实 验 报 告

Raspberry Pi-based Basic Module Experiment

课程名称：微机原理与微系统

实验题目：基于树莓派的模块实验

学　　号：*********

姓　　名：*********

专　　业：自动化

目录

基于树莓派的模块实验

Raspberry Pi-based Basic Module Experiment

实验一 RaspberryPi 系统安装与 SSH/VNC 服务

一、实验目的

1. 下载、安装，和烧录 Raspberry Pi 操作系统到 SD 卡。
2. 连接 Raspberry Pi 到显示器和电源，并配置基本设置。
3. 启用 SSH 和 VNC 服务，以便进行远程访问。
4. 使用 SSH 和 VNC 远程连接到 Raspberry Pi。

二、实验过程

1. 下载系统映像文件

访问树莓派官方网站，并点击“Software”选项。

在“Our software”页面中，选择“See all download options”。

选择“Raspberry Pi OS with desktop and recommended software”下载文件，并验证文件完整性。

2. 烧录系统

使用 Raspberry Pi Imager 软件将下载的 zip 文件解压。

（其实上面的内容不是必须的因为 Raspberry Pi Imager 自带映像）

下载并安装 Raspberry Pi Imager 软件。

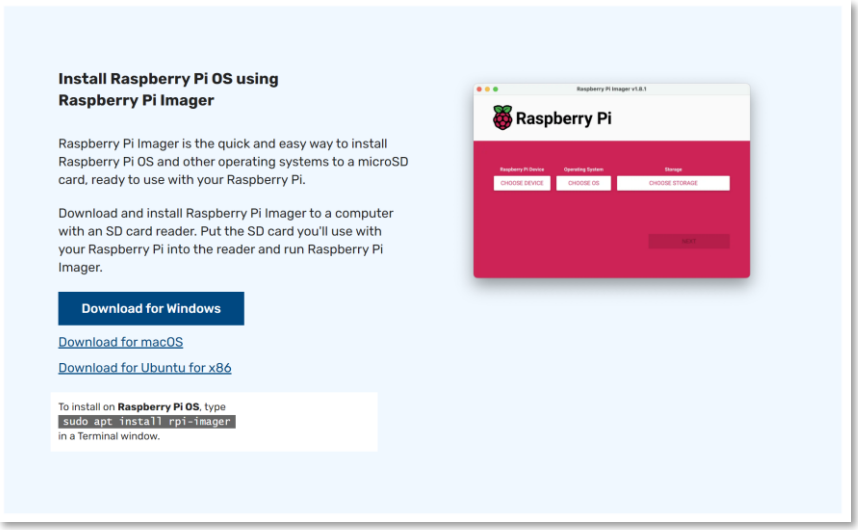


Figure 1 下载 Imager

下载并安装 SD Card Formatter 软件，对 SD 卡进行格式化。（**使用 Windows 自带的格式化选项通常不能满足树莓派系统烧录需要**，会导致在下一步烧录中发生“校验失败”

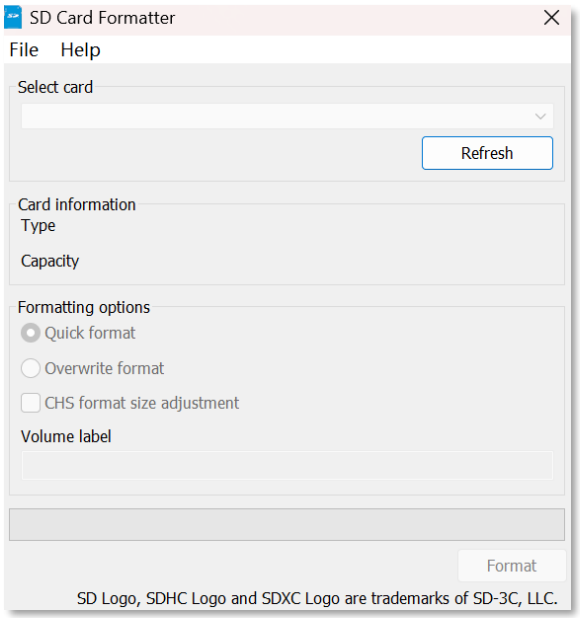


Figure 2 格式化 SD 卡

在 Raspberry Pi Imager 中，选择“Choose OS”并选取你下载的镜像文件，

然后选择你的 SD 卡。

烧录系统。之后，选择“弹出”。（不做可能导致系统损坏

3. 硬件接线与开机设置

插入 SD 卡到 Raspberry Pi 的卡槽，连接 USB 供电线和 HDMI 数据线。接通电源，Raspberry Pi 启动并在屏幕上显示图形界面，进入系统设置，开通所需的交互接口功能如 GPIO、SSH、VNC 等。

4. SSH 与 VNC 远程连接

4.1 SSH 远程连接

连接 Raspberry Pi 到 Wi-Fi 并查看其局域网 IP 地址。使用 SSH 客户端，（墙裂推荐 MobaXterm!! ）连接到 Raspberry Pi，输入 IP 地址，连接后输入用户名和密码登录。

