Powersensor API 手册

API reference for Powersensor

作者：xiaobo & 师

组织：Powersensor Developer

时间：2020 年 12 月

版本：V1.5

创新、便捷、高效、愉悦

更新说明

这个 API 手册是从 tutorial 中摘出来的整理汇编的我们自行开发的函数库的说明手册，这里分模块地介绍了各个类和函数的功能，适合有经验的用户查阅使用。由于我们的固件版本、tutorial 版本、api 版本是同步更新的，因此版本号也是同步，从 1.5 开始。

2020/12/25 更新：Powersensor API 手册 1.5，配套固件：img_2020.12

第一次发布

① 图像传感器模块

② 通信接口模块

③ 拓展板模块

④ fpga 加速模块

⑤ 其他模块

目录

第一章 图像传感器模块

内容提要

- 摄像头对象
- 参数配置

这章节介绍图像传感器相关的 API 函数。这些函数主要包含在Powersensor库中。引用这个库的正常写法是：

```
import PowerSensor as ps
```

1.1 摄像头对象

读取摄像头图像数据相关的函数封装在ImageSensor类中。

1.1.1 函数说明

其构造函数为：

```
# 获取主摄像头的图像对象
cam1 = ps.ImageSensor()
# 获取USB摄像头的图像对象
usbCam = ps.ImageSensor(source='usb')
```

获取摄像头对象后，读取图像的函数为：

```
imgMat = cam1 .read_img_ori()
```

笔记 Powersensor（1.5 及以上固件）支持 UVC 协议的 usb 摄像头。

1.1.2 参考例程

```
while(True):
  start = time.time()
  # 读取图像
  imgMat = cam1.read_img_ori()
  # 清空以往的显示
  clear_output(wait=True)
  # 显示最终的图像
  ps.CommonFunction.show_img_jupyter(imgMat)
  # 计算消耗时间
  end = time.time()
  # 打印时间
  print(end - start)
```

1.2 摄像头参数配置

由于相机的感光元件无法像人眼那样具备强大的自适应能力，因此需要在不同情景下设置不同的采集参数，以满足任务需求。

相机曝光是一个非常复杂的过程，曝光准确是使画面的亮度处于用户需求的状态。影响曝光的最重要的因素有三个，环境亮度、曝光时间、曝光增益，采集到的图像的亮度与环境亮度、曝光时间、曝光增益均正相关，因此在明亮的环境下要试用小的曝光时间和小的曝光增益，而黑暗的环境反之。

虽然加大曝光时间和曝光增益可以提高画面亮度，但是各有缺点，曝光时间大了会导致画面变糊，就是那种拖尾的感觉，曝光增益大了会增加画面的噪声，就是那种椒盐的感觉。所以用户要根据情况调整参数设置，拍快速运动的物体用高增益，拍静止高画质的东西用低增益。

Powersenser 支持自动曝光和手动曝光两种模式，自动曝光主要设置的参数有期望的亮度（流明数）和最大增益，即相机会根据设置的亮度和允许的最大增益自动调节曝光时间和曝光增益；复杂的多干扰的环境下，需要使用手动曝光模式，手动曝光需要设置的主要参数有曝光时间和曝光增益，即相机直接使用用户设置的参数来曝光成像。

白平衡的作用是让画面看起来更白，是某些场合非常有用。颜色识别建议使用固定白平衡。固定白平衡即使用出厂标定的固定的白平衡参数，自动白平衡使用实时估计的白平衡参数

本节相关的函数封装在`SensorConfig`类中。

笔记 仅适用于 powersensor 的原生摄像头。

1.2.1 函数说明

构造函数为：

```
config = ps.SensorConfig()
```

函数列表：

```
# 类名：SensorConfig
# ----------------------------------------
# 构造函数：SensorConfig(),无参数
# ----------------------------------------
# 明亮/弱光模式设置函数：set_light_mode(mode),
# -mode，必须，模式，可以是PowerSensor.SensorPara.Light_Dark（弱光模式）和
    PowerSensor.SensorPara.Light_Normal（明亮模式）种的一个
# ----------------------------------------
# 自动/手动曝光设置函数：set_exposure(mode, desired_lumin, max_gain, period,
    gain),
# -mode，必须，模式，可以是PowerSensor.SensorPara.EXPOSURE_MODE_MANUAL（手动
    曝光）和PowerSensor.SensorPara.EXPOSURE_MODE_AUTO（自动曝光，默认）种的一
    个；
```

```
# —desired_lumin，自动曝光需要，期望的亮度，值范围是（10—64），一般地明亮模式下推荐为56左右，弱光模式下推荐为15左右；
# —max_gain，自动曝光需要，最大增益，值范围是（16—63），按需要设置；
# —period，手动曝光需要，曝光时间，值范围是（10—30000），按需要设置；
# —gain，手动曝光需要，曝光增益，值范围是（16—63），按需要设置；
# ————————————————————————————————
# 白平衡设置函数：set_wb_mode(mode),
# —mode，必须，模式，可以是PowerSensor.SensorPara.AWB_Fix（固定白平衡）和PowerSensor.SensorPara.AWB_GrayWold（自动白平衡）种的一个
```

1.2.2 参考例程

1. 弱光模式

```
config.set_exposure(mode=ps.SensorPara.EXPOSURE_MODE_AUTO,
  desired_lumin=48, max_gain=64)
config.set_light_mode(ps.SensorPara.Light_Dark)
for i in range(50):
    start = time.time()          # 记录开始时间
    clear_output(wait=True)      # 清除图片，在同一位置显示，不使用会打印多张图片
    imgMat = cam1.read_img_ori()         # 读入图像

    # 缩小图像为320x240尺寸
    origin = cv2.resize(imgMat, (320,240))

    ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
    end = time.time()          # 记录结束时间
    print(end — start)
    time.sleep(0.1)

```

2. 明亮模式

```
config.set_exposure(mode=ps.SensorPara.EXPOSURE_MODE_AUTO,
  desired_lumin=48, max_gain=64)
config.set_light_mode(ps.SensorPara.Light_Normal)
for i in range(50):
    start = time.time()          # 记录开始时间
    clear_output(wait=True)      # 清除图片，在同一位置显示，不使用会打印多张图片
    imgMat = cam1.read_img_ori()         # 读入图像

    # 缩小图像为320x240尺寸
    origin = cv2.resize(imgMat, (320,240))

    ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
    end = time.time()          # 记录结束时间
```

```
    print(end - start)
    time.sleep(0.1)

```

3. 自动曝光模式

```
# 自动曝光，亮度=58，最大增益64
config.set_exposure(mode=ps.SensorPara.EXPOSURE_MODE_AUTO,
  desired_lumin=58, max_gain=64)
for i in range(30):
    start = time.time()         # 记录开始时间
    clear_output(wait=True)     # 清除图片，在同一位置显示，不使用会打印多张图片
    imgMat = cam1.read_img_ori()        # 读入图像

    # 缩小图像为320x240尺寸
    origin = cv2.resize(imgMat, (320,240))

    ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
    end = time.time()         # 记录结束时间
    print('0-58-64', end - start)
    time.sleep(0.1)

# 自动曝光，亮度=32，最大增益64
config.set_exposure(mode=ps.SensorPara.EXPOSURE_MODE_AUTO,
  desired_lumin=32, max_gain=64)
for i in range(30):
    start = time.time()         # 记录开始时间
    clear_output(wait=True)     # 清除图片，在同一位置显示，不使用会打印多张图片
    imgMat = cam1.read_img_ori()        # 读入图像

    # 缩小图像为320x240尺寸
    origin = cv2.resize(imgMat, (320,240))

    ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
    end = time.time()         # 记录结束时间
    print('0-32-64', end - start)
    time.sleep(0.1)

# 自动曝光，亮度=32，最大增益32
config.set_exposure(mode=ps.SensorPara.EXPOSURE_MODE_AUTO,
  desired_lumin=32, max_gain=32)
for i in range(30):
    start = time.time()         # 记录开始时间
    clear_output(wait=True)     # 清除图片，在同一位置显示，不使用会打印多张图片
    imgMat = cam1.read_img_ori()        # 读入图像
```

```
37
38      # 缩小图像为320x240尺寸
39      origin = cv2.resize(imgMat, (320,240))
40
41      ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
42      end = time.time()         # 记录结束时间
43      print('0-32-32', end - start)
44      time.sleep(0.1)
45
```

