

RESUMEN

Este trabajo tiene como objetivo principal el análisis de un hardware comercial recientemente aparecido en el mercado, denominado Pyboard v.1.0, que se basa en un microcontrolador ARM de 32 bits y una versión reducida y eficiente de Python, denominado Micro Python. A partir de este TFG, los profesores de la ETSEIB podrán decidir si este hardware y software es adecuado para la enseñanza de microcontroladores y sus aplicaciones en distintas asignaturas impartidas en la escuela.

El producto resultante de este trabajo también debe ser la síntesis en forma de tutorial, de toda una experiencia vivida como alumno de la ETSEIB y para un usuario del mismo tipo.

El método que se ha utilizado en el análisis de este trabajo comienza con seguir los primeros pasos indicados en el tutorial de Micro Python. Una vez acabada esta pequeña introducción, se pasa a elaborar ejemplos propios, que hagan uso del hardware diverso de la placa, y comprobar su funcionamiento y eficiencia. Se han probado los temporizadores, interrupciones, los dos pulsadores, la UART, el bus CAN y el USB, así como el acelerómetro y los LEDs de la placa.

Se analizan, por lo tanto, factores como la sencillez de uso. Se comprueba el funcionamiento de los comandos y se anotan los errores que se puedan encontrar. También se investiga en funcionalidades poco explicadas y definidas, a través de una documentación que, en muchos casos, es difícil de digerir, además de estar escrita totalmente en inglés.

Los resultados obtenidos son satisfactorios hasta cierto punto. La sencillez característica de Python contrasta con cierta lentitud en algunos procesos, pudiendo llegar a ser tres veces más lento el intérprete ejecutando Python que ensamblador, en la ejecución de instrucciones sencillas. Se han encontrado errores en el firmware, algunos de ellos se han podido solucionar y otros no, como por ejemplo la recepción de mensajes a través del bus CAN.

También se ha indagado sin mucho éxito en la programación del firmware, buscando una manera de añadirle funciones compiladas en C, para después ejecutarlas desde el intérprete de Python. Aunque no se logra ninguna utilidad determinada, sí que se abre la senda en ese sentido.

SUMARIO

RESUMEN.....	1
SUMARIO.....	3
1. PREFACIO	9
1.1. Origen y motivación del proyecto	9
2. INTRODUCCIÓN.....	9
2.1. Objetivos del proyecto	9
2.2. Antecedentes y estado del arte	9
2.3. Alcance del proyecto	10
3. MICRO PYTHON.....	11
4. EJEMPLOS DE LA DOCUMENTACIÓN OFICIAL	12
4.1. Instalación de drivers	12
4.1.1. Mass Storage Class (MSC).....	12
4.1.2. Communications Device Class (CDC).....	13
4.1.3. Human Interface Device (HID)	15
4.1.4. Device Software Upgrade (DFU).....	15
4.2. Ejemplos del tutorial	19
4.2.1. Primeros pasos	19
4.2.2. La “REPL prompt”	19
4.2.3. LEDs.....	21
4.2.4. El pulsador SWITCH.....	23
4.2.5. Acelerómetro	23
4.2.6. Modo seguro y restablecimiento de fábrica	25
4.2.7. Modo HID: mouse	26
4.2.8. Temporizadores	28
4.2.9. Ensamblador.....	30
4.3 Conclusiones.....	31
5. REGISTRADOR DE DATOS	33
5.1. Guardar datos del acelerómetro.....	34
5.2. Ejecución del programa.....	36
5.3. Conclusiones.....	37
6. DEPURADOR	39
6.1. Contador de microsegundos	39
6.2. Construcción del contador de microsegundos.....	40

6.2.1.	Ecuación de prescaler, period y freq	40
6.2.2.	Características de los temporizadores	41
6.2.3.	Comprobación de los datos del fabricante	42
6.2.4.	Comprobación de frecuencia mínima	43
6.2.5.	Comprobación de frecuencia máxima	44
6.3.	Construcción del depurador	44
6.4.	Funciones a comparar.....	45
6.5.	Ejecución del programa.....	46
6.6.	Conclusiones.....	46
7.	SALIDAS Y ENTRADAS ANALÓGICAS	48
7.1.	Generador de señales.....	48
7.2.	Puente virtual entre entrada y salida	51
7.3.	Puente virtual con desfase	54
7.4.	Conclusiones.....	55
8.	FRECUENCIA DE ITERACIÓN DE PROCESOS.....	56
8.1.	Alternativa 1	56
8.2.	Alternativa 2	57
8.2.1.	Ejemplo práctico	58
8.3.	Conclusiones.....	61
9.	BUS CAN	62
9.1.	Montaje con transceptores	62
9.2.	Pruebas con Kvaser Canking	64
9.3.	Conclusiones.....	66
10.	APLICACIÓN ACELERÓMETRO.....	67
10.1.	Demo USB	67
10.2.	Demo UART	69
10.3.	Demo PC.....	70
10.4.	Conclusiones.....	72
11.	COMPILAR FUNCIONES EN C.....	73
11.1.	Micro-Ctypes	73
11.2.	Modificación del firmware	73
11.2.1.	Compilación del código fuente	73
11.2.2.	Incorporación de una función propia al firmware	74
11.3.	CONCLUSIONES.....	75
	CONCLUSIONES.....	77



AGRADECIMIENTOS	79
BIBLIOGRAFÍA.....	81
Referencias bibliográficas	81
ANEXO	83
A1. Código: bucle en ensamblador.....	83
A2. Código: find_prescaler_period.....	84
A3. Código: find_freq	85
A4. Código: depurador.....	85
A5. Código: demo acelerómetro PC	86
A6. Presupuesto	88

ÍNDICE DE ILUSTRACIONES

Fig. 3.1: Comando de cambio de modo a MSC.....	12
Fig. 3.2: Pyboard como dispositivo no reconocido.....	13
Fig. 3.3: Instalación automática o manual.	14
Fig. 3.4: Alerta de fallo de reconocimiento de drivers.	14
Fig. 3.5: Pyboard reconocida como dispositivo serial.	15
Fig. 3.6: Comando de cambio de modo HID.....	15
Fig. 3.7: Conexión de pin BOOT a 3,3V.	16
Fig. 3.8: Dispositivo no reconocido (Pyboard Virtual Comm Port in FS Mode).	16
Fig. 3.9: Dispositivo reconocido en modo DFU.....	17
Fig. 3.10: Interfaz DfuSe con pasos de actualización.	18
Fig. 3.11: Comandos para importar el módulo PYB y encender un LED.	19
Fig. 3.12: Interfaz de PuTTY 20	
Fig. 3.13: Comandos probados en el terminal PuTTY.	21
Fig. 3.14: Soft Reset de la Pyboard (recorte de la imagen del terminal).	21
Fig. 3.15: Comandos de asignación, encendido y apagado del LED1.....	21
Fig. 3.16: Bucle de alternación de encendido del LED3 21	
Fig. 3.17: Programación secuencial de LEDs..... 22	
Fig. 3.18: Programación secuencial de LEDs (estructura de intento y finalización).	22
Fig. 3.19: Bucle de variación de la luminosidad del LED4.	22
Fig. 3.20: Creación del objeto pulsador USR y comprobación de estado.	23
Fig. 3.21: Asignación del LED2 al pulsador USR mediante lambda.	23
Fig. 3.22: Asignación de una función al pulsador USR..... 23	
Fig. 3.23: Creación objeto acelerómetro y consulta al eje de coordenadas X..... 24	
Fig. 3.24: Programa de encendido de LED para un límite de inclinación..... 24	
Fig. 3.25: Encendido de cada LED acorde a la dirección de la inclinación.	24
Fig. 3.26: Código del ejemplo "Spirit level" 25	
Fig. 3.27: Código BOOT.py para configuración en modo HID.	26
Fig. 3.28: Pyboard no reconocida por Windows.	27
Fig. 3.29: Programa de control del mouse (Linux): movimiento circular..... 28	
Fig. 3.30: Programa de control del mouse: movimiento ligado al acelerómetro.	28
Fig. 3.31: Creación de un temporizador y atributos de la clase Timer.	28
Fig. 3.32: Atributos por defecto del temporizador en el Tutorial oficial..... 29	
Fig. 3.33: Interruptor del temporizador 29	
Fig. 3.34: Dos interrupciones simultaneas de temporizador 29	
Fig. 3.35: Temporizador de microsegundos 29	
Fig. 3.36: Función de retorno de valor (Ensamblador)..... 30	



Fig. 3.37: Comprobación de función de retorno de valor (Ensamblador).....	30
Fig. 3.38: Programa de encendido del LED1 (Ensamblador).....	31
Fig. 3.39: Operación suma (Ensamblador).....	31
Fig. 3.40: Comprobación en terminal de la función “flash_led()” (Ensamblador).....	31
Fig. 4.1 Comandos de construcción, escritura y guardado de archivo (Python).	33
Fig. 4.2: Creación de un archivo en la tarjeta SD.	33
Fig. 4.3: Creación de un archivo en la memoria FLASH.....	33
Fig. 4.4: Lectura y escritura en serie del acelerómetro en un archivo.....	34
Fig. 4.5: Programa de muestreo de acelerómetro no viable.	34
Fig. 4.6: Programa de muestreo de acelerómetro viable.	35
Fig. 4.7: Programa completo de captación y escritura del acelerómetro.....	35
Fig. 4.8: Función de cambio de estado booleano no viable.	36
Fig. 4.9: Hoja de cálculo con datos del acelerómetro	37
Fig. 5.1: Fórmula para calcular la frecuencia de un temporizador.	40
Fig. 5.2: Tabla de propiedades de temporizadores según el fabricante.....	41
Fig. 5.3: OverflowError en terminal PuTTY.....	42
Fig. 5.4: Inicialización de temporizador con valores fuera del límite.	43
Fig. 5.5: Inicialización del temporizador 2 a mínima frecuencia.	43
Fig. 5.6: Iteración de encendido de LED 1000 veces (Micro Python).	45
Fig. 5.7: Iteración de encendido de LED 1000 veces (Micro Python+Ensamblador).....	45
Fig. 6.1: Creación objeto DAC y definición de valor en la salida.....	48
Fig. 6.2: Método del objeto DAC para señales frecuenciales.	49
Fig. 6.3: Fórmula para el muestreo de una señal sinusoidal de salida.	49
Fig. 6.4: Código para el muestreo de una señal sinusoidal de salida.	49
Fig. 6.5: Código de la clase sineGenerator: generador de salida sinusoidal.....	50
Fig. 6.6: Señal captada con el osciloscopio.....	50
Fig. 6.7: Pyboard, osciloscopio y generador de señales conectados en una protoboard.	51
Fig. 6.8: Comando de cambio de valor de entrada a valor de salida poco eficiente.....	51
Fig. 6.9: Comando de cambio de valor de entrada a valor de salida eficiente.	51
Fig. 6.10: Señales superpuestas (adc_dac_v1). Muestreo a 10khz y 20kHz respectivamente.....	52
Fig. 6.11: Programa de puente “entrada-salida analógicas” con objeto clase (adc_dac_v1).	53
Fig. 6.12: Señales superpuestas (adc_dac_v2). Muestreo a 10khz y 20kHz respectivamente.....	53
Fig. 6.13: Programa de puente entrada-salida analógicas código directo (adc_dac_v2).	53
Fig. 6.14: Cola circular.	54
Fig. 6.15: Programa de puente entre salida y entrada con desfase.....	55
Fig. 7.1: Depurador para proceso iterativo con y sin interrupciones.	57
Fig. 7.2: Diagrama temporal del programa de iteración a frecuencia constante.	58