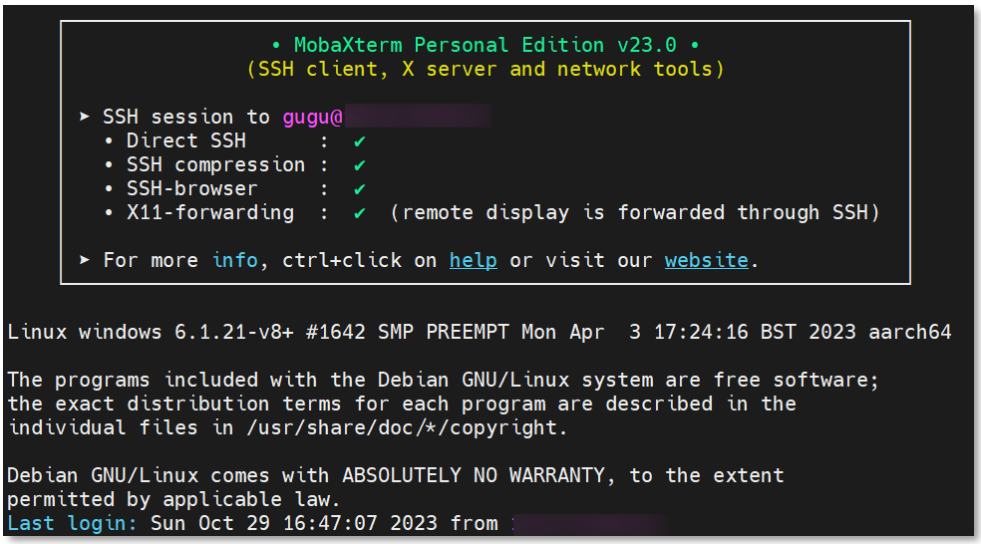


Figure 3 SSH 连接

4.2 VNC 虚拟远程桌面连接

使用 sudo apt install 安装 VNC Server，启动服务并配置账户，输入 Raspberry Pi 的 IP 地址和 VNC 服务用户名和密码，连接到其虚拟桌面。

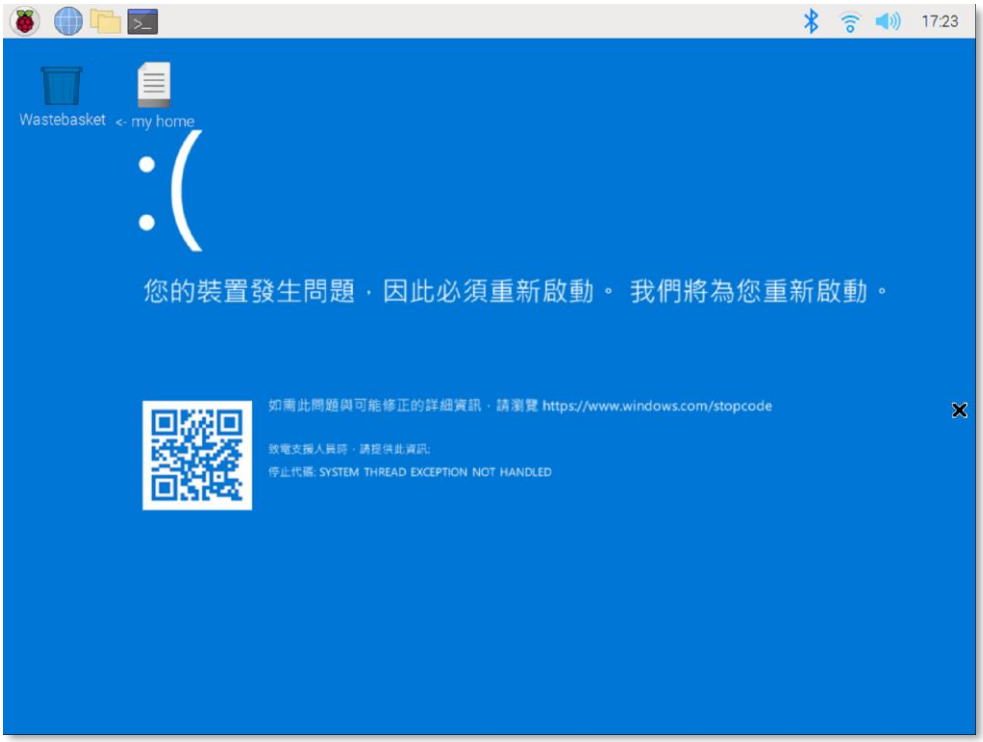


Figure 4 VNC 连接

4.3 使用 xrdp（可选）

在 Raspberry Pi 上运行 sudo apt-get install xrdp 安装远程桌面服务。

在 Windows 中搜索“远程桌面连接”，输入 Raspberry Pi 的 IP 地址、用户名和密码，连接到桌面。

实验二 开发环境配置与双色 LED 实验

一、树莓派开发环境配置

树莓派 GPIO 参考：

```
Shell
gpio readall
```

Figure 5 Shell 命令查看 GPIO 指示图

使用 Shell 命令查看树莓派标准及各个附加库对 GPIO 引脚的命名配置，如下图示：

```
gugu@windows:~ $ gpio readall
 +-----+-----+---------+------+---+---Pi 4B--+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 | ALT0 | 1 |  3 ||  4  |   |      | 5v      |     |     |
 |   3 |   9 |   SCL.1 | ALT0 | 1 |  5 ||  6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 0 |  7 ||  8  | 1 | ALT5 | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 ||  10 | 1 | ALT5 | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 ||  12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 ||  14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 1 | 15 ||  16 | 0 | OUT  | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 ||  18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI | ALT0 | 0 | 19 ||  20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO | ALT0 | 0 | 21 ||  22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK | ALT0 | 0 | 23 ||  24 | 1 | OUT  | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 ||  26 | 1 | OUT  | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 ||  28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 ||  30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 ||  32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 ||  34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 ||  36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 ||  38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 ||  40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 4B--+---+------+---------+-----+-----+
```

