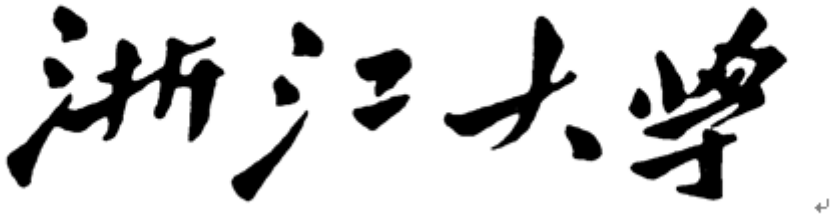


LoT Lab1:基于RIOT的基础设备控制实验

姓名：蔡佳伟

学号：3220104519

一、实验目的和要求

- 了解常用的硬件平台(esp32-wroom-32)
- 理解RIOT操作系统的多线程控制原理和方法
- 掌握基于GPIO引脚的LED灯控制技术
- 学习通过I2C总线读取IMU传感器数据的方法
- 识别设备运动状态并根据运动状态展示不同LED颜色的功能

二、实验内容

基于ESP32硬件以及RIOT系统，

根据实验手册(https://gitee.com/emnets/emnets_experiment/blob/master/part0_base.md)完成以下实验：

1. IMU传感器读取及LED控制实验

- 通过GPIO控制LED灯状态，展示不同颜色
- 读取IMU惯性传感器数据，并打印至串口
- 识别设备运动状态，根据运动状态展示不同灯颜色

三、实验背景

- RIOT操作系统

RIOT(https://github.com/RIOT-OS/RIOT) 是一个开源的微控制器操作系统，旨在满足物联网(IoT)设备和其他嵌入式设备的需求。它支持一系列通常在物联网(IoT)中发现的设备:8位，16位和32位微控制器。RIOT基于以下设计原则:节能、实时功能、内存占用小、模块化和统一的API访问，独立于底层硬件(该API提供部分POSIX遵从性)。

- MPU6050惯性传感器介绍

MPU6050是一款六轴运动处理传感器，它集成了3轴MEMS陀螺仪和3轴MEMS加速度计，以及一个可扩展的数字运动处理器DMP。这种传感器能够检测物体在空间中的姿态变化，包括俯仰角(Pitch)、翻滚角(Roll)和偏航角(Yaw)。MPU6050通过I2C接口读取数据，经过数据处理后，可以提供姿态角的精确测量。它广泛应用于平衡车、无人机、智能手环、手机等设备中，用于保持设备平衡和稳定，以及实现姿态控制 。MPU6050模块的引脚功能包括两组I2C信号线SDA/SCL和XDA/XCL，其中SDA/SCL用于与外部主机通讯，而XDA/XCL则用于MPU6050与其他I2C传感器通讯。模块的I2C设备地址可以通过AD0引脚的电平控制，提供灵活的地址配置。

四、主要仪器设备

- PC
- ESP32-WROOM-32、MPU6050惯性传感器、LED RGB灯。

五、实验题目简答

a ）列举生活中常见的一些传感器，其应用场景有哪些?

气体传感器（例如CO2传感器、甲烷传感器）

空气质量检测：用于检测室内外空气中的CO2浓度，应用于智能家居空气净化器或气象站

工业安全：部分危险气体传感器用于检测矿井中的可燃气体，防止爆炸、窒息风险等

骑车排放控制：用于监测尾气中的有害气体，例如CO2，NOx，确保汽车排放符合环保标准。

LIDAR（光探测和测距传感器）

自动驾驶汽车：LIDAR用于创建高精度的3D环境地图，帮助自动驾驶车辆识别周围的物体并导航。

无人机：用于测绘、地形测量和避障，在复杂环境中提供精准的高度和距离感知。

机器人：用于SLAM（即时定位与地图构建），帮助机器人在不熟悉的环境中自主移动。

心率传感器（光电容积脉搏波描记法，PPG）

智能手表和健身手环：通过手腕测量脉搏，实时监测心率并提供运动和健康数据分析。

医疗设备：如心率监测仪，用于医院或家庭中的心脏健康监测，帮助医生随时了解患者的心脏情况。

运动训练设备：跑步机或自行车等设备中集成心率监控功能，帮助用户调整运动强度

霍尔传感器

电动车和电机：用于检测转速和位置，确保电机的精确控制。

智能手机：在一些手机中用于检测手机套的开关，从而自动唤醒或关闭屏幕。

汽车安全系统：用于车速传感器，检测车轮的转动速度，以辅助ABS（防抱死刹车系统）和车身稳定控制系统。

b ）MPU6050惯性传感器SDA和SCL引脚的作用是什么?请详细介绍I2C是如何通信以及如何通过GPIO控制LED灯?

SDA（Serial Data Line） 和 **SCL（Serial Clock Line）** 是 **I²C（Inter-Integrated Circuit）** 通信协议的两个核心信号线，用于在 MPU6050 传感器和微控制器（如 ESP32 或 Arduino）之间传输数据和时钟信号。

- **SDA（数据线）**：用于传输数据。该引脚负责在主机（microcontroller）和从设备（MPU6050）之间的双向数据传输。
- **SCL（时钟线）**：提供时钟信号，同步主机和从设备的数据传输。主设备生成时钟信号以控制数据传输的速率。

I²C 通信协议

I²C（Inter-Integrated Circuit） 是一种常用的串行通信协议，主要用于短距离的板上通信，如 MPU6050 传感器与微控制器之间的通信。I²C 是一种 **双线制协议**。通信由主设备发起，通常按以下步骤进行：

- **起始条件**：主设备将 SDA 拉低，然后拉低 SCL，标志着通信开始。
- **发送从设备地址**：主设备发送 7 位从设备地址（例如 MPU6050 的 `0x68`）和一个 **读/写位**，表明接下来是读操作还是写操作。
- **应答（ACK/NACK）**：从设备应答该地址，如果识别到自己的地址，它会发送一个 **ACK（应答）** 信号。
- **数据传输**：主设备和从设备之间的数据通过 SDA 线按字节传输，主设备控制时钟（SCL）以同步传输。
- **停止条件**：主设备发送一个停止条件，拉高 SDA 和 SCL，结束通信。

