

# EE351实验报告

---

微机原理与微系统

12212635 韩骐骏

EE351

# 目录

---

1. EE351实验报告	3
1.1 实验报告列表	3
2. 实验报告列表	4
2.1 树莓派开发环境搭建	4
2.2 双色灯实验	8
2.3 轻触开关实验	10
2.4 PCF8591模数转换器实验	15
2.5 模拟温度传感器实验	19
2.6 Lab6实验报告：超声波传感器测距实验	22
2.7 蜂鸣器实验	25
2.8 PS2操纵杆实验	29
2.9 红外遥控实验	32
2.10 中断实验	36

# 1. EE351实验报告

此处是EE351“微机原理与微系统”24Fall课程实验报告的主页，你可以在[这里](#)找到所有的实验报告。



本课程所有实验在树莓派4B上进行，使用的操作系统为RaspberryPi OS-64-bit-desktop。

本实验用到的硬件设备包括：

- 树莓派4 model B
- PCF8591模数转换器
- 传感器模块（如温度传感器、超声波传感器等）
- LED灯、蜂鸣器、电位器等
- PS2操纵杆、红外遥控器等
- 面包板、杜邦线等

本实验用到的软件工具包括：

- RPi.GPIO库（Python）
- wiringPi库（C/C++）
- python-smbus库（I2C通信）

## 1.1 实验报告列表

- [实验一：树莓派开发环境搭建](#)
- [实验二：双色灯实验](#)
- [实验三：轻触开关实验](#)
- [实验四：PCF8591模数转换器实验](#)
- [实验五：模拟温湿度传感器实验](#)
- [实验六：超声波传感器实验](#)
- [实验七：蜂鸣器实验](#)
- [实验八：PS2操纵杆实验](#)
- [实验九：红外遥控实验](#)
- [实验十：中断实验](#)

作者: 12212635 韩骐骏



本文档使用[mkdocs](#)生成静态网页，使用[mkdocs-with-pdf](#)生成PDF文档。访问[网页版](#)，获得更好的阅读体验。

## 2. 实验报告列表

### 2.1 树莓派开发环境搭建

#### Lab1实验报告：Raspberry Pi开发环境搭建

##### 一、实验介绍

本次实验将配置后续实验用到的软硬件环境，包括操作系统、网络、远程连接等。

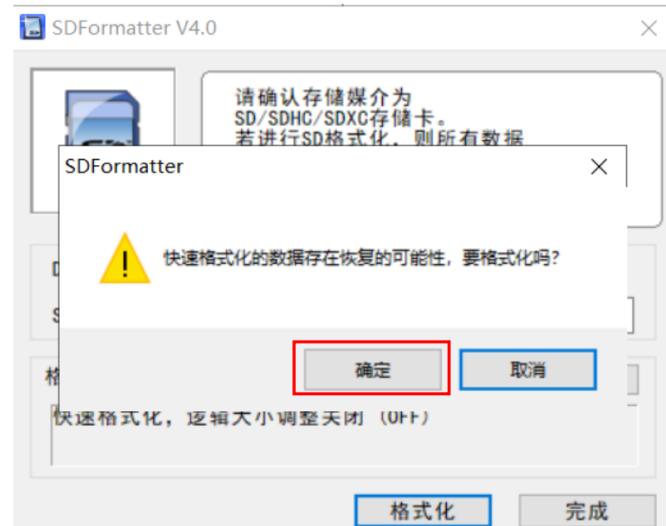
##### 二、实验目标

1. 熟悉Raspberry Pi硬件组成及其引脚布局。
2. 完成Raspberry Pi OS镜像的下载与烧录。
3. 配置Wi-Fi，确保能够访问互联网。

##### 三、实验步骤

###### (1) 硬件准备与检查

1. 确认所需材料：
  2. Raspberry Pi 4 Model B
  3. microSD卡
  4. 电源适配器（最好使用官方提供的USB-C电源，否则可能出问题）
  5. HDMI显示器及HDMI线缆
  6. 键盘和鼠标
  7. 检查硬件状态：
  8. 插入microSD卡到Raspberry Pi的卡槽中。
- 如果是使用过的卡，先用[SDFormatter](#)工具格式化。



9. 连接显示器、键盘和鼠标（如果打算使用GUI）。
10. 将电源线插入Raspberry Pi，并确保另一端连接到合适的电源插座上。

###### (2) 操作系统安装

1. 下载Raspberry Pi Imager工具：

2. 访问Raspberry Pi官方网站下载Imager工具。

## Install Raspberry Pi OS using Raspberry Pi Imager

Raspberry Pi Imager is the quick and easy way to install Raspberry Pi OS and other operating systems to a microSD card, ready to use with your Raspberry Pi.

Download and install Raspberry Pi Imager to a computer with an SD card reader. Put the SD card you'll use with your Raspberry Pi into the reader and run Raspberry Pi Imager.

[Download for Ubuntu for x86](#)

[Download for Windows](#)

[Download for macOS](#)

To install on **Raspberry Pi OS**, type

```
sudo apt install rpi-imager
```

in a Terminal window.



### 3. 选择并写入OS镜像：

4. 打开Raspberry Pi Imager，点击“CHOOSE OS”按钮，选择推荐的Raspberry Pi OS (32-bit)版本。
5. 点击“CHOOSE STORAGE”，挑选之前准备好的microSD卡作为存储介质。
6. 确认无误后，点击“WRITE”开始烧录过程。请耐心等待，直到提示写入完成。

### 7. 配置屏幕参数：

8. 在microSD卡的根目录下，找到 config.txt 文件，编辑并添加以下内容：

```
1 max_usb_current=1
2 hdmi_force_hotplug=1
3 config_hdmi_boost=7
4 hdmi_group=2
5 hdmi_mode=1
6 hdmi_mode=87
7 hdmi_drive=1
8 display_rotate=0
9 hdmi_cvt 1024 600 60 0 0 0 0
```

#### Warning

在 hdmi\_cvt 1024 600 60 0 0 0 这里填入实际显示屏的分辨率，不同显示器分辨率不同。

- 保存文件后，将microSD卡插回Raspberry Pi中。
- 初次启动与初始化设置：
  - 将烧录好OS镜像的microSD卡重新插回Raspberry Pi后，给它通电。
  - 第一次启动时，根据屏幕上的指示进行语言、地区、时区等基本信息的配置。

### (3) 网络配置

### 1. 连接Wi-Fi：

2. 在命令行中输入 `sudo raspi-config` 打开配置菜单。
3. 选择“Network Options”，然后按照提示输入您的Wi-Fi SSID和密码。
4. 验证网络连接：
5. 使用 `ping www.bing.com` 测试是否能成功访问外部网站。

(4) 配置开发环境

1. 更新软件包列表：
2. 执行 `sudo apt-get update` 刷新本地数据库以获取最新的软件包信息。
3. 升级已安装的软件包：
4. 使用 `sudo apt-get upgrade` 命令来更新所有现有的软件包至最新版本。
5. 安装额外的开发工具：
6. 安装Git用于版本控制：

```
1 sudo apt-get install git
```

7. 安装Vim编辑器： bash

```
sudo apt-get install vim
```

8. 安装远程连接工具：
9. 启动SSH服务：

```
1 sudo systemctl start ssh
```

10. 验证SSH服务是否正常运行：

```
1 sudo systemctl status ssh
```

11. 设置SSH服务开机自启动： bash

```
sudo systemctl enable ssh
```

12. 在本地通过vscode连接远程Raspberry Pi：

13. 安装Remote - SSH插件。

14. 在树莓派终端运行 `ifconfig` 命令查看IP地址。

15. 使用 Ctrl+Shift+P 打开命令面板，输入 Remote-SSH: Connect to Host，然后输入Raspberry Pi的IP地址和用户名。

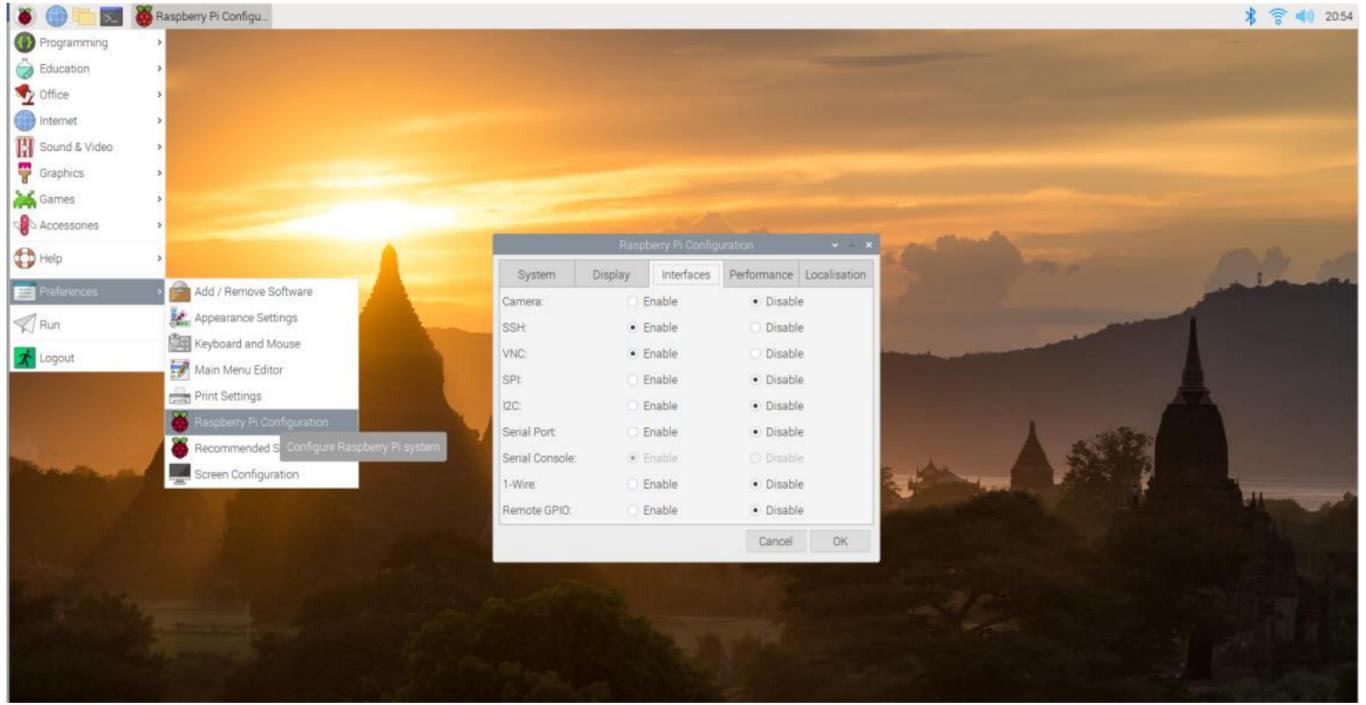
16. 输入密码后，即可通过VSCode连接到远程的Raspberry Pi。

17. 把 `ssh-rsa` 公钥添加到 `~/.ssh/authorized_keys` 文件中，实现无密码登录。

18. 远程可视化开发：

19. 在本地安装[VNC Viewer](#)。

## 20. 在树莓派上启用VNC Server：



21. 确保电脑和 Raspberry 在同一局域网。打开软件，在框内输入获取到的 Raspberry Pi ip 地址，回车。

22. 输入用户名和密码，即可远程连接到 Raspberry Pi 的桌面环境。

## 四、总结

本实验完成了Raspberry Pi的初步设置，包括操作系统的安装、网络的配置以及开发环境的搭建。

## 2.2 双色灯实验

### Lab2实验报告：学习知识准备与双色LED实验

#### 一、实验介绍

本实验旨在了解Raspberry Pi的IO接口及其引脚编号方式，并通过实际操作掌握使用wiringPi库和RPi.GPIO库控制硬件的方法，最终目标是实现一个简单的双色LED红绿交替闪烁效果。

#### 二、实验原理

##### 1. Raspberry Pi IO口：

2. Raspberry Pi拥有40个GPIO管脚，这些管脚可以通过不同的编号系统来引用，包括物理位置编号、wiringPi指定的编号以及BCM2837 SOC指定的编号。
3. 在本次实验（以及后续实验）中，我们使用BCM编码来连接和编程。