Figure 6 回显示意图

也可以使用 “pinout” 命令显示板上资源，如图所示：

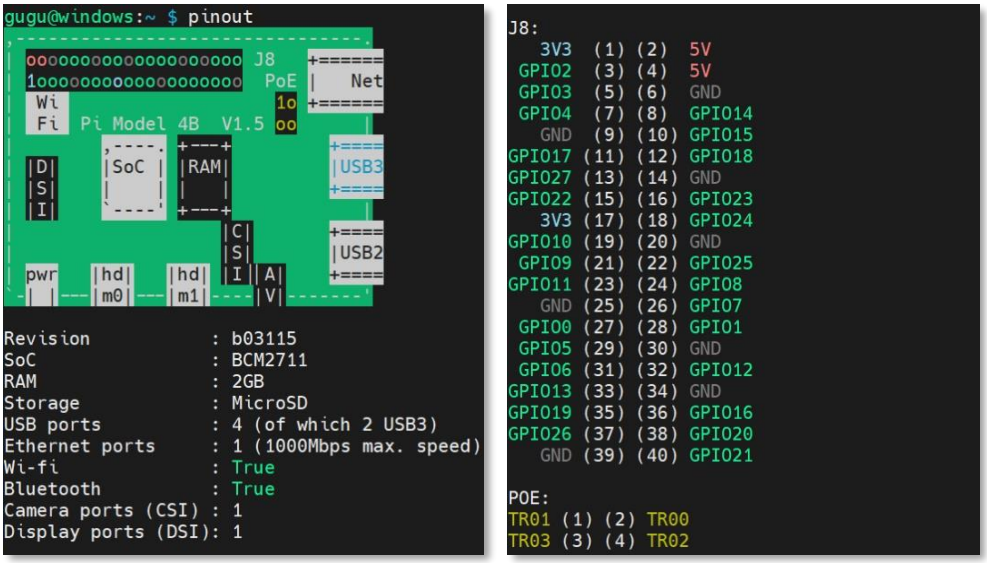


Figure 7 回显示意图

根据各方规定不同，板上引脚资源被赋予了不同的编号序列，在编程时需要使引脚编号正确对应。

目前，Raspberry Pi 有三种引脚编号方法，分别是：根据引脚的物理位置编号（Header）、由 C 语言 GPIO 库 wiringPi 指定的编号（wiringPi Pin）、由 BCM2837SOC 指定的编号（BCM GPIO）。

- 如果选择使用 C/C++编程，需要安装 wiringPi 库。首先，获取 https://project-downloads.drogon.net/wiringpi-latest.deb 并安装二进制 deb 文件。使用如下命令测试是否成功安装： gpio –v。
- 如果使用 Python 编程，可以使用 RPi.GPIO 提供的 API 对 GPIO 进行编程， 该软件包提供了一个类来控制 Raspberry Pi 上的 GPIO。Rspberry Pi 的 Raspbian OS 镜像中默认安装 RPi.GPIO，因此可以直接使用它。 如果需要使用，安装 python-dev 包即可。

双色 LED 实验

二、实验目的

本实验旨在通过使用双色 LED 模块，实现红绿两种颜色的交替闪烁效果，并控制 LED 模块的颜色变化。具体目标包括：

1. 了解双色 LED 模块的基本特性和工作原理。
2. 连接双色 LED 模块到树莓派的 GPIO 接口。
3. 使用树莓派的编程控制实现红绿双色的交替闪烁效果。

三、实验原理

本双色 LED 模块的工作原理如下：

1. 双色 LED 模块可以显示两种颜色，通常是红色和绿色。
2. 通过控制模块的引脚，可以选择点亮红色或绿色的 LED。
3. 交替控制引脚状态可以实现红绿两种颜色的交替闪烁效果。

在本实验中，我们将通过连接双色 LED 模块到树莓派的 GPIO 接口，并编写树莓派的程序，交替控制引脚状态，实现红绿两种颜色的交替闪烁效果。

Figure 8 双色 LED 灯

四、实验记录

1. 实验具体进行过程

1. 连接硬件：

将双色 LED 模块的 S 引脚连接到树莓派的 GPIO 口，GND 引脚连接到树莓派的 GND。

2. 编写控制器代码：

a. 编写树莓派的 GPIO 口控制代码。

b. 配置两个 GPIO 口为输出模式。

c. 编写程序循环控制两个引脚状态，实现红绿双色的交替闪烁效果。

2. 程序框图

Figure 9 程序框图

3. 实验具体实现代码

```
print('hi')

import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(13, GPIO.OUT)
GPIO.setup(19, GPIO.OUT)

while True:
    GPIO.output(19, GPIO.LOW)
    GPIO.output(13, GPIO.HIGH)
    time.sleep(0.25)
    GPIO.output(13, GPIO.LOW)
    GPIO.output(19, GPIO.HIGH)
    time.sleep(0.25)
```

