Project-16.01.2013

Small Self Balancing robot

Arkadi Rafalovich

Feel free to use any part or all in the spirit of open source and DIY

The project`s goal is creating a small self-balancing robot capable of using 2 or 4 of its wheels. The project was intended to create the smallest possible footprint and have Bluetooth connectivity.

System components: (most bought from eBay)

Part	Price
arduino nano v3 ATmega328 microcontroller	13$
4 micro motors, 2 at 300 RPM and 2 at 540	6$ each 24$ total
2 H bridge chips l293d	1.25$ each 2.5$ total
4 wheels 2 42 mm and 2 70 mm	10$ total (pololu.com is expensive)
Gyro and accelerometer - MPU6050	15$ (its cheaper now)
Bluetooth module BTM-5	10$
2 Li-Poly 7.4V batteries 500 mah	5$ each 10$ total
Misc: leds, resistor, regulators, wires etc..	5.5$ (rounding up)
Total:	90$ *

* Of course the development is more expensive due to some trial and errors.

Choice of the parts:

I have chosen the parts for the project with respect to what is available to me and in the budget.

All of the major parts (expansive) are easily unassembled from the robot so the design have been with this consideration as well.

Motors at high speed are a good choice, in order that the system will respond fast enough, around 300-500 RPM is a good choice. Regarding the required torque I have performed a simple calculation, (by considering the weight that the motors will require to lift); I have estimated that the robot will weight half a kilo. (It is 350 grams) the mass will be pretty centered so it is around 5 cm from the motor shaft. And I will need to lift it at approximately from up to 30 degrees. Resulting in around 2 kg*cm required torque and a 1 kg*cm per motor. My actual motors have a torque of half of that and they work great with a lot to spare. This method is an over estimate but gives some numbers to work with.

Batteries: 7.4 volts small enough to fit inside the robot and with enough capacity to work for an hour. I have separated the power for the controller and the motors in order to isolate the controller from the noise caused by the motors (and overdue with a back thought)

H bridge: l293 for its internal kickback diodes and they fit the required current (which is 0.5 amp per motor at stall).

Wheels and motors have been chosen with motor ratio in mind.

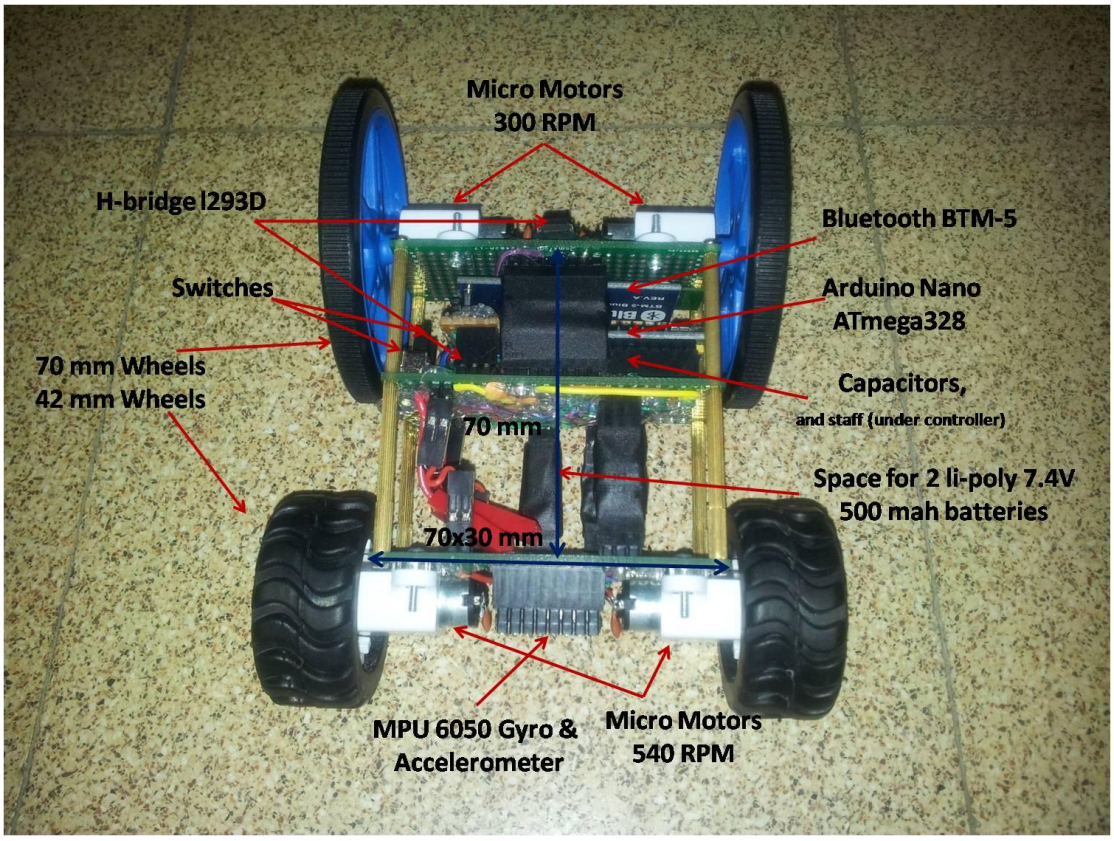
Bluetooth BTM 5 is a popular module on eBay and cheap enough. I have added a voltage divider for the Arduino's TX to the module's RX in order to reduce the input voltage from 5 to 3.3 by using a 22 and 12 kOhm resistors. I use the BlueTerm Application on the android, which is a free application that monitors the communication through Bluetooth, allowing sending and receiving information.