LED 通常由 GPIO 输出的高低电平来控制：

- **高电平**：LED 亮。
- **低电平**：LED 灭。

下面是一个通过 GPIO 控制 LED 的简单代码示例（以 ESP32 为例）：

```
#include <driver/gpio.h>

#define LED_PIN GPIO_NUM_2  // 定义 LED 的 GPIO 引脚

void app_main() {
    // 1. 配置 GPIO 引脚为输出模式
    gpio_pad_select_gpio(LED_PIN);
    gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);

    while (1) {
        // 2. 点亮 LED (设置 GPIO 引脚为高电平)
        gpio_set_level(LED_PIN, 1);
        vTaskDelay(1000 / portTICK_PERIOD_MS);  // 延时 1 秒

        // 3. 熄灭 LED (设置 GPIO 引脚为低电平)
        gpio_set_level(LED_PIN, 0);
        vTaskDelay(1000 / portTICK_PERIOD_MS);  // 延时 1 秒
    }
}
```

工作流程

- **配置 GPIO 引脚**：首先需要将 GPIO 引脚设置为 **输出模式。**
- **设置电平**：调用 `gpio_set_level()` 函数设置引脚的电平值，`1` 表示高电平（点亮 LED），`0` 表示低电平（熄灭 LED）。
- **延时和循环**：通过 `vTaskDelay()` 实现每秒切换 LED 的亮灭状态，形成闪烁效果。

c） RIOT系统如何创建线程？线程之间如何通信？列举出其他线程通信方法。

在 **RIOT** 操作系统中，创建线程的方式通常是通过调用 `thread_create()` 函数。这个函数用于创建一个新的线程并指定其堆栈大小、优先级、入口函数等。以下是创建线程的主要步骤和参数：

创建线程的函数：`thread_create()`

```
kernel_pid_t thread_create(
    char *stack,                // 线程的堆栈内存
    int stacksize,              // 堆栈大小
    int priority,               // 线程优先级
    int flags,                  // 线程标志位 (如调度策略等)
    thread_task_func_t func,    // 线程入口函数
    void *arg,                  // 传递给入口函数的参数
    const char *name            // 线程名称（调试用）
);
```

线程创建示例代码

```
#include <thread.h>
#include <stdio.h>

#define THREAD_STACKSIZE 1024  // 定义线程的堆栈大小

char thread_stack[THREAD_STACKSIZE];  // 为线程分配堆栈内存

// 线程入口函数
void *thread_func(void *arg) {
    while (1) {
        printf("This is the new thread\n");
        thread_yield();  // 让出 CPU，让其他线程运行
    }
    return NULL;
}

int main(void) {
    // 创建一个新线程
    kernel_pid_t thread_pid = thread_create(
        thread_stack,                // 堆栈指针
        sizeof(thread_stack),        // 堆栈大小
        THREAD_PRIORITY_MAIN - 1,    // 线程优先级（优先级较高）
        THREAD_CREATE_STACKTEST,     // 线程标志，创建堆栈测试
        thread_func,                 // 线程入口函数
        NULL,                        // 传递给线程的参数
        "my_thread"                  // 线程名称
```

```
    );

    // 检查线程创建是否成功
    if (thread_pid <= 0) {
        printf("Thread creation failed\n");
    } else {
        printf("Thread created, PID: %d\n", thread_pid);
    }

    return 0;
}
```

- `thread_create()` 返回一个 **PID**，表示线程的 ID。若返回值为正数，说明线程创建成功；若返回值为负数，则表示创建失败。
- 堆栈大小和优先级是创建线程时必须合理分配的资源。

---

2. RIOT 系统中的线程间通信

RIOT 提供了几种线程间通信的机制，最常见的是通过 **消息传递机制（Message Passing）**，使用 `msg` 结构体在线程之间传递消息。

`msg` 结构体

```
typedef struct {
    uint32_t type;       // 消息类型
    union {
        void *ptr;       // 数据指针
        int value;       // 整数值
    } content;           // 消息内容
    kernel_pid_t sender_pid;  // 发送者线程的 PID
} msg_t;
```

消息传递函数

- **发送消息：** `msg_send()`
  将消息发送给目标线程，目标线程会通过 `msg_receive()` 获取消息。

  ```
  int msg_send(msg_t *msg, kernel_pid_t target);
  ```

- **接收消息：** `msg_receive()`
  阻塞接收线程的消息，直到消息到达。

  ```
  void msg_receive(msg_t *msg);
  ```

线程通信示例

```
#include <thread.h>
#include <msg.h>
#include <stdio.h>

#define THREAD_STACKSIZE 1024
```

```
char thread_stack[THREAD_STACKSIZE];
msg_t msg;

void *receiver_thread(void *arg) {
    while (1) {
        msg_receive(&msg);  // 阻塞等待消息
        printf("Received message with value: %d\n", msg.content.value);
    }
    return NULL;
}

int main(void) {
    // 创建一个接收消息的线程
    kernel_pid_t thread_pid = thread_create(thread_stack, sizeof(thread_stack),
THREAD_PRIORITY_MAIN - 1, 0, receiver_thread, NULL, "receiver_thread");

    if (thread_pid <= 0) {
        printf("Thread creation failed\n");
        return -1;
    }

    // 向接收线程发送消息
    msg_t msg;
    msg.content.value = 42;  // 发送消息中的数据
    msg_send(&msg, thread_pid);

    return 0;
}
```

- `msg_receive()` 阻塞当前线程，直到收到其他线程通过 `msg_send()` 发送的消息。
- 消息中可以传递简单的整数值或指针等数据。

---

3. 其他线程通信方法

除了消息传递，RIOT 系统还支持其他一些常用的线程间通信机制：

1. 互斥锁（Mutex）

- **互斥锁** 用于确保多个线程不会同时访问共享资源。RIOT 提供 `mutex_t` 结构体和相关的锁机制。
- 示例：

```
mutex_t my_mutex;
mutex_init(&my_mutex);
mutex_lock(&my_mutex);   // 锁定资源
// 执行临界区代码
mutex_unlock(&my_mutex); // 释放锁
```

2. 信号量（Semaphore）

- **信号量** 是用于控制线程同步和资源访问的计数器，RIOT 提供 `sema_t` 结构体来实现。
- 示例：

```
sema_t my_semaphore;
sema_create(&my_semaphore, 1);  // 初始化信号量

sema_wait(&my_semaphore);  // 等待信号量
// 执行临界区代码
sema_post(&my_semaphore);  // 释放信号量
```

3. 队列（Message Queue）

- **消息队列** 用于在生产者和消费者线程之间传递消息，保证线程之间的数据交换有序且安全。
- `msg_queue` 是 RIOT 提供的一种基于消息的队列，用于线程通信。
- 使用 `msg_queue_send()` 和 `msg_queue_receive()` 函数来实现。

