Micro Python Documentation

Выпуск 1.3.6

Damien P. George

окт. 25, 2017

Оглавление

Глава 1

Краткое описание микроконтроллера pyboard

Основные команды микроконтроллера

Подробнее *pyb*.

```
import pyb

pyb.delay(50) # ожидание 50 миллисекунд
pyb.millis() # количество миллисекунд с момента запуска
pyb.repl_uart(pyb.UART(1, 9600)) # duplicate REPL on UART(1)
pyb.wfi() # pause CPU, waiting for interrupt
pyb.freq() # получить частоты центрального процессора и магистрали
pyb.freq(60000000) # установить частоту центрального процессора 60MHz
pyb.stop() # остановка центрального процессора, ожидание внешнего прерывания
```

Светодиоды (LEDs)

Подробнее *pyb.LED*.

```
from pyb import LED

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

Пины и интерфейсы ввода/вывода (GPIO)

Подробнее *pyb.Pin*.

```
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() # получить значение, 0 или 1
```

Внешние прерывания

Подробнее *pyb.ExtInt*.

```
from pyb import Pin, ExtInt

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

Таймеры

Подробнее *pyb.Timer*.

```
from pyb import Timer

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

Широтно-импульсная модуляция (PWM)

Подробнее *pyb.Pin* и *pyb.Timer*.

```
from pyb import Pin, Timer

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

Конвертация аналогового в цифровой (ADC)

Подробнее *pyb.Pin* и *pyb.ADC*.

```
from pyb import Pin, ADC

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

Конвертация цифрового в аналоговый (DAC)

Подробнее *pyb.Pin* и *pyb.DAC*.

```
from pyb import Pin, DAC

dac = DAC(Pin('X5'))
dac.write(120) # вывод от 0 до 255
```

Универсальный асинхронный приёмопередатчик (UART)

Подробнее *pyb.UART*.

```
from pyb import UART

uart = UART(1, 9600)
uart.write('hello')
uart.read(5) # читать 5 байт
```

Интерфейс системного программированния (SPI)

Подробнее *pyb.SPI*.

```
from pyb import SPI

spi = SPI(1, SPI.MASTER, baudrate=200000, polarity=1, phase=0)
spi.send('hello')
spi.recv(5) # получить 5 байт из шины
spi.send_recv('hello') # send a receive 5 bytes
```

Интерфейсная шина IIC (I2C)

Подробнее *pyb.I2C*.

```
from pyb import I2C

i2c = I2C(1, I2C.MASTER, baudrate=100000)
i2c.scan() # возвращает список ведомых адресов
i2c.send('hello', 0x42) # отправить 5 байт для ведомого устройства с адресом 0x42
i2c.recv(5, 0x42) # получить 5 байт от ведомого устройства
i2c.mem_read(2, 0x42, 0x10) # прочитать 2 байта от ведомого устройства 0x42, ведомого устройства␣
↪памяти 0x10
i2c.mem_write('xy', 0x42, 0x10) # написать 2 байта ведомому устройству 0x42, 0x10 ведомого␣
↪устройства памяти
```

Глава 2

Основная информация о микроконтоллере

Внуренняя файловая система и микро SD

У pyboard есть небольшая внутренняя файловая система: `/flash`; она находится во флеш-памяти микроконтроллера. Если карта памяти micro SD вставлена в гнездо, то её файловая система находится в `/sd`.

Перед запуском pyboard, необходимо выбрать загрузочную файловую систему. Если карта памяти micro SD вставлена в гнездо, то загрузка будет происходить из `/sd`. Когда карты памяти нет - загрузка происходит из внутренней памяти `/flash`.

(Стоит отметить, что в старых версиях микроконтроллера, `/flash` называется `0:/`, а `/sd` - `1:/`).

Во время загрузки используется два файла в файловой системе: `boot.py` и `main.py`. Они доступны на компьютере при подключении к нему микроконтроллера.

При подключении микроконтроллера к компьютеру, последний его распознает как USB накопитель. В результате вы можете свободно работать с файлами `boot.py` и `main.py`.

Не забывайте извлекать (размонтировать на Linux) USB накопитель перед перезагрузкой микроконтроллера.

Режимы загрузки

Если включение произошло в штатном режиме или была нажата кнопка перезагрузки микроконтроллера, то pyboard запустится в стандартном режиме: первым выполнится файл `boot.py`, затем сконфигурируется USB и запустится `main.py`

Вы можете изменить последовательность загрузки, удерживая переключатель пользователя когда микроконтроллер будет запускаться. Удерживайте переключатель и нажимите reset, продолжайте удерживать переключатель - светодиоды начнут считать в двоичной системе. Когда светодиоды достигнут значения нужного вам режима - отпустите переключатель. Светодиоды выбранного режима будут быстро мигать и микроконтроллер будет быстро загружаться.

Режимы:

1. Только зелёный светодиод, *стандартная загрузка*: запустить `boot.py` и затем `main.py`.
2. Только оранжевый светодиод, *безопасная загрузка*: не запускать никаких скриптов при загрузке.
3. Зелёный и оранжевый светодиоды одновременно, *сброс файловой системы*: сбросить внутреннюю файловую систему до заводских параметров и затем загрузиться в безопасном режиме.

Если ваша файловая система испортилась - загрузитесь в 3-м режиме чтобы исправить ошибки.

Ошибки: мигание светодиодов

в настоящий момент есть только два типа ошибок, которые вы можете увидеть:

1. Зелёный и красный светодиоды поочерёдно горят: ваш код содержит ошибки (например в `main.py`)
2. **Все четыре светодиода медленно мигают: произошла критическая ошибка.** Не подлежит исправлению, требуется сделать аппаратный сброс.

Глава 3

Руководство Micro Python

Это руководство предназначено для того, чтобы вы начали работать с вашим микроконтроллером. Всё что вам нужно это микроконтроллер и микро-USB кабель чтобы соединить его с вашим компьютером. Для начала рекомендуется следовать инструкции, изложенной ниже.

Введение

Чтобы получить максимальную отдачу от Вашего pyboard, нужно изучить несколько базовых вещей для понимания того, как она работает.

Уход за pyboard

Так как pyboard не имеет защиты - следует соблюдать небольшие правила безопасности:

- Будьте внимательны при подключении / отключении кабеля USB. Несмотря на то, что разъем USB спаян через микроконтроллер и является относительно прочным; если он оторвётся - это будет очень трудно исправить.
- Статическое электричество может повредить компоненты на pyboard. Если вы чувствуете статическое электричество вокруг вас (например, сухой и холодный климат), следует принять дополнительные меры, чтобы не повредить pyboard. Если ваш pyboard пришел в чёрной пластиковой коробке, то используйте её для хранения и транспортировки pyboard так как это антистатическая коробка (она выполнена из проводящего пластика, с проводящим пенопластом внутри).

Пока вы заботитесь об оборудовании, вы можете быть спокойны. Программное обеспечение на pyboard сломать почти невозможно, так что не стесняйтесь играть с написанием кода, так много, как вам нравится. Если файловая система портится - ниже о том, что делать, чтобы сбросить её. В худшем случае вам придется перепрошить программное обеспечение MicroPython, но это может быть сделано через USB.

Макет pyboard

Разъем micro-USB находится в правом верхнем углу, micro-SD слот для карты в левом верхнем углу платы. Есть 4 светодиода между слотом SD и Разъем USB. Цвета: красный внизу, выше зеленый, оранжевый, и синий сверху. Есть 2 кнопки(переключателя): справа сброс(reset) переключатель, слева переключатель пользователя.

Подключение и включение

Pyboard может получать питание через USB. Подключите его к компьютеру через micro-USB кабель. Существует только один способ расположения в гнезде, которому кабель будет соответствовать. После подключения, зеленый светодиод на плате должен быстро мигать.

Питание от внешнего источника

Pyboard может питаться от аккумулятора или от другого внешнего источника питания.

Обязательно подключите положительный провод питания к VIN, и землю к GND. На pyboard нет защиты от неправильной полярности, так что вы должны быть осторожны при подключении чего-либо к VIN.

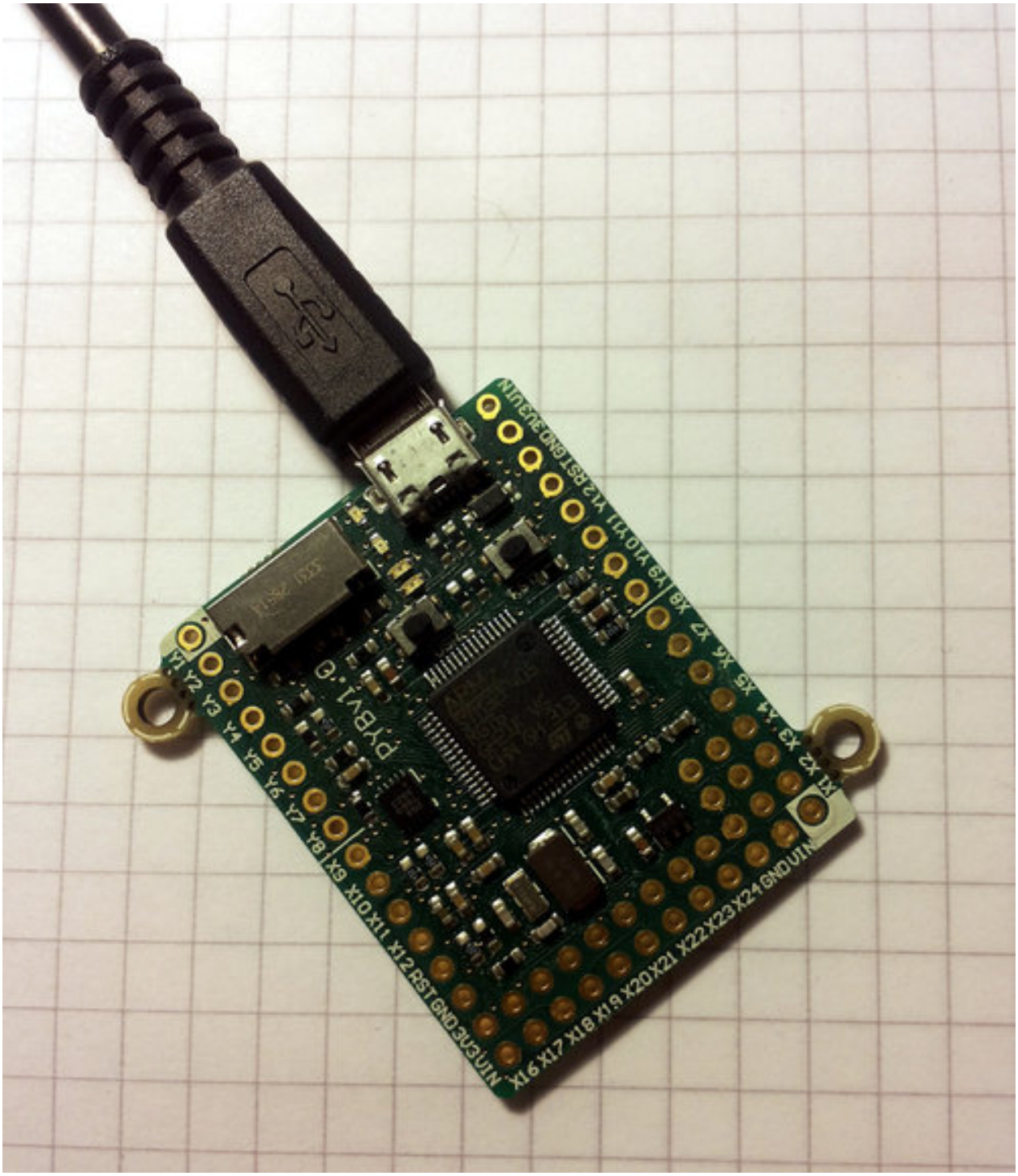
Входное напряжение должно быть в пределах от 3.6V до 10V.

Запуск первого скрипта

Давайте приступим и получим код, который будет работать на pyboard! В конце концов, это то, ради чего всё затевалось!

Подключение pyboard

Подключите pyboard к компьютеру (Windows, Mac или Linux) с помощью микро-USB. Существует только один способ, которым можно вставить кабель в гнездо, так что вы не ошибётесь.



Когда pyboard подключен к компьютеру, он включается и начинает процесс запуска (the boot process). Зеленый светодиод должен загореться на пол секунды или меньше, и, когда он отключается - процесс загрузки завершен.

Открытие pyboard как USB накопителя

Теперь Ваш компьютер должен распознать pyboard. От типа вашего компьютера зависит то, что произойдет далее:

- **Windows**: Ваш pyboard будет отображаться как съемный USB накопитель. Появится всплывающее окно или вам понадобится открыть Проводник.

  Windows может распознать pyboard как последовательное устройство и попытаться автоматически настроить это устройство. Если это произойдёт, то процесс следует отменить. Мы узнаем как работает последовательное устройство в следующем уроке.

- **Mac**: Ваш pyboard появится на рабочем столе в качестве съемного диска. И, вероятно, будет называться "NONAME". Нажмите на него, чтобы открыть папку pyboard.

- **Linux**: Ваш pyboard будет отображаться как съемный носитель. В Ubuntu он подключится автоматически. В других дистрибутивах Linux pyboard может установится автоматически, либо вам потребуется сделать это вручную. В командной строке терминала введите `lsblk`, чтобы увидеть список подключенных дисков, а затем `mount/dev/sdb1` (замените `sdb1` на соответствующее устройство). Вы, возможно, должны сделать это с правами суперпользователя (root).

Итак, теперь у вас есть pyboard, подключённый как USB накопитель и окно (или командную строку) со списком файлов на диске микроконтроллера.

Диск, который вы видете, у pyboard называется `/flash` и должен содержать следующие 4 файла:

- boot.py – этот скрипт выполняется, когда pyboard загружается. Он устанавливает различные варианты конфигурации для pyboard.
- main.py – это основной скрипт, в котором будет находиться ваша программа. Он выполняется после `boot.py`.
- README.txt – содержит некоторую основную информацию.
- pybcdc.inf – файл драйвера для Windows, чтобы настроить последовательное USB устройство. Подробнее об этом в следующем уроке.

Резактирование `main.py`

Теперь мы собираемся написать нашу программу на Python, так что открываем `main.py` в текстовом редакторе. В Windows вы можете использовать блокнот или любой другой редактор. На Mac и Linux, используйте ваш любимый текстовый редактор. Когда откроете файл - увидите, что содержит одну строку:

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

Эта строка начинается с символа #, который означает, что это * комментарий *. Такие строки не будут ничего делать, они нужны, чтобы написать заметки в вашей программе.

Давайте добавим две строки:

```
# main.py -- put your code here!
import pyb
pyb.LED(4).on()
```

Первая строка, которую мы написали, говорит, что мы хотим использовать `pyb` модуль. Этот модуль содержит все функции и классы для управления pyboard.

Вторая строка включает синий индикатор: сначала получаем `LED` класс из модуля `pyb`, задаём светодиодный номер 4 (синий светодиод), а затем включаем его.

Сброс pyboard

Для запуска этого небольшого скрипта, вы должны сначала сохранить и закрыть `main.py`, а затем извлечь (или отключить/размонтировать) pyboard USB диск. Сделайте это, так как бы вы извлекали обычный USB флэш-диск.

Когда диск безопасно отключился - вы можете добраться до забавной части: нажмите переключатель RST на pyboard для сброса и запуска вашего скрипта. Переключатель RST это маленькая чёрная кнопка, чуть ниже разъема USB на плате по правому краю.

When the drive is safely ejected/unmounted you can get to the fun part: press the RST switch on the pyboard to reset and run your script. The RST switch is the small black button just below the USB connector on the board, on the right edge.

После того как вы нажмёте RST - зеленый светодиод будет быстро мигать, а затем синий светодиод должен загореться и остаться.

Поздравляем! Вы написали и запустили свою первую программу на Micro Python!

Простая интерактивная среда программирования REPL

REPL - Read(Чтение) Evaluate(Оценка) Print(Печать) Loop(Цикл) - интерактивная строка Micro Python, с помощью которой вы можете получить доступ к pyboard. Использование REPL является самым простым способом проверки своего кода и выполнения команд. Вы можете использовать REPL как вспомогательное средство для написания сценариев в `main.py`.

Для использования REPL, необходимо подключиться к последовательному устройству USB на pyboard. Как это сделать, зависит от вашей операционной системы.

Windows

Вы должны установить драйвер pyboard с использованием последовательного USB устройства. Драйвер USB флэш-диска pyboard называется `pybcdc.inf`.

Для установки этого драйвера вам необходимо перейти в Диспетчер Устройств для вашего компьютера, найти pyboard в списке устройств (он должн иметь предупреждающий знак, потому что он еще не работает), щелкните правой кнопкой мыши на pyboard устройство, выберите Свойства и установите драйвер. Затем необходимо выбрать опцию, чтобы найти драйвер потоков вручную (не используйте автоматическое обновление Windows), перейдите к USB флеш-диску pyboard, и выберите его. Он должен установиться. После установки, вернитесь в Диспетчер Устройств, чтобы найти установленный pyboard, и посмотрите, какой это COM порт (например, COM4).

Теперь вам нужно запустить свою терминальную программу. Вы можете использовать HyperTerminal, если он у вас установлен, или скачать бесплатную программу PuTTY: putty.exe. Используя программу удалённого доступа (возможно терминальную), вы должны подключиться к тому COM порту, который нашли в предыдущем шаге. Для PuTTY: нажмите “Session” на левой панели и нажмите радио-кнопку “Serial” справа, затем в поле “Serial Line” введите найденный COM порт (например, COM4), нажмите кнопку “Open”.

Mac OS X

Откройте терминал и запустите:

```
screen /dev/tty.usbmodem*
```

Когда вы закончите и захотите выйти, нажмите CTRL-A CTRL-\.

Linux

Откройте терминал и запустите:

```
screen /dev/ttyACM0
```

Вы можете попробовать вместо этого `picocom` или `ttyACM`. Возможно вам придётся использовать `/dev/ttyACM1` или с большим номером `ttyACM`. Также вам могут понадобиться права для доступа к этому устройству (например, группа `uucp` или `dialog` или использовать sudo).

Использование REPL

Теперь давайте попробуем запустить код Micro Pythom непосредственно на pyboard.

В вашей программе удалённого доступа (PuTTY, screen, picocom, и т.д.) вы увидите пустой экран с мигающим курсором. Чтобы попасть в простой интерпретатор Micro Python - нажмите Enter. Вы увидите строку, начинающуюся с `>>>`. Давайте удостоверимся, что он работает:

```
>>> print("hello pyboard!")
hello pyboard!
```

Символы `>>>` указывают, что далее вы должны ввести текст в командной строке. После ввода `print("hello pyboard!")` и нажатия Enter - в выводе мы увидили `hello, pyboard!`

Если вы уже знакомы с python - попробуйте выполнить основные команды.

Если какая-либо из них не сработает - попробуйте hard reset or a soft reset; смотри ниже.

Пойдём дальше и попробуем ввести некоторые другие команды. Например:

```
>>> pyb.LED(1).on()
>>> pyb.LED(2).on()
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypy'
```

Сброс микроконтроллера

Если что-то пойдёт не так - вы можете сбросить микроконтроллер двумя способами. Первый - нажать CTRL-D в командной строке Micro Python, произойдёт мягкая перезагрузка (soft reset). Вы увидите сообщение наподобие следующего:

```
>>>
PYB: sync filesystems
PYB: soft reboot
Micro Python v1.0 on 2014-05-03; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>>
```

Если же это не сработает - вы можете выполнить аппаратный сброс (hard reset) (turn-it-off-and-on-again). Для этого нажмите RST на микроконтроллере (маленькая черная кнопка ближе к разъему micro-USB на плате). Это остановит сеанс, отсоединится от любой программы (PuTTY, screen, и т.д.), которая используется для подключения к pyboard.

Перед аппаратным сбросом рекомендуется предварительно отключить программу удалённого доступа и извлечь/размонтировать pyboard.

Включение светодиодов и основные понятия Python

На pyboard проще всего сделать включение светодиодов, подключённых к плате. Соедините плату с компьютером и войдите как описано в руководстве 1. Начнём с переключения светодиода. Введите в интерпретаторе:

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

Эти команды включат и выключат светодиод.

Это все очень хорошо, но мы бы хотели автоматизировать этот процесс. Откройте файл MAIN.PY на pyboard в вашем любимом текстовом редакторе. Напишите или вставьте следующие строки в файл. Если вы новичок в Python, то убедитесь, что вы сделали корректный отступ, это имеет значение!

```
led = pyb.LED(2)
while True:
    led.toggle()
    pyb.delay(1000)
```

Когда вы сохраните изменения, красный светодиод должен на секунду загореться. Чтобы запустить скрипт, сделать мягкую перезагрузку (CTRL-D). Pyboard перезапустится, и вы увидите как зеленый светодиод непрерывно мигает: включается и выключается. Успех! Первый шаг на пути к созданию армии злых роботов! Когда вам наскучит назойливый мигающий зелёный светодиод - нажмите CTRL-C в вашем терминале чтобы остановить его работу.

Итак, что же делает этот код? Для начала нам нужна терминология. Python объектно-ориентированный язык, почти всё в Python есть *class* и когда вы создаёте экземпляр класса - вы получаете *object*. У классов есть *methods*, связанные с ними. Метод используется для взаимодействия с объектами.

В первой строке кода создаётся LED объект, который мы называли led. При создании, этот объект принемает один параметр - число в диапазоне от 1 до 4, соответствующее четырём светодиодам на плате. Класс pyb.LED имеет три важных метода: on(), off() и toggle(). Другой метод, который мы используем: pyb.delay() - он просто ожидает в течение заданного в миллисекундах времени. Создаётся LED объект и затем, в бесконечном цикле (while True), происходит переключение (toggle()) светодиода (то есть on, off) и ожидание в 1 секунду.

Упражнение: попробуйте изменить время между переключением светодиода и изменить сам светодиод.

Упражнение: подключитесь к pyboard, создайте объект pyb.LED и включите его, используя метод on().

Дискотека на вашем pyboard

До сих пор мы использовали только один светодиод, но на pyboard их четыре. Давайте создадим объекты для каждого светодиода чтобы контролировать их. Сделаем это, создав список (массив) объектов pyb.LED:

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

Если вы попробуете вызвать pyb.LED() с числом за пределами данного (1..4) диапазона - поглучите сообщение об ошибке. Далее давайте создадим бесконечный цикл, который будет включать и выключать все светодиоды:

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

Здесь n содержит текущий светодиод и каждую следующую итерацию цикла переходим к следующему n (символ % является бинарной операцией: получение модуля числа; то есть n находится в диапазоне от 0 до 3 включительно). Получаем доступ к n-ому светодиоду и переключаем его. Когда вы запустите этот скрипт - увидите как поочерёдно включаются и выключаются светодиоды.

Вы можете заметить одну проблему: если остановить скрипт и запустить снова - светодиоды застрянут на предыдущей сессии, разрушая нашу дискотеку. Мы можем исправить это, выключив все светодиоды при инициализации с помощью блока try/finally.

One problem you might find is that if you stop the script and then start it again that the LEDs are stuck on from the previous run, ruining our carefully choreographed disco. We can fix this by turning all the LEDs off when we initialise the script and then using a try/finally block. При нажатии CTRL-C, Micro Python генерирует VCPInterrupt исключение. Исключение обычно означают что что-то пошло не так и вы можете использовать try: команду, которая “ловит” исключения. В данном случае это всего лишь пользовательское прерывание скрипта. Нам не нужно ловить ошибку - просто скажем Micro Python что он должен делать когда мы останавливаем скрипт. Используем это чтобы убедиться что все светодиоды выключены. Код:

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

n = 0
try:
   while True:
      n = (n + 1) % 4
      leds[n].toggle()
      pyb.delay(50)
finally:
    for l in leds:
        l.off()
```

Четвёртый специальный светодиод

Синий светодиод особенный - вы можете контролировать интенсивность его свечения. Это делается с помощью специального метода intensity(). Интенсивность это число от 0 до 255, которое определяет на сколько ярко светится синий светодиод. Следующий код постепенно повышает яркость и затем выключает этот светодиод:

```
led = pyb.LED(4)
intensity = 0
while True:
    intensity = (intensity + 1) % 255
    led.intensity(intensity)
    pyb.delay(20)
```

Вы можете вызвать intensity() и для других светодиодов, но они могут быть только включены или выключены. 0 их выключает, а любое другое число до 255 - включает.

Переключатель, обратные вызовы и прерывания

На pyboard есть две небольшие кнопки, называющиеся USR и RST.

RST переключатель делает аппаратную перезагрузку (hard-reset) и, если вы на него - pyboard перезагрузится; аналогично выключению и включению микроконтроллера.

USR переключатель для общего пользования и управляется с помощью объекта Switch. Создание объекта переключателя:

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

Помните, вам скорее всего потребуется подключить pyb: `import pyb`, если вы получите ошибку “the name `pyb` does not exist”.

С помощью переключателя вы можете получить свой статус:

```
>>> sw()
False
```

Будет получено `False` если переключатель не нажат, или `True` если он нажат. Попробуйте удерживать USR пока выполняется вышеуказанная команда.

Переключение функции обратного вызова (callback)

Переключатель очень простой объект, но у него одно расширение функционала: функция `sw.callback()`. Функция обратного вызова устанавливает алгоритм действий когда кнопка нажата и использует прерывания. Как работают прерывания, наверное, лучше всего понять на примере. Попробуйте запустить следующий код:

```
>>> sw.callback(lambda:print('press!'))
```

Каждый раз при нажатии на USR выводится `press!` Пойдём далее и нажмём USR переключатель - смотрите что выводится на экран. Обратите внимание что этот вывод прерывает всё, что вы вводите: пример асинхронных прерываний.

В качестве другого примера, попробуйте:

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

Кнопка USR будет включать и выключать красный светодиод. И это будет работать даже вовремя выполнения другого кода.

Чтобы отключить коллбэк - передайте в него `None`:

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

Вы можете передать в коллбэк любую функцию, которая не содержит аргументов. Выше мы использовали `lambda` - особенная функция для создания анонимных функций налету. Вместо неё мы могли бы сделать:

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

Здесь создаётся функция `f` и передаётся коллбэку переключателя USR. Такой способ имеет смысл, если ваша функция более сложная, чем позволяет сделать `lambda`.

Обратите внимание, что ваши функции обратного вызова не должны выделять никакую память (например они не могут создать кортеж или список). Функции обратного вызова должны быть относительно простыми. Если вам нужно создать список - сделайте это заранее и сохраните его в глобальной переменной (или создайте его внутри коллбэка и в нём же обязательно удалите). Если вам нужны долгие сложные расчёты - используйте коллбэк-функцию чтобы установить флаг (ссылку) на код, где будут реализованы требуемые расчёты.

Технические детали прерываний

Давайте рассмотрим подробнее что происходит с переключателем обратного вызова. Когда вы передаёте функцию в `sw.callback()` - переключатель устанавливает внешний триггер (falling edge) прерывания на пин (pin), который соединён с кнопкой USR. Это означает, что микроконтроллер слушает этот пин и отлавливает любые изменения. Происходит следующее:

1. При нажатии на кнопку происходит изменение на контакте (пине): замыкание; и микроконтроллер регистрирует это.
2. Микроконтроллер завершает выполнение текущей машинной инструкции, останавливает выполнение и сохраняет текущее состояние (записывает регистры в стек). Это прерывает выполнение любого скрипта на плате.
3. Микроконтроллер начинает выполнять специальный обработчик прерывания, связанного с внешним триггером переключателя USR. Этот обработчик прерываний получает функцию, которую вы передали в `sw.callback()`, и выполняет её.
4. Функция обратного вызова выполняется пока не закончит; затем вернёт управление обработчику прерывания.
5. Обработчик прерывания сообщает микроконтроллеру, что прерывание завершилось.
6. Микроконтроллер восстанавливает состояние, сохранённое на шаге 2.
7. Выполнение программы продолжается с того места, на котором остановилось. Пауза на выполнении кода никак не скажется.

Данная последовательность несколько усложняется когда одновременно происходит несколько прерываний. В этом случае прерывание с наибольшим приоритетом выполняется раньше и так далее по приоритету в порядке очереди. Приоритет прерывания переключателя USR является самым низким.

Акселерометр

Здесь вы изучите как использовать акселерометр и включать светодиоды относительно наклона платы.

Использование акселерометра

На puboard есть акселерометр (крошечная гирька на крошечной пружине) который можно использовать для того, чтобы обнаружить угол и движение платы. Есть и другой датчик для каждого измерения x, y, z. Для получения значения акселерометра, создайте объект pyb.Accel() и вызовите x() метод.

```
>>> accel = pyb.Accel()
>>> accel.x()
7
```

Этот метод возвращает целое число от -30 до 30. Обратите внимание, что результат измерений очень зашумлён; так что даже если вы будите держать плату идеально ровно - будут неболшьшие изменения в значениях, полученных от измерительных функций. Поэтому вы не должны использовать точные значения функции x() - работать следует с диапазоном значений.

Давайте напишем код, который будет включать светодиод тогда, когда плата расположена не горизонтально относительно одной из осей.

```
accel = pyb.Accel()
light = pyb.LED(3)
SENSITIVITY = 3

while True:
    x = accel.x()
    if abs(x) > SENSITIVITY:
        light.on()
    else:
        light.off()

    pyb.delay(100)
```

Мы создали Accel и LED объекты, затем получили значение акселерометра оси x. Если абсолютная величина значения x больше чем определённая константа `SENSITIVITY`, то светодиод включается, в противном случае - выключается. Цикл имеет небольшую задержку `pyb.delay()`: если бы его не было, то светодиод бы очень чато мигал при значениях x близких к `SENSITIVITY`. Попробуйте запустить этот код на вашем микроконтроллере и покрутите его влево/вправо, чтобы активировать светодиод.