The MPU 6050 is the holy grail of the Gyro Accelerometers in my opinion. It is very cheap (today you can get one for just 8 $) and it is coming with a built in microprocessor which reduces the load from the Arduino board. In addition there is a library very easy to find on the web that simplifies a lot of the communication with the device. The library can give you the actual angles of the sensor already filtered and calibrated which simplifies your life a lot.

As a side project I have tinkered a bit with a separated analog accelerometer and a gyro and my conclusions for that in case you intend to use it as the angle measurement are: if you intend to use only an accelerometer I suggest placing it on the geometric line of the shaft of the wheels such as my sensor in my robot this is for removing unwanted noise from the robots angle accelerations which can ruin the measured angle with the accelerometer. In general the accelerometer is good for measuring static angles because it relies on the gravitational vector for the computation of the angles, and the gyro is good for the dynamic change of the angle for it measures the angular velocity and by integration the angle is calculated. For the Gyro the integration and the sensor calibration are causing an error for the measured angle which accumulates in time, thus the gyro isn`t good enough for measuring the robots angle precisely, but with the combination of both sensors the precise angle can be measured correctly. The math is pretty simple but tedious especially on the controller so I prefer the use of the MPU6050 which handles all the issues by itself resulting with a clear and precise measurements.

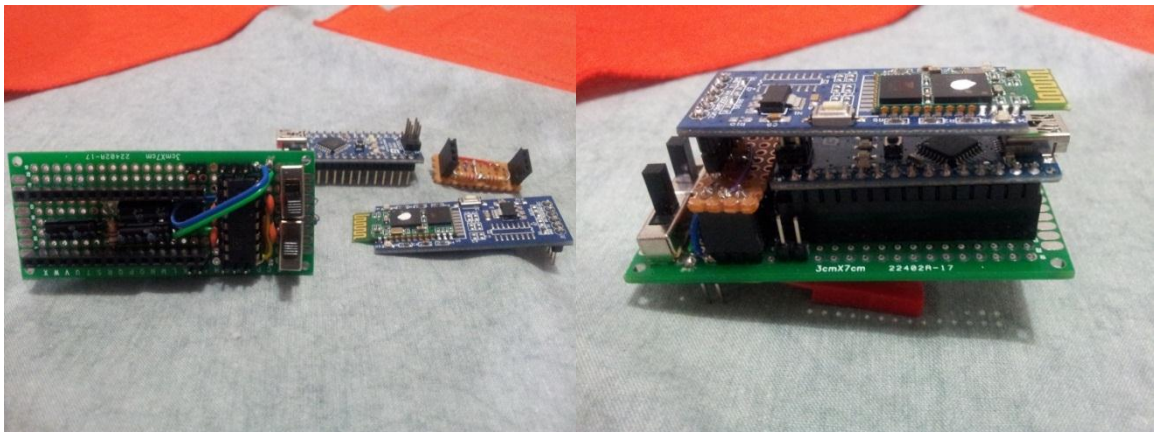
The controller is one that will be easy to work with and small to fit the system.

