OPENMV 中文参考文档

所有资料来自 OPENMV 官网，我只做整理与翻译。

http://www.openmv.net.cn/

http://docs.openmv.io/

时间：2018-03-25　　Rev1.3　　作者：se7en　　QQ：770609835

如有错误以官网英文版为准。（更新了 Image.Methods）

特别注意！由于国内 OpenMV 生产不统一，管脚绑定可能存在差距，使用时需要咨询商家。

一、PYBoard 核心板快速指南

1、通用板级控制

开发板基本控制

```
import pyb
pyb.delay(50)                           #延时 50 毫秒
pyb.millis()                            #自从 bootup 开始的毫秒数
pyb.repl_uart(pyb.UART(3, 9600))        #同步 repl 与串口 3 配置
pyb.wfi()                               #休眠 CPU
pyb.stop()                              #停止 CPU
```

LED 控制

```
from pyb import LED
led = LED(1)                            #1 号为红色 LED
led.toggle()                            #LED 状态取反
led.on()                                #LED 亮
led.off()                               #LED 灭
```

这是 LED 引脚排列：

```
LED(1) -> 红色 LED
LED(2) -> 绿色 LED
LED(3) -> 蓝色 LED
LED(4) -> 红外 LEDs
```

引脚和 GPIO 控制

```
from pyb import Pin
p_out = Pin('P7', Pin.OUT_PP)           #设置 Pin7 为推挽输出模式
p_out.high()                            #输出高电平
p_out.low()                             #输出低电平
p_in = Pin('P7', Pin.IN, Pin.PULL_UP)   #设置 Pin7 为上拉输入模式
p_in.value()                            #读取当前管脚值，结果 0 或 1
```

这是 GPIO 引脚分配：

```
Pin(‘P0’) -> P0 (PB15) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
Pin(‘P1’) -> P1 (PB14) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
Pin(‘P2’) -> P2 (PB13) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
Pin(‘P3’) -> P3 (PB12) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
Pin(‘P4’) -> P4 (PB10) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
Pin(‘P5’) -> P5 (PB11) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
```

Pin(‘P6’) -> P6 (PA5) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
Pin(‘P7’) -> P7 (PD12) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
Pin(‘P8’) -> P8 (PD13) - 5V 耐压，3.3V 输出，高达 25 mA 驱动。
在 OpenMV M7 上：
Pin(‘P9’) -> P9 (PD14) - 5V 耐压，3.3V 输出，最高可达 25 mA 驱动。
所有 I/O 引脚上的电流之和不要超过 120mA。

舵机控制

```
from pyb import Servo
s1 = Servo(1)                          # 定义一号舵机(P7)
s1.angle(45)                           # 移动到 45 度
s1.angle(-60, 1500)                    # 在 1500ms 内移动到-60 度
s1.speed(50)                           # 以 50 的速度持续旋转
```

舵机引脚分配：
Servo(1) -> P7 (PD12)
Servo(2) -> P8 (PD13)
在 OpenMV M7 上:
Servo(3) -> P9 (PD14)

外部中断

```
from pyb import Pin, ExtInt
callback = lambda e: print("intr")     #外部中断函数
ext = ExtInt(Pin('P7'), ExtInt.IRQ_RISING, Pin.PULL_NONE, callback)
```

GPIO 引脚分配：
Pin(‘P0’) -> P0 (PB15)
Pin(‘P1’) -> P1 (PB14)
Pin(‘P2’) -> P2 (PB13)
Pin(‘P3’) -> P3 (PB12)
Pin(‘P4’) -> P4 (PB10)
Pin(‘P5’) -> P5 (PB11)
Pin(‘P6’) -> P6 (PA5)
Pin(‘P7’) -> P7 (PD12)
Pin(‘P8’) -> P8 (PD13)
在 OpenMV M7 上:
Pin(‘P9’) -> P9 (PD14)

计时器

```
from pyb import Timer
tim = Timer(4, freq=1000)                      #设置定时 4 器频率为 1000HZ
```

```
tim.counter()                                   #获取计数值
tim.freq(0.5)                                   #设置当前定时器频率为 0.5HZ
tim.callback(lambda t: pyb.LED(1).toggle())     #定时器中断函数
```

定时器引脚分配：

```
Timer 1 Channel 3 Negative -> P0
Timer 1 Channel 2 Negative -> P1
Timer 1 Channel 1 Negative -> P2
Timer 2 Channel 3 Positive -> P4
Timer 2 Channel 4 Positive -> P5
Timer 2 Channel 1 Positive -> P6
Timer 4 Channel 1 Negative -> P7
Timer 4 Channel 2 Negative -> P8
```

在 OpenMV M7 上：

```
Timer 4 Channel 3 Positive -> P9
```

PWM（脉宽调制）

```
from pyb import Pin, Timer
p = Pin('P7')                               #Pin7 是定时器 4 通道 1
tim = Timer(4, freq=1000)                   #配置定时器 4 频率
ch = tim.channel(1, Timer.PWM, pin=p)       #配置定时器通道 1
ch.pulse_width_percent(50)                  #配置通道占空比
```

这是定时器引脚分配：

```
Timer 1 Channel 3 Negative -> P0
Timer 1 Channel 2 Negative -> P1
Timer 1 Channel 1 Negative -> P2
Timer 2 Channel 3 Positive -> P4
Timer 2 Channel 4 Positive -> P5
Timer 2 Channel 1 Positive -> P6
Timer 4 Channel 1 Negative -> P7
Timer 4 Channel 2 Negative -> P8
```

在 OpenMV M7 上：

```
Timer 4 Channel 3 Positive -> P9
```

ADC（模数转换）

```
from pyb import Pin, ADC
adc = ADC('P6')                             #配置 ADC 输入引脚
adc.read()                                  #获取读进的数值 0~4095
```

这是 ADC 引脚分配：

ADC(‘P6’) -> P6 (PA5) - 只有 3.3V（非 5V）电压在这个模式!

DAC (数模转换)

```
from pyb import Pin, DAC
dac = DAC('P6')                          #设置 DAC 输出管脚
dac.write(120)                           #给 DAC 数字值 0~255
```

这是 ADC 引脚分配：
DAC(‘P6’) -> P6 (PA5) - 只有 3.3V（非 5V）电压在这个模式!

UART (串口)

```
from pyb import UART
uart = UART(3, 9600)                     #设置串口 3 波特率
uart.write('hello')                      #串口写数据
uart.read(5)                             #串口读 5 字节
```

这是 UART 引脚分配：
UART 3 RX -> P5 (PB11)
UART 3 TX -> P4 (PB10)
在 OpenMV M7 上：
UART 1 RX -> P0 (PB15)
UART 1 TX -> P1 (PB14)

SPI 总线

```
from pyb import SPI
spi = SPI(2, SPI.MASTER, baudrate=200000, polarity=1, phase=0)
spi.send('hello')                        #SPI 发送
spi.recv(5)                              #SPI 接收 5 字节
spi.send_recv('hello')
```

这是 SPI 引脚分配：
SPI 2 MOSI (Master-Out-Slave-In) -> P0 (PB15)
SPI 2 MISO (Master-In-Slave-Out) -> P1 (PB14)
SPI 2 SCLK (Serial Clock) -> P2 (PB13)
SPI 2 SS (Serial Select) -> P3 (PB12)

I2C 总线

```
from pyb import I2C
i2c = I2C(2, I2C.MASTER, baudrate=100000) #I2C 总线 2，主模式，波特率 100000
i2c.scan()                               #返回从机地址
i2c.send('hello', 0x42)                  #给地址 0x42 发送
```

```
i2c.recv(5, 0x42)                    #从 0x42 接收 5 字节
i2c.mem_read(2, 0x42, 0x10)          #从 0x42 内存地址 0x10 开始读 2 字节
i2c.mem_write('xy', 0x42, 0x10)      #从 0x42 内存地址 0x10 开始写 'xy'
```

这是 I2C 引脚排列：
I2C 2 SCL (Serial Clock) -> P4 (PB10)
I2C 2 SDA (Serial Data) -> P5 (PB11)
在 OpenMV M7 上：
I2C 4 SCL (Serial Clock) -> P7 (PD13)
I2C 4 SDA (Serial Data) -> P8 (PD12)