4. 条件变量（Condition Variable）

- **条件变量** 通常与互斥锁一起使用，允许线程等待某个条件发生后继续执行。
- 当某个条件满足时，可以通过条件变量唤醒等待中的线程。

5. 信号（Signal）

- **信号** 是一种用于通知线程某个事件发生的机制。一个线程可以向另一个线程发送信号，触发其执行某个操作。信号通常是异步的。

六、实验数据记录和处理

1. 基础设备控制实验

0）常用指令

`ls /dev/tty*` 查看tty接口(接口名)

`sudo ln -s /dev/ttyUSB1 /dev/ttyUSB0` 把USB1暂时绑到USB0上，用于找不到USB0的报错同时查看接口名是ttyUSB1

`esp_idf all` 一个自定义复合命令，用来初始化 ESP-IDF 环境、设置工具链等，每次连接设备后都要使用

`make BOARD=esp32-wroom-32 flash term` 用于编译项目、烧录到开发板并监控串口输出

1）代码截图

1. ledcontroller.cpp

该函数用于灯光赋值和初始化

home > cjw > RIOT > examples > emnets_experiment > 03_threads_led_and_imu_experiment > ledcontroller.cpp > change_led_color(uint8_t)

```cpp
#include "ztimer.h"
#include "ledcontroller.hh"
#include "periph/gpio.h"
#include <stdio.h>
/**
 * Initialize the RGB pins as output mode, and initially, the lights should be off.
 * gpio_init(gpio_pin, GPIO_OUT);
 * gpio_write(pin, 0);
 */
LEDController::LEDController(uint8_t gpio_r, uint8_t gpio_g, uint8_t gpio_b){
    printf("LED Controller initialized with (RGB: GPIO%d, GPIO%d, GPIO%d)\n", gpio_r, gpio_g, gpio_b);
    // input your code
    led_gpio[0] = gpio_r;
    led_gpio[1] = gpio_g;
    led_gpio[2] = gpio_b;
    for(int i=0;i<3;i++){
        gpio_init(led_gpio[i],GPIO_OUT); // init all gpio GPIO_OUT
        gpio_write(led_gpio[i],0); // close all led default
    }
}
```

该函数用于定义LED灯光颜色和LED状态

```cpp
void LEDController::change_led_color(uint8_t color){
    // input your code
    switch (color)
    {
    case COLOR_NONE:
        gpio_write(led_gpio[0],0); //close red
        gpio_write(led_gpio[1],0); //close green
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_RED:
        gpio_write(led_gpio[0],1); //open red
        gpio_write(led_gpio[1],0); //close green
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_GREEN:
        gpio_write(led_gpio[0],0); //close red
        gpio_write(led_gpio[1],1); //open green
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_YELLO:
        gpio_write(led_gpio[0],1); //open red
        gpio_write(led_gpio[1],1); //open green
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_BLUE:
        gpio_write(led_gpio[0],0); //close red
        gpio_write(led_gpio[1],0); //close green
        gpio_write(led_gpio[2],1); //open blue
        break;
    case COLOR_MAGENTA:
```

```
            gpio_write(led_gpio[0],1); //open red
            gpio_write(led_gpio[1],0); //close green
            gpio_write(led_gpio[2],1); //open blue
            break;
        case COLOR_CYAN:
            gpio_write(led_gpio[0],0); //close red
            gpio_write(led_gpio[1],1); //open green
            gpio_write(led_gpio[2],1); //open blue
            break;
        default:
            break;
        }
    }
```

2. **main.cpp**

该函数对应传感器获取的运动状态和LED灯光状态，从而使得传感器运动可以控制灯光颜色的变化，不同的状态使灯光显示不同的颜色

```
void *_led_thread(void *arg)
{
    (void) arg;
    LEDController led(LED_GPIO_R, LED_GPIO_G, LED_GPIO_B);
    led.change_led_color(0);
    while(1){
        // Input your codes
        // Wait for a message to control the LED
        // Display different light colors based on the motion state of the
        msg_t msg;
        msg_receive(&msg);
        if(msg.content.value==Stationary){
            led.change_led_color(COLOR_NONE);
        }
        else if(msg.content.value==Moving_X){
            led.change_led_color(COLOR_CYAN);
        }
        else if(msg.content.value==Moving_Y){
            led.change_led_color(COLOR_WHITE);
        }
        else if(msg.content.value==Tilted){
            led.change_led_color(COLOR_RED);
        }
        else if(msg.content.value==Rotating){
            led.change_led_color(COLOR_BLUE);
        }
        else if(msg.content.value==Shaking){
            led.change_led_color(COLOR_YELLO);
        }
        else if(msg.content.value==Moving){
            led.change_led_color(COLOR_GREEN);
        }
    }
    return NULL;
}
```

detectmovement模块，使用传感器获取的数据进行状态判定，参数的数值不同从而判定不同的运动状态

其中判定所用参数如下

```
#define g_acc (9.8)
#define ACC_THRESHOLD (1.5) // acceleration
#define ROT_THRESHOLD (50)
```

```
MoveState detectMovement(MPU6050Data &data)
{
    // Input your code
    // Please use your imagination or search online
    // to determine the motion state of the device
    // based on the data obtained from the MPU6050 sensor.
    float acceleration_x=fabs(data.ax);     // calculate acceleration
    float acceleration_y=fabs(data.ay);

    float total_acceleration=sqrt(data.ax*data.ax+data.ay*data.ay+data.az*dat
    float total_rotation=sqrt(data.gx*data.gx+data.gy*data.gy+data.gz*data.gz

    if(fabs(total_acceleration-g_acc)<ACC_THRESHOLD*3&&acceleration_x<ACC_THR
    <ACC_THRESHOLD*3&&total_rotation<ROT_THRESHOLD/2){
        return Stationary; // z_acceleration-g x y ->0 and no rotation
    }
    else if(fabs(total_acceleration-g_acc)<ACC_THRESHOLD*3&&total_rotation<RO
        return Tilted; // z_acceleration-g ->0 and no rotation
    }
    else if(acceleration_x>ACC_THRESHOLD&&acceleration_y<ACC_THRESHOLD&&total
        return Moving_X; // x moving y no moving and no rotation
    }
    else if(acceleration_x<ACC_THRESHOLD&&acceleration_y>ACC_THRESHOLD&&total
        return Moving_Y; // x no moving y moving and no rotation
    }
    else if(total_rotation>ROT_THRESHOLD){
        return Rotating;
    }
    else if(fabs(total_acceleration-g_acc)>ACC_THRESHOLD){
        return Moving;
    }
    return Stationary;
}
```