4. 手动曝光模式

```
1   # 手动曝光，时间500个单位，增益32
2   config.set_exposure(mode=ps.SensorPara.EXPOSURE_MODE_MANUAL, period
      =500, gain=32)
3   for i in range(30):
4     start = time.time()         # 记录开始时间
5     clear_output(wait=True)     # 清除图片，在同一位置显示，不使用会打印
      多张图片
6     imgMat = cam1.read_img_ori()        # 读入图像
7
8     # 缩小图像为320x240尺寸
9     origin = cv2.resize(imgMat, (320,240))
10
11    ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
12    end = time.time()         # 记录结束时间
13    print('1-500-32', end - start)
14    time.sleep(0.1)
15
16  # 手动曝光，时间2000个单位，增益32
17  config.set_exposure(mode=ps.SensorPara.EXPOSURE_MODE_MANUAL, period
      =2000, gain=32)
18  for i in range(30):
19    start = time.time()         # 记录开始时间
20    clear_output(wait=True)     # 清除图片，在同一位置显示，不使用会打印
      多张图片
21    imgMat = cam1.read_img_ori()        # 读入图像
22
23    # 缩小图像为320x240尺寸
24    origin = cv2.resize(imgMat, (320,240))
25
26    ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
27    end = time.time()         # 记录结束时间
28    print('1-2000-32', end - start)
29    time.sleep(0.1)
30
```

5. 固定白平衡

```
# 普通白平衡
config.set_wb_mode(ps.SensorPara.AWB_Fix)
for i in range(30):
  start = time.time()         # 记录开始时间
  clear_output(wait=True)     # 清除图片，在同一位置显示，不使用会打印多张图片
  imgMat = cam1.read_img_ori()        # 读入图像

  # 缩小图像为320x240尺寸
  origin = cv2.resize(imgMat, (320,240))

  ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
  end = time.time()         # 记录结束时间
  print('1—2000—32', end — start)
  time.sleep(0.1)

```

6. 自动白平衡

```
# 自动白平衡
config.set_wb_mode(ps.SensorPara.AWB_GrayWold)
for i in range(30):
  start = time.time()         # 记录开始时间
  clear_output(wait=True)     # 清除图片，在同一位置显示，不使用会打印多张图片
  imgMat = cam1.read_img_ori()        # 读入图像

  # 缩小图像为320x240尺寸
  origin = cv2.resize(imgMat, (320,240))

  ps.CommonFunction.show_img_jupyter(imgMat)# 打印用于差分的两张图片
  end = time.time()         # 记录结束时间
  print('1—2000—32', end — start)
  time.sleep(0.1)

```

第 二 章 通信接口模块

内容提要

- SPI
- I2C
- 串 口
- PWM 输出
- 普通 IO

这章节介绍基础通信接口相关的 API 函数，这些接口函数是与常见外部拓展模块通信的关键。这些接口位于主板的 2.54 间距的排针和一个 1.25 间距的 molex 接口上（串口），具体位置如下所示：

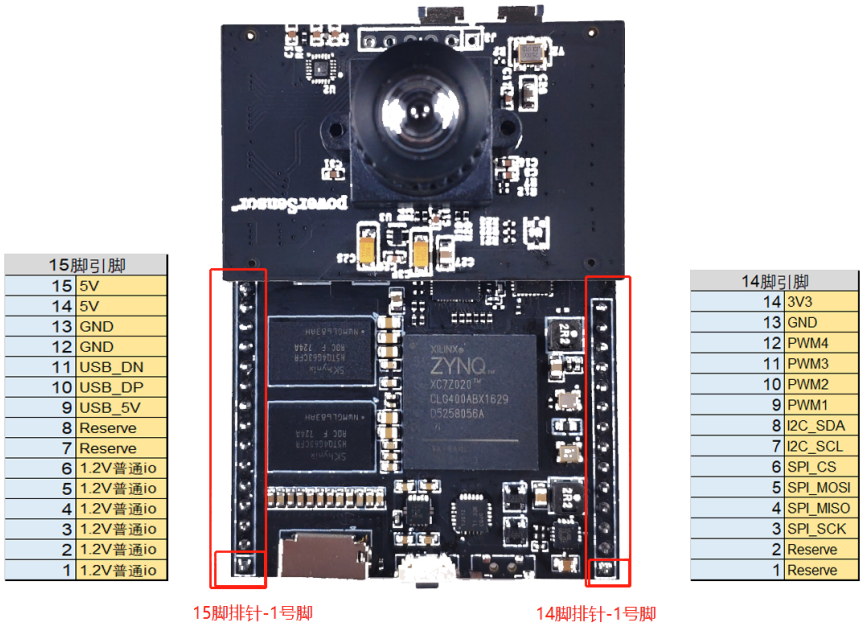


图 2.1: 背板管脚定义

molex 的串口接口管脚定义如图：

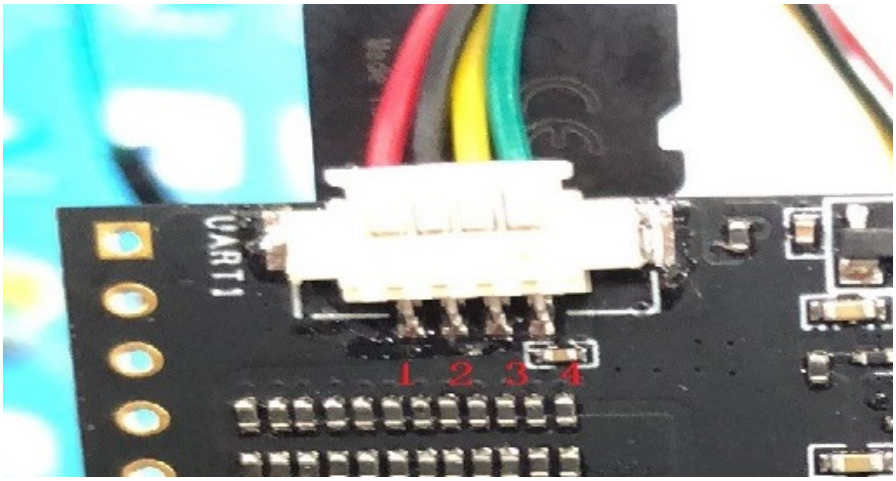


图 2.2: 串口管脚定义

从左往右 1，2，3，4 依次为：

表 2.1: 默认串口参数

管脚号：	1	2	3	4
功能：	TXD0	RXD0	GND	3V3

笔记 本章介绍的大部分通信接口的电平是 3.3V 的 TTL，除了普通 IO 的电平是 1.2V。

这些函数主要包含在Powersensor库中。引用这个库的正常写法是：

```
1 import PowerSensor as ps
```

2.1 SPI 接口

SPI 接口相关的函数封装在SpiPort类中。

2.1.1 原理介绍

SPI(Serial peripheral interface) 即串行外围设备接口，SPI 通讯需要使用 4 条线：3 条总线和 1 条片选：

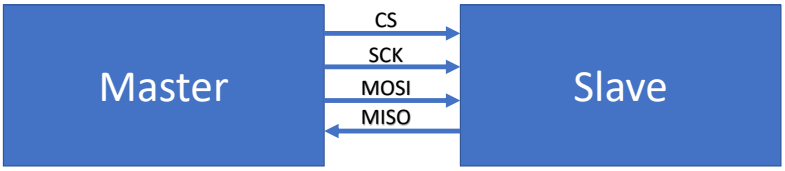


图 2.3: spi 主从

SPI 还是遵循主从模式，主机提供 sck、MOSI、CS（低电平有效），spi 协议支持多个从机，但 Powersenosr 只提供了一个 cs 脚，所以只能接一个外设。

1. cs(Slave Select)：片选信号线，用于选中 SPI 从设备。每个从设备独立拥有这条 cs 信号线，占据主机的一个引脚。设备的其他总线是并联到 SPI 主机的，即无论多少个从设备，都共同使用这 3 条总线。当从设备上的 cs 引脚被置拉低时表明该从设备被主机选中。
2. SCK(Serial Clock)：时钟信号线，通讯数据同步用。时钟信号由通讯主机产生，它决定了 SPI 的通讯速率。
3. MOSI(Master Ouput Slave Input)：主机 (数据) 输出/从设备 (数据) 输入引脚，即这条信号线上传输从主机到从机的数据。
4. MISO(Master Input Slave Ouput)：主机 (数据) 输入/从设备 (数据) 输出引脚，即这条信号线上传输从机从到主机的数据主从机通过两条信号线来传输数据。

SPI 协议根据时钟信号的时钟极性（CPOL）和时钟相位（CPHA）可以分为四种情况：

时钟极有两种选择，0 代表空闲状态 sck 为低，1 代表空闲状态 sck 为高，时钟相位也有两种选择，0 代表第一个沿采样，1 代表第二个沿采样。两两结合，所以总共有四种情况。

