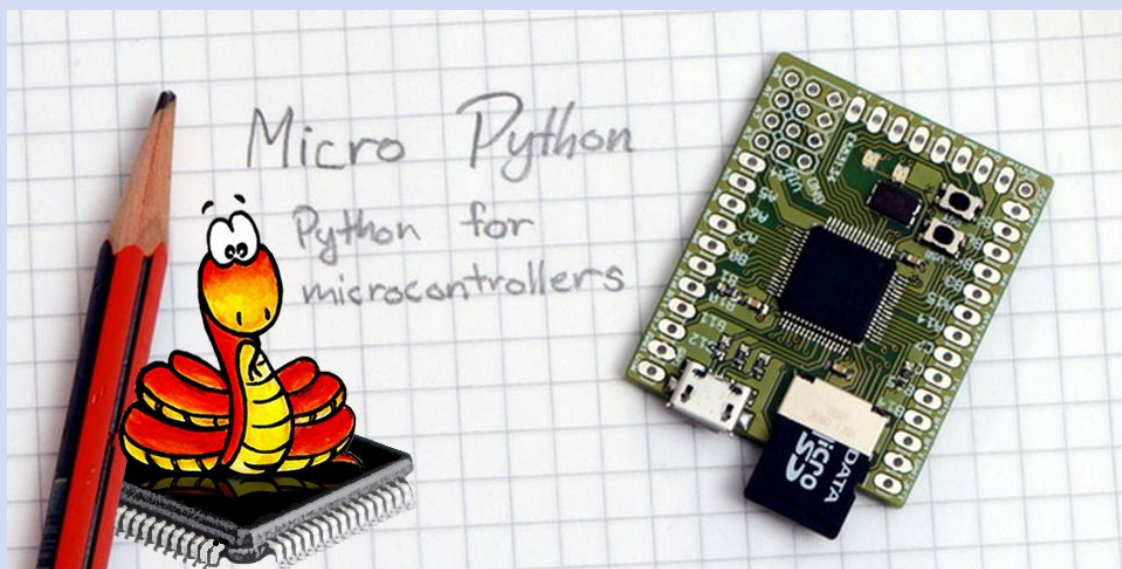


MicroPython中文教程

V1.0

Micropython中文论坛

<http://bbs.micro-python.com/>



shaoziyang

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

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
```

注意不能有空格

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。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

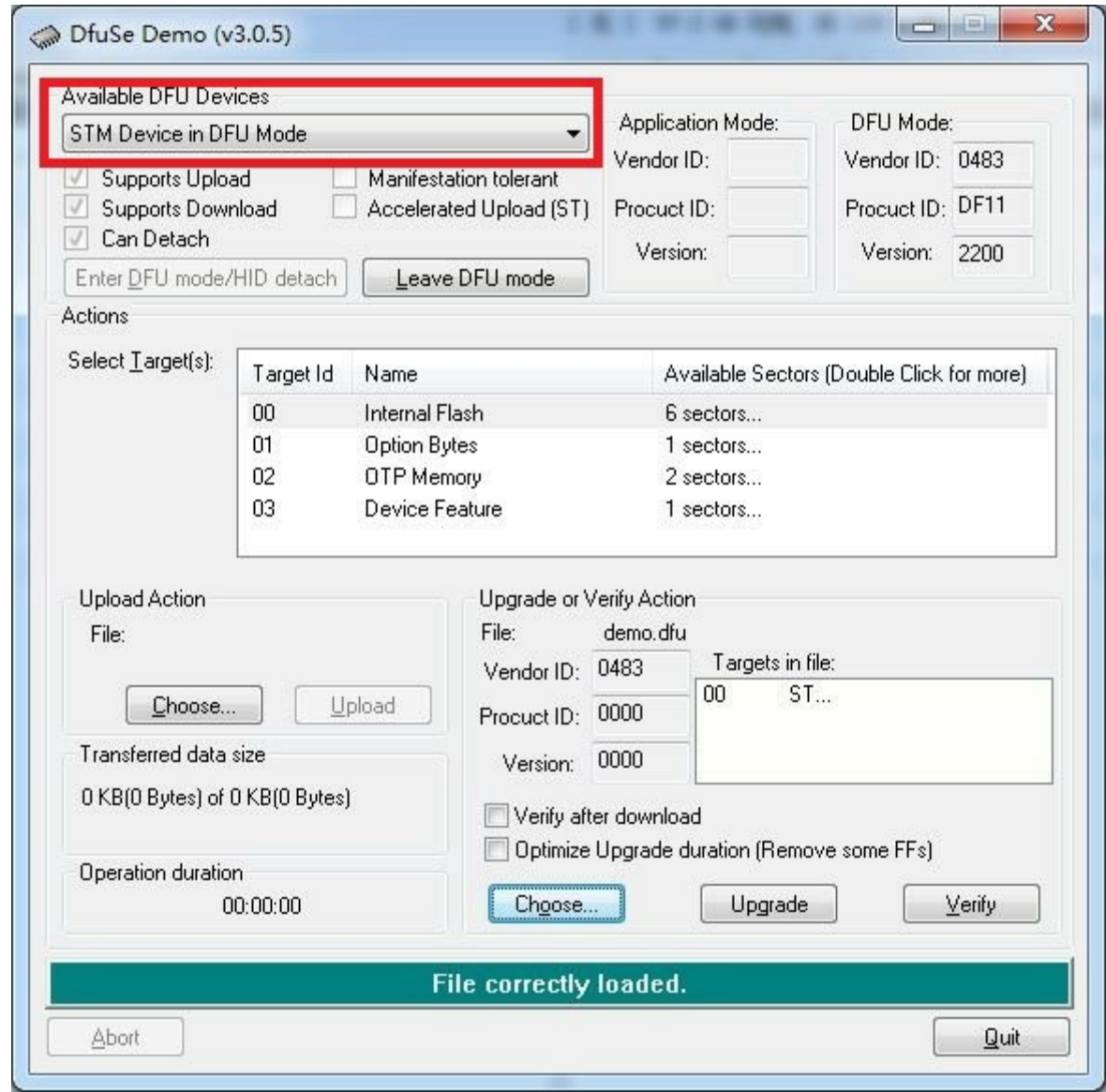
CTRL-E -- 粘贴模式

上下方向键 -- 调出以前输入命令

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/SS1533/PF257916>
