```
//HAND:

#include <SPI.h>

#include <nRF24L01.h>

#include <RF24.h>

#include "Wire.h"

#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"

#include <Adafruit_GFX.h>

#include <Adafruit_SSD1306.h>


#define LOW_POWER_TIMEOUT 60000 // 1 minute in milliseconds

#define LOW_POWER_INTERVAL 1000 // Check every 1 second in low power mode

#define SCREEN_WIDTH 128

#define SCREEN_HEIGHT 32

#define OLED_RESET -1

#define SCREEN_ADDRESS 0x3C

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);


MPU6050 mpu;

RF24 radio(8, 9); // CE = 8, CSN = 9


const uint64_t pipeOut = 0xF9E8F0F0E1LL;

const uint64_t pipeIn = 0xF9E8F0F0E2LL;

int lastXValue = 0;

int lastYValue = 0;


const int GESTURE_THRESHOLD = 3; // Minimum change in degrees to count as
movement

const unsigned long GESTURE_IDLE_TIMEOUT = 1800000;  // 1 min to enter low
power

const unsigned long GESTURE_WAKE_DURATION = 50000;  // Must move for at least
2s to wake
```

```
unsigned long gestureStillStart = 0;
unsigned long gestureMoveStart = 0;

struct PacketData {
byte xAxisValue;
byte yAxisValue;
byte speedCommand; // 0-254 representing speed
byte commandFlags; // Bitmask for commands (bit 0: stop, bit 1: brake)
} data;

struct FeedbackData {
bool obstacleDetected;
byte batteryLevel;
bool isEmergencyStopped;
float currentSpeed; // in mph
unsigned long uptime; // in seconds
byte systemStatus; // Bitmask for system status
} carStatus;

uint8_t fifoBuffer[64];
uint16_t packetSize;
Quaternion q;
VectorFloat gravity;
float ypr[3];
bool dmpReady = false;
unsigned long lastDisplayUpdate = 0;
unsigned long lastActivityTime = 0;
bool lowPowerMode = false;

// Speed settings (calibrated for 8mph max)
```

```
const byte MIN_SPEED = 100; // Corresponds to ~3mph

const byte MAX_SPEED = 254; // Corresponds to ~8mph

const byte BRAKE_SPEED = 50; // Reduced speed for braking


void enterLowPowerMode();

void exitLowPowerMode();

void updateDisplay();


void setup() {

Serial.begin(115200);

Wire.begin();

if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {

Serial.println("OLED failed");

while (1);

}

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(0,0);

display.println("Starting...");

display.display();


mpu.initialize();

if (mpu.dmpInitialize() == 0) {

mpu.setDMPEnabled(true);

dmpReady = true;

packetSize = mpu.dmpGetFIFOPacketSize();

Serial.println("✅ MPU6050 Ready.");

} else {

Serial.println("❌ MPU6050 Failed.");
}

if (!radio.begin()) {
Serial.println("❌ NRF24L01 not responding.");
while (1);
}

radio.setPALevel(RF24_PA_LOW);
radio.setDataRate(RF24_250KBPS);
radio.setChannel(108);
radio.enableAckPayload();
radio.setRetries(5,5);
radio.openWritingPipe(pipeOut);
radio.openReadingPipe(1, pipeIn);


Serial.println("✅ RF Transmitter Ready.");

// Initialize data structure
data.speedCommand = MIN_SPEED;
data.commandFlags = 0;

display.clearDisplay();
display.setCursor(0,0);
display.println("Ready!");
display.display();
lastActivityTime = millis();
}

void loop() {
```

```
if (!dmpReady) return;
// Function prototypes

// Check if we should enter low power mode
if (!lowPowerMode && (millis() - lastActivityTime > LOW_POWER_TIMEOUT)) {
enterLowPowerMode();
}
// In low power mode, we check less frequently
if (lowPowerMode) {
delay(LOW_POWER_INTERVAL);
// Check if we should exit low power mode (if there's any activity)
if (radio.available() || mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {
exitLowPowerMode();
}
return; // Skip the rest of the loop in low power mode
}

if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {
mpu.dmpGetQuaternion(&q, fifoBuffer);
mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

int xValue = constrain(ypr[2] * 180 / M_PI, -90, 90); // Roll
int yValue = constrain(ypr[1] * 180 / M_PI, -90, 90); // Pitch

// Gesture change detection
  if (abs(xValue - lastXValue) > GESTURE_THRESHOLD || abs(yValue -
lastYValue) > GESTURE_THRESHOLD) {
    if (!lowPowerMode) {
      lastActivityTime = millis(); // Reset idle timer
    } else {
      if (gestureMoveStart == 0) gestureMoveStart = millis();
      if (millis() - gestureMoveStart > GESTURE_WAKE_DURATION) {
        exitLowPowerMode();
      }
    }
  } else {
    if (!lowPowerMode && millis() - lastActivityTime > GESTURE_IDLE_TIMEOUT) {
      enterLowPowerMode();
    }
    gestureMoveStart = 0; // Reset if no movement
  }

  lastXValue = xValue;
  lastYValue = yValue;

data.xAxisValue = map(xValue, -90, 90, 0, 254);
data.yAxisValue = map(yValue, -90, 90, 254, 0);
// Detect stop gesture (hand flat)
if (abs(ypr[1]) < 0.2 && abs(ypr[2]) < 0.2) {
data.commandFlags |= 0b00000001; // Set stop flag
} else {
data.commandFlags &= ~0b00000001; // Clear stop flag
}
// Detect brake gesture (hand tilted back slightly)
if (ypr[1] < -0.3) {
data.commandFlags |= 0b00000010; // Set brake flag
} else {
data.commandFlags &= ~0b00000010; // Clear brake flag
}
```

```
bool success = radio.write(&data, sizeof(data));

if (success) {
Serial.print("✅ Sent | ");
} else {
Serial.print("❌ Failed | ");
}

Serial.print("X: "); Serial.print(xValue);
Serial.print(" | Y: "); Serial.println(yValue);
// Set speed based on forward tilt (only when moving forward)
if (yValue > 10) { // Forward tilt
data.speedCommand = map(yValue, 10, 90, MIN_SPEED, MAX_SPEED);
} else {
data.speedCommand = MIN_SPEED;
}
/// Send control data
radio.stopListening();
bool sent = radio.write(&data, sizeof(data));
radio.startListening();

// Receive feedback
if (radio.available()) {
radio.read(&carStatus, sizeof(carStatus));
//Exit low power mode if we receive data
if (lowPowerMode) {
exitLowPowerMode();
}
}
```

```
// Update display every 100ms to avoid flickering
if (millis() - lastDisplayUpdate > 100) {
updateDisplay();
lastDisplayUpdate = millis();
}

}
}


void updateDisplay() {
display.clearDisplay();
display.setCursor(0,0);
// First line: Status and battery
display.print("S:");
display.print(carStatus.currentSpeed, 1); // Print speed with 1 decimal place
display.print("mph B:");
display.print(carStatus.batteryLevel); // Print battery level
display.print("%");
// Second line: Status indicators
display.setCursor(0, 8);
if (carStatus.batteryLevel < 10) {
display.print("LOW BAT! ");
}
if (carStatus.obstacleDetected) {
display.print("OBSTACLE ");
}
if (data.commandFlags & 0b00000001) {
display.print("STOP ");
} else if (data.commandFlags & 0b00000010) {
display.print("BRK ");
```

```
}
if (carStatus.isEmergencyStopped) {
display.print("EMERG STOP ");
}
// Third line: System status
display.setCursor(0, 16);
display.print("Uptime: ");
unsigned long hours = carStatus.uptime / 3600;
unsigned long minutes = (carStatus.uptime % 3600) / 60;
unsigned long seconds = carStatus.uptime % 60;
display.print(hours);
display.print(":");
if (minutes < 10) display.print("0");
display.print(minutes);
display.print(":");
if (seconds < 10) display.print("0");
display.print(seconds);
// Fourth line: Error status
display.setCursor(0, 24);
if (carStatus.systemStatus & 0b00000001) {
display.print("MOTOR ERR ");
}
if (carStatus.systemStatus & 0b00000010) {
display.print("SENSOR ERR ");
}
if (carStatus.systemStatus & 0b00000100) {
display.print("RADIO ERR ");
}
if (carStatus.systemStatus & 0b00001000) {