A schematic of the system assembled (some modifications were performed for the added batteries as you may notice in later images):

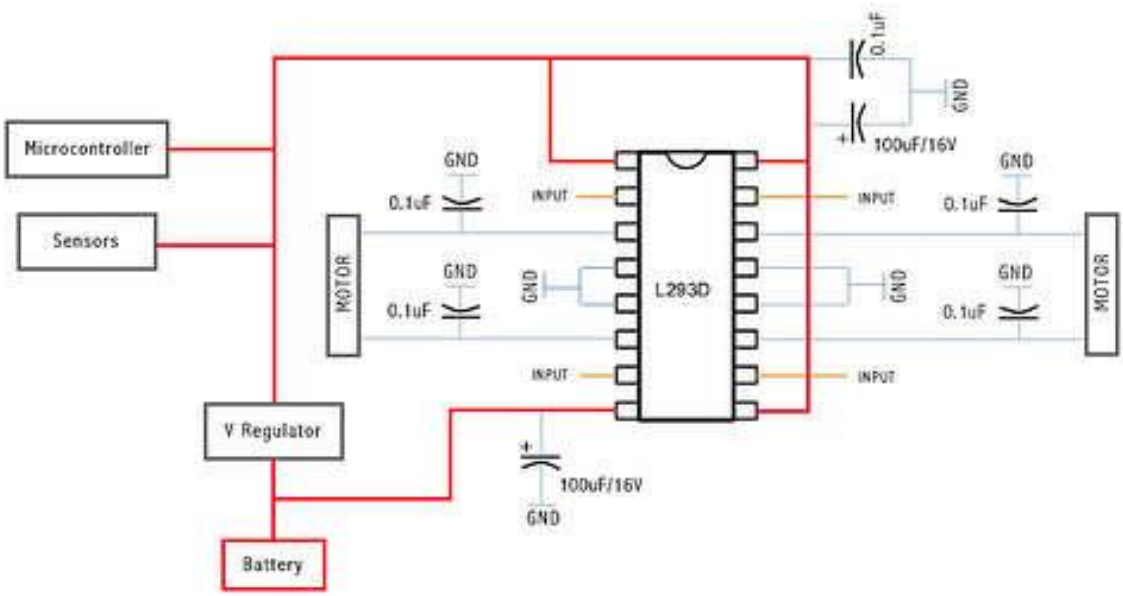


Actual size is: 95x70x30 mm without wheels.
With wheels 140x70x[70:42] mm.

The internal unit consists of the controller Bluetooth H bridge voltage regulators and wiring and switches:



The H bridge schematics are approximately as follows: (based on a tutorial found on the web, didn`t find it to add here just kept the image)



The overall layout of the controller connections are as follows:

Arduino nano pin	Connection:
0 – RX	Serial TX BTM 5
1 – TX	Serial RX BTM 5
2 – Interrupt	MPU 6050 Interrupt
3 – Interrupt, PWM	
4	H-Bridge Direction1
5 – PWM	
6 – PWM	H-Bridge PWM1
7	H-Bridge Direction2
8	H-Bridge Direction3
9 – PWM	H-Bridge PWM2
10 – PWM	H-Bridge PWM3
11 – PWM	H-Bridge PWM4
12	H-Bridge Direction4
13 Led	On board Led
A0	
A1	
A2	
A3	
A4 – SDA	MPU 6050 SDA
A5 – SCL	MPU 6050 SCL
A6	
A7	

Tuning the PID controller:

First if you are not familiar with the concept of PID controller, check the wiki page it has a good explanation http://en.wikipedia.org/wiki/PID_controller. I will later comment on how it affects the self-balancing. Regarding the implementation of the standard control low in c you can check the following link for the presentation of the PID controller in discrete time: http://www.regnumelectronic.com/Docs/PID.pdf and later on how I have implemented the controller in my code.

