第 3 章

传感器接口与编程

本章从基本传感器模块入手，介绍树莓派 GPIO 接口与各种传感器的连接与编程，学习树莓派的硬件资源与接口设计。本章使用的各种传感器模块价格便宜，购买方便。

3.1 GPIO 接口简介

树莓派拥有 40 个可编程的 GPIO(通用输入/输出端口)引脚。GPIO 应用非常广泛，掌握了 GPIO 的使用，就掌握了树莓派硬件设计的能力。使用者可以通过 GPIO 输出高低电平来控制 LED、蜂鸣器、电动机等各种外设工作，也可以通过它们实现树莓派和外接传感器模块之间的交互。树莓派 3B/3B+的 GPIO 接口及引脚分布如图 3-1 所示，除了包括多个

BCM 编码	功能名	物理引脚 BOARD编码		功能名	BCM 编码
	3.3V	1	2	5V	
2	SDA.1	3	4	5V	
3	SCL.1	5	6	GND	
4	GPIO.7	7	8	TXD	14
	GND	9	10	RXD	15
17	GPIO.0	11	12	GPIO.1	18
27	GPIO.2	13	14	GND	
22	GPIO.3	15	16	GPIO.4	23
	3.3V	17	18	GPIO.5	24
10	MOSI	19	20	GND	
9	MISO	21	22	GPIO.6	25
11	SCLK	23	24	CE0	8
	GND	25	26	CE1	7
0	SDA.0	27	28	SCL.0	1
5	GPIO.21	29	30	GND	
6	GPIO.22	31	32	GPIO.26	12
13	GPIO.23	33	34	GND	
19	GPIO.24	35	36	GPIO.27	16
26	GPIO.25	37	38	GPIO.28	20
	GND	39	40	GPIO.29	21

图 3-1　树莓派 GPIO 引脚

5V、3.3V 以及接地引脚以外，还具有 I^2C、SPI 和 UART 接口等双重功能。需要说明的是，电源和接地引脚可用于给外部模块或元器件供电，但过大的工作电流或峰值电压可能会损坏树莓派。

GPIO 接口有以下两种常用的编码方式：

(1) BOARD 编码，按照树莓派主板上引脚的物理位置进行编号，分别对应 1～40 号引脚。

(2) BCM 编码，属于更底层的工作方式，它和 Broadcom 片上系统中信道编号相对应，在使用引脚时需要查找信道编号和物理引脚编号的对应关系。

树莓派操作系统里包含了 RPi.GPIO 库，使用该库可以指定 GPIO 接口的编码方式，代码如下：

```
import RPi.GPIO as GPIO                    #导入 RPi.GPIO 模块

GPIO.setmode(GPIO.BCM)                     #引脚采用 BCM 编码方式
GPIO.setmode(GPIO.BOARD)                   #引脚采用 BOARD 编码方式
```

使用 RPi.GPIO 库也可以轻松实现对 GPIO 引脚功能的设置，例如，

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BOARD)
GPIO.setup(11, GPIO.IN)                                    #将引脚 11 设置为输入模式
GPIO.input(11)                                             #读取输入引脚的值
GPIO.setup(12, GPIO.OUT)                                   #将引脚 12 设置为输出模式

#可以通过软件实现输入引脚的上拉/下拉
GPIO.setup(11, GPIO.IN, pull_up_down = GPIO.PUD_UP)        #输入引脚 11 上拉
GPIO.setup(11, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)      #输入引脚 11 下拉
'''也可以设置输出引脚的初始状态，输出引脚 12 的初始状态为高电平，状态可以表示为
0/GPIO.LOW/False 或者 1/GPIO.HIGH/True'''
GPIO.setup(12, GPIO.OUT, initial = GPIO.HIGH)
```

程序运行结束后，需要释放硬件资源，同时避免因意外损坏树莓派。使用 GPIO.cleanup()会释放使用的 GPIO 引脚，并清除设置的引脚编码方式。

串口配置与 GPS 模块编程

3.2　GPS 定位

3.2.1　树莓派串口配置

树莓派 3B/3B＋提供了两类串口，即硬件串口(/dev/ttyAMA0)和 mini 串口(/dev/ttyS0)。硬件串口由硬件实现，有单独的时钟源，性能高、工作可靠；而 mini 串口性能低，功

能简单,波特率受到内核时钟的影响。树莓派 3B/3B+新增了板载蓝牙模块,硬件串口被默认分配给与蓝牙模块通信,而把由内核提供时钟参考源的 mini 串口分配给了 GPIO 接口中的 TXD 和 RXD 引脚。在终端输入 **ls -l /dev**,查看当前的串口映射关系,如图 3-2 所示,UART 接口映射的串口 serial0 默认是 mini 串口。

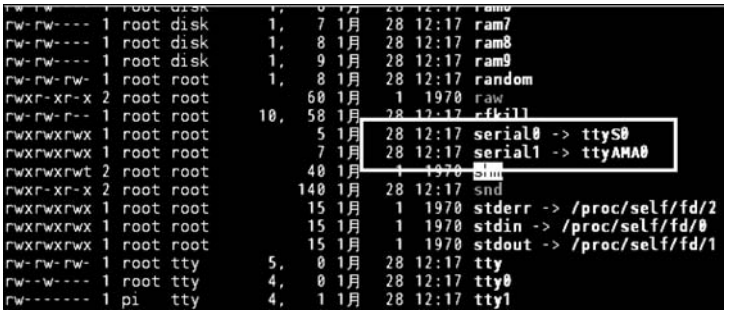


图 3-2 树莓派默认串口映射关系

由于 mini 串口速率不稳定,通过 UART 接口外接模块时可能会出现无法正常工作的情况。为了通过 GPIO 使用高性能的硬件串口,需要将树莓派 3B/3B+的蓝牙模块切换到 mini 串口,并将硬件串口恢复到 GPIO 引脚中,步骤如下:

(1) 终端输入 **sudo raspi-config**,如图 3-3 所示,依次选择 Interfacing Options→Serial 选项,回车后选择"否",禁用串口的控制台功能(树莓派 GPIO 引出的串口默认用来做控制台使用,需要禁用该功能,使得串口可以自由使用),随后选择"是",使能树莓派串口,如图 3-4 所示。

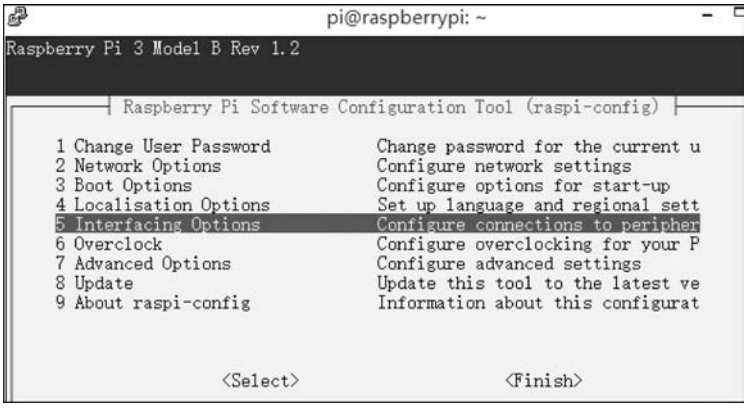


图 3-3 树莓派串口配置

(2) 终端输入 **sudo nano /boot/config.txt** 打开配置文件,在文件最后一行添加"dtoverlay=pi3-disable-bt"释放蓝牙占用的串口,保存后退出,重启树莓派使上述修改生效。

(3) 在终端输入 **ls -l /dev** 再次查看当前的串口映射关系,如图 3-5 所示,树莓派已经恢复了硬件串口与 GPIO 的映射关系。

图 3-4　使能树莓派串口

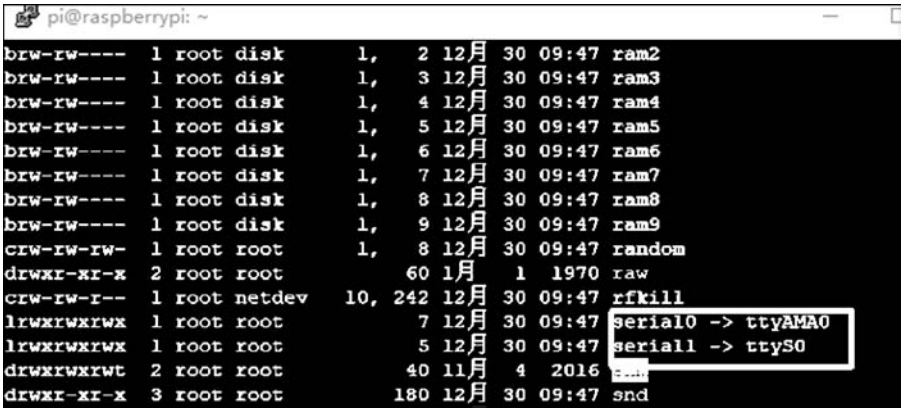


图 3-5　恢复硬件串口与 GPIO 的映射关系

注意：禁用串口的控制台功能也可以通过编辑 cmdline.txt 文件来实现。输入 sudo nano /boot/cmdline.txt 打开文件，将/dev/ttyAMA0 有关的配置去掉，例如，原 cmdline.txt 的内容为：dwc_otg.lpm_enable=0 console= ttyAMA0,115200 console=tty1 root=…，只需将其中的“console= ttyAMA0,115200”删掉即可。

3.2.2　GPS 模块接口与编程

选用的 GPS 模块型号为 ATGM336H，它基于中科微低功耗 GNSS SOC 芯片 AT6558，支持 GPS/BDS/GLONASS 卫星导航系统，具有高灵敏度、低功耗、低成本的特点。该模块供电电压为 3.3～5V，采用 IPX 陶瓷有源天线，定位精度 2.5m，冷启动捕获灵敏度为－148dBm，跟踪灵敏度为－162dBm，工作电流小于 25mA，通信方式为串口通信，波特率默认为 9600bps。GPS 模块及其与树莓派的引脚连接如图 3-6 所示，4 个引脚 VCC、

GND、TX 和 RX 分别与树莓派 GPIO 接口的 1 脚(3.3V)、6 脚(GND)、10 脚(RXD)和 8 脚(TXD)相连。

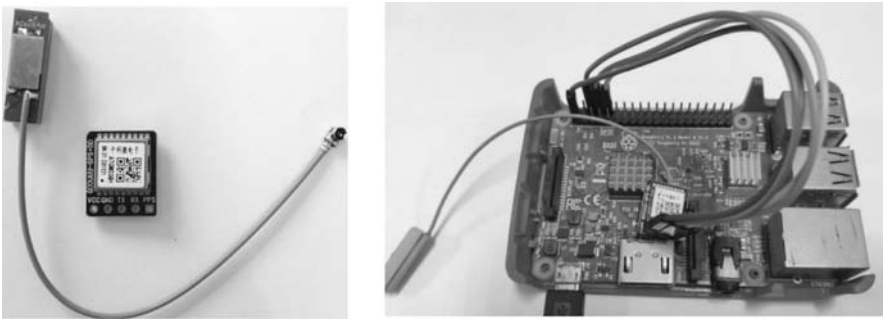


图 3-6　树莓派连接 GPS 模块

minicom 是运行在 Linux 系统下的轻量级串口调试工具,类似 Windows 系统中的串口调试助手,在命令行输入 **sudo apt-get install minicom** 即可完成安装。将 GPS 模块与树莓派连接好后,在终端输入 **minicom -b 9600 -D /dev/ttyAMA0** 打开 minicom 获取串口数据,其中-b 设定波特率,视模块参数而定; -D 指定的是接口。随后可以看到树莓派通过串口接收到 GPS 模块的定位数据,如图 3-7 所示。测试时需将 GPS 模块置于室外或者窗户边,有利于 GPS 搜星与定位。

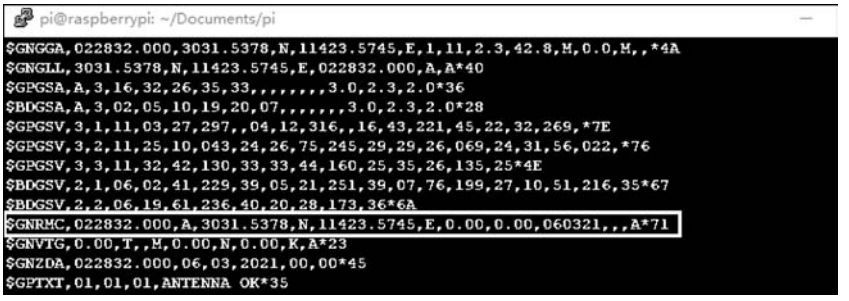


图 3-7　串口读取 GPS 模块的数据

GPS 模块按照 NMEA-0183 协议格式输出数据,包括 GPS 定位信息(GGA)、当前卫星信息(GSA)、可见卫星信息(GSV)、推荐定位信息(RMC)和地面速度信息(VTG)等内容。通常根据推荐定位信息($GNRMC 开头的数据行)来获取有用数据,$GNRMC 语句的基本格式与数据详解如图 3-8 所示。

以图 3-7 中标记的 $GNRMC 语句为例,选用其中<1><3><4><5><6><9>这 6 个数据项就可以得到时间和经纬度信息。从<9>和<1>可知,当前的时间是 2021 年 3 月 6 日 10 时 28 分 32 秒; <3>和<4>表明当前位置是北纬 30 度 31.5378 分,即北纬 30.525630 度(31.5378 分/60 可以转化为度); 类似地,<5>和<6>表明当前位置是东经 114 度 23.5745 分,即东经 114.392908 度(23.5745 分/60 转化为度)。为了从 GPS 原始数据中解析出时间

$GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>*hh
<1> UTC 时间,hhmmss.sss(时分秒)格式
<2> 定位状态,A=有效定位,V=无效定位
<3> 纬度 ddmm.mmmm(度分)格式
<4> 纬度半球 N(北半球)或 S(南半球)
<5> 经度 dddmm.mmmm(度分)格式,其中的分/60 可以转化为度
<6> 经度半球 E(东经)或 W(西经)
<7> 地面速率(000.0～999.9 节,前面的 0 也将被传输)
<8> 地面航向(000.0～359.9 度,以真北为参考基准)
<9> UTC 日期,ddmmyy(日月年)格式
<10> 磁偏角(000.0～180.0 度,前面的 0 也将被传输)
<11> 磁偏角方向,E(东)或 W(西)
<12> 模式指示(A=自主定位,D=差分,E=估算,N=数据无效),后面的 *hh 表示校验值

图 3-8　$GNRMC 语句的基本格式与数据详解

与位置信息,新建 gps_test.py 脚本文件,输入以下代码:

