E29 Final Project: Smart Car

Group 9
Youssef Kharrat and Christian Bignotti

Introduction and Motivation:

Smart cars are vehicles that provide safety, convenience and efficiency for its users by utilizing advanced technology to maneuver in the never resting real world. Inspired by what seems to be a surreal idea of vehicles driving themselves, we decided we would try and mimic this at a smaller scale. With the intention of better understanding the software and the embedded systems behind a smart vehicle,we plan to build a RC car that can go from one location to another while also avoiding obstacles along its way.

Apart from smart cars we were also inspired by autonomous devices that move around and interact with the real world. From autonomous vacuum cleaners to drones that deliver packages, there are currently companies using smart vehicles for several different applications. One example we found was the company ZipLine that delivers medical supplies to rural areas in countries like Rwanda, where hospitals in small towns are hard to reach in time, by using autonomous RC airplanes [1].

Design Requirements:

As stated, our project consists of creating a smart car. This smart car should be able to take in a single action from a user through a keypad. Actions will span from go forward or backwards a certain distance, to turn right or left a certain amount of degrees. After confirming the input the smart car should perform the action. In the case it comes across an obstacle, it should be able to slow down as it approaches the obstacle, and eventually come to a stop, to then resume its course whenever the obstacle moves off of the car's path. The car should stay in its place if it approaches an obstacle that does not move (i.e. a wall, a parked car, etc). The car will only be able to detect obstacles it is approaching from the front or the back, considering it will only have distance sensors on the front and back of it.

With this design in mind, our system will have two main states. The first one would be awaiting a user's keypad input. The user has to first choose an action through one of the 4 letters : Forward, Backward, Turn right, Turn left ('A', 'B', 'C', 'D' respectively) and then give a number followed by '#'. The number will be interpreted as distance in cm for the first two options, and angle in degrees for the other two. The user will know he is being prompted by a flashing LED. In case the user fails to adhere to these constraints, he will get prompted again.

Once the input is successfully accepted, the system goes to the second state: the drive mode. The system now will drive in the direction and distance that was prompted. In order for this to work, our car is also supposed to locate itself based on the distance it has traveled using its odometry system.

In case an unexpected obstacle appears on course, we expect our car to stop before collision. Considering we will be using ultrasonic, obstacles that are not perpendicular or parallel to the vehicle's path will not be detected.

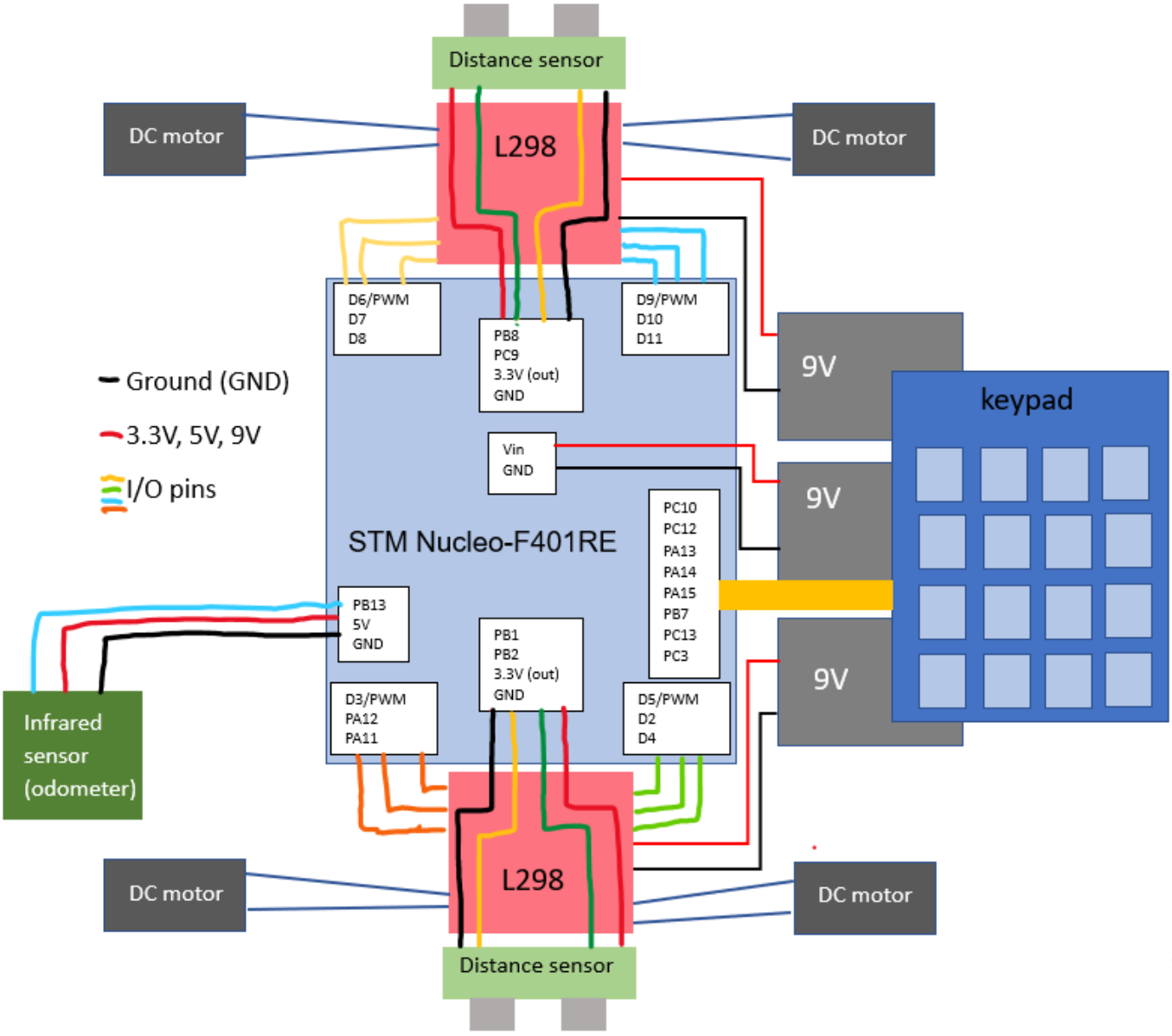
This design comes with the assumption that we will be able to actively update the distance traveled and angle turned. This will be achieved through a feedback loop that continuously asks our odometry system for the actual distance our car has traveled in the given direction. Our car is either in a state of turning or a state of rolling (forward or backwards). The amount of turning that has been achieved will also be calculated with the same feedback loop.