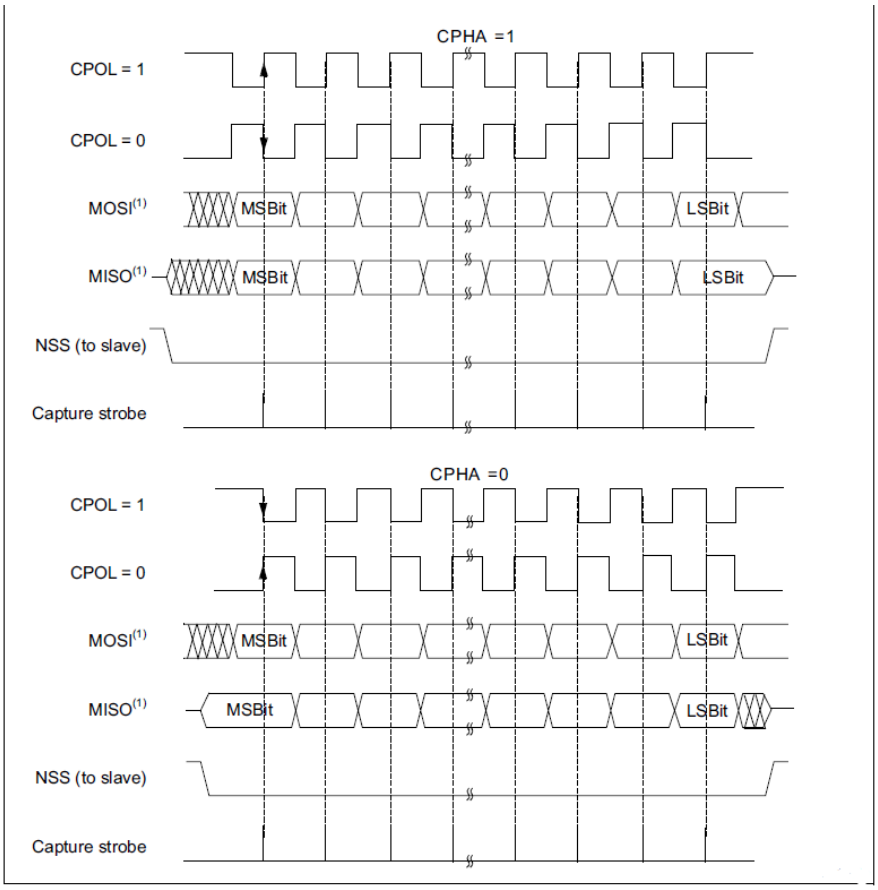


图 2.4: CPOL 和 CPHA 对 sck 的影响

注 Powersensor 目前支持的数据结构都是 byte 类型的（即 8 位的整形），数据是 MSB 的，就是最高位先发。

2.1.2 函数说明

```
# 类名：SpiPort
# ----------------------------------------
# 构造函数：SpiPort(),无参数
# ----------------------------------------
# 配置函数：set_clock(speed, cpol, cpha),
# -speed, 必须, 通信速率1Hz-10MHz
# -cpol, 必须, 时钟极性, 见原理介绍
# -cpha, 必须, 时钟相位, 见原理介绍
# ----------------------------------------
# 读写函数：swap_data(to_send),
# -to_send, 需要发送的数据, 必须是list类型, 数据类型为uint8
# -return, 返回值是接受到的数据, spi是双工的, 发送的同时会收到数据
```

2.1.3 参考例程

```
# 初始化SPI, 运行一次即可
spi = ps.SpiPort()

spi.set_clock(1000000, 0, 0)

# 读写数据, 回环测试：请将mosi和miso短接, 这样发送数据的时候就可以收到数据
# 读写的数据类型为list, dtype为uint8
to_send = [1, 3, 5, 7]
print(spi.swap_data(to_send))
```

单机实验时需要进行回环测试，把 spi 接口的 mosi 和 miso 用杜邦线短接，运行程序即可看到接受到的发送的数据。

2.2 I2C 接口

I2C 接口相关的函数封装在`I2cPort`类中

2.2.1 原理介绍

I2C 是一种两线的总线结构通信协议，分主从机，一般一个总线上有一个主机和多个从机，写数据的主要过程如下:

1. 主机发出开始信号
2. 主机接着发出一字节的从机地址信息，包含 7 位从机地址，即最低位的读写信号（1 为读、0 为写）
3. 从机发出应答信号
4. 主机开始发送信号，每发完一字节后，从机发出应答信号给主机
5. 主机发出停止信号

时序图如下：

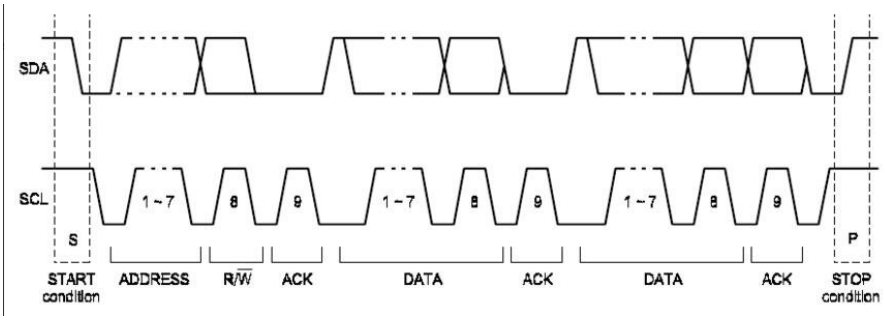


图 2.5: I2C 时序

读的过程与写数据的过程类似，不过在发完从机地址后，接的是重启动信号加地址+1（读信号)，然后主机释放总线，由从机控制数据线，主机负责应答。

I2C 通信的详细过程可以上网搜索，不过现在这些过程一般都硬件实现了，Powersensor 只需要通过简单的函数调用即可完成 I2C 通信。

笔记 Powersensor 支持的从机地址是 7 位的。考虑到兼容性，Powersenosr 的 I2C 管脚没有焊上拉电阻，因此在使用时需要根据实际情况外接合适的上拉电阻。

2.2.2 函数介绍

```
# 类名：I2cPort
# ----------------------------------------
# 构造函数：I2cPort(),无参数
# ----------------------------------------
# 配置函数：set_slave_addr(slave_addr),
# -slave_addr，必须，设置从机地址
# ----------------------------------------
# 单字节写函数：write_byte(reg_addr, value),
# -reg_addr，需要写入的地址（寄存器地址，不是从机地址）
# -value，需要写入的值
# ----------------------------------------
# 单字节读函数：write_byte(reg_addr),
# -reg_addr，需要读的地址（寄存器地址，不是从机地址）
# -retuan，uint8类型，读到的数据
# ----------------------------------------
# 块写函数：write_block(reg_addr, data),
# -reg_addr，需要写入的地址（寄存器地址，不是从机地址）
# -data，需要写入的数组，list类型，数据类型位uint8
# ----------------------------------------
# 块写函数：read_block(reg_addr, data, num),
# -reg_addr，需要读取的地址（寄存器地址，不是从机地址）
# -num，需要读取的数据的数量
# -return，  返回的读取的数据
```

2.2.3 参考例程

i2c 无法回环测试，需要外接一个设备，这里以 mpu6050 举例，示例程序如下

```
import PowerSensor as ps
# 初始化i2c，运行一次即可
i2c = ps.I2cPort()

# 配置从机地址，注意，这里使用的是7位的从机地址，
# mpu6050文档上写的从机地址是0xD0，这个是8位地址，需要右移一位变成0x68
i2c.set_slave_addr(0x68)

# 这里读取mpu6050的ID寄存器，这个寄存器是只读的，值为104
imu_id = i2c.read_byte(117)
print(imu_id)
```

2.3 异步串口通信

异步串口接口相关的函数封装在`UsartPort()`类中。

2.3.1 原理介绍

串口是 powersensor 把图像处理的结果（跟踪物体的位置，速度，二维码读取结果等）交付给单片机/pc 等设备的重要方式。

Powersenosr 的串口是一种单机对单机的双线全双工异步通信方式，没有时钟线，双方通过波特率校准，因此需要较高的时钟精度，默认的串口参数是

表 2.2: 默认串口参数

项	项	项	值
波特率：	115200	奇偶校验：	无校验
数据长度：	8	停止位：	1

串口接口的接线顺序见本章的第一节，串口接线时注意 tx 接 rx，rx 接 tx。

2.3.2 函数介绍

```
# 类名：UsartPort
# ----------------------------------------
# 构造函数：UsartPort(),无参数
# ----------------------------------------
# 波特率设置函数：set_baudrate(baudrate),
# -baudrate，必须，波特率
# ----------------------------------------
# 字符串打印函数：u_print(str),
```

```
# -str, 需要打印的字符串
# ----------------------------------------
# 字节数组打印函数: u_send_bytes(arr),
# -arr, 需要输出的字节数组
# ----------------------------------------
# 按字节数读取函数: u1.read(num),
# -num, 需要读取的字节数
# ----------------------------------------
# 读取缓存区所有字节函数: u1.read_all(),
# -无参数
```