- 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/>

•

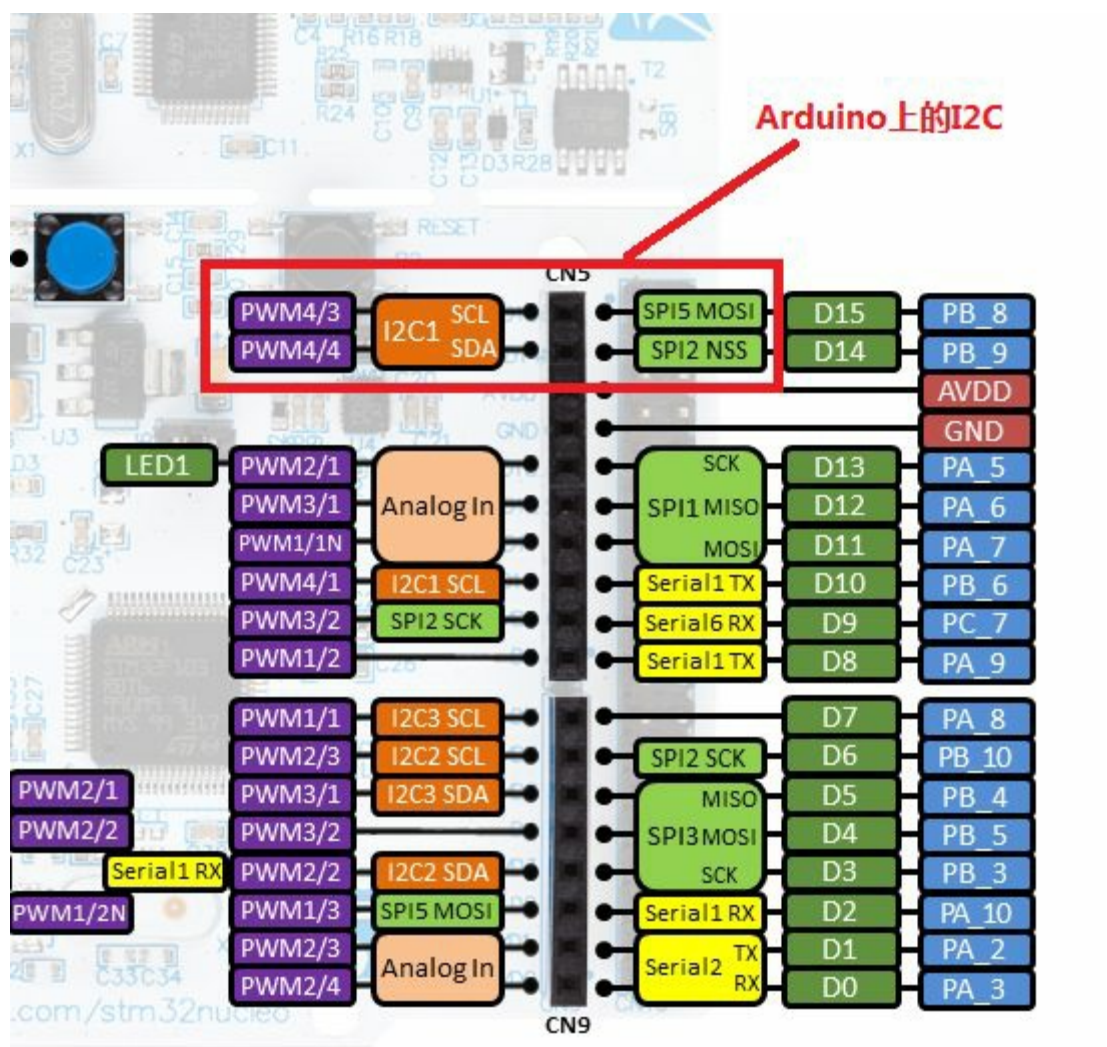
其他

- 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

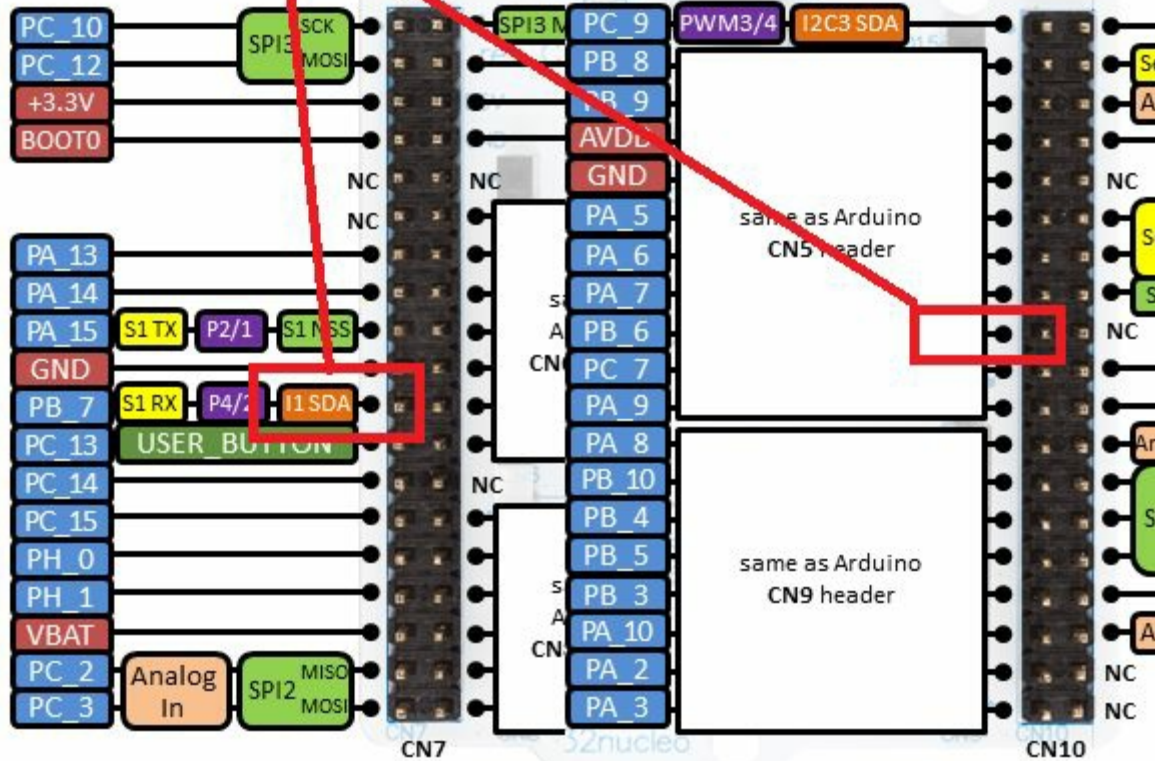
因为某些原因造成pyboard故障，可以恢复到出厂设置，就像Windows系统重新用ghost恢复一样。

- 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。

在MicroPython的NUCLEO_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。



MicroPython默认的I2C1



怎样将MicroPython的I2C改为和标准的Arduino方式呢？

首先用文本编辑器打开stmhal/board/NUCLE0_F411RE文件夹下的mpconfigboard.h文件，找到I2C相关的定义：