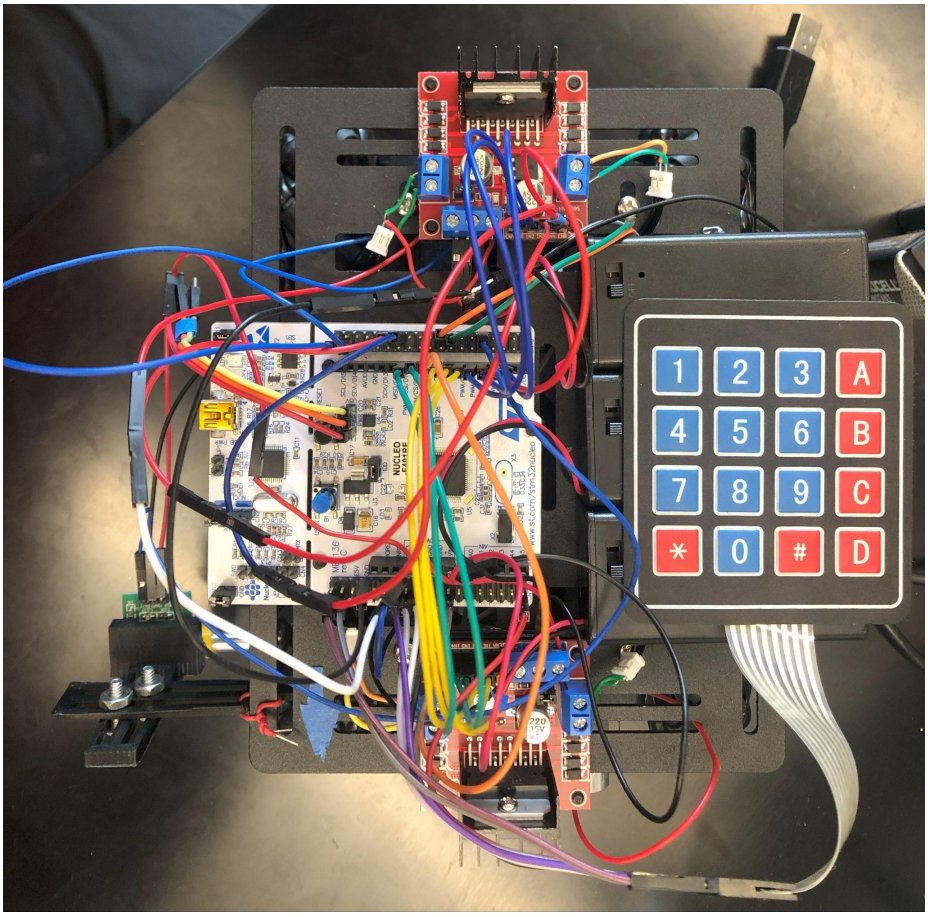
Final Product:

Hardware:

- STM Nucleo-F401RE microcontroller
- Two L298 DC motor drivers
- Lewan Soul Mecanum Chassis car kit which includes four DC motors
- Two ultrasonic distance sensors
- One notch infrared sensor with encoder wheel
- Three 9V batteries
- Three 9V battery holders with ON/OFF switches
- A 4x4 keypad (numbers 0-9, letters A-D, # and *)

Block Diagram:

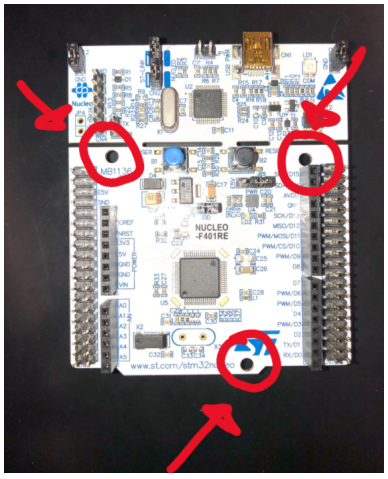


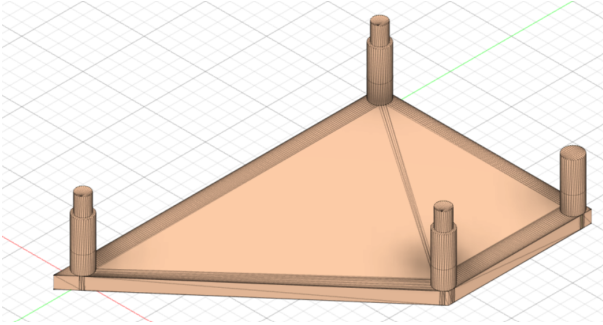


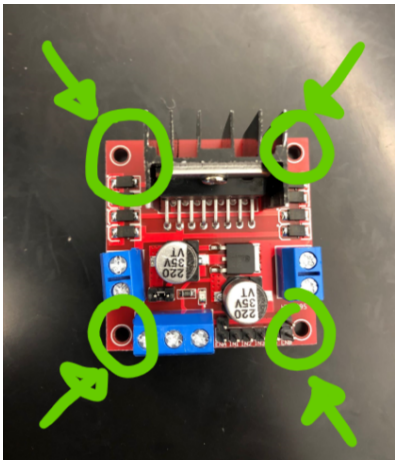
Assembly:

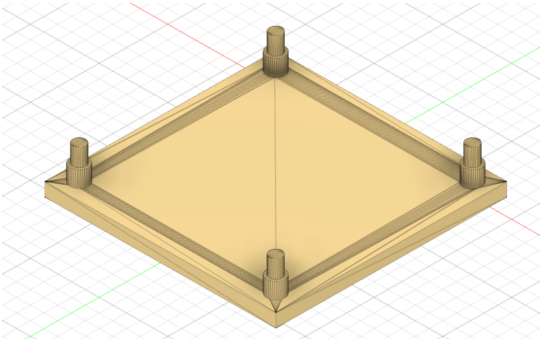
One of the big obstacles we came across was mounting all of our hardware onto the car. We needed everything to be on the car in an organized fashion, and firm enough to not fall if the car stopped or sped up quickly.

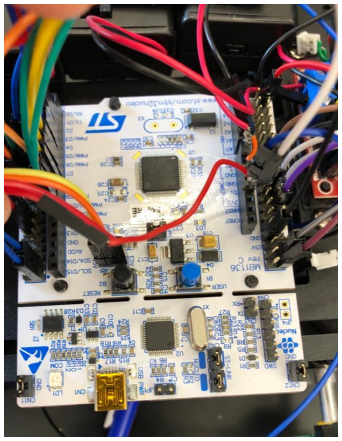
On top of the car we needed to put our microcontroller, for easy access, and both motor drivers. These boards (both the Nucleo board and the motor drivers) have holes specifically for mounting. So after some quick measuring and prototyping we were able to 3D print mounts for them:

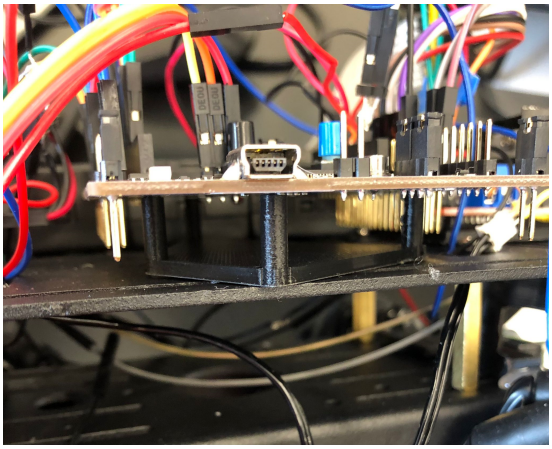


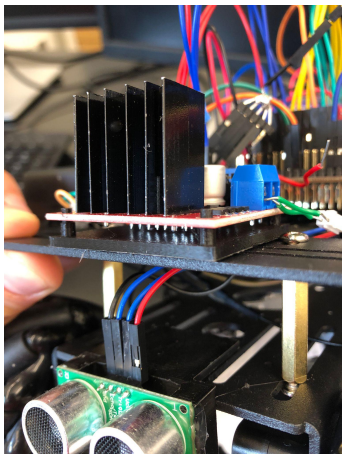






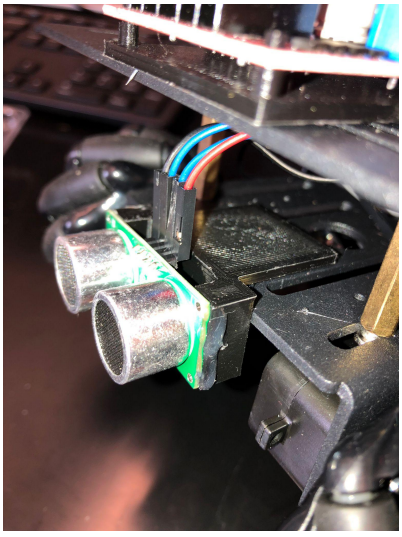


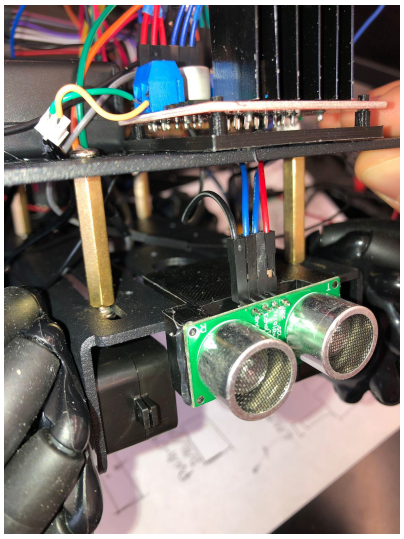


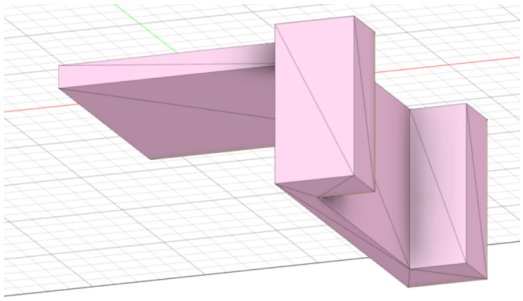


The mounts were then hot glued onto the car. The boards can be easily removed, but are sturdy enough to stay in place when the car is in action.

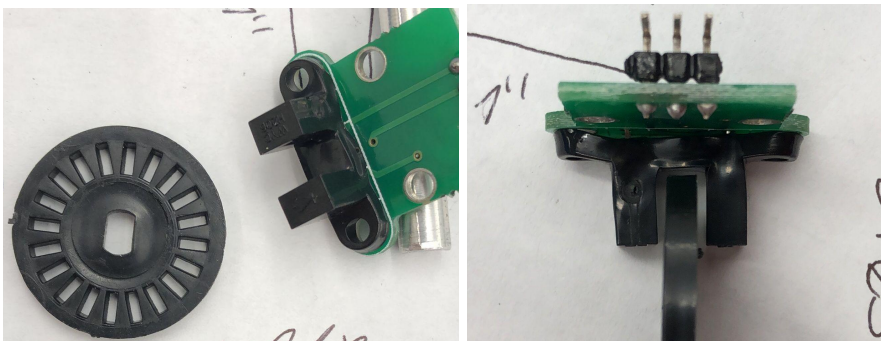
For the distance sensor we took a similar approach. We measured the areas we could use to mount it, and after some prototyping we were able to secure our sensors to the car. The sensors were glued to the mounts, and the mounts glued to the car chassis..



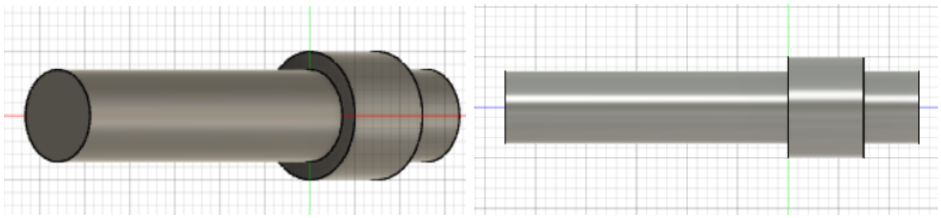




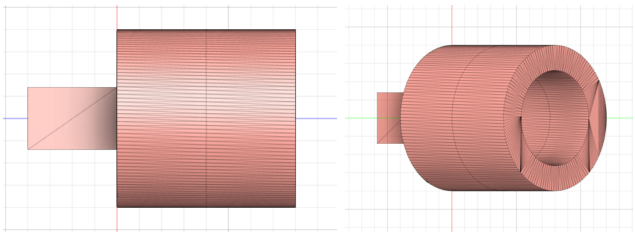
Lastly was our odometer. For this we needed to mount an encoder wheel to one of the car wheels. The encoder had to be far enough from the wheels so we could fit a notched infrared sensor over the encoder wheel, to measure revolutions of the car wheel over time. The encoder wheel is just a small wheel with holes distributed symmetrically around its circumference, so that the notched infrared sensor, which you can imagine to be like field goal posts, can count how many times it sees a hole go through the field goal posts.



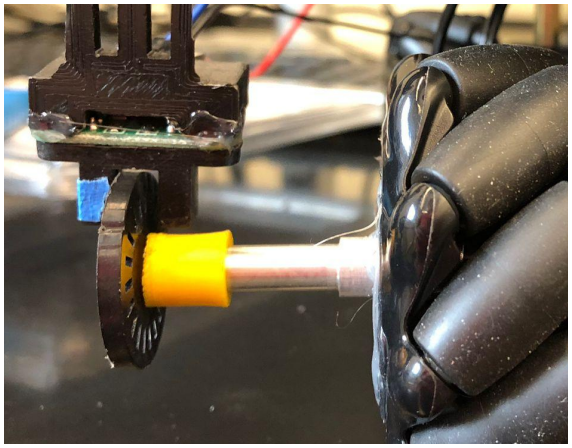
To mount the enconder wheel on the car wheel far enough to where we could fit the notched infrered sensor we made a sahft. We made the shaft out of aluminum with the help of J. Johnson, from the engineering machine shop. Using a lathe we were able to get the exact diameters we needed.