2、pyb 板级其他函数

PYB——和 pyboard 相关的函数,pyb 模块包含了和 pyboard 相关的函数。
时间函数
pyb.delay(ms)
延时毫秒。

pyb.udelay(us)
延时微秒。

pyb.millis()
返回启动后运行的时间（毫秒）。
返回值是 micropython smallint 类型 (31 位有符号整数), 因此在 2^30 毫秒后 (大约 12.4 天) 它将变为负数。注意如果调用 pyb.stop() 将停止硬件计数器，因此在 “休眠” 时将计数。它也会影响 pyb.elapsed_millis().pyb.micros()返回复位后的微秒。返回值是 micropython smallint 类型 (31 位有符号整数), 因此在 2^30 微秒后 (约 17.8 分钟) 将变为负数。

pyb.elapsed_millis(start)
返回从 start 时刻后到现在的时间（毫秒）。
这个函数考虑到计数器的回绕，因此返回值总是正数。因此它可以用于测量最高 12.4 天。
例子:

```
start = pyb.millis()
while pyb.elapsed_millis(start) < 1000:
   # Perform some operation
```

pyb.elapsed_micros(start)
返回从 start 时刻到现在的时间（微秒）。

这个函数考虑到计数器的回绕，因此返回值总是正数。因此它可以用于测量最高 17.8 分钟。
例子:

```
start = pyb.micros()
while pyb.elapsed_micros(start) < 1000:
   # Perform some operation
   Pass
```

pyb.hard_reset()
复位，和按下复位键的效果相同。

pyb.bootloader()
直接进入 bootloader 。

中断函数
pyb.disable_irq()
禁止中断。返回之前的中断允许状态。返回值可以在 enable_irq 函数中用于恢复中断允许状态。

pyb.enable_irq(state=True)
state 是 True 时（默认）允许中断，是 False 时禁止中断。常用于退出关键时区时恢复中断。

时钟，功率函数
pyb.freq([sysclk[, hclk[, pclk1[, pclk2]]]])
无参数时，返回当前时钟频率，包括: (sysclk, hclk, pclk1, pclk2)。它对应着:
sysclk: CPU 时钟频率
hclk: AHB、内存和 DMA 总线频率
pclk1: APB1 总线频率
pclk2: APB2 总线频率
如果指定参数，将设置 CPU 频率。频率单位是 Hz，如 freq(120000000) 设置 sysclk (CPU 频率) 到 120MHz。注意并非任何参数都能使用，支持的时钟频率有(MHz): 8, 16, 24, 30, 32, 36, 40, 42, 48, 54, 56, 60, 64, 72, 84, 96, 108, 120, 144, 168 等。最大的 hclk 是 168MHz, pclk1 是 42MHz, pclk2 是 84MHz。不要设置频率超过这个范围。
hclk, pclk1 和 pclk2 频率由系统时钟分频而来，hclk 支持的分频比是: 1, 2, 4, 8, 16, 64, 128, 256, 512。pclk1 和 pclk2 的分配比是: 1, 2, 4, 8。
sysclk 在 8MHz 时直接使用 HSE (外部振荡器) ，在 16MHz 时直接使用 HSI (内部振荡器)。高于这个频率时使用 HSE 驱动 PLL (锁相环)输出。

注意改变时钟频率时如果通过 USB 连接到计算机，将使 USB 变为不可用。因此最好在 boot.py 中改变时钟，这时 USB 外设还没有启用。此外当系统时钟低于 36MHz 时 USB 功能将不能使用。

pyb.wfi()
等待内部或外部中断。执行 wfi 指令以降低功耗，直到发生任何中断（内部或外部），然后继续运行。注意 system-tick 中断每毫秒（1000Hz）发生一次，因此它最多阻塞 1ms.

pyb.stop()
进入 “sleeping” 状态，可以降低功耗到 500 uA。从睡眠模式唤醒，需要外部中断或者实时时钟事件，唤醒后从睡眠的位置继续运行。
查看 rtc.wakeup() 配置实时时钟唤醒事件。

pyb.standby()
进入 “deep sleep” 状态， 功耗将低于 50 uA。从深度睡眠模式唤醒需要实时时钟事件，或 X1 引脚外部中断（PA0=WKUP），或者 X18（PC13=TAMP1）。唤醒后将执行复位。
查看 rtc.wakeup() 配置实时时钟唤醒事件。

其他函数
pyb.have_cdc()
如果 USB 连接并作为串口设备就返回 True，否则返回 False。
注意这个函数已废弃，以后请使用 pyb.USB_VCP().isconnected() 。

pyb.hid((buttons, x, y, z))
获取 4 参数元组（或列表）并发送到 USB 主机（PC），驱动 HID 鼠标。
注意这个函数已经废弃，请使用 pyb.USB_HID().send(…)。

pyb.info([dump_alloc_table])
打印开发板的信息。

pyb.main(filename)
设置在 boot.py 运行后启动的 main 脚本的文件名，如果没有调用这个函数，将执行默认文件 main.py。只有在 boot.py 中调用这个函数才有效。

pyb.mount(device, mountpoint, *, readonly=False, mkfs=False)
加载设备，并作为文件系统的一部分。设备必须提供下面的协议:

```
readblocks(self, blocknum, buf)
writeblocks(self, blocknum, buf) (optional)
count(self)
sync(self) (optional)
```

readblocks 和 writeblocks 需要在 buf 和设备之间复制数据。buf 是 512 倍数的 bytearray。如果没有定义 writeblocks，那么设备将加载为只读模式，两个函数的返回值被忽略。

count 返回 device 的块数量，sync, 同步数据到设备。

mountpoint 在文件系统的 root 下，必须以右斜杠开头。

如果 readonly 是 True，设备会加载为只读模式，否则加载为读写模式。

如果 mkfs 是 True，那么将创建新的文件系统。

卸载设备，以 None 作为设备名参数，挂载点作为 mountpoint。

pyb.repl_uart(uart)
获取或设置 REPL 的串口。

pyb.rng()
返回 30 位随机数，它由 RNG 硬件产生。（注如果没有 RNG 模块，这个函数将不可用）

pyb.sync()
同步所有文件系统。

pyb.unique_id()
返回 12 字节 (96 位) 的唯一 ID 号。

二、内置 Classes

class Accel - 加速度传感器
class ADC - 模拟到数字转换
class CAN - CAN 总线通信
class DAC - 数字到模拟转换
class ExtInt - 外部中断
class I2C - 两线通信协议
class LCD - pyskin LCD 控制 LCD
class LED - LED 对象
class Pin - 控制 I/O
class PinAF - Pin 替代功能
class RTC - 实时时钟
class Servo - 3 线伺服电机驱动
class SPI - 主 SPI 驱动
class Switch - 按键对象
class Timer - 内部定时器
class TimerChannel — 定时器通道控制
class UART - 串口通信

class USB_VCP – USB 虚拟串口

三、板级外设详细教程

ADC

ADC 的基本用法：

```
---------------------------------------------------------------------------
import pyb
adc = pyb.ADC(Pin('Y11'))           #将管脚 Y11 设置为 ADC 输入并创建对象
adc = pyb.ADC(pyb.Pin.board.Y11)    #将管脚 Y11 设置为 ADC 输入并创建对象
val = adc.read()                    #读取 ADC 转换值
adc = pyb.ADCAll(resolution)        #将 ADC 所有引脚设置为输入并创建对象
val = adc.read_channel(channel)     #读取 channel 所指通道的转换值
val = adc.read_core_temp()          #读取 MCU 温度
val = adc.read_core_vbat()          #读取 MCU 电源电压
val = adc.read_core_vref()          #读取 MCU 参考电压
---------------------------------------------------------------------------
pyb.ADC (pin)                       #通过 GPIO 定义一个 ADC
pyb.ADCAll(resolution)              #定义 ADC 的分辨率，可以设置为 8/10/12
adc.read()                          #读取 adc 的值，返回值与 adc 分辨率有关，8 位最
                                    #大 255，10 位最大 1023，12 位最大 4095
adc.read_channel(channel)           #读取指定 adc 通道的值
adc.read_core_temp()                #读取内部温度传感器
adc.read_core_vbat()                #读取 vbat 电压
vback = adc.read_core_vbat() * 1.21 / adc.read_core_vref()
adc.read_core_vref()                #读取 vref 电压（1.21V 参考）

3V3 = 3.3 * 1.21 / adc.read_core_vref()
adc.read_timed(buf, timer)          #以指定频率读取 adc 参数到 buf
    buf，缓冲区
    timer，频率（Hz）
注：使用这个函数会将 ADC 的结果限制到 8 位，这个函数是阻塞式的，会延时
len(buf)/timer
```

例子：

```
---------------------------------------------------------------------------
adc = pyb.ADC(pyb.Pin.board.X19)     # create an ADC on pin X19
buf = bytearray(100)                 # create a buffer of 100 bytes
adc.read_timed(buf, 10)              # read analog values into buf at 10Hz
```

```
                                    #  this will take 10 seconds to finish
for val in buf:                     # loop over all values
   print(val)                       # print the value out
```

DAC

DAC 基本用法：

```
from pyb import DAC
dac = DAC(1)                        #创建 dac 对象对应输出带 X5
dac.write(128)                      #给 DAC 写值 0~255
dac = DAC(1, bits=12)               #使用 12 位分辨率
```

输出正弦波：

```
import math
from pyb import DAC
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 math.sin(2 math.pi * i / len(buf)))
dac = DAC(1)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

输出 12 位精度正弦波:

```
import math
from array import array
from pyb import DAC
buf = array('H', 2048 + int(2047 math.sin(2 math.pi * i / 128)) for i in range(128))
dac = DAC(1, bits=12)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

