Designated Driver

Smart vehicle project documentation

Group 9

Members: Sara Johansson, Simon Lobo, Peili Ge, Flutra Tahiraj, Mengjiao Wei, Markus Erlach, Fredric Eidsvik, Malcolm R. Kente

Table of contents.

Project & Milestone Organisation:

// Malcolm Kente

The project's overall structure is broken down to sectors of hardware and software respectively. The hardware sector focuses on the assembling and connectivity of components to the car. Also included is the functionality testing of components so as to ensure compatibility and steady operation on the car. The software sector focuses on the implementation of code on the virtual environment (Open DaVinci). Methods and functions are set on the virtual components within Open DaVinci to have the virtual car operational. The virtual implementations are then later integrated with the actual RC vehicle.

The group split into two groups that worked on the hardware and software sectors. For each milestone, both teams would prioritize the goals to be met before presentation. The hardware team's main priorities were to assemble and test components to make certain that everything worked as it should. The software team's main priorities were to ensure functionality of the differents states the RC vehicle would operate on (Lane following, Parking, Overtaking,Intersection). Both teams integrate their parts at the end for a fully operational RC vehicle.

1.Milestone Table

	22/1/2015 -11/2/2015	**1st Milestone (12/2 - 18/2)**	**2nd Milestone (19/2 -11/3)**	**3rd Milestone (12/3 - 01/4)**	**4th Milestone (02/4 -16/4)**
Markus	Research	Platform Independent Logic & LLB->HLB	Platform Independent Logic & LLB->HLB	Proxy	Proxy & Hardware testing
Peili	Research	logic for parking, test parking in simulator with hardcoding	logic for parking, test parking in simulator with hardcoding	Set up Ordroid connections	Data recording for LaneFollowing, and Serial link logic
Mengjiao	Research	overtaking logic IrUs chart recrdings	platform-independent logic for overtaking	parking with all sensors in simulator	parking with all sensors in simulator
Sara	Research	Sensor	OpenCv/Lanedet	LaneFollowing	Get Image to

		Investigation	ection		work on live camera.
Flutra	Research	State machine. Logic for intersection. IrUs chart recordings	potential connection plan for the different hardware components	overtaking in simulation	Overtaking with the sensors in simulation. and getting more understanding for lanefollowing, to get overtaking to work
Malcolm	Research	Platform Independent Logic & LLB->HLB	Platform Independent Logic & LLB->HLB	Hardware pick-up and scheduling @ CRF	Hardware
Fredric	Research	state machine & making the car drive in OpenDaVinci	Parallel parking	vacation	Hardware
Simon	Research	Making the car drive in ODV	potential connection plan for the different hardware components	Intersection	Intersection

	5th Milestone (17/4 - 22/04)	**6th Milestone (23/4 - 29/4)**	**7th Milestone (30/4 - 7/5)**	**8/5/2015-22/5/2015**
Markus	Proxy & Hardware testing	Proxy	Proxy & Track testing	Proxy & Track testing
Peili	Data recording for parking and overtaking	encoding and decoding for LLB and HLB sides	encoding and decoding for LLB and HLB sides	Test-cases for encoding and decoding. Read sensor board data and store in Proxy, Documentation

Mengjiao	parking without using ultra sonic right Rear in simulator	parking without using ultra sonic right Rear in simulator	parking without using vehicle's heading in simulator	parking without using vehicle's heading and works in real track scenarios in simulator. And documentation
Sara	Help collide Overtaking and Parking with Lanefollowing	Get LaneFollowing into car	Get LaneFollowing into car & organize documentation	Documentation
Flutra	Overtaking and getting more understanding for lanefollowing,and intersection to get overtaking to work Got a menu in the terminal for the integration. Can choose if parking or normal driving.	Improving intersection handling to make it work with overtaking. (With simon)	Got overtaking to work with the original scenarios, Overtakning and Parking, integrated with the menu.	Changing the code to work with the acceptance test scenarios. Easy pass,medium works Refactoring code Documentation
Malcolm	Hardware	Hardware	Hardware/Documentation	Documentation
Fredric	Hardware	Hardware	Hardware/Documentation	Documentation
Simon	Intersection integration with Parking	Proxy	Proxy/Overtaking	Documentation

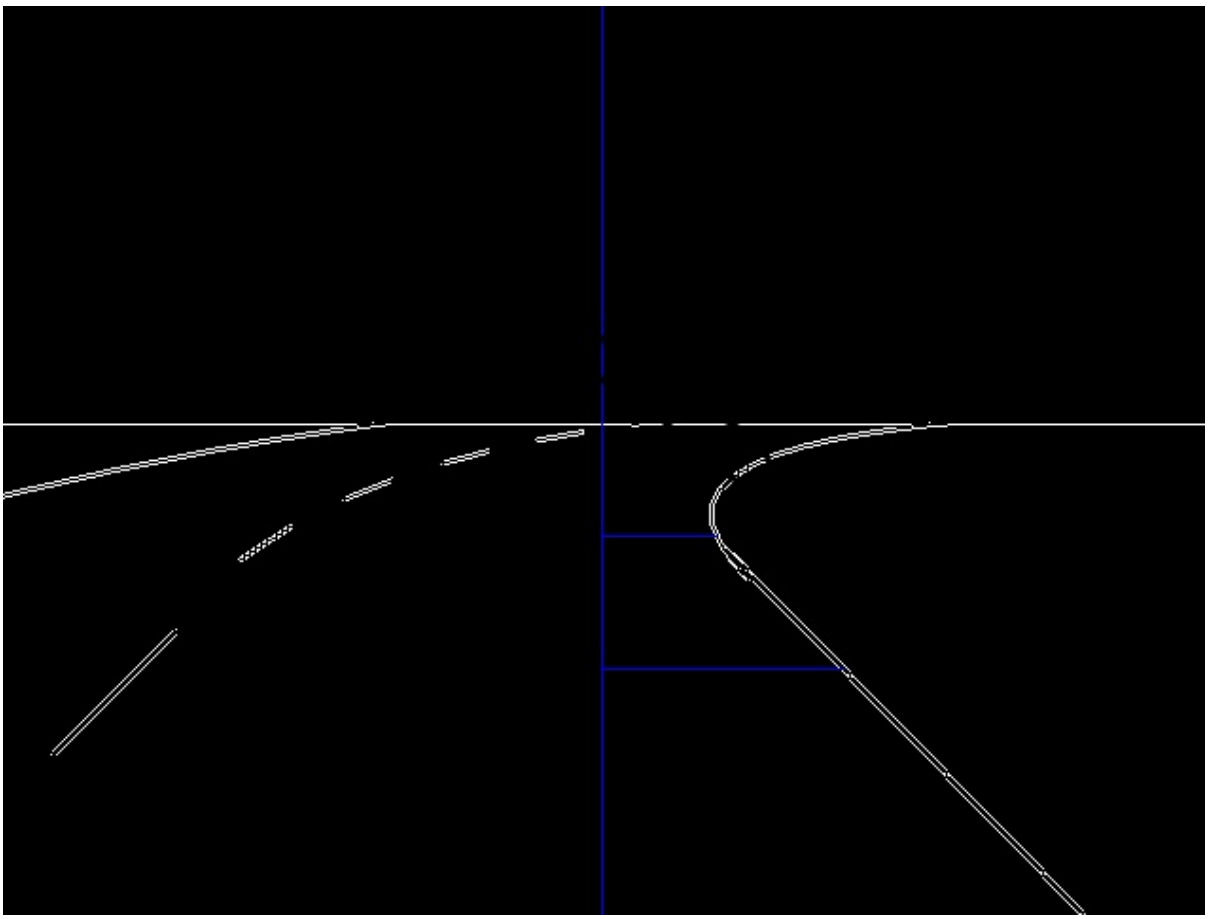
Algorithmic Aspects:

//Sara Johansson

LaneDetection and Lanefollowing:

To detect the lane we use lines that are drawable with the opencv libraries. One line is drawn vertically in the middle of the image by using the width and length. From the middle line another line is drawn towards the right using a while loop to count how much should be added to the x-variable of the end point.

2.Picture of LaneDetection



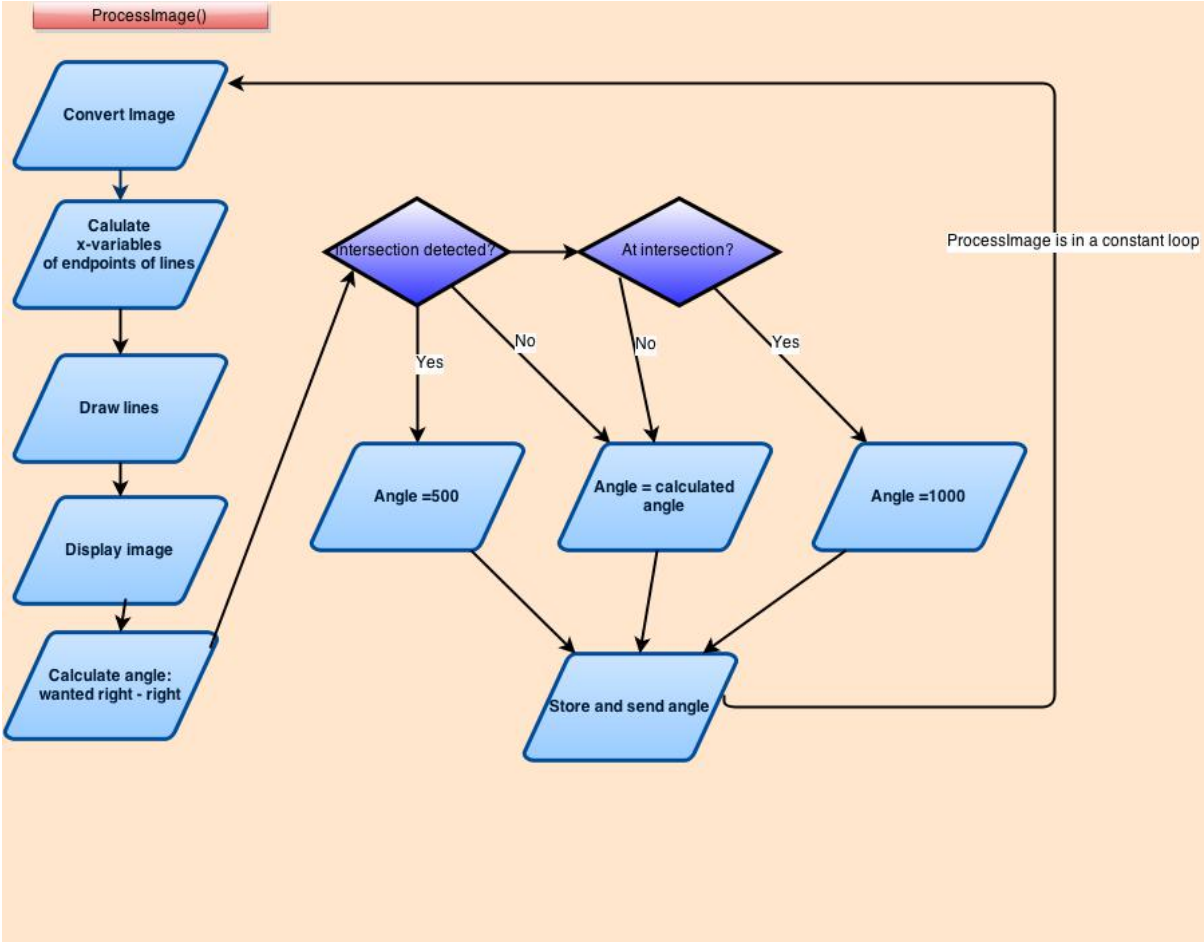
The lanedetection worked fine in the simulator but in the real camera it would not detect the white line. This was easily fixed by first converting the image to Grayscale and then implementing canny edge detection on the image.