```
// I2C busses
#define MICROPY_HW_I2C1_SCL (pin_B6) // Arduino D10, pin 17 on CN10
#define MICROPY_HW_I2C1_SDA (pin_B7) // pin 21 on CN7
#define MICROPY_HW_I2C2_SCL (pin_B10) // Arduino D6, pin 25 on CN10
#define MICROPY_HW_I2C2_SDA (pin_B3) // Arduino D3, pin 31 on CN10
#define MICROPY_HW_I2C3_SCL (pin_A8) // Arduino D7, pin 23 on CN10
#define MICROPY_HW_I2C3_SDA (pin_C9) // pin 1 on CN10
```

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。

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。

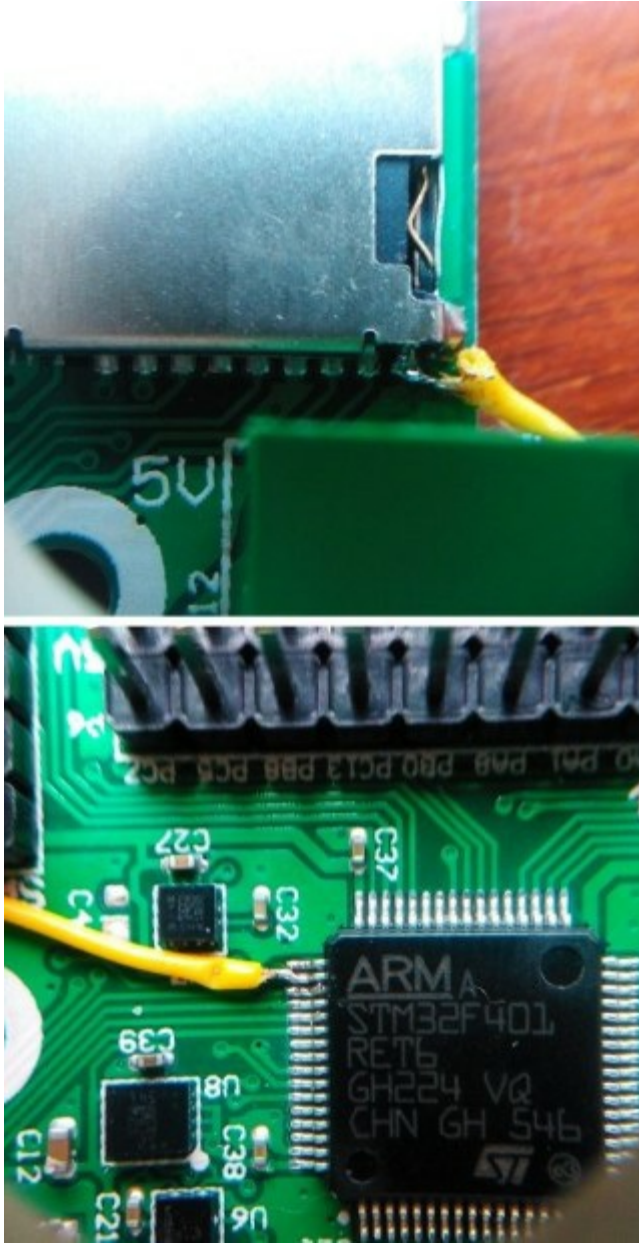
1. 首先要下载并安装GNU Tools for ARM Embedded Processors。
 1. <https://launchpad.net/gcc-arm-embedded>
2. 下载并安装ST的DfuSe软件，<http://www.st.com/web/en/catalog/tools/FM147/CL1794/SC961/SS1533/PF257>
3. 下载MicroPython的源码，[micropython-master.zip](#)。
4. 展开MicroPython源码，打开 stmhal\boards\ 目录
5. 新建一个CANNON目录，将NUCLEO_F401RE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BOOT0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)
tml=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。

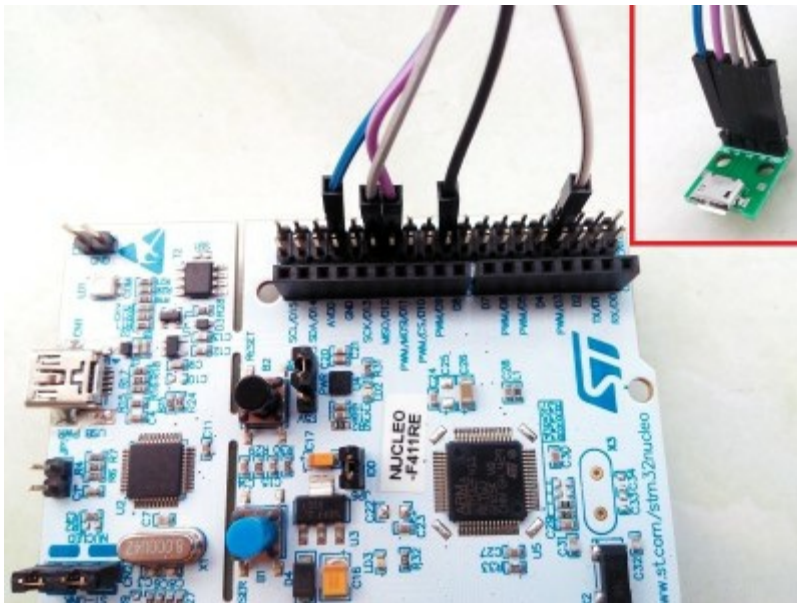
在NUCLE0-F411RE上使用MicroPython

在MicroPython的源码中，已经包含了NUCLE0-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NUCLE0-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NUCLE0-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。



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](#)等，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
```