display.print("LOW BATTERY ");
}
```

```
display.display();
} // <-- This was the missing closing brace

void enterLowPowerMode() {
Serial.println("Entering low power mode");
lowPowerMode = true;
// Update display with low power message
display.clearDisplay();
display.setCursor(0, 0);
display.println("Low Power Mode");
display.println("%");
display.println("Move to wake");
display.display();
radio.powerDown();
mpu.setSleepEnabled(true);
}
void exitLowPowerMode() {
Serial.println("Exiting low power mode");
lowPowerMode = false;
gestureMoveStart = 0;
lastActivityTime = millis();

// No need to turn display on since it was never off
radio.powerUp();
mpu.setSleepEnabled(false);
// Clear the low power message by updating display normally
updateDisplay();
}

//CAR
#include <SPI.h>
```

```
#include <nRF24L01.h>
#include <RF24.h>
#include <Servo.h>

#define SIGNAL_TIMEOUT 500
#define MAX_MOTOR_SPEED 228
#define OBSTACLE_DISTANCE_CM 30
#define TURN_DURATION 500
#define EMERGENCY_PAUSE 5000
#define SPEED_CALIBRATION 0.035
#define BATTERY_PIN A6 // Analog pin connected to voltage divider
#define REFERENCE_VOLTAGE 5.0
#define VOLTAGE_DIVIDER_RATIO 1.0 // R1 = R2 = 10kΩ
#define FULL_BATTERY_VOLTAGE 4.2 // 2S LiPo fully charged
#define EMPTY_BATTERY_VOLTAGE 3.3// 2S LiPo empty
#define LOW_BATTERY_THRESHOLD 10

// NRF Configuration
const uint64_t pipeIn = 0xF9E8F0F0E1LL;
const uint64_t pipeOut = 0xF9E8F0F0E2LL;
RF24 radio(8, 9);

unsigned long lastRecvTime = 0;
int obstacleCount = 0;
bool emergencyStopped = false;
bool signalConnected = false;
unsigned long systemStartTime = 0;
float currentSpeed = 0;
byte systemStatus = 0;
unsigned long lastSpeedUpdate = 0;
unsigned long lastBatteryCheck = 0;
```

```
byte simulatedBatteryLevel = 100;

struct PacketData {
byte xAxisValue;
byte yAxisValue;
byte speedCommand;
byte commandFlags;
};
PacketData receiverData;

struct FeedbackData {
bool obstacleDetected;
byte batteryLevel;
bool isEmergencyStopped;
float currentSpeed;
unsigned long uptime;
byte systemStatus;
} carStatus;

// Motor Pins
const int enableRightMotor = 5;
const int rightMotorPin1 = 2;
const int rightMotorPin2 = 3;

const int enableLeftMotor = 6;
const int leftMotorPin1 = 4;
const int leftMotorPin2 = 7;

// Ultrasonic + Servo
const int trigPin = A1;
const int echoPin = A2;
const int servoPin = A0;

Servo sensorServo;


// Traffic Light Pins

const int greenLED = A3;

const int yellowLED = A4;

const int redLED = A5;


// Motor speed tracking

int currentLeftSpeed = 0;

int currentRightSpeed = 0;


// Function prototypes

void rotateMotor(int rightMotorSpeed, int leftMotorSpeed);

float detectObstacle();

int scanSide(int angle);

void blinkRed();

void blinkYellow();

void greenLight();

void redLight();

void updateLED(String state);


void setup() {

Serial.begin(115200);

pinMode(BATTERY_PIN, INPUT);

pinMode(enableRightMotor, OUTPUT);

pinMode(rightMotorPin1, OUTPUT);

pinMode(rightMotorPin2, OUTPUT);

pinMode(enableLeftMotor, OUTPUT);

pinMode(leftMotorPin1, OUTPUT);

pinMode(leftMotorPin2, OUTPUT);
pinMode(trigPin, OUTPUT);
pinMode(echoPin, INPUT);

pinMode(greenLED, OUTPUT);
pinMode(yellowLED, OUTPUT);
pinMode(redLED, OUTPUT);

sensorServo.attach(servoPin);
sensorServo.write(90);

if (!radio.begin()) {
Serial.println("❌ NRF24L01 not responding. Check wiring.");
while (1);
}

radio.setPALevel(RF24_PA_LOW);
radio.setDataRate(RF24_250KBPS);
radio.setChannel(108);
radio.enableAckPayload();
radio.openReadingPipe(1, pipeIn);
radio.openWritingPipe(pipeOut);
radio.startListening();

systemStartTime = millis();
Serial.println("✅ Car listening for commands...");