##### 4. wiringPi库：

5. wiringPi是一个用于C/C++语言的GPIO控制库，它简化了对Raspberry Pi GPIO的操作。安装此库后，可以方便地在命令行或程序中控制GPIO引脚。

##### 6. RPi.GPIO库：

7. RPi.GPIO是一个Python库，允许用户直接从Python代码中控制Raspberry Pi的GPIO。它是Raspbian操作系统的一部分，默认已安装，因此可以直接调用其API进行编程。

##### 8. Mu编辑器与Geany IDE：

9. Mu是一款适合初学者使用的Python编辑器，提供了基本的IDE功能如语法检查、运行和调试等。
10. Geany则是一款轻量级的跨平台IDE，支持多种编程语言，对于C/C++项目来说非常适合。



后续经验表明，这两款IDE在树莓派下渲染效果不佳，推荐使用VSCode远程开发插件。

#### 1. 双色LED模块：

2. 双色LED通常指的是包含两个独立发光单元（红色和绿色）在一个封装内的LED。通过改变输入电压或电流的方向，可以使不同的颜色发光或者两者同时亮起形成黄色。

#### 三、实验步骤

##### 1. 硬件连线：

2. 将双色LED的S引脚（绿色）、中间引脚（红色）分别连接到Raspberry Pi的GPIO接口上，GND引脚连接到Raspberry Pi的地线。记住使用的GPIO引脚编号（本次实验使用的是BCM编号下的GPIO19, GPIO20, GND）。

##### 3. 编写并上传代码：

4. 使用Mu编辑器创建一个新的Python脚本文件，编写一段代码来控制双色LED的红绿交替闪烁。代码应该设置好相应的GPIO模式（输入/输出），然后按照设定的时间间隔切换LED的状态。

5. 如果使用C/C++，则可以在Geany中新建一个源文件，同样需要配置GPIO模式，并且要记得在编译时链接wiringPi库。

##### 6. 运行测试：

7. 执行编写的Python脚本或编译后的C/C++程序，观察双色LED是否能够按照预期的顺序红绿交替闪烁。

8. 除了代码逻辑，一定要检查硬件连接是否正确无误！！！（比如是否插紧T型板、是否接错了引脚等）

##### 9. 清理工作：

10. 实验结束后，记得关闭所有运行中的进程，否则LED灯可能会一直亮着。

#### 四、编写代码

程序框图：



Python代码，用于实现双色LED的红绿交替闪烁：

```

1 import RPi.GPIO as GPIO
2 import time
3
4 RED_PIN = 19 # 红色LED
5 GREEN_PIN = 20 # 绿色LED
6
7 # 设置GPIO模式
8 GPIO.setmode(GPIO.BCM)
9 GPIO.setup(RED_PIN, GPIO.OUT)
10 GPIO.setup(GREEN_PIN, GPIO.OUT)
11
12 try:
13     while True:
14         # 打开红色LED
15         GPIO.output(RED_PIN, GPIO.HIGH)
16         GPIO.output(GREEN_PIN, GPIO.LOW)
17         print("红色LED已打开")
18         time.sleep(1) # 等待1秒钟
19
20         # 打开绿色LED
21         GPIO.output(RED_PIN, GPIO.LOW)
22         GPIO.output(GREEN_PIN, GPIO.HIGH)
23         print("绿色LED已打开")
24         time.sleep(1) # 等待1秒
25
26 except KeyboardInterrupt:
27     print("程序被终止")
28
29 finally:
30     # 退出前清理GPIO设置
31     GPIO.cleanup()

```

这段代码将使双色LED按照红-绿-红-绿的顺序交替闪烁，每次持续1秒钟。

C++代码

```

1 #include <wiringPi.h>
2 #include <iostream>
3
4 #define RED_PIN 19
5 #define GREEN_PIN 20
6
7 int main() {
8     wiringPiSetupGpio();
9     pinMode(RED_PIN, OUTPUT);
10    pinMode(GREEN_PIN, OUTPUT);
11
12    while (true) {
13        digitalWrite(RED_PIN, HIGH);
14        digitalWrite(GREEN_PIN, LOW);
15        std::cout << "Red LED is ON" << std::endl;
16        delay(1000);
17
18        digitalWrite(RED_PIN, LOW);
19        digitalWrite(GREEN_PIN, HIGH);
20        std::cout << "Green LED is ON" << std::endl;
21        delay(1000);
22    }
23
24    return 0;
25 }

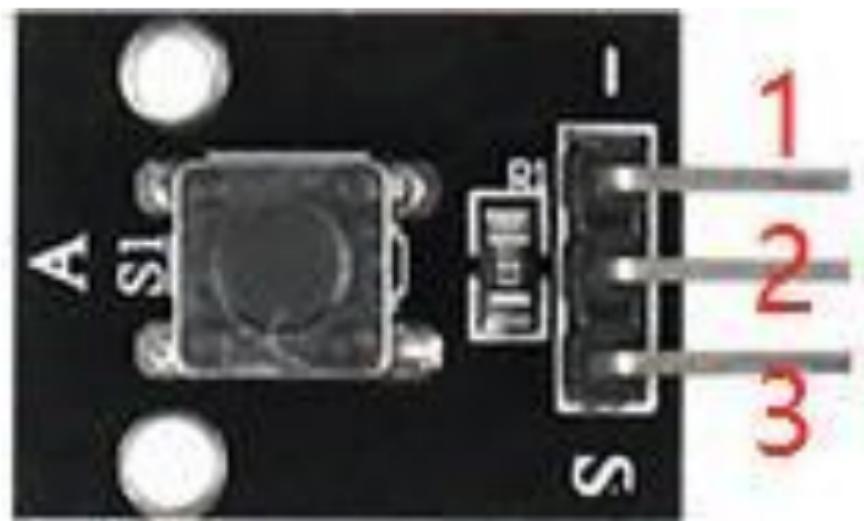
```

## 2.3 轻触开关实验

### Lab3实验报告：轻触开关实验

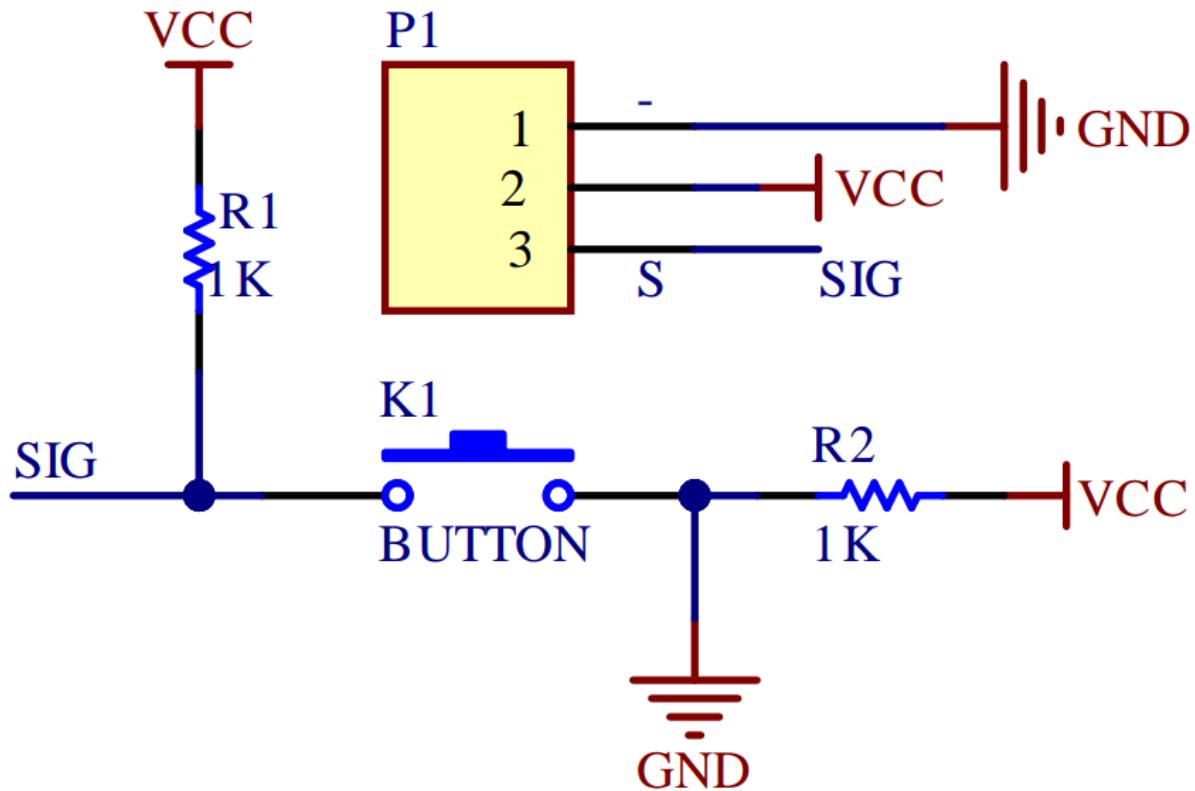
#### 一、实验介绍

轻触开关模块是最常见的开关模块，内部有一个轻触开关（按键开关）。-引脚接地，中间引脚接VCC。按下按键时，S脚输出为低电平；松开按键时，S脚输出为高电平。



该模

块的原理图为：



## 二、实验原理

在本实验中，使用轻触开关作为树莓派的输入设备。将树莓派某 GPIO 口设置为输入模式，通过此 GPIO 口检测轻触开关的 S 引脚。当检测到 S 引脚为低电平时，表示按键被按下，检测到 S 引脚为高电平时，表示按键松开。通过两种不同颜色的 LED 指示按键的状态。即当按键按下时，一种颜色 LED 亮；按键松开时，另一种 LED 亮。

