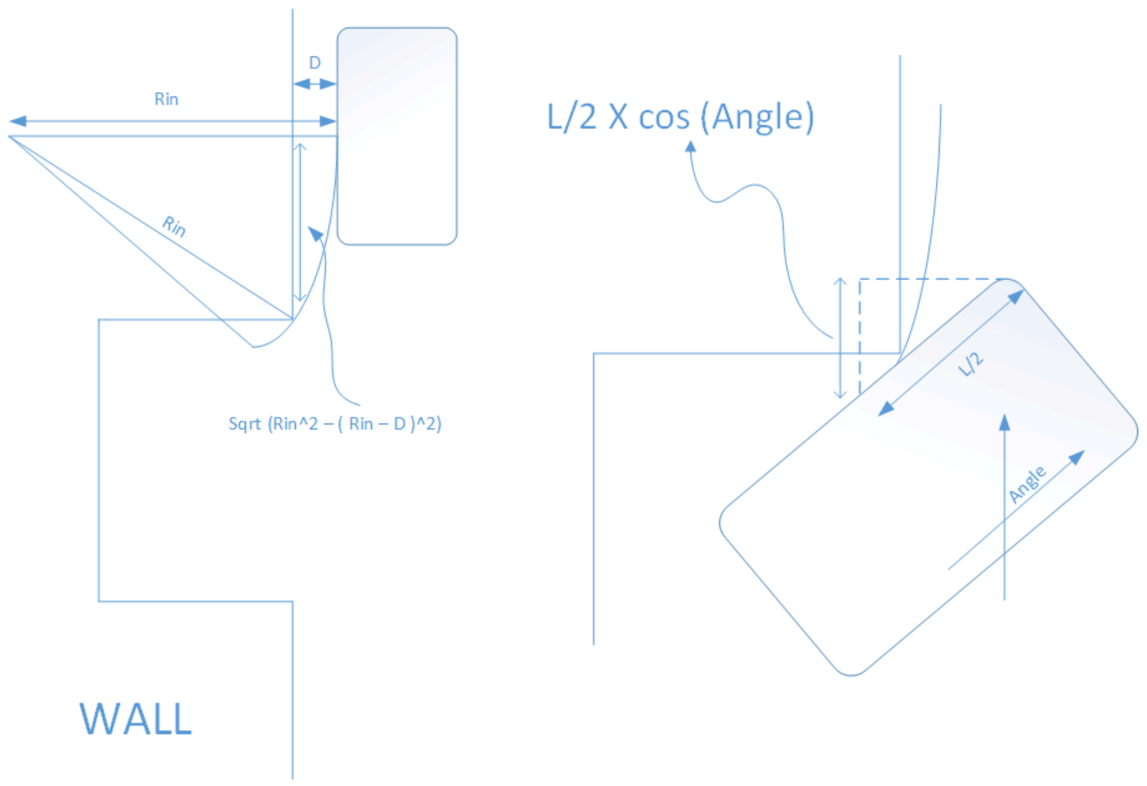


Figure 8 crashing limits model

As seen from the drawing, in order to avoid crashes, the initial distance in y direction between the center of the car and the upper side of the parking lot should be at most

$$\sqrt{{R_{in}}^2 - (R_{in} - D)^2}$$

In this condition, we can make sure that the car will be safe when moving through the first arc. But it is still likely to hit the wall in the second condition as the figure indicates above. To solve this problem, the left center of the car must be positioned lower than the upper side of the parking lot in the first arc. According to the figure on the right, if we can shift down the left top of the car by $\frac{L}{2} \times \cos(Angle)$ in advance, the upper left corner of the car will move below the upper side of the parking lot, meaning that the car will be safe while moving through the second arc.

Therefore, the initial maximum distance in y direction between the center of the car and the upper side of the parking lot now turns to be as the following

$$\sqrt{{R_{in}}^2 - (R_{in} - D)^2} - \frac{L}{2} \times \cos(Angle)$$

The left side of the diagram below shows the maximum distance condition discussed above whereas the right side of the diagram shows the case without margin.

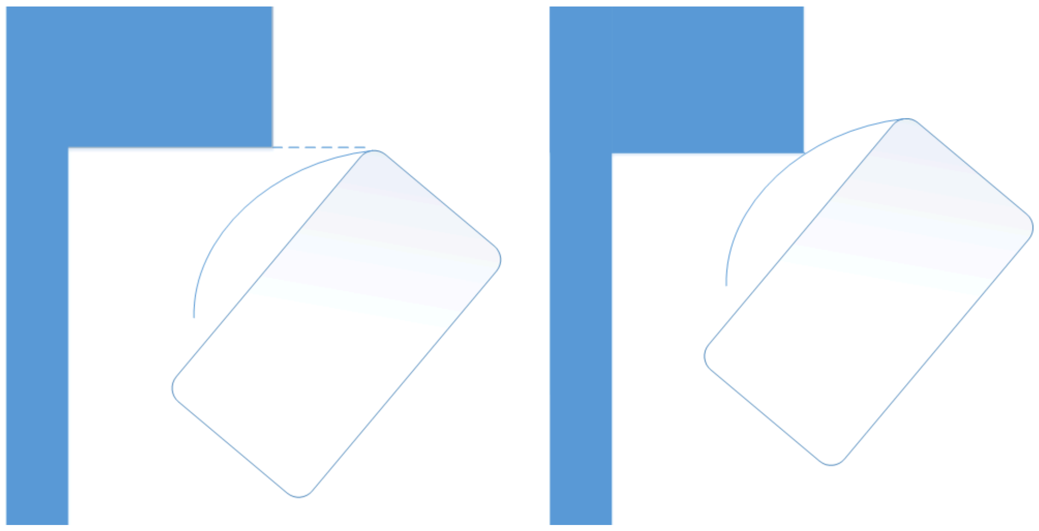


Figure 9 margin demonstration diagram

The mathematical calculation and concept for this aggressive algorithm are too lengthy to show in the report. However, we implemented this model in the Matlab model. As you can see from the picture below, the upper left side of the corner moves just enough to pass the wall.

But in reality, $\sqrt{{R_{in}}^2 - (R_{in} - D)^2}$ is the dominating factor while other factors such as ultrasonic inaccuracy have way less impact on the position calculation. Also based on our experiment, we need to leave a margin for the car to move from the theoretically calculated starting point. As indicated in the figures above, the car should move by $\sqrt{{R_{in}}^2 - (R_{in} - D)^2} - \text{Margin}$ from the initial position before it starts running through the first curve. Margin is a small number acquired through experiment.

Garage Parking Algorithm

We designed the garage parking algorithm based on the parallel parking model. The initial conditions and parameters are the same as those used in parallel parking case. The algorithm was modified where the car finishes its first curve. Instead of having the car keep moving in the second curve as in parallel parking, we put in a set of conditions with ultrasonic modules to break the curve and change its path to straight. As soon as the right front and right back ultrasonic modules detect almost the same distance, and the back right ultrasonic modules detects a reasonable distance from the wall, the car is programmed to stop.

Automatic Trace Adjustment Parking

The 3rd parking demo includes automatic trace adjustment parking techniques. The car starts crooked. While it is moving, both right front and right back ultrasonic modules will start collecting data. The right front ultrasonic measured data will be checked against a distance range with max and min limits. If the car is not moving straight and falls out of this distance range, the car will adjust itself to make reasonable turns and moves away from where it is shifting to.

The algorithm also implements a counter that tracks the time after the car successfully detects the space. If this time period is reasonably long and gets up to the max limit that indicates the parking spot is a couple times more than what is needed, the parking algorithm will be activated and move the car into the parking spot.

Our model idealized a portion of testing conditions, logic and parameters. However in real implementations, due to ultrasonic imprecision and other disturbing factors, we modified the parking algorithm with a slightly different structure and experimental data as opposed to the golden model.

Difficulties/Challenges

Signal/Power Integrity was an unexpected challenge for our implementation. Due to lack of protection mechanism in our original design, the hardware modules presented circuit damage and failure. We then added diode-clamping circuitry to prevent overvoltage on all the critical pins. The details of all these critical pins with protection circuitry (labeled as PD) can be found on the top level schematic. The diode clamping circuit is shown as the following:

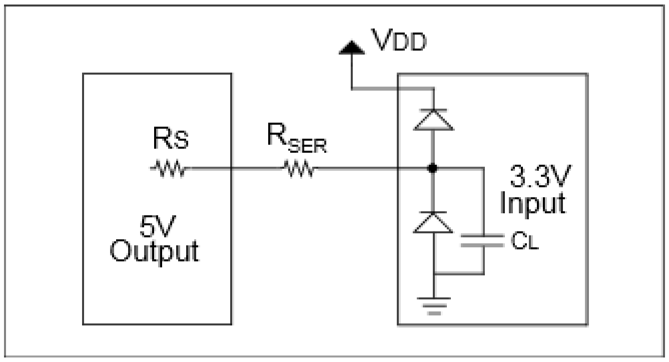


Figure 10-Diode Protection Circuit 1

"3.3V Input " is regarded as the FPGA pin in our design. "5V Output" is one example of the connections made to each FPGA pin. Two diodes are placed against each other. VDD is tied to 3.3V, same as FPGA pin working voltage. If an input that goes into FPGA pins is larger than Vdd by the amount of a diode turn-on voltage, one diode path will be activated and therefore isolates FPGA pins. On the other hand, if the input that goes into FPGA pin is negative and less than Gnd by the amount of a diode turn-on voltage, another diode path will be triggered and prevent currents from going into FPGA. The following table includes the current and voltage measurements of our circuit to prevent all the signal/power integrity issues.

COMPONENTS	PINS	MAXIMUM VOLTAGE	MAXIMUM CURRENT	CORRESPONDING PINS
Ultrasonic	Vcc	5V	7mA	N/A
	Trigger	3.3V (pulse)	0.5mA	HSMC-RX-p0 HSMC-TX-p0 HSMC-TX-p2 HSMC-TX-p1
	Echo	5V (pulse)	0.031mA	N/A
	GND	0V	7mA	N/A
Level shifter	HV	5V	1uA	N/A
	LV	3.3V	0.01mA	N/A
	RXO	3.3V (pulse)	0.2mA	HSMC-RX-n0 HSMC-TX-n0 HSMC-TX-n2 HSMC-TX-n1
	RXO	3.3V (pulse)	0.2mA	
	GND (HV)	0V	1uA	N/A
	GND (LV)	0V	0.02mA	N/A
Servo motor	Vcc	12V	5.3mA	N/A
	C1	3.3V	<1uA	HSMC-TX-n9

	C2	3.3V	0.2mA	HSMC-TX-p9
	GND	0V	5.3mA	N/A
Regular motor	C1	12V	<1uA	N/A
	C2	12V	21uA	N/A
L298N	12V	12V	300mA	N/A
	EN	3.3V	3uA	HSMC-RX-n16
	IN1	3.3V	<1uA	HSMC-RX-p16
	IN2	3.3V	<1uA	HSMC-RX-n15

Pin Measurement Table 1

Conclusion

The initial project goals have been accomplished. In fact, the robot can perform more parking operations than what we originally planned such as garage parking and automatic trace adjustment. Starting from project design, component selection, hardware/software implementation, testing and debugging ,we all benefit from this great learning experience and deepened our understanding of FPGA. In the broader picture, we plan to implement more features for robots to get more closer to human parking ability.

Acknowledgments

The team would like to thank Professor Stephen Edwards and our TA Qiushi for all their guidance and support throughout the semester.

Code Attachment

Matlab Code:

AutoParking.m

```
cla;
% This is a model for parallel auto parking, for more detailed
explanation,
% please refer to the algorithm part in the final report.

% Parameter for world Setting
global LOT_BACK; % the coordinate of the back side of the parking lot
global LOT_WIDTH; % the width of the parking lot
global LOT_LENGTH; % The length of the parking lot
global ST_LENGTH; % length of street
global ST_WITDTH; % width of street
global CAR_LENGTH; % length of the car
global CAR_WIDTH; % width of the car

CAR_LENGTH = 40;
CAR_WIDTH = 20;
LOT_BACK = 40;
LOT_WIDTH = 25;
LOT_LENGTH = 2*CAR_LENGTH; % Assume the parking is twice the length of
the car
ST_LENGTH = 200;
ST_WITDTH = 80;



% Parameter for My Car





% Initial Position setting



% The angle between y-axis and the direction of the car ( facing front
= > theta = 0)
theta = 0;

% parkPitch is the distance between the car and the wall
parkPitch = 10;

% iniCarY = initial Car Y coordiante
iniCarY = LOT_LENGTH + LOT_BACK;

% carCenter is a two dimentional vector that stores the coordinate of
the
% curren car position. We initially set that the back side of the car is
% align with the front side of the parking lot, but it can be set to any
% value and will not affect the parking.
```

```
carCenter = [0 + LOT_WIDTH + parkPitch, iniCarY] +  0.5.*[CAR_WIDTH,
CAR_LENGTH];

% Carplot is a function to plot the car
carPlot(carCenter(1),carCenter(2),theta,CAR_WIDTH,CAR_LENGTH);

%world is a function used to plot the street and wall.
world(LOT_WIDTH,LOT_BACK,LOT_LENGTH,ST_WITDTH,ST_LENGTH); % Plot the
world ( street and wall)


% Setting the turninig radius of the car
radius = 40;

% Calculate some intermediate value to put in the formular after. This
can make the formula looks more tidy
lotFront = LOT_BACK + LOT_LENGTH; % The Y-coordinate of the front of the
parking lot
lotRear = LOT_BACK; % The Y- coordinate of the rear of the parking lot
inRadius = radius - 0.5* CAR_WIDTH; % turning radius of the  inner side
of the car


% Calculate the the angle to turn when doing parallel parking, the
explanation is documented in the final report.
angleToTurn = acos((radius  - 0.5*(0.5*CAR_WIDTH +
parkPitch+radius+LOT_WIDTH-
sqrt((radius+0.5*CAR_WIDTH)^2+(0.5*CAR_LENGTH)^2)))/radius);

% overhead is an aggresive algorithm to make the parking car precisely
not
% hit the wall, it generate an output "out" and put it into distance to
% move
[phi,out,angle_p,angle_n] =
overhead(CAR_LENGTH,CAR_WIDTH,angleToTurn,parkPitch,radius);

% Calculate the distance the car needs to move before starting to turn
and
% parking
distanceToMove = sqrt(inRadius^2 - (inRadius - parkPitch)^2) + lotFront
- carCenter(2) - out   ;

% straight is a function that makes the car going forward or backward
while
% you can choose the distance you want, minus value correspond to move
backward
% further it returns a matrix carTrack that record the track of the
% car (coordiante)
[carCenter(1),carCenter(2),theta,carTrack]...
 =
straight(carCenter(1),carCenter(2),theta,distanceToMove,CAR_WIDTH,CAR_LE
NGTH,LOT_WIDTH,LOT_BACK,LOT_LENGTH,ST_WITDTH,ST_LENGTH);

% store the the data into matrix carTrajectory
```

```
carTraject = carTrack;

% turn is a function that makes the car turning while you can choose the
% angle to turn, positive radius corresponds to turn, and positive
% angle corresponds to go counter clockwise, also generate carTrack
[carCenter(1),carCenter(2),theta,carTrack]...
=turn(carCenter(1),carCenter(2),theta,-
(angleToTurn),radius,CAR_WIDTH,CAR_LENGTH,LOT_WIDTH,LOT_BACK,LOT_LENGTH,
ST_WITDTH,ST_LENGTH);

% store the the data into matrix carTrajectory
carTraject = [carTraject;carTrack];


% similarly from the code above, but this represent the second arc.
[carCenter(1),carCenter(2),theta,carTrack]...
=turn(carCenter(1),carCenter(2),theta,angleToTurn,-
radius,CAR_WIDTH,CAR_LENGTH,LOT_WIDTH,LOT_BACK,LOT_LENGTH,ST_WITDTH,ST_L
ENGTH);

% store the the data into matrix carTrajectory
carTraject = [carTraject;carTrack];



hold on;

% the program has utilize one straight and two curvy move to complete
the
% parallel parking. The trajectory of the car has been stored in the
matrix
% carTrajectory. carTrajectory is a matrix with 10 columes, storing five
2
% dimentional coordinate. The first two colume are for the center of the
% car, and the other 8 columes are for the four corners of the car.

% For example, to plot the trajectory of front-left corner of the car,
use the
% following:

plot(carTraject(:,5),carTraject(:,6),'ro');

% r means the plot it in red, o means use circle to plot


 World.m
function world(lotWidth,lotBack,lotLength,stWidth,stLength)

patch([0,0,lotWidth,lotWidth],[0,lotBack,lotBack,0],[0,0,0]); % Bottom
patch([0,0,lotWidth,lotWidth],[lotBack + lotLength, stLength, stLength,
lotBack + lotLength],[0,0,0]); % Top
patch([-10,0,0,-10],[0,0,stLength,stLength],[0,0,0]); %Wall

axis ([-10 stWidth 0 stLength],'equal');
```

```
end

Turn.m

function  [centerX,centerY,theta,carTrack] =
turn(centerX,centerY,theta,angle,radius,myCarWidth,myCarLength,lotWidth,
lotBack,lotLength,stWidth,stLength)

stepsize = pi/180;
centerX1 = centerX;
centerY1 = centerY;
theta1 = theta;
loopCount = 0;
carTrack = zeros(abs(angle)/stepsize,10);
if (radius >=0 && angle >= 0)
    for k = stepsize:stepsize:angle
            loopCount = loopCount +1 ;
            D = [-(1-cos(k)) sin(k)];
            Drot = D*[1;1i]*exp(1i*(theta1));
            theta = theta1 + k;
            centerX =  centerX1 + radius*real(Drot);
            centerY =  centerY1 + radius*imag(Drot);
            cla;
            carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
            world(lotWidth,lotBack,lotLength,stWidth,stLength);
            pause(0.01)
    end
            loopCount = loopCount + 1;
            D = [-(1-cos(angle)) sin(angle)];
            Drot = D*[1;1i]*exp(1i*(theta1));
            theta = theta1 + angle;
            centerX =  centerX1 + radius*real(Drot);
            centerY =  centerY1 + radius*imag(Drot);
            cla;
            carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
            world(lotWidth,lotBack,lotLength,stWidth,stLength);
            pause(0.01)

else if (radius>= 0 && angle < 0)
        angle = -angle;
        for k = stepsize:stepsize:angle
            loopCount = loopCount +1 ;
             D = [-(1-cos(k)) -sin(k)];
            Drot = D*[1;1i]*exp(1i*(theta1));
            theta = theta1 - k;
            centerX =  centerX1 + radius*real(Drot);
            centerY =  centerY1 + radius*imag(Drot);
            cla;
            carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
            world(lotWidth,lotBack,lotLength,stWidth,stLength);
            pause(0.01)
        end
            loopCount = loopCount + 1;
            D = [-(1-cos(angle)) -sin(angle)];
            Drot = D*[1;1i]*exp(1i*(theta1));
            theta = theta1 - angle;
            centerX =  centerX1 + radius*real(Drot);
            centerY =  centerY1 + radius*imag(Drot);
            cla;
            carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
            world(lotWidth,lotBack,lotLength,stWidth,stLength);
            pause(0.01)

    else if (radius < 0 && angle > 0)
            radius = - radius ;
            for k = stepsize:stepsize:angle
            loopCount = loopCount +1 ;
            D = [1-cos(k) -sin(k)];
            Drot = D*[1;1i]*exp(1i*(theta1));
            theta = theta1 + k;
            centerX =  centerX1 + radius*real(Drot);
            centerY =  centerY1 + radius*imag(Drot);
            cla;
            carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
            world(lotWidth,lotBack,lotLength,stWidth,stLength);
            pause(0.01)
            end
            loopCount = loopCount + 1;
            D = [1-cos(angle) -sin(angle)];
            Drot = D*[1;1i]*exp(1i*(theta1));
            theta = theta1 + angle;
            centerX =  centerX1 + radius*real(Drot);
            centerY =  centerY1 + radius*imag(Drot);
            cla;
            carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
            world(lotWidth,lotBack,lotLength,stWidth,stLength);
            pause(0.01)
    else if (radius < 0 && angle < 0)
            angle = -angle ;
            radius = -radius;
            for k = stepsize:stepsize:angle
             loopCount = loopCount +1 ;
             D = [(1-cos(k)) sin(k)];
            Drot = D*[1;1i]*exp(1i*(theta1));
            theta = theta1 - k;
            centerX =  centerX1 + radius*real(Drot);
            centerY =  centerY1 + radius*imag(Drot);
            cla;
            carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
            world(lotWidth,lotBack,lotLength,stWidth,stLength);
            pause(0.01)
            end
```

```
          loopCount = loopCount + 1;
          D = [1-cos(angle) sin(angle)];
          Drot = D*[1;1i]*exp(1i*(theta1));
          theta = theta1 - angle;
          centerX =  centerX1 + radius*real(Drot);
          centerY =  centerY1 + radius*imag(Drot);
          cla;
          carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
          world(lotWidth,lotBack,lotLength,stWidth,stLength);
          pause(0.01)
       end
       end
    end
end
end

 straight.m
function [centerX,centerY,theta,carTrack] =
straight(centerX,centerY,theta,distance,myCarWidth,myCarLength,lotWidth,
lotBack,lotLength,stWidth,stLength)


stepsize = 1;
centerX1 = centerX;
centerY1 = centerY;
loopCount = 0;
carTrack = zeros(distance/stepsize,10);
if distance >= 0
    for i = stepsize:stepsize:distance
        loopCount = loopCount + 1 ;
        centerX = centerX1 - i*sin(theta);
        centerY = centerY1 + i*cos(theta);
        cla;
        carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
        world(lotWidth,lotBack,lotLength,stWidth,stLength);
        pause(0.01)

    end
else
    distance = -distance;
    for i = stepsize:stepsize:distance
        loopCount = loopCount + 1 ;
        centerX = centerX1 + i*sin(theta);
        centerY = centerY1 - i*cos(theta);
        cla;
        carTrack(loopCount,:) =
carPlot(centerX,centerY,theta,myCarWidth,myCarLength);
        world(lotWidth,lotBack,lotLength,stWidth,stLength);
        pause(0.01)
    end
end
end
```

Overhead.m

```
function [ phi_out,out,angle_p,angle_n ] =
overhead(myCarLength,myCarWidth,theta,parkPitch,radius)

% This is a function to utilize an aggressive parking algorithm

phi_out = atan(myCarLength/(2*(radius + myCarWidth/2)));
phi_in = atan(myCarLength/(2*(radius - myCarWidth/2)));
rin_p = sqrt((myCarLength/2)^2 + (radius - myCarWidth/2)^2);
rout_p = sqrt((myCarLength/2)^2 + (radius + myCarWidth/2)^2);
phi_p = phi_out + theta;
A = (parkPitch + rin_p*(cos(phi_in - theta) - cos(phi_in)))/rout_p;
assignin('base','rout_p',rout_p);
assignin('base','A',A);
assignin('base','phi_p',phi_p);

b = -(A+cos(phi_p))*cos(phi_p);
c = (A+cos(phi_p))^2-(sin(phi_p))^2;
a = 1;
angle_p = acos((-b+sqrt(b^2-a*c))/a);
angle_n = acos((-b-sqrt(b^2-a*c))/a);
assignin('base','b',b);
assignin('base','c',c);


out = A*rout_p*cot(theta) - rout_p*((sin(phi_p) - sin(phi_p -
angle_p)));
end
```

CarPlot.m

```
% This is a function to plot a car with a given postion, width, length,
and
% direction. It also output the coordinates of the center and four
corners
% of the car

function [carData] = carPlot(centerX,centerY,direction,width,length)
colordef white;
rotationMatrix = [cos(direction), -sin(direction); +sin(direction),
cos(direction)];

RFCorner = 0.5.*[ width, length];
LFCorner = 0.5.*[ -width, length];
RRCorner = 0.5.*[ width, -length];
LRCorner = 0.5.*[ -width, -length];

RFCorner = (rotationMatrix*RFCorner')';
LFCorner = (rotationMatrix*LFCorner')';
RRCorner = (rotationMatrix*RRCorner')';
LRCorner = (rotationMatrix*LRCorner')';
```

```
carData =
[centerX,centerY,RFCorner+[centerX,centerY],LFCorner+[centerX,centerY],R
RCorner+[centerX,centerY],LRCorner+[centerX,centerY]];


patch ([LRCorner(1),RRCorner(1),RFCorner(1),LFCorner(1)] + centerX...
     ,[LRCorner(2),RRCorner(2),RFCorner(2),LFCorner(2)] + centerY...
     ,[0,0,1],'EraseMode','none');


end
```

Parking Robot SW C code:

`Software File Description:`

(1) Drivers:

steering_motor.c/h -> driver for servo/steering motor

regulrar_motor.c/h -> driver for regular motor

ultrasonic_sensor.c/h -> driver for ultrasonic sensors

driver_functions.c/h -> helper functions for all peripherals above.

socfpga.dts/dtsi -> device tree

(2) Main program:

parking_robot.c/h -> user interface for command input. It's also responsible dispatching the following thread:
(a) Data collecting thread -> used to collect the four ultrasonic sensors' data
(b) Auto parking thread -> the main functionality of this thread is to let the car drive along the wall and keep a certain distance (trace adjustment). A requirement for making the mechanism work correctly is that the car need to be within 30 cm from the wall when starting the thread. If you choose auto_parking -> parallel_parking in the user interface, it will drive along the wall, and will start parallel_parking when finding an enough parking space. Note that the parallel parking function it calls is in parallel_parking.c, not parallel_parking_independent.c

parallel_parking.c/h -> the parallel parking procedure that is able to collaborate with trace adjustment mechanism. This is modified from parallel_parking_independent.c

parallel_parking_independent.c/h -> this is the parallel parking procedure that can run itself without the help from auto parking thread (trace adjustment). However there's a possibility that it will go out of trace and then failed to park. Note that when running this function, you're not able to do other things since it's running on main thread.

garage_parking.c -> this is the garage parking procedure that can run itself without the help from auto parking thread (trace adjustment). However there's a possibility that it will go out of trace and then failed to park. Note that when running this function, you're not able to do other things since it's running on main thread.

To compile the drivers, please run:
make

To compile our program, please run:
gcc -pthread -o parking_robot parking_robot.c parallel_parking.c garage_parking.c driver_functions.c parking_robot_independent.c -lm

Driver Functions.c

```
#include "driver_functions.h"
#include "parameters.h"

#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>



static const char u_filename[] = "/dev/ultrasonic_sensor";
static const char r_filename[] = "/dev/regular_motor";
static const char s_filename[] = "/dev/steering_motor";


//resources for fpga peripherals
int regular_motor_fd, steering_motor_fd, ultrasonic_sensor_fd;


char open_drivers(){

  if ( (ultrasonic_sensor_fd = open(u_filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", u_filename);
    return FALSE;
  }

  if ( (regular_motor_fd = open(r_filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", r_filename);
    return FALSE;
  }

  if ( (steering_motor_fd = open(s_filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", s_filename);
    return FALSE;
  }

  return TRUE;

}

//=== functions for regular motor control ===
void write_r_motor_dir(const r_direction_t r_dir)
{
  if (ioctl(regular_motor_fd, R_MOTOR_SET_DIR, &r_dir)) {
     printf("ioctl(R_MOTOR_SET_DIR) faiball");
     return;
  }
}

void write_r_motor_speed(const __u32 speed)
{
  if (ioctl(regular_motor_fd, R_MOTOR_SET_SPEED, &speed)) {
     printf("ioctl(R_MOTOR_SET_SPEED) faiball");
     return;
  }
}

r_direction_t read_r_motor_dir()
{
  r_direction_t r_dir;
  if (ioctl(regular_motor_fd, R_MOTOR_GET_DIR, &r_dir)) {
     printf("ioctl(R_MOTOR_GET_DIR) faiball");
     return FORWARD; //randomly return one
  }

  return r_dir;
}

__u32 read_r_motor_speed()
{
  __u32 speed;
  if (ioctl(regular_motor_fd, R_MOTOR_GET_SPEED, &speed)) {
     printf("ioctl(R_MOTOR_GET_SPEED) faiball");
     return MAX_SPEED_LV + 1;//impossible value in real, so used as error
val
  }

  return speed;
}

//=== functions for steering motor control ===
void write_s_motor_dir(const s_direction_t s_dir)
{
  if (ioctl(steering_motor_fd, S_MOTOR_SET_DIR, &s_dir)) {
     printf("ioctl(S_MOTOR_SET_DIR) faiball");
     return;
  }
}

void write_s_motor_angle(const __u32 angle)
{
  if (ioctl(steering_motor_fd, S_MOTOR_SET_ANGLE, &angle)) {
     printf("ioctl(S_MOTOR_SET_ANGLE) faiball");
     return;
  }
}

s_direction_t read_s_motor_dir()
{
  s_direction_t s_dir;
  if (ioctl(steering_motor_fd, S_MOTOR_GET_DIR, &s_dir)) {
     printf("ioctl(S_MOTOR_GET_DIR) faiball");
     return RIGHT; //randomly return one
  }

  return s_dir;
}

__u32 read_s_motor_angle()
{
  __u32 angle;
  if (ioctl(steering_motor_fd, S_MOTOR_GET_ANGLE, &angle)) {
     printf("ioctl(S_MOTOR_GET_ANGLE) faiball");
     return MAX_ANGLE_LV + 1;//impossible value in real, so used as error
val
  }

  return angle;
}

//=== functions for ultrasonic sensor control ===
__u32 read_ultrasonic_distance(us_position_t pos)
{
  ultrasonic_t us;
  us.pos = pos;

  if (ioctl(ultrasonic_sensor_fd, ULTRASONIC_GET_DIST_WITH_POS, &us)) {
     printf("ioctl(ULTRASONIC_GET_DIST_WITH_POS) faiball");
     return 0;//impossible value in real, so used as error val
  }

  //since the position of us_RF and us_RB is not exactly the same
  if(us.pos == RF)
    us.distance = us.distance - US_RF_OFFSET;

  return us.distance;
}

//conver cycles from read_ultrasonic_distance to cm value
float cycles_to_cm(__u32 cycles){
  return ((float)cycles) * 340 * 100 / 2 / 50000000;
}
```