imu线程函数，处理获得的传感器数据，换算成物理数据，以便于状态判定，需要先获取传感器数据，再转换，不同量程范围的陀螺仪输出值需要除以不同的转换因子来获得实际的角速度，加速度也是同理。

```c
void *_imu_thread(void *arg)
{
    (void) arg;
    // Input your code
    // 1. initial mpu6050 sensor
    // 2. Acquire sensor data every 100ms
    // 3. Determine the motion state
    // 4. notify the LED thread to display the light color through a message.
    MPU6050 mpu;
    mpu.initialize();
    uint8_t gyro_fs=mpu.getFullScaleGyroRange();
    uint8_t accel_fs_g=mpu.getFullScaleAccelRange(); // detemine the motion sta
    float gyro_fs_convert=1.0;
    float accel_fs_real=1.0;
    switch(gyro_fs){
        case MPU6050_GYRO_FS_250:
            gyro_fs_convert=131.0;
            break;
        case MPU6050_GYRO_FS_500:
            gyro_fs_convert=65.5;
            break;
        case MPU6050_GYRO_FS_1000:
            gyro_fs_convert=32.8;
            break;
        case MPU6050_GYRO_FS_2000:
            gyro_fs_convert=16.4;
            break;
        default:
            printf("{IMU_THREAD} unknown GYRO_FS: 0x%x\n",gyro_fs);
            break;
    }

    switch (accel_fs_g)
    {
    case MPU6050_ACCEL_FS_2:
        accel_fs_real=g_acc*2;
        break;
    case MPU6050_ACCEL_FS_4:
        accel_fs_real=g_acc*4;
        break;
    case MPU6050_ACCEL_FS_8:
        accel_fs_real=g_acc*8;
        break;
    case MPU6050_ACCEL_FS_16:
        accel_fs_real=g_acc*16;
        break;
    default:
        printf("{IMU_THREAD} unknown ACCEL_FS: 0x%x\n",accel_fs_g);
        break;
    }

    float accel_fs_convert=32768.0/accel_fs_real;
    int16_t ax,ay,az,gx,gy,gz;

    delay_ms(1000);
```

其中imu线程主循环用于周期性读取传感器数据并输出，通过调用 `detectMovement(data)` 函数，代码分析当前的传感器数据，判断设备的运动状态，根据检测到的运动状态，代码通过消息机制将运动状态传递给 LED 线程，以改变 LED 的显示颜色，每次采集数据之后，代码会等待 200 毫秒再进行下一次数据采集。

```
while(1){
    mpu.getMotion6(&ax,&ay,&az,&gx,&gy,&gz);
    MPU6050Data data;
    data.ax=ax/accel_fs_convert;
    data.ay=ay/accel_fs_convert;
    data.az=az/accel_fs_convert;
    data.gx=gx/gyro_fs_convert;
    data.gy=gy/gyro_fs_convert;
    data.gz=gz/gyro_fs_convert;

    printf("------------------------\n");
    printf("{IMU_THREAD} Accelerometer (X,Y,Z): (%.02f,%.02f,%.02f) m/s^2\n
    printf("{IMU_THREAD} Gyroscope (X,Y,Z): (%.02f,%.02f,%.02f) 0/s\n",data
    MoveState state=detectMovement(data);
    msg_t msg;
    msg.content.value=(int)state;
    msg_send(&msg,_led_pid);

    delay_ms(200);
}

return NULL;
```

2）串口惯性传感器数据及运动状态实验结果截图

Stationary：Z轴加速度接近g，X,Y轴加速度小于阈值，旋转角速度很小。（没有截到合适的）

Moving_X: X轴加速度

```
2024-09-25 21:14:10,259 # {IMU_THREAD} Accelerometer (X,Y,Z): (-1.81,0.02,10.
86) m/s^2
2024-09-25 21:14:10,266 # {IMU_THREAD} Gyroscope (X,Y,Z): (6.71,0.79,2.38) 0/
s
2024-09-25 21:14:10,457 # ------------------------
2024-09-25 21:14:10,461 # {IMU_THREAD} Accelerometer (X,Y,Z): (-1.69,0.04,10.
81) m/s^2
2024-09-25 21:14:10,465 # {IMU_THREAD} Gyroscope (X,Y,Z): (6.65,0.73,2.38) 0/
s
```

Moving_Y: X轴加速度小于阈值，Y轴加速度大于阈值，且没有旋转

```
2024-09-25 21:13:04,402 # ------------------------
2024-09-25 21:13:04,407 # {IMU_THREAD} Accelerometer (X,Y,Z): (-1.44,2.89,9.9
2) m/s^2
2024-09-25 21:13:04,411 # {IMU_THREAD} Gyroscope (X,Y,Z): (7.07,-9.39,-16.16)
 0/s
```

Tilted: 输出数据中，设备的加速度在X、Y、Z轴上较为稳定，且没有旋转

```
2024-09-25 21:13:49,462 # {IMU_THREAD} Gyroscope (X,Y,Z): (6.46,0.61,2.38) 0/
s
2024-09-25 21:13:49,655 # ------------------------
2024-09-25 21:13:49,660 # {IMU_THREAD} Accelerometer (X,Y,Z): (-1.71,0.04,10.
84) m/s^2
2024-09-25 21:13:49,664 # {IMU_THREAD} Gyroscope (X,Y,Z): (6.65,0.55,2.56) 0/
s
2024-09-25 21:13:49,856 # ------------------------
2024-09-25 21:13:49,861 # {IMU_THREAD} Accelerometer (X,Y,Z): (-1.75,-0.04,10
.75) m/s^2
2024-09-25 21:13:49,866 # {IMU_THREAD} Gyroscope (X,Y,Z): (6.52,0.67,2.50) 0/
s
2024-09-25 21:13:50,058 # ------------------------
2024-09-25 21:13:50,063 # {IMU_THREAD} Accelerometer (X,Y,Z): (-1.73,0.05,10.
87) m/s^2
2024-09-25 21:13:50,067 # {IMU_THREAD} Gyroscope (X,Y,Z): (6.59,0.67,2.50) 0/
s
2024-09-25 21:13:50,261 # ------------------------
2024-09-25 21:13:50,265 # {IMU_THREAD} Accelerometer (X,Y,Z): (-1.73,-0.03,10
.86) m/s^2
```