Using the right line we could figure out the desired length of the right line, this desired length signifies that the car is straight. Any change in the right line would then be calculated and the difference between the current length and the desired length is converted to degrees. This is then sent as an angle to the driver and is later

on saved as the steering angle unless overtaking,parking or intersection is happening.

The overall idea of lanedetection and lanefollowing is to keep it as simple as possible. Therefore we are using just one line that keeps track of the movement, whether the car needs to turn or continue straight on. The four right lines, were a backup plan that were created for the scenario where the lanefollowing would would work poorly on the real track then the backup lines could be used as well. The backup left lines work such that it skips the line that is long, between the white middle lines, and goes to the next line to retrieve the angle. The backup left lines do work without the left line although it is not as safe as using the right line. Therefore, the right line is being used instead of the backup lines.

3.Diagram.



//Simon Lobo

Intersection.

The intersections handling is very simple but effective and one could say it uses both the lanedetector and the driver. Intersections works by using all the lines drawn to

the left, the road markings, to check if the car is approaching an intersection. The driver receives a special “angle” for this and therefore it knows what lays ahead. When all the lines to the left cannot detect road marking we ́know that we are at an intersection and the lanedetector sends another special “angle” to the driver who then stops. The ultrasonic sensor pointed to the front right checks for oncoming vehicles, if another car approaches it continues to stay still. If no vehicles are approaching and after the car has stopped for a while at the stop line, the driver continues to cross the intersection. The car then continues the lane following and receiving the special “angle” so that it knows what lays ahead .

In summary, when the car is approaching an intersection it detects that by the no longer existing road markers and the driver stops. If there is no other vehicles coming it continues forward, past the intersection and resumes to follow the lane.

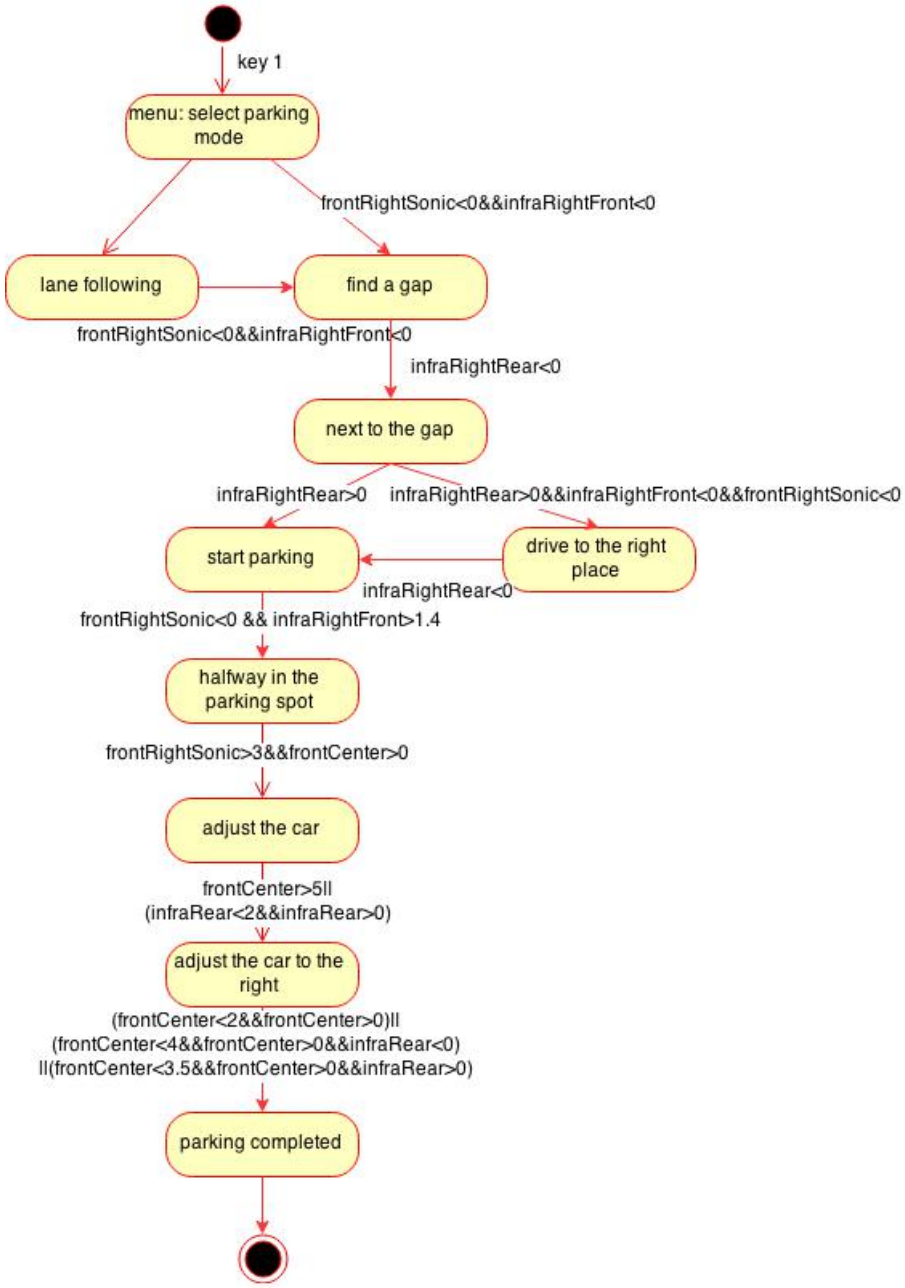
//mengjiao

Parking.

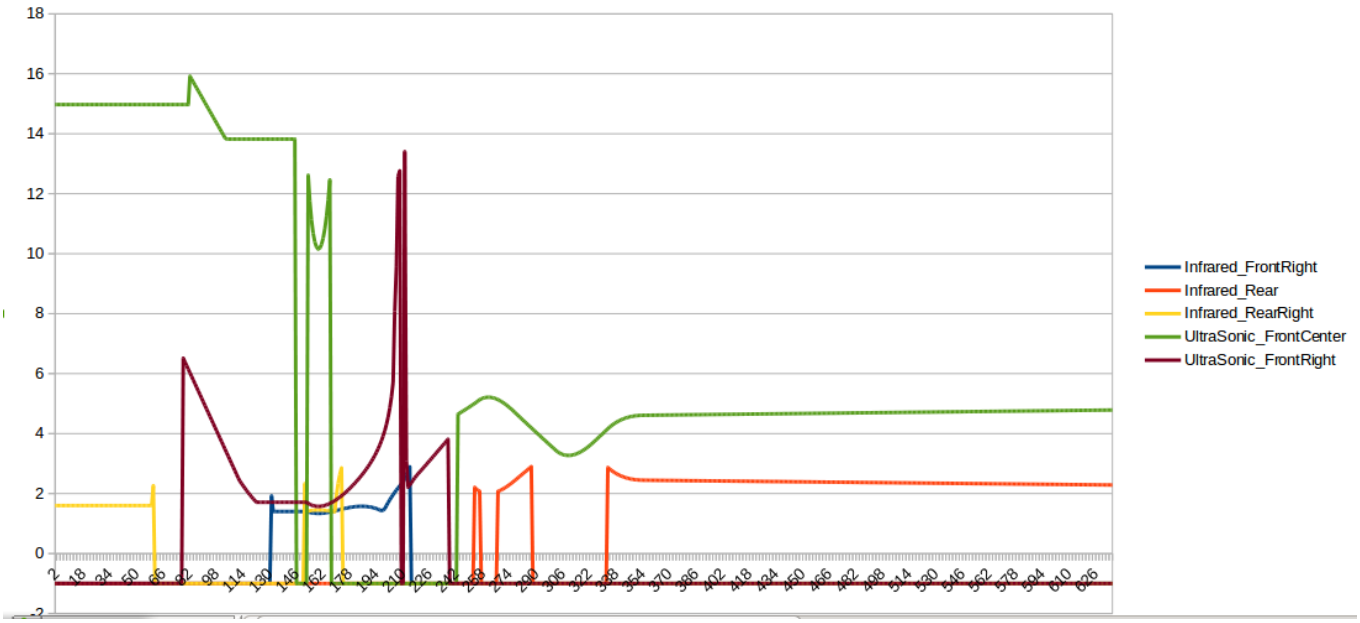
we divided the parking function into four parts, looking for the parking spot, drive to the right place, parking and adjust the car.
We tried to use UltraSonic_RightRear sensor to detect a parking spot and move the vehicle pass the parking space before we plan to park in and use the car’s heading to parking the car, but we do not have UltraSonic_RightRear on the real car and we can’t get heading of the car. So we needed to change the algorithm of the whole parking function without using UltraSonic_RightRear sensor or car’s heading. Our parking algorithm works for parking-easy, parking-medium and parking-hard, but
In the parking algorithm, we use switch-case statement. In the default statement, we use the lane following function to make the vehicle move.
For detect the parking space, we use UltraSonic_FrontRight sensor. When there is nothing on the right front of the car, the UltraSonic_FrontRight get the distance is -1. Then the car can keep move to the next stage to drive to the right place.
For drive to the right place, the car keep moving until the right place to start parking. We use infra_RightRear sensor to check if there is an obstacle next to the car, infra_FrontRear sensor get the distance is -1, then the car move until the distance is larger than 0. However, for the parking-hard scenarios, the gap is too small to park so we need to drive a little bit more. So we check the infra_FrontRight and UltraSonic_FrontRight, vehicle knows this is the hard scenario if there is nothing on the front right of the car. The vehicle will move forward a short distance and then start parking the car.
For parking the car, we use Infra_FrontRight and Infra_RearRight to park in the car. When the vehicle is half part in the parking spot start adjust the car. We use Infra_Rear and UltraSonic_FrontCenter for adjust the car into right position. We have

three if statements to check the car and to make it stop without using check car's heading.