Fig. 7.3: Código simplificado programa de iteración a frecuencia constante.....	58
Fig. 7.4: Código de comprobación para el programa de iteración a frecuencia constante.	59
Fig. 7.5: Código para la comprobación del tiempo que dura el proceso principal.	60
Fig. 7.6: Captura de pantalla de la ejecución del programa y valor del programa.....	61
Fig. 8.1: Izquierda: transceptor con placa y acoplador. Derecha: Esquema del transceptor.	62
Fig. 8.2: Conexión entre los buses CAN1 y CAN2.....	63
Fig. 8.3: Comandos de creación de objeto CAN, envío y recepción de datos.....	63
Fig. 8.4: Comandos de comunicación entre CAN1 y CAN2.....	63
Fig. 8.5: Conexión de bus CAN entre placa y puerto serial del PC.....	64
Fig. 8.6: Interfaz Kvaser CanKing.....	65
Fig. 8.7: Programa de ejemplo de envío de datos por el bus CAN.	65
Fig. 9.1: Programa de ejemplo de envío de datos vía USB.	68
Fig. 9.2: Entorno Python para leer datos recibidos desde el Pyboard.....	68
Fig. 9.3: Programa de envío de datos para la demostración del acelerómetro.	69
Fig. 9.4: Montaje de la Pyboard con adaptador UART-USB.....	69
Fig. 9.5: Captura de pantalla de la demostración con el acelerómetro.	71
Fig. 9.6: Ecuaciones de la cinemática de una partícula.....	71
Fig. 9.7: Ecuaciones discretas de la cinemática de una partícula.....	72
Fig. 10.1: Comandos de preparación de Linux y compilación del código fuente.....	74
Fig. 10.2: Declaración de objeto GOC en goc.c	74
Fig. 10.3: Adición de goc.c al archivo Makefile.....	75
Fig. 10.4: Asignación del nombre GOC.....	75



1. PREFACIO

1.1. ORIGEN Y MOTIVACIÓN DEL PROYECTO

Este proyecto se escogió por interés personal en los temas relacionados con Micro Python como son la robótica, la electrónica y el control automático. Por lo tanto, este proyecto brinda la oportunidad al autor de sumergirse por primera vez en un mundo con infinitud de posibilidades tanto laborales como pedagógicas o lúdicas. Además de ser un importante aspecto en la ingeniería.

La historia pragmática de la ingeniería en el capitalismo busca el beneficio mejorando el rendimiento de máquinas y mecanismos u ofreciendo nuevas comodidades y productos que solo la más desarrollada tecnología puede producir. Por contra, la motivación de este proyecto se basa en el más puro sentimiento creativo que la automática y la electrónica pueden ofrecer.

2. INTRODUCCIÓN

2.1. OBJETIVOS DEL PROYECTO

El objetivo de este proyecto es ofrecer un análisis que ayude a conocer las posibilidades de la placa Pyboard v.1.0 y determinar si es viable o no como material didáctico en las aulas de la ETSEIB, en la UPC. Además, también se tiene como objetivo elaborar un tutorial que se base en toda la experiencia vivida y que sea de utilidad, primordialmente, al alumnado de dicha escuela.

2.2. ANTECEDENTES Y ESTADO DEL ARTE

La comercialización de microcontroladores para su uso en diferentes tipos de proyectos comenzó en los años 1970 con los conocidos PIC. Estos microcontroladores continúan utilizándose aunque están perdiendo popularidad frente a los nuevos microcontroladores ARM Cortex.

No ha sido hasta el desarrollo de Arduino y Raspberry-pi, en los últimos años, cuando se ha dirigido este tipo de herramientas a un público con menor experiencia y conocimiento. La placa de desarrollo Arduino debe su fama al hecho de ser *Open Source*, siendo ésta más barata que los PIC. El lenguaje de programación que utiliza es C y va dirigida tanto al uso pedagógico, como lúdico o en proyectos experimentales en organizaciones de más

prestigio. Es una herramienta flexible y potente y, hasta la fecha, es de las más extendidas.

En su lugar, Raspberry-Pi, que no es un microcontrolador sino un microcomputador, requiere la instalación de un sistema operativo, lo que la convierte en una herramienta más compleja que Arduino. Raspberry-Pi puede ser programado tanto en Python como en C. En muchos casos se utiliza con fines pedagógicos, haciendo honor a los motivos de su creación.

Micro Python, por su parte, surge pocos años después como proyecto Kickstarter, de la mano de Damien George. Un proyecto que sobrepasó con creces su objetivo de financiación. La idea inicial del proyecto propone rediseñar Python para crear Micro Python, una versión adaptada y mejorada para microcontroladores. Por lo tanto, se ejecuta el intérprete de Micro Python en un microcontrolador.

Los objetivos esenciales de este proyecto son conseguir un microcontrolador “más potente que Arduino y más sencillo que Raspberry-pi”, además de ser barato y pequeño. El proyecto de Micro Python está en constante desarrollo aunque ya se distribuye a un precio razonable.

2.3. ALCANCE DEL PROYECTO

El alcance de este proyecto cubre la documentación, experimentación y la utilización de algunas de las características que ofrece la Pyboard:

- Ejecución de código y escritura de programas.
- Creación de archivos de datos desde el entorno de Micro Python.
- Caracterización de los temporizadores y medición del tiempo.
- Comparación de Micro Python con lenguaje ensamblador.
- Precisión, fiabilidad y velocidad de captura del acelerómetro.
- Velocidad de muestreo, precisión y utilización de las entradas y salidas analógicas.
- Análisis del malfuncionamiento del bus CAN.
- Análisis de velocidad de la UART y del USB.
- Introducción a la programación en C de la Pyboard.



3. Micro Python

A continuación se listan brevemente las características esenciales de la placa Micro Python v.1.0.

- Microcontrolador STM32F405RGT6.
- Procesador ARM Cortex-M4 Core.
- Micro SD slot.
- Memoria flash (95 kB).
- Pulsador de usuario (USR).
- Pulsador de *reset* (RST).
- USB micro.
- 4 LEDs: rojo, naranja, verde y azul.
- 16x ADC (4 shielded)
- 2x DAC
- 6x UART
- 2x CAN
- 2x SPI
- 2x I2C
- 14 temporizadores
- Acelerómetro: Freescale MMA7660

4. EJEMPLOS DE LA DOCUMENTACIÓN OFICIAL

Los primeros pasos para introducirse en el uso de la placa de Micro Python (Pyboard) deben ser sencillos. La página oficial de Micro Python cuenta con un breve tutorial para ponerse en marcha, explicando los pasos más necesarios y básicos. Este tutorial está disponible únicamente en inglés. También hay un documento dedicado a la instalación de drivers y actualización de software de la Pyboard para Windows. Uno de los inconvenientes de la página oficial de Micro Python es que continuamente cambia no solo de contenido sino también de estructura.

La idea en este capítulo es comprobar que todos los pasos funcionan tanto en Windows como en Linux y familiarizarse con la Pyboard. Por eso, cada apartado de este punto corresponde a un apartado del tutorial y del manual de instalación. El primer punto del tutorial se obviará, ya que no es más que una serie de recomendaciones técnicas de alimentación y cuidados de la Pyboard.

Cuando se adquiere la placa Pyboard, ésta viene sin tarjeta SD y con una serie de archivos guardados en la memoria Flash del microcontrolador, como son:

- Boot.py
- Main.py
- README.py
- Pyboard.inf

4.1. INSTALACIÓN DE DRIVERS

La placa tiene cuatro modos de utilización que vienen explicados en el manual de instalación. A continuación se explicará brevemente cada modo y los pasos necesarios que se han seguido para la puesta en marcha de algunos de ellos en Windows. Linux no ha requerido ningún procedimiento para la actualización de drivers.

4.1.1. Mass Storage Class (MSC)

Éste es el modo de almacenamiento masivo, dado que la Pyboard es reconocida como si de un lápiz de memoria se tratase. No necesita instalación de drivers. Para cambiar a modo MSC se utiliza el comando de la Fig. 4.1. Este comando viene por defecto escrito en el archivo “boot.py”.

```
pyb.usb_mode('CDC+MSC')
```

Fig. 4.1: Comando de cambio de modo a MSC



4.1.2. Communications Device Class (CDC)

Este modo permite la comunicación con el microcontrolador desde un puerto serie del PC. A diferencia del modo anterior, éste sí que necesita instalación de drivers. Estos drivers se encuentran en la memoria flash de la placa con el nombre “pybcdcd.inf”. En Linux la instalación es automática. Los pasos necesarios en el sistema operativo Windows se explican a continuación.

Al conectar la placa vía USB se intentarán instalar los drivers automáticamente. Se ha ignorado este mensaje de Windows, ya que la instalación debe ser manual, tal y como lo recomienda el fabricante.

Se ha abierto el *Panel de Control*>*Hardware*>*Administrar Dispositivos*. Seguidamente, se ha hecho un clic derecho sobre el dispositivo no reconocido “*Pyboard Virtual Comm Port in FS Mode*” y se ha seleccionado “*actualizar software de dispositivo*” en la pestaña emergente (Fig. 4.2).

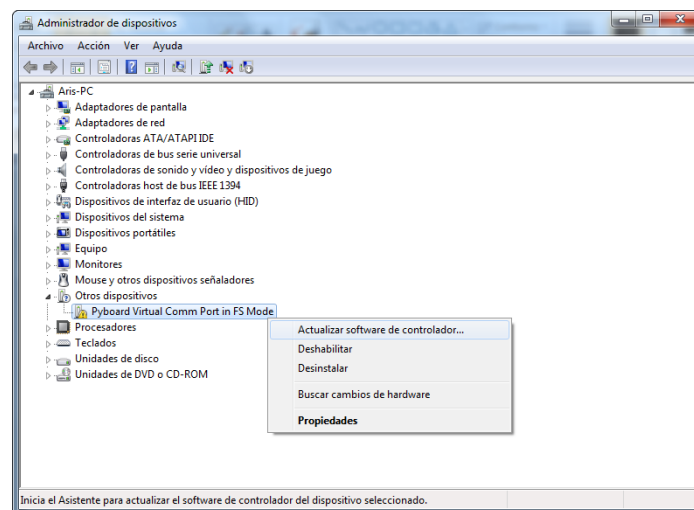


Fig. 4.2: Pyboard como dispositivo no reconocido

Tal y como se explica en el manual oficial, se debe actualizar “con software en el equipo” y buscar la letra correspondiente a la Pyboard (Fig. 4.3).

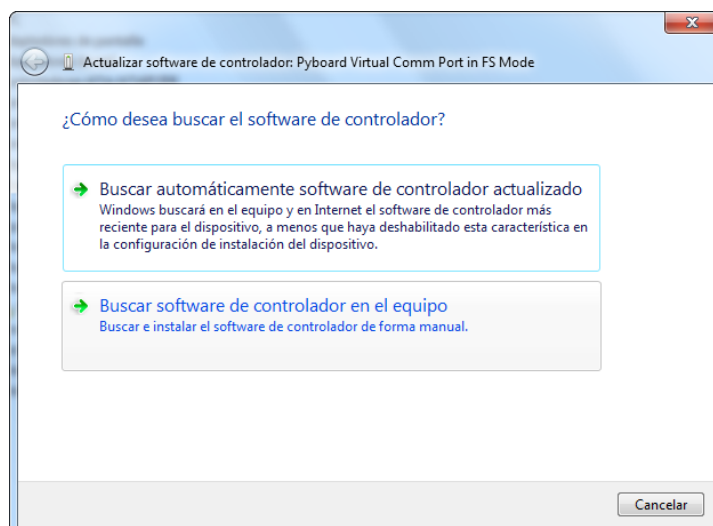


Fig. 4.3: Instalación automática o manual.