2.3.3 参考例程

1. 字符串输出实验，直接用 print 输出 ascii 字符串，适合与 pc 通信

```
# 设置波特率, 只需要设置一次, 默认为115200
s1 = ps.UsartPort()
s1.set_baudrate(115200)
for i in range(2):
s1.u_print('hello world!\n')

```

笔记 注意串口调试助手要设置正确的参数。

2. 字节数组输出实验，输出可以自定义数据包结构的字节数组，适合与单片机通信

```
# 设置波特率, 只需要设置一次, 默认为115200
s1.set_baudrate(9600)
for i in range(2):
s1.u_send_bytes([1, 3, 5, 7, 9])

```

这里只是举例，输出的是 1，3，5，7，9 数组，正常与对单片机通信时应该有完整的包头包尾校验字，方便单片机捡包和校验，如

```
pack = np.aray([0xa5, 0x08, 数据0, 数据1, 数据2, 数据3, 数据4, 数据5,
  校验)

```

注 其中 0xa5 是包头，0x08 是包的长度，校验可以是 5 个数据的求和的后 8 位。

2.4 PWM 输出

异步串口接口相关的函数封装在`PwmActuator()`类中。

2.4.1 原理介绍

PWM 脉冲宽度调制技术，是一系列可以对脉冲的宽度进行调制的方波，来等效地获得所需要波形（含形状和幅值），在电机控制、led 亮度等场合有广泛的应用。

PWM 波有几个重要的概念：

1. 时钟源 (设频率为 f_s)，PWM 一般是一种频率固定，高电平时间可以调节的矩形波，时钟源决定 pwm 的精度、最高频率。Powersensor 的 pwm 时钟源是 100Mhz
2. 预分频 (设为 n_f)，由于时钟源的频率较高，为了计算方便以及得到低频的信号，会使用预分频器对时钟源进行分频。Powersensor 的预分频器是 16 位的。
3. 计数器，计数器的单位是时钟源经过预分频后的时钟信号的周期，Powersensor 使用的计数器是 32 位的。
4. 周期/频率 (设周期为 n_p)，周期是指生成的 PWM 的一个高电平和一个低电平的总时长，单位是计数器计数的个数，频率是周期的倒数。
5. 高电平时间/占空比，高电平时间是指 PWM 一个周期内信号的高电平时间，单位是计数器计数的个数，占空比是高电平时间与周期的比值。

pwm 波的频率与时钟源的关系为：

$$f_{pwm} = \frac{f_s}{(n_f + 1)(n_p + 1)}$$

PWM 产生的时序图如下所示:

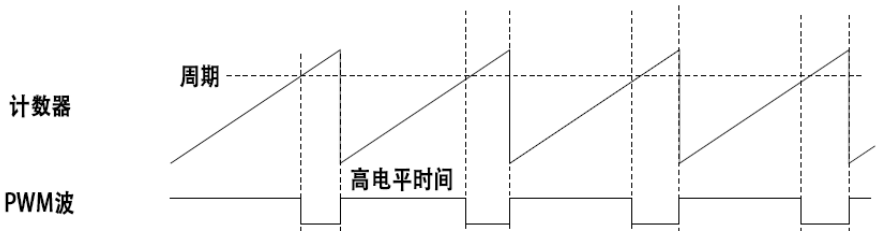


图 2.6: PWM 时序

在电机控制的半桥电路中，一般需要两个极性相反的 PWM 来控制两个 mos 管的开关，为了防止瞬间出现的上下两个 mos 同时开启导致的电源短路问题，一般要设置死区，死区的单位也是计数器计数的个数，死区就是高电平前的延时时间（为了等待另一路 mos 管彻底关闭）。Powersensor 提供了两种 pwm 模式，普通模式和互补模式，普通的模式是 4 路周期固定，脉冲宽度可以调节的 pwm 波，互补模式是两路（也是 4 个信号）宽度可调节的 pwm 信号，每路含两个极性相反的 pwm 信号，同时带死区控制。带死区控制的波形如下所示：

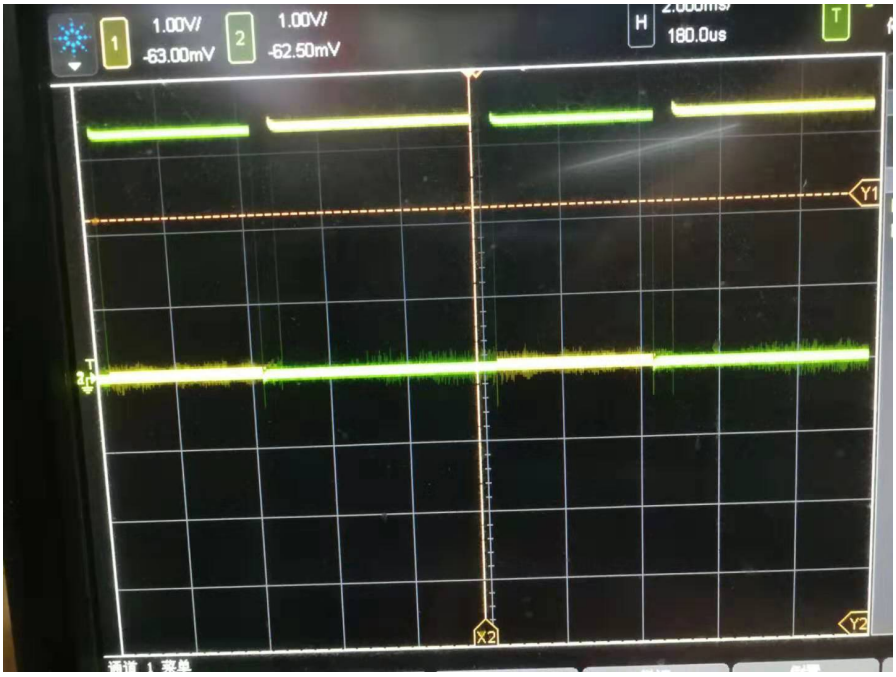


图 2.7: 互补 pwm

2.4.2 函数介绍

```
# 类名：PwmActuator
# ------------------------------------------
# 构造函数：PwmActuator()，无参数
# ------------------------------------------
# 属性：
# -preDividor，预分频，16位整形
# -period，  周期，32位整型
# -mode，模式，0关闭输出，1普通模式，2互补输出模式
# -preDeadZonee，死区时间，互补模式下起作用
# -pwm0,pwm1,pwm2,pwm3，高电平时间，整形，最大值位周期，互补模式时只有pwm0和
    pwm2起作用
# ------------------------------------------
# 配置函数：pwm_setup(enable)，
# -enable，必须，设置为True时，上述配置的属性才会起作用，否则输出低电平
# ------------------------------------------
# 通道高电平时间设置：pwm_set_width(chanel, width)，
# -chanel，通道选择，0-3
# -width，高电平时间，整形，最大为周期
```

2.4.3 参考例程

```
import PowerSensor as ps
# 初始化pwm对象，运行一次即可
pwmx = ps.PwmActuator()

```

```
# 普通模式
# ------------------------------------------------
# 预分频，实际分频数=设定值+1
pwmx.preDividor = 10 - 1
# 死区，实际的周期=设定值+1
pwmx.period = 1000 - 1
# 高电平时间，最大为周期的设定值，即999
pwmx.pwm0 = 200
pwmx.pwm1 = 400
pwmx.pwm2 = 600
pwmx.pwm3 = 800
# 模式设置，0：低电平；1：普通模式；2：互补输出模式；
pwmx.mode = 1
# 写入设置，写入这句后，上述配置的内容才起效
pwmx.pwm_setup(True)

# 互补模式
# ------------------------------------------------
# 互补模式下的pwm输出
# 预分频，实际分频数=设定值+1
pwmx.preDividor = 10 - 1
# 周期，实际的周期=设定值+1
pwmx.period = 1000 - 1
# 死区，即互补的两个信号的高电平间的间隔，实际值为设定值+1
pwmx.preDeadZonee = 50 - 1
# 互补滤波通道1，高电平时间，最大为周期的设定值，即999
pwmx.pwm0 = 200
# 互补滤波通道2，高电平时间，最大为周期的设定值，即999
pwmx.pwm2 = 600
## 此模式下，pwm1和pwm3不起作用
# 模式设置，0：低电平；1：普通模式；2：互补输出模式；
pwmx.mode = 1
# 写入设置，写入这句后，上述配置的内容才起效
pwmx.pwm_setup(True)
# 预分频，实际分频数=设定值+1
pwmx.preDividor = 10 - 1

# 动态改变高电平时间
# ------------------------------------------------
pwmx.pwm_set_width(0, 300)
```