4. diagram for parking



5.Sensor chart



This is the irus chart from one of the parking scenarios. From this diagram, we can five sensors which we used for the parking function changed the data.

//Flutra Tahiraj

Overtaking:

When the user runs driver.cpp there will be a menu at the start of the terminal. The user gets to choose between the mode options; normal driving and parking. We decided to give the user this option because it's not possible to know when the car should actually park. When the parking mode is chosen the code for parking will then be executed.

When normal driving has been chosen the car will start driving normally with lane following and will be able to manage overtaking and intersections.

The code for these functionalities are all integrated in the Driver file and easy to handle with help of this plain menu.

The theory behind Driver is that it is written in if-statements that will correspond to the states of the car. The code will handle boolean values where a state will only be executed if the arguments are true. The first value that will be sent to the car is the data of lane detection, that will then activate lane following if the data is correctly received.

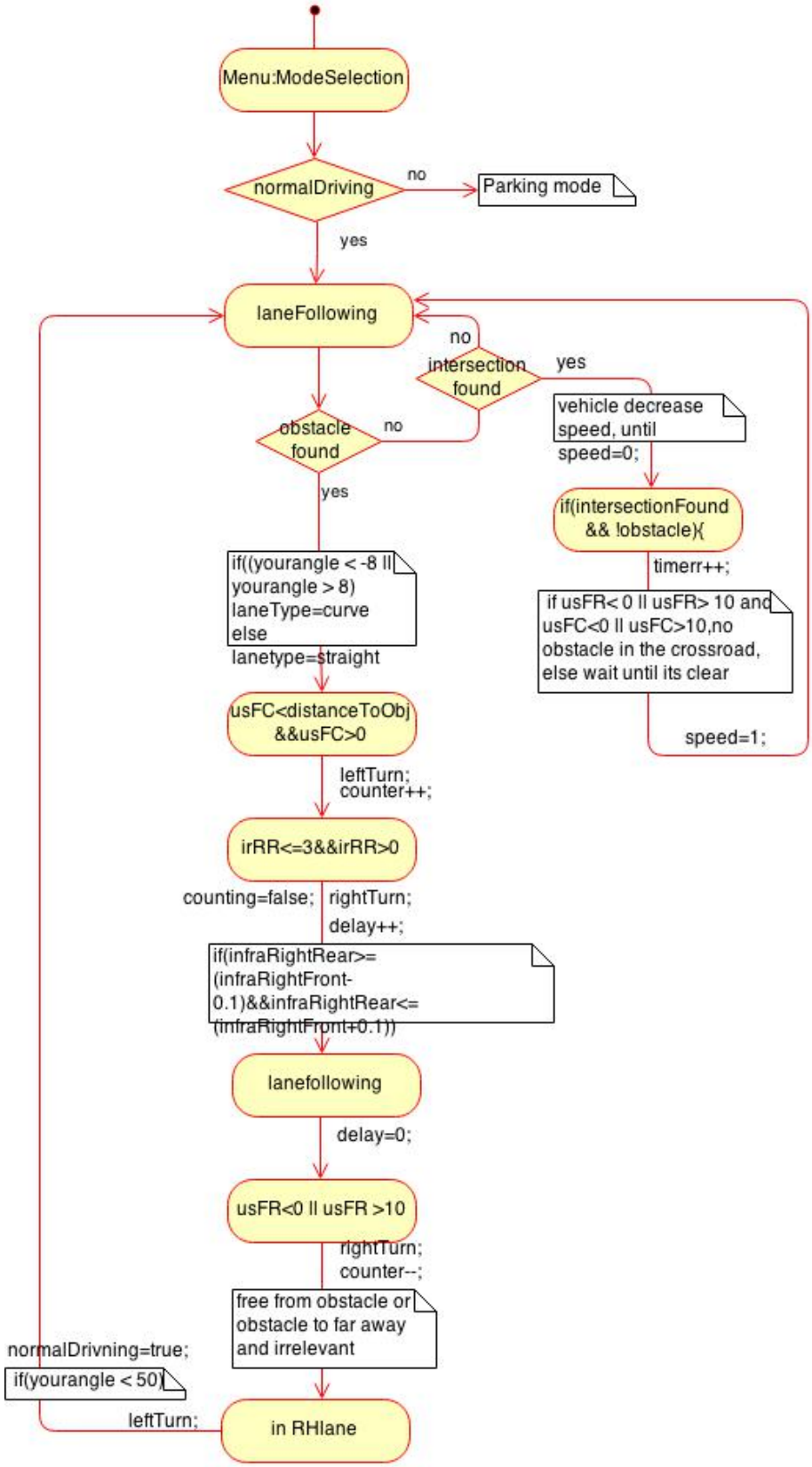
The car will get out of the lane following handling state when it's triggered by another receiving data and the arguments are no longer true, such as when data that enables overtaking and intersection handling. Lane following will be set to false to be able to set a solid steering angle for the car to drive parallel to the obstacle and re connect with lane following when lane is found and overtaking is completed. When a crossroad has been detected the car enters another state where there will be a delay and a flag that will return if the intersection is clear or not and if it is clear the steering angle is fixed at 0 until the lane is detected again and the car can enable lane following again.

The algorithm behind the driver is based on states. Because it has been written by more than one person the style is different.

Overtaking algorithm:

The process of the overtaking is managed by five states, and only one state can be true at a time.
Depending on what lane type the vehicle is on, ie. a curve or straight, the code had to be adjusted to be able to complete the action. This was fixed by knowing the retrieved steering angle, the distance to the obstacle, the speed and the turns could then be adjusted to make overtaking possible.
When an obstacle is detected lane following is stopped and the vehicle makes a left turn, until the right rear infrared has detected the obstacle it will continue turning. When this sensor has detected the obstacle the vehicle enters a new state, where a counter is implemented for measuring this turn. When the front and rear right infrareds have almost the same value (+-0.1) the wheel angles are fixed so the vehicle drives in parallel with the obstacle working with lane following until the front right ultrasonic is free from the obstacle and the vehicle can safely return back to right lane. By decreasing the counter the turn back will have the same curve as before until the wheel angle is in a certain angle to get enough data so the current action can be clipped by lane following again. This routine will constantly happen when the vehicle detects an obstacle except from when it's on an intersection.

6. State Machine chart



Implementation of algorithmic fundamentals:

//Markus Erlach

Proxy:

Our proxy is built using termios which is a UNIX api for terminal input and output. When using termios you first Open a serial device. You then setup the communication options you want to use, using the different termios options. When you have setup the communication options you use the termios standard read and write functions in order to read from the serial device as well as write to it. The device can then at the end also be closed using the close() function.

There are a lot of different ways of going about setting up the options for the communication but we chose to use a fairly standard way of doing it with settings mostly gathered from similar applications and following guides like:
http://en.wikibooks.org/wiki/Serial_Programming/termios
The more important options used are VMIN which waits a minimum of characters before continuing. In our case we have chosen 1 to make sure that some data is being read.

HLB:

When the termios settings were configured, we began by using the connection without all the additionals of the regular proxy.cpp file handed to us with the Open DaVinci. This made it a little simpler to run and test reading and writing char arrays or strings back and forth. To begin with reading and writing data we primarily focused on using hardcoded values that we created to make it simpler to see if something did not work or went wrong.
Once we had it running well using the arduino api with our test proxy, we started to test using the car. This was when we had to tweak a lot of values to make sure the ESC and servo could receive and use the values sent from the test proxy. When the ESC and servo were working fine we wanted to start using the original proxy and started running in to some minor problems again. Our camera reacted a few seconds late which made the wheels turn late which meant our car had already driven off the road before it started to turn. We weren't sure what was wrong or how to fix this issue and struggled for a few days trying to fix it. The problem was that inside the arduino file we were using, we had a few delays when we were testing to be able to better see what we received and sent and all these delays added up quite a delay that in turn slowed down the camera. When we had realized this and removed the

delays the car started reacting in time and we were able to drive it correctly on the track.

The data we send to the LLB is first retrieved from the driver which has been stored in containers by using the camera and lanefollowing. We then handle that data by making it into degrees from radians for the angle we wish to send to the car. That angle is checked if is positive or negative, based on what is is, it gets handled differently so that we can correctly add it to a char array to be sent. We either add the value to 90 or remove it from 90 depending on if it was negative or positive. This lets us send a value that is usable by the servo. After that we create an int out of to round off the value.
Once it is checked and made into an int, we do the same thing for the speed value and then we add the values along with either padding, which is additional zeros if a value is below a certain point or as the values are. We then create a checksum value to send with the values to be checked on the LLB to make sure it is the correct values received. All these values are added to a char array because a char takes less space and is sent more quickly. That array is then send to the LLB through the serial link.