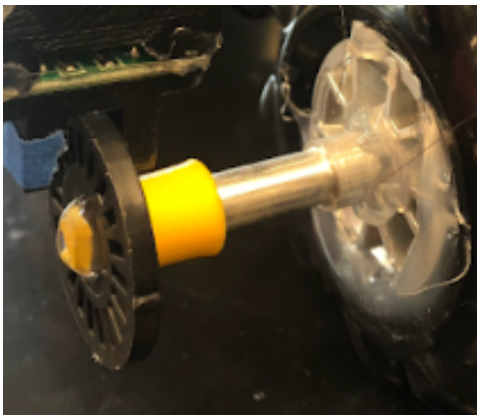


One side of the shaft was attached to the car wheel. The shaft had a very tight on the car wheel, and was secured with glue. On the other side we 3D printed a hub that we could attach to the encoder wheel. The hub also had a tight fit on the shaft, and was later secured with glue.

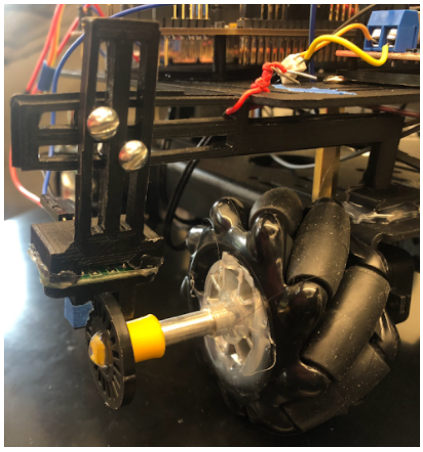


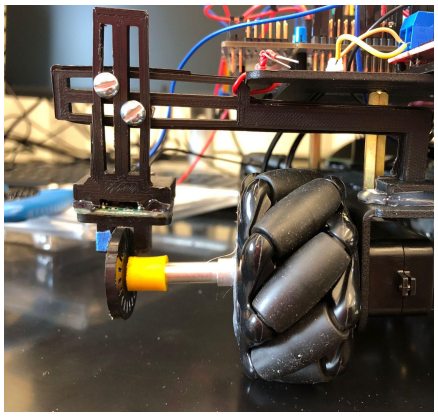
All three parts assembled (shaft on car wheel, hub on shaft, enconder wheel on hub):

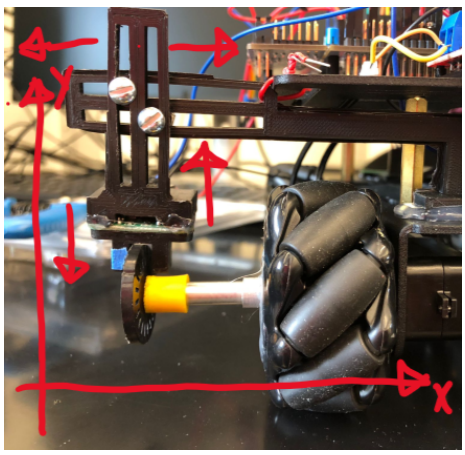


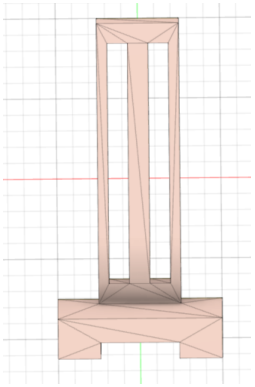


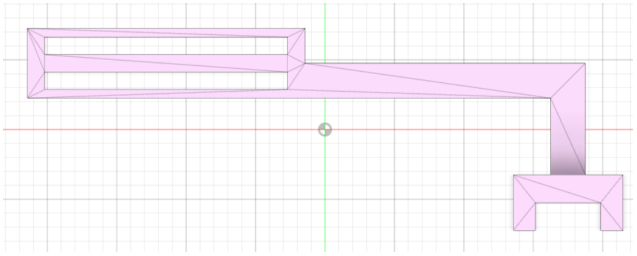
Considering we had to get the sensor right above the encoder wheel, we 3D printed two long slotted mounts. The slots were there so we could easily adjust the x and y positions of the sensor over the encoder. One of the mounts was attached to the car frame, whilst the other had the infrared sensor attached to the bottom of it.







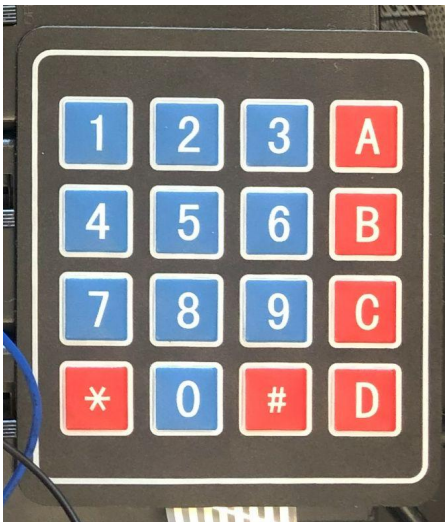




The key pad and three batteries were simply glued to the the top of the the car’s chassis.

Software:

User manual:



Key pad (takes the input)

The user has to first choose an action through one of the 4 letters : Forward, Backward, Turn right, Turn left (‘A’, ‘B’, ‘C’, ‘D’ respectively).

A - Forwards

B - Backwards

C - Turn Right

D - Turn Left

The user then must enter a number which will be interpreted as distance in cm or angle in degrees, depending on what action was choosen. The user will know he is being prompted by a flashing LED. In case the user fails to adhere to these constraints, he will get prompted again. After entering a number, the user must press the pound sign (#) to confirm the input, and the smart car will begin its action. The sensors are only active when moving forward or backwards.

Software:

One of the intended goals when working on this project was to expose ourselves to some new software design, such as object oriented programming (OOP). Firstly we defined a SmartCar object. This object encapsulates the speed, the gear and all the possible movements that the car is capable of. As we can see in the .h file, we made the speed and gear private variables since we do not want the user to change those values outside of the intended use. By defining all the pins and functions in a separate SmartCar.cpp file, our code gained a lot of readability.

```
#ifndef SMARTCAR_H
#define SMARTCAR_H

class SmartCar {
private:
  int speed_;
  // 0 = park; 1 = forward; -1 = backward; 2 = right; 3 = left
  int gear_;


public:
  SmartCar();
  void goForward(float speed);
  void goBackward(float speed);
  void goDiagonalRight(float speed);
  void goDiagonalLeft(float speed);
  void turnRight(float speed);
  void turnLeft(float speed);
  void slideLeft(float speed);
  void slideRight(float speed);
  float smoothAcc(float desired_vel);
  void stop();
  int mySpeed();
  int myGear();
};

#endif // SMARTCAR_H
```

The second object that we created was the keypad. We were able to reuse the code that we developed back in lab 1. In fact, this again shows the power of OOP where using this whole new functionality only required us to add an “include” line at the top of our program and adjust the right pins. This keypad allowed us to interact with the system outside of the terminal and give it a finished product feeling.