2.5 普通 IO

异步串口接口相关的函数封装在GpioPort()类中。

2.5.1 原理介绍

对应的硬件是排针引脚上的 6 个 1.2V 的普通 IO。普通 IO 有输出和输入两种模式，输出模式下可以输出高电平或者低电平，输入模式下可以读取外界输入的电平。

笔记 这几个 Gpio 是 1.2V 的，不要输入过高电平，若需要其他电平请设置电平转换电路。一个简单的参考电路如下：

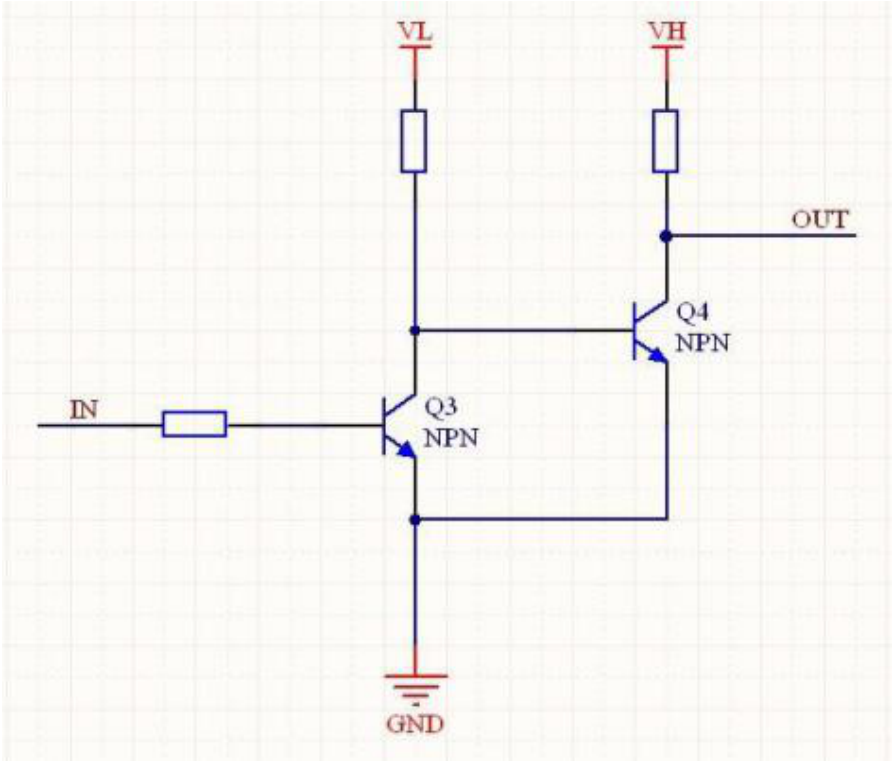


图 2.8: 输出电平转换电路

2.5.2 函数介绍

```
# 类名: GpioPort
# ----------------------------------------
# 构造函数: GpioPort(),无参数
# ----------------------------------------
# 配置函数: set_mode(pos, mode),
# -pos, 必须, 管脚号, 从0开始, 到5结束
# -mode, 必须, 可以是.PortPara.Gpio_Mode_Out和PortPara.Gpio_Mode_In中的一个,
    即输出模式或输入模式
# ----------------------------------------
# 配置函数: set_value(pos, value),
# 仅对输出模式的管脚有效
# -pos, 必须, 管脚号, 从0开始, 到5结束
# -value, 必须, 可以是.PortPara.Gpio_Value_High和PortPara.Gpio_Value_Low中的
    一个, 即输出高电平或低电平
# ----------------------------------------
# 电平读取函数: read_value(pos),
```

```
# 仅对输入模式的管脚有效
# —pos，必须，管脚号，从0开始，到5结束
# —return，返回值，高电平返回1，低电平返回0
# ----------------------------------------
```

2.5.3 参考例程

```
# 初始化控制对象
gpiox = ps.GpioPort()

# 设置输出
gpiox.set_mode(1, ps.PortPara.Gpio_Mode_Out)
gpiox.set_mode(3, ps.PortPara.Gpio_Mode_Out)
gpiox.set_mode(4, ps.PortPara.Gpio_Mode_Out)
gpiox.set_mode(5, ps.PortPara.Gpio_Mode_Out)

gpiox.set_value(1, ps.PortPara.Gpio_Value_High)
gpiox.set_value(3, ps.PortPara.Gpio_Value_Low)
gpiox.set_value(4, ps.PortPara.Gpio_Value_Low)
gpiox.set_value(5, ps.PortPara.Gpio_Value_High)

# 设置输入
gpiox.set_mode(0, ps.PortPara.Gpio_Mode_In)
gpiox.set_mode(2, ps.PortPara.Gpio_Mode_In)
# 设置电平
for i in range(15):
x1 = gpiox.read_value(0)
x2 = gpiox.read_value(2)
print(x1, x2)
time.sleep(1)
```

输出模式的管脚可以通过万用表测量来验证程序，注意第一个管脚是 0 号脚。

输入模式可以这样验证：输入程序是进行一秒一次的轮询管脚读取，可以使用金属短路管脚，0-1，或者 1-2 管脚，此时读书会变成 1（因为上一段程序把 1 号脚设成输出，输出高电平）。

第 三 章 拓展板模块

内容提要

- IO 拓展板
- 电机拓展板
- 云台多机板
- 16pwm 舵机板
- tof 测距板
- 红外测温板
- 热成像板
- SPI 屏幕

这章节介绍各个拓展模块相关的函数。一部分模块是基于原始通信接口，但为了保证通信可靠性而设计的带隔离功能的拓展板，其相关的函数包含在`Powersensor`库中。另一部分是引入其他芯片，增加额外的功能，每个拓展板独立，其相关的函数包含不同的库中，分别是`Pca9685`，`Vl531x\_helper`，`Mlx90621`。引用这些库的正常写法是：

```
import PowerSensor as ps
import Mlx90621
import Vl531x_helper
import Pca9685
import LeptonHelper
import Lcd_helper
```

3.1 IO 和串口板

这块拓展板主要用于与 plc 之类工业现场其他设备通信时起信号隔离功能。提供了 4 路输入（0-3），两路输出（4-5）的光耦隔离 IO。串口提供了 ttl 电平到 232 电平转换的功能。

GPIO 和串口相关的函数在前面章节介绍 (2)，此处不再赘述。

3.2 电机驱动板

这块拓展板主要通过功率元件将 pwm 信号转换为电机控制信号，可以直接驱动两路小型直流电机。具体的使用案例就是双轮小车。

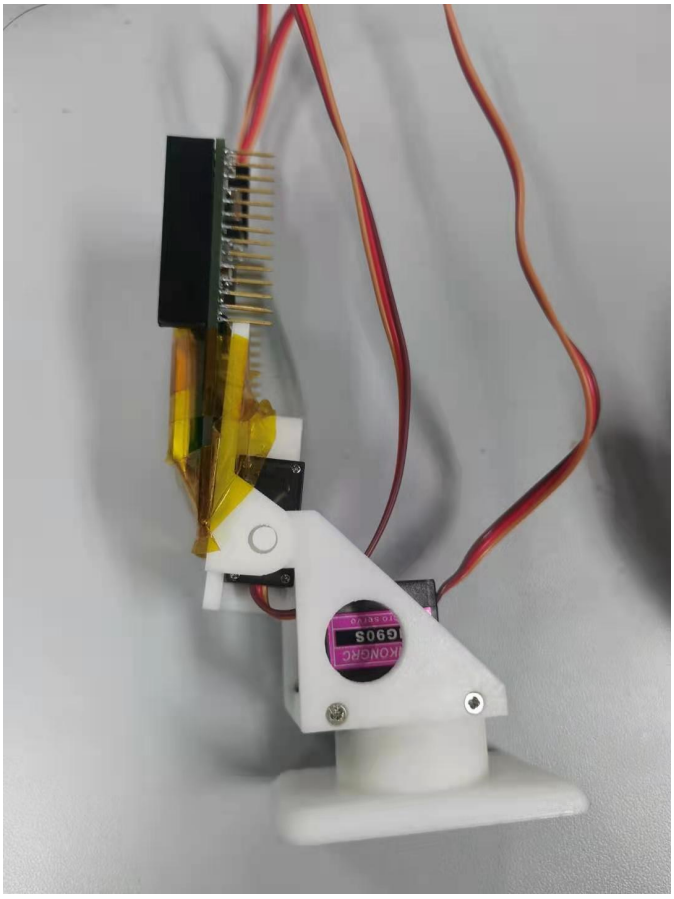
需要用到的函数有 GPIO 和 PWM 相关的部分，在前面章节介绍过 (2)。此处不再介绍。