//Simon Lobo Roca
The lane detector sends what angle the wheels should turn in containers, the driver access this data and sets the steering angle. The proxy reads the set angle which is in radians and converts it to degrees. When the cars wheels point forward the degree is 90, the minimum value is 65 when the wheels point to the left and maximum is 115 when the wheels are pointing to the right. When the driver sets the steering angle to a negative value it means the we must turn to the left. As briefly explained before if we receive the value -15 we subtract 15 from 90 and if we receive the value 15 we add to 90. After this is converted to an integer we can send the values to the low level board. Since we insert the values in character arrays we need padding if the number is below 100, we perform the same process as we did for the speed.

As the proxy receives the container data from the driver it must handle the speed data in an appropriate way. The speed data is received as a double, there are several ways to handle this double. The way we chose to handle this value is to convert the double into an integer. The drivers most common speed values are 0.0, 0.5, 0.8 and 1.0 so we must change these values which are in double in order to convert them to integers. Our solution for this is that we simply multiply them with ten so that our new values are 5.0, 8.0 and 10.0 respectively. Now these new values; 5.0, 8.0 and 10.0 are more useful and manageable.

Our next step was converting them to integers, then put them into character arrays together with our steering angle values and sending it to the arduino. Unfortunately, the arduino expects two digit values for the speed so sending in the values of 5 or 8 would not work. We had to make sure to send in the values of 05 or 08, which has been previously done for the steering angle. In order to fix this issue we start by checking if the value is under 10. If that is true we will use padding, adding zeroes in front of the value in the character array. Thus when we send it to the arduino it will get two digits no matter what the original speed value was. For example, if the value is 8, it checks to see that the value is under 10 and if that is the case it padd in a zero in front of the value in the character array and send the two digits 08 to the arduino. This solved the problem with the handling of the speed data.

// Peili Ge
Logic for Encoding and Decoding in Proxy for HLB
After we received data from LLB, a string like “<16:150:1020304050>”, we need to decoded the data and store it. By using the function readSerial(), first we check the length. If the string is too short or too long, and then to check if the string contents ‘:’ , then we check if it contents any letters, if the string we received matching each of these, it means we received wrong data. All these steps are checking if we received right data or not. Then we start reading the string, we start reading from ‘<’ where it begins, and we skip the first 3 digits with the length of the data, and we will finishing reading until we find/reach the ending ‘>’. After we finish reading the data,we then remove the ‘<’ and ‘>’ away from the string, as well as the length and ‘:’, the pure data we are going to check the sum looks like “150:1020304050"
After we successfully read the data, we need to check if its the matching data. By check the sum of the value. Then we decode the string by break it down to single digits, and assign which sensor to which matching data. Finally, we save this data values in our sensor Board container.

We will send encoded data to LLB which we stored in the container from HLB, we use net string to encode the format by adding all the sensor data to a list, and converting to string. Then we check the length and the sum of the data, and set it before the value. Simply like the string we received from LLB.

//Markus Erlach

LLB:
When we assembled all the different sensors and parts on the car we created test files to test and use each sensor. These test files used the different parts or sensors to retrieve data from the surroundings to be able to send it to the HLB in order to use

the retrieved data to change direction or even speed. These functions are all in the arduino file to be used in order to retrieve and use the data in the loop.
For the arduino side of the proxy we created an arduino file that contains a loop that continuously writes to the ESC and Servo as well as read data from the different sensors in order to be able to send it on to the HLB via the serial link.
When the data is retrieved, we have a few if statements so that we can control the values we want to send. For example, an IR value greater than 99 centimeters is a value we do not need so we set it to 99 if greater and then make a string out of it.
Once all the data has gone through all the if statements, we add up all the values to create a checksum to send first in our char array. The newly created strings of sensor data is inserted into a created char array that we wish to send along with the checksum separated by two colons.
Like: 100:2020202020 where all the sensor data together add up to the checksum of 100.
To this we also add the length to send through so that we know how much to reach on the HLB side. Then finally we have a start and an end char to make it: “<16:100:2020202020>”.
After our char array is created we send it through the serial link with Serial.write() and that is the last thing done in the loop.

After the loop we have a function called serialEvent which runs between each iteration of our loop. This function is what reads data received through the serial link in order to be able to use data sent from the HLB. When we read in the data, we add it to a char array so that we can manipulate each char received. When the read does not read any more chars we null terminate the array at the current index of the array. Once the array is null terminated we check the first, the second and third index value if they are digits as well as the index to make sure it received the correct length. If they are and it passes the if statement check, we call a function called setData which makes an int out of all the chars in the array so that we can add them up to check the values and compared them to a checksum that is part of the char array we receive. Before using these values, we check them in if statements to make sure they are within the bound we deem reasonable for the values that should be received to avoid errors. If these values are correct, we use them to change the direction and speed of the car as long as they are within max and min values allowed.

Reasoning for using padding in LLB and HLB proxy:
By padding I mean adding a zero in front of a digit that for example might be less than 10 when our values range from 0 to 99. This makes it so that certain values always have the size 2 in a char array. And we know the index of each value.
The reason I chose to pad the values stored in different char arrays is that I find it to be easier to use and looks better in my opinion. It does not change the length of an

array that is being sent so therefor I felt that using padding also made it easier to use with reading in the array when we will know and can have control over what will be where and how to extract the different values we are receiving. Another major reason I decided to use padding was that I wasn't sure if the group would have time to implement a read function that could use the length to read in the correct amount of data in the HLB.

//Simon Lobo
The arduino reads the data sent and stores it in a char array where it checks each value. The values come with a checksum as previously described and all the values for both speed and steering angle. The first steering value is multiplied by onehundred, the second by ten and both the resulting values are added together with the third steering value. The angle is then set and sent to the servo. The first speed value is multiplied by ten and then added with the second speed value. Now we have a correct speed value of either 0, 5, 8 or 10. The arduino will now set the throttle accordingly: speed value 0 gives throttle 1500, speed value 5 gives throttle 1570, speed value 8 gives throttle 1600 and speed value 10 should give throttle 1620. Throttle 1500 means that the car does not move and of course it increases after that. Throttle of 1620 is the maximum and it should not go over that, so we have a guard making sure that it will not surpass 1620.

The car is going to need to reverse at some point, during the parking scenario. That means that the driver will send the negative value of -0.5. If we simply multiply this -0.5 with ten we will get -5 (-0.5*10) and it will be two characters in the array. The first character will however be “-” (minus) and the arduino cannot handle that character. To solve this minor issue with if the value is negative, we simply multiply it with -30 in order to get the positive value 15 (-0.5*-30). Since the driver will never send the value 1.5, it is safe to assume that when the arduino receives 15 it means that the driver wants to reverse. Consequently, the throttle will be set to 1450 which means reverse.

//Peili Ge
Logic for Encoding and Decoding in Proxy for LLB
From Arduino side, after we received data from IR sensors and UR sensors, we need to encode them and send those datas to High level board. By encoding the data, we used net strings as format. First we get the length of the string (store as N) and place it in the beginning, then we separate the N and the data by ‘ : ’, then it follows with the data we received from 5 sensors (we have 3 IR sensors, 2 US sensors), and we make all 5 sensor datas into the same string. It looks like: “150:1020304050”
After received this encoded string sent from Arduino, we need to know where to read from the start and where to end, therefore we added ‘<’ and ‘>’ for let HLB knowing

how long is the string and where it should ends. Additionally, we added checkSum() function in the Proxy in HLB for checking if the received string is matching the sent string, we send the sum of the 5 sensors as well, after the length of the string. Thus, the encoded string looks like : “<16:150:1020304050>”

When we decoding the data we received back from HLB, for example the data for parking, then we convert the data to make the car running. To decode the received data, we first count the length of the string, and break each value done to 2 digits for each sensor data to count the sum of the values, to see if its matching. Then we convert each sensor data to make the car running.

We have tested this functions in proxy.ino file with sensors, it works well by receiving the data and decode them, but we use this method as backup.

//Fredric Eidsvik
The checksum function takes in the array of “100:2020202020” and then it will first check the first digits before the :. If it is above 99 it will go into the first if statement. Otherwise it will go into else.
The difference between those is where we start to count. Let's say that we get: 50:1010101010. Then we can’t start to count the first sensor on [4]. So to summarize. Checksum above 99? start to count the first sensor on [4]. There are 5 sensors each with two digits(sensor1[2]). We set two digits to max so any sensor can’t go above 99.
If the checksum matches the added sensors it will store it in a container.
If it does not match it will do nothing

//Simon Lobon Roca
Odroid
Compiling the OpenDaVinci software on the odroid proved more difficult than we expected. We faced many obstacles and challenges simply with compiling the OpenDaVinci software. Some of the obstacles were minor, but still caused a lot of problems and manly time delays. Two of the problems were quite ironic considering the fact that we conduct our work in an university of technology, nonetheless it proved nearly impossible to get a computer screen with a hdmi input, since all the computer screens available was without hdmi input and the fact that the transfer speed to the odroid reminded of the early dial-up modem internet.

These obstacles and challenges were resolved by borrowing a computer screen with a hdmi input from a senior lecturer and head of the Software engineering department, Imed Hammouda, at the university, spending a lot of time waiting for

the transfer to be complete and we even considered bring everything home and have it run in the background, in order to save time for the group as a whole.

After finally getting all the necessary files into the odroid we faced a couple of additional challenges. One of the challenges was that it proved impossible to install the software itself, major compiling problems arose and it said that several libraries had to be removed. This issue has also been pointed out by several other students and their groups. The hesperia and libvehiclecontext folder was not to be created. After commenting or removing the lines on the CMakeLists we found out that several testing scenarios used hesperia/vehiclecontext. We solved this issue by checking each error line that we could find and remove the code that was trying to install these test scenarios. At long last, we could finally install the OpenDaVinci software on the odroid.

After we got the odroid to work it worked just as any computer running the software. We would run code out of our computers to test and then transfer it to the odroid. Since other students set up a VNC we could easily access the odroid remotely and transfer all our code into it.

Basically, everything had to be tested, as soon as we completed something on our personal computers we would upload it to the odroid and test it. Everything that was tested, that worked on the computer, worked on the odroid. Thanks to the VNC we could simply connect remotely to the odroid and send all our files. Which made the process of checking and testing our work much easier and fast.