Упражнение: измените предыдущий скрипт так, чтобы синий светодиод становился тем ярче чем больший угол наклона будет у платы. Совет: Вам нужно будет масштабировать значения, шнтенсивность светодиода 0-255.

Создаём уровень

Пример выше бал чувствителен только лишь к наклонам относительно оси x, но если мы используем значения `y()` и больше светодиодов - то можем превратить pyboard в уровень.

```
xlights = (pyb.LED(2), pyb.LED(3))
ylights = (pyb.LED(1), pyb.LED(4))

accel = pyb.Accel()
SENSITIVITY = 3

while True:
    x = accel.x()
    if x > SENSITIVITY:
        xlights[0].on()
        xlights[1].off()
```

```
    elif x < -SENSITIVITY:
        xlights[1].on()
        xlights[0].off()
    else:
        xlights[0].off()
        xlights[1].off()

    y = accel.y()
    if y > SENSITIVITY:
        ylights[0].on()
        ylights[1].off()
    elif y < -SENSITIVITY:
        ylights[1].on()
        ylights[0].off()
    else:
        ylights[0].off()
        ylights[1].off()

    pyb.delay(100)
```

Мы начинаем с создания кортежа LED объектов для x, y. Кортежи - неизменяемые объекты в python, это означает что их нельзя изменить при создании. Затем мы делаем то же, что и раньше, но включаем разные светодиоды для положительного и отрицательного наклонов относительно оси x. Далее делаем то же самое для оси y. Не особенно изощрённый метод, но он работает. Запустите это на вашем pyboard и вы увидите как включаются различные светодиоды относительно того как наклонена плата.

Безопасный режим и возврат к заводским настройкам

Если что-то пошло не так с вашим pyboard - не паникуйте! Сломать pyboard с помощью программирования практически невозможно.

В первую очередь попробуйте войти в безопасный режим: при этом временно пропускается выполнение `boot.py` и `main.py`, используя базовые настройки USB.

Если у вас возникли проблемы с файловой системой - вы можете сделать возврат к заводским настройкам, который восстановит файловую системы в исходное состояние.

Безопасный режим

Чтобы войти в безопасный режим, выполните следующее:

1. Подключите pyboard по USB.
2. Нажмите и удерживайте кнопку USR.
3. Удерживая USR, нажмите и отпустите RST.
4. Светодиоды начнут поочереди переключаться между режимами: зелёный, оранжевый, зелёный+оранжевый.
5. В тот момент когда будет гореть *только оранжевый светодиод* - отпустите USR.
6. Оранжевый светодиод должен быстро помигать 4 раза и выключиться.
7. Теперь вы в безопасном режиме.

В безопасном режиме файлы `boot.py` и `main.py` не выполняются и, значит, pyboard загружается с настройками по умолчанию. Это означает, что вы теперь имеете доступ к файловой системе (USB диск должен появиться) и можете редактировать `boot.py` и `main.py` чтобы устранить любые неполадки.

Вход в безопасный режим является временным и не влечёт никаких изменений в файлах на pyboard.

Возврат файловой системы к заводским настройкам

Если ваша файловая система будет повреждена (к примеру вы забудите извречь/размонтировать микроконтроллер) или у вас есть код в `boot.py` или `main.py`, который вы не можете отключить - вы всегда можете сбросить файловую систему.

Сброс файловой системы удаляет все файлы во внутренней памяти pyboard (не SD карты), а также восстанавливает файлы `boot.py`, `main.py`, `README.txt` и `pybcdc.inf` обратно к их базовому состоянию.

Чтобы сделать возврат к заводским настройкам, вы должны повторить процедуру входа в безопасный режим, но отпустить кнопку USR в тот момент когда будут гореть зелёный+оранжевый:

1. Подключите pyboard по USB.
2. Нажмите и удерживайте кнопку USR.
3. Удерживая USR, нажмите и отпустите RST.
4. Светодиоды начнут поочереди переключаться между режимами: зелёный, оранжевый, зелёный+оранжевый.
5. В тот момент когда будут гореть *оба светодиода (зелёный и оранжевый)* - отпустите USR.
6. Зелёный и оранжевый светодиоды должны быстро помигать 4 раза.
7. Красный светодиод включится (теперь горят 3 светодиода: красный, зелёный и оранжевый).
8. Сейчас pyboard сбрасывает файловую систему (это займёт несколько секунд).
9. Все светодиоды выключатся.
10. Теперь ваша файловая система в её первозданном виде.
11. Нажмите и отпустите RST для запуска.

Делаем из pyboard USB мышь

Pyboard это USB устройство, которое можно превратить в обычную USB мышь.

Для того чтобы сделать это, мы должны изменить конфигурацию USB в файле `boot.py`. Он будет выглядеть так

```
# boot.py -- run on boot-up
# can run arbitrary Python, but best to keep it minimal

import pyb
#pyb.main('main.py') # main script to run after this one
#pyb.usb_mode('CDC+MSC') # act as a serial and a storage device
#pyb.usb_mode('CDC+HID') # act as a serial device and a mouse
```

Чтобы включить режим мыши - раскомментируйте последнюю строку, сделав её похожей на

```
pyb.usb_mode('CDC+HID') # act as a serial device and a mouse
```

Если вы уже изменили ваш файл `boot.py`, то остаётся написать совсем немного кода:

```
import pyb
pyb.usb_mode('CDC+HID')
```

Это настроит pyboard как CDC (последовательный/serial) и HID (устройство для взаимодействия с человеком, в нашем случае - мышь) USB устройство когда он загрузится.

Извлеките/размонтируйте pyboard и перезапустите, используя кнопку RST. Теперь ваш компьютер должен распознать pyboard как мышь!

Отправка событий ру-мыши компьютеру

Чтобы получить ру-мышь - мы должны отправлять события ру-мыши компьютеру. Первым делом используем REPL. Подключитесь к вашему pyboard, используя программу удалённого доступа и введите следующее

```
>>> pyb.hid((0, 10, 0, 0))
```

Ваш курсор должен передвинуться на 10 пикселей вправо! В команде выше вы отправили 4 единицы информации: о состоянии кнопки, x, y, и скроллинг. В данном случае число 10 означает что положение курсора должно измениться на десять пикселей в положительном направлении относительно оси x.

Давайте сделаем так, чтобы курсор вибрировал (влево и право):

```
>>> import math
>>> def osc(n, d):
...   for i in range(n):
...     pyb.hid((0, int(20 * math.sin(i / 10)), 0, 0))
...     pyb.delay(d)
...
>>> osc(100, 50)
```

Первый аргумент функции `osc` - количество событий, которые будут отправлены компьютеру. Второй - задержка (в миллисекундах) между событиями. Поэкспериментируйте с различными значениями.

Упражнение: сделайте так, чтобы курсор двигался по кругу.

Мышка из акселерометра

Тепреь давайте сделаем из pyboard настоящую мышку при помощи акселерометра. Следующий код можно ввести непосредственно в интерактивной строке REPL или написать в файл `main.py`. В данном случае мы расположим код в `main.py` так как, сделав это, мы научимся работать в безопасном режиме.

На данный момент pyboard выступает в роли последовательного USB устройства и HID (мышь). Поэтому мы не имеем доступа к файловой системе и не можем редактировать файл `main.py`.

Так же и файл `boot.py` мы не можем отредактировать чтобы выйти из HID-режима и вернуться к нормальному режиму USB устройства...

Чтобы выйти из этой ситуации нам необходимо перейти в *безопасный режим*. Это было описано в [безопасный режим и возврат к заводским настройкам](tut-reset), но мы повторим инструкции:

1. Подключите pyboard по USB.
2. Нажмите и удерживайте кнопку USR.
3. Удерживая USR, нажмите и отпустите RST.

4. Светодиоды начнут поочереди переключаться между режимами: зелёный, оранжевый, зелёный+оранжевый.
5. В тот момент когда будет гореть *только оранжевый светодиод* - отпустите USR.
6. Оранжевый светодиод должен быстро помигать 4 раза и выключиться.
7. Теперь вы в безопасном режиме.

В безопасном режиме файлы `boot.py` и `main.py``не выполняются и pyboard загружается с настройками по умолчанию. Это означает, что теперь у нас есть доступ к файловой системе (должен появиться USB диск) и мы можем редактировать ``main.py`. (Оставим `boot.py` без изменений так как, после внесения изменений в `main.py`, мы по-прежнему хотим использовать HID-режим.)

В `main.py` напишем следующий код:

```
import pyb

switch = pyb.Switch()
accel = pyb.Accel()

while not switch():
    pyb.hid((0, accel.x(), accel.y(), 0))
    pyb.delay(20)
```

Сохраним наш файл, извлечём/размонтируем pyboard и перезагрузим его, используя кнопку RST. Теперь pyboard будет работать как мышь: Любое отклонение платы от горизонтального положения передвигает курсор мыши.

Нажмите USR чтобы прекратить работу программы.

Вы заметите, что ось y инвертирована. Это легко исправить: влего лишь поставим минус перед y-координатами в `pyb.hid()`.

Восстановление pyboard к нормальному состоянию

Если мы оставим всё как есть, то каждый раз при подключении, pyboard будет вести себя как мышь. Возможно вы не хотите этого. Чтобы вернуться нормальное состояние, нужно первым делом войти в безопасный режим (смотри выше) и отредактировать файл `boot.py`. В `boot.py` нужно закомментировать (поставить в начало строки символ #) строку с `CDC+HID` настройками. Должно получиться следующее:

```
#pyb.usb_mode('CDC+HID') # act as a serial device and a mouse
```

Сохраним наш файл, извлечём/размонтируем pyboard и перезапустим его. Мы вернулись к нормальному режиму.

Таймеры

У pyboard есть 14 таймеров, каждый из которых является независимым счётчиком, работающим на определённой пользователем частоте. Их можно использовать для запуска функций через определённые промежутки времени. Все 14 таймеров пронумерованы от 1 до 14, но 3-й зарезервирован внутренних нужд, также как 5-й и 6-й используются для сервоприводов и ADC/DAC контроллеров. Старайтесь не использовать эти зарезервированные таймеры.

Давайте создадим таймер:

```
>>> tim = pyb.Timer(4)
```

Теперь давайте посмотрим что мы только что создали:

```
>>> tim
Timer(4)
```

Pyboard говорит нам, что `tim` прикреплён к таймеру номер 4, но он ещё не инициализирован. Итак, давайте инициализируем его с частотой 10 Гц (это 10 раз в секунду):

```
>>> tim.init(freq=10)
```

Теперь таймер инициализирован и мы можем информацию о нём:

```
>>> tim
Timer(4, prescaler=255, period=32811, mode=0, div=0)
```

Из этого следует, что этот таймер настроен на цикличную работу с тактовой частотой в 255 и он будет считать до 32811, после чего он вызывает прерывание, и начинает отчёт снова с 0. Этот набор чисел задаёт частоту вызова прерывания в 10 Гц.

Счётчик таймера

Так что мы можем сделать с нашим таймером? Самое простое, что можно получить - текущее значение счётчика:

```
>>> tim.counter()
21504
```

Этот счётчик будет непрерывно меняться и подсчитывать.

Таймер с функцией обратного вызова

Следующее что мы можем сделать - это создать функцию обратного вызова для таймера чтобы выполнить её при срабатывании (смотри [switch tutorial](tut-switch) введение в функции обратного вызова):

```
>>> tim.callback(lambda t:pyb.LED(1).toggle())
```

Красный светодиод немедленно начнёт мигать с частотой 5 Гц (два переключения необходимы для одного такта "горения" светодиода, так что переключение с частотой 10 Гц означает, что гореть светодиод будет с частотой 5 Гц). Мы можем изменить частоту, повторно инициализировав таймер:

```
>>> tim.init(freq=20)
```

Мы можем отключить функцию обратного вызова, передав ей `None`:

```
>>> tim.callback(None)
```

Функция, которую мы передаём в коллбэк-функцию должна принемать 1 аргумент - таймер. Это позволяет нам управлять таймером внутри функции обратного вызова.

Мы можем создать 2 таймера и запустить их независимо друг от друга:

```
>>> tim4 = pyb.Timer(4, freq=10)
>>> tim7 = pyb.Timer(7, freq=20)
>>> tim4.callback(lambda t: pyb.LED(1).toggle())
>>> tim7.callback(lambda t: pyb.LED(2).toggle())
```

Так как коллбэк-функции это настоящие аппаратные прерывания - мы можем продолжать использовать pyboard на своё усмотрение при работающих таймерах.

Создание счётчика микросекунд

Мы можем использовать таймер для создания микросекуного счётчика - это может быть полезно когда вы делаете что-то, что требует точной синхронизации. Для этого мы будем использовать 2 таймера: таймер №2 имеет 32-х разрядный счётчик (так же как и таймер №5, но если мы используем его - не сможем использовать сервомотор).

Мы создадим таймер №2 следующим образом:

```
>>> micros = pyb.Timer(2, prescaler=83, period=0x3fffffff)
```

Заданная тактовая частота равна (prescaler) 83, это делает из таймера счётчик с частотой 1 МГц. Это происходит потому что тактовая частота процессора, работающего на частоте 168 МГц, делится на 2 и тогда к заданной частоте прибавляется единица (prescaler+1); всё это даём нам частоту таймера №2: 168МГц/2/(83 + 1) = 1МГц. Период равен огромному числу, так что таймер не скоро вернётся к нулевому значению. В данном случае таймер вернётся к нулю примерно через 17 минут.

Перед тем как использовать этот таймер, его лучше всего сбросить в ноль:

```
>>> micros.counter(0)
```

и затем выполнить свой код:

```
>>> start_micros = micros.counter()

... код ...

>>> end_micros = micros.counter()
```

Ассемблерная вставка

Здесь вы узнаете как сделать ассемблерную вставку в код Micro Python.

Примечание: эта часть учебника для опытных программистов, для тех, кто уже немного знаком с микроконтроллерами и ассемблером.

С помощью ассемблерных вставок вы можете писать процедуры ассемблера и вызывать их как обычные функции в Python.

Возвращение значения

Ассемблерные вставки обозначаются специальным символом. Начнём с простого примера:

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

Вы можете написать этот скрипт в REPL. Эта функция не принемает никаких аргументов и возвращает число 42. `r0` - это регистр, значение этого регистра вернётся вместе с 42. Micro Python всегда распознаёт `r0` как integer и конвертирует его в объект для получателя.

Если мы запустим `print(fun())` - то увидим как на экране появится 42.

Доступ к периферийным устройствам

Для того, чтобы немного усложнить задачу, давайте включим светодиод:

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

Здесь есть несколько новых элементов:

- `stm` - это модуль, который предоставляет множество констант для простого доступа к регистрам микроконтроллера. Попытайтесь импортировать stm (`import stm`) и затем вызовите `help(stm)` в REPL. Должен появиться список всевозможных конастант.
- `stm.GPIOA` - это адресс в памяти периферийного устройства GPIOA. На pyboard красный светодиод на порт A, пин PA13.
- `movwt` перемещает 32-разрядное число в регистр. Это удобная функция, которая превращается в две отличные инструкции: `movw` следует за `movt`. Также, `movt` немедленно сдвигает значение вправо на 16 бит.
- `strh` хранит полу-слова (16 бит). Инструкция выше сохраняет нижние 16 бит из `r1` в ячейку памяти `r0 + stm.GPIO_BSRRL`. This has the effect of setting high all those pins on port A for which the corresponding bit in `r0` is set. In our example above, the 13th bit in `r0` is set, so PA13 is pulled high. Это включает красный светодиод.