```
import serial #导入串口库,以便通过串口访问 GPS 模块
import time

ser = serial.Serial("/dev/ttyAMA0",9600) #使用/dev/ttyAMA0 建立串口,波特率 9600

def GPS():
    str_gps = ser.read(1200) #从串口读取 1200 字节数据,包括完整的 GPS 数据集合
    #转换成 UTF-8 编码输出,避免乱码
    str_gps = str_gps.decode(encoding = 'utf-8', errors = 'ignore')
    pos1 = str_gps.find("$GNRMC") #找到 $GNRMC 字符串首次出现的位置
    pos2 = str_gps.find("\n",pos1) #找到 $GNRMC 行的结尾处
    loc = str_gps[pos1:pos2]        #提取完整的 $GNRMC 行数据
    data = loc.split(",")          #以逗号为分隔符,将 $GNRMC 行进行分割,分解到 data 列表中

    if data[2] == 'V': #GPS 数据无效
            print("No location found")
    else:
        position_lat = float(data[3][0:2]) + float(data[3][2:9]) / 60.0 #计算纬度
        position_lng = float(data[5][0:3]) + float(data[5][3:10]) / 60.0 #计算经度
        time = data[1]
        time_h = int(time[0:2]) + 8 #调整为北京时间,北京时间 = UTC + 时区差 8
        time_m = int(time[2:4])
        time_s = int(time[4:6])
        print("纬度: %f %s" % (position_lat, data[4]))
        print("经度: %f %s" % (position_lng, data[6]))
        print("时间: %d h %d m %d s\n" % (time_h, time_m, time_s))
        #返回的经纬度只取小数点后面 6 位
```

```
            return [round(position_lng,6), data[6],round(position_lat,6), data[4]]

