树莓派IO接口 实验报告

# 实验目的

* 熟悉树莓派的GPIO
* 掌握Python对GPIO的调用方式
* 了解中断的概念和编程

# GPIO介绍

## GPIO – General Purpose I/O

“通用目的输入/输出端口”,是一种灵活的软件控制的端口。通俗地说，就是一些引脚，可以通过它们输出高低电平或者通过它们读入引脚的状态-是高电平或是低电平。在嵌入式系统中，经常需要控制许多结构简单的外部设备或者电路，使用传统的串口或者并口就显得比较复杂，而GPIO解决了这个问题。

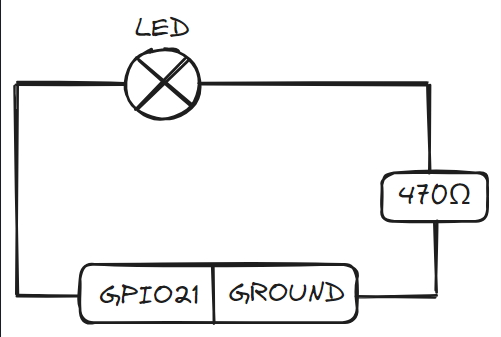
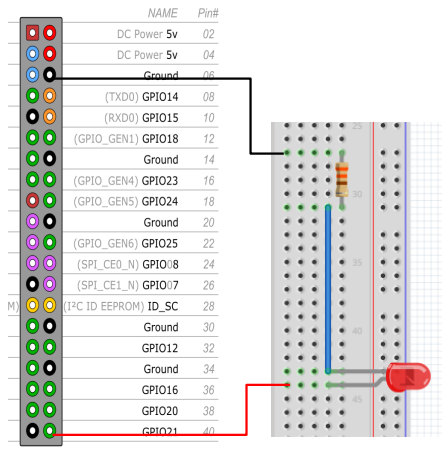
## 基于Python的树莓派GPIO开发方式

本课程的GPIO编程基于Rpi.GPIO库。Rpi.GPIO是一个小型的python库，非常简单好用，但是暂时还没有支持SPI、I2C或者1-wire等总线接口。

* 如果希望进行I2C的开发，请安装smbus库和I2C tools。
* 如果希望进行SPI开发，请安装spi-dev库。
* 如果希望进行串口开发，请安装PySerial库。

# 树莓派GPIO驱动LED

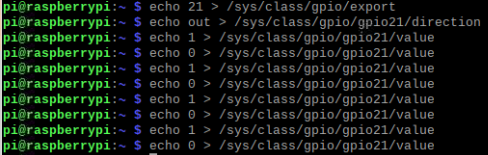
## 电路连接

* 在GPIO21和地线之间接 接入一个LED灯，还有一个用于防止短路的470欧电阻
* 红色直插LED的导通压 降为2.1V左右。 I=（3.3-2.2）V/470Ω ≈ 2mA
* 当GPIO21位于高电平时，将有电流通过电路，从而点亮LED灯  
  

## shell 方法

我们用shell命令来控制GPIO21。在Linux中，外部设备经常被表示成文件。向文件写入或读取字符，就相当于向设备输出或者从设备输入。树莓派上的GPIO端口也是如此，其代表文件位于/sys/class/gpio/下。