garage_parking.c

```
/*
 * Parking Robot Team
 */

 /* This is a automatic garage parking system, it is very similar to
parallel_parking_independent .c
    Their code are the same as parallel_parking_independent.c until Ln
150
 */



#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

#include "parameters.h"
#include "driver_functions.h"
#include "independent_parking.h"





void garage_parking()
{




 // Auto parking parameter
int state, adjust;
float angleToTurn;
float parkPitch;
float distanceToMove;
float lotWidth,lotLength;
float CURF, CURB, CUBR, CUFR, CURF_MAX, CURB_MAX;
float turnPosition;


printf("Garage parking started\n");
```

```
// Initializing, please refer to parallel_parking_independent.c

  state = 0;
  CURF = cycles_to_cm(read_ultrasonic_distance(RF));
  CURB = cycles_to_cm(read_ultrasonic_distance(RB));
  parkPitch = ( CURF + CURB )/ 2 ;


  write_r_motor_dir(FORWARD);
  write_r_motor_speed(0);

  write_s_motor_dir(RIGHT);
  write_s_motor_angle(0);

    lotWidth = 0;
    CURF_MAX = 0;
    CURB_MAX = 0;


// garage parking starts


  while(TRUE) {


    if (state == 0){

    write_r_motor_dir(FORWARD);
    write_r_motor_speed(1);

    write_s_motor_dir(RIGHT);
    write_s_motor_angle(0);
    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
        if (CURF > parkPitch + CAR_WIDTH + 10){
        state = 1;
        }

    }
    if (state == 1){

    write_r_motor_dir(FORWARD);
    write_r_motor_speed(1);

    write_s_motor_dir(RIGHT);
    write_s_motor_angle(0);
    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
    CURB = cycles_to_cm(read_ultrasonic_distance(RB));
    if (CURF > CURF_MAX){
    CURF_MAX = CURF;
    }
    if (CURB > CURB_MAX){
    CURB_MAX = CURB;
    }

    lotWidth = (CURF_MAX+CURB_MAX)/2 - parkPitch;
            usleep(200000);
        if (abs(parkPitch - CURF) < 30 && abs(parkPitch - CURB) < 30 ){
        state = 2;
        }
    }
    if (state == 2){
    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
    CURB = cycles_to_cm(read_ultrasonic_distance(RB));
    parkPitch = ( CURF + CURB )/ 2 ;



    write_r_motor_dir(FORWARD);
    write_r_motor_speed(0);

    write_s_motor_dir(RIGHT);
    write_s_motor_angle(0);

    angleToTurn = acos((RADIUS - 0.5*( CAR_WIDTH/2 + parkPitch + RADIUS
+  lotWidth  -  RIGHT_MARGIN  -  sqrt((RADIUS  +  CAR_WIDTH/2)*(RADIUS  +
CAR_WIDTH/2) +(CAR_LENGTH/2)*(CAR_LENGTH/2))) )/RADIUS);

    distanceToMove = - CAR_LENGTH/2 + sqrt((RADIUS - CAR_WIDTH/2)*(RADIUS
- CAR_WIDTH/2) - (RADIUS -CAR_WIDTH/2 -parkPitch)*(RADIUS -CAR_WIDTH/2 -
parkPitch) ) - 5 ;



    turnPosition  =  (0.5*CAR_WIDTH  +  0.5*(  lotWidth  +  parkPitch)  +
0.5*(sqrt((RADIUS   +   0.5*CAR_WIDTH)*(RADIUS   +   0.5*CAR_WIDTH)   +
(0.5*CAR_LENGTH)*(0.5*CAR_LENGTH))          -          (RADIUS          +
0.5*CAR_WIDTH)))/(sin(angleToTurn))       -       0.5*CAR_LENGTH       -
ULTRA_POSITION/(tan(angleToTurn));

            if (distanceToMove >=0){
                write_r_motor_dir(FORWARD);
            }
            else{
                write_r_motor_dir(BACKWARD);
            }
            write_r_motor_speed(1);
            usleep(STEERING_WAIT_TIME
*(int)(distanceToMove/STRAIGHT_SPEED));

            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);

                write_s_motor_dir(RIGHT);
                write_s_motor_angle(7);
            usleep(STEERING_WAIT_TIME);

            write_r_motor_dir(BACKWARD);
            write_r_motor_speed(1);
            CURF = cycles_to_cm(read_ultrasonic_distance(RF));
            CURB = cycles_to_cm(read_ultrasonic_distance(RB));

            while(abs(CURF - CURB) < 10){
                usleep(ULTRASONICS_READ_INTERVAL);
                CURF = cycles_to_cm(read_ultrasonic_distance(RF));
                CURB = cycles_to_cm(read_ultrasonic_distance(RB));
                CUBR = cycles_to_cm(read_ultrasonic_distance(BR));
            }

            // Try to pull the car straight, unless reach the end of the
parking lot
            while(abs(CURF - CURB) > 2){

                CUBR = cycles_to_cm(read_ultrasonic_distance(BR));
                if (CUBR < 7){
                    printf("Parking completed at stage 2, the car may not
be completely straight, CUBR = %f", CUBR);
                break;
                }
                usleep(ULTRASONICS_READ_INTERVAL);
                CURF = cycles_to_cm(read_ultrasonic_distance(RF));
                CURB = cycles_to_cm(read_ultrasonic_distance(RB));
            }
            while(CUBR > 15){
                CUBR = cycles_to_cm(read_ultrasonic_distance(BR));
                usleep(ULTRASONICS_READ_INTERVAL);

            }

            write_s_motor_dir(LEFT);
            write_s_motor_angle(0);
            state=3;




    }
    if(state==3){

            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);

            write_s_motor_dir(LEFT);
            write_s_motor_angle(0);
            printf("Garage parking finished\n");

            return;



    }

  }
```

```
}

Parallel_Parking_Indepent.c

/*
 * Parking Robot Team
 * Columbia University
 */



#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

#include "parameters.h"
#include "driver_functions.h"
#include "independent_parking.h"




void parallel_parking_independent()
{



 // Auto parking parameter

int state, adjust, counter;
float angleToTurn;
float parkPitch;
float distanceToMove;
float lotWidth,lotLength;
float CURF, CURB, CUBR, CUFR, CURF_MAX, CURB_MAX;
float turnPosition;

 printf("Parallel parking started\n");


 // Initializing parameter

  state = 0;
  CURF = cycles_to_cm(read_ultrasonic_distance(RF));
  CURB = cycles_to_cm(read_ultrasonic_distance(RB));
  // The distance between the car and the wall
  parkPitch = ( CURF + CURB )/ 2 ;


  write_r_motor_dir(FORWARD);
  write_r_motor_speed(0);

  write_s_motor_dir(RIGHT);
  write_s_motor_angle(0);

    lotWidth = 0;
    CURF_MAX = 0;
    CURB_MAX = 0;



// Start parallel parking

  while(TRUE) {


    if (state == 0){


    // Go forward with speed level "1"
    write_r_motor_dir(FORWARD);
    write_r_motor_speed(1);

    // Set servo motor angle to zero, meaning to go straight.
    write_s_motor_dir(RIGHT);
    write_s_motor_angle(0);
    // CURF = Current Ultrasonic Value for rightside - front ultrasonic
    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
        if (CURF > parkPitch + 10){
        // If CURF gets larger than parkpitch, meaning that parking lot
hase been detected, then go to the next state
        state = 1;
        }

    }
    if (state == 1){

    write_r_motor_dir(FORWARD);
    write_r_motor_speed(1);

    write_s_motor_dir(RIGHT);
    write_s_motor_angle(0);
    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
    CURB = cycles_to_cm(read_ultrasonic_distance(RB));

    // Find the maximum distance detected to calculate the width of the
parking lot
    if (CURF > CURF_MAX){
```

```
    CURF_MAX = CURF;
    }
    if (CURB > CURB_MAX){
    CURB_MAX = CURB;
    }
    // Width of parking lot = maximum distance detected (average for two
rightside ultrasonic) - parkPitch
    lotWidth = (CURF_MAX+CURB_MAX)/2 - parkPitch;
            usleep(200000);
        // If CURF and CURB get closer to parkPitch, then the car has
reach the end of the parking lot
        if (abs(parkPitch - CURF) < 30 && abs(parkPitch - CURB) < 30 ){
        state = 2;
        }
    }
    if (state == 2){
    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
    CURB = cycles_to_cm(read_ultrasonic_distance(RB));

    // Calculate parkPitch again to calibrate, because the car may deviate
during previous state.
    parkPitch = ( CURF + CURB )/ 2 ;



    write_r_motor_dir(FORWARD);
    write_r_motor_speed(0);

    write_s_motor_dir(RIGHT);
    write_s_motor_angle(0);

    // Auto parking start here

    // Calculate angle and distance for parking

    angleToTurn = acos((RADIUS - 0.5*( CAR_WIDTH/2 + parkPitch + RADIUS
+ lotWidth - RIGHT_MARGIN - sqrt((RADIUS + CAR_WIDTH/2)*(RADIUS +
CAR_WIDTH/2) +(CAR_LENGTH/2)*(CAR_LENGTH/2))) )/RADIUS);

    distanceToMove = - CAR_LENGTH/2 + sqrt((RADIUS - CAR_WIDTH/2)*(RADIUS
- CAR_WIDTH/2) - (RADIUS -CAR_WIDTH/2 -parkPitch)*(RADIUS -CAR_WIDTH/2 -
parkPitch) ) -5 ;


    turnPosition = (0.5*CAR_WIDTH + 0.5*( lotWidth + parkPitch) +
0.5*(sqrt((RADIUS + 0.5*CAR_WIDTH)*(RADIUS + 0.5*CAR_WIDTH) +
(0.5*CAR_LENGTH)*(0.5*CAR_LENGTH)) - (RADIUS +
0.5*CAR_WIDTH)))/(sin(angleToTurn)) - 0.5*CAR_LENGTH -
ULTRA_POSITION/(tan(angleToTurn));




            if (distanceToMove >= 0 ){
                write_r_motor_dir(FORWARD);

            }
            else{
                write_r_motor_dir(BACKWARD);

            }

            // Go with speed level 1
            write_r_motor_speed(1);
            usleep(STEERING_WAIT_TIME
*(int)(distanceToMove/STRAIGHT_SPEED));

            // Stop
            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);

            // Turn the wheels to the right max

            write_s_motor_dir(RIGHT);
            write_s_motor_angle(7);
            usleep(STEERING_WAIT_TIME);

            // Moving backward toward the parking lot

            write_r_motor_dir(BACKWARD);
            write_r_motor_speed(1);
            CUBR =  cycles_to_cm(read_ultrasonic_distance(BR));
            while(CUBR > 20){
                usleep(ULTRASONICS_READ_INTERVAL);
                CUBR =  cycles_to_cm(read_ultrasonic_distance(BR));
            }

            write_r_motor_dir(FORWARD);
            while(CUBR < turnPosition - 5 ){
                usleep(ULTRASONICS_READ_INTERVAL);
                CUBR = cycles_to_cm(read_ultrasonic_distance(BR));
            }



            //stop
            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);

            //to turn the wheels to the left max
            write_s_motor_dir(LEFT);
            write_s_motor_angle(7);
            usleep(STEERING_WAIT_TIME);

            write_r_motor_dir(BACKWARD);
            write_r_motor_speed(1);
            CURF = cycles_to_cm(read_ultrasonic_distance(RF));
            CURB = cycles_to_cm(read_ultrasonic_distance(RB));
            // adjust is a state for adjusting the car position while the
car is moving through the second arc.
            // For adjust = 0, the car is now moving backward and turning
left.
            // For adjust = 1, the car is moving forward and turning right.
            // In each one states, the car will try to park parallel with
the wall (CURF == CURB)
            // But if there is a near obstacle, it will jump to another
state and try to keep park parallel.

            adjust = 0;

            // This counter is for checking CURF == CURB, the equal is
recognize only when it has been detected twice.
            counter = 0;
            while(abs(CURF - CURB) >= 4 || (counter < 2) ){
                if (abs(CURF - CURB) >= 2){
                    counter = 0;
                }
                while (adjust == 0){

                    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
                    CURB = cycles_to_cm(read_ultrasonic_distance(RB));
                    CUBR = cycles_to_cm(read_ultrasonic_distance(BR));

                    // abs(CURF - CURB) < 2 means they are equal. But we
want them to be less than 30,
                    // to make sure we did't measure the distance from an
irrelevent, far away place.
                    if ( abs(CURF - CURB) < 2 && CURF < 30 && CURB < 30){
                        counter ++;
                        usleep(ULTRASONICS_READ_INTERVAL);
                        break;
                    }
                    else {
                        // if the obstacle is too close, switch to another
adjust
                        if ( CUBR < 15){
                            //stop
                            write_r_motor_dir(FORWARD);
                            write_r_motor_speed(0);
                            //to turn the wheels to the right max
                            write_s_motor_dir(RIGHT);
                            write_s_motor_angle(7);
                            usleep(STEERING_WAIT_TIME);
                            //forward right
                            write_r_motor_dir(FORWARD);
                            write_r_motor_speed(1);
                            adjust = 1;
                            counter = 0;
                        }
                        else{
                            usleep(ULTRASONICS_READ_INTERVAL);
                        }
                    }
                }
                while (adjust == 1){
                    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
                    CURB = cycles_to_cm(read_ultrasonic_distance(RB));
                    CUFR = cycles_to_cm(read_ultrasonic_distance(FR));
                    if ( abs(CURF - CURB) < 2 && CURF < 30 && CURB < 30 ){
                        counter ++;
                        usleep(ULTRASONICS_READ_INTERVAL);
                        break;
                    }
                    else {
                        if ( CUFR < 15){
                            //stop
                            write_r_motor_dir(FORWARD);
                            write_r_motor_speed(0);
                            //to turn the wheels to the left max
                            write_s_motor_dir(LEFT);
                            write_s_motor_angle(7);
                            usleep(STEERING_WAIT_TIME);
                            //backward left
                            write_r_motor_dir(BACKWARD);
                            write_r_motor_speed(1);
                            adjust = 0;
                            counter = 0;
                            }
                        else{
                            usleep(ULTRASONICS_READ_INTERVAL);
                            }
                    }
                }
            }
            //Stop the steering motor and go to state 3
            //
            write_s_motor_dir(LEFT);
            write_s_motor_angle(0);
            state=3;

            }
    if(state==3){
            // Stop the car and finish the parking
            //stop the regular motor
            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);
            //to stop the steering motor
            write_s_motor_dir(LEFT);
            write_s_motor_angle(0);
            printf("Parallel parking finished\n");

            return;


    }
    }

}
```

```
Parallel Parking.c

/*
 * Parking Robot Team
 * Columbia University
 */



#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

#include "parameters.h"
#include "driver_functions.h"
#include "parallel_parking.h"

void parallel_parking(float parkPitch, float lotWidth){

    int state, adjust;
    float angleToTurn;
    //float parkPitch;
    float distanceToMove;
    float phi_in,phi_out,rin_p, phi_p,rout_p,out;
    //float a,b,c,A,angle_p,angle_n;
    //float speed;
    float CURF, CURB, CUBR, CUFR;
    float turnPosition;
    int counter;




    printf("Parallel parking started\n");



    //Initialize
    state = 2;
    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
    CURB = cycles_to_cm(read_ultrasonic_distance(RB));
    //parkPitch = ( CURF + CURB )/ 2 ;
    //lotWidth = 0;

    write_r_motor_dir(FORWARD);
    write_r_motor_speed(0);
    write_s_motor_dir(RIGHT);
    write_s_motor_angle(0);
    while(TRUE) {




        if (state == 2){
            CURF = cycles_to_cm(read_ultrasonic_distance(RF));
            CURB = cycles_to_cm(read_ultrasonic_distance(RB));
            //parkPitch = ( CURF + CURB )/ 2 ;


            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);

            write_s_motor_dir(RIGHT);
            write_s_motor_angle(0);

            // Auto parking start here
            printf("parkPitch = %f\n", parkPitch);
            printf("lotWidth = %f\n", lotWidth);


            angleToTurn = acos((RADIUS - 0.5*( CAR_WIDTH/2 + parkPitch +
RADIUS + lotWidth - RIGHT_MARGIN - sqrt((RADIUS + CAR_WIDTH/2)*(RADIUS +
CAR_WIDTH/2) +(CAR_LENGTH/2)*(CAR_LENGTH/2))) )/RADIUS);

            distanceToMove   =   -   CAR_LENGTH/2   +   sqrt((RADIUS   -
CAR_WIDTH/2)*(RADIUS   -   CAR_WIDTH/2)   -   (RADIUS   -CAR_WIDTH/2   -
parkPitch)*(RADIUS -CAR_WIDTH/2 -parkPitch) ) ;


            turnPosition = (0.5*CAR_WIDTH + 0.5*( lotWidth + parkPitch)
+  0.5*(sqrt((RADIUS  +  0.5*CAR_WIDTH)*(RADIUS  +  0.5*CAR_WIDTH)  +
(0.5*CAR_LENGTH)*(0.5*CAR_LENGTH))         -         (RADIUS         +
0.5*CAR_WIDTH)))/(sin(angleToTurn))       -      0.5*CAR_LENGTH       -
ULTRA_POSITION/(tan(angleToTurn));

            printf("angleToTurn = %f\n", angleToTurn );

            printf("turnPosition = %f\n", turnPosition );
            printf("  Y   =  %f\n",  (0.5*CAR_WIDTH  +  0.5*(  lotWidth  +
parkPitch) + 0.5*(sqrt((RADIUS + 0.5*CAR_WIDTH)*(RADIUS + 0.5*CAR_WIDTH)
+ (0.5*CAR_LENGTH)*(0.5*CAR_LENGTH)) - (RADIUS + 0.5*CAR_WIDTH))));

            printf("distance to move = %f\n",distanceToMove);


            if (distanceToMove >= 0 ){
                write_r_motor_dir(FORWARD);
            }
            else{
                write_r_motor_dir(BACKWARD);

            }
            write_r_motor_speed(1);
            usleep(STEERING_WAIT_TIME
*(int)(distanceToMove/STRAIGHT_SPEED));// go forward
            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);//stop

            write_s_motor_dir(RIGHT);
            write_s_motor_angle(7);//to turn the wheels to the right max
            usleep(STEERING_WAIT_TIME);

            write_r_motor_dir(BACKWARD);
            write_r_motor_speed(1);
            CUBR =  cycles_to_cm(read_ultrasonic_distance(BR));
            while(CUBR > 20){
                usleep(ULTRASONICS_READ_INTERVAL);
                CUBR =  cycles_to_cm(read_ultrasonic_distance(BR));
            }

            write_r_motor_dir(FORWARD);
            while(CUBR < turnPosition - 5 ){
                usleep(ULTRASONICS_READ_INTERVAL);
                CUBR = cycles_to_cm(read_ultrasonic_distance(BR));
                //printf("CUBR = %f\n",CUBR);
            }

            //usleep(STEERING_WAIT_TIME*
(int)(abs(angleToTurn*RADIUS/TURN_SPEED)));// to go backward

            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);//stop

            write_s_motor_dir(LEFT);
            write_s_motor_angle(7);//to turn the wheels to the left max
            usleep(STEERING_WAIT_TIME);

            write_r_motor_dir(BACKWARD);
            write_r_motor_speed(1);
            CURF = cycles_to_cm(read_ultrasonic_distance(RF));
            CURB = cycles_to_cm(read_ultrasonic_distance(RB));
            printf("adjust preread CURF = %f, CURB = %f\n", CURF,CURB);
            adjust = 0;
            counter = 0;
            while(abs(CURF - CURB) >= 4 || (counter < 2) /* can set by
user*/){
                printf("adjust W1 CURF = %f, CURB = %f\n", CURF,CURB);
                if (abs(CURF - CURB) >= 2){
                    counter = 0;
                }
                while (adjust == 0){
                    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
                    CURB = cycles_to_cm(read_ultrasonic_distance(RB));
                    CUBR = cycles_to_cm(read_ultrasonic_distance(BR));
                    printf("adjust W2 adjust0 CURF = %f, CURB = %f\n",
CURF,CURB);
                    if ( abs(CURF - CURB) < 1 && CURF < 30 && CURB < 30/*
can set by user*/){
                        counter ++;
                        printf("counter = %d\n",counter);
                        usleep(ULTRASONICS_READ_INTERVAL);
                        break;
                    }
                    else {
                        if ( CUBR < 15/* can set by user*/){

                            write_r_motor_dir(FORWARD);
                            write_r_motor_speed(0);//stop

                            write_s_motor_dir(RIGHT);
                            write_s_motor_angle(7);
                            usleep(STEERING_WAIT_TIME);//to   turn   the
wheels to the right max

                            write_r_motor_dir(FORWARD);
                            write_r_motor_speed(1);//forward right
                            adjust = 1;
                            counter = 0;
                            printf("counter reset\n");
                        }
                        else{
                            usleep(ULTRASONICS_READ_INTERVAL);
                            printf("break 0.26s\n");
                        }
                    }
                }
                while (adjust == 1){
                    CURF = cycles_to_cm(read_ultrasonic_distance(RF));
                    CURB = cycles_to_cm(read_ultrasonic_distance(RB));
                    CUFR = cycles_to_cm(read_ultrasonic_distance(FR));
                    printf("adjust W2 adjust1 CURF = %f, CURB = %f\n",
CURF,CURB);
                    if ( abs(CURF - CURB) < 1 && CURF < 30 && CURB < 30
/* can set by user*/){
                        counter ++;
                        printf("counter = %d\n",counter);
                        usleep(ULTRASONICS_READ_INTERVAL);
                        break;
                    }
                    else {
                        if ( CUFR < 12/* can set by user*/){

                            write_r_motor_dir(FORWARD);
                            write_r_motor_speed(0);//stop
                            write_s_motor_dir(LEFT);
                            write_s_motor_angle(7);
                            usleep(STEERING_WAIT_TIME);//to   turn   the
wheels to the right max

                            write_r_motor_dir(BACKWARD);
                            write_r_motor_speed(1);//backward left
                            adjust = 0;
                            counter = 0;
                            printf("counter reset");
                        }
                        else{
                            usleep(ULTRASONICS_READ_INTERVAL);
                            printf("break 0.26s\n");
                        }
                    }
                }
            }
            //usleep(STEERING_WAIT_TIME
*(int)(abs(angleToTurn*RADIUS/TURN_SPEED)));

            write_s_motor_dir(LEFT);
            write_s_motor_angle(0);//to stop the steering motor
            state=3;

            }
    if(state==3){

            write_r_motor_dir(FORWARD);
            write_r_motor_speed(0);//stop the regular motor

            write_s_motor_dir(LEFT);
            write_s_motor_angle(0);//to stop the steering motor
            printf("Parallel parking finished\n");
            //pthread_join(data_collector_thread, NULL);
            return;


    }
    }

}
```

```
Parking Robot.c

/*
 * Parking Robot Team
 * Columbia University
 */


#include "parameters.h"
#include "driver_functions.h"
#include "parallel_parking.h"
#include "garage_parking.h"
#include "independent_parking.h"

#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>



//for data collect thread
typedef struct{
    char isRecording;
    int timeToRecord;
} data_collector_t;





//============= Function declaration ===================
//Check whether is in specified distance, and adjust servo motor if not
void keep_in_dist(const float keep_dist, const float keep_dist_max, const
float keep_dist_min,
                  const float tolerable_delta_f_b,
                  const float us_R_dist, const float us_R_reverse);


//====== thread functions ======
void *data_collect_thread_f(void *);
void *auto_parking_thread_f(void *);
//=============================

int main()
{
  //=========== threads ===========
  /* Used to collect data */
  pthread_t data_collect_thread ;
  pthread_t auto_parking_thread;
  //================================



  r_direction_t r_dir;
  s_direction_t s_dir;
  __u32 speed, angle, distance;
  char cmd;
  int tmpDir;
  int timeToRecord;
  int parking_mode;
  data_collector_t data_collector_arg;

  int thread_condition;

  char had_auto_parking_created = FALSE;
  char quit = FALSE;

  printf("Parking Robot program started\n");

  if(!open_drivers()){
    puts("Open drivers failed! Program stopped.");
    return -1;
  }

  //initlize(Maybe should put some in driver in init)
  //ball_pos.x = X_RANGE/2;
  //ball_pos.y = Y_RANGE/2;
  r_dir = FORWARD;
  speed = 0;
  write_r_motor_dir(r_dir);
  write_r_motor_speed(speed);
  s_dir = RIGHT;
  angle = 0;
  write_s_motor_dir(s_dir);
  write_s_motor_angle(angle);
  data_collector_arg.isRecording = FALSE;

  //start to cmd loop
  while(!quit) {

    //usleep(20000); //controll how fast the ball runs, lower value ->
run faster
    puts("");
    puts(SEPERATE_BAR_1);
    printf("What  to  do?\tr:  r_motor\ts:  s_motor\tu:  ultrasonic\td:
record_data\na:          auto_parking\tp:          parallel_parking\tg:
garage_parking\ni:status\te: emergency_stop\tq: quit \n");
    scanf ("%c", &cmd);
    switch (cmd){
        //Emergemcy stop (set speed to 0, but all other states remains)
        case 'e':
            write_r_motor_speed(0);
            break;

        //start a thread to colloect data for n sec, at most 30sec
        case 'd':
            puts("");
            printf("Please enter how many seconds you want to record
data(at most 100sec):\n");
            scanf("%d", &timeToRecord);
            if(!data_collector_arg.isRecording){
                if(timeToRecord <= 0 || timeToRecord > 100){
                    puts("Invalid value. It should be 0~100.");
                    continue;
                }
                data_collector_arg.isRecording = TRUE;
                data_collector_arg.timeToRecord =  timeToRecord;
                pthread_create(&data_collect_thread,              NULL,
data_collect_thread_f, (void *)&data_collector_arg);
            }else{
                printf("It's  already  recording,  please  try  again
later.\n");
            }
            break;
        case 'r':
            puts("");
            printf("Please enter direction(0: FORWARD, 1: BACKWARD) and
speed(0~8).\n");
            scanf("%d %d", &tmpDir, &speed);
            r_dir = (tmpDir == 0) ? FORWARD : BACKWARD;
            write_r_motor_dir(r_dir);
            write_r_motor_speed(speed);
            break;
        case 's':
            puts("");
            printf("Please  enter  direction(0:  RIGHT,  1:  LEFT)  and
angle(0~7).\n");
            scanf("%d %d", &tmpDir, &angle);
            s_dir = (tmpDir == 0) ? RIGHT : LEFT;
            write_s_motor_dir(s_dir);
            write_s_motor_angle(angle);
            break;

        //fire up/cancel auto parking mechanism
        case 'a':
            puts("");

            if(!had_auto_parking_created){
                printf("Please enter the parking mode (0: go_along_wall,
1: parallel_parking):\n");
                scanf("%d", &parking_mode);
                if(parking_mode < 0 || parking_mode > 1){
                    puts("Invalid value. It should be 0~1.");
                    continue;
                }
                puts("Starting auto parking...");
                pthread_create(&auto_parking_thread,              NULL,
auto_parking_thread_f, (void *)&parking_mode);
                had_auto_parking_created = TRUE;

            }else{
                //used to check wheter the thread is running
                thread_condition = pthread_kill(auto_parking_thread, 0);
                if(thread_condition == EINVAL){ //shouldn't occur indeed

                    puts("Invalid signal for pthread_kill");

                }else if(thread_condition == ESRCH){
                    printf("Please   enter   the   parking   mode   (0:
go_along_wall, 1: parallel_parking):\n");
                    scanf("%d", &parking_mode);
                    if(parking_mode < 0 || parking_mode > 1){
                        puts("Invalid value. It should be 0~1.");
                        continue;
                    }

                    puts("Starting auto parking...");
                    pthread_create(&auto_parking_thread,           NULL,
auto_parking_thread_f, (void *)&parking_mode);

                }else{

                    puts("Trying to stop auto parking ...");
                    /* Terminate the auto_parking_thread */
                    pthread_cancel(auto_parking_thread);
                    /* Wait for the auto_parking_thread to finish */
                    pthread_join(auto_parking_thread, NULL);
                    puts("Stopped");
                }
            }
            break;

        case 'g':
            garage_parking();
            break;

        case 'p':
            parallel_parking_independent();
            break;

        //print out all info
        case 'i':
            puts("");
            if(!had_auto_parking_created){

                printf("Auto parking status: off\n");

            }else{
                thread_condition = pthread_kill(auto_parking_thread, 0);

                if(thread_condition == ESRCH){

                    printf("Auto parking status: off\n");

                }else if(thread_condition == EINVAL){

                    printf("Auto parking status: unknown\n");

                }else{

                    printf("Auto   parking   status:   on,   mode:   %d\n",
parking_mode);

                }


            }


            printf("Data        recording        status:        %s\n",
data_collector_arg.isRecording ? "on" : "off" );

            printf("Regular  motor:  dir  is  %s,  speed  is  %d\n",
(read_r_motor_dir()==FORWARD)?"FORWARD":"BACKWARD",
                                                                   read_r_m
otor_speed());

            printf("Steering  motor:  dir  is  %s,  angle  is  %d\n",
(read_s_motor_dir()==RIGHT)?"RIGHT":"LEFT",
                                                                   read_s_m
otor_angle());
        case 'u':
            puts("");
            printf("Ultrasonic     FR     distance:     %.2f     cm\n",
cycles_to_cm(read_ultrasonic_distance(FR)));
            printf("Ultrasonic     RF     distance:     %.2f     cm\n",
cycles_to_cm(read_ultrasonic_distance(RF)));
            printf("Ultrasonic     RB     distance:     %.2f     cm\n",
cycles_to_cm(read_ultrasonic_distance(RB)));
            printf("Ultrasonic     BR     distance:     %.2f     cm\n",
cycles_to_cm(read_ultrasonic_distance(BR)));

            break;

        case 'q':
            quit = TRUE;
            break;

        default:;
```

```
    }


  }

  if(had_auto_parking_created){

    thread_condition = pthread_kill(auto_parking_thread, 0);

    if(thread_condition != ESRCH && thread_condition != EINVAL){
        puts("Trying to stop auto parking ...");
        /* Terminate the auto_parking_thread */
        pthread_cancel(auto_parking_thread);
        /* Wait for the auto_parking_thread to finish */
        pthread_join(auto_parking_thread, NULL);
        puts("Stopped");
    }
  }
  //Todo: close data collect thread
  write_r_motor_speed(0);
  write_s_motor_angle(0);
  printf("Parking robot terminated\n");
  return 0;
}

//data collect thread function
//Todo : add pthread_cleanup_push and pop when have time
void *data_collect_thread_f(void *arg)
{

  time_t start_time, end_time, tmp_time;
  data_collector_t *data_collector_arg;
  FILE *us_FR, *us_RF, *us_RB, *us_BR;

  data_collector_arg = (data_collector_t *)arg;

  //Get current time
  start_time = time(0);

  puts(SEPERATE_BAR_2);
  printf("Start to record data at ");
  puts(SEPERATE_BAR_2);
  printf(ctime(&start_time));

  //open files
  us_FR = fopen("us_FR.txt","w+");
  us_RF = fopen("us_RF.txt","w+");
  us_RB = fopen("us_RB.txt","w+");
  us_BR = fopen("us_BR.txt","w+");
  if(!us_FR){
    puts("Can't open us_FR.txt");
    goto end_thread;
  }
  if(!us_RF){
    puts("Can't open us_RF.txt");
    goto end_thread;
  }
  if(!us_RB){
    puts("Can't open us_RB.txt");
    goto end_thread;
  }
  if(!us_BR){
    puts("Can't open us_BR.txt");
    goto end_thread;
  }


  while(difftime(time(NULL),                start_time)               <=
data_collector_arg->timeToRecord){
    tmp_time = time(0);

    fprintf(us_FR,                                            "%.2f\t%s",
cycles_to_cm(read_ultrasonic_distance(FR)), ctime(&tmp_time));
    fprintf(us_RF,                                            "%.2f\t%s",
cycles_to_cm(read_ultrasonic_distance(RF)), ctime(&tmp_time));
    fprintf(us_RB,                                            "%.2f\t%s",
cycles_to_cm(read_ultrasonic_distance(RB)), ctime(&tmp_time));
    fprintf(us_BR,                                            "%.2f\t%s",
cycles_to_cm(read_ultrasonic_distance(BR)), ctime(&tmp_time));

    //fprintf(us_FR, "%.2f\n", (float)read_ultrasonic_distance(FR) * 340
* 100 /2 / 50000000);
    //fprintf(us_RF, "%.2f\n", (float)read_ultrasonic_distance(RF) * 340
* 100 /2 / 50000000);
    //fprintf(us_RB, "%.2f\n", (float)read_ultrasonic_distance(RB) * 340
* 100 /2 / 50000000);
    //fprintf(us_BR, "%.2f\n", (float)read_ultrasonic_distance(BR) * 340
* 100 /2 / 50000000);

    //sleep for specified time
    usleep(ULTRASONICS_READ_INTERVAL);
  }



  end_time = time(0);
  puts(SEPERATE_BAR_2);
  printf("Finished recording at ");
  puts(SEPERATE_BAR_2);
  printf(ctime(&end_time));

end_thread:
  if(us_FR)
    fclose(us_FR);
  if(us_RF)
    fclose(us_RF);
  if(us_RB)
    fclose(us_RB);
  if(us_BR)
    fclose(us_BR);
  data_collector_arg->isRecording = FALSE;
  return NULL;
}

//used to auto parking
void *auto_parking_thread_f(void *arg)
{
    puts(SEPERATE_BAR_2);
    puts("Auto parking started");
    puts(SEPERATE_BAR_2);

    //keep the certain distance from wall
    static const float short_dist = 10;
    //need a tolerate range since ultrasonic value is not always exactly
the same
    static const float tolerable_range = 0.25;
    //max and min   (need to be modify according to above values)
    static const float short_dist_max = 10.25;
    static const float short_dist_min = 9.75;

    //define what value can be said it's a obstacle in long
    //area.
    static const float gap_leng = 3;
    //define after how many cycle we say we see a gap
    static const char gap_cons = 3;

    //indicate the longest distance that the us can receive
    static const float largest_dist = 100;

    //define what is large distance that should change area;
    static const float long_dist = 30;
    //determin after how many consecutive long dist detection
    //will we say that is in long dist area
    static const char long_dist_cons = 3;
    //same for short distance
    static const char short_dist_cons = 3;

    //tolerable us_RF, RB difference
    //(should be < tolerable_range*2 , in general I think)
    static const float tolerable_delta_f_b = 0.2;

    //define whether us_RF and us_RB are both in short or long
    //static const float in_same_area_diff = 5;

    //determine when to turn left
    static const float turn_left_dist = 25;

    //current long dist value that should keep
    float cur_long_dist, cur_long_dist_max, cur_long_dist_min;

    //indicate how long want to keep from wall currently, could be short
or long
    float keep_dist, keep_dist_max, keep_dist_min;
    //indicate in what times of the detection of the issue that when
    //ultrasonic sensor is too close the value will become the maximum
    //when short distant -> long distant, this will start counting.
    //This machenism can also avoid small hole/glitch
    char long_dist_counter = 0;
    char short_dist_counter = 0;

    //the 0 will be the newest value, and 1 is one time prior to 0, and
so on
    //float us_RF[val_count] = {0};
    //float us_RB[val_count] = {0};
    float us_RF, us_RB, us_FR, us_BR;
    //could be us_RF or us_RB, depend on which direction is going
    float us_R_dist, us_R_reverse;

    //last value before seeing a gap in long area
    float last_long_dist_bef_gap;
    float delta_cur_last_long_dist;
    char gap_counter = 0;

    //
    float last_short_dist_bef_in_long_area[3] = {short_dist};

    //could be FR or BR
    float us_FB_dist;

    //indicate whehter really going into long distnat area
    char is_in_long_dist_area = FALSE;

    //remember previous regular motor direction
    r_direction_t prev_r_dir = FORWARD;
    r_direction_t cur_r_dir;

    //remember previous servo motor direction
    //s_direction_t prev_s_dir = RIGHT;
    //s_direction_t cur_s_dir;

    //depth of lot in long area(minimum)
    float lot_depth;

    //decide whether the space is long enough to park
    long parallel_leng_counter = 0;
    long garage_leng_counter = 0;

    //determine whether has turn left and need turn right now
    char has_turned_left = FALSE;

    //parking mode
    char parallel_mode, garage_mode;

    int i;

    //convert use choice
    switch(*(int *)arg){
```

```
    case 0: //along the wall
        parallel_mode = FALSE;
        garage_mode = FALSE;
        break;
    case 1:
        parallel_mode = TRUE;
        garage_mode = FALSE;
        break;

    case 2:
        parallel_mode = FALSE;
        garage_mode = TRUE;
        break;

    case 3: //auto choose
        parallel_mode = TRUE;
        garage_mode = TRUE;
        break;

    default:
        puts("Invalid choice, auto parking aborted");
}

write_r_motor_speed(1);

while(TRUE){

    //sleep here so that when we use continue will sleep
    usleep(ULTRASONICS_READ_INTERVAL);

    //if not moving, continue to next loop
    if(read_r_motor_speed() == 0){
        continue;
    }

    //======== for protection =====================

    us_FR = cycles_to_cm(read_ultrasonic_distance(FR));
    us_BR = cycles_to_cm(read_ultrasonic_distance(BR));
    if(us_FR < 6 || us_BR < 6){
        write_r_motor_speed(0);
        continue;
    }
    //=============================================

    //shift the previous values
    //for(int i=val_count-1; i > 0; i--){
    //  us_RF[i] = us_RF[i-1];
    //  us_RB[i] = us_RB[i-1];
    //}

    //fetch new value
    //us_RF[0] = cycles_to_cm(read_ultrasonic_distance(RF));
    //us_RB[0] = cycles_to_cm(read_ultrasonic_distance(RB));
```

```
        us_RF = cycles_to_cm(read_ultrasonic_distance(RF));
        us_RB = cycles_to_cm(read_ultrasonic_distance(RB));



        //clear some memorized states if direction changed
        cur_r_dir = read_r_motor_dir();
        if(cur_r_dir != prev_r_dir){
            long_dist_counter = 0;
            short_dist_counter = 0;
        }
        prev_r_dir = cur_r_dir; //cannot put at the end of while loop, or
will be skipped if use continue



        // //Determine whether need to turn left
        // us_FB_dist = (cur_r_dir == FORWARD)?
        //  cycles_to_cm(read_ultrasonic_distance(FR)):cycles_to_cm(read
_ultrasonic_distance(BR));

        // if(us_FB_dist < turn_left_dist){ //start to turn left to avoid
bump the wall

        //  write_s_motor_dir(LEFT);
        //  write_s_motor_angle(MAX_ANGLE_LV);
        //  has_turned_left = TRUE;
        //  continue;
        // }

        // if(has_turned_left){

        //  is_in_long_dist_area = FALSE;

        //  has_turned_left = FALSE;
        //  continue;
        // }

        //Determine whether it's really a long/short dist or just some
accident occurred
        us_R_dist = (cur_r_dir == FORWARD)? us_RF : us_RB;
        us_R_reverse = (cur_r_dir == FORWARD)? us_RB : us_RF;

        printf("RF:%.2f, RB:%.2f\n", us_RF, us_RB);

        if(us_R_dist > long_dist){
            short_dist_counter = 0;
            if(!is_in_long_dist_area){

                //turn straight first so that it won't go bias
                write_s_motor_angle(0);
                long_dist_counter ++ ;
                if(long_dist_counter >= long_dist_cons){
                    is_in_long_dist_area = TRUE;
                    cur_long_dist = us_R_dist;
                    cur_long_dist_max = cur_long_dist + tolerable_range;
                    cur_long_dist_min = cur_long_dist - tolerable_range;
                    long_dist_counter = 0;

                    lot_depth = us_R_dist;
                    last_long_dist_bef_gap = us_R_dist;
                    parallel_leng_counter = 0;
                    garage_leng_counter = 0;
                    gap_counter = 0;
                    //prev_was_in_range = FALSE;
                    //steering_feedback_counter = 0;
                }else{ // in uncertain state, we don't modify track, so
go another run
                    continue;
                }
            }else{ //is in long distant area

                delta_cur_last_long_dist       =       us_R_dist       -
last_long_dist_bef_gap;
                if(abs(delta_cur_last_long_dist) > gap_leng){
                    write_s_motor_angle(0);
                    gap_counter++;
                    if(gap_counter >= gap_cons){
                        lot_depth = (us_R_dist > lot_depth)? us_R_dist :
lot_depth;
                        last_long_dist_bef_gap = us_R_dist;

                        cur_long_dist = us_R_dist;
                        cur_long_dist_max     =     cur_long_dist     +
tolerable_range;
                        cur_long_dist_min     =     cur_long_dist     -
tolerable_range;



                    }else{
                        continue;
                    }
                }

                gap_counter = 0;

                lot_depth  =  (us_R_dist  >  lot_depth)?  us_R_dist  :
lot_depth;

                //use the reverse one to determin
                if(us_R_reverse > last_short_dist_bef_in_long_area[2] +
CAR_WIDTH + CAR_LENGTH_WIDTH_SAFE_MARGIN)
                    parallel_leng_counter++;
                else
                    parallel_leng_counter = 0;

                if(us_R_reverse > last_short_dist_bef_in_long_area[2] +
CAR_LENGTH + CAR_LENGTH_WIDTH_SAFE_MARGIN)
                    garage_leng_counter++;
                else
                    garage_leng_counter = 0;
                if(parallel_leng_counter  >=  PARALLEL_ENOUGH_COUNT  &&
parallel_mode){

                    parallel_parking(last_short_dist_bef_in_long_area[2],
lot_depth - last_short_dist_bef_in_long_area[2]);
                    puts("Done parking");
                    return NULL;
                }




            }
        }else{

            long_dist_counter = 0;
            if(is_in_long_dist_area){

                write_s_motor_angle(0);
                short_dist_counter ++;
                if(short_dist_counter >= short_dist_cons){
                    is_in_long_dist_area = FALSE;
                    short_dist_counter = 0;
                    //prev_was_in_range = FALSE;
                    //steering_feedback_counter = 0;
                    last_short_dist_bef_in_long_area[0]                =
last_short_dist_bef_in_long_area[2];
                    last_short_dist_bef_in_long_area[1]                =
last_short_dist_bef_in_long_area[2];
                }else{
                    continue;
                }
            }else{


                if(abs(us_R_reverse  -  us_R_dist)  >  8){  //reverse  side
still in long area

                    //use the reverse one to determin
                    if(us_R_reverse > last_short_dist_bef_in_long_area[2]
+ CAR_WIDTH + CAR_LENGTH_WIDTH_SAFE_MARGIN)
                        parallel_leng_counter++;
                    else
                        parallel_leng_counter = 0;

                    if(us_R_reverse > last_short_dist_bef_in_long_area[2]
+ CAR_LENGTH + CAR_LENGTH_WIDTH_SAFE_MARGIN)
                        garage_leng_counter++;
                    else
                        garage_leng_counter = 0;

                    if(parallel_leng_counter >= PARALLEL_ENOUGH_COUNT &&
parallel_mode){
```

```
                        parallel_parking(us_R_dist,      lot_depth      -
us_R_dist);
                        puts("Done parking");
                        return NULL;
                    }

                }else{
                    //shitft
                    for(i=2; i > 0; i--){
                        last_short_dist_bef_in_long_area[i]             =
last_short_dist_bef_in_long_area[i-1];
                    }
                    last_short_dist_bef_in_long_area[0] = us_R_dist;
                }
            }
        }

        if(is_in_long_dist_area){

            keep_dist = cur_long_dist;
            keep_dist_max = cur_long_dist_max;
            keep_dist_min = cur_long_dist_min;

        }else{

            keep_dist = short_dist;
            keep_dist_max = short_dist_max;
            keep_dist_min = short_dist_min;
        }

        //printf("Current   is%s   in   long_dist_area.   keep_dist:%.2f,
max: %.2f, min: %.2f\n",
        //      (is_in_long_dist_area)?     "":"     not",     keep_dist,
keep_dist_max,keep_dist_min);

        //cur_s_dir = read_s_motor_dir();


        keep_in_dist(keep_dist,       keep_dist_max,       keep_dist_min,
tolerable_delta_f_b, us_R_dist, us_R_reverse);

        //prev_s_dir = cur_s_dir;

    }


    return NULL;
}



void keep_in_dist(const float keep_dist,  const float keep_dist_max,  const
float keep_dist_min,
                  const float tolerable_delta_f_b,
                  const float us_R_dist, const float us_R_reverse){

    //float for tmperary calculation
    float delta_cur_keep;
    float delta_f_b_us;

    //start to modify track
    //check whether in tolerable range (most common case, so detect first)
    if(us_R_dist < keep_dist_max && us_R_dist > keep_dist_min){



        delta_f_b_us = us_R_dist - us_R_reverse;
        if(delta_f_b_us > tolerable_delta_f_b){
            write_s_motor_dir(RIGHT);
            write_s_motor_angle(1);
        }else if(delta_f_b_us < (-tolerable_delta_f_b)){
            write_s_motor_dir(LEFT);
            write_s_motor_angle(1);
        }else{
            write_s_motor_angle(0);
        }


    }else{


        delta_cur_keep = us_R_dist - keep_dist;

        if(delta_cur_keep > 0){ //bias outside

            write_s_motor_dir(RIGHT);

            if(delta_cur_keep < 2){
                write_s_motor_angle(1);
            }else if(delta_cur_keep < 5){
                write_s_motor_angle(2);
            }else if(delta_cur_keep < 10){
                write_s_motor_angle(3);
            }else if(delta_cur_keep < 20){
                write_s_motor_angle(4);
            }else if(delta_cur_keep < 30){
                write_s_motor_angle(5);
            }else if(delta_cur_keep < 45){
                write_s_motor_angle(6);
            }else{
                write_s_motor_angle(7);
            }
        }else{ //bias toward wall
            write_s_motor_dir(LEFT);
            if(delta_cur_keep > -1){
                write_s_motor_angle(1);
            }else if(delta_cur_keep > -3){
                write_s_motor_angle(4);
            }else{
                write_s_motor_angle(7);
            }

        }

    }

}
```