    if __name__ == "__main__":
        try:
            while True:
                GPS()
                time.sleep(5)
        except KeyboardInterrupt:    #按 Ctrl + C 组合键退出
            ser.close()              # 释放串口
```

此外,还可以通过第三方库访问 GPS。首先,运行命令 **sudo apt-get install gpsd gpsd-clients** 安装 gpsd 软件。接着输入 **sudo gpsd /dev/ttyAMA0 -F /var/run/gpsd.sock** 运行 gpsd 软件,该命令会开启一个后台程序并由它负责与 GPS 模块通信。然后输入命令 **cgps** 开启 GPS 客户端,显示 gpsd 接收的数据信息,如图 3-9 所示。如果运行软件时客户端一直没有数据显示或出现超时错误,则可以使用命令 **sudo systemctl stop gpsd.socket** 以及 **sudo systemctl disable gpsd.socket** 禁用 gpsd 系统服务,再执行命令 **sudo killall gpsd** 结束 gpsd 进程并通过命令 **sudo gpsd /dev/ttyAMA0 -F /var/run/gpsd.sock** 重启 gpsd 软件。

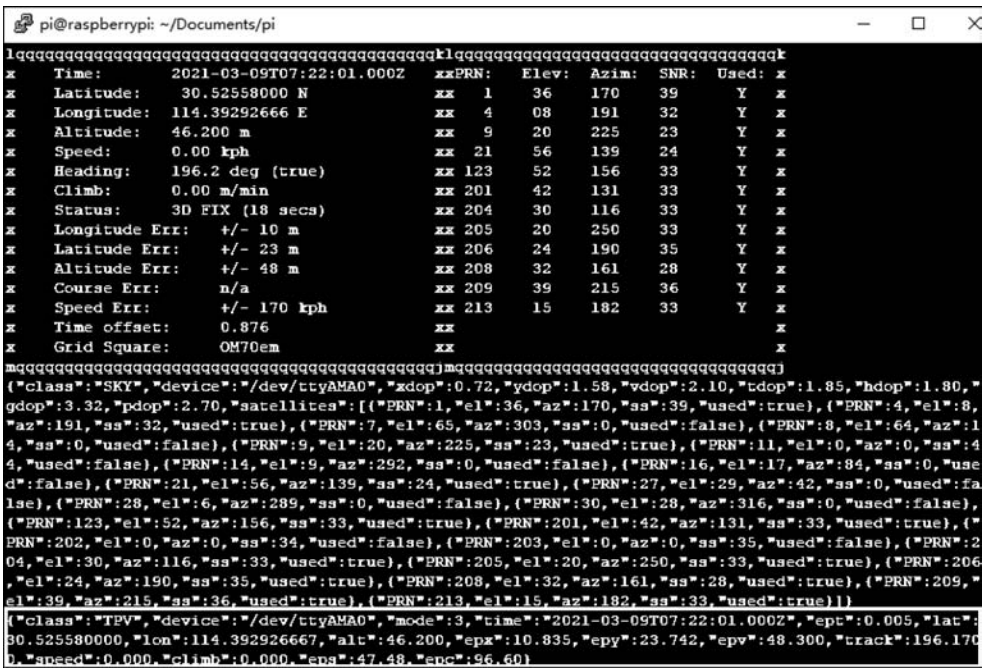


图 3-9　客户端窗口显示数据信息

图 3-8 中的标注框中包含了经纬度、海拔、航向等信息,借助 gps3 库可以解析并获取这些数据。运行 **sudo pip3 install gps3** 命令安装 gps3 库(负责提供 gpsd 接口,默认为 host='127.0.0.1',port=2947),新建 Python 脚本 gps_test2.py,输入以下代码:

```
from gps3 import gps3                                    #用来访问 gpsd
import time

def GPS3():
    gps_socket = gps3.GPSDSocket()                       #创建 gpsd 套接字连接并请求 gpsd 输出
    data_stream = gps3.DataStream()                      #将流式 gpsd 数据解压到字典中
    gps_socket.connect()                                 #建立连接
    gps_socket.watch()                                   #寻找新的 GPS 数据
    for new_data in gps_socket:
        if new_data:                                     #数据非空
            data_stream.unpack(new_data)                 #将字节流转换成数据
            if not (isinstance(data_stream.TPV['alt'],str)|
                    isinstance(data_stream.TPV['lat'],str)|
                    isinstance(data_stream.TPV['lon'],str)|
                    isinstance(data_stream.TPV['track'],str)):   #防止出现 n/a
                return [data_stream.TPV['alt'],data_stream.TPV['lat'],
                        data_stream.TPV['lon'],data_stream.TPV['track']]

if __name__ == "__main__":
    altitude,latitude,longitude,heading = GPS3()  #提取海拔、纬度、经度和航向
    print('海拔: ', altitude)                        #输出结果
    print('纬度: ', latitude)
    print('经度: ', longitude)
    print('航向: ', heading)
```

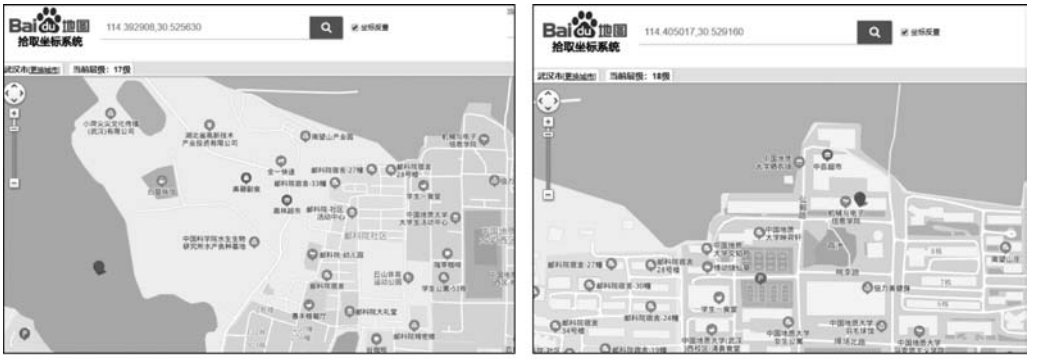
3.2.3　百度地图 GPS 定位

通过百度地图拾取坐标系统(http://api.map.baidu.com/lbsapi/getpoint/index.html)坐标反查发现,GPS 模块获取的数据在地图上显示的位置与实际位置有较大偏差,如图 3-10(a)所示。主要原因是坐标系之间不兼容,GPS 坐标遵循 WGS-84 标准,而百度对外接口的坐标系并不是 GPS 采集的经纬度。为了在百度地图上精准定位,需要对 GPS 坐标进行转换。具体过程是:先将 WGS-84 标准转换为 GCJ-02 标准(国内 Google、高德以及腾讯地图遵循该标准),再进行 BD-09 标准加密转换为百度地图坐标系。新建名为 gps_transform.py 的文件,输入如下代码:

```
import math

pi = 3.14159265358979324
a = 6378245.0                          #长半轴
ee = 0.00669342162296594323            #偏心率平方
```

```
x_pi = 3.14159265358979324 * 3000.0 / 180.0

def transformlat(lng, lat):
    ret = -100.0 + 2.0 * lng + 3.0 * lat + 0.2 * lat * lat + 0.1 * lng * lat +
          0.2 * math.sqrt(math.fabs(lng))
    ret += (20.0 * math.sin(6.0 * lng * pi) + 20.0 * math.sin(2.0 * lng * pi)) * 2.0 / 3.0
    ret += (20.0 * math.sin(lat * pi) + 40.0 * math.sin(lat / 3.0 * pi)) * 2.0 / 3.0
    ret += (160.0 * math.sin(lat / 12.0 * pi) + 320 * math.sin(lat * pi / 30.0)) * 2.0 / 3.0
    return ret

def transformlng(lng, lat):
    ret = 300.0 + lng + 2.0 * lat + 0.1 * lng * lng + 0.1 * lng * lat +
          0.1 * math.sqrt(math.fabs(lng))
    ret += (20.0 * math.sin(6.0 * lng * pi) + 20.0 * math.sin(2.0 * lng * pi)) * 2.0 / 3.0
    ret += (20.0 * math.sin(lng * pi) + 40.0 * math.sin(lng / 3.0 * pi)) * 2.0 / 3.0
    ret += (150.0 * math.sin(lng / 12.0 * pi) + 300.0 * math.sin(lng / 30.0 * pi)) * 2.0 / 3.0
    return ret

def wgs84_to_gcj02(lng, lat):
    dlat = transformlat(lng - 105.0, lat - 35.0)
    dlng = transformlng(lng - 105.0, lat - 35.0)
    radlat = lat / 180.0 * pi
    magic = math.sin(radlat)
    magic = 1 - ee * magic * magic
    sqrtmagic = math.sqrt(magic)
    dlat = (dlat * 180.0) / ((a * (1 - ee)) / (magic * sqrtmagic) * pi)
    dlng = (dlng * 180.0) / (a / sqrtmagic * math.cos(radlat) * pi)
    mglat = lat + dlat
    mglng = lng + dlng
    return [mglng, mglat]

