

|  |  |  |
| --- | --- | --- |
| **Auteur :** | **Encadrant :** | **Année:** |
| Aymeric Maurice Vincent Desnos | Pierre Gaucher | DII3 |

Programmation microcontrôleur en Python

Tutoriel d’installation de l’IDE Python et gestion des ressources matérielles

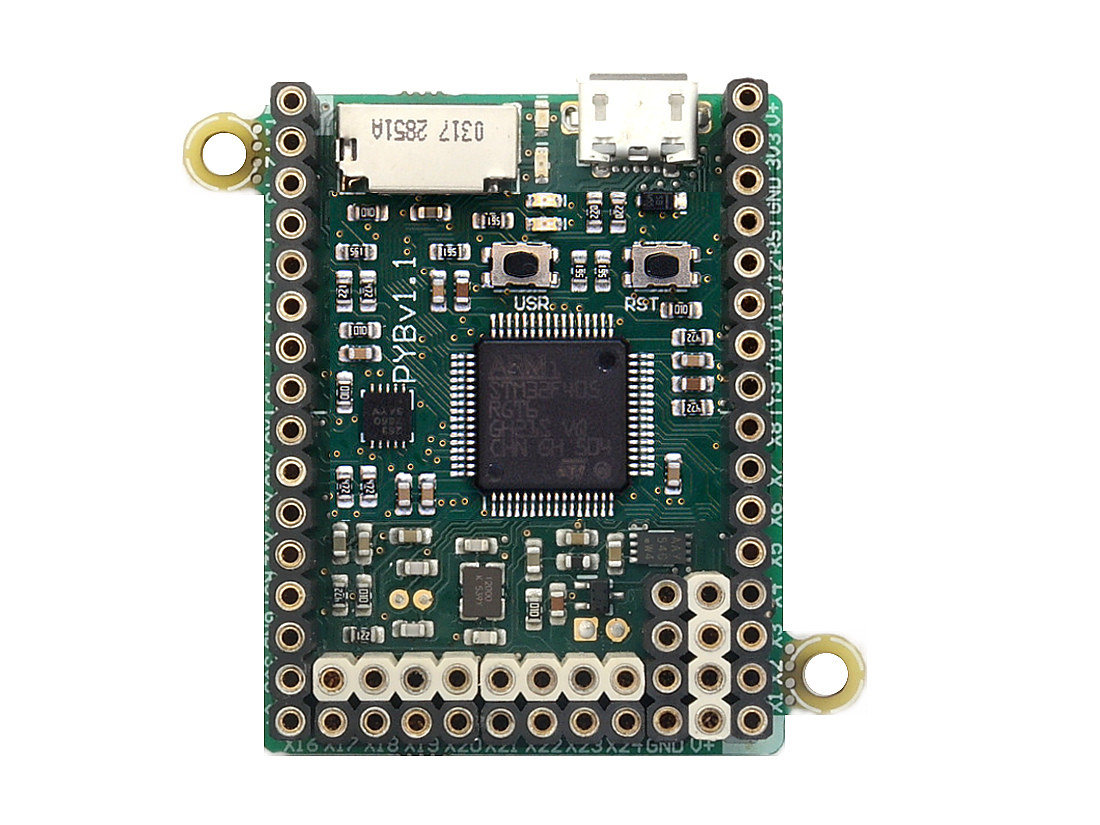


Table des matières

1. [Introduction 2](#_Toc507615007)

[Contexte](#_Toc507615008)

[Objectifs](#_Toc507615009)

[Caractéristiques de la Pyboard](#_Toc507615010)

1. [Installation de l’IDE de développement 2](#_Toc507615011)
2. [Gestion des ressources matérielles du microcontrôleur 2](#_Toc507615012)

[Système de fichiers local et carte SD](#_Toc507615013)

[Modes de démarrage](#_Toc507615014)

[Erreurs possibles](#_Toc507615015)

[Délai](#_Toc507615016)

[Composants internes](#_Toc507615017)

[Interruptions externes](#_Toc507615018)

[Les timers](#_Toc507615019)

[RTC ( real time clock)](#_Toc507615020)

[PWM ( pulse width modulation)](#_Toc507615021)

