

MicroPython中 文教程 V2.0

shaoziyang

本教程由MicroPython中文社区制作，资料来源于MicroPython官方文档、国内社区活动、网友经验分享等。

在保留本说明和版权的情况下，欢迎转载和分享

作者：[shaoziyang](#)

时间：2016年11月

[MicroPython中文社区](#)版权所有，保留所有权利

Copyright by [MicroPython Chinese community](#)

使用MicroPython前，需要了解一些基本的使用方法：

- 硬件

大部分MicroPython开发板都是通过串口（或者USB转串口方式）和计算机连接，部分版本支持Wifi，可以通过wifi进行连接。

通过USB连接时，通常会产生两个设备：虚拟磁盘和虚拟串口。用户程序可以复制到虚拟磁盘运行，也可以通过串口编写和调试程序。

- 软件

软件不要使用一般的串口助手类软件，而需要使用终端软件，如超级终端、putty、xshell、SecureCRT、MobaXterm等。

通过串口连接时，绝大部分使用了下面的参数：

```
115200, n, 8, 1, 无流量控制
```

- 驱动

在Windows下使用STM32的pyboard开发板时，第一次连接会提示安装串口驱动，同时会出现一个PYBFLASH虚拟磁盘，串口驱动就在这个虚拟磁盘上。

在Linux和MacOS下无需安装串口驱动。

- REPL

通过终端连接到开发板后，就可以通过REPL发送命令，编写和调试程序了，REPL下的用法和标准的python类似。

下面是网上关于MicroPython的介绍

Damien George是一名计算机工程师，他每天都要使用Python语言工作，同时也在做一些机器人项目。有一天，他突然冒出了一个想法：能否用Python语言来控制单片机，进行实现对机器人的操控呢？

要知道，Python是一款比较容易上手的脚本语言，而且有强大的社区支持，一些非计算机专业领域的人都选它作为入门语言。遗憾的是，它不能实现一些非常底层的操控，所以在硬件领域并不起眼。

Damien为了突破这种限制，他花费了六个月的时间来打造Micro Python。它基于ANSI C，语法跟Python 3基本一致，拥有自家的解析器、编译器、虚拟机和类库等。目前他支持基于32-bit的ARM处理器，比如说STM32F405。

借助Micro Python，用户完全可以通过Python脚本语言实现硬件底层的访问和控制，比如说控制LED灯泡、LCD显示器、读取电压、控制电机、访问SD卡等。

与此同时，Damien还给大家带来了一款专门为Micro Python而打造的开发板，它基于STM32F405单片机，通过USB接口进行数据传输。该开发板内置4个LED灯、一个加速传感器、时钟模块，可在3V-10V之间的电压正常工作。值得一提的是，它遵守MIT协议开源，被授权人拥有复制、修改、发行和再授权的权利。

MicroPython在2014成功的在kickstarter上众筹，获得很高的评价，现在越来越多的开发者开始使用MicroPython。

目前MicroPython有多个不同硬件平台的移植版本，包括STM32F4/F7/L4系列、ESP8266、ESP32、NXP MK20DX256、microchip PIC33、Infineon XMC4700、nRF51822、CC3200、MSP432等。其中以STM32和ESP8266为主要版本。除了官方维护的版本外，还有众多爱好者移植的版本。

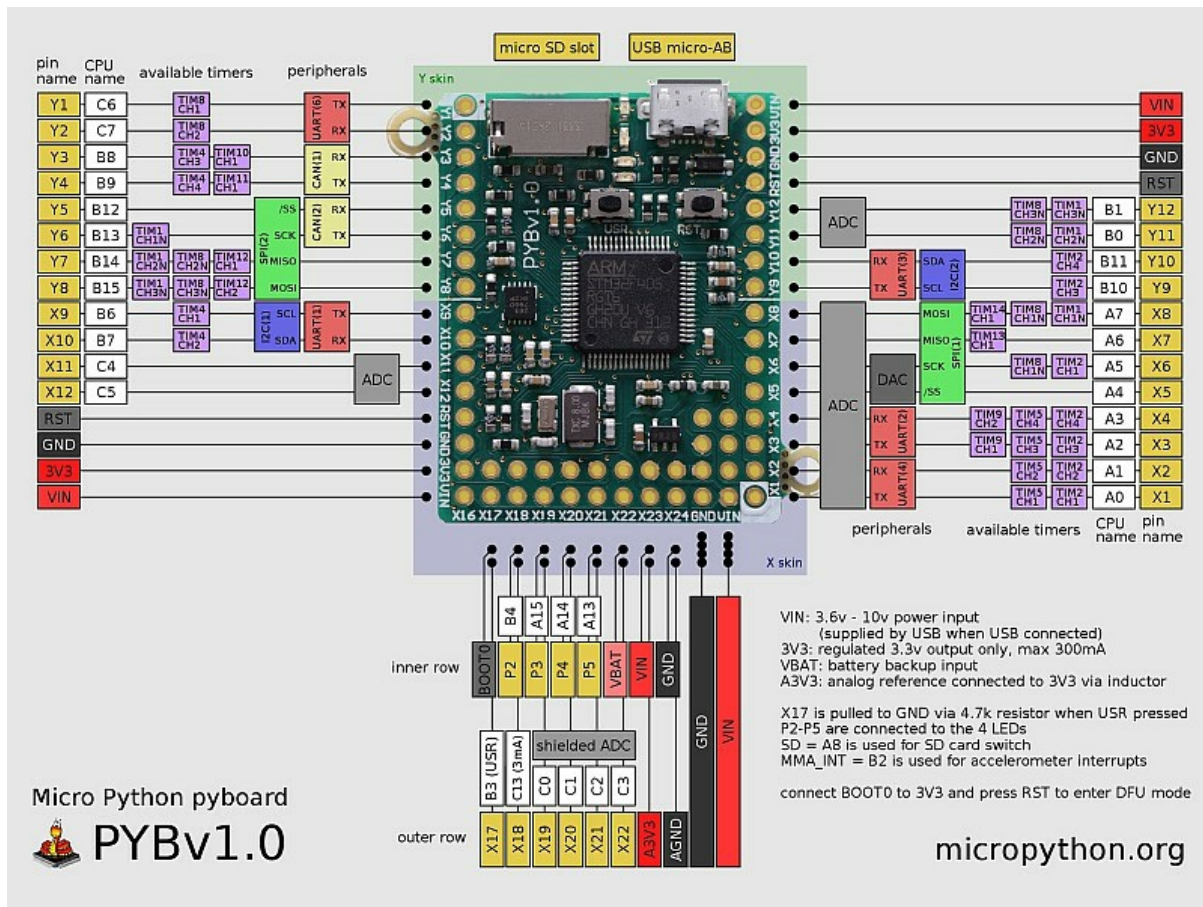
MicroPython采用了MIT授权方式，这是最宽松的授权方式，任何人都可以使用它，也可以用于商业应用。

实际上，除了MicroPython外，在嵌入式上还有其它一些python版

本，只是影响力不大。如更早的时候有pymite项目，可以在avr上运行python；还有一些不开源的商业项目，如：SNAPpy，能够在C8051和AVR上运行。但是这些项目的功能不如micropython，。

pyboard主要是针对STM32系列的单片机

PYBoard 快速指南



通用控制

LEDs

```
from pyb import LED

led = LED(1) # red led
led.toggle()
```

```
led.on()
led.off()
```

Pins 和 GPIO

```
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
```

电机控制

```
from pyb import Servo

s1 = Servo(1) # servo on position 1 (X1, VIN, GND)
s1.angle(45) # move to 45 degrees
s1.angle(-60, 1500) # move to -60 degrees in 1500ms
s1.speed(50) # for continuous rotation servos
```

外中断

```
from pyb import Pin, ExtInt

callback = lambda e: print("intr")
ext = ExtInt(Pin('Y1'), ExtInt.IRQ_RISING, Pin.PULL_NONE,
callback)
```

定时器

```
from pyb import Timer

tim = Timer(1, freq=1000)
tim.counter() # get counter value
tim.freq(0.5) # 0.5 Hz
tim.callback(lambda t: pyb.LED(1).toggle())
```

PWM (pulse width modulation)

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

```
p = Pin('X1') # X1 has TIM2, CH1
tim = Timer(2, freq=1000)
ch = tim.channel(1, Timer.PWM, pin=p)
ch.pulse_width_percent(50)
```

ADC (analog to digital conversion)

```
from pyb import Pin, ADC

adc = ADC(Pin('X19'))
adc.read() # read value, 0-4095
```

DAC (digital to analog conversion)

```
from pyb import Pin, DAC

dac = DAC(Pin('X5'))
dac.write(120) # output between 0 and 255
```

UART (serial bus)

```
from pyb import UART

uart = UART(1, 9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

SPI bus

```
from pyb import SPI

spi = SPI(1, SPI.MASTER, baudrate=200000, polarity=1, phase=0)
spi.send('hello')
spi.recv(5) # receive 5 bytes on the bus
spi.send_recv('hello') # send a receive 5 bytes
```

I2C bus

```
from pyb import I2C

i2c = I2C(1, I2C.MASTER, baudrate=100000)
i2c.scan() # returns list of slave addresses
i2c.send('hello', 0x42) # send 5 bytes to slave with address 0x42
i2c.recv(5, 0x42) # receive 5 bytes from slave
```

```
i2c.mem_read(2, 0x42, 0x10) # read 2 bytes from slave 0x42, slave  
memory 0x10  
i2c.mem_write('xy', 0x42, 0x10) # write 2 bytes to slave 0x42,  
slave memory 0x10
```

- [GPIO](#)
 - [PYB中未公开的Pin用法](#)
- [LED](#)
- [按键](#)
- [RTC](#)
- [ADC](#)
- [DAC](#)
- [UART](#)
- [Timer](#)
- [PWM](#)
- [I2C](#)
- [外中断](#)
- [USB VCP](#)
- [文件操作](#)
- [mascrSD](#)

GPIO的用法

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

- 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.cpu.B3, Pin.IN, Pin.PULL_UP)
```

- 返回GPIO的第二功能列表:`Pin.af_list()`
`Pin.af_list(pyb.Pin.board.X1)`
`Pin.af_list(LED)`

- 获取/设置debug状态: `Pin.debug(state)`
`Pin.debug(True)`
- 获取/设置GPIO映射字典: `Pin.dict(dict)`
`MyMapperDict = { 'LeftMotorDir' : pyb.Pin.cpu.C12 }`
`pyb.Pin.dict(MyMapperDict)`
- 获取/设置Pin映射: `Pin.mapper(func)`
- 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
 - pull
 - `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
- 获取/设置GPIO逻辑电平
`Pin.value(sw) # sw and LED has predefine`
`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())
```

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。

预定义的GPIO名称，仅对pyboard有效。

X1	PA0
X2	PA1
X3	PA2
X4	PA3
X5	PA4
X6	PA5
X7	PA6
X8	PA7
X9	PB6
X10	PB7
X11	PC4
X12	PC5
X13	Reset
X14	GND
X15	3.3V
X16	VIN
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

SW	PB3
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
SD	PA8
SD_SW	PA8
USB_VBUS	PA9
USB_ID	PA10
USB_DM	PA11
USB_DP	PA12

LED用法

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.
- `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：

```
import pyb

pyb.LED(1).on()
```

```
myled = pyb.LED(1)
myled.on()
myled.off()
myled.toggle()
```

设置LED3亮度

```
pyb.LED(3),intensity(10)
```

跑马灯

```
leds = [pyb.LED(i) for i in range(1,5)]

n = 0
while True:
    n = (n + 1) % 4
    leds[n].toggle()
    pyb.delay(50)
```

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)`
- 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()
```

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。

- 定义RTC对象

pyb.RTC

- 读取/设置rtc

rtc.datetime([datetimeuple])

datetimeuple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 倒计时

- 设置唤醒定时器

rtc.wakeup(timeout, callback=None)

timeout是毫秒

- 获取RTC启动时间和复位源

rtc.info()

- 获取/设置校正

rtc.calibration(cal)

无参数时读取校正值得，有参数时设置校正值得

-
- 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())
```

ADC的使用方法

基本用法

```
import pyb

adc = pyb.ADC(Pin('Y11'))      # create an analog object from a pin
adc = pyb.ADC(pyb.Pin.board.Y11)
val = adc.read()               # read an analog value

adc = pyb.ADCall(resolution)    # create an ADCall object
val = adc.read_channel(channel) # read the given channel
val = adc.read_core_temp()      # read MCU temperature
val = adc.read_core_vbat()      # read MCU VBAT
val = adc.read_core_vref()      # read MCU VREF
```

- `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
$$v_{\text{back}} = \text{adc.read_core_vbat}() * 1.21 / \text{adc.read_core_vref}()$$
- `adc.read_core_vref()`
读取vref电压（1.21V参考）
$$3V3 = 3.3 * 1.21 / \text{adc.read_core_vref}()$$

- `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 $\text{len(buf)}/\text{timer}$

```
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基本用法

```
from pyb import DAC

dac = DAC(1)           # create DAC 1 on pin X5
dac.write(128)         # write a value to the DAC (makes X5
1.65V)

dac = DAC(1, bits=12)  # use 12 bit resolution
dac.write(4095)        # output maximum value, 3.3V
```

输出正弦波

```
import math
from pyb import DAC

# create a buffer containing a sine-wave
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 * math.sin(2 * math.pi * i / len(buf)))

# output the sine-wave at 400Hz
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

# create a buffer containing a sine-wave, using half-word samples
buf = array('H', 2048 + int(2047 * math.sin(2 * math.pi * i /
128)) for i in range(128))

# output the sine-wave at 400Hz
dac = DAC(1, bits=12)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

- 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

UART的基本用法

```
from pyb import UART

u1 = UART(1, 9600)
u1.writechar(65)
u1.write('123')
u1.readchar()
u1.readall()
u1.readline()
u1.read(10)
u1.readinto(buf)
u1.any()
```

串口方法

- `class pyb.UART(bus, ...)`
 - `bus`: 1-6, 或者 'XA', 'XB', 'YA', 'YB' .
- `uart.init(baudrate, bits=8, parity=None, stop=1, *, timeout=1000, flow=None, timeout_char=0, read_buf_len=64)`, 初始化
 - `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
- `uart.deinit()`: 关闭串口
- `uart.any()`: 返回缓冲区数据个数, 大于0代表有数据
- `uart.writechar(char)`: 写入一个字节
- `uart.read([nbytes])`: 读取最多nbytes个字节。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
- `uart.readall()`: 读取所有数据

- `uart.readchar()`: 读取一个字节
- `uart.readinto(buf[, nbytes])`
 - `buf`: 数据缓冲区
 - `nbytes`: 最大读取数量
- `uart.readline()`: 读取一行
- `uart.write(buf)`: 写入缓冲区。在9bits模式下，两个字节算一个数据
- `uart.sendbreak()`: 往总线上发送停止状态，拉低总线13bit时间

串口对应GPIO

The physical pins of the UART busses are:

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)

定时器的用法

使用定时器前需要导入Timer库

```
from pyb import Timer
```

- 定义Timer

```
tm=Timer(n)
```

n=1-14，但是3用于内部程序，5/6用于伺服系统和ADC。

更多定义方式：

```
tm=Timer(1, freq=100)
```

```
tm=Timer(4, freq=200, callback=f)
```

- 设置频率

```
tm.freq(100)
```

- 定义回调函数

```
tm.callback(f)
```

- 禁用回调函数

```
tm.callback(None)
```

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

- `class pyb.Timer(id, ...)`

创建定时器对象，id范围是[1..14]

- `timer.init(*, freq, prescaler, period)`

初始化。

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

- `timer.deinit()`

禁止定时器, 禁用回调函数, 禁用任何定时器通道

- `timer.callback(fun)`

设置定时器回调函数

- `timer.channel(channel, mode, ...)`

设置定时器通道

- `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, 输入捕捉模式
 - 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

- timer.counter([value])

设置或获取定时器计数值

- `timer.freq([value])`
设置或获取定时器频率
- `timer.period([value])`
设置或获取定时器周期
- `timer.prescaler([value])`
设置或获取定时器预分频
- `timer.source_freq()`
获取定时器源频率（无预分频）
- `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。

方法

- `timerchannel.callback(fun)`
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.capture([value])`
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.compare([value])`
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.pulse_width([value])`
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。

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。

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

PWM的用法

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）。

PWM更多函数见[Timer](#)小节。

class SPI - 主SPI驱动

主SPI驱动，物理层上需要三根数据线：SCK, MOSI, MISO.

构造函数

方法

- `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。

Constants

- `SPI.MASTER`
初始化为主SPI模式

- SPI. MSB
设置高位在前模式

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。

外中断的用法

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)
```

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。

- `class pyb.USB_VCP`
创建虚拟串口对象
- `usb_vcp.setinterrupt(chr)`
设置中断python运行键，默认是3（Ctrl+C）。
-1是禁止中断功能，在需要发送原始字节时需要。
- `usb_vcp.isconnected()`
如果USB连接到串口设备，返回True
- `usb_vcp.any()`
如果缓冲区有数据等待接收，返回True
- `usb_vcp.close()`
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.read([nbytes])`
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.readall()`
读取缓冲区全部数据
- `usb_vcp.readinto(buf[, maxlen])`
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.readline()`
读取整行数据
- `usb_vcp.readlines()`
读取所有数据并分行存储，返回字节对象列表
- `usb_vcp.write(buf)`

写入缓冲区数据，返回写入数据的个数

- `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
- `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()
```

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()
```

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0V 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。

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
```

复位函数

- `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。

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。

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](#)

标准库，可以直接import后使用。

pyboard带有的标准库有：

- [cmath](#) - 复数运算
- [gc](#) - 垃圾回收
- [math](#) - 数学计算
- [select](#) - 等待事件
- [sys](#) - 系统函数
- [ubinaascii](#) - binary/ASCII 转换
- [ucollections](#) - 集合和容器类型
- [uhashlib](#) - hashing algorithm
- [uheapq](#) - heap queue algorithm
- [uio](#) - input/output streams
- [ujson](#) - JSON encoding and decoding
- [uos](#) - basic “operating system” services
- [ure](#) - regular expressions
- [usocket](#) - socket module
- [ustruct](#) - pack and unpack primitive data types
- [utime](#) - time related functions
- [uzlib](#) - zlib decompression

全部的内置函数

abs()
all()
any()
bin()
class bool
class bytearray
class bytes
callable()
chr()
classmethod()
compile()
class complex
class dict
dir()
divmod()
enumerate()
eval()
exec()
filter()
class float
class frozenset
getattr()
globals()
hasattr()
hash()
hex()
id()
input()
class int
isinstance()
issubclass()
iter()
len()
class list
locals()
map()
max()
class memoryview

min()
next()
class object
oct()
open()
ord()
pow()
print()
property()
range()
repr()
reversed()
round()
class set
setattr()
sorted()
staticmethod()
class str
sum()
super()
class tuple
type()
zip()

复数运算库

`cmath` 提供了基本的复数运算功能。它不支持 WiPy 和 ESP8266，因为需要浮点库支持。

函数

- `cmath.cos(z)`
余弦计算
- `cmath.exp(z)`
指数计算
- `cmath.log(z)`
自然对数计算
- `cmath.log10(z)`
常用对数计算（底数是10）
- `cmath.phase(z)`
相位，范围是 $(-\pi, +\pi)$ ，以弧度表示
- `cmath.polar(z)`
返回极坐标
- `cmath.rect(r, phi)`
产生复数
- `cmath.sin(z)`
计算正弦
- `cmath.sqrt(z)`
计算开平方

常数

- `cmath.e`
自然对数的底数
- `cmath.pi`
圆周率

garbage collector 垃圾回收

gc库可以回收系统运行中产生的垃圾。

- `gc.enable()`
允许自动回收垃圾
- `gc.disable()`
禁止自动回收，但可以手动进行回收
- `gc.collect()`
回收垃圾
- `gc.mem_alloc()`
返回已分配的内存数量
- `gc.mem_free()`
返回剩余的内存数量

数学计算库 (math)

函数

- `math.acos(x)`
计算反余弦
- `math.acosh(x)`
计算反双曲余弦
- `math.asin(x)`
计算反正弦
- `math.asinh(x)`
计算反双曲正弦
- `math.atan(x)`
计算反正切
- `math.atan2(y, x)`
计算 y/x 反正切.
- `math.atanh(x)`
计算反双曲正切
- `math.ceil(x)`
向上计算整数部分
- `math.copysign(x, y)`
返回 x 带有 y 的符号位
- `math.cos(x)`
计算余弦
- `math.cosh(x)`
计算双曲余弦
- `math.degrees(x)`
弧度转为角度

- `math.erf(x)`
返回误差函数
- `math.erfc(x)`
返回余误差函数
- `math.exp(x)`
计算指数
- `math.expm1(x)`
计算 $\exp(x) - 1$.
- `math.fabs(x)`
计算绝对值
- `math.floor(x)`
向下计算整数部分
- `math.fmod(x, y)`
计算余数
- `math.frexp(x)`
分解浮点数为尾数和指数。返回结果是元祖格式 (m, e) ，对应关系是 $x == m * 2^{**}e$ 。如果 $x == 0$ 就返回 $(0.0, 0)$ ，否则 otherwise $0.5 \leq \text{abs}(m) < 1$ holds.
- `math.gamma(x)`
计算伽马函数
- `math.isfinite(x)`
返回 `True` 如果是有限数
- `math.isinf(x)`
返回 `True` 如果是无穷大
- `math.isnan(x)`
如果不是数字返回 `True`
- `math.ldexp(x, exp)`
返回 $x * (2^{**}\text{exp})$.

- `math.lgamma(x)`
返回伽马函数的自然对数
- `math.log(x)`
计算自然对数
- `math.log10(x)`
计算常用对数（10为底）
- `math.log2(x)`
计算2为底的对数
- `math.modf(x)`
浮点数分解为小数和整数，小数在前
- `math.pow(x, y)`
计算指数
- `math.radians(x)`
角度转换为弧度
- `math.sin(x)`
计算正弦
- `math.sinh(x)`
计算双曲正弦
- `math.sqrt(x)`
计算开平方
- `math.tan(x)`
计算正切
- `math.tanh(x)`
计算双曲正切
- `math.trunc(x)`
取整数部分

常数

- `math.e`
自然对数的底数
- `math.pi`
圆周率

`select` - 提供了等待数据流的事件功能

Pyboard上功能

在读写多个对象时轮询是一种很有效的方式。目前支持轮询的对象有：`pyb.UART`, `pyb.USB_VCP`.

函数

- `select.poll()`
创建轮询实例
- `select.select(rlist, wlist, xlist[, timeout])`
等待活动对象。这个函数是为了兼容，效率不高，推荐用 `Poll` 函数.

`class Poll`

方法

- `poll.register(obj[, eventmask])`
注册轮询对象。`eventmask` 支持下面参数的逻辑 OR 操作：
 - `select.POLLIN` - 有数据可读取
 - `select.POLLOUT` - 可以写入数据
 - `select.POLLERR` - 发生错误
 - `select.POLLHUP` - 数据流结束/连接中断
- `poll.unregister(obj)`
解除轮询对象
- `poll.modify(obj, eventmask)`
修改对象的 `eventmask`
- `poll.poll([timeout])`
等待对象就绪。返回 `(obj, event, ...)` 的列表，`event` 元素是组合的已发生数据流事件。不同版本的函数可能包含其它参数，因此不用假设它的大小是2。超时后返回空列表。

Timeout 单位是毫秒。

sys - 系统函数

函数

- `sys.exit(retval=0)`
使用指定参数退出当前程序。它也会产生 `SystemExit` 异常，同时产生系统软复位。
- `sys.print_exception(exc, file=sys.stdout)`
打印异常到文件对象，默认是 `sys.stdout`。

和 CPython 的差异

这是CPython中回溯模块的简化版本。不同于 `traceback.print_exception()`，这个函数用异常值代替了异常类型、异常参数和回溯对象。文件参数在对应位置，不支持更多参数。CPython 兼容回溯模块在 `micropython-lib`。

常数

- `sys.argv`
启动参数列表
- `sys.byteorder`
字节顺序（“小” 或 “大”）。
- `sys.implementation`
当前 Python 情况，如 `(name='micropython', version=(1, 8, 1))`。对于 MicroPython，它返回下面属性：

名称 - “micropython”

版本 - (主，次，微)，如 (1, 7, 0)

这个方法推荐用来识别 MicroPython 和其它的 Python（注意少数移植版不支持）。

和 CPython 的差异

CPython 包含了更多属性，MicroPython支持基本功能。

- `sys.maxsize`
整数类型最大的数值。或MicroPython如果它小于os最大值（当MicroPython 移植版不支持 `long int` 时）。

这个属性可以用来检测平台的“bitness”（32位或64位等）。推荐不要字节比较属性值，而是象下面这样计算：

```
bits = 0
v = sys.maxsize
while v:
    bits += 1
    v >>= 1
if bits > 32:
    # 64-bit (or more) platform
    ...
else:
    # 32-bit (or less) platform
    # Note that on 32-bit platform, value of bits may be less than
    32
    # (e.g. 31) due to peculiarities described above, so use ">
    16",
    # "> 32", "> 64" style of comparisons.
```

- `sys.modules`
已载入模块字典。在某些移植版中，它可能不包含在内建模块中。
- `sys.path`
系统路径，
- `sys.platform`
MicroPython 运行的平台。在 OS/RTOS 移植版本中，通常表示 OS，如“linux”。在一般移植中它代表使用的开发板，如在最初的 MicroPython 中是“pyboard”。它可以用来识别不同的板子，如果需要识别运行环境(在其它 Python 环境下)，请使用 `sys.implementation`。
- `sys.stderr`
标准错误输出设备（默认是USB虚拟串口，可选其他串口）
- `sys.stdin`
标准输入设备（默认是USB虚拟串口，可选其他串口）

- `sys.stdout`
标准输出设备（默认是USB虚拟串口，可选其他串口）
- `sys.version`
Python 语言版本，字符串格式。
- `sys.version_info`
Python 语言版本，整数元祖格式。

ubinascii - binary/ASCII 转换

函数

- `ubinascii.hexlify(data[, sep])`
转换二进制数据为16进制字符串。如：

```
ubinascii.hexlify(b' \x11\x22123')  
b' 1122313233'
```

和 CPython 的差异

如果指定了第二个参数，它将用于分隔两个HEX参数，如：

```
ubinascii.hexlify(b' \x11\x22123', ' ' )  
b' 11 22 31 32 33'
```

```
ubinascii.hexlify(b' \x11\x22123', ', ' )  
b' 11, 22, 31, 32, 33'
```

如果sep设定了多个字符，只有第一个有效。

- `ubinascii.unhexlify(data)`
转换HEX数据为二进制字符串，功能和hexlify反向。

```
ubinascii.unhexlify(' 313233')  
b' 123'
```

- `ubinascii.a2b_base64(data)`
转换 Base64 编码数据为二进制字符串。
- `ubinascii.b2a_base64(data)`
编码二进制数据为 Base64 格式。

ucollections - 集合和容器类型

该模块实现了高级集合和容器类型，以容纳各种对象。

Classes

- `ucollections.namedtuple(name, fields)`
使用指定名称和字段用工厂函数创建新的命名元组类型。命名元组类型是元组子集，不但可以用索引访问，也可以通过符号字段名访问，字段是指定名称的字符串序列。为了兼容 CPython，它也可以是用空格分隔的字符串字段名(但是效率很低)。例如：

```
from ucollections import namedtuple

MyTuple = namedtuple("MyTuple", ("id", "name"))
t1 = MyTuple(1, "foo")
t2 = MyTuple(2, "bar")
print(t1.name)
assert t2.name == t2[1]
```

- `ucollections.OrderedDict(...)`
字典类型子集，它会按顺序保存添加的键值。当字典迭代完成，按照添加时的顺序：

```
from ucollections import OrderedDict

# To make benefit of ordered keys, OrderedDict should be
# initialized
# from sequence of (key, value) pairs.
d = OrderedDict([("z", 1), ("a", 2)])
# More items can be added as usual
d["w"] = 5
d["b"] = 3
for k, v in d.items():
    print(k, v)
```

输出结果：

```
z 1
a 2
w 5
b 3
```

uhashlib - 哈希算法（散列算法）

这个模块执行哈希算法，目前可以使用 SHA256 算法。选择 SHA256 是经过仔细考虑的，因为它是流行的、安全的算法。这意味着一个单一的算法可以覆盖“任何哈希算法”和安全相关的应用，省去传统算法如MD5或SHA1从而节省空间。

构造函数

- `class uhashlib.sha256([data])`
创建哈希对象，可以选择填充数据

方法

- `hash.update(data)`
填充数据
- `hash.digest()`
返回经过散列的数据，结果是字节对象。调用这个方法后，不能在输送数据到散列。
- `hash.hexdigest()`
这个函数不再使用，请使用 `ubinscii.hexlify(hash.digest())` 代替。

uheapq - 堆排队算法

提供堆排队算法，堆队列是一个简单列表，它的元素以特定方式存储
函数

- `uheapq.heappush(heap, item)`
项目推入堆中
- `uheapq.heappop(heap)`
弹出并返回堆中的第一个项目。如果堆是空的将引起 `IndexError` 异常。
- `uheapq.heapify(x)`
转换列表到堆。

uio - 输入/输出流

包含流类型（类似文件）对象和帮助函数

函数

- `uio.open(name, mode='r', **kwargs)`
打开一个文件，关联到内建函数 `open()`。所有端口（用于访问文件系统）需要支持模式参数，但支持其他参数不同的端口。

Classes

- `class uio.FileIO(...)`
这个文件类型用二进制方式打开文件，等于使用 `open(name, "rb")`。不应直接使用这个实例。
- `class uio.TextIOWrapper(...)`
这个类型以文本方式打开文件，等同于使用 `open(name, "rt")`。不应直接使用这个实例。
- `class uio.StringIO([string])`
- `class uio.BytesIO([string])`
内存文件对象。`StringIO` 用于文本模式 I/O（用 “t” 打开文件），`BytesIO` 用于二进制方式（用 “b” 方式）。文件对象的初始内容可以用字符串参数指定（`stringio` 用普通字符串，`bytesio` 用 `bytes` 对象）。所有的文件方法，如 `read()`，`write()`，`close()` 都可以用在这些对象上，包括下面方法：

`getvalue()`
获取缓冲区内容。

ujson - JSON 编码解码

提供 Python 对象到 JSON (JavaScript Object Notation) 数据格式的转换。

函数

- `ujson.dumps(obj)`
返回 JSON 字符串
- `ujson.loads(str)`
解析 JSON 字符串并返回对象。如果字符串格式错误将引发 `ValueError` 异常。

uos - 基础 “操作系统” 服务

os模块提供文件函数和随机数函数

移植说明

文件系统将 / 做为根目录，其它物理驱动器从根目录访问。目前支持：

/flash - 内部 flash 文件系统

/sd - SD 卡（如果存在）

启动时，如果没有SD卡，当前目录就是 /flash，否则是 /sd。

函数

- uos.chdir(path)
改变当前目录
- uos.getcwd()
获取当前目录
- uos.listdir([dir])
无参数时列出当前目录文件，否则列出指定目录的文件
- uos.mkdir(path)
创建新目录
- uos.remove(path)
删除文件
- uos.rmdir(path)
删除目录
- uos.rename(old_path, new_path)
文件改名
- uos.stat(path)
获取文件或目录状态

- `uos.sync()`
同步所有文件系统
- `uos.urandom(n)`
返回 `n` 字节的随机数，随机数由硬件随机数发生器产生。

常数

- `uos.sep`
路径的分隔符

ure - 正则表达式

执行正则表达式操作。正则表达式支持 CPython 子集 re 模块（实际是 POSIX 扩展正则表达式的子集）。

支持操作符：

- `'.'`：匹配任意字符
- `'[]'`：匹配字符集合，支持单个字符和一个范围。
- `'^'`
 - `'$'`
 - `'?'`
 - `'*'`
 - `'+'`
 - `'??'`
 - `'*?'`
- `'+?'`：重复计数 (`{m,n}`)，不支持高级的断言、命名组等。

函数

- `ure.compile(regex)`
编译正则表达式，返回 regex 对象
- `ure.match(regex, string)`
用 string 匹配 regex，匹配总是从字符串的开始匹配
- `ure.search(regex, string)`
在 string 中搜索 regex。不同于匹配，它搜索第一个匹配位置的正则表达式字符串（结果可能会是0）。
- `ure.DEBUG`
标志值，显示表达式的调试信息。

Regex 对象

编译正则表达式，使用 `ure.compile()` 创建实例

```
regex.match(string)
regex.search(string)
regex.split(string, max_split=-1)
```

匹配对象

匹配对象是 `match()` 和 `search()` 方法返回值

- `match.group([index])`
只支持数字组

usocket - socket module

BSD 套接字接口

查看 [CPython 对应的模块](#) 进行比较

Socket 地址格式

下面函数使用 ipv4 格式（地址：端口）网络地址，ipv4 地址是由点和数字组成的字符串，如 "8.8.8.8"，端口是 1-65535 的数字。注意不能使用域名做为 ipv4 地址，域名需要先用 `socket.getaddrinfo()` 进行解析。

函数

- `socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP)`
创建新的套接字，使用指定的地址、类型和协议号。
- `socket.getaddrinfo(host, port)`
传递 主机/端口 到一个5个数据的元组。元组列表的结构如下：

(family, type, proto, canonname, sockaddr)

下面显示了怎样连接到一个网址：

```
s = socket.socket()
s.connect(socket.getaddrinfo('www.micropython.org', 80)
[0][-1])
```

常数

- `socket.AF_INET`
family 类型
- `socket.SOCK_STREAM`
- `socket.SOCK_DGRAM`
socket 类型

- `socket.IPPROTO_UDP`
- `socket.IPPROTO_TCP`

套接字类型

方法

- `socket.close()`
关闭套接字。一旦关闭后，套接字所有的功能都将失效。远端将接收不到任何数据（清理队列数据后）。

在回收垃圾时套接字会自动关闭，但还是推荐在必要时用 `close()` 去关闭，或，`or to use a with statement around them.`

- `socket.bind(address)`
将套接字绑定到地址，套接字不能是已经绑定的。
- `socket.listen([backlog])`
允许服务器接收连接。如果指定了 `backlog`，它不能小于0（如果小于0将自动设置为0）；超出后系统将拒绝新的连接。如果没有指定，将使用默认值。
- `socket.accept()`
接收连接。套接字需要指定地址并监听连接。返回值是 `(conn, address)`，其中`conn`是用来接收和发送数据的套接字，`address`是绑定到另一端的套接字。
- `socket.connect(address)`
连接到指定地址的远端套接字。
- `socket.send(bytes)`
发送数据。套接字需要已连接到远程。
- `socket.sendall(bytes)`
发送数据。套接字已连接到远程。
- `socket.recv(bufsize)`

接收数据，返回值是数据字节对象。bufsize是接收数据的最大数量。

- `socket.sendto(bytes, address)`
发送数据。套接字没有连接到远程，目标套接字由地址参数指定。
- `socket.recvfrom(bufsize)`
接收数据。返回值是 (bytes, address)，其中 bytes 是字节对象，address 是发送数据的套接字。
- `socket.setsockopt(level, optname, value)`
设置套接字参数。需要的符号常数定义在套接字模块 (SO_* 等)。value 可以是整数或字节对象。
- `socket.settimeout(value)`
设置阻塞套接字超时时间。value 参数可以是代表秒的正浮点数或 None。如果设定大于 0 的参数，在后面套接字操作超出指定时间后将引起 timeout 异常。如果参数是 0，套接字将使用非阻塞模式。如果是 None，套接字使用阻塞模式。
- `socket.setblocking(flag)`
设置阻塞或非阻塞模式：如果 flag 是 false，设置非阻塞模式。

这是调用 `settimeout()` 的一种简便方法：

```
sock.setblocking(True) 等于 sock.settimeout(None)
sock.setblocking(False) 等于 sock.settimeout(0.0)
```

- `socket.makefile(mode='rb')`
返回关联到套接字的文件对象，返回值类型与指定的参数有关。仅支持二进制模式 ('rb' 和 'wb')，CPython 的 encoding、errors 和 newline 不被支持。

套接字必须是阻塞模式，它可以指定超时，但是当发生超时异常后文件内部缓存状态可能不一致。

和 CPython 的不同

关闭文件也会同时关闭套接字。

- `socket.read(size)`
读取指定字节数据，返回参数是字节对象。如果没有指定 `size`，结果和 `socket.readall()` 相同。
- `socket.readall()`
读取全部数据，直到 EOF。函数直到套接字关闭才返回。
- `socket.readinto(buf[, nbytes])`
读取到缓冲区。如果指定了 `nbytes`，那么最多只读取 `nbytes` 字节，否则最多读取 `len(buf)` 字节。

返回值是读取的字节数。

- `socket.readline()`
读取一行，以换行符结束。

返回读取的数据行。

- `socket.write(buf)`
写入缓冲区数据。

返回值是写入的数据数量。

ustruct - 压缩和不压缩原始数据类型

请参考 [Python struct](#)。

支持的 size/byte 前缀: @, <, >, !.

支持的格式代码: b, B, h, H, i, I, l, L, q, Q, s, P, f, d (最后两个需要浮点库支持).

函数

- `ustruct.calcsize(fmt)`
返回存放 `fmt` 需要的字节数.
- `ustruct.pack(fmt, v1, v2, ...)`
按照格式字符串 `fmt` 压缩参数 `v1, v2, ...`。返回值是参数编码后的字节对象。
- `ustruct.pack_into(fmt, buffer, offset, v1, v2, ...)`
按照格式字符串 `fmt` 压缩参数 `v1, v2, ...` 到缓冲区 `buffer`, 开始位置是 `offset`。 `offset` 可以是负数, 从缓冲区末尾开始计数。0
- `ustruct.unpack(fmt, data)`
从 `fmt` 中解压数据。返回值是解压后参数的元组。
- `ustruct.unpack_from(fmt, data, offset=0)`
从 `fmt` 的 `offset` 开始解压数据, 如果 `offset` 是负数就是从缓冲区末尾开始计算。返回值是解压后参数元组。

utime - 时间函数库

utime 库提供获取时间和日期、测量时间间隔、延时等功能。

初始时刻：Unix 使用 POSIX 系统标准，从 1970-01-01 00:00:00 UTC开始。嵌入式程序从 2000-01-01 00:00:00 UTC 开始。

日期/时间：需要一个实时时钟（RTC）。在底层系统（包括一些 RTOS 中），RTC 已经包含在其中。设置时间是通过 OS/RTOS 而不是 MicroPython 完成，查询日期/时间也需要通过系统 API。对于裸板系统时钟依赖于 machine.RTC() 对象。设置时间通过 machine.RTC().datetime(tuple) 函数，并通过下面方式维持：

- 后备电池（可能是选件、扩展板等）。
- 使用网络时间协议（需要用户设置）。
- 每次上电时手工设置（大部分只是在硬复位时需要设置，少部分每次复位都需要设置）。

如果实际时间不是通过系统/MicroPython RTC维持，那么下面函数结果可能不是和预期的相同。

函数

- `utime.localtime([secs])`
从初始时间的秒转换为元组：(年, 月, 日, 时, 分, 秒, 星期, yearday)。如果 secs 是空或者 None，那么使用当前时间。
 - 年包括了 (如 2014)。
 - 月是 1-12
 - 日是 1-31
 - 小时是 0-23
 - 分钟是 0-59
 - 秒是 0-59
 - 星期是 0-6 代表周一到周日
 - yearday 是 1-366
- `utime.mktime()`
时间的反函数，它的参数是完整8参数的元组，返回值是从 Jan 1, 2000开始的秒数。
- `utime.sleep(seconds)`
休眠指定的时间（秒），Seconds 可以是浮点数。注意有些版本

的 MicroPython 不支持浮点数，为了兼容可以使用 `sleep_ms()` 和 `sleep_us()` 函数。

- `utime.sleep_ms(ms)`
延时指定毫秒，参数不能小于0。
- `utime.sleep_us(us)`
延时指定微秒，参数不能小于0。
- `utime.ticks_ms()`
返回不断递增的毫秒计数器，在某些值后会重新计数(未指定)。计数值本身无特定意义，只适合用在 `ticks_diff()`。
- `utime.ticks_us()`
和上面类似，只是返回微秒。
- `utime.ticks_cpu()`
和 `ticks_ms/ticks_us` 类似，具有更高精度（使用 CPU 时钟）。
- `utime.ticks_diff(old, new)`
计算两次调用 `ticks_ms()`，`ticks_us()`，或 `ticks_cpu()` 之间的时间。因为这些函数的计数值可能会回绕，所以不能直接相减，需要使用 `ticks_diff()` 函数。“旧”时间需要在“新”时间之前，否则结果无法确定。这个函数不要用在计算很长的时间（因为 `ticks_*` 函数会回绕，通常周期不是很长）。通常用法是在带超时的轮询事件中调用：

```
# Wait for GPIO pin to be asserted, but at most 500us
start = time.ticks_us()
while pin.value() == 0:
    if time.ticks_diff(start, time.ticks_us()) > 500:
        raise TimeoutError
```

- `utime.time()`
返回从开始时间的秒数（整数），假设 RTC 已经按照前面方法设置好。如果 RTC 没有设置，函数将返回参考点开始计算的秒数（对于 RTC 没有后备电池的板子，上电或复位后的情况）。如果你开发便携版的 MicroPython 应用程序，你不要依赖函数来提供超过秒级的精度。如果需要高精度，使用 `ticks_ms()` 和 `ticks_us()` 函数。如果需要日历时间，使用不带参数的

`localtime()` 是更好选择。

与 CPython 的不同

在 CPython 中，这个函数用浮点数返回从 Unix 开始时间（1970-01-01 00:00 UTC）的秒数，通常是毫秒级的精度。在 MicroPython 中，只有 Unix 版才使用相同开始时间，如果允许浮点精度，将返回亚秒精度。嵌入式硬件通常没有用浮点数表示长时间访问和亚秒精度，所以返回值是整数。一些嵌入式系统硬件不支持 RTC 电池供电方式，所以返回的秒数是从最后上电、或相对某个时间、以及特定硬件时间（如复位）。

uzlib - zlib 解压缩

使用 DEFLATE 算法解压缩二进制数据（常用于 zlib 库和 gzip 文档）。压缩尚未实现。

函数

- `uzlib.decompress(data)`
返回解压后的 bytes 对象。

MicroPython 特殊库

machine — 板级函数

提供和硬件相关的函数

复位相关函数

- `machine.reset()`
设备复位，效果和按下复位键一样。
- `machine.reset_cause()`
获取复位原因。

中断相关函数

- `machine.disable_irq()`
禁止中断。返回先前的 IRQ 状态: `False/True` 对应 `disabled/enabled` IRQs, 返回值可以用来恢复 IRQ 状态。
- `machine.enable_irq(state=True)`
允许中断。如果 `state` 是 `True` (默认) 将允许 IRQs, 否则禁止 IRQs, 最常用的方式是在退出临界代码时将 `disable_irq` 返回值传递到函数。

功耗相关函数

- `machine.freq()`
返回 CPU 频率 (Hz)。
- `machine.idle()`
中断到 CPU 的时钟, 减少系统功耗。外设继续工作, 发生任意中断后恢复运行 (大部分版本中, 包括了系统定时器中断)。
- `machine.sleep()`
停止 CPU 并禁止所有外设, 除了 WLAN。唤醒后从休眠位置继续运行, 休眠前需要首先配置唤醒源。
- `machine.deepsleep()`
停止 CPU 和所有外设 (包括网络)。唤醒后从 `main.py` 开始运行,

就像复位一样，可以通过 `reset_cause` 查看从什么地方运行。如果需要唤醒，需要先配置好唤醒源，，如 Pin 状态改变或 RTC 超时。

其它函数

- `machine.unique_id()`
获取 board/SoC 的唯一序列号。如果底层硬件允许这个功能，每个板的 ID 都是不同的，ID 的长度由硬件决定（因此可以使用完整值的子串如果希望得到短 ID）。在某些 MicroPython 版本中，使用网络 MAC 地址代表 ID。
- `machine.time_pulse_us(pin, pulse_level, timeout_us=1000000)`
在指定引脚上输出脉冲，返回脉冲持续时间（微秒）。

首先等待引脚上输入电平等于 `pulse_level`，然后进行计时直到引脚电平和 `pulse_level` 电平不同。如果引脚电平已经和 `pulse_level` 相同，那么将立即开始计时。

函数在等待时间超时后会引发 `ETIMEDOUT` 异常。

常数

- `machine.IDLE`
- `machine.SLEEP`
- `machine.DEEPSLEEP`
irq 唤醒参数
- `machine.POWER_ON`
- `machine.HARD_RESET`
- `machine.WDT_RESET`
- `machine.DEEPSLEEP_RESET`
- `machine.SOFT_RESET`
复位原因
- `machine.WLAN_WAKE`
- `machine.PIN_WAKE`
- `machine.RTC_WAKE`
- `wake reasons`

唤醒原因

Classes

- class ADC - 模数转换
- class ADCChannel — read analog values from internal or external sources
- class I2C - a two-wire serial protocol
- class Pin - control I/O pins
- class RTC - real time clock
- class SD - secure digital memory card
- class SPI - a master-driven serial protocol
- class Timer - control internal timers
- class TimerChannel — setup a channel for a timer
- class UART - duplex serial communication bus
- class WDT - watchdog timer

micropython - 访问和控制MicroPython

函数

- `micropython.mem_info([verbose])`
打印使用的内存。如果指定 `verbose` 参数将显示更多信息。

显示的信息和移植版本有关，一般包括堆栈使用情况，在冗余模式下，还将显示使用的块和自由块。

- `micropython.qstr_info([verbose])`
显示使用的字符串，如果指定 `verbose` 参数将显示更多信息。

显示的信息和移植版本有关，一般显示使用的字符串数量和占用的内存，冗余模式下还将显示所有字符串的名称。

- `micropython.alloc_emergency_exception_buf(size)`
为紧急异常分配的缓存（一个合适的大小约为 100 字节）。这个缓存用于在异常情况下使用常规方式分配内存失败时（如中断处理），给出有用的回溯信息。

一个好的用法是将这个函数放在主程序的开始（如 `boot.py` 或 `main.py`），紧急异常缓存将对后面所有代码都生效。

network — 网络配置

这个模块提供了网络驱动和程序配置。它可以驱动特定硬件的网络，配置网络接口，然后通过 `socket` 模块使用网络。使用网络模块必须安装带有网络驱动的固件。

例如：

```
# configure a specific network interface
# see below for examples of specific drivers
import network
nic = network.Driver(...)
print(nic.ifconfig())

# now use socket as usual
import socket
addr = socket.getaddrinfo('micropython.org', 80)[0][-1]
s = socket.socket()
s.connect(addr)
s.send(b'GET / HTTP/1.1\r\nHost: micropython.org\r\n\r\n')
data = s.recv(1000)
s.close()
```

`class CC3K`

提供 TI 的 CC3000 wifi 模块驱动。使用方法：

```
import network
nic = network.CC3K(pyb.SPI(2), pyb.Pin.board.Y5,
pyb.Pin.board.Y4, pyb.Pin.board.Y3)
nic.connect('your-ssid', 'your-password')
while not nic.isconnected():
    pyb.delay(50)
print(nic.ifconfig())

# now use socket as usual
...
```

这个例子需要连接下面端口：

- MOSI 连接到 Y8
- MISO 连接到 Y7
- CLK 连接到 Y6

- CS 连接到 Y5
- VBEN 连接到 Y4
- IRQ 连接到 Y3

可以使用其它 SPI 总线和其它端口连接到 CS, VBEN 和 IRQ。

构造函数

- `class network.CC3K(spi, pin_cs, pin_en, pin_irq)`
创建 CC3K 驱动对象，用指定的 spi 和 gpio 初始化 CC3000 模块，返回 CC3K 对象。

参数：

- spi, 连接到 CC3000 模块的 SPI 对象 (MOSI, MISO 和 CLK 端口)。
- pin_cs 连接到 CC3000 的 CS 端口。
- pin_en 连接到 CC3000 的 VBEN 端口。
- pin_irq 连接到 CC3000 的 IRQ 端口。

所有对象由驱动进行初始化，所以用户不用自己进行初始化。例如，你可以：

```
nic = network.CC3K(pyb.SPI(2), pyb.Pin.board.Y5,
pyb.Pin.board.Y4, pyb.Pin.board.Y3)
```

方法

- `cc3k.connect(ssid, key=None, *, security=WPA2, bssid=None)`
用给定的 SSID 和密码等参数连接到 wifi 访问点。
- `cc3k.disconnect()`
断开 wifi 连接。
- `cc3k.isconnected()`
返回 True 如果已经连接到 wifi 并获取有效的 IP 地址，否则返回 False。

- `cc3k.ifconfig()`
返回 7 参数元组 (ip, subnet mask, gateway, DNS server, DHCP server, MAC address, SSID)。
- `cc3k.patch_version()`
返回补丁程序版本 (固件)。
- `cc3k.patch_program('pgm')`
上传固件到 CC3000。必须将 'pgm' 做为第一个参数上传。

常数

- `CC3K.WEP`
- `CC3K.WPA`
- `CC3K.WPA2`

使用的安全类型

`class WIZNET5K`

这个类允许控制 WIZnet5x00 使用了 W5200 或 W5500 芯片的以太网适配器 (仅测试了 W5200)。

例如:

```
import network
nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5,
pyb.Pin.board.X4)
print(nic.ifconfig())

# now use socket as usual
...
```

这个例子中连接了下面端口到 WIZnet5x00 模块:

- MOSI 连接到 X8
- MISO 连接到 X7
- SCLK 连接到 X6
- nSS 连接到 X5

- nRESET 连接到 X4

可以使用其它 SPI 总线和其它端口。

构造函数

`class network.WIZNET5K(spi, pin_cs, pin_rst)`
创建 WIZNET5K 对象，使用指定 SPI 和端口进行初始化，返回 WIZNET5K 对象。

参数：

- `spi` 是连接 WIZnet5x00 的 SPI 对象（包含 MOSI, MISO 和 SCLK 引脚）。
- `pin_cs` 是连接 WIZnet5x00 nSS 引脚的 Pin 对象。
- `pin_rst` 是连接到 WIZnet5x00 nRESET 的 Pin 对象。

所有这些对象由驱动进行初始化，因此无需预先初始化。例如：

```
nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5,  
pyb.Pin.board.X4)
```

方法

- `wiznet5k.ifconfig([(ip, subnet, gateway, dns)])`
获取/设置 IP 地址，子网掩码，网关和 DNS。

当不带参数时，返回上述参数的 4 参数元组。设置参数时，传递 4 参数元组。例如：

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1',  
'8.8.8.8'))
```

- `wiznet5k.regs()`
转储 WIZnet5x00 寄存器，通常用于调试。

uctypes - 有条理的访问二进制数据

这个模块实现 MicroPython 的“外部数据接口”。它背后的想法类似于 CPython 的加密模块，但是 API 不同，优化了大小。基本思路是定义和 C 语言相同功能的数据结构层，并通过类似的方法访问子域。

参考 [ustruct 模块](#)

标准的 Python 方法访问二进制数据结构（不适合大量和复杂的结构）。

定义数据结构层

结构层定义是通过“描述符” - Python 字典编码字段名称作为键值和其它属性。目前，uctypes 要求每个字段明确规范偏移量，偏移量从结构开始字节计算。

下面是编码例子：

- 标量类型：

```
"field_name": uctypes.UINT32 | 0
```

换句话说，数值是带有偏移量的标量类型标识符（字节）。

- 递归结构：

```
"sub": (2, {  
    "b0": uctypes.UINT8 | 0,  
    "b1": uctypes.UINT8 | 1,  
})
```

例如，数值是一个 2 元组，第一个元素是偏移量，第二个是结构描述字典（注：偏移量在递归描述符是相对于结构定义的）。

- 基本类型数组：


```
"arr": (uctypes.ARRAY | 0, uctypes.UINT8 | 2),
```

例如数值是一个 2 元组，第一个元素是带有偏移量数组标志，第二个是数组中元素数量的标量元素类型

- 聚集类型数组：

```
"arr2": (uctypes.ARRAY | 0, 2, {"b": uctypes.UINT8 | 0}),
```

参数是三元组，第一个元素是带有偏移量的数组标志，第二个是数字，第三个是元素类型说明。

- 指向原始类型的指针：

```
"ptr": (uctypes.PTR | 0, uctypes.UINT8),
```

参数是二元组，第一个元素是带偏移量的 PTR 标志，第二个是标量元素类型。

- 指向集合类型的指针：

```
"ptr2": (uctypes.PTR | 0, {"b": uctypes.UINT8 | 0}),
```

参数是二元组，第一个元素是带偏移量的 PTR 标志，第二个是指针说明。

- 位域：

```
"bitf0": uctypes.BFUINT16 | 0 | 0 << uctypes.BF_POS | 8 <<  
uctypes.BF_LEN,
```

参数是位域标量（类型名类似标量类型，但带有前缀“BF”），或与标量偏移量，或与位域长度，分别由 BF_POS 和 BF_LEN 进行位移。位域是从最低有效位开始，到右边最高位（换句话说，它是一个位通过右移而成）。

上面例子中，第一个 UINT16 参数在偏移量 0 处提取（在访问硬

件寄存器时这很重要，需要特定大小和对齐)，最右边是 UINT16 的最低位，长度是 8 bits - 实际上它访问的是 UINT16 低字节。

注意位域运算是独立于目标的字节顺序，上面例子中在小端或大端结构中都会访问 UINT16 的低字节。它取决于最低有效位编号 0，一些系统使用与原生编号不同的方式，但 ctypes 总是使用上述编号规范。

模块内容

- `class ctypes.struct(addr, descriptor, layout_type=NATIVE)`
“外部数据结构”对象，基于内存中结构地址，说明（编码为字典），以及类型（看下面）。
- `ctypes.LITTLE_ENDIAN`
小端压缩结构。（压缩意味着每一个字段都占用的字节数正好是描述符中定义的字节数，即对齐方式为1）
- `ctypes.BIG_ENDIAN`
大端压缩结构类型。
- `ctypes.NATIVE`
原生结构 - 数据的字节顺序、对齐按照系统方式。
- `ctypes.sizeof(struct)`
按字节返回数据的大小。参数可以是类或者数据对象（或集合）。
- `ctypes.addressof(obj)`
返回对象地址。参数需要是 `bytes`, `bytearray` 或缓存（返回缓存的实际地址）。
- `ctypes.bytes_at(addr, size)`
捕获给定大小和地址内存作为 `bytes` 对象。因为字节对象是不可变的，实际是复制内存到字节对象，如果内存内容发生改变，创建的字节对象还是原始值。
- `ctypes.bytearray_at(addr, size)`

捕捉给定大小和地址内存为 `bytearray` 对象，不像 `bytes_at()` 函数，捕捉的内存是引用值，因此它可以被再次写入，可以访问给定地址的参数。

结构说明和实例化结构对象

给定一个结构描述词典及其类型，就可以使用 `uctypes.struct()` 在指定内存地址构建实例。内存地址通常来自：

- 预定义地址，在裸机中访问硬件寄存器时，可以在数据手册中查找这些地址。
- 作为某些 FFI（外部函数接口）函数调用的返回值。
- 来自 `uctypes.addressof()`，当传递参数到 FFI 函数，或通过 I/O 访问数据（例如，从文件或网络套接字读取数据）。

结构对象

结构对象允许使用标准的点访问字段：

`my_struct.substruct1.field1`。如果字段是标量类型，就会产生一个相应的原始值（Python 整数或浮点数）。也可以分配标量到字段。

如果字段是数组，它的每个元素可以通过标志下标操作符 `[]` 访问 - 读或分配。

如果字段是指针，可以使用 `[0]` 语法引用（对应 C 的 `*` 操作符，在 C 语言中也可以使用 `[0]`）。指针也可以使用非 0 的整数作为下标，和 C 语言类似。

总结来说，访问结构字段通常遵循 C 语言的语法，除了指针有所不同，需要用 `[0]` 代替 `*`。

限制

访问非标量域导致分配临时对象代表它们，这意味着在禁止内存分配时访问它们需要特别注意（如在中断里）。推荐：

- 避免嵌套。例如不要使用 `mcu_registers.peripheral_a.register1` 这样用法，而是每一层

- 单独定义，再访问 `peripheral_a.register1`。
- 避免非标量数据，如数组。例如不要使用 `peripheral_a.register[0]`，而是使用 `peripheral_a.register0`。

请注意这些建议会降低可读性和简洁性，因此只在不能分配内存时使用它（甚至可以定义两个并行的结构，一个用在正常模式，一个用在禁止分配内存时）。

在pyboard中，我们通常是直接用LED对象来控制LED，如：

```
pyb.LED(1).on()
pyb.LED(1).off()
```

但是，pyb中也可以通过寄存器去控制LED，如：

```
import stm

LED1 = 13
stm.mem16[stm.GPIOA+stm.GPIO_BSRR] |= (1<<LED1)
stm.mem16[stm.GPIOA+stm.GPIO_BSRRH] |= (1<<LED1)

LED4 = 4
stm.mem16[stm.GPIOB+stm.GPIO_BSRR] |= (1<<LED4)
stm.mem16[stm.GPIOB+stm.GPIO_BSRRH] |= (1<<LED4)
```

这种方式不比用pyb.LED更好，但是可以通过这种方式控制一些MicroPython尚不支持的外设模块，如看门狗。

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。

- 连接USB线
- 按住USER键，然后按下复位键
- 松开复位键，保持USER键
- 这时LED将循环显示：绿—》黄—》绿+黄—》灭
- 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
- 然后红灯亮起（这时红绿黄灯同时亮）
- 红灯灭，pyboard开始进行恢复到出厂状态
- 所有灯都灭，恢复出厂设置完成。

摘自: <http://www.oschina.net/news/76812/micropython-assembly>

MicroPython 包涵可内联的汇编, 允许用户使用汇编语言作为 Python 的子程序, 且你可以像正常使用函数般使用它们。

1、返回值

内联汇编函数用特定的函数装饰器标示。我们从最简单的例子下手:

```
@micropython.asm_thumb
def fun():
    movw(r0, 42)
```

你可以在脚本或是解释器里边使用该函数。该函数没有任何参数且返回数值42。r0 是一个寄存器, 其中的数值在函数返回值返回时被更改。MicroPython 一直将 r0视为一个整数并将其作为整数变量供使用者调用。

如果使用了命令 `print(fun())` 将能看到数值42被打印出来。

2、汇编语言基础

稍微复杂一些些, 我们尝试点亮一盏灯:

```
@micropython.asm_thumb
def led_on():
    movwt(r0, stm.GPIOA)
    movw(r1, 1 << 13)
    strh(r1, [r0, stm.GPIO_BSRR])
```

上述代码使用了一些新的概念:

. stm 为 pyboard 的微处理器提供了一系列内容以便于连接寄存器。尝试在 REPL 里运行 `import stm` 和 `help(stm)`。这将得到一清单的有用内容:

. stm.GPIOA 对应外围设备GPIOA 在内存中的地址。在 pyboard 板上红色的led 灯对应 A端口, PA13 引脚;

. movwt 将32位数值放入寄存器中。其可视为由两个指令集组成的简便函数：先是 movw 然后 movt 。movt 将16位立即数移动。

. strh 存储半字数据。上述代码里将r1的低16位数值存入 r0 +stm.GPIO_BSRRL 的内存地址中。这将按照 r0 里设定的数值将 A 端口对应的引脚设置为高。例程中r0的第13位值被置位，故PA13 被拉高。因此红色LED 灯被点亮。

3、接受参数

内联汇编语言最多可以接收四个参数。一旦被使用，必须为 r0,r1,r2,r3 的寄存器或其里边的调用内容。以下是使用了这些参数的函数：

```
@micropython.asm_thumb
def asm_add(r0, r1):
    add(r0, r0, r1)
```

这里使用了 r0=r0+r1 的计算。由于将结果放入了 r0 中，故其为返回结果。尝试运行asm(1,2)，将能得到 3 的返回值。

4、循环

我们可以分配 label(my_label)的标号，然后使用 b(my_label) 跳转到该分支，或者用 bgt(my_label)进行有条件的跳转。下面例程使绿色的 LED 灯闪烁，闪烁次数存放在 r0 里边。

```
@micropython.asm_thumb
def flash_led(r0):
    # get the GPIOA address in r1
    movwt(r1, stm.GPIOA)
    # get the bit mask for PA14 (the pin LED #2 is on)
    movw(r2, 1 << 14)
    b(loop_entry)
    label(loop1)
    # turn LED on
    strh(r2, [r1, stm.GPIO_BSRRL])
    # delay for a bit
```

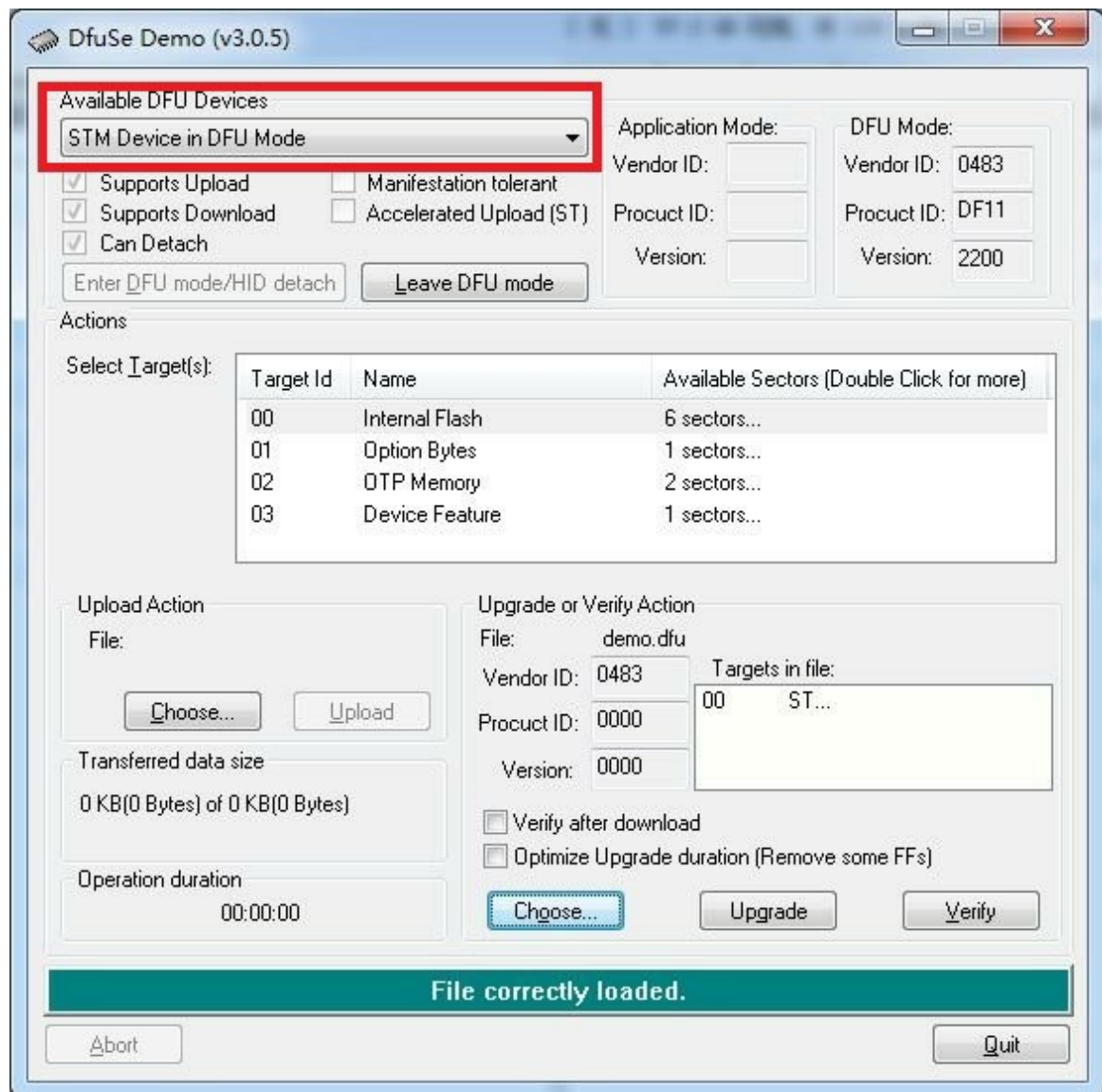


```
movwt(r4, 5599900)
label(delay_on)
sub(r4, r4, 1)
cmp(r4, 0)
bgt(delay_on)
# turn LED off
strh(r2, [r1, stm.GPIO_BSRRH])
# delay for a bit
movwt(r4, 5599900)
label(delay_off)
sub(r4, r4, 1)
cmp(r4, 0)
bgt(delay_off)
# loop r0 times
sub(r0, r0, 1)
label(loop_entry)
cmp(r0, 0)
bgt(loop1)
```

在pyboard上使用MicroPython时，有时因为各种原因可能需要下载固件到开发板，比如原有程序损坏、固件版本升级、DIY新的开发板等。下面就介绍升级的方法。

在pyboard上，除了可以用SWD写入（开发板并没有设计SWD插座，但是都连接到开发板四周的排针上了），更方便的方法是通过USB使用DFU模式烧写固件。

- 首先下载并安装DfuSedemo
<http://www.st.com/web/en/catalog/tools/FM147/CL1794/SC961/>
- 运行DfuSedemo
- pyboard连接USB
- 短路BOOT0和3V3，然后按下RST按键。
- 松开RST，保持BOOT0和GND短路。
- 释放BOOT0
- DfuSeDemo识别成功，红框中显示出连接的设备



- 如果连接不成功，请重复前面步骤。
- 载入下载的DFU固件文件，按下Upgrade按钮就可以升级了。

<https://micropython.org/download/>

•

其他

- 少量单片机需要先短路BOOT0和GND，然后接入USB才能进入DFU模式。
- 如果是第一次进入DFU模式，会提示安装驱动，驱动就在DfuSeDemo的安装目录下

- . 首先需要安装Linux版的arm-gcc编译器
- . 安装dfu-util
`sudo apt-get install dfu-util`
- . 编译固件
- . 用dfu-util写入固件
`sudo make BOARD=XXXX deploy USE_PYDFU=0`
- . 也可以使用pydfu.py下载固件，需要先安装python-usb模块
`sudo apt-get install python-usb python3-usb`



这是针对ESP8266版的快速指南，与PYBOARD版的有很多区别。



machine 和频率控制

```
import machine

machine.freq()           # get the current frequency of the CPU
machine.freq(160000000)  # set the CPU frequency to 160 MHz
```

ESP模块

```
import esp

esp.osdebug(None)        # turn off vendor O/S debugging messages
esp.osdebug(0)           # redirect vendor O/S debugging messages
                           # to UART(0)
```

GPIO

```
from machine import Pin

p0 = Pin(0, Pin.OUT)     # create output pin on GPIO0
p0.high()                # set pin to high
p0.low()                 # set pin to low
p0.value(1)              # set pin to high

p2 = Pin(2, Pin.IN)      # create input pin on GPIO2
print(p2.value())        # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up
resistor
```

```
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

定时器

```
from machine import Timer

tim = Timer(-1)
tim.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))
```

延时

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get millisecond counter
delta = time.ticks_diff(start, time.ticks_ms()) # compute time
                        # difference
```

PWM

```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))      # create PWM object from a pin
pwm0.freq()             # get current frequency
pwm0.freq(1000)         # set frequency
pwm0.duty()             # get current duty cycle
pwm0.duty(200)          # set duty cycle
pwm0.deinit()           # turn off PWM on the pin

pwm2 = PWM(Pin(2), freq=500, duty=512) # create and configure in
one go
```

ADC

```
from machine import ADC
```

```
adc = ADC(0)           # create ADC object on ADC pin
adc.read()             # read value, 0-1024
```

SPI

```
from machine import Pin, SPI

# construct an SPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means
the second
spi = SPI(baudrate=100000, polarity=1, phase=0, sck=Pin(0),
mosi=Pin(2), miso=Pin(4))

spi.init(baudrate=200000) # set the baudrate

spi.read(10)             # read 10 bytes on MISO
spi.read(10, 0xff)      # read 10 bytes while outputting 0xff on
MOSI

buf = bytearray(50)      # create a buffer
spi.readinto(buf)        # read into the given buffer (reads 50
bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output
0xff on MOSI

spi.write(b'12345')      # write 5 bytes on MOSI

buf = bytearray(4)       # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from
MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO
back into buf
```

I2C

```
from machine import Pin, I2C

# construct an I2C bus
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)

i2c.readfrom(0x3a, 4)    # read 4 bytes from slave device with
address 0x3a
i2c.writeto(0x3a, '12') # write '12' to slave device with
address 0x3a
```



```
buf = bytearray(10)      # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)   # write the given buffer to the slave
```

休眠

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# check if the device woke from a deep sleep
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# set RTC.ALARM0 to fire after 10 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 10000)

# put the device to sleep
machine.deepsleep()
```

onewire总线

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # create a OneWire bus on GPIO12
ow.scan()                    # return a list of devices on the bus
ow.reset()                   # reset the bus
ow.readbyte()                # read a byte
ow.read(5)                   # read 5 bytes
ow.writebyte(0x12)           # write a byte on the bus
ow.write('123')              # write bytes on the bus
ow.select_rom(b'12345678')   # select a specific device by its ROM
                              # code
```

驱动DS18B20

```
import time
ds = onewire.DS18B20(ow)
roms = ds.scan()
ds.convert_temp()
```

```
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

网络

```
import network

wlan = network.WLAN(network.STA_IF) # create station interface
wlan.active(True)                   # activate the interface
wlan.scan()                          # scan for access points
wlan.isconnected()                  # check if the station is connected to
an AP
wlan.connect('ssid', 'password') # connect to an AP
wlan.config('mac')                 # get the interface's MAC address
wlan.ifconfig()                     # get the interface's IP/netmask/gw/DNS
addresses

ap = network.WLAN(network.AP_IF) # create access-point interface
ap.active(True)                   # activate the interface
ap.config(essid='ESP-AP') # set the ESSID of the access point
```

```
def do_connect():
    import network
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('ssid', 'password')
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ifconfig())
```

NeoPixel 驱动

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT) # set GPIO0 to output to drive NeoPixels
np = NeoPixel(pin, 8) # create NeoPixel driver on GPIO0 for 8
pixels
np[0] = (255, 255, 255) # set the first pixel to white
np.write()               # write data to all pixels
r, g, b = np[0]          # get first pixel colour
```

底层驱动

```
import esp
esp.neopixel_write(pin, grb_buf, is800khz)
```

APA102驱动

```
from machine import Pin
from apa102 import APA102

clock = Pin(14, Pin.OUT)      # set GPIO14 to output to drive the
clock                                  clock
data = Pin(13, Pin.OUT)       # set GPIO13 to output to drive the
data                                  data
apa = APA102(clock, data, 8)   # create APA102 driver on the clock
and the data pin for 8 pixels
apa[0] = (255, 255, 255, 31)  # set the first pixel to white with
a maximum brightness of 31
apa.write()                   # write data to all pixels
r, g, b, brightness = apa[0] # get first pixel colour
```

底层驱动

```
import esp
esp.apa102_write(clock_pin, data_pin, rgb_buf)
```

webrepl

```
import webrepl
webrepl.start()
webrepl.stop()
```

Pin的用法

在ESP8266中，Pin的用法和在pyboard（STM32）中有很大的区别。

定义Pin

- `class machine.Pin(id, ...)`
id的范围是： [0, 1, 2, 3, 4, 5, 12, 13, 14, 15, 16]，其中16只能做DIO，不支持PWM和中断。[9, 10]是ESP - 12E以后才提供的，但是GPIO9不能做通用IO，GPIO10和GPIO16类似。

方法

- `Pin.init(mode, pull=None, *, value)`
初始化
mode:
 - `Pin.IN`，输入
 - `Pin.OUT`，输出pull:
 - `NONE`，无
 - `Pin.PULL_UP`，上拉value: 输出电平
- `Pin.value([value])`
不带参数时是读取输入电平，带参数时是设置输出电平。参数可以是True/False，也可以是1/0。
- `Pin.low()`
- `Pin.high()`
设置输出电平
- `Pin.irq(*, trigger, handler=None)`
中断
trigger, 触发方式
 - `Pin.IRQ_FALLING`，下降沿
 - `Pin.IRQ_RISING`，上升沿
 - `Pin.IN`，上升下降沿handler, 回调函数

常数

下面的常数用于配置 pin。注意不是没股份端口都有全部的属性。

- Pin.IN
 - Pin.OUT
 - Pin.OPEN_DRAIN
 - Pin.ALT
 - Pin.ALT_OPEN_DRAIN
选择 pin 模式
 - Pin.PULL_UP
 - Pin.PULL_DOWN
设置上拉/下拉电阻。
 - Pin.LOW_POWER
 - Pin.MED_POWER
 - Pin.HIGH_POWER
设置驱动能力
 - Pin.IRQ_FALLING
 - Pin.IRQ_RISING
 - Pin.IRQ_LOW_LEVEL
 - Pin.IRQ_HIGH_LEVEL
设置 IRQ 触发类型。
-

例子

```
from machine import Pin

CS = Pin(2, Pin.OUT)
CS(1)
CS(0)
CS.value()
CS.value(1)
CS.high()
CS.low()

sw = Pin(0, Pin.IN)
sw()
sw.irq(trigger=Pin.IRQ_FALLING, handler=lambda t:led.value(not
```

```
led.value()))
```

class ADC - 模数转换

ESP8266只有一个专用的ADC输入端口，官方文档中的class ADCChannel是WiPY中的，在ESP8266上没有。

用法：

```
import machine

adc = machine.ADC(0)          # create an ADC object
val = adc.read()             # read an analog value
```

构造函数

- class machine.ADC(id=0)
创建 ADC 对象。ADC 的输入电压范围 0~1V，参数范围：0~1024

方法

- ADC.read()
读取ADC结果。

class I2C

I2C 是设备之间的两线通信协议。在物理层它只需要两个信号线：SCL 和 SDA，分别是时钟和数据线。

I2C 对象关联到总线，它可以创建时初始化，也可以稍后在初始化。

构造函数

- `class machine.I2C(scl, sda, *, freq=400000)`
创建并返回新的 I2C 对象，参数部分可以参考下面的初始化函数。

方法

- `I2C.init(scl, sda, *, freq=400000)`

初始化 I2C:

- `scl` 代表 SCL 引脚
- `sda` 代表 SDA 引脚
- `freq` 代表 SCL 时钟速率

注：实际上第一个参数默认是SDA，或者用scl强制指定Pin。

- `I2C.scan()`
扫描 0x08 到 0x77 之间的 I2C 地址，并返回设备列表。如果收到地址后设备拉低 SDA（包括读取位）代表设备有响应。

原始 I2C 操作

下面方法执行原始的 I2C 主操作，它们可以组合产生各种 I2C 事务。提供这些方法是可以更好的控制总线，否则使用标准方法就足够。

- `I2C.start()`
在总线上发送 start 位（当 SCL 是高时 SDA 变低）。
- `I2C.stop()`

发送停止位（当 SCL 是高时 SCL 变高）。

- `I2C.readinto(buf)`
从总线上读取数据并存放到 `buf`，读取的数量是 `buf` 的长度。接收到倒数第二个数据时发送 ACK 信号，接收到全部数据后发送 NACK 信号。
- `I2C.write(buf)`
写入缓冲区数据到总线。每发送一个字节后将检查是否收到 ACK，如果没有收到将引发 `OSError` 异常。

标准总线操作

下面方法是标准的 I2C 主模式读写操作。

- `I2C.readfrom(addr, nbytes)`
从指定地址设备读取数据，返回读取对象。
- `I2C.readfrom_into(addr, buf)`
从指定地址设备读取数据到缓冲区，读取的数量是 `buf` 的长度。

只有 WiPy 返回读取数据的数量，其他模块都返回 `None`。

- `I2C.writeto(addr, buf, *, stop=True)`
写入数据到设备。`stop` 参数（仅 WiPy）代表发送完成后需要再发送停止位。

只有 WiPy 返回读取数据的数量，其他模块都返回 `None`。

内存操作

某些 I2C 设备作为存储设备（或一组寄存器），可以读取或者写入。这种情况下，有两个地址和 I2C 事务相关：从设备地址和内存地址。下面方法用于和这些设备进行通信。

- `I2C.readfrom_mem(addr, memaddr, nbytes, *, addrsize=8)`
读取从设备 `addr` 的内存地址 `memaddr` 数据，`addrsize` 指定地址大小（在 ESP8266 上这个参数无效，地址大小总是 8 位），返回读取数据的字节对象。

- `I2C.readfrom_mem_into(addr, memaddr, buf, *, addrsize=8)`
读取从设备 `addr` 的内存地址 `memaddr` 数据到缓冲区，`addrsize` 指定地址大小（在 ESP8266 上这个参数无效，地址大小总是 8 位），读取数据数量是 `buf` 的长度。

在 WiPy 上返回值是读取数据数量，其他模块都返回 `None`。

- `I2C.writeto_mem(addr, memaddr, buf, *, addrsize=8)`
写入 `buf` 到从设备 `addr` 的内存 `memaddr`，`addrsize` 代表地址大小（在 ESP8266 上这个参数无效，地址大小总是 8 位）。

在 WiPy 上返回值是读取数据数量，其他模块都返回 `None`。

参考例子：

```
from machine import Pin, I2C

i2c = I2C(Pin(14), Pin(2))
i2c = I2C(scl = Pin(2), sda = Pin(14), freq = 100000)
```

ESP8266的库和Pyb的库差异较大，主要在参数上，如参数个数、参数位置等。从这里可以看出两个库不是同一个人开发的。

pyb中没有I2C.start()、I2C.stop()等函数

pyb中的I2C.send()函数，在ESP8266中对应的是I2C.writeto()函数。

在I2C.send()中，参数在前，地址在后；I2C.writeto()函数正好相反。I2C.writeto()函数只能用bytearray变量做参数。

class SPI - 主模式串行通信协议

主模式 SPI串口通信，在物理层需要3根信号线：SCK, MOSI, MISO.

构造函数

方法

- `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` 代表 SCK 时钟频率.

`polarity` 是 0 或 1, 代表空闲状态时钟电平.

`phase` 是 0 或 1, 代表数据在第一或第二时钟沿进行采样.

`bits` 是数据传输宽度, 可以是 8, 16 或 32.

`firstbit` 只能是 `SPI.MSB`.

`pins` 是 SPI 相关的 GPIO.

- `SPI.deinit()`
关闭 SPI 总线.
- `SPI.write(buf)`
写缓冲区数据到总线, 返回写入的数据数量。
- `SPI.read(nbytes, *, write=0x00)`
写入的同时读取数据, 返回读取数量。
- `SPI.readinto(buf, *, write=0x00)`
读取数据到缓冲区, 返回读取数量。
- `SPI.write_readinto(write_buf, read_buf)`
写入 `write_buf` 并读取到 `read_buf`, 两个缓冲区长度相同, 返回写入数量。

常数

- `SPI.MASTER`

初始化 SPI 为主模式

- SPI. MSB
设置最高位优先

class RTC - 实时时钟

RTC 模块提供了时钟、日期和时间功能。

使用方法：

```
from machine import RTC

rtc = RTC()
rtc.datetime()
rtc.datetime((2016, 8, 1, 1, 12, 10, 0, 0))
```

构造函数

class machine.RTC()
创建 RTC 对象。

方法

- RTC.datetime(datetime)
设置或读取时间，时间的格式是 (year, month, day, weekday, hours, minutes, seconds, microsecond)
- RTC.alarm(id=RTC.ALARM0, time)
设置RTC唤醒的时间。
id目前只能是RTC.ALARM0，或者就是0。
time是超时时间 (ms)
- RTC.irq(*, trigger=RTC.ALARM0, wake=machine.DEEPSLEEP)
设置RTC唤醒方式，默认是RTC.ALARM0模块和 machine.DEEPSLEEP。
- RTC.memory
未知

常数

- RTC.ALARM0
irq 触发源

RTC唤醒例程，需要连接GPIO16到RST（旧版本的固件需要手工复位）：

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# check if the device woke from a deep sleep
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# set RTC.ALARM0 to fire after 2 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 2000)

# put the device to sleep
machine.deepsleep()
```

class Timer - 控制定时器

注:

在中断处理函数中不能分配内存，在中断异常里不会给出更多提示信息。查看 `micropython.alloc_emergency_exception_buf()` 获取如何绕过这个限制。

构造函数

- `class machine.Timer(id, ...)`

方法

- `Timer.deinit()`
停止定时器，禁止所有通道和相关中断。
- `Timer.init(period, mode, callback)`
设置定时器参数
 - `period`，定时器间隔时间（ms）
 - `mode`，定时器模式
`Timer.ONE_SHOT`，一次性
`Timer.PERIODIC`，周期
 - `callback`，回调函数

常数

- `Timer.ONE_SHOT`
- `Timer.PERIODIC`

例程:

```
from machine import Timer

t0 = Timer(-1)
t0.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(1))
```

和PYBOARD不同，ESP8266有一个独立的PWM模块（感觉像软件PWM，并不是硬件PWM，Timer也是一样），频率的范围是1-1000Hz，占空比是10位的，范围是0-1023。

因为只有一个PWM模块，所以对于所有通道，频率是相同的。

不是所有的GPIO都支持PWM功能，PWM支持GPIO有 [0, 2, 4, 5, 12, 13, 14, 15]

PWM的基本用法是：

构造函数

- `class PWM(pin, ...)`
定义一个GPIO为PWM模式，pin是GPIO。

方法

- `PWM.init(freq, duty)`
freq, PWM输出频率，设置频率会影响所有通道
duty, 占空比
- `PWM.freq(freq)`
freq, 设置PWM输出频率，范围是1-1000。
- `PWM.duty(duty)`
duty, 输出占空比。范围是0-1023。
- `PWM.deinit()`
禁用PWM。

基本用法

```
from machine import PWM, Pin

led = PWM(Pin(2), freq=100)
led.duty(800)
```

class UART - 全双工串行通信总线

UART 执行标准 UART/USART 双工串行通信协议。物理层上需要两根线：RX 和 TX。通信数据单位是字符（不是字符串字符），可以是 8 或 9 bit。

UART 参考用法：

```
from machine import UART

uart = UART(1, 9600) # init with given
baudrate
uart.init(9600, bits=8, parity=None, stop=1) # init with given
parameters
```

UART 对象的使用类似 stream 对象，可以使用标准 stream 的方法进行读写：

```
uart.read(10) # read 10 characters, returns a bytes object
uart.readall() # read all available characters
uart.readline() # read a line
uart.readinto(buf) # read and store into the given buffer
uart.write('abc') # write the 3 characters
```

构造函数

- class machine.UART(id, baudrate=115200, bits=8, parity=None, stop=1, timeout=0, timeout_char=1)
 - id, 串口号
串口号从0开始，目前支持0和1。
串口1只有发送，没有接收功能。UART1_TXD使用GPIO2。
 - baudrate, 波特率
 - bits, 数据位
 - parity, 奇偶校验
 - stop, 停止位数量
 - timeout, 超时时间

方法

- `UART.read([nbytes])`
读取字符。如果指定 `nbytes`，那么最多读取 `nbytes` 字节。

返回参数：包含读取数据的 `bytes` 对象，超时返回 `None`。
- `UART.readall()`
读取全部数据

返回参数：包含读取数据的 `bytes` 对象，超时返回 `None`。
- `UART.readinto(buf[, nbytes])`
读取数据到缓冲区。如果指定 `nbytes`，那么最多读取 `nbytes` 字节，否则最多读取 `len(buf)` 字节。

返回值：读取字节数，超时返回 `None`。
- `UART.readline()`
读取一行，以换行符结束。

返回读取的数据，超时返回 `None`。
- `UART.write(buf)`
写缓存数据

返回写入数据的数量，超时返回 `None`。

ESP8266版的标准库

- [array](#) - 数组
- [gc](#) - 垃圾回收
- [math](#) - 数学函数
- `sys` - 系统函数
- `ubinascii` - binary/ASCII 转换
- `ucollections` - collection and container types
- `uhashlib` - hashing algorithm
- `uheapq` - heap queue algorithm
- `uio` - input/output streams
- `ujson` - JSON encoding and decoding
- `uos` - basic “operating system” services
- `ure` - regular expressions
- `usocket` - socket module
- `ussl` - ssl module
- `ustruct` - pack and unpack primitive data types
- `utime` - time related functions
- `uzlib` - zlib decompression

array - 数据数组

更多请参考 [Python array](#)。

支持格式代码: b, B, h, H, i, I, l, L, q, Q, f, d (最后两个依赖于浮点数的支持)。

Classes

- `class array.array(typecode[, iterable])`
创建指定类型的数组。数组初始内容通过 `iterable` 设置。如果没有指定 `iterable`, 那么将创建空数组。

方法

- `append(val)`
附加新元素到数组末尾。
- `extend(iterable)`
附加包含 `iterable` 的新元素到数组末尾。

和pybaord的gc库相同。

[参考pybaord的math库](#)。

大部分标准库和PYB的是相同的，下面就不再列出。

machine — 板级库函数

machine 模块包含了和开发板相关的库函数。

复位函数

- `machine.reset()`
复位系统，和按下复位键效果一样。
- `machine.reset_cause()`
获取复位原因。参考下面的常数值。

中断函数

- `machine.disable_irq()`
禁止中断。返回先前中断允许状态，它将用于函数 `enable_irq()` 恢复初始中断状态。
- `machine.enable_irq(state)`
允许中断。`state` 参数通常是 `disable_irq()` 函数返回值。

功率控制函数

- `machine.freq()`
返回 CPU 频率(Hz)。
- `machine.idle()`
禁止 CPU 的时钟，用于减少一段时间的功耗。外设将继续运行，程序在任何中断发生后将恢复运行（在很多移植版本中包括周期触发的系统时钟中断）。
- `machine.sleep()`
停止 CPU 并禁用除了 WLAN 外所有的外设。唤醒后从休眠处继续执行，休眠前需要先配置唤醒方式。
- `machine.deepsleep()`
停止 CPU 和所有外设（包括网络）。唤醒后从 `main.py` 开始执行，类似于复位。可以通过检查复位原因是否为

`machine.DEEPSLEEP_RESET`。为了确保可以被唤醒，需要先配置好唤醒方式，如引脚电平变化或者 RTC 超时。

其它函数

- `machine.unique_id()`
返回唯一序号字符串。如果底层硬件支持这个功能，每个芯片的 ID 都不相同。ID 的长度由硬件决定（因此如果需要短 ID 可以使用子字符串）。在某些 MicroPython 移植版中，ID 是网络 MAC 地址。
- `machine.time_pulse_us(pin, pulse_level, timeout_us=1000000)`
测量指定端口上脉冲信号，返回脉冲持续时间。`pulse_level` 参数代表脉冲的电平。

如果端口输入电平不等于 `pulse_level`，函数将先等待，当电平相同时开始测量持续时间。如果端口输入电平等于 `pulse_level`，那么立即开始测量。

当等待时间或者持续时间超过 `timeout_us`，将产生一个 `OSError: ETIMEDOUT` 异常。

常数

中断唤醒常数（**目前仅支持**`machine.DEEPSLEEP`）

- `machine.IDLE`
- `machine.SLEEP`
- `machine.DEEPSLEEP`

复位常数

- `machine.PWRON_RESET`
- `machine.HARD_RESET`
- `machine.WDT_RESET`
- `machine.DEEPSLEEP_RESET`
- `machine.SOFT_RESET`

唤醒常数（ESP8266目前不支持）

- machine.WLAN_WAKE
- machine.PIN_WAKE
- machine.RTC_WAKE

Classes

class ADC - analog to digital conversion
class ADCChannel - read analog values from internal or external sources
class I2C - a two-wire serial protocol
class Pin - control I/O pins
class RTC - real time clock
class SD - secure digital memory card
class SPI - a master-driven serial protocol
class Timer - control internal timers
class TimerChannel - setup a channel for a timer
class UART - duplex serial communication bus
class WDT - watchdog timer

ESP8266没有USB，文件传输只能通过wifi或者串口，不如pyb方便。

webrepl是MicroPython官方提供的文件管理工具。micropython还专门提供了一个webrepl工具，使我们可以通过浏览器来访问ESP8266。

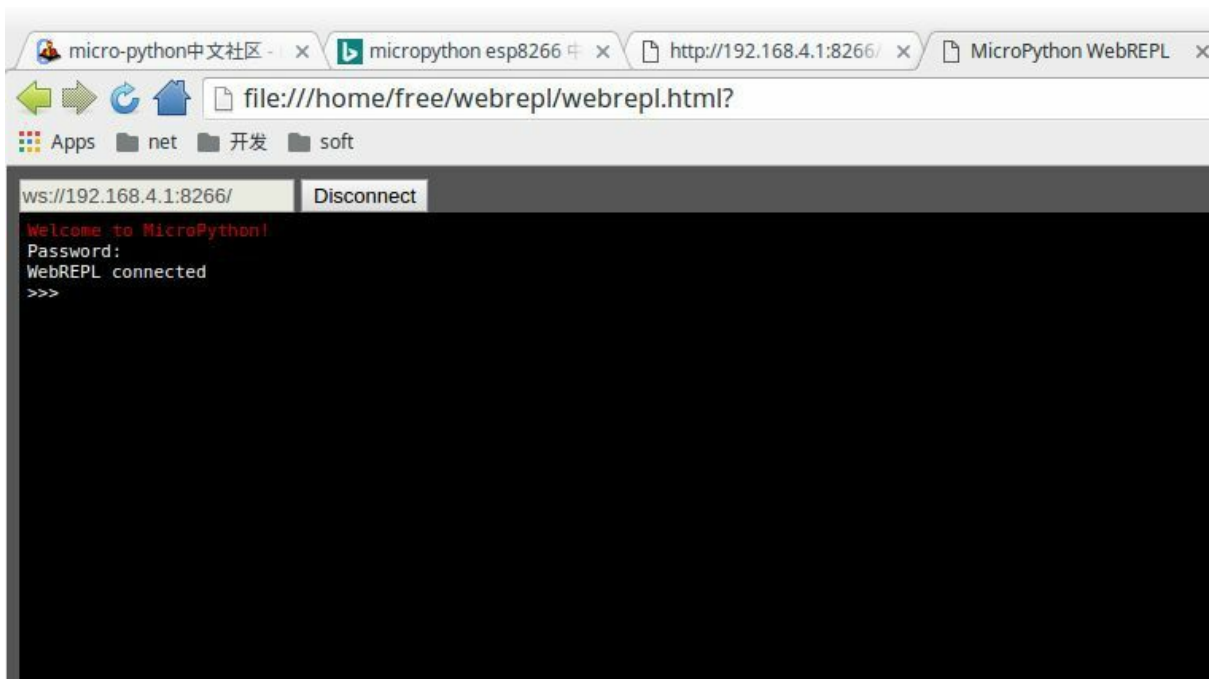
webrepl，从名称看，就是用web方式使用repl的功能。因为官网的介绍非常简单，只有几句话，而且还分散到几个部分，让我们不太容易掌握。这里将它的使用方法总结出来，方便大家使用。它的使用方法如下：

- 首先通过串口方式连接ESP8266
- 在串口端，发送命令，启动webrepl（可以在boot.py中打开这个功能，默认是关闭的）。
`import webrepl`
`webrepl.start()`
- 网络连接有两种方式，一种是连接ESP8266热点，一种是通过路由器连接。
 - 计算机连接到ESP8266的热点，micropython-xxxxxx（后面的代号每个模块都是不同的）。连接热点的密码是：**micropythoN**（注意最后的N是大写）
 - 通过路由器连接，需要先配置网络。配置网络只需一次，下一次会自动连接，和计算机连接路由器一样