def gcj02_to_bd09(lng, lat):
    z = math.sqrt(lng * lng + lat * lat) + 0.00002 * math.sin(lat * x_pi)
    theta = math.atan2(lat, lng) + 0.000003 * math.cos(lng * x_pi)
    bd_lng = z * math.cos(theta) + 0.0065
    bd_lat = z * math.sin(theta) + 0.006
    return [bd_lng, bd_lat]
```

修改 gps_test.py 代码，调用 wgs84_to_gcj02()和 gcj02_to_bd09()两个转换函数对读取到的经纬度数据进行处理，另存为 gps.py。运行该脚本，将得到的经纬度再次通过百度地图拾取坐标系统坐标反查，就能够准确定位到真实位置，结果如图 3-10(b)所示。



(a) 未进行坐标转换的结果　　(b) 坐标转换后的结果

图 3-10　百度地图 GPS 定位

3.3　烟雾/可燃气体检测

MQ-2 属于二氧化锡半导体气敏材料，适用于可燃性气体、酒精、烟雾等的探测。MQ-2 传感器模块如图 3-11 所示，4 个接口从上到下分别为 VCC、GND、DO 和 AO（具有 TTL 电平输出和模拟量输出），烟雾/可燃气体浓度越大，输出的模拟信号越大；输入电压为 5V，AO 输出 0.1～0.3V 相对无污染，最高浓度电压 4V 左右；使用前必须预热 20s 左右，使测量的数据稳定，使用中传感器发热属于正常现象。

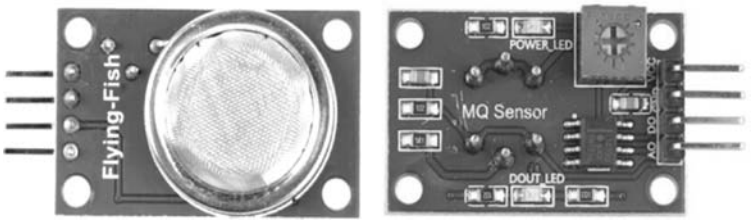


图 3-11　MQ-2 气敏传感器模块

为了获取浓度，树莓派需要外接模数转换器读取 MQ-2 模块 AO 引脚的输出值，这里选用模数转换芯片 MCP3002，如图 3-12 所示。该芯片是双通道 10 位 A/D 转换器，采用 2.7～5.5V 电源和参考电压输入，通过 SPI 串行总线与树莓派 GPIO 接口直接相连。具体的电路

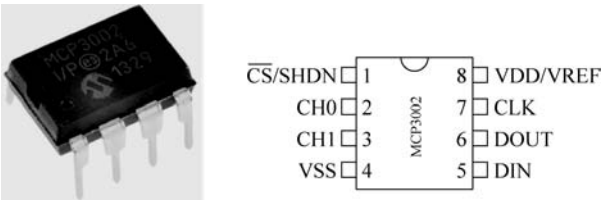


图 3-12　A/D 转换器 MCP3002

连接如下,MCP3002 的 VDD/VREF 和 VSS 分别连接树莓派的 17 脚(3.3V)和 9 脚(GND),DIN、DOUT、CLK 和 $\overline{CS}$/SHDN 引脚分别连接树莓派 GPIO 接口的 19 脚(MOSI)、21 脚(MISO)、23 脚(SCLK)和 24 脚(CE0);MQ-2 传感器的 VCC 连接树莓派 GPIO 接口的 2 脚(5V),GND 连接 MCP3002 的 VSS,A0 引脚通过 330Ω 和 470Ω 电阻串联分压后连到 MCP3002 的 CH0 引脚(保证输入的模拟电压不超过 3.3V)。

为了访问 MCP3002,先要启动树莓派的 SPI 硬件接口。操作过程如下:在终端输入 **sudo raspi-config** 进入配置界面,依次选择 Interfacing Options →SPI 选项,使能 SPI 接口。通过 **ls -l /dev** 命令可以看到两个 SPI 设备(spidev0.0 和 spidev0.1),GPIO 引脚 CE0 和 CE1 分别对应 spidev0.0 和 spidev0.1。MCP3002 使用 SPI 通信协议,可以使用 spidev 库来驱动 SPI 接口,简化程序设计。新建名为 mcp3002.py 的 Python 脚本,开启 SPI 总线设备并通过其获取 MQ-2 模块 AO 引脚输出的电压值,代码如下:

```
import spidev                                    #导入 spidev 库
import time

def read_Analog(channel):
    spi = spidev.SpiDev()                        #创建 SPI 总线设备对象
    spi.open(0, 0)                               #打开 SPI 总线设备,此处设备为/dev/spidev0.0
    spi.max_speed_hz = 15200                     #设置最大总线速度
    reply = spi.xfer2([(((6 + channel)<< 1 ) + 1 )<< 3, 0])
                                                 #向 spi 设备发送命令,见数据手册
    adc_out = ((reply[0]& 3) << 8) + reply[1]#读取 10 位转换数据
    value = adc_out * 3.3/1024                   #转化为电压值
    value = value/4.7 * (3.3 + 4.7)              #将串联分压折算为 MQ-2 的输出电压
    return value

if __name__ == "__main__":
    try:
        while True:
            value = read_Analog(0)               #读取 A/D 转换结果
            print("AO_voltage = %f" % value)
            time.sleep(5)
    except KeyboardInterrupt:
        spi.close()                              #中断退出关闭 SPI 设备
```

在终端输入 **python3 mcp3002.py** 运行程序,将点燃的蚊香靠近 MQ-2 传感器并不断吹气,测试结果如图 3-13 所示。

注意:spi.open(bus,device)用于开启 SPI 总线设备,bus 和 device 分别对应设备/dev/spidev0.0(或者 spidev0.1)后面的两个数字。此外,树莓派 SPI 接口默认的最大总线速度是 125.0MHz(spi.max_speed_hz =125000000),工作时应根据需要设置为合适的值,否则有可能会读不到正确的数据。

```
pi@raspberrypi: ~/Documents/pi
pi@raspberrypi:~/Documents/pi $ python3 mcp3002.py
AO_voltage=0.071310
AO_voltage=0.142620
AO_voltage=0.208444
AO_voltage=0.191988
AO_voltage=0.252327
AO_voltage=0.318152
AO_voltage=0.323637
```

图 3-13　气敏传感器模块测试结果

3.4　温湿度检测

DHT11 是一款含有已校准数字信号输出的温湿度传感器。如图 3-14 所示，该传感器模块体积小、功耗低，采用单线制串行接口，3 个接口分别为 VCC、OUT 和 GND。其主要参数特性包括：供电电压为 3.3～5.5V，相对湿度测量范围为 20%～95%（测量误差±5%），温度测量范围为 0～50℃（测量误差±2℃）。DHT11 与树莓派接口简单，VCC 和 GND 分别连接树莓派 GPIO 的 17 脚（3.3V）、20 脚（GND），OUT 连接 GPIO 的 18 引脚。

DHT11 遵循单总线通信协议，对时序有较为严格的要求。一次完整的工作流程如下：首先树莓派发送开始信号，将总线由高电平拉低，时长至少需要 18ms，以保证被 DHT11 检测到。待开始信号结束后，树莓派释放总线，总线从输出模式变为输入模式，保持高电平延时等待 20～40μs 后，DHT11 发送 80μs 低电平响应信号，紧接着输出 80μs 的高电平通知树莓派准备接收数据。DHT11 一次传送 40b 的数据（高位先出），1b 数据都以 50μs 低电平时隙开始，电平的长短决定了数据位是 0 还是 1。具体来说，数据位 0 的格式是 50μs 的低电平和 26～28μs 的高电平；数据位 1 的格式是 50μs 的低电平和 70μs 的高电平。

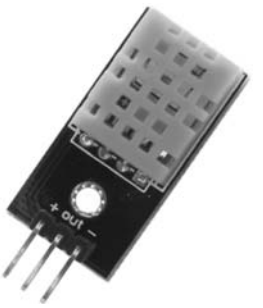


图 3-14　数字温湿度传感器 DHT11

40b 数据格式为：8b 湿度整数数据＋8b 湿度小数数据＋8b 温度整数数据＋8b 温度小数数据＋8b 校验和。数据传送正确时校验和等于前面 4 个 8b 数据之和。如果数据接收不正确，则放弃本次数据，重新接收。

编写树莓派读取 DHT11 温湿度数据的代码时，有两点需要说明：一是传感器上电后，要等待 1s 以越过不稳定状态；二是树莓派的实时性较弱，不像单片机那样严格可控，在读取 DHT11 的数据脉冲时要注意控制时序，否则可能无法正确读取到数据。新建 dht11.py 脚本，输入以下代码：

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

```
def init():
    GPIO.setmode(GPIO.BOARD)
    time.sleep(1)                                    #时延 1s,越过不稳定状态

def get_readings(ch):                                #ch:DHT11 数据引脚
    data = []                                        #存储温湿度值
    j = 0                                            #数据位计数器
    OUT = ch
    GPIO.setup(OUT , GPIO.OUT)                       #设置引脚为输出
    GPIO.output(OUT, GPIO.LOW)                       #发送开始信号,将总线由高拉低
    time.sleep(0.02)                                 #时长需要超过 18ms
    GPIO.output(OUT, GPIO.HIGH)                      #释放总线,变为高电平
    GPIO.setup(OUT, GPIO.IN,pull_up_down = GPIO.PUD_UP) #设置引脚为输入,上拉
    while GPIO.input(OUT) == GPIO.LOW:               #等待 DHT11 发送的低电平响应信号
        continue
    while GPIO.input(OUT) == GPIO.HIGH:              #等待 DHT11 拉高总线结束
        continue
    while j < 40:                                    #开始接收 40bit 数据
        k = 0                                        #通过计数的方式判断数据位高电平的时间
        while GPIO.input(OUT) == GPIO.LOW:
            continue
        while GPIO.input(OUT) == GPIO.HIGH:
            k += 1
        if k > 100:                                  #数据线为高时间过长,放弃本次数据
            break
        if k < 8:                                    #数据位为 0
            data.append(0)
        else:                                        #数据位为 1
            data.append(1)
        j += 1
    return data

def data_check(data):
    humidity_bit = data[0:8]                         #湿度整数
    humidity_point_bit = data[8:16]                  #湿度小数
    temperature_bit = data[16:24]                    #温度整数
    temperature_point_bit = data[24:32]              #温度小数
    check_bit = data[32:40]                          #检验位
    humidity = 0
    humidity_point = 0
    temperature = 0
    temperature_point = 0
    check = 0

    for i in range(8):
        humidity += humidity_bit[i] * 2 ** (7 - i)       #转换成十进制数据
```