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。

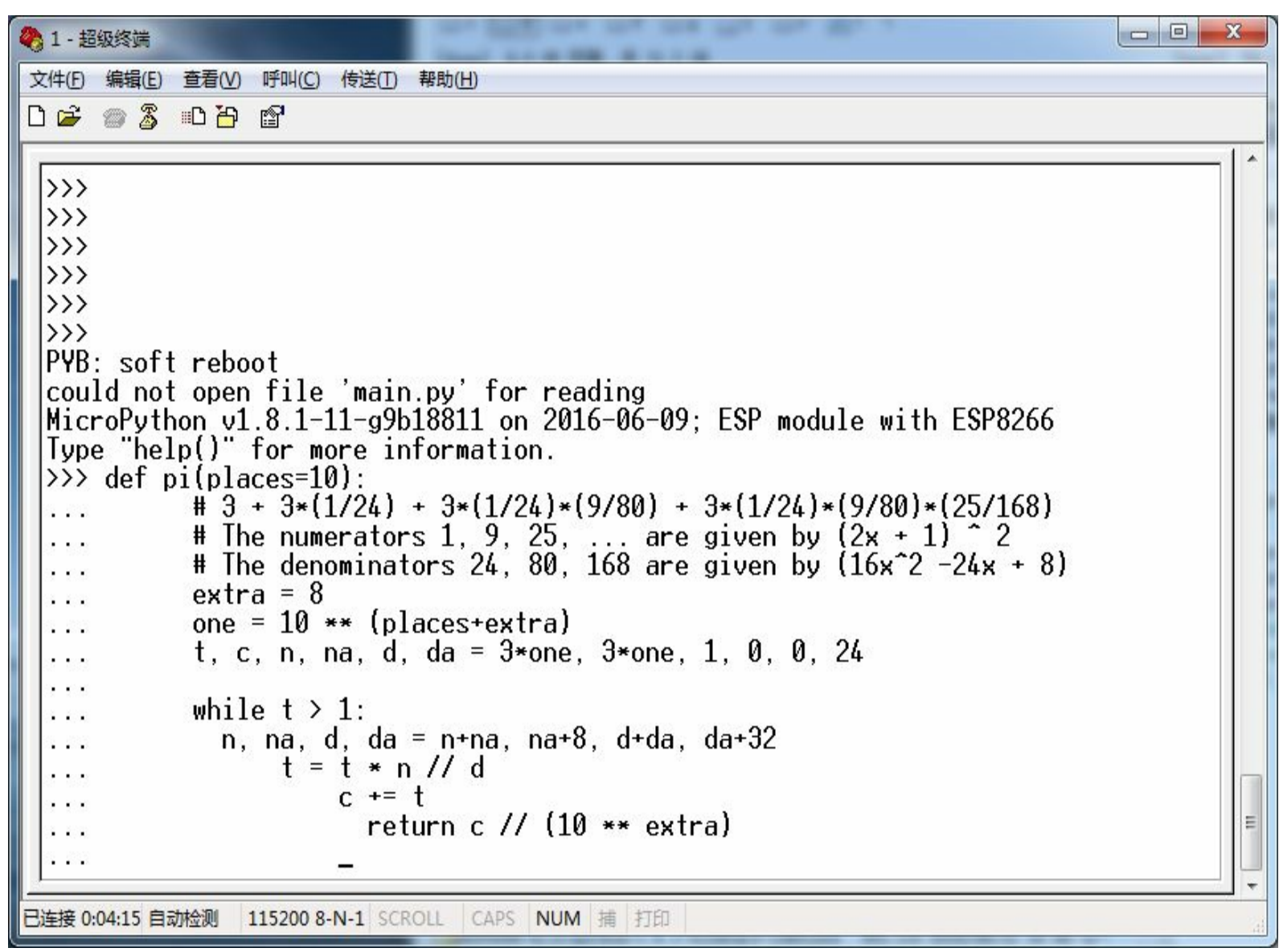
1. 首先需要安装Linux版的arm-gcc编译器
2. 安装dfu-util
`sudo apt-get install dfu-util`
3. 编译固件
4. 用dfu-util写入固件
`sudo make BOARD=XXXX deploy USE_PYDFU=0`
5. 也可以使用pydfu.py下载固件，需要先安装python-usb模块
`sudo apt-get install python-usb python3-usb`

```
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。



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。

1 - 超级终端

文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

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):
=== # $3 + 3 \cdot (1/24) + 3 \cdot (1/24) \cdot (9/80) + 3 \cdot (1/24) \cdot (9/80) \cdot (25/168)$
=== # 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 捕 打印

在线文档:

<http://docs.micropython.org/en/latest/pyboard/>

pdf版文档:

<http://docs.micropython.org/en/latest/micropython-pyboard.pdf>

昨天拿到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。

- [配置文件](#)

在ESP8266中，是以数字代表GPIO的，如：

```
LED = Pin(2, Pin.OUT)
```

而在pyboard中，是以名称代表GPIO，如：

```
Pin(Pin.cpu.A0, Pin.IN) Pin(Pin.board.X1, Pin.OUT)
```

这样显得名称很长，也不方便。其实，在pyboard中，也可以这样用

```
Pin('A13', Pin.OUT)
```

这样不但好记，也简短。

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



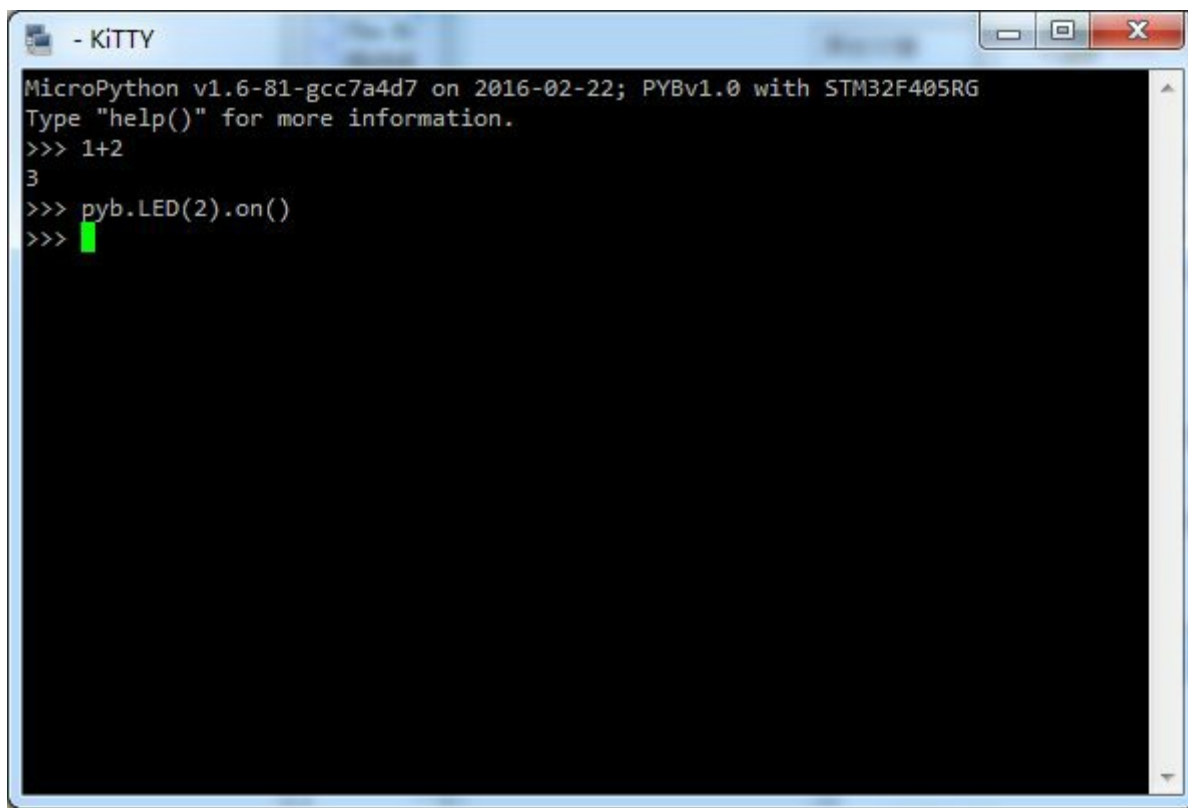
我们可以在main.py中增加代码，加入需要开机自动完成的功能。EEWORLD版的pyboard中，我在main.py中增加了一小段代码，用于LED自检，然后用LED3（橙色）做呼吸灯。参考程序可以在[EEWORLD版pyboard说明](#)中找到。

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。