### Warning

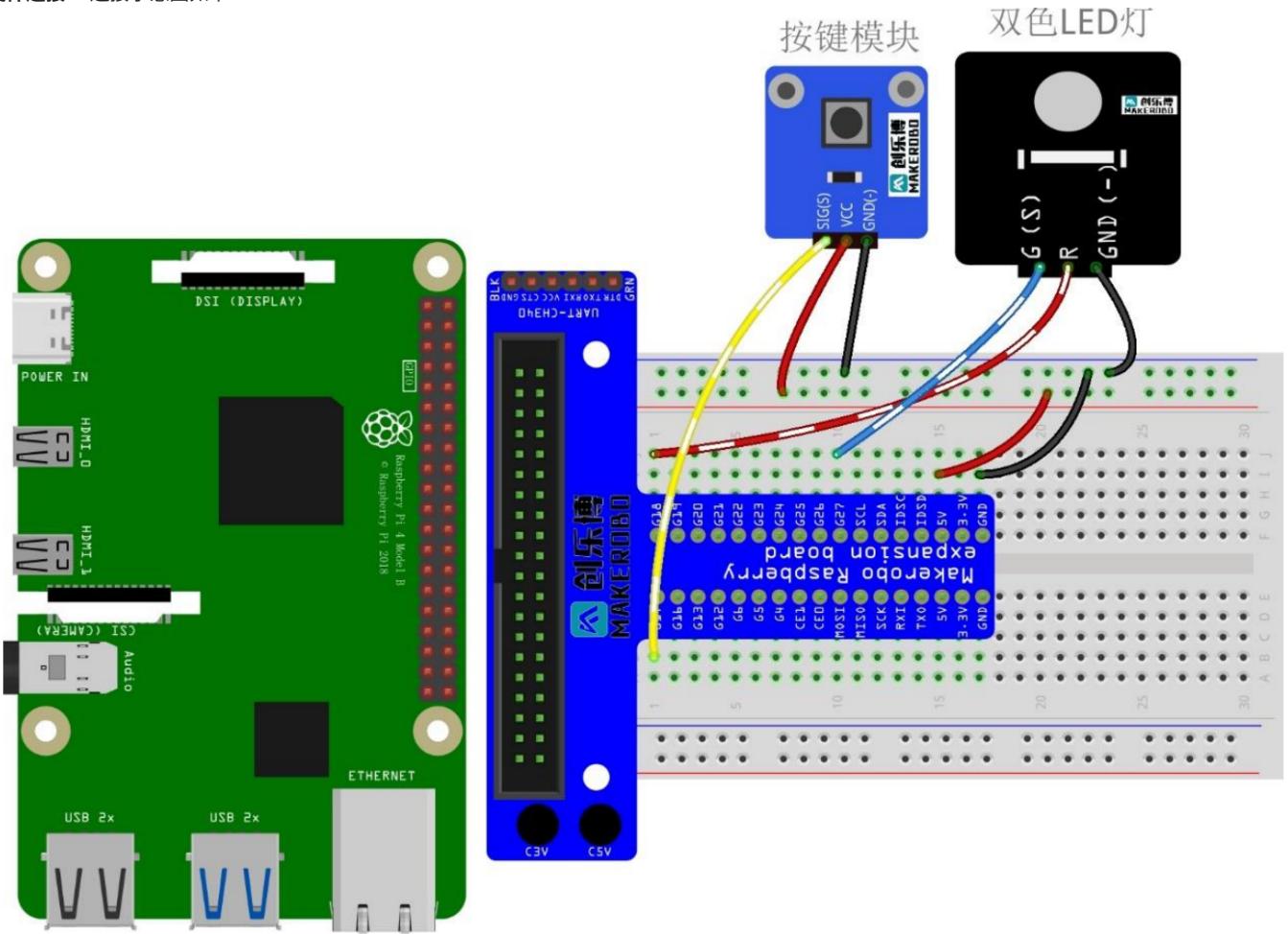
在编写代码时，需要注意消除好按键抖动。

### Tip

在 Python 中，`time.sleep()` 函数可以让程序暂停执行一段时间（秒），通过延时函数能有效地减少按键抖动带来的误触发问题。

## 三、实验步骤

1. 硬件连接：连接示意图如下：



2. 编写代码：使用Mu编辑器或VSCode等工具，编写Python代码来控制双色LED和轻触开关。代码逻辑如下：

3. 设置GPIO引脚的模式（输入/输出）。
4. 不断检测轻触开关的状态，根据按键的状态控制LED的亮灭。
5. 为了防止按键抖动，可以在检测到按键状态变化后加入一个小小的延时。

程序框图：

```

flowchart TD
    A[开始] --> B[设置GPIO模式]
    B --> C[循环]
    C --> D[检测按键状态]
    D --> E[按键按下]
    E --> F[点亮红灯]
    F --> G[熄灭绿灯]
    G --> H[等待0.1秒]
    H --> I[检测按键状态]
    I --> J[按键松开]
    J --> K[点亮绿灯]
    K --> L[熄灭红灯]
    L --> M[等待0.1秒]
    M --> C
  
```

```

1 import RPi.GPIO as GPIO
2 import time
3
4 RED_PIN = 19
5 GREEN_PIN = 20
6 SWITCH_PIN = 21 # 开关引脚
7
8 GPIO.setmode(GPIO.BCM)
9 GPIO.setup(RED_PIN, GPIO.OUT)
10 GPIO.setup(GREEN_PIN, GPIO.OUT)
11 GPIO.setup(SWITCH_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
12
13 def switch_with_delay(pin, delay=0.1):
14     """检测开关状态"""
15     state = GPIO.input(pin)
16     time.sleep(delay)
17     return state == GPIO.input(pin)
18
19 try:
20     while True:
21         if switch_with_delay(SWITCH_PIN):
22             GPIO.output(RED_PIN, GPIO.HIGH)
23             GPIO.output(GREEN_PIN, GPIO.LOW)
24         else:
25             GPIO.output(RED_PIN, GPIO.LOW)
26             GPIO.output(GREEN_PIN, GPIO.HIGH)
27
28     except KeyboardInterrupt:
29         print("Exiting...")
30     finally:
31         GPIO.cleanup()

```

## 1. 实验拓展

通过开关和 LED 及相应的编程，实现以下功能：1. 按一下按键，LED 红灯亮起；2. 再次按一下按键，LED 红灯闪烁；3. 再次按一下按键，LED 绿灯亮；4. 再次按一下按键，LED 绿灯闪烁；再次按下按键红灯亮起……如此循环。

程序框图：



```
1 import RPi.GPIO as GPIO
2 import time
3
4 RED_PIN = 19
5 GREEN_PIN = 20
6 SWITCH_PIN = 21 # 开关引脚
7
8 GPIO.setmode(GPIO.BCM)
9 GPIO.setup(RED_PIN, GPIO.OUT)
10 GPIO.setup(GREEN_PIN, GPIO.OUT)
11 GPIO.setup(SWITCH_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
12
13 def switch_with_delay(pin, delay=0.1):
14     """检测开关状态"""
15     state = GPIO.input(pin)
16     time.sleep(delay)
17     return state == GPIO.input(pin)
18
19 def toggle_led(pin):
20     """切换LED状态"""
21     GPIO.output(pin, not GPIO.input(pin))
22
23 try:
24     while True:
25         if switch_with_delay(SWITCH_PIN):
26             toggle_led(RED_PIN)
27             time.sleep(0.5)
28             toggle_led(RED_PIN)
29         else:
30             toggle_led(GREEN_PIN)
31             time.sleep(0.5)
32             toggle_led(GREEN_PIN)
33
34 except KeyboardInterrupt:
35     print("Exiting...")
36 finally:
37     GPIO.cleanup()
```

## 2.4 PCF8591模数转换器实验

### Lab4实验报告：PCF8591模数转换器实验

#### 一、实验介绍

PCF8591 是一款单芯片，单电源，低功耗 8 位 CMOS 数据采集设备，具有四个模拟输入，一个模拟输出和一个串行(I<sup>2</sup>C)总线接口。三个地址引脚(A\_0, A\_1 和 A\_2)用于对硬件地址进行编程，从而允许使用多达 8 个连接到(I<sup>2</sup>C)总线的设备，而无需额外的硬件。通过两行双向(I<sup>2</sup>C)总线串行传输与设备之间的地址，控制和数据。该设备的功能包括模拟输入多路复用，片上跟踪和保持功能，8 位模数转换和 8 位数模转换。最大转换率由(I<sup>2</sup>C)总线的最大速度决定。本次实验目标为：通过控制 PCF8591，将 LED 灯点亮。

#### 二、实验原理

##### 1. PCF8591特性：

2. PCF8591是一款单芯片、低功耗的CMOS数据采集设备，它包含模拟输入多路复用、片上跟踪保持功能、8位A/D转换和8位D/A转换。
3. 设备通过I<sup>2</sup>C总线接口与主控制器通信，默认地址为0x48，但可以通过设置地址引脚A0, A1, 和A2改变其硬件地址，最多允许连接8个相同类型的从设备到同一(I<sup>2</sup>C)总线上。
4. 发送到PCF8591 器件的第二个字节将被存储在其控制寄存器中，并且需要 控制器件功能。控制寄存器的高半字节用于使能模拟输出，并将模拟输入编程为 单端或差分输入。下半字节选择由上半字节定义的一个模拟输入通道。如果设置 了自动增加标志，则在每次 A/D 转换后，通道编号会自动递增。
5. 在本实验中，AIN0(模拟输入 0)端口用于接收来自电位计模块的模拟信号，AOUT(模拟输出)用于将模拟信号输出到双色 LED 模块，以便改变 LED 的亮度。该模块的原理图如下所示：



需要注意的是，除了电位器，PCF8591 模块还带有光电二极管和负温度系数（NTC）热敏电阻，原理图如下所示。当外部光强或温度变化时，光敏或热敏电阻的阻值也会发生变化，通过采集 INPUT1 和 INPUT2 的电压值，可以实现光强和温度感知。

##### 6. I<sup>2</sup>C总线通信：

7. (I<sup>2</sup>C)是一种简单的两线式串行通信标准，由SDA（数据线）和SCL（时钟线）组成。在本实验中，Raspberry Pi作为主设备，负责发送命令给PCF8591并接收来自它的响应。

##### 8. 模拟信号采集与处理：

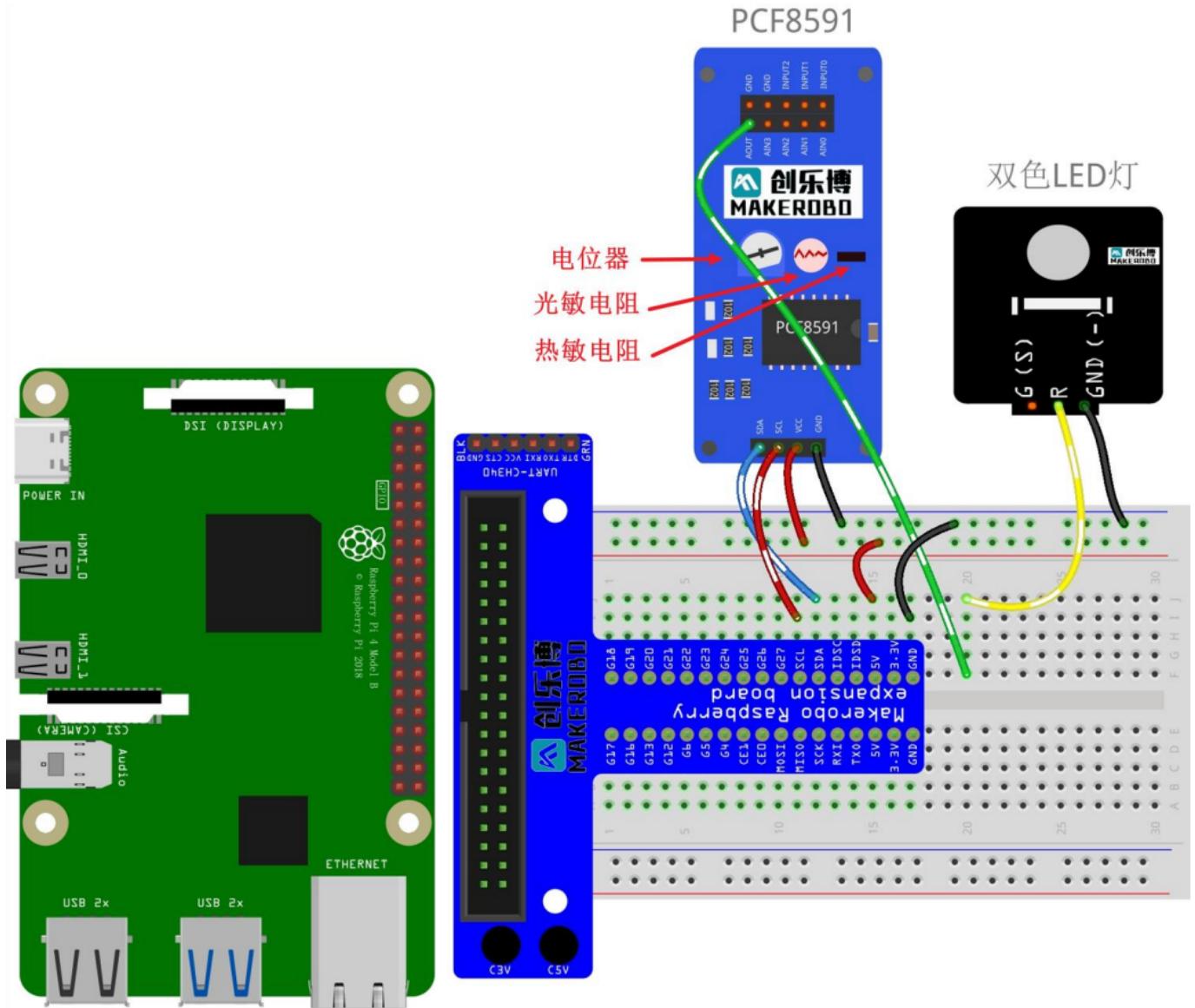
9. 在这个实验里，AIN0端口被用来接收来自电位计模块的模拟信号，而AOUT端口则输出模拟电压以驱动双色LED模块，从而改变LED的亮度。
10. 当外部条件发生变化时（例如光照强度或温度变化），相应的传感器（如光电二极管或NTC热敏电阻）的阻值也会随之变化，通过测量这些元件两端的电压，我们可以得知环境的变化情况。