Получение аргументов

Встроенные ассемблерные функции могут принемать до 3-х аргументов. Если они используются, то должны быть названы `r0`, `r1` и `r2` для отображения регистров и вызова соглашения(???).

Ниже функция, которая добавляет эти аргументы:

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

Эта функция выполняет вычисление `r0 = r0 + r1`. Результат помещается в `r0` и возвращается. Попробуйте `asm_add(1,2)` - это должно вернуть 3ю

Циклы

Мы можем назначать метки `label(my_label)` и ветвить их, используя, `b(my_label)` или условно ветвить как `bgt(my_label)`.

В следующем примере мигает залёный светодиод. Он мигает `r0` раз.

```
@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)
```

Управление питанием

pyb.wfi() используется для снижения энергопотребления при ожидании таких событий как прерывание. Мы можем использовать его в следующих ситуациях:

```
while True:
    do_some_processing()
    pyb.wfi()
```

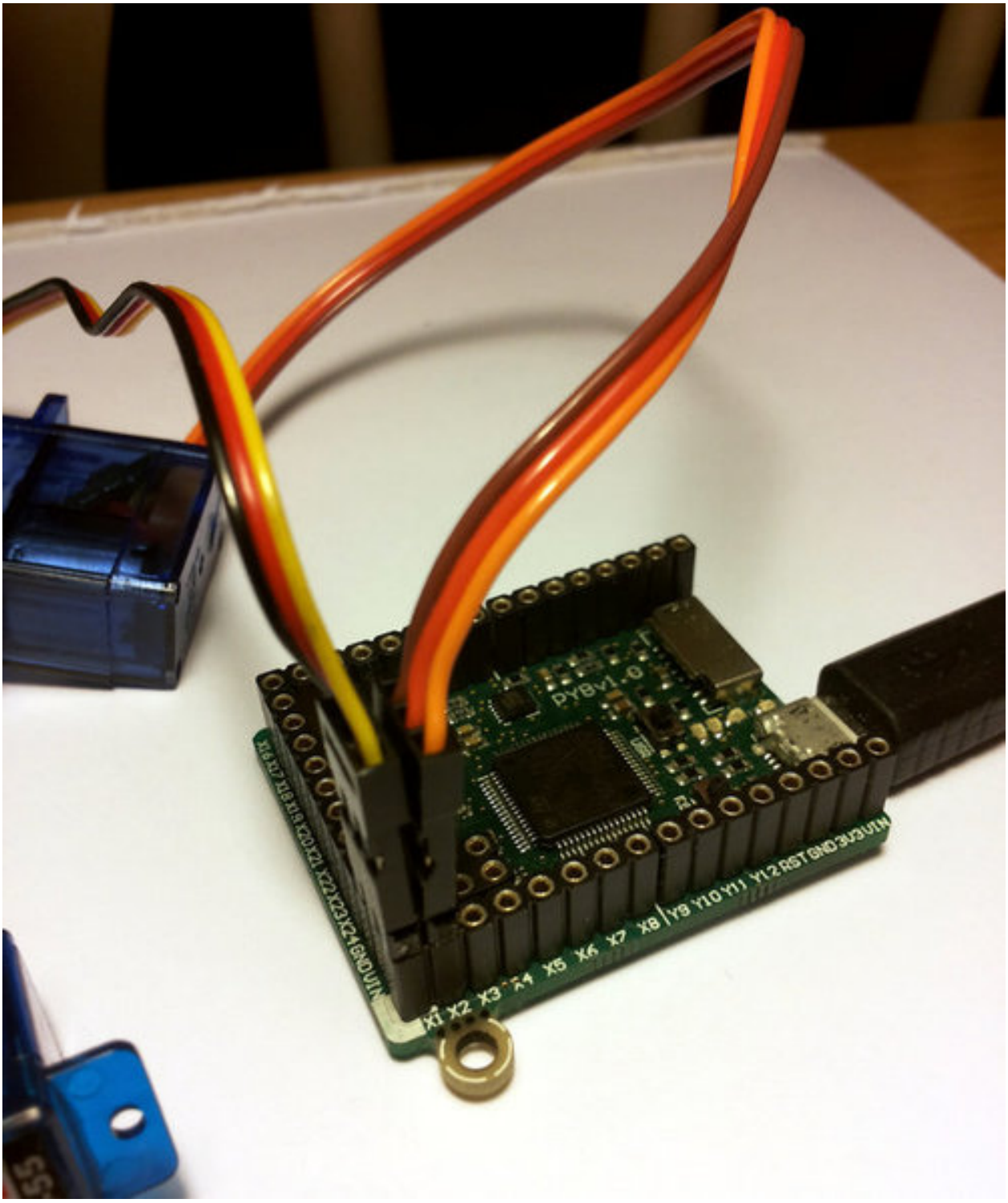
Управление частотой с помощью *pyb.freq()*:

```
pyb.freq(30000000) # set CPU frequency to 30MHz
```

Руководства требующие дополнительных компонент

Controlling hobby servo motors

There are 4 dedicated connection points on the pyboard for connecting up hobby servo motors (see eg [Wikipedia](http://en.wikipedia.org/wiki/Servo_%28radio_control%29)). These motors have 3 wires: ground, power and signal. On the pyboard you can connect them in the bottom right corner, with the signal pin on the far right. Pins X1, X2, X3 and X4 are the 4 dedicated servo signal pins.



In this picture there are male-male double adaptors to connect the servos to the header pins on the pyboard.

The ground wire on a servo is usually the darkest coloured one, either black or dark brown. The power wire will most likely be red.

The power pin for the servos (labelled VIN) is connected directly to the input power source of the pyboard. When powered via USB, VIN is powered through a diode by the 5V USB power line. Connect to USB, the pyboard can power at least 4 small to medium sized servo motors.

If using a battery to power the pyboard and run servo motors, make sure it is not greater than 6V, since this is the maximum voltage most servo motors can take. (Some motors take only up to 4.8V, so check what type you are using.)

Creating a Servo object

Plug in a servo to position 1 (the one with pin X1) and create a servo object using:

```
>>> servo1 = pyb.Servo(1)
```

To change the angle of the servo use the `angle` method:

```
>>> servo1.angle(45)
>>> servo1.angle(-60)
```

The angle here is measured in degrees, and ranges from about -90 to +90, depending on the motor. Calling `angle` without parameters will return the current angle:

```
>>> servo1.angle()
-60
```

Note that for some angles, the returned angle is not exactly the same as the angle you set, due to rounding errors in setting the pulse width.

You can pass a second parameter to the `angle` method, which specifies how long to take (in milliseconds) to reach the desired angle. For example, to take 1 second (1000 milliseconds) to go from the current position to 50 degrees, use

```
>>> servo1.angle(50, 1000)
```

This command will return straight away and the servo will continue to move to the desired angle, and stop when it gets there. You can use this feature as a speed control, or to synchronise 2 or more servo motors. If we have another servo motor (`servo2 = pyb.Servo(2)`) then we can do

```
>>> servo1.angle(-45, 2000); servo2.angle(60, 2000)
```

This will move the servos together, making them both take 2 seconds to reach their final angles.

Note: the semicolon between the 2 expressions above is used so that they are executed one after the other when you press enter at the REPL prompt. In a script you don't need to do this, you can just write them one line after the other.

Continuous rotation servos

So far we have been using standard servos that move to a specific angle and stay at that angle. These servo motors are useful to create joints of a robot, or things like pan-tilt mechanisms. Internally, the motor has a variable resistor (potentiometer) which measures the current angle and applies power to the motor proportional to how far it is from the desired angle. The desired angle is set by the width of a high-pulse on the servo signal wire. A pulse width of 1500 microsecond corresponds to the centre position (0 degrees). The pulses are sent at 50 Hz, ie 50 pulses per second.

You can also get **continuous rotation** servo motors which turn continuously clockwise or counterclockwise. The direction and speed of rotation is set by the pulse width on the signal wire. A pulse width of 1500 microseconds corresponds to a stopped motor. A pulse width smaller or larger than this means rotate one way or the other, at a given speed.

On the pyboard, the servo object for a continuous rotation motor is the same as before. In fact, using `angle` you can set the speed. But to make it easier to understand what is intended, there is another method called `speed` which sets the speed:

```
>>> servo1.speed(30)
```

`speed` has the same functionality as `angle`: you can get the speed, set it, and set it with a time to reach the final speed.

```
>>> servo1.speed()
30
>>> servo1.speed(-20)
>>> servo1.speed(0, 2000)
```

The final command above will set the motor to stop, but take 2 seconds to do it. This is essentially a control over the acceleration of the continuous servo.

A servo speed of 100 (or -100) is considered maximum speed, but actually you can go a bit faster than that, depending on the particular motor.

The only difference between the `angle` and `speed` methods (apart from the name) is the way the input numbers (angle or speed) are converted to a pulse width.

Calibration

The conversion from angle or speed to pulse width is done by the servo object using its calibration values. To get the current calibration, use

```
>>> servo1.calibration()
(640, 2420, 1500, 2470, 2200)
```

There are 5 numbers here, which have meaning:

1. Minimum pulse width; the smallest pulse width that the servo accepts.
2. Maximum pulse width; the largest pulse width that the servo accepts.
3. Centre pulse width; the pulse width that puts the servo at 0 degrees or 0 speed.
4. The pulse width corresponding to 90 degrees. This sets the conversion in the method `angle` of angle to pulse width.
5. The pulse width corresponding to a speed of 100. This sets the conversion in the method `speed` of speed to pulse width.

You can recalibrate the servo (change its default values) by using:

```
>>> servo1.calibration(700, 2400, 1510, 2500, 2000)
```

Of course, you would change the above values to suit your particular servo motor.

Fading LEDs

In addition to turning LEDs on and off, it is also possible to control the brightness of an LED using Pulse-Width Modulation (PWM), a common technique for obtaining variable output from a digital pin. This allows us to fade an LED:

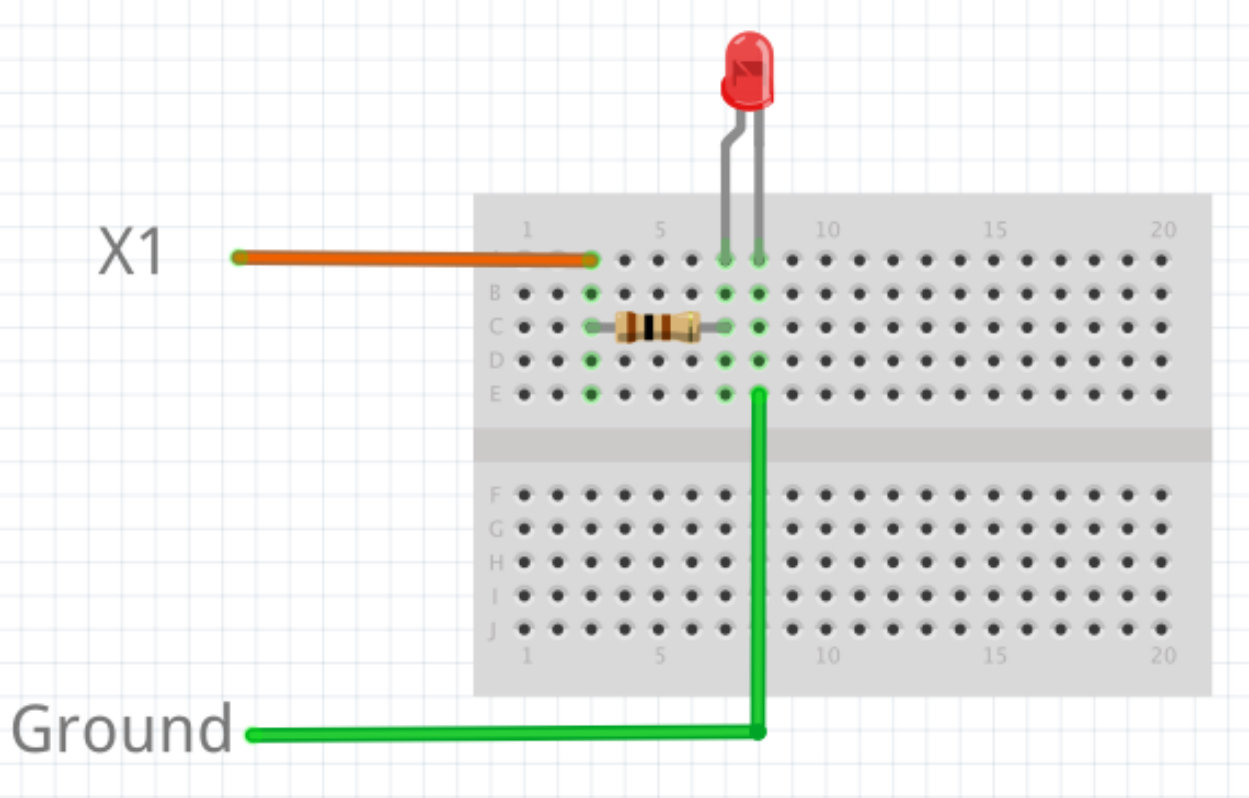
Components

You will need:

- Standard 5 or 3 mm LED
- 100 Ohm resistor
- Wires
- Breadboard (optional, but makes things easier)

Connecting Things Up

For this tutorial, we will use the `X1` pin. Connect one end of the resistor to `X1`, and the other end to the **anode** of the LED, which is the longer leg. Connect the **cathode** of the LED to ground.



Code

By examining the *Краткое описание микроконтроллера pyboard*, we see that `X1` is connected to channel 1 of timer 5 (`TIM5 CH1`). Therefore we will first create a `Timer` object for timer 5, then create a `TimerChannel` object for channel 1:

```
from pyb import Timer
from time import sleep

# timer 5 will be created with a frequency of 100 Hz
tim = pyb.Timer(5, freq=100)
tchannel = tim.channel(1, Timer.PWM, pin=pyb.Pin.board.X1, pulse_width=0)
```

Brightness of the LED in PWM is controlled by controlling the pulse-width, that is the amount of time the LED is on every cycle. With a timer frequency of 100 Hz, each cycle takes 0.01 second, or 10 ms.