```
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()
>>>
```

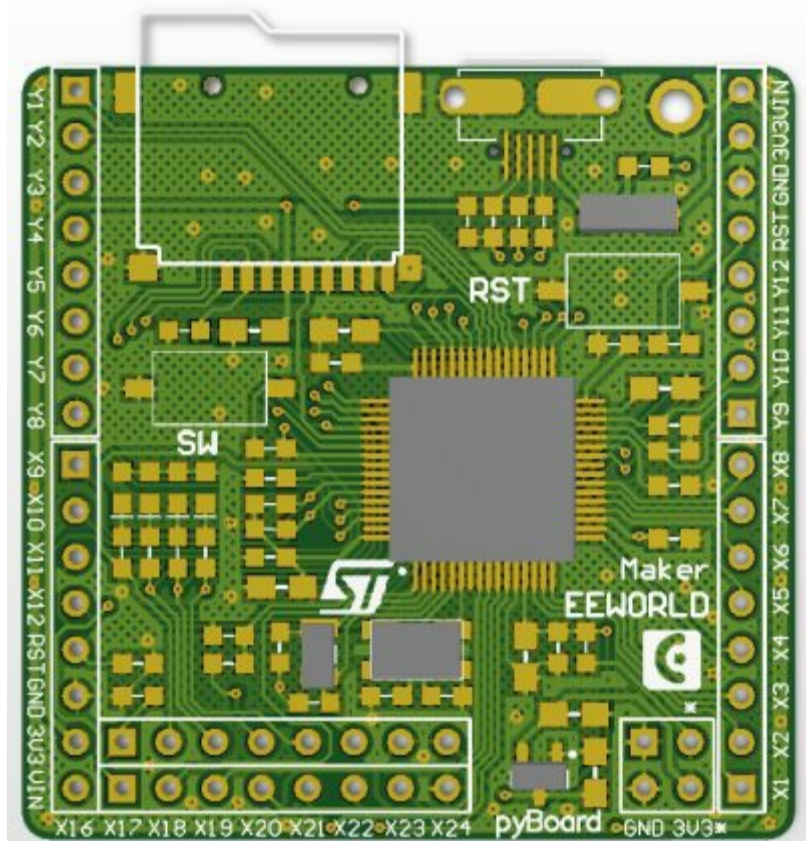
对于micropython的初学者，如果对python语言还不熟悉的，需要先看看python的基本教程，然后在开始。python比C语言简单一些，通常花一两天掌握初步的语法就可以开始玩了。如果对python比较了解，那么可以直接开始，看看LED、按键、定时器、串口的用法，很快就可以掌握基本方法。

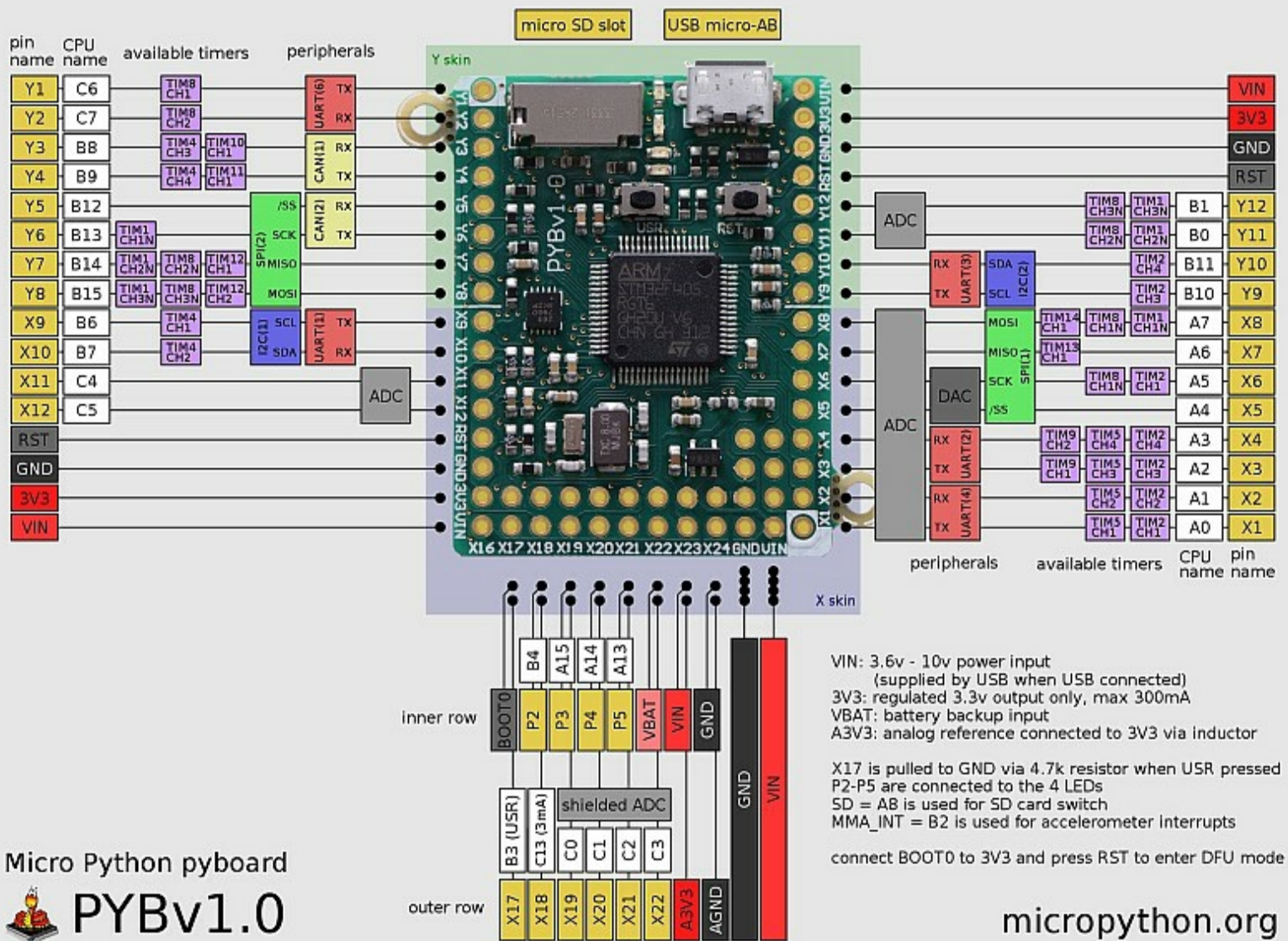
MicroPython的作者非常勤奋，软件升级也很快。我在调试和烧写固件时，版本是pybv10-2016-03-28-v1.6-310-g1937953.dfu，而写这个帖子的时候，固件已经升级到pybv10-2016-04-07-v1.6-379-g5e7fa7c.dfu。如果需要升级固件，可以参考一下教程中的[【MicroPython】怎样升级固件](#)。如果是在Linux下，参考[这里](#)？
<https://github.com/micropython/micropython/wiki/Pyboard-Firmware-Update>。如果你的pyboard没有正常运行，可以先尝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





General board control