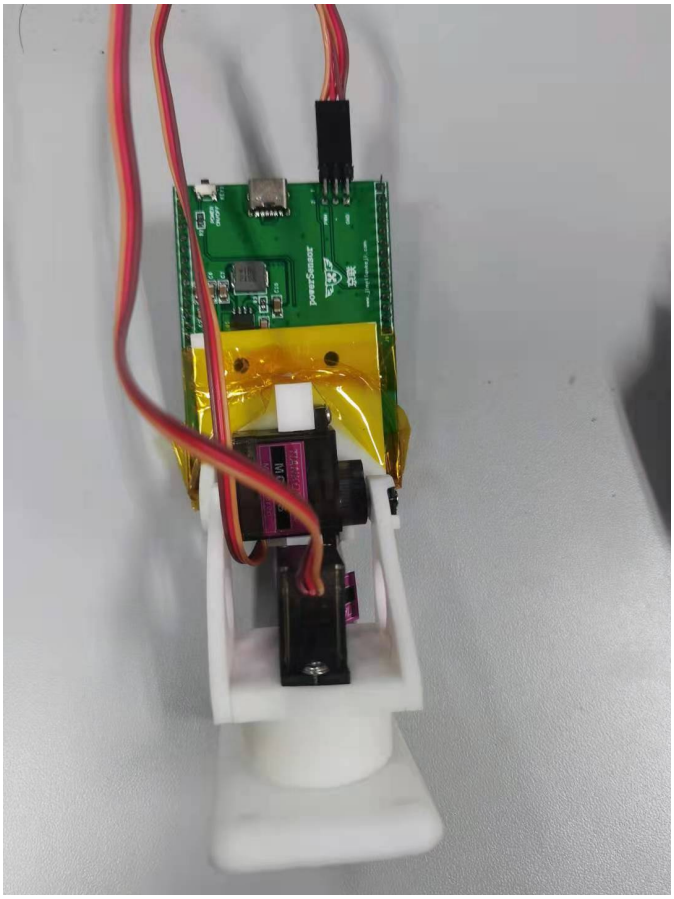
板子具体使用的例程见总教程。

3.3 舵机云台板

这块拓展板主要用于双舵机云台的驱动。

舵机云台如下图所示，主要由两个舵机和相应的控制模块组成





两个舵机可分别控制云台进行水平和垂直方向上的运动。

需要用到的函数是 PWM 相关的部分，在前面章节介绍过 (2)。此处不再介绍。

3.4 16 路 pwm 拓展板 - PCA9685

这块拓展板提供 16 路 pwm 波的输出功能，主要用于多关节舵机机械臂等领域。拓展板采用 I2C 协议控制 PCA9685，完成 16 路 pwm 波信号的驱动和控制。

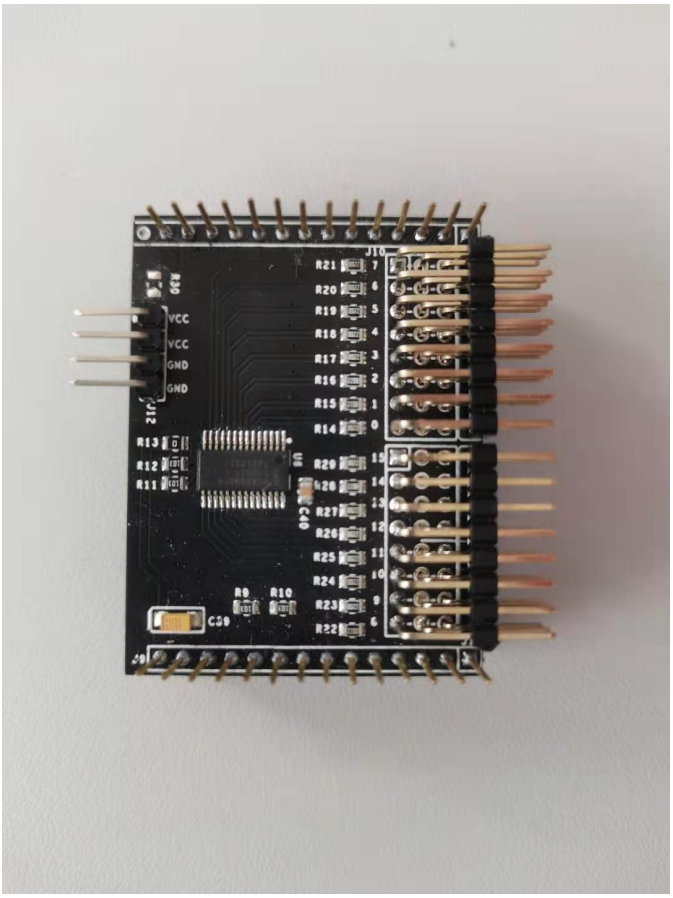


图 3.1: PCA9685-16 路 pwm 拓展板

3.4.1 函数说明

其构造函数为：

```
# 0x60是Pca9685的从机地址，用户可根据自己需求修改
pwm16 = Pca9685.PCA9685(0x60)
```

函数列表：

```
# 类名：PCA9685
# ------------------------------------------
# 构造函数：PCA9685(),无参数
# ------------------------------------------
# 频率设置函数：setPWMFreq(fre),
# -fre，必须，需要输出的pwm的频率，单位Hz
```

```
# ----------------------------------------
# 脉冲宽度设置函数: setServoPulse(chanel, value),
# -chanel, 必须, 通道选择, 选择范围0-16;
# -value, 必须, 高电平时间, 单位微秒;
# ----------------------------------------
```

注 舵机驱动的高电平时间一般是 0.5 毫秒到 2.5 毫秒。另外使用舵机注意机械零位校准和打死的问题。

3.4.2 参考例程

```
pwm16 = Pca9685.PCA9685(0x60)
pwm16.setPWMFreq(50) # 频率

for i in range(16):
pwm16.setServoPulse(i, i * 50 + 1000) # PWM frequency is 50HZ,the period is
    20000us
```

3.5 TOF 测距板 - Vl531X

这块拓展板提供激光测距功能，测距范围大致 0.02 米到 4 米。拓展板采用 I2C 协议控制 Vl531X，完成测距和数据采集。

图 3.2: TOF 测距模块-Vl531X

注 ToF 是 time of fly 的缩写，即测量的是光飞行的时间，因此相比于对焦测距而言精度较高，稳定性较好。

3.5.1 函数说明

其构造函数为：

```
i2c1 = ps.I2cPort()
sensor = Vl531x_helper(i2c1)
```

函数列表：

```
# 类名：Vl531x_helper
# ----------------------------------------
# 构造函数：Vl531x_helper(i2c),
# - i2c，必须，powersensor的i2c对象
# ----------------------------------------
# 探测范围设置函数：vl53l1x_setDistanceMode(dis)
# - 配置函数，设置最大测量距离，距离越长需要的探测时间越长；
# - dis，必须，探测范围，字符串类型，可以是'long', 'medium', 'short'中的一个；
# - 返回值，无
# ----------------------------------------
# 最大探测时间设置函数：vl53l1x_setMeasurementTimingBudget(tim)
# - tim，必须，最大探测时间，单位微秒，long模式建议33000；
# - 返回值，无
# ----------------------------------------
# 启动测量函数：sensor.sensor_start(tim)
# - tim，必须，采样周期，单位毫秒，原则上要大于最大测量时间；
# - 返回值，无
# ----------------------------------------
# 接收到新数据事件函数：sensor_new_data()
# - 无参数
# - 返回值，布尔型，True为收到数据，Flase没有收到数据；
# ----------------------------------------
# 读取测量值函数：sensor_read_mm()
# - 无参数
# - 返回值，浮点数，读到的距离值，单位为毫米
# ----------------------------------------
# 停止传感器函数：sensor.sensor_stop()
# - 无参数
# - 返回值，无
```

3.5.2 参考例程

```
i2c1 = ps.I2cPort()
sensor =  Vl531x_helper.vl531x_helper(i2c1)
# 1. 传感器初始化
sensor.sensor_init()
# 2. 设置参数
# 2.1 最大距离参数，可以是'long', 'medium', 'short'中的一个
sensor.vl53l1x_setDistanceMode('short')
# 2.2 设置最大测量时间，单位微秒，medium模式建议10000
```

```
sensor.vl53l1x_setMeasurementTimingBudget(33000)
# 3. 启动测量
sensor.sensor_start(50)
# 4. 周期性数据读取
for i in range(500):
  clear_output(wait=True)
  over_time = 0
  while(not sensor.sensor_new_data()):
    over_time = over_time + 1
    time.sleep(0.001)
  if over_time > 200:
    print('over tiem')
    break
  print(sensor.sensor_read_mm())
  time.sleep(0.1)
```