To achieve the fading effect shown at the beginning of this tutorial, we want to set the pulse-width to a small value, then slowly increase the pulse-width to brighten the LED, and start over when we reach some maximum brightness:

```
# maximum and minimum pulse-width, which corresponds to maximum
# and minimum brightness
max_width = 200000
min_width = 20000

# how much to change the pulse-width by each step
wstep = 1500
cur_width = min_width

while True:
  tchannel.pulse_width(cur_width)

  # this determines how often we change the pulse-width. It is
  # analogous to frames-per-second
  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = min_width
```

Breathing Effect

If we want to have a breathing effect, where the LED fades from dim to bright then bright to dim, then we simply need to reverse the sign of `wstep` when we reach maximum brightness, and reverse it again at minimum brightness. To do this we modify the `while` loop to be:

```
while True:
  tchannel.pulse_width(cur_width)

  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = max_width
    wstep *= -1
  elif cur_width < min_width:
    cur_width = min_width
    wstep *= -1
```

Advanced Exercise

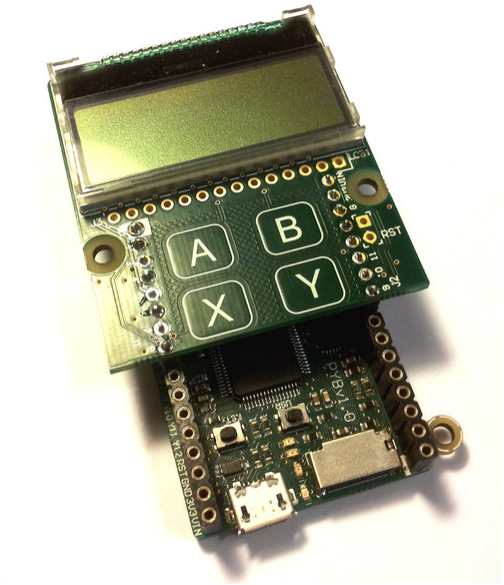
You may have noticed that the LED brightness seems to fade slowly, but increases quickly. This is because our eyes interprets brightness logarithmically (Weber's Law), while the LED's brightness changes linearly, that is by the same amount each time. How do you solve this problem? (Hint: what is the opposite of the logarithmic function?)

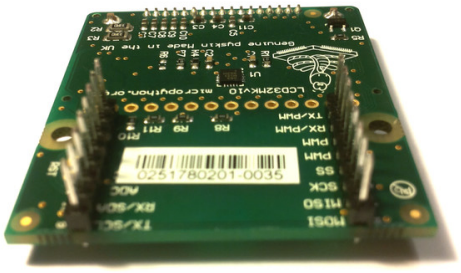
Addendum

We could have also used the digital-to-analog converter (DAC) to achieve the same effect. The PWM method has the advantage that it drives the LED with the same current each time, but for different lengths of time. This allows better control over the brightness, because LEDs do not necessarily exhibit a linear relationship between the driving current and brightness.

The LCD and touch-sensor skin

Soldering and using the LCD and touch-sensor skin.





The following video shows how to solder the headers onto the LCD skin. At the end of the video, it shows you how to correctly connect the LCD skin to the pyboard.

Using the LCD

To get started using the LCD, try the following at the Micro Python prompt. Make sure the LCD skin is attached to the pyboard as pictured at the top of this page.

```
>>> import pyb
>>> lcd = pyb.LCD('X')
>>> lcd.light(True)
>>> lcd.write('Hello uPy!\n')
```

You can make a simple animation using the code:

```
import pyb
lcd = pyb.LCD('X')
lcd.light(True)
for x in range(-80, 128):
    lcd.fill(0)
    lcd.text('Hello uPy!', x, 10, 1)
    lcd.show()
    pyb.delay(25)
```

Using the touch sensor

To read the touch-sensor data you need to use the I2C bus. The MPR121 capacitive touch sensor has address 90.

To get started, try:

```
>>> import pyb
>>> i2c = pyb.I2C(1, pyb.I2C.MASTER)
>>> i2c.mem_write(4, 90, 0x5e)
>>> touch = i2c.mem_read(1, 90, 0)[0]
```

The first line above makes an I2C object, and the second line enables the 4 touch sensors. The third line reads the touch status and the `touch` variable holds the state of the 4 touch buttons (A, B, X, Y).

There is a simple driver here which allows you to set the threshold and debounce parameters, and easily read the touch status and electrode voltage levels. Copy this script to your pyboard (either flash or SD card, in the top directory or `lib/` directory) and then try:

```
>>> import pyb
>>> import mpr121
>>> m = mpr121.MPR121(pyb.I2C(1, pyb.I2C.MASTER))
>>> for i in range(100):
...   print(m.touch_status())
...   pyb.delay(100)
...
```

This will continuously print out the touch status of all electrodes. Try touching each one in turn.

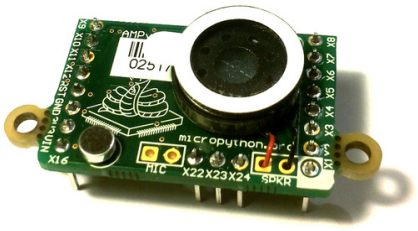
Note that if you put the LCD skin in the Y-position, then you need to initialise the I2C bus using:

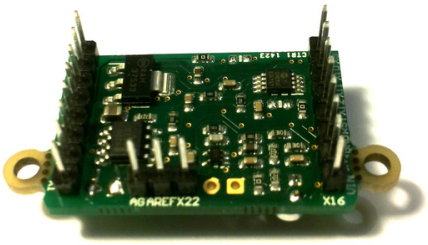
```
>>> m = mpr121.MPR121(pyb.I2C(2, pyb.I2C.MASTER))
```

There is also a demo which uses the LCD and the touch sensors together, and can be found here.

The AMP audio skin

Soldering and using the AMP audio skin.





The following video shows how to solder the headers, microphone and speaker onto the AMP skin.

Example code

The AMP skin has a speaker which is connected to `DAC(1)` via a small power amplifier. The volume of the amplifier is controlled by a digital potentiometer, which is an I2C device with address 46 on the `IC2(1)` bus.

To set the volume, define the following function:

```
import pyb
def volume(val):
    pyb.I2C(1, pyb.I2C.MASTER).mem_write(val, 46, 0)
```

Then you can do:

```
>>> volume(0)   # minimum volume
>>> volume(127) # maximum volume
```

To play a sound, use the `write_timed` method of the `DAC` object. For example:

```
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)
```

You can also play WAV files using the Python `wave` module. You can get the wave module here and you will also need the chunk module available here. Put these on your pyboard (either on the flash or the SD card in the top-level directory). You will need an 8-bit WAV file to play, such as this one. Then you can do:

```
>>> import wave
>>> from pyb import DAC
>>> dac = DAC(1)
>>> f = wave.open('test.wav')
>>> dac.write_timed(f.readframes(f.getnframes()), f.getframerate())
```

This should play the WAV file.

Советы, фишки и полезные штуки, которые стоит знать

Debouncing a pin input

A pin used as input from a switch or other mechanical device can have a lot of noise on it, rapidly changing from low to high when the switch is first pressed or released. This noise can be eliminated using a capacitor (a debouncing circuit). It can also be eliminated using a simple function that makes sure the value on the pin is stable.

The following function does just this. It gets the current value of the given pin, and then waits for the value to change. The new pin value must be stable for a continuous 20ms for it to register the change. You can adjust this time (to say 50ms) if you still have noise.

```
import pyb

def wait_pin_change(pin):
    # wait for pin to change value
    # it needs to be stable for a continuous 20ms
    cur_value = pin.value()
    active = 0
    while active < 20:
        if pin.value() != cur_value:
            active += 1
        else:
            active = 0
        pyb.delay(1)
```

Use it something like this:

```
import pyb

pin_x1 = pyb.Pin('X1', pyb.Pin.IN, pyb.Pin.PULL_DOWN)
while True:
    wait_pin_change(pin_x1)
    pyb.LED(4).toggle()
```

Making a UART - USB pass through

It's as simple as:

```
import pyb
import select

def pass_through(usb, uart):
    while True:
```

```
        select.select([usb, uart], [], [])
        if usb.any():
            uart.write(usb.read(256))
        if uart.any():
            usb.write(uart.read(256))

pass_through(pyb.USB_VCP(), pyb.UART(1, 9600))
```

Глава 4

Библиотеки Micro Python

Стандартные библиотеки Python

Следующие стандартные библиотеки Python встроены в Micro Python.

Для установки дополнительных библиотек, пожалуйста загрузите их из micropython-lib repository.

cmath – mathematical functions for complex numbers

The `cmath` module provides some basic mathematical funtions for working with complex numbers.

Functions

`cmath.cos(`*z*`)`
: Return the cosine of `z`.

`cmath.exp(`*z*`)`
: Return the exponential of `z`.

`cmath.log(`*z*`)`
: Return the natural logarithm of `z`. The branch cut is along the negative real axis.

`cmath.log10(`*z*`)`
: Return the base-10 logarithm of `z`. The branch cut is along the negative real axis.

`cmath.phase(`*z*`)`
: Returns the phase of the number `z`, in the range (-pi, +pi].

`cmath.polar(`*z*`)`
: Returns, as a tuple, the polar form of `z`.

`cmath.rect(`*r*, *phi*`)`
: Returns the complex number with modulus `r` and phase `phi`.

`cmath.sin(`*z*`)`
Return the sine of `z`.

`cmath.sqrt(`*z*`)`
Return the square-root of `z`.

Constants

`cmath.e`
base of the natural logarithm

`cmath.pi`
the ratio of a circle's circumference to its diameter

gc – control the garbage collector

Functions

`gc.enable()`
Enable automatic garbage collection.

`gc.disable()`
Disable automatic garbage collection. Heap memory can still be allocated, and garbage collection can still be initiated manually using *`gc.collect()`*.

`gc.collect()`
Run a garbage collection.

`gc.mem_alloc()`
Return the number of bytes of heap RAM that are allocated.

`gc.mem_free()`
Return the number of bytes of available heap RAM.

math – mathematical functions

The `math` module provides some basic mathematical funtions for working with floating-point numbers.

Note: On the pyboard, floating-point numbers have 32-bit precision.

Functions

`math.acos(`*x*`)`
Return the inverse cosine of `x`.

`math.acosh(`*x*`)`
Return the inverse hyperbolic cosine of `x`.

`math.asin(`*x*`)`
Return the inverse sine of `x`.

`math.asinh(`*x*`)`
Return the inverse hyperbolic sine of `x`.

`math.atan(`*x*`)`
Return the inverse tangent of `x`.

`math.atan2(`*y*, *x*`)`
Return the principal value of the inverse tangent of `y/x`.

`math.atanh(`*x*`)`
Return the inverse hyperbolic tangent of `x`.

`math.ceil(`*x*`)`
Return an integer, being `x` rounded towards positive infinity.

`math.copysign(`*x*, *y*`)`
Return `x` with the sign of `y`.

`math.cos(`*x*`)`
Return the cosine of `x`.

`math.cosh(`*x*`)`
Return the hyperbolic cosine of `x`.

`math.degrees(`*x*`)`
Return radians `x` converted to degrees.

`math.erf(`*x*`)`
Return the error function of `x`.

`math.erfc(`*x*`)`
Return the complementary error function of `x`.

`math.exp(`*x*`)`
Return the exponential of `x`.

`math.expm1(`*x*`)`
Return `exp(x) - 1`.

`math.fabs(`*x*`)`
Return the absolute value of `x`.

`math.floor(`*x*`)`
Return an integer, being `x` rounded towards negative infinity.

`math.fmod(`*x*, *y*`)`
Return the remainder of `x/y`.

`math.frexp(`*x*`)`
Converts a floating-point number to fractional and integral components.

`math.gamma(`*x*`)`
Return the gamma function of `x`.

`math.isfinite(`*x*`)`
Return `True` if `x` is finite.

`math.isinf(`*x*`)`
Return `True` if `x` is infinite.

`math.isnan(`*x*`)`
Return `True` if `x` is not-a-number

`math.ldexp(`*x*, *exp*`)`
Return `x * (2**exp)`.

`math.lgamma(`*x*`)`
Return the natural logarithm of the gamma function of `x`.

`math.log(`*x*`)`
Return the natural logarithm of `x`.

`math.log10(`*x*`)`
Return the base-10 logarithm of `x`.

`math.log2(`*x*`)`
Return the base-2 logarithm of `x`.

`math.modf(`*x*`)`
Return a tuple of two floats, being the fractional and integral parts of `x`. Both return values have the same sign as `x`.

`math.pow(`*x*, *y*`)`
Returns `x` to the power of `y`.

`math.radians(`*x*`)`
Return degrees `x` converted to radians.

`math.sin(`*x*`)`
Return the sine of `x`.

`math.sinh(`*x*`)`
Return the hyperbolic sine of `x`.

`math.sqrt(`*x*`)`
Return the square root of `x`.

`math.tan(`*x*`)`
Return the tangent of `x`.

`math.tanh(`*x*`)`
Return the hyperbolic tangent of `x`.

`math.trunc(`*x*`)`
Return an integer, being `x` rounded towards 0.

Constants

`math.e`
base of the natural logarithm

`math.pi`
the ratio of a circle's circumference to its diameter

os – basic "operating system" services

The `os` module contains functions for filesystem access and `urandom`.

Pyboard specifics

The filesystem on the pyboard has `/` as the root directory and the available physical drives are accessible from here. They are currently:

`/flash` – the internal flash filesystem

`/sd` – the SD card (if it exists)

On boot up, the current directory is `/flash` if no SD card is inserted, otherwise it is `/sd`.

Functions

`os.chdir(path)`
Change current directory.

`os.getcwd()`
Get the current directory.

`os.listdir([dir])`
With no argument, list the current directory. Otherwise list the given directory.

`os.mkdir(path)`
Create a new directory.

`os.remove(path)`
Remove a file.

`os.rmdir(path)`
Remove a directory.

`os.stat(path)`
Get the status of a file or directory.

`os.sync()`
Sync all filesystems.

`os.urandom(n)`
Return a bytes object with n random bytes, generated by the hardware random number generator.

Constants

`os.sep`
separation character used in paths

`select` – Provides select function to wait for events on a stream

This module provides the select function.

Pyboard specifics

Polling is an efficient way of waiting for read/write activity on multiple objects. Current objects that support polling are: *pyb.UART*, *pyb.USB_VCP*.

Functions

`select.poll()`
Create an instance of the Poll class.

`select.select(rlist, wlist, xlist[, timeout])`
Wait for activity on a set of objects.

class `Poll`

Methods

`poll.register(`*obj*`[,` *eventmask*`])`
Register `obj` for polling. `eventmask` is 1 for read, 2 for write, 3 for read-write.

`poll.unregister(`*obj*`)`
Unregister `obj` from polling.

`poll.modify(`*obj*, *eventmask*`)`
Modify the `eventmask` for `obj`.

`poll.poll(`[*timeout*]`)`
Wait for one of the registered objects to become ready.

Timeout is in milliseconds.

`struct` – pack and unpack primitive data types

See Python struct for more information.

Functions

`struct.calcsize(`*fmt*`)`
Return the number of bytes needed to store the given `fmt`.

`struct.pack(`*fmt*, *v1*, *v2*, ...`)`
Pack the values `v1`, `v2`, ... according to the format string `fmt`. The return value is a bytes object encoding the values.

`struct.unpack(`*fmt*, *data*`)`
Unpack from the `data` according to the format string `fmt`. The return value is a tuple of the unpacked values.

`sys` – system specific functions

Functions

`sys.exit(`[*retval*]`)`
Raise a `SystemExit` exception. If an argument is given, it is the value given to `SystemExit`.