Figure 10 实验代码

实验三 轻触按键开关实验

一、实验目的

本实验旨在使用轻触开关模块作为输入设备，学习如何在树莓派上检测按键的状态，并通过 LED 指示按键的不同状态。具体目标包括：

1. 了解轻触开关模块的基本特性和工作原理。
2. 配置树莓派的 GPIO 口为输入模式，用于检测轻触开关的状态。
3. 使用 LED 灯指示按键的不同状态。

二、实验原理

轻触开关模块的工作原理如下：

1. 轻触开关模块内部有一个轻触开关（按键开关）。
2. 当按下按键时，S 引脚输出低电平。
3. 当松开按键时，S 引脚输出高电平。

在本实验中，我们将使用树莓派的 GPIO 口检测轻触开关的 S 引脚状态，并使用 LED 灯指示按键的状态。

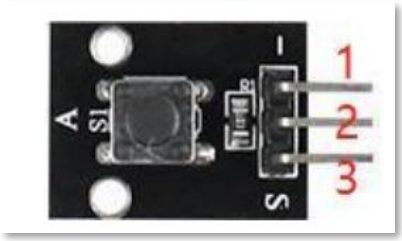


Figure 11 按键开关

三、实验记录

1. 实验具体进行过程

1. 连接硬件：

将轻触开关模块的 S 引脚连接到树莓派的 GPIO 口，VCC 引脚连接到树莓派的 5V 电源，GND 引脚连接到树莓派的 0V。

2. 编写控制器代码：

a. 编写树莓派的 GPIO 口控制代码。

b. 配置 GPIO 口为输入模式。

c. 不断检测 S 引脚的状态，当检测到低电平时，表示按键被按下。

d. 设置状态机，依次循环 LED 的灯光模式。

2. 程序框图

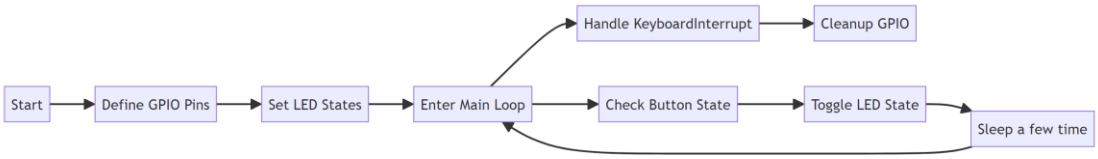


Figure 12 程序框图

其中，我们实现了 LED“颜色”与“是否闪烁”的循环切换。

3. 实验具体实现代码

```python
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

button_pin = 26
led_pin_red = 19
led_pin_green = 13

GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(led_pin_red, GPIO.OUT)
GPIO.setup(led_pin_green, GPIO.OUT)

LED_OFF = 0
LED_RED_ON = 1
LED_RED_BLINK = 2
LED_GREEN_ON = 3
LED_GREEN_BLINK = 4

current_led_state = LED_OFF
GPIO.output(led_pin_red, GPIO.LOW)
GPIO.output(led_pin_green, GPIO.LOW)

now_time = time.perf_counter()

print("Init!")
```

```python
def toggle_led_state():
    global current_led_state
    if current_led_state == LED_OFF:
        current_led_state = LED_RED_ON
        GPIO.output(led_pin_red, GPIO.HIGH)
    elif current_led_state == LED_RED_ON:
        current_led_state = LED_RED_BLINK
        now_time = time.perf_counter()
        # blink_led(led_pin_red)
    elif current_led_state == LED_RED_BLINK:
        current_led_state = LED_GREEN_ON
        GPIO.output(led_pin_red, GPIO.LOW)
        GPIO.output(led_pin_green, GPIO.HIGH)
    elif current_led_state == LED_GREEN_ON:
        current_led_state = LED_GREEN_BLINK
        # blink_led(led_pin_green)
        now_time = time.perf_counter()
    elif current_led_state == LED_GREEN_BLINK:
        current_led_state = LED_OFF
        GPIO.output(led_pin_red, GPIO.LOW)
        GPIO.output(led_pin_green, GPIO.LOW)

def blink_led(led_pin):
    for _ in range(10):
        GPIO.output(led_pin, GPIO.HIGH)
        time.sleep(0.5)
        GPIO.output(led_pin, GPIO.LOW)
        time.sleep(0.5)
```

```
try:
    while True:
        button_state = GPIO.input(button_pin)
        if button_state == GPIO.LOW:
            time.sleep(0.005)
            button_state = GPIO.input(button_pin)
            if button_state == GPIO.LOW:
                toggle_led_state()

        if current_led_state == LED_RED_BLINK:
            last_time = now_time
            now_time = time.perf_counter()
            if (now_time - last_time)*1000 > 250:
                GPIO.output(led_pin_red, not GPIO.input(led_pin_red))
        elif current_led_state == LED_GREEN_BLINK:
            last_time = now_time
            now_time = time.perf_counter()
            if (now_time - last_time)*1000 > 250:
                GPIO.output(led_pin_green, not GPIO.input(led_pin_green))

        time.sleep(0.3)

except KeyboardInterrupt:
    pass

GPIO.cleanup()
```

Figure 13 实验代码

实验四 PCF8591 模数转换器实验

一、实验目的

本实验旨在通过使用 PCF8591 模数转换器，控制 LED 灯的亮度，以理解 PCF8591 的工作原理和使用方法。具体目标包括：

1. 了解 PCF8591 模数转换器的基本特性和功能。
2. 掌握 I2C 总线通信协议，实现与 PCF8591 的通信。
3. 使用电位器模拟输入和模拟输出功能，调节 LED 灯的亮度。

二、实验原理

PCF8591 模数转换器的工作原理如下：

1. I2C 总线系统中的每个 PCF8591 设备通过向该设备发送有效地址而被激活，地址由固定部分和可编程部分组成。可编程部分由地址引脚 A0、A1、A2 设置。
2. 在 I2C 总线协议中，地址字节的最后一个位是读/写位，用于设置数据传输的方向。（实际情况似乎在前面补了一位 0 而没有把 R/W 算入
3. 发送到 PCF8591 设备的第二个字节将被存储在其控制寄存器中，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，下半字节选择一个模拟输入通道。
4. 在本实验中，AIN0（模拟输入 0）端口用于接收电位器模块的模拟信号，AOUT）用于将模拟信号输出到 LED 模块，以改变 LED 的亮度。
5. PCF8591 模块还带有光电二极管和负温度系数（NTC）热敏电阻，可以用