```
            humidity_point += humidity_point_bit[i] * 2 ** (7 - i)
            temperature += temperature_bit[i] * 2 ** (7 - i)
            temperature_point += temperature_point_bit[i] * 2 ** (7 - i)
            check += check_bit[i] * 2 ** (7 - i)

        return [humidity,humidity_point,temperature,temperature_point,check]

    if __name__ == "__main__":
        init()
        while True:
            dat = get_readings(18)
            humidity,humidity_point,temperature,temperature_point,check = data_check(dat)
            tmp = humidity + humidity_point + temperature + temperature_point
            if check == tmp:                    #数据校验
                T_value = str(temperature) + "." + str(temperature_point) #温度的整数与小数结合
                H_value = str(humidity) + "." + str(humidity_point)    #湿度的整数与小数结合
                print ("temperature :", T_value, "*C, humidity :", H_value, "%")
                time.sleep(5)
```

在树莓派终端输入 **python3 dht11.py** 运行程序，可以得到从 DHT11 读取的温湿度数据，结果如图 3-15 所示。

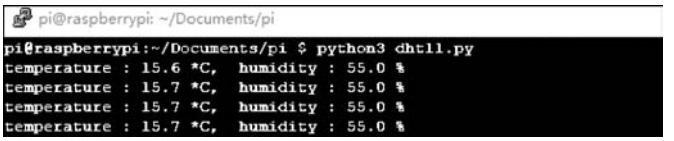


图 3-15　温湿度传感器测试结果

3.5　大气压检测

BMP180 是一款性能优越的数字气压传感器，具有高精度、体积小和超低能耗的特点。该模块采用 I^2C 接口，如图 3-16 所示，4 个引脚分别是 VIN、GND、SCL 和 SDA。其主要特点如下：与 BMP085 兼容，电源电压 1.8～3.6V（VIN 需 5V 供电，该模块上带有电源转换芯片，可将 5V 转化为 3.3V），低功耗（标准模式下电流仅为 5μA），压力范围为 300～1100hPa（海拔 9000m～－500m），低功耗模式下分辨率为 0.06hPa（0.5m）。

除了测量大气压力，BMP180 还能测量温度，同时还可以根据式（3-1）推测出当前海拔高度。

$$\text{altitude} = 44330 \times \left(1 - \left(1 - \frac{p}{p_0}\right)^{1/5.255}\right) \tag{3-1}$$

式中，p 为测得的大气压值，p_0 是海平面大气压力，默认值取 1013.25hPa。

为了通过 GPIO 接口访问外接 I^2C 设备，要先启动树莓派的 I^2C 硬件接口。输入 **sudo**

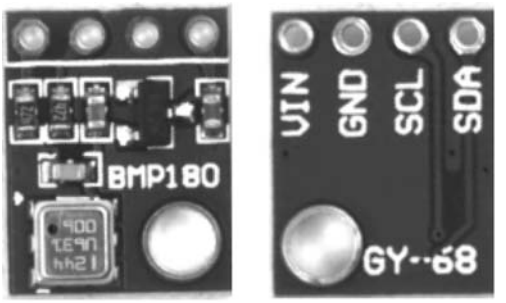


图 3-16　数字气压传感器 BMP180

raspi-config 打开配置界面，依次选择 Interfacing Options →I2C 选项，使能 I²C 接口。通过 **ls -l /dev** 命令可以查看到 I²C 设备 i2c-1。使用 smbus 库对 I²C 设备进行读写操作，可以避免编写烦琐的 I²C 时序，简化程序设计。另外，使用 i2cdetect 工具可以查看连接到树莓派上的 I²C 设备，这二者的安装命令分别为 **sudo apt-get install python-smbus** 和 **sudo apt-get install i2c-tools**。

树莓派 3 脚和 5 脚分别预设为 I²C 总线的数据信号 SDA 和时钟信号 SCL，能够与外接 I²C 设备进行通信。I²C 总线可以同时连接多个 I²C 设备，每个设备都有一个唯一的 7 位地址。树莓派与某个特定 I²C 设备通信时是通过地址进行区分的，只有被呼叫的设备会做出响应。将 BMP180 的 VIN、GND、SDA、SCL 引脚分别与树莓派 GPIO 的 4 脚(5V)、14 脚(GND)、3 脚(SDA)和 5 脚(SCL)连接。当 BMP180 连接好后，输入命令 **i2cdetect -y 1**，可以看到 BMP180 的设备地址为 0x77，如图 3-17 所示。

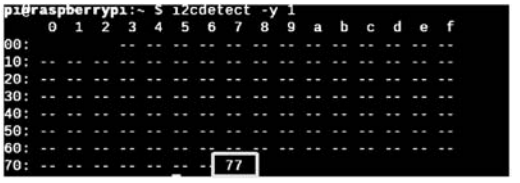


图 3-17　i2cdetect 查看已连接的 I²C 设备

BMP180 的工作流程大致如下：首先读取相关寄存器获得校准数据，然后分别对相应寄存器进行读写，获取原始温度和气压数据；接下来通过前面得到的校准数据和原始数据计算出实际的温度和气压值；最后再根据气压值推测出海拔高度。读者可自行查看数据手册了解 BMP180 的工作模式、寄存器设置以及具体计算公式等内容。下面编写测试程序，创建名为 bmp180.py 的脚本，输入以下内容：

```
import time
import smbus                                    #导入 smbus 库实现树莓派和 BMP180 的 I2C 通信

class BMP180():                                 #定义 BMP180 类
```

```
def __init__(self, address = 0x77, mode = 1):    #默认 OSS = 1(标准模式),设备地址 0x77
    self._mode = mode                             #单下画线开头的表示伪私有变量
    self._address = address
    self._bus = smbus.SMBus(1)                    #创建 smbus 实例,1 代表/dev/i2c - 1

def read_u16(self,cmd):                           #读 16 位无符号数据
    MSB = self._bus.read_byte_data(self._address,cmd)
    LSB = self._bus.read_byte_data(self._address,cmd + 1)
    return (MSB << 8) + LSB

def read_s16(self,cmd):                           #读 16 位有符号数据
    result = self.read_u16(cmd)
    if result > 32767:
        result -= 65536
    return result

def write_byte(self,cmd,val):
    self._bus.write_byte_data(self._address, cmd, val)      #I2C 总线写字节操作

def read_byte(self,cmd):
    return self._bus.read_byte_data(self._address,cmd)      #I2C 总线读字节操作

def read_Calibration(self):                       #从 22 个寄存器读取校准数据
    caldata = []
    caldata.append(self.read_s16(0xAA))
    caldata.append(self.read_s16(0xAC))
    caldata.append(self.read_s16(0xAE))
    caldata.append(self.read_u16(0xB0))
    caldata.append(self.read_u16(0xB2))
    caldata.append(self.read_u16(0xB4))
    caldata.append(self.read_s16(0xB6))
    caldata.append(self.read_s16(0xB8))
    caldata.append(self.read_s16(0xBA))
    caldata.append(self.read_s16(0xBC))
    caldata.append(self.read_s16(0xBE))
    return caldata

def read_rawTemperature(self):                    #读取原始温度数据
    self.write_byte(0xF4, 0x2E)                   #向控制寄存器 0xF4 发送读取温度命令 0x2E
    time.sleep(0.005)                             #等待测量完毕,延时至少 4.5ms
    rawTemp = self.read_u16(0xF6)
    return rawTemp

def read_rawPressure(self):                       #读取原始气压数据
    self.write_byte(0xF4, 0x34 + (self._mode << 6))
    time.sleep(0.008)                             #OSS = 1,延时不少于 7.5ms
```