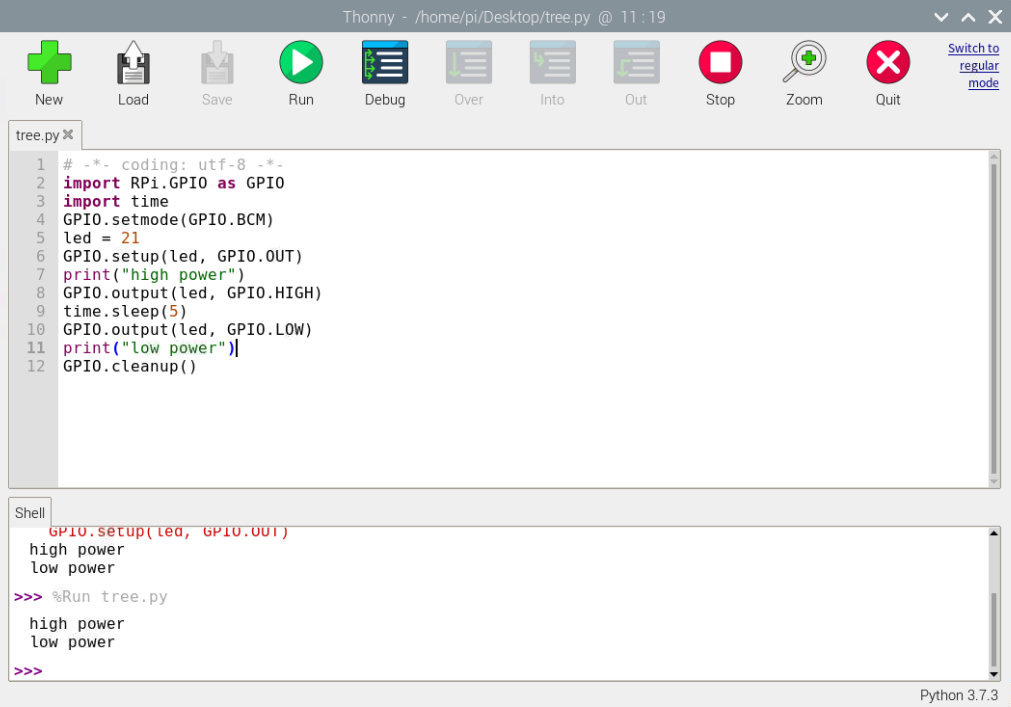
# 初始化GPIO21  
echo 21 > /sys/class/gpio/export  
# 执行后，/sys/class/gpio/下面增加了代表GPIO21的一个目录，目录名就是gpio21。  
  
# 下一步，把GPIO21置于输出状态：  
echo out > /sys/class/gpio/gpio21/direction  
  
# 最后，向GPIO21写入1，从而让PIN处于高电压——LED灯亮  
echo 1 > /sys/class/gpio/gpio21/value  
  
# 通过写入0让PIN处于低电压——LED灯灭  
echo 0 > /sys/class/gpio/gpio21/value  
  
# 使用完毕GPIO21，可以删除该端口  
echo 21 > /sys/class/gpio/unexport  
复制



## python方法

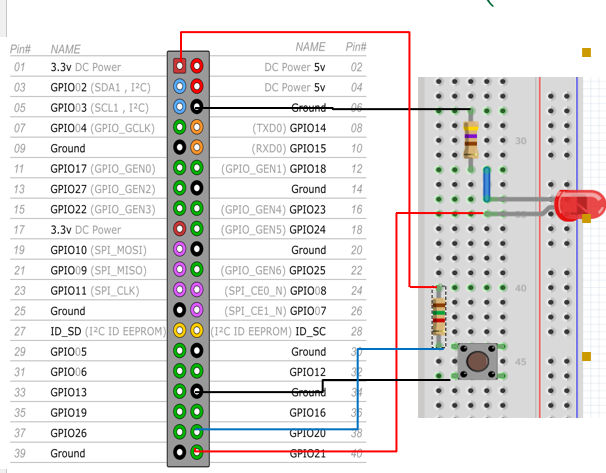
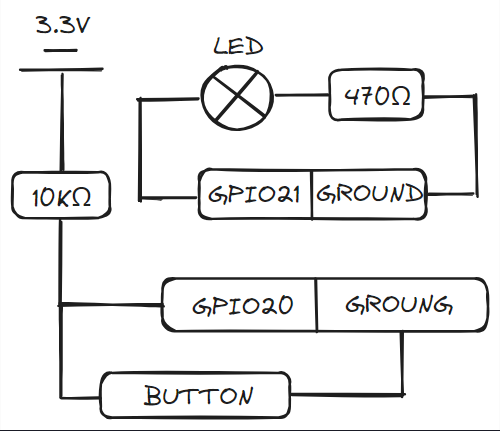
**使用Rpi.GPIO库**  
通常Rpi.GPIO已经包含在树莓派系统中。