El manual oficial explica que hasta la fecha Windows no ha verificado el archivo "pybcdcd.inf", por lo que emergerá una ventana de alerta (Fig. 4.4). El archivo no contiene ningún driver, solo direcciona la Pyboard hacia drivers ya existentes en Windows, por lo que es seguro instalarlo.

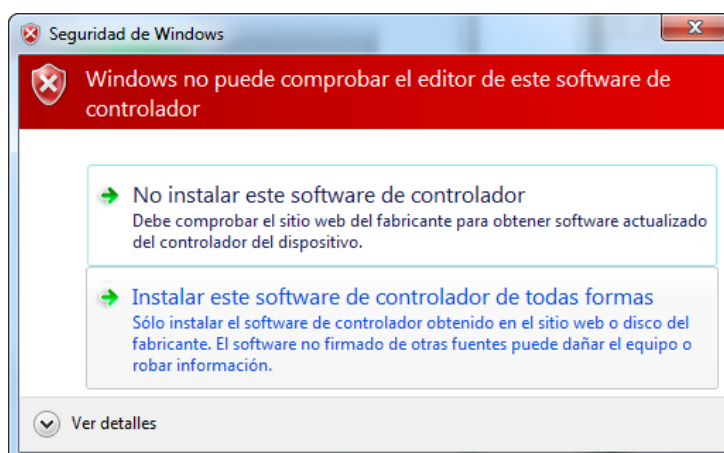


Fig. 4.4: Alerta de fallo de reconocimiento de drivers.

Una vez se han instalado los drivers, se ha anotado el número del puerto COMM asignado. En este caso se trata del puerto "COM3" (Fig. 4.5).



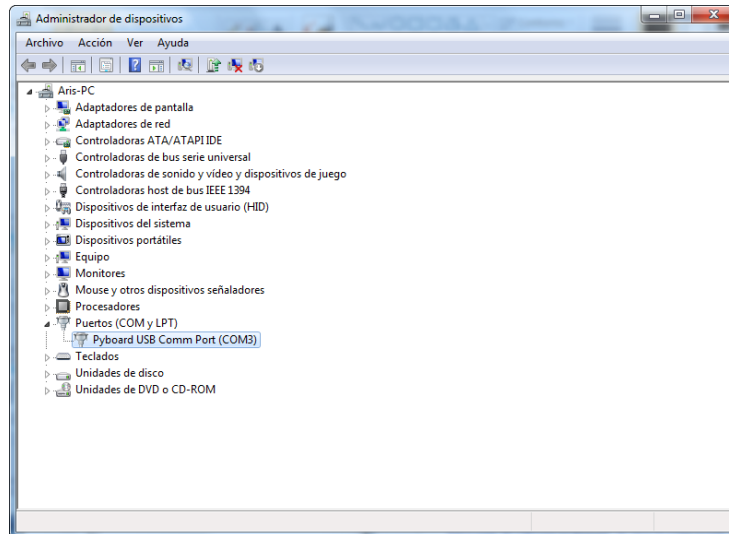


Fig. 4.5: Pyboard reconocida como dispositivo serial.

4.1.3. Human Interface Device (HID)

Este modo sirve para utilizar la Pyboard como un periférico de tipo mouse o teclado. No necesita instalación de drivers pero no viene configurado por defecto. Para cambiar a modo HID se puede usar el comando de la Fig. 4.6. Éste se substituye por el comando de la Fig. 4.1 en el archivo *boot.py*. Para que el cambio surta efecto se debe desconectar y conectar la Pyboard.

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

Fig. 4.6: Comando de cambio de modo HID.

4.1.4. Device Software Upgrade (DFU)

El modo DFU sirve para actualizar el firmware de la placa. Este modo requiere instalación de drivers y además solo está disponible conectando el pin BOOT a 3,3V. La Fig. 4.7 muestra como se ha realizado la conexión.



Fig. 4.7: Conexión de pin BOOT a 3,3V.

Se ha conectado la placa al PC después de hacer un puente del pin BOOT a 3,3V. Esta acción provoca la ejecución de un programa tipo “bootloader” almacenado en la memoria Flash ROM del microcontrolador. De esta manera, siempre será posible iniciar la placa desde el modo DFU y actualizar el firmware de Micro Python si éste está corrupto.

Tal y como muestra en la Fig. 4.8, el dispositivo no se reconoce.

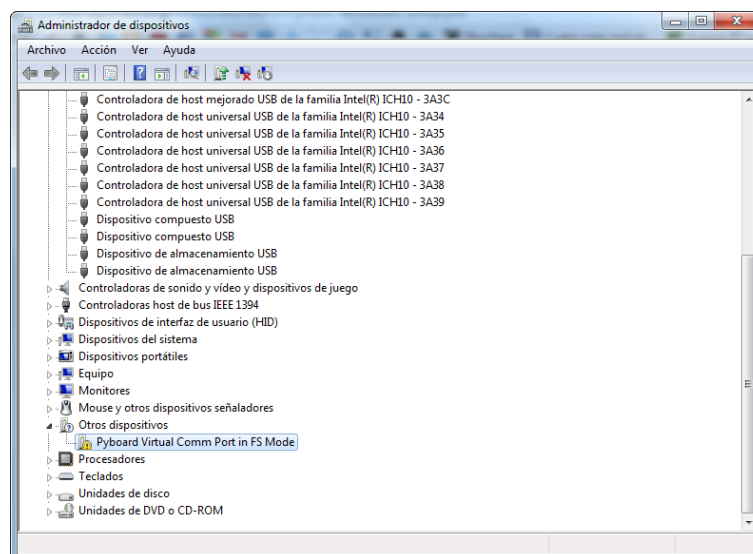


Fig. 4.8: Dispositivo no reconocido (Pyboard Virtual Comm Port in FS Mode).



Los siguientes pasos que se han seguido consisten en instalar los drivers adecuados para reconocer la Pyboard en modo DFU. Windows descarga e instala los drivers sin ningún problema. En Linux la instalación no es necesaria.

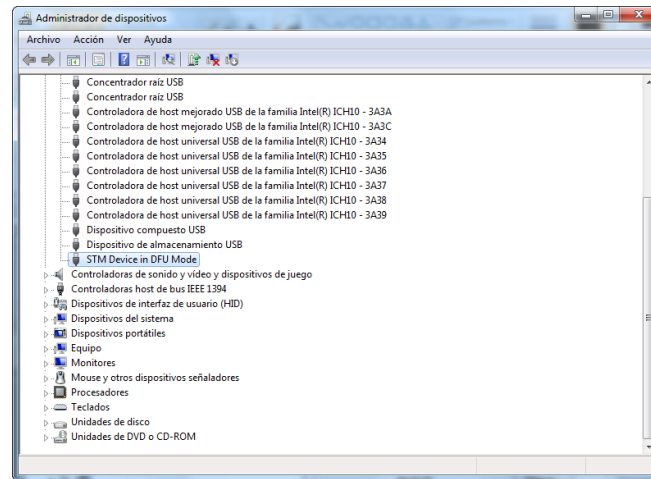


Fig. 4.9: Dispositivo reconocido en modo DFU.

Véase, en la Fig. 4.9, cómo los drivers han sido actualizados y la placa se reconoce como un dispositivo en modo DFU.

El siguiente paso debe ser utilizar este modo para actualizar el firmware de la Pyboard. Los archivos de firmware se van actualizando periódicamente, normalmente a diario, y se pueden descargar de la página oficial de Micro Python [1]:

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

Para actualizar el firmware en Windows se ha descargado el programa DfuSe (v3.0.2) de la página de STMicroelectronics [2]:

<http://www.st.com/web/en/catalog/tools/FM147/CL1794/SC961/SS1533/PF257916>

En la Fig. 4.10 se muestran los pasos 1 y 2 que se han seguido. En el paso 1 se selecciona el archivo de *firmware* descargado. En el paso 2 se instala el *firmware* seleccionado en la Pyboard.

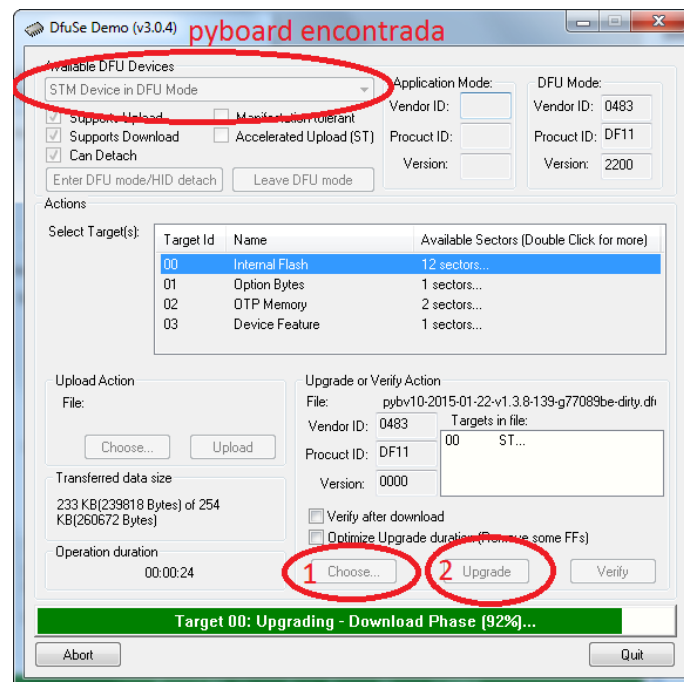


Fig. 4.10: Interfaz DfuSe con pasos de actualización.



4.2. EJEMPLOS DEL TUTORIAL

Los siguientes puntos corresponden a ejemplos del tutorial oficial que se han probado tanto en Windows como en Linux:

4.2.1. Primeros pasos

4.2.1.1. Editar main.py

Para empezar se ha abierto el archivo “main.py” en la tarjeta SD. En él se ha escrito el comando para importar la librería “pyb” y encender el LED 1 (rojo).

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

Fig. 4.11: Comandos para importar el módulo PYB y encender un LED.

4.2.1.2. Resetear la Pyboard

Para hacer funcionar el comando anterior se ha guardado el archivo “main.py” y desconectado y conectado el cable USB. El tutorial indica que es necesario pulsar el botón RST, después de conectar la Pyboard, para hacer funcionar el código.

Se ha comprobado que no es necesario desconectar/conectar la placa y apretar el pulsador RST al mismo tiempo. Son acciones aparentemente equivalentes. Ambas reinician y ejecutan el programa. No obstante, en el tutorial se recomienda seguir ambas acciones. Si se hace un gran uso del pulsador RST, Windows avisará de que la Pyboard puede estar dañada y dará la opción de repararla.

4.2.2. La “REPL prompt”

Una alternativa que simplifica la experimentación con comandos es el uso de un terminal. De esta manera se pueden probar comandos sin necesidad de resetear la Pyboard. El tipo de terminal depende del sistema operativo que se utilice. Aquí se ha utilizado tanto Windows como Linux.

Windows

En Windows se ha descargado el terminal PuTTY, que es portable (no requiere instalación) y gratuito (<http://www.putty.org/>).