Our main file also included some definitions at the top, in order to give some sense to the seemingly meaningless numbers. In fact our gear system was defined through these integers.

```
#define WAIT_TIME_MS 500

#define Forward 1

#define Backward -1

#define Park 0

#define Right 2

#define Left 3
```

Our implementation of the odometry system is as follows. We check for every positive edge of the IR sensor connected to our wheel by comparing it to its previous value. We then increment a global counter variable. This happens every 5ms since that sensor is fairly sensitive and we were getting better results with these very frequent ISRs.

```
sampleCounter.attach(&updateCounter,5ms);

void updateCounter(){

    if ((IRsensor.read() == 0) && (preV == 0 )){

        counter ++;

        preV = 1;

    } else if ((IRsensor.read()!= 0) && (preV > 0 )){

        preV = 0 ;

    }

}
```

Our odometry worked by translating the number of revolutions that our wheel has made into a distance through the getDistanceTraveled function, where wheelRadius is the measured radius of our wheel. Depending on the kind of movement the car is making we update our x and y coordinates accordingly, based on which we update our variable d. This variable refers to the distance traveled. If our gear is in turning state then, we instead measure the new angle Theta where the variable turnRadius refers to the radius from the center of the car to the wheel. This update ISR only triggers every 50ms since the car does not go at very high speeds we do not need to update our location that often.

```
sampleOdometry.attach(&updateLocation,50ms);

float getDistanceTraveled(){

    int localCount = counter;

    counter = 0;

    return wheelRadius * 2 * pi * localCount / 20;

}

void updateLocation(){

    float distance = getDistanceTraveled();

    if (Car.myGear() == Forward) {

        x = x + distance*cos(theta);

        y = y + distance *sin(theta);

        d = sqrt(x*x + y*y);
```

```
    } else if (Car.myGear() == Backward) {
        x = x + distance*cos(theta + pi);
        y = y + distance *sin(theta + pi);
        d = sqrt(x*x + y*y);
    } else if (Car.myGear() == Right) {
        theta = theta - distance/turnRadius;
        if (theta<(-2*pi)){
            theta = theta  + 2*pi;
        }
    } else if (Car.myGear() == Left) {
        theta = theta + distance/turnRadius;
        if (theta>(2*pi)){
            theta = theta - 2*pi;
        }
    }
}
```

We also use an ISR in order to update the distances of our two ultrasonic sensors. This follows pretty closely to what we have previously done in Lab3.

```
    sampleDistance.attach(&distanceADC,50ms);
```

Based on the distance returned by these sensors we control the speed of our SmartCar. The farther things are from us the faster we go. Once we get too close, the car goes to full stop. The threshold values used were experimentally determined.

```
            if (distanceFront < 20.0) {
                speed = GearPark;
            } else if (distanceFront < 30.0 ) {
                speed = SpeedLow;
            } else if (distanceFront < 40.0) {
                speed = SpeedMid;
```

```
        } else {
            speed = SpeedHigh;
        }
```

Finally we implemented a state machine inside our main function that is controlled by two booleans : inRead and inAction. inRead is a blocking state where we keep on reading until we satisfy the input conditions after which we set inAction to true. However, since the updating speed snippet of code happens inside the main loop, we had to make sure that the inAction state is non blocking. Inside inAction, we set the right direction and speed of movement of our car depending on the input. Once we have traveled the desired distance inAction sets itself to false and goes back into input mode by setting inRead to true. It also re-initializes the distance traveled and the angles turned.

Videos:

Forward 20cm (A-2-0-#):

https://drive.google.com/file/d/1gTH4nPG5DhgyTuxu737xgApVZxC_sr7L/view?usp=share_link

Forward 120cm (A-1-2-0-#):

https://drive.google.com/file/d/1eVl1YT2_hhJukwv6H2Xle3w0JBKjZZOM/view?usp=share_link

Backwards 20cm (B-2-0-#):

https://drive.google.com/file/d/1witBOkS081JWxYf3eu7gq-b6QbE40LhA/view?usp=share_link

Turn Right 45° (C-4-5-#):

https://drive.google.com/file/d/1Y6ew410cZ4uGcjD01xGg33EbXxtWsVIs/view?usp=share_link

Turn Right 90° (C-9-0-#):

https://drive.google.com/file/d/1k820gCzLhfNHZdclf7TJ2tN1gmaaFVmG/view?usp=share_link

Turn Left 180° (D-1-8-0-#):

https://drive.google.com/file/d/1dy4PrTCep1t7byLN7GehQ0GEsFmV7acm/view?usp=share_link

Approaching an object:

https://drive.google.com/file/d/1lP9fNHlJag8m5OYkRhwNFpdhyy6I_I7M/view?usp=share_link

Turning Right 90° (C-9-0-#), forward 120cm (A-1-2-0-#), approaching an object that then moves:

https://drive.google.com/file/d/1mL9M2eAZFfR3-sDDf3H17ZPuqdfl1Ls3/view?usp=share_link

Reflection:

In the end we are very pleased with our final product. Even though we are councious of the things we can add to the smart car, such as a coordinate system, re-routing when approaching obstacles, and improving accuracy, we think we used our time well. When building and coding we were very particular about slowly adding new hardware and new blocks of code. For example, we started by making one motor rotate clockwise, then counter clockwise, then controlled its speed, then went on to all four motors. We followed this “baby steps” approach the entire process. This proved to be very successful considering we had a lot of small parts in the entire system. We were able to constantly check any progress, and easily debug both hardware and software if needed.

One of the biggest challenges we had was powering the entire system. At first we thought two nine volt batteries would be enough. One battery for each motor driver, and then we power the Nucleo board with a five volt output pin from one of the motor drivers. This worked for a large part of the building process, but after adding sensors, it seemed as if it was not enough. The biggest challenge was pinpointing the problem. Our motors would rotate slower sometimes but not all the time. Considering we were adding code and other hardware we were not sure what was the problem. After some testing, we were able to recognize it was a power issue, and with the Nucleo’s documentation were able to figure out how to power the Nucleo board with its own nine volt battery.

We also had some problems with I/O pins. Because we needed to use so many more pins than we ever needed for previous labs, we depended on I/O pins we found in the documentation. Documentation was easy to read but we made a big mistake in not inquering to know if some of the pins were connected. For example, for the distance sensor and one of the motors we used two different I/O pins that we did not know were connected. So our wheel would rotate in spurts, and the distance sensor had very inconsistent readings. Again, it was hard to pinpoint the issue, we thoguh it may be a power issue again, but with help from Professor Delano we were able to figure it out.

Future improvements:

Even though we are satisfied with our final product, we also know of the potential we have to improve the “Smart Car”. One idea that we had before starting to build and code, was to have the car have a coordinate system. With this we could tell it to go to a point (i.e. x=100 and y=100) for example. It would then start going towards that point. If it reached any obstacles it could reroute, keeping track of how far it has gone in the x direction and y direction, whilst also keeping track of how far it is from the origin and how many degrees it has turned from its initial orientation. It is very ambitious in terms of software, but the hardware we need is all there. Additionally we would have wanted to add sensors on both sides, not only the front and the back, so if we wanted to reroute we would know if we could turn right or left before turning. With these additions the car would be able to go through a maze for example, constantly rerouting until reaching a point.