```
import pyb
```

```
pyb.delay(50) # wait 50 milliseconds
pyb.millis() # number of milliseconds since bootup
pyb.repl_uart(pyb.UART(1, 9600)) # duplicate REPL on UART(1)
pyb.wfi() # pause CPU, waiting for interrupt
pyb.freq() # get CPU and bus frequencies
pyb.freq(60000000) # set CPU freq to 60MHz
pyb.stop() # stop CPU, waiting for external interrupt
```

LEDs


```
from pyb import LED
```

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

Pins and 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
```

Servo control

```
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
```

External interrupts

```
from pyb import Pin, ExtInt
```

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

Timers

```
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
```

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(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())
```

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

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')
```

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。

LED用法:

- `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

```
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)
```

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()
```

- 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** is 1-7 for Monday through Sunday.

subseconds counts down from 255 to 0

- 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

`vback = adc.read_core_vbat() * 1.21 / adc.read_core_vref()`

- `adc.read_core_vref()` 读取vref电压（1.21V参考）

`3V3 = 3.3 * 1.21 / adc.read_core_vref()`

- `adc.read_timed(buf, timer)` 以指定频率读取adc参数到buf
○ `buf`，缓冲区
○ `timer`，频率（Hz）

使用这个函数会将ADC的结果限制到8位

```
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
```

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

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`: 读取打一个字节超时时间 (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)

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）

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。

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。

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。

- `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()
```

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

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

- 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。

使用库之前，需要先import库

- `pyb.delay(ms)`
延时XX毫秒
- `pyb.udelay(us)`
延时XX微秒
- `pyb.millis()`
上电后运行时间（毫秒）
- `pyb.micros()`
上电后运行时间（微秒）
- `pyb.elapsed_millis(start)`
从start开始到现在的时间（毫秒）
- `pyb.elapsed_micros(start)`
和上面类似，单位是微秒
- `pyb.hard_reset()`
硬件复位，和按下复位键功能一样。在windows下这样会丢失VCP串口连接，必须插拔一次USB线才能恢复。
- `pyb.bootloader()`
直接进入Bootloader，无需按下BOOT0。
- `pyb.disable_irq()`
禁止中断，返回以前的中断状态
- `pyb.enable_irq(state=True)`
允许/禁止中断
- `pyb.freq([sysclk[, hclk[, pclk1[, pclk2]]]])` ○ `sysclk`: CPU时钟频率（有效值为 8, 16, 24, 30, 32, 36, 40, 42, 48, 54, 56, 60, 64, 72, 84, 96, 108, 120, 144, 168.）
 - `hclk`: AHB 总线、核心内存和 DMA 频率
 - `pclk1`: APB1 总线频率
 - `pclk2`: APB2 总线频率

注意修改时钟频率后CPU会重启，在命令方式下会丢失VCP串口连接。

最大频率分别是168M/168M/42M/84M，不要超过这个限制。

- `pyb.wfi()`
等待中断
- `pyb.stop()`
进入睡眠模式，功耗降低到500uA。需要通过外中断或RTC唤醒。进入停止模式会丢失VCP连接。
- `pyb.standby()`
深度睡眠（休眠）模式，功耗进一步降低到50uA。只能通过RTC、X1 (PA0=WKUP) 或 X18 (PC13=TAMP1) 外中断唤醒。
- `pyb.have_cdc()`
如果USB做为串口设备连接，返回True
- `pyb.info([dump_alloc_table])`
开发板信息
- `pyb.main(filename)`
指定boot.py完成后自行的程序。这个函数只有在boot.py中调用才有效
- `pyb.mount(device, mountpoint, *, readonly=False, mkfs=False)`
 - device，加载块设备做为文件系统一部分。device 必须是提供下面协议的对象，readblocks 和 writeblocks 在块设备和buf之间复制数据，buf是512倍数的 bytearray，如果writeblocks没有定义那么设备是只读的。count返回设备中块的数量，sync用于数据同步。
 - `readblocks(self, blocknum, buf)` ■ `writeblocks(self, blocknum, buf)` (optional) ■ `count(self)` ■ `sync(self)` (optional)
 - mountpoint，加载点，需要带有前导斜杠/
 - readonly，是否为只读
 - mkfs，创建文件系统如果它不存在

卸载设备，将device设置为None，而mountpoint需要设置。
- `pyb.repl_uart(uart)`
获取或设定REPL使用的串口对象
- `pyb.rng()`
产生30位硬件随机数

- `pyb.sync()`
同步文件系统
- `pyb.unique_id()`
CPU的唯一序列号 (UID)

system specific functions

需要先import sys

- `sys.exit([retval])` 产生`SystemExit` 异常，如果给出参数，参数将传递到`SystemExit`。这个函数会引起软复位。
- `sys.print_exception(exc[, file])`
- `sys.argv` 程序的参数列表
- `sys.byteorder` 系统字节顺序，“little” or “big”
- `sys.path` 系统搜索路径
- `sys.platform` 系统名称。不同板子中名称不同，在pyboard上名称是 “pyboard” 。
- `sys.implementation` name - string “micropython”
version - tuple (major, minor, micro), e.g. (1, 7, 0)
- `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，字符串方式
- `sys.version_info` python语言版本，int元组方式

- `micropython.mem_info([verbose])`

当前系统内存状态

- `micropython.qstr_info([verbose])`

内部字符串状态

- `micropython.alloc_emergency_exception_buf(size)`

分配关键缓存

在MicroPython的源码中，带有了单总线的驱动，可以很方便的驱动单总线器件，如读取温度传感器DS1820。

首先在MicroPython的源码目录中，进入drivers\onewire\目录，然后将目录下的文件ds18x20.py和onewire.py复制到PYBFLASH磁盘的根目录。复制文件后要安全退出磁盘，然后重新接入，不然找不到文件。

用Y11、Y10、Y9三个引脚做为DS1820的控制，其中Y11是GND，Y9是VCC，Y10是DQ。

先将DS1820接到Y11、Y10、Y9，然后输入下面代码：

```
>>> from pyb import Pin
>>> Pin("Y11", Pin.OUT_PP).low()
>>> Pin("Y9", Pin.OUT_PP).high()
>>> pyb.delay(100)
>>> from ds18x20 import DS18X20
>>> d = DS18X20(Pin('Y10'))
>>> d.read_temp()
34.0
>>> d.read_temp()
33.375
>>> d.read_temps()
[32.625]
>>>
```