#### 三、实验步骤

##### 1. 硬件连接：

2. 连接Raspberry Pi、T型转接板和PCF8591模块之间的SDA、SCL、VCC和GND引脚。

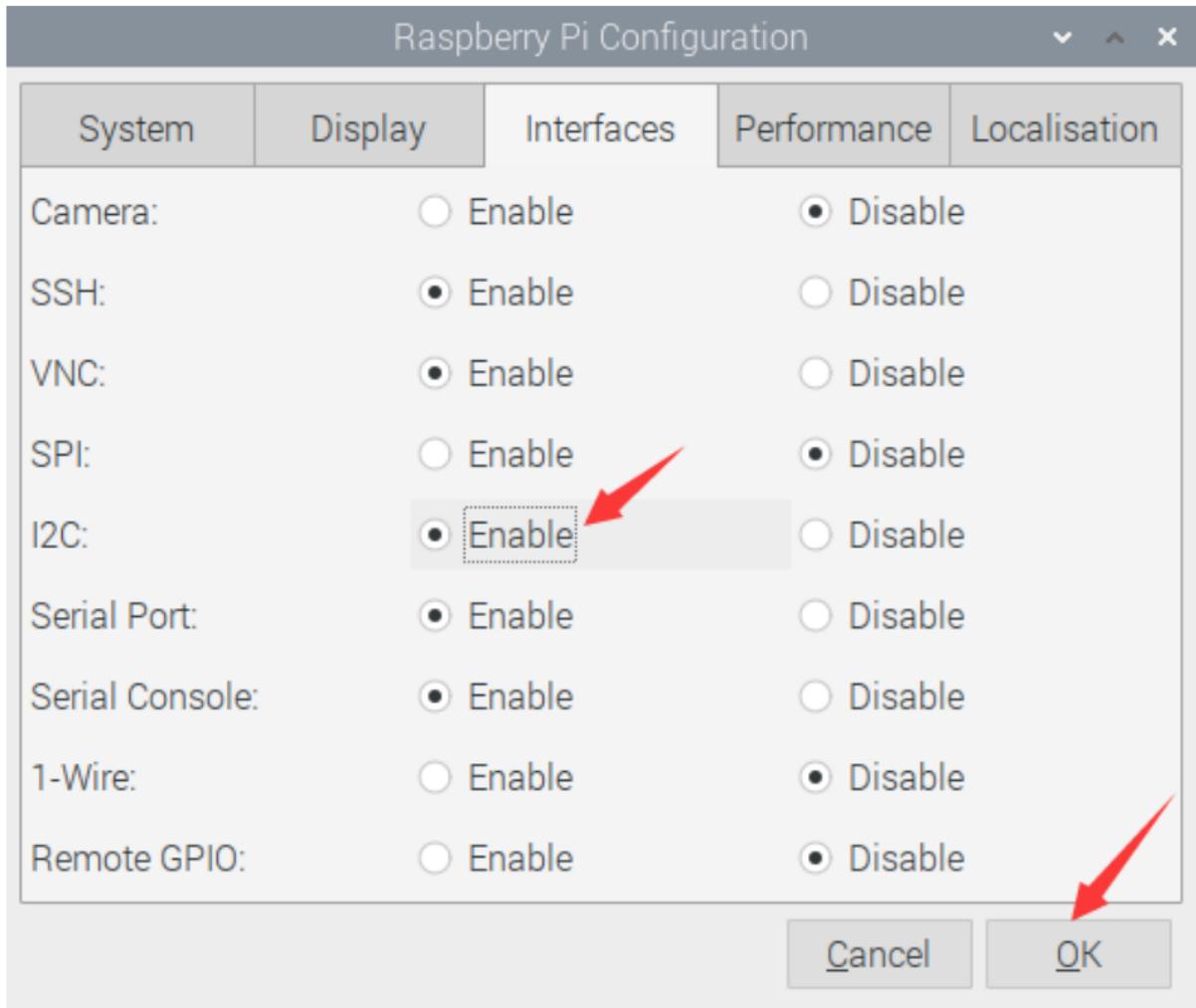
3. 将双色LED的中间引脚（红色）连接到PCF8591的AOUT引脚，GND引脚接地。



4. 配置I2C总线：

5. 点击Raspberry Pi桌面环境中的开始菜单，选择Preferences -> Raspberry Pi Configuration。

6. 进入Interfaces标签页，开启I2C选项，点击OK保存更改并重启系统。



7. 查看设备地址：

8. 在终端中输入 `sudo i2cdetect -y 0` 命令，查看I2C总线上所有设备的地址。

9. 如果一切正常，应该能看到PCF8591的地址（默认为0x48）显示在对应的位置上。

10. 编写代码：

11. 使用Python语言编写程序，首先需要安装 `smbus` 库，它可以方便地操作I2C设备。

12. 导入必要的库后，创建一个SMBus实例并与PCF8591建立连接，读取AIN0上的模拟值并根据该值调整AOUT输出，进而控制LED亮度。

程序框图：

```

graph TD
    A[开始] --> B[初始化I2C总线]
    B --> C[设置PCF8591地址和控制位]
    C --> D[进入循环]
    D --> E[向PCF8591写入控制字节]
    E --> F[启动A/D转换]
    F --> G[读取模拟值]
    G --> H[打印模拟值]
    H --> I[将模拟值映射到LED亮度范围]
    I --> J[打印LED亮度百分比]
    J --> K[延时]
    K --> D
    D --> L[检测到键盘中断?]
    L -- 是 --> M[结束]
    L -- 否 --> D
  
```

## Python代码

```
1 import smbus
2 import time
3
4 address = 0x48 # 地址
5 control_bit = 0x40 # 控制字
6
7 bus = smbus.SMBus(1)
8
9 try:
10     while True:
11         # 向PCF8591写入控制字节
12         bus.write_byte(address, control_bit)
13
14         analog_value = bus.read_byte(address)
15
16         print("Analog Value:", analog_value)
17
18         # 把模拟值映射到LED亮度范围
19         led_brightness = int((analog_value / 255.0) * 100)
20
21         print("LED Brightness (%):", led_brightness)
22
23         time.sleep(0.1)
24
25 except KeyboardInterrupt:
26     print("Exiting...")
```

## 2.5 模拟温度传感器实验

### Lab5实验报告：模拟温度传感器实验

#### 一、实验介绍

温度感测模块提供易于使用的传感器，它带有模拟和数字输出。该温度模块 使用 NTC（负温度系数）热敏电阻来检测温度变化，其对温度感应非常灵敏。NTC 热敏电阻电路相对简单，价格低廉，组件精确，可以轻松获取项目的温度数据，因此广泛应用于各种温度的感测与补偿中。简而言之，**NTC 热敏电阻将随温度变化传递为电阻变化**，利用这种特性，我们可以通过测量电阻网络(例如分压器)的电压来检测室内/环境温度。本次实验的任务为：获取当前环境的温度值。

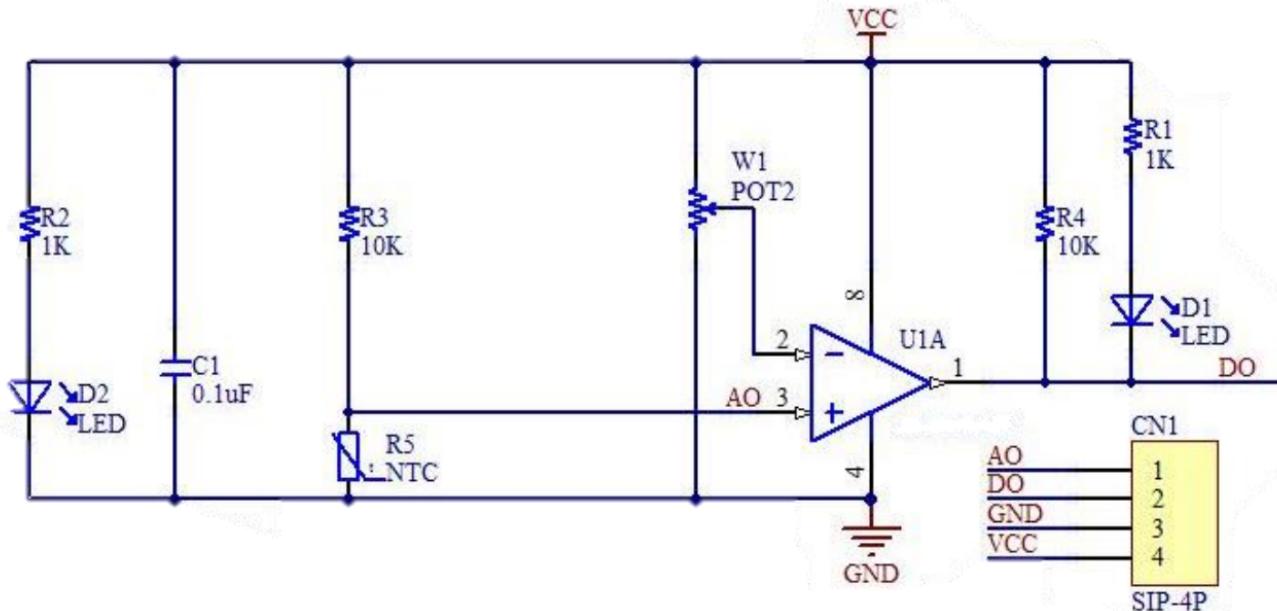
#### 二、实验原理

##### 1. NTC热敏电阻特性：

- 在本实验中，我们将使用Steinhart-Hart方程来计算热敏电阻的精确温度，这是一个用于描述热敏电阻电阻-温度特性的经验公式，即Steinhart-Hart方程。
- Steinhart-Hart方程表达式为  $\frac{1}{T} = A + B \ln(R) + C(\ln(R))^3$ ，其中  $T$  是以开尔文为单位的绝对温度，  $R$  是热敏电阻在给定温度下的电阻值，而  $A$ ,  $B$ ,  $C$  则是取决于具体型号的常数参数。对于本次实验，假设  $R_0$  为  $10k\Omega$ ,  $B$  值为  $3950K$ 。