# -\*- coding: utf-8 -\*-  
import RPi.GPIO as GPIO  
import time  
  
#首先调用GPIO.setmode函数来确定引脚的模式。  
# 在RPi.GPIO包中定义GPIO针脚的两种模式：BCM模式和BOARD模式。  
GPIO.setmode(GPIO.BCM)  
led = 21  
GPIO.setup(led, GPIO.OUT)  
print("输出高电平")  
GPIO.output(led, GPIO.HIGH)  
time.sleep(5)  
GPIO.output(led, GPIO.LOW)  
print("输出低电平")  
  
# GPIO.cleanup()函数清除掉之前GPIO.setup()设置的状态，恢复所有使用过的GPIO状态为输入，避免由于短路意外损坏树莓派。  
# 注意，该操作仅会清理你的代码使用过的GPIO通道。退出程序之前一定要调用，否则下次调用该GPIO的时候会报错。  
GPIO.cleanup()  
复制

**运行结果**  


# GPIO按键控制LED

## 电路连接

* 在GPIO驱动LED灯的电路基础上，加入一个按键开关，以及一个10kΩ电阻。
* 用 GPIO20 监测按键是否被按下，从而控制LED灯状态。
* 当按键没有按下时，GPIO20上的电平被上拉到3.3V，当按键按下时，GPIO20接地。  
    
  

## 轮询方法

* 轮询方法通过程序循环和延时读取获取引脚的状态及变化，以此判断按钮是否被按下
* 直接通过程序改变GPIO21的输出控制LED灯

### 代码

import RPi.GPIO as GPIO  
import time  
  
GPIO.setmode(GPIO.BCM)  
led = 21 # led由GPIO21控制  
bt = 20 # button状态由GPIO20读取  
GPIO.setup(led,GPIO.OUT) # GPIO21设置为输出模式  
GPIO.setup(bt,GPIO.IN,pull\_up\_down=GPIO.PUD\_UP) # GPIO20设置为读入模式  
ledStatus=False  
n=1  
try: # 提供退出方法，键盘输入Ctrl+c即可终止循环  
 while True:  
 time.sleep(0.01)  
 if(GPIO.input(bt)==GPIO.LOW): # 读取到按键按下  
 time.sleep(0.03) # 睡0.03秒  
 if(GPIO.input(bt)==GPIO.HIGH): # 如果此时读取到按键恢复  
 print('button pressed',n) # 输出语句  
 n=n+1  
 ledStatus=not ledStatus # 逆转ledStatus  
 if ledStatus: # 根据ledStatus设置GPIO21的输出电平，控制led亮灭  
 GPIO.output (led,GPIO.HIGH)  
 else:  
 GPIO.output (led,GPIO.LOW)  
except KeyboardInterrupt:  
 pass  
GPIO.cleanup() # 清除setup的配置  
复制

## 中断方法

* 中断的概念  
  CPU在运行目前任务时，当有特别事件发生时，CPU应当暂停正在执行的程序，转向执行处理该事件的子程序；事件处理完毕后，恢复原来的状态，再继续执行原来的程序。这种对这些事件的处理模式，称为程序中断。
* 通过设计一个事件函数来提供中断后的输出和LED控制
* 通过为GPIO20（bt）引脚添加一个事件检测来触发事件函数，检测基准是下降沿，即按钮被按下后接地
* 事件函数将在另一个线程执行，使得反应及时

### 代码

# -\*- coding: utf-8 -\*-  
import RPi.GPIO as GPIO  
import time  
# 同上初始化  
GPIO.setmode(GPIO.BCM)  
led = 21  
bt = 20  
GPIO.setup(led,GPIO.OUT)  
GPIO.setup(bt,GPIO.IN,pull\_up\_down=GPIO.PUD\_UP)  
ledStatus=True  
  
def my\_callback(channel): # 事件函数/回调函数  
 print("button pressed") # 输出  
 global ledStatus  
 ledStatus = not ledStatus   
 if ledStatus: # 控制LED  
 GPIO.output (led,GPIO.HIGH)  
 else:  
 GPIO.output (led,GPIO.LOW)  
   