Our lane following was the first thing we tested, along with intersection handling. As stated before we did not foresee the wall posing as a threat or an issue to our intersection handling. Therefore our first test was mainly lane following, and it passed with flying colors. When the delay problem was fixed on the arduino, everything worked just as well on the odroid as it did on our own personal computers. Running the proxy and the lane detector at a frequency of 30 proved to work efficiently.The car received the correct values and was able to run the track without problems. When we fixed the intersection handling it worked as expected without any major problems. The extra lines drawn by the lane detector worked somewhat acceptable but it seemed that our vehicle was handling everything as it was supposed to.

Implementation details of source code:

//Sara Johansson
In order to detect the lane we use the following which counts the x variable of the right lines end point. We use the widthstep that exists in the image and named it ”move”.So if the widthstep of (height-sample_right+2) and ((width/2+right) *3) is zero that means that the widthstep is still not at the white line so it adds one to the variable right. It counts how many steps from the middle to the right the line has to be drawn until it comes to the white line.

```
while((image + move * (height- sample_right+2)) [(width/2+right)*3]==0 && right < width/2 ){ right ++;
}
```

We then implement and use this variable to set the right lines endpoint.

```
cv::Point pt2;//right end
        pt2.x=width/2+right;
        pt2.y=height -60;
```

Then the difference between the current length (right) and the distance we want the car to be positioned at is found:

```
double angle = ((right + 1) – desired_right);
```

This is then converted to degrees and sent to the driver as well as stored in a container.

The driver consists of the overtaking, parking and intersection. The intersection is done partially in the lanefollowing and partially in the driver.

To handle all of these aspects in an ordered manner we used states. The normalDriving state is the state that the car is always in, unless it is told to park or an obstacle is detected, and the normalDriving state assigns the lanefollowing and the intersection to always be active.

The intersection is detected in the lanefollowing and sends absurdly large angle to alert. The first angle that is more than or equal to 102 signifies that an intersection is detected thus decreasing the speed of the car. The second angle that the intersection part sends is to signify that the car is at the stop line thus setting the speed to zero.

```
if(normalDriving){
        if(yourangle <100 ){
                cerr <<" lane following" <<endl;
                steerAngle = yourangle;
                sec = 0;
        }
        if(yourangle >=102 ){
                vc.setSpeed(0.5);
                steerAngle = 0;
        }
        if(yourangle >= 204 && !obstacle){
                intersectionFound = true;
                normalDriving = false;
                vc.setSpeed(0.0);
                steerAngle = 0;
        }
}
```

When the steering wheel angle is stored it is stored as radians so when the proxy gets the angle it is converted back to degrees again since it is easier to work with larger number rather than decimal numbers. We filter out the angles before we send it to the arduino.

In this example we make sure that a value less than 100 is padded with a zero before it is stored in the character array and sent to the arduino

```
 if (intAngle < 100) {
            sprintf(fromIntAngle, "%d", intAngle);
            strcpy(padAngle, "0");
            strcat(padAngle, fromIntAngle);
}
```

//Flutra Tahiraj
The following code snip has been very necessary and important during the development while working with all the different scenarios. This part of the code had to be modified several times to successfully complete an overtaking maneuver.
The current code that will be shown below is for the acceptance test (medium), and the only change is the value for the vehicles right turn. Mainly, it is this part that should be changed to make it work with other scenarios.
But the idea and algorithm behind this code will still be explained here because as mentioned, during the development this was something that had to be modified.

The actual steering angle that the vehicle has determines what values that needs to be used by the overtaking algorithm.

```
if(( wheel_angle < - 8 || wheel_angle > 8 ) && !obstacle){
  vc.setSpeed(0.6);
  rightTurn = 21;
  distToObstacle = 6;
  laneType = “curve”;
}
```

This if statement will be true if the vehicle is on a curved track and the obstacle is not yet detected. The value of the right turn is adjusted, compared to if the vehicle would be on a straight road.

```
if(( wheel_angle > -8 && wheel_angle < 8 ) && !obstacle){
  vc.setSpeed(0.6);
  rightTurn = 25;
  distToObstacle = 6;
  laneType = “straight”;
}
```

This if statement will be true if the vehicle is on a fairly straight path. As clearly shown the code is similar and only the right turn was needed to be modified, but for previous scenarios in ODV this snip differed from each other. The distance to the obstacle and speed were also modified when overtaking in other scenarios.

//Mengjiao

```
default:{
        if(normalDriving){
                yourangle = sd.getExampleData();
```

```
                steerAngle = 0;
                vc.setSpeed(2);
                    if(yourangle <100){

                  steerAngle = yourangle;
                  sec=0;
                }
                if(yourangle >=102){
                  vc.setSpeed(0.5);
                }
                if(yourangle >= 204){
                  if( sec <= 50){
                    sec= sec + 1;
                  }
                }
              }
              if(frontRightSonic<0&&infraRightFront<0){
                stage = 1
              }
            }break;
```

This part of code is for we start the parking mode. UltraSonic_FrontRight sensor and infra_RightFront start looking for the gap, if both of them does not detect anything that means there is a gap. However, if one of them detect something the car will keep lane following. Outside the parking mode we set normalDriving to true.

```
case 2:{
        normalDriving = false;
        vc.setSpeed(1);
        steerAngle = 0;
        if(infraRightRear>0){
            stage = 4;
        }
        if (infraRightRear>0&&infraRightFront<0&&frontRightSonic<0){
             stage = 3;
 }
}break;
```

In case 2, the car already found a parking space, so we set normalDriving to false in case the car start to do something else. There are two if statements, one is when the infraRightRear drive pass the parking space then go to case 4 start park the car. Another if statement is when there is no obstacle in front and right of the vehicle, the car knows it is the parking-hard scenario, and then it gonna go to stage 3.

```
case 3:{
  normalDriving = false;
  vc.setSpeed(1);
    steerAngle = 0;
    if (infraRightRear<0){
          stage = 4;
    }
}break;
```

Stage 3 is aim to the parking-hard scenario, in the parking-hard scenario the gap is smaller than easy and hard. The car need to drive a little bit more and when the infraRightRear detect nothing next to the car then start park the car.

Hardware architecture and manufacturing details:

// Fredric Eidsvik

The hardware consists of 2 main layers. On these different layers all the hardware that is necessary to build a self driving car exist.

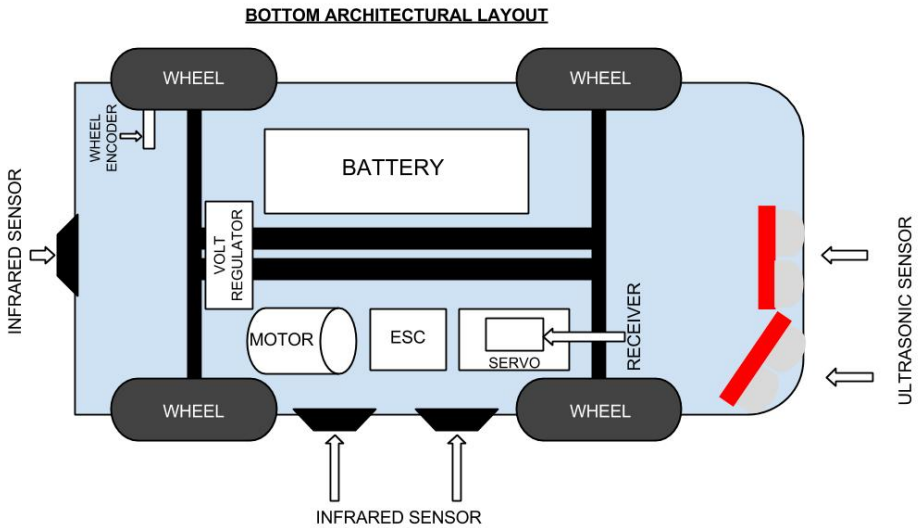
On the first level we have the wheels, suspensors, ESC, engine, battery, servo, receiver, infrared, volt regulator, ultrasonic and the wheel encoder.

On the second level we have the odroid, arduino, STM, voltmeter, power switch and a pillar where the webcam is mounted. On the back of the pillar the razor is mounted. The reason for the razor being mounted on the back of the pillar is that it can interfere with arduino and/or odroid. Because of this we decided to place it on the back of the pillar so when the chassis is placed it will be hard for the razor to interfere.

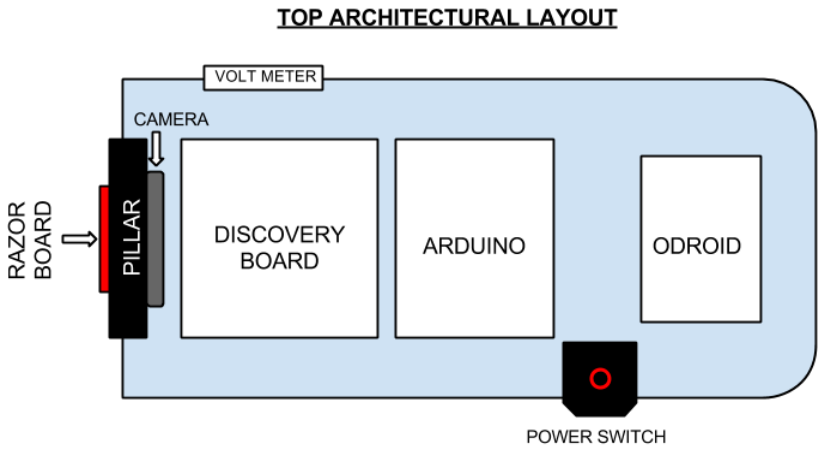
The reason for the boards being placed on the second level is because we want easy access to the boards so we can easily see what's going on with the boards when we are coding. The remaining hardware not described are components necessary for the project but not requiring any ease of access. These components will not be fiddled with once they are mounted. The exception being the ESC, which is placed in such a manor that the on and off button is easy to access.

// Malcolm Kente

7.Hardware layout



8.Hardware layout



Component connections

// Fredric Eidsvik

Attention: There are red wires(power) & blue wires(GND) that are soldered together with all the sensor wires so we can easily connect Vcc and GND with two wires into the arduino. Description of the wires on the sensors are description of colour before the soldering connection. With the exception off razor.