```
import network
w=network.wlan(network.STA_IF)
w.scan()
w.connect('SSID', 'PASSWROD') #SSID和PASSWORD需要填写实际值
```

- 下载webrepl: <https://github.com/micropython/webrepl>
- 在Chrome或者Firefox浏览器（不支持IE）中，打开webrepl目录中的webrepl.html文件。
- 在浏览器中，热点方式时一般不需要修改IP（192.168.4.1），直接连接；路由器方式需要修改为实际IP（如192.168.1.xxx）。第一次连接后需要设置密码（密码需要3位以上），以后就需要用这个密码登录了。设置后，ESP8266会重新启动，因此浏览器这里需要再次连接。
- 连接ESP8266，并在浏览器中用webrepl连接，使用设置的密码登录。
- 这时浏览器就会显示一个终端界面，可以输入各种命令和代码了，和一般的串口终端一样。
- 如果刷新页面，就需要重新登录才能继续使用
- 可以使用webrepl下的webrepl_cli.py下载或者上传文件，这也是目前最方便的方法。

- 查看帮助
`webrepl_cli.py --help`
- 基本用法
`webrepl_cli.py <host>:<remote_file> <local_file> -`
Copy remote file to local file
`webrepl_cli.py <local_file> <host>:<remote_file> -`
Copy local file to remote file
- 上传文件
`webrepl_cli.py main.py 192.168.4.1:/`
`webrepl_cli.py user.py 192.168.4.1:/app/`
- 下载文件到D盘
`webrepl_cli.py 192.168.1.110:/boot.py d:/`



uPyLoader需要安装python3、Qt5、PyQt5三个软件才能运行。

在Windows下，可以下载并安装套装，一次搞定。

<https://sourceforge.net/projects/pyqt/files/PyQt5/PyQt-5.6/>

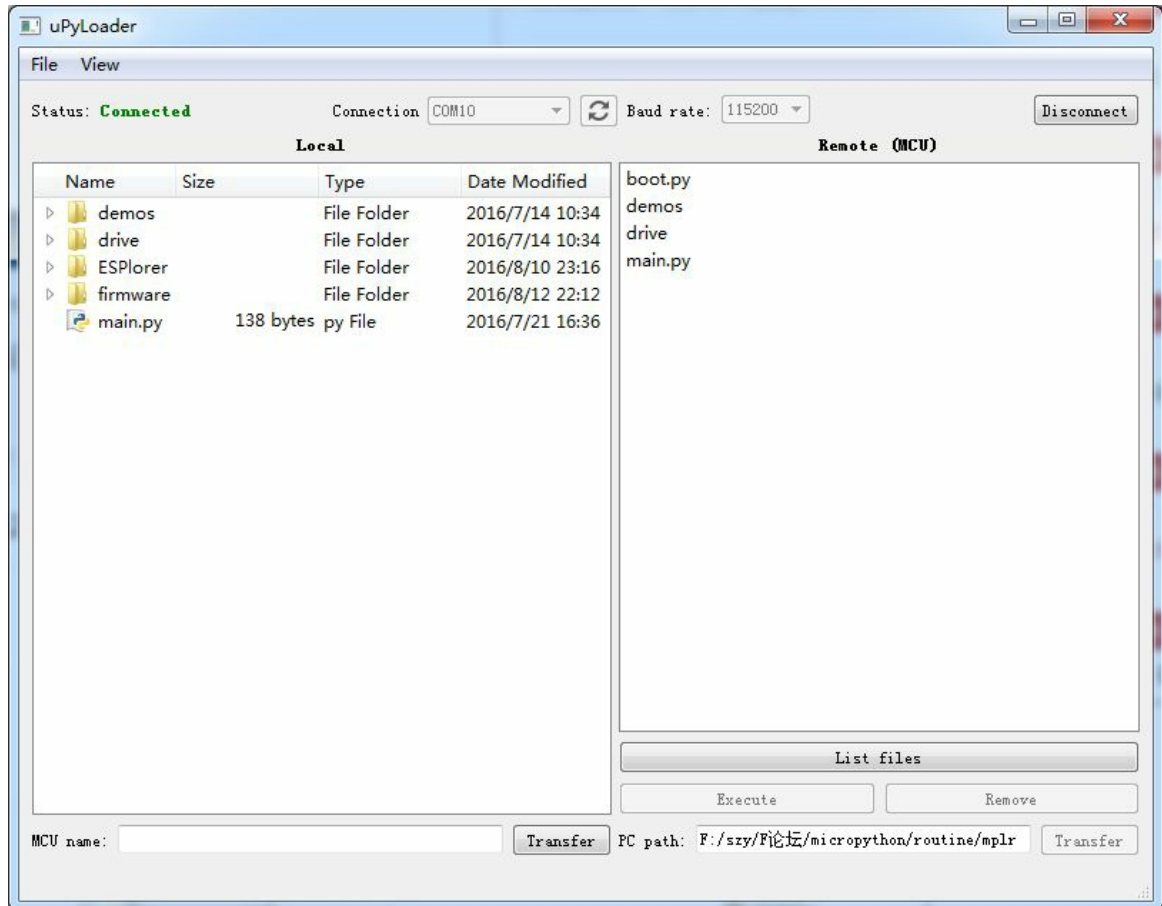
在Linux下，可以用apt-get分别安装。

安装好这几个文件后，就可以在uPyLoader的目录下运行了。

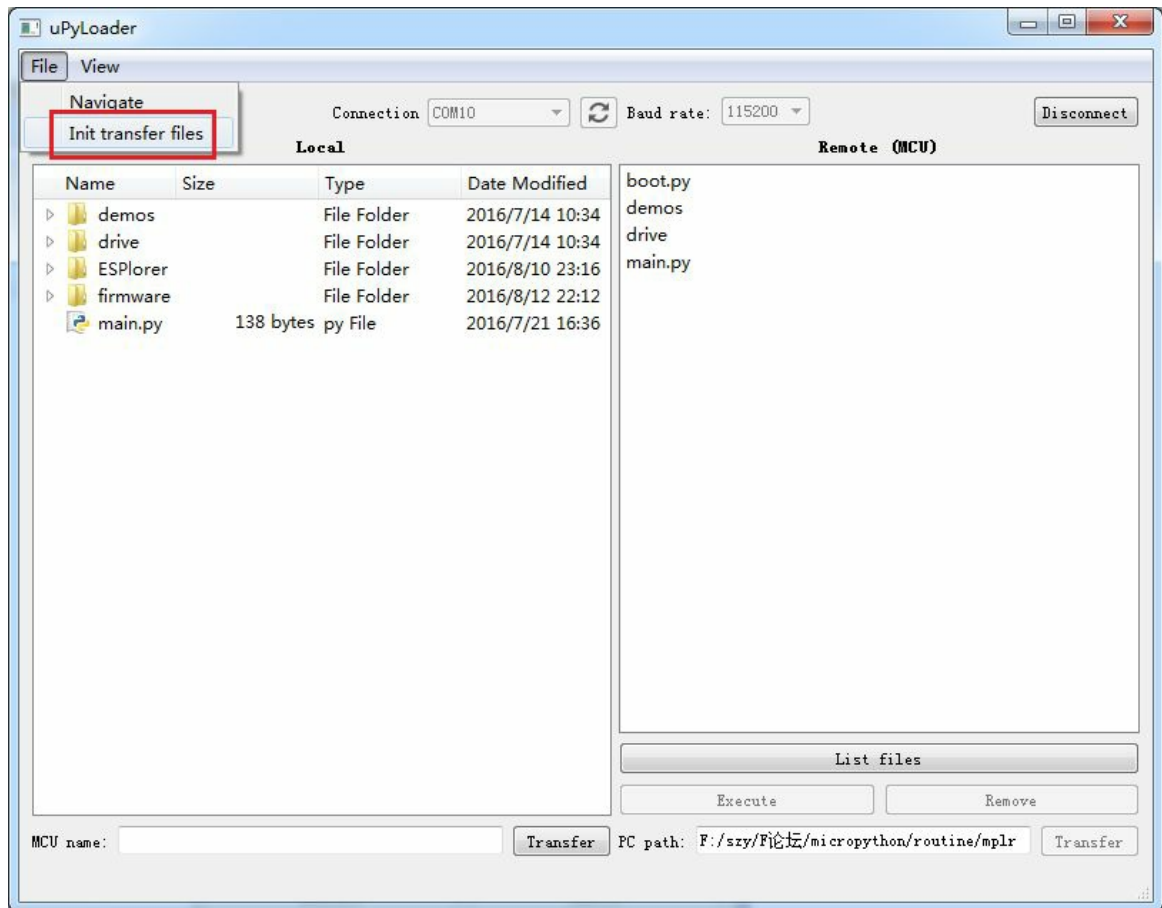
```
python3 main.py
```

下面介绍ESP8266的文件传输工具软件uPyLoader。

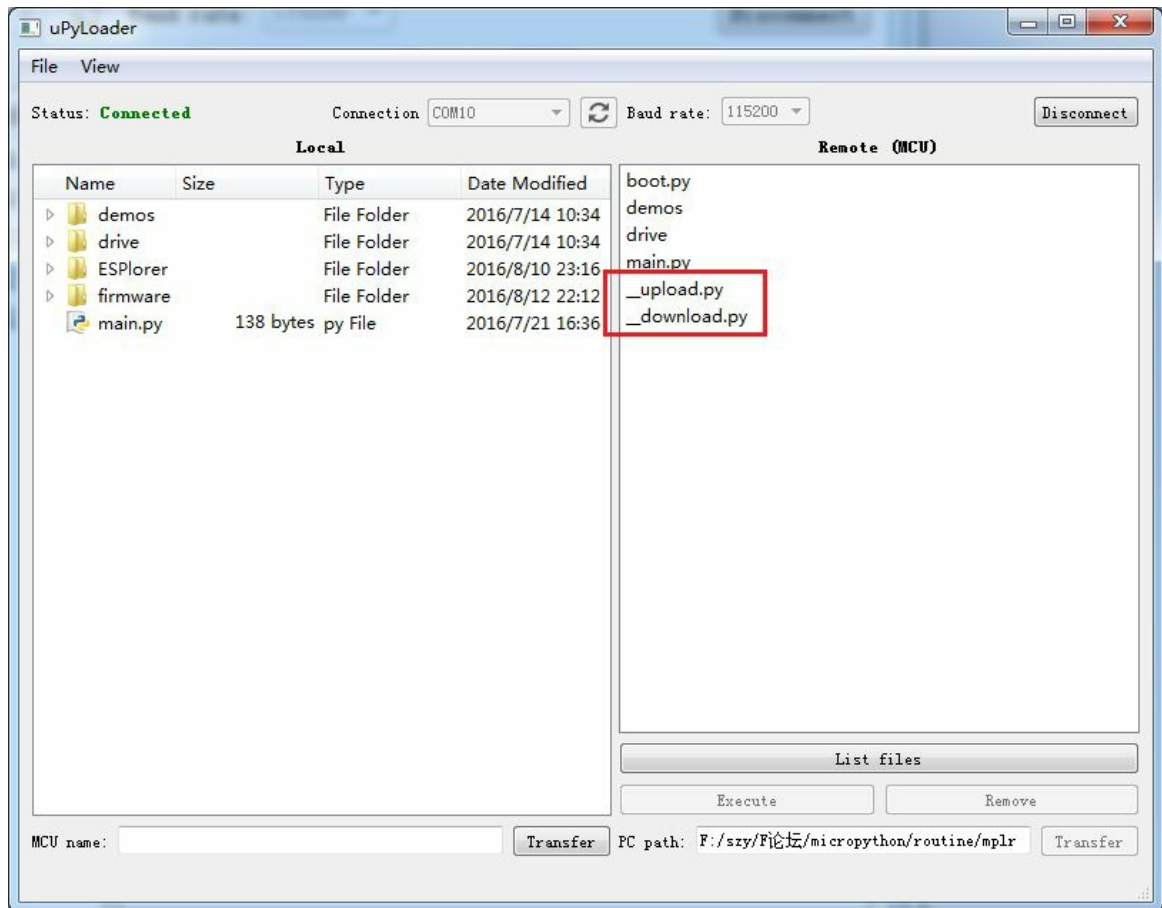
- 首先连接ESP8266到串口
- 运行uPyLoader。
- 连接到ESP8266模块的串口上，这时会自动列出ESP8266模块上的文件。



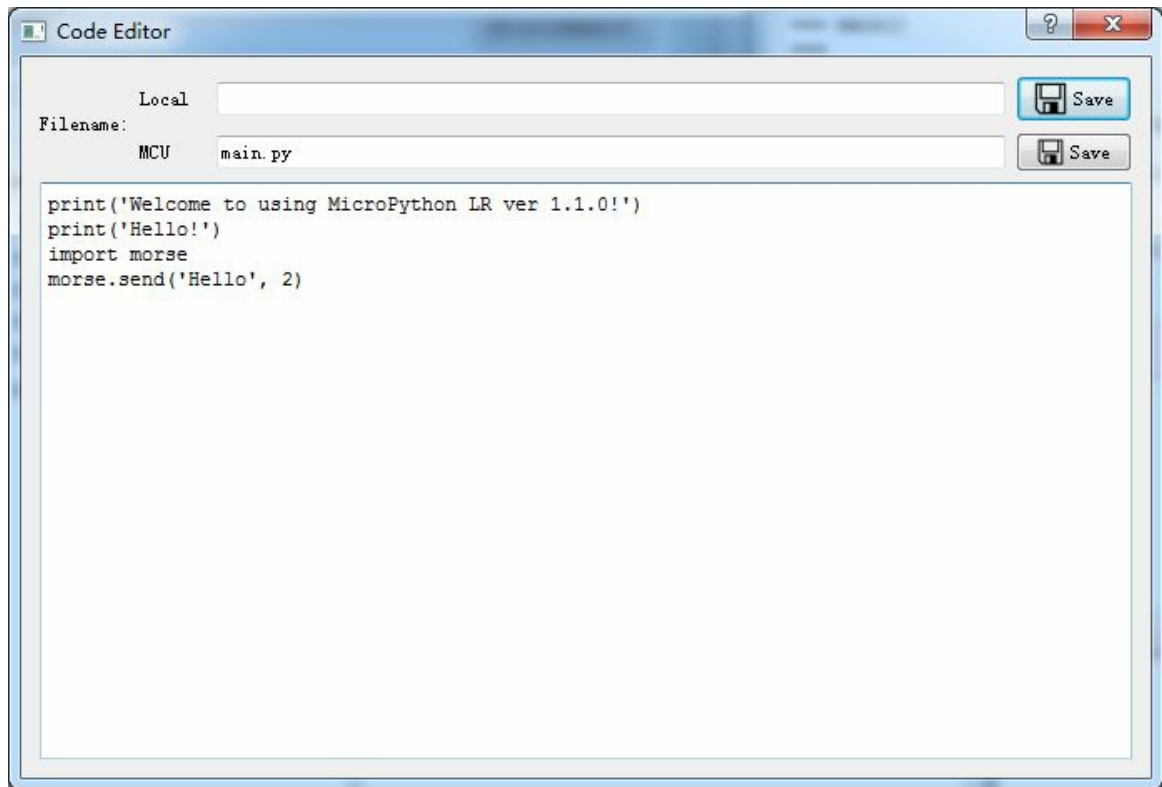
- 使用uPyLoader其他前，需要先初始化ESP8266模块，将两个必要的文件传输过去，才能实现后续的功能。如果已经初始化过，就可以跳过这一步。



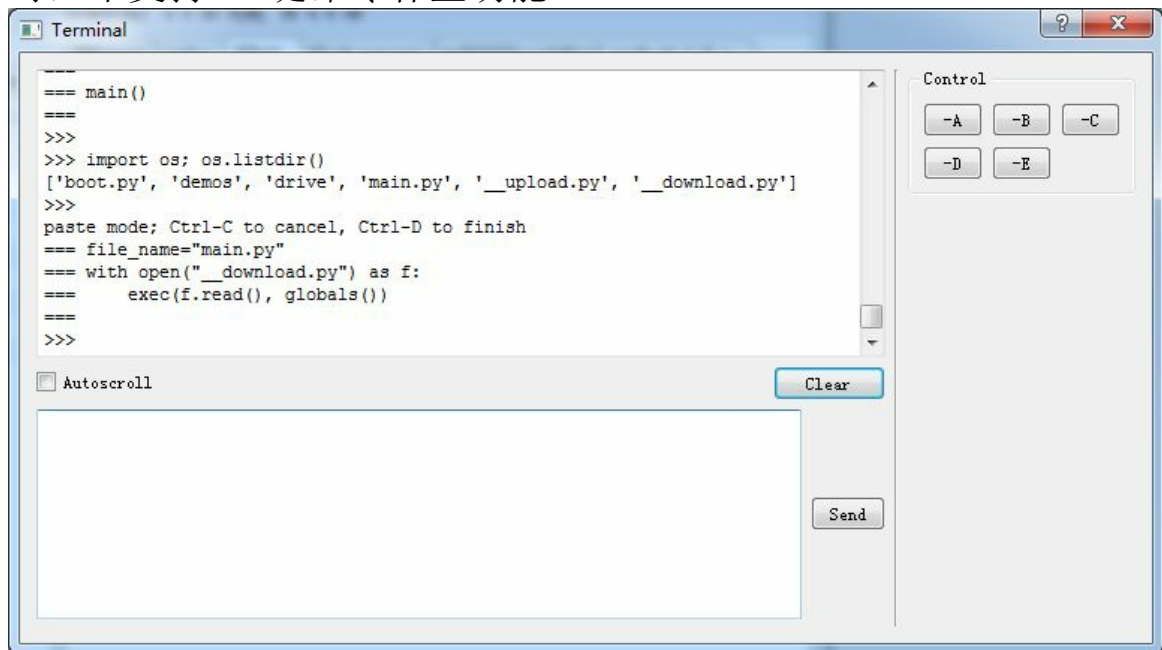
- 初始化后，文件列表中会多出两个文件：



- 这时就可以进行其他操作了，如双击一个文件就是打开Code Editor（代码编辑），可以编辑计算机上的文件或者ESP8266上的文件，编辑后可以保存到任意位置。也可以在主界面上将文件进行复制操作。



- 还可以打开终端窗口，查看命令。但是在这个终端窗口输入命令时，不支持Tab键命令补全功能。



uPyLoader还支持Wifi方式，使用方法和串口方式类似。

uPyLoader常见问题

- 不能运行uPyLoader
uPyLoader需要python3，python2不能运行。
还需要安装pyserial。
- 初始化失败
如果在进行初始化步骤时失败，通常是因为没有在uPyLoder目录下启动造成的，找不到需要传输的文件。在uPyLoader目录下运行，或者通过IDLE运行，就没有这个问题。
- 文件传输失败
首先请检查MicroPython的版本，一定要大于 1.8.3。
其次检查USB电压是否稳定，电源输出电压是否大于4.5V以上
CH340的驱动是否升级到最新版本

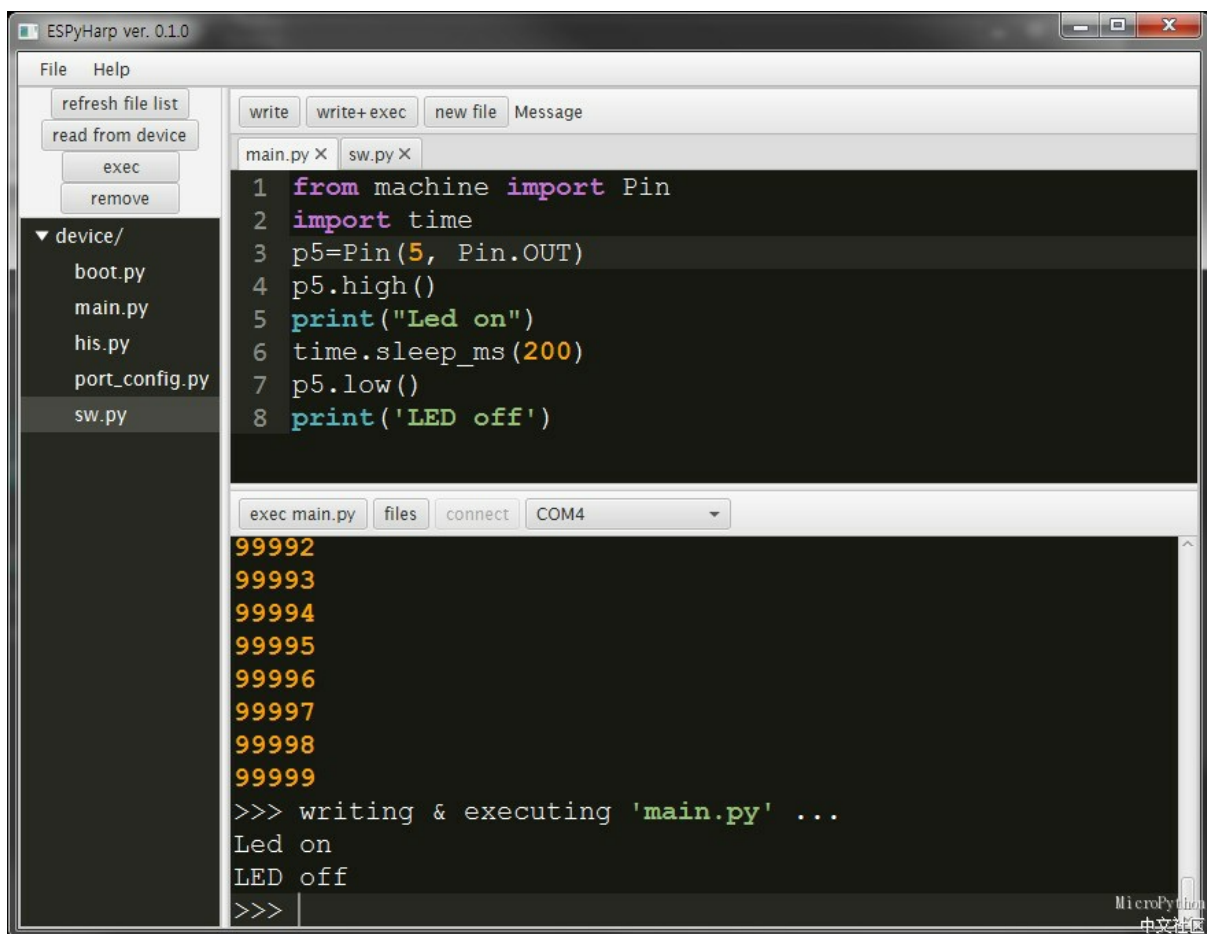
ESPyHarp是韩国人写的一个简单的MicroPython IDE程序。它可以在ESP8266模块 上（如NodeMCU）。

软件的网址是：<http://studymake.tistory.com/584>

代码在

github：<https://github.com/salesiopark/ESPyHarp>

软件下载：https://github.com/salesiopark/ESPyHarp_v010.zip



特点

- 嵌入式REPL
- 文件列表中的装置
- 与Python关键字和内建高亮度代码编辑器
- 上传和/或执行python脚本

系统需求

- Java运行环境

注

- 软件目前有bug，退出软件后，进程没有完全退出，造成串口被占用，不能被其它软件使用。这时需要通过任务管理器将javaw进程关掉才行。
- 文件列表有时会出现一个文件显示两次。

这是一个比webrepl功能更强的shell，支持串口和wifi，可以像Linux的shell那样管理文件。

主要特点：

- Support for serial connections (ESP8266 and WiPi)
- Support for websockets (via WebREPL) connections (ESP8266 only)
- Support for telnet connections (WiPy only)
- Full directory handling (enter, create, remove)
- Transfer (upload/download) of multiple files matching a reg.-exp.
- All files are transferred in binary mode, and thus it should be possible to also upload pre-compiled code (.mpy) too.
- Integrated REPL (supporting a workflow like: upload changed files, enter REPL, test, exit REPL, upload ...)
- Fully scriptable
- Tab-completion
- Command history
- Best of all: it comes with color

它也是一个开源项目，网址在：

<https://github.com/wendlers/mpfshell>

```
** Micropython File Shell v0.6, 2016 sw@kaltpost.de **

mpfs [/]> open ws:192.168.1.111
webrepl passwd:
Connected to esp8266
mpfs [/]> ls

Remote files in '/':

<dir> test
    boot.py
    port_config.py

mpfs [/]> repl
>
*** Exit REPL with Ctrl+] ***

MicroPython v1.8.1-8-gb1533c4 on 2016-06-06; ESP module with ESP8266
Type "help()" for more information.
>>> print("Hello World")
Hello World
>>> 
```

在使用ESP8266模块时，通常会用到一些刷机软件。官方提供了nodemcu_flasher、ESPFlashDownloadTool、ESP8266Flasher等下载工具，但是缺少更底层的工具，不能进行读取、擦写、校验等功能。

Esptool.py是一个python开发的针对ESP8266的小工具，可以实现底层的操作，弥补ESP8266官方工具的不足。它也是一个开源项目，项目在github上进行托管：

<https://github.com/themadinventor/esptool>

虽然可以直接从github上下载使用，但是更好的方法是通过网络的方式进行安装，这样不会缺少依赖模块，减少运行中的故障，也方便后续更新。下面就介绍它的安装方法。

- 因为esptool.py需要使用python，所以我们先需要安装python，并将python加入系统路径（path）。
- 安装python的包管理器pip（目前python安装版本中通常已经集成了pip），通常是使用get-pip.py进行安装。
在 <https://pip.pypa.io/en/latest/installing/> 可以找到安装的说明和需要下载的文件，按照说明可以很容易安装pip。（如果同时安装了python2和python3，pip可能默认是pip3，需要用pip2来代替下面的pip，在Linux上需要用sudo权限安装）。
- 用pip安装esptool
`pip install esptool`
- 因为esptool需要使用串口，所以还需要安装pyserial。
`pip install pyserial`

安装后，在Linux下，通常就可以直接运行esptool.py，在Windows下，esptool一般安装在python2\Scripts\目录下，需要输入完整目录才能运行，如：

`c:\Python27\Scripts\esptool.py`

如果不清楚esptool.py的用法，可以输入-h查看帮助，如

`esptool.py -h`

甚至可以查看某个用法的帮助：

`esptool.py read_flash -h`

乐鑫没有直接提供读取工具，但是可以用esptools.py读取，例如：

```
esptool.py --port XXXX read_flash 0 0xB0000 1.bin
```

基本用法是：

```
esptool.py read_flash start_addr size filename
```

当因为某些意外原因导致MicroPython运行后输出乱码，刷固件也不能恢复时，需要清理（擦除）Flash，然后在刷固件才能恢复正常。

使用esptool.py可以很方便的擦除ESP8266的Flash。注意下面命令中的串口需要根据实际串口设定，如果擦除的速度过快（不到1S），很可能没有完全擦除，这是可以再擦除一次。擦除时和刷固件一样，需要保持按住Flash按键然后再按下并释放RESET按键，然后在进行擦除。

- 在Linux下
`esptool.py --port /dev/ttyUSB0 erase_flash`
- 在Windows下
`c:\Python27\Scripts\esptool.py --port /COM9 erase_flash`

esp8266的刷机工具esptool.py稳定版升级到了1.2.1（开发版1.3），大家可以去升级一下。

```
pip install --upgrade esptool
```

上周pip也升级到了9.0.1，也可以顺便升级一下。

```
pip install --upgrade pip
```

今天测试ESP8266时，发现使用超级终端时偶尔会出现一些问题。比如大数计算，经常数据会丢失，如：

```
>>>44**55
```

```
2454127978827264271087665173489762127868017461208347716238318
```

而正确的结果是：

```
2454127978827264271087665173489762127868017461208347716238318  
94280372224
```

如果换成putty进行putty，结果就是正常的。感觉超级终端可能在数据量较大时，偶尔会出现问题。

从MicroPython的1.8版本开始，因为固件越来越大，超过了512K，所以就不在支持512K的ESP8266模块，如ESP-01。

在1.8.6版本中，ESP8266的SDK升级到了2.0版本，API作出了很多调整。现在又可以重新支持512K的模块了。

具体方法是：

- 更新MicroPython的源码到1.8.6
- 更新编译器
- 在esp8266目录下
 - 先要清理以前编译的内容
 - `make clean`
 - 然后在命令行指定512k参数
 - `make 512k`

编译后产生的bin文件，就小于512k了，但其实剩余的也不多（23k），放不下多少程序了。

虽然没有SDIO接口，但ESP8266现在可以通过SPI方式挂载SD卡了。下面介绍具体方法：

将macroSD通过SPI和ESP8266连接，一共需要4个GPIO和VCC、GND。

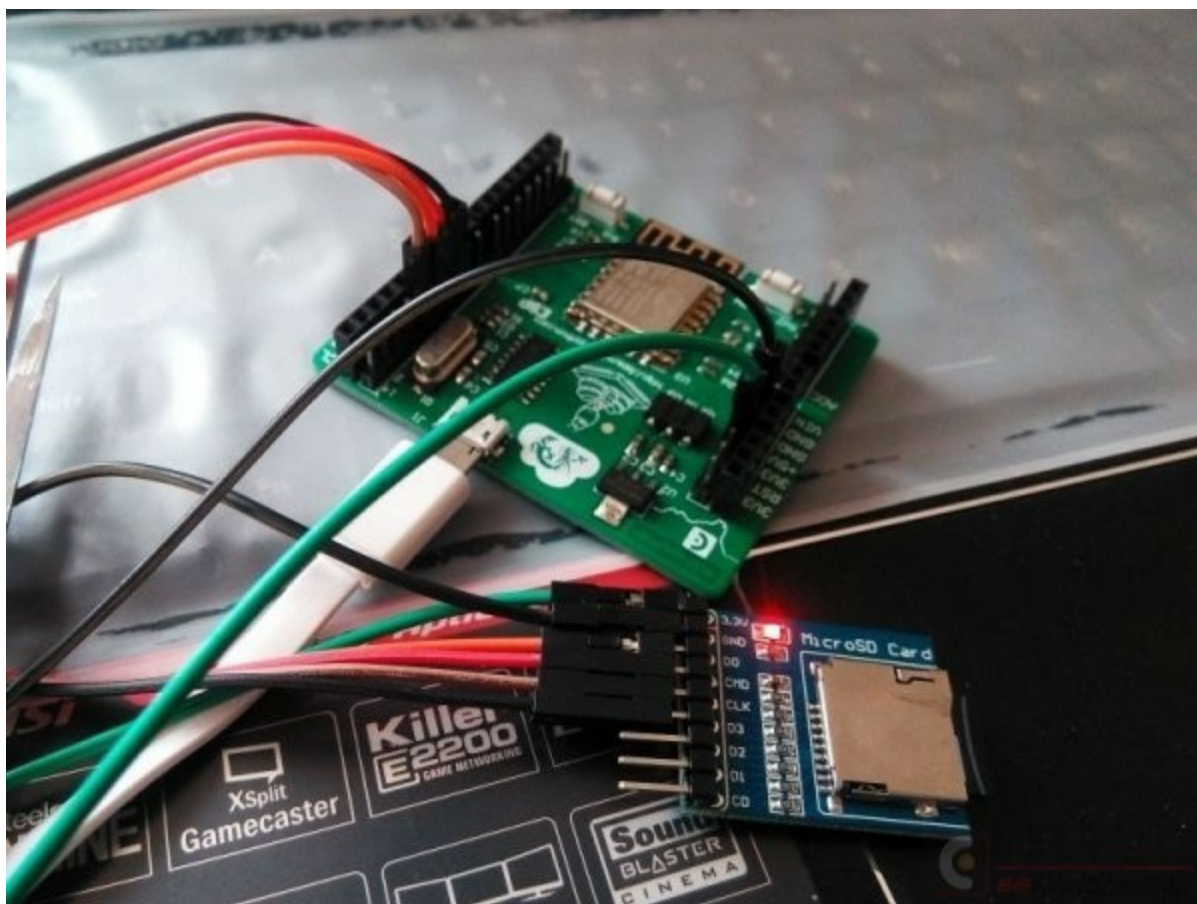
SPI的连接方式是：

	ESP8266	MacroSD
1	MI	MISO
2	MO	MOSI
3	SCK	SCK
4	G16	CS

CS可以接任意IO，在官方例程中使用了GPIO15，但是GPIO15会影响启动，启动时如果GPIO15不是低电平会造成启动时故障，因此不能连接GPIO15启动，换成GPIO16就没有这个问题。

连接后，先将sdcard.py通过传输工具下载到ESP8266，然后用下面命令就可以挂载了。

```
>>> import machine, sdcard, os
>>> sd = sdcard.SDCard(machine.SPI(0), machine.Pin(16))
>>> os.umount()
>>> os.VfsFat(sd, "")
<VfsFat>
>>> os.listdir()
['LOST.DIR', 'backup', '10.bmp', '09.bmp', '08.bmp', '07.bmp',
'06.bmp', '05.bmp',
', '04.bmp', '03.bmp', '02.bmp', '01.bmp']
```



注：

- ESP8266固件版本不低于1.8.3
- 使用sdcard.SDCard函数时需要先插卡
- 这个sdcard驱动有些挑卡，如果提示不支持的卡或者os.VfsFat失败，可以换卡试试。

要自己编译ESP8266的源码，就需要安装相应的工具链。官方给出的工具链是在Debian/Ubuntu下，具体步骤如下：

- 在github下载工具链源码

<https://github.com/pfalcon/esp-open-sdk>

或者直接用git克隆一个

```
git clone --recursive https://github.com/pfalcon/esp-open-sdk
```

- 安装依赖库

```
$ sudo apt-get install make unrar-free autoconf automake libtool  
gcc g++ gperf \  
flex bison texinfo gawk ncurses-dev libexpat-dev python-dev  
python python-serial \  
sed git unzip bash help2man wget bzip2
```

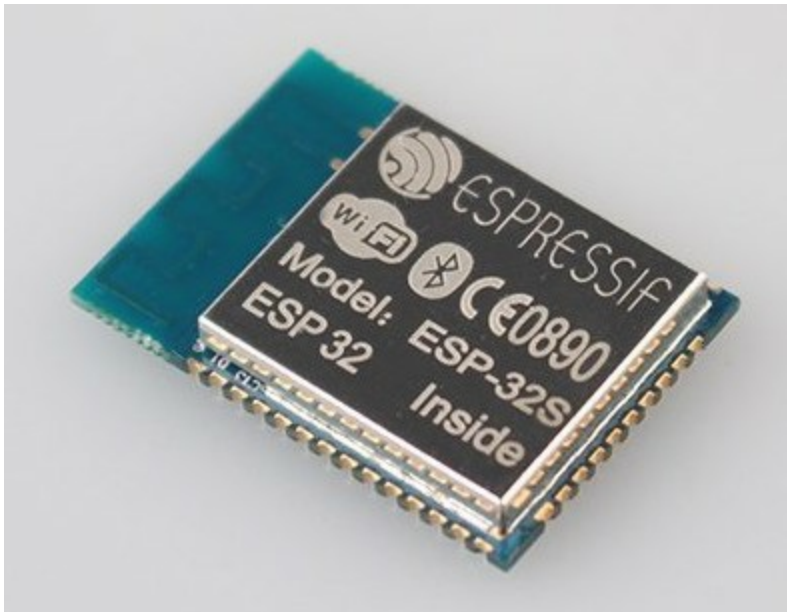
某些版本还需要

```
$ sudo apt-get install libtool-bin
```

- 进行编译

```
$ make
```

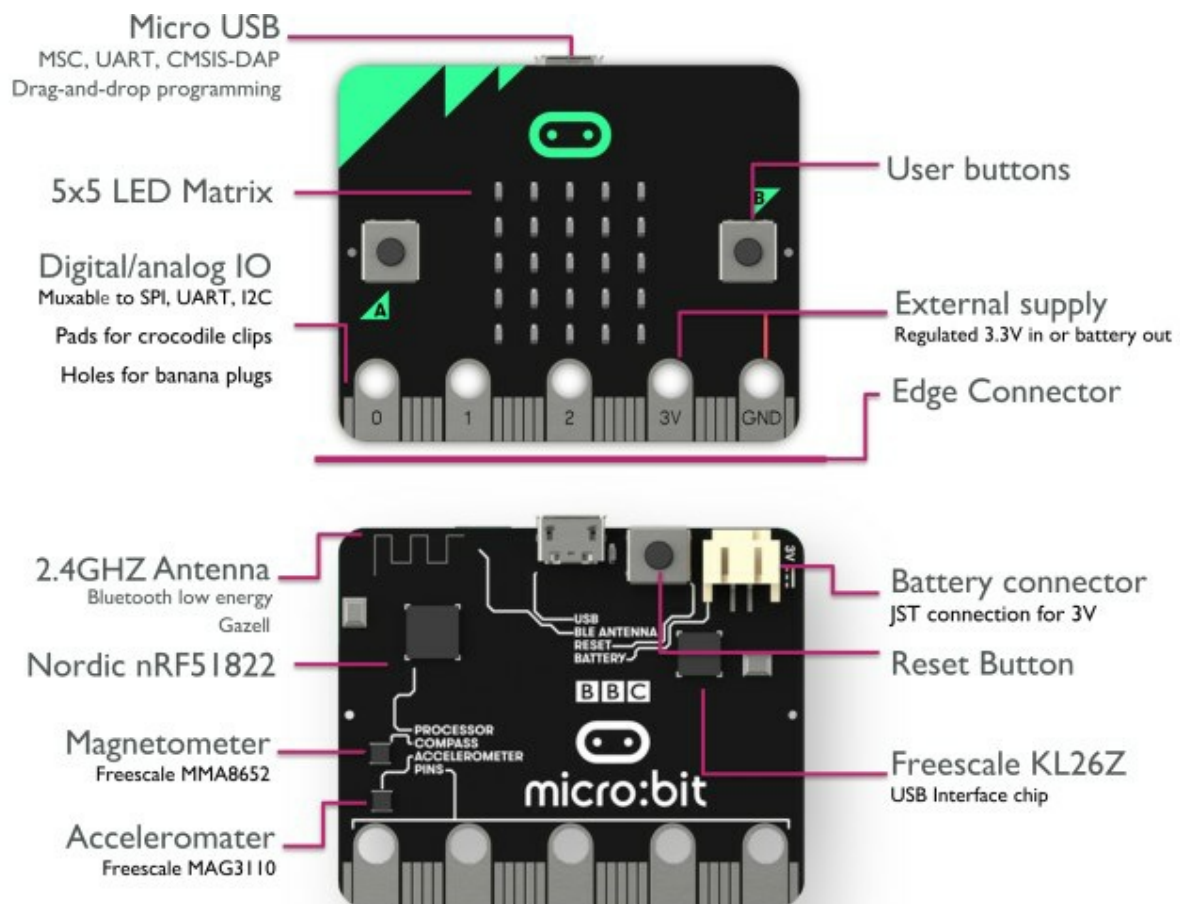
在编译过程中，会自动下载需要的编译器和组件，大约有100多M，在网络情况不好时会经常中断，可以多试几次。下载后会自动编译源码，产生完整的工具链文件，整个过程与网络速度和CPU速度有关。



近日，micro:bit 开发板的硬件方案完全公开了，大家可以自己DIY了。

参考网站：

- <http://tech.microbit.org/hardware/>
- <https://github.com/microbit-foundation/microbit-reference-design>



micro:bit:developer community

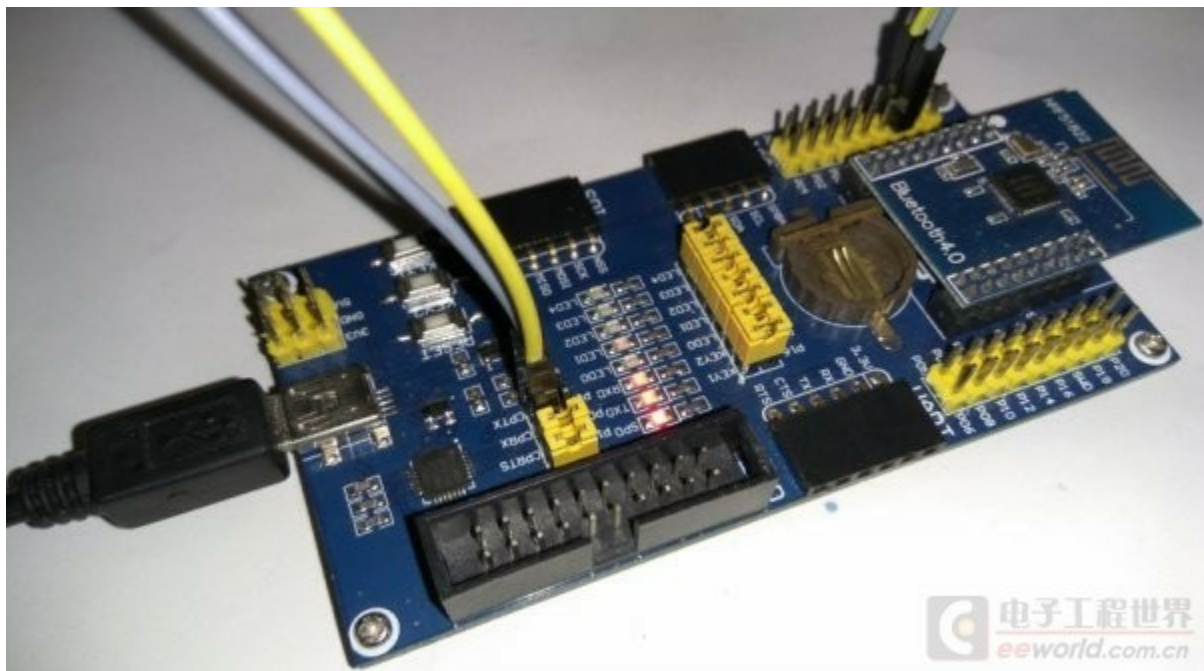
有没有网友发现上面图中的错误？

现在虽然 micro:bit 开发板很难得到，但是想到它的主控芯片是 nRF51822，而很多蓝牙模块和开发板上，也有这个芯片，正好在网上又找到现成的固件文件，就想应该可以将固件下载进去。

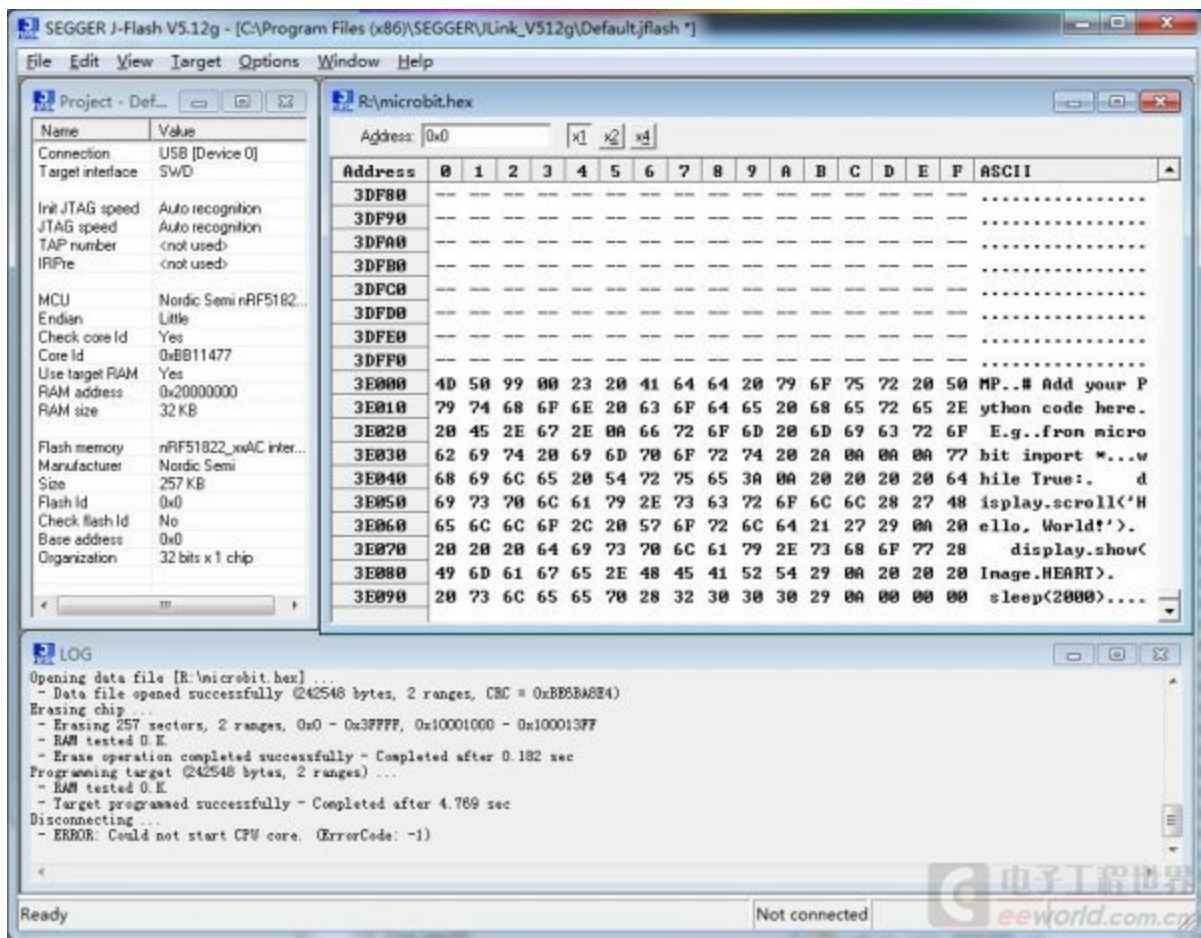
nRF51822 有几种型号，区别在于 Flash 和 SRAM 的大小不同。而 micropython 需要大于 128K 的 Flash 才能运行，因此需要 nRF51822AA 或者 nRF51822AC，nRF51822AB 则不行。

找了一下已有的开发板和蓝牙模块，发现有三种带有 nRF51822，其中微雪的 BLE400 使用了 nRF51822AC，另外一个蓝牙串口模块使用了 nRF51822AA，可以尝试，另外一个 Seeed Arch BLE 使用了 nRF51822AB，不能使用。在研究了一下电路图，发现 microbit 的 REPL 使用的串口是 P0.24/P0.25，只有微雪的 BLE400 可以通过飞线方式实现。

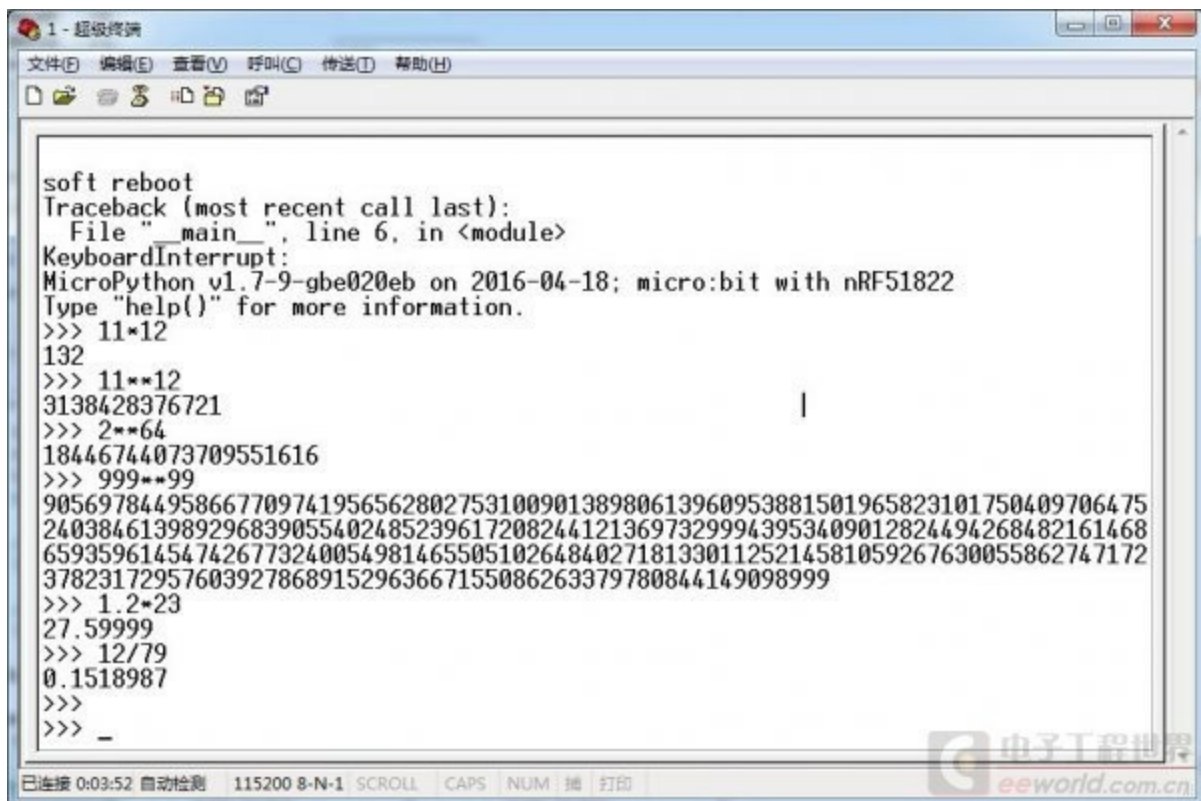
在 BLE400 上，取下连接串口的两个短路帽，用杜邦线连接到 P24/P25，其中 P24 连接 CPRX，P25 连接 CPTX。



找了一个 Jlink OB，将 microbit.hex 下载进去。下载后 LED 开始有规律的闪动，说明程序开始运行了。另外从 J-Flash 的文件窗口可以看到，Flash 已经使用到 0x3E090，256K 的空间已经所剩不多了，不到 8K。



因为BLE400上带有CP2102芯片，可以直接通过它连接串口，所以不用另外的USB转串口了。连上超级终端，设置好串口参数（115200，无流量控制），发现没有任何反映。感觉可能是hex文件中带有默认的程序，已经在运行默认的程序了，所以没有提示，于是按下Ctrl+C尝试终止，果然就看到了熟悉的MicroPython的REPL界面，做了简单的尝试，大部分功能都在，支持浮点运算和大数计算，这一点比CC3200好多了。当然板级库不再是pyb或者machine，而是变为了microbit。



```
soft reboot
Traceback (most recent call last):
  File "__main__", line 6, in <module>
KeyboardInterrupt:
MicroPython v1.7-9-gbe020eb on 2016-04-18; micro:bit with nRF51822
Type "help()" for more information.
>>> 11*12
132
>>> 11**12
3138428376721
>>> 2**64
18446744073709551616
>>> 999**99
90569784495866770974195656280275310090138980613960953881501965823101750409706475
24038461398929683905540248523961720824412136973299943953409012824494268482161468
65935961454742677324005498146550510264840271813301125214581059267630055862747172
378231729576039278689152963667155086263379780844149098999
>>> 1.2*23
27.59999
>>> 12/79
0.1518987
>>>
>>> _
```

如果大家也有nRF51288AA或者nRF51822AC，并且P24/P25可以引出来，可以试试下面的固件，体验一下。

[固件文件](#)

也可以到[社区下载](#)

国外网友写的旋转编码器库

使用方法:

```
from machine import sleep_ms
from encoder import Encoder # or from pyb_encoder import Encoder

e = Encoder('X11', 'X12') # optional: add pin_mode=Pin.PULL_UP
lastval = e.value

while True:
    val = e.value
    if lastval != val:
        lastpos = val
        print(val)
    sleep_ms(100)
```

<https://github.com/SpotlightKid/micropython-stm-lib/tree/master/encoder>

utelnetserver是一个ESP8266的MicroPython模块，使用它可以
将ESP8266变为一个Telnet服务器，通过telnet和ESP8266相连接。

功能：

- Telnet服务器
- REPL

使用方法：

```
import utelnetserver
utelnetserver.start()
```

限制：

- 一次只能连接一个客户端
- 暂时不支持授权

注：

- 需要先配置好网络参数
- 支持AP模式和普通模式

<https://github.com/cpopp/MicroTelnetServer>

在Ubuntu下，首先要添加软件仓库

```
ubuntu 16.0  
deb http://ppa.launchpad.net/team-gcc-arm-embedded/ppa/ubuntu xenial main
```

```
ubuntu 14.04  
deb http://ppa.launchpad.net/team-gcc-arm-embedded/ppa/ubuntu  
trusty main
```

然后需要添加ppa文件

```
sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
```

更新仓库

```
sudo apt-get update
```

然后就可以安装gcc编译器

```
sudo apt-get install gcc-arm-embedded
```


先安装git软件

Linux下:

```
sudo apt-get install git
```

Windows下，可以安装[SourceTree](#)、Github desktop等软件，然后使用命令就可以克隆一个新仓库了

```
git clone --recursive
```

```
https://github.com/micropython/micropython.git
```

在windows编译时，如果gcc编译器的路径没有添加到系统路径，同时又不希望改动makefile文件时，可以在编译是输入下面命令：

```
make CROSS_COMPILE=e:/gcc-arm/bin/arm-none-eabi- BOARD=XXXX
```

注意不能make编译参数在等号后不能带有空格

在make BOARD=XXXX后，加上编译开关 -j8，后面数字是线程数。

不是所有的终端软件都适合操作micropython，因为有些软件的热键会有冲突，或者操作习惯区别太大。比较适合终端有putty、kitty、Windows的超级终端等，而MobaXterm就不合适，因为快捷键有一些不同。Linux下可以用putty、gtkterm。

在终端下，控制命令有：

CTRL-A	-- on a blank line, enter raw REPL mode
CTRL-B	-- on a blank line, enter normal REPL mode
CTRL-C	-- 中断运行的程序
CTRL-D	-- 软复位
CTRL-E	-- 粘贴模式
上下方向键	-- 调出以前输入命令

MicroPython提供了一个Linux版本的Micropython，方便在PC上模拟运行。使用时，需要先编译源码，然后才能运行。

- 根据官方文档，在编译前需要先安装依赖文件。

```
sudo apt-get install build-essential libffi-dev pkg-config
```

- 然后更新子仓库

```
git submodule update --init
```

- 再编译依赖库

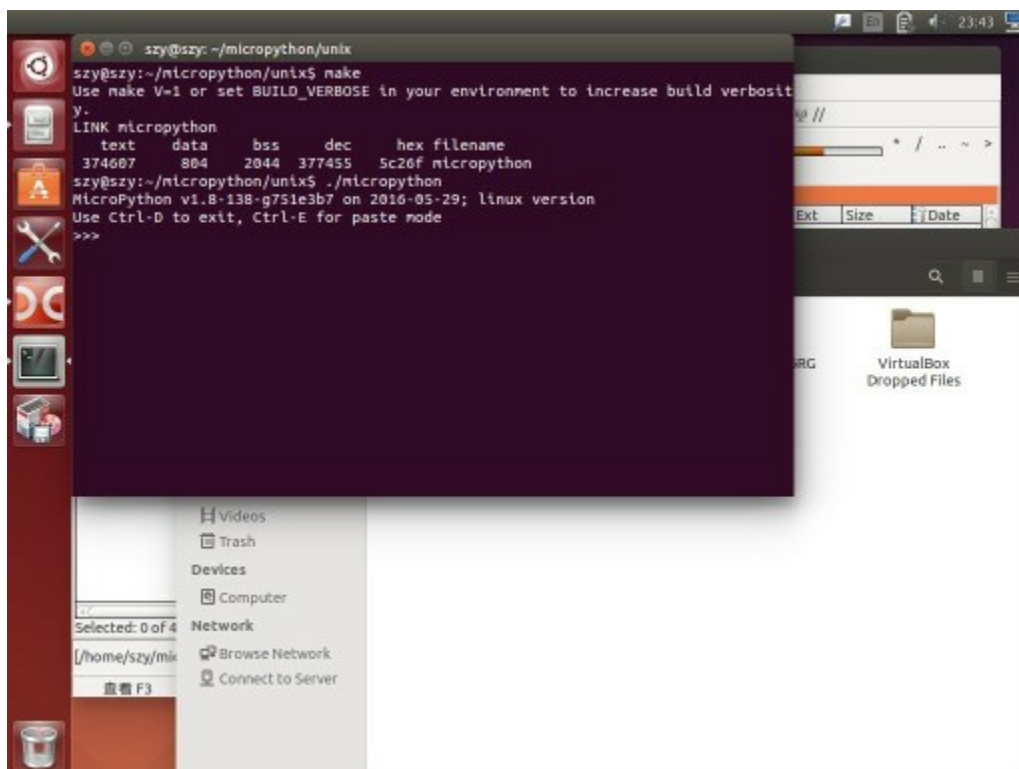
```
make deplibs
```

- 最后编译源码