}

void loop() {
static bool signalLost = true;
static bool obstacleHandled = false; // Add this to track if we've handled the current obstacle
static unsigned long lastObstacleTime = 0; // For debouncing
// Update system uptime
carStatus.uptime = (millis() - systemStartTime) / 1000;
if (radio.available()) {
radio.read(&receiverData, sizeof(receiverData));
lastRecvTime = millis();
signalLost = false;

// Get the current distance
float obstacleDistance = detectObstacle();
// Prepare feedback data
carStatus.obstacleDetected = (obstacleDistance > 0 && obstacleDistance < OBSTACLE_DISTANCE_CM);
carStatus.isEmergencyStopped = emergencyStopped;
carStatus.systemStatus = systemStatus;
carStatus.batteryLevel = readBatteryLevel();
// Calculate current speed
if (millis() - lastSpeedUpdate > 200) {
float avgSpeed = (abs(currentLeftSpeed) + abs(currentRightSpeed)) / 2.0;
currentSpeed = avgSpeed * SPEED_CALIBRATION;
carStatus.currentSpeed = currentSpeed;
lastSpeedUpdate = millis();
}

// Send feedback
radio.stopListening();
radio.write(&carStatus, sizeof(carStatus));
radio.startListening();
```

```
// Handle emergency stop
if (emergencyStopped) {
if (carStatus.batteryLevel > 10 && (receiverData.commandFlags & 0b00000001))
{
emergencyStopped = false;
obstacleCount = 0;
obstacleHandled = false; // Reset obstacle handling state
}
return;
}

// Handle stop command
if (receiverData.commandFlags & 0b00000001) {
rotateMotor(0, 0);
updateLED("red");
return;
}
// Obstacle detection with debouncing
if (!emergencyStopped && obstacleDistance > 0 && obstacleDistance <
OBSTACLE_DISTANCE_CM && !obstacleHandled &&
(millis() - lastObstacleTime > 1000)) { // 1 second debounce
obstacleCount++;
obstacleHandled = true; // Mark that we've handled this obstacle
lastObstacleTime = millis();
rotateMotor(0, 0);
// Handle obstacle detection
if (obstacleCount == 1) {
updateLED("yellow");
delay(300);
unsigned long turnStart = millis();
while (millis() - turnStart < TURN_DURATION) {
```

```
updateLED("yellow_blink");
rotateMotor(-MAX_MOTOR_SPEED, MAX_MOTOR_SPEED);
}
rotateMotor(0, 0);
sensorServo.write(90);
} else if(obstacleCount == 2) {
updateLED("red_blink");
rotateMotor(0, 0);
emergencyStopped = true;
delay(EMERGENCY_PAUSE);
rotateMotor(-MAX_MOTOR_SPEED, -MAX_MOTOR_SPEED);
delay(600);
rotateMotor(0, 0);
obstacleCount = 0;
emergencyStopped = false;
sensorServo.write(90);
}
}
// Reset obstacle handled state when no obstacle is detected
else if (obstacleDistance >= OBSTACLE_DISTANCE_CM || obstacleDistance <= 0) {
obstacleHandled = false;
}
// Normal driving when no obstacles
if (!emergencyStopped && !obstacleHandled) {
if (receiverData.yAxisValue >= 175) {
rotateMotor(MAX_MOTOR_SPEED, MAX_MOTOR_SPEED); //FORWARD
updateLED("right");
} else if (receiverData.yAxisValue <= 75) {
rotateMotor(-MAX_MOTOR_SPEED, -MAX_MOTOR_SPEED); //BACKWARD
updateLED("left");
} else if (receiverData.xAxisValue >= 175) {
```

```
rotateMotor(-MAX_MOTOR_SPEED, MAX_MOTOR_SPEED); //RIGHT
updateLED("green");
} else if (receiverData.xAxisValue <= 75) {
rotateMotor(MAX_MOTOR_SPEED, -MAX_MOTOR_SPEED); //LEFT
updateLED("green");
} else {
rotateMotor(0, 0);
updateLED("red");
}
}
}
else if (millis() - lastRecvTime > SIGNAL_TIMEOUT) {
rotateMotor(0, 0);
blinkRed();
signalLost = true;
}
}

void rotateMotor(int rightMotorSpeed, int leftMotorSpeed) {
rightMotorSpeed = constrain(rightMotorSpeed, -255, 255);
leftMotorSpeed = constrain(leftMotorSpeed, -255, 255);
currentLeftSpeed = leftMotorSpeed;
currentRightSpeed = rightMotorSpeed;
digitalWrite(rightMotorPin1, rightMotorSpeed > 0);
digitalWrite(rightMotorPin2, rightMotorSpeed < 0);