使用 DAC 产生周期数据，写入 data

```
class pyb.DAC(port, bits=8)         #定义 DAC; port，1 或 2，对应 X5（PA4）/X6（PA5）;
                                    #bits，输出精度，可以是 8 或 12
dac.init(bits=8)                    #初始化 DAC
dac.noise(freq)                     #以指定频率，产生伪随机噪声信号
dac.triangle(freq)                  #以指定频率产生三角波
dac.write(value)                    #写入参数。在 8bits 时，参数范围[0-255]；在
                                    #12bits 时，参数范围[0-4095]
dac.write_timed(data, freq, *, mode=DAC.NORMAL)   #使用 DMA 方式周期写入数据
```

data，缓冲区数组

freq，默认使用 Timer(6)，用指定频率更新。也可以指定另外的定时器，有效的定时器是[2, 4, 5, 6, 7, 8]。

mode，DAC.NORMAL or DAC.CIRCULAR

GPIO

GPIO 的定义

所有的 GPIO 都在 pyb.Pin.board.Name 中预先定义了：

```
x1_pin = pyb.Pin.board.X1
g = pyb.Pin(pyb.Pin.board.X1, pyb.Pin.IN)
```

也可以这样使用：

```
g = pyb.Pin('X1', pyb.Pin.OUT_PP)
```

也可以自己定义 GPIO 名称：

```
MyMapperDict = { 'LeftMotorDir' : pyb.Pin.cpu.C12 }
pyb.Pin.dict(MyMapperDict)
g = pyb.Pin("LeftMotorDir", pyb.Pin.OUT_OD)
```

可以映射 GPIO：

```
pin = pyb.Pin("LeftMotorDir")
```

甚至可以通过函数进行映射：

```
def MyMapper(pin_name):
  if pin_name == "LeftMotorDir":
      return pyb.Pin.cpu.A0
pyb.Pin.mapper(MyMapper)
```

基本用法

1.定义 GPIO: pyb.Pin(id)

```
LED1=Pin(Pin.cpu.A13, Pin.OUT_PP)
sw = Pin("X17")
sw = Pin('X17', Pin.IN, Pin.PULL_UP)
sw = Pin(Pin(Pin.cpu.B3, Pin.IN, Pin.PULL_UP)
```

2.返回 GPIO 的第二功能列表:Pin.af_list()

```
Pin.af_list(pyb.Pin.board.X1)
Pin.af_list(LED)
```

3.获取/设置 debug 状态：Pin.debug(state)

```
Pin.debug(True)
```

4.获取/设置 GPIO 映射字典：Pin.dict(dict)

```
MyMapperDict = { 'LeftMotorDir' : pyb.Pin.cpu.C12 }
pyb.Pin.dict(MyMapperDict)
```

5.获取/设置 Pin 映射:Pin.mapper(func)

6.初始化: Pin.init(mode, pull=Pin.PULL_NONE, af=-1)

```
mode:
Pin.IN - 输入
```

Pin.OUT_PP - 推挽输出(push-pull)
Pin.OUT_OD - 开漏输出(open-drain)
Pin.AF_PP - 第二功能，推挽模式
Pin.AF_OD - 第二功能，开漏模式
Pin.ANALOG - 模拟功能
Pin.PULL_NONE - 无上拉下拉
Pin.PULL_UP - 上拉
Pin.PULL_DOWN - 下拉
af，当 mode 是 Pin.AF_PP 或 Pin.AF_OD 时，选择第二功能索引或名称

7.获取/设置 GPIO 逻辑电平

Pin.value(sw)
Pin.value(LED, 1)
Pin.value(LED, 0)
LED.value(1)
LED.value(0)

当前 GPIO 第二功能索引：pin.af()

当前 GPIO 关联基本地址：pin.gpio()

GPIO 的模式:pin.mode()

GPIO 的名称:pin.name()

GPIO 和预定义的名称:pin.names()

引脚序号:pin.pin()

端口序号:pin.port()

上拉状态:pin.pull()

例程

```
from pyb import Pin
p_out = Pin('X1', Pin.OUT_PP)
p_out.high()
p_out.low()
p_in = Pin('X2', Pin.IN, Pin.PULL_UP)
p_in.value() # get value, 0 or 1
LED = Pin(Pin.cpu.A14, Pin.OUT_PP)
LED.value(not LED.value())
```

---

pyborad Pin 对应关系

预定义的 GPIO 名称，仅对 pyboard 有效。

芯片 IO 预定义名称

PA0 X1	PA1 X2	PA2 X3	PA3 X4	PA4 X5	PA5 X6	PA6 X7
PA7 X8	PB6 X9	PB7 X10	PC4 X11	PC5 X12	Reset X13	GND X14
3.3V X15	VIN X16	X17 PB3	X18 PC13	X19 PC0	X20 PC1	X21 PC2
X22 PC3	X23 A3.3V	X24 AGND	Y1 PC6	Y2 PC7	Y3 PB8	Y4 PB9
Y5 PB12	Y6 PB13	Y7 PB14	Y8 PB15	Y9 PB10	Y10 PB11	Y11 PB0

Y12 PB1　　Y13 Reset　Y14 GND　　Y15 3.3V　　Y16 VIN　　SWP B3
LED_RED PA13　　LED_GREEN PA14　　LED_YELLOW PA15　　LED_BLUE PB4
MMA_INT PB2　　MMA_AVDD PB5　　SD_D0 PC8　　SD_D1 PC9　　SD_D2 PC10
SD_D3 PC11　　SD_CMD PD2　　SD_CK PC12　　SDP A8　　SD_SW PA8
USB_VBUS PA9　　USB_ID PA10　　USB_DM PA11　　USB_DP PA12

PYB 中未公开的 Pin 用法

在 pyb 中，定义 Pin 的方法是：

pyb.Pin(‘X1’)

或者

pyb.Pin(pyb.Pin.cpu.A0)。

第一种方法是官方 pyboard 中定义的，虽然简明，但是不直观，容易搞错具体代表的 GPIO。第二种方法直观，但是名称较长，也比较繁琐。其实在 pyb 中，有隐藏的简单用法（官方文档中没有写），如：

pyb.Pin(‘B0’)

pyb.Pin(‘PB0’) # PYBV1.0 中不支持这个用法

这个用法既直观又方便。只要是使用 STM32 的板子，带有 pyb 库的都可以使用，不像 pyb.Pin(‘X1’)只能用在官方的 pyboard 上。

IIC

I2C 的用法

先看看基本用法：

```
from pyb import I2C
i2c = I2C(1)                           # create on bus 1
i2c = I2C(1, I2C.MASTER)               # create and init as a master
i2c.init(I2C.MASTER, baudrate=20000)   # init as a master
i2c.init(I2C.SLAVE, addr=0x42)         # init as a slave with given address
i2c.deinit()                           # turn off the peripheral

i2c.init(I2C.MASTER)
i2c.send(‘123’, 0x42)                  # send 3 bytes to slave with address 0x42
i2c.send(b’456’, addr=0x42)            # keyword for address

i2c.is_ready(0x42)                     # check if slave 0x42 is ready
i2c.scan()                             # scan for slaves on the bus, returning
                                       # a list of valid addresses
i2c.mem_read(3, 0x42, 2)               # read 3 bytes from memory of slave 0x42,
                                       # starting at address 2 in the slave
i2c.mem_write(‘abc’, 0x42, 2, timeout=1000) # write ‘abc’ (3 bytes) to memory
                                                of slave 0x42
```

```
                                        # starting at address 2 in the slave,
                                                  timeout after 1 second
```

I2C 的用法：

```
class pyb.I2C(bus, …)                    #bus，I2C 总线的序号
i2c.deinit()                             #解除 I2C 定义
i2c.init(mode, *, addr=0x12, baudrate=400000, gencall=False)
#初始化 mode，只能是 I2C.MASTER 或 I2C.SLAVE；addr，7 位 I2C 地址；baudrate，时钟
#频率；gencall，通用调用模式
i2c.is_ready(addr)                       #检测 I2C 设备是否响应，只对主模式有效
i2c.mem_read(data, addr, memaddr, *, timeout=5000, addr_size=8)
#读取数据 data，整数或者缓存；addr，设备地址；memaddr，内存地址；timeout，读取等
#待超时时间；addr_size，memaddr 的大小，8 位或 16 位。
i2c.mem_write(data, addr, memaddr, *, timeout=5000, addr_size=8)
#写入数据，参数含义同上
i2c.recv(recv, addr=0x00, *, timeout=5000)
#从总线读取数据；recv，需要读取数据数量，或者缓冲区；addr，I2C 地址；timeout，超
#时时间
i2c.send(send, addr=0x00, *, timeout=5000)
#send，整数或者缓冲区；addr，I2C 地址；timeout，超时时间。
i2c.scan()                               #搜索 I2C 总线上设备。
```

LED

LED 用法
LED1 绿色
LED2 红色
LED4 蓝色
LED 是特殊的 GPIO，它的用法如下：

```
pyb.LED(id)
```

定义一个 LED 对象，id 是 LED 序号，1-4。在 OpenIOE pyboard 板上，引出 LED1 LED2 LED4 三个 LED RGB 指示灯。

```
led.on()#亮灯
led.off()#关灯
led.toggle()#翻转
led.intensity([value]) #LED 亮度，value 是亮度值，0-255，0 是关，255 最亮，仅 LED3
                       #和 LED4 支持
```