```
        MSB = self.read_byte(0xF6)
        LSB = self.read_byte(0xF7)
        XLSB = self.read_byte(0xF8)
        rawPressure = ((MSB << 16) + (LSB << 8) + XLSB) >> (8 - self._mode)
        return rawPressure

    def read_Temperature(self,caldata):        #计算实际温度值,公式参见数据手册
        UT = self.read_rawTemperature()
        X1 = ((UT - caldata[5]) * caldata[4]) >> 15
        X2 = (caldata[9] << 11) / (X1 + caldata[10])
        B5 = X1 + X2
        temp = (B5 + 8) / 16
        return temp / 10.0

    def read_Pressure(self,caldata):           #计算实际气压值,公式参见数据手册
        UT = self.read_rawTemperature()
        UP = self.read_rawPressure()
        X1 = ((UT - caldata[5]) * caldata[4]) >> 15
        X2 = (caldata[9] << 11) / (X1 + caldata[10])
        B5 = X1 + X2
        B6 = B5 - 4000
        X1 = (caldata[7] * (B6 * B6) / 2 ** 12) / 2 ** 11
        X2 = (caldata[1] * B6) / 2 ** 11
        X3 = X1 + X2
        B3 = (((int(caldata[0] * 4 + X3)) << self._mode) + 2) / 4
        X1 = (caldata[2] * B6) / 2 ** 13
        X2 = (caldata[6] * (B6 * B6) / 2 ** 12) / 2 ** 16
        X3 = ((X1 + X2) + 2) / 2 ** 2
        B4 = (caldata[3] * (X3 + 32768)) / 2 ** 15
        B7 = (UP - B3) * (50000 >> self._mode)
        if B7 < 0x80000000:
            p = (B7 * 2) / B4
        else:
            p = (B7 / B4) * 2
        X1 = (p / 2 ** 8) * (p / 2 ** 8)
        X1 = (X1 * 3038) / 2 ** 16
        X2 = ( - 7357 * p) / 2 ** 16
        p = p + ((X1 + X2 + 3791) / 2 ** 4)
        return p / 100.0

    def read_Altitude(self,sealevel_hpa,pressure): #sealevel_hpa 是海平面大气压
        altitude = 44330 * (1.0 - pow(pressure / sealevel_hpa, (1.0/5.255)))
        return altitude
```

```
def read_BMP180_data():
    bmp = BMP180()                                          # 创建 BMP180 实例
    while True:
        caldata = bmp.read_Calibration()
        temp = bmp.read_Temperature(caldata)
        pressure = bmp.read_Pressure(caldata)
          altitude = bmp.read_Altitude(1013.25,pressure)          # sealevel_hpa =
1013.25hPa
        print("Altitude : %.2f m" % altitude)
        print("Pressure : %.2f hPa" % pressure)
        print("Temperature : %.2f C\n" % temp)
        time.sleep(5)

if __name__ == '__main__':
    read_BMP180_data()
```

运行上面的程序可以发现，计算出来的海拔高度为负数，这与所在地区的真实海拔明显不一致。其原因是式(3-1)中 p_0 的默认值是 0℃时的海平面大气压值，需要根据实际情况进行校正。这里的做法是，在不同楼层通过树莓派读取 BMP180 输出的大气压值 p，同时用智能手机内置的指南针工具获取对应的海拔高度，按式(3-2)反向计算当前的 p_0，然后再将 p_0 代入式(3-1)即可计算出比较精准的海拔高度。

$$p_0 = p \Big/ \left(1 - \frac{\text{altitude}}{44330}\right)^{5.255} \tag{3-2}$$

如表 3-1 所示，根据当前环境下 8 个楼层测得的大气压值和对应的海拔高度，计算出的 p_0 值比较稳定(平均值 1023.66hPa)。由于 p_0 的值会随着温度改变发生变化，所以实际应用时需要进行修正。将 bmp180.py 脚本中 sealevel_hpa 的值替换为 1023.66，再次运行程序，可得到准确的海拔高度，结果如图 3-18 所示。

表 3-1　不同楼层大气压与海拔测量值的对应关系

楼层	大气压测量值/hPa	海拔高度/m	p_0 计算值/hPa
1	1021.96	15	1023.78
2	1021.26	20	1023.68
3	1020.73	24	1023.64
4	1020.24	28	1023.63
5	1019.87	32	1023.75
6	1019.35	36	1023.71
7	1018.85	39	1023.57
8	1018.16	44	1023.49

```
pi@raspberrypi: ~/Documents/pi
pi@raspberrypi:~/Documents/pi $ python3 bmp180.py
Altitude : 44.49 m
Pressure : 1018.27 hPa
Temperature : 18.41 C

Altitude : 44.74 m
Pressure : 1018.24 hPa
Temperature : 18.25 C
```

图 3-18　BMP180 测试结果

3.6　空气质量检测

空气质量检测采用集 CO_2、PM2.5、PM10、温湿度、总挥发性有机物(TVOC)及甲醛(CH_2O)于一体的综合型传感器模块,如图 3-19 所示。该模块供电电压 5V,工作温度为 0～50℃,采用串口通信协议,波特率默认为 9600bps,数据传输周期默认为 1s(可通过指令修改)。每次传输的数据共 19 字节,格式为:报文头(0x01)＋功能码(0x03)＋数据长度(0x0E)＋7 个双字节数据(CO_2、TVOC、CH2O、PM2.5、湿度、温度、PM10)＋2 字节的 CRC16 校验。

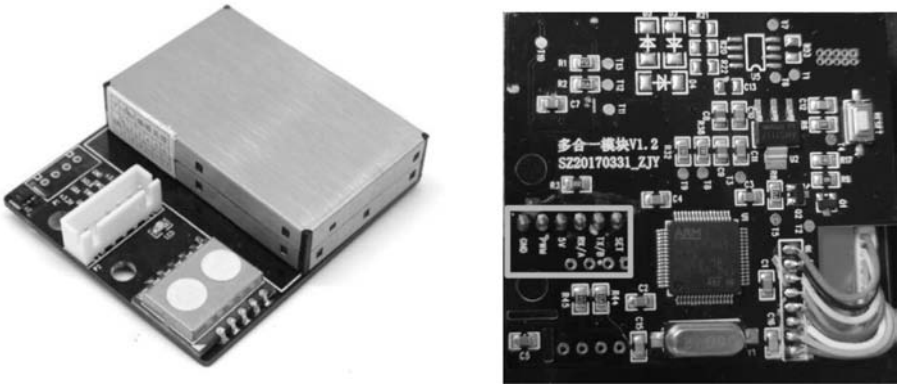


图 3-19　综合型空气质量传感器

由于树莓派的 UART 接口已分配给 GPS 模块使用,所以这里通过 USB 外接串口模块(CH340E)实现树莓派与空气质量传感器模块的连接,如图 3-20 所示。传感器模块和 CH340E 的具体接口如下:5V 和 GND 引脚分别直接相连,TXD 和 RXD 引脚交叉相连,此外,传感器模块的 SET 引脚连接树莓派的 2 脚(5V)。

树莓派系统集成了 USB 转串口驱动,将 CH340E 插入树莓派 USB 接口就可以使用。在命令行输入 **lsusb** 查看连接的 USB 设备,输入 **ls -l /dev/tty*** 查看设备的串口号,结果分别如图 3-21(a)和图 3-21(b)所示。在树莓派系统中,USB 串口设备一般是根据设备插入顺序进行命名,依次是/dev/ttyUSB0、/dev/ttyUSB1 等。

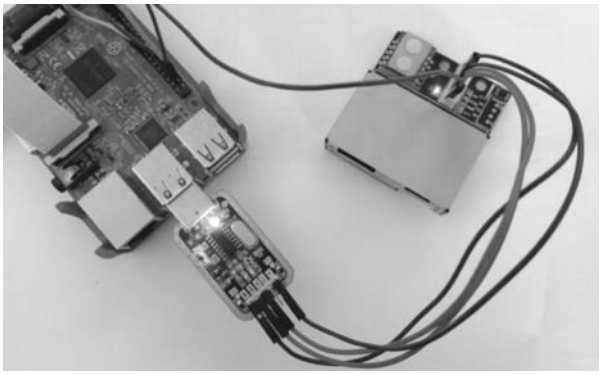


图 3-20　树莓派连接空气质量传感器

```
pi@raspberrypi:~/Documents/pi $ lsusb
Bus 001 Device 004: ID 1a86:7523 QinHeng Electronics HL-340 USB-Serial adapter
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast
Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp. SMC9514 Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

(a) 连接的 USB 设备

```
crw--w---- 1 root tty       4,  8 3月  11 20:14 /dev/tty8
crw--w---- 1 root tty       4,  9 3月  11 20:14 /dev/tty9
crw-rw---- 1 root dialout 204, 64 3月  11 20:14 /dev/ttyAMA0
crw------- 1 root root      5,  3 3月  11 20:14 /dev/ttyprintk
crw-rw---- 1 root dialout   4, 64 3月  11 20:14 /dev/ttyS0
crw-rw---- 1 root dialout 188,  0 3月  11 20:14 /dev/ttyUSB0
```