##### 4. 电路：

- 温度传感器模块由一个NTC热敏电阻和一个固定电阻组成分压电路。当环境温度发生变化时，热敏电阻的阻值也会随之改变，从而影响分压点处的电压输出。



- 通过连接到PCF8591的模拟输入端口AIN0，我们可以采集这个电压信号，并将其转换为数字形式以便后续分析。

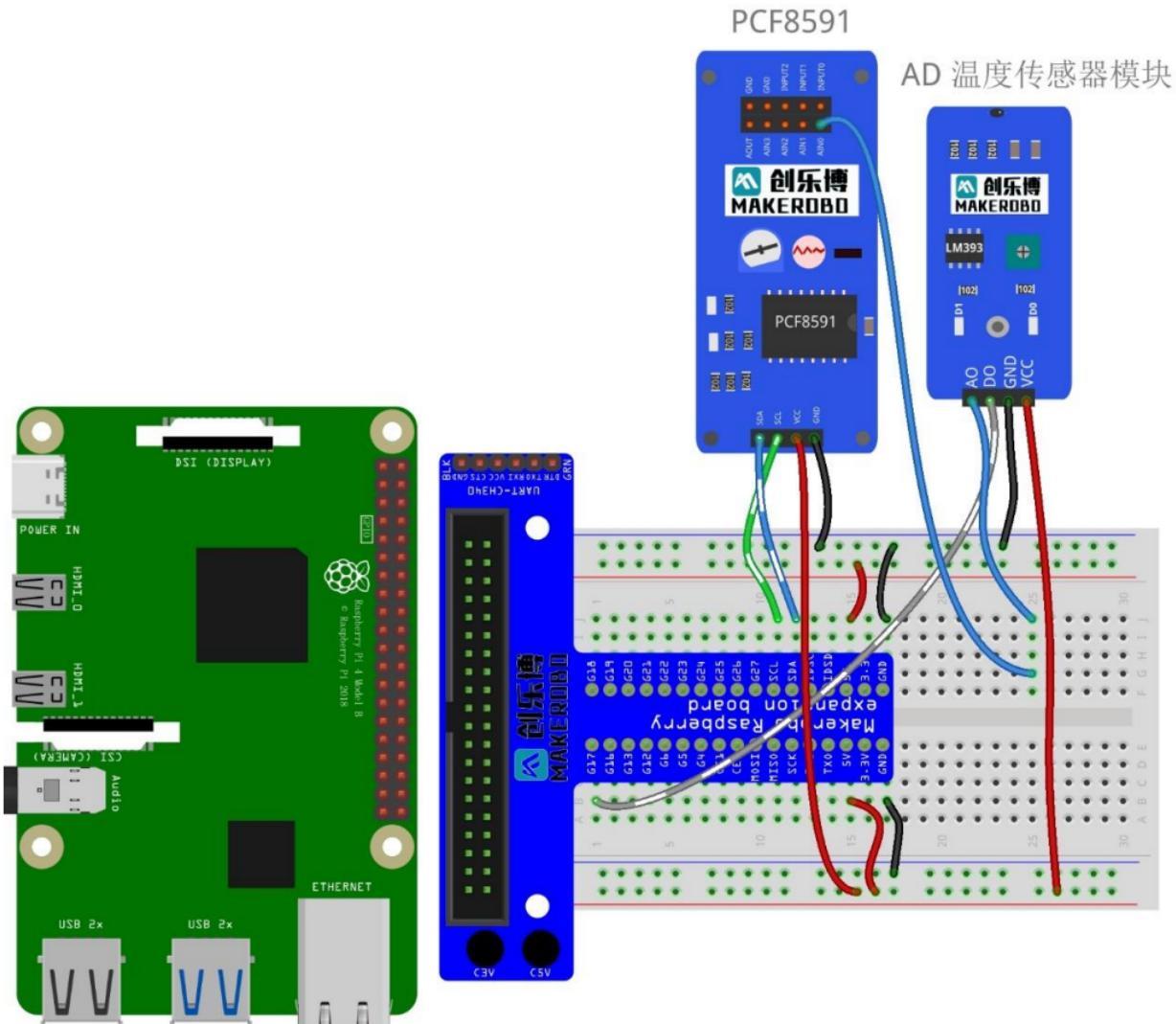
##### 7. 数据处理：

- 首先从PCF8591读取经过A/D转换后的数值  $(analogVal = PCF8591.read(0))$ ，然后根据已知条件（如供电电压  $(V_{cc} = 5V)$ ，ADC分辨率为8位即  $255$  对应  $0V$ ）计算出对应的模拟电压  $(V_r = V_{cc} * float(analogVal) / 255.0)$ 。
- 接着利用分压比公式计算得到热敏电阻的实际阻值  $(R_t = R_0 * V_r / (V_{cc} - V_r))$ 。
- 再代入Steinhart-Hart方程求解温度  $T$ 。

#### 三、实验步骤

##### 1. 硬件连接：

2. 连接Raspberry Pi、T型转接板和PCF8591模块之间的SDA、SCL、VCC和GND引脚。



3. 将模拟温度传感器的AO引脚连接到PCF8591模块的AIN0，DO引脚可以留空或接地，VCC引脚接5V电源，GND引脚接地。

4. 配置I2C总线：见[Lab4实验报告](#)中的第2步。

5. 编写代码

6. 导入必要的库。

7. 创建一个SMBus实例并与PCF8591建立连接，读取AIN0上的模拟值并根据该值计算温度。

程序框图：

```
flowchart TD
    A[开始] --> B(初始化I2C总线);
    B --> C{设置PCF8591地址和控制位};
    C --> D{读取AIN0模拟值};
    D --> E{计算温度};
    E --> F{打印温度};
    F --> G{延时};
    G --> B
```

Python代码：

```

1 import smbus
2 import math
3 import time
4
5 address = 0x48 # 地址
6 control_bit = 0x40 # 控制字
7
8 # 常数
9 R0 = 10000
10 B = 3950
11 T0 = 298.15 # 25°C -> 开氏温度
12 Vcc = 5.0 # 5V供电
13
14 bus = smbus.SMBus(1)
15
16 def read_temperature():
17     try:
18         # 设置PCF8591地址和控制位
19         bus.write_byte(address, control_bit)
20
21         analog_value = bus.read_byte(address)
22
23         Vr = (analog_value / 255.0) * Vcc
24
25         # 计算热敏电阻的阻值
26         Rt = R0 * Vr / (Vcc - Vr)
27
28         # 计算温度
29         temp_kelvin = 1 / (math.log(Rt / R0) / B + 1 / T0)
30         temp_celsius = temp_kelvin - 273.15
31
32     return round(temp_celsius, 2)
33
34 except Exception as e:
35     print("Error reading temperature:", str(e))
36     return None
37
38 try:
39     while True:
40         temperature = read_temperature()
41         if temperature is not None:
42             print(f"Temperature: {temperature}°C")
43         else:
44             print("Failed to read temperature.")
45
46         time.sleep(1)
47
48 except KeyboardInterrupt:
49     print("\nExiting program.")

```

## 2.6 Lab6实验报告：超声波传感器测距实验

### 一、实验介绍

超声波测距模块主要是由两个通用的压电陶瓷超声传感器，并加外围信号处理电路构成的。超声传感器中的一个用作发射器，将电信号转换为 40KHz 超声波脉冲信号；另一个一个用作接收器，监听发射的脉冲。超声波距离传感器体积小，易于在项目中使用，可以提供 2cm 至 400cm 左右的非接触距离检测，精度为 3mm。由于它的工作电压为 5 伏，因此可以直接连接到 Raspberry 或任何其他 5V 逻辑微控制器。该传感器有 4 个引脚：- **VCC**：5V 电源供电；- **Trig**：触发引脚，用于启动超声波发射；- **Echo**：回波引脚，表示是否检测到返回的超声波；- **GND**：接地。

该模块的原理图如下：



### 二、实验原理

#### 1. 超声波传感器工作流程：

2. 超声波传感器包括一个发射器和一个接收器。当触发引脚 (Trig) 接收到至少 10 微秒的高电平脉冲时，它会发送 8 个周期的 40kHz 超声波脉冲。
3. 接收器监听反射回来的超声波，并将 Echo 引脚置为高电平直到接收到回波为止。此时，Echo 引脚保持高电平的时间长度与超声波往返一次所需的时间成正比。
4. 由于声音传播速度约为 343 米/秒（在 20 摄氏度空气中），因此可以通过测量 Echo 引脚高电平持续时间（秒）\* 34300 / 2
5. 测试距离（单位：厘米）= Echo 引脚高电平持续时间（秒）\* 34300 / 2

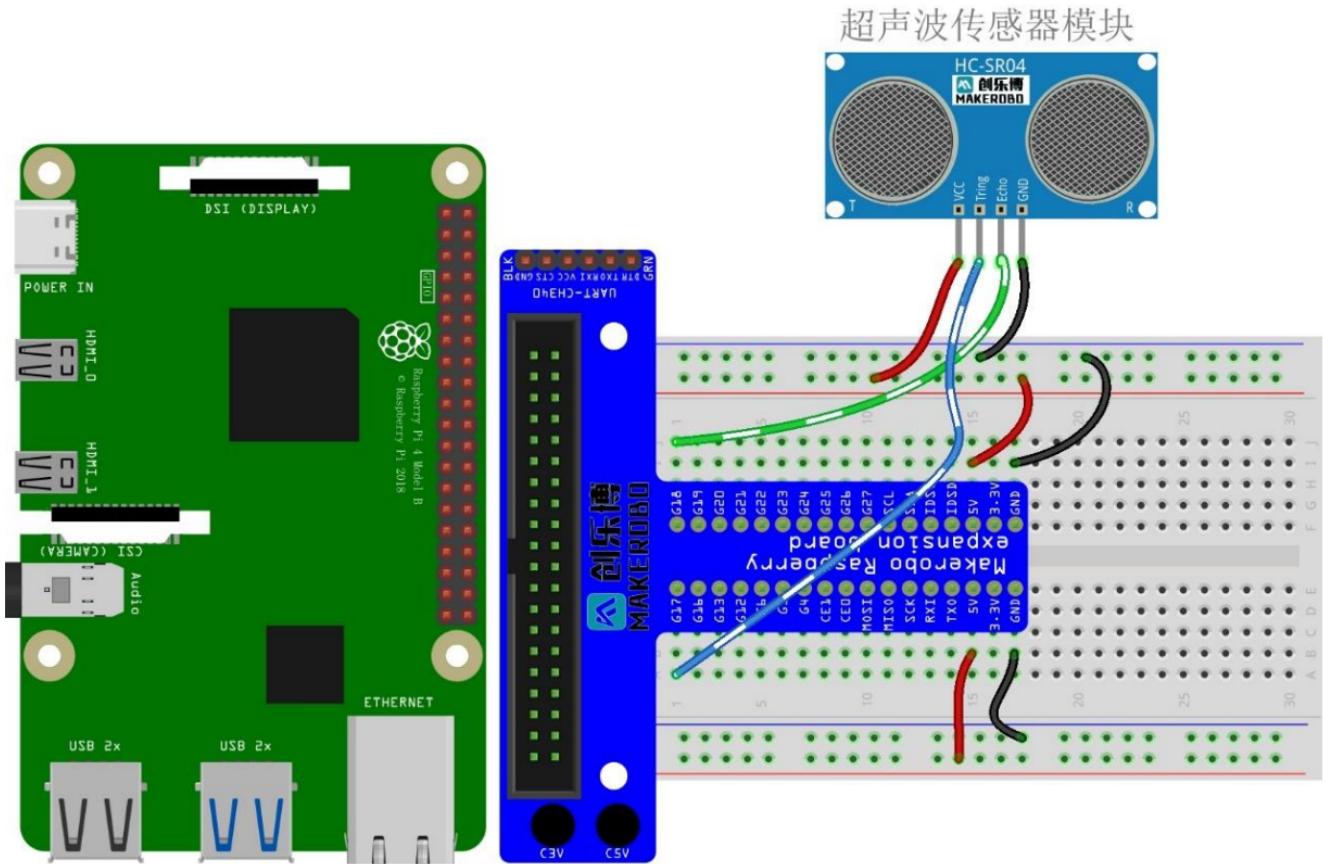
#### 6. 注意事项：

7. 注：Echo 引脚变为高电平时为 5V，树莓派 GPIO 输入一般不能超过 3.3V，故应使用分压器测量。但由于本次实验 Echo 引脚高电平时间非常短，故可不使用分压。

### 三、实验步骤

#### 1. 硬件连接：

2. 连接Raspberry Pi、T型转接板和超声波传感器之间的VCC、Trig、Echo和GND引脚。



3. 将超声波传感器的Trig引脚连接到Raspberry Pi的GPIO17（BCM编号），Echo引脚连接到GPIO18（BCM编号），同时确保VCC接到5V电源，GND接地。

#### 4. 编写代码：

5. 导入 RPi.GPIO 库，设置Trig和Echo引脚的BCM编号。

6. 编写函数 `get_distance()`，该函数负责设置Trig引脚输出10微秒的高电平脉冲，然后等待Echo引脚变为高电平，记录开始时间；接着再次等待Echo引脚变为低电平，记录结束时间。最后利用这两个时间点计算出超声波往返一次所花费的时间，并据此换算成实际距离。

程序框图：

```

graph TD
    A[开始] --> B{设置GPIO引脚}
    B --> C{循环}
    C --> D[发射超声波]
    D --> E{等待Echo引脚高电平}
    E --> F{等待Echo引脚低电平}
    F --> G[计算距离]
    G --> H{打印距离}
    H --> I[延时]
    I --> C
  
```

```
1 import RPi.GPIO as GPIO
2 import time
3
4 TRIG = 17 # 传感器的Trig引脚
5 ECHO = 18 # Echo引脚
6
7 GPIO.setmode(GPIO.BCM)
8 GPIO.setup(TRIG, GPIO.OUT)
9 GPIO.setup(ECHO, GPIO.IN)
10
11 def get_distance():
12     # 确保Trig引脚为低电平
13     GPIO.output(TRIG, False)
14     time.sleep(0.2)
15     GPIO.output(TRIG, True)
16     time.sleep(0.00001)
17     GPIO.output(TRIG, False)
18
19     # 记录开始时间
20     while GPIO.input(ECHO) == 0:
21         pulse_start = time.time()
22
23     # 记录结束时间
24     while GPIO.input(ECHO) == 1:
25         pulse_end = time.time()
26
27     pulse_duration = pulse_end - pulse_start
28
29     distance = pulse_duration * 17150
30     distance = round(distance, 2)
31
32     return distance
33
34 try:
35     print("Measuring distance...")
36     while True:
37         dist = get_distance()
38         print(f"Distance: {dist} cm")
39         time.sleep(1)
40
41 except KeyboardInterrupt:
42     print("Measurement stopped by user")
43
44 finally:
45     GPIO.cleanup()
```