可以看到已经可以读出传感器的温度。用d.read_temp()可以读取一个传感器，默认是第一个传感器，d.read_temp(d.roms[1])可以读取第二个传感器；如果只有一个传感器，可以用d.read_temp(rom=None)忽略地址

用d.read_temps()可以读取全部传感器。

下面连接两个DS18B20进行测试

```
>>> from pyb import Pin
>>> Pin("Y11", Pin.OUT_PP).low()
>>> Pin("Y9", Pin.OUT_PP).high()
>>> pyb.delay(100)
>>> from ds18x20 import DS18X20
>>> d = DS18X20(Pin('Y10'))
>>> d.read_temp()
33.75
>>> d.read_temp(d.roms[0])
33.875
```

```
>>> d.read_temp(d.roms[1])
33.625
>>> d.read_temps()
[33.5625, 32.75]
>>> d.read_temp()
34.0625
>>> d.read_temps()
[34.0, 34.75]
```

注：

- 如果DS1820没有连接好就输入了`d = DS18X20(Pin('Y10'))`命令，会因为搜索不到器件而出错。
- 千万不要接反VCC和GND，不然会损坏传感器。
- 不同开发板上，可能需要调整`onewire.py`中的延时时间。例如在STM32F746DISC上，需要修改为：
`self.write_delays = (6, 64, 60, 10)`
`self.read_delays = (6, 9, 55)`

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/si7005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参数数。

```
# 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)
```

先将hts221.py复制到小钢炮的PYFLASH磁盘，然后就可以使用函数读取温度湿度

```
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

```
>>>
```

```
"""
```

```
DS3231 RTC drive  
Author: shaoziyang  
2016.May
```

```
>>> from DS3231 import DS3231
```

```
>>> ds = DS3231(1)
```

```
>>> ds.sec()
```

```
>>> ds.sec(10)
```

```
>>> ds.TIME()
```

```
>>> ds.TEMP()
```

```
"""
```

```
import pyb
```

```
from pyb import I2C
```

```
DS3231_ADDR = const(0x68)  
DS3231_REG_SEC = const(0x00)  
DS3231_REG_MIN = const(0x01)  
DS3231_REG_HOUR = const(0x02)  
DS3231_REG_WEEKDAY= const(0x03)  
DS3231_REG_DAY = const(0x04)  
DS3231_REG_MONTH = const(0x05)  
DS3231_REG_YEAR = const(0x06)  
DS3231_REG_A1SEC = const(0x07)  
DS3231_REG_A1MIN = const(0x08)  
DS3231_REG_A1HOUR = const(0x09)  
DS3231_REG_A1DAY = const(0x0A)  
DS3231_REG_A2MIN = const(0x0B)  
DS3231_REG_A2HOUR = const(0x0C)  
DS3231_REG_A2DAY = const(0x0D)  
DS3231_REG_CTRL = const(0x0E)  
DS3231_REG_STA = const(0x0F)  
DS3231_REG_OFF = const(0x10)  
DS3231_REG_TEMP = const(0x11)
```

```
class DS3231(object):
```

```
def __init__(self, i2c_num):
```

```
self.i2c = I2C(i2c_num, I2C.MASTER, baudrate = 100000)
```

```
def DATE(self, dat=[]):
```

```
if dat==[]:
```

```
t = []
```

```
t.append(self.year())
```

```
t.append(self.month())
```

```
t.append(self.day())
```

```
return t
```

```
else:
```

```
self.year(dat[0])
```

```
self.month(dat[1])
```

```
self.day(dat[2])
```

```
def TIME(self, dat=[]):
    if dat==[]:
        t = []
        t.append(self.hour())
        t.append(self.min())
        t.append(self.sec())
    return t
    else:
        self.hour(dat[0])
        self.min(dat[1])
        self.sec(dat[2])

def DateTime(self, dat=[]):
    if dat==[]:
        return self.DATE() + self.TIME()
    else:
        self.year(dat[0])
        self.month(dat[1])
        self.day(dat[2])
        self.hour(dat[3])
        self.min(dat[4])
        self.sec(dat[5])

def dec2hex(self, dat):
    return (int(dat/10)<<4) + (dat%10)

def setREG(self, dat, reg):
    buf = bytearray(2)
    buf[0] = reg
    buf[1] = dat
    self.i2c.send(buf, DS3231_ADDR)

def getREG_DEC(self, reg):
    self.i2c.send(reg, DS3231_ADDR)
    t = self.i2c.recv(1, DS3231_ADDR)[0]
    return (t>>4)*10 + (t%16)

def sec(self, sec=''):
    if sec == '':
        return self.getREG_DEC(DS3231_REG_SEC)
    else:
        self.setREG(self.dec2hex(sec), DS3231_REG_SEC)

def min(self, min=''):
    if min == '':
        return self.getREG_DEC(DS3231_REG_MIN)
    else:
        self.setREG(self.dec2hex(min), DS3231_REG_MIN)

def hour(self, hour=''):
```



```
if hour=='':
    return self.getREG_DEC(DS3231_REG_HOUR)
else:
    self.setREG(self.dec2hex(hour), DS3231_REG_HOUR)

def day(self, day=''):
    if day=='':
        return self.getREG_DEC(DS3231_REG_DAY)
    else:
        self.setREG(self.dec2hex(day), DS3231_REG_DAY)

def month(self, month=''):
    if month=='':
        return self.getREG_DEC(DS3231_REG_MONTH)
    else:
        self.setREG(self.dec2hex(month), DS3231_REG_MONTH)

def year(self, year=''):
    if year=='':
        return self.getREG_DEC(DS3231_REG_YEAR)
    else:
        self.setREG(self.dec2hex(year), DS3231_REG_YEAR)

def TEMP(self):
    self.i2c.send(DS3231_REG_TEMP, DS3231_ADDR)
    t1 = self.i2c.recv(1, DS3231_ADDR)[0]
    self.i2c.send(DS3231_REG_TEMP+1, DS3231_ADDR)
    t2 = self.i2c.recv(1, DS3231_ADDR)[0]
    if t1>0x7F:
        return t1 - t2/256 -256
    else:
        return t1 + t2/256
```

字符串转换

- `ubinascii.hexlify(data)`

转换字符串为hex字符串

- `ubinascii.unhexlify(data)`

转换HEX字符串为普通字符串

访问C结构体

这个模块实现MicroPython的“外部数据接口”，它类似与CPython的ctypes，但是API并不相同，为更小代码大小做了优化。

定义结构

结构体通过描述进行定义—将Python字典编码字段名称做为访问相关值的键和其他属性。目前，uctypes要求明确指定每个字段偏移。从结构开始以字节给定偏移量。

下面是各种字段编码示例：

标准类型

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

递归结构体

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

基本数组类型

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

集合数组类型

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

指针类型

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

指针集合类型

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

位定义

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

模块包含内容

`class ctypes.struct(addr, descriptor, layout_type=NATIVE)` 定义结构

`ctypes.LITTLE_ENDIAN`

小端紧凑结构

`ctypes.BIG_ENDIAN`

大端紧凑结构

`ctypes.NATIVE`

自然结构

`ctypes.sizeof(struct)` 结构大小

`ctypes.addressof(obj)` 结构地址

`ctypes.bytes_at(addr, size)` 指定地址和大小的字节对象

`ctypes bytearray_at(addr, size)` 指定地址和大小的字节数组对象

密码是：micropythoN
注意最后一个N是大写的。

IP: 192.168.4.1

ESP8266带有Wifi功能，所以除了通过串口和它通信外，还可以通过Wifi的方式进行访问。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。