Battery > red wire(Vcc), black wire(GND), blue wire(signal), > Voltmeter

Battery > Power Switch > red wire(Vcc), black wire(GND) > ESC

Battery > Power Switch > white wire(Vcc), grey wire(GND) > Arduino

Battery > Power Switch > white wire(Vcc), grey wire(GND) > Volt Regulator > white wire(Vcc), grey wire(GND) > Odriod

Arduino > red wire(GND), orange wire(Vcc), green wire(OUT B), yellow wire(OUT A) > Wheel encoder

Arduino > purple wire(), white wire(), grey wire(), black wire(), > Ultrasonic Front

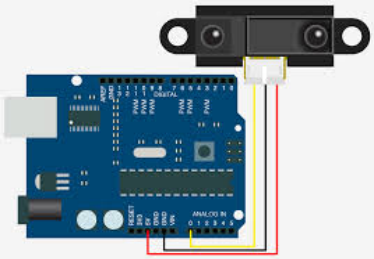
Arduino > purple wire(), white wire(), grey wire(), blue wire(), > Ultrasonic Right Front

Arduino > left wire(Analog in), middle wire(GND), right wire(Vcc) > Infrared (See figure 1.0

Arduino > black wire(GND), red wire(Vcc), brown wire(RXI), orange wire(TXO) > Razor

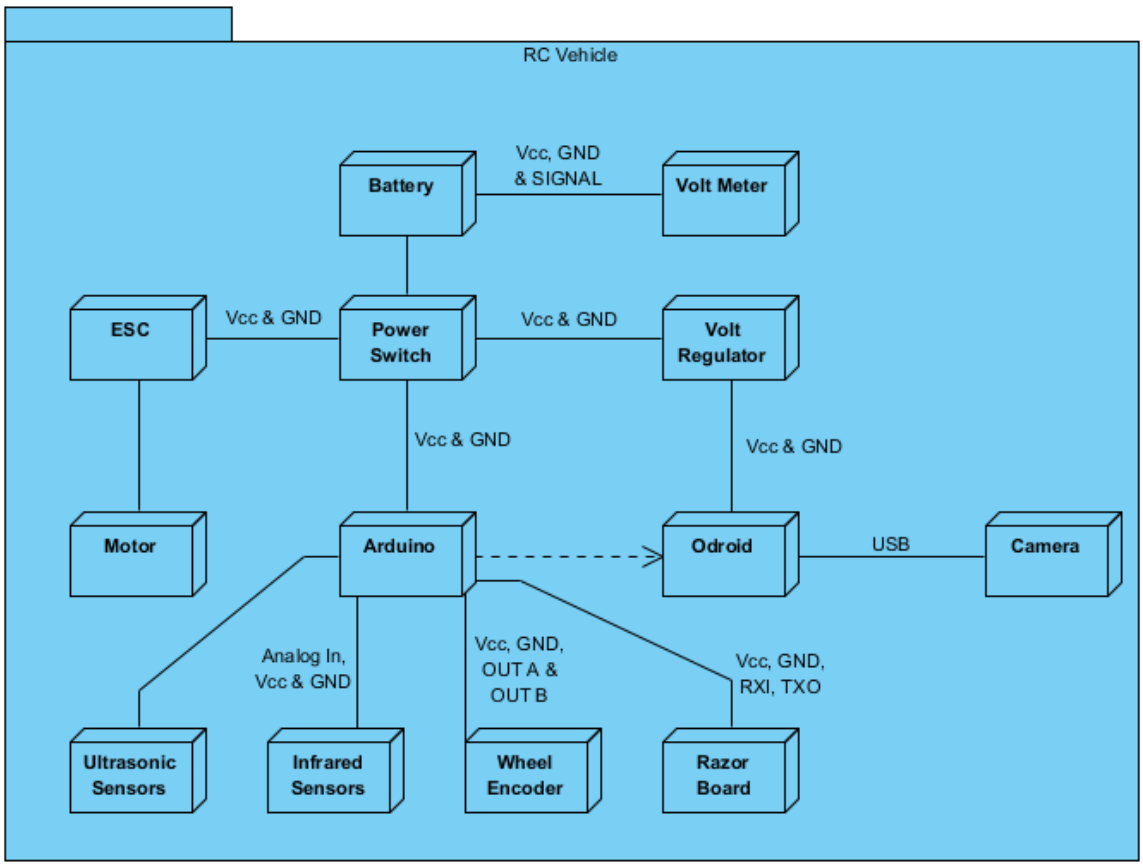
Odriod > USB > Webcam

9. Infrared



// Malcolm Kente

10 Diagram



Manufacturing Process

// Malcolm Kente

The process started with the removal of the default motor, ESC & servo from the car. The new servo was assembled and mounted onto the base of the car with tape adhesive. The new ESC and motor came in one package and we connected their respective wires to each other before mounting them on the car. The ESC was also

mounted with adhesive tape and the motor was mounted with two screws in the back end of car's base. When the car's new parts were assembled, we tested the compatibility & functionality by help of the RC remote.

// Fredric Eidsvik
We then mounted the rubber tires on the wheels. It proved to be quite a hassle because the tires was a bit hard to get on the wheels After this process the car was driveable with a remote control.

The IR sensors were mounted on the side of the body on the car. By saying the body i mean the bottom plate where the wheels, battery, Servo, Esc are mounted on. There is one IR on the back side and two on the right side. We simply used two torx screws and two nuts.
The back one was more complicated because it ended up pointing downwards. We solved that with adding hot glue on the lower side of the IR making it compensate for the crooked frame.

After this we created and framework from a bit of plastic that would later become the surface where Arduino, Odriod and STM would be placed. We received a template from Emma Jarvi, unfortunately it was a faulty template. The Odriod was placed too close to the Arduino to be able to connect the usbs. This had to be fixed and was time consuming.
The main reason for it being very time consuming was that it was impossible to realize that the Odriod had a bad position until we actually mounted everything on the surface. So we had to drill new holes into the plastic piece. We also received an template for the front bumper, this template was correct and only had to be filed down to the proper size.
We also created two small plastic parts, we heated up the plastic so it could be bent. We then placed the plastic with 2 screws and 2 nuts on each part in front of the bumper. This was later declined by the stakeholders so we had to reheat the plastic parts and bend it the other way and place it behind the bumper to meet the customers requirements.
We also created an plastic pillar where we would later mount the webcam and the razor. The pillar had to be double layered so it would be stable and not vibrate so much that the camfeed would be bad. I just bent the two plastic parts at the bottom, screwed it on the surface where the boards go. i glued the pieces together so it would get an even surface stick. Instead of having screws and nuts. That would equal to more points of tension rather than have a surface of tension. We also cut out an extra plastic piece that we will later mount the power switch on

// Malcolm Kente

The US (Ultrasonic) sensors proved also a hassle due to the fact that one of them wasn't operational. After running tests on the computer with connection to the Arduino, we realized that no information was being received from one of the sensors. This slowed down our process of building the as we had to send an email to Federico Giaimo to report and request of a new sensor. Upon receiving a new and functioning US sensor, we set to mount them on to the car. First, we mounted the sensors onto the plastic provided to us with 2 torx screws on each. The plastic mounted with US sensors was then mounted on the front plastic frame of the car with 2 more screws. Initially we had two sensors mounted in a hanging position on the front end of the car. This would later need revision as both the customer's & software team's requirements didn't match with the design. So as to match the software side's criteria, we moved the one sensor from the side to the middle of the front-end of the car. The other sensor was tilted and mounted at a 45° angle on the front right-end of the car. We then matched the customer's requirements by switching the position from hanging to a more upright safe position. This way, if the car would crash then the sensors wouldn't obtain any damage.

After the Odroid and Arduino were mounted on the car, the wheel encoder was placed inside the back left wheel of the car. The initial mounting of the wheel encoder wasn't successful as we never got any readings back after testing. We placed striped bit of dark and white tapes in the wheel and used hot glue to mount the wheel encoder inside the wheel.

// Fredric Eidsvik
We decided to use spacers on the US, Odriod and Arduino so we would not hurt the boards with tension on the plastic surface. The STM was simply glued on the car. This decision was made due to the fact that the STM did not have any screw holes. The screws on the Arduino proved to be impossible to mount because they were too wide on the head. We solved this with a mechanical metal grinder and we simply grinded the head of the screws down.

The razor board was mounted on the back of the pillar, we were advised by Emma to put it there because it would mostly likely disturb the other boards if it was placed any closer to the other boards. The webcam was placed on the very top pillar so we could get an nice angle when we use lane following. The razor was mounted with 2 torx, nuts and spacers. The webcam was mounted with a torx screw and two bricks.

The wiring part was solved by soldering all Vcc wires and all GND wires from all sensors into two wires so we could easily connect them into the Arduino. This meant that we did not need an breadboard with was nice for cable management and

design. The soldering part might have been time consuming but it has payed of in the long run.

When we received the new battery we decided that we wanted an powerswitch for the car. So we had to cut the Vcc wire so we could mount an attachment to the end that was connected to the power switch input. We created another red wire for the output on the power switch this would later go into the connection on the ESC. On the red wire we soldered two wires. On the ESC connection we also soldered two wires on, this were the GND.
One GND and one Vcc went into the volt regulator input and then two wires went out on output that we soldered on. The two wires from output was soldered on to an powerplug that would fit the Odriod. The other two wires were directly soldered to an powerplug that would fit the Arduino.