例程 1:

```
import pyb
pyb.LED(1).on()
myled = pyb.LED(1)
```

```
myled.on()
myled.off()
myled.toggle()
pyb.LED(3).intensity(10)   #设置 LED3 亮度
```

例程 2，LED 交替闪烁：

```
import pyb
led1 = pyb.LED(1)
led2 = pyb.LED(2)
led3 = pyb.LED(4)
while True:
    led1.off()
    led2.on()
    led3.on()
    pyb.delay(200)
    led1.on()
    led2.off()
    led3.on()
    pyb.delay(200)
    led1.on()
    led2.on()
    led3.off()
    pyb.delay(200)
    print("OpenIOE LED FLASH")
```

PWM

PWM 是 Timer 的一种工作模式，它需要使用到 Timer 和 Pin 两个库

```
from pyb import Pin, Timer
tm2=Timer(2, freq=100)
tm3=Timer(3, freq=200)
led3=tm2.channel(1, Timer.PWM, pin=Pin.cpu.A15)
led3.pulse_width_percent(10)
led4=tm3.channel(1, Timer.PWM, pin=Pin.cpu.B4, pulse_width_percent=50)
```

首先使用 Timer 设定定时器，然后指定 Timer 的通道，并设定 PWM 模式、关联的 Pin，最后设置输出脉冲宽度或者脉冲宽度百分比（占空比）。

例子（OpenIOE pyboard 结合 OPenIOE ESP8266 扩展板 LED01 闪烁）：

```
from pyb import Pin, Timer
```

```
tm3=Timer(3, freq=5)
led2=tm3.channel(1, Timer.PWM, pin=Pin.cpu.B4, pulse_width_percent=50)
```
PWM 更多函数见 Timer 小节。

RTC

RTC 的用法

pyb 中已经定义好了 RTC，可以直接使用。RTC 除了可以读取/设置时间，还支持中断，也可以做为通用定时器。

1、定义 RTC 对象

```
pyb.RTC
```

2、读取/设置 rtc

```
rtc.datetime([datetimetuple])
```

datetimetuple 格式：(year, month, day, weekday, hours, minutes, seconds, subseconds)

weekday 是 1-7 代表周一到周日

subseconds 从 255 到 0 倒计数

3、设置唤醒定时器

```
rtc.wakeup(timeout, callback=None)
```

timeout 是毫秒

4、获取 RTC 启动时间和复位源

```
rtc.info()
```

5、获取/设置校正

```
rtc.calibration(cal)
```

无参数时读取校正值，有参数时设置校正值

6 例子：

RTC 定时器 2S 翻转一次 LED1

```
rtc.wakeup(2000, lambda t:pyb.LED(1).toggle())
```

设置/读取 RTC 时间

```
rtc = pyb.RTC()
set date time
rtc.datetime((2014, 5, 1, 4, 13, 0, 0, 0))
get date time
print(rtc.datetime())
```

SPI

class SPI - 主 SPI 驱动

主 SPI 驱动，物理层上需要三根数据线: SCK, MOSI, MISO.

1 构造函数：

SPI.init(mode, baudrate=1000000, *, polarity=0, phase=0, bits=8,
firstbit=SPI.MSB, pins=(CLK, MOSI, MISO))

初始化 SPI 总线:

mode 必须是 SPI.MASTER.

baudrate 是 SCK 时钟频率.

polarity 可以是 0 或 1, 代表空闲时时钟电平.

phase 可以是 0 或 1, 代表采样数据时第一或第二时钟沿.

bits 是数据位, 只能是 8, 16 或 32.

firstbit 只能是 SPI.MSB.

pins 代表 SPI 总线使用的 GPIO 元组.

SPI.deinit()

关闭 SPI.

SPI.write(buf)

写入数据，然后实际写入数据数量。

SPI.read(nbytes, *, write=0x00)

读取数据到 nbytes 同时写入制定数据，返回读取数据的数量。

SPI.readinto(buf, *, write=0x00)

读取到缓冲区，同时写入制定数据，返回读取数据的数量

SPI.write_readinto(write_buf, read_buf)

将 write_buf 写入 SPI，同时读取到 read_buf。两个缓冲区的长度需要相同，返回实际写入数据的数量。

2 常量：

SPI.MASTER

初始化为主 SPI 模式

SPI.MSB

设置高位在前模式

UART

使用串口前必须引入 UART 库

from pyb import UART

串口使用方法

1.1 定义串口

class pyb.UART(bus, …)

bus: 1-6，或者‘XA’, ‘XB’, ‘YA’, ‘YB’.

1.2 初始化串口

uart.init(baudrate, bits=8, parity=None, stop=1, *, timeout=1000, flow=None,
timeout_char=0, read_buf_len=64)

baudrate：波特率

bits：数据位，7/8/9

parity：校验，None, 0 (even) or 1 (odd)
stop：停止位，1/2
flow：流控，可以是 None, UART.RTS, UART.CTS or UART.RTS | UART.CTS
timeout：读取一个字节超时时间（ms）
timeout_char：两个字节之间超时时间
read_buf_len：读缓存长度

1.3 关闭串口

uart.deinit()

1.4 返回缓冲区数据个数，大于 0 代表有数据

uart.any()

1.5 写入一个字节

uart.writechar(char)

1.6 读取最多 nbytes 个字节。

uart.read([nbytes])

如果数据位是 9bit，那么一个数据占用两个字节，并且 nbytes 必须是偶数。

1.7 读取所有数据

uart.readall()

1.8 读取一个字节

uart.readchar()

1.9 读指定参数的数据

uart.readinto(buf[, nbytes])

buf：数据缓冲区
nbytes：最大读取数量

1.10 读取一行

uart.readline()

1.11 写入缓冲区

uart.write(buf)

在 9bits 模式下，两个字节算一个数据。

1.12 往总线上发送停止状态，拉低总线 13bit 时间

uart.sendbreak()：

1.13 串口对应 GPIO

串口总线的物理管脚对应如下：

UART(4) is on XA: (TX, RX) = (X1, X2) = (PA0, PA1)
UART(1) is on XB: (TX, RX) = (X9, X10) = (PB6, PB7)
UART(6) is on YA: (TX, RX) = (Y1, Y2) = (PC6, PC7)
UART(3) is on YB: (TX, RX) = (Y9, Y10) = (PB10, PB11)
UART(2) is on: (TX, RX) = (X3, X4) = (PA2, PA3)

USB_VCP（USB 虚拟串口）

使用 USB_VCP（USB 虚拟串口）

micropython 上的 USB 兼做 VCP，可以通过函数去控制 VCP，和 PC 进行数据通信。发送的数据会在终端上直接显示出来。

1.1 创建虚拟串口对象

class pyb.USB_VCP

1.2 设置中断 python 运行键，默认是 3（Ctrl+C）

usb_vcp.setinterrupt(chr)

-1 是禁止中断功能，在需要发送原始字节时需要。

1.3 检测 USB 是否连接，如果 USB 连接到串口设备，返回 True

usb_vcp.isconnected()

1.4 如果缓冲区有数据等待接收，返回 True

usb_vcp.any()

1.5 这个函数什么也不做，它的目的是为了让 vcp 可以做为文件来使用。

usb_vcp.close()

1.6 最多读取 nbytes 字节。如果不指定 nbytes 参数，那么这个函数和 readall()功能相同。

usb_vcp.read([nbytes])

1.7 读取缓冲区全部数据

usb_vcp.readall()

1.8 读取串口数据并存放到 buf。如果指定 maxlen 参数，那么最多读取 maxlen 个字节

usb_vcp.readinto(buf[, maxlen])

1.9 读取整行数据

usb_vcp.readline()

1.10 读取所有数据并分行存储，返回字节对象列表

usb_vcp.readlines()

1.11 写入缓冲区数据，返回写入数据的个数

usb_vcp.write(buf)

1.12 读取指定数据大小

usb_vcp.recv(data, *, timeout=5000)

data，可以是读取数据个数，或者是缓冲区

timeout，等待接收超时时间

1.13 发送指定数据大小

usb_vcp.send(data, *, timeout=5000)

data，缓冲区或者整数

timeout，发送超时时间

参考例子：

```
vs = pyb.USB_VCP()
vs.send(‘123’)
vs.send(65)
```

vs.write(‘123’)
vs.readline()

macroSD 卡的使用

pyboard 可以插 TF 卡，只要将卡格式化为 FAT/FAT32 格式就可以使用。插卡后，可以做为 TF 读卡器使用，只是速度稍慢（大约 450k/s）。

如果不插卡，就会从内部 flash 启动，如果插卡启动，就会从 TF 卡启动，就像使用 U 盘启动系统那样。如果 TF 卡上有 boot.py 和 main.py，在启动时也会自动执行。