# 添加一个事件检测，检测条件为GPIO20的下降沿，即按键被按下  
# 设置bouncetime参数为程序提供去抖  
GPIO.add\_event\_detect(bt,GPIO.FALLING,callback = my\_callback,bouncetime=200)  
  
try: # 循环输出并提供程序结束方法  
 while True:  
 print("I LOVE Raspberry Pi")  
 time.sleep(2)  
except KeyboardInterrupt:  
 pass  
  
GPIO.cleanup()  
复制

## 轮询式与中断式的主要区别

### 1.询问方式不同

* 轮询式是一种软件式的询问，需要CPU不断询问引脚状态来实现按键是否按下的判断
* 中断式是一种硬件式的询问，本质上，CPU通过读取外部信号来判断CPU的下一步状态，即未检测到中断请求时，CPU主进程并不主动询问，如本次代码中CPU实际上一直在循环输出语句"I LOVE Raspberry Pi"

### 2.处理方式（提供服务的方式）不同

* 轮询式的服务提供（改变LED灯的状态）直接由主进程提供，当轮询机制检测到按钮按下时，主程序的if语句被执行
* 中断式的则可以由另一个进程提供（如果是单线程就会中断主线程并执行约定的事件函数，而RPi.GPIO库会为回调函数另外开启一个线程），可以做到不影响主进程的运行

### 3.CPU消耗不同

* 轮询式由CPU软件机制判断事件，CPU消耗大，且浪费大量的处理器周期（大部分时间内检测结果都是无变化）
* 中断式由硬件机制判断事件，平时CPU并不监视按键状态，CPU消耗相对较小，效率高

# 故障排查方案

## 基本原理

依据上文给出的电路示意图，不难发现电路分为两个系统

* 由GPIO21提供的LED灯控制系统
* 由GPIO20提供的按键检测系统

## 排查思路——以轮询法为基础

### 首先检查LED灯控制系统的问题

考虑到轮询法中初始化为ledStatus=False,我们可以先使用Ctrl+C终止程序，将程序初始化为ledStatus=True再运行一次程序

* 如果LED灯亮，基本可以认定LED灯控制系统无问题
* 如果LED灯无反应，那么调整整个电路的连线，直到LED灯亮为止

### 其次检查按键控制系统的问题

考虑到轮询法中程序第一个if为检查按键是否按下，我们可以添加一个debug语句

while True:  
 time.sleep(0.01)  
 if(GPIO.input(bt)==GPIO.LOW): # 读取到按键按下  
 print("DEBUG Button down!") # DEBUG语句  
 time.sleep(0.03) # 睡0.03秒  
 if(GPIO.input(bt)==GPIO.HIGH): # 如果此时读取到按键恢复  
复制

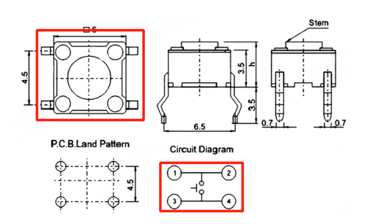
* 如果系统一切正常，那么理论上只有按键按下后，控制台不断输出DEBUG语句
* 如果3.3V电源接口连接有问题，那么控制台将直接不断输出DEBUG语句，这时应该调整3.3V电源和10kΩ电阻这条电路的连接状态，直到不再出现异常的DEBUG语句
* 如果发现按键按下后毫无反应，这时可能是按键或者连线的问题

### 最后排查按键和电路连接的问题

延续上一部分的DEBUG程序，我们可以直接用一根线模拟按键，强行让GPIO20接地

* 如果用线直连后控制台开始不断输出DEBUG语句，那么表明连线没有问题
* 如果用线直连后毫无反应，那么保留直连的线，调整其他连线直到控制台正常输出

如果确定是按键的问题，那么只需注意按键的方向：

* 引脚在左右两边时，按键按下1与3连通，2与4连通。  
    
  如果确定按键安装无误，那就只好换一个按键了

### 中断法的后续排查方案

由于中断法之前至少轮询法已经正常运行，所有基本不会出现什么问题  
唯一要注意的点是中断法需要一定时间长度的下降沿信号（按键被按下）来触发中断，所以可以适当延长按下按键的时间，如果又不行就再用电线直连