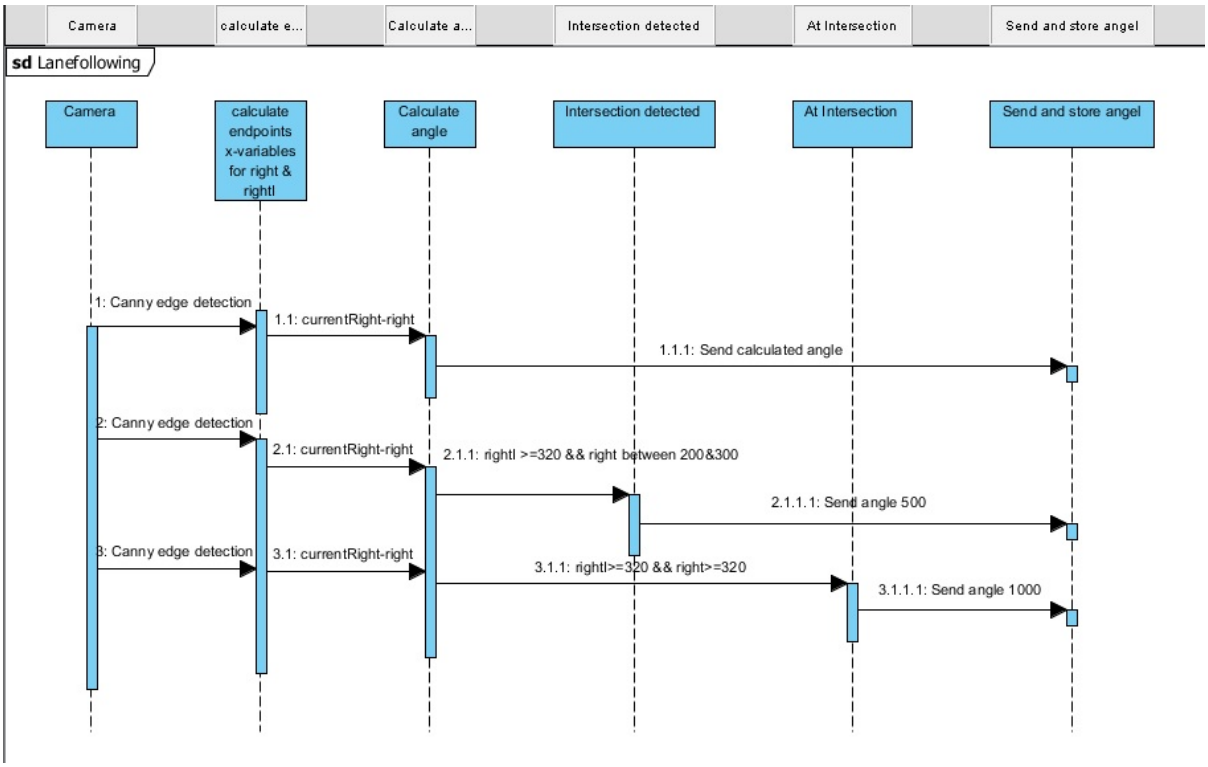
The LED lights were mounted on the inside on the chassi. The board was taped on with double sided tape. We screwed several holes into the chassi so we could push through the lights they all fitted perfectly except the blinkers which we had to glue. We had to cut hole in the chassi for the pillar, we also had to cut on in the front and on the right side on the chassi so we could use the sensors without any disturbance. We cut a hole for the power switch so we could turn the car on and off with the chassi on.

The last thing we did was lifting the suspension so the car, this was to due the new battery and all the hardware on the car made the car really low. On the track we had some tape that were like speedbumps. Our car could not pass this bumps because it was too low, after we lifted the car it ran more smoothly.

Software Architecture:

//Sara Johansson

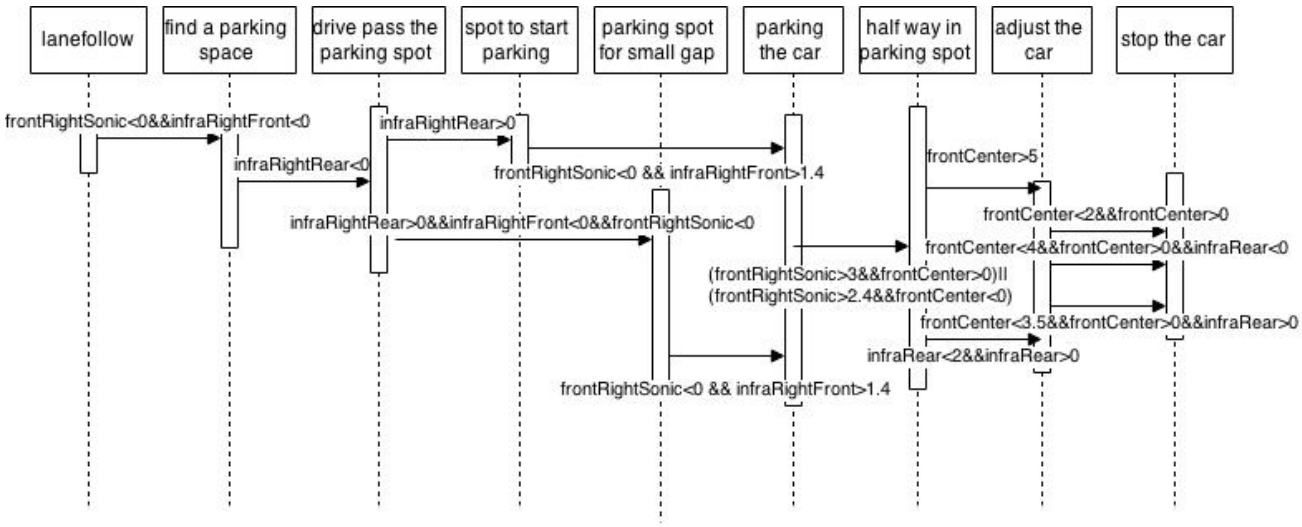
11. Diagram of Lanefollowing



There are three different scenarios that the lanefollowing can go through, the first is the normal one where the angle is calculated and sent, the second is when the lanefollowing detects an intersection and sends an angle of 500, the last one is where the lanefollowing recognises that it is at an intersection and sends an angle of 1000.

//Mengjiao Wei

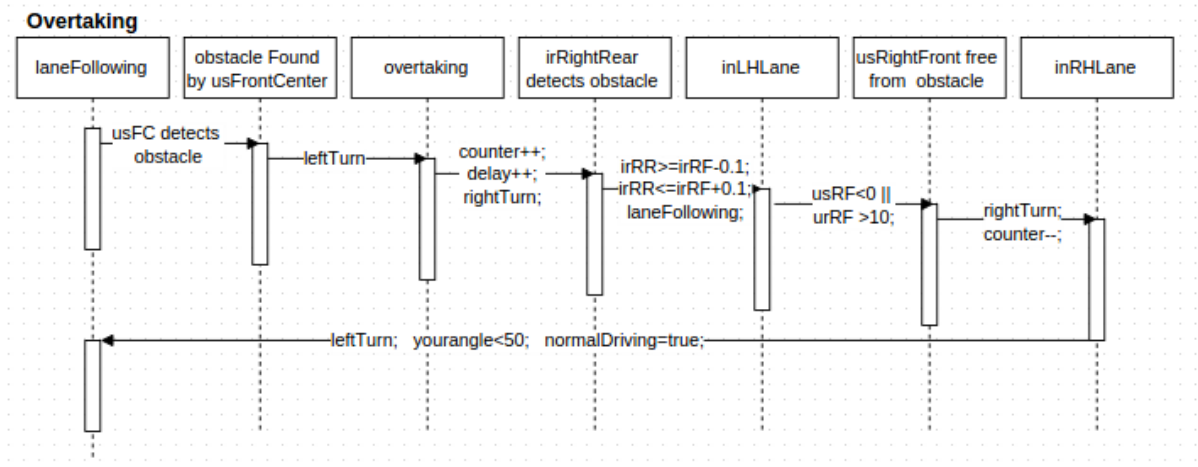
12. Diagram for parking



For parking, we have three scenarios. We use almost the same logic for parking-easy and parking-medium, but we change a little bit for parking-hard. As you can see from above diagram, the vehicle use the lane follow to move until it detect a parking space. The change is before the vehicle start parking. In the parking-hard, the car will drive a little bit more than in parking-easy and parking-medium, and then start parking. We divide the parking the car part into 3 steps, parking the car, halfway in parking spot and adjust the car.