Regular_motor.c

```
/*
 * Parking Robot Team
 * Columbia University
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "regular_motor.h"
#include <linux/types.h>

#define DRIVER_NAME "regular_motor"

/*
 * Information about our device
 */
struct regular_motor_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory
*/
    r_motor_t r_motor_val; /* Current position of the ball, is useless
unless we want to read it */
} dev;

/*
 * Write position
 * Assumes position is in range and the device information has been set
up
 */
static void write_dir(r_direction_t r_dir)
{
    u32 writedata;
    if(r_dir == FORWARD)
        writedata = 0;
    else
        writedata = 1;

    iowrite32(writedata, dev.virtbase);
    dev.r_motor_val.r_dir = r_dir;
}

static void write_speed(__u32 speed)
{
    iowrite32(speed, dev.virtbase + 4);
    dev.r_motor_val.speed = speed;
}

/*
 * Handle ioctl() calls from userspace:
 * Write position to peripheral.
 * Note extensive error checking of arguments
 */
static long regular_motor_ioctl(struct file *f, unsigned int cmd, unsigned
long arg)
{
    r_direction_t r_dir;
    __u32 speed;


    switch (cmd) {
    case R_MOTOR_SET_DIR:
        if    (copy_from_user(&r_dir,     (r_direction_t     *)     arg,
sizeof(r_direction_t)))
            return -EACCES;
        //check whether it's in valid range
        if (r_dir != FORWARD && r_dir != BACKWARD) //shouldn't happen
actually, but in case
            return -EINVAL;
        write_dir(r_dir);
        break;

    case R_MOTOR_SET_SPEED:
        if (copy_from_user(&speed, (__u32 *) arg, sizeof(__u32)))
            return -EACCES;
        //check whether it's in valid range
        if (speed < MIN_SPEED_LV || speed > MAX_SPEED_LV) //shouldn't
happen actually, but in case
            return -EINVAL;
        write_speed(speed);
        break;
    case R_MOTOR_GET_DIR:
        if (copy_to_user((r_direction_t *) arg, &dev.r_motor_val.r_dir,
sizeof(r_direction_t)))
            return -EACCES;
        break;

    case R_MOTOR_GET_SPEED:
        if   (copy_to_user((__u32   *)   arg,   &dev.r_motor_val.speed,
sizeof(__u32)))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations regular_motor_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = regular_motor_ioctl,
};

/* Information about our device for the "misc" framework -- like a char
dev */
static struct miscdevice regular_motor_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &regular_motor_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init regular_motor_probe(struct platform_device *pdev)
{
    int ret;

    /* Register ourselves as a misc device: creates /dev/regular_motor */
    ret = misc_register(&regular_motor_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                   DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }


    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&regular_motor_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int regular_motor_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&regular_motor_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id regular_motor_of_match[] = {
    { .compatible = "altr,regular_motor" },
    {},
};
MODULE_DEVICE_TABLE(of, regular_motor_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver regular_motor_driver = {
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(regular_motor_of_match),
    },
    .remove = __exit_p(regular_motor_remove),
};

/* Calball when the module is loaded: set things up */
static int __init regular_motor_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return                  platform_driver_probe(&regular_motor_driver,
regular_motor_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit regular_motor_exit(void)
{
    platform_driver_unregister(&regular_motor_driver);
    pr_info(DRIVER_NAME ": exit\n");
}
```

```
module_init(regular_motor_init);
module_exit(regular_motor_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment ball Emulator");
```