于光强和温度感知。

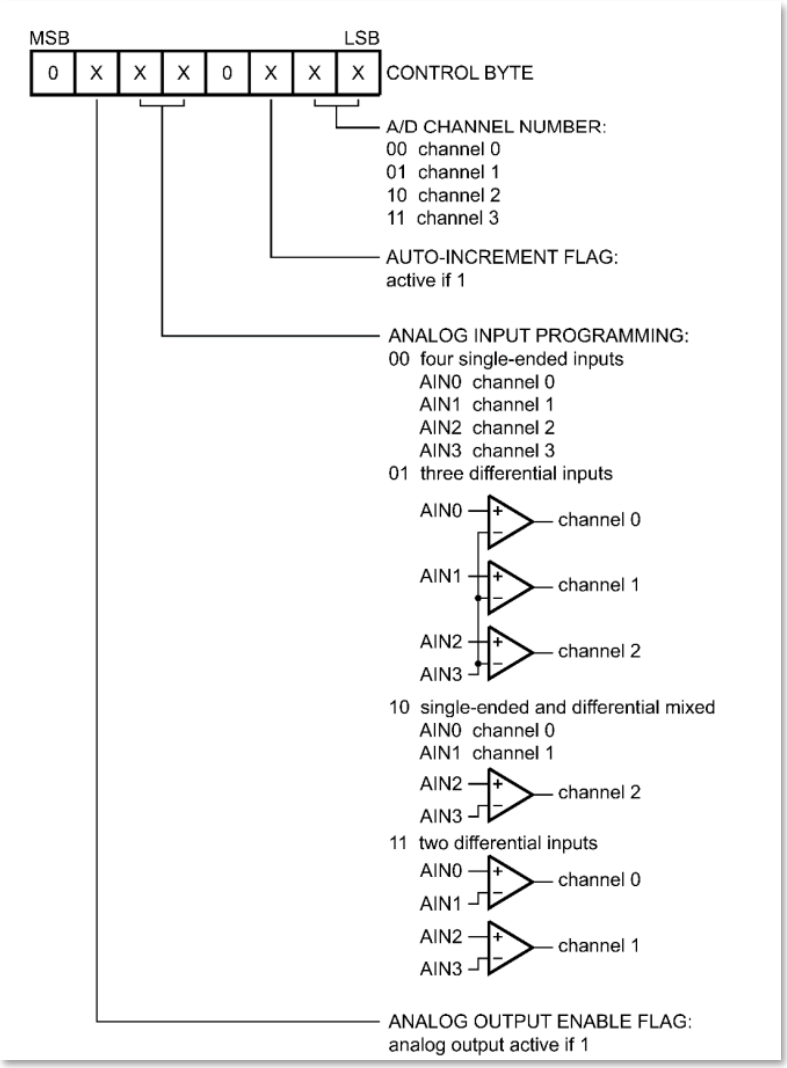


Figure 14 控制字节

三、实验记录

1. 实验具体进行过程

1. 连接 PCF8591 模块：

   a. 将 PCF8591 模块连接到 I2C 总线。

   b. 设置 PCF8591 的硬件地址。

2. 编写控制器代码：

根据控制器的类型，编写代码以通过 I2C 总线与 PCF8591 通信。

3. 实验步骤：

a. 启动实验装置和控制器。

b. 发送有效地址和控制字节以配置 PCF8591，选择模拟输入通道。

c. 读取 AIN0 端口的模拟输入信号。

d. 根据模拟输入信号，调节 LED 灯的亮度，将模拟输出信号输出到 AOUT 端口。

2. 程序框图

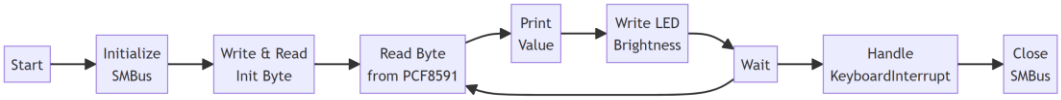


Figure 15 程序框图

3. 实验具体实现代码

```
import smbus
import time

bus = smbus.SMBus(1)

bus.write_byte(0x48, 0x40)
bus.read_byte(0x48)

while True:
    try:
        value = bus.read_byte(0x48)
        print('value:', value)
        bus.write_byte_data(0x48, 0x40, value)
        time.sleep(0.5)

    except KeyboardInterrupt:
        break

bus.close()
```

Figure 16 实验代码

实验五 模拟温度传感器实验

一、实验目的

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

温度传感器模块由 NTC 热敏电阻和定值电阻组成。热敏电阻的电阻值随环境温度变化而变化，使用 Steinhart-Hart 方程来得出热敏电阻的精确温度。

在本实验中，我们使用模数转换器 PCF8591 将模拟信号转换为数字信号。模拟温度传感器的原理图如下图所示：

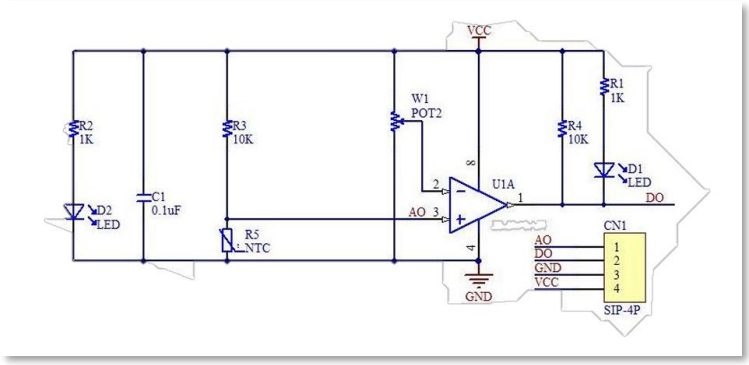


Figure 17 原理图

具体温度计算步骤如下：

1. 通过函数读取得到传感器模拟输出通过 A/D 转化后的数字值：

   analogVal = PCF8591.read(0)

2. 利用上面的值计算热敏电阻的原始模拟电压值 Vr：

   *Vr = 5 * float(analogVal) / 255*

3. 从上面的电路图可知，R3 与 R5 串联，所以电流值相等，R 电阻阻值为 10K， 即 10000，所以热敏电阻值:

   *Rt = 10000 * Vr / (5 - Vr)*

4. Steinhart-Hart 方程，热敏电阻的阻值为：

$$R = R_0 exp\,[B(\frac{1}{T} - \frac{1}{T_0})]$$

T_0: Standard Temperature 298.15 K (25 °C)
R_0: Resistance at T_0 K
B: Thermistor Constant (K)

注意：由于温度传感器型号未知，在这次实验中，*R*0 的值暂时选取为 10k，B 的值为 3950。

我们已经测出 R 的值，即为 Rt。根据 Steinhart-Hart 方程即可推导出温度 T 的表达式。

三、实验记录

1. 实验具体进行过程

首先，根据端口示意图对树莓派和温度传感器进行连线；其次，在树莓派文件系统内进行编程，验证并运行代码，在屏幕上打印出当前的环境温度。

在此之后，实现**附加功能**：检验当前温度是否达到 40 摄氏度并据此控制 LED 亮灭，实验中要求采用 DO 口进行控制，由电路原理可知，为了完成这个任务，只需把 DO 端口对接 LED 的供电端口，通过调节模块电阻值大小，实现恰好在 40 摄氏度时熄灭 LED。