//Flutra Tahiaj

13. Sequence Diagram for Overtaking

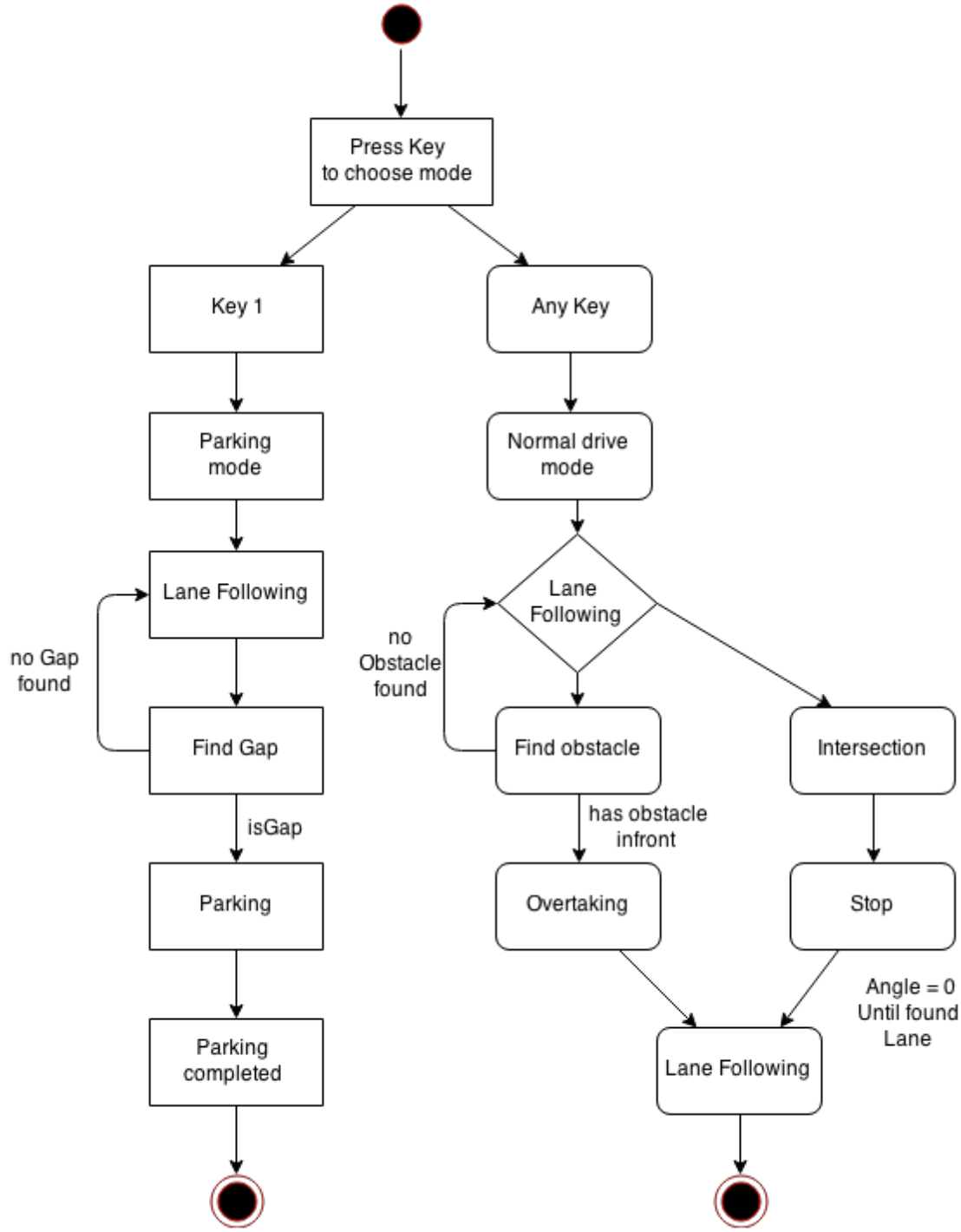


This diagram shows a very basic overview of the overtaking maneuver, this provides other users to be able to understand an unfamiliar code. Overtaking gets triggered by the ultrasonicFrontCenter and gets into overtaking state. What's not showing in this diagram is that the usFC can be triggered in two different occasions, depending on the wheel angle that the vehicle has, the distance from the usFC to the obstacle is different, exactly why and more detailed have previously been explained, in "Implementation details of source code". When the obstacle has been detected by

the ultrasonicFrontCenter the vehicle will turn left sharply until the infraredRighrRear detects the obstacle, which then straighten up the vehicle and when the two infrared sensors on the right side are getting the same values means that the obstacle is in parallel to the vehicle, and lane detection will correct the angels so that lane can be followed again. When the ultrasonicFrontRight is free from obstacles i

//Peili Ge
Software Architecture for overall project - parking, lane following, overtaking and intersection:
Diagram: Statecharts
There are two modes we can chose to drive: Parking mode and Normal driving mode.
Before you start driving, you press key in terminal to chose which model you want to enter. Key 1 for parking, and any key else for normal driving.
While in Parking mode: you start by driving with lane following, the sensor is always finding the gap, when you find the suitable gap for parking, then you start the parking. By the parking method written above After parking is done, you stop the car and finished driving. When you can’t find the gap, you continue driving with lane following.
While in the Normal driving mode: you start by driving with lane following, when your sensor find there is any obstacle in the front, you start doing overtaking. After finished overtaking, you follow back to lane following until the end you stop driving. If you can’t find the obstacle in the front, you continue with lane following. Or there is one option you will meet intersection, the car will stop at intersection (before the stopping line). After intersection, the angle will be 0, therefore it will drive straight for a while, until it finds lane again and continue with lane following, until the end you stop driving.

14. Diagram of architecture



Test results:

//Markus Erlach

Hardware Testing:

When we started assembling the car and replacing all the old parts such as the ESC, Servo and battery, we needed to figure out how we wanted to place all the new parts as well as test all the new parts in order to have fully functioning parts when fully assembled. The new parts were things such as Ultrasonic sensors and Infrared sensors.
These sensors we needed to place in a way that the software group could use them in the way that their parts managed them in simulation. This meant having the sensors at certain locations or angles in relation to the surroundings or the car.

For each sensor or new part we got, we tested it before assembling it, when it was assembled as well as together with more of the same kind. We also made sure to test each part whenever a new part was added to make sure that everything still worked.
These tests consisted of individual test codes for each part in order to make sure the data read was correct. Minor problems arose when testing sensors. Problems such as sensor addresses became minor challenges as well as problems trying to read wheel encoder data without using some kind of tape inside the wheel to give the encoder something to read off of.

When testing the proxy with the hardware, we began quite simple only trying to send single chars or short char arrays to make sure what we sent was received on each side correctly as well being done quickly without something slowing it down. Once we had a simple read and write done in the HLB proxy as well as the LLB arduino file, we began slowly including minor parts such as one US sensor or one IR sensor and adding that to the array being sent to make sure each new thing worked. Once we had added the speed and angle on the HLB side, we needed a checksum so that we could always compare our data to make sure the correct values were received. This checksum is quite simple and just adds up what we send and puts it at the beginning of the char array we send so we know where to look for it. We also use padding here.
The HLB char array structure:
Structure: CheckSumSpeedAngle Example: 02808020
This is checked on LLB side when received before continuing.

We then did the same thing for the LLB, but here we also added the length, a start char as well as an end char since it was a bit trickier to read in what we needed to use. This made our array sent from the LLB have the structure:
Structure: <LEN:CSUM:IR1IR2IR3US1US2> Example: <16:100:2020202020>
We added the start char < and the end char > so that we could know where it began and ended in order to make it easier to check the array we have received.

//Simon Lobo Roca

Integration testing

We started with sending hardcoded values from the HLB to the LLB. The LLB would use serialEvent() to read what the HLB sends. The Proxy would send {4,20,90}, the LLB would store each character in arrays and check for start char = “{“ and end char = “}”. The values in between would be divided commas so we know what value belongs to what. The first value was a check so we would see if it was a digit the following two values were meant to be speed values and the final values were steering angles. As we continued with the testing we figured out that the number we send could always be the same amount

//Flutra Tahiraj

Software Testing and Integration

The driver file that was provided to us were copied and given to each developer with the responsibility for software functionality of the vehicle. Each one tested their own code and made sure it worked before it was integrated together with all code in driver.cpp. It was easiest to handle the integration with the simple menu in the terminal, which would make the vehicle park or drive normally. The challenging part was to make sure that lane following and intersection was working well. Which we many times had to modify to make it work with both parking and overtaking.

Eventually when the integration in driver was done, what was left to do was to change the scenarios and run the program again, and again. Continuously until it was succeeded in every scenario. Then the conclusion would be that the integration would be fine, and each developer could continue to modify their code, with the small risk of breaking the integration in Driver.

References:

For lanefollowing: http://docs.opencv.org/
For proxy: http://en.wikibooks.org/wiki/Serial_Programming/termios
For overtaking: C.Berger “From a Competition for Self-Driving Miniature Cars to a Standardized Experimental Platform: Concept, Models, Architecture, and Evaluation”, published Jun 2014. [Online]: Available: http://http://arxiv.org/abs/1406.7768

Retrospective:

//mengjiao

Software: The problem for lane following is we did not consider we need to release the images so it didn’t draw the line when we were using the real camera. Before we tested the lane following on real car we created some backup plan, but actually lane following works really well on the real car. When we doing the parking and overtaking, it’s taking a lot of time to figure out one algorithm works for all the scenarios. We didn’t have enough communication with hardware group, so we had to change the algorithm of parking. And also we spent time work on make lane following and intersection work when the vehicle is doing overtaking. To integrate lane following, intersection, overtaking and parking together we with the simple menu.

//Markus Erlach

Proxy:

Our first solution after talking to supervisors was to begin with using google protocol buffers between the HLB and the LLB. This seemed like a great idea at first because gpb is a very fast and easy to use serialization tool. The problem we ran into here was that gpb is not yet compatible with using arduino from what we have understood. And implementing it would require a lot of tweaking and fixing to make it work the way we wanted so therefor we chose to give up on using gpb and instead focused on implementing it another way.

Another minor issue we had was lack of communication and that is why we ended up having two different encode/decode parts written by two different people. Markus needed to make his when he was working on the proxy because PeiLi had not started yet and so we ended up having two very similar code snippets.

//mengjiao

Hardware:

Our hardware group worked together with other groups and supervisors, so when we met problems we got some suggestions from people with experience which make our work easier. And also we distribute our work very well, so everyone worked on their work. We also had some problems that made our efficiency slow. Some part of hardware was broken so we have to wait until we got a new one which wasted our time. We did not get enough Ultra Sonic sensors. We didn't know discovery board should put on the car when we put everything together, so we failed the first hardware check. Plastic template which we got is bad. We did the wrong thing in the beginning, so we had to redo it.

// Peili Ge

Overall Retrospective:

We have strict meeting time each Monday at 10:00am, and additional meetings several times per week. We have good communications within the group if anyone cannot make the meeting in time.

We made mostly all the milestones' deadlines in time, we tried as much as as we could by helping each others, and handin works as much as we have finished.

We also learned from other groups' feedback to prove ourselves next time.

Our Lane Following works really well by both testing on the real car and in simulator, it follows the lane when turning and or following the straight line.

Our Parking works very well in the simulator for all scenarios.

Our Overtaking works very well in the simulator and past the medium tests.

We received data successfully from all the sensors and encode them and send to HLB, and from HLB we can send the stored data to LLB, so the communication between LLB and HLB works well.

We have done much testings and data recordings which all went really well.

We did manage the hardware really well by providing the final product, and the connections with proxy. It takes abit longer time than we expected on proxy side by making different functions of read data and write data for transfer connections. Successfully, the proxy for both arduino and odriod sides works well with transfer datas to each other.

We had several problems within the processes of making the hardware part as well as connect hardware with software to make the car running. We got our hardware part down a bit late then we expected. As well as the misunderstanding between hardware group and software group, it's very time-consuming in the whole project process.

For hardware part, we only had limited materials, since other groups already took all the wires before us, we could not manage the car really well. We could not use red wires for Vcc, and black wires for GND, as well as the signal wires has mixed colors, therefore the hardware work was delayed. There were also limited amount of spacers, screws and nuts. We also did not allocate the workload very well, this was the main reason that we had much time-consuming of planning and divide works up, we just find what need to do step by step, we did not get a knowledge of the overall plan about what we need to do and how to divide the whole project. We get too less knowledge of this project in the beginning and in the processes. That's also the reason we didn't update informations and communications in connect with both hardware and software groups.

Appendix (Test protocols and charts):