Socfpga.dts

```
/*
 *  Copyright (C) 2012 Altera Corporation <www.altera.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

/dts-v1/;
/include/ "socfpga.dtsi"

/ {
	model = "Altera SOCFPGA Cyclone V";
	compatible = "altr,socfpga-cyclone5", "altr,socfpga";

	chosen {
		bootargs = "console=ttyS0,57600";
	};

	memory {
		name = "memory";
		device_type = "memory";
		reg = <0x0 0x40000000>; /* 1 GB */
	};

	aliases {
		/* this allow the ethaddr uboot environmnet variable contents
		 * to be added to the gmac1 device tree blob.
		 */
		ethernet0 = &gmac1;
	};
```

```
soc {
        clkmgr@ffd04000 {
                clocks {
                        osc1 {
                                clock-frequency = <25000000>;
                        };
                };
        };

        dcan0: d_can@ffc00000 {
                status = "disabled";
        };

        dcan1: d_can@ffc10000 {
                status = "disabled";
        };

        dwmmc0@ff704000 {
                num-slots = <1>;
                supports-highspeed;
                broken-cd;
                altr,dw-mshc-ciu-div = <4>;
                altr,dw-mshc-sdr-timing = <0 3>;

                slot@0 {
                        reg = <0>;
                        bus-width = <4>;
                };
        };
        lightweight_bridge: bridge@0xff200000 {
                #address-cells = <1>;
                #size-cells = <1>;
                ranges = < 0x0 0xff200000 0x200000 >;

                compatible = "simple-bus";


                ultrasonic_sensor: ultrasonic_sensor@0 {
                          compatible = "altr,ultrasonic_sensor";
                          reg = <0x0 0x10>;
                };

                regular_motor: regular_motor@0 {
                          compatible = "altr,regular_motor";
                          reg = <0x10 0x8>;
                };

                steering_motor: steering_motor@0 {
                          compatible = "altr,steering_motor";
                          reg = <0x18 0x8>;
                };
        };
```

```
ethernet@ff700000 {
        status = "disabled";
};

ethernet@ff702000 {
        phy-mode = "rgmii";
        phy-addr = <0xffffffff>; /* probe for phy addr */
};

i2c1: i2c@ffc05000 {
        status = "disabled";
};

i2c2: i2c@ffc06000 {
        status = "disabled";
};

i2c3: i2c@ffc07000 {
        status = "disabled";
};

qspi: spi@ff705000 {
                compatible = "cadence,qspi";
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <0xff705000 0x1000>,
                        <0xffa00000 0x1000>;
                interrupts = <0 151 4>;
                master-ref-clk = <400000000>;
                ext-decoder = <0>;    /* external decoder */
                num-chipselect = <4>;
                fifo-depth = <128>;
                bus-num = <2>;

                flash0: n25q00@0 {
                        #address-cells = <1>;
                        #size-cells = <1>;
                        compatible = "n25q00";
                        reg = <0>;      /* chip select */
                        spi-max-frequency = <100000000>;
                        page-size = <256>;
                        block-size = <16>; /* 2^16, 64KB */
                        quad = <1>;          /* 1-support quad */
                        tshsl-ns = <200>;
                        tsd2d-ns = <255>;
                        tchsh-ns = <20>;
                        tslch-ns = <20>;

                        partition@0 {
                                /* 8MB for raw data. */
                                label = "Flash 0 Raw Data";
                                reg = <0x0 0x800000>;
```

```
						};
						partition@800000 {
							/* 8MB for jffs2 data. */
							label = "Flash 0 jffs2 Filesystem";
							reg = <0x800000 0x800000>;
						};
					};

				};

			sysmgr@ffd08000 {
				cpu1-start-addr = <0xffd080c4>;
			};

			timer0@ffc08000 {
				clock-frequency = <100000000>;
			};

			timer1@ffc09000 {
				clock-frequency = <100000000>;
			};

			timer2@ffd00000 {
				clock-frequency = <25000000>;
			};

			timer3@ffd01000 {
				clock-frequency = <25000000>;
			};

			serial0@ffc02000 {
				clock-frequency = <100000000>;
			};

			serial1@ffc03000 {
				clock-frequency = <100000000>;
			};

			usb0: usb@ffb00000 {
				status = "disabled";
			};

			usb1: usb@ffb40000 {
				ulpi-ddr = <0>;
			};

			i2c0: i2c@ffc04000 {
				speed-mode = <0>;
			};

			leds {
				compatible = "gpio-leds";
				hps0 {
```

```
                        label = "hps_led0";
                        gpios = <&gpio1 15 1>;
                };

                hps1 {
                        label = "hps_led1";
                        gpios = <&gpio1 14 1>;
                };

                hps2 {
                        label = "hps_led2";
                        gpios = <&gpio1 13 1>;
                };

                hps3 {
                        label = "hps_led3";
                        gpios = <&gpio1 12 1>;
                };
            };
        };
};

&i2c0 {
        lcd: lcd@28 {
                compatible = "newhaven,nhd-0216k3z-nsw-bbw";
                reg = <0x28>;
                height = <2>;
                width = <16>;
                brightness = <8>;
        };

        eeprom@51 {
                compatible = "atmel,24c32";
                reg = <0x51>;
                pagesize = <32>;
        };

        rtc@68 {
                compatible = "dallas,ds1339";
                reg = <0x68>;
        };
};
```