Rotating：输出数据中，设备的总角速度大于阈值

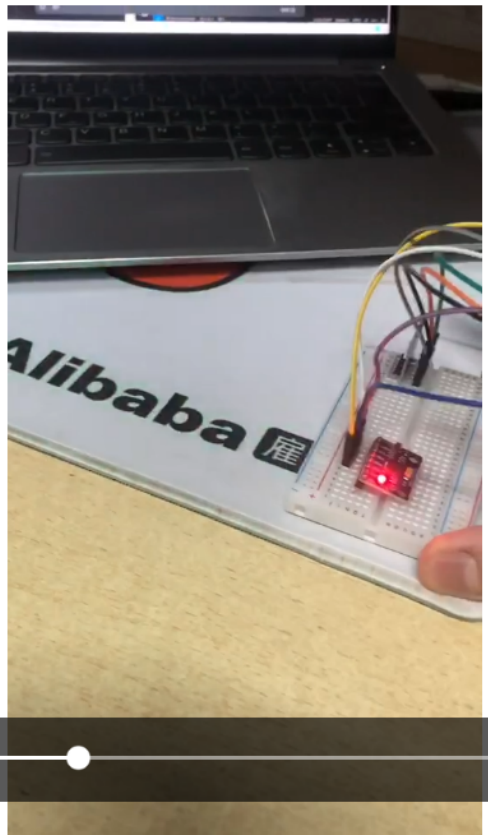
```
2024-09-25 21:13:42,793 # {IMU_THREAD} Accelerometer (X,Y,Z): (-5.27,-0.52,6.68)
 m/s^2
2024-09-25 21:13:42,798 # {IMU_THREAD} Gyroscope (X,Y,Z): (-42.26,-89.09,-77.13)
 0/s
2024-09-25 21:13:42,991 # ------------------------
2024-09-25 21:13:42,995 # {IMU_THREAD} Accelerometer (X,Y,Z): (-4.59,-0.09,16.83
) m/s^2
2024-09-25 21:13:43,000 # {IMU_THREAD} Gyroscope (X,Y,Z): (21.52,-13.54,-120.73)
 0/s
```

Moving: 输出数据中，设备的总加速度明显偏离重力加速度

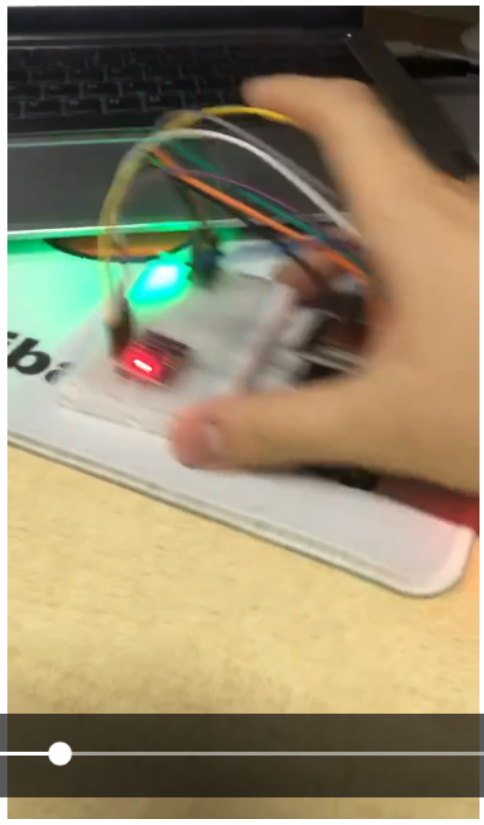
```
2024-09-25 21:12:51,672 # ------------------------
2024-09-25 21:12:51,678 # {IMU_THREAD} Accelerometer (X,Y,Z): (-4.28,-4.08,9.
73) m/s^2
2024-09-25 21:12:51,683 # {IMU_THREAD} Gyroscope (X,Y,Z): (-226.40,-201.10,-8
4.70) 0/s
2024-09-25 21:12:51,874 # ------------------------
2024-09-25 21:12:51,880 # {IMU_THREAD} Accelerometer (X,Y,Z): (9.73,-19.60,3.
36) m/s^2
2024-09-25 21:12:51,884 # {IMU_THREAD} Gyroscope (X,Y,Z): (58.72,24.02,-18.54
) 0/s
```