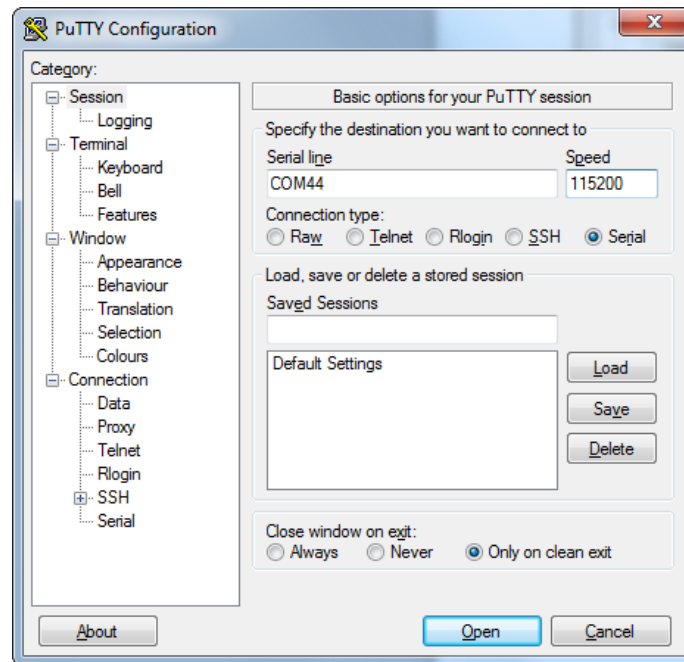


Fig. 4.12: Interfaz de PuTTY

La Fig. 4.12 muestra el interfaz de PuTTY antes de conectarse a un dispositivo. La opción que se ha seleccionado es la conexión a un puerto serie cuyo nombre en este caso y, como se ha visto anteriormente, es COM3. Se ha probado de establecer dos velocidades (Speed, Fig. 4.12) de comunicación diferentes: 9600b/s y 115200b/s. En ambos casos ha funcionado correctamente.

Linux

Para establecer un terminal de comunicación en Linux ha sido únicamente necesario escribir el comando `screen /dev/ttyACM0` en el terminal de Linux.

Es posible que sea necesario escribir `ttyACM1` o algún número superior en función del PC y los dispositivos que se tenga conectados.

4.2.2.1. Utilizar la REPL prompt

Se han hecho varias pruebas utilizando comandos típicos de Python y clases propias de Micro Python. La Fig. 4.13 muestra algunos de ellos.



```
>>> print("hola")
'hola'
>>> pyb.LED(1).on()
>>> pyb.LED(2).on()
>>> 1/2
0.5
>>> 4*'ja'
'jajajaja'
```

Fig. 4.13: Comandos probados en el terminal PuTTY.

Nótese que los guiones “>>>” no forman parte del comando, pero servirán para expresar cuándo el código se escribe en el terminal y no en un archivo ejecutable.

4.2.2.2. Resetear la Pyboard

Para hacer un “soft reset” de la Pyboard se puede pulsar CTRL+D. Haciendo esto se ha ejecutado el programa que estaba escrito en el archivo “main.py”. La Fig. 4.14 muestra la información que nos ofrece al provocar un “soft reset”.

```
>>>
PYB: sync filesystems
PYB: soft reboot
Micro Python v1.4.3 on 2015-05-17; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> █
```

Fig. 4.14: Soft Reset de la Pyboard (recorte de la imagen del terminal).

4.2.3. LEDs

Se ha comenzado asignando nombres a los objetos del tipo LED como se muestra en la Fig. 4.15.

```
>>> led1 = pyb.LED(1)
>>> led1.on()
>>> led1.off()
```

Fig. 4.15: Comandos de asignación, encendido y apagado del LED1.

Se ha escrito un pequeño bucle que alterne el encendido del LED verde y que espere un segundo entre encendido y apagado como muestra la Fig. 4.16.

```
>>> led2 = pyb.LED(2)
>>> while True:
...     led2.toggle()
...     pyb.delay(1000)
```

Fig. 4.16: Bucle de alternación de encendido del LED3

El bucle bloquea el terminal, por lo que es necesario resetear la Pyboard si se quiere continuar probando comandos o interrumpir el proceso utilizando la combinación de teclas CTRL+C.

Seguidamente se ha creado una lista para contener todos los LEDs. En la Fig. 4.17 se muestra el código escrito en el archivo “main.py”. Este programa enciende todos los LEDs, uno tras otro, dejando un lapso de 50ms entre uno y otro. Cuando están todos encendidos, sigue el mismo proceso para apagarlos.

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

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

Fig. 4.17: Programación secuencial de LEDs.

Si se interrumpe el bucle desde el terminal (CTRL+C) algunos LEDs quedan encendidos. Para evitarlo, se ha utilizado la estructura de la Fig. 4.18.

```
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()
```

Fig. 4.18: Programación secuencial de LEDs (estructura de intento y finalización).

También se ha probado específicamente el cuarto LED, ya que éste tiene una característica especial que es la regulación de la intensidad mediante una señal PWM. Se pueden probar valores de intensidad entre 0 y 255.

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

Fig. 4.19: Bucle de variación de la luminosidad del LED4.

La Fig. 4.19 corresponde al código que cambia la intensidad del LED azul gradualmente de forma que sea visible para el ojo humano.



4.2.4. El pulsador SWITCH

El pulsador USR de la Pyboard es una herramienta adicional para uso del usuario. Se han probado comandos básicos, como puede apreciarse en la Fig. 4.20.

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

Fig. 4.20: Creación del objeto pulsador USR y comprobación de estado.

Cuando el pulsador USR está apretado el método `pyb.Switch()` devuelve cierto: "True". En caso contrario devuelve falso: "False".

Este pulsador puede servir también de interruptor para lanzar funciones básicas que no requieran asignación de memoria. Se ha probado de controlar el encendido y apagado del LED verde con el pulsador mediante el constructor `lambda`, tal y como se refleja en la Fig. 4.21.

```
>>> sw.callback(lambda:pyb.LED(2).toggle())
```

Fig. 4.21: Asignación del LED2 al pulsador USR mediante `lambda`.

Se ha comprobado que, en general, el funcionamiento del pulsador USR presenta ciertos rebotes mecánicos.

Para desactivar la función `callback` se puede utilizar `sw.callback(None)`. Además de utilizar el constructor `lambda`, también se puede utilizar mediante la definición de una función, tal y como se ve en la Fig. 4.22. Las funciones no pueden tomar nunca ningún argumento.

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

Fig. 4.22: Asignación de una función al pulsador USR.

4.2.5. Acelerómetro

El acelerómetro devuelve valores de aceleración en los tres ejes x, y, z del espacio. Los valores que proporcionan van, según la documentación, de -30 a 30. Se ha empezado probando que devuelva, efectivamente, los valores del acelerómetro con los comandos sugeridos por el tutorial oficial de Micro Python. En la Fig. 4.23 se crea un objeto acelerómetro y se pregunta el valor en ese instante del eje X.

```
>>> accel = pyb.Accel()
>>> accel.x()
11
```

Se ha intentado situar los tres ejes en sus posiciones de máxima aceleración y, aun así, no se han logrado valores por encima del 23, o por debajo del -23. Además, aun dejando la Pyboard quieta, los valores varían en un rango de hasta 3 unidades. Se puede utilizar el acelerómetro, no para valores exactos, sino para rangos o en un sistema realimentado.

Se ha probado un ejemplo básico que enciende una luz si la Pyboard no está alineada con el eje X. Este ejemplo está escrito en la Fig. 4.24.

```
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)
```

Fig. 4.24: Programa de encendido de LED para un límite de inclinación.

Siguiendo el mismo concepto, se ha escrito un programa que utilice las dos coordenadas que forman el plano horizontal (x, y) y los 4 LEDs. Para valores negativos de la coordenada X se enciende el LED3 y para valores positivos de la misma el LED2. Para valores negativos de la coordenada Y se enciende el LED4 y para valores positivos el LED1. La Fig. 4.25 muestra la placa en estas cuatro posiciones mientras que la Fig. 4.26 muestra el código del programa.

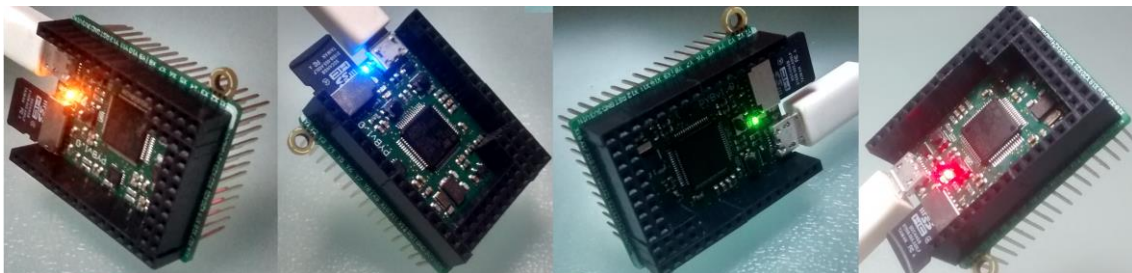


Fig. 4.25: Encendido de cada LED acorde a la dirección de la inclinación.




```
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)
```

Fig. 4.26: Código del ejemplo “Spirit level”

4.2.6. Modo seguro y restablecimiento de fábrica

4.2.6.1. Modo seguro

En modo seguro los archivos “boot.py” y “main.py” no se ejecutan. Así se pueden arreglar bucles infinitos o errores que hubiera en dichos archivos. Para hacer esto se han seguido los siguientes pasos:

1. Conectar la Pyboard al PC.
2. Mantener el pulsador USR.
3. Mientras se mantiene el pulsador USR se presiona y se suelta el pulsador RST.
4. Los LEDs se permutan de forma cíclica. Primero verde, luego naranja y finalmente naranja + verde.
5. Se mantiene el pulsador USR hasta que solo el LED naranja esté encendido.
6. El LED naranja hace cuatro flashes rápidos y luego se apaga.
7. Finalmente se está en modo seguro.

4.2.6.2. Restablecimiento de fábrica

El restablecimiento de fábrica únicamente substituye los archivos “boot.py”, “main.py” y “pybcdc.inf” por los originales, en caso de que estos estén corruptos. Esto solo lo hace para la memoria interna flash y no para la memoria SD. Los pasos seguidos son similares a los seguidos para iniciar en modo seguro. Sin embargo, en este caso, el pulsador USR debe soltarse cuando tanto el LED naranja como el verde estén encendidos.

4.2.7. Modo HID: mouse

En este apartado, existen diferencias sustanciales si se hace en Linux o en Windows.

Windows

La Pyboard puede actuar para el PC como un *mouse*, en vez de actuar como un dispositivo de almacenamiento. Para ello, se ha cambiado el archivo “boot.py”, donde aparecen los comandos que se muestran en la Fig. 4.27. Para hacer que funcione como un mouse, se debe poner en modo HID. Esto se ha hecho eliminando el símbolo de comentario # en la última línea.

```
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
```

Fig. 4.27: Código BOOT.py para configuración en modo HID.

Efectivamente, al desconectar y volver a conectar la Pyboard, ésta no aparecerá como disco extraíble. No obstante, tampoco se muestra como dispositivo reconocido por el PC (véase Fig. 4.28).

Se ha discutido este tema en el foro de Micro Python [3]. El enlace de la discusión es el siguiente: <http://forum.Micro Python.org/viewtopic.php?f=2&t=617>.