A few things needed to be considered before adjusting the controller itself:

- The control loop, the loop which updates the motors commands should run at a more or less fixed frequency otherwise the system may be hard to control.
- The frequency of the control loop should be at list an order higher of the frequency of the system. For mechanical systems 50 Hz is usually enough unless it is a quadracopter or something very fast. In my system it is set to 50 Hz. (by the MPU 6050 library).
- The PID controller is best in linear systems. The cheap or high geared motors are usually not very linear meaning that they have a high width of input at which they don`t respond due to internal friction. For that purpose I have added the "Preamplifier" and "dead zone". In practice you measure the input at which the motors are just starting to move and it will be your preamplify value which you add to the command to the motor. In addition I suggest adding a dead zone, an arbitrary width at which the motors won`t be active at all, it reduces some issues with high frequency switching of the motors. (50 Hz as our control low).

Now regarding the tuning of the parameters:

It is always handy to be able to tune the parameters on the fly so consider some wireless communication for that purpose.

Start with the deferential gain and integral gain at zero, and only apply the proportional gain up until the system will start responding for your liking. Meaning standing up, perhaps with a bit tilting and falling after a few seconds.

Now for adding the deferential and integral gains, the integral gain will improve the rising time of the system, but increase the overshoot and remove the steady state error. The deferential will do more or less the opposite regarding the response time and overshoot. So you just need to tinker with them to adjust. A too high integral gain will result in tilting at high amplitudes and perhaps falling. A high differential will add ripples when it stands upright which I don`t think are good for the motors.

The Arduino code: (attached at the end of the file)

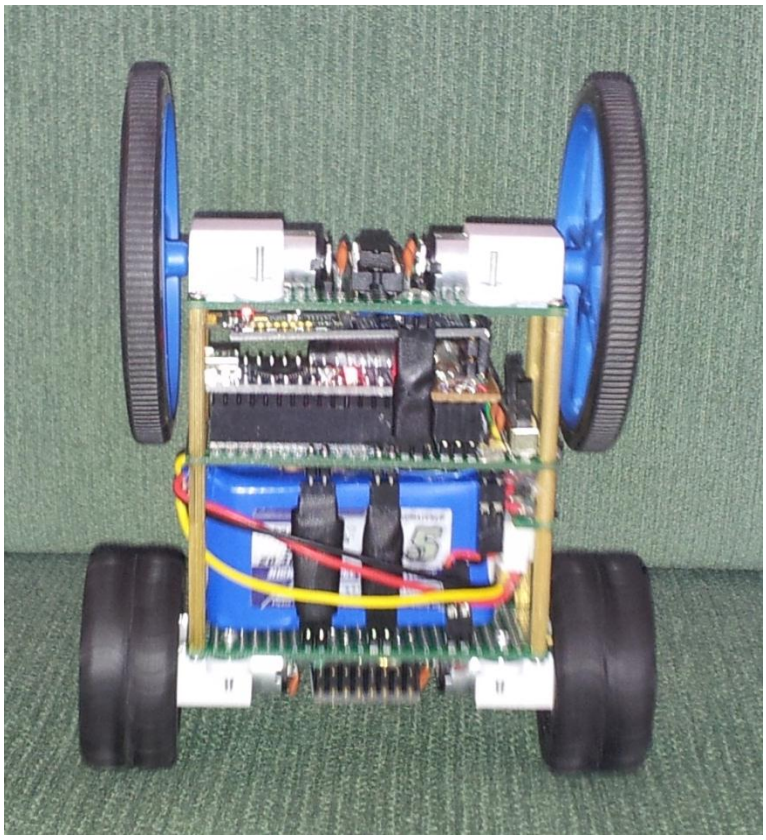
The code is basically self-explanatory; I made sure to insert as many comments as possible in between the lines so it will be easy to understand.