Steering motor.c

```
/*
 * Parking Robot Team
 * Columbia University
 */

#include <linux/module.h>
#include <linux/init.h>
```

```
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "steering_motor.h"
#include <linux/types.h>

#define DRIVER_NAME "steering_motor"

/*
 * Information about our device
 */
struct steering_motor_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory
*/
    s_motor_t s_motor_val; /* Current position of the ball, is useless
unless we want to read it */
} dev;

/*
 * Write position
 * Assumes position is in range and the device information has been set
up
 */
static void write_dir(s_direction_t s_dir)
{
    u32 writedata;
    if(s_dir == RIGHT)
        writedata = 0;
    else
        writedata = 1;

    iowrite32(writedata, dev.virtbase);
    dev.s_motor_val.s_dir = s_dir;
}

static void write_angle(__u32 angle)
{
    iowrite32(angle, dev.virtbase + 4);
    dev.s_motor_val.angle = angle;
}

/*
 * Handle ioctl() calls from userspace:
 * Write position to peripheral.
 * Note extensive error checking of arguments
 */
```

```
static long steering_motor_ioctl(struct file *f, unsigned int cmd, unsigned
long arg)
{
    s_direction_t s_dir;
    __u32 angle;


    switch (cmd) {
    case S_MOTOR_SET_DIR:
        if    (copy_from_user(&s_dir,     (s_direction_t     *)     arg,
sizeof(s_direction_t)))
            return -EACCES;
        //check whether it's in valid range
        if (s_dir != RIGHT && s_dir != LEFT) //shouldn't happen actually,
but in case
            return -EINVAL;
        write_dir(s_dir);
        break;

    case S_MOTOR_SET_ANGLE:
        if (copy_from_user(&angle, (__u32 *) arg, sizeof(__u32)))
            return -EACCES;
        //check whether it's in valid range
        if (angle  <  MIN_ANGLE_LV  ||  angle  >  MAX_ANGLE_LV)  //shouldn't
happen actually, but in case
            return -EINVAL;
        write_angle(angle);
        break;

    case S_MOTOR_GET_DIR:
        if (copy_to_user((s_direction_t  *)  arg,  &dev.s_motor_val.s_dir,
sizeof(s_direction_t)))
            return -EACCES;
        break;

    case S_MOTOR_GET_ANGLE:
        if   (copy_to_user((__u32    *)    arg,    &dev.s_motor_val.angle,
sizeof(__u32)))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations steering_motor_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = steering_motor_ioctl,
};
```

```
/* Information about our device for the "misc" framework -- like a char
dev */
static struct miscdevice steering_motor_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &steering_motor_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init steering_motor_probe(struct platform_device *pdev)
{
    int ret;

    /* Register ourselves as a misc device: creates /dev/steering_motor
*/
    ret = misc_register(&steering_motor_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                   DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }


    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&steering_motor_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int steering_motor_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&steering_motor_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id steering_motor_of_match[] = {
    { .compatible = "altr,steering_motor" },
    {},
};
MODULE_DEVICE_TABLE(of, steering_motor_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver steering_motor_driver = {
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(steering_motor_of_match),
    },
    .remove = __exit_p(steering_motor_remove),
};

/* Calball when the module is loaded: set things up */
static int __init steering_motor_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return                platform_driver_probe(&steering_motor_driver,
steering_motor_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit steering_motor_exit(void)
{
    platform_driver_unregister(&steering_motor_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(steering_motor_init);
module_exit(steering_motor_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment ball Emulator");
```

```
Ultrasonic_sensor.c

/*
 * Parking Robot Team
 * Columbia University
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "ultrasonic_sensor.h"
#include <linux/types.h>

#define DRIVER_NAME "ultrasonic_sensor"

/*
 * Information about our device
 */
struct ultrasonic_sensor_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory
*/
    //s_motor_t s_motor_val;
} dev;

/*
 * Write position
 * Assumes position is in range and the device information has been set
up
 */
static __u32 read_distance(us_position_t pos)
{
    return ioread32(dev.virtbase + pos*4);
}

/*
 * Handle ioctl() calls from userspace:
 * Write position to peripheral.
 * Note extensive error checking of arguments
 */
static long ultrasonic_sensor_ioctl(struct file  *f,  unsigned  int cmd,
unsigned long arg)
{
    ultrasonic_t us;
    switch (cmd) {
    case ULTRASONIC_GET_DIST_WITH_POS:
        if     (copy_from_user(&us,      (ultrasonic_t      *)      arg,
sizeof(ultrasonic_t)))
            return -EACCES;
        //check whether in range. Idealy it's already checked by compiler,
but
        //in case compiler has different behavior for enum default values
or
        //someone try to assign number for enum manually in .h file
        if(us.pos < 0 || us.pos >= NUM_OF_US)
            return -EINVAL;

        us.distance = read_distance(us.pos);
        if (copy_to_user((ultrasonic_t *) arg, &us, sizeof(ultrasonic_t)))
            return -EACCES;

        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations ultrasonic_sensor_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = ultrasonic_sensor_ioctl,
};

/* Information about our device for the "misc" framework -- like a char
dev */
static struct miscdevice ultrasonic_sensor_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &ultrasonic_sensor_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init ultrasonic_sensor_probe(struct platform_device *pdev)
{
    int ret;

    /* Register ourselves as a misc device: creates /dev/ultrasonic_sensor
*/
    ret = misc_register(&ultrasonic_sensor_misc_device);
    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                   DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }


    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&ultrasonic_sensor_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int ultrasonic_sensor_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&ultrasonic_sensor_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id ultrasonic_sensor_of_match[] = {
    { .compatible = "altr,ultrasonic_sensor" },
    {},
};
MODULE_DEVICE_TABLE(of, ultrasonic_sensor_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver ultrasonic_sensor_driver = {
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(ultrasonic_sensor_of_match),
    },
    .remove = __exit_p(ultrasonic_sensor_remove),
};

/* Calball when the module is loaded: set things up */
static int __init ultrasonic_sensor_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return              platform_driver_probe(&ultrasonic_sensor_driver,
ultrasonic_sensor_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit ultrasonic_sensor_exit(void)
{
    platform_driver_unregister(&ultrasonic_sensor_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(ultrasonic_sensor_init);
module_exit(ultrasonic_sensor_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment ball Emulator");
```