References:
[1] https://www.flyzipline.com/

Appendix:

Appendix I:

main.cpp:

```
/* mbed Microcontroller Library
 * Copyright (c) 2019 ARM Limited
 * SPDX-License-Identifier: Apache-2.0
 */

#include "mbed.h"
#include "SmartCar.h"
#include <cstdio>
#include <type_traits>
#include "calculator.h"

#define WAIT_TIME_MS 500
#define Forward 1
#define Backward -1
#define Park 0
#define Right 2
#define Left 3

DigitalOut led1(LED1);
DigitalIn IRsensor(PB_13);

Ticker sampleDistance;
Ticker sampleOdometry;
Ticker sampleCounter;

// example setup for echo and trigger using D2 and D3
DigitalIn echo_back(PC_9);
DigitalOut trigger_back(PB_8);
```

```
// example setup for echo and trigger using D2 and D3
DigitalIn echo_front(PB_1);
DigitalOut trigger_front(PB_2);

//Define the threads
// Thread t1;
// Create a BufferedSerial object with a default baud rate
static BufferedSerial pc(USBTX, USBRX);

//global variables to determine the distance to certain ojects
volatile float distanceFront;
volatile float distanceBack;
volatile float speed;

//different possible speeds
const float SpeedHigh = 1.0f;
const float SpeedMid = 0.5f;
const float SpeedLow = 0.1f;
const float GearPark = 0.0f;

const float pi = 3.1415926;

const int wheelRadius = 3; // in cm
const int turnRadius = 11; // in cm


//counter used for measuring distance
volatile int counter = 0;
volatile bool preV = 0;
//location on the map in terms of distane and angle
volatile float d = 0;
volatile float theta = 0;
volatile float x = 0;
volatile float y = 0;

// timer for echo code
Timer echoDuration;
using namespace std::chrono; // namespace for timers
//constructor for our car
SmartCar Car = SmartCar();
```

```
float getDistanceTraveled(){
   int localCount = counter;
   counter = 0;
   return wheelRadius * 2 * pi * localCount / 20;
}

void updateLocation(){
   float distance = getDistanceTraveled();

   if (Car.myGear() == 1) {
      x = x + distance*cos(theta);
      y = y + distance *sin(theta);
      d = sqrt(x*x + y*y);
      // if ((x == 0)&&(y==0)){
      //     theta = 0;
      // }else {
      //     theta = atan(y/x);
      // }

   } else if (Car.myGear() == Backward) {
      x = x + distance*cos(theta + pi);
      y = y + distance *sin(theta + pi);
      d = sqrt(x*x + y*y);
      // if ((x == 0)&&(y==0)){
      //     theta = 0;
      // }else {
      //     theta = atan(y/x);
      // }
   } else if (Car.myGear() == Right) {
      theta = theta - distance/turnRadius;
      if (theta<(-2*pi)){
         theta = theta  + 2*pi;
      }

   } else if (Car.myGear() == Left) {
      theta = theta + distance/turnRadius;
      if (theta>(2*pi)){
         theta = theta - 2*pi;
      }
   }
}

void updateCounter(){
   if ((IRsensor.read() == 0) && (preV == 0 )){
```

```
        counter ++;
        preV = 1;
    } else if ((IRsensor.read()!= 0) && (preV > 0 )){
        preV = 0 ;
    }

}


float getInputFront(){
    // trigger pulse
    trigger_front = 1;
    wait_us(10);
    trigger_front = 0;

    while (echo_front != 1); // wait for echo to go high
    echoDuration.start(); // start timer
    while (echo_front == 1);
    echoDuration.stop(); // stop timer

     float distance =  ((float)duration_cast<microseconds>(echoDuration.elapsed_time()).count()/58.0);
    // print distance in cm (switch to divison by 148 for inches)
    echoDuration.reset(); // need to reset timer (doesn't happen automatically)
    return distance;

}

float getInputBack(){

    // trigger pulse
    trigger_back = 1;
    wait_us(10);
    trigger_back = 0;

    while (echo_back != 1); // wait for echo to go high
    echoDuration.start(); // start timer
    while (echo_back == 1);
    echoDuration.stop(); // stop timer

    float distance =  ((float)duration_cast<microseconds>(echoDuration.elapsed_time()).count()/58.0);
    // print distance in cm (switch to divison by 148 for inches)
    echoDuration.reset(); // need to reset timer (doesn't happen automatically)
    return distance;
```

```
}

void distanceADC(){
   distanceFront = getInputFront();
   distanceBack = getInputBack();
}


//Display the song name on the LCD and the RGB LEDs
void update_distance_thread(){
   while(1){

      distanceFront = getInputFront();
      distanceBack = getInputBack();
     ThisThread::sleep_for(500ms);
   }
}


int main()
{
   printf("This is the bare metal blinky example running on Mbed OS %d.%d.%d.\n",
MBED_MAJOR_VERSION, MBED_MINOR_VERSION, MBED_PATCH_VERSION);




   sampleDistance.attach(&distanceADC,50ms);
   sampleCounter.attach(&updateCounter,5ms);
   sampleOdometry.attach(&updateLocation,50ms);

   char action = 'Z';
   char buff;
   int desiredDistance = 0;
   int state = 1;
   bool inAction = false;
   bool inRead = true;
   speed = SpeedLow;
   float thetaNow = 0;
   while (true)
   {

      // thread_sleep_for(WAIT_TIME_MS);
      thread_sleep_for(100);
```

```
if(inRead){
   led1 = !led1;
   // printf("Action : \n");
   while((action !='A' && action !='B' && action !='C' && action != 'D')){
      action = getCharacter(true);
   }
   // printf("distance or direction : \n");
   buff = getCharacter(true);
   inAction = false;
   while (buff != '#') {
      desiredDistance = desiredDistance * 10 + (buff - 0x30);
      buff = getCharacter(true);
      inRead= false;
      inAction = true;
   }
}
if(inAction){
   led1 = 1;
   switch (action) {
      case 'A':
      if(d < desiredDistance){
         Car.goForward(speed);
         thread_sleep_for(50);
      }else{
      Car.stop();
      inRead= true;
      inAction = false;
      desiredDistance = 0;
      theta = 0;
      action = 'Z';
      }
         break;
      case 'B':
      if(d < desiredDistance){
         Car.goBackward(speed);
         thread_sleep_for(50);
      }else{
      Car.stop();
      inRead= true;
      inAction = false;
      desiredDistance = 0;
      theta = 0;
      action = 'Z';
      }
```

```
        break;
        case 'C':
        if(theta < (desiredDistance*pi/180)){
           Car.turnLeft(0.2f);
           thread_sleep_for(50);
        }else{
        inRead= true;
        inAction = false;
        desiredDistance = 0;
        action = 'Z';
        theta = 0;
        Car.stop();

        }
           break;
        case 'D':
        if(theta > ((-1) * desiredDistance * pi/180)){
           Car.turnRight(0.2f);
           thread_sleep_for(50);
        }else{
        inRead= true;
        inAction = false;
        desiredDistance = 0;
        theta = 0 ;
        action = 'Z';
        Car.stop();
        }
           break;
        default:
           Car.stop();
    }
}


if (Car.myGear() == Forward) {
    if (distanceFront < 20.0) {
        speed = GearPark;
    } else if (distanceFront < 30.0 ) {
        speed = SpeedLow;
    } else if (distanceFront < 40.0) {
        speed = SpeedMid;
    } else {
        speed = SpeedHigh;
```

```
            }
        } else if (Car.myGear() == Backward){
            if (distanceBack < 20.0) {
                speed = GearPark;
            } else if (distanceBack < 30.0 ) {
                speed = SpeedLow;
            } else if (distanceBack < 40.0) {
                speed = SpeedMid;
            } else {
                speed = SpeedHigh;
            }
        }

         }
}
```