2. 实验具体实现代码（程序思路和流程参见详细备注）

```
import smbus
import math
import time

# 定义PCF8591的I2C地址和通道
pcf8591_address = 0x48
analog_channel = 0

# 初始化I2C总线
bus = smbus.SMBus(1)  # 使用I2C总线1

# 定义Steinhart-Hart方程的参数
R0 = 10000  # 标准温度下的电阻值 (10kΩ)
B = 3950    # 热敏电阻的热敏常数

while True:
    try:
        # 读取传感器的模拟输出通过A/D转换后的数字值
        analog_val = bus.read_byte_data(pcf8591_address, analog_channel)

        # 计算热敏电阻的原始模拟电压值
        Vr = 5.0 * float(analog_val) / 255

        # 计算热敏电阻值
        Rt = 10000 * Vr / (5 - Vr)

        # 使用Steinhart-Hart方程计算温度
        T0 = 298.15  # 标准温度 (25°C)
        T = 1 / (1 / T0 + (1 / B) * math.log(Rt / R0))

        # 温度转换为摄氏度
        temperature_celsius = T - 273.15

        # 打印温度值
        print("Now Temp: {:.2f} *C".format(temperature_celsius))

        time.sleep(0.5)

    except KeyboardInterrupt:
        break
```

Figure 18 实验代码

实验六 超声波传感器实验

一、实验目的

本实验旨在通过超声波传感器测距模块，掌握超声波传感器的触发和接收信号处理方法，学习如何使用超声波测量非接触距离，并理解超声波测距的原理和工作过程。

本次实验的任务为：**获取障碍物与超声波传感器的距离**。

二、实验原理

超声波测距模块的工作原理如下：

1. 控制器给 Trig 引脚至少 10μs（10 微秒）的高电平信号，触发传感器开始工作。
2. 超声波传感器经过一定的延时（2ms）后，发送 8 个 40KHz 的超声波脉冲信号。同时，Echo 引脚变为高电平。
3. 当检测到有回声信号返回或大于 38ms 时，Echo 引脚变为低电平。
4. 测试距离=（高电平时间*声速）/2

Figure 19 超声波模块

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、实验记录

1. 实验具体进行过程

1. 连接超声波测距模块：

a. 将 VCC 引脚连接到 5V 电源。

b. 将 Trig 引脚连接到控制器的触发引脚。

c. 将 Echo 引脚连接到控制器的接收引脚。

d. 将 GND 引脚连接到地。

2. 编写控制器代码：根据控制器的类型，编写相应的代码以触发 Trig 引脚并测量 Echo 引脚的高电平时间。

3. 实验步骤：

a. 启动实验装置和控制器。

b. 控制器发送高电平信号给 Trig 引脚，触发超声波传感器工作。

c. 记录 Echo 引脚的高电平时间。

d. 使用上述原理计算距离并打印结果。

2. 程序框图

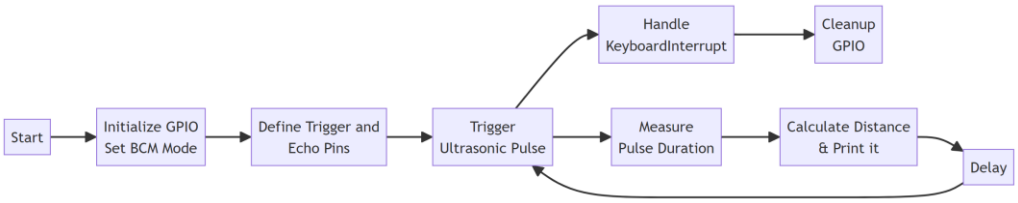


Figure 20 程序框图

3. 实验具体实现代码

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

trigger_pin = 17
echo_pin = 18

print('1')

GPIO.setup(trigger_pin, GPIO.OUT)
GPIO.setup(echo_pin, GPIO.IN)

try:
    while True:
        GPIO.output(trigger_pin, GPIO.HIGH)
        time.sleep(0.00001)
        GPIO.output(trigger_pin, GPIO.LOW)

        start_time = time.time()
        end_time = time.time()

        while GPIO.input(echo_pin) == 0:
            start_time = time.time()

        while GPIO.input(echo_pin) == 1:
            end_time = time.time()

        pulse_duration = end_time - start_time
        distance = (pulse_duration * 34300) / 2

        print("d: {:.2f} cm".format(distance))
        time.sleep(1)

except KeyboardInterrupt:
    print("Program terminated by user.")
    GPIO.cleanup()
```

Figure 21 实验代码

实验七 蜂鸣器实验

一、实验目的

本实验旨在通过使用蜂鸣器模块，播放一段音乐。具体目标包括：

1. 了解蜂鸣器模块的基本特性和工作原理。
2. 配置树莓派的 GPIO 口来驱动蜂鸣器模块。
3. 编写程序以产生不同频率的脉冲信号，模拟音乐的声音。

二、实验原理

1. 蜂鸣器模块的工作原理

a. 有源蜂鸣器： 有源蜂鸣器内部包含一个振荡电路，可以将恒定的直流电源转化为一定频率的脉冲信号。这个振荡电路产生的脉冲信号的频率是固定的，一般为单一音调。有源蜂鸣器的工作原理比较简单，它通过输入一个低电平信号（通常为低电平触发），就可以发出特定频率的声音。在实验中，只需要控制蜂鸣器的开关状态即可发出声音，频率不可调。

b. 无源蜂鸣器： 无源蜂鸣器内部没有振荡电路，它需要接在特定的音频输出电路中才能发出声音。为了发出声音，需要通过脉冲宽度调制（PWM）或脉冲频率调制（PFM）的方式控制蜂鸣器的工作。在本实验中，我们使用的是 PFM 方式。

2. 脉冲频率调制（PFM）的工作原理

脉冲频率调制是一种调制方式，它通过改变脉冲信号的频率来模拟声音的音调。具体原理如下：

- 方波信号： PFM 方式通常使用方波信号，即信号以高电平和低电平交替的方式变化。
- 频率控制： 在改变方波频率的同时固定方波脉冲的宽度度（一般高电平宽度占空比（Duty）为 50%）。
- 音符与频率： 音乐中的音符与信号的频率之间存在对应关系。不同音符对应不同的频率，例如 C4 音符对应的频率是 261.63 Hz。

在蜂鸣器实验中，我们将使用树莓派的 GPIO 口来控制蜂鸣器模块。通过编写程序，在 GPIO 口产生不同频率的信号，就可以模拟不同音符的声音，从而播放音乐。

三、实验记录

1. 实验具体进行过程

1. 连接硬件，将无源蜂鸣器模块连接到树莓派的 GPIO 口。

2. 编写控制器代码：

    a. 编写程序以产生不同频率的脉冲信号，模拟音乐的声音。

    b. 可以使用音符的频率值来定义不同的音调。

    c. 将一段音乐处理为序列化的音符并储存为文本。

3. 实验步骤：

a. 启动树莓派。

b. 执行编写的程序，读取音符序列文件

c. 播放音乐

2. 程序框图

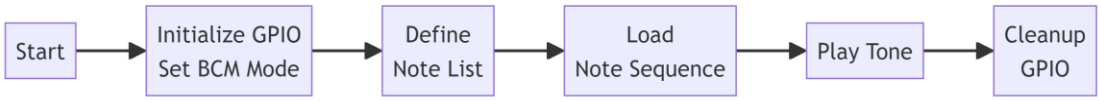


Figure 22 程序框图

3. 实验具体实现代码