## 2.7 蜂鸣器实验

---

### Lab7实验报告：蜂鸣器实验

#### 一、实验介绍

蜂鸣器属于声音模块，一般可以分为有源蜂鸣器和无源蜂鸣器。有源和无源是指内部是否有震荡源。有源蜂鸣器内置振荡器，没有频率变化，直接接上合适的直流电源即可发声，常用于发出单一的提示性报警声音；无源蜂鸣器由于内部没有震荡源，所以其驱动方式为脉冲频率调制（Pulse-Frequency Modulation, PFM），可以通过调控脉冲频率发出不同频率的声音信号。本次实验任务为利用蜂鸣器播放一段音乐（音乐自选），并通过编程控制蜂鸣器发出相应的音符。

#### 二、实验原理

##### 1. 有源蜂鸣器：

2. 内部含有振荡电路，可以将恒定的直流电转化为一定频率的脉冲信号，因此只需给它施加合适的直流电压即可让它发出声音。

3. 在本实验中使用的有源蜂鸣器为低电平触发，即当GPIO引脚设置为低电平时，蜂鸣器会响起；反之则停止发声。

##### 4. 无源蜂鸣器：

5. 没有内置驱动电路，必须由外部提供特定频率的方波信号才能工作。由于声音频率可控，可以发出“do re mi fa so la xi”的声效。在一些特例中，可以和LED复用一个控制口。

6. PFM (Pulse-Frequency Modulation) 是一种仅使用两个电平（高/低）表示模拟信号的调制方式，在这里用来生成可变频率的脉冲序列以驱动无源蜂鸣器。

7. PWM (Pulse-Width Modulation) 虽然不是本次实验的重点，但作为一种常见的调制技术，它同样适用于控制蜂鸣器或其他设备的输出特性。

#### 8. 编程思路：

9. 对于有源蜂鸣器，只需要简单地配置对应的GPIO引脚状态为高或低就可以控制其开关。

10. 对于无源蜂鸣器，则需要创建一个包含多个音符频率值的列表，并依次遍历这个列表，每次根据当前音符设定适当的PWM频率，使蜂鸣器按照指定旋律发声。

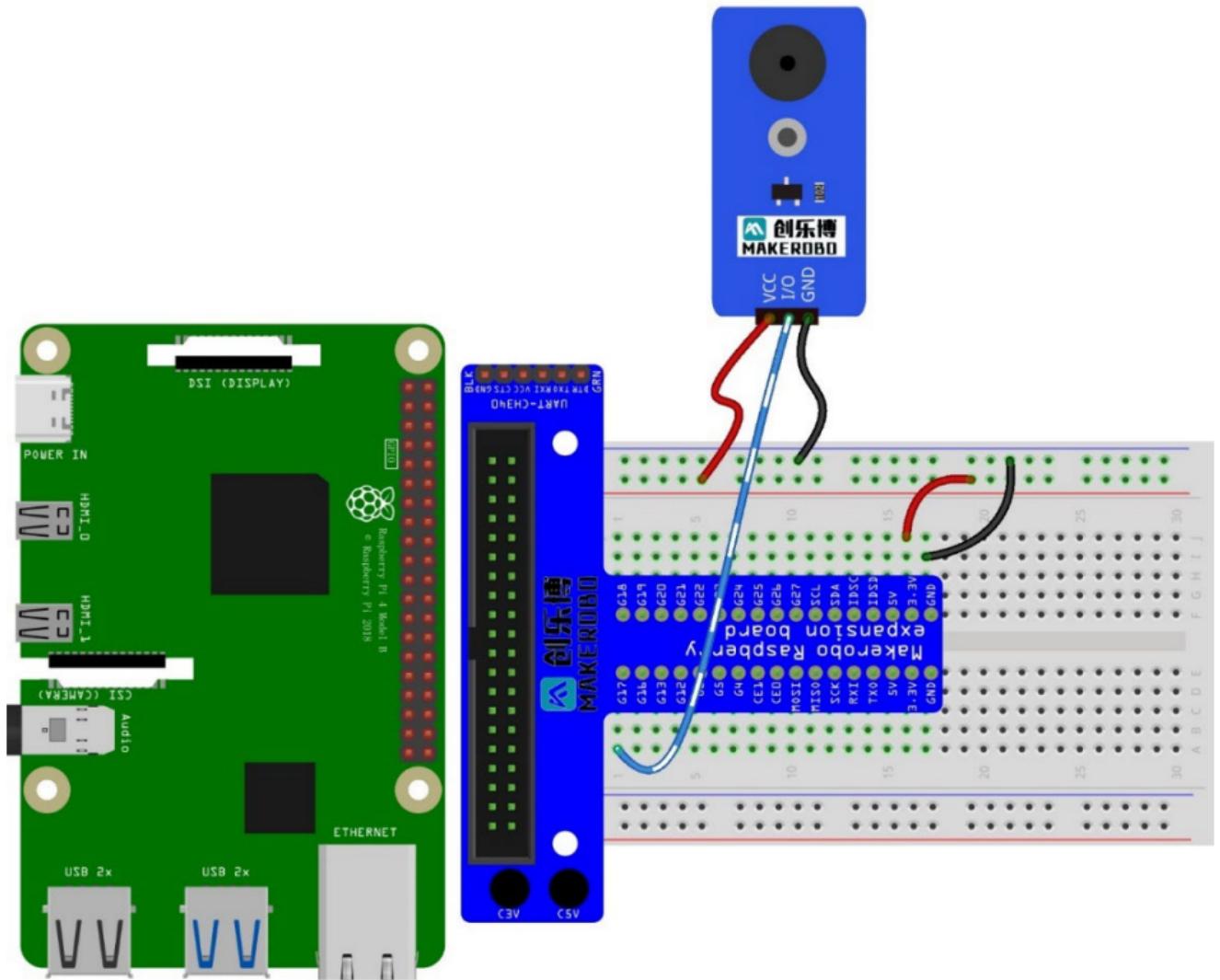
#### 三、实验步骤

##### (1) 有源蜂鸣器实验

##### 1. 硬件连接：

2. 连接Raspberry Pi、T型转接板和有源蜂鸣器模块之间的I/O、VCC3.3V和GND引脚。

有源蜂鸣器

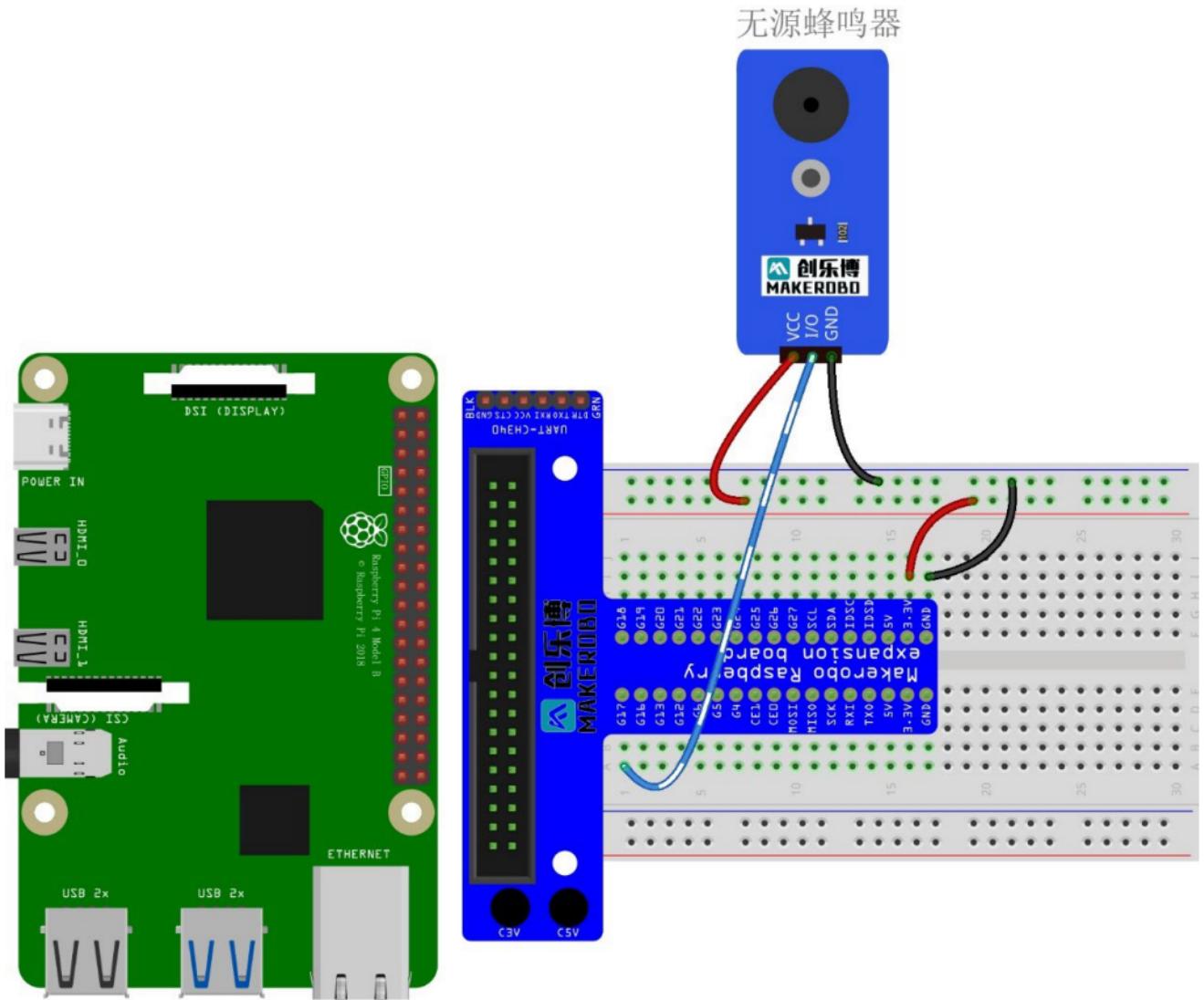


3. 通电后，蜂鸣器会发出持续的“滴滴”声音。

(2) 无源蜂鸣器实验

1. 硬件连接：

2. 连接Raspberry Pi、T型转接板和无源蜂鸣器模块之间的I/O、VCC和GND引脚。



3. 确保选择支持PWM输出的GPIO引脚（本次使用GPIO11，BCM编号）。

4. 编写代码：

5. 导入RPi.GPIO 和 pigpio 库，设置蜂鸣器的GPIO引脚和频率参数。

6. 编写函数 play\_music()，该函数定义了一系列音符及其对应的频率，并通过循环调用这些频率来驱动蜂鸣器发出音乐。

程序框图：



```

1 import RPi.GPIO as GPIO
2 import pigpio
3 import time
4
5 BUZZER_PIN = 18
6
7 pi = pigpio.pi()
8
9 # 音符及其对应频率 (Hz)
10 NOTES = {
11     "C4": 262,
12     "D4": 294,
13     "E4": 330,
14     "F4": 349,
15     "G4": 392,
16     "A4": 440,
17     "B4": 494,
18     "C5": 523,
19     "D5": 587,
20     "E5": 659,
21     "F5": 698,
22     "G5": 784,
23     "A5": 880,
24     "B5": 988,
25 }
26
27 MELODY = [
28     "C4",
29     "D4",
30     "E4",
31     "C4",
32     "E4",
33     "D4",
34     "C4",
35     "C4",
36     "D4",
37     "D4",
38     "E4",
39     "E4",
40     "C4",
41     "D4",
42     "E4",
43 ]
44
45
46 def set_frequency(freq):
47     """通过PWM设置蜂鸣器的频率"""
48     pi.hardware_PWM(BUZZER_PIN, freq, 500000)
49
50
51 def play_music(melody):
52     try:
53         for note in melody:
54             if note in NOTES:
55                 set_frequency(NOTES[note])
56                 time.sleep(0.5) # 每个音符持续0.5秒
57                 set_frequency(0) # 停止发声
58                 time.sleep(0.1) # 间隔0.1秒
59
60     except KeyboardInterrupt:
61         print("Music stopped by user")
62
63     finally:
64         pi.stop()
65         GPIO.cleanup()
66
67
68 if __name__ == "__main__":
69     print("Playing music...")
70     play_music(MELODY)