3.6 红外温度阵列测量板 - mlx90621

这块拓展板提供红外非接触测温功能，阵列大小为 16x4。拓展板采用 I2C 协议控制 mlx90621，完成测温功能。

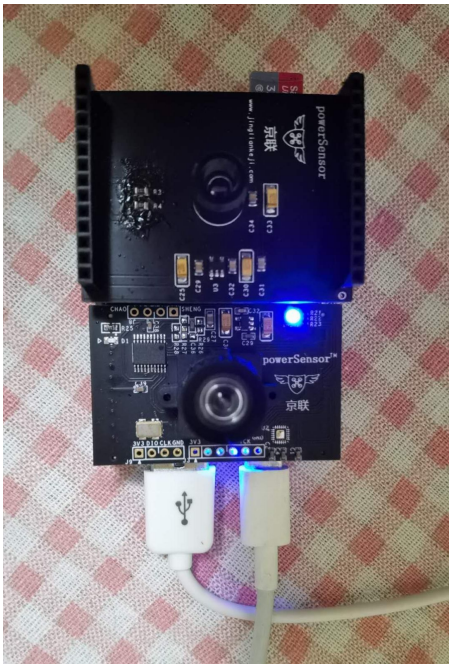


图 3.3: 红外测温模块 - mlx90621

注 测量温度范围：−20° 到 300°。

3.6.1 函数说明

其构造函数为：

```
i2c1 = ps.I2cPort()
sensor = Mlx90621(i2c1)
```

函数列表：

```
# 类名：Mlx90621
# ----------------------------------------
# 构造函数：Mlx90621(i2c),
# - i2c，必须，powersensor的i2c对象
# ----------------------------------------
# 初始化函数：sensor_init(),
# - 无参数
# - 返回值，无
# ----------------------------------------
# 读取到校正的温度数据函数：sensor_read()
# - 无参数
# - 返回值，字典类型，4项：(1)传感器本身温度，浮点型；(2)温度阵列(16x4)，浮点型；(3)最小温度，浮点型；(4)最大温度，浮点型；
# ----------------------------------------
# 伪色图像读取函数：sensor_get_img(block_size=15)
# - block_size，填充的色块大小；
# - 返回值，三维的温度伪色图像；
```

3.6.2 参考例程

```
# 1. 初始化i2c，即准备用于操作红外测温传感器的i2c
i2c = ps.I2cPort()
# 2. 实例化红外测温传感器的对象
sensor = Mlx90621.mlx90621_helper(i2c)
# 3. 初始化红外测温传感器
sensor.sensor_init()
# 4. 读取传感器温度信息
x1, x2, x3, x4 = sensor.sensor_read()
print("传感器本身温度是：  " + str(x1))
print("测量（看到）温度阵列是：  " + str(x2))
print("测量温度最小值是：  " + str(x3))
print("测量温度最大值是：  " + str(x4))
```

3.7 热成像仪板 - lepton 3.5

这块拓展板提供红外非接触测温功能，阵列大小为 160x120。拓展板采用 I2C 协议配置参数，采用 SPI 传输数据，主芯片是 lepton 3.5。

图 3.4: 热成像仪 - Lepton 3.5

注 与 mlx90621 相比，lepton 的测量范围和效果大大提升。温度探测范围 −10° 到 400°。

3.7.1 函数说明

其构造函数为：

```
# 初始化i2c，运行一次即可
i2c = ps.I2cPort()
spi = ps.SpiPort()
spi.set_clock(40000000, 1, 1)
# 调用LeptonHelper辅助类
lepton = LeptonHelper.Lapton_Helper(i2c, spi)
```

函数列表：

```
# 类名：Lapton_helper
# ----------------------------------------
# 构造函数：Lapton_helper
# - i2c，Powersenosr的i2c对象
# - spi，Powersensor的spi对象
# ----------------------------------------
# 指令函数：commond(moduleId, commondId, comond),
# - moduleId
# - commondId
# - comond
# - 返回值，无
# ----------------------------------------
# 寄存器读取函数：read_reg(cmd),
# - cmd，必须，指令
# - 返回值，指令处理结果
```

```
# ----------------------------------------
# 指令结果读取函数: lread_data(),
# - 无参数
# - 返回值, 指令执行结束后的结果
# ----------------------------------------
# 以上3个函数提供给专业用户使用
# ----------------------------------------
# 读取热成像图像数据: sensor_read_frame()
# - 无参数
# - 返回值, 二维数组, 整形, 14位ad采样结果;
```

3.7.2 参考例程

```
import PowerSensor as ps
import time
import numpy as np
import LeptonHelper
# 初始化i2c, 运行一次即可
i2c = ps.I2cPort()
spi = ps.SpiPort()
spi.set_clock(40000000, 1, 1)
# 调用LeptonHelper辅助类
lepton = LeptonHelper.Lapton_helper(i2c, spi)
print('左边是冷水, 右边是热水')
img_buf = lepton.sensor_read_frame()
plt.imshow(img_buf)
plt.title('Ir image')
plt.show()
cam1 = ps.ImageSensor()
img_color = cam1.read_img_ori()
plt.imshow(img_color)
plt.title('Color image')
plt.show()
```

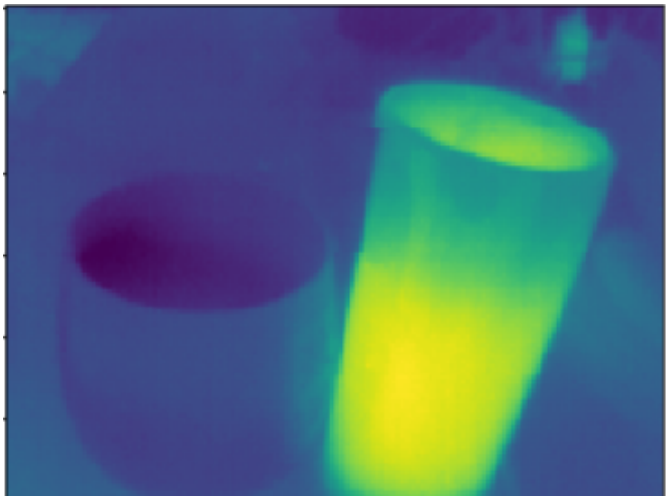


图 3.5: Lepton 测量效果

3.8 2.8 寸 TFT 显示屏

这块拓展板提供 TFT 显示功能，分辨率是 320x240，通信协议是 spi。全屏的刷新率大概 8 帧左右。

图 3.6: TFT 显示屏

注 使用 120x160 的分辨率进行小区域的刷屏可以有效提高帧率。

3.8.1 函数说明

屏幕的坐标系为：

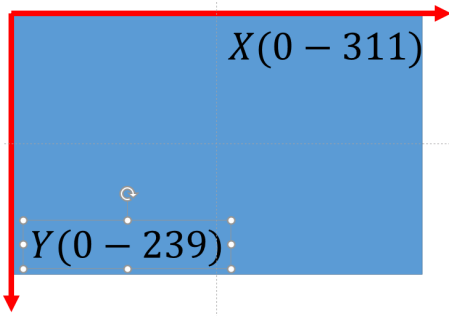


图 3.7: TFT 显示屏的坐标系

其构造函数为：

```
# 初始化spi和相关io，运行一次即可
spi = ps.SpiPort()
spi.set_clock(100000000, 1, 1)
gpio = ps.GpioPort()
gpio.set_mode(0, ps.PortPara.Gpio_Mode_Out)
gpio.set_value(0, ps.PortPara.Gpio_Value_Low)
gpio.set_mode(1, ps.PortPara.Gpio_Mode_Out)
gpio.set_value(1, ps.PortPara.Gpio_Value_High)
# 调用LeptonHelper辅助类
lcd = Lcd_helper.Ili9341_helper(spi=spi, dc=gpio)
```

函数列表：

```
# 类名：Ili9341_helper
# ----------------------------------------
# 构造函数：Ili9341_helper(spi, dc)
# - spi，Powersensor的spi对象
# - dc，Powersenosr的gpio对象，用于控制IO管脚
# ----------------------------------------
# 绘图函数：LCD_Draw(sx, sy, img_data, color=0xff00),
# 这个函数在指定的起点开始，填充一个小图片
# - sx，整形，起点的x坐标
# - sy，整形，起点的y坐标
# - img_data，整形的2维，3维数组，opencv的mat对象；如果是3维图像，为彩色图像；如果是2维图像，会绘制成color指定颜色的单色图像，所以请保证img_data为二值化后的图像。
# - color，整形的16位，当img_data为二维图像时起作用，为单色图像的颜色
# - 返回值，无
# ----------------------------------------
# 清屏函数：LCD_Clear(color),
# - color，清屏的颜色，16位无符号整形，rgb565格式。在Lcd_helper模块中提供了几种参考色，WHITE，BLACK，BLUE，BRED，GRED，GBLUE，RED，MAGENTA，GREEN，CYAN，YELLOW，BROWN，BRRED，GRAY；
# - 返回值，无
```