(b) 设备串口号

图 3-21　查看 USB 串口设备

创建脚本文件 air_quality_senor.py，输入如下代码：

```
import serial
import time
import binascii                                    #用于二进制(byte类型数据)和 ASCII 的转换

class Multisensor():                               #定义 Multisensor 类
    def __init__(self):
        self.ser = serial.Serial("/dev/ttyUSB0", 9600)  #打开 USB 串口,波特率设为 9600bps
        self.time_sent = bytes.fromhex('42 78 01 00 00 00 00 FF')   #设置数据传输周期 1s
        self.ser.write(self.time_sent)                  #通过串口向传感器模块写入指令
        self.ser.flushInput()                           #清空串口接收缓存中的数据

    def serial_rec(self):
        count = self.ser.inWaiting()                    #返回串口接收缓存中的字节数
        while count != 0:
            recv = self.ser.read(count)                 #从串口读入指定的字节数
```

```
            '''下条语句返回二进制数据的十六进制表示形式,将串口接收的 19 字节转
            换成 38 位十六进制字符串,其中[2:-1]表示截取该行从第三位到最后一个字符
            (换行符)之间的部分,该部分对应真正的有效数据'''
            recv = str(binascii.b2a_hex(recv))[2:-1]
            self.ser.flushInput()        #清空接收缓存区
            return recv

    #以下各项空气指标的具体计算公式参见传感器模块文档
    def co2_count(self,recv):
        recv_co2 = recv[6:10]                    #从 38 位十六进制字符串中截取 CO2 数据
        recv_co2_h = int(recv_co2[0:2],16)       #高字节十六进制转换成十进制
        recv_co2_l = int(recv_co2[2:4],16)       #低字节十六进制转换成十进制
        co2 = recv_co2_h*256 + recv_co2_l
        print('(1).CO2 : %d ppm' %co2)
        return co2

    def tvoc_count(self,recv):
        recv_tvoc = recv[10:14]                  #从 38 位十六进制字符串中截取 TVOC 数据
        recv_tvoc_h = int(recv_tvoc[0:2],16)
        recv_tvoc_l = int(recv_tvoc[2:4],16)
        tvoc = float(recv_tvoc_h*256 + recv_tvoc_l)/10.0
        print('(2).TVOC : %f ug/m3' %tvoc)
        return tvoc

    def ch20_count(self,recv):
        recv_ch20 = recv[14:18]                  #从 38 位十六进制字符串中截取 CH2O 数据
        recv_ch20_h = int(recv_ch20[0:2],16)
        recv_ch20_l = int(recv_ch20[2:4],16)
        ch20 = float(recv_ch20_h*256 + recv_ch20_l)/10.0
        print('(3).CH20 : %f ug/m3' %ch20)
        return ch20

    def pm25_count(self,recv):
        recv_pm25 = recv[18:22]                 #从 38 位十六进制字符串中截取 PM2.5 数据
        recv_pm25_h = int(recv_pm25[0:2],16)
        recv_pm25_l = int(recv_pm25[2:4],16)
        pm25 = recv_pm25_h*256 + recv_pm25_l
        print('(4).PM2.5 : %d ug/m3' %pm25)
        return pm25

    def humidity_count(self,recv):
        recv_humidity = recv[22:26]             #从 38 位十六进制字符串中截取湿度数据
        recv_humidity_h = int(recv_humidity[0:2],16)
        recv_humidity_l = int(recv_humidity[2:4],16)
        srh = recv_humidity_h*256 + recv_humidity_l
        humidity = -6 + 125*float(srh)/2**16
```

```
            print('(6).Humidity : %f %%RH' %humidity)
            return humidity

        def temp_count(self,recv):
            recv_temp = recv[26:30]              #从38位十六进制字符串中截取温度数据
            recv_temp_h = int(recv_temp[0:2],16)
            recv_temp_l = int(recv_temp[2:4],16)
            stem = recv_temp_h*256 + recv_temp_l
            temp = -46.85 + 175.72*float(stem)/2**16
            print('(7).Temperature : %f ℃' %temp)
            return temp

        def pm10_count(self,recv):
            recv_pm10 = recv[30:34]              #从38位十六进制字符串中截取PM10数据
            recv_pm10_h = int(recv_pm10[0:2],16)
            recv_pm10_l = int(recv_pm10[2:4],16)
            pm10 = recv_pm10_h*256 + recv_pm10_l
            print('(5).PM10 : %d ug/m3' %pm10)
            return pm10

        def read_sensor_data(self):              #获取空气指标参数
            while True:
                recv = self.serial_rec()
                #判断接收数据的格式是否正确
                if recv != None and len(recv) == 38 and recv[0:6] == '01030e':
                    sto_co2 = self.co2_count(recv)
                    sto_tvoc = self.tvoc_count(recv)
                    sto_ch20 = self.ch20_count(recv)
                    sto_pm25 = self.pm25_count(recv)
                    sto_pm10 = self.pm10_count(recv)
                    sto_humidity = self.humidity_count(recv)
                    sto_temp = self.temp_count(recv)
                    break                        #直至接收到一次完整数据后退出本次循环

            return sto_co2,sto_tvoc,sto_ch20,sto_pm25,sto_pm10,sto_humidity,sto_temp

    if __name__ == '__main__':
        try:
            multisensor = Multisensor()          #创建实例
            while True:
                multisensor.read_sensor_data()   #调用Multisensor类的方法
                print('------------------------')
        except KeyboardInterrupt:
            if multisensor.ser != None:
                multisensor.ser.close()          #关闭USB串口
```

运行程序,可以同时监测 7 种空气指标参数,结果如图 3-22 所示。

```
pi@raspberrypi: ~/Documents/pi
pi@raspberrypi:~/Documents/pi $ python3 air_quality_senor.py
(1).CO2 :  443 ppm
(2).TVOC :  7.300000 ug/m3
(3).CH20 :  2.600000 ug/m3
(4).PM2.5 :  58 ug/m3
(5).PM10 :  72 ug/m3
(6).Humidity :  48.485321 %RH
(7).Temperature :  19.498135 °C
-----------------------
(1).CO2 :  439 ppm
(2).TVOC :  6.100000 ug/m3
(3).CH20 :  2.600000 ug/m3
(4).PM2.5 :  58 ug/m3
(5).PM10 :  72 ug/m3
(6).Humidity :  48.485321 %RH
(7).Temperature :  19.498135 °C
-----------------------
```

图 3-22　空气质量传感器测试结果

3.7　数字指南针

数字指南针也称作电子罗盘或磁力计,用于测量地球磁场的方向和大小。HMC5883L 是一种带有数字接口的弱磁传感器芯片,采用各向异性磁阻(AMR)技术,灵敏度高、可靠性好,内置 12 位模数转换器,可以测量沿 X、Y 和 Z 轴 3 个方向上的地球磁场值,测量范围从毫高斯到 8 高斯。HMC5883L 模块及其引脚如图 3-23 所示,该模块工作电压为 2.16～3.6V,工作电流 100μA,罗盘航向精度 1°～2°。

图 3-23　磁场传感器模块 HMC5883L

和前面介绍的 BMP180 一样,HC5883L 也遵循 I^2C 协议。它和树莓派相连只需要 4 根线,即 VCC、GND、SCL 和 SDA,具体连接如下:将 SDA 和 SCL 引脚分别连接至树莓派 GPIO 接口的 3 脚和 5 脚,GND 和 VCC 分别连接 GPIO 的 20 脚(GND)和 17 脚(3.3V)。连线接好后,在终端输入命令 **sudo i2cdetect -y 1**,如图 3-24 所示,可以看到在地址 0x1e 处检测到了一个设备,这就是外接的 HMC5883L 传感器。

同样,树莓派使用 smbus 库对 HC5883L 模块进行读写操作。下面编写程序,通过树莓派读取 HMC5883L 模块沿 X、Y 和 Z 轴的磁场强度并计算其航向角。新建脚本文件

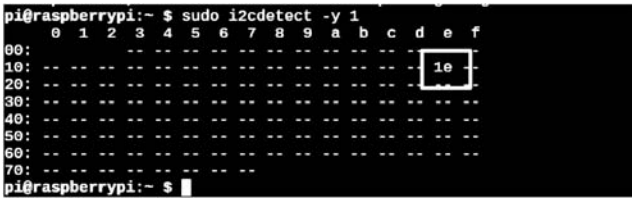


图 3-24　查看 HC5883L 的地址

hmc5883l.py，输入以下代码：

```
import smbus
import time
import math

class HMC5883():                                    #定义 HMC5883 类
    def __init__(self, address = 0x1e,x_offset = 0.041304,y_offset = -0.132608):
    '''HMC5883L 设备地址 0x1e, x_offset 和 y_offset 分别为 x、y 方向校准量'''
        self._address = address
        self._bus = smbus.SMBus(1)                  #创建 smbus 实例,1 代表/dev/i2c-1
        self.Magnetometer_config()                  #设置寄存器
        self.x_offset = x_offset
        self.y_offset = y_offset

    def read_raw_data(self,addr):                   #addr 为数据输出寄存器的高字节地址
        high = self._bus.read_byte_data(address, addr)    #读取高字节数据
        low = self._bus.read_byte_data(address, addr+1) #读取低字节数据
        value = (high << 8) + low
        if (value >= 0x8000):                               #两个字节以补码的形式存储
            return -((65535 - value) + 1)
        else:
            return value

    def Magnetometer_config(self):      #设置配置寄存器 A、B 和模式寄存器,参看数据手册
        self._bus.write_byte_data(self._address, 0, 0x74)    #配置寄存器 A 地址 0x00
        self._bus.write_byte_data(self._address, 1, 0xe0)    #配置寄存器 B 地址 0x01
        self._bus.write_byte_data(self._address, 2, 0)       #模式寄存器地址 0x02

'''XYZ 轴数据输出寄存器高字节地址分别为 0x03,0x07 和 0x05 读取的原始数据除以增益'''
    def get_magnetic_xyz(self):
        x_data = self.read_raw_data(3)/230.0                #X 轴输出数据
        y_data = self.read_raw_data(7)/230.0                #Y 轴输出数据
        z_data = self.read_raw_data(5)/230.0                #Z 轴输出数据
        return [x_data,y_data,z_data]
```

```
    def read_HMC5883_data(self):
        x_data,y_data,z_data = self.get_magnetic_xyz()
        x_data = x_data - self.x_offset                    #校正偏差
        y_data = y_data - self.y_offset
        print('x轴磁场强度: ', x_data, 'Gs')
        print('y轴磁场强度: ', y_data, 'Gs')
        print('z轴磁场强度: ', z_data, 'Gs')
        #计算航向角
        bearing = math.atan2(y_data, x_data)
        if (bearing < 0):
            bearing += 2 * math.pi
        print("航向角: ", math.degrees(bearing),"\n")      #将弧度转换为角度
        return math.degrees(bearing)
        #return round(math.degrees(bearing),2)             #保留小数点后2位

if __name__ == '__main__':
    hmc = HMC5883()
    while True:
        hmc.read_HMC5883_data()
        time.sleep(5)
```