```

## 2.8 PS2操纵杆实验

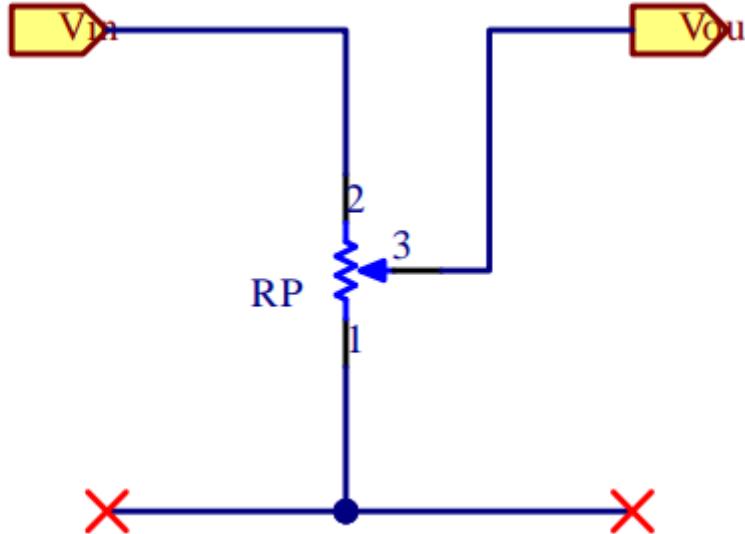
### Lab8实验报告：PS2操纵杆实验

#### 一、实验介绍

PS2 操纵杆模块类似于手柄中的模拟游戏杆，是一种输入设备，其在许多项目中得到应用。它是通过以 90 度角安装两个电位计来制成的。电位计连接到以 弹簧为中心的短杆上。本次实验任务为用 PS2 操纵杆控制不同的 LED 以及其亮度变化。一个轴控制红灯亮度，另一个轴控制绿灯亮度，按下按钮则熄灭两者。

#### 二、实验原理

**1. PS2操纵杆工作原理：** PS2 操纵杆有两个模拟输出(对应 X 和 Y 坐标)和一个数字输出，表示是否在 Z 轴上按下。处于静止位置时，其在 X 和 Y 方向产生约 2.5V 的输出，移动操纵杆将导致输出在 0v 到 5V 之间变化，具体取决于其方向。按下按钮时，其 SW 引脚输出为低电平。其内部结构实际上就是两个 X, Y 方向上的滑动变阻器。当 VCC 连接 5V 电压时，X, Y 方向电压常态时为 2.5V，最大值 5V，最小值 0V，用 PCF8591 模数转换模块的两个通道分别检测电压值的变化就可以知道摇杆指向的位置了。



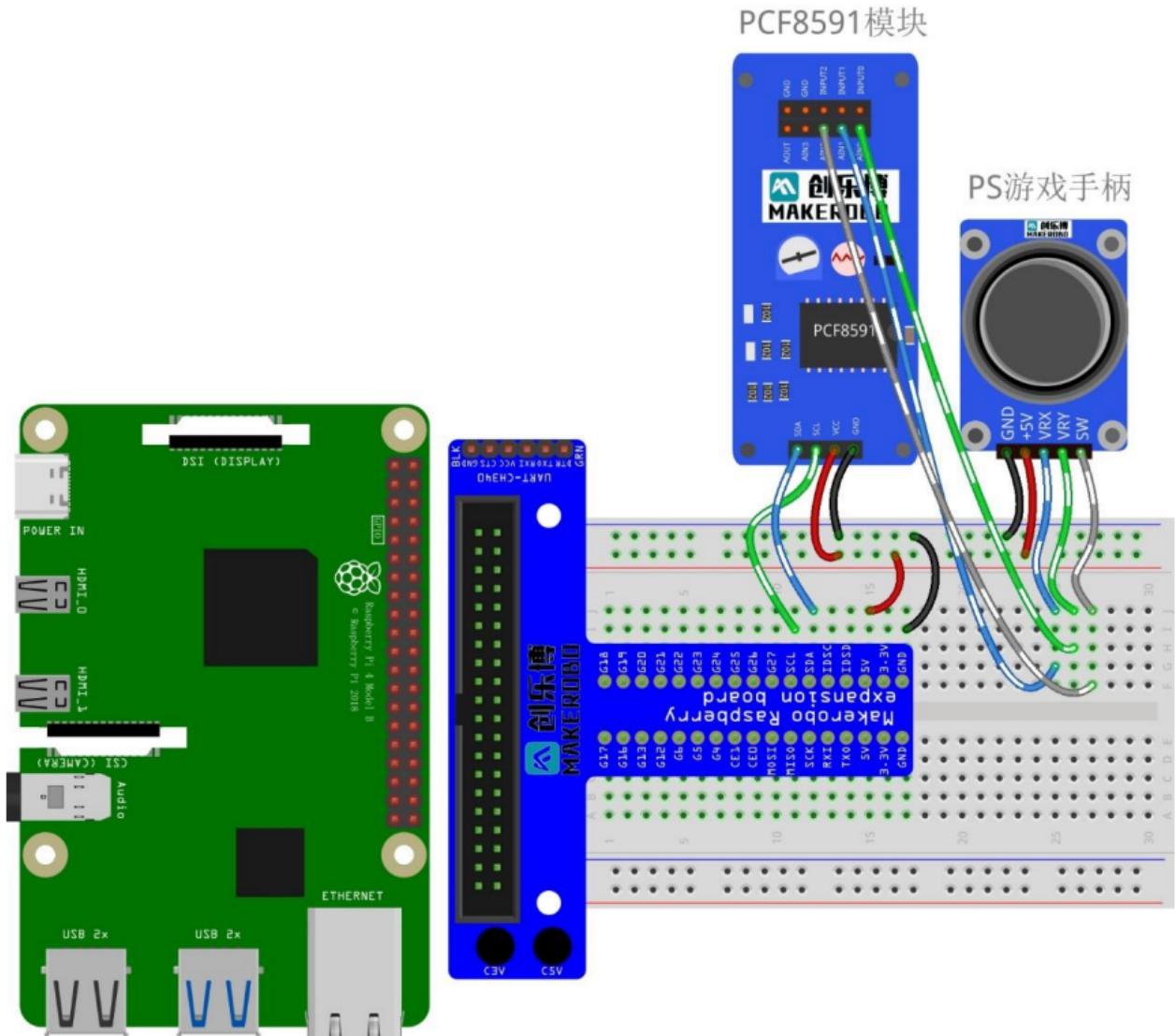
#### 2. 电路连接：

3. 在本实验中，我们将PS2操纵杆的X轴（VRX）和Y轴（VRY）连接到PCF8591的模拟输入端口AIN0和AIN1，而按钮（SW）则可以连接到另一个GPIO引脚或者留空。
4. PCF8591负责将来自操纵杆的模拟电压信号转换为数字值，这些数值可以在Raspberry Pi上进一步处理以确定操纵杆的具体位置。
5. 数据处理与控制逻辑：
6. 通过读取PCF8591提供的数字化后的X轴和Y轴数据，我们可以得知当前操纵杆指向的位置。
7. 根据操纵杆的位置，我们可以改变连接到PCF8591模拟输出端口AOUT的LED亮度（0-255）。例如，当操纵杆位于中心位置时，LED保持一定亮度；随着操纵杆向任意方向偏移，相应地增加或减少LED的亮度。

#### 三、实验步骤

1. 硬件连接：
2. 连接Raspberry Pi、T型转接板、PCF8591模块以及PS2操纵杆之间的SDA、SCL、VCC、GND、VRX、VRY和SW引脚。

3. 将PS2操纵杆的VRX引脚连接到PCF8591模块的AIN0，VRY引脚连接到AIN1，SW引脚可以根据需要选择性连接到额外的GPIO引脚，VCC引脚接5V电源，GND引脚接地。



4. 配置I2C总线：见[Lab4实验报告](#)中的第2步。

5. 编写代码：

6. 导入必要的库。

7. 创建一个SMBus实例并与PCF8591建立连接，读取AIN0和AIN1上的模拟值，并根据这些值计算出对应的LED亮度。

程序框图：

```

flowchart TD
    A[开始] --> B{读取 PS2 操纵杆 X 轴和 Y 轴};
    B --> C[将 X 轴值映射到红色 LED 亮度];
    C --> D[将 Y 轴值映射到绿色 LED 亮度];
    D --> E[设置红色 LED 的 PWM 占空比];
    E --> F[设置绿色 LED 的 PWM 占空比];
    F --> G{检测按键是否按下?};
    G -- 是 --> H[熄灭所有 LED];
    H --> I[延时];
    I --> G;
    G -- 否 --> B;
  
```

## Python代码

```

1 import smbus
2 import RPi.GPIO as GPIO
3 import time
4
5 # I2C 地址
6 PCF8591_ADDRESS = 0x48
7
8 # GPIO 引脚
9 RED_LED_PIN = 27
10 GREEN_LED_PIN = 22
11 BUTTON_PIN = 17
12
13 # 初始化
14 bus = smbus.SMBus(1) # 使用 I2C 总线 1
15 GPIO.setmode(GPIO.BCM)
16 GPIO.setup(RED_LED_PIN, GPIO.OUT)
17 GPIO.setup(GREEN_LED_PIN, GPIO.OUT)
18 GPIO.setup(BUTTON_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
19
20
21 def read_adc(channel):
22     bus.write_byte(PCF8591_ADDRESS, 0x40 | channel) # 选择通道
23     bus.read_byte(PCF8591_ADDRESS) # 去除第一次读取
24     return bus.read_byte(PCF8591_ADDRESS)
25
26
27 try:
28     while True:
29         x_value = read_adc(0) # 读取 X 轴
30         y_value = read_adc(1) # 读取 Y 轴
31
32         # 将 ADC 值 (0-255) 映射到 PWM 值 (0-100)
33         red_brightness = int(x_value / 255 * 100)
34         green_brightness = int(y_value / 255 * 100)
35
36         # 使用 PWM 控制 LED 亮度 (你需要设置 GPIO 为 PWM 输出)
37         red_pwm = GPIO.PWM(RED_LED_PIN, 100) # 100Hz 频率
38         green_pwm = GPIO.PWM(GREEN_LED_PIN, 100)
39         red_pwm.start(red_brightness)
40         green_pwm.start(green_brightness)
41
42         if GPIO.input(BUTTON_PIN) == GPIO.LOW: # 按键按下
43             red_pwm.stop()
44             green_pwm.stop()
45             GPIO.output(RED_LED_PIN, GPIO.LOW)
46             GPIO.output(GREEN_LED_PIN, GPIO.LOW)
47             time.sleep(0.5) # 延时防止抖动
48         else:
49             red_pwm.ChangeDutyCycle(red_brightness)
50             green_pwm.ChangeDutyCycle(green_brightness)
51
52 except KeyboardInterrupt:
53     red_pwm.stop()
54     green_pwm.stop()
55     GPIO.cleanup()

```

## 2.9 红外遥控实验

### Lab9实验报告：红外遥控实验

#### 一、实验介绍

遥控器接收头是用于接收遥控器所发射信号进而读取按键信息而执行操作的一种光电信号转换器件。遥控器接收头是利用最新的 IC 技术开发和设计出来的小型红外控系统接收器。在支架上装着 PIN 二极管和前置放大器，环氧树脂包装成一个红外过滤器，解调输出信号可以由微处理器解码，一般三条腿的红外线遥控接收头是接收、放大、解调一体头，接收头输出的是解调后的数据信号，Raspberry pi 里面需要相应的读取程序。红外通信是利用红外技术实现两点间的近距离保密通信和信息转发。它一般由红外发射和接收系统两部分组成。发射系统对一个红外辐射源进行 调制后发射红外信号，而接收系统用光学装置和红外探测器进行接收，就构成红外通信系统。