```
import RPi.GPIO as GPIO
import time

buzzer_pin = 18

notes = {...}

def read_music_sequence(filename):
    with open(filename, 'r') as file:
        music_sequence = file.read().split()
    return music_sequence

def play_tone(note, duration):
    if note == 'P':
        time.sleep(duration)
        return
    p = GPIO.PWM(buzzer_pin, notes[note])
    p.start(50)
    time.sleep(duration)
    p.stop()
    time.sleep(0.01)

GPIO.setmode(GPIO.BCM)
GPIO.setup(buzzer_pin, GPIO.OUT)

music_sequence = read_music_sequence('musicsq.txt')

for note_duration in music_sequence:
    note, duration = note_duration.split(':')
    duration = float(duration)
    play_tone(note, duration)

GPIO.cleanup()
```

Figure 23 实验代码

实验八 PS2 操纵杆实验

一、实验目的

本实验旨在通过使用 PS2 操纵杆模块，控制不同的 LED 及其亮度变化。具体目标包括：

1. 了解 PS2 操纵杆模块的基本特性和工作原理。
2. 连接 PS2 操纵杆模块到树莓派，并使用模数转换模块（PCF8591）将模拟输出转换为数字量。
3. 编写树莓派的程序以检测操纵杆的移动位置。
4. 根据操纵杆的移动状态，控制不同的 LED 颜色亮度。

二、实验原理

在本实验中，我们使用 PS2 操纵杆模块，它包含两个模拟输出（X 和 Y 坐标）和一个数字输出（按钮按下状态）。操纵杆内部实际上是两个滑动变阻器，它们连接到 X 和 Y 方向上，产生模拟输出电压。

实验的主要原理如下：

1. 连接 PS2 操纵杆模块到树莓派的模数转换模块（PCF8591）。
2. 配置树莓派的 GPIO 口和 I2C 总线以与 PCF8591 通信。
3. 通过读取 PCF8591 模块的两个通道的数值，可以获取操纵杆在 X 和 Y 方向上的位置信息。这些数值在静止位置约为 128，根据移动操纵杆的方向，数值会在 0 到 255 之间变化。

根据操纵杆的位置，编写树莓派的程序来控制 LED 的亮度和颜色，从而实

现不同 LED 的控制效果。

三、实验记录

1. 实验具体进行过程

1. 连接硬件：

a. 将 PS2 操纵杆模块的 X 和 Y 输出连接到 PCF8591 模块的输入端口。

b. 配置树莓派的 GPIO 口和 I2C 总线。

2. 编写控制器代码：

a. 读取 PCF8591 模块的两个通道的数值，获取操纵杆的位置信息。

b. 根据操纵杆的位置和按钮的状态，控制 GPIO 口输入的 PWM 值，从而控制不同颜色 LED 的亮度。

3. 实验步骤：

a. 启动树莓派和控制器。

b. 执行编写的程序，监测操纵杆的位置状态，并相应地控制 LED。

2. 程序框图

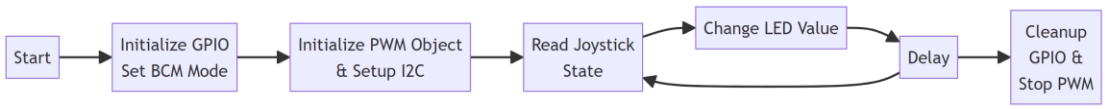


Figure 24 程序框图

3. 实验具体实现代码

```python
import RPi.GPIO as GPIO
import smbus
import time

GPIO.setmode(GPIO.BCM)
x_led_pin = 18
y_led_pin = 17

GPIO.setup(x_led_pin, GPIO.OUT)
GPIO.setup(y_led_pin, GPIO.OUT)
x_pwm = GPIO.PWM(x_led_pin, 100)
y_pwm = GPIO.PWM(y_led_pin, 100)

x_pwm.start(0)
y_pwm.start(0)

bus = smbus.SMBus(1)
pcf8591_address = 0x48

def read_ps2_joystick():
    x_value = bus.read_byte_data(pcf8591_address, 1)
    y_value = bus.read_byte_data(pcf8591_address, 0)
    return x_value, y_value

try:
    while True:
        x, y = read_ps2_joystick()

        x_duty_cycle = (x / 255.0) * 100
        x_pwm.ChangeDutyCycle(x_duty_cycle)
        y_duty_cycle = (y / 255.0) * 100
        y_pwm.ChangeDutyCycle(y_duty_cycle)