内部 flash 的路径是”/flash”，TF 卡的路径是”/sd”，区分大小写。

使用内部文件操作，需要 import os

os.chdir(path) 修改路径
os.getcwd() 获取当前路径
os.listdir(dir) 目录列表
os.mkdir(dir) 创建目录
os.remove(path) 删除文件
os.rmdir(dir) 删除目录
os.rename(old_path, new_path) 文件改名
os.stat(path) 文件/目录状态
os.sync() 同步文件
os.urandom(n) 返回 n 个硬件产生的随机数

其他问题

microPython 不能显示中文文件名和路径名。

文件操作后，不会立即更新到 TF 卡，需要从系统中安全移出磁盘后才会生效，如果不先移出磁盘，可能会丢失文件，甚至破坏 TF 卡上的文件系统。

pyboard 有些挑卡，试过创见 8G 和 OV 32G 的都可以，但是十铨 16G 的就只能偶尔识别出来一次。所以如果你的 TF 卡识别不出来也不要紧，可能换一个卡就好了。

EXTI(外部中断)

一共有 22 个中断行，其中 16 个来自 GPIO，另外 6 个来自内部中断。

中断行 0 到 15，可以映射到对应行的任意端口。中断行 0 可以映射到 Px0，x 可以是 A/B/C；中断行 1 可以映射到 Px1，x 可以是 A/B/C，依次类推。

使用外中断时，GPIO 自动配置为输入。

基本用法

定义中断

pyb.ExtInt(pin, mode, pull, callback)
pin，中断使用的 GPIO，可以是 pin 对象或者已经定义 GPIO 的名称
Mode：
ExtInt.IRQ_RISING 上升沿

ExtInt.IRQ_FALLING 下降沿
ExtInt.IRQ_RISING_FALLING 上升下降沿

Pull：
pyb.Pin.PULL_NONE 无
pyb.Pin.PULL_UP 上拉
pyb.Pin.PULL_DOWN 下拉

callback，回调函数
extint.disable()，禁止中断
extint.enable()，允许中断
extint.line()，返回中断映射的行号
extint.swint()，软件触发中断
ExtInt.regs()，中断寄存器值

例子，设置用户按键下降沿中断

```
from pyb import Pin, ExtInt
def callback(line):
    print(“line =”, line)
    extint = pyb.ExtInt(Pin(“X17”), pyb.ExtInt.IRQ_FALLING, pyb.Pin.PULL_UP,
    callback)
```

TIMER(定时器)

定时器的用法

```
    from pyb import Timer                 #使用定时器前需要导入 Timer 库！！！
```

1.1 多项同时定义方式：

```
    tm=Timer(1, freq=100)                 #tm 为定时器 1，频率 100HZ
    tm=Timer(4, freq=200, callback=f)     #tm 为定时器 4，频率 200HZ，回调函数为 f
```

2.1 定义定时器：

```
    tm=Timer(n)                           #定义 Timer，n=1-14，但是 3 用于内部程序，
                                          #5/6 用于伺服系统和 ADC
```

2.2 设置频率：

```
    tm.freq(100)                          #设置定时器频率为 100HZ
```

2.3 定义回调函数

```
    tm.callback(f)                        #设置定时器中断回调函数为 f
```

2.4 禁用回调函数

```
    tm.callback(None)                     #禁用回调函数
```

例子：

翻转 LED4：

```
from pyb import Timer
```

```
tim = Timer(1, freq=1)
tim.callback(lambda t: pyb.LED(4).toggle())
```

呼吸灯：

```
from pyb import Timer
i = 0
def f(t):
   global i
   i = (i+1)%255
   pyb.LED(4).intensity(i)
tm=Timer(4, freq=200, callback=f)
```

Timer 库说明

3.1 创建定时器对象，id 范围是[1..14]

class pyb.Timer(id, ...)

3.2 初始化。

timer.init(id, freq, prescaler, period)

freq，频率

prescaler，预分频，[0-0xffff]，定时器频率是系统时钟除以(prescaler + 1)。定时器 2-7 和 12-14 最高频率是 84MHz，定时器 1、8-11 是 168MHz

peroid，周期值（ARR）。定时器 1/3/4/6-15 是 [0-0xffff]，定时器 2 和 5 是 [0-0x3fffffff]。

mode，计数模式：

Timer.UP - 从 0 到 ARR (默认)

Timer.DOWN - 从 ARR 到 0.

Timer.CENTER - 从 0 到 ARR，然后到 0.

div，用于数值滤波器采样时钟，范围是 1/2/4。

callback，定义回调函数，和 Timer.callback()功能相同

deadtime，死区时间，通道切换时的停止时间（两个通道都不会工作）。范围是 [0..1008]，它有如下限制：

0-128 in steps of 1.

128-256 in steps of 2,

256-512 in steps of 8,

512-1008 in steps of 16

deadtime 的测量是用 source_freq 除以 div，它只对定时器 1-8 有效。

3.3 禁止定时器，禁用回调函数，禁用任何定时器通道

timer.deinit()

3.4 设置定时器回调函数

timer.callback(fun)

3.5 设置定时器通道

timer.channel(channel, mode, ...)

channel，定时器通道号

mode，模式：

Timer.PWM，PWM 模式（高电平方式）

Timer.PWM_INVERTED，PWM 模式（反相方式）

Timer.OC_TIMING，不驱动 GPIO

Timer.OC_ACTIVE，比较匹配，高电平输出

Timer.OC_INACTIVE，比较匹配，低电平输出

Timer.OC_TOGGLE，比较匹配，翻转输出

Timer.OC_FORCED_ACTIVE，强制高，忽略比较匹配

Timer.OC_FORCED_INACTIVE，强制低，忽略比较匹配

Timer.IC，输入扑捉模式

Timer.ENC_A，编码模式，仅在 CH1 改变时修改计数器

Timer.ENC_B，编码模式，仅在 CH2 改变时修改计数器

callback，每个通道的回调函数

pin，驱动 GPIO，可以是 None

在 Timer.PWM 模式下的参数

pulse_width，脉冲宽度

pulse_width_percent，百分比计算的占空比

在 Timer.OC 模式下的参数

compare，比较匹配寄存器初始值

polarity，极性

Timer.HIGH，输出高

Timer.LOW，输出低

在 Timer.IC 模式下的参数（捕捉模式只有在主通道有效）

Polarity:

Timer.RISING，上升沿捕捉

Timer.FALLING，下降沿捕捉

Timer.BOTH，上升下降沿同时捕捉

Timer.ENC 模式：

需要配置两个 Pin，使用 timer.counter() 方法读取编码值，只在 CH1 或 CH2 上工作（CH1N 和 CH2N 不工作），编码模式时忽略通道号。

3.6 设置或获取定时器计数值

timer.counter([value])

3.7 设置或获取定时器频率

timer.freq([value])

3.8 设置或获取定时器周期

timer.period([value])

3.9 设置或获取定时器预分频

timer.prescaler([value])

3.10 获取定时器源频率（无预分频）

timer.source_freq()

class TimerChannel — 设置定时器通道

定时器通道用于产生/捕捉信号

TimerChannel 对象需要用 Timer.channel() 方法创建。

4.1 设置回调函数，fun 是回调函数的参数，如果 fun 是 None 那么将禁用回调函数

```
timerchannel.callback(fun)
```

4.2 获取或设置通道的捕捉值。捕捉、比较、脉宽都是同一个功能的别名，当用于输入捕捉功能时叫捕捉。

```
timerchannel.capture([value])
```

4.3 获取或设置通道的比较值（捕捉、比较和脉宽都是相同功能的别称）。比较是使用输出比较模式时的逻辑名称。

```
timerchannel.compare([value])
```

4.4 获取或设置通道的脉宽值（捕捉、比较和脉宽都是相同功能的别称）。脉宽是使用 PWM 模式时的逻辑名称。

```
timerchannel.pulse_width([value])
```

在边沿对齐模式下，脉宽 period＋1 对应占空比 100%；在中心对齐模式，脉宽 period 对应 100% 占空比。

4.5 获取或设置脉宽百分比。这个参数的范围在 0 到 100 之间，它可以是整数，也可以是浮点数（提高精度）。

```
timerchannel.pulse_width_percent([value])
```

KEY(按键)

开发板按键的使用

在 pyboard 上，有一个用户按键。MicroPython 已经预先定义好了按键的类，按键可以这样使用：

定义按键

```
sw = pyb.Switch()
```

读取按键状态

```
sw()
```

定义按键回调函数

```
sw.callback(lambda:pyb.LED(1).toggle())
```

禁用按键回调函数

```
sw.callback(None)
```

例子：

更复杂的使用回调函数（按键后翻转 LED1）

```
def f():
    pyb.LED(1).toggle()
    sw.callback(f)
```

当然还可以直接当作 GPIO 使用：

```
import pyb from Pin
```

```
sw=Pin("X17", Pin.IN, Pin.PULL_UP)
sw()
```

FILE(文件操作)

micropython 中的文件操作和 C 语言中类似。
写文件

```
f = open(“1:/hello.txt”, “w”)
f.write(“Hello World from Micro Python”)
f.close()
```