SmartCar.cpp

```
#include "SmartCar.h"
#include "mbed.h"


DigitalOut motorBackLeftF(PA_11);
DigitalOut motorBackLeftB(PA_12);
PwmOut speedPWMBackLeft(ARDUINO_UNO_D3);
PwmOut speedPWMBackRight(ARDUINO_UNO_D5);
DigitalOut motorBackRightB(ARDUINO_UNO_D2);
DigitalOut motorBackRightF(ARDUINO_UNO_D4);
PwmOut speedPWMFrontLeft(ARDUINO_UNO_D6);
DigitalOut motorFrontLeftB(ARDUINO_UNO_D7);
DigitalOut motorFrontLeftF(ARDUINO_UNO_D8);
PwmOut speedPWMFrontRight(ARDUINO_UNO_D9);
DigitalOut motorFrontRightF(ARDUINO_UNO_D10);
DigitalOut motorFrontRightB(ARDUINO_UNO_D11);

SmartCar::SmartCar() : speed_(0) {}

const float max_change = 0.05f;

float absolute(float difference){
   if (difference<0) return -1 * difference;
   else return difference;
}

float SmartCar::smoothAcc(float desired_vel){
   // float difference = desired_vel - speed_;
   // if (absolute(difference) < max_change){
   //     return desired_vel;
   // } else {
   //     if (difference < 0){
   //         desired_vel = speed_ - max_change;
   //     } else {
   //         desired_vel = speed_ + max_change;
   //     }
   // return desired_vel;
   // }
   return desired_vel;
}

void SmartCar::goForward(float speed) {
```

```
    motorFrontLeftF = 1;
    motorFrontLeftB = 0;
    speed = smoothAcc(speed);
    speedPWMFrontLeft.write(speed);
    motorFrontRightF = 1;
    motorFrontRightB = 0;
    speedPWMFrontRight.write(speed);
    motorBackLeftF = 1;
    motorBackLeftB = 0;
    speedPWMBackLeft.write(speed);
    motorBackRightF = 1;
    motorBackRightB = 0;
    speedPWMBackRight.write(speed);
    gear_ = 1;
    speed_ = speed;
}

void SmartCar::goBackward(float speed) {
    speed = smoothAcc(speed);
    motorFrontLeftF = 0;
    motorFrontLeftB = 1;
    speedPWMFrontLeft.write(speed);
    motorFrontRightF = 0;
    motorFrontRightB = 1;
    speedPWMFrontRight.write(speed);
    motorBackLeftF = 0;
    motorBackLeftB = 1;
    speedPWMBackLeft.write(speed);
    motorBackRightF = 0;
    motorBackRightB = 1;
    speedPWMBackRight.write(speed);
    gear_ = -1;
    speed_ = speed;
}

void SmartCar::turnRight(float speed) {
    speed = smoothAcc(speed);
    motorFrontLeftF = 0;
    motorFrontLeftB = 1;
    speedPWMFrontLeft.write(speed);
    motorFrontRightF = 1;
    motorFrontRightB = 0;
    speedPWMFrontRight.write(speed);
    motorBackLeftF = 0;
```

```
	motorBackLeftB = 1;
	speedPWMBackLeft.write(speed);
	motorBackRightF = 1;
	motorBackRightB = 0;
	speedPWMBackRight.write(speed);
	gear_ = 2;
	speed_ = speed;
}

void SmartCar::turnLeft(float speed) {
	speed = smoothAcc(speed);
	motorFrontLeftF = 1;
	motorFrontLeftB = 0;
	speedPWMFrontLeft.write(speed);
	motorFrontRightF = 0;
	motorFrontRightB = 1;
	speedPWMFrontRight.write(speed);
	motorBackLeftF = 1;
	motorBackLeftB = 0;
	speedPWMBackLeft.write(speed);
	motorBackRightF = 0;
	motorBackRightB = 1;
	speedPWMBackRight.write(speed);
	gear_ = 3;
	speed_ = speed;
}

void SmartCar::goDiagonalLeft(float speed) {
	motorFrontLeftF = 0;
	motorFrontLeftB = 0;
	speedPWMFrontLeft.write(speed);
	motorFrontRightF = 1;
	motorFrontRightB = 0;
	speedPWMFrontRight.write(speed);
	motorBackLeftF = 0;
	motorBackLeftB = 0;
	speedPWMBackLeft.write(speed);
	motorBackRightF = 1;
	motorBackRightB = 0;
	speedPWMBackRight.write(speed);
}

void SmartCar::goDiagonalRight(float speed) {
	motorFrontLeftF = 1;
```

```
	motorFrontLeftB = 0;
	speedPWMFrontLeft.write(speed);
	motorFrontRightF = 0;
	motorFrontRightB = 0;
	speedPWMFrontRight.write(speed);
	motorBackLeftF = 1;
	motorBackLeftB = 0;
	speedPWMBackLeft.write(speed);
	motorBackRightF = 0;
	motorBackRightB = 0;
	speedPWMBackRight.write(speed);
}



void SmartCar::slideRight(float speed) {
	motorFrontLeftF = 1;
	motorFrontLeftB = 0;
	speedPWMFrontLeft.write(speed);
	motorFrontRightF = 0;
	motorFrontRightB = 1;
	speedPWMFrontRight.write(speed);
	motorBackLeftF = 0;
	motorBackLeftB = 1;
	speedPWMBackLeft.write(speed);
	motorBackRightF = 1;
	motorBackRightB = 0;
	speedPWMBackRight.write(speed);
}

void SmartCar::slideLeft(float speed) {
	motorFrontLeftF = 0;
	motorFrontLeftB = 1;
	speedPWMFrontLeft.write(speed);
	motorFrontRightF = 1;
	motorFrontRightB = 0;
	speedPWMFrontRight.write(speed);
	motorBackLeftF = 1;
	motorBackLeftB = 0;
	speedPWMBackLeft.write(speed);
	motorBackRightF = 0;
	motorBackRightB = 1;
	speedPWMBackRight.write(speed);
}
```

```
void SmartCar::stop() {
        motorFrontLeftF = 0;
        motorFrontLeftB = 0;
        motorFrontRightF = 0;
        motorFrontRightB = 0;
        motorBackLeftF = 0;
        motorBackLeftB = 0;
        motorBackRightF = 0;
        motorBackRightB = 0;
        gear_ = 0;
        speed_ = 0;
}


int SmartCar::mySpeed() {
   return speed_;
}

int SmartCar::myGear() {
   return gear_;
}
```