[DAC (digital vers analogique conversion)](#_Toc507615022)

[ADC (convertisseur analogique numérique)](#_Toc507615023)

[UART- bus de communication série en duplex](#_Toc507615024)

[Bus I2C](#_Toc507615025)

[Bus SPI](#_Toc507615026)

1. [Gestion des ressources périphériques du microcontrôleur 10](#_Toc507615027)

[Pins et GPIO](#_Toc507615028)

[Contrôle des moteurs](#_Toc507615029)

1. [Annexe 11](#_Toc507615030)

# Introduction

## Contexte

## Objectifs

## Caractéristiques de la Pyboard

Au niveau hardware, la carte Pyboard possède les caractéristiques suivantes :

* STM32F405RG microcontrôleur
* 168 MHz Cortex M4 CPU
* 1024KiB flash ROM et 192KiB RAM
* Un connecteur Micro USB pour l’alimenter et pour la communication série
* Un emplacement pour les cartes Micro SD
* Un accéléromètre 3 axes (MMA7660)
* Horloge temps réel (possibilité d’ajouter une batterie de secours)
* 24 GPIO , LED and switch GPIO available on bottom row
* 3x 12-bit convertisseur analogique vers numérique, disponible sur 16 pins
* 2x 12-bit convertisseur numérique vers analogique, disponible sur les pins X5 and X6
* 4 LEDs (rouge, vert, jaune et bleu)
* 1 bouton reset et 1 switch
* Régulateur de tension 3,3V, délivre jusqu’à 250mA, tension d’entrée allant de 3.6V à 16V
* DFU bootloader dans la ROM pour mettre facilement à jour le firmware

# Installation de l’IDE de développement

# Gestion des ressources matérielles du microcontrôleur

## Système de fichiers local et carte SD

On dispose d’un petit lecteur de fichier système dans la Pyboard appelé /flash, qui correspond à la « RAM » de la carte, si une carte micro SD est branché à la Pyboard alors la mémoire sera stocké dans /sd au lieu de /flash.

|  |  |
| --- | --- |
|  | Ne pas oublier d’éjecter la Pyboard de l’ordinateur avant de faire un reset |

## Modes de démarrage

Quand on allume la Pyboard ou qu’on appuie sur le bouton Reset, la carte démarre en mode « standard » :

Le fichier **boot.py** s’exécute puis lance à son tour **main.py**

On peut changer le mode de démarrage en maintenant le bouton utilisateur enfoncé pendant le démarrage de la carte, puis tout en maintenant ce bouton, maintenir enfoncé le bouton Reset. On voit alors les leds verte et orange compter en mode binaire, ce sont les différents modes de démarrage. Lorsque les voyants ont atteint le mode souhaité, relâchez le bouton :

1. LED verte uniquement, démarrage standard : exécutez boot.py ensuite main.py.
2. LED orange uniquement, démarrage sécurisé : n'exécutez aucun script au démarrage.
3. LED verte et orange ensemble, réinitialisation à son état d'usine, puis démarre en mode sans échec.

|  |  |
| --- | --- |
|  | Si vos fichiers sont corrompus démarrez le mode 3 |

## Erreurs possibles

1. LED rouges et vertes clignotent alternativement : un script Python a une erreur, Utilisez le REPL pour le déboguer.
2. les 4 voyants s'allument et s'éteignent lentement : erreur grave, débrancher et rebrancher la carte

## Délai

Une librairie time est mise à notre disposition pour faire des délais avec une précision réglable :

1. **import** time
3. time.sleep(1)           # délai de 1 seconde
4. time.sleep\_ms(500)      # délai de 500 millisecondes
5. time.sleep\_us(10)       # délai pour 10 microsecondes
6. start = time.ticks\_ms() # valeur du compteur de millisecondes
7. delta = time.ticks\_diff(time.ticks\_ms(), start) # valeur de la difference de temps entre le temps actuel et et le temps de « start »

## Composants internes

### Leds

Le constructeur pyb.LED permet de controler les leds internes à la carte :