Constants

`sys.argv`
a mutable list of arguments this program started with

`sys.byteorder`
the byte order of the system ("little" or "big")

`sys.path`
a mutable list of directories to search for imported modules

`sys.platform`
: the platform that Micro Python is running on

`sys.stderr`
: standard error (connected to USB VCP, and optional UART object)

`sys.stdin`
: standard input (connected to USB VCP, and optional UART object)

`sys.stdout`
: standard output (connected to USB VCP, and optional UART object)

`sys.version`
: Python language version that this implementation conforms to, as a string

`sys.version_info`
: Python language version that this implementation conforms to, as a tuple of ints

`time` – time related functions

The `time` module provides functions for getting the current time and date, and for sleeping.

Functions

`time.localtime(`[*secs*]`)`
: Convert a time expressed in seconds since Jan 1, 2000 into an 8-tuple which contains: (year, month, mday, hour, minute, second, weekday, yearday) If secs is not provided or None, then the current time from the RTC is used. year includes the century (for example 2014).

  - month is 1-12
  - mday is 1-31
  - hour is 0-23
  - minute is 0-59
  - second is 0-59
  - weekday is 0-6 for Mon-Sun
  - yearday is 1-366

`time.mktime()`
: This is inverse function of localtime. It's argument is a full 8-tuple which expresses a time as per localtime. It returns an integer which is the number of seconds since Jan 1, 2000.

`time.sleep(`*seconds*`)`
: Sleep for the given number of seconds. Seconds can be a floating-point number to sleep for a fractional number of seconds.

`time.time()`
: Returns the number of seconds, as an integer, since 1/1/2000.

Микро-библиотеки Python

Следующие стандартные библиотеки Python были немного модифицированы для соответствия философии Micro Python. Они сохранили базовую функциональность.

Такие библиотеки имеют в имени приставку u и само название без приставки. Название может быть изменено с помощью файла, имя которого в вашем пути к пакету. Например, `import.json` будет в первую очередь искать файл `json.py` или папку `json` и загрузит этот пакет, если найдёт. Однако, если ничего не наёдёт - будет загружен встроенный модуль `ujson`.

`usocket` – socket module

Socket functionality.

Functions

`usocket.getaddrinfo(`*host*, *port*`)`

`usocket.socket(`*family=AF_INET*, *type=SOCK_STREAM*, *fileno=-1*`)`
Create a socket.

`uheapq` – heap queue algorithm

This module implements the heap queue algorithm.

A heap queue is simply a list that has its elements stored in a certain way.

Functions

`uheapq.heappush(`*heap*, *item*`)`
Push the `item` onto the `heap`.

`uheapq.heappop(`*heap*`)`
Pop the first item froh the `heap`, and return it. Raises IndexError if heap is empty.

`uheapq.heapify(`*x*`)`
Convert the list `x` into a heap. This is an in-place operation.

`ujson` – JSON encoding and decoding

This modules allows to convert between Python objects and the JSON data format.

Functions

`ujson.dumps(`*obj*`)`
Return `obj` represented as a JSON string.

`ujson.loads(`*str*`)`
Parse the JSON `str` and return an object. Raises ValueError if the string is not correctly formed.

Специальные библиотеки для pyboard

Следующие библиотеки являются специфическими для pyboard:

pyb — functions related to the pyboard

The pyb module contains specific functions related to the pyboard.

Time related functions

pyb.delay(*ms*)
Delay for the given number of milliseconds.

pyb.udelay(*us*)
Delay for the given number of microseconds.

pyb.millis()
Returns the number of milliseconds since the board was last reset.

The result is always a micropython smallint (31-bit signed number), so after 2^30 milliseconds (about 12.4 days) this will start to return negative numbers.

pyb.micros()
Returns the number of microseconds since the board was last reset.

The result is always a micropython smallint (31-bit signed number), so after 2^30 microseconds (about 17.8 minutes) this will start to return negative numbers.

pyb.elapsed_millis(*start*)
Returns the number of milliseconds which have elapsed since start.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods upto about 12.4 days.

Example:

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

pyb.elapsed_micros(*start*)
Returns the number of microseconds which have elapsed since start.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods upto about 17.8 minutes.

Example:

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

Reset related functions

pyb.hard_reset()
Resets the pyboard in a manner similar to pushing the external RESET button.

pyb.bootloader()
Activate the bootloader without BOOT* pins.

Interrupt related functions

`pyb.disable_irq()`
Disable interrupt requests. Returns the previous IRQ state: `False`/`True` for disabled/enabled IRQs respectively. This return value can be passed to enable_irq to restore the IRQ to its original state.

`pyb.enable_irq(`*state=True*`)`
Enable interrupt requests. If `state` is `True` (the default value) then IRQs are enabled. If `state` is `False` then IRQs are disabled. The most common use of this function is to pass it the value returned by `disable_irq` to exit a critical section.

Power related functions

`pyb.freq(`[*sys_freq*]`)`
If given no arguments, returns a tuple of clock frequencies: (SYSCLK, HCLK, PCLK1, PCLK2).

If given an argument, sets the system frequency to that value in Hz. Eg freq(120000000) gives 120MHz. Note that not all values are supported and the largest supported frequency not greater than the given sys_freq will be selected.

Supported frequencies are (in MHz): 8, 16, 24, 30, 32, 36, 40, 42, 48, 54, 56, 60, 64, 72, 84, 96, 108, 120, 144, 168.

8MHz uses the HSE (external crystal) directly and 16MHz uses the HSI (internal oscillator) directly. The higher frequencies use the HSE to drive the PLL (phase locked loop), and then use the output of the PLL.

Note that if you change the frequency while the USB is enabled then the USB may become unreliable. It is best to change the frequency in boot.py, before the USB peripheral is started. Also note that frequencies below 36MHz do not allow the USB to function correctly.

`pyb.wfi()`
Wait for an interrupt. This executies a `wfi` instruction which reduces power consumption of the MCU until an interrupt occurs, at which point execution continues.

`pyb.standby()`

`pyb.stop()`

Miscellaneous functions

`pyb.have_cdc()`
Return True if USB is connected as a serial device, False otherwise.

`pyb.hid(`*(buttons, x, y, z)*`)`
Takes a 4-tuple (or list) and sends it to the USB host (the PC) to signal a HID mouse-motion event.

`pyb.info(`[*dump_alloc_table*]`)`
Print out lots of information about the board.

`pyb.repl_uart(`*uart*`)`
Get or set the UART object that the REPL is repeated on.

`pyb.rng()`
Return a 30-bit hardware generated random number.

`pyb.sync()`
Sync all file systems.

`pyb.unique_id()`
Returns a string of 12 bytes (96 bits), which is the unique ID for the MCU.

Classes

class Accel – accelerometer control

Accel is an object that controls the accelerometer. Example usage:

```
accel = pyb.Accel()
for i in range(10):
    print(accel.x(), accel.y(), accel.z())
```

Raw values are between -32 and 31.

Constructors

class `pyb.Accel`
Create and return an accelerometer object.

Note: if you read accelerometer values immediately after creating this object you will get 0. It takes around 20ms for the first sample to be ready, so, unless you have some other code between creating this object and reading its values, you should put a `pyb.delay(20)` after creating it. For example:

```
accel = pyb.Accel()
pyb.delay(20)
print(accel.x())
```

Methods

`accel.filtered_xyz()`
Get a 3-tuple of filtered x, y and z values.

`accel.tilt()`
Get the tilt register.

`accel.x()`
Get the x-axis value.

`accel.y()`
Get the y-axis value.

`accel.z()`
Get the z-axis value.

class ADC – analog to digital conversion: read analog values on a pin

Usage:

```
import pyb

adc = pyb.ADC(pin)              # create an analog object from a pin
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
```

Constructors

class pyb.ADC(*pin*)
Create an ADC object associated with the given pin. This allows you to then read analog values on that pin.

Methods

adc.read()
Read the value on the analog pin and return it. The returned value will be between 0 and 4095.

adc.read_timed(*buf*, *freq*)
Read analog values into the given buffer at the given frequency. Buffer can be bytearray or array.array for example. If a buffer with 8-bit elements is used, sample resolution will be reduced to 8 bits.

Example:

```
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
```

This function does not allocate any memory.

class CAN – controller area network communication bus

CAN implements the standard CAN communications protocol. At the physical level it consists of 2 lines: RX and TX. Note that to connect the pyboard to a CAN bus you must use a CAN transceiver to convert the CAN logic signals from the pyboard to the correct voltage levels on the bus.

Note that this driver does not yet support filter configuration (it defaults to a single filter that lets through all messages), or bus timing configuration (except for setting the prescaler).

Example usage (works without anything connected):

```
from pyb import CAN
can = pyb.CAN(1, pyb.CAN.LOOPBACK)
can.send('message!', 123)   # send message to id 123
can.recv(0)                 # receive message on FIFO 0
```

Constructors

class pyb.CAN(*bus*, ...)
Construct a CAN object on the given bus. `bus` can be 1-2, or 'YA' or 'YB'. With no additional parameters, the CAN object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

The physical pins of the CAN busses are:

- `CAN(1)` is on `YA`: `(RX, TX) = (Y3, Y4) = (PB8, PB9)`
- `CAN(2)` is on `YB`: `(RX, TX) = (Y5, Y6) = (PB12, PB13)`

Methods

`can.init(`*mode*, *extframe=False*, *prescaler=100*, ***, *sjw=1*, *bs1=6*, *bs2=8*`)`
Initialise the CAN bus with the given parameters:

- `mode` is one of: NORMAL, LOOPBACK, SILENT, SILENT_LOOPBACK

If `extframe` is True then the bus uses extended identifiers in the frames (29 bits). Otherwise it uses standard 11 bit identifiers.

`can.deinit()`
Turn off the CAN bus.

`can.any(`*fifo*`)`
Return `True` if any message waiting on the FIFO, else `False`.

`can.recv(`*fifo*, ***, *timeout=5000*`)`
Receive data on the bus:

- `fifo` is an integer, which is the FIFO to receive on
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: buffer of data bytes.

`can.send(`*send*, *addr*, ***, *timeout=5000*`)`
Send a message on the bus:

- `send` is the data to send (an integer to send, or a buffer object).
- `addr` is the address to send to
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: `None`.

Constants

`CAN.NORMAL`

`CAN.LOOPBACK`

`CAN.SILENT`

`CAN.SILENT_LOOPBACK`
the mode of the CAN bus

class DAC – digital to analog conversion

The DAC is used to output analog values (a specific voltage) on pin X5 or pin X6. The voltage will be between 0 and 3.3V.

This module will undergo changes to the API.

Example usage:

```
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)
```

To output a continuous sine-wave:

```
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)
```

Constructors

class pyb.DAC(*port*)
Construct a new DAC object.

`port` can be a pin object, or an integer (1 or 2). DAC(1) is on pin X5 and DAC(2) is on pin X6.

Methods

`dac.noise(`*freq*`)`
Generate a pseudo-random noise signal. A new random sample is written to the DAC output at the given frequency.

`dac.triangle(`*freq*`)`
Generate a triangle wave. The value on the DAC output changes at the given frequency, and the frequence of the repeating triangle wave itself is 256 (or 1024, need to check) times smaller.

`dac.write(`*value*`)`
Direct access to the DAC output (8 bit only at the moment).

`dac.write_timed(`*data*, *freq*, *, *mode=DAC.NORMAL*`)`
Initiates a burst of RAM to DAC using a DMA transfer. The input data is treated as an array of bytes (8 bit data).

`mode` can be `DAC.NORMAL` or `DAC.CIRCULAR`.

TIM6 is used to control the frequency of the transfer.

class ExtInt – configure I/O pins to interrupt on external events

There are a total of 22 interrupt lines. 16 of these can come from GPIO pins and the remaining 6 are from internal sources.

For lines 0 thru 15, a given line can map to the corresponding line from an arbitrary port. So line 0 can map to Px0 where x is A, B, C, ... and line 1 can map to Px1 where x is A, B, C, ...

```
def callback(line):
    print("line =", line)
```

Note: ExtInt will automatically configure the gpio line as an input.

```
extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING, pyb.Pin.PULL_UP, callback)
```

Now every time a falling edge is seen on the X1 pin, the callback will be called. Caution: mechanical pushbuttons have "bounce" and pushing or releasing a switch will often generate multiple edges. See: http://www.eng.utah.edu/~cs5780/debouncing.pdf for a detailed explanation, along with various techniques for debouncing.

Trying to register 2 callbacks onto the same pin will throw an exception.

If pin is passed as an integer, then it is assumed to map to one of the internal interrupt sources, and must be in the range 16 thru 22.

All other pin objects go through the pin mapper to come up with one of the gpio pins.

```
extint = pyb.ExtInt(pin, mode, pull, callback)
```

Valid modes are pyb.ExtInt.IRQ_RISING, pyb.ExtInt.IRQ_FALLING, pyb.ExtInt.IRQ_RISING_FALLING, pyb.ExtInt.EVT_RISING, pyb.ExtInt.EVT_FALLING, and pyb.ExtInt.EVT_RISING_FALLING.

Only the IRQ_xxx modes have been tested. The EVT_xxx modes have something to do with sleep mode and the WFE instruction.

Valid pull values are pyb.Pin.PULL_UP, pyb.Pin.PULL_DOWN, pyb.Pin.PULL_NONE.

There is also a C API, so that drivers which require EXTI interrupt lines can also use this code. See extint.h for the available functions and usrsw.h for an example of using this.

Constructors

class `pyb.ExtInt`(*pin*, *mode*, *pull*, *callback*)

Create an ExtInt object:

- `pin` is the pin on which to enable the interrupt (can be a pin object or any valid pin name).
- `mode` can be one of: - `ExtInt.IRQ_RISING` - trigger on a rising edge; - `ExtInt.IRQ_FALLING` - trigger on a falling edge; - `ExtInt.IRQ_RISING_FALLING` - trigger on a rising or falling edge.
- `pull` can be one of: - `pyb.Pin.PULL_NONE` - no pull up or down resistors; - `pyb.Pin.PULL_UP` - enable the pull-up resistor; - `pyb.Pin.PULL_DOWN` - enable the pull-down resistor.
- `callback` is the function to call when the interrupt triggers. The callback function must accept exactly 1 argument, which is the line that triggered the interrupt.

Class methods

`ExtInt.regs()`
Dump the values of the EXTI registers.

Methods

`extint.disable()`
Disable the interrupt associated with the ExtInt object. This could be useful for debouncing.

`extint.enable()`
Enable a disabled interrupt.

`extint.line()`
Return the line number that the pin is mapped to.

`extint.swint()`
Trigger the callback from software.

Constants

`ExtInt.IRQ_FALLING`
interrupt on a falling edge

`ExtInt.IRQ_RISING`
interrupt on a rising edge

`ExtInt.IRQ_RISING_FALLING`
interrupt on a rising or falling edge

class I2C – a two-wire serial protocol

I2C is a two-wire protocol for communicating between devices. At the physical level it consists of 2 wires: SCL and SDA, the clock and data lines respectively.