读取文件

```
f = open(“main.py”, “r”)
f.readall()
```

如何脱离电脑运行

当 OpenMV 连接电脑后，会出现一个存储盘。
把此存储盘下的 main.py 的内容修改为你自己要运行的代码。
重新上电，就会看到 LED 灯开始闪烁。这时你的 OpenMV 就已经脱离电脑运行了。

四、恢复出厂设置

因为某些原因造成 pyboard 故障,可以恢复到出厂设置,就像 Windows 系统重新用 ghost 恢复一样。

1. 连接 USB 线
2. 按住 USER 键，然后按下复位键
3. 松开复位键，保持 USER 键
4. 这时 LED 将循环显示：绿－》黄－》绿+黄－》灭
5. 等黄绿灯同时亮时松开 USER 键，这时黄绿灯会同时快速闪 4 次
6. 然后红灯亮起（这时红绿黄灯同时亮）
7. 红灯灭，pyboard 开始进行恢复到出厂状态
8. 所有灯都灭，恢复出厂设置完成。

五、机器视觉——图像处理函数

1、基本图像处理函数

image.rgb_to_lab(rgb_tuple)
返回 RGB 元组的 LAB 元组
#red、green、blue 色彩模式使用 rgb 模型为图像中每一个像素的 rgb 分量分配一个 0~255 范围内的强度值。Lab 色彩模型是由照度 L 和有关色彩的 a, b 三个要素组成。L 表示照度相当于亮度 0~100，a 表示从红色至绿色的范围-128~127，b 表示从蓝色至黄色的范围-128~127。

image.lab_to_rgb(lab_tuple)
返回 LAB 元组的 RGB 元组

image.rgb_to_grayscale(rgb_tuple)
返回 RGB 元组的灰度值 0~255

image.grayscale_to_rgb(g_value)
返回灰度值所对应的 RGB 元组

image.load_decriptor(path)
从磁盘加载描述对象

image.save_descriptor(path, descriptor)
将描述对象保存到磁盘

image.match_descriptor(descritor0, descriptor1, threshold=70, filter_outliers=False)
对于 LBP 描述符，这个函数返回一个整数，表示两个描述符之间的差异。然后，您可以根据需要对这个距离度量阈值进行比较。距离是相似性的度量。它是越接近零越好 LBP 关键点匹配。

2、百分比函数（Percentile Object）

percentile.value()
返回灰度百分比值（介于 0 和 255 之间）。

percentile.l_value()
返回 RGB565LAB L 通道百分位值（0 至 100）。

percentile.a_value()

返回 RGB565LAB A 通道百分位值（-128 至 127）。

percentile.b_value()
返回 RGB565LAB B 通道百分位值（-128 至 127）。

3、统计对象函数（Statistics Object）

statistics.mean() #返回的灰度均值（0-255）（int）
statistics.median() #返回灰度中值（0-255）（int）
statistics.mode() #返回灰度模式（0-255）（int）
statistics.stdev() #返回灰度标准差（0-255）（int）
statistics.min() #返回灰度最小值（0-255）（int）
statistics.max() #返回灰度最大值（0-255）（int）
statistics.lq() #返回灰度下四分位数（0-255）（int）
statistics.uq() #返回灰度下四分位数（0-255）（int）。
#下面依次对应，为 RGB565LAB 的各通道值
Statistics.l_mean()
statistics.l_median()
statistics.l_mode()
statistics.l_stdev()
statistics.l_min()
statistics.l_max()
statistics.l_lq()
statistics.l_uq()
statistics.a_mean()
statistics.a_median()
statistics.a_mode()
statistics.a_stdev()
statistics.a_min()
statistics.a_max()
statistics.a_lq()
statistics.a_uq()
statistics.b_mean()
statistics.b_median()
statistics.b_mode()
statistics.b_stdev()
statistics.b_min()
statistics.b_max()
statistics.b_lq()
statistics.b_uq()

4、块，色点，像素区函数（blob object）

blob.rect()

返回像“image.draw_rectangle”等函数画出的长方形的元组（x，y，w，h）
x，y 是长方形起点坐标，w 是横向宽度，h 是纵向高度。

blob.x() #返回块的 x 坐标（int）
blob.y() #返回块的 y 坐标（int）
blob.w() #返回块的宽度 w（int）
blob.h() #返回块的高度 h（int）
blob.pixels() #返回块的像素数
blob.cx() #返回块的质心 x 位置（int）
blob.cy() #返回块的质心 y 位置（int）

blob.code()

返回一个 16 位二进制数，其中包含一个位设置为每个颜色阈值，这是这个数据块的一部

blob.count()

返回合并到此块中的斑点数量。这是 1 除非调用“image.find_blobs”函数时令 merge=True。

blob.rotation()

返回以弧度表示的 BLOB 的旋转（浮动）。如果斑点像一支铅笔或钢笔这个值将为 0-180 度独特。如果这个圆是圆的，这个值就不起作用了。

blob.area()

返回包围盒周围的面积。（W * H）

blob.density()

返回块的密度比。这是在包围区域中的像素数。

5、直线函数（Line object）

line.line()

返回像“image.draw_line”等函数画出直线的元组（x1,y1,x2,y2）
直线从（x1，y1）到（x2，y2）

line.x1() #返回线的起点（point1）x 分量
line.y1() #返回线的起点（point1）y 分量
line.x2() #返回线的终点（point2）x 分量
line.y2() #返回线的终点（point2）y 分量
line.length() #返回直线的长度：sqrt(((x2-x1)^2) + ((y2-y1)^2)
line.magnitude() #从霍夫变换返回线的大小。
line.theta() #从霍夫变换（0 - 179）度返回直线的角度。
line.rho() #从霍夫变换返回该行的 ρ 值。

6、圆函数（Circle object）

circle.x() #返回圆的 x 坐标。
circle.y() #返回圆的 y 坐标。
circle.r() #返回圆的半径 r。
circle.magnitude() #返回圆的大小。

7、长方形函数（Rectangle Object）

rect.corners()
返回对象的 4 个角中 4 个（x，y）元组的列表。角总是按左上角从顺时钟顺序返回。rect.rect() 返回像“mage.draw_rectangle”等函数画出的长方形的元组（x，y，w，h）。
rect.x() #返回长方形的左上角 x 坐标。
rect.y() #返回长方形的左上角 y 坐标。
rect.w() #返回长方形的宽度。
rect.h() #返回长方形的高度。
rect.magnitude() #返回长方形的大小。

8、二维码函数（QRCode object）

9、数据矩阵函数（DataMatrix object）

10、条形码函数（BarCode object）

11、图像函数（Image object）

Class image.Image(path, copy_to_fb=False)
从 path 所指路径文件创建一个图像目标。
如 copy_to_fb=True 图像直接加载到帧缓冲区，允许你加载大量的图片。如果为 false，图像加载到 micropython 堆，比帧缓冲区小得多。

image.copy(roi=Auto)
创建一个图像目标的副本。
有参数 roi 是指定矩形区域（X，Y，W，H）进行复制。如果未指定，它等于复制整个图像的图像矩形。此参数不适用于 JPEG 图像。

image.save(path, roi=Auto, quality=50)
将图像的副本保存在 path 所指路径下。
参数 roi 是指定矩形区域（X，Y，W，H）进行复制。如果未指定，它等于复制整个图像的图像矩形。此参数不适用于 JPEG 图像。
参数 quality 是 JPEG 压缩质量，用于图像未压缩时保存为 JPEG 格式的图像。

image.compress(quality=50)
把图像进行 JPEG 压缩，压缩率 quality（0-100）。

image.compress_for_ide(quality=50)
为 IDE 把图像进行 JPEG 压缩。
用这种 JPEG 方法压缩然后传输到 OpenMV IDE 的编码每六位作为一个字节值显示 JPEG 数据。这样做是为了防止 JPEG 数据被误解为字节流中的其他文本数据。你需要使用这个方法来显示终端窗口通过“打开终端”OpenMV IDE 格式图像数据。

image.compressed(quality=50)
JPEG 压缩图像解压缩——原始图像未被解压缩。然而，这种方法需要占用大量的堆空间，因此图像压缩质量必须低，图像分辨率必须低。

image.compressed_for_ide(quality=50)
为 IDE 将 JPEG 压缩图像解压缩。

image.width() #以像素形式返回图像宽度。
image.height() #以像素形式返回图像高度。
image.format()
灰度图像返回 sensor.GRAYSCALE，RGB 图像返回 sensor.RGB565，JPEG 图像返回 ensor.JPEG。

image.size()
返回图像的大小（单位字节）。

image.clear()
清空灰度或者 RGB565 图像，不能将此函数使用在 JPEG 图像。

image.get_pixel(x, y)
灰度图像返回像素点（x，y）处的灰度像素值，RGB 图像返回像素点（x，y）处的 rgb888 像素组元。不支持压缩图像。

image.set_pixel(x, y, pixel)
设置像素值，也不支持压缩图像。

image.draw_line(line_tuple, color=White)