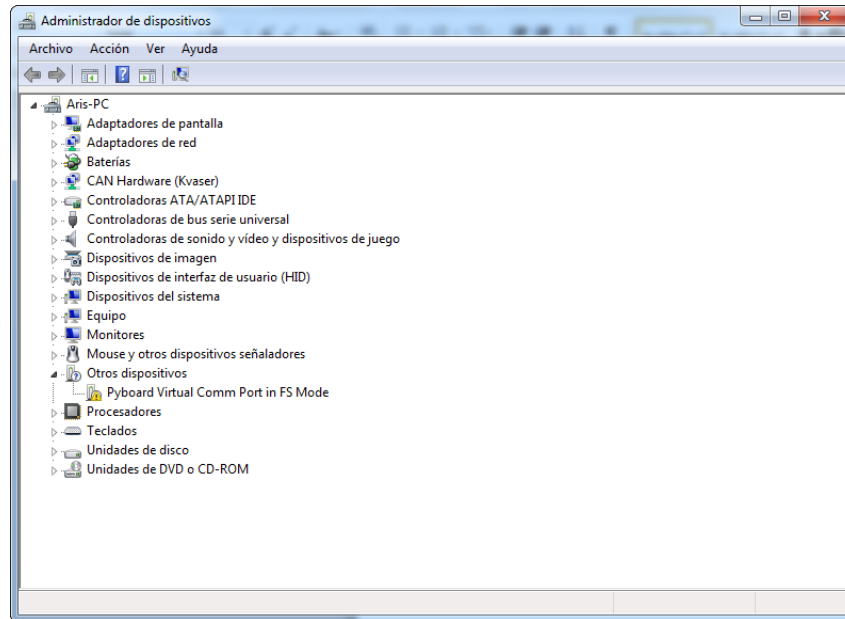


Fig. 4.28: Pyboard no reconocida por Windows.

Linux

En Linux se ha seguido el mismo procedimiento que con Windows sin aparecer ningún problema.

4.2.7.1. Enviar eventos desde el terminal

Windows

Se ha intentado abrir el terminal PuTTY para enviar comandos a la Pyboard, pero no ha funcionado correctamente. No se reconoce el puerto COM3. Según usuarios del foro de Micro Python, este problema es debido a un error en los drivers que hasta la fecha está pendiente de actualización.

Linux

En Linux sí que se puede utilizar el terminal para enviar instrucciones a la Pyboard. Se ha comprobado el correcto funcionamiento de la función mostrada en la Fig. 4.29.

```
>>> 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)
```

Fig. 4.29: Programa de control del mouse (Linux): movimiento circular.

4.2.7.2. Crear un mouse con el acelerómetro

Para acceder de nuevo a la tarjeta SD de la Pyboard para escribir un programa, se ha iniciado en modo seguro. En el archivo “main.py” se han escrito los comandos, tal como se indica en la Fig. 4.30 y después se ha hecho un reset.

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

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

Fig. 4.30: Programa de control del mouse: movimiento ligado al acelerómetro.

Este programa mueve el cursor del ratón, equiparando la velocidad de éste a los valores del acelerómetro de la Pyboard. Funciona tanto en Windows como en Linux.

4.2.8. Temporizadores

En este punto se explican las primeras pruebas que se hicieron siguiendo el tutorial sin llegar a profundizar en gran medida en el uso y características de los temporizadores.

4.2.8.1. Temporizador e interrupciones

Para empezar se ha creado un temporizador desde el terminal PuTTY en Windows.

```
>>> tim = pyb.Timer(4)
>>> tim
Timer(4)
>>> tim.init(freq=10)
>>> tim
Timer(4, freq=10, prescaler=624, period=13439, mode=UP, div=1)
```

Fig. 4.31: Creación de un temporizador y atributos de la clase Timer.



En la Fig. 4.31 se muestra primero la creación del temporizador. Después, el objeto que se ha creado. En la tercera línea se inicializa a la frecuencia de 10Hz y, por último, cómo la inicialización influye en el resto de parámetros. En el tutorial, para el mismo procedimiento, genera diferentes parámetros (Fig. 4.32). Esto puede ser debido a la versión de firmware.

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

Fig. 4.32: Atributos por defecto del temporizador en el Tutorial oficial.

La clase *Timer*, al igual que la clase *Switch*, también dispone del método *callback* para poder utilizar un temporizador como interruptor de una función. Véase la Fig. 4.33, donde se muestran los comandos básicos cuyo funcionamiento se ha comprobado.

```
>>> tim.callback(lambda t:pyb.LED(1).toggle())
>>> tim.init(freq=20)
>>> tim.callback(None)
```

Fig. 4.33: Interruptor del temporizador

También se ha probado cómo dos temporizadores diferentes pueden generar interrupciones de forma simultánea (Fig. 4.34).

```
>>> 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())
```

Fig. 4.34: Dos interrupciones simultaneas de temporizador

4.2.8.2. Contador de microsegundos

El contador de microsegundos es un aspecto importante, ya que nos permitirá controlar y medir el tiempo de los procesos. Se ha comenzado comprobando la inicialización de un temporizador mediante la definición de los parámetros *prescaler* y *period* (Fig. 4.35).

```
>>> micros = pyb.Timer(2, prescaler=83, period=0x3fffffff)
>>> micros.counter(0), micros.counter()
None, 7
```

Fig. 4.35: Temporizador de microsegundos

La segunda línea de la Fig. 4.35 pone el contador a cero mediante el uso de `micros.counter(0)`, seguido de `micros.counter()`, que ayuda a comprobar que el

comando anterior funcione. Efectivamente, la segunda línea de comandos devuelve un 7. Esto es el tiempo en microsegundos que ha pasado desde que se ha puesto a 0. Éste es prácticamente nulo.

Se ha hecho una pequeña prueba comparando el contador (asignado al encendido de un LED) con el cronómetro del teléfono móvil. El resultado es satisfactorio ya que, obviando un pequeño desfase por error humano, corresponden los segundos desde el primer instante.

4.2.9. Ensamblador

El objetivo del trabajo no contempla el aprendizaje del lenguaje ensamblador. No obstante, sí que es importante comprobar su funcionamiento y comprobar su viabilidad.

4.2.9.1. Retornar un valor

La función que se muestra en la Fig. 4.36 retorna el valor del registro “r0”, al cual se le asigna el valor 42.

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

Fig. 4.36: Función de retorno de valor (Ensamblador).

Se ha importado esta función desde el archivo “boot.py” y ejecutado en el terminal (Fig. 4.37).

```
>>> fun()
42
```

Fig. 4.37: Comprobación de función de retorno de valor (Ensamblador).

4.2.9.2. Acceso a periféricos

El ejemplo del tutorial enciende el LED rojo con los comandos de la Fig. 4.38. La función, escrita en lenguaje ensamblador del microcontrolador, activa determinados bits en el registro adecuado. El tutorial oficial explica brevemente algunas de las instrucciones que se utilizan.

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



Fig. 4.38: Programa de encendido del LED1 (Ensamblador).

4.2.9.3. Aceptación de argumentos

Las funciones escritas en ensamblador aceptan un máximo de tres argumentos, que corresponden a los registros r0, r1 y r2. Se ha probado la función de suma que aparece en el tutorial, tal y como muestra la Fig. 4.39, haciendo uso del terminal PuTTY.

```
>>> @Micro Python.asm_thumb
... def asm_add(r0,r1):
...     add(r0,r0,r1)
...
>>> asm_add(1,2)
3
```

Fig. 4.39: Operación suma (Ensamblador).

4.2.9.4. Bucles

También se pueden añadir etiquetas propias del lenguaje ensamblador para saltar de una parte del código a otra. En la figura A1 del anexo (Bucle en ensamblador) se puede ver el código que itera un número dado de veces el encendido de un LED.

```
>>> flash_led(5)
0
>>> flash_led(10)
0
```

Fig. 4.40: Comprobación en terminal de la función “flash_led()” (Ensamblador).

Como se puede observar en la Fig. 4.40, la función retorna 0 cuando acaba el proceso iterativo. En el primer caso, el LED verde se enciende y se apaga 5 veces. En el segundo, lo hace 10 veces.

Se ha probado de interrumpir la ejecución de la función utilizando CTRL+C. Se ha interrumpido quedando el LED encendido. Esto ocurrirá la mitad de las veces. Si se vuelve a ejecutar la misma función, ésta partirá de cero con el LED apagado sin importar su estado inicial. No ocurriría lo mismo si se utilizase Python para conseguir el mismo efecto con un bucle `for` y el comando `toggle()`.

4.3 CONCLUSIONES

Micro Python ofrece un entorno muy sencillo y fácil de utilizar, tanto como lo es Python en un PC.

Windows tiene problemas de compatibilidad con la Pyboard en modo HID y no parece existir una solución a corto plazo. Los colaboradores de Micro Python utilizan fundamentalmente Linux, ya que ofrece herramientas más sólidas que Windows. Además, la solución a cualquier problema llega antes para Linux que para Windows.



5. REGISTRADOR DE DATOS

En este capítulo se pretende crear archivos en la tarjeta SD y en la memoria flash interna desde el entorno de Micro Python. Es preciso conocer las funciones que en Python permiten crear archivos y escribir en ellos.

En Python, los parámetros de la función `open()` deben incluir la ruta donde se situará el archivo a partir del directorio donde se ejecute Python, además del nombre y el formato del archivo. El parámetro 'w' indica que se abre el archivo para escribir. Para que el archivo se guarde, con lo que se ha escrito en él, es necesario cerrarlo utilizando `file.close()`.

Antes de nada, se ha optado por abrir el terminal de PuTTY con la placa en modo "CDC+MSC" para probar los comandos de la Fig. 5.1.

```
>>> file = open('file.txt','w')
>>> file.write('hola mundo')
>>> file.close()
```

Fig. 5.1 Comandos de construcción, escritura y guardado de archivo (Python).

Se ha comprobado cómo el archivo no se ha creado buscándolo en el disco extraíble (memoria de la Pyboard) con el explorador de Windows. Se ha descubierto, después de varias pruebas, que es necesario especificar la ruta a la SD ('/sd/') o a la memoria FLASH interna ('/flash/'). Por lo tanto, para que funcione de forma correcta se debe escribir, tal y como muestra la Fig. 5.2.

```
>>> file = open('/sd/file.txt','w')
>>> file.write('hola mundo')
>>> file.close()
```

Fig. 5.2: Creación de un archivo en la tarjeta SD.

O bien como muestra la Fig. 5.3, en el caso de que no haya una tarjeta SD insertada.

```
>>> file = open('/flash/file.txt','w')
>>> file.write('hola mundo')
>>> file.close()
```

Fig. 5.3: Creación de un archivo en la memoria FLASH.

Una vez hecho esto, el archivo no es visible en la SD si se busca con el explorador de Windows. Tampoco es útil actualizar la carpeta. La única solución es hacer un *hard reset* de la placa para poder buscar el archivo desde el explorador de Windows.

5.1. GUARDAR DATOS DEL ACELERÓMETRO

Como ejemplo se ha probado de capturar los datos del acelerómetro durante un intervalo de tiempo y guardarlos en un archivo de texto o una hoja de cálculo. Este proceso se entiende como el programa principal que se pretende repetir periódicamente para acumular diferentes datos en el tiempo.

Se ha comenzado creando un archivo de texto desde el terminal Putty, con los comandos vistos en el punto anterior y separando los datos del acelerómetro con saltos de línea. La Fig. 5.4 muestra un ejemplo de código probado desde el terminal.

```
>>> file.write(str(pyb.Accel().x())+'\n')
>>> file.write(str(pyb.Accel().x())+'\n')
```

Fig. 5.4: Lectura y escritura en serie del acelerómetro en un archivo.

Se ha visto que no se pueden guardar datos en columna utilizando un salto de línea '\n', ya que éste no funciona. Se ha probado escribiendo un archivo para ejecutarse tras un reset de la Pyboard y tampoco ha funcionado el salto de línea.