1. **from** pyb **import** LED
3. led = LED(1) # 1=rouge, 2=vert, 3=jaune, 4=bleu
4. led.toggle() # changer etat
5. led.on() # allumer
6. led.off() #eteindre
8. # Les Led 3 et 4 supporte la gestion d'intensité par PWM (0-255)
9. LED(4).intensity()    # valeur de l'intensité
10. LED(4).intensity(128) # mettre la valeur de l'intensité à moitié

### Commutateurs

Le constructeur pyb.Switch permet de controler les boutons internes à la carte :

1. **from** pyb **import** Switch
3. btn = Switch() #instanciation
4. btn.value() # retourne Vrai ou Faux
5. btn.callback(**lambda**: pyb.LED(1).toggle()) # appelle la fonction, ici définie en paramètre

## Interruptions externes

Le constructeur pyb.ExtInt( Pin, Mode, Pull, Callback) permet de créer une interruption :

* Pin
  + On choisit le pin qui active l’interruption avec Pin(<monPin>)
* Mode
  + **ExtInt.IRQ\_FALLING :** interruption sur front montant
  + **ExtInt.IRQ\_RISING :** interruption sur front descendant
  + **ExtInt.IRQ\_RISING\_FALLING :** interruption sur front montant ou descendant
* Pull
  + **Pin.PULL\_NONE :** Pas de résistance de Pull up
  + **Pin.PULL\_UP** **:** Présence de résistance de Pull up
* Callback

Fonction qui est appelée

1. **from** pyb **import** Pin, ExtInt
3. callback = **lambda** e: **print**("intr") # définition de la fonction e dans callback
4. ext = ExtInt(Pin('Y1'), ExtInt.IRQ\_RISING, Pin.PULL\_NONE, callback) # chaque fois qu'on détecte un front montant, la fonction callback est appelé

|  |  |
| --- | --- |
|  | Attention à l’effet « bounce » pour certain bouton poussoir : lorsqu’on appui dessus ils génèrent parfois plusieurs interruption. |

## Les timers

Le constructeur pyb.Timer( id,..) permet de créer un timer :

### .init(…)

Voici quelques paramètres utiles que l’on peut spécifier pour cette méthode:

* freq : permet de donner la fréquence du timer
* prescaler : permet de diviser par (prescaler + 1) l’horloge source du timer. Les horloges sources sont différentes selon les timers :
  + Timers 2-7 et 12-14 → 84 MHz
  + Timers 1 et 8-11 → 168 MHz
* mode :
  + Timer.UP : compte de 0 à AutoReload Register (ARR)
  + Timer.DOWN : compte de ARR à 0
  + Timer.CENTER : compte de 0 à ARR puis de ARR à 0

### .deinit()

Désactive le timer

### .callback(fun)

Fonction à appeler à chaque fois que le compteur s’incrémente

### .counter([valeur])

Récupérer ou assigner la valeur du compteur

### .freq([valeur])

### .period([valeur])

### .prescaler([valeur])

Récupérer ou assigner la valeur du pré-diviseur

### .source\_freq([valeur])

Récupérer ou assigner la valeur de la fréquence source du Timer

## RTC ( real time clock)

Le constructeur pyb.RTC( ...) permet de paramétrer et démarer une horloge à une date précise :

### .datetime([year, month, day, weekday, hours, minutes, seconds, subseconds ])

Récupérer ou assigner la valeur de la date

1. **from** pyb **import** RTC
3. horaire = RTC()
4. horaire.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) # assigner une date et une heure
5. horaire.datetime() # Avoir les information sur l'heure et la date

## PWM ( pulse width modulation)

Avec le constructeur pyb.Timer et pyb.Pin, on peut générer une PWM sur certains Pin de la carte Python:

### .channel( channel, mode, ...)

Voici les paramètres principaux de la méthode channel:

#### mode

* Timer.PWM : configure le Timer en mode PWM (active à l’état haut)
* Timer.PWM\_INVERTED : configure le Timer en mode PWM (active à l’état bas)

#### pin

Choisir le pin sur lequel on souhaite activer la PWM

#### Pulse\_width

Choisir la largeur d’impulsion (fréquence) de la PWM

#### Pulse\_width\_percent

Choisir le rapport cyclique (le pulse)