在图像上画一条以直线组元 line_tuple（x0，y0，x1，y1）为参数的 color 色直线。其中直线从点（x0，y0）开始，结束于（x1，y1）；当为灰度图像时 color 值 0-255，当为 RGB 图像时，color 为 RGB888 组元。默认为白色。不支持压缩图像。

image.draw_rectangle(rect_tuple, color=White)
画长方形，参数类似画直线。

image.draw_circle(x, y, radius, color=White)
画圆，参数类似。

image.draw_string(x, y, text, color=White)
画字符串，参数类似。

image.draw_cross(x, y, size=5, color=White)
画十字，参数类似。

image.draw_keypoints(keypoints, size=Auto, color=White)
画 keypoints object，参数类似。

image.binary(thresholds, invert=False)
图像二值化。
对于灰度图像，阈值（thresholds）是一组（lower，upper）像素灰度阈值，用来分割图像。分割将阈值内的所有像素转换为 1（白色），之外的所有像素为 0（黑色）。
对于 RGB 图像，阈值（thresholds）是一组（l_lo, l_hi, a_lo, a_hi, b_lo, b_hi）像素 LAB 阈值，用来分割图像。分割将阈值内的所有像素转换为 1（白色），之外的所有像素为 0（黑色）。被交换的 LO / HI 阈值是自动处理的。
Inverts=False 时不反置结果，否则反置。不支持压缩图像。

image.invert()
反置二值化的图像。像素点 0 变 1，1 变 0。不支持压缩图像。

image.b_and(image)
图像逻辑与。
参数 image 可以是一个图像对象或一个未压缩的图像文件的路径（BMP / PGM / ppm）。
这两个图像必须是相同的大小和相同的类型（灰度/ RGB）。不支持压缩图像。

image.b_or(image) #图像逻辑或，类似图像逻辑与。
image.b_nand(image) #图像逻辑与非，类似图像逻辑与。
image.b_nor(image) #图像逻辑非，类似图像逻辑与。
image.b_xor(image) #图像逻辑异或，类似图像逻辑与。
image.b_xnor(image) #图像逻辑同或，类似图像逻辑与。

image.erode(size, threshold=Auto)
从分段区域的边缘移除像素。
该方法通过卷积核（（size×2）+ 1）*（（size×2）+ 1）计算像素的图像和零内核的中心像素，如果邻居像素集的总和不大于 thanthreshold。此方法被设计用于二进制图像。

image.dilate(size, threshold=Auto)
将像素添加到分段区域的边缘。与上函数类似，被设计用于二进制图像。

image.negate()
将每个色彩通道的值数字转置。

image.difference(image)
从图像中减去另一个图像。对于每个颜色通道，每个像素都用求 ABS 代替。

image.replace(image)
用 image 替换当前图像（这是比混合快得多）。

image.blend(image, alpha=128)
用 image 混合当前图像。
参数 alpha 控制透明度。256 用于不透明覆盖。0 表示完透明。这两个图像必须是相同的大小和相同的类型（灰度/ RGB）。

image.morph(size, kernel, mul=Auto, add=0)
通过滤波核卷积图像。

image.midpoint(size, bias=0.5)
在图像上运行中点筛选器。

image.mean(size)
标准平均模糊滤波器（比使用变形更快）。

image.median(size, percentile=0.5)
在图像上运行中值滤波器。中值滤波是平滑曲面、保留边缘的最佳滤波器，但速度较慢。
image.mode(size)
在图像上运行模式过滤器，用相邻的模式替换每个像素。这种方法在灰度图像上效果很好。然而，在 RGB 图像上，由于操作的非线性性质，它在边缘上创造了大量的噪点。

image.gaussian(size)
用高斯内核平滑的图像。 Size 可以是 3 或 5 个 3x3 或 5x5 的内核。

image.chrominvar()
从图像中移除光照效果，只适用于 RGB565 图像。

image.histeq()
在图像上运行直方图均衡化算法。直方图均衡化的图像的对比度和亮度。

image.lens_corr(strength=1.8, zoom=1.0)
对镜头造成的鱼眼镜头进行镜头校正。

image.get_similarity（Image）
返回一个 similarity 对象，描述两个图像之间的相似性，使用 SSIM 算法比较 8*8 像素块。
Image 可以使一个图像，也可以是一个图像文件夹（bmp/pgm/ppm），两个图像必须相同尺寸，相同类型。不支持压缩图像。

image.get_statistics(roi=Auto, bins=Auto, l_bins=Auto, a_bins=Auto, b_bins=Auto)
计算平均值，中位数，模式，标准偏差，min，max，下分位数，和上部的四分位数为 ROI 所有颜色通道并返回一个 统计 对象。请看“统计对象”的更多信息。你也可以通过下面的函数调用此方法，get_stats 或 image.statistics。

image.get_linear_regression(thresholds[, roi=Auto, x_stride=2, y_stride=1, invert=False, robust=False])
计算线性回归的所有阈值的图像中的像素。

image.find_blobs(thresholds[, roi=Auto, x_stride=2, y_stride=1, invert=False, area_threshold=10, pixels_threshold=10, merge=False, margin=0, threshold_cb=None, merge_cb=None])
查找图像中所有通过阈值测试的像素（返回的像素区域），并返回一个描述每一个团块的对象的列表。请参阅 BLOB 对象更多的信息。

阈值必须是一个元组列表[（LO，HI），（LO，HI），…，（LO，HI）]定义要跟踪的颜色范围。你可以通过高达 16 的阈值在一 image.find_blobs 叫元组。对于灰度图像，每个元组需要包含两个值——一个最小灰度值和一个最大灰度值。只有像素区域，介于这些阈值将被考虑。每个元组为 RGB565 图像需要六个值（l_lo，l_hi，a_lo，a_hi，b_lo，b_hi）-这是实验室的 L，最小值和最大值，和 B 通道分别。为方便使用这个函数会自动修复用最小和最大值。此外，如果一个元组是大于六的值被忽略的休息。相反，如果元组太短的阈值，其余的都是假定为零。

要获取要跟踪的对象的阈值，只需选择要在 IDE 帧缓冲区中跟踪的对象（单击并拖动）。直方图将更新只是在那个地区。然后写下颜色分布在每个直方图通道中的起始位置和下降位置。这将是你的低值和高值为阈值。最好是手动确定阈值，而不是使用上、下四分位数的统计数字，因为它们太紧了。

最新版本的 OpenMV IDE 功能阈值编辑器帮助采摘阈值容易。它允许你用滑块控制阈值，这样你就可以看到阈值是如何分割的。

roi 是感兴趣区域的矩形元组（X，Y，W，H）。如果没有指定，它等于图像的矩形。在 ROI 中只有像素操作。

x_stride 是搜索时跳过的 X 像素数。一旦找到一个 blob，线填充算法将像素精确。当斑点很大时，增加 x_stride 加快斑点搜索。

y_stride 是搜索时跳过的像素数。Y 一旦找到一个 blob，线填充算法将像素精确。当斑点很大时，增加 y_stride 加快斑点搜索。

invert 反转阈值操作，而不是匹配像素内已知的颜色边界像素进行匹配，这是已知的颜色范围之外。

如果一个块的包围盒面积小于 area_threshold 被过滤掉。

如果一个块的像素数小于 pixel_threshold 被过滤掉。

merge 如果 TRUE 合并所有未过滤出的斑点谁的边界矩形相交对方。在交叉测试期间，可以使用边缘来增加或减少用于像素的边界矩形的大小。例如，有 1 个边距的边界矩形是 1 个像素彼此距离将被合并。

合并斑点允许你实现色标跟踪。每一个对象都有一个代码值，它是由每个颜色阈值的 1s 组成的位向量。例如，如果你通过 image.find_blobs 两颜色阈值然后第一阈值有 1 和第二 2 码（第三的门槛将 4 和第四会 8 等）。将逻辑或所有代码合并在一起，这样您就可以知道它们是由什么颜色生成的。这允许你跟踪两种颜色，如果你得到一个有两种颜色的物体，那么你知道它可能是一种颜色代码。

如果您使用的是严格的颜色边界，而不能完全跟踪您试图跟踪的对象的所有像素，那么您也可能希望合并斑点。

最后，如果你想合并的斑点，但不想让两个颜色阈值进行合并，然后调用两次 image.find_blobs 与单独的阈值，使斑点不合并。

threshold_cb 可能设置的函数调用每个 blob 的阈值滤波后从斑点的列表合并。回调函数将收到一个参数——要过滤的对象。然后，回调必须返回 true，以保持团块和 false 过滤它。

merge_cb 可能设置的函数调用上每两块约被阻止或允许合并。回调函数将收到两个参数——两个要合并的对象。回调函数必须返回 true 以合并斑点或 false，以防止合并斑点。

不支持压缩图像。

image.find_lines(roi=Auto, x_stride=2, y_stride=1, threshold=1000, theta_margin=25, rho_margin=25)

使用霍夫变换查找图像中的所有直线。返回一个直 线对象列表（见上文）。

image.find_line_segments(roi=Auto, x_stride=2, y_stride=1, threshold=1000, theta_margin=25,rho_margin=25, segment_threshold=100)