Parking_Robot FPGA code:

SoCKit_top.v

```
	wire stop; //emergency stop signal that will be sent from correct_distance to r_motor and s_motor
	assign stop = SW[0];
	//Qsys generated entity
	parking_robot u0 (
		.clk_clk                               (OSC_50_B4A),                            //
clk.clk
		.reset_reset_n                          (hps_fpga_reset_n),                      //
reset.reset_n
		.memory_mem_a                                                             (memory_mem_a),
//                          memory.mem_a
		.memory_mem_ba                                                           (memory_mem_ba),
//                         .mem_ba
		.memory_mem_ck                                                           (memory_mem_ck),
//                         .mem_ck
		.memory_mem_ck_n                                                       (memory_mem_ck_n),
//                         .mem_ck_n
		.memory_mem_cke                                                         (memory_mem_cke),
//                         .mem_cke
		.memory_mem_cs_n                                                       (memory_mem_cs_n),
//                         .mem_cs_n
		.memory_mem_ras_n                                                     (memory_mem_ras_n),
//                         .mem_ras_n
		.memory_mem_cas_n                                                     (memory_mem_cas_n),
//                         .mem_cas_n
		.memory_mem_we_n                                                       (memory_mem_we_n),
//                         .mem_we_n
		.memory_mem_reset_n                                                 (memory_mem_reset_n),
//                         .mem_reset_n
		.memory_mem_dq                                                           (memory_mem_dq),
//                         .mem_dq
		.memory_mem_dqs                                                         (memory_mem_dqs),
//                         .mem_dqs
		.memory_mem_dqs_n                                                     (memory_mem_dqs_n),
//                         .mem_dqs_n
		.memory_mem_odt                                                         (memory_mem_odt),
//                         .mem_odt
		.memory_mem_dm                                                           (memory_mem_dm),
//                         .mem_dm
		.memory_oct_rzqin                                                     (memory_oct_rzqin),
//                         .oct_rzqin
		.hps_io_hps_io_emac1_inst_TX_CLK        (hps_io_hps_io_emac1_inst_TX_CLK),       //
                           .hps_0_hps_io.hps_io_emac1_inst_TX_CLK
		.hps_io_hps_io_emac1_inst_TXD0          (hps_io_hps_io_emac1_inst_TXD0),
//                         .hps_io_emac1_inst_TXD0
		.hps_io_hps_io_emac1_inst_TXD1          (hps_io_hps_io_emac1_inst_TXD1),
//                         .hps_io_emac1_inst_TXD1
		.hps_io_hps_io_emac1_inst_TXD2          (hps_io_hps_io_emac1_inst_TXD2),
//                         .hps_io_emac1_inst_TXD2
```

```
		.hps_io_hps_io_emac1_inst_TXD3		(hps_io_hps_io_emac1_inst_TXD3),
//		.hps_io_emac1_inst_TXD3
		.hps_io_hps_io_emac1_inst_RXD0		(hps_io_hps_io_emac1_inst_RXD0),
//		.hps_io_emac1_inst_RXD0
		.hps_io_hps_io_emac1_inst_MDIO		(hps_io_hps_io_emac1_inst_MDIO),
//		.hps_io_emac1_inst_MDIO
		.hps_io_hps_io_emac1_inst_MDC		(hps_io_hps_io_emac1_inst_MDC),
//		.hps_io_emac1_inst_MDC
		.hps_io_hps_io_emac1_inst_RX_CTL		(hps_io_hps_io_emac1_inst_RX_CTL),
//		.hps_io_emac1_inst_RX_CTL
		.hps_io_hps_io_emac1_inst_TX_CTL		(hps_io_hps_io_emac1_inst_TX_CTL),
//		.hps_io_emac1_inst_TX_CTL
		.hps_io_hps_io_emac1_inst_RX_CLK			(hps_io_hps_io_emac1_inst_RX_CLK),
//		.hps_io_emac1_inst_RX_CLK
		.hps_io_hps_io_emac1_inst_RXD1		(hps_io_hps_io_emac1_inst_RXD1),
//		.hps_io_emac1_inst_RXD1
		.hps_io_hps_io_emac1_inst_RXD2		(hps_io_hps_io_emac1_inst_RXD2),
//		.hps_io_emac1_inst_RXD2
		.hps_io_hps_io_emac1_inst_RXD3		(hps_io_hps_io_emac1_inst_RXD3),
//		.hps_io_emac1_inst_RXD3
		.hps_io_hps_io_qspi_inst_IO0		(hps_io_hps_io_qspi_inst_IO0),
//		.hps_io_qspi_inst_IO0
		.hps_io_hps_io_qspi_inst_IO1		(hps_io_hps_io_qspi_inst_IO1),
//		.hps_io_qspi_inst_IO1
		.hps_io_hps_io_qspi_inst_IO2		(hps_io_hps_io_qspi_inst_IO2),
//		.hps_io_qspi_inst_IO2
		.hps_io_hps_io_qspi_inst_IO3		(hps_io_hps_io_qspi_inst_IO3),
//		.hps_io_qspi_inst_IO3
		.hps_io_hps_io_qspi_inst_SS0		(hps_io_hps_io_qspi_inst_SS0),
//		.hps_io_qspi_inst_SS0
		.hps_io_hps_io_qspi_inst_CLK		(hps_io_hps_io_qspi_inst_CLK),
//		.hps_io_qspi_inst_CLK
		.hps_io_hps_io_sdio_inst_CMD		(hps_io_hps_io_sdio_inst_CMD),
//		.hps_io_sdio_inst_CMD
		.hps_io_hps_io_sdio_inst_D0		(hps_io_hps_io_sdio_inst_D0),
//		.hps_io_sdio_inst_D0
		.hps_io_hps_io_sdio_inst_D1		(hps_io_hps_io_sdio_inst_D1),
//		.hps_io_sdio_inst_D1
		.hps_io_hps_io_sdio_inst_CLK		(hps_io_hps_io_sdio_inst_CLK),
//		.hps_io_sdio_inst_CLK
		.hps_io_hps_io_sdio_inst_D2	(hps_io_hps_io_sdio_inst_D2),
//		.hps_io_sdio_inst_D2
		.hps_io_hps_io_sdio_inst_D3		(hps_io_hps_io_sdio_inst_D3),
//		.hps_io_sdio_inst_D3
		.hps_io_hps_io_usb1_inst_D0		(hps_io_hps_io_usb1_inst_D0),
//		.hps_io_usb1_inst_D0
		.hps_io_hps_io_usb1_inst_D1		(hps_io_hps_io_usb1_inst_D1),
//		.hps_io_usb1_inst_D1
		.hps_io_hps_io_usb1_inst_D2		(hps_io_hps_io_usb1_inst_D2),
//		.hps_io_usb1_inst_D2
		.hps_io_hps_io_usb1_inst_D3		(hps_io_hps_io_usb1_inst_D3),
//		.hps_io_usb1_inst_D3
```

```
		.hps_io_hps_io_usb1_inst_D4		(hps_io_hps_io_usb1_inst_D4),
//			.hps_io_usb1_inst_D4
		.hps_io_hps_io_usb1_inst_D5		(hps_io_hps_io_usb1_inst_D5),
//			.hps_io_usb1_inst_D5
		.hps_io_hps_io_usb1_inst_D6		(hps_io_hps_io_usb1_inst_D6),
//			.hps_io_usb1_inst_D6
		.hps_io_hps_io_usb1_inst_D7		(hps_io_hps_io_usb1_inst_D7),
//			.hps_io_usb1_inst_D7
		.hps_io_hps_io_usb1_inst_CLK		(hps_io_hps_io_usb1_inst_CLK),
//			.hps_io_usb1_inst_CLK
		.hps_io_hps_io_usb1_inst_STP		(hps_io_hps_io_usb1_inst_STP),
//			.hps_io_usb1_inst_STP
		.hps_io_hps_io_usb1_inst_DIR		(hps_io_hps_io_usb1_inst_DIR),
//			.hps_io_usb1_inst_DIR
		.hps_io_hps_io_usb1_inst_NXT		(hps_io_hps_io_usb1_inst_NXT),
//			.hps_io_usb1_inst_NXT
		.hps_io_hps_io_spim0_inst_CLK		(hps_io_hps_io_spim0_inst_CLK),
//			.hps_io_spim0_inst_CLK
		.hps_io_hps_io_spim0_inst_MOSI		(hps_io_hps_io_spim0_inst_MOSI),
//			.hps_io_spim0_inst_MOSI
		.hps_io_hps_io_spim0_inst_MISO		(hps_io_hps_io_spim0_inst_MISO),
//			.hps_io_spim0_inst_MISO
		.hps_io_hps_io_spim0_inst_SS0		(hps_io_hps_io_spim0_inst_SS0),
//			.hps_io_spim0_inst_SS0
		.hps_io_hps_io_spim1_inst_CLK		(hps_io_hps_io_spim1_inst_CLK),
//			.hps_io_spim1_inst_CLK
		.hps_io_hps_io_spim1_inst_MOSI		(hps_io_hps_io_spim1_inst_MOSI),
//			.hps_io_spim1_inst_MOSI
		.hps_io_hps_io_spim1_inst_MISO		(hps_io_hps_io_spim1_inst_MISO),
//			.hps_io_spim1_inst_MISO
		.hps_io_hps_io_spim1_inst_SS0		(hps_io_hps_io_spim1_inst_SS0),
//			.hps_io_spim1_inst_SS0
		.hps_io_hps_io_uart0_inst_RX		(hps_io_hps_io_uart0_inst_RX),
//			.hps_io_uart0_inst_RX
		.hps_io_hps_io_uart0_inst_TX		(hps_io_hps_io_uart0_inst_TX),
//			.hps_io_uart0_inst_TX
		.hps_io_hps_io_i2c1_inst_SDA		(hps_io_hps_io_i2c1_inst_SDA),
//			.hps_io_i2c1_inst_SDA
		.hps_io_hps_io_i2c1_inst_SCL		(hps_io_hps_io_i2c1_inst_SCL),
//			.hps_io_i2c1_inst_SCL
		//custom conduits
		.r_motor_en						(HSMC_RX_n[16]),
//			.en
		.r_motor_c1				(HSMC_RX_p[16]),			//
r_motor.c1
		.r_motor_c2						(HSMC_RX_n[15]),
//			.c2
		.stop_rmotor_export			(stop),			//	stop_rmotor.export
		.s_motor_c1				(HSMC_TX_n[9]),			//
s_motor.c1
		.s_motor_c2						(HSMC_TX_p[9]),
//			.c2
		.stop_smotor_export			(stop),			//	stop_smotor.export
```

```
            .stop_cordist_export                 (),                    // stop_cordist.export
            //echo and trigger: 3,2,1,0 -> BR, RB, RF, FR
            .ultrasonic_echo                                     ({HSMC_TX_n[11],   HSMC_TX_n[2],
HSMC_TX_n[0], HSMC_RX_n[2]}),                         //      ultrasonic.echo
           .ultrasonic_trigger                    ({HSMC_TX_p[11], HSMC_TX_p[2], HSMC_TX_p[0],
HSMC_RX_p[2]})                      //                    .trigger
  );


correct_distance_qsys.sv
//F: Front, B: Back, R: Right
//e.g. FR: on the right corner of the front
//e.g. RF: in the front of right side
module correct_distance_qsys(
                input logic                         clk,
                input logic                  reset,
                output logic [31:0]  readdata, // Only accept 8, 16, 32, 64,... bits;
                input logic    [1:0]          address, //00:FR, 01:RF, 10:RB, 11:BR
                input logic                  read,
                input logic                  chipselect,

                //ultrasonic part
                input logic        [3:0]      echo, //echo[0]:FR, echo[1]:RF, etc.
                output logic [3:0]    trigger,

                output logic                 stop); //for emergency stop that directly send to motors



     //                              ======================                          Ultrasonic
=======================================

     logic [19:0] us_distance_000, us_distance_001, us_distance_010, us_distance_011;

     logic [3:0] pause;

     logic [3:0] finished;


     ultrasonic us_000(  .clk(clk),
                                   .reset(reset),
                                   .pause(pause[0]),
                                   .finished(finished[0]),
                                   .echo(echo[0]),
                                   .trigger(trigger[0]),
                                   .distance(us_distance_000));

     ultrasonic us_001(  .clk(clk),
                                   .reset(reset),
                                   .pause(pause[1]),
                                   .finished(finished[1]),
                                   .echo(echo[1]),
                                   .trigger(trigger[1]),
                                        .distance(us_distance_001));

ultrasonic us_010(   .clk(clk),
                                        .reset(reset),
                                        .pause(pause[2]),
                                        .finished(finished[2]),
                                        .echo(echo[2]),
                                        .trigger(trigger[2]),
                                        .distance(us_distance_010));

ultrasonic us_011(   .clk(clk),
                                        .reset(reset),
                                        .pause(pause[3]),
                                        .finished(finished[3]),
                                        .echo(echo[3]),
                                        .trigger(trigger[3]),
                                        .distance(us_distance_011));

always_comb begin

        readdata = 32'b0; //default

        if(chipselect && read) begin
                case(address)
                        2'b00: readdata = {12'h000, us_distance_000};

                        2'b01: readdata = {12'h000, us_distance_001};

                        2'b10: readdata = {12'h000, us_distance_010};

                        2'b11: readdata = {12'h000, us_distance_011};

                endcase
        end
end


//control ultrasonic sensors by pause signals
logic [1:0] pointer; //indicate which ultrasonic sensor is working now
always_ff @ (posedge clk) begin
        if(reset) begin
                pause <= 4'b1110;
                pointer <= 2'd0;
        end else begin
                if(~pause[pointer])begin
                        pause[pointer] <= 1'b1;
                end else begin
                        if(finished[pointer]) begin //done, change to next one
                                pause[pointer_next] <= 1'b0;
                                pointer <= pointer_next;
                        end
                end
        end
```

```
	end


//	always_ff @ (posedge clk) begin
//		if(reset) begin
//
//		end else begin
//			if(finished[pointer]) begin
//
//			end
//		end
//	end

	//pointer + 1
	logic [1:0] pointer_next;
	assign pointer_next = (pointer == 2'd3) ? 2'd0 : pointer + 1'b1;


endmodule
```

```
ultrasonic.sv
module ultrasonic(
					input logic clk,reset,
					input logic echo,

					//Used by correct_distance_module that every
					//ultrasonic can be controlled when to measure
					//distance.(In order to avoid conflict of multi
					//ultrasonic sensors measure at the same time)
					//When counter of a ultrasonic is zero, it will
					//wait until pause = 0 to start. Once start, it
					//doesn't care about pause signal.
					//When finished, the counter reset to zero and
					//raise finished singal until start to count again.
					input logic pause,
					output logic finished,

					output logic trigger,
					output logic [19:0] distance);

	parameter COUNT_INTERVAL = 26'd3_250_000;

	logic [19:0] distance_counter; //temp distance that will be counted depend on echo width
	logic [25:0] counter;

	always_ff @ (posedge clk)
	begin
		if(reset) begin
			counter <= 0;
			distance <= 0;
			distance_counter <=0;
```

```
        end else begin
            if(counter == 0) begin //when counter == 0, only start counting when pause == 0
                if(~pause) begin
                    counter <= counter + 1;
                end else begin
                    counter <= counter;
                end
            end else if (counter >= COUNT_INTERVAL) begin //reach counter maximum, set
distance and reset counter to zero
                distance <= distance_counter;
                counter <= 0;
                distance_counter <= 0;
            end else if (counter >= 26'd510) begin //after trigger deactive a while, start to monitor
echo and count width
                counter <= counter + 1;
                if (echo) begin
                    distance_counter <= distance_counter + 1;
                end
            end else begin
                counter <= counter + 1;
            end
        end
    end

    //trigger
    always_ff @ (posedge clk) begin
        if (counter > 0 && counter < 26'd502) //1 ~ 501 (0 might be in pause, shouldn't send trigger)
            trigger <= 1;
        else
            trigger <= 0;
    end

    //finished singnal
    assign finished = (counter == 0) ? 1'b1 : 1'b0;

endmodule



regular_motor_qsys.sv
module regular_motor_qsys(
            input logic                 clk,
            input logic             reset,
            //Note: we use 32 bits instead of 8bits because ARM is 32bits,
            //and if we use e.g. 16bits, avalon bus will actually send us
            //singal twice. 16 bits each, but one with byteenable=11 and
            //another with byteenable = 00. Therefore we will need to receive
            //and handle byteenable signal, or it may execute twice. More over,
            //it may not be as simple as we might though. When I tried to use
            //8bits writedata and 0x10 of base, I found that when I test it with
            //.tcl in System Console, all master_write_8 to 0x10, 0x11, 0x12 and
           //0x13 will result in controlling this regular motor, which should
            //only occupy 0x10. Furthermore, the steering motor which I set to
```

```
                        //also 8bits and base at 0x11 can't be controlled in this case, no matter
                        //which address I write to using master_write_8.
                        //That means, not only this peripheral, all peripherals
                      //near this one may also need to handle byteenable signal!
                        input logic [31:0]    writedata, // Only accept 8, 16, 32, 64,... bits;
                        input logic                     write,
                        input logic                     address,
                        input logic                     chipselect,

                        input logic                     stop, // emergency stop signal send from other peripheral
(Ultrasonic)

                        output logic                    motor_c1, motor_c2, motor_en);


      //parameter
      //parameter

      //seperate direction and speed from writedata
      logic             direction;
        logic[3:0]speed;


        always_ff @ (posedge clk) begin
                if(reset) begin
                        direction <= 0;
                        speed <= 0;
                end else if (chipselect & write) begin
                        if(address == 0) //set direction
                                direction <= writedata[0];
                        else //set speed
                                speed <= writedata[3:0]; //since the speed higher than MAX_SPEED_LV will be
deemed as MAX_SPEED_LV, we don't check here.
                end

        end
      regular_motor r_motor(.*);
endmodule
```