Para la utilidad de guardar datos iterativamente se puede utilizar un archivo de extensión ".csv". Así se ha continuado intentando utilizar la función de interrupción en el cambio de ciclo de un temporizador (callback) para recoger datos y guardarlos con una frecuencia dada (Fig. 5.5). Sin embargo, se ha descubierto y probado que el método de interrupción no puede asignar memoria, por lo que las funciones que ejecuta deben ser muy sencillas. En el caso del código de la Fig. 5.5 se genera un error de asignación de memoria.

```
file = open('/sd/file.csv','w')
def write():
    x,y,z = accel.x(), accel.y(), accel.z()
    file.write('{},{},{}\n'.format(x,y,z))
timer = pyb.Timer(1)
timer.init(freq = 1)
timer.callback(write)
```

Fig. 5.5: Programa de muestreo de acelerómetro no viable.

Una posible solución que se ha probado consiste en utilizar un bucle infinito donde se aloje el programa principal (Fig. 5.6). En este caso, no se puede cambiar la frecuencia de iteración o de ejecución del programa principal de forma precisa. Dicha frecuencia dependerá del tiempo que tarde el programa principal, por lo que ésta se puede cambiar añadiendo un tiempo de espera que aumente el periodo de iteración. Este concepto se ilustra en el código de la Fig. 5.6.



```

file = open('/sd/file.csv','w')
while True:
    x,y,z = accel.x(), accel.y(), accel.z()
    file.write('{},{},{}\n'.format(x,y,z))
    pyb.delay(1000)

```

Fig. 5.6: Programa de muestreo de acelerómetro viable.

Para más sofisticación se podría utilizar otro bucle dentro de dicho bucle infinito que contenga el programa principal y que dependa del pulsador de usuario (USR). Esto permitiría escoger cuándo se están recogiendo datos y cuándo no. En la Fig. 5.7 se muestra el código con este nuevo enfoque.

```

import pyb

accel = pyb.Accel()
switch = pyb.Switch()
f = False
t = True

def condicion():
    global t,f
    f,t = t,f

switch.callback(condicion)

while True:
    pyb.wfi()
    if f:
        pyb.LED(4).on()
        file = open('/sd/file.csv','w')
        while f:
            x,y,z = accel.filtered_xyz()
            file.write('{},{},{}\n'.format(x,y,z))
            pyb.delay(1000)
        file.close()

```

Fig. 5.7: Programa completo de captación y escritura del acelerómetro

Es posible que el programa siga iterando en el bucle infinito sin ejecutar ninguna acción. En ese caso se puede ahorrar energía utilizando la función “pyb.wfi()”.

El pulsador de usuario se utiliza para cambiar el booleano que condiciona el comienzo del almacenamiento de datos del acelerómetro. Esto se hace mediante la ejecución de la función “condicion”. El código visto para esta función es suficientemente ligero como para que el método *callback* lo pueda ejecutar.

Existe otra forma intuitiva de hacer esto, pero que no cumple los límites de memoria de la interrupción (Fig. 5.8).

```
def condicion():  
    global f  
    if f:  
        f = False  
    else:  
        f = True
```

Fig. 5.8: Función de cambio de estado booleano no viable.

5.2. EJECUCIÓN DEL PROGRAMA

Para ejecutar el código completo del programa se debe escribir el comando `pyb.main('registrador.py')` en el archivo `boot.py` y hacer un primer *hard reset*. El nombre del archivo donde se localiza el programa puede ser, según el ejemplo, 'registrador.py'.

Éste es el procedimiento habitual para ejecutar programas, aunque no siempre es útil si se ejecuta estando la placa conectada al PC. Una vez se resetea la Pyboard, ésta se reinicia ejecutando primero el archivo `boot.py` y después el archivo indicado en el comando `pyb.main()`. El PC detecta la placa como dispositivo de almacenamiento a la par que se ejecuta este último archivo. Esto implica que a partir de un cierto momento en el que se está ejecutando el programa, éste no es capaz de guardar documentos en la memoria (SD o flash), ya que está siendo utilizada por el PC.

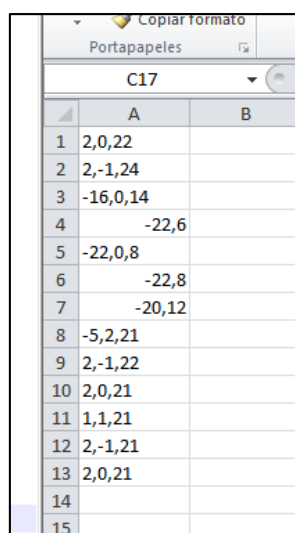
El programa registrador está en ejecución con la Pyboard en modo "CDC+MSC". Para empezar el registro de datos se debe presionar el pulsador USR una vez. Si se presiona el pulsador USR por segunda vez, el programa guarda todos los datos recogidos del acelerómetro en un documento. En ese preciso instante no es posible ver dicho documento a través del explorador de Windows. Para verlo, será necesario un segundo reset de la Pyboard.

El programa ya no es necesario una vez se han capturado los datos. Lo lógico entonces sería eliminar el comando `pyb.main('registrador.py')` antes de hacer el segundo reset. Así se evitaría volver a ejecutar el programa y sobrescribir el archivo de los datos recogidos, si los datos ya se han recogido satisfactoriamente. No obstante, se ha comprobado que haciendo esto, el documento con los datos no se guarda en la memoria al resetear la placa. En cambio, en el caso de que no se elimine dicho comando, sí que encontraremos el documento guardado en la memoria tras el segundo reset. Sin embargo, el programa volverá a ejecutarse. Lo siguiente es borrar el comando de ejecución del programa después del segundo reset y hacer un tercer reset para que el programa deje de correr.



Para evitar esto se podría cambiar el comando `pyb.main('registrador.py')` por el comando `import registrador.py`. De este modo, se aseguraría que el PC no interfiera con la placa hasta que se hayan guardado todos los datos. No obstante, sería necesario ejecutar la Pyboard en modo seguro para romper el bucle infinito del programa y así poder utilizar la placa nuevamente como dispositivo de almacenamiento masivo en el PC. Lo que ocurre cuando se inicia el modo seguro es que se presiona el pulsador USR. De acuerdo con el programa, el pulsador USR inicia y finaliza el guardado de datos. Es posible entonces que se acabe reemplazando el documento con los datos por uno nuevo sin datos.

Una vez ejecutado el programa se han comprobado los datos guardados. Como se puede observar en la Fig. 5.9, hay una serie de datos en los que falta uno de los ejes. Particularmente en las muestras 4,6 y 7. Esto es un error del firmware de Micro Python, ya que el hardware funciona y el programa escrito en Python es correcto.



	A	B
1	2,0,22	
2	2,-1,24	
3	-16,0,14	
4	-22,6	
5	-22,0,8	
6	-22,8	
7	-20,12	
8	-5,2,21	
9	2,-1,22	
10	2,0,21	
11	1,1,21	
12	2,-1,21	
13	2,0,21	
14		
15		

Fig. 5.9: Hoja de cálculo con datos del acelerómetro

5.3. CONCLUSIONES

Cuando se ejecuta un programa en la Pyboard que requiere la utilización de la memoria flash o SD, es necesario que el PC no controle dicha memoria. Para ello, se puede bloquear el reseteo de la Pyboard en un punto en el que el PC no tome control de la memoria. En ese punto es cuando se puede ejecutar el programa en cuestión. Otro modo puede ser utilizar el modo Human Interface Device ("CDC+HID").

Para el tipo de programas que requiere de un bucle infinito y del uso de la memoria, podría ser necesario el uso de inicialización en modo seguro. Lo que convierte el procedimiento en algo relativamente pesado.



6. DEPURADOR

En este capítulo se pretende crear un software que compare el tiempo de ejecución de dos programas diferentes o simplemente nos diga el tiempo que dura un proceso. Este programa puede servir, entre otras cosas, para comparar dos funciones iguales pero escritas en diferentes lenguajes, que en el caso de la Pyboard podrían ser python y ensamblador.

6.1. CONTADOR DE MICROSEGUNDOS

Este depurador debe utilizar un temporizador con el periodo suficientemente grande para que no se reinicie el ciclo antes de que los procesos a depurar se hayan finalizado. Además, debe garantizar que cada salto de bit equivalga a una unidad de tiempo del sistema internacional. De esta manera se pueden consultar instantes a dicho temporizador y restarlos para saber el lapso de tiempo entre un instante y otro.

La alternativa sería disponer de un contador que vaya contando el número de ciclos que ha superado el temporizador hasta el momento. Así, el tiempo total entre instantes sería el resultado de la multiplicación del número de ciclos superados por el periodo de cada ciclo, y a esto sumarle el tiempo del ciclo que aún no ha acabado y restarle el tiempo que llevaba del primer ciclo.

$$\text{Incremento de tiempo} = (N_{\text{ciclos}} \cdot T_{\text{ciclo}}) + t_{\text{final}} - t_{\text{inicial}}$$

Esta última alternativa podría ser útil en caso de necesitar un periodo entre una acción y la siguiente (la misma acción repetida en el tiempo) más grande que el máximo periodo disponible de entre todos los temporizadores. En el siguiente apartado se exponen una serie de experimentos que se han hecho siguiendo esta línea para ver el alcance de cada temporizador. En cualquier caso, para periodos muy grandes, se puede utilizar la clase RTC que mantiene un registro de la fecha y hora con una precisión milimétrica y con métodos de calibrado.

Dado que los procesos que se pretenden depurar no durarán demasiado en conjunto, se ha decidido crear un contador cuyo ciclo dure unos 17 minutos como máximo, que será más que suficiente. En el tutorial oficial de Micro Python se muestra cómo crear un temporizador únicamente de estas características.

6.2. CONSTRUCCIÓN DEL CONTADOR DE MICROSEGUNDOS

A continuación se describen los pasos que se han seguido para analizar el funcionamiento de los temporizadores y así poder crear un “cronómetro” de cualquier configuración. En el tutorial oficial ya se ejemplifican los temporizadores creando un contador de microsegundos.

Para construir un contador (o temporizador) se pueden especificar la “*freq*” por un lado, o el “*period*” y el “*prescaler*” por el otro. Con una serie de pruebas básicas se ha comprobado que la frecuencia (*freq*) que se especifica en la inicialización del temporizador es la frecuencia de reinicio de ciclo.

6.2.1. Ecuación de prescaler, period y freq

No se ha encontrado la ecuación que relaciona las tres propiedades del temporizador (*prescaler*, *period*, *freq*) en el *datasheet* del microcontrolador. Sin embargo, sí que se ha podido deducir del tutorial de Micro Python. Dicha ecuación se muestra en la Fig. 6.1.

$$freq \text{ de cambio de ciclo} = \frac{freq \text{ asociada}}{(prescaler + 1) \cdot period}$$

Fig. 6.1: Fórmula para calcular la frecuencia de un temporizador.

A la hora de deducir la ecuación, el tutorial puede llevar a confusión, ya que en un ejemplo se utiliza el término 168/2, en vez de 84, refiriéndose a la velocidad de reloj asociada al temporizador 2. Por los datos que proporciona el fabricante sabemos que la velocidad de reloj asociada al temporizador 2 es de 84MHz y no de 168MHz, por lo que el término 2 del denominador se refiere al divisor interno que el fabricante ya tiene en cuenta en las especificaciones.

Existe también otra incoherencia con dos ejemplos del tutorial. En uno de ellos la frecuencia de cambio de bit se iguala a la frecuencia de reloj dividida entre el *prescaler*, mientras que en otro ejemplo se divide entre *prescaler+1*.