```
cd unix
```

```
make
```

编译完成后，就可以在Linux下运行体验，快速测试了。当然和底层硬件相关的库是没有的，比如不能import pyb。

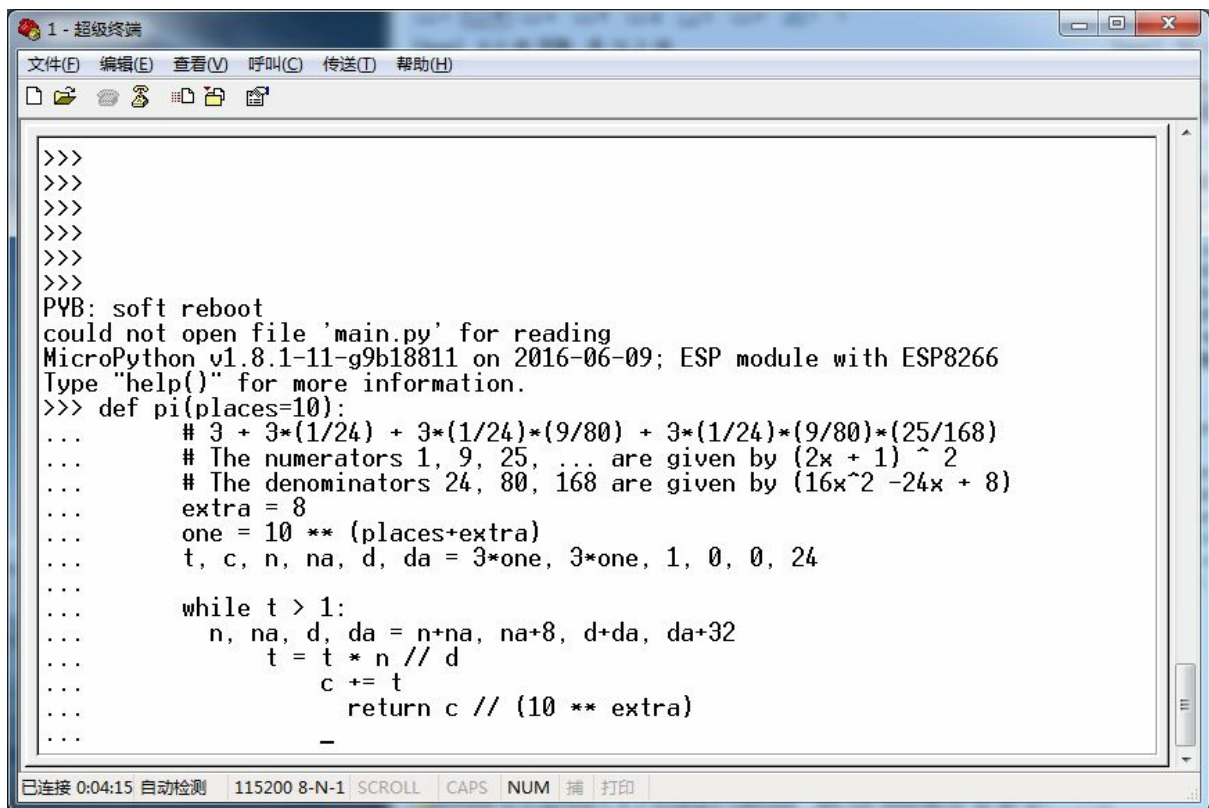


```
sudo screen /dev/ttyACM0
```

如果没有安装screen，需要先安装screen软件

```
sudo apt-get install screen
```

使用Micropython时，我们可以在REPL下输入代码。但是如果代码比较长，输入就比较麻烦。但是如果直接复制，就会出现下面的问题，换行的代码不能被正确识别，不能正常运行。

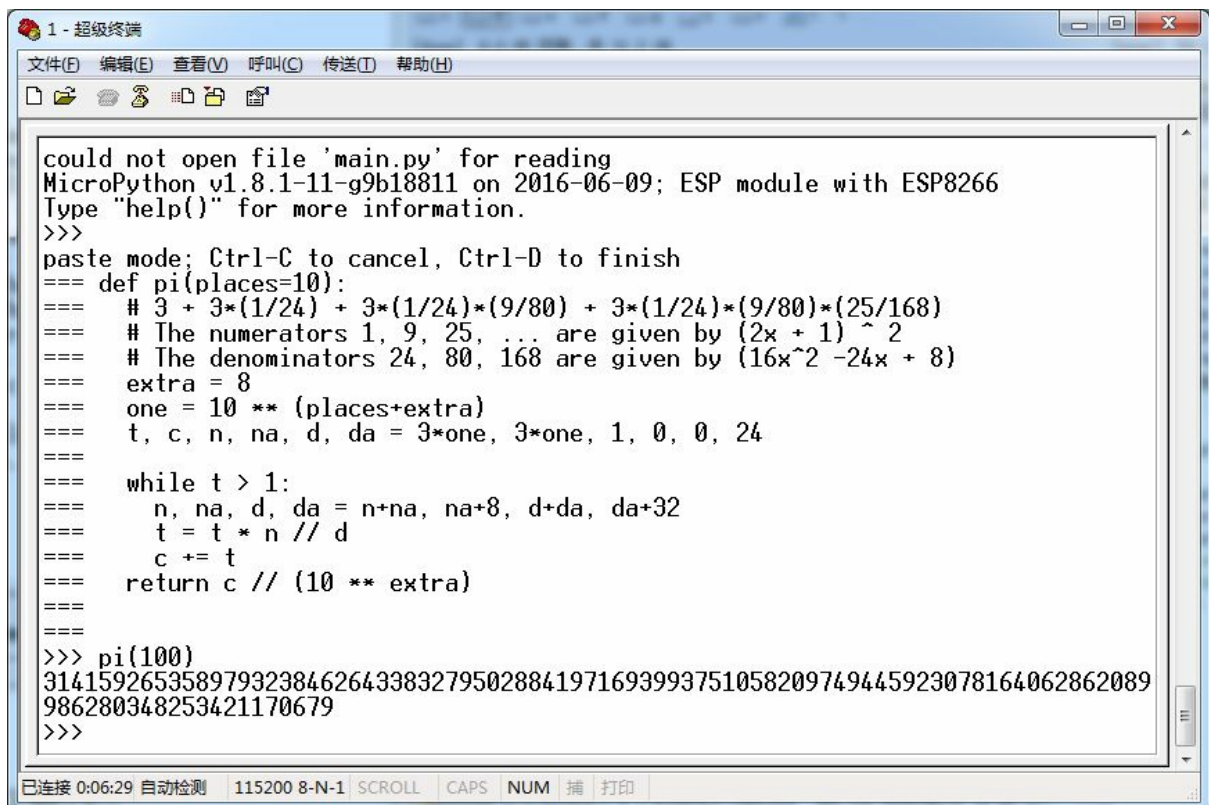


```
>>>
>>>
>>>
>>>
>>>
>>>
PYB: soft reboot
could not open file 'main.py' for reading
MicroPython v1.8.1-11-g9b18811 on 2016-06-09; ESP module with ESP8266
Type "help()" for more information.
>>> def pi(places=10):
...     # 3 + 3*(1/24) + 3*(1/24)*(9/80) + 3*(1/24)*(9/80)*(25/168)
...     # The numerators 1, 9, 25, ... are given by (2x + 1) ^ 2
...     # The denominators 24, 80, 168 are given by (16x^2 -24x + 8)
...     extra = 8
...     one = 10 ** (places+extra)
...     t, c, n, na, d, da = 3*one, 3*one, 1, 0, 0, 24
...
...     while t > 1:
...         n, na, d, da = n+na, na+8, d+da, da+32
...         t = t * n // d
...         c += t
...         return c // (10 ** extra)
...
-
```

其实micropython的作者考虑到了这个问题，他特别设计一个粘贴模式。

- 先在命令行提示符状态下，按下Ctrl-E组合键，就会出现提示：
paste mode; Ctrl-C to cancel, Ctrl-D to finish
===
- 然后在粘贴代码，完成后按下Ctrl-D推出粘贴模式

我们在尝试粘贴刚才的代码，就是正确的了。



```
could not open file 'main.py' for reading
MicroPython v1.8.1-11-g9b18811 on 2016-06-09; ESP module with ESP8266
Type "help()" for more information.
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def pi(places=10):
===   # 3 + 3*(1/24) + 3*(1/24)*(9/80) + 3*(1/24)*(9/80)*(25/168)
===   # The numerators 1, 9, 25, ... are given by (2x + 1) ^ 2
===   # The denominators 24, 80, 168 are given by (16x^2 - 24x + 8)
===   extra = 8
===   one = 10 ** (places+extra)
===   t, c, n, na, d, da = 3*one, 3*one, 1, 0, 0, 24
===
===   while t > 1:
===       n, na, d, da = n+na, na+8, d+da, da+32
===       t = t * n // d
===       c += t
===   return c // (10 ** extra)
===
>>> pi(100)
31415926535897932384626433832795028841971693993751058209749445923078164062862089
986280348253421170679
>>>
```

已连接 0:06:29 自动检测 115200 8-N-1 SCROLL CAPS NUM 插 打印

系统路径是`sys.path`，在路径中的文件可以被python搜索到并直接使用。默认系统路径是：

```
sys.path  
['', '/flash', '/flash/lib']
```

如果需要添加自己的目录，可以用`append`命令，如：

```
sys.path.append('usr')
```

在MicroPython的drive目录下提供了SPI模式连接SD卡的驱动，下面介绍一下具体的使用方法：

- 将micropython源码drive/sdcard目录下的sdcard.c文件复制到开发板
- 导入sdcard库
- 通过spi驱动sdcard（这时需要插卡）
- 挂载文件系统
- 使用文件系统

下面是在小钢炮开发板上连接macroSD的例子：

```
import pyb, sdcard, os
sd = sdcard.SDCard(pyb.SPI(2), pyb.Pin('B12'))
pyb.mount(sd, '/sd2')
os.listdir('/sd2')
```

注：

- 这个程序比较挑SD卡，试过几种不同的卡，有的可以读取，有的提示不支持的格式，有的挂载时会出错
- micropython的版本需要高于1.8.2才行

今天在pyb上遇到一个问题，开始使用的时候正常，突然找不到pybflash了，插上电脑也没有反应。

开始以为是虚焊，重新焊接主要地方后，问题依然存在。

连接BOOT0到VCC，重新启动可以进入DFU模式，说明USB和芯片没有问题。检查系统电压，发现VCC只有2.9V，比预期的3.3V低。按下复位键，发现电压回复到3.2V。

用万用表测量二极管，正方向导通都正常，正向压降也正常，但是通电后输出电压只有2.9V。用镊子短路二极管D1，VCC变为3.3V，最后定位故障到二极管D1。更换新的二极管后，故障消失。

正常情况下，二极管上电流不大(实测小于10mA)，小封装的SS5819是可以满足要求的。二极管也不容易损坏，估计是遇到国产二极管中质量不好的。

一块pyb的USB连接有问题，无论是dfu模式或者其他方式，都无法识别出来。

反复检查，没有发现问题，后来偶然发现D-上电压偏低，在仔细排查，发现USB插座上D-和地短路，重新焊接USB插座后，故障消失。

可以运行Micropython的各种开发板

为了方便大家使用和测试，在Github上创建MicroPython开发板的固件库，收集整理各种可以运行MicroPython开发板的固件。

所有发布的固件都是社区编译，都经过了测试，保证是可以运行的，不会出现官方下载中某些固件（机器自动编译）不能运行的情况。

 **GitHub**

https://github.com/shaoziyang/MicroPython_firmware

 **oschina** 镜像

https://git.oschina.net/shaoziyang/MicroPython_firmware

EEWORLD版pyboard说明

经过前后两次改进，EEWORLD版的pyboard终于完工了。它是在pyboard1.0基础上，做了少量修改而成。

主要改进

- 使用低功耗的LD0（XC6206）取代了不常见的MCP1802
- 取消了较难焊接的三轴传感器MMA7660
- 增加了VIN/3V3电流测试功能（需要断开反面的连线）



- 替换了部分元件，更换为更常见的型号
- 增加了ST和EEWORLD的Logo
- 在main.py中增加了一个启动程序，自检LED，4个LED轮流闪一次，然后用LED3（橙色）做呼吸灯。

```
from pyb import Timer

# LED loop test
def LED_loop_test():
    for i in range(1, 5):
        pyb.LED(i).on()
        pyb.delay(100)
        pyb.LED(i).off()
        pyb.delay(100)

LED_loop_test()

# LED3 breathing lamp
ia = 1
```

```

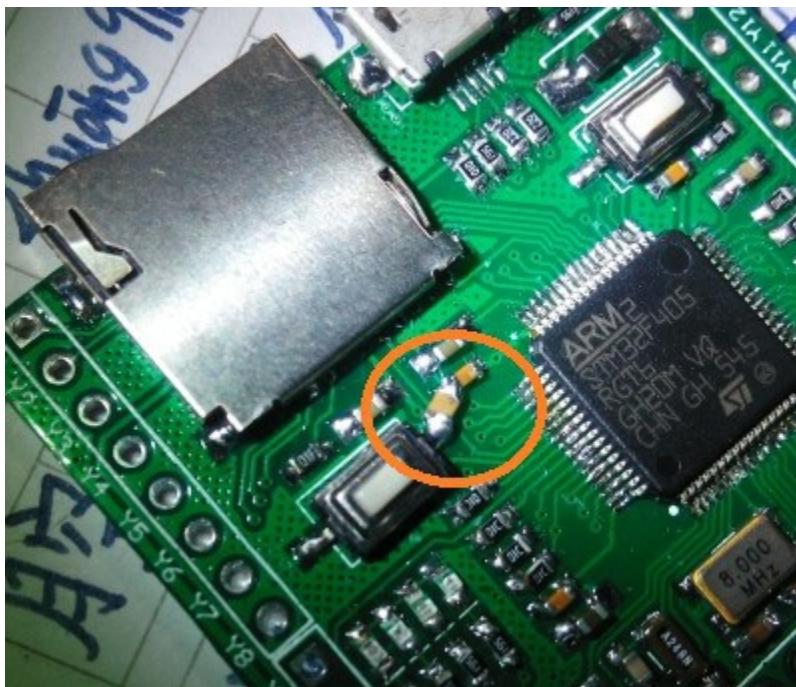
da = 1
def fa(t):
    global ia, da
    if (ia==0) or (ia==255):
        da=256-da
    ia=(ia+da)%256
    pyb.LED(3).intensity(ia)

tm=Timer(1, freq=200, callback=fa)

```

已知问题:

- 因为贴片时一个配合失误，造成LED焊接反了。目前的LED都是我手工重新修正，因为数量不够，所以部分LED用了其他规格。虽然每个板子都做了测试，但是难免会有疏忽，运输中也可能有摔碰，如果有LED不亮的，请大家包涵一二，自行修理一下。
- 如果固件损坏，或者升级固件或者自己DIY时，[可以参考这里烧写固件](#)
- 原版按键SW上没有并联电容，在按下时会产生抖动信号，如果使用中断方式容易产生多次触发。可以自行增加一个100nF电容，就可以有效消除抖动。



当第一次使用装载了MicroPython的pyboard时，我们只要一根安卓手机的数据线（macroUSB）就足够了。当连接了macroUSB线后，在windows中会自动安装移动磁盘驱动和虚拟串口驱动。移动磁盘的驱动系统自带了，可以自动识别出来，而虚拟串口的驱动可以在这个移动磁盘中找到。在Linux和MacOS下，无需另外安装驱动。

移动磁盘中默认会有4个文件，它们分别是：

- main.py，开机自动运行文件，可以将自己的代码放在里面
- boot.py，开机引导文件，由它加载main.py
- pyboard.inf，windows下的虚拟串口驱动文件
- readme.txt，简要说明



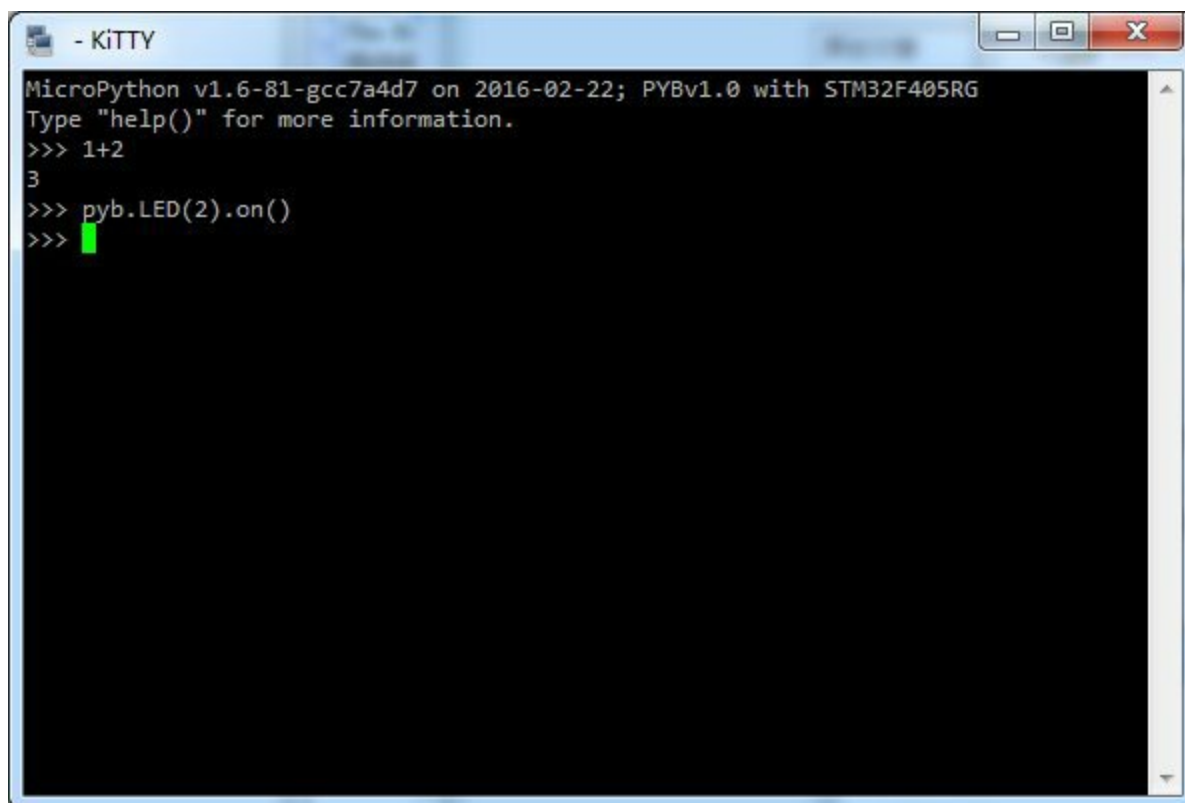
我们可以在main.py中增加代码，加入需要开机自动完成的功能。EEWORLD版的pyboard中，我在main.py中增加了一小段代码，用于LED自检，然后用LED3（橙色）做呼吸灯。参考程序可以在[EEWORLD版pyboard说明](#)中找到。

连上开发板后，我们需要一个支持串口功能终端软件，推荐使用[putty](#)、[kitty](#)和winxp下的超级终端，它们都可以很好的支持micropython。putty和kitty还支持多种操作系统。

设置串口波特率为115200，连接到pyboard，可以看到提示画面，可以直接在命令行中输入指令和代码，运行程序，和在标准的python软件环境下一样，可以输入help()查看简单的帮助。

上下键可以切换历史命令，鼠标右键可以复制剪贴板的内容。按下Ctrl+C可以中止当前程序，Ctrl+D软复位，通常情况下不要按pyboard板子上的复位键，因为这样会丢失串口连接，使终端软件无法连接到开发板。

如果程序较长和复杂，在命令行方式编写就不方便，可以用其他编辑器编写文件，然后复制到pyboard的磁盘中在运行。复制文件后一定要安全退出磁盘，不然pyboard上的文件系统很可能会被破坏，需要进行恢复出厂设置。



```
- KITTY
MicroPython v1.6-81-gcc7a4d7 on 2016-02-22; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> 1+2
3
>>> pyb.LED(2).on()
>>> 
```

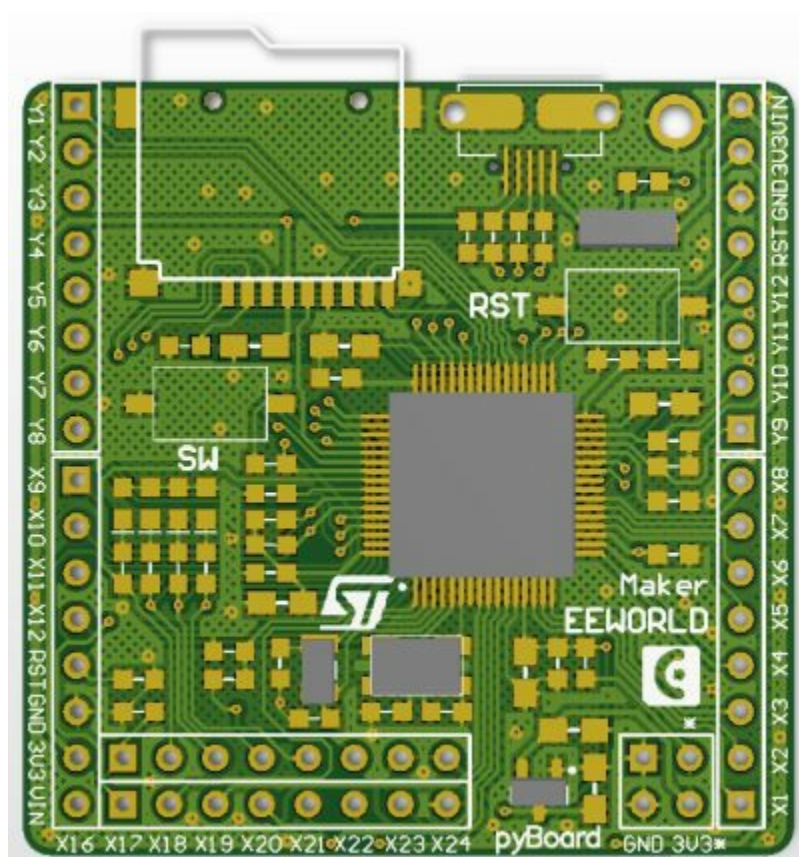
对于micropython的初学者，如果对python语言还不熟悉的，需要先看看python的基本教程，然后在开始。python比C语言简单一些，通常花一两天掌握初步的语法就可以开始玩了。如果对python比较了解，那么可以直接开始，看看LED、按键、定时器、串口的用法，很快就可以掌握基本方法。

MicroPython的作者非常勤奋，软件升级也很快。我在调试和烧写固件时，版本是pybv10-2016-03-28-v1.6-310-g1937953.dfu，而写这个帖子的时候，固件已经升级到pybv10-2016-04-07-v1.6-379-g5e7fa7c.dfu。如果需要升级固件，可以参考一下教程中的[【MicroPython】怎样升级固件](#)。如果是在Linux下，参考这里？<https://github.com/micropython/micropython/wiki/Pyboard-Firmware-Update>。如果你的pyboard没有正常运行，可以先尝试[恢复出厂设置](#)，如果连移动磁盘都无法识别出来，可以重新下载一下固件。

pyboard支持TF卡，如果没有插入TF卡，系统是从内部flash启动；如果插入TF卡并能识别出来，就将TF卡做为默认的磁盘。pyboard使用SDIO方式驱动TF卡，速度比SPI方式快很多。可以将pyboard做为TF读卡器，虽然速度不是很快（约450K/s），应急还是不错的。

硬件上，EEWORLD版本的pyboard是以官方的pyboard v1.0为基础修改而来，具体的改进和一些问题请参考《[EEWORLD版pyboard说明](#)》。

使用过程中我们都会遇到一些问题，很多问题可以在[【MicroPython】教程](#)中找到解决方法。如果没有找到的，可以提出来大家一起讨论。



STM32L476RG和STM32F405RG都是64pin的，基本兼容，所以尝试了用STM32L476替换PYBV10上的STM32F405RG。这样不但可以使用micropython，还可以支持低功耗。



当然，PYBV10的固件和STM32L476DISC的固件是不能直接用在这个板子上，需要进行移植。

ESP-mp-01开发板是使用ESP-12F模块设计的一个低成本MicroPython开发板，也是我们的第二个MicroPython开发板。

ESP-MP-01开发板说明

功能和特点：

- 板载USB转串口功能
- 带有标准Arduino插座（支持排母和排针）
- 支持VIN和USB两种供电方式
- USB电源可以输出到+5V
- 带有复位键和用户按键
- 支持USB升级固件
- 可以通过webrepl上传下载文件

其他：

- GPIO9暂时不能使用
- GPIO10和GPIO16只支持DIO功能，不支持PWM和IRQ功能。
- 默认开启了webrepl，需要配置路由器才能使用。
- ESP8266对电源的稳定性要求较高，通过USB供电时，如果USB电源不稳（如USB接触电阻大、USB线阻抗高），容易造成Flash错误，破坏固件。
- ESP8266的运行功耗较高，运行一段时间后，模块会有一定的发热，这是正常的。



ESP-MP-01开发板使用指南

ESP8266上的MicroPython的用法和pyb上有些不同，有些功能更加简单，如：

```
from machine import Pin

LED = Pin(2, Pin.OUT) # 设置 GPIO2 为输出
LED(0)                 # GPIO2 输出低电平
LED(1)                 # GPIO2 输出高电平
LED.value(0)           # GPIO2 输出低电平
LED.value(1)
LED.high()
LED.low()
```

读取GPIO的电平：

```
from machine import Pin

sw = Pin(0, Pin.IN)
sw() # 读取输入电平
```

大部分GPIO可以支持PWM功能：

```
from machine import Pin, PWM

LED = PWM(Pin(2)) # 设置GPIO2为PWM模式
LED.freq(100)     # 设置PWM频率100Hz
LED.duty(100)     # 设置占空比为100（范围从0-1023）
```


这次ESP8266的活动，我疏忽了很多网友是初次接触MicroPython，并没有参加上一次的pyboard的MicroPython活动，不太熟悉MicroPython。在加上官方只有英文文档，而且文档中还存在不少bug，给使用增加了一些困难。因此我编写了这个【ESP8266】MicroPython入门教程，希望可以帮助大家尽快掌握【ESP8266】MicroPython的用法，享受使用MicroPython的乐趣。

准备工作

在开始玩MicroPython前，我们需要做好准备工作。

- 硬件上，只需要一根macroUSB数据线，大部分安卓手机的数据线都可以，很多开发板也带有macroUSB线，即使没有，淘宝上几元包邮的也可以使用。
- 软件上，需要准备的东西多一点。
 - CH340的USB驱动
 - 终端软件。

MicroPython需要使用支持串口功能的终端，而不能使用普通的串口调试工具。很多只在windows下工作的嵌入式开发者可能不太习惯使用终端软件，这可能需要一点时间去适应。论坛搜集了常用的几种终端软件，大家可以试试。大部分需要的软件和参考资料，在最后的【MicroPython】ESP8266教程和资源中都已提供。

[【MicroPython】常用串口终端软件](#)

无论哪种MicroPython，物理串口目前都只使用了一种波特率：115200, n, 8, 1, none。当然对于象pyb这样使用的USB虚拟串口，其实串口参数随便设置成什么效果都是一样的，但是使用标准参数还是可以减少不必要的问题。

因为ESP8266本身没有USB接口，因此也无法象pyb那样使用虚拟磁盘功能，虽然内部有很大的Flash，却无法直接象磁盘那样访问，只能通过串口或wifi方式访问。文件传输只能通过象webrepl_cli、ESPlorer等软件完成，相比pyboard显得不够方便，希望后续会有更好的文件传输和文件管理工具。

通过USB连接开发板

不像STM32版本的pyboard，ESP8266本身没有USB，只能通过TTL串口和Wifi访问ESP8266。我们先介绍通过串口方式进行连接。

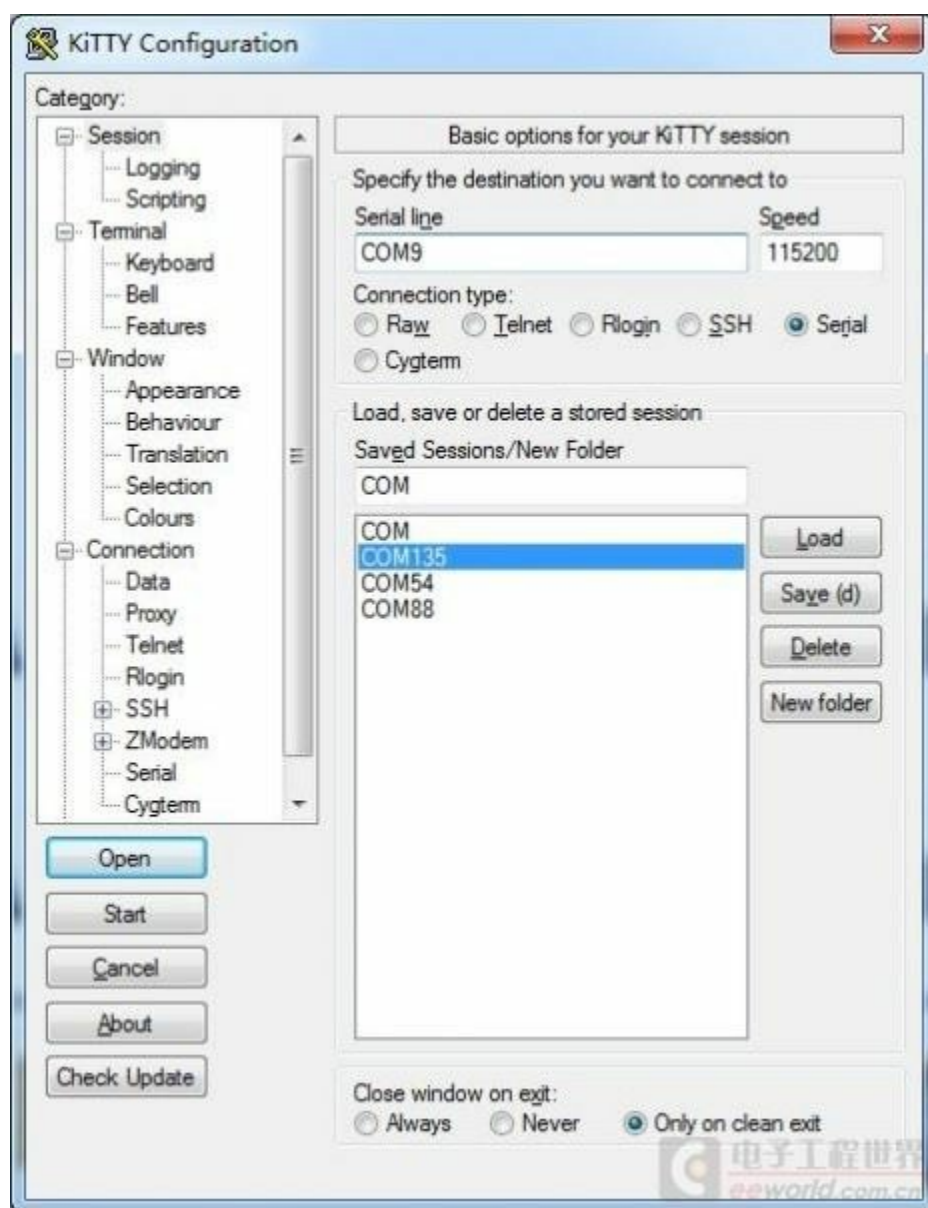
在MicroPython ESP8266开发板上，带有macroUSB接口，以及USB转串口芯片CH340，它可以方便实现计算机与ESP8266模块的连接。使用前需要先安装CH340的驱动，这样当MicroPython开发板连接到计算机，就会出现一个串口设备。下面是windows上显示的虚拟串口，Linux下通常是/dev/ttyUSB0。



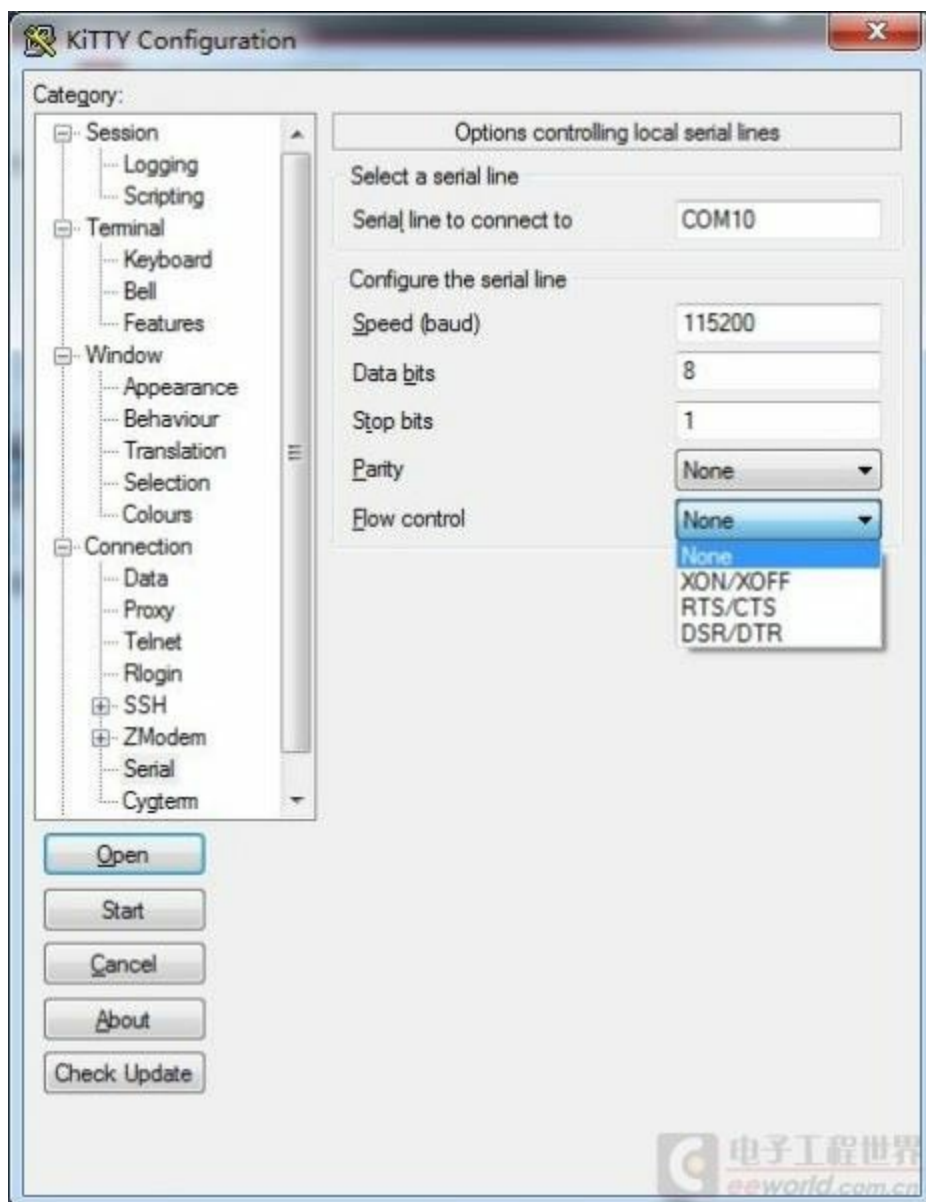
使用终端连接开发板

为了使用MicroPython，我们需要运行一个终端软件，下面以putty为例，其他软件用法也类似。

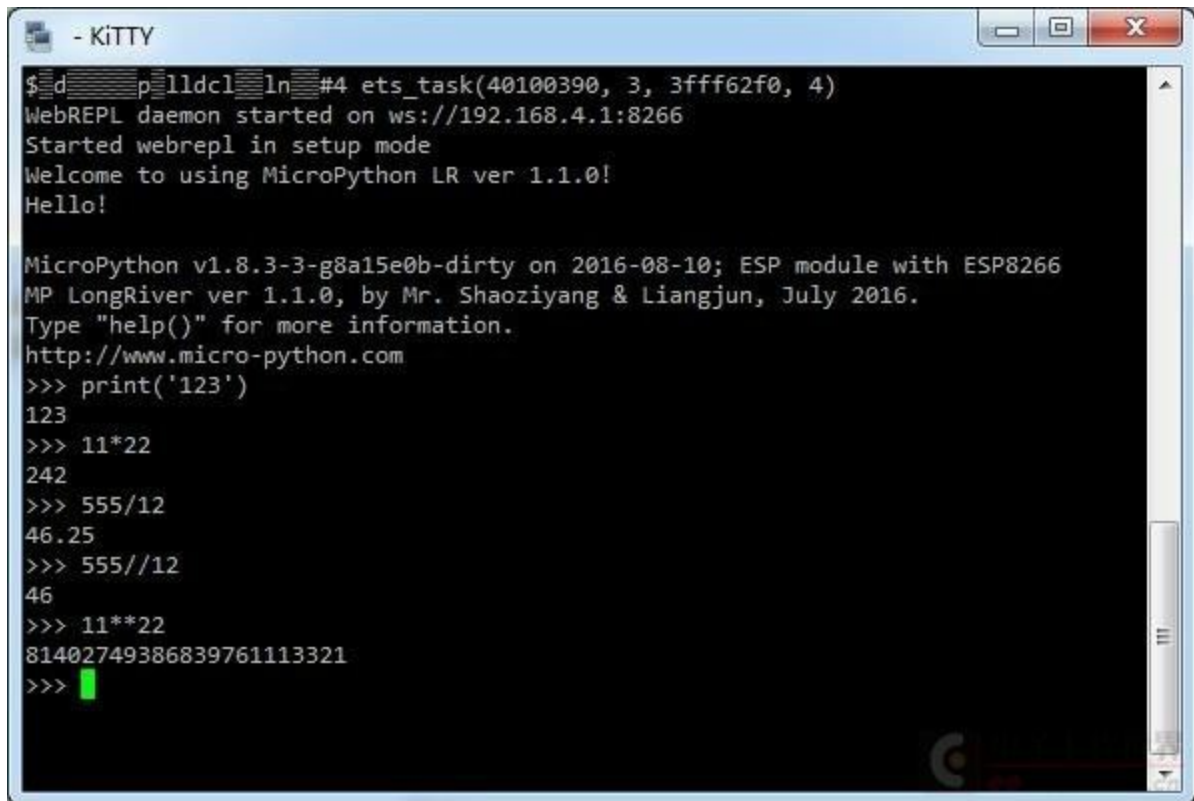
先要设置串口，选择CH340的串口（Windows上在设备管理器中查看串口，Linux下在/dev/中查看），并设置波特率为115200。有些软件还要设置更多参数，一般设置8位数数据，无校验，1位停止位，无流量控制等。



然后在串口设置中将Flow Control改为None。



然后按下open，就可以进入终端界面了。这时界面上可能什么也没有，因为MicroPython已经运行，正在等待输入命令。我们可以按下开发板的复位键，就可以看到屏幕上的提示信息。最开始有一段乱码，这是ESP8266模块开机时内部的调试信息，波特率和我们的不相同，所以是乱码，后面就正常了。等出现三个尖括号的提示符，就可以输入命令了。



```
- KITTY
$ d p lldcl ln #4 ets_task(40100390, 3, 3fff62f0, 4)
WebREPL daemon started on ws://192.168.4.1:8266
Started webrepl in setup mode
Welcome to using MicroPython LR ver 1.1.0!
Hello!

MicroPython v1.8.3-3-g8a15e0b-dirty on 2016-08-10; ESP module with ESP8266
MP LongRiver ver 1.1.0, by Mr. Shaoziyang & Liangjun, July 2016.
Type "help()" for more information.
http://www.micro-python.com
>>> print('123')
123
>>> 11*22
242
>>> 555/12
46.25
>>> 555//12
46
>>> 11**22
81402749386839761113321
>>> █
```

开机时会有一小段延时，屏幕显示Hello，同时LED在闪。这是在运行我们的一个Morse例程，向大家发送Hello。大家可以打印main.py，就知道运行的程序了。

在终端中输入程序

为什么要使用终端软件，而不是串口调试软件，最主要的原因就是在终端里可以灵活的输入程序，运行程序。

一般情况下，我们都是repl交互模式下输入代码，在python的命令提示符(>>>)后就可以输入代码，可以用左右方向键改变字符位置，插入新的字母。也可以用Del或者BS键删除字母。输入时，可以灵活使用TAB键进行代码补全。完成一行后用回车键换行。这些和标准的python环境一样。

在终端下，灵活使用快捷键可以帮助我们。常用的快捷键有：

CTRL-A	-- on a blank line, enter raw REPL mode (这个快捷键不是为了输入程序，一般不要使用)
CTRL-B	-- 在空命令行下，回到正常 REPL 交互模式
CTRL-C	-- 中断正在运行的程序
CTRL-D	-- 软复位
CTRL-E	-- 粘贴模式

上下方向键 -- 调出以前输入命令

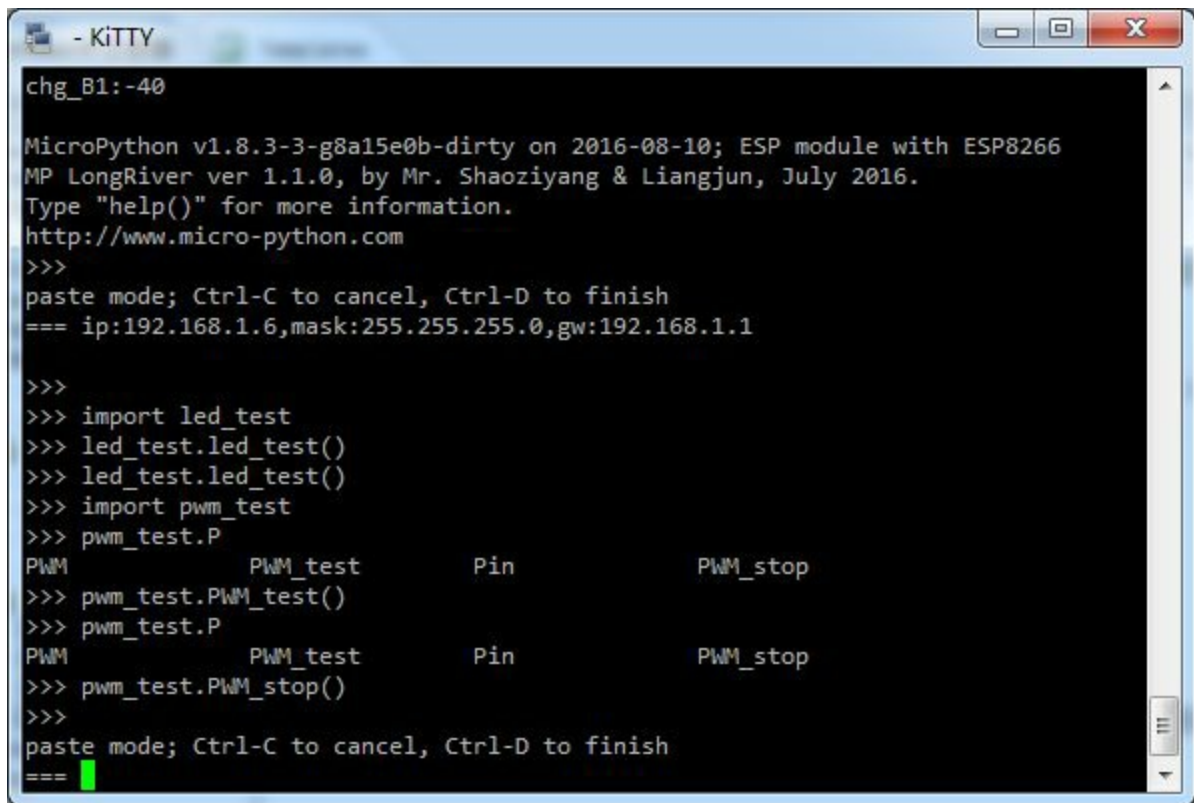
运行程序时，如果出现问题可以随时用Ctrl-C中止运行，或者在空命令行下用Ctrl-D软复位。如果还不能解决问题，就直接按复位键进行硬复位。

遇到有疑问的地方，可以输入help()查看帮助，甚至可以查看一个函数或者库的帮助，如help(machine)。

还可以用dir()查看已经载入的模块、函数、变量，也可以用dir查看一个库里面包含的内容，如dir(machine)。

粘贴代码

对于较长的程序，用键盘输入不但麻烦，效率低，也容易输入错误。一种方法是先将程序复制到剪贴板，然后粘贴进去。在空命令行下按下快捷键Ctrl-E就会进入粘贴模式：



```
- KITTY
chg_B1:-40

MicroPython v1.8.3-3-g8a15e0b-dirty on 2016-08-10; ESP module with ESP8266
MP LongRiver ver 1.1.0, by Mr. Shaoziyang & Liangjun, July 2016.
Type "help()" for more information.
http://www.micro-python.com
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== ip:192.168.1.6,mask:255.255.255.0,gw:192.168.1.1

>>>
>>> import led_test
>>> led_test.led_test()
>>> led_test.led_test()
>>> import pwm_test
>>> pwm_test.P
PWM          PWM_test          Pin          PWM_stop
>>> pwm_test.PWM_test()
>>> pwm_test.P
PWM          PWM_test          Pin          PWM_stop
>>> pwm_test.PWM_stop()
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
===
```

在putty下，鼠标右键就可以将剪贴板内容复制到repl中。在其他软件中，可能稍有区别。粘贴后，用Ctrl-D完成粘贴，或者用Ctrl-C取消粘贴。


```
- KITTY
===
=== import time
===
=== def pi(places=10):
===     # 3 + 3*(1/24) + 3*(1/24)*(9/80) + 3*(1/24)*(9/80)*(25/168)
===     # The numerators 1, 9, 25, ... are given by (2x + 1) ^ 2
===     # The denominators 24, 80, 168 are given by (16x^2 -24x + 8)
===     extra = 8
===     one = 10 ** (places+extra)
===     t, c, n, na, d, da = 3*one, 3*one, 1, 0, 0, 24
===
===     while t > 1:
===         n, na, d, da = n+na, na+8, d+da, da+32
===         t = t * n // d
===         c += t
===     return c // (10 ** extra)
===
=== def pi_t(n=10):
===     t1=time.time()
===     t=pi(n)
===     t2=time.time()
===     print('elapsed: ', time.time_diff(t1,t2)/1000000, 's')
===     return t
===
>>>
```

查看开发板上的文件

NicroPython开发板都是带有文件系统的，它将剩余的Flash空间，模拟成磁盘，可以通过多种方式访问。在pyboard上，因为带有USB接口，所以可以模拟成虚拟磁盘，通过系统的文件管理器访问，非常方便。而在ESP8266上，没有USB接口（只有USB转TTL串口），所以只能通过串口或者Wifi方式访问。

- 文件列表

通过串口访问文件的方法，和pyb上是一样的，都是通过os模块。先看看下面例子，列出当前目录下的文件和目录：

```
>>> import os
>>> os.listdir()
['boot.py', 'demos', 'drive', 'main.py']
```

上面就是开发板默认带有的文件。os模块中的listdir()函数提供查看文件列表的功能。它还支持目录，例如：

```
>>> os.listdir('demos')
['led_test.py', 'morse.py', 'pwm_test.py', 'timer_test.py',
'webservicedemo.py']
```

这就是开发板的demos目录下的文件。

- 查看当前目录

```
os.getcwd()
```

- 改变当前目录

```
os.chdir()
```

例如：

```
>>> os.chdir('/demos')
>>> os.getcwd()
'/demos'
```

- 查看文件内容

MicroPython没有提供Linux下的cat或者windows的type这样直接查看文件的方法，但是可以通过文件读写的方式查看文件内容。如：