I2C objects are created attached to a specific bus. They can be initialised when created, or initialised later on:

```
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
```

Printing the i2c object gives you information about its configuration.

Basic methods for slave are send and recv:

```
i2c.send('abc')      # send 3 bytes
i2c.send(0x42)       # send a single byte, given by the number
data = i2c.recv(3)   # receive 3 bytes
```

To receive inplace, first create a bytearray:

```
data = bytearray(3)  # create a buffer
i2c.recv(data)       # receive 3 bytes, writing them into data
```

You can specify a timeout (in ms):

```
i2c.send(b'123', timeout=2000)   # timout after 2 seconds
```

A master must specify the recipient's address:

```
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
```

Master also has other methods:

```
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)
```

Constructors

class pyb.I2C(*bus*, ...)
Construct an I2C object on the given bus. `bus` can be 1 or 2. With no additional parameters, the I2C object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

The physical pins of the I2C busses are:

- `I2C(1)` is on the X position: `(SCL, SDA) = (X9, X10) = (PB6, PB7)`
- `I2C(2)` is on the Y position: `(SCL, SDA) = (Y9, Y10) = (PB10, PB11)`

Methods

i2c.deinit()
Turn off the I2C bus.

i2c.init(*mode*, *, *addr=0x12*, *baudrate=400000*, *gencall=False*)
Initialise the I2C bus with the given parameters:

- `mode` must be either `I2C.MASTER` or `I2C.SLAVE`
- `addr` is the 7-bit address (only sensible for a slave)
- `baudrate` is the SCL clock rate (only sensible for a master)
- `gencall` is whether to support general call mode

i2c.is_ready(*addr*)
Check if an I2C device responds to the given address. Only valid when in master mode.

i2c.mem_read(*data*, *addr*, *memaddr*, *timeout=5000*, *addr_size=8*)
Read from the memory of an I2C device:

- `data` can be an integer or a buffer to read into

- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the read
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns the read data. This is only valid in master mode.

`i2c.mem_write(`*data*, *addr*, *memaddr*, *timeout=5000*, *addr_ size=8*`)`
Write to the memory of an I2C device:

- `data` can be an integer or a buffer to write from
- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the write
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns `None`. This is only valid in master mode.

`i2c.recv(`*recv*, *addr=0x00*, *timeout=5000*`)`
Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes
- `addr` is the address to receive from (only required in master mode)
- `timeout` is the timeout in milliseconds to wait for the receive

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

`i2c.scan()`
Scan all I2C addresses from 0x01 to 0x7f and return a list of those that respond. Only valid when in master mode.

`i2c.send(`*send*, *addr=0x00*, *timeout=5000*`)`
Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object)
- `addr` is the address to send to (only required in master mode)
- `timeout` is the timeout in milliseconds to wait for the send

Return value: `None`.

Constants

`I2C.MASTER`
for initialising the bus to master mode

`I2C.SLAVE`
for initialising the bus to slave mode

class LCD – LCD control for the LCD touch-sensor pyskin

The LCD class is used to control the LCD on the LCD touch-sensor pyskin, LCD32MKv1.0. The LCD is a 128x32 pixel monochrome screen, part NHD-C12832A1Z.

The pyskin must be connected in either the X or Y positions, and then an LCD object is made using:

```
lcd = pyb.LCD('X')      # if pyskin is in the X position
lcd = pyb.LCD('Y')      # if pyskin is in the Y position
```

Then you can use:

```
lcd.light(True)                 # turn the backlight on
lcd.write('Hello world!\n')     # print text to the screen
```

This driver implements a double buffer for setting/getting pixels. For example, to make a bouncing dot, try:

```
x = y = 0
dx = dy = 1
while True:
    # update the dot's position
    x += dx
    y += dy

    # make the dot bounce of the edges of the screen
    if x <= 0 or x >= 127: dx = -dx
    if y <= 0 or y >= 31: dy = -dy

    lcd.fill(0)                 # clear the buffer
    lcd.pixel(x, y, 1)          # draw the dot
    lcd.show()                  # show the buffer
    pyb.delay(50)               # pause for 50ms
```

Constructors

class pyb.LCD(*skin_position*)
Construct an LCD object in the given skin position. `skin_position` can be 'X' or 'Y', and should match the position where the LCD pyskin is plugged in.

Methods

`lcd.command`(*instr_data*, *buf*)
Send an arbitrary command to the LCD. Pass 0 for `instr_data` to send an instruction, otherwise pass 1 to send data. `buf` is a buffer with the instructions/data to send.

`lcd.contrast`(*value*)
Set the contrast of the LCD. Valid values are between 0 and 47.

`lcd.fill`(*colour*)
Fill the screen with the given colour (0 or 1 for white or black).

This method writes to the hidden buffer. Use `show()` to show the buffer.

`lcd.get`(*x*, *y*)
Get the pixel at the position `(x, y)`. Returns 0 or 1.

This method reads from the visible buffer.

`lcd.light(`*value*`)`
Turn the backlight on/off. True or 1 turns it on, False or 0 turns it off.

`lcd.pixel(`*x*, *y*, *colour*`)`
Set the pixel at `(x, y)` to the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

`lcd.show()`
Show the hidden buffer on the screen.

`lcd.text(`*str*, *x*, *y*, *colour*`)`
Draw the given text to the position `(x, y)` using the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

`lcd.write(`*str*`)`
Write the string `str` to the screen. It will appear immediately.

class LED – LED object

The LED object controls an individual LED (Light Emitting Diode).

Constructors

class `pyb.LED(`*id*`)`
Create an LED object associated with the given LED:

- `id` is the LED number, 1-4.

Methods

`led.intensity(`[*value*]`)`
Get or set the LED intensity. Intensity ranges between 0 (off) and 255 (full on). If no argument is given, return the LED intensity. If an argument is given, set the LED intensity and return `None`.

`led.off()`
Turn the LED off.

`led.on()`
Turn the LED on.

`led.toggle()`
Toggle the LED between on and off.

class Pin – control I/O pins

A pin is the basic object to control I/O pins. It has methods to set the mode of the pin (input, output, etc) and methods to get and set the digital logic level. For analog control of a pin, see the ADC class.

Usage Model:

All Board Pins are predefined as pyb.Pin.board.Name

```
x1_pin = pyb.Pin.board.X1

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

CPU pins which correspond to the board pins are available as `pyb.cpu.Name`. For the CPU pins, the names are the port letter followed by the pin number. On the PYBv1.0, `pyb.Pin.board.X1` and `pyb.Pin.cpu.B6` are the same pin.

You can also use strings:

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

Users can add their own names:

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

and can query mappings

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

Users can also add their own mapping function:

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

pyb.Pin.mapper(MyMapper)
```

So, if you were to call: `pyb.Pin("LeftMotorDir pyb.Pin.OUT_PP)` then `"LeftMotorDir"` is passed directly to the mapper function.

To summarise, the following order determines how things get mapped into an ordinal pin number:

1. Directly specify a pin object
2. User supplied mapping function
3. User supplied mapping (object must be usable as a dictionary key)
4. Supply a string which matches a board pin
5. Supply a string which matches a CPU port/pin

You can set `pyb.Pin.debug(True)` to get some debug information about how a particular object gets mapped to a pin.

When a pin has the `Pin.PULL_UP` or `Pin.PULL_DOWN` pull-mode enabled, that pin has an effective 40k Ohm resistor pulling it to 3V3 or GND respectively (except pin Y5 which has 11k Ohm resistors).

Constructors

class pyb.Pin(*id*, *...*)
Create a new Pin object associated with the id. If additional arguments are given, they are used to initialise the pin. See *pin.init()*.

Class methods

`Pin.af_list()`
Returns an array of alternate functions available for this pin.

`Pin.debug(`[*state*]`)`
Get or set the debugging state (`True` or `False` for on or off).

`Pin.dict(`[*dict*]`)`
Get or set the pin mapper dictionary.

`Pin.mapper(`[*fun*]`)`
Get or set the pin mapper function.

Methods

`pin.init(`*mode*, *pull=Pin.PULL_NONE*, *af=-1*`)`
Initialise the pin:

- `mode` can be one of: - `Pin.IN` - configure the pin for input; - `Pin.OUT_PP` - configure the pin for output, with push-pull control; - `Pin.OUT_OD` - configure the pin for output, with open-drain control; - `Pin.AF_PP` - configure the pin for alternate function, pull-pull; - `Pin.AF_OD` - configure the pin for alternate function, open-drain; - `Pin.ANALOG` - configure the pin for analog.
- `pull` can be one of: - `Pin.PULL_NONE` - no pull up or down resistors; - `Pin.PULL_UP` - enable the pull-up resistor; - `Pin.PULL_DOWN` - enable the pull-down resistor.
- when mode is Pin.AF_PP or Pin.AF_OD, then af can be the index or name of one of the alternate functions associated with a pin.

Returns: `None`.

`pin.high()`
Set the pin to a high logic level.

`pin.low()`
Set the pin to a low logic level.

`pin.value(`[*value*]`)`
Get or set the digital logic level of the pin:

- With no argument, return 0 or 1 depending on the logic level of the pin.
- With `value` given, set the logic level of the pin. `value` can be anything that converts to a boolean. If it converts to `True`, the pin is set high, otherwise it is set low.

`pin.__str__()`
Return a string describing the pin object.

`pin.af()`
Returns the currently configured alternate-function of the pin. The integer returned will match one of the allowed constants for the af argument to the init function.

`pin.gpio()`
Returns the base address of the GPIO block associated with this pin.

`pin.mode()`
Returns the currently configured mode of the pin. The integer returned will match one of the allowed constants for the mode argument to the init function.

`pin.name()`
Get the pin name.

`pin.names()`
Returns the cpu and board names for this pin.

`pin.pin()`
Get the pin number.

`pin.port()`
Get the pin port.

`pin.pull()`
Returns the currently configured pull of the pin. The integer returned will match one of the allowed constants for the pull argument to the init function.

Constants

`Pin.AF_OD`
initialise the pin to alternate-function mode with an open-drain drive

`Pin.AF_PP`
initialise the pin to alternate-function mode with a push-pull drive

`Pin.ANALOG`
initialise the pin to analog mode

`Pin.IN`
initialise the pin to input mode

`Pin.OUT_OD`
initialise the pin to output mode with an open-drain drive

`Pin.OUT_PP`
initialise the pin to output mode with a push-pull drive

`Pin.PULL_DOWN`
enable the pull-down resistor on the pin

`Pin.PULL_NONE`
don't enable any pull up or down resistors on the pin

`Pin.PULL_UP`
enable the pull-up resistor on the pin

class PinAF – Pin Alternate Functions

A Pin represents a physical pin on the microcprocessor. Each pin can have a variety of functions (GPIO, I2C SDA, etc). Each PinAF object represents a particular function for a pin.

Usage Model:

```
x3 = pyb.Pin.board.X3
x3_af = x3.af_list()
```

x3_af will now contain an array of PinAF objects which are availble on pin X3.

For the pyboard, x3_af would contain: [Pin.AF1_TIM2, Pin.AF2_TIM5, Pin.AF3_TIM9, Pin.AF7_USART2]

Normally, each peripheral would configure the af automatically, but sometimes the same function is available on multiple pins, and having more control is desired.

To configure X3 to expose TIM2_CH3, you could use:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.AF_PP, af=pyb.Pin.AF1_TIM2)
```

or:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.AF_PP, af=1)
```

Methods

`pinaf.__str__()`
Return a string describing the alternate function.

`pinaf.index()`
Return the alternate function index.

`pinaf.name()`
Return the name of the alternate function.

`pinaf.reg()`
Return the base register associated with the peripheral assigned to this alternate function. For example, if the alternate function were TIM2_CH3 this would return stm.TIM2

class RTC – real time clock

The RTC is and independent clock that keeps track of the date and time.

Example usage:

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

Constructors

class `pyb.RTC`
Create an RTC object.

Methods

`rtc.datetime(`[*datetimetuple*]`)`
Get or set the date and time of the RTC.

With no arguments, this method returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time.

The 8-tuple has the following format:

(year, month, day, weekday, hours, minutes, seconds, subseconds)

`weekday` is 1-7 for Monday through Sunday.

`subseconds` counts down from 255 to 0

`rtc.info()`
Get information about the startup time and reset source.

- The lower 0xffff are the number of milliseconds the RTC took to start up.
- Bit 0x10000 is set if a power-on reset occurred.
- Bit 0x20000 is set if an external reset occurred

class Servo – 3-wire hobby servo driver

Servo controls standard hobby servos with 3-wires (ground, power, signal).

Constructors

class `pyb.Servo(`*id*`)`
Create a servo object. `id` is 1-4.

Methods

`servo.angle(`[*angle*, *time=0*]`)`
Get or set the angle of the servo.

- `angle` is the angle to move to in degrees.
- `time` is the number of milliseconds to take to get to the specified angle.

`servo.calibration(`[*pulse_ min*, *pulse_ max*, *pulse_ centre*[, *pulse_ angle_ 90*, *pulse_ speed_ 100*]]`)`
Get or set the calibration of the servo timing.

`servo.pulse_width(`[*value*]`)`
Get or set the pulse width in milliseconds.

`servo.speed(`[*speed*, *time=0*]`)`
Get or set the speed of a continuous rotation servo.

- `speed` is the speed to move to change to, between -100 and 100.
- `time` is the number of milliseconds to take to get to the specified speed.

class SPI – a master-driven serial protocol

SPI is a serial protocol that is driven by a master. At the physical level there are 3 lines: SCK, MOSI, MISO.

See usage model of I2C; SPI is very similar. Main difference is parameters to init the SPI bus:

```
from pyb import SPI
spi = SPI(1, SPI.MASTER, baudrate=600000, polarity=1, phase=0, crc=0x7)
```

Only required parameter is mode, SPI.MASTER or SPI.SLAVE. Polarity can be 0 or 1, and is the level the idle clock line sits at. Phase can be 0 or 1 to sample data on the first or second clock edge respectively. Crc can be None for no CRC, or a polynomial specifier.

Additional method for SPI:

```
data = spi.send_recv(b'1234')        # send 4 bytes and receive 4 bytes
buf = bytearray(4)
spi.send_recv(b'1234', buf)          # send 4 bytes and receive 4 into buf
spi.send_recv(buf, buf)              # send/recv 4 bytes from/to buf
```

Constructors

class pyb.SPI(*bus*, ...)
Construct an SPI object on the given bus. `bus` can be 1 or 2. With no additional parameters, the SPI object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

The physical pins of the SPI busses are:

- `SPI(1)` is on the X position: `(NSS, SCK, MISO, MOSI) = (X5, X6, X7, X8) = (PA4, PA5, PA6, PA7)`
- `SPI(2)` is on the Y position: `(NSS, SCK, MISO, MOSI) = (Y5, Y6, Y7, Y8) = (PB12, PB13, PB14, PB15)`

At the moment, the NSS pin is not used by the SPI driver and is free for other use.

Methods

`spi.deinit()`
Turn off the SPI bus.