Some notes regarding the code:

For communication between the android and Arduino through Bluetooth I use the BlueTerm application on the android, and by sending characters from the android I update the transmitting data from the arduino using the Case function. In addition this mode allows the update for the PID parameter on the fly for fast tuning of the parameters.

Note the implementation of the linearization of the motors by using the preamp value for the motors commands and the added dead zone in order to remove the high frequency switching of the motors.

And finally the robot working and kicking:



4 wheels mode as a racing car: http://www.youtube.com/watch?v=bdeWWh5-a7U

Self-balancing mode: http://www.youtube.com/watch?v=K2tlKteO_ZU

Combining the two modes with android control using the application "Bluetooth RC Car" by Andi.Co (great app cheers to the developer) :
http://www.youtube.com/watch?v=GIyY5aSZYEo&feature=youtu.be

Future work because I have just started with the fun ☺

Implementation of the 4 wheels mode drive (as a 4 wheel speed car, can get to 6-7 Km/h) with the self-balancing mode by measuring the angle of the robot and setting the working mode by the angle. Allowing the user to bring the car to a wall climb it a bit and thus returning to a two wheels mode. Done update 16.01.2012

Adding some sensors and led to the robot. I have yet pins to spare so an RGB led is going to be placed for the fun of it. And some analog sensor such as light sensor, magnetic sensor,

temperature sensor and we will see what more. There is a chance in the far future to place an analog ultrasonic range sensor but because it is expensive (25$) not in the close future. But when it will happen be expecting for an autonomous robot ☺

Have fun, and feel free to contact me with any questions, suggestion and whatever you like.

Publish share use copy edit and have fun!

Arkadi Rafalovich

MSc student, Tel Aviv University

(The project was done as a home hobby without any connections to the university)

Thanks for all who supported the project and helped in writing the documentation.

For the open source library of the MPU 6050 sensor from Jeff Rowberg
https://github.com/jrowberg/i2cdevlib/blob/master/Arduino/MPU6050/Examples/MPU6050_DMP6/MPU6050_DMP6.ino

For the free android applications "BlueTerm" and "Bluetooth RC Car"

And especially my girlfriend and life partner Inbal gelz, for being tolerant and supporting of this hobby of mine which is so much time consuming (and not so healthy for the bank account either).