#### 二、实验原理

在本实验中，我们使用 lirc 库读取遥控器按钮返回的红外信号，并将它们转换为按钮值。

### 三、实验步骤

1. 连接电路 连接遥控器接收头到 Raspberry Pi 的 GPIO 引脚上，如下图所示：



### 2. 安装LIRC及相关配置：

3. 使用以下命令安装LIRC软件包及其依赖项：

```
1 sudo apt-get update
2 sudo apt-get install lirc
```

4. 修改 /boot/config.txt 文件中的红外模块部分，确保启用了红外接收功能，并指定了正确的GPIO引脚编号（例如接收引脚为22，发射引脚为23）。添加或修改如下行：

```
1 dtoverlay= gpio-ir,gpio_pin=22
2 dtoverlay= gpio-ir-tx,gpio_pin=23
```

### 5. 调整驱动设置：

6. 编辑位于 /etc/lirc/lirc\_options.conf 的LIRC配置文件，更改默认驱动程序和设备路径：

```
1 sudo nano /etc/lirc/lirc_options.conf
```

将内容更改为：

```
1 driver = default
2 device = /dev/lirc0
```

### 7. 重启系统：

8. 执行完上述配置更改后，请重启Raspberry Pi以使新的设置生效：

```
1 sudo reboot
```

### 9. 测试IR接收器：

10. 重启完成后，可以使用 irw 命令查看当前接收到的红外信号。打开终端窗口并输入：

```
1  irw
```

11.

```
pulse 227
timeout 127608
pulse 203
timeout 134804
pulse 201
timeout 131259
pulse 202
timeout 130188
pulse 201
timeout 127711
pulse 201
timeout 129438
pulse 199
space 40855
pulse 231
timeout 131769
pulse 229
space 43554
pulse 232
timeout 127633
pulse 201
timeout 134456
^Xpulse 202
timeout 132576
pulse 206
timeout 132497
^C
pi@raspberrypi:~ $
```

此时按下遥控器上的任意按键，观察屏幕上的十六进制代码输出。

12. **记录红外代码:** 在终端输入 irrecord -l 命令，按照提示操作，记录红外遥控器的按键代码。屏幕上输出的十六进制代码如下：

```
1  KEY_0
2  KEY_102ND
3  KEY_10CHANNELDOWN
4  KEY_10CHANNELUP
5  KEY_1
6  KEY_2
7  KEY_3
8  KEY_3D_MODE
9  KEY_4
10 KEY_5
11 KEY_6
12 KEY_7
13 KEY_8
14 KEY_9
15 KEY_A
16 KEY_AB
17 KEY_ADDRESSBOOK
18 KEY AGAIN
19 KEY_ALTERASE
20 KEY_ANGLE
21 KEY_APOSTROPHE
22 KEY_APPSELECT
23 KEY_ARCHIVE
24 KEY_ASPECT_RATIO
25 KEY_ASSISTANT
26 KEY_ATTENDANT_OFF
```

13. **开始录制 :** 在终端输入 irrecord -d /dev/lirc0 ~/lircd.conf 命令，按照提示操作，录制红外遥控器的按键代码。

14. **保存文件:** 保存文件后，将其复制到 /etc/lirc/lircd.conf 目录下：

```
1  sudo cp ~/lircd.conf /etc/lirc/lircd.conf
```

**15. 重命名文件:** 重命名文件后，重启LIRC服务：

```

1 cd /etc/lirc/lircd.conf.d
2 sudo mv devinput.lircd.conf devinput.lircd.dist
3 sudo service lircd restart
4 sudo lircd --nodaemon --device /dev/lirc1 --driver default

```

**16. 测试遥控器:** 在终端输入 sudo irw 命令，按下遥控器上的按键，观察屏幕上的输出。

**17. 关联Python程序:** 修改文件名

```

1 cd /etc/lirc
2 sudo mv irexec.lircrc lircrc

```

编辑 /etc/lirc/lircrc 文件，添加以下内容：

```

1 begin
2     prog = irexec
3     button = KEY_1
4     config = echo "Button 1 pressed"
5 end
6 begin
7     prog = irexec
8     button = KEY_2
9     config = echo "Button 2 pressed"
10 end
11 begin
12     prog = irexec
13     button = KEY_3
14     config = echo "Button 3 pressed"
15 end
16 begin
17     prog = test.py

```

**18. 编写Python程序:** 程序框图

```

graph TD
    A[开始] --> B{导入lirc库}
    B --> C{定义解析函数}
    C --> D{建立连接}
    D --> E{读取数据}
    E --> F{解析按键}
    F --> G{打印按键}
    G --> H{循环}
    H --> E
    H -- 中断 --> I[退出]

```

Python代码

```

1 import lirc
2
3
4 def pasreset(data): # 解析按键
5     if data == 'echo "KEY_1"':
6         print("1 按下") # 遥控器按下1:
7     elif data == 'echo "KEY_2"':
8         print("2 按下") # 遥控器按下2:
9     elif data == 'echo "KEY_3"':
10        print("3 按下") # 遥控器按下3:
11
12
13 with lirc.LircdConnection(
14     "test.py",
15 ) as conn:
16     while True:
17         string = conn.readline()
18         pasreset(string)
19         print("收到:", end="")
20         print(type(string))

```

**19. 运行Python程序:** 在终端输入 python3 test.py 命令，按下遥控器上的按键，观察终端上的输出。

## 2.10 中断实验

### Lab10实验报告：中断实验

#### 一、实验介绍

在计算机系统中，外部中断是指在程序执行过程中，由外部输入触发的一种机制，用于暂停当前程序的执行，转而执行其他指定任务。外部中断的存在主要是为了处理紧急事件，例如硬件设备的输入、定时器的到期等。树莓派也可实现类似的功能。本次实验任务为通过中断机制在树莓派上实现对不同外接设备的及时响应。任务一为通过中断实现对按键的响应，改变LED灯的状态；任务二为通过中断实现按键每按一次，超声波传感器测距一次。最终的效果为：按下按键，LED灯亮，超声波传感器测距一次；松开按键，LED灯灭，超声波传感器不再测距。

#### 二、实验原理

##### 1. 树莓派中断函数：

2. 使用 `GPIO.add_event_detect()` 方法来监控指定GPIO引脚的状态改变。此方法接受四个参数：

- `channel`：需要监测的GPIO引脚编号。
- `edge`：指定要监测的边沿类型，可以是上升沿（`GPIO.RISING`）、下降沿（`GPIO.FALLING`）或者两者皆可（`GPIO.BOTH`）。



There are 2 kind of interrupts:

- RISING: when the state goes from LOW to HIGH.
- FALLING: when the state goes from HIGH to LOW.

- `callback`：当检测到状态变化时调用的回调函数（可选）。
  - `bouncetime`：用于消除机械按键抖动的时间间隔（ms），即两次有效状态变化之间所需的最长时间差（可选）。
- !!! note "回调函数" 回调函数：在编程中，回调函数是指在某个事件发生时由系统调用的函数。在这里，回调函数用于处理按键按下时的逻辑，例如切换LED的状态。

##### 3. `GPIO.wait_for_edge(channel, GPIO.RISING)` :

4. 另一种方式是使用 `GPIO.wait_for_edge()` 函数，在检测到指定边沿之前阻止程序继续执行。这种方法占用较少CPU资源。

#### 5. 按键去抖动：

6. 由于物理按键按下时可能会产生短暂的电压波动（即“抖动”），因此在实际应用中通常会加入软件延时或者硬件滤波来确保每个按键动作只被记录一次，在lab3中已经有所涉及，当时是通过手动设置延时来实现的消抖。

#### 三、实验步骤

##### 1. 建立电路：

2. 连接Raspberry Pi、T型转接板和轻触按键模块之间的SIG(S)、VCC和GND引脚。
3. 将轻触按键模块的SIG(S)引脚连接到Raspberry Pi的GPIO23（BCM编号），VCC引脚接5V电源，GND引脚接地。
4. 把LED灯和超声波传感器的电路连接到树莓派上，具体连接方式见[lab6](#)。
- 5. 编写Python程序：**
6. 导入 RPi.GPIO 库，设置轻触按键的GPIO引脚编号。
7. 编写函数 button\_callback()，该函数用于处理轻触按键按下时的逻辑，例如切换LED灯的状态。
8. 使用 GPIO.add\_event\_detect() 方法监控轻触按键的状态变化，当检测到按键按下时调用 button\_callback() 函数。
9. 编写函数 ultrasonic\_callback()，该函数用于处理超声波传感器测距的逻辑，每次按键按下时调用一次。
10. 使用 GPIO.wait\_for\_edge() 方法等待轻触按键按下，然后调用 ultrasonic\_callback() 函数。

程序框图：

```
flowchart TD
    A[程序开始] --> B{按键按下?};
    B -- 是 --> C[点亮 LED];
    C --> D[触发超声波];
    D --> E[接收到回波?];
    E -- 是 --> F[计算距离并打印];
    F --> G[按键松开?];
    G -- 是 --> H[熄灭 LED];
    H --> B;
    E -- 否 --> G;
    B -- 否 --> I[程序循环];
    I --> B;
```

Python代码：

```

1 import RPi.GPIO as GPIO
2 import time
3
4 BUTTON_PIN = 23
5 LED_PIN = 18
6 TRIG = 17
7 ECHO = 27 # 确保 ECHO 引脚正确
8
9 # 设置 GPIO 模式
10 GPIO.setmode(GPIO.BCM)
11 GPIO.setup(BUTTON_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
12 GPIO.setup(LED_PIN, GPIO.OUT)
13 GPIO.setup(TRIG, GPIO.OUT)
14 GPIO.setup(ECHO, GPIO.IN)
15
16 # 全局变量, 用于控制超声波测量
17 measuring = False
18 pulse_start = 0
19 pulse_end = 0
20
21
22 def button_pressed_callback(channel):
23     """按键按下回调函数:点亮 LED 并启动测距"""
24     global measuring
25     GPIO.output(LED_PIN, GPIO.HIGH)
26     measuring = True # 设置为 True 以允许测距
27     GPIO.output(TRIG, False) # 先拉低, 再拉高, 使得输出一个干净的脉冲
28     time.sleep(0.01)
29
30     GPIO.output(TRIG, True)
31     time.sleep(0.00001) # 10us 脉冲
32     GPIO.output(TRIG, False)
33
34
35 def button_released_callback(channel):
36     """按键松开回调函数:熄灭 LED"""
37     global measuring
38     GPIO.output(LED_PIN, GPIO.LOW)
39     measuring = False # 设置为 False 以停止测距
40
41
42 def echo_callback(channel):
43     """ECHO 引脚状态变化回调函数:计算距离"""
44     global measuring, pulse_start, pulse_end
45     if measuring: # 只有在测量状态下才进行计算
46         if GPIO.input(ECHO) == GPIO.HIGH:
47             pulse_start = time.time()
48         elif GPIO.input(ECHO) == GPIO.LOW:
49             pulse_end = time.time()
50             pulse_duration = pulse_end - pulse_start
51             distance = pulse_duration * 34300 / 2
52             print(f"Distance: {distance:.2f} cm")
53
54
55 # 添加按键按下和松开事件检测
56 GPIO.add_event_detect(
57     BUTTON_PIN, GPIO.FALLING, callback=button_pressed_callback, bouncetime=200
58 ) # 按下
59 GPIO.add_event_detect(
60     BUTTON_PIN, GPIO.RISING, callback=button_released_callback, bouncetime=200
61 ) # 松开
62
63 # 添加 ECHO 引脚状态变化检测
64 GPIO.add_event_detect(ECHO, GPIO.BOTH, callback=echo_callback)
65
66 try:
67     print("Press the button...")
68     while True:
69         time.sleep(0.1)
70
71 except KeyboardInterrupt:
72     print("\nCleaning up...")
73     GPIO.cleanup()

```



None