### Exemples:

1. **from** pyb **import** Pin, Timer
3. p = Pin('X1') # le pin X1 possède le timer TIM2,et la channel CH1
4. tim = Timer( 2, freq=1000)
5. ch = tim.channel(1, Timer.PWM, pin=p)
6. ch.pulse\_width\_percent(50)

## DAC (digital vers analogique conversion)

Le constructeur pyb.DAC permet d’ecrire des valeurs analogiques sur un pin donné

Voici les paramètres principaux du constructeur:

#### Pin

On peut mettre soit un objet pin ou bien 1 ou 2. Sachant que 1 correspond au pin ‘X5’ et 2 correspond au pin ‘X6’

#### Bits

La résolution du convertisseur qui peut être 8 ou 12 bits

### .noise(freq)

Permet de générer un signal de bruit aléatoire. Aussi un échantillon aléatoire est créé à cette fréquence.

### .write(valeur)

Ecrire une valeur à la sortie du DAC comprise entre 0 et 2n-1 bits (spécifié dans le paramètres bits )

Exemple: Générer une sinusoïdale avec une résolution de 12 bits

1. **import** math
2. **from** array **import** array
3. **from** pyb **import** DAC
5. # creation d'un buffer qui contient la forme sinosoïdale
6. buf = array('H', 2048 + int(2047 \* math.sin(2 \* math.pi \* i / 128)) **for** i **in** range(128))
8. # sortie de la sinusoïdale à 400 Hz
9. CAN = DAC(1, bits=12)
10. CAN.write\_timed(buf, 400 \* len(buf), mode=DAC.CIRCULAR)

## ADC (convertisseur analogique numérique)

Le constructeur pyb.ADC permet de lire des valeurs analogiques sur un pin donné. Il fait donc l’inverse.

Voici les paramètres principaux du constructeur:

#### Pin

Les pin possibles sont disponibles en annexe.

### .read()

Lit la valeur analogique sur le pin sélectionné et retourne une valeur numérique comprise entre 0 et 4095.

### .read\_timed(buf, timer)

Cette fonction permet de stocker dans un buffer (buf) les valeurs lut par le ADC. Le nombre d’échantillons est égale à la taille du buffer.

### Exemple:

1. convertisseur = pyb.ADC(pyb.Pin.board.X19)    # creation du convertisseur sur le pin X19
2. tim = pyb.Timer(6, freq=10)         # creation du timer à 10Hz
3. buf = bytearray(100)                # creation du buffer pour stocker 100 échantillons
4. convertisseur.read\_timed(buf, tim)            # il y a 100 valeur à stocker à 10Hz, dans 10s

## UART- bus de communication série en duplex

Le constructeur **pyb.UART**, implémente les 2 standards UART et USART protocole. Physiquement il y a seulement 2 fils: RX et TX. Chaque unité de communication sont des caractère d’une taille de 7, 8 ou 9.

UART(4) est XA: (TX, RX) = (X1, X2) = (PA0, PA1)

UART(1) est XB: (TX, RX) = (X9, X10) = (PB6, PB7)

UART(6) est YA: (TX, RX) = (Y1, Y2) = (PC6, PC7)

UART(3) est YB: (TX, RX) = (Y9, Y10) = (PB10, PB11)

UART(2) est : (TX, RX) = (X3, X4) = (PA2, PA3)

Voici les paramètres principaux du constructeur:

#### Bus

Peut-être 1-6 ou XA, XB, YA, YB

### .init(baudrate, bits=8, parity=None, stop=1, \*, timeout=1000, flow=0, timeout\_char=0, read\_buf\_len=64)

#### Baudrate

#### Bits

Nombre de bits par caractère, peut être 7, 8, ou 9

#### Parity (0 even, 1 odd)

#### Stop (1 ou 2 bits de stop)

#### Flow

Définir le contrôle de flux: UART.RTS ou UART.CTS

#### Timeout

Temps en millisecondes à attendre avant d’écrire/lire le premier caractère

#### Timeout\_char

Temps en millisecondes à attendre avant d’écrire/lire chaques caractères

#### Read\_buf\_len

Taille du buffer de lecture

### .any()

Retourne le nombre d’ octets en attente

### .read([noctets])

Si “noctets” est spécifié alors si ce nombre est disponible sur buffer, il les octets sont immédiatement lut sinon la method attend qu’ils soient disponible.