使用霍夫变换查找图像中的线段。返回一个线对象列表（见上文）。

image.find_circles([roi=Auto, x_stride=2, y_stride=1, threshold=1600, x_margin=10, y_margin=10,r_margin=10])

使用霍夫变换在图像中找到圆。返回一个圈对象列表（见上文）。

image.find_qrcodes（toi=AUto）

找二维码，返回 qrcode 对象列表。

image.find_apriltags(roi=Auto,families=image.TAG36H11, fx=Auto, fy=Auto, cx=Auto, cy=Auto)
找到所有的 AprilTags，返回一个对象列表。（AprilTags 是类似二维码的二维图像）
Roi 是感兴趣区域参数，为一个四元组 tupl（x，y，w，h）。

image.find_barcodes(roi=Auto)
找到所有一维的条形码。

image.find_rects([roi=Auto, threshold=10000])
在图像中用 apriltags 四检测算法找到矩形。

Image.midpoint_pooled(x_div, y_div, bias=0.5)
找到 x_div*y_div 正方形的中点，返回由这些中点组成的新的正方形图像。

image.find_keypoints(roi=Auto, threshold=20, normalized=False, scale_factor=1.5, max_keypoints=100,corner_detector=CORNER_AGAST)
从感兴趣区域提取特征点（x，y，w，h）元组。然后你可以再使用 image.match_descriptor 函数来比较两套要点得到匹配的区域。如果没有找到关键点返回 None。
threshold 是关键点个数。当 corner_detector 参数为 AGAST 时，这个值在 20 左右；参数为 FAST 时这个值在 60-80。参数越小的得到的跟踪点越多。
normalized 是一个 bool 值，如果为真，关闭多分辨率关键点跟踪。
scale_factor 是一个浮点数，必须大于 1.0，高的比例因子会让扫描更快但是匹配不好。一个较好的实验值在 1.35 到 1.5 之间。
max_keypoints 最大关键点。关键点太多内存占用多。
corner_detector 角检测算法，用来关键点跟踪，有两个值 image.FAST 和 image.AGAST。FAST 算法跟快但更不准确。
只适用于灰度图像。

image.find_edges(edge_type, threshold=[100,200])
仅用于灰度图像。对图像进行边缘检测，只有边缘的图像被替换成边。 edge_type 可以是：
image.edge_simple 简单阈值高通滤波算法。
image.edge_canny - Canny 边缘检测算法。
threshold 是一个包含低阈值和高阈值的二值元组。您可以通过调整这些值来控制边缘的质量。

image.find_template(template,threshold, roi=Auto,step=2,search=image.SEARCH_EX)
模版匹配。返回第一个匹配位置 tuple（x，y，w，h）。
template 是模版，所有图像必须为灰度图像。
threshold 是相似程度阈值，相似度高于这个值返回 tuple，否则不返回值。
roi 是感兴趣区域，也就是只在 roi 区域（x，y，w，h）内进行此函数操作。
step 是查找模版每次跳过的像素个数。

search 可以是 image.SEARCH_DS 或者 image.SEARCH_EX。image.SEARCH_DS 更快，但如果图像靠近图像边缘，则可能找不到模板。image.SEARCH_EX 会详尽的查找所以相对更慢。

image.find_features(cascade, roi=Auto, threshold=0.5, scale=1.5)
查找图像中所有匹配 HaarCascade 特征并返回列表。

image.find_eye(roi)
查找瞳孔。
使用此函数，首先要使用 image.find_features 函数特征参数为 frontalface 找到脸，然后用 image.find_features 特征参数为 eye 找到眼睛，最后调用此函数返回瞳孔坐标。只支持灰度图像。

六、相机传感器

1、基本函数

sensor.reset()
初始化相机传感器。

sensor.flush()
将帧缓冲区中的任何内容复制到 IDE。如果不是一个无限循环的运行脚本，你应该调用这个函数来显示你 OpenMV Cam 最后拍摄的图像。

sensor.snapshot(line_filter=None)
用相机拍照并返回图像目标。
参数 line_filter 可能是一个 Python 函数回调函数，用于处理从摄像机中传入的每一行像素。
Note：OpenMV Cam M4 不要使用这个参数，应为它计算不够快。

sensor.skip_frames([n, time])
以 n 次快拍让相机更改设置后图像稳定。 n 作为正常传递的参数，例如 skip_frames（10） 跳过 10 帧。你应该在改变相机设置后调用这个函数。
或者，你可以通过关键字参数 time 跳过帧几毫秒数，例如 skip_frames（time = 2000） 跳过帧 2000 毫秒。

sensor.width()
返回传感器分辨率宽度。

sensor.height()
返回传感器分辨率高度。

sensor.get_fb()
（获取帧缓冲）返回由调用 sensor.snapshot()函数所返回的图像，如果没有调用 sensor.snapshot()则返回 None。

sensor.get_id()
返回相机模块 ID。
sensor.OV2640: 旧传感器模块
sensor.OV7725: 新传感器模块

sensor.set_pixformat(pixformat)
设置像素点格式。
sensor.GRAYSCALE: 每像素点 8 位
sensor.RGB565:每像素点 16 位

sensor.set_framerate(rate)
设置相机模块的帧速率。

sensor.set_framesize(framesize)
设置相机模块的帧大小。
framesize 值见下文常量列表。

sensor.set_windowing(roi)
将摄像机的分辨率设置为当前分辨率内的子分辨率。

sensor.set_quality(quality)
设置摄像机图像 jpeg 压缩质量。0 - 100。只针对 OV2640 相机。

sensor.set_colorbar(enable)
打开颜色条模式（true）或关闭（false）。默认关闭。

sensor.set_auto_gain(enable, value=-1)
enable 参数打开（True）/关闭（False）相机自动增益，默认打开。value 强制给定增益值。

sensor.set_auto_exposure(enable, value=-1)
enable 参数打开（True）/关闭（False）相机自动曝光，默认打开。value 强制给定曝光值。

sensor.set_auto_whitebal(enable, value=(-1, -1, -1))
enable 参数打开（True）/关闭（False）相机自动白平衡，默认打开。value 强制给定白平衡值。

sensor.set_hmirror(enable)
打开（True）或关闭（False）镜面模式，默认关闭。

sensor.set_vflip(enable)
打开（True）或关闭（False）垂直翻转模式，默认关闭。

sensor.set_special_effect(effect)
设置相机特效。
sensor.NORMAL: 正常图片
sensor.NEGATIVE: 负片

sensor.set_lens_correction(enable, radi, coef)
打开（True）或关闭（False）图像校正，radi 为像素校正半径（int），coef 为校正强度。

sensor.set_vsync_output(pin_object)
在管脚 pin_object 产生 vsync 垂直同步信号。

sensor.__write_reg(address, value)
向相机寄存器 address 开始写 value。

sensor.__read_reg(address)
读相机寄存器 address 开始的值。

2、常量

sensor.GRAYSCALE	#灰度像素格式（YUV422）。每个像素是 8 位字节。
sensor.RGB565	#RGB565 格式。每个像素为 16 位，2 字节。五位用于红、六位 #用于绿色，和五位用于蓝色。
sensor.JPEG	#JPEG 模式。只针对 OV2640 摄像头。
sensor.OV9650:	#使用函数 sensor.get_id() 返回此常亮，若相机是当前相机
sensor.OV2640:	#同上
sensor.OV7725:	#同上
sensor.QQQQCIF	#22x18 的分辨率
sensor.QQQCIF	#44x36 的分辨率
sensor.QQCIF	#88x72 的分辨率
sensor.QCIF	#176x144 的分辨率
sensor.CIF	#352x288 的分辨率
sensor.QQQQSIF	#22x15 的分辨率

sensor.QQQSIF #44x30 的分辨率
sensor.QQSIF #88x60 的分辨率
sensor.QSIF #176x120 的分辨率
sensor.SIF #352x240 的分辨率

sensor.QQQQVGA #40x30 的分辨率
sensor.QQQVGA #80x60 的分辨率
sensor.QQVGA #160x120 的分辨率
sensor.QVGA #320x240 的分辨率
sensor.VGA #640x480 的分辨率，只针对 OpenMV Cam M7 的 OV2640

sensor.HQQQQVGA #30x20 的分辨率
sensor.HQQQVGA #60x40 的分辨率
sensor.HQQVGA #120x80 的分辨率
sensor.HQVGA #240x160 的分辨率
sensor.HVGA #480x320 的分辨率，只针对 OpenMV Cam M7 的 OV2640

sensor.LCD #128x160 的分辨率，用于液晶屏
sensor.QQVGA2 #128x160 的分辨率，用于液晶屏

sensor.B40x30 #40x30 的分辨率，用于 image.find_displacement
sensor.B64x32 #64x32 的分辨率，用于 image.find_displacement
sensor.B64x64 #64x64 的分辨率，用于 image.find_displacement
sensor.SVGA #800x600 的分辨率，只针对 OV2640
sensor.SXGA #1280x1024 的分辨率，只针对 OV2640
sensor.UXGA #1600x1200 的分辨率，只针对 OV2640
sensor.NORMAL #将相机特效设为正常
sensor.NEGATIVE #将相机特效设为负片