analogWrite(enableRightMotor, abs(rightMotorSpeed));
digitalWrite(leftMotorPin1, leftMotorSpeed > 0);
digitalWrite(leftMotorPin2, leftMotorSpeed < 0);
analogWrite(enableLeftMotor, abs(leftMotorSpeed));
if ((rightMotorSpeed != 0 || leftMotorSpeed != 0) &&
(abs(rightMotorSpeed - leftMotorSpeed) > 100)) {
systemStatus |= 0b00000001;
} else {
systemStatus &= ~0b00000001;
}
}

// Change the function to return distance instead of boolean
float detectObstacle() {
digitalWrite(trigPin, LOW); delayMicroseconds(2);
digitalWrite(trigPin, HIGH); delayMicroseconds(10);
digitalWrite(trigPin, LOW);
long duration = pulseIn(echoPin, HIGH, 30000);
float distance = duration * 0.034 / 2;
return distance; // Return the actual distance
}


int scanSide(int angle) {
sensorServo.write(angle); // Move servo to specified angle
delay(250); // Wait for servo to settle
// Take 3 readings for reliability
float totalDistance = 0;
int validReadings = 0;
for(int i = 0; i < 3; i++) {
digitalWrite(trigPin, LOW);
delayMicroseconds(2);
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);
digitalWrite(trigPin, LOW);
long duration = pulseIn(echoPin, HIGH, 30000);
```

```
if(duration > 0) {

totalDistance += duration * 0.034 / 2;

validReadings++;

}

delay(50);

}

return validReadings > 0 ? (totalDistance / validReadings) : 999; // Return
average or large number if no valid readings

}


void blinkRed() {

digitalWrite(redLED, millis() % 500 < 250);

digitalWrite(greenLED, LOW);

digitalWrite(yellowLED, LOW);

}


void blinkYellow() {

digitalWrite(yellowLED, millis() % 300 < 150);

digitalWrite(greenLED, LOW);

digitalWrite(redLED, LOW);

}


void greenLight() {

digitalWrite(greenLED, HIGH);

digitalWrite(redLED, LOW);

digitalWrite(yellowLED, LOW);

}


void redLight() {

digitalWrite(redLED, HIGH);

digitalWrite(greenLED, LOW);
digitalWrite(yellowLED, LOW);
}

byte readBatteryLevel() {
int rawValue = analogRead(BATTERY_PIN);
float voltage = rawValue * (REFERENCE_VOLTAGE / 1023.0);
// Protect against over-voltage just in case
voltage = constrain(voltage, 0, REFERENCE_VOLTAGE);
// Convert voltage to percentage (simple linear approximation)
byte level = map(voltage * 100,
EMPTY_BATTERY_VOLTAGE * 100,
FULL_BATTERY_VOLTAGE * 100,
0, 100);
return constrain(level, 0, 100); // Ensure we stay within 0-100%
}
void updateLED(String state) {
if (state == "red") {
digitalWrite(redLED, HIGH);
digitalWrite(yellowLED, LOW);
digitalWrite(greenLED, LOW);
}
else if (state == "yellow") {
digitalWrite(redLED, LOW);
digitalWrite(yellowLED, HIGH);
digitalWrite(greenLED, LOW);
}
else if (state == "green") {
digitalWrite(redLED, LOW);
digitalWrite(yellowLED, LOW);
digitalWrite(greenLED, HIGH);
}
```

```
else if (state == "right") {

digitalWrite(redLED, HIGH);

delay(100); // 100ms is about the maximum you should use

digitalWrite(redLED, LOW);

digitalWrite(yellowLED, HIGH);

delay(100);

digitalWrite(yellowLED, LOW);

digitalWrite(greenLED, HIGH);

delay(100);

digitalWrite(greenLED, LOW);

}

else if (state == "left") {

digitalWrite(redLED, HIGH);

delay(100); // 100ms is about the maximum you should use

digitalWrite(redLED, LOW);

digitalWrite(yellowLED, HIGH);

delay(100);

digitalWrite(yellowLED, LOW);

digitalWrite(greenLED, HIGH);

delay(100);

digitalWrite(greenLED, LOW);

}

else if (state == "red_blink") {

digitalWrite(redLED, millis() % 1000 < 500);

digitalWrite(yellowLED, LOW);

digitalWrite(greenLED, LOW);

}

else if (state == "yellow_blink") {

digitalWrite(redLED, LOW);

digitalWrite(yellowLED, millis() % 1000 < 500);

digitalWrite(greenLED,LOW);
}

}
```