```
>>> f = open('main.py', 'r')
>>> f.readall()
"print('Welcome to using MicroPython LR ver
1.1.0!')\nprint('Hello!')\nimport morse\nmorse.send('Hello',
2)\n"
```

运行板载例程

大家拿到的开发板，已经带有例程了。如果刷了MicroPython官方固件，就会丢失例程，这时可以重新刷一下我们提供的带有例程的开发板固件。

开发板的例程在/demos目录下，用`os.listdir('/demos')`就可以查看文件列表，里面为我们提供了几个例程。

- `led_test.py`，LED测试，演示了LED的基本控制，

```
>>> import led_test
>>> led_test.led_test()
```

- `morse.py`，莫尔斯码，开机时的LED闪烁，其实就是在发送Hello的莫尔斯码。如果感觉比较耽误时间，可以随时用Ctrl-C中止运行。

```
>>> import morse
>>> morse.send('123', 2, 0)
```

`morse.send()`函数有三个参数，第一个是要发送的字符串，第二个是IO，在ESP8266上LED是GPIO2，如果用在pybaord上，可以是'A13'这样的用法。第三个参数可以不用，它代表LED是正极驱动还是负极驱动。

- `pwm_test.py`，PWM测试，通过PWM改变LED亮度，实现呼吸灯功能。

```
>>> import pwm_test
>>> pwm_test.PWM_test()
```

如果要停止呼吸等，运行下面命令

```
>>> pwm_test.PWM_stop()
```

- `timer_test.py`，演示了定时器的用法，每秒翻转一次LED

```
>>> import timer_test
>>> timer_test.timer_test()
```

要停止定时器，使用下面命令

```
>>> timer_test.tm.deinit()
```

更多用法，大家可以参考一下[快速参考](#)里的用法，以及[ESP8266的教程](#)。

大家可能发现，运行上面的例程时，没有加上路径，因为我们已经将demos目录添加到系统路径了。

```
>>> import sys
>>> sys.path
['', '/', '/lib', '/drive', '/demos']
```

在AP模式下，micropython会显示一个micropython-xxxxxx热点，连接时

密码是：micropythoN

注意最后一个N是大写的。

连接后的IP：192.168.4.1

另外，热点名称后面的字符串，也就是mac地址的后三位。

开发板使用了CH340芯片转换串口，在win10上有些网友不能正常使用，是因为没有安装最新的驱动，升级驱动程序后就可以正常使用了。

[驱动下载](#)

目前最小的MicroPython开发板

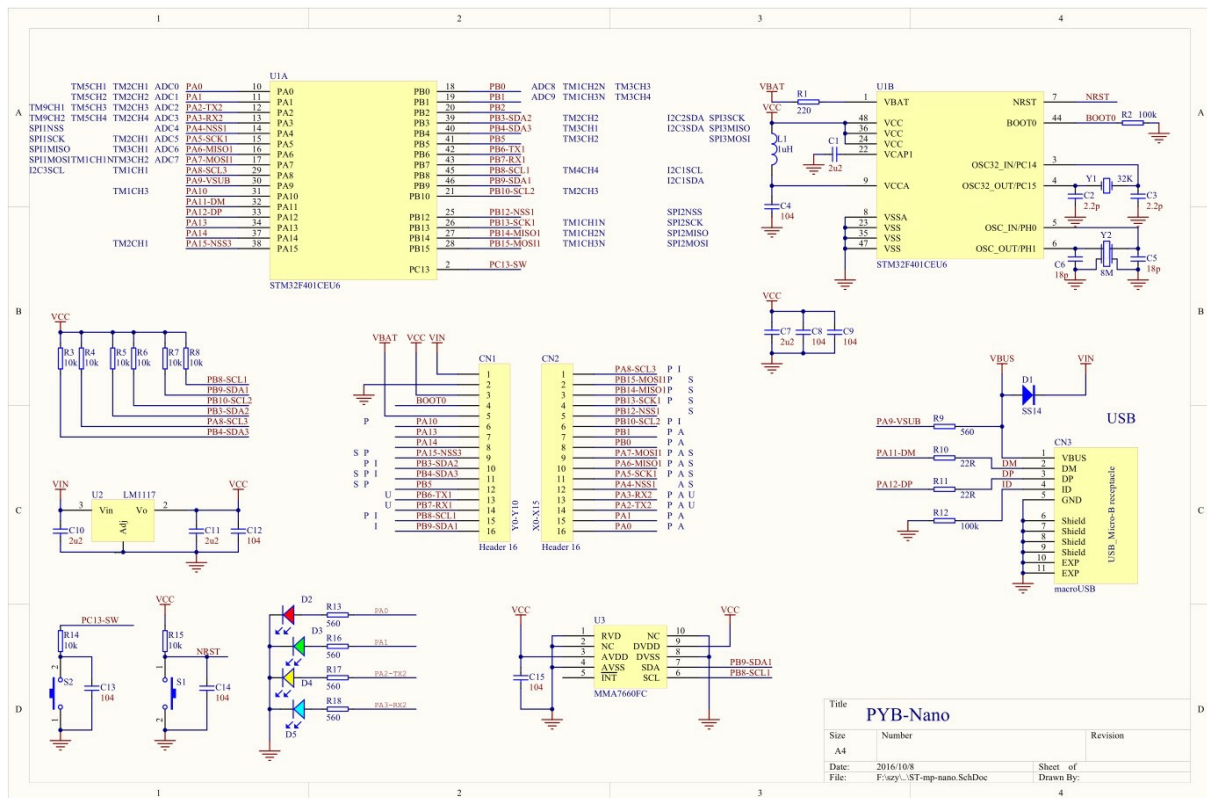
PYB Nano的主要特点：

- 支持 macroUSB
- 2路UART
- 3路I2C
- 3路SPI
- 10路ADC
- 支持RTC
- 支持后备电池输入
- 支持USB供电和VIN输入（最高12V）
- 一个用户按键和一个复位键
- 带有 4个 LED，LED支持亮度调节功能
- 带有加速度传感器（MMA7660）
- 支持USB升级功能
- 支持SWD
- 低成本、高性能
- 开源（所有资料近期将整理出来，请大家[关注论坛](#)和我们的[微信订阅号](#)）

应用范围：

- 教育、学习
- 电子竞技
- 机器人
- 智能硬件
- 物联网开发
- 快速原型设计
- 创客、DIYer

PYB Nano非常适合作为MicroPython的入门开发板，它支持绝大部分MicroPython的功能和函数，成本却只有官方版本的几分之一，是学习MicroPython的首选开发板之一。



pdf版请见[论坛](#)

PYB Nano 开发板快速指南

这篇快速指南是为 MicroPython 和 PYB Nano 的初学者写的，通过这个文档，可以快速掌握 MicroPython 和 PYB Nano 的基本使用方法。



PYB Nano开发板简介

PYB Nano开发板是目前体积最小、成本最低的MicroPython开发板

PYB Nano的主要特点

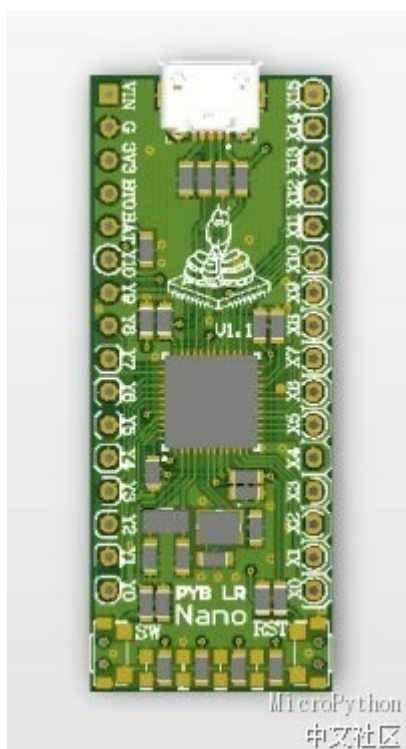
- STM32F401CEU6微控制器
- 16M主时钟
- 支持 macroUSB
- 2路UART
- 3路I2C
- 3路SPI
- 10路12位ADC
- 支持RTC
- 支持后备电池输入
- 支持USB供电和VIN输入（最高12V）
- 一个用户按键和一个复位键
- 带有 4个支持亮度调节功能的 LED
- 带有加速度传感器（MMA7660）
- 支持USB升级功能
- 低成本、高性能
- 开源

应用范围

- 教育、学习
- 电子竞技

- 机器人
- 智能硬件
- 物联网开发
- 快速原型设计
- 创客、DIYer

PYB Nano非常适合作为MicroPython的入门开发板，它支持绝大部分MicroPython的功能和函数，成本却只有官方 PyBoard 的几分之一，是学习MicroPython的首选开发板之一，也是从Arduino进阶到其它应用的最好选择。



系统需求

在开始使用 PYB Nano 前，需要做一点准备工作：

- 一台计算机，安装了不低于Win7，或者Linux、MacOS操作系统，32位/64位都可以
- 串口终端软件，如超级终端、putty、MobaXterm、SecureCRT等
- macroUSB数据线（可以使用安卓手机的数据线）

因为Windows的用户最多，所以下面的介绍也以Windwos为主，但其它操作系统下的用法也是类似的，甚至更简单。

另外，为了顺利使用 MicroPython，大家还需要对 Python 语言有基本了解，因为MicroPython 是基于 python3 的。

安装驱动

在Windows系统中，第一次连接开发板时，会出现一个PYBFLASH磁盘，同时提示需要安装新的设备。这个新的设备就是虚拟串口，它的驱动程序就在新出现的PYBFLASH磁盘上，浏览到这个磁盘安装驱动，安装后就可以使用。

在大部分的Linux、MacOS下无需安装任何驱动程序。

终端软件设置

在调试时，通常都使用串口终端软件。在使用前，需要对终端软件的串口参数进行设置。

先用数据线连接开发板，然后运行任一串口终端软件，并设置串口为PYB Nano 对应的虚拟串口，在将串口参数设置如下：

波特率	115200
数据位	8
奇偶校验	无
停止位	1
流量控制	无

个别软件还需要设置字符集才能正常显示。

REPL 的用法

通常调试程序时，都是在 MicroPython 的 REPL（read-eval-print loop，循环交互解释器）环境下运行。在REPL下可以直接输入命令，有内置的解释器执行。如果命令输入正常
MicroPython支持几个常用的快捷键，如果你熟悉串口终端，会发现它们的习惯是一样的。

- Ctrl-C，停止正在的程序或终止当前的命令行
- Ctrl-D，软复位（soft reset）

- Ctrl-B, 显示系统提示
- Ctrl-E, 进入粘贴模式。可以按下Ctrl-C退出粘贴模式, Ctrl-D完成粘贴
- Tab, 键盘上的Tab键, 可以补全命令

除了Ctrl-C, 其它快捷键需要在空命令行下(没有输入任何字符)才能生效。此外, 还可以使用上下左右光标键

- 上下键, 调出以前输入的命令。MicroPython可以保存最后输入的6条命令
- 左右键, 在当前命令行中移动, 编辑命令

基本用法

LED

无论在哪个开发板中, LED都是最常用的的例程, 我们也从这里开始。

PYB Nano上有4个LED, 分别是红、绿、黄、蓝色。我们可以通过 `pyb.LED(n)` [`n = 1-4`] 去使用它们。如:

```
pyb.LED(1).on()    # LED1亮
pyb.LED(2).off()   # LED2灭
pyb.LED(3).toggle() # 翻转LED3
pyb.LED(4).intensity(20) # 设置LED4亮度20, 范围是[0-255]
```

呼吸灯, 下面程序将LED3设置为呼吸灯

```
from pyb import Timer

ia = 1
da = 1
def fa(t):
    global ia, da
    if (ia==0) or (ia==255):
        da=256-da
    ia=(ia+da)%256
    pyb.LED(3).intensity(ia)

tm=Timer(1, freq=200, callback=fa)
```

在其它MicroPython开发板上, LED的数量可能是1-4个, 但用法是一样的。此外不是每种 MicroPython 开发板的 LED 都支持亮度调整功

能 (intensity()) 。

按键

在MicroPython的Pyboard中，预定义了按键开关对象，它的使用方法如下：

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

如果按键按下，返回True，否则返回False。

GPIO

使用GPIO，需要导入pyb库的Pin对象。

```
from pyb import Pin

p_out = Pin('X1', Pin.OUT_PP) # 定义 X1 引脚为输出
p_out.high()                  # 输出高电平
p_out.low()                    # 输出低电平
p_out.value(1)                 # 等同于 p_out.high()

p_in = Pin('X2', Pin.IN, Pin.PULL_UP) # 定义 X2 为输入
p_in.value() # get value, 0 or 1
```

外中断

下面代码将按键（PC13）定义为外中断输入，上升沿触发模式。每当按下一次按键，LED1就会翻转一次。

```
from pyb import Pin, ExtInt

callback = lambda e: pyb.LED(1).toggle()
ext = ExtInt(Pin('C13'), ExtInt.IRQ_RISING, Pin.PULL_UP,
callback)
```

定时器

定时器需要使用到pyb库的Timer对象。下面程序中，先设置定时器1的频率为1000Hz，然后读取计数器的值，再设置定时器的频率为0.5Hz，并在定时器回调函数（中断）里翻转

LED2。

```
from pyb import Timer

tim = Timer(1, freq=1000)
tim.counter() # get counter value
tim.freq(0.5) # 0.5 Hz
tim.callback(lambda t: pyb.LED(2).toggle())
```

PWM

PWM是定时器模块的一个子功能，可以将定时器的某一通道设置为PWM输出。下面代码将PA3 设置为定时器2的CH4输出，最后设置占空比是10%。注意占空比参数可以是浮点数。

```
from pyb import Pin, Timer

p = Pin('A3') # X1 has TIM2, CH4
tim = Timer(2, freq=1000)
ch = tim.channel(4, Timer.PWM, pin=p)
ch.pulse_width_percent(10)
```

ADC

PYB Nano带有10路12位ADC输入，可以非常容易读取ADC的参数。下面的代码读取引脚 X8（PB0）的输入

```
from pyb import Pin, ADC

adc = ADC(Pin('X8'))
adc.read() # read value, 0-4095
```

另外一种读取ADC的方法是

```
from pyb import Pin, ADC

a=pyb.ADCAll(12) # 设置ADC为12位模式
a.read_channel(8) # 读取ADC8
a.read_core_temp() # 读取内部温度传感器
```

UART

```
from pyb import UART
```

```
uart = UART(1, 9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

I2C

```
from pyb import I2C

i2c = I2C(1, I2C.MASTER, baudrate=100000)
i2c.scan() # returns list of slave addresses
i2c.send('hello', 0x42) # send 5 bytes to slave with address 0x42
i2c.recv(5, 0x42) # receive 5 bytes from slave
i2c.mem_read(2, 0x42, 0x10) # read 2 bytes from slave 0x42, slave
memory 0x10
i2c.mem_write('xy', 0x42, 0x10) # write 2 bytes to slave 0x42,
slave memory 0x10
```

SPI

```
from pyb import SPI

spi = SPI(1, SPI.MASTER, baudrate=200000, polarity=1, phase=0)
spi.send('hello')
spi.recv(5) # receive 5 bytes on the bus
spi.send_recv('hello') # send a receive 5 bytes
```

加速度传感器

```
acc = pyb.Accel()
while True:
    print(acc.x(), acc.y(), acc.z())
    pyb.delay(500)
```

其它常用功能

- `pyb.delay(500)` # 延时500ms
- `pyb.udelay(20)` # 延时20us
- `pyb.unique_id()` # 读取芯片的唯一序列号
- `pyb.millis()` # 复位后的运行时间 (ms)
- `pyb.hard_reset()` # 复位, 和按下复位键的效果相同
- `pyb.bootloader()` # 直接进入 bootloader 模式升级
- `pyb.disable_irq()` # 禁止中断

- `pyb.enable_irq()` # 恢复中断
- `pyb.freq()` # 读取系统时钟
- `pyb.wfi()` # 等待内部或外部中断
- `pyb.stop()` # 休眠模式，需要外部中断或者实时时钟唤醒

MicroPython的启动模式

在启动后，MicroPython会先运行 `boot.py` 文件，加载用户驱动，然后在运行`main.py`，执行用户程序。可以将用户程序放在`main.py`中，也可以在`main.py`中再加载其它的文件。

常见故障

在使用过程中，我们需要注意下面问题，避免造成文件系统破坏、数据丢失。

- 取下数据线前，需要先U盘那样安全删除硬件，弹出PYBFLASH磁盘，否则可能会造成文件系统破坏，特别在修改了文件或复制新文件到PYBFLASH磁盘后。
- 不要轻易按复位键，这样会造成当前的USB通讯中断。一般的问题，可以通过按下Ctrl-D软复位接近。

出厂模式

使用时间长了，因为各种原因可能会出现故障，造成无法正常启动，不能进入REPL，文件系统破坏等现象，这时就需要通过出厂模式进行恢复。

进入出厂模式的方法是：

- 按下复位键（RESET）的同时，按住用户按键SW。
- 然后保持用户按键不放，释放复位键。
- 这时LED将循环显示：绿—》黄—》绿+黄—》灭
- 等黄绿灯同时亮时，松开用户键，这时黄绿灯会同时快速闪4次
- 然后红灯亮起（这时红绿黄三个灯同时亮）
- 红灯灭，开始进行恢复到出厂状态
- 所有灯都灭，恢复出厂设置完成。

恢复出厂设置后，PYBFLASH中的内容会丢失，变为默认文件。

升级固件

MicroPython的更新速度很快，每次更新都会带来一些新的功能，修正错误。所以掌握 MicroPython 的固件升级方法是有必要的。

PYB Nano支持下面几种升级方法：

- 通过DFU模式升级
- 通过SWD方式升级

使用DFU模式，需要安装ST的DfuSe_demo软件（Windows）或者dfu-util（Linux）。使用SWD需要将开发板的SWD接口（PA13/PA14）连接到编程器，通过编程软件下载。

参考资料

以上是 MicroPython 和 PYB Nano 的最基础知识，掌握后大家就可以逐步深入了。下面是一些参考网站和论坛，供大家参考。

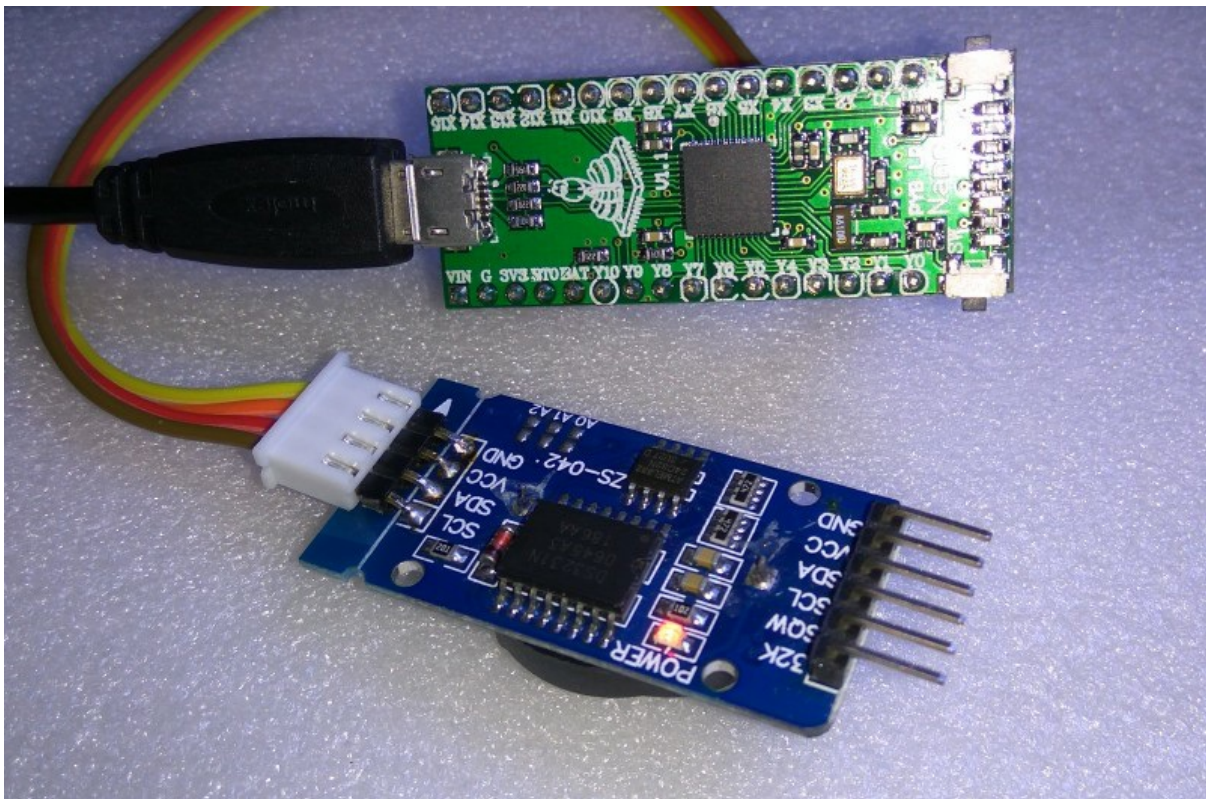
- [MicroPython的官方网站](#)
- [官方英文论坛](#)
- [MicroPython源码](#)
- [官方在线英文帮助](#)
- [官方维基](#)

- [MicroPython中文论坛](#)
- [PYB Nano 版块](#)
- [PYB Nano 原理图、BOM](#)
- [PYB Nano 开发板最新固件](#)
- [MicroPython中文教程](#)

PYB Nano连接DS3231时钟模块

接线方式和连线图如下:

DS3231	PYB Nano
GND	GND
VCC	3V3
SDA	Y0/PB9
SCL	Y1/PB8



将DS3231库导入，就可以直接使用了。

```
>>> from DS3231 import DS3231
>>> ds=DS3231(1)
>>> ds.sec()
46
>>> ds.sec()
47
>>> ds.TIME()
[22, 35, 56]
>>> ds.DATE()
```

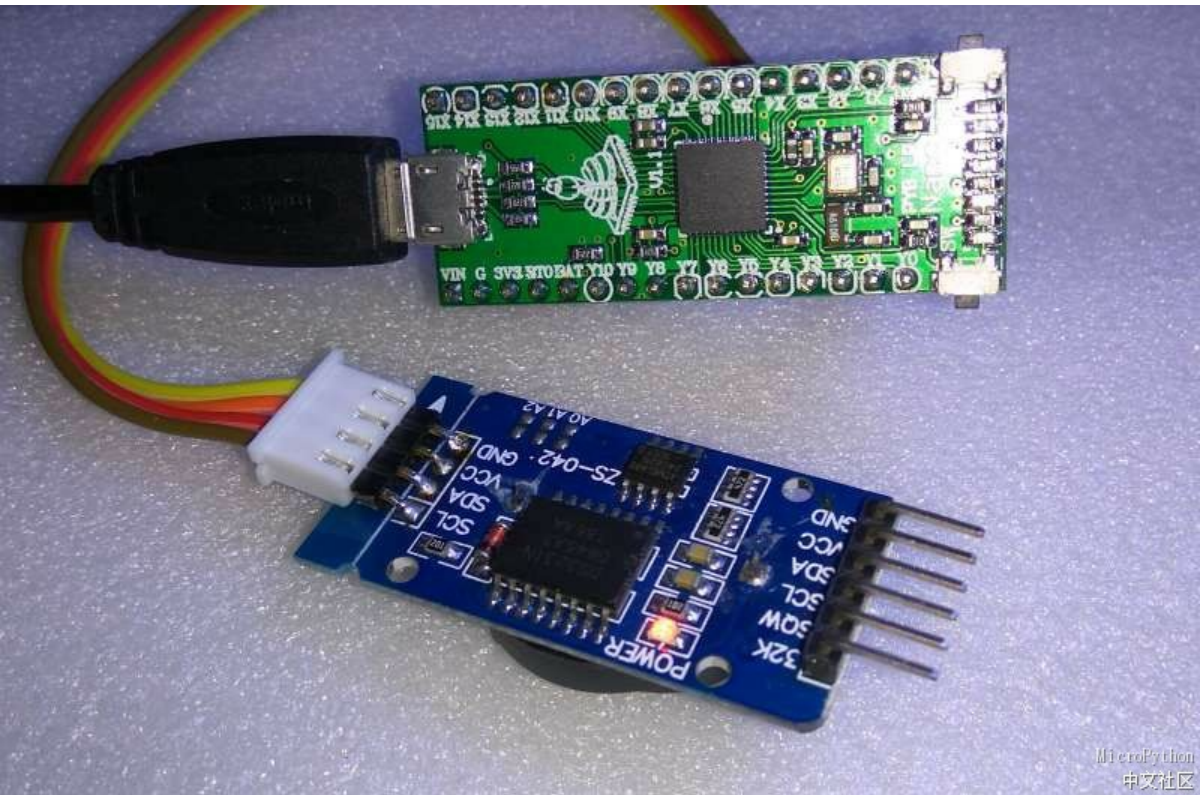
```
[16, 10, 20]  
>>> ds.TEMP()  
26.5  
>>>
```

完整程序见[论坛](#)

DS3231模块上还有一个EEPROM AT24C32，可以保存数据，它也是I2C接口的，和DS3231共用I2C。

接线方式和连线图和DS3231一样，就照搬过来了：

DS3231	PYB Nano
GND	GND
VCC	3V3
SDA	Y0/PB9
SCL	Y1/PB8



```
from pyb import I2C

_24L64_ADDR = const(0x57)

class _24L64(object):
    def __init__(self, i2c_num, i2c_addr=_24L64_ADDR,
i2c_baud=100000):
        self.i2c_addr = i2c_addr
        self.i2c_buad = i2c_baud
        self.r = bytearray(2)
        self.w = bytearray(3)
        self.i2c = I2C(i2c_num, I2C.MASTER, baudrate = i2c_baud)
```

```
def read(self, addr):
    self.r[0] = addr//256
    self.r[1] = addr%256
    self.i2c.send(self.r, self.i2c_addr)
    return self.i2c.recv(1, self.i2c_addr)[0]

def write(self, addr, dat):
    self.w[0] = addr//256
    self.w[1] = addr%256
    self.w[2] = dat
    self.i2c.send(self.w, self.i2c_addr)
```

运行效果

```
>>> from _24L64 import _24L64
>>> ee=_24L64(1)
>>> ee.read(0)
0
>>> ee.read(1)
2
>>> ee.write(1, 5)
>>> ee.read(1)
5
```

程序请到[论坛下载](#)

BMP180也是DIY时常用的气压传感器，它的特点是体积小，接口简单（I2C）。下面演示在PYB Nano上使用BMP180。

接线方式和连线图如下：

BMP180	PYB Nano
GND	GND
VCC	3V3
SDA	Y0/PB9
SCL	Y1/PB8



然后导入BMP180的库，就可以使用了。

```
from bmp180 import BMP180
b=BMP180(1)
b.getPress()
```

主要的函数有：

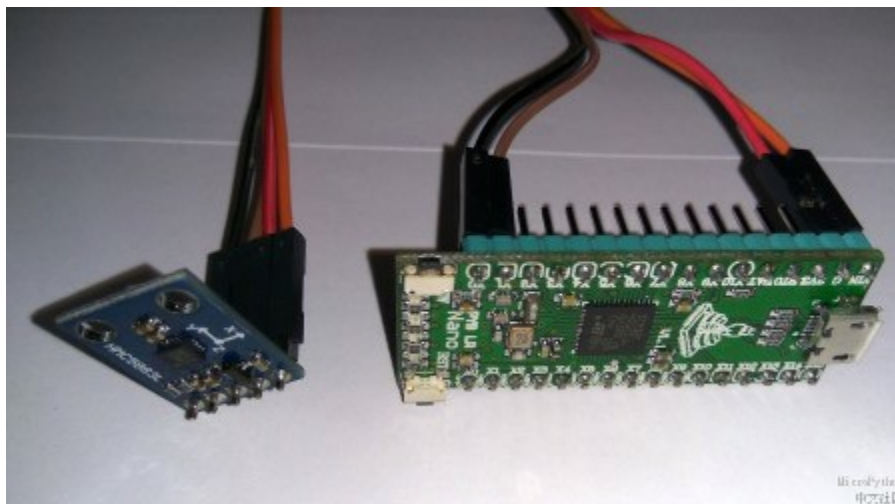
- b.getTemp(), 获取温度
- b.getPress(), 获取气压
- b.getAltitude(), 获取高度
- b.get(), 获取全部参数

完整BMP180库见[论坛](#)

HMC5883是一个I2C接口的磁传感器芯片，应用于低成本罗盘和磁场检测。

因为HMC5883模块上带有LD0，所以它的VCC接到了PYB Nano的VIN上（4.8V左右）。接线方式和连线图如下：

HMC5883	PYB Nano
GND	GND
VCC	3V3
SDA	Y0/PB9
SCL	Y1/PB8



```
>>> from HMC5883 import HMC5883
>>> h = HMC5883(1)
>>> while True:
...     pyb.delay(500)
...     print("%5d"%h.getX(), "%5d"%h.getY(), "%5d"%h.getZ())
...
65387    407     73
65384    406     74
65386    403     78
65386    403     78
65383    404     77
65384    406     76
65385    404     77
   56    402  65215
  124    351  65162
  117    341  65161
   44    313  65110
  149    375  65362
```


60	405	65528
65490	413	49
65424	421	28

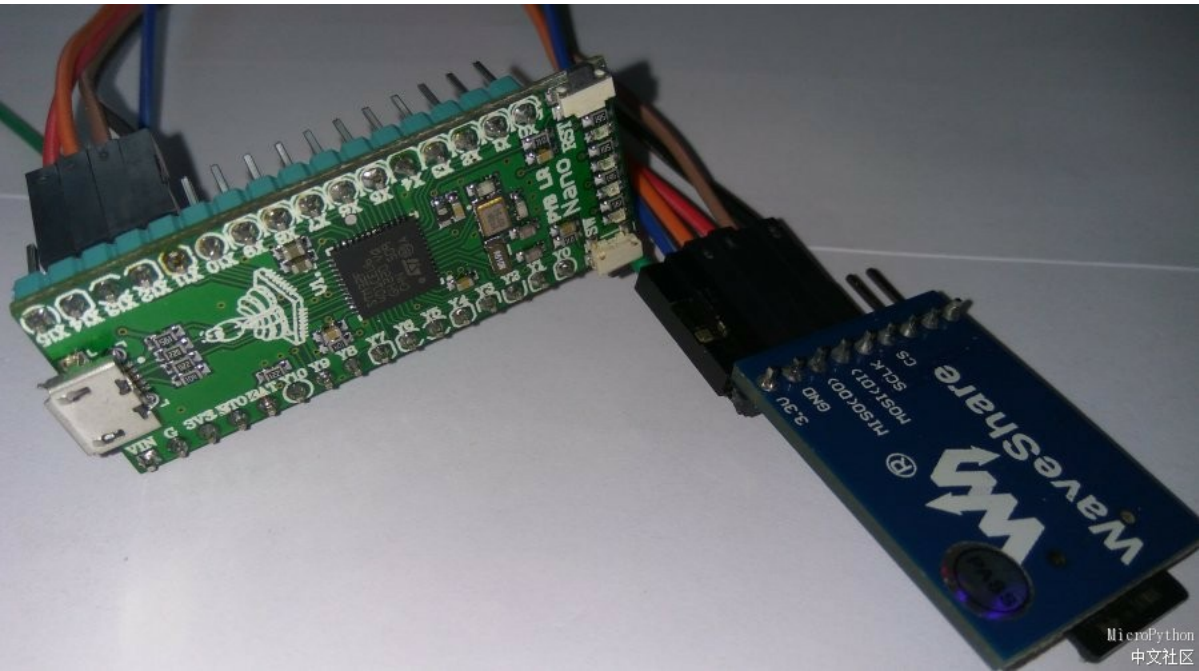
完整BMP180库见[论坛](#)

因为大小限制，所以STM32F401CEU6没有SD接口，因此PYB Nano上也没有了SD/macroSD，这给存储大数据带来一点不便。不过我们可以通过SPI接口挂载SD，下面介绍具体的连接方法。

通过SPI方式连接，需要6根线（包括电源）。

接线方式和连线图如下：

DS3231	PYB Nano
GND	GND
VCC	3V3
MOSI	X14/PB15
MISO	X13/PB14
SCK	X12/PB13
CS	X11/PB12



?

连接好后，将sdcard.py复制到PYB Nano中，在使用下面的命令进行挂载

```
import pyb, sdcard, os
sd = sdcard.SDCard(pyb.SPI(2), pyb.Pin('B12'))
pyb.mount(sd, '/sd2')
os.listdir('/')
```

运行效果图

[illegible]

下面的Demo演示了用定位器控制LED亮度。

使用到一个外部的定位器，并将定位器连用杜邦线接到PYB Nano的引脚：X9/PB1。

然后输入下面的程序

```
>>> from pyb import ADC, Pin
>>> adc = ADC(Pin('X9'))
>>> while True:
...     dat = adc.read()
...     pyb.LED(1).intensity(dat>>4)
...
...     print(dat)
...     pyb.delay(200)
```

改变定位器，就会在屏幕上输出当前ADC的结果，同时会改变LED1的亮度。

注：

- 上面的X9可以改为X0-X9（支持ADC功能）。
- LED(1)可以改为1-4，它们都支持亮度调整功能。

ST 的 Nucelo64 系列开发板



[开发板网站](#)

ST 的 Nucelo64 系列开发板



[开发板网站](#)

NUCLEO-F411的micropython官方版本是不支持macroSD的。在配置文件中可以看到，只有下面三行：

```
#define MICROPY_HW_HAS_SWITCH          (1)
#define MICROPY_HW_HAS_FLASH           (1)
#define MICROPY_HW_ENABLE_RTC          (1)
```

没有SD卡的配置。因此，我们需要自己加上。从PYBV10的配置中，复制SD卡的配置：

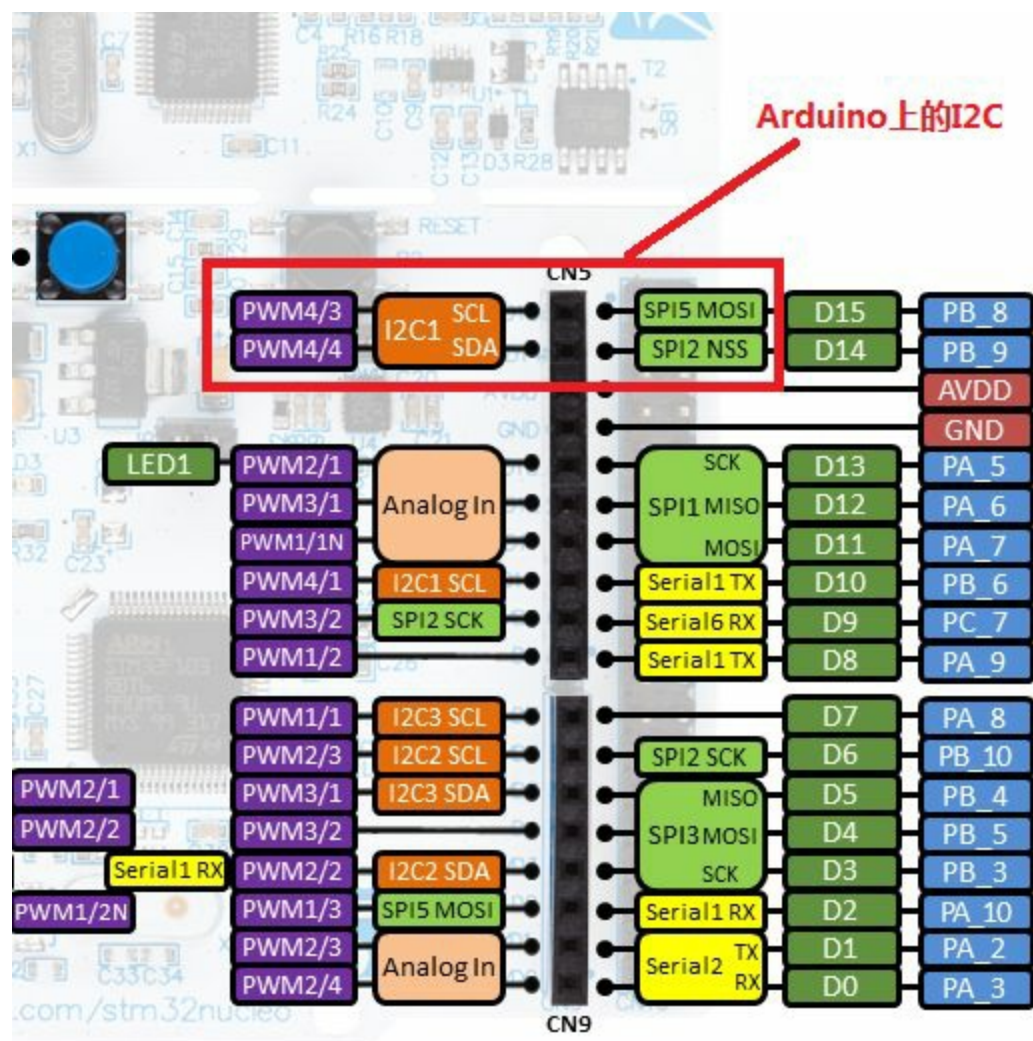
```
#define MICROPY_HW_HAS_SDCARD          (1)
```

然后加上下面定义

```
// SD card detect switch
#define MICROPY_HW_SDCARD_DETECT_PIN
    (pin_A8)
#define MICROPY_HW_SDCARD_DETECT_PULL
    (GPIO_PULLUP)
#define MICROPY_HW_SDCARD_DETECT_PRESENT
    (GPIO_PIN_RESET)
```

注意在编译前，需要先clean，否则可能会编译出错。编译后更新固件，并连上我刚做好的扩展板，插入SD卡，最后连接USB线，可以识别出SD卡，一起运行正常。😊

在MicroPython的NUCLE0_F411RE版本中，I2C默认的引脚使用了PB6/PB7，和Arduino上使用的不一致，这样使用起来不是太方便。



然后将I2C1相关的定义进行修改

```
#define MICROPY_HW_I2C1_SCL (pin_B8)           // Arduino D15
#define MICROPY_HW_I2C1_SDA (pin_B9)           //           D14
```

然后重新编译源码，下载后，在连接一个I2C模块（我使用了DS3231模块）就可以发现，I2C1的确已经改为了B8/B9上了。

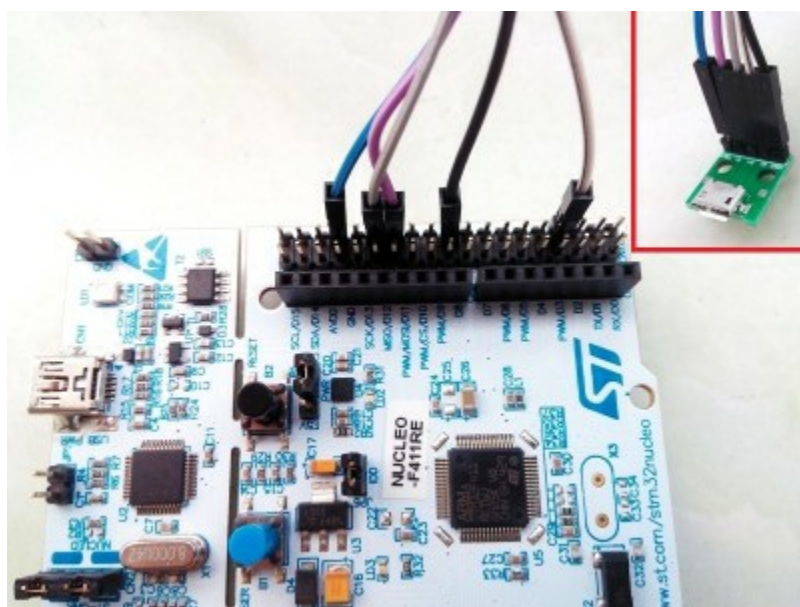
在NUCLEO-F411RE上使用MicroPython

在MicroPython的源码中，已经包含了NUCLEO-F411RE开发板，大家只要重新编译一下，将固件下载进去就可以运行。编译的方法请参考小钢炮那个帖子，就不重复了。下面说明其他需要注意问题。

- 虽然NUCLEO-F411RE开发板带有Mbed编程接口，但是不能直接将HEX文件复制到Mbed磁盘进行更新，需要用STM32 ST-LINK Utility或其他软件下载。
- STM32F411是有USB功能的，但是NUCLEO-F411RE开发板没有预留USB接口（不算STLink的）。虽说通过STlink的串口是可以运行MicroPython，但是这样无法使用PYFlash磁盘，很多驱动程序就无法复制进去。幸好它将USB的GPIO引出来了，我们通过一个macroUSB转接板就可以使用USB功能。具体接线如下：

PA12 — DP
PA11 — DM
ID — GND
AVDD — VBUS

开发板的供电跳线不用改，还是U5V，但是STLink上的miniUSB还是需要连接，不然单片机的RST会被STLink拉低。

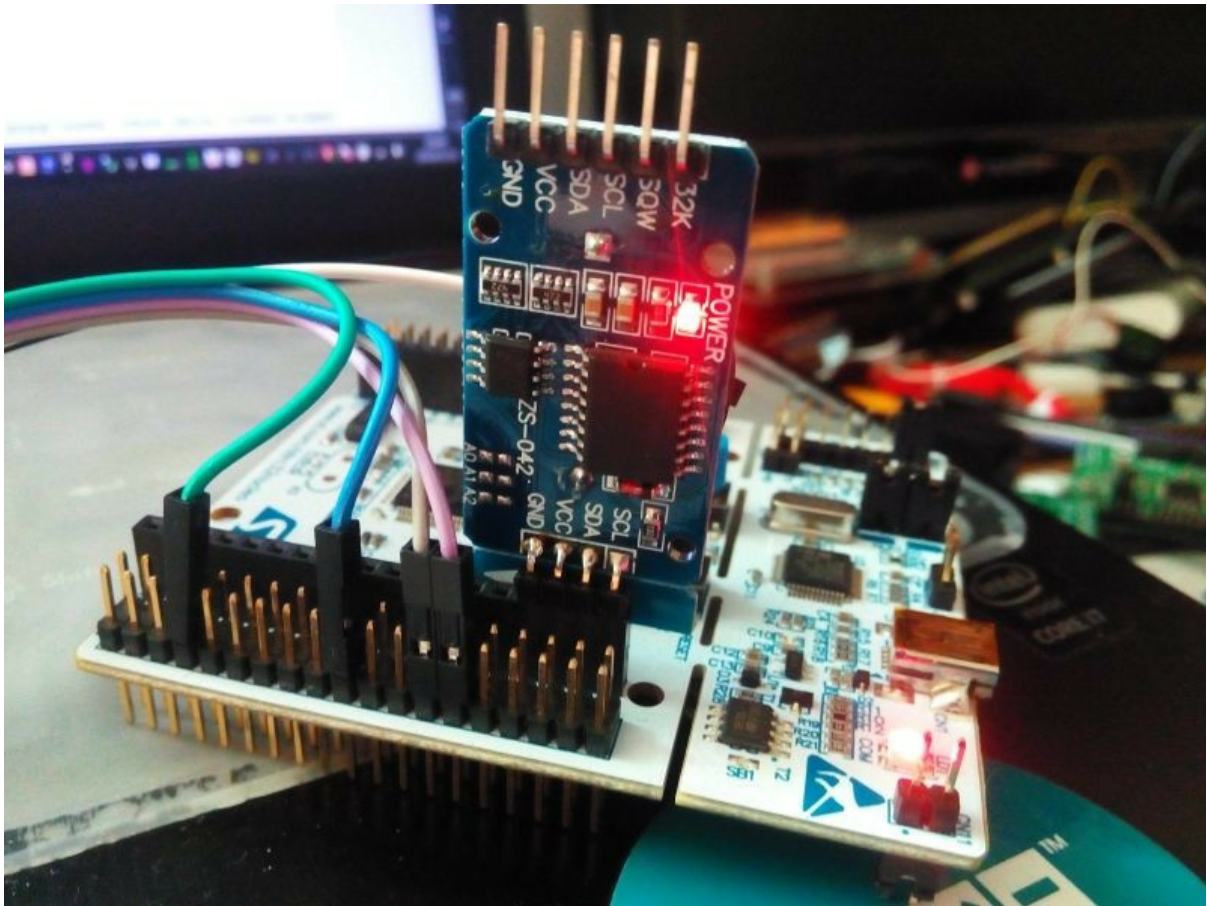


ST 的 Nucelo64 系列开发板



[开发板网站](#)

DS3231可以直接插在Arduino排座上，顺序正好一样。



然后将DS3231.py库（完整程序见[论坛](#)）复制到PYFLASH中，运行下面的程序，可以看到可以正确的读取时间和温度，说明I2C1工作正常。

```
>>> from DS3231 import DS3231
>>> ds = DS3231(1)
>>> ds.TEMP()
25.75
>>> ds.TIME()
[16, 35, 38]
>>> ds.DATE()
[16, 5, 15]
>>> while True:
...     pyb.delay(1000)
...     ds.sec()
...
14
15
16
17
18
```

Nucleo开发板的默认开机程序是使用按键控制LED的频率，下面使用MicroPython实现同样的功能。

```
from pyb import Pin, Timer
tm = Timer(2, freq=1)
led = tm.channel(1, Timer.PWM, pin=Pin.cpu.A5)
led.pulse_width_percent(50)

f = 1
def sw_isr():
    global f
    f=(f+5)%10
    tm.freq(f)
    led.pulse_width_percent(50)

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

ST 的 Nucleo64 系列开发板



[开发板网站](#)

ST 的 Nucelo144 系列开发板



[开发板网站](#)

ST 的 Nucelo144 系列开发板



[开发板网站](#)

NUCLEO_F746ZG开发板，这是一个NUCLEO-144系列的开发板，并不在micropython直接支持的列表中，但是同型号系列中的STM32F746DISC是支持micropython。不过STM32F746DISC的固件并不能直接用在这个开发板上，一个是芯片型号不同，另外就是时钟配置不同。

虽然不能使用使用STM32F746DISC的固件，但是我们可以通过修改这个开发板的配置，实现程序的移植。

主要需要修改的地方有：

- 时钟
- LED
- 按键
- GPIO
- I2C
- SPI
- UART

等。因为时间原因，先只修改了前面3个，后面的等有空了在进行。

修改配置后，重新编译代码，得到初步可以运行的固件，经过在NUCLEO_F746ZG开发板上初步测试，的确可以使用了。

```
#define STM32F7DISC

#define MICROPY_HW_BOARD_NAME      "NUCLEO_F746ZG"
#define MICROPY_HW_MCU_NAME        "STM32F746"

#define MICROPY_HW_HAS_SWITCH      (1)
#define MICROPY_HW_HAS_FLASH      (1)
#define MICROPY_HW_HAS_SDCARD      (0)
#define MICROPY_HW_HAS_MMA7660    (0)
#define MICROPY_HW_HAS_LIS3DSH    (0)
#define MICROPY_HW_HAS_LCD        (0)
#define MICROPY_HW_ENABLE_RNG      (1)
#define MICROPY_HW_ENABLE_RTC      (1)
#define MICROPY_HW_ENABLE_TIMER    (1)
#define MICROPY_HW_ENABLE_SERVO    (0)
#define MICROPY_HW_ENABLE_DAC      (0)
#define MICROPY_HW_ENABLE_CAN      (1)

// HSE is 25MHz
#define MICROPY_HW_CLK_PLLM (8)
#define MICROPY_HW_CLK_PLLN (336)
#define MICROPY_HW_CLK_PLLP (RCC_PLLP_DIV2)
```

```

#define MICROPY_HW_CLK_PLLQ (7)

#define MICROPY_HW_FLASH_LATENCY FLASH_LATENCY_6

// UART config
#define MICROPY_HW_UART2_TX_PORT (GPIOD)
#define MICROPY_HW_UART2_TX_PIN (GPIO_PIN_5)
#define MICROPY_HW_UART2_RX_PORT (GPIOD)
#define MICROPY_HW_UART2_RX_PIN (GPIO_PIN_6)

// #define MICROPY_HW_UART6_PORT (GPIOC)
// #define MICROPY_HW_UART6_PINS (GPIO_PIN_6 | GPIO_PIN_7)
// #define MICROPY_HW_UART7_PORT (GPIOF)
// #define MICROPY_HW_UART7_PINS (GPIO_PIN_6 | GPIO_PIN_7)

#define MICROPY_HW_UART_REPL PYB_UART_1
#define MICROPY_HW_UART_REPL_BAUD 115200

// I2C busses
#define MICROPY_HW_I2C1_SCL (pin_B8)
#define MICROPY_HW_I2C1_SDA (pin_B9)

#define MICROPY_HW_I2C3_SCL (pin_H7)
#define MICROPY_HW_I2C3_SDA (pin_H8)

// The STM32F7 uses a TIMINGR register which is configured using
an Excel
// Spreadsheet from AN4235:
http://www.st.com/web/en/catalog/tools/PF258335
// We use an array of baudrates and corresponding TIMINGR values.
//
// The value 0x40912732 was obtained from the
DISCOVERY_I2Cx_TIMING constant
// defined in the STM32F7Cube file Drivers/BSP/STM32F746G-
Discovery/stm32f7456g_discovery.h
#define MICROPY_HW_I2C_BAUDRATE_TIMING {{100000, 0x40912732}}
#define MICROPY_HW_I2C_BAUDRATE_DEFAULT 100000
#define MICROPY_HW_I2C_BAUDRATE_MAX 100000

// SPI
#define MICROPY_HW_SPI2_NSS (pin_I0)
#define MICROPY_HW_SPI2_SCK (pin_I1)
#define MICROPY_HW_SPI2_MISO (pin_B14)
#define MICROPY_HW_SPI2_MOSI (pin_B15)

// USRSW is pulled low. Pressing the button makes the input go
high.
#define MICROPY_HW_USRSW_PIN (pin_C13)
#define MICROPY_HW_USRSW_PULL (GPIO_NOPULL)
#define MICROPY_HW_USRSW_EXTI_MODE (GPIO_MODE_IT_RISING)
#define MICROPY_HW_USRSW_PRESSED (1)

// LEDs
#define MICROPY_HW_LED1 (pin_B0) // blue

```

```

#define MICROPY_HW_LED2                (pin_B7) // red
#define MICROPY_HW_LED3                (pin_B14) // green
// #define MICROPY_HW_LED1_PWM         { TIM4, 2, TIM_CHANNEL_2,
GPIO_AF2_TIM4 }
// #define MICROPY_HW_LED2_PWM         { TIM1, 2, TIM_CHANNEL_2,
GPIO_AF1_TIM1 }
//
#define MICROPY_HW_LED_OTYPE            (GPIO_MODE_OUTPUT_PP)
#define MICROPY_HW_LED_ON(pin)         (pin->gpio->BSRR = pin-
>pin_mask)
#define MICROPY_HW_LED_OFF(pin)        (pin->gpio->BSRR = (pin-
>pin_mask << 16))

// USB config (CN13 - USB OTG FS)
// The Hardware VBUS detect only works on pin PA9. The STM32F7
Discovery uses
// PA9 for VCP_TX functionality and connects the VBUS to pin J12
(so software
// only detect). So we don't define the VBUS detect pin since that
requires PA9.