```
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）
 - 配置网络，连接到路由器。配置网络只需一次，下一次会自动连接，和计算机上一样

```
import network
```

```
w=network.wlan(network.STA_IF)
```

```
w.scan()
```

```
w.connect(SSID, PASSWR0D)
```

• 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。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，因此需要再次连接。
- 再次连接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。

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
data = Pin(13, Pin.OUT) # set GPIO13 to output to drive the 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, rgbi_buf)
```

webrepl

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

GPIO的用法

定义GPIO

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

方法

- 初始化

`Pin.init(mode, pull=None, *, value)` mode:

- `Pin.IN`, 输入
- `Pin.OUT`, 输出

`pull`:

- `NONE`, 无
- `Pin.PULL_UP`, 上拉

`value`: 输出电平

- 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

- 中断

`Pin.irq(*, trigger, handler=None)` `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

例子

```
CS = Pin(2, Pin.OUT) CS(1)
```

```
CS(0)
```

```
CS.value()
```

```
CS.value(1)
```

```
CS.high()
```

```
CS.low()
```

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

首先要在ubuntu下

在github下载工具链源码

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

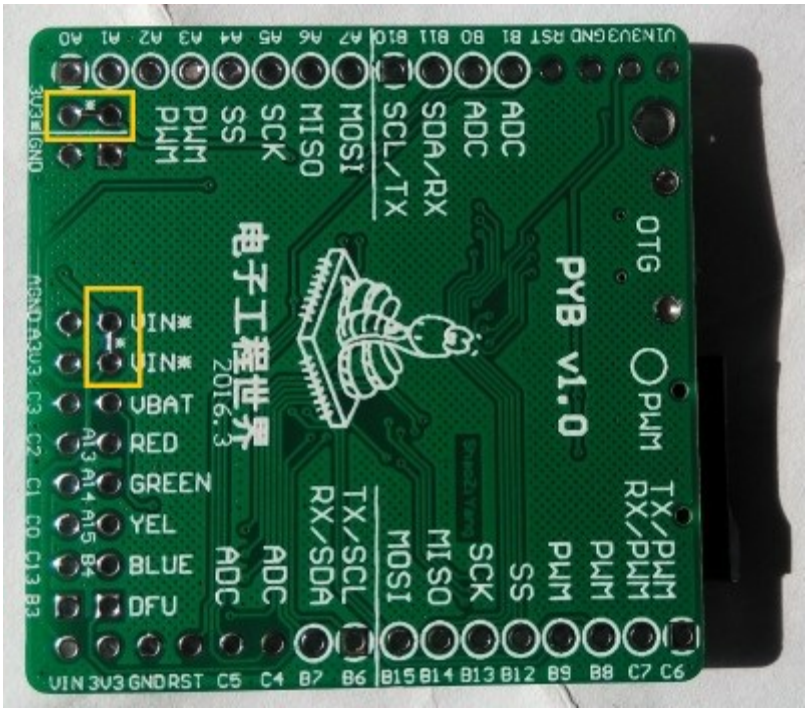
或者直接用git克隆一个

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

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

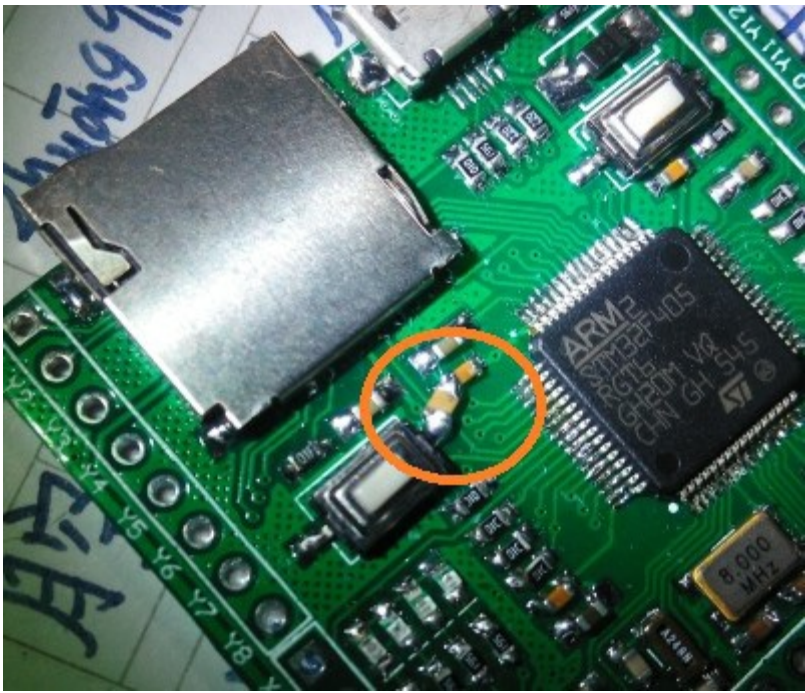
# LED loop test
def LED_loop_test():
    for i in range(1, 5):
        pyb.LED(i).on()
        pyb.delay(100)
        pyb.LED(i).off()
        pyb.delay(100)

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。



这里收集了一些常见问题

- 在修改pybFlash磁盘上的boot.py/main.py后，需要安全退出USB，才能拔下USB线，否则容易造成文件系统损坏。
- 有问题时，先尽量用Ctrl-C中断运行，以及Ctrl-D软复位，而不要按复位键硬复位，因为硬复位会丢失串口连接。

各种相关资源

- 官网: <https://micropython.org/>
- 下载: <https://micropython.org/download/>
- 快速参考: <http://docs.micropython.org/en/latest/pyboard/pyboard/quickref.html>
- github: <https://github.com/micropython/micropython>
- 推荐终端软件:
 - putty: <http://www.chiark.greenend.org.uk/~sgtatham/putty/>
 - kitty: <http://www.9bis.net/kitty/>
 - xshell: http://www.netsarang.com/download/down_xsh.html
 - 超级终端 (WinXP自带)

•