SmartCar.h:

```
#ifndef SMARTCAR_H
#define SMARTCAR_H

class SmartCar {
private:
  int speed_;
  // 0 = park; 1 = forward; -1 = backward; 2 = right; 3 = left
  int gear_;


public:
  SmartCar();
  void goForward(float speed);
  void goBackward(float speed);
  void goDiagonalRight(float speed);
  void goDiagonalLeft(float speed);
  void turnRight(float speed);
  void turnLeft(float speed);
  void slideLeft(float speed);
  void slideRight(float speed);
  float smoothAcc(float desired_vel);
  void stop();
  int mySpeed();
  int myGear();




};

#endif // SMARTCAR_H
```

calculator.cpp:

```
#include "calculator.h"
#include "mbed.h"
#include <cstdio>

// array of keys used on the keyPad
// you can modify this if you like
char keys[16] = {'1','2','3','A',
                 '4','5','6','B',
                 '7','8','9','C',
                 '*','0','#','D'};

// Create a BufferedSerial object with a default baud rate
static BufferedSerial pc(USBTX, USBRX);

// code for debounce timer used later in the lab
Timer debounce;
using namespace std::chrono;

// YOUR JOB:
// Declare additional variables for keyPad as needed

// NOTE:
// All functions currently return dummy values so the code will
// compile while you are working. Be sure to update the return values!


// function used to setup the board at the beginning of main()
void setupBoard(void){

    // needed to use thread_sleep_for in debugger
    // your board will get stuck without it :(
    #if defined(MBED_DEBUG) && DEVICE_SLEEP
        HAL_DBGMCU_EnableDBGSleepMode();
    #endif

    // YOUR JOB
    // Initialize keyPad (as needed)
}

// perform one of four operations on num1, num2 using given operation
// A = addition
```

```
// B = subtraction
// C = multiplication
// D = division
// return the result as a float
// If the operation is invalid, return nanf("");
float computation(int num1, int num2, char operation){
   switch (operation) {
      case 'A':
         return num1 + num2 ;
         break;
      case 'B':
         return num1 - num2;
         break;
      case 'C':
         return num1 * num2;
         break;
      case 'D':
         return (float)num1/ num2;
         break;
      default:
         return nanf("");
   }
}

// Get a number from the serial terminal or keyPad. Return
// the absolute value of the number as an int.
// If the number is invalid, returns error code -1.
// Also returns the operation and whether or not the number is negative.
int getNumber(int number, bool keyPad, char* operation, bool* negative){
   char buff = getCharacter(keyPad);
   int result;
   //checking for negative
   if (buff == '*') {
      *negative = true;
      buff = getCharacter(keyPad);
      result = (buff - 0x30);
   } else {
      *negative = false;
      //if we didn't write in a *, we need a number whether it's first number or second one
      if ((buff >= 0x30) && (buff <= 0x39)) {
         result = (buff - 0x30);
      } else {
         return -1;
      }
```

```
    }
    //at this point we need to either keep writing in numbers or write in the operation or # if it's second
number.
    buff = getCharacter(keyPad);
    while ((buff >= 0x30) && (buff <= 0x39)){
        result = result * 10 + (buff - 0x30);
        buff = getCharacter(keyPad);
    }
    if (number == 1) {
        switch (buff) {
            case 'A':
                *operation = 'A';
                break;
            case 'B':
                *operation = 'B';
                break;
            case 'C':
                *operation = 'C';
                break;
            case 'D':
                *operation = 'D';
                break;
            default:
                return -1;
        }
    } else if (number == 2) {
        if (buff == '#') {
            return result;
        } else {
            return -1;
        }
    }
    return result;
}

// Scan the keyPad for a valid keypress and return it as a character variable.
// Input the key mapping as an array.
// The keyPad will need to be debounced using the debounce timer.
char getKeyPress(char keys[]){
// BusIn rows (PB_7, PC_13, PC_14, PC_15);
// BusOut cols (PH_0, PH_1, PC_2, PC_3);
    // THE FIRST WAY OF DOING THIS

//     DigitalIn row1(PB_7, PullUp);
```

```
//    DigitalIn row2(PC_13,PullUp);
//    DigitalIn row3(PC_14,PullUp);
//    DigitalIn row4(PC_15,PullUp);

//    DigitalOut col1(PH_0);
//    DigitalOut col2(PH_1);
//    DigitalOut col3(PC_2);
//    DigitalOut col4(PC_3);
//    col1= 1;
//    col2= 1;
//    col3 = 1;
//    col4 = 1;

//    DigitalOut* col[] = {&col1, &col2, &col3, &col4};
//    DigitalOut* currentCol;
//    DigitalIn* row[] = {&row1, &row2, &row3, &row4};
//    DigitalIn* currentRow;
//    while(1){
//        for (int i = 0; i<4 ; i++) {
//            currentCol = col[i];
//            *currentCol = 0;
//            for (int j = 0; j<4; j++){
//                currentRow = row[j];
//                if (!*currentRow){
//                    debounce.start();
//                    while (duration_cast<milliseconds>(debounce.elapsed_time()).count() < 400);
//                    debounce.stop();
//                    debounce.reset();
//                    return keys[j* 4 + i];
//                }
//            }
//            *currentCol = 1;
//        }
//    }
// }


//    //THE SECOND WAY OF DOING THIS

    BusIn busIn(PC_10, PC_12, PA_13, PA_14);
    busIn.mode(PullUp);
    BusOut busOut(PA_15, PB_7, PC_13, PC_3);
    busOut.write(0b1111);
    int numberOld = 0;
```

```
    while(1){
        for (int i = 0; i<4; i++){
            busOut.write(15 - (1<<i));
            if(busIn.read() != 15){
                int number = 15 - busIn.read();
                int j = -1;
                bool test = false;
                while(j<4 && (!test)) {
                    j++;
                    test = ~busIn.read() & 1<<j;
                }
                debounce.start();
                while (duration_cast<milliseconds>(debounce.elapsed_time()).count() < 400);
                debounce.stop();
                debounce.reset();
                return keys[j*4 + i];
            }
        }
    }
}

// Get a single character from the keyPad (keyPad = true)
// or serial monitor (keyPad = false).
// Code should also "echo" the character to the serial terminal
// using pc.write(&buff,1);
char getCharacter(bool keyPad){
    //right now only reads and writes from serial monitor.
    char buff;
    if (keyPad) {
        char buff = getKeyPress(keys);
        pc.write(&buff,1);
        return buff;
    } else {
        pc.read(&buff, 1);
        pc.write(&buff,1);
        return buff;
    }
}
```

calculator.h

```
#ifndef __CALC__
#define __CALC__


// declaring functions used in calculator.cpp
// see calculator.cpp for more details
float computation(int num1, int num2, char operation);
int getNumber(int number, bool keyPad, char* operation, bool* negative);
char getCharacter(bool keyPad);
char getKeyPress(char keys[]);
void setupBoard(void);

#endif
```