/* #define MICROPY_HW_USB_VBUS_DETECT_PIN (pin_J12) */
// #define MICROPY_HW_USB_OTG_ID_PIN      (pin_A10)

```

ST 的 Nucelo144 系列开发板



[开发板网站](#)

下载MicroPython源码后，展开到一个目录，然后进入/stmal/board目录，新建一个NUCLEO_F767ZI目录，并将STM32F7DISC目录下的文件复制过来。

先打开文件stm32f7xx_hal_conf.h，将 #define HSE_VALUE 后的数字从25000000改为8000000。因为NUCLEO_F767ZI上使用的时钟频率是8MHz。

再打开文件mpconfigboard.h，修改部分配置，主要修改内容有（只列出主要部分）：

```
#define MICROPY_HW_BOARD_NAME
"NUCLEO_F767ZI"
#define MICROPY_HW_MCU_NAME           "STM32F767"

#define MICROPY_HW_HAS_SDCARD         (0)

#define MICROPY_HW_CLK_PLLM (8)

#define MICROPY_HW_UART3_PORT         (GPIOD)
#define MICROPY_HW_UART3_PINS         (GPIO_PIN_8 |
GPIO_PIN_9)

#define MICROPY_HW_UART_REPL          PYB_UART_3

#define MICROPY_HW_I2C4_SCL            (pin_F14)
#define MICROPY_HW_I2C4_SDA           (pin_F15)

#define MICROPY_HW_USRSW_PIN          (pin_C13)

#define MICROPY_HW_LED1                (pin_B0)    //
green
#define MICROPY_HW_LED2                (pin_B7)    //
blue
#define MICROPY_HW_LED3                (pin_B14)   //
red
```

编译源码前，需要安装arm-none-eabi-gcc编译器，编译器在<https://launchpad.net/gcc-arm-embedded>，大家可以根据自己的操作系统选择合适的版本安装。

安装编译器后，就可以在stmhal目录下，输入make进行编译了。在windows下，如果没有安装过make，还需要安装msys或者其他带有make工具的环境。推荐在Linux下编译代码，因为编译速度比Windows下快很多。如果在widnwos下编译，可以使用多线程编译，比默认的单线程也要快一点。如：`make BOARD=NUCLEO_F767ZI -j8`

编译后，就会在stmhal目录下产生一个build-NUCLEO_F767ZI目录，里面包含有编译的中间文件和最终二进制文件。

用STM32 ST-LINK Utility或者其他软件，加载编译后的firmware.hex文件，通过STlink下载固件到NUCLEO_F767ZI开发板，就可以体验MicroPython了。可以通过STlink的串口或者NUCLEO_F767ZI开发板的macroUSB（CN13）连接到终端软件，连接时将波特率设置为115200。如果通过macroUSB（CN13）连接，会自动安装一个pybflash虚拟磁盘，还会安装一个虚拟串口，这个串口的驱动就在pybflash磁盘上。

在MicroPython中用PWM控制LED的亮度，需要使用Timer和Pin两个模块

```
from pyb import Pin, Timer

# led1使用TIM1_CH2
tm1=Timer(1, freq=1000)
led1=tm1.channel(2, Timer.PWM, pin=Pin('B0'))

# 设置亮度（绿），0最亮，100最暗
led1.pulse_width_percent(0)
led1.pulse_width_percent(100)

# led2（兰）使用TIM4_CH2
tm4=Timer(4, freq=1000)
led2=tm4.channel(2, Timer.PWM, pin=Pin('B7'))

# 设置亮度，0最暗，100最亮
led2.pulse_width_percent(10)

# led3（红）使用TIM4_CH2N

tm8=Timer(8, freq=1000)
led3=tm8.channel(2, Timer.PWM, pin=Pin('B14'))

# 设置亮度，0最亮，100最暗
led3.pulse_width_percent(90)

#led3也可以使用TIM12_CH1
tm12=Timer(12, freq=1000)
led3=tm12.channel(1, Timer.PWM, pin=Pin('B14'))
led3.pulse_width_percent(0)
```


ST 的 Nucelo144 系列开发板



[开发板网站](#)

ST Discovery系列开发板



[开发板网站](#)

ST Discovery系列开发板



[开发板网站](#)

ST Discovery系列开发板



[开发板网站](#)

ST Discovery系列开发板



[开发板网站](http://www.st.com/stm3244-discovery)

聚码科技的小钢炮开发板，带有很多传感器和蓝牙BLE



周末抽空将气压计LPS25H的驱动完成了，可以很方便的读取温度、气压值。目前驱动只做了查询部分，没有处理中断部分。这个等后面有空的时候在做了。

首先import LPS25H库，然后定义lps25对象，就可以使用lps25.PRESS()读取气压， lps25.TEMP()读取温度。

```
MicroPython v1.7 on 2016-04-17; CANNON with STM32F401xE
Type "help()" for more information.
>>> from LPS25H import LPS25H
>>> lps25=LPS25H(1, 1)
>>> lps25.PRESS()
1021.375
>>> lps25.PRESS()
1021.313
>>> lps25.PRESS()
1021.25
>>> lps25.PRESS()
1021.25
>>> lps25.TEMP()
21.99375
>>> lps25.TEMP()
21.99792
>>>
```

定义LPS25H对象时，使用

```
lps25=LPS25H(1, 1)
```

其中第一个参数是I2C模块的序号，第二个参数是SA0，在小钢炮上，SA0=1

此外还增加了休眠处理，使用lps25.poweron()工作，lps25.poweroff()休眠。

[程序下载](#)

小钢炮开发板带有HTS221温湿度传感器，这个温湿度传感器和STH22/si7002不同，不能直接读取温度和湿度，需要通过插值进行计算。

下面是我写的HTS221驱动，可以通过函数直接读取温度和湿度常数。

```
# HTS221 Humidity and temperature micropython drive
# Author: shaoziyang
# 2016.4

import pyb
from pyb import I2C

HTS_I2C_ADDR = 0x5F

class HTS221(object):
    def __init__(self, i2cn):
        self.i2c = I2C(i2cn, I2C.MASTER, baudrate = 100000)
        # HTS221 Temp Calibration registers
        self.T0_OUT = self.get2Reg(0x3C)
        self.T1_OUT = self.get2Reg(0x3E)
        if self.T0_OUT >= 0x8000 :
            self.T0_OUT -= 65536
        if self.T1_OUT >= 0x8000 :
            self.T1_OUT -= 65536
        t1 = self.getReg(0x35)
        self.T0_degC = (self.getReg(0x32) + (t1%4)*256)/8
        self.T1_degC = (self.getReg(0x33) + ((t1%16)/4)*256)/8
        # HTS221 Humi Calibration registers
        self.H0_OUT = self.get2Reg(0x36)
        self.H1_OUT = self.get2Reg(0x3A)
        self.H0_rH = self.getReg(0x30)/2
        self.H1_rH = self.getReg(0x31)/2
        # set av conf: T=4 H=8
        self.setReg(0x81, 0x10)
        # set CTRL_REG1: PD=1 BDU=1 ODR=1
        self.setReg(0x85, 0x20)

    def setReg(self, dat, reg):
        buf = bytearray(2)
        buf[0] = reg
        buf[1] = dat
        i2c = self.i2c
        i2c.send(buf, HTS_I2C_ADDR)

    def getReg(self, reg):
        i2c = self.i2c
        i2c.send(reg, HTS_I2C_ADDR)
        t = i2c.recv(1, HTS_I2C_ADDR)
        return t[0]
```



```

def get2Reg(self, reg):
    a = self.getReg(reg)
    b = self.getReg(reg + 1)
    return a + b * 256

def av(self, av=''):
    i2c = self.i2c
    if av != '':
        #buf = bytearray(2)
        #buf[0] = 0x10;
        #buf[1] = av;
        #i2c.send(buf, HTS_I2C_ADDR)
        self.setReg(av, 0x10)
    else:
        #i2c.send(0x10, HTS_I2C_ADDR)
        #t = i2c.recv(1, HTS_I2C_ADDR)
        #return t[0]
        return self.getReg(0x10)

def T0_OUT(self):
    return self.T0_OUT

def T1_OUT(self):
    return self.T1_OUT

def T0_degC(self):
    return self.T0_degC

def T1_degC(self):
    return self.T1_degC

# calculate Temperature
def getTemp(self):
    t = self.get2Reg(0x2A)
    return self.T0_degC + (self.T1_degC - self.T0_degC) * (t -
self.T0_OUT) / (self.T1_OUT - self.T0_OUT)

def H0_OUT(self):
    return self.H0_OUT

def H1_OUT(self):
    return self.H1_OUT

def H0_rH(self):
    return self.H0_rH

def H1_rH(self):
    return self.H1_rH

```

```
# calculate Humidity
def getHumi(self):
    t = self.get2Reg(0x28)
    return self.H0_rH + (self.H1_rH - self.H0_rH) * (t -
self.H0_OUT) / (self.H1_OUT - self.H0_OUT)
```

运行效果

```
PYB: sync filesystems
PYB: soft reboot
MicroPython v1.7 on 2016-04-17; CANNON with STM32F401xE
Type "help()" for more information.
>>> from hts221 import HTS221
>>> hts=HTS221(1)
>>> hts.getTemp()
22.95221
>>> hts.getHumi()
82.62943
>>>
```

小钢炮（CANNON）开发板是最近比较热门的一个蓝牙开发板，它是XXXXX（为了避免告，此处省略XX字）。这里介绍小钢炮开发板，主要因为它使用了STM32F401RE这个MCU，而这个MCU是MicroPython支持的型号，而且这个开发板比较容易获取，从去年年底开始做活动，到现在也还有不少地方可以申请或低价购买。所以下面就介绍在小钢炮开发板上移植MicroPython的方法。

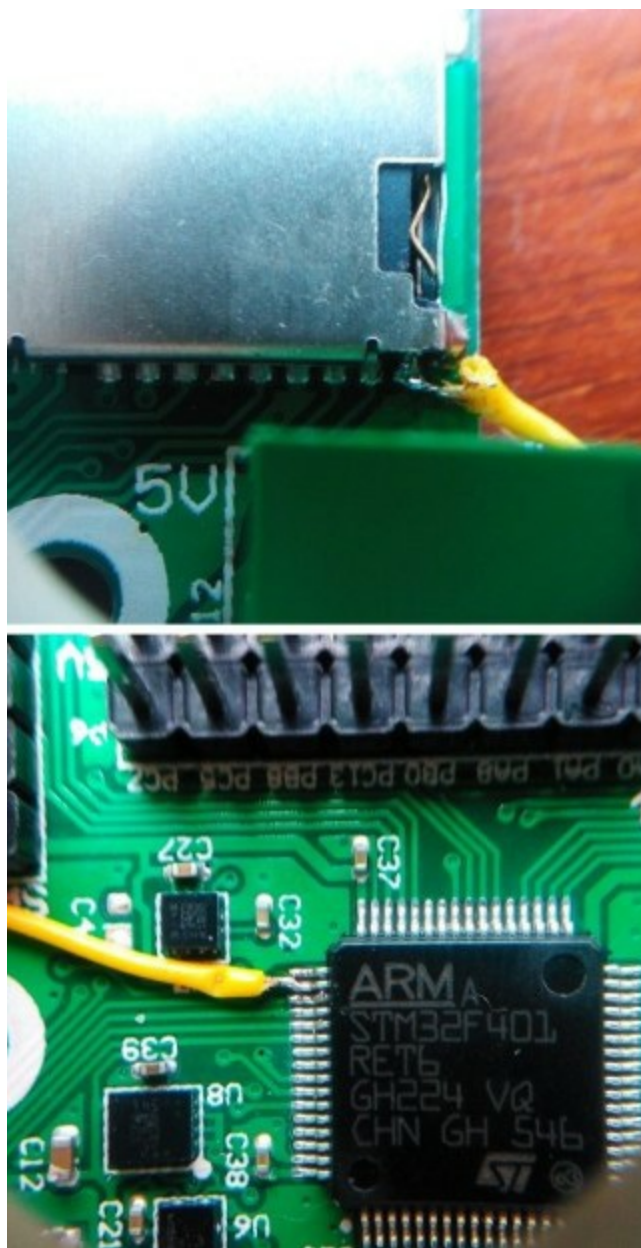
1. 首先要下载并安装GNU Tools for ARM Embedded Processors。
 1. <https://launchpad.net/gcc-arm-embedded>
2. 下载并安装ST的DfuSe软件，<http://www.st.com/web/en/catalog/tools/FM147/CL1794/>
3. 下载MicroPython的源码，[micropython-master.zip](#)。
4. 展开MicroPython源码，打开 stmhal\boards\ 目录
5. 新建一个CANNON目录，将NUCLEO_F401RE下的文件复制到CANNON目录下
6. 如果GNU Tools for ARM已经添加到系统路径，就可以跳到步骤8，直接编译
7. 打开 stmhal 下的 makefile 文件，修改 `CROSS_COMPILE = arm-none-eabi-` 这一行，在 arm-none-eabi- 前添加编译器的实际路径，注意路径需要使用右斜杠
8. 在 stmhal 目录下，输入 `make BOARD=CANNON`，就可以编译了。不过这时编译出的代码是不能运行的，因为两个板子的参数不同。
9. 打开 stmhal\boards\CANNON目录，先修改文件 `stm32f4xx_hal_conf.h`
 - 找到`#define HSI_VALUE ((uint32_t)8000000)`，将数字8000000改为16000000，因为小钢炮使用了16M的外部时钟
10. 打开文件 `mpconfigboard.h`
 - 找到`#define MICROPY_HW_CLK_PLLM (8)`，将数字8改为16
 - 修改`#define MICROPY_HW_HAS_SWITCH (1)`将1改为0，因为小钢炮上没有用户按键
 - 修改`#define MICROPY_HW_LED1 (pin_A5) // Green LD2 LED on Nucleo`，将pin_A5改为pin_B3，因为两个板子的LED使用不同的GPIO
 - 修改`#define MICROPY_HW_LED_ON(pin) (pin->gpio->BSRRL = pin->pin_mask)`，将BSRRL改为BSRRH
 - 修改`#define MICROPY_HW_LED_OFF(pin) (pin->gpio->BSRRH = pin->pin_mask)`，将BSRRH改为BSRRL，这是因为两个板子的LED驱动方式不同
 - 添加下面RTC的定义

```
// The pyboard has a 32kHz crystal for the RTC
#define MICROPY_HW_RTC_USE_LSE      (1)
#define MICROPY_HW_RTC_USE_US      (0)
#define MICROPY_HW_RTC_USE_CALOUT  (1)
```

- 添加sdcard的定义，因为小钢炮支持TF(macroSD)卡。如果不想改线，或者不需要使用TF卡，可以忽略这一步和下面一步。

```
#define MICROPY_HW_HAS_SDCARD      (1)
// SD card detect switch
#define MICROPY_HW_SDCARD_DETECT_PIN      (pin_A15)
#define MICROPY_HW_SDCARD_DETECT_PULL    (GPIO_PULLUP)
#define MICROPY_HW_SDCARD_DETECT_PRESENT (GPIO_PIN_RESET)
```

- 小钢炮开发板没有做TF卡的插入检测，所以需要自己飞一根线。开发板上A15(50)和B4(56)是空脚，我选择了A15，因为它更容易焊接一些。



- 打开文件pins.cvs，这里预定义了GPIO的名称
 - 修改LED的GPIO为PB3
 - 修改SW的GPIO为PC13
 - 如果还有时间和精力，可以适当修改其他GPIO
- 11. 现在可以再次编译源文件了。编译时建议在Linux下编译，因为速度快很多，在windows下编译速度很慢，需要等数分钟。
- 12. 准备3个短路块，连接P1，将BOOT0连接到VCC，BOOT1连接到GND。
- 13. 将开发板用macroUSB线连接到计算机，因为设置了BOOT0/BOOT1，所以上电后会进入DFU模式。在Windows下如果是第一次使用，会提示安装驱动，驱动程序就在DfuSe软件的安装

目录下。

14. 使用DfuSe打开编译后的dfu文件，并下载到开发板。
15. 将B00T0连接到GND，开发板重新上电。这时会自动安装USB磁盘，出现PYBFLASH驱动器。在windows下还会安装虚拟串口，如果找不到驱动程序，可以到新出现的PYBFLASH驱动器上查找。
16. 打开一个串口终端软件，如kitty、xshell、超级终端等，设置波特率为115200，就可以开始玩micropython了。

先试试直接控制LED

```
import pyb
pyb.LED(1).on()
pyb.LED(1).off()
```

在试试用GPIO控制LED。

```
from pyb import Pin
led=Pin.cpu.B3
led.init(Pin.OUT_PP)
led.value(1)
led.value(0)
```

用PWM控制LED的亮度

```
from pyb import Pin, Timer

tm2=Timer(2, freq=100)
led=tm2.channel(2, Timer.PWM, pin=Pin.cpu.B3,
pulse_width=100)
led.pulse_width_percent(100)
led.pulse_width_percent(1)
```

呼吸灯

```
# main.py -- put your code here!

from pyb import Timer, Pin

tm2=Timer(2, freq=200)
led=tm2.channel(2, Timer.PWM, pin=Pin.cpu.B3)

# LED breathing lamp
ia = 1
```

```
da = 1
def fa(t):
    global ia, da
    if (ia==0) or (ia==100):
        da=100-da
    ia=(ia+da)%100
    led.pulse_width_percent(ia)

tm1=Timer(1, freq=100, callback=fa)
```

更多使用方法可以参考MicroPython网站的文档，以及【MicroPython】教程。

TI的CC3200 LaunchPAD开发板

经过几天的实验，终于在CC3200-LAUNCHXL开发板上成功运行了MicroPython，下面将具体方法分享给大家。

首先，需要做一些准备工作。需要下载几个软件：

1. [CCS Uniflash](#)：闪存编程工具
2. [CC3200SDK-SERVICEPACK](#)：系统服务包
3. [MicroPython源码](#) 或 [github上同步](#)
4. [GNU ARM Embedded Toolchain](#)

然后安装[CCS Uniflash](#)和[CC3200SDK-SERVICEPACK](#)，CC3200-LAUNCHXL开发板的驱动是FTDI驱动，如果安装过其它FTDI的驱动就不需要在安装了，如果没有安装那么在[CCS Uniflash](#)安装目录中带有这个驱动。不需要安装CCS6和其它软件。

另外需要注意CC3200-LAUNCHXL开发板硬件版本必须是4.1或以上，如果低于4.1就不用尝试了。

下面就开始具体的操作，因为步骤比较多，如果是第一次尝试，可能需要非常小心，避免操作错误。

编译源码，得到固件

编译源码和编译STM32的源码差不多，不同在于需要编译application和bootloader两个源码。

在micropython/cc3200目录下，输入下面的目录进行编译：

- `make BTARGET=application BTYPE=release`

- ```
BOARD=LAUNCHXL
```
- `make BTARGET=bootloader BTYPE=release`  
`BOARD=LAUNCHXL`

分别得到cc3200\build\LAUNCHXL\mcuimg.bin和cc3200\bootmgr\build\LAUNCHXL\bootloader.bin两个固件。

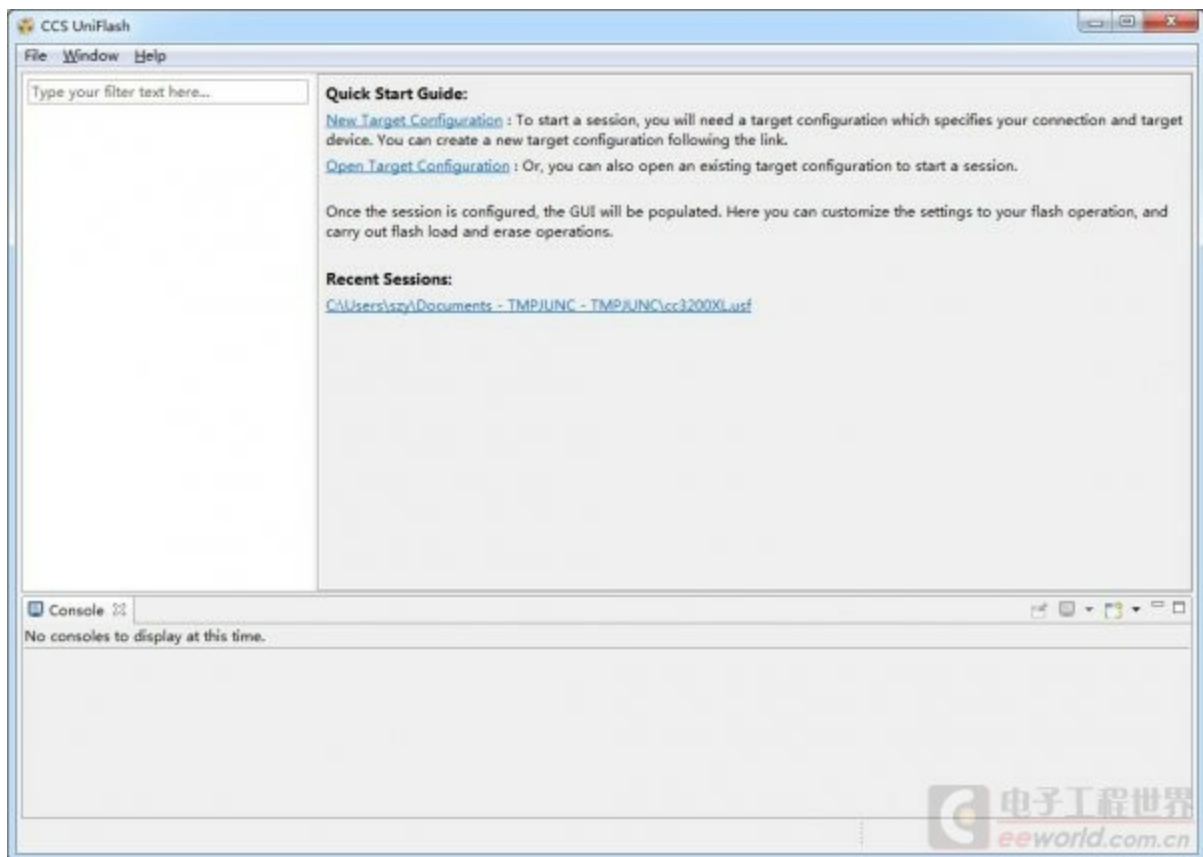
和STM32一样，在Linux下编译速度会快很多，在Windows下可以加上-j8加快编译，但比Linux下还是慢不少。

## 下载固件

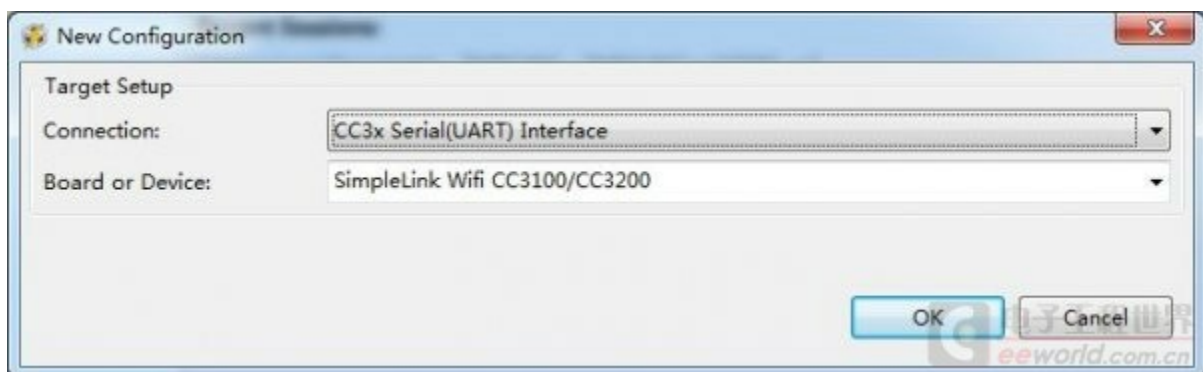
- 先将开发板的SOP2用短路块连接（如图黄色框所示），然后连接USB



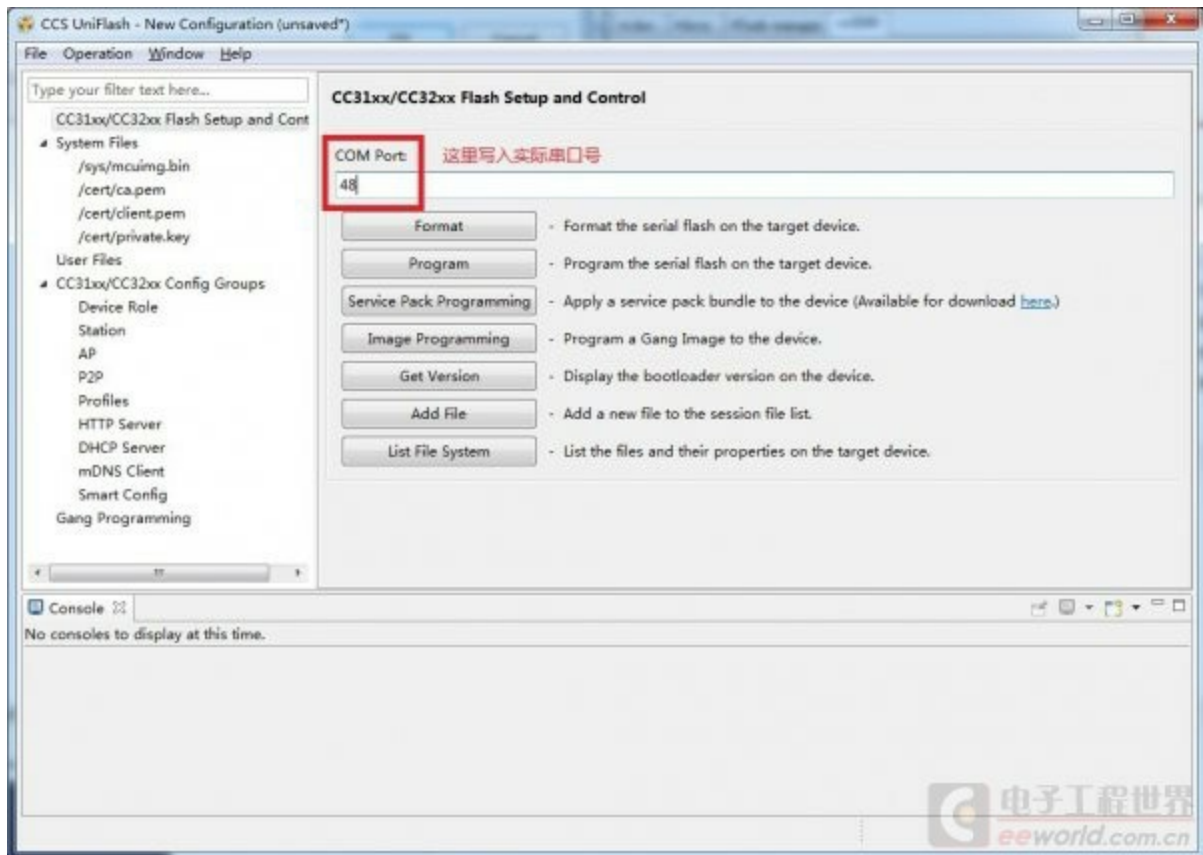
- 运行CCS Uniflash软件



- 新建一个配置，选择最下面的CC3X系列



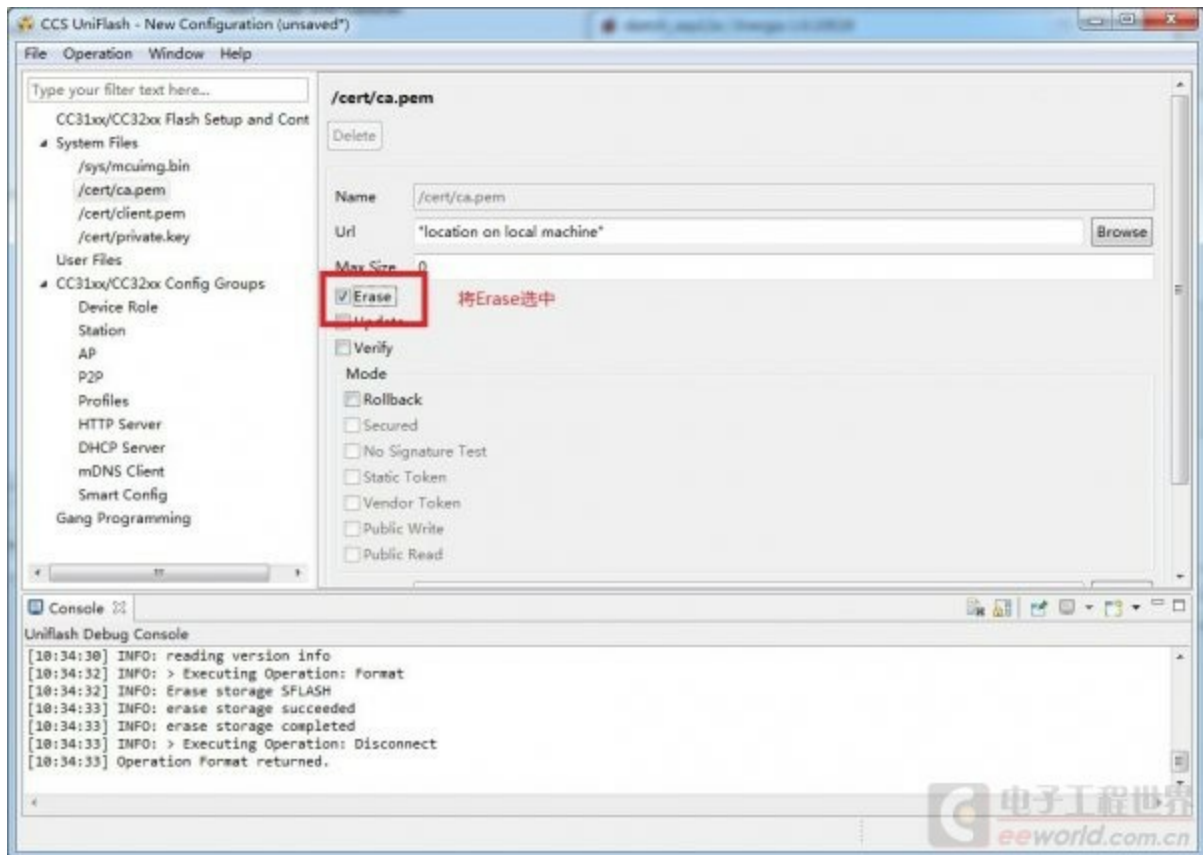
- 在第一个配置页面中（CC31xx/CC32xx Flash Setup and Control），输入实际的串口号，如下图所示。



- 格式化Flash。在第一个页面按下Format，在弹出选择框选择1M容量，然后确定就开始格式化。

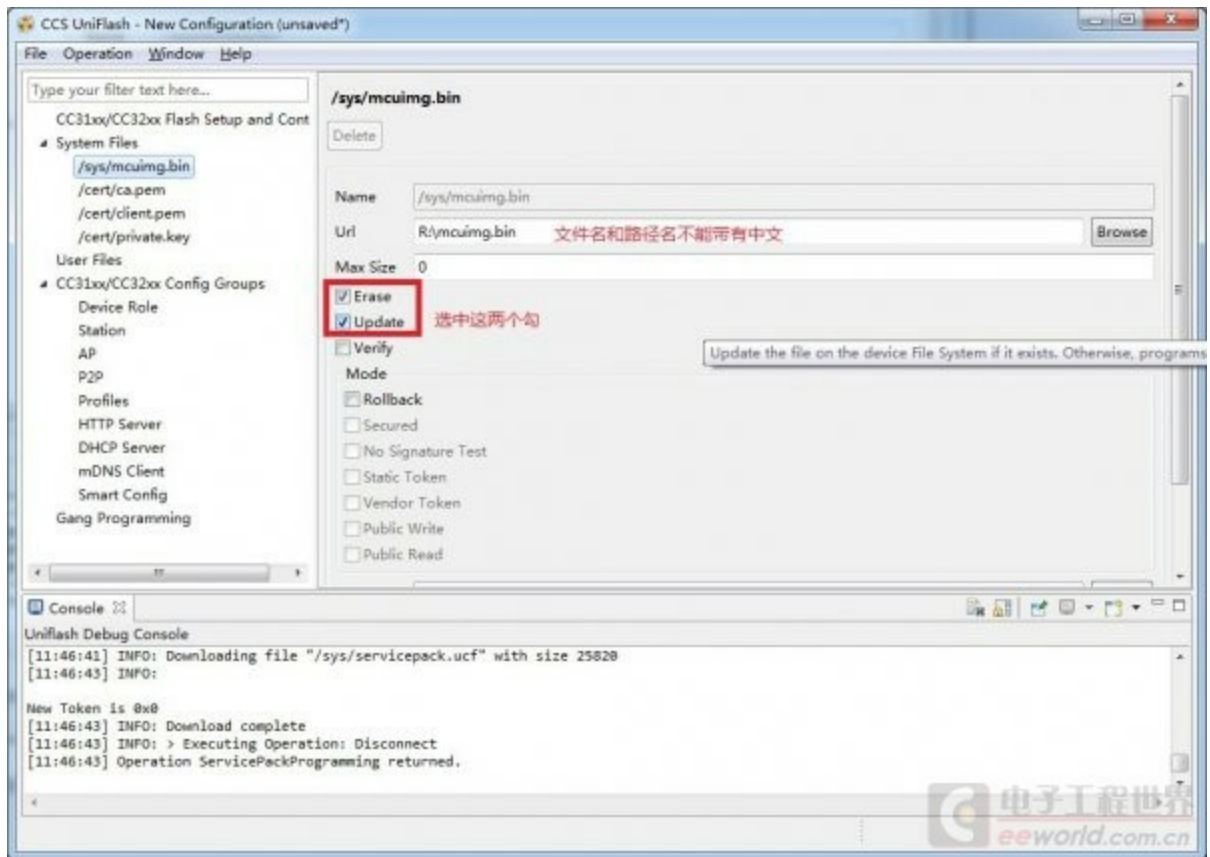


- 在System Files下，选择/cert/ca.pem，只选中Erase框。然后对后面两个文件进行同样的操作。



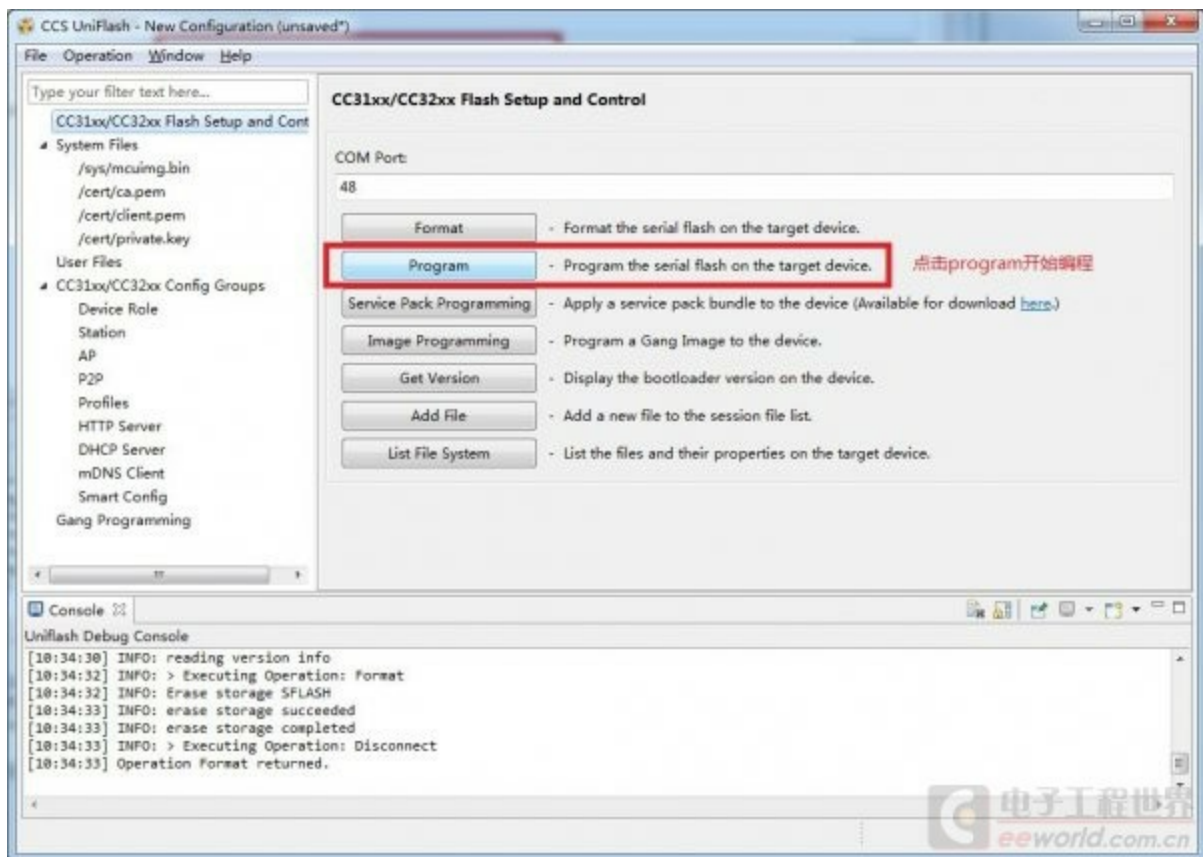
- 选择sys/mcuimg.bin文件，选中Erase和Update两个勾。然后在下面的Url中，用Browse选择我们前面编译出的mcuimg.bin文件。特别注意文件不要带有中文路径和中文文件名，否则后面的下载会出错。





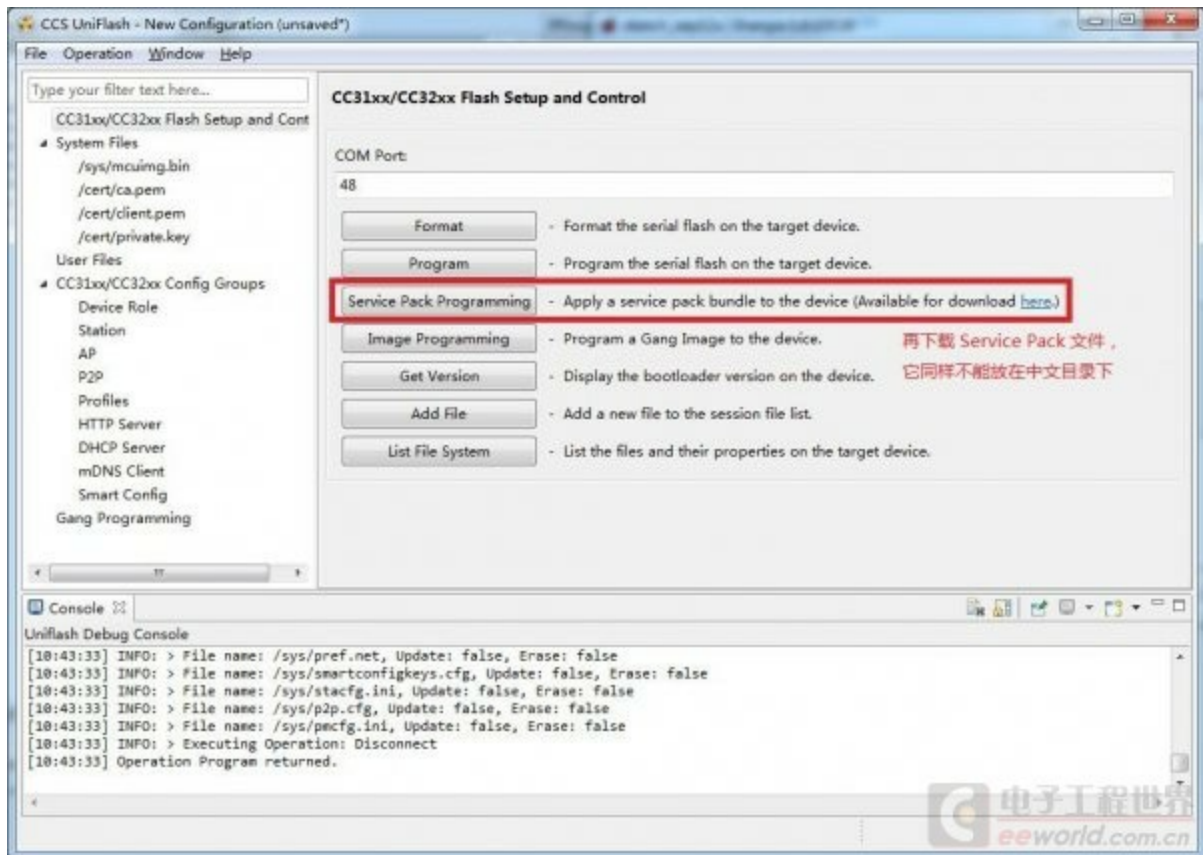
- 回到第一个页面，按下Program，下载固件。





## 下载Service Pack

- 按下Service Pack Programming按钮，下载系统服务包，这个文件在前面安装的[CC3200SDK-SERVICEPACK](#)文件夹中，通常是一个servicepack\_XXXX.bin这样的文件，后面的XXXX与版本有关。



## 运行MicroPython

- 完成前面的操作后，如果没有错误（如果出现错误可以重新再次操作，出错通常与USB线有关），就可以取下SOP2的短路块，拔下USB线，然后重新上电（只按复位键是不行的）。等几秒后查看Wifi，如果出现一个wipy-wlan-XXXX的热点，就说明MicroPython已经成功运行了。

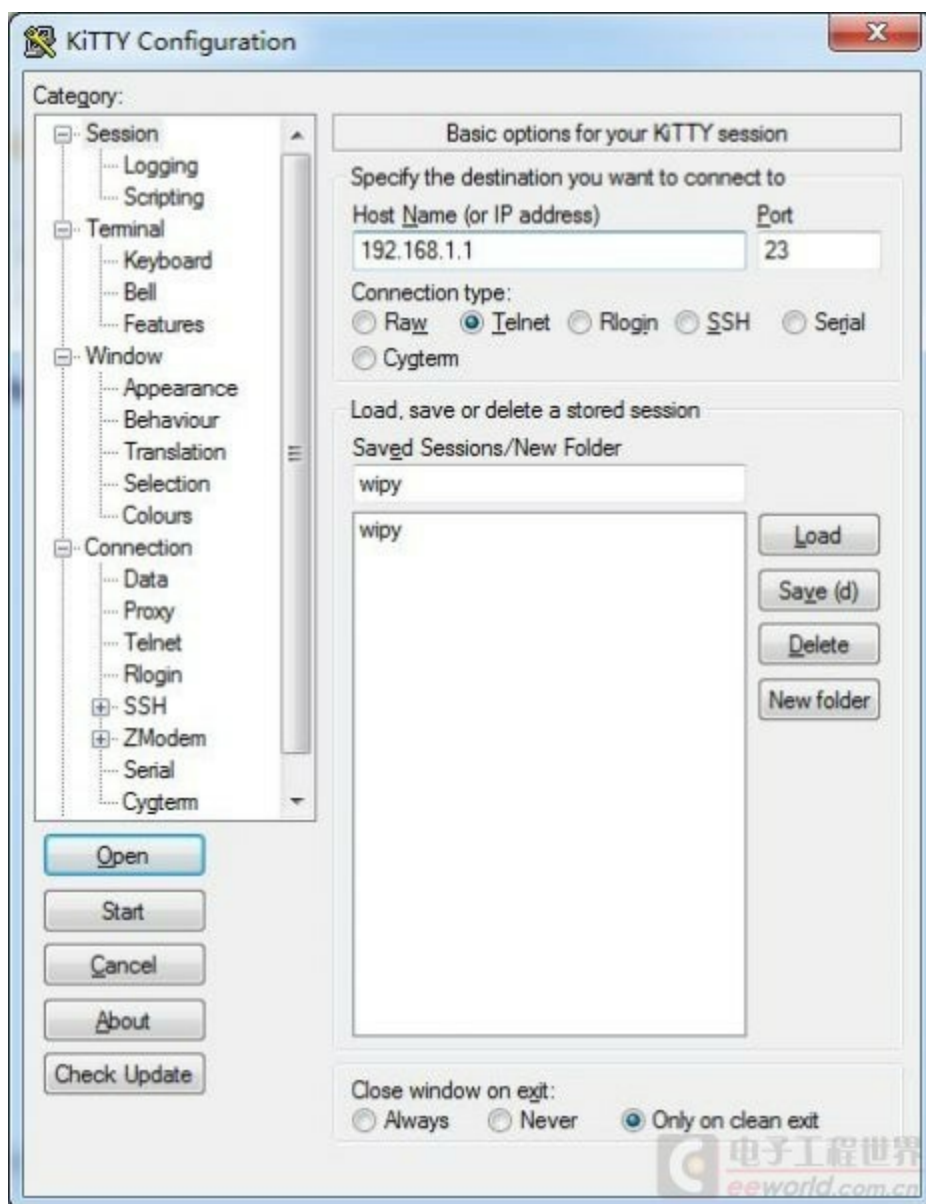


- 出现wipy-wlan热点后，就可以进行连接了。连接的密码是wipy公司的网址：[www.wipy.io](http://www.wipy.io)

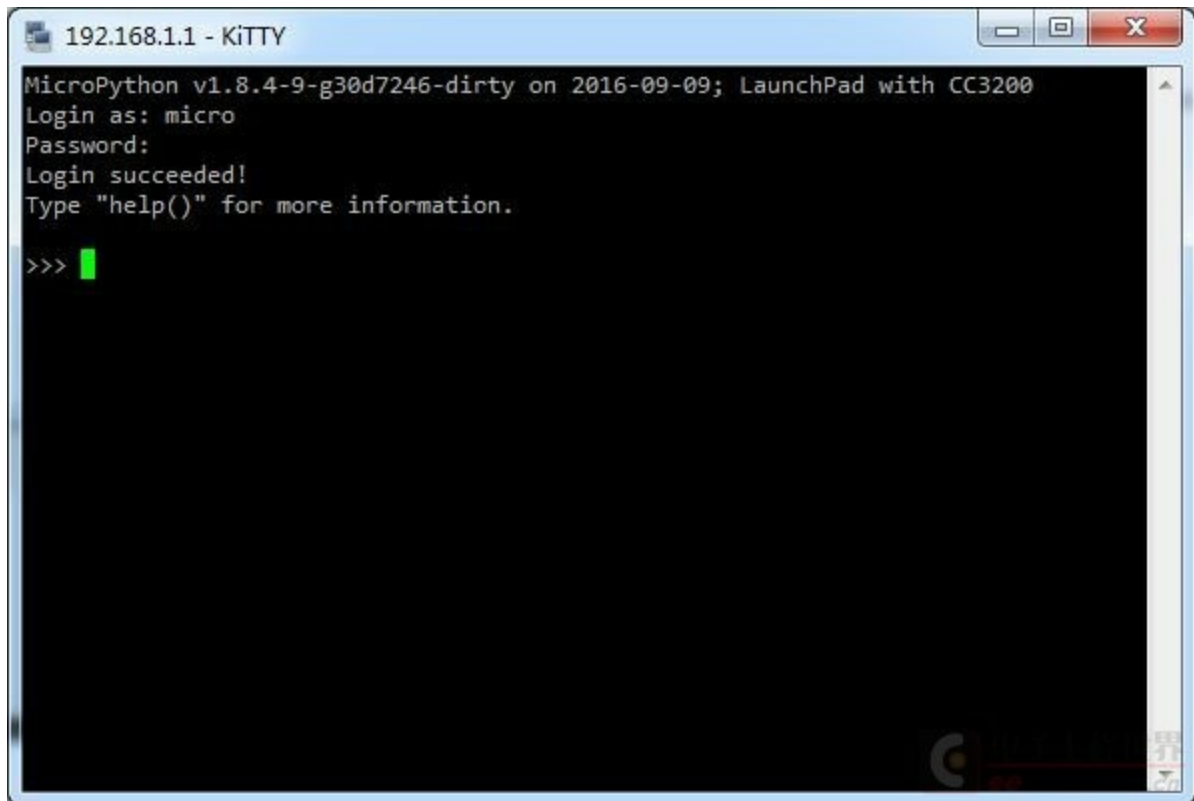


- 然后就可以通过一个支持telnet的终端进行连接，比

如putty，将连接方式设置为Telnet，主机设置为192.168.1.1，端口是23。

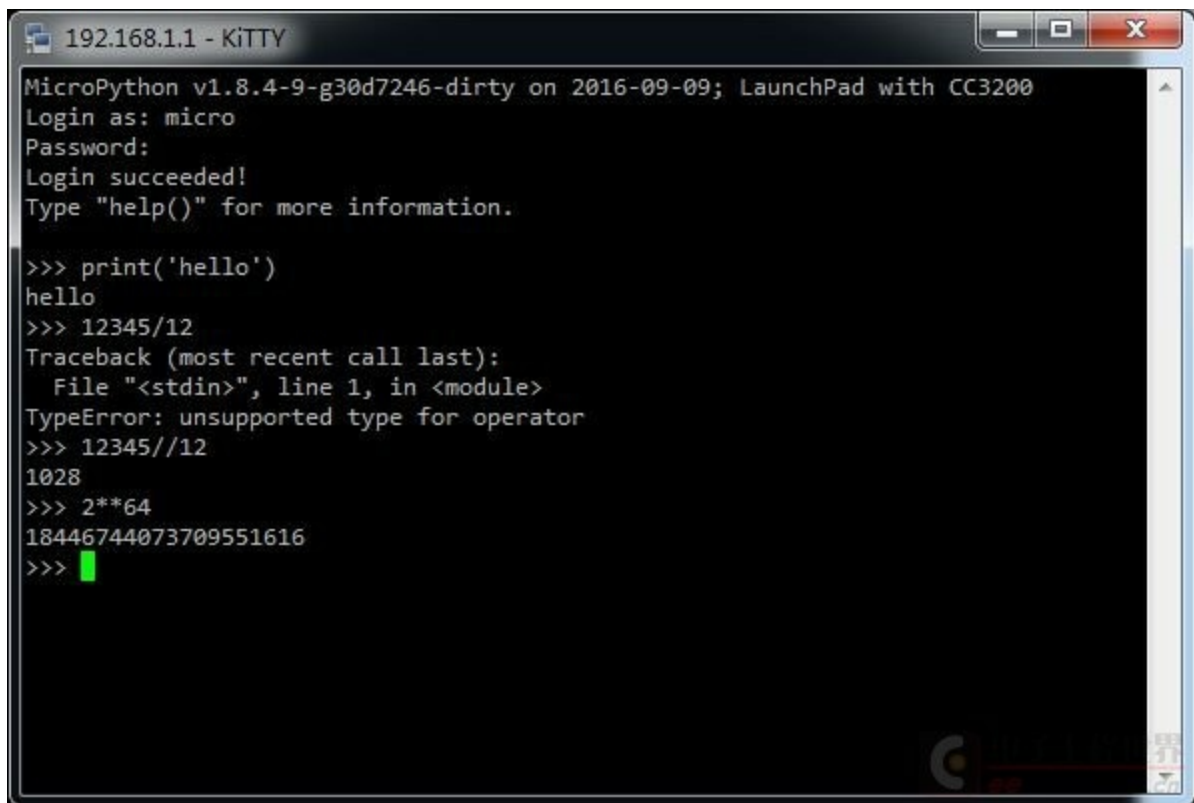


- 在出现登录提示时，用户名是：micro，密码是：python。登录成功就会出现我们熟悉的>>>提示符。



```
192.168.1.1 - KITTY
MicroPython v1.8.4-9-g30d7246-dirty on 2016-09-09; LaunchPad with CC3200
Login as: micro
Password:
Login succeeded!
Type "help()" for more information.
>>> █
```

- 在提示符后就可以输入各种命令，值得注意的是CC3200没有浮点单元，不支持浮点计算。就连普通的除法计算也会出错，只能用整数除法//代替一般除法/。



```
192.168.1.1 - KITTY
MicroPython v1.8.4-9-g30d7246-dirty on 2016-09-09; LaunchPad with CC3200
Login as: micro
Password:
Login succeeded!
Type "help()" for more information.