regular_motor.sv

```
module regular_motor(
                                        input logic clk, reset,
                                        input logic [3:0] speed, // 0 is stop; speed level from 1 to 8
                                        input logic stop, //stop signal for emergency stop that will feed from
ultrasonic sensors
                                        input logic direction, //forward and backward
                                        output logic motor_c1, motor_c2, motor_en //for motor pins
                                );


        parameter       MAX_SPEED_LV = 8; //from 0 to 8. (should <= 15)
```

```
                    //MAX_SPEED_LV_P_ONE = MAX_SPEED_LV + 1'b1; //used to optimized
circuit, no need to change

    enum logic {FORWRAD,BACKWARD} dir;
    //enum logic [3:0] {STOP, SPEED1, SPEED2, SPEED3, SPEED4, SPEED5, SPEED6, SPEED7,
SPEED8} spd;

    //convert input signals to what we use and do verifying
    assign dir = (direction == 1'b1) ? FORWRAD : BACKWARD;



    logic [14:0] counter_10000;
    always_ff @ (posedge clk)
    begin
        if(reset) begin
            counter_10000 <= 0;
        end else begin
            if (counter_10000 < 10000)
                counter_10000 <= counter_10000 + 1'b1;
            else
                counter_10000 <= 1'b0;
        end
    end

    logic tick;
    assign tick = (counter_10000 == 10000)? 1'b1: 1'b0;

    logic [3:0] counter; //count pulse width
    always_ff @ (posedge clk)
    begin
        if(reset) begin
            counter <= 0;
        end else begin
            if(tick == 1'b1) begin
                if (counter < MAX_SPEED_LV)
                    counter <= counter + 1'b1;
                else
                    counter <= 1'b0;
            end
        end
    end

    //motor control
    always_comb begin
        motor_c1 = 1'b0;
        motor_c2 = 1'b0;

        if(!stop) begin
            if(dir == FORWRAD)
                motor_c1 = 1'b1;
            else
                motor_c2 = 1'b1;
```

```
		end
	end

	assign motor_en = (speed == 0 || stop)? 1'b0 :
						(counter <= speed) ? 1'b1 : 1'b0;

endmodule
```

steering_motor_qsys.sv

```
module steering_motor_qsys(
			input logic			clk,
			input logic		reset,
			//Please refer to regular motor one for the reason why I use 32bits instead of 8bits
			input logic [31:0]	writedata, // Only accept 8, 16, 32, 64,... bits;
			input logic		write,
			input logic		address,
			input logic		chipselect,

			input logic		stop, // emergency stop signal send from other peripheral
(Ultrasonic)

			output logic		motor_c1, motor_c2);


	//parameter
	//parameter

	//seperate direction and angle from writedata
	logic		direction;
	logic[2:0]angle;


	always_ff @ (posedge clk) begin
		if(reset) begin
			direction <= 0;
			angle <= 0;
		end else if (chipselect & write) begin
			if(address == 0) //set direction
				direction <= writedata[0];
			else //set angle
				angle <= writedata[2:0]; //since the angle higher than MAX_ANGLE_LV will be
deemed as MAX_ANGLE_LV, we don't check here.
		end
	end
	steering_motor s_motor(.*);
endmodule
```

steering_motor.sv

```
module steering_motor(
					input logic clk, reset,
					input logic [2:0] angle, // 0 is stop; angle level from 1 to 7
```

```
                        input logic stop, //stop signal for emergency stop that will feed from
ultrasonic sensors
                        input logic direction, //right and left
                        output logic motor_c1, motor_c2 //for motor pins
                    );


    parameter       MAX_ANGLE_LV = 7; //from 0 to 7. (should <= 7)

    enum logic {RIGHT,LEFT} dir, old_dir;
    //enum logic [3:0] {STOP, SPEED1, SPEED2, SPEED3, SPEED4, SPEED5, SPEED6, SPEED7,
SPEED8} spd;

    //convert input signals to what we use and do verifying
    assign dir = (direction == 1'b1) ? RIGHT : LEFT;


    //we want reduce PWM clock freq from 50M to aruond 5k
    logic [14:0] counter_10000;
    always_ff @ (posedge clk)
    begin
        if(reset) begin
            counter_10000 <= 0;
        end else begin
            if (counter_10000 < 10000)
                counter_10000 <= counter_10000 + 1'b1;
            else
                counter_10000 <= 1'b0;
        end
    end

    logic tick;
    assign tick = (counter_10000 == 10000)? 1'b1: 1'b0;

    logic [2:0] counter; //count pulse width
    always_ff @ (posedge clk)
    begin
        if(reset) begin
            counter <= 0;
        end else begin
            if(tick == 1'b1) begin
                if (counter < MAX_ANGLE_LV)
                    counter <= counter + 1'b1;
                else
                    counter <= 1'b0;
            end
        end
    end

    //use to save old angle and dir so that when receive stop signal steering motor can
    //keep current angle no matter how input angle changes.
    logic [2:0] old_angle;
    always_ff @ (posedge clk) begin
```

```
		if(reset) begin
			old_angle <= 0;
			old_dir <= RIGHT;
		end else if(!stop) begin
			old_angle <= angle;
			old_dir <= dir;
		end
	end

	//motor control
	always_comb begin
		motor_c1 = 1'b0;
		motor_c2 = 1'b0;

		if(!stop) begin
			if(angle != 0) begin
				if(counter <= angle) begin
					if(dir == RIGHT)
						motor_c1 = 1'b1;
					else
						motor_c2 = 1'b1;
				end
			end
		end else begin
			if(old_angle != 0) begin
				if(counter <= old_angle) begin
					if(old_dir == RIGHT)
						motor_c1 = 1'b1;
					else
						motor_c2 = 1'b1;
				end
			end
		end
	end
endmodule
```