3）LED灯颜色照片

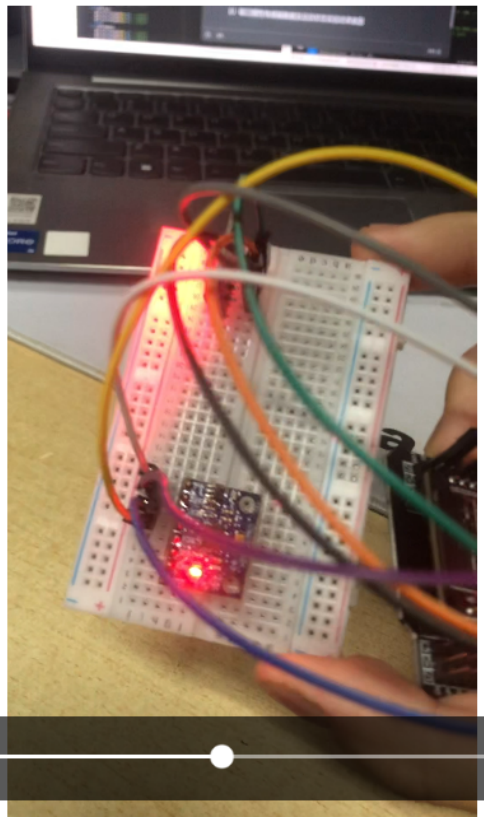
1 Stationary：None



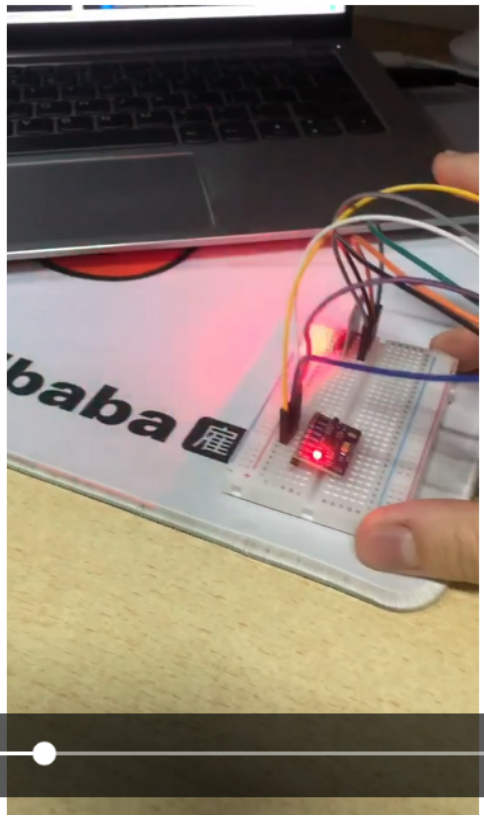
2 Moving_X：CYAN



3 Moving_Y：Yellow

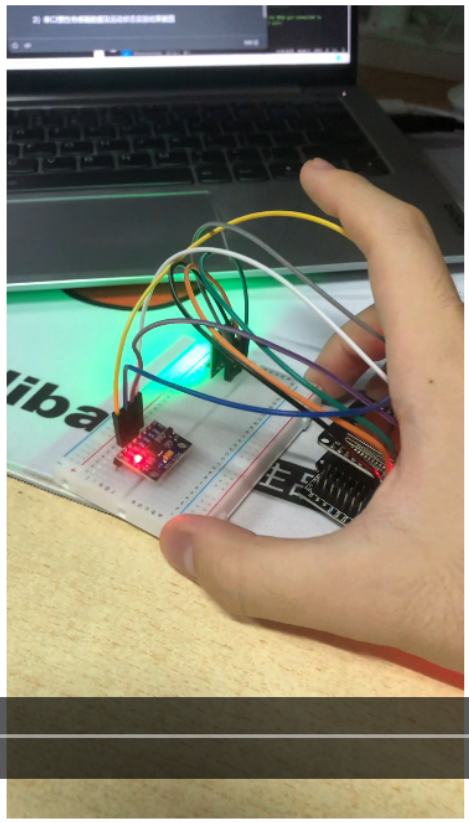


4 Tilted：Red



5 Rotating：Blue

6 Moving：Green



七、实验结果与分析

- 基于该实验，总结开发一个完整的传感器应用基本流程

1. 硬件初始化

(1) 初始化传感器硬件接口

- **选择传感器接口**：确定传感器的通信接口，如 **I²C** 或 **SPI**。通过配置相应的硬件引脚（如 SDA 和 SCL）来初始化通信总线。
- **初始化通信协议**：调用操作系统中的通信接口库函数（如 I²C 或 SPI），进行设备初始化。

示例：I²C 初始化

```
i2c_init(I2C_DEV(0));  // 初始化 I²C 总线设备 0
```

(2) 配置传感器

- 设置传感器的工作模式、数据输出格式、采样频率等。例如，可以通过向传感器的寄存器写入配置数据来完成。

示例：配置 MPU6050 传感器

```
uint8_t config_data = 0x01;  // 假设 0x01 是某个配置
i2c_write_reg(I2C_DEV(0), MPU6050_ADDR, CONFIG_REG, config_data);
```

---

2. 创建线程

- **数据采集线程**：创建一个线程来周期性读取传感器数据。这是传感器应用的核心线程，负责从传感器采集原始数据。
- **数据处理线程（可选）**：如果需要对采集的数据进行处理或分析，可以创建一个专门的线程用于数据处理。

示例：创建数据采集线程

```
char thread_stack[THREAD_STACKSIZE_DEFAULT];

void *sensor_read_thread(void *arg) {
    while (1) {
        read_sensor_data();  // 读取传感器数据
        thread_sleep();      // 等待下一次采样
    }
}

kernel_pid_t pid = thread_create(thread_stack, sizeof(thread_stack),
THREAD_PRIORITY_MAIN - 1, 0, sensor_read_thread, NULL, "sensor_reader");
```

---

3. 读取传感器数据

(1) 定期采集传感器数据

- 根据传感器的采样频率或应用需求，定期通过接口（如 I²C）读取传感器数据。
- 采集的数据通常为原始传感器值，需要后续处理才能使用。

示例：读取 MPU6050 数据

```
int16_t acc_data[3];  // 用于存储加速度数据

void read_sensor_data(void) {
    // 假设 0x3B 是 MPU6050 加速度数据寄存器的地址
    i2c_read_regs(I2C_DEV(0), MPU6050_ADDR, 0x3B, (uint8_t *)acc_data,
sizeof(acc_data));
}
```

(2) 处理传感器数据

- 对原始数据进行必要的转换或过滤。例如，将从传感器读取的原始二进制数据转换成可用的物理量（如温度、加速度、角速度等）。
- 可以对数据进行单位转换、校准、滤波等处理。

---

4. 线程间通信

- 当有多个线程负责处理数据时，线程间的通信机制很重要。可以使用消息传递、互斥锁、信号量等机制来同步数据。
- 常见的做法是，采集线程将数据发送给处理线程。

示例：线程间通信通过消息传递

```
msg_t msg;
msg.content.value = (int)value;  // 将传感器数据打包到消息中
msg_send(&msg, processing_thread_pid);  // 发送消息到数据处理线程
```

---

5. 数据处理

- **在处理线程中处理数据**：接收从采集线程发送的传感器数据，并进行进一步的分析或处理。
- **数据处理操作**：可能包括数据平滑、检测异常、计算均值、或将数据转换为物理量。
- **数据存储或输出**：处理完成后，数据可能需要存储到非易失性存储器或通过通信接口传输到其他设备。

示例：接收消息并处理数据

```
void *data_processing_thread(void *arg) {
    msg_t msg;
    while (1) {
        msg_receive(&msg);  // 接收消息
        int processed_data = process_data(msg.content.value);  // 处理数据
        display_data(processed_data);  // 显示或存储数据
    }
}
```

---

6. 数据展示与输出

- **数据输出**：处理后的数据可以输出到串口、显示器、无线通信模块（如 LoRa、WiFi 等），或其他外部系统。
- **显示数据**：如果设备配有显示屏，可以实时显示传感器数据。对于调试，可以使用串口输出。

示例：通过串口输出传感器数据

```
printf("Processed sensor data: %d\n", processed_data);
```

---

7. 错误处理与恢复机制

- **错误检测**：开发过程中需要考虑传感器可能发生的错误，如通信失败、数据超出范围等。
- **错误处理**：在通信失败时，可以加入重试机制或错误提示；当传感器数据异常时，进行合理的恢复操作。