除了上述的上层函数外，我们还提供了一些底层函数给高级用户使用：

```
# ----------------------------------------
# 设置spi波特率：set_spi_rate(bauterate),
# 可以通过这个函数设置spi通信速度，也就可以修改spi通信的速度。大于40M的速度可能带来不稳定
# - bauterate，整形，波特率
# - 返回值，无
# ----------------------------------------
# 画图窗口设置函数：LCD_Set_Window(sx, sy, width, height),
# 注意这个坐标系是竖屏的坐标系，与上层函数的横屏坐标系不一样。
# - sx，整形，起点的x坐标
# - sy，整形，起点的y坐标
```

```
# - width, 整形, 宽度
# - height, 整形, 高度
# - 返回值, 无
# ---------------------------------------
# 像素RAM预写入函数: LCD_WriteRAM_Prepare(),
# 这个函数用在设置窗口函数之后, 代表准备写入像素值。
# - 无参数
# - 返回值, 无
# ---------------------------------------
# 像素RAM写入: LCD_Write_Ram(data),
# 这个函数用在像素RAM预写入函数之后, 比如, 窗口设置了20x20共400点后, 使用预写入函数后, 即可通过这个函数写入400x2个byte到ram区。图像的格式是rgb565, 因此需要乘2。
# - data, list类型, 每个元素为byte, 像素值
# - 返回值, 无
```

3.8.2 参考例程

```
import PowerSensor as ps
import time
import cv2
import numpy as np
from IPython.display import clear_output
import Lcd_helper

spi = ps.SpiPort()
spi.set_clock(100000000, 1, 1)
# spi.set_clock(1000000, 1, 1)
gpio = ps.GpioPort()
gpio.set_mode(0, ps.PortPara.Gpio_Mode_Out)
gpio.set_value(0, ps.PortPara.Gpio_Value_Low)
gpio.set_mode(1, ps.PortPara.Gpio_Mode_Out)
gpio.set_value(1, ps.PortPara.Gpio_Value_High)

lcd = Lcd_helper.Ili9341_helper(spi=spi, dc=gpio)

lcd.device_init()

img = cv2.imread('/home/debian/bk.jpg')
# img = cv2.imread('./img/flower.jpg')
img = cv2.resize(img, (320,240))
lcd.LCD_Draw(0, 0, img)

time.sleep(3)
lcd.LCD_Clear(Lcd_helper.BLUE)
```

第 四 章 FPGA 加速模块

内容提要

- 人工智能
- 并行加速

这章节介绍 FPGA 加速相关的函数。目前主要支持 xilinx 的 dpu 加速库。引用这些库的正常写法是：

```
from dnndk import n2cube
import PowerSensor as ps
```

4.1 dpu 加速模块

这里主要介绍 dpu 调度程序，将编译好 elf 模型部署到 powersensor 上面运行。模型生成、编译相关的教程详情见 tutorial。

4.1.1 函数说明

其构造函数为：

```
dpu1 = ps.DpuHelper()
```

函数列表：

```
# 类名: DpuHelper
# ------------------------------------------
# 构造函数: DpuHelper(),
# - 无参数
# ------------------------------------------
# 模型加载函数: load\_kernel(elf\_name, input\_node\_name, output\_node\_name),
# - elf\_name, 神经网络相关的模型文件, 虚拟机生成的, 拓展名elf
# - input\_node\_name, 输入节点的名称, 在虚拟机编译模型时会打印
# - output\_node\_name, 输出节点的名称, 在虚拟机编译模型时会打印
# - 返回值, 无
# ------------------------------------------
# 神经网络推断函数: predit(img\_scale)
# - img\_scale, 神经网络输入, 应该与训练时保持一致
# - 返回值, 推断的结果
# ------------------------------------------
# 带softmax的神经网络推断函数: predit\_softmax(img\_scale)
# - img\_scale, 神经网络输入, 应该与训练时保持一致
# - 返回值, 推断的结果经过softmax运算后的结果
# ------------------------------------------
# 关闭dpu函数: close()
```

```
# - 无参数
# - 返回值，无
```

4.1.2 参考例程

这部份例程需要现场编译的 elf 文件，具体请查看 tutorial 文档。

第 五 章 其他模块

内容提要

❑ TcpUdp 上位机

这章节介绍其他的一些库，目前主要有采用多线程传输图像和参数的 tcpudp 上位机。引用这些库的正常写法是：

```
import TcpUdpHelper
```

5.1 TcpUdp 上位机

Tcp 是稳定可靠的传输，用于上位机（电脑）设置 Powersensor 的参数，udp 是实时性较好的传输，用于 Powersensor 向上位机参数。

上位机的下载地址（目前支持 windows 操作系统）：

```
链接：https://pan.baidu.com/s/1JxLUMGoxo4PvUets-89Nlw
提取码：1cfj
复制这段内容后打开百度网盘手机App，操作更方便哦
```

这里仅介绍函数的功能和参数的作用，具体教程请参考 tutorial。

5.1.1 函数说明

其构造函数为：

```
server = TcpUdpHelper.PowersensorServer(port=10775, para=x)
```

函数列表：

```
# 类名：PowersensorPara，
# ------------------------------------------
# 抽象类，必须被继承实现，主要用于数据包解析
# ------------------------------------------
# 属性：
# - img，把需要远程显示的图像写入到这个变量
# ------------------------------------------
# 包解析函数：unpackBuffer(data_buf)，
# 必须实现，把字节数组解析成用户需要的变量形式（整形/浮点）
# - 返回值，无
# ------------------------------------------
# 打包函数：packParas()
# 必须实现，把用户的变量打包成字节数组形式
# - 无参数
# - 返回值，字节数组包
```

```
# ================================================
# ================================================
# 类名: TcpUdpHelper
# ------------------------------------------------
# 构造函数: TcpUdpHelper(port, psPara_obj),
# - port, 用户指定的在powersensor上运行的tcp服务器的端口号
# - psPara_obj, 继承PowersensorPara类的用户参数类的对象
# ------------------------------------------------
# ps服务器启动函数: startServer(),
# - 无参数
# - 返回值, 继承PowersensorPara类的用户参数类的对象, 用户可以通过这个对象获取和设置实时设置的变量
```

5.1.2 参考例程

```
import numpy as np
import cv2
import time
import numpy as np
import struct
import socket
import TcpUdpHelper
import PowerSensor as ps

class PowersensorPara_ex(TcpUdpHelper.PowersensorPara):
  def __init__(self, debug=False):
    super(PowersensorPara_ex, self).__init__()
    self.paraSwitch = [False, False, False, False, False, False, False, False]
    self.paraInt = np.array(np.zeros(2), dtype=np.int32)
    self.paraFloat = np.array(np.zeros(8), dtype=np.float32)

  def unpackBuffer(self, data_buf):
    for i in range(8):
      # print(data_buf[4 + i] )
    if data_buf[4 + i] > 0:
      self.paraSwitch[i] = True
    else:
      self.paraSwitch[i] = False
      temp = struct.unpack('ii', data_buf[12:20])
      self.paraInt = np.array(temp, np.int32)
      temp = struct.unpack('ffffffff', data_buf[20:52])
      self.paraFloat = np.array(temp, np.float32)

  def packParas(self):
    buffer = struct.pack("iifffffffff", self.paraInt[0], self.paraInt[1], self
```

```
    .paraFloat[0], self.paraFloat[1],
    self.paraFloat[2], self.paraFloat[3], self.paraFloat[4], self.paraFloat
    [5],
    self.paraFloat[6], self.paraFloat[7])
    cmd_buf = bytearray(52)
    cmd_buf[0] = 0xfa
    cmd_buf[1] = 0xf5
    cmd_buf[2] = 0
    cmd_buf[3] = 0x02
    for i in range(8):
      if self.paraSwitch[i]:
      cmd_buf[i + 4] = 1
    else:
      cmd_buf[i + 4] = 0
    for i in range(len(buffer)):
      cmd_buf[i + 12] = buffer[i]
    return cmd_buf
      x = PowersensorPara_ex()
# 端口号必须和上位机定义的一致
server = TcpUdpHelper.PowersensorServer(port=10775, para=x)
x = server.startServer()
# 打印原始数据
print('开关型数据:' + str(x.paraSwitch))
print('整型数据:' + str(x.paraInt))
print('浮点型数据:' + str(x.paraFloat))

# 在上位机上修改数据后，点写入参数，然后打印查看参数的变化
print('开关型数据:' + str(x.paraSwitch))
print('整型数据:' + str(x.paraInt))
```