Long Live Arduino

The Arduino code: (attached as a separated file)

```
  // Self balancing Mini robot with 4 motors.//
//////////////////////////////////////////////////////////////////
//////////////////////////////////////
///////Created by///////////////
// Arkadi Rafalovich///////
//Last Update 16.01.2013////
//////////////////////////////////////

// libraries in use:
// for running MPU 6050 6/21/2012 by Jeff Rowberg
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
//thatnks for all of the open source libraries

// ================================================================
// ===               Define Variables                ===
// ================================================================

//define motor parameters:
#define MpwmBL 6   //BOTTOM LEFT
#define MpwmBR 9   //BOTTOM RIGHT
#define MpwmTR 10 // TopRight
#define MpwmTL 11 // TopLeft

#define MdirectionR1 4
#define MdirectionR2 7
#define MdirectionL1 8
#define MdirectionL2 12

// Define motors preamp and deadzone.
//A method to remove the dead zone of the motors and keep them silent when not working.
#define Preamp 35
#define DeadZone 15

//Define the angle at which the robot moves between 4 wheel control and 2 wheel self balancing control
#define ModeAngle 50

// adjust motor speed and wheel size 42 mm with 540 rpm 70 mm with 300 rpm: 1.08 for faster motors =>> neglectable
// #define MSSDiff 1.08


// desired motor speed right and left:
int DMSR=0;
int DMSL=0;

// Bluetooth command
char command='S';

 // state true for self balance and state false for 4WD car
boolean state;

// control variables to change the direction of movment
// turn right
int RMotorDiff=0;
// turn left
int LMotorDiff=0;

// Speed adjustment variable
byte ComStep=0;

// ================================================================
// ===               Define PID Controller  Variables              ===
// ================================================================

// proportional gain
float Kp=7.5;
// deferential gain
float Kd=0.5;
// integral gain
float Ki=12.5;
```

```
// variables for PID controller

//// Type A PID controller
//// error=Set angle-actual angle
//float error=0;
//float Lasterror=0;
//float LLasterror=0;

// Type C PID controller
float error=0;
float Aangle=0;
float Lastangle=0;
float LLastangle=0;

// Angle adjustment for Forward Backward control Is through the Desired angle
// Desired angle
float Dangle=0;

// PID controller command
float COcommand =0;


// ================================================================
// ===               Define MPU 6050  Variables               ===
// ================================================================
MPU6050 mpu;
#define LED_PIN 13
bool blinkState = false;

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorInt16 aa;         // [x, y, z]            accel sensor measurements
VectorInt16 aaReal;     // [x, y, z]            gravity-free accel sensor measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame accel sensor measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
VectorInt16 aaOffset; // [x, y, z]            accel sensor measurements offset

float euler[3];         // [psi, theta, phi]    Euler angle container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and gravity vector
float yprOffset[] = {0, 0, 0};           // [yaw, pitch, roll]  offset  yaw/pitch/roll container and gravity vector



// ================================================================
// ===               INTERRUPT DETECTION ROUTINE                ===
// ================================================================

volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin has gone high
void dmpDataReady() {
   mpuInterrupt = true;
}



// ================================================================
// ===                      INITIAL SETUP                       ===
// ================================================================

// the setup routine runs once when you press reset:
void setup()  {
     // join I2C bus (I2Cdev library doesn't do this automatically)
   Wire.begin();
     // initialize serial communication
  Serial.begin(115200);

  ////////////////////////////////////
```

```
  // initialize mpu6050///
  ////////////////////////////////////

     // initialize device
    Serial.println(F("Initializing I2C devices..."));
    mpu.initialize();

    // verify connection
    Serial.println(F("Testing device connections..."));
    Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") : F("MPU6050 connection failed"));

       // load and configure the DMP
    Serial.println(F("Initializing DMP..."));
    devStatus = mpu.dmpInitialize();

    // make sure it worked (returns 0 if so)
    if (devStatus == 0) {
       // turn on the DMP, now that it's ready
       Serial.println(F("Enabling DMP..."));
       mpu.setDMPEnabled(true);

       // enable Arduino interrupt detection
       Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));
       attachInterrupt(0, dmpDataReady, RISING);
       mpuIntStatus = mpu.getIntStatus();