`spi.init(`*mode*, *baudrate=328125*, ***, *polarity=1*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *ti=False*, *crc=None*`)`
Initialise the SPI bus with the given parameters:

- `mode` must be either `SPI.MASTER` or `SPI.SLAVE`.
- `baudrate` is the SCK clock rate (only sensible for a master).

`spi.recv(`*recv*, ***, *timeout=5000*`)`
Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

`spi.send(`*send*, ***, *timeout=5000*`)`
Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: `None`.

`spi.send_recv(`*send*, *recv=None*, ***, *timeout=5000*`)`
Send and receive data on the bus at the same time:

- `send` is the data to send (an integer to send, or a buffer object).

- `recv` is a mutable buffer which will be filled with received bytes. It can be the same as `send`, or omitted. If omitted, a new buffer will be created.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: the buffer with the received bytes.

Constants

`SPI.MASTER`

`SPI.SLAVE`
for initialising the SPI bus to master or slave mode

`SPI.LSB`

`SPI.MSB`
set the first bit to be the least or most significant bit

class Switch – switch object

A Switch object is used to control a push-button switch.

Usage:

```
sw = pyb.Switch()       # create a switch object
sw()                    # get state (True if pressed, False otherwise)
sw.callback(f)          # register a callback to be called when the
                        #   switch is pressed down
sw.callback(None)       # remove the callback
```

Example:

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

Constructors

class `pyb.Switch`
Create and return a switch object.

Methods

`switch()`
Return the switch state: `True` if pressed down, `False` otherwise.

`switch.callback(`*fun*`)`
Register the given function to be called when the switch is pressed down. If `fun` is `None`, then it disables the callback.

class Timer – control internal timers

Timers can be used for a great variety of tasks. At the moment, only the simplest case is implemented: that of calling a function periodically.

Each timer consists of a counter that counts up at a certain rate. The rate at which it counts is the peripheral clock frequency (in Hz) divided by the timer prescaler. When the counter reaches the timer period it triggers an event, and the counter resets back to zero. By using the callback method, the timer event can call a Python function.

Example usage to toggle an LED at a fixed frequency:

```
tim = pyb.Timer(4)              # create a timer object using timer 4
tim.init(freq=2)                # trigger at 2Hz
tim.callback(lambda t:pyb.LED(1).toggle())
```

Further examples:

```
tim = pyb.Timer(4, freq=100)    # freq in Hz
tim = pyb.Timer(4, prescaler=0, period=99)
tim.counter()                   # get counter (can also set)
tim.prescaler(2)                # set prescaler (can also get)
tim.period(199)                 # set period (can also get)
tim.callback(lambda t: ...)     # set callback for update interrupt (t=tim instance)
tim.callback(None)              # clear callback
```

Note: Timer 3 is reserved for internal use. Timer 5 controls the servo driver, and Timer 6 is used for timed ADC/DAC reading/writing. It is recommended to use the other timers in your programs.

Constructors

class `pyb.Timer`(*id*, *...*)
Construct a new timer object of the given id. If additional arguments are given, then the timer is initialised by `init(...)`. `id` can be 1 to 14, excluding 3.

Methods

`timer.callback`(*fun*)
Set the function to be called when the timer triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

`timer.channel`(*channel*, *mode*, *...*)
If only a channel number is passed, then a previously initialized channel object is returned (or `None` if there is no previous channel).

Othwerwise, a TimerChannel object is initialized and returned.

Each channel can be configured to perform pwm, output compare, or input capture. All channels share the same underlying timer, which means that they share the same timer clock.

Keyword arguments:

- `mode` can be one of:
  - `Timer.PWM` — configure the timer in PWM mode (active high).
  - `Timer.PWM_INVERTED` — configure the timer in PWM mode (active low).
  - `Timer.OC_TIMING` — indicates that no pin is driven.
  - `Timer.OC_ACTIVE` — the pin will be made active when a compare match occurs (active is determined by polarity)
  - `Timer.OC_INACTIVE` — the pin will be made inactive when a compare match occurs.

– `Timer.OC_TOGGLE` — the pin will be toggled when an compare match occurs.

– `Timer.OC_FORCED_ACTIVE` — the pin is forced active (compare match is ignored).

– `Timer.OC_FORCED_INACTIVE` — the pin is forced inactive (compare match is ignored).

– `Timer.IC` — configure the timer in Input Capture mode.

- `callback` - as per TimerChannel.callback()
- `pin` None (the default) or a Pin object. If specified (and not None) this will cause the alternate function of the the indicated pin to be configured for this timer channel. An error will be raised if the pin doesn't support any alternate functions for this timer channel.

Keyword arguments for Timer.PWM modes:

- `pulse_width` - determines the initial pulse width value to use.
- `pulse_width_percent` - determines the initial pulse width percentage to use.

Keyword arguments for Timer.OC modes:

- `compare` - determines the initial value of the compare register.
- `polarity` can be one of: - `Timer.HIGH` - output is active high - `Timer.LOW` - output is acive low

Optional keyword arguments for Timer.IC modes:

- `polarity` can be one of: - `Timer.RISING` - captures on rising edge. - `Timer.FALLING` - captures on falling edge. - `Timer.BOTH` - captures on both edges.

Note that capture only works on the primary channel, and not on the complimentary channels.

PWM Example:

```
timer = pyb.Timer(2, freq=1000)
ch2 = timer.channel(2, pyb.Timer.PWM, pin=pyb.Pin.board.X2, pulse_width=210000)
ch3 = timer.channel(3, pyb.Timer.PWM, pin=pyb.Pin.board.X3, pulse_width=420000)
```

`timer.counter(`[*value*]`)`
Get or set the timer counter.

`timer.deinit()`
Deinitialises the timer.

Disables the callback (and the associated irq). Disables any channel callbacks (and the associated irq). Stops the timer, and disables the timer peripheral.

`timer.freq(`[*value*]`)`
Get or set the frequency for the timer (changes prescaler and period if set).

`timer.init(`*, *freq*, *prescaler*, *period*`)`
Initialise the timer. Initialisation must be either by frequency (in Hz) or by prescaler and period:

```
tim.init(freq=100)                  # set the timer to trigger at 100Hz
tim.init(prescaler=83, period=999)  # set the prescaler and period directly
```

Keyword arguments:

- `freq` — specifies the periodic frequency of the timer. You migh also view this as the frequency with which the timer goes through one complete cycle.

- `prescaler` [0-0xffff] - specifies the value to be loaded into the timer's Prescaler Register (PSC). The timer clock source is divided by (`prescaler + 1`) to arrive at the timer clock. Timers 2-7 and 12-14 have a clock source of 84 MHz (pyb.freq()[2] * 2), and Timers 1, and 8-11 have a clock source of 168 MHz (pyb.freq()[3] * 2).
- `period` [0-0xffff] for timers 1, 3, 4, and 6-15. [0-0x3fffffff] for timers 2 & 5. Specifies the value to be loaded into the timer's AutoReload Register (ARR). This determines the period of the timer (i.e. when the counter cycles). The timer counter will roll-over after `period + 1` timer clock cycles.
- `mode` can be one of:
  - `Timer.UP` - configures the timer to count from 0 to ARR (default)
  - `Timer.DOWN` - configures the timer to count from ARR down to 0.
  - `Timer.CENTER` - confgures the timer to count from 0 to ARR and then back down to 0.
- `div` can be one of 1, 2, or 4. Divides the timer clock to determine the sampling clock used by the digital filters.
- `callback` - as per Timer.callback()
- `deadtime` - specifies the amount of "dead" or inactive time between transitions on complimentary channels (both channels will be inactive) for this time). `deadtime` may be an integer between 0 and 1008, with the following restrictions: 0-128 in steps of 1. 128-256 in steps of 2, 256-512 in steps of 8, and 512-1008 in steps of 16. `deadime` measures ticks of `source_freq` divided by `div` clock ticks. `deadtime` is only available on timers 1 and 8.

You must either specify freq or both of period and prescaler.

`timer.period(`[*value*]`)`
Get or set the period of the timer.

`timer.prescaler(`[*value*]`)`
Get or set the prescaler for the timer.

`timer.source_freq()`
Get the frequency of the source of the timer.

class TimerChannel — setup a channel for a timer

Timer channels are used to generate/capture a signal using a timer.

TimerChannel objects are created using the Timer.channel() method.

Methods

`timerchannel.callback(`*fun*`)`
Set the function to be called when the timer channel triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

`timerchannel.capture(`[*value*]`)`
Get or set the capture value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. capture is the logical name to use when the channel is in input capture mode.

`timerchannel.compare([value])`
Get or set the compare value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. compare is the logical name to use when the channel is in output compare mode.

`timerchannel.pulse_width([value])`
Get or set the pulse width value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. pulse_width is the logical name to use when the channel is in PWM mode.

In edge aligned mode, a pulse_width of `period + 1` corresponds to a duty cycle of 100% In center aligned mode, a pulse width of `period` corresponds to a duty cycle of 100%

`timerchannel.pulse_width_percent([value])`
Get or set the pulse width percentage associated with a channel. The value is a number between 0 and 100 and sets the percentage of the timer period for which the pulse is active. The value can be an integer or floating-point number for more accuracy. For example, a value of 25 gives a duty cycle of 25%.

class UART – duplex serial communication bus

UART implements the standard UART/USART duplex serial communications protocol. At the physical level it consists of 2 lines: RX and TX. The unit of communication is a character (not to be confused with a string character) which can be 8 or 9 bits wide.

UART objects can be created and initialised using:

```
from pyb import UART

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

Bits can be 7, 8 or 9. Parity can be None, 0 (even) or 1 (odd). Stop can be 1 or 2.

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

A UART object acts like a stream object and reading and writing is done using the standard stream methods:

```
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
```

Individual characters can be read/written using:

```
uart.readchar()     # read 1 character and returns it as an integer
uart.writechar(42)  # write 1 character
```

To check if there is anything to be read, use:

```
uart.any()               # returns True if any characters waiting
```

Note: The stream functions `read`, `write` etc Are new in Micro Python since v1.3.4. Earlier versions use `uart.send` and `uart.recv`.

Constructors

class pyb.UART(*bus*, ...)
Construct a UART object on the given bus. `bus` can be 1-6, or 'XA', 'XB', 'YA', or 'YB'. With no additional parameters, the UART object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

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)`

Methods

uart.init(*baudrate*, *bits=8*, *parity=None*, *stop=1*, ***, *timeout=1000*, *timeout_char=0*, *read_buf_len=64*)
Initialise the UART bus with the given parameters:

- `baudrate` is the clock rate.
- `bits` is the number of bits per character, 7, 8 or 9.
- `parity` is the parity, `None`, 0 (even) or 1 (odd).
- `stop` is the number of stop bits, 1 or 2.
- `timeout` is the timeout in milliseconds to wait for the first character.
- `timeout_char` is the timeout in milliseconds to wait between characters.
- `read_buf_len` is the character length of the read buffer (0 to disable).

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

uart.deinit()
Turn off the UART bus.

uart.any()
Return `True` if any characters waiting, else `False`.

uart.read([*nbytes*])
Read characters. If `nbytes` is specified then read at most that many bytes.

Note: for 9 bit characters each character takes two bytes, `nbytes` must be even, and the number of characters is `nbytes/2`.

Return value: a bytes object containing the bytes read in. Returns `b''` on timeout.

uart.readall()
Read as much data as possible.

Return value: a bytes object.

`uart.readchar()`
Receive a single character on the bus.

Return value: The character read, as an integer. Returns -1 on timeout.

`uart.readinto(`*buf*`[, `*nbytes*`])`
Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value: number of bytes read and stored into `buf`.

`uart.readline()`
Read a line, ending in a newline character.

Return value: the line read.

`uart.write(`*buf*`)`
Write the buffer of bytes to the bus. If characters are 7 or 8 bits wide then each byte is one character. If characters are 9 bits wide then two bytes are used for each character (little endian), and `buf` must contain an even number of bytes.

Return value: number of bytes written.

`uart.writechar(`*char*`)`
Write a single character on the bus. `char` is an integer to write. Return value: `None`.

class USB_VCP – USB virtual comm port

The USB_VCP class allows creation of an object representing the USB virtual comm port. It can be used to read and write data over USB to the connected host.

Constructors

class `pyb.USB_VCP`
Create a new USB_VCP object.

Methods

`usb_vcp.any()`
Return `True` if any characters waiting, else `False`.

`usb_vcp.close()`

`usb_vcp.read(`[*nbytes*]`)`

`usb_vcp.readall()`

`usb_vcp.readline()`

`usb_vcp.recv(`*data*, *, *timeout=5000*`)`
Receive data on the bus:

- `data` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `data` is an integer then a new buffer of the bytes received, otherwise the number of bytes read into `data` is returned.

usb_vcp.send(*data*, ***, *timeout=5000*)
Send data over the USB VCP:

- `data` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: number of bytes sent.

usb_vcp.write(*buf*)

network — network configuration

This module provides network drivers and routing configuration.

class CC3k

Constructors

class network.CC3k(*spi*, *pin_cs*, *pin_en*, *pin_irq*)
Initialise the CC3000 using the given SPI bus and pins and return a CC3k object.

Methods

cc3k.connect(*ssid*, *key=None*, ***, *security=WPA2*, *bssid=None*)

class WIZnet5k

This class allows you to control WIZnet5x00 Ethernet adaptors based on the W5200 and W5500 chipsets (only W5200 tested).

Example usage:

```
import wiznet5k
w = wiznet5k.WIZnet5k()
print(w.ipaddr())
w.gethostbyname('micropython.org')
s = w.socket()
s.connect(('192.168.0.2', 8080))
s.send('hello')
print(s.recv(10))
```

Constructors

class network.WIZnet5k(*spi*, *pin_cs*, *pin_rst*)
Create and return a WIZnet5k object.

Methods

`wiznet5k.ipaddr(`[*(ip, subnet, gateway, dns)*]`)`
Get/set IP address, subnet mask, gateway and DNS.

`wiznet5k.regs()`
Dump WIZnet5k registers.

Глава 5

Аппаратные средства pyboard

- PYBv1.0 schematics and layout (2.4MiB PDF)
- PYBv1.0 metric dimensions (360KiB PDF)
- PYBv1.0 imperial dimensions (360KiB PDF)

Глава 6

Спецификации компонент pyboard

- Микроконтроллер: STM32F405RGT6 (link to manufacturer's site)
- Акселерометр: Freescale MMA7660 (800kiB PDF)
- Стабилизатор напряжения LDO (малым падением напряжения вход/выход): Microchip MCP1802 (400kiB PDF)

Глава 7

Спецификации прочих компонент

- ЖК экран с наружным сенсорным слоем: Newhaven Display NHD-C12832A1Z-FSW-FBW-3V3 (460KiB PDF)
- Чип для сенсорной ЖК оболочки: Freescale MPR121 (280KiB PDF)
- Цифровой потенциометр для аудио оболочки: Microchip MCP4541 (2.7MiB PDF)

Глава 8

Лицензия Micro Python

The MIT License (MIT)

Copyright (c) 2013, 2014 Damien P. George, and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Глава 9

Содержание документации Micro Python

Глава 10

Indices and tables

- genindex
- modindex
- search

Содержание модулей Python

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z