上例中,设置配置寄存器 A 的值为 0x74,其功能是数据输出频率为 30Hz,每次测量采样 8 个样本并将其平均值作为输出;配置寄存器 B 的值为 0xe0,其功能是将增益设置为 230,输出数据的范围为 0xF800～0x07FF;模式寄存器的值为 0x00,表示选择连续测量操作模式。读取 X 轴和 Y 轴数据寄存器的原始值,除以增益后得到各方向上的磁场强度,最后再计算出航向角。运行程序,以正北方为初始方向顺时针旋转传感器,航向角不断增大,结果如图 3-25 所示。

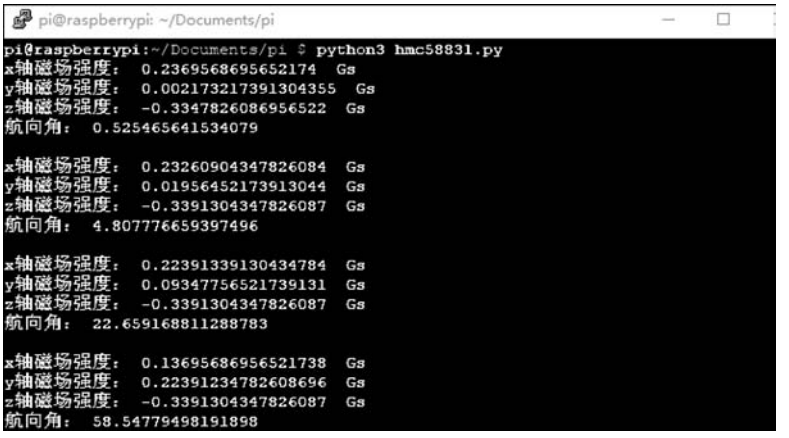


图 3-25　磁场强度与航向角测试结果

如果根据 HMC5883L 读取值计算出来的角度和指南针的角度有偏差，需要按如下步骤进行校正。首先将传感器模块水平放置，匀速旋转找出 X 轴和 Y 轴方向上磁场强度的最大值与最小值，即 x_max、x_min、y_max、y_min，然后计算得到两个方向上的偏移量 x_offset＝(x_max＋x_min)/2 和 y_offset＝(y_max＋y_min)/2。新建 hmc5883l_calibration.py 输入以下代码：

```
import time
from hmc5883l import HMC5883

def calibrateMag():                          #进行 X 轴和 Y 轴方向的校准,绕 Z 轴慢速转动
    minx = 0
    maxx = 0
    miny = 0
    maxy = 0

    hmc = HMC5883()                          #创建实例
    hmc.Magnetometer_config()

    for i in range(0,200):                   #旋转过程中读取 200 个数据
        x_out,y_out,z_out = hmc.get_magnetic_xyz()

        if x_out < minx:
            minx = x_out
        if y_out < miny:
            miny = y_out
        if x_out > maxx:
            maxx = x_out
        if y_out > maxy:
            maxy = y_out
        time.sleep(0.1)
    print("minx: ", minx)
    print("miny: ", miny)
    print("maxx: ", maxx)
    print("maxy: ", maxy)
    x_offset = (maxx + minx) / 2
    y_offset = (maxy + miny) / 2
    print("x_offset: ", x_offset)
    print("y_offset: ", y_offset)

if __name__ == '__main__':
    calibrateMag()                           #测试中,X 箭头初始朝向北方,匀速旋转
```

校正测试结果如图 3-26 所示，其中的 x_offset、y_offset 即为 hmc5883l.py 中 X 轴和 Y 轴磁场强度的校准量。实际应用时，读者需要根据当前位置进行校正并重新设定。

```
pi@raspberrypi:~/Documents/pi $ python3 hmc5883l_calibration.py
minx:  -0.20869565217391303
miny:  -0.4217391304347826
maxx:  0.29130434782608694
maxy:  0.1565217391304348
x_offset:  0.041304347826086954
y_offset:  -0.13260869565217392
```

图 3-26　HMC5883l 校正结果

3.8　超声波测距

利用 HC-SR04 超声波传感器可以检测前方的障碍物，实现超声波测距与避障功能。HC-SR04 模块如图 3-27 所示，包括超声波发射器、接收器与控制电路，4 个接口从左到右分别为 VCC、Trig(触发控制信号输入端)、Echo(回响信号输出端)和 GND。该模块采用 5V 电压供电，工作电流 15mA，工作频率 40kHz，测量角度不大于 15°，探测距离 2～400cm，精度为 0.3cm。

图 3-27　HC-SR04 超声波传感器

HC-SR04 模块使用简单，只需给 Trig 引脚至少 10μs 的高电平信号即可触发测距。超声波发射器会对外连续发送 8 个 40kHz 的脉冲。如果接收器检测到返回信号，则 Echo 引脚输出一个高电平，且该高电平的持续时间是超声波从发射到返回的时间。由此可以计算出前方障碍物的距离，即距离等于高电平持续时间乘以声速的积的一半。

将 HC-SR04 的 VCC 和 GND 分别连接树莓派 GPIO 的 4 脚(5V)和 30 脚(GND)，Trig 端接树莓派 GPIO 的 29 脚，由于 Echo 端输出电压为 5V，需要经过 330Ω 和 470Ω 电阻串联分压后连接到树莓派 GPIO 的 31 脚。新建脚本 hcsr04.py，输入超声波测距程序，代码如下：

```
import RPi.GPIO as GPIO
import time

class HCSR04():                                    #定义 HCSR04 类
```

```
    def __init__(self,trigger = 29,echo = 31):      #默认定义 TRIG 为 29 脚,ECHO 为 31 脚
        self.TRIG = trigger
        self.ECHO = echo
        GPIO.setwarnings(False)                     #禁用引脚设置的警告信息
        GPIO.setmode(GPIO.BOARD)
        GPIO.setup(self.TRIG, GPIO.OUT, initial = False)
        GPIO.setup(self.ECHO, GPIO.IN)

    def readDistanceCm(self):
        GPIO.output(self.TRIG, True)                #设置 TRIG 为高电平
        time.sleep(0.00001)                         #等待 10μs
        GPIO.output(self.TRIG, False)
        while GPIO.input(self.ECHO) == 0:
            pass
        start_time = time.time()                    #获取 ECHO 为高的起始时间
        while GPIO.input(self.ECHO) == 1:
            pass
        stop_time = time.time()                     #获取 ECHO 为高的终止时间
        time_elapsed = stop_time - start_time       #计算超声波发射到返回的时间
        distance = (time_elapsed * 34000) / 2       #计算距离,单位为 cm
        return distance

if __name__ == "__main__":
    try:
        hcsr04 = HCSR04(29,31)                      #创建实例,可以根据需要替换为其他引脚
        while True:
            d = hcsr04.readDistanceCm()
            print("Distance is %.2f cm" % d)
            time.sleep(1)
    except KeyboardInterrupt:
        GPIO.cleanup()                              #释放 GPIO 资源
```

运行程序,以书本作为障碍物不断靠近超声波传感器模块,距离测量值不断变小,结果如图 3-28 所示。

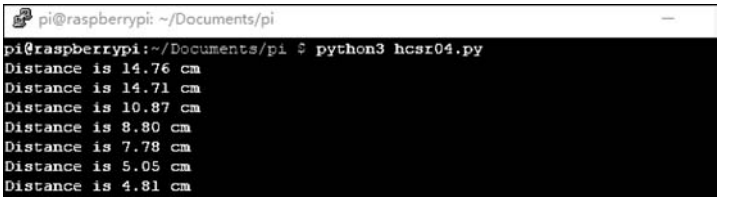


图 3-28　超声波测距结果