       // set our DMP Ready flag so the main loop() function knows it's okay to use it
       Serial.println(F("DMP ready! Waiting for first interrupt..."));
       dmpReady = true;

       // get expected DMP packet size for later comparison
       packetSize = mpu.dmpGetFIFOPacketSize();
    } else {
       // ERROR!
       // 1 = initial memory load failed
       // 2 = DMP configuration updates failed
       // (if it's going to break, usually the code will be 1)
       Serial.print(F("DMP Initialization failed (code "));
       Serial.print(devStatus);
       Serial.println(F(")"));
    }

    // configure LED for output
    pinMode(LED_PIN, OUTPUT);

  ////////////////////////////////////////
  // initializemotor pins///
  ////////////////////////////////////////
  // declare direction pins as aoutup
  pinMode(MdirectionR1, OUTPUT);
  pinMode(MdirectionR2, OUTPUT);
  pinMode(MdirectionL1, OUTPUT);
  pinMode(MdirectionL2, OUTPUT);

    // declare PWM pins as aoutup
  pinMode(MpwmBL, OUTPUT);
  pinMode(MpwmBR, OUTPUT);
  pinMode(MpwmTR, OUTPUT);
  pinMode(MpwmTL, OUTPUT);
}


// ================================================================
// ===                  MAIN PROGRAM LOOP                     ===
// ================================================================

void loop()  {

     // if MPU 6050 initialization failed exit.
    if (!dmpReady) return;

  // check if  MPU 6050  ready
  if (!mpuInterrupt && fifoCount < packetSize) {

    // check if there is incoming command
```

```
 if (Serial.available() > 0) {
            // read the incoming byte:
            command = Serial.read();
switch (command) {
  // Controller commands:
                        case 'S':
                         Dangle=0;
                         RMotorDiff=0;
                         LMotorDiff=0;
                        break;

                        case 'F':
                          Dangle=ComStep*0.5;
                        break;
                          case 'B':
                         Dangle=-ComStep*0.5;
                          break;
                         case 'R':
                         RMotorDiff=ComStep*10;
                        break;
                          case 'L':
                        LMotorDiff=ComStep*10;
                          break;

                         case '1':
                        ComStep=1;
                          break;
                          case '2':
                       ComStep=2;
                          break;
                          case '3':
                        ComStep=3;
                          break;
                          case '4':
                        ComStep=4;
                          break;
                          case '5':
                       ComStep=5;
                          break;
                          case '6':
                        ComStep=6;
                          break;
                          case '7':
                        ComStep=7;
                          break;
                          case '8':
                       ComStep=8;
                          break;
                          case '9':
                        ComStep=9;
                          break;

// turned off when not adjusting control due to conflicts with the BluetoothRC android app.
//   // PID Adjusting Commands:
//                         case 'P':
//                           Kp=Kp+0.5;
//                         break;
//                           case 'p':
//                          Kp=Kp-0.5;
//                           break;
//                          case 'I':
//                          Ki=Ki+0.5;
//                         break;
//                           case 'i':
//                         Ki=Ki-0.5;
//                           break;
//                         case 'D':
//                          Kd=Kd+0.1;
//                         break;
//                           case 'd':
//                         Kd=Kd-0.1;
//                           break;
//
                      default:
                     // nothing
```

```
                    ;

                }

          // end incoming command
          }


// ================================================================
// ===                  Control Low                 ===
// ================================================================

//  // PID type A controller
//  LLasterror=Lasterror;
//  Lasterror=error;
//  // error is the Desired angle - actual angle
//  error=Dangle-ypr[1]*180/M_PI;
//
//  // PID controller at 50 HZ
//  COcommand=COcommand+Kp*(error-Lasterror)+(Ki/50*error)+Kd*50*(error-2*Lasterror+LLasterror);

// PID type C controller
  LLastangle=Lastangle;
  Lastangle=Aangle;
  Aangle=ypr[1]*180/M_PI;
  // error is the Desired angle - actual angle
  error=Dangle-Aangle;

  // PID controller at 50 HZ
  COcommand=COcommand-Kp*(Aangle-Lastangle)+Ki/50*error-Kd*50*(Aangle-2*Lastangle+LLastangle);
  // constrain the Control low command to a valid range