>>> print('hello')
hello
>>> 12345/12
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unsupported type for operator
>>> 12345//12
1028
>>> 2**64
18446744073709551616
>>> █
```

- 出了putty，还可以使用很多其它软件，如超级终端、MobaXterm、XShell等。



- CC3200目前只支持热点（AP）方式，不支持终端方式。因此连接上wipy-wlan热点后，计算机就不能上网了，会带来很多不便。一个解决方法是使用有线网络，或者使用两个无线网卡，一个连接外网，一个连接wipy-wlan。注意需要用第一个网卡连接wipy-wlan。



固件文件



为了方便大家，下面提供编译好的固件和service pack文件，可以节约大家编译和下载的时间，但是CCS Uniflash还是需要大家自己下载和安装，它目前也只有windows版本，没有Linux和MacOS版本，在这些OS下，只能通过虚拟机方式使用。

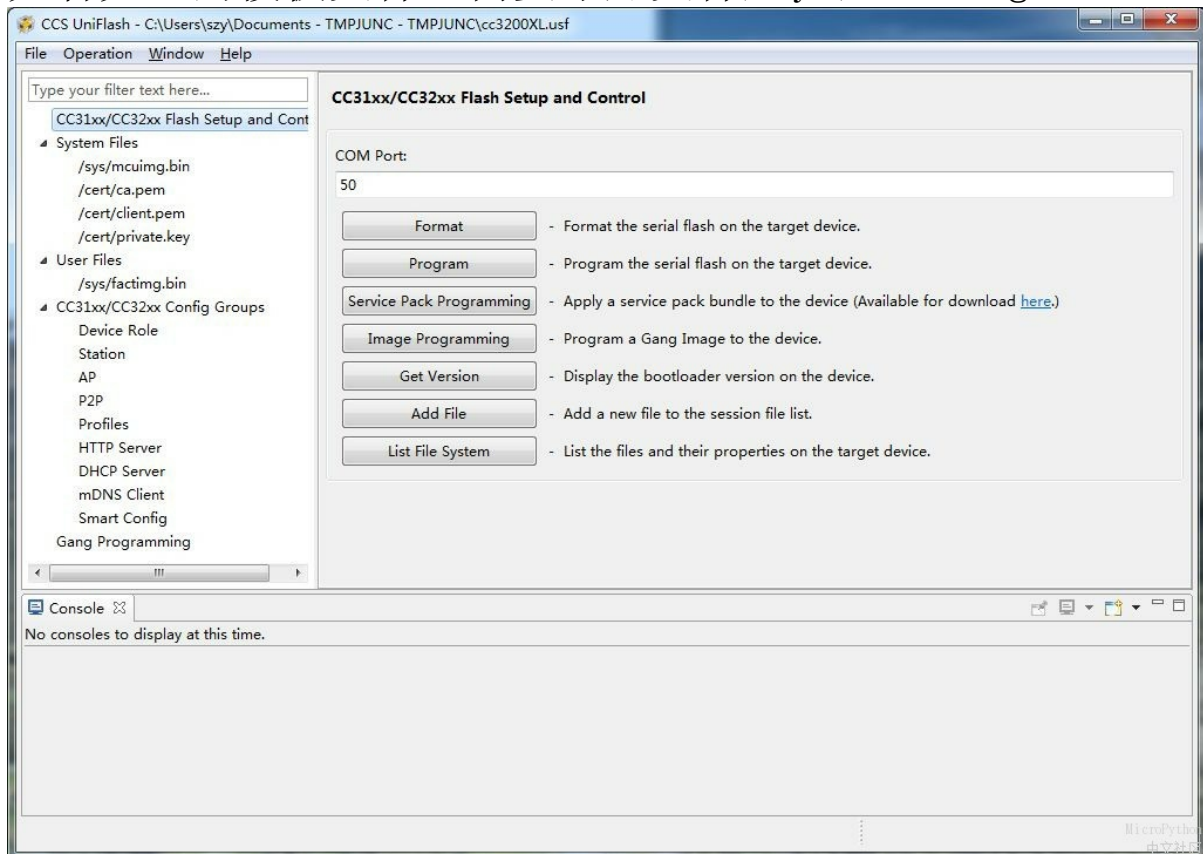
## [下载固件](#)

### 注：

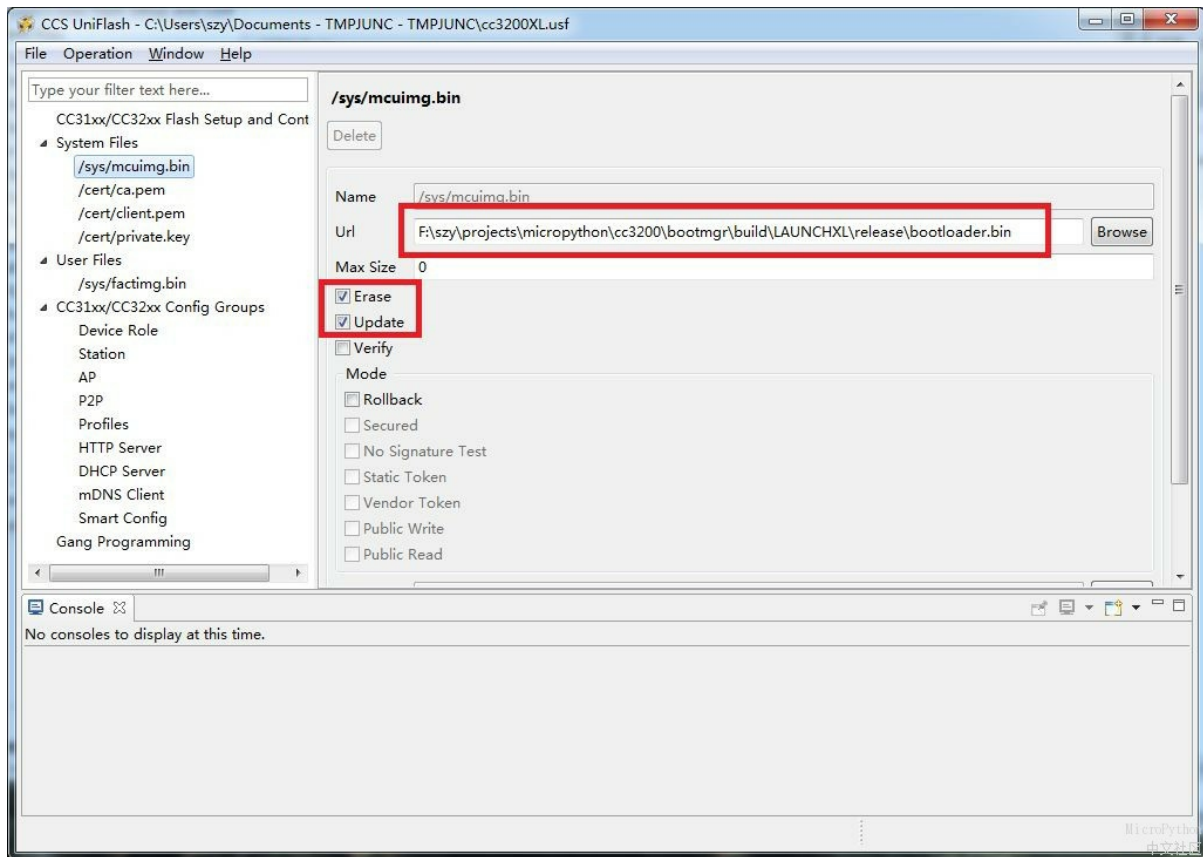
- 下载固件的操作和官方介绍有很大不同，官方操作中，是需要添加两个用户文件，然后将bootloader.bin写入/sys/mcuimg.bin，将mcuimg.bin写入/sys/factimg.bin，但是我尝试过很多次都没有成功，下载后没有任何反应。
- 现有的方式运行没有问题，但是不知道是否会影响无线方式升级，暂时还没有试过。

经过几天的摸索，终于找到正确的写入方法，修正了上次帖子中不能下载bootloader的问题，可以完美的通过wifi方式升级固件了。如果不下载bootloader.bin，虽然可以运行，但是升级功能是无法实现的。

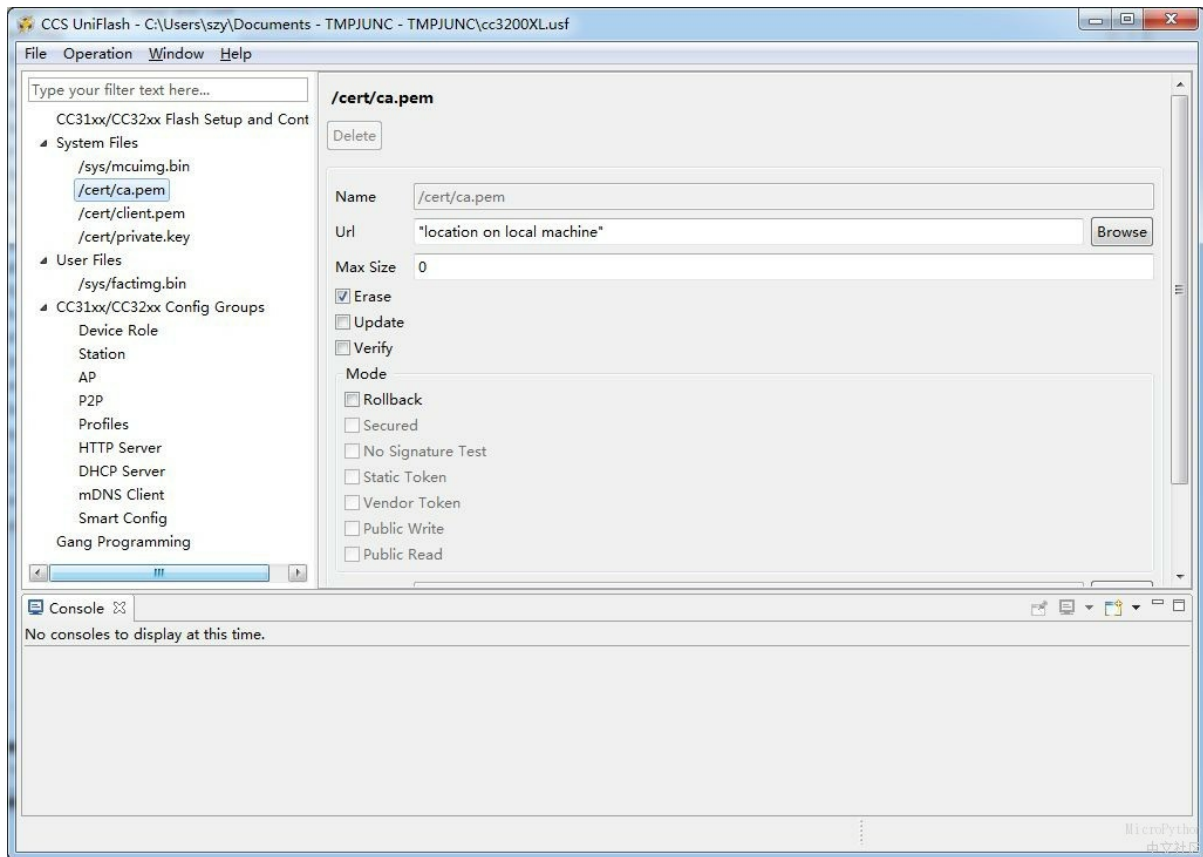
先看完整的模板文件，需要添加文件/sys/factimg.bin



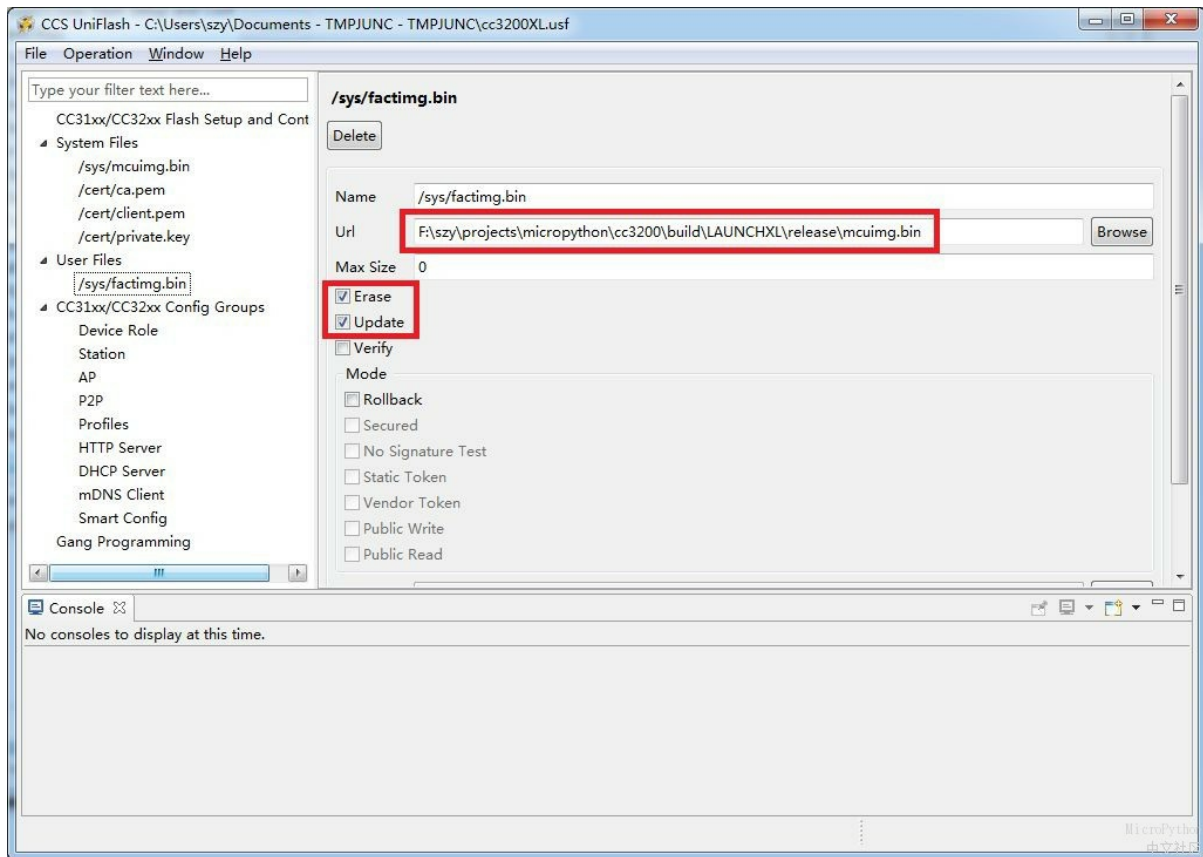
原来/sys/mcuimg.bin文件改为写入bootloader.bin，同时设置Erase和update。



/cert/ca.pem等文件和以前一样，需要设置Erase。



新添加的文件文件/sys/factimg.bin，需要写入mcuimg.bin，同时也需要设置Erase和Update，update如果不设置，程序就不会运行，这一点readme.me没有写，一直就卡在这个地方。



剩下的就和以前那个帖子一样，先Format清除Flash，再programm写入固件，最后是Service Pack Programming写入系统服务包，在复位就可以运行了。另外不要忘记设置SOP2短路块，具体步骤大家可以参考一下：

[在CC3200-LAUNCHXL上运行MicroPython](#)

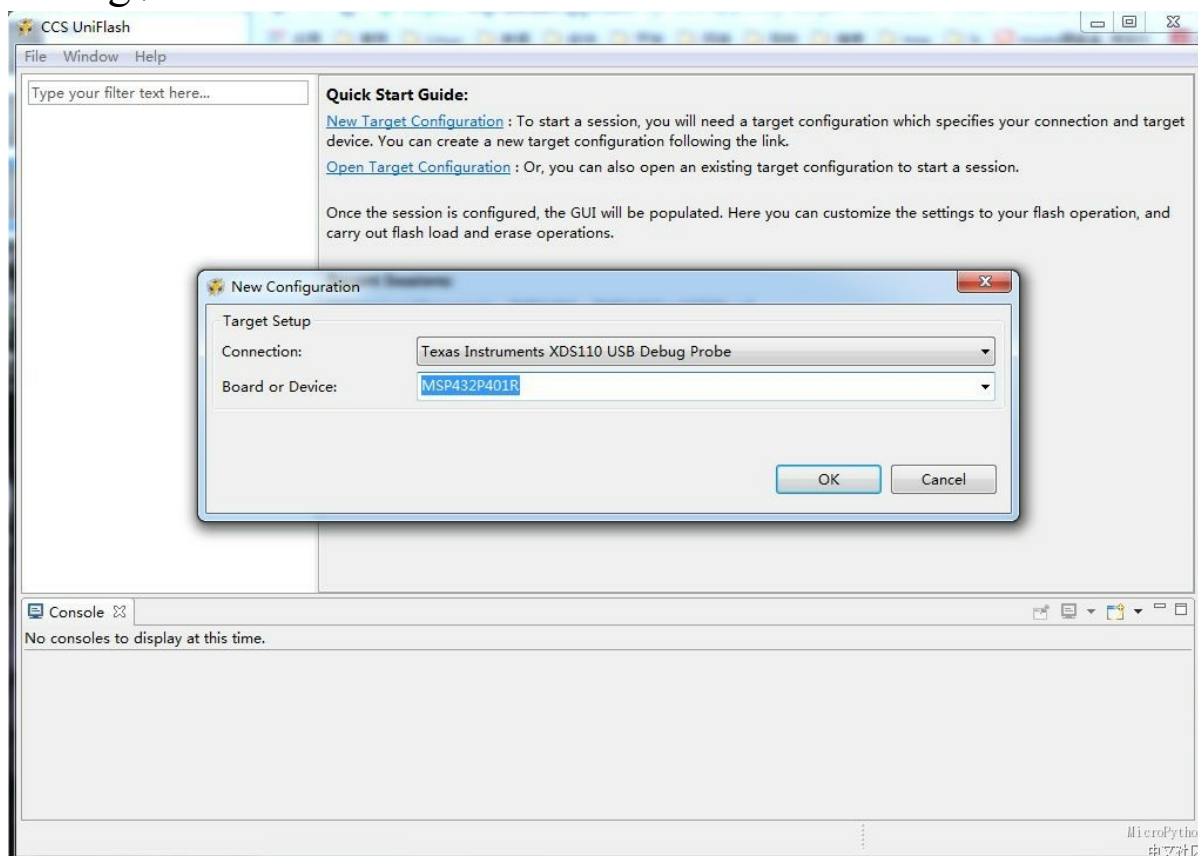


<http://www.ti.com/tool/msp-exp432p401r>

请到[社区](#)下载

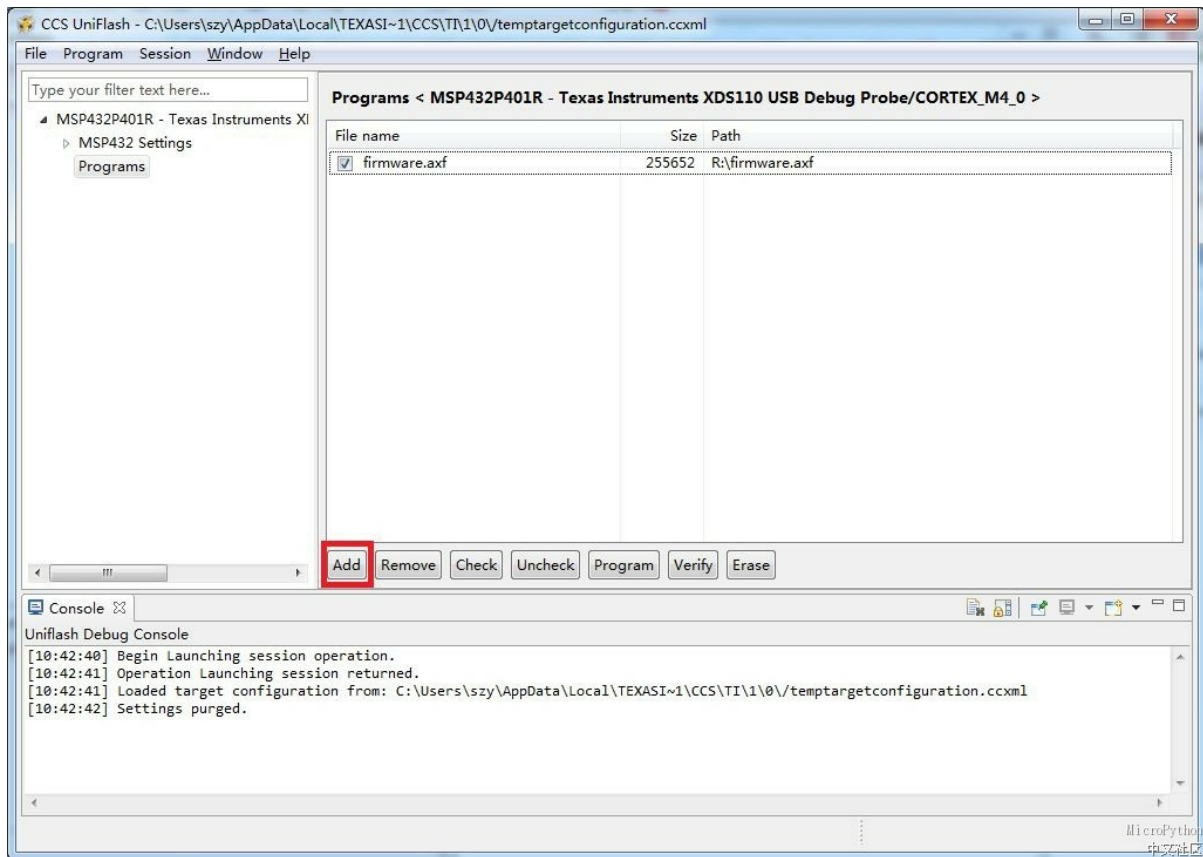
首先需要下载国外网友BonifaceBassey移植的[MicroPython](#)，并编译源码，得到固件firmware.axf。编译的方法和编译STM32的方法差不多，需要安装gcc-arm-none-eabi。如果怕麻烦，就直接下载论坛提供的[二进制文件吧](#)。

在运行TI的CCS Uniflash，新建一个配置，选择XDS110 USB Debug和MSP432R401。

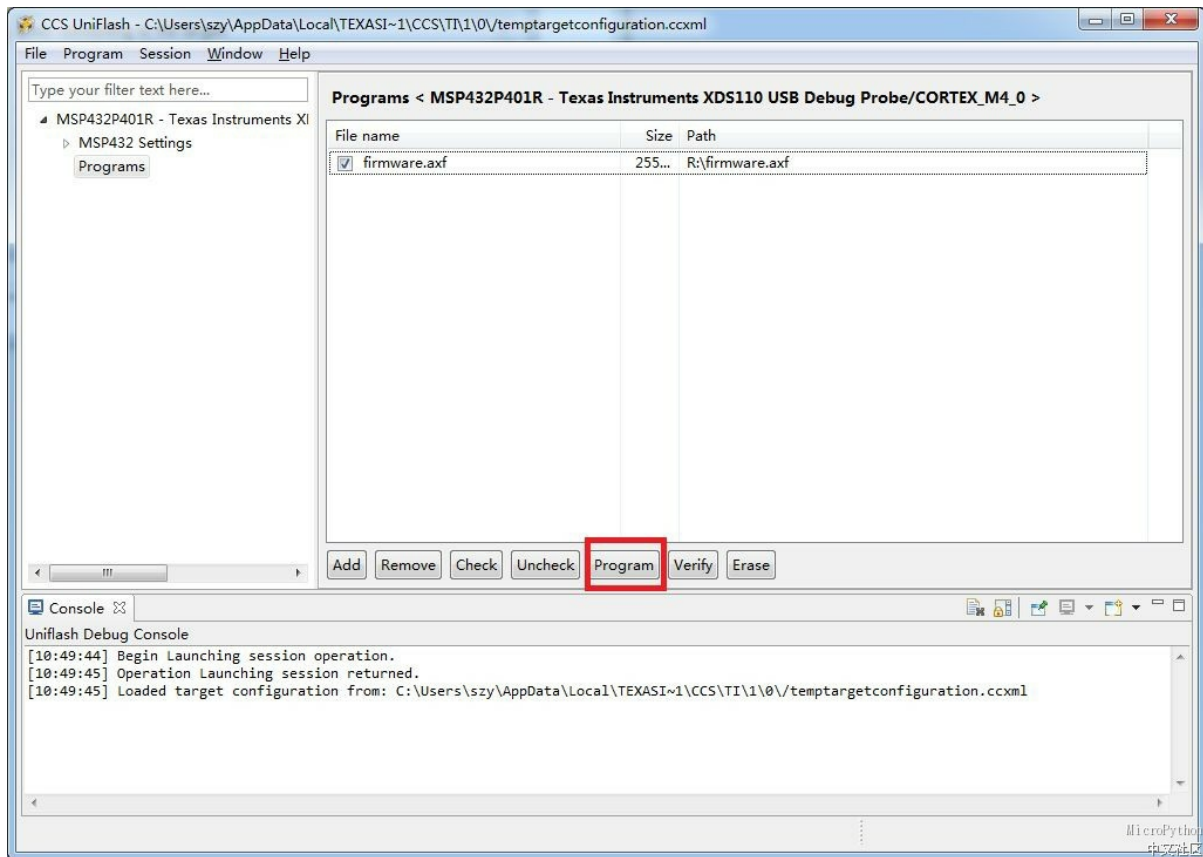


在Programs下，添加固件文件。

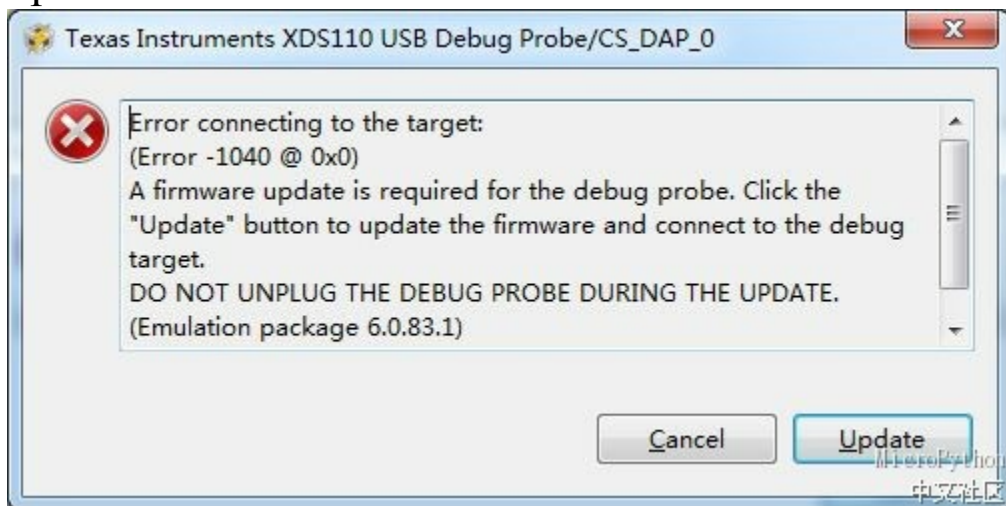




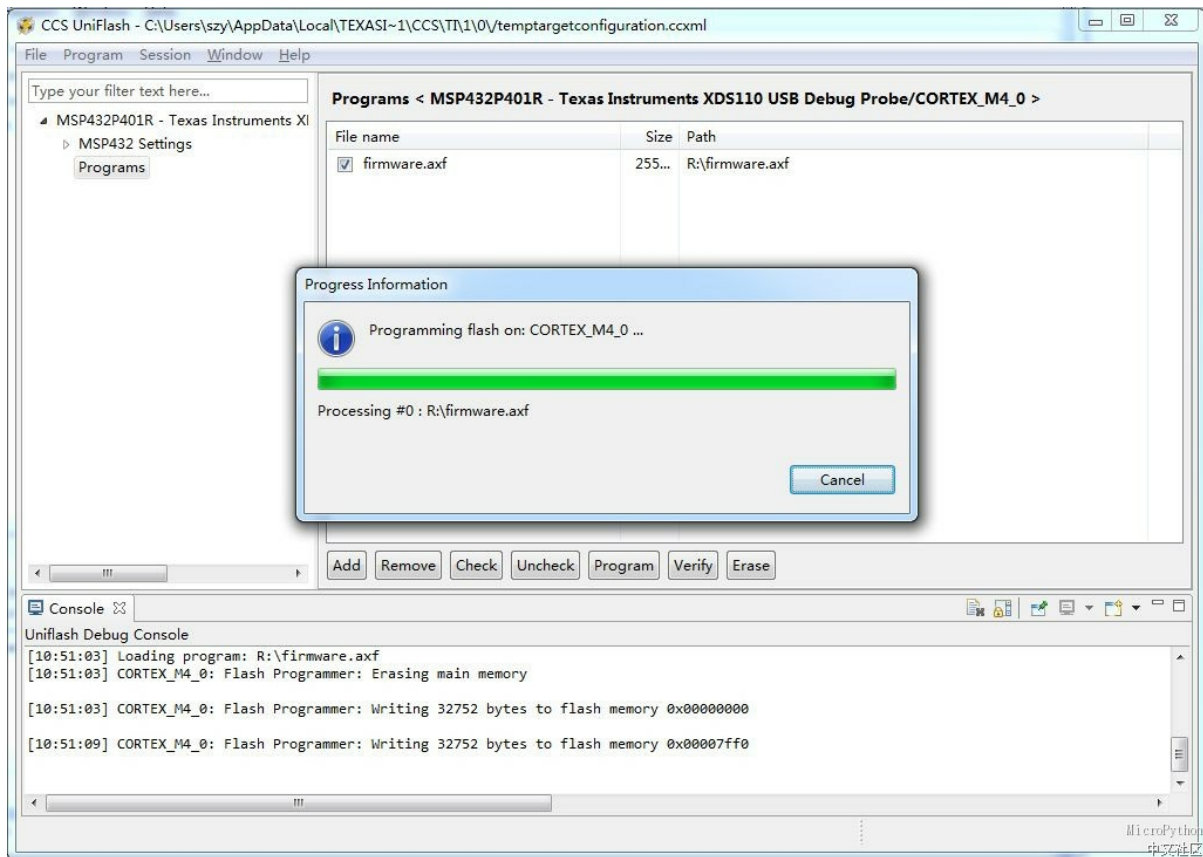
再点击Program下载固件。



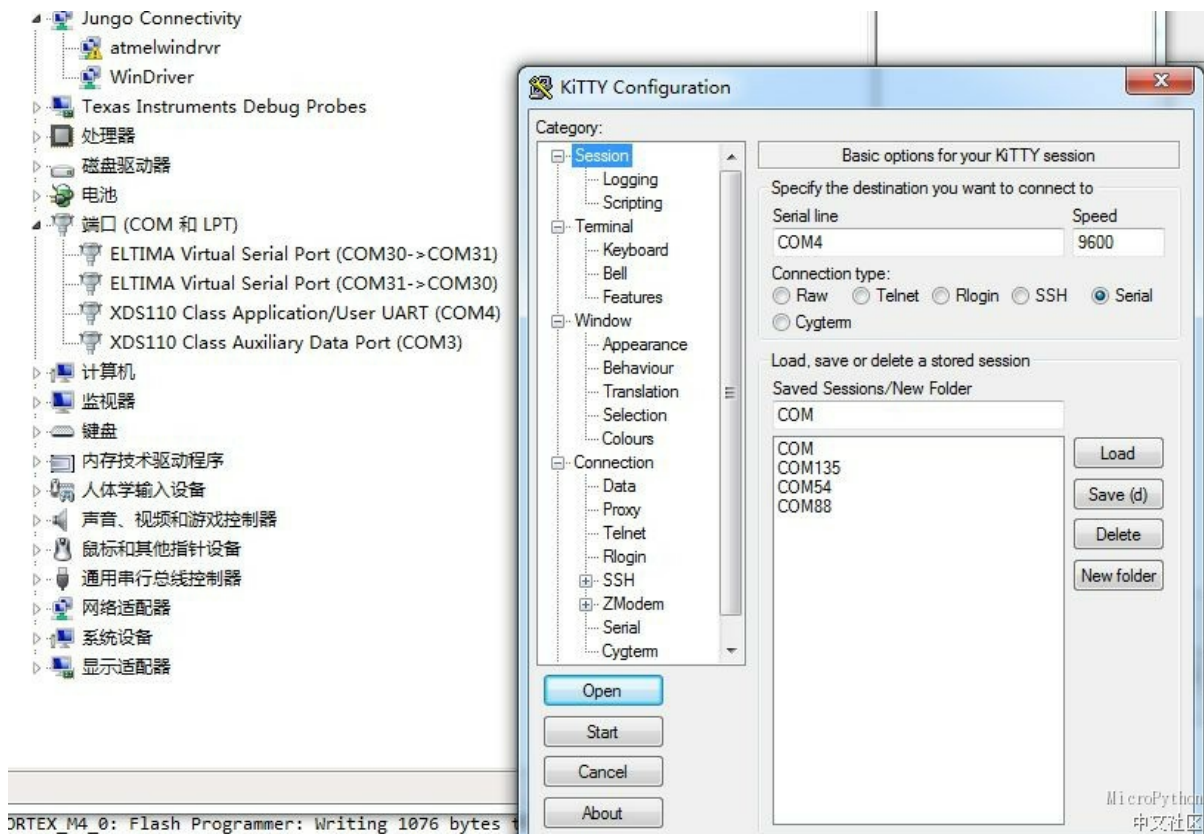
如果是首次连接开发板，多半会提示升级仿真器固件，选择 update。



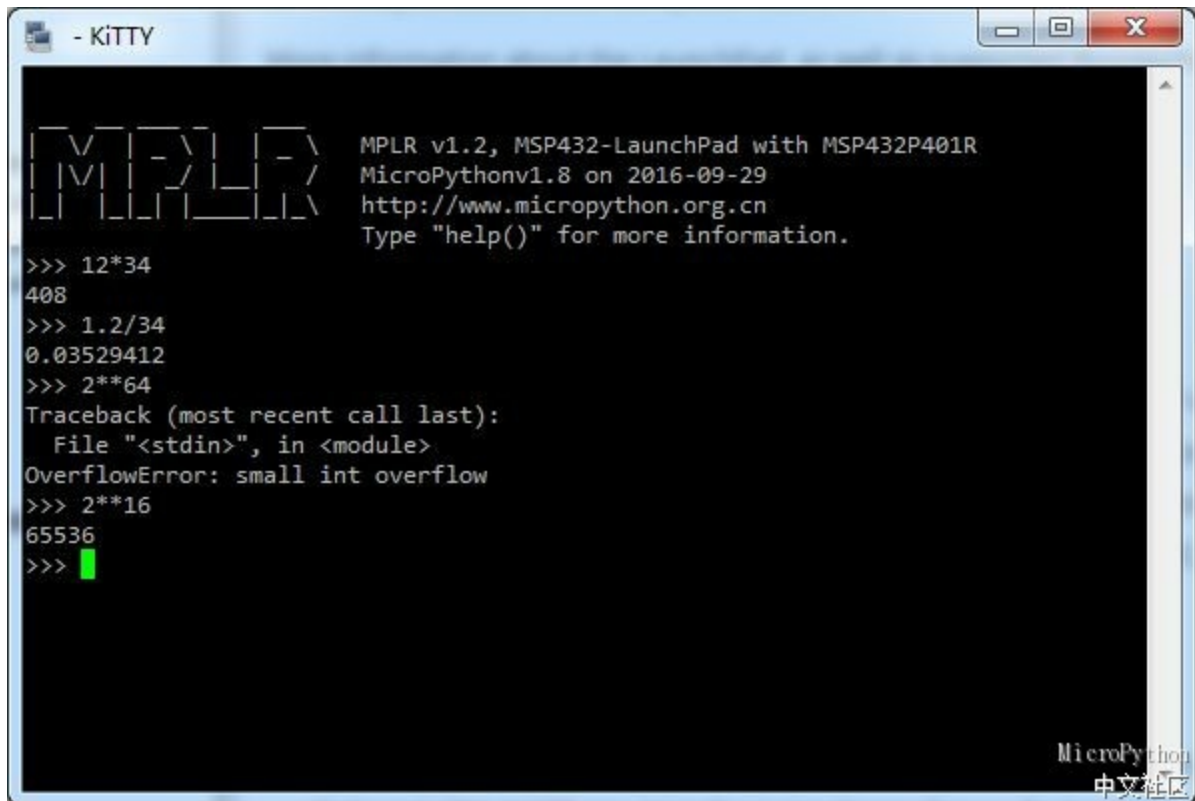
升级后会继续下载固件。如果提示错误，检查一下JTAG Switch是否不小心拨到Ext Debug了。



下载后，就可以用终端软件连接了。XDS仿真器会出现两个串口，选择User UART那个。和其它版本不同，这里波特率需要设置为9600。



进入后，可以按下Ctrl-D软复位，或者Ctrl-B，看看提示符是否出现。比CC3200好的是MSP432支持浮点运算，但是不支持大数计算。大家可能也注意到这个移植版本比较旧，还是1.8的版本，因此很多标准库还没有移植过来，如os、sys、machine等，所以大部分功能也就无法使用。



```
- KITTY

MPLR v1.2, MSP432-LaunchPad with MSP432P401R
MicroPythonv1.8 on 2016-09-29
http://www.micropython.org.cn
Type "help()" for more information.