Para verificar que la frecuencia de reloj se divide entre *prescaler+1* y no entre *prescaler*, se ha iniciado un temporizador con un valor de *prescaler* muy pequeño tal que el término “+1” influya considerablemente en la frecuencia final. Se ha visto que la frecuencia del temporizador se puede consultar con el comando `timer.freq()`, tanto si está inicializado con el valor *freq* como si lo está con los valores *period* y *prescaler*. De ésta manera, se verifica la utilización del divisor de frecuencia “*prescaler+1*”.



Para verificar la fórmula se han creado varios objetos *Timer* con valores aleatorios de *period* y *prescaler* que ofrezcan una frecuencia mensurable manualmente. Se ha sometido el encendido y apagado del LED naranja al cambio de ciclo de cada temporizador. Después se ha cronometrado manualmente el periodo de encendido y apagado para comparar la frecuencia experimental con la teórica dada por la fórmula. Más adelante se comparará la frecuencia con el osciloscopio y la salida de valores analógicos.

En conclusión, con el *prescaler* y el *period* se controlan, respectivamente, la frecuencia de cambio de bit y el número de bits que se contarán.

6.2.2. Características de los temporizadores

Los valores de *period* y *prescaler* vienen especificados en el *datasheet* del microcontrolador [4]. Cada temporizador tiene 16bits de memoria para los valores de *prescaler* y *period* exceptuando el 2 y el 5, que tienen 32 bits para el valor de *period*. Además cada temporizador tiene una frecuencia asociada de reloj. Los temporizadores 1, 8, 9, 10 y 11 tienen una frecuencia de reloj máxima asociada de 168 MHz mientras que el resto tienen una frecuencia de 84 MHz.

Con los datos obtenidos del fabricante se ha elaborado la tabla de la Fig. 6.2. Nótese que el número 3 no aparece ya que, como se menciona en el tutorial, es de uso interno.

clock (MHz)	84	84	168
<i>period</i> máximo (bit)	65535	4294967495	65535
Temporizadores	4,6,7,12,13,14	2,5	1,8,9,10,11
<i>prescaler</i> máximo (bit)	65535	65535	65535

Fig. 6.2: Tabla de propiedades de temporizadores según el fabricante.

Con los datos recogidos hasta ahora, se pueden definir las frecuencias máximas de los temporizadores haciendo *prescaler* = 1. Se obtienen las siguientes frecuencias:

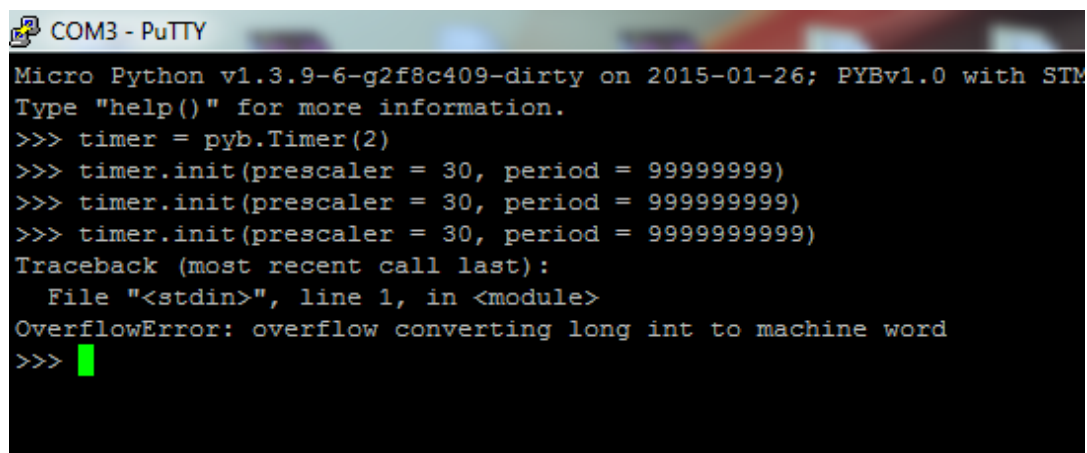
- Frecuencia máxima de 84 MHz para los temporizadores 1, 8, 9,10 y 11.
- Frecuencia máxima de 42 MHz para los temporizadores 2, 4, 5, 6, 12,13 y 14.

Como se indica en el tutorial, el cronómetro con más definición que podemos construir es un contador de microsegundos.

Los procesos que podemos iterar con el uso de un temporizador y el método *callback*, utilizan la frecuencia de cambio de ciclo para lanzar una interrupción. Con esa interrupción se ejecuta el proceso a iterar. Cada proceso tiene un periodo de duración propio que obliga una frecuencia de iteración mucho más pequeña. Por lo tanto, las frecuencias muy elevadas en temporizadores, solo son útiles para cronometrar.

6.2.3. Comprobación de los datos del fabricante

A pesar de los datos que se presentan en la Fig. 6.2, los temporizadores se pueden inicializar con parámetros fuera de los límites. Para comprobar estos datos se ha escrito un programa que utiliza el método de la bisección para encontrar qué valores límite permite la inicialización de cada temporizador. Sabiendo siempre que al sobrepasar el límite de un parámetro (*prescaler* o *period*) en la inicialización, se genera el error *OverflowError* de la Fig. 6.3. Este programa se puede encontrar en la figura A2 del anexo con el nombre de *find_prescaler_period*.



```
COM3 - PuTTY
Micro Python v1.3.9-6-g2f8c409-dirty on 2015-01-26; PYBv1.0 with STM
Type "help()" for more information.
>>> timer = pyb.Timer(2)
>>> timer.init(prescaler = 30, period = 999999999)
>>> timer.init(prescaler = 30, period = 999999999)
>>> timer.init(prescaler = 30, period = 999999999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: overflow converting long int to machine word
>>>
```

Fig. 6.3: *OverflowError* en terminal PuTTY.

El programa *find_prescaler_period* encuentra primero el número más elevado para el número de cifras más elevada que se permita. Luego resta la mitad de ese número al mismo número y, si da error, sigue restando, esta vez, la mitad del decremento anterior. Si por lo contrario, no da error, suma la mitad del decremento anterior en valor absoluto. Siguiendo esta lógica se alcanza un número con decimales que se redondea adecuadamente al final del programa.

Con este código se han encontrado los límites que impone Micro Python a los valores de *period* y *prescaler*, al inicializar un temporizador. Para el *prescaler* de todos los temporizadores, se permite asignarle un valor de 32 bits mientras que al *period*, también de todos los temporizadores, se permite asignarle un valor de 30 bits.



Estos datos no son coherentes con el diseño del microcontrolador (ver Fig. 6.2). Lo que ocurre en este caso es que se recorta todo lo que sobresalga de 16 bits. Por lo tanto, si inicializamos con un *period* de 80.000, que equivale al número binario “00010011100010000000”, utilizará los primeros 16 bits que equivalen al 14464 (0011100010000000 en binario). Esto se muestra en la Fig. 6.4.

```
>>> t.init(prescaler = 80000, period = 80000)
>>> t
Timer(1, freq=0, prescaler=14464, period=14464, mode=UP, div=1, deadtime=0)
```

Fig. 6.4: Inicialización de temporizador con valores fuera del límite.

6.2.4. Comprobación de frecuencia mínima

La frecuencia mínima (de ciclo entero) debería conseguirse para los temporizadores 2 y 5 con los máximos valores de *prescaler* y *period*. El problema es que Micro Python no deja inicializar con un *period* de 2^{32} . No obstante, se puede deducir siguiendo la fórmula, que para esos valores la frecuencia de cambio de ciclo es de $2,98423 \cdot 10^{-7}$ Hz. Se ha inicializado el temporizador 2 a dicha frecuencia y comprobado su definición como se muestra en la Fig. 6.5. En esta figura se puede ver como no se cumple la frecuencia que se ha impuesto.

```
>>> t.init(freq = 0.000000298423382)
>>> t
Timer(2, freq=0, prescaler=9999, period=4294967294, mode=UP, div=1)
>>> t.freq()
0.01955782
```

Fig. 6.5: Inicialización del temporizador 2 a mínima frecuencia.

Después de probar repetidamente valores más pequeños, se ha visto que el 0,01955782 es el número más pequeño de frecuencia de cambio de ciclo que se puede definir, independientemente de la frecuencia de cambio de bit. Por lo menos, este es el valor mínimo que retorna el comando `timer.freq()`.

Se ha comprobado rápidamente utilizando el cronómetro de un teléfono móvil y el método `counter()` que sí que se pueden definir frecuencias de cambio de ciclo por debajo de 0,019 aunque el método `freq()` no las revela. Se ha intentado utilizar el método de interrupción `callback` para encender un LED y así poder comprobar con más facilidad si la frecuencia es correcta. Sin embargo, se ha descubierto que el método `callback` no funciona para frecuencias de cambio de ciclo inferiores a 1 Hz.

6.2.5. Comprobación de frecuencia máxima

En el apartado 4.2.8 se ha comprobado el correcto funcionamiento de un contador de microsegundos. No obstante, como ejemplo, se ha escrito un programa que enciende un LED cada segundo si la frecuencia definida corresponde a la frecuencia real. Éste es un método poco exacto, ya que cuenta con error humano porque se debe comparar la iteración del estado del LED con el cronómetro de un teléfono móvil. No obstante, es suficiente para ilustrar a qué frecuencias permite Micro Python iterar un proceso tan básico, y qué ocurre cuando se definen frecuencias muy elevadas.

Este programa se ha llamado `find_freq` y se puede encontrar en la figura A3 del anexo. El programa permuta el estado del led cuando el temporizador ha finalizado X ciclos. El valor de X es la frecuencia que se especifica al iniciarlo. Eso tiene el efecto de encender y apagar el LED cada 1 segundo. Sea cual sea la frecuencia especificada, siempre será igual a la real, si el LED se conmuta cada segundo.

Se han probado frecuencias de hasta 45 kHz. A partir de ese valor se aprecia cómo se ralentiza la iteración del LED. Esto implica que la frecuencia real es inferior a la especificada. También se observa cómo a partir de esa frecuencia, el terminal se bloquea y el microcontrolador no responde a interrupciones externas. Posteriormente, se ha podido comprobar esta frecuencia en el laboratorio midiendo los pulsos en un pin del LED con un osciloscopio.

Usando un contador de microsegundos sabemos que la permutación de un LED tarda $22\mu s$. La frecuencia máxima teórica que permite sería de 45 kHz, que corresponde con la medida anteriormente.

6.3. CONSTRUCCIÓN DEL DEPURADOR

El archivo que contiene el programa se ha nombrado *depurador.py*. La estructura del programa es sencilla. Es necesario crear dos funciones que contengan los procesos a comparar, para poder llamarlas posteriormente. En este ejemplo se han llamado `proceso1` y `proceso2`.

Se debe inicializar el contador de microsegundos y luego ejecutar los dos procesos uno tras otro. Es necesario guardar el tiempo que tarda cada proceso en una variable. Este tiempo es la resta de las consultas al contador antes y después de ejecutar cada función. Por último, se abre un archivo de texto y se guardan los datos.

Es evidente que en este programa se podrían comparar tantos procesos como se quiera. Solo sería necesario añadir la función y las variables para almacenar el tiempo que tarda



cada uno. Por lo contrario puede servir simplemente para medir el tiempo de un solo proceso. El código de este programa se puede encontrar en el apartado A4 del anexo.