Si rien n’est donné pour “noctets”, alors la method lit autant données que possible.

|  |  |
| --- | --- |
|  | Pour des caractère de taille 9 bits, chaques caractères occuperont un espace de 2 octets |

### .readchar()

Cette méthode lit et retourne 1 caractère ou -1 si elle ne peut rien lire.

|  |  |
| --- | --- |
|  | Le caractère retourné est de type **int** |

### .write(buf)

Écrit sur le bus le buffer d’octets voulu. Pour des caractères de taille 7 ou 8 chaques octets représente 1 caractère.

### .writechar(char)

Écrit un seul caractère sur le bus.

### Exemple

#### Initialisation

1. **from** pyb **import** UART
3. uart = UART(1, 9600)                         # initialisation du bus et de la vitesse du bus
4. uart.init(9600, bits=8, parity=None, stop=1) # initialisation d'autres paramètres

#### Lecture

1. uart.read(10)       # lit 10 caractère et les retournent sous forme d'octets
2. uart.read()         # lit tous les caractère disponible
3. uart.readinto(buf)  # lit et stock les données dans 'buf'
4. uart.write('abc')   # lit 3 caractère
6. uart.readchar()     # lit 1 caractère et le retourne sous forme d'un entier
7. uart.writechar(42)  # ecrit 1 caractère

## Bus I2C

Protocol de communication entre 2 appareils, 2 cable de connexion : SCL, c’est l’horloge du bus, et SDA, pour le données. Le constructeur est pyb.I2C.

Voici les paramètres principaux du constructeur:

#### Bus

Construire un objet I2C sur le bus donné. Bus peut être 1 ou 2, 'X' ou 'Y'. Sans paramètre supplémentaire, l'objet I2C est créé mais pas initialisé (il possède les paramètres de la dernière initialisation du bus,s’ils existe). Si des arguments supplémentaires sont donnés, le bus est initialisé. Voir **.init** pour les paramètres d'initialisation.

Les broches physiques des bus I2C sur Pyboards V1.0 et V1.1 sont:

I2C(1) est sur la position X: (SCL, SDA) = (X9, X10) = (PB6, PB7)

I2C(2) est sur la position Y: (SCL, SDA) = (Y9, Y10) = (PB10, PB11)

### .init(mode , \* , addr = 0x12 , débit binaire = 400000 , gencall = faux , dma = faux )

On peut initialiser le bus avec les paramètres suivant:

#### Mode

Doit être **I2C.MASTER** ou **I2C.SLAVE**

#### Addr

Adresse de l’esclave sur 7 bits

#### baudrate

Vitesse de l’horloge SCL

### .is\_ready(addr)

Vérifiez si un périphérique I2C répond à l'adresse donnée. Valable uniquement en mode maître.

### .mem\_write(data , addr , memaddr , \* , timeout = 5000 , addr\_size = 8 )

Data

 Peut être un nombre entier ou un tampon pour écrire à partir de

#### Addr est l'adresse de l'appareil I2C

Memaddr

Emplacement de la mémoire dans le dispositif I2C

timeout

Délai d'attente en millisecondes pour faire l'écriture

addr\_size

Sélectionne la taille de memaddr: 8 ou 16 bits

### .recv(recv, addr, timeout)

Recevoir des données sur le bus:

Recv

Peut être un entier, qui est le nombre d'octets à recevoir, ou un buffer(tampon) mutable, qui sera rempli avec des octets reçus

Addr

L'adresse à recevoir de (uniquement requise en mode maître)

Timeout

Délai d'attente en millisecondes pour attendre la réception

### .send(send, addr, timeout)

Send

Peut être un entier, ou un buffer(tampon) mutable.

### .scan()

Analyser les adresses I2C disponible

## Bus SPI

# Gestion des ressources périphériques du microcontrôleur

## Pins et GPIO

La gestion des pins et GPIO se fait de manière très simple via le constructeur *Pin.*

Voici les paramètres principaux du constructeur:

## Contrôle des moteurs

### Servomoteurs

### Moteurs DC

### Moteurs pas à pas

# Annexe