>>> 12*34
408
>>> 1.2/34
0.03529412
>>> 2**64
Traceback (most recent call last):
 File "<stdin>", in <module>
OverflowError: small int overflow
>>> 2**16
65536
>>> █
```

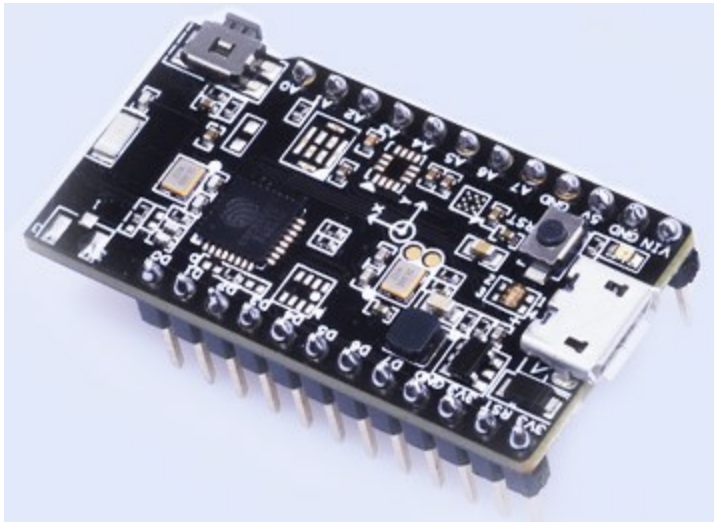
MicroPython  
中文社区

今天检查了一下MicroPython的更新，发现已经加入GPIO的支持了

```
import time
from machine import Pin

LED = Pin("GPIO_1", 21, Pin.OUT)
while True:
 LED.value(1)
 time.sleep(0.5)
 LED.value(0)
 time.sleep(0.5)
```

IntoRobot Neutron是深圳摩仑公司的物联网开发板，上面使用了STM32F411CUE6和ESP8266。



IntoRobot Neutron开发板上带有STM32F411控制器，它支持USB接口的DFU模式，这样给程序升级带来了方便，可以不使用任何编程器将程序下载到单片机中。

单片机进入DFU模式，在复位的时候需要有几个条件：

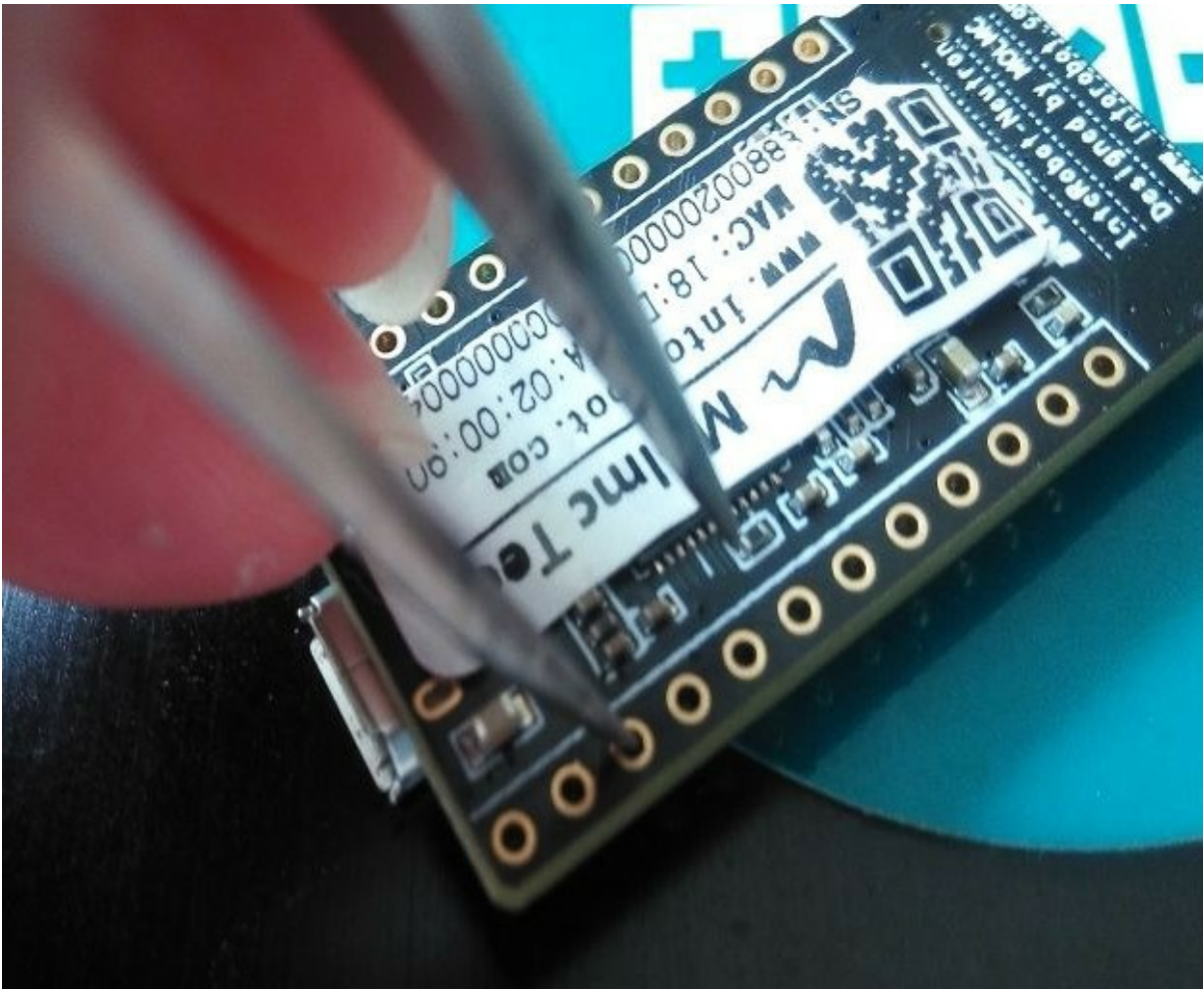
- USB接入到计算机
- BOOT0接VCC
- BOOT1悬空或接地
- 外部时钟频率和精度在一定范围

IntoRobot Neutron开发板的外部时钟是26M，满足条件4，USB已经引出，条件1也满足。所以重点就看条件2和条件3。

在原理图上，我们可以看到BOOT0通过R104接地，没有直接连出来，而BOOT1连接到SW5，通过R97接VCC，因此条件2和条件3都不能直接满足。条件3可以通过按下SW5实现，条件2就需要自己想办法将BOOT0连接到VCC上。

通过安装图可以找到R104的位置，它就在STM32F411的旁边，我们用一个镊子就可以方便的将它和VCC连接。一个问题是R104是0402封装的，比较小，所以要小心不能短路了。先将USB连接，然后按下SW5和SW6（复位键），在用镊子小心连接R104和VCC，然后释放SW5，在放开SW6和镊子。如果顺序没有错误，我们就可以在设备管理器中找到DFU设备了，在DfuSe Demo中也可以看到设备已经连接上了。





IntoRobot Neutron的主MCU是STM32F411，它在MicroPython的支持范围，因此是可以移植MicroPython的。

这几天尝试了移植MicroPython，具体步骤如下：

- 下载[MicroPython源码](#)
- 安装[GNU ARM Embedded Toolchain](#)编译器
- 在stmhal/board目录下，新建一个Neutron目录。
- 将nucleo-F401下的文件复制过来
- 修改配置文件，主要有：
  - 时钟
  - LED
  - I2C
  - SPI
  - 串口
  - 等，可以根据需要自己去配置
- 编译源码
- 将开发板复位，并进入DFU模式（参考 [DFU模式](#)）
- 下载固件
- 重新上电，如果出现PYBFLASH磁盘，就说明移植成功了。
- 运行一个支持串口的终端软件，就可以开始玩MicroPython了。

为了方便大家，节约重复编译的时间，社区提供了编译好的固件，可以直接下载使用。

[固件下载](#)

可能不少网友知道，普通的LED其实也是可以测量光强的，这是因为LED也是一个二极管，它具有光电效应，光越强，PN节的放电速度越快。

使用这个功能，驱动LED的GPIO需要具有ADC功能。在原理图上我们可以看到，只有LED\_R对应的B1具有ADC功能，下面我们就使用它做光强测试的实验。

首先下载了MicroPython固件，然后在串口终端中输入下面的代码：

```
from pyb import Pin, ADC

def test(count):
 pn = Pin('B1', Pin.OUT)
 for i in range(count):
 tmp = 0
 pn(1)

 pn = Pin('B1', Pin.IN, pull=Pin.PULL_NONE)
 pyb.delay(2)

 adc = ADC(Pin('B1'))
 for n1 in range(8):
 tmp += adc.read()
 print(tmp - 19500)

 pn = Pin('B1', Pin.OUT)
 pn(0)
 pyb.delay(500)
```

然后输入 `test(100)`，运行程序进行测试。并改变环境的光强，比如用手遮挡LED，将LED放到光线较强的位置等。

```
1152
1078
1205
1406
950
892
900
902
899
```

892  
1077  
1413  
1394  
1405  
1407  
1422  
1276  
1202

从上面可以发现，效果还是很明显的，光线越强，放电越快，数值越小。

```

from pyb import Pin, Timer

tm=Timer(1, freq=1000)
ch1=tm.channel(1, Timer.PWM, pin=Pin('B13'))
ch2=tm.channel(2, Timer.PWM, pin=Pin('B14'))
ch3=tm.channel(3, Timer.PWM, pin=Pin('B1'))

r=g=0.0
b=100.0
dr=1.1
dg=2.6
db=3.2
while True:
 ch1.pulse_width_percent(g)
 ch2.pulse_width_percent(b)
 ch3.pulse_width_percent(r)

 r += dr
 g += dg
 b += db

 if(r>260)or(r<-5):
 dr = -dr
 if(g>260)or(g<-5):
 dg = -dg
 if(b>260)or(b<-5):
 db = -db

 print("%4d"%r, "%4d"%g, "%4d"%b)
 pyb.delay(20)

```



<http://www.beidouapp.com/>

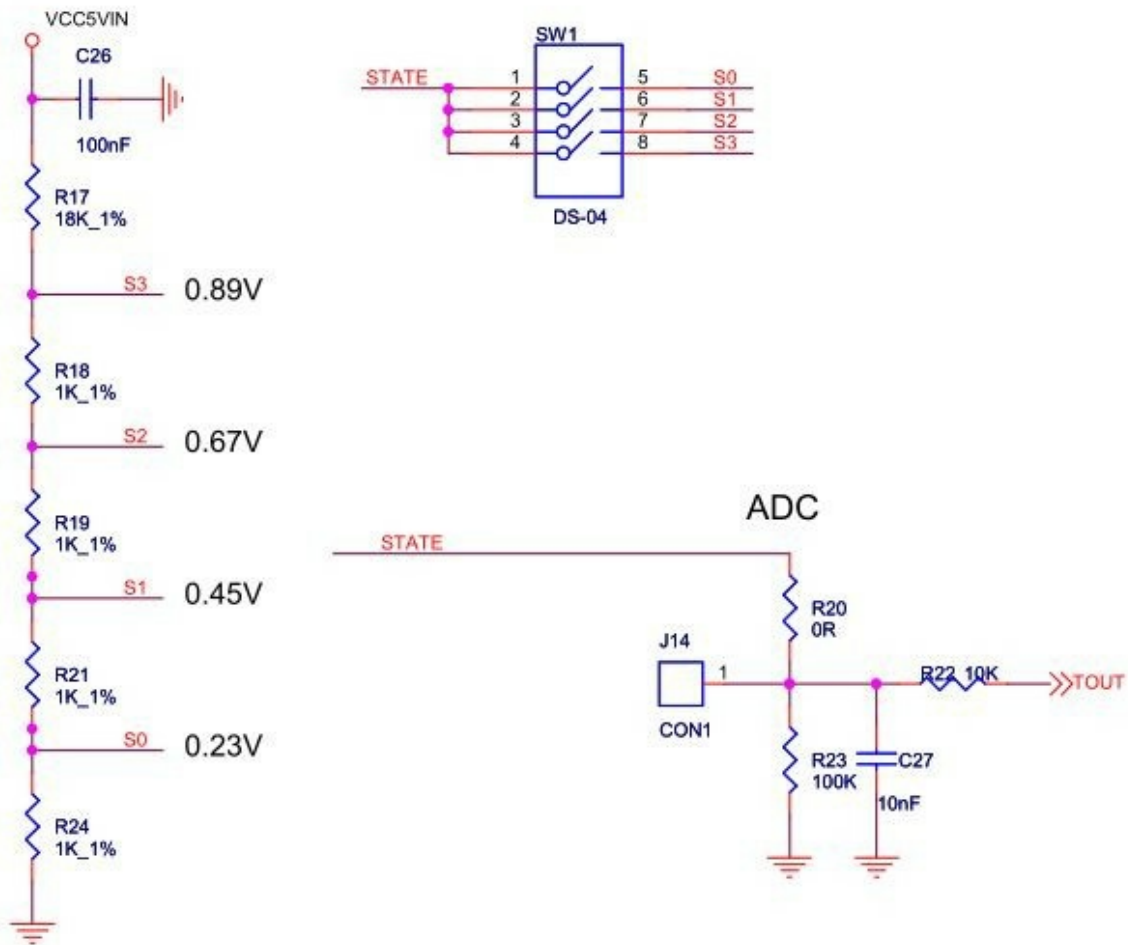
```
import machine, time
from machine import ADC

while True:
 print(ADC(0).read())
 time.sleep_ms(500)
```

运行这段程序后，就会每500ms读取一次ADC。改变拨码开关，就可以看到ADC读数的变化。

停止程序，可以按下Ctrl-C

小e的拨码开关是通过ADC读取分压电阻来识别的。



因此我们只要通过电压的范围就可以知道按下哪个开关。

```
import machine, time
from machine import ADC

def DSW():
 d = ADC(0).read()
 if(d < 200):
 return 0
 elif(d < 450):
 return 4
 elif(d < 650):
 return 3
 elif(d < 900):
 return 2
 else:
 return 1
```



改变拨码开关，然后输入DSW()就可以知道开关状态了。

DSW()

注：因为分压精度问题，所以只能识别一个开关，不能识别同时按下多个开关。

小e开发板上带有BMP180气压传感器，使用micropython，可以很方便的读取传感器参数。

首先BMP180驱动，然后就可以直接读取。

```
from bmp180 import BMP180

b=BMP180(2,14)
b.getTemp()
b.getPress()
b.getAltitude()
```

BMP180的micropython驱动，完整驱动文件请到[社区下载](#)。

```
import machine, time
from machine import I2C, Pin

BMP180_I2C_ADDR = const(0x77)

class BMP180():
 def __init__(self, pSDA, pSCL):
 self.i2c = I2C(scl=Pin(pSCL), sda=Pin(pSDA), freq =
100000)
 self.AC1 = self.short(self.get2Reg(0xAA))
 self.AC2 = self.short(self.get2Reg(0xAC))
 self.AC3 = self.short(self.get2Reg(0xAE))
 self.AC4 = self.get2Reg(0xB0)
 self.AC5 = self.get2Reg(0xB2)
 self.AC6 = self.get2Reg(0xB4)
 self.B1 = self.short(self.get2Reg(0xB6))
 self.B2 = self.short(self.get2Reg(0xB8))
 self.MB = self.short(self.get2Reg(0xBA))
 self.MC = self.short(self.get2Reg(0xBC))
 self.MD = self.short(self.get2Reg(0xBE))
 self.UT = 0
 self.UP = 0
 self.B3 = 0
 self.B4 = 0
 self.B5 = 0
 self.B6 = 0
 self.B7 = 0
 self.X1 = 0
 self.X2 = 0
 self.X3 = 0

 def short(self, dat):
 if dat > 32767:
```

```

 return dat - 65536
 else:
 return dat

def setReg(self, dat, reg):
 buf = bytearray(2)
 buf[0] = reg
 buf[1] = dat
 self.i2c.writeto(BMP180_I2C_ADDR, buf)

def getReg(self, reg):
 buf = bytearray(1)
 buf[0] = reg
 self.i2c.writeto(BMP180_I2C_ADDR, buf)
 t = self.i2c.readfrom(BMP180_I2C_ADDR, 1)
 return t[0]

def get2Reg(self, reg):
 a = self.getReg(reg)
 b = self.getReg(reg + 1)
 return a*256 + b

def measure(self):
 self.setReg(0x2E, 0xF4)
 time.sleep_ms(5)
 self.UT = self.get2Reg(0xF6)
 self.setReg(0x34, 0xF4)
 time.sleep_ms(5)
 self.UP = self.get2Reg(0xF6)

def getTemp(self):
 self.measure()
 self.X1 = (self.UT - self.AC6) * self.AC5/(1<<15)
 self.X2 = self.MC * (1<<11) / (self.X1 + self.MD)
 self.B5 = self.X1 + self.X2
 return (self.B5 + 8)/160

def getPress(self):
 self.getTemp()
 self.B6 = self.B5 - 4000
 self.X1 = (self.B2 * (self.B6*self.B6/(1<<12))) / (1<<11)
 self.X2 = (self.AC2 * self.B6)/(1<<11)
 self.X3 = self.X1 + self.X2
 self.B3 = ((self.AC1*4+self.X3) + 2)/4

```

...

先输入DHT11的库

```
import esp
from machine import Pin

class DHT11(object):
 def __init__(self, pin):
 self.pin = Pin(pin)
 self.buf = bytearray(5)

 def measure(self):
 buf = self.buf
 esp.dht_readinto(self.pin, buf)
 if (buf[0] + buf[1] + buf[2] + buf[3]) & 0xff != buf[4]:
 raise Exception("checksum error")
 return buf

 def humi(self):
 return self.measure()[0]

 def temp(self):
 return self.measure()[2]

 def get(self):
 return [self.measure()[0], self.measure()[2]]
```

然后就可以读取传感器了

```
from DHT11 import DHT11
dht = DHT11(5)
dht.get()
dht.temp()
dht.humi()
```

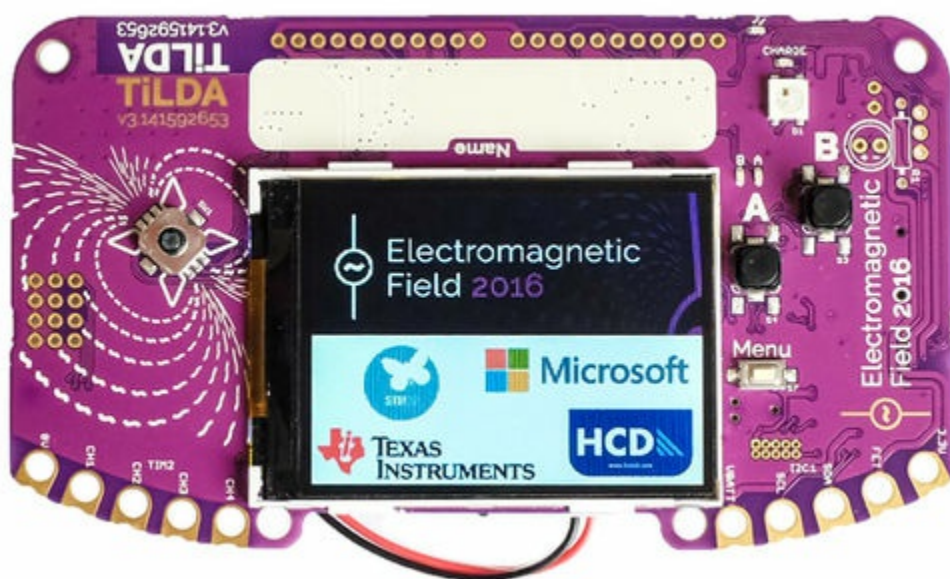
在micropython中，可以很方便的同步网络时间。

```
import ntptime

#查询时间
ntptime.time()

#设置时间
ntptime.settime()
```

国外网友的作品。



[https://badge.emfcamp.org/wiki/TiLDA\\_MK3](https://badge.emfcamp.org/wiki/TiLDA_MK3)

## 国内主要技术论坛的MicroPython版块

- Micropython中文论坛: <http://www.micropython.org.cn>
- EEORLD micropython开源版  
块: <http://bbs.eeworld.com.cn/forum-243-1.html>
- ICKEY MicroPython讨论  
区: <http://bbs.ickey.cn/community/forum.php?mod=forumdisplay&fid=213>
- JUMA论坛Micropython版  
块: <http://www.52cannon.com/bbs/forum-microphython-1.html>
- 电子互动社区MicroPython技术交流: <https://community.ednchina.com/c/group/micropython>

MicroPython需要使用串口终端进行代码调试，常用的终端软件有：

- 超级终端（WinXP）
- putty
- kitty
- xshell
- SecureCRT
- MobaXterm

Linux下可以用：

- putty
- screen



## 其它的相关资源

- BBC  
micro:bit: <https://github.com/bbcmicrobit/micropython>
- webrepl (ESP8266网络终端): <https://github.com/micropython/webrepl>
- mpfshell (ESP8266网络shell): <https://github.com/wendlers/mpfshell>
- ESP8266工具链: <https://github.com/pfalcon/esp-open-sdk>
- Espruino(JavaScript开发单片机): <http://www.espruino.com/>
- Espruino源码: <https://github.com/espruino/Espruino>
- EspruinoWebIDE (图形化编程的浏览器应用): <https://github.com/espruino/EspruinoWebIDE>
- python网站上的pypi相关: <https://pypi.python.org/pypi?%3Aaction=search&term=micropython>

## 官方资源

- 网站: <https://micropython.org/>
- 官方固件下载: <https://micropython.org/download/>
- 源码: <http://micropython.org/resources/micropython-master.zip>
- pyboard设计文件: <http://micropython.org/resources/pyboard-master.zip>
- 官方帮助文件:
  - ESP8266在线帮助: <http://docs.micropython.org/en/latest/esp8266>
  - ESP8266帮助 (PDF): <http://docs.micropython.org/en/latest/micropython-esp8266.pdf>
  - ESP8266快速指南: [http://docs.micropython.org/en/latest/esp8266/esp8266\\_quickstart.pdf](http://docs.micropython.org/en/latest/esp8266/esp8266_quickstart.pdf)
  - pyboard在线帮助: <http://docs.micropython.org/en/latest/pyboard>
  - pyboard帮助 (PDF): <http://docs.micropython.org/en/latest/micropython-pyboard.pdf>
  - pyboard快速指南: [http://docs.micropython.org/en/latest/pyboard/pyb\\_quickstart.pdf](http://docs.micropython.org/en/latest/pyboard/pyb_quickstart.pdf)
  - wipy在线帮助: <http://docs.micropython.org/en/latest/wipy>
  - wipy帮助 (PDF): <http://docs.micropython.org/en/latest/micropython-wipy.pdf>
  - wipy快速指南: [http://docs.micropython.org/en/latest/wipy/wipy\\_quickstart.pdf](http://docs.micropython.org/en/latest/wipy/wipy_quickstart.pdf)
- github源码: <https://github.com/micropython/micropython>
- wiki: <http://wiki.micropython.org/Home>

脚本文件一般用 .py 后缀

中文用户一定得先用这行来声明编码,同时文件本身也得存储成UTF-8编码!

单行注释

导入其它代码模块

注意!Python最好也最个性的语法:  
使用缩进来代替其它语句块声明;  
一般建议每个层级用4个空格来缩进.

变量得先实例化  
才可进一步计算

单行的语句块,其实可以不换行的,  
但是,建议清晰起见,规范点:  
- 另起一行  
- 缩进一级

函数声明,  
注意使用冒号结束声明

多行注释的内容不用遵守当前缩进  
只要开始的''' 缩进正确就成!

每级语法块不用}之类的括号引领!  
直接回车+4空格  
(当然,要在当前缩进基础上)

模块名,其实导入了 os.py

函数名"main"在这儿并不是必须的,调用在这段脚本的最后部分;

声明单行字符串,使用双/单引号都成,  
注意对字符串中的引号进行转义处理!

函数调用,声明在后述代码;

字符串乘,等于:'\*'

调用了os 模块中的函数

连接字符串

内置的列表类型对象,其实可以包含不同类型数据,  
甚至可以包含其它列表对象;

在循环中,i 指代了列表中按顺序的每个"food"

range() 内置函数,返回类似  
[0,1,2,3,4,5,6,7,8,9]  
的数字列表,注意 for in 循环语句使用冒号结束声明!

字符串的格式化输出基本类似C语言的

判定式也基本和C语言的相同

逻辑运算符,不使用 && 和 ||,使用直观的E文单词

这都是合法注释

用冒号来结束判断句,  
在 if elif else 行最后

一般在脚本最后调用主函数 main();而且使用 内置的运行脚本名来判定;  
当且仅当我们直接运行当前脚本时, name\_ 才为 main  
这样当脚本被当作模块进行 import 导入时,并不运行 main()  
所以,一般这里是进行测试代码安置的...

关于  
UliPad 4.0  
作者: Limodou (limodou@gmail.com)  
如果你有任何问题请与我联系。  
[The UliPad project homepage](#)  
[The UliPad maillist](#)  
[The UliPad Snippets Site](#)  
[My Blog](#)  
[Contact me](#)  
确定

一直有网友担心，MicroPython运行是否足够快，性能不够用。所以我做了一个小测试，用python计算阶乘，看看速度到底会有多快，是否可以达到大家的期望值。

我们先编写一小段测试程序，这段代码先计算阶乘，然后计算总的耗时。测试程序如下

```
import time

def fact(n):
 r=1
 t1=time.time()
 while n>1:
 r=r*n
 n=n-1
 t2=time.time()
 print('elapsed: ', time.time_diff(t1,t2), 'us')

 return r
```

我们先在小钢炮开发板上进行测试，它使用了STM32F401RE，最高主频84MHz，SRAM是64KB。

MicroPython v1.7 on 2016-04-17; CANNON with  
STM32F401xE

Type "help()" for more information.

```
>>> import fact_t
>>> n=fact_t.fact(100)
elapsed: 2534 us
```

```
>>> n=fact_t.fact(1000)
elapsed: 174353 us
```

```
>>> n=fact_t.fact(5000)
elapsed: 4223822 us
```

```
>>> n=fact_t.fact(10000)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
File "fact_t.py", line 7, in fact
MemoryError: memory allocation failed, allocating
6498 bytes
```

计算速度还不错，计算5000阶乘也只用了4.2秒。在计算10000的阶乘时因为SRAM不足而出错。因为python支持大数计算，所以计算的位数只受SRAM大小的限制。在进行大数计算的时候，数字范围远远超出了一般数据的范围，因此比较容易测试计算性能。

在换用PYBOARD进行测试，PYBOARD使用了STM32F405RG，它的最高主频是168MHz，SRAM达到192KB。运行同样的测试，结果如下：

```
MicroPython v1.7 on 2016-04-17; PYBv1.0 with
STM32F405RG
Type "help()" for more information.
>>> import fact_t
>>> n=fact_t.fact(100)
elapsed: 1417 us
>>> n=fact_t.fact(1000)
elapsed: 108004 us
>>> n=fact_t.fact(5000)
elapsed: 2333655 us
>>> n=fact_t.fact(10000)
elapsed: 9839525 us
>>> n=fact_t.fact(20000)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "fact_t.py", line 7, in fact
MemoryError: memory allocation failed, allocating
16210 bytes
```

得益于更高的主频和大SRAM，计算用时缩短了接近一半，计

算直到20000时才内存不足。最后在试试性能更好的STM32F746DISC，它使用了STM32F746NGH6，主频达到216MHz，SRAM有320KB。

```
MicroPython v1.8 on 2016-05-15; F7DISC with
STM32F746
```

```
Type "help()" for more information.
```

```
>>> import fact_t
>>> n=fact_t.fact(100)
elapsed: 979 us
>>> n=fact_t.fact(1000)
elapsed: 45384 us
>>> n=fact_t.fact(5000)
elapsed: 1174590 us
>>> n=fact_t.fact(10000)
elapsed: 4386010 us
>>> n=fact_t.fact(20000)
elapsed: 18040829 us
```

```
>>> n=fact_t.fact(25000)
elapsed: 33982479 us
>>> n=fact_t.fact(30000)
```

```
Traceback (most recent call last):
```

```
 File "<stdin>", line 1, in <module>
```

```
 File "fact_t.py", line 7, in fact
```

```
MemoryError: memory allocation failed, allocating
49620 bytes
```

为了进一步了解MicroPython的运行性能，又做了加法和乘法的计算测试，并和用C语言计算进行对比。分别进行1000,000次加法和乘法，看计算各需要多长时间。

使用MicroPython的测试程序如下：

```
import time

def add_test(n):
 t1=time.ticks_us()
 sum=0
 for i in range(n):
 sum = i + i
 t2=time.ticks_us()
 print(n, '次加法计算测试用时：', time.ticks_diff(t1, t2), 'us')

def mul_test(n):
 t1=time.ticks_us()
 sum=0
 for i in range(n):
 sum = i * i
 t2=time.ticks_us()
 print(n, '次乘法计算测试用时：', time.ticks_diff(t1, t2), 'us')
```

使用C语言的测试程序如下。为了测试计算性能，将程序的优化等级设置为0（不优化）。

```
#include "mbed.h"

#define NUM 1000000

DigitalOut myled(LED1);

Timer t;

volatile uint32_t sum, i;

int main() {

 printf("Mbed计算测试\r\n");

 t.reset();
 t.start();

 sum = 0;
```

```

for(i = 0; i < NUM; i++)
 sum = i + i;

t.stop();

printf("%d 次加法计算用时 %f 秒\r\n", NUM, t.read());

t.reset();
t.start();

sum = 0;
for(i = 0; i < NUM; i++)
 sum = i * i;

t.stop();

printf("%d 次乘法计算用时 %f 秒\r\n", NUM, t.read());

while(1) {
 myled = !myled;
 wait(1.0); // 1 sec
}
}

```

运行MicroPython的结果是：

```
>>> test.add_test(1000000)
```

1000000 次加法计算测试用时： 2126868 us

```
>>> test.mul_test(1000000)
```

1000000 次乘法计算测试用时： 14458043 us

使用C语言计算的结果是：

Mbed计算测试

1000000 次加法计算用时 0.097270 秒

1000000 次乘法计算用时 0.101901 秒

加法上，MicroPython使用时间是C语言的21倍，而乘法是141倍。



计算任意精度的圆周率是个有趣的主题，得益于python的强大计算能力，我们在MicroPython中也可以轻松的计算pi的数值。

先输入下面的代码：

```
"""
文件: pi.py
说明: 用MicroPython计算任意精度圆周率计算
作者: 未知
版本
时间:
修改: 邵子扬
 2016.5 v1.1
 http://bbs.micro-python.com/forum.php
"""
import time

def pi(places=10):
 # 3 + 3*(1/24) + 3*(1/24)*(9/80) + 3*(1/24)*(9/80)*(25/168)
 # The numerators 1, 9, 25, ... are given by (2x + 1) ^ 2
 # The denominators 24, 80, 168 are given by (16x^2 -24x + 8)
 extra = 8
 one = 10 ** (places+extra)
 t, c, n, na, d, da = 3*one, 3*one, 1, 0, 0, 24

 while t > 1:
 n, na, d, da = n+na, na+8, d+da, da+32
 t = t * n // d
 c += t
 return c // (10 ** extra)

def pi_t(n=10):
 t1=time.ticks_us()
 t=pi(n)
 t2=time.ticks_us()
 print('elapsed: ', time.ticks_diff(t1,t2)/1000000, 's')
 return t

def pi2(n=10):
 r=6*(10**n)*1000
 p=0
 k=0
 c=r//2
```

```

d=c//(2*k+1)
while d>0:
 p=p+d
 k=k+1
 k2=2*k
 c=c*(k2-1)/(4*k2)
 d=c/(k2+1)
return p//1000

def pi2_t(n=10):
 t1=time.ticks_us()
 t=pi2(n)
 t2=time.ticks_us()
 print('elapsed: ', time.ticks_diff(t1,t2)/1000000, 's')
 return t

```

我们将它复制到STM32F7DISC中（STM32F7DISC已经下载了MicroPython固件），然后测试一下计算速度。（在其它MicroPython开发板上也可以进行这个测试）

```

MicroPython v1.8 on 2016-05-15; F7DISC with STM32F746
Type "help()" for more information.
>>> import pi
>>> np=pi.pi_t(1000)
elapsed:??0.221486 s
>>> np=pi.pi_t(2000)
elapsed:??0.793141 s
>>> np=pi.pi_t(5000)
elapsed:??4.981964 s
>>> np=pi.pi_t(10000)
elapsed:??21.02012 s
>>>

```

从运行结果可以看出，虽然不是很快，但是考虑到STM32的资源 and 性能，结果已经出乎预料了，毕竟计算函数还不到10行代码，没有做深度优化。最后打印出1000位的圆周率，大家可以和标准结果比较看看。

```

>>> pi.pi(1000)
314159265358979323846264338327950288419716939937510
>>>

```

ESP8266的MicroPython（也包括pyb版）使用了FAT磁盘格式。大家知道FAT格式是有一个专门的（File Allocation Table文件分配表），一旦FAT被破坏，文件就无法访问了。如果不安全退出磁盘，pyboard上文件系统被容易损坏，也是这个原因造成的。

国外网友Jon Schneider因此将spiffs文件系统（SPI Flash File System）移植到MicroPython上希望解决这个问题。目前只提供了测试版本，没有源码，大家可以下载固件试试。

## [固件下载](#)

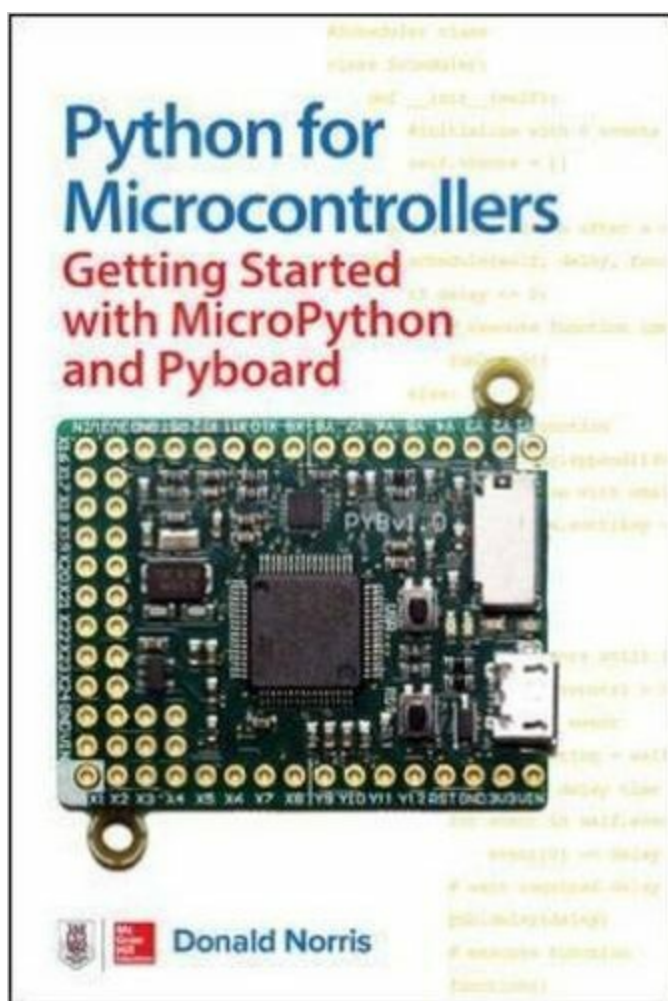
使用方法：

```
import spiffs
fs = spiffs.SPIFFS()

open = fs.open
with fs.open('log.txt', 'a') as f: # Only file modes r, w, a and
+
f.write('Isn't this fun')
fs.rename('oldname', 'newname')
fs.remove('name')
```

注：目前好像spiffs和FAT系统会共存，使用fs.listdir()和os.listdir()得到的结果不同。

第一本MicroPython书籍即将在亚马逊上开售



<https://www.amazon.com/Python-Microcontrollers-Getting-Started-MicroPython/dp/1259644537/>

《微控制器的Python：MicroPython和Pyboard指南》

立即用MicroPython建立和编辑自己的电子项目

这本实用的指南介绍了用micropython在开源硬件平台pyboard上编程。这本书一步一步介绍如何建立与开发板连接，安装必要的软件和程序开发。

本书由有丰富经验的爱好者编写，Python开发单片机：micropython和pyboard入门指南，怎样开始，DIY项目，清楚地展示每个技术。您将学习如何使用内置的传感器、存储数据到SD卡，控制LCD和矩阵键盘，通过Web界面 — 甚至建立一个很酷的机器人车！从那里，你会发现如何，编程和解决各种有趣的和实际的项目。

- 作为业余爱好者的引导，以及针对工程师和技术人员介绍
- 功能清晰的解释，以及说明的例子，和动手的项目
- 书写清晰，由有经验的创客书写易于遵循的语言

---

## 英文介绍

### 《Python for Microcontrollers: Getting Started with MicroPython》

Build and program your own electronics projects with MicroPython in no time!

This practical guide offers a hands-on introduction to Python-based microcontroller programming with MicroPython and the open-source Pyboard hardware platform. The book shows, step-by-step, how to set up and interface with the board, install the necessary software, and develop custom MicroPython programs.

Written by an experienced hobbyist, Python for Microcontrollers: Getting Started with MicroPython and Pyboard features start-to-finish, DIY projects that clearly demonstrate each technique. You will learn how to use the built-in sensor, store data to an SD card, control the LCD and matrix keyboard, interface with the Web—even build a

cool robotic car! From there, you will discover how to assemble, program, and troubleshoot all kinds of entertaining and practical projects of your own.

- Serves both as a hobbyists' guide and as an introduction for engineers and techs
- Features clear explanations, well-illustrated examples, and hands-on projects
- Written in clear, easy-to-follow language by an experienced maker

又一个使用micropython的众筹项目：OpenMV Cam。可以给任何项目增加机器视觉。



<https://www.kickstarter.com/projects/botthoughts/openmv-cam-embedded-machine-vision/description>

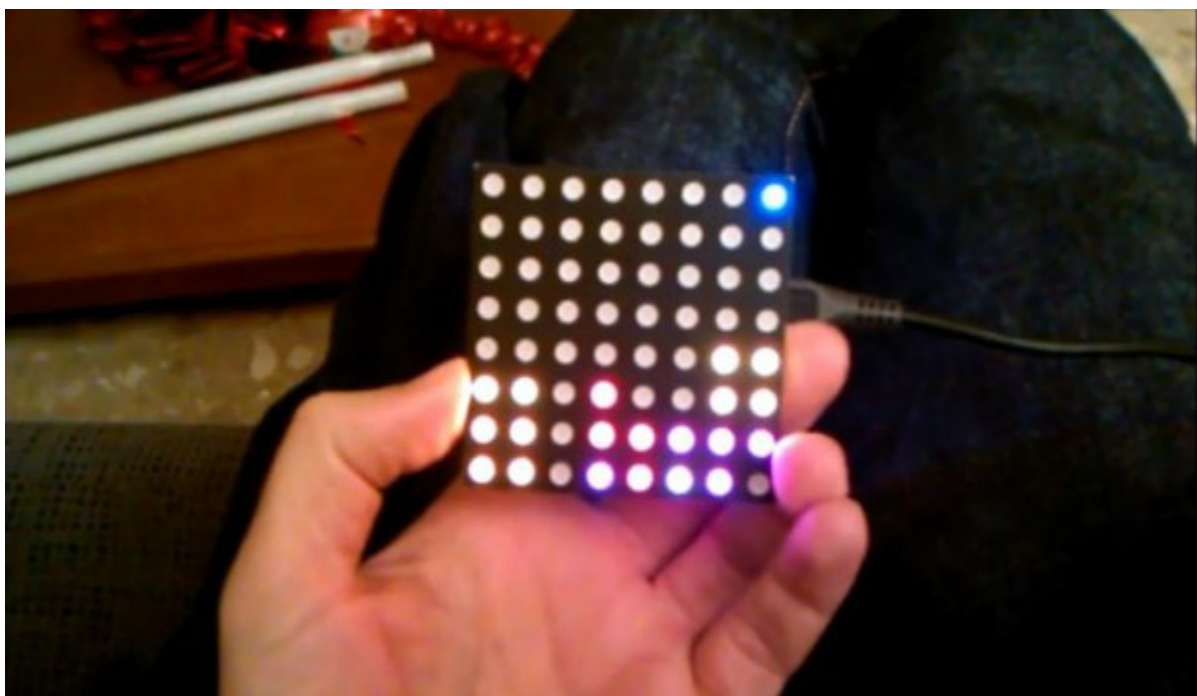
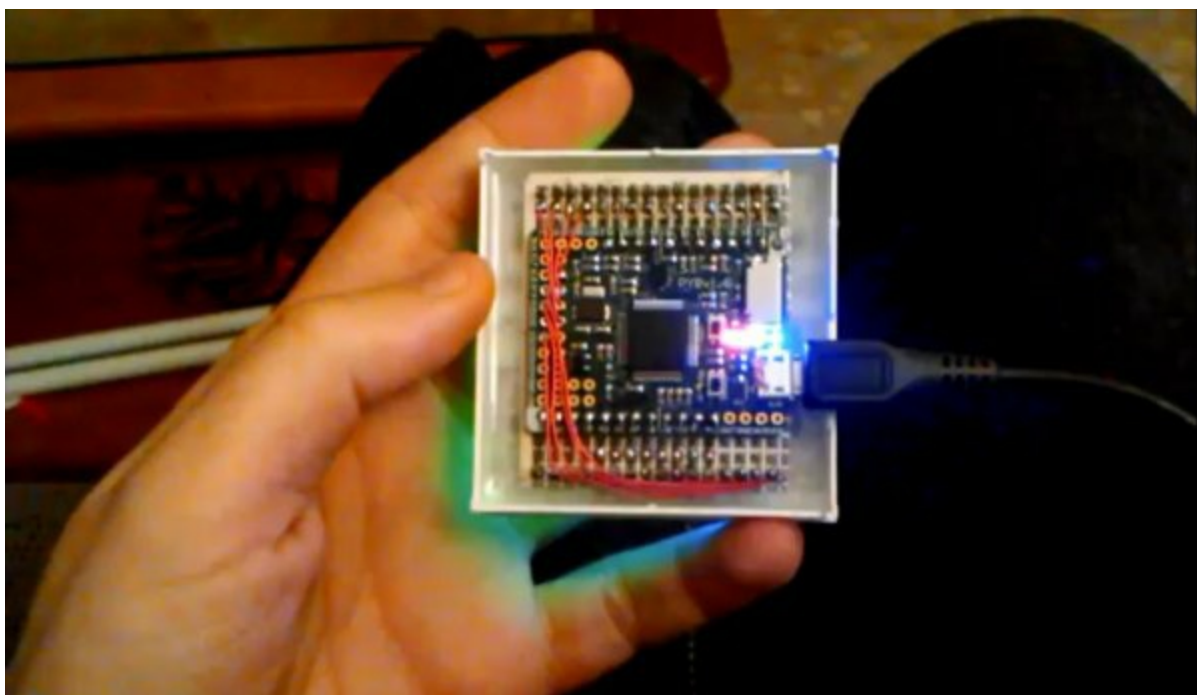
以前的Logo是



现在改为贪吃蛇了。



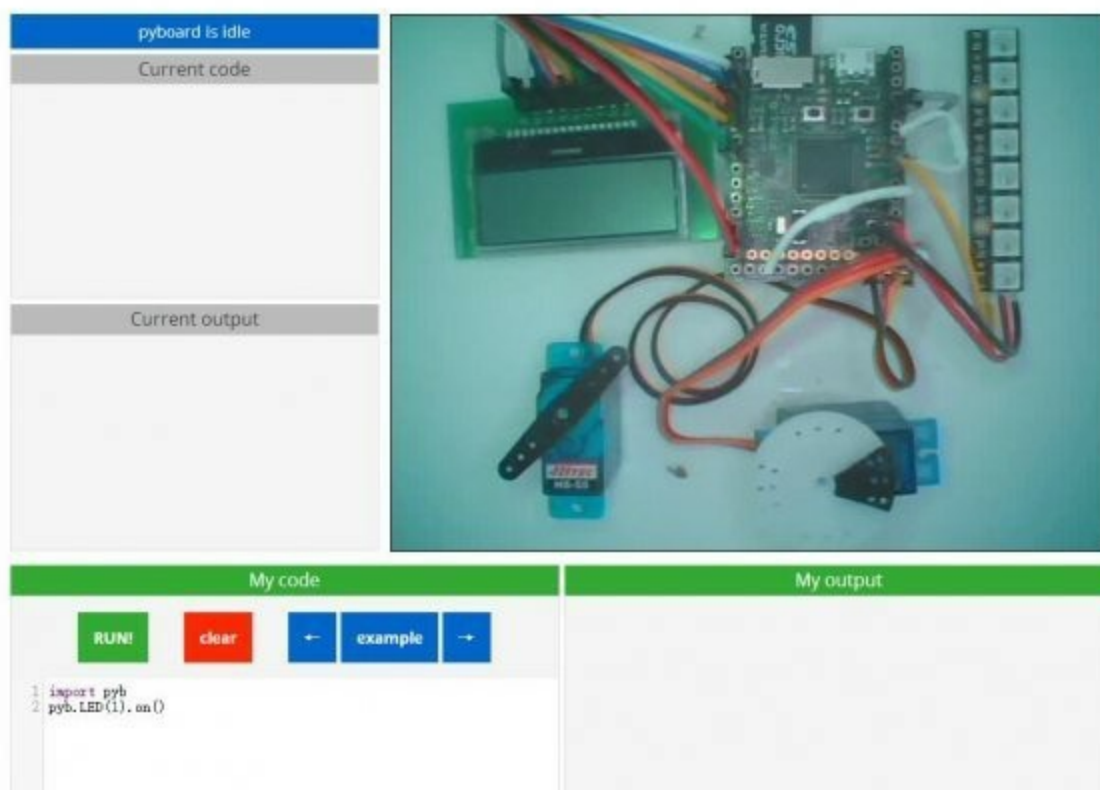




[http://v.youku.com/v\\_show/id\\_XMTUxMDAxNzQ2O](http://v.youku.com/v_show/id_XMTUxMDAxNzQ2O)

在MicroPython的官网，提供了一个在线体验MicroPython的方法，可以在浏览器中输入代码，观察运行效果。

这个网页有些慢，也可能是只允许同时接入一个用户（只有一个板子），所以如果连接不上，可以多试几次，或者等一段时间再试。



<http://micropython.org/live/>

目前已经举行的各种活动

- 2016年3月 [MicroPython抢鲜玩 Python遇上MCU=? 由你定](#)
- 2016年6月 [免费提供Nucelo扩展板，将Nucleo开发板变为MicroPython](#)
- 2016年8月 [盛夏嗨学有礼——和dcexpert一起学ESP8266](#)
  - [【福利真的来了】 免费玩Micropython开发板！你和它会有什么火花？](#)
- 2016年10月 [PYB Nano 开发板包邮团购活动开始了](#)
  - [爱板网团购](#)
- 2016年11月 ICKey [MicroPython 创客开发板试用活动](#)

# 目录

|                 |    |
|-----------------|----|
| 关于              | 2  |
| 基本用法            | 3  |
| MicroPython简介   | 4  |
| pyboard         | 6  |
| 快速指南            | 7  |
| 教程              | 11 |
| GPIO            | 12 |
| PYB中未公开的Pin用法   | 15 |
| Pyboard中Pin对应关系 | 16 |
| LED             | 18 |
| 按键              | 19 |
| RTC             | 20 |
| ADC             | 22 |
| DAC             | 24 |
| UART            | 26 |
| Timer           | 28 |
| PWM             | 33 |
| SPI             | 34 |
| I2C             | 36 |
| 外中断             | 38 |
| USB VCP (虚拟串口)  | 40 |
| 文件操作            | 42 |
| macroSD卡        | 43 |
| MicroPython库    | 43 |
| pyboard相关库      | 43 |
| pyb             | 44 |
| 标准库             | 50 |
| buildin 函数      | 51 |
| cmath           | 53 |
| gc              | 54 |
| math            | 55 |

|                     |     |
|---------------------|-----|
| select              | 59  |
| sys                 | 61  |
| ubinascii           | 64  |
| ucollections        | 65  |
| uhashlib            | 66  |
| uheapq              | 67  |
| uio                 | 68  |
| ujson               | 69  |
| uos                 | 70  |
| ure                 | 72  |
| usocket             | 74  |
| ustruct             | 78  |
| utime               | 79  |
| uzlib               | 82  |
| MicroPython库        | 83  |
| machine             | 84  |
| micropython         | 87  |
| network             | 88  |
| uctypes             | 92  |
| 其它                  | 96  |
| 用寄存器控制PYB的LED       | 97  |
| 恢复出厂设置              | 98  |
| MicroPython 如何嵌入汇编  | 99  |
| 升级固件                | 102 |
| 在Linux下更新固件         | 104 |
| ESP8266             | 105 |
| 快速参考                | 106 |
| 教程                  | 111 |
| Pin                 | 112 |
| ADC                 | 115 |
| I2C                 | 116 |
| ESP8266和PYB的I2C用法对比 | 119 |
| SPI                 | 120 |
| RTC                 | 122 |

|                              |     |
|------------------------------|-----|
| Timer                        | 124 |
| PWM                          | 126 |
| UART                         | 127 |
| 标准库                          | 129 |
| array                        | 130 |
| gc                           | 131 |
| math                         | 132 |
| MicroPython库                 | 132 |
| machine                      | 133 |
| 工具                           | 135 |
| 文件传输                         | 136 |
| webrepl                      | 136 |
| webrepl的用法                   | 137 |
| uPyLoader                    | 138 |
| 安装uPyLoader                  | 139 |
| uPyLoader使用教程                | 140 |
| 常见问题                         | 144 |
| ESPyHarp                     | 144 |
| ESPyHarp 简介                  | 145 |
| mpfshell                     | 147 |
| 编程工具                         | 147 |
| esptool.py                   | 147 |
| 安装esptool.py                 | 148 |
| 读取flash                      | 149 |
| 清除Flash内容                    | 150 |
| esptool.py 升级到 1.2.1         | 151 |
| 其它                           | 151 |
| 调试ESP8266时不要用超级终端            | 152 |
| MicroPython 1.8.6重新支持512K的模块 | 153 |
| ESP8266挂载SD卡                 | 154 |
| 创建ESP8266工具链                 | 156 |
| ESP32                        | 157 |
| micro:bit                    | 157 |
| micro:bit 硬件方案               | 158 |

|                             |     |
|-----------------------------|-----|
| 在BLE400上体验microbit的运行效果     | 159 |
| CC3200                      | 161 |
| 软件库                         | 161 |
| 旋转编码器库                      | 162 |
| telenet服务器                  | 163 |
| 技巧                          | 163 |
| 在ubuntu中安装gcc-arm-none-eabi | 164 |
| 用git克隆micropython仓库         | 165 |
| 编译时指定编译器路径                  | 166 |
| 加快编译速度                      | 167 |
| 快捷键                         | 168 |
| Linux版的Micropython          | 169 |
| 在Linux下用Screen连接pyboard     | 170 |
| 怎样在REPL下粘贴程序                | 171 |
| 添加目录到系统路径                   | 173 |
| SPI方式连接SD卡                  | 174 |
| 故障修复                        | 174 |
| pyb识别不能pybflash             | 175 |
| pyb连接问题USB                  | 176 |
| 开发板                         | 177 |
| 开发板固件                       | 178 |
| EEWORLD版 PYBV10             | 178 |
| EEWORLD版pyboard说明           | 179 |
| EEWORLD版使用指南                | 181 |
| STM32L476版的PYBV1.0          | 184 |
| ESP-mp-01                   | 185 |
| ESP-MP-01开发板说明              | 186 |
| ESP-MP-01开发板使用指南            | 188 |
| 入门教程                        | 188 |
| 1                           | 189 |
| 2                           | 191 |
| 3                           | 192 |
| 4                           | 195 |

|                              |     |
|------------------------------|-----|
| 5                            | 198 |
| 6                            | 200 |
| 连接Micropython热点              | 202 |
| win10的驱动问题                   | 203 |
| PYB Nano                     | 204 |
| 原理图                          | 205 |
| PYB Nano 开发板快速指南             | 206 |
| 连接DS3231模块                   | 215 |
| 使用EEPROM                     | 217 |
| 驱动气压传感器BMP180                | 219 |
| 使用HMC5883                    | 220 |
| 连接SD卡                        | 222 |
| 用定位器控制LED亮度                  | 224 |
| ST Nucleo-F401RE             | 225 |
| ST Nucleo-F411RE             | 226 |
| 让NUCLEO-F411支持SD卡            | 227 |
| 将NUCLEO上的I2C1改为PB8/PB9       | 228 |
| 在NUCLEO-F411RE上使用MicroPython | 231 |
| ST Nucleo-L476RG             | 232 |
| 使用DS3231                     | 233 |
| 使用按键控制LED的频率                 | 234 |
| ST NUCLEO-F446RE             | 235 |
| ST NUCLEO-F446ZE             | 236 |
| ST NUCLEO-F746ZG             | 237 |
| NUCLEO_F746ZG上运行MicroPython  | 238 |
| ST NUCLEO-F767ZI             | 241 |
| 在NUCLEO_F767ZI上移植MicroPython | 242 |
| 用PWM控制LED亮度                  | 244 |
| ST NUCLEO-F429ZI             | 245 |
| ST DISCO-F429ZI              | 246 |
| ST DISCO-L476VG              | 247 |
| ST DISCO-F746NG              | 248 |
| STM32F4DISCOVERY             | 249 |
| 聚码 小钢炮开发板                    | 250 |



|                                 |     |
|---------------------------------|-----|
| LPS25H驱动                        | 251 |
| 读取HTS221传感器                     | 252 |
| 在小钢炮开发板上运行MicroPython           | 255 |
| CC3200LaunchXL                  | 260 |
| 在CC3200-LAUNCHXL上运行MicroPython  | 261 |
| CC3200-LAUNCHXL的正确烧写固件方法        | 278 |
| MSP432R                         | 282 |
| MSP432 LaunchPad的固件             | 283 |
| 在MSP432 LaunchPad上运行MicroPython | 284 |
| FRDM-K64F                       | 289 |
| Zephyr分支加入GPIO功能                | 290 |
| XMC4700                         | 290 |
| IntoRobot Neutron               | 291 |
| DFU模式                           | 292 |
| 移植MicroPython到STM32F411CEU6     | 294 |
| 用LED测光强                         | 295 |
| 控制RGB彩灯                         | 297 |
| 小e开发快                           | 298 |
| 读取ADC                           | 299 |
| 判断小e的拨码开关                       | 300 |
| 读取小e开发板上的BMP180传感器              | 302 |
| 读取DHT11传感器                      | 304 |
| 同步网络时间                          | 305 |
| 可以运行MicroPython的TiLDA MK3       | 306 |
| 资源                              | 306 |
| 论坛                              | 307 |
| 常用终端软件                          | 308 |
| 其它相关资源                          | 309 |
| 官方资源                            | 310 |
| 其它                              | 310 |
| python入门神图                      | 311 |
| 计算性能测试                          | 312 |
| 计算性能测试2                         | 315 |
| 使用MicroPython计算任意位数圆周率          | 317 |

|                           |     |
|---------------------------|-----|
| 支持spiffs格式的移植版            | 319 |
| 第一本MicroPython书籍即将在亚马逊上开售 | 320 |
| 正在众筹的OpenMV Cam           | 323 |
| MicroPython官网换Logo了       | 324 |
| MicroPython 视频分享 - 俄罗斯方块  | 325 |
| 在线体验MicroPython           | 326 |
| 已举行的MicroPython的活动        | 327 |