        # print(f"X: {x}, Y: {y}")
        time.sleep(0.1)

except KeyboardInterrupt:
    x_pwm.stop()
    y_pwm.stop()
    GPIO.cleanup()
    pass
```

Figure 25 实验代码

实验九 红外遥控实验

一、实验目的

本实验旨在通过使用红外遥控接收头和树莓派，实现对遥控器发射的红外信号的接收和解码，以识别遥控器按键并执行相应的操作。具体目标包括：

1. 理解红外遥控技术的基本原理和工作方式。
2. 配置树莓派以与红外接收头协同工作，以捕获红外信号。
3. 通过 lirc 库将接收到的红外信号转化为对应的按键信息。
4. 探索红外通信系统的应用和潜在用途。

二、实验原理

1. 红外遥控技术：红外遥控技术是一种利用红外光信号进行通信和控制的技术。它的基本原理是利用红外光的特性，即人眼无法看到红外光，但红外光在一定距离内可以传播，从而实现了遥控通信的目的。在这个实验中，我们使用红外遥控器来发送经过调制的红外信号，以便控制目标设备，例如树莓派。

Figure 26 红外遥控

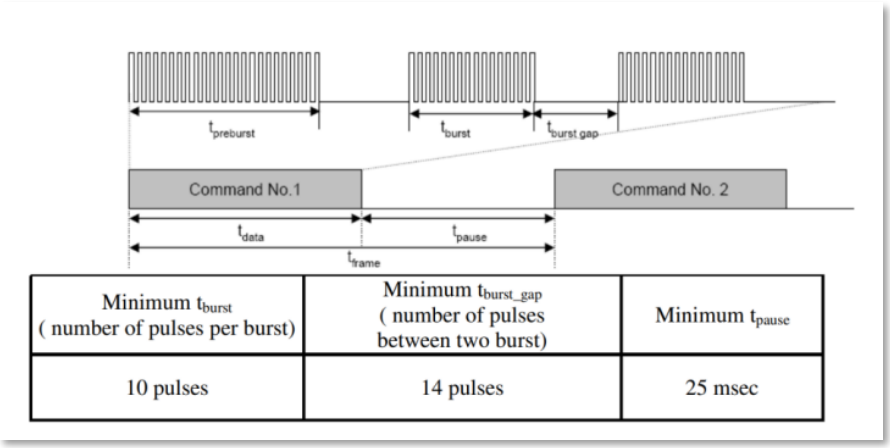
2. 遥控器接收头：遥控器接收头是实验中使用的红外接收设备。它通常包括以下组件：

- 红外二极管：用于接收红外信号，其特性是能够感知红外光。

- 前置放大器：用于放大接收到的弱红外信号，以便进一步处理。

- 解调电路：用于解调接收到的信号，将其转化为数字脉冲序列，以便进一步分析。

4. lirc 库：lirc（Linux Infrared Remote Control）库是一个用于 Linux 系统的红外遥控器接口库。它允许我们在 Linux 系统上轻松地捕获和解码红外信号，以便识别遥控器按键并执行相应的操作。
5. 配置树莓派：在本实验中，树莓派充当了红外信号的接收和处理设备。我们需要对树莓派进行配置，以确保它能够与红外接收头协同工作。这包括修改树莓派的引脚设置，以便正确连接到红外接收头，并修改驱动文件以指定使用哪个设备来接收红外信号。



Minimum t_{burst} (number of pulses per burst)	Minimum t_{burst_gap} (number of pulses between two burst)	Minimum t_{pause}
10 pulses	14 pulses	25 msec

Figure 27 IRM-2638 接收信号

6. 信号捕获和解码：当我们按下遥控器上的某个键时，遥控器会发射一连串经过调制的红外信号。这些信号由红外接收头接收，并通过前置放大器和解调电路进行处理。最终，接收头会输出解调后的数字脉冲序列，其中每个按键对应不同的脉冲序列。通过解码这些脉冲序列，我们可以识别出按下的是哪个按键，从而执行相应的操作。

三、实验记录

1. 实验具体进行过程。

a) 安装了 lirc 库以准备接收和解码红外信号。

b) 修改了配置文件 config.txt，配置了树莓派的引脚，以确保与红外接收头的连接正确。

c) 修改了驱动文件 lirc_options.conf，将驱动设定为默认，并指定了设备为/dev/lirc0。

d) 重启树莓派以应用配置更改。

e) 使用命令"irw"捕获接收到的原始红外数据，以便进一步分析。

f) 通过按下遥控器上的某个键，观察到一连串经过调制的信号，这些信号经过红外接收头后，输出解调后的数字脉冲。

g) 开启配置记录程序，记录具体的按钮对应的红外信号码值。

h) 对接收到的红外信号进行解码，以确定不同按键所对应的脉冲序列。

2. 程序框图

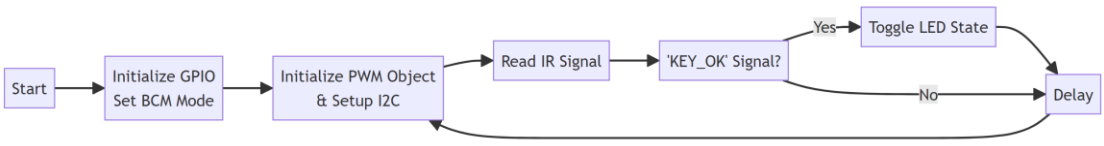


Figure 28 程序框图

3. *附加实验*：使用红外遥控器控制 LED 颜色切换

详情可见代码和程序框图，根据所接受的红外遥控信号，执行相应的任务指令。

示例中，演示了切换 LED 颜色的功能，根据需要，也可以使不同的遥控器按键对应不同的硬件功能，拓展使用场景。

```
import lirc
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(13, GPIO.OUT)
GPIO.setup(19, GPIO.OUT)

LED_OFF = 0
LED_RED_ON = 1
LED_GREEN_ON = 2

current_led_state = LED_OFF

def toggle_led_state():
    global current_led_state
    if current_led_state == LED_OFF:
        current_led_state = LED_RED_ON
        GPIO.output(13, GPIO.HIGH)
        GPIO.output(19, GPIO.LOW)
    elif current_led_state == LED_RED_ON:
        current_led_state = LED_GREEN_ON
        GPIO.output(19, GPIO.HIGH)
        GPIO.output(13, GPIO.LOW)
    elif current_led_state == LED_GREEN_ON:
        current_led_state = LED_OFF
        GPIO.output(13, GPIO.LOW)
        GPIO.output(19, GPIO.LOW)

try:
    with lirc.LircdConnection("irexec",) as conn:
        while True:
            string = conn.readline()
            print(string)
            if string == "echo \"KEY_OK\"":
                toggle_led_state()
                time.sleep(0.25)

except KeyboardInterrupt:
    pass
```

Figure 29 实验与附加实验代码