示例：I²C 通信错误处理

```
if (i2c_read_reg(I2C_DEV(0), MPU6050_ADDR, DATA_REG, &data) != 0) {
    printf("Error reading sensor data\n");
}
```

---

8. 应用优化

- **功耗优化**：对于嵌入式应用，功耗非常重要。可以通过控制传感器的休眠模式，或者通过线程睡眠节省功耗。
- **性能优化**：如果数据采集频率较高或需要处理大量数据，可以考虑通过优化线程调度、提高通信效率等方式提升应用的性能。

---

9. 总结：开发传感器应用的流程

1. **硬件初始化**：配置 I²C/SPI 等接口，初始化传感器硬件。
2. **创建线程**：为数据采集和处理创建相应的线程。
3. **传感器数据读取**：通过通信接口读取传感器的原始数据。
4. **数据处理**：对采集的数据进行转换、分析或过滤。
5. **线程间通信**：通过消息传递或其他机制同步不同线程之间的数据。
6. **数据展示与输出**：将处理后的数据通过串口、网络或显示器输出。
7. **错误处理**：设计合理的错误检测与恢复机制，确保应用的稳定性。
8. **应用优化**：通过功耗控制和性能优化，使应用更加高效。

通过这个流程，可以在 RIOT 操作系统上实现一个完整的传感器应用，从硬件初始化到数据采集、处理、输出，再到优化和稳定性提升。

八、附录

8.1 ledcontroller.cpp

```
#include "ztimer.h"
#include "ledcontroller.hh"
```

```
#include "periph/gpio.h"
#include <stdio.h>
/**
 * Initialize the RGB pins as output mode, and initially, the lights should be
off.
 * gpio_init(gpio_pin, GPIO_OUT);
 * gpio_write(pin, 0);
 */
LEDController::LEDController(uint8_t gpio_r, uint8_t gpio_g, uint8_t gpio_b){
    printf("LED Controller initialized with (RGB: GPIO%d, GPIO%d, GPIO%d)\n",
gpio_r, gpio_g, gpio_b);
    // input your code
    led_gpio[0] = gpio_r;
    led_gpio[1] = gpio_g;
    led_gpio[2] = gpio_b;
    for(int i=0;i<3;i++){
        gpio_init(led_gpio[i],GPIO_OUT); // init all gpio GPIO_OUT
        gpio_write(led_gpio[i],0); // close all led default
    }
}

/**
 * Implement a light that displays at least 5 status colors through the RGB
three pins.
 * ----------------------------------------------------------------
 * @note Method 1
 * Utilizes the gpio_write function to set the GPIO pin connected to the LED to
the current LED state.
 * void gpio_write(uint8_t pin, int value);
 * @param pin The GPIO pin connected to the LED.
 * @param value The value to set the GPIO pin to (0 for LOW, 1 for HIGH).
 * ----------------------------------------------------------------
 * @note Method 2
 * Uses the gpio_set and gpio_clear functions to set the GPIO pin connected to
the LED to the current LED state.
 * void gpio_set(uint8_t pin); void gpio_clear(uint8_t pin);
 * @param pin The GPIO pin connected to the LED.
 */
void LEDController::change_led_color(uint8_t color){
    // input your code
    switch (color)
    {
    case COLOR_NONE:
        gpio_write(led_gpio[0],0); //close red
        gpio_write(led_gpio[1],0); //close green
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_RED:
        gpio_write(led_gpio[0],1); //open red
        gpio_write(led_gpio[1],0); //close green
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_GREEN:
        gpio_write(led_gpio[0],0); //close red
        gpio_write(led_gpio[1],1); //open green
```

```
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_YELLO:
        gpio_write(led_gpio[0],1); //open red
        gpio_write(led_gpio[1],1); //open green
        gpio_write(led_gpio[2],0); //close blue
        break;
    case COLOR_BLUE:
        gpio_write(led_gpio[0],0); //close red
        gpio_write(led_gpio[1],0); //close green
        gpio_write(led_gpio[2],1); //open blue
        break;
    case COLOR_MAGENTA:
        gpio_write(led_gpio[0],1); //open red
        gpio_write(led_gpio[1],0); //close green
        gpio_write(led_gpio[2],1); //open blue
        break;
    case COLOR_CYAN:
        gpio_write(led_gpio[0],0); //close red
        gpio_write(led_gpio[1],1); //open green
        gpio_write(led_gpio[2],1); //open blue
        break;
    case COLOR_WHITE:
        gpio_write(led_gpio[0],1); //open red
        gpio_write(led_gpio[1],1); //open green
        gpio_write(led_gpio[2],1); //open blue
        break;
    default:
        break;
    }
}
```

8.2 main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <cmath>
#include <string>
#include <log.h>
#include <errno.h>
#include "clk.h"
#include "board.h"
#include "periph_conf.h"
#include "timex.h"
#include "ztimer.h"
#include "periph/gpio.h"
#include "thread.h"
#include "msg.h"
#include "shell.h"
// #include "xtimer.h"
#include "ledcontroller.hh"
#include "mpu6050.h"
```

```
#define THREAD_STACKSIZE        (THREAD_STACKSIZE_IDLE)
static char stack_for_led_thread[THREAD_STACKSIZE];
static char stack_for_imu_thread[THREAD_STACKSIZE];

static kernel_pid_t _led_pid;
#define LED_MSG_TYPE_ISR     (0x3456)
#define LED_GPIO_R GPIO26
#define LED_GPIO_G GPIO25
#define LED_GPIO_B GPIO27
struct MPU6050Data
{
    float ax, ay, az;
    float gx, gy, gz;
};
enum MoveState{Stationary, Tilted, Rotating, Moving, Shaking, Moving_X,
Moving_Y, None};

void delay_ms(uint32_t sleep_ms)
{
    ztimer_sleep(ZTIMER_USEC, sleep_ms * US_PER_MS);
    return;
}
/**
 * LED control thread function.
 * Then, it enters an infinite loop where it waits for messages to control the
LED.
 * @param arg Unused argument.
 * @return NULL.
 */
void *_led_thread(void *arg)
{
    (void) arg;
    LEDController led(LED_GPIO_R, LED_GPIO_G, LED_GPIO_B);
    led.change_led_color(0);
    while(1){
        // Input your codes
        // Wait for a message to control the LED
        // Display different light colors based on the motion state of the
device.
        msg_t msg;
        msg_receive(&msg);
        if(msg.content.value==Stationary){
            led.change_led_color(COLOR_NONE);
        }
        else if(msg.content.value==Moving_X){
            led.change_led_color(COLOR_CYAN);
        }
        else if(msg.content.value==Moving_Y){
            led.change_led_color(COLOR_YELLO);
        }
        else if(msg.content.value==Tilted){
            led.change_led_color(COLOR_RED);
        }
        else if(msg.content.value==Rotating){
            led.change_led_color(COLOR_BLUE);
```