     COcommand=constrain(COcommand, -255, 255);

// before updating the motors commad chek the angle of the robot if it is in the desire range:
if (abs(Aangle)<ModeAngle) {

   DMSR= (int)   COcommand + RMotorDiff;
   DMSL=  (int)   COcommand + LMotorDiff;

 // set motor speeds: // state true for self balance and state false for 4WD car
 SetMotorSpeed(DMSR, DMSL, true);
} else{
  // not in the range so the motoors COcommad will be zeroed in order to delete the memory of the control low
   COcommand=0;
   // use same variable with adjusted parameters
   DMSR= (int)   (Dangle*10 + RMotorDiff - LMotorDiff)*3;
   DMSL=  (int)   (Dangle*10 + LMotorDiff- RMotorDiff)*3;
   SetMotorSpeed(DMSR, DMSL, false);
}


    // end main code
   }

     // check if  MPU 6050  not ready again before sending data
  if (!mpuInterrupt && fifoCount < packetSize) {

    switch (command) {
                    case 'a':
          //Display Variables
          Serial.print("MPU ");
          Serial.print("ypr\t");
          Serial.print(ypr[0] * 180/M_PI);
          Serial.print("\t");
          Serial.print(ypr[1] * 180/M_PI);
          Serial.print("\t");
          Serial.println(ypr[2] * 180/M_PI);
                    break;

                     case 'c':
          Serial.print("command:    ");
          Serial. print(command);
          Serial.print("  RM Speed:    ");
          Serial. print(DMSR);
          Serial.print("  LM Speed:    ");
```

```
        Serial. println(DMSL);
                    break;

                     case 'C':
        Serial.print("PID");
        Serial.print("  Kp:");
        Serial.print(Kp);
        Serial.print("  Ki:");
        Serial.print(Ki);
        Serial.print("  Kd:");
        Serial.println(Kd);
                    break;

                    default:
                  // nothing
                  ;
  }

  // end interface. update
 }


    // reset interrupt flag and get INT_STATUS byte
  mpuInterrupt = false;
  mpuIntStatus = mpu.getIntStatus();

  // get current FIFO count
  fifoCount = mpu.getFIFOCount();

  // check for overflow (this should never happen unless our code is too inefficient)
  if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
    // reset so we can continue cleanly
    mpu.resetFIFO();
    Serial.println(F("FIFO overflow!"));

  // otherwise, check for DMP data ready interrupt (this should happen frequently)
  } else if (mpuIntStatus & 0x02) {
    // wait for correct available data length, should be a VERY short wait
    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    // read a packet from FIFO
    mpu.getFIFOBytes(fifoBuffer, packetSize);

    // track FIFO count here in case there is > 1 packet available
    // (this lets us immediately read more without waiting for an interrupt)
    fifoCount -= packetSize;

       // Get Yaw Pitch Roll
       mpu.dmpGetQuaternion(&q, fifoBuffer);
       mpu.dmpGetGravity(&gravity, &q);
       mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

       // Set offset
       ypr[0] -= yprOffset[0];
       ypr[1] -=yprOffset[1];
       ypr[2] -=yprOffset[2];

     // blink LED to indicate activity
    blinkState = !blinkState;
    digitalWrite(LED_PIN, blinkState);
  }

  // end main code
}

// ===============================================================
// ===                  SetMotorSpeed function                 ===
// ===============================================================
// set motor speed parameters recieaved right motor desired speed and left motor desired speed.
// the preamp and deadzone is applied in this part of the code.
// ==> leading for the approximate liniarization of the system by removing the motors dead zone.
  void SetMotorSpeed(int  DMSR, int DMSL, boolean state) {

    // handlig each motor independently
    // start by setting the motor direction:
```

```
  if (DMSR>0) {
    // set forward
  digitalWrite(MdirectionR1,LOW);
  digitalWrite(MdirectionR2,HIGH);
  }
  else{
  // set reverse
  digitalWrite(MdirectionR1,HIGH);
  digitalWrite(MdirectionR2,LOW);
   // change baluse to positive
   DMSR=-DMSR;
 }

   if (DMSL>0) {
    // set forward
  digitalWrite(MdirectionL1,LOW);
  digitalWrite(MdirectionL2,HIGH);
   }
  else {
  // set reverse
  digitalWrite(MdirectionL1,HIGH);
  digitalWrite(MdirectionL2,LOW);
   // change valuse to positive
  DMSL=-DMSL;
  }

  // implement dead zone and preamp:
  if (DMSR<DeadZone){
    // set motor speed to zero
    DMSR=0;
  }
    else{
    DMSR=DMSR+Preamp;
  }

  if (DMSL<DeadZone){
    // set motor speed to zero
    DMSL=0;
  }
    else{
    DMSL=DMSL+Preamp;
  }

  // apply constrain for valid input
  DMSR=constrain(DMSR, 0, 255);
  DMSL=constrain(DMSL, 0, 255);

  if (state) {
  // set motor speed:
    analogWrite(MpwmBR, DMSR);
    analogWrite(MpwmTR, 0);
    analogWrite(MpwmBL, DMSL);
    analogWrite(MpwmTL, 0);
  }
    else{
    analogWrite(MpwmBR, DMSR);
    analogWrite(MpwmTR, DMSR);
    analogWrite(MpwmBL, DMSL);
    analogWrite(MpwmTL, DMSL);
     }


// end set motor speed function
  }
```