```
        }
        else if(msg.content.value==Shaking){
            led.change_led_color(COLOR_WHITE);
        }
        else if(msg.content.value==Moving){
            led.change_led_color(COLOR_GREEN);
        }
    }
    return NULL;
}


#define g_acc (9.8)
#define ACC_THRESHOLD (1.5) // acceleration
#define ROT_THRESHOLD (50)

MoveState detectMovement(MPU6050Data &data)
{
    // Input your code
    // Please use your imagination or search online
    // to determine the motion state of the device
    // based on the data obtained from the MPU6050 sensor.
    float acceleration_x=fabs(data.ax);     // calculate acceleration
    float acceleration_y=fabs(data.ay);

    float
total_acceleration=sqrt(data.ax*data.ax+data.ay*data.ay+data.az*data.az);
    float total_rotation=sqrt(data.gx*data.gx+data.gy*data.gy+data.gz*data.gz);

    if(fabs(total_acceleration-g_acc)
<ACC_THRESHOLD*3&&acceleration_x<ACC_THRESHOLD*3&&acceleration_y
    <ACC_THRESHOLD*3&&total_rotation<ROT_THRESHOLD/2){
        return Stationary; // z_acceleration-g x y ->0 and no rotation
    }
    else if(fabs(total_acceleration-g_acc)
<ACC_THRESHOLD*3&&total_rotation<ROT_THRESHOLD/2){
        return Tilted; // z_acceleration-g ->0 and no rotation
    }
    else
if(acceleration_x>ACC_THRESHOLD&&acceleration_y<ACC_THRESHOLD&&total_rotation<RO
T_THRESHOLD){
        return Moving_X; // x moving y no moving and no rotation
    }
    else
if(acceleration_x<ACC_THRESHOLD&&acceleration_y>ACC_THRESHOLD&&total_rotation<RO
T_THRESHOLD){
        return Moving_Y; // x no moving y moving and no rotation
    }
    else if(total_rotation>ROT_THRESHOLD){
        return Rotating;
    }
    else if(fabs(total_acceleration-g_acc)>ACC_THRESHOLD){
        return Moving;
    }
    return Stationary;
```

```
}

void *_imu_thread(void *arg)
{
    (void) arg;
    // Input your code
    // 1. initial mpu6050 sensor
    // 2. Acquire sensor data every 100ms
    // 3. Determine the motion state
    // 4. notify the LED thread to display the light color through a message.
    MPU6050 mpu;
    mpu.initialize();
    uint8_t gyro_fs=mpu.getFullScaleGyroRange();
    uint8_t accel_fs_g=mpu.getFullScaleAccelRange(); // detemine the motion
state
    float gyro_fs_convert=1.0;
    float accel_fs_real=1.0;
    switch(gyro_fs){
        case MPU6050_GYRO_FS_250:
            gyro_fs_convert=131.0;
            break;
        case MPU6050_GYRO_FS_500:
            gyro_fs_convert=65.5;
            break;
        case MPU6050_GYRO_FS_1000:
            gyro_fs_convert=32.8;
            break;
        case MPU6050_GYRO_FS_2000:
            gyro_fs_convert=16.4;
            break;
        default:
            printf("{IMU_THREAD} unknown GYRO_FS: 0x%x\n",gyro_fs);
            break;
    }

    switch (accel_fs_g)
    {
    case MPU6050_ACCEL_FS_2:
        accel_fs_real=g_acc*2;
        break;
    case MPU6050_ACCEL_FS_4:
        accel_fs_real=g_acc*4;
        break;
    case MPU6050_ACCEL_FS_8:
        accel_fs_real=g_acc*8;
        break;
    case MPU6050_ACCEL_FS_16:
        accel_fs_real=g_acc*16;
        break;
    default:
        printf("{IMU_THREAD} unknown ACCEL_FS: 0x%x\n",accel_fs_g);
        break;
    }

    float accel_fs_convert=32768.0/accel_fs_real;
```

```
    int16_t ax,ay,az,gx,gy,gz;

    delay_ms(1000);

    while(1){
        mpu.getMotion6(&ax,&ay,&az,&gx,&gy,&gz);
        MPU6050Data data;
        data.ax=ax/accel_fs_convert;
        data.ay=ay/accel_fs_convert;
        data.az=az/accel_fs_convert;
        data.gx=gx/gyro_fs_convert;
        data.gy=gy/gyro_fs_convert;
        data.gz=gz/gyro_fs_convert;

        printf("-------------------------\n");
        printf("{IMU_THREAD} Accelerometer (X,Y,Z): (%.02f,%.02f,%.02f)
m/s^2\n",data.ax,data.ay,data.az);
        printf("{IMU_THREAD} Gyroscope (X,Y,Z): (%.02f,%.02f,%.02f)
0/s\n",data.gx,data.gy,data.gz);
        MoveState state=detectMovement(data);
        msg_t msg;
        msg.content.value=(int)state;
        msg_send(&msg,_led_pid);

        delay_ms(200);
    }

    return NULL;
}
static const shell_command_t shell_commands[] = {
    { NULL, NULL, NULL }
};

int main(void)
{
    _led_pid = thread_create(stack_for_led_thread, sizeof(stack_for_led_thread),
THREAD_PRIORITY_MAIN - 2,
                             THREAD_CREATE_STACKTEST, _led_thread, NULL,
                             "led_controller_thread");
    if (_led_pid <= KERNEL_PID_UNDEF) {
        printf("[MAIN] Creation of receiver thread failed\n");
        return 1;
    }
    thread_create(stack_for_imu_thread, sizeof(stack_for_imu_thread),
THREAD_PRIORITY_MAIN - 1,
                             THREAD_CREATE_STACKTEST, _imu_thread, NULL,
                             "imu_read_thread");
    printf("[Main] Initialization successful - starting the shell now\n");
    while(1)
    {
        ztimer_sleep(ZTIMER_USEC, 1000000);
    }
    return 0;
}
```