6.4. FUNCIONES A COMPARAR

Las funciones que se comparan consisten en encender y apagar el led rojo en Python y en ensamblador. El tiempo que tarda el proceso en Python es 31.214µs mientras que el tiempo que tarda en ensamblador es de 11.446µs. Con esta prueba se puede ver como el intérprete de Python tarda aproximadamente tres veces más cuando se utiliza Python que cuando se utiliza ensamblador. La Fig. 6.6 muestra el código en Python mientras que la Fig. 6.7 muestra el código en lenguaje ensamblador.

```
def procesol() :  
    s=0  
    while s<1000:  
        pyb.LED(1).on()  
        pyb.LED(1).off()  
        s+=1
```

Fig. 6.6: Iteración de encendido de LED 1000 veces (Micro Python).

```
@Micro Python.asm_thumb  
def ledON() :  
    movwt(r0, stm.GPIOA)  
    movw(r1, 1 << 13)  
    strh(r1, [r0, stm.GPIO_BSRR])  
  
@Micro Python.asm_thumb  
def ledOFF() :  
    movwt(r0, stm.GPIOA)  
    movw(r1, 1 << 13)  
    strh(r1, [r0, stm.GPIO_BSRH])  
  
def proceso2() :  
    s=0  
    while s<1000:  
        ledON()  
        ledOFF()  
        s+=1
```

Fig. 6.7: Iteración de encendido de LED 1000 veces (Micro Python+Ensamblador).

La idea en este punto no es elaborar una comparación muy detallada. Para hacer eso se necesitarían comparar otros procesos, pero el objetivo de este proyecto no es aprender a escribir en ensamblador. Por lo tanto, estos datos ya nos dan una idea general. Además de Python y ensamblador, se podría optar por utilizar C para programar funciones que, escritas en Python son demasiado lentas. La utilización de C en Micro Python se estudiará más adelante.

6.5. EJECUCIÓN DEL PROGRAMA

En el punto 5.2 se ha explicado cómo ejecutar un programa y qué inconvenientes pueden surgir si lo que se busca es crear y guardar un documento en la SD o en la memoria flash.

En el caso del programa depurador, si el documento no se crea y se guarda rápidamente después de su ejecución, el PC tomará control de la memoria de la placa y será imposible guardarlo instantáneamente. Esto puede pasar para procesos no muy rápidos.

El procedimiento a seguir en este caso es igual que en el punto 5.2. No obstante, puede importarse (`import depurador`) el programa depurador (bloqueando temporalmente el reinicio de la placa) desde el boot.py sin necesidad de iniciar la placa en modo seguro, ya que el programa depurador no tiene un bucle infinito. Una vez lea todo el programa y escriba los datos, se iniciará la Pyboard normalmente y el documento con los datos aparecerá en la SD.

6.6. CONCLUSIONES

Para concluir este apartado se hacen una serie de consideraciones:

- El firmware no está del todo adaptado a las características de los temporizadores. Puede llevar a confusión al creer que se ha construido un temporizador que en realidad no es viable.
 - o Los temporizadores 2 y 5 no se pueden inicializar con un contador de 32bits como permite el fabricante, sino con un contador máximo de 30bits.
 - o El resto de parámetros de todos los temporizadores están sobredimensionados con respecto a los valores que permite el microcontrolador.
- No se pueden utilizar métodos de interrupción para temporizadores con frecuencia inferior a 1Hz.
- Para frecuencias inferiores a 1Hz, los temporizadores no se inicializan con normalidad. Las propiedades que muestra el temporizador no siempre se corresponden con las reales.
- Dada la velocidad máxima de los temporizadores, el contador de unidades temporales del SI más preciso que se podría (y que se puede) construir es un contador de microsegundos.
- Las frecuencias de iteración que se consiguen, para procesos sencillos escritos en Python, alcanzan los 45kHz aproximadamente.



- Crear un programa de medida de tiempo en Micro Python es muy sencillo, y se convierte en una herramienta útil para otros problemas.

7. SALIDAS Y ENTRADAS ANALÓGICAS

En este apartado se pretende comprobar el funcionamiento de las entradas y salidas analógicas. Es preciso hacer una serie de consideraciones a modo de introducción:

La tensión máxima para las entradas y salidas de la Pyboard es de 3,3 V, que corresponde a la tensión de alimentación de la placa. En el caso de la entrada no se especifica la tensión máxima en la documentación oficial, por lo que es necesario mirar las hojas características del microcontrolador de la placa Pyboard para conocer los valores máximos absolutos. Dependiendo de si es entrada o salida, se especificará este valor con dos o tres bytes respectivamente.

La clase de entrada analógica recibe el nombre de ADC de las siglas *Analog to Digital Converter*, mientras que la de salida recibe el nombre de DAC de las siglas *Digital to Analog Converter*.

Es evidente que el tratamiento entre la entrada y la salida es digital. Por lo tanto, el tratamiento de valores se hará mediante muestreo. Esto implica que existe un periodo de muestreo determinado entre la captura de una muestra y la siguiente.

Para conectar los pines a la placa se han comprado 8 hileras de 8 pines macho-hembra. Se han cortado, para que encajen con las hileras de la placa, y soldado.

7.1. GENERADOR DE SEÑALES

En este punto se desea programar una salida sinusoidal que funcione como un objeto y que tenga ciertos métodos para activar o desactivar, cambiar la amplitud y frecuencia entre otros.

Para generar señales es necesario utilizar la clase DAC del módulo pyb ("pyb.DAC"). El método de dicho objeto que parece más sencillo y suficiente es el de escribir un valor determinado, en un instante determinado, en la salida, tal y como muestra la Fig. 7.1. Se podría escribir un programa en Python que jugase con el valor que hay en la salida en cada instante.

```
dac = pyb.DAC(pin)
dac.write(valor)
```

Fig. 7.1: Creación objeto DAC y definición de valor en la salida.

Lo primero que se ha intentado es generar un cambio de valor en la salida del objeto DAC mediante el uso del método *callback* en un temporizador. El problema con el que



nos encontramos siempre es la incapacidad de asignar memoria de dicha función. Afortunadamente, la clase DAC tiene otro método (Fig. 7.2) preparado para generar señales periódicas a partir de los siguientes atributos: un *buffer* de valores, una frecuencia y el modo de ejecución.

```
self.dac.write_timed(buffer, frecuencia, mode=DAC.CIRCULAR)
```

Fig. 7.2: Método del objeto DAC para señales frecuenciales.

En el buffer se especifican los valores o muestras que irá escribiendo en la salida analógica. Estos valores se expresan en 8 bits por lo que están comprendidos entre 0 y 255, que equivalen a valores de 0 a 3,3V. Por lo tanto, debe contener un número determinado de valores resultado de calcular el seno entre 0 y 360°. Esto se puede hacer automáticamente con un bucle finito de X iteraciones que genere X muestras. Siendo “len(buffer)” igual al número de muestras, la función para generar dicho buffer se muestra en la Fig. 7.3.

$$buffer[i] = 128 + \text{int}\left(127 \cdot \sin\left(\frac{2 \cdot \pi \cdot i}{\text{len}(buffer)}\right)\right)$$

Fig. 7.3: Fórmula para el muestreo de una señal sinusoidal de salida.

En la Fig. 7.4 se muestra el código equivalente a la fórmula de la Fig. 7.3. Nótese que la resolución corresponde al número de muestras. Es, por lo tanto, un parámetro de calidad de la señal.

```
def createBuffer(amplitud,resolucion):
    buffer = bytearray(resolucion)
    for i in range(len(buffer)):
        buffer[i] = 128 + int(amplitud*sin(2*pi*i/len(buffer)))
    return buffer
```

Fig. 7.4: Código para el muestreo de una señal sinusoidal de salida.

La frecuencia que se especifica en el método “*write_timed*” es la de iteración de muestras, por lo que la frecuencia de oscilación del seno será igual a la frecuencia especificada dividida entre el número de muestras. Esto se debe tener en cuenta si se quiere especificar directamente la frecuencia del seno en un programa que genere señales. El modo de ejecución *circular* nos permite cerrar el *buffer* de modo que una vez acabe vuelva a empezar (cola circular/fifo buffer).

Esta utilidad se puede programar bien como una función a la que se pueda llamar, como un objeto clase o simplemente como el cuerpo del programa principal en un bucle infinito. Se ha optado por crear un objeto clase llamado “sinGenerator”. En un principio se ha intentado que ésta herede los atributos y métodos de la clase DAC, pero se han tenido

problemas. Sin embargo, una buena alternativa es crear un objeto DAC dentro del objeto sinGenerator.

Los métodos básicos por lo tanto del objeto sinGenerator son activar y desactivar. Los atributos son la amplitud del seno, la frecuencia, el pin de salida, la resolución y el buffer. En la Fig. 7.5 se muestra el código para esta nueva clase creada.

```
class sineGenerator():
    def __init__(self, amplitud, frecuencia, pin, resolucion):
        self.dac = pyb.DAC(pin)
        if amplitud > 128:
            amplitud = 128
        self.amplitud = amplitud
        self.frecuencia = frecuencia
        self.pin = pin
        self.buffer = createBuffer(amplitud, resolucion)
        self.resolucion = resolucion

    def activar(self):
        self.dac.write_timed(self.buffer,
        self.frecuencia * self.resolucion, mode=DAC.CIRCULAR)

    def desactivar(self):
        self.dac.write(0)
```

Fig. 7.5: Código de la clase sineGenerator: generador de salida sinusoidal.

Se ha comprobado el funcionamiento del programa con un osciloscopio y funciona correctamente hasta alcanzar la frecuencia máxima de 44 kHz. Esta frecuencia es la de cambio de byte en el buffer que almacena la señal muestreada.

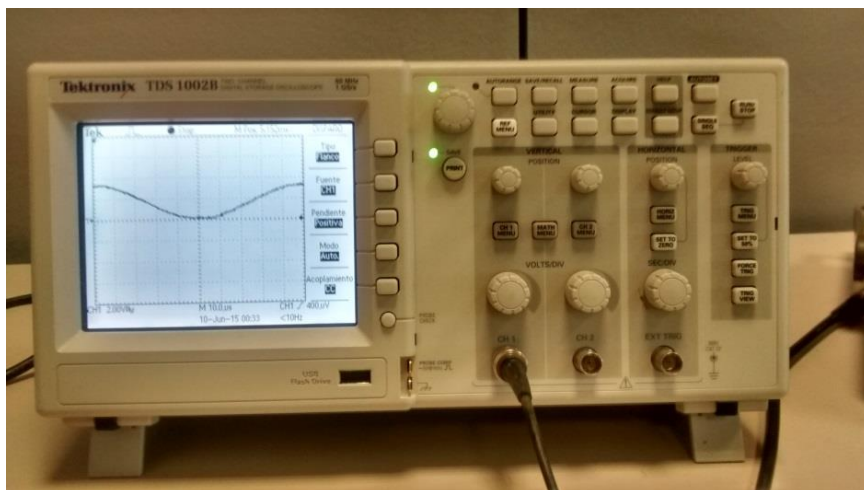


Fig. 7.6: Señal captada con el osciloscopio.



7.2. PUENTE VIRTUAL ENTRE ENTRADA Y SALIDA

Una forma de analizar los retardos a la hora de trabajar con las entradas analógicas y salidas analógicas es establecer un puente virtual entre ellas. Esto significa capturar el valor en una entrada y generar una salida del mismo valor. No existe una función predeterminada que establezca un vínculo de este tipo. Por lo tanto, se ha optado por utilizar un interruptor temporal *callback* que desarrolle la acción de capturar un dato en la entrada y guardarlo en la salida.

Para hacer esto se necesita un generador de señales y un osciloscopio. Para hacer el montaje se ha utilizado una protoboard. La Fig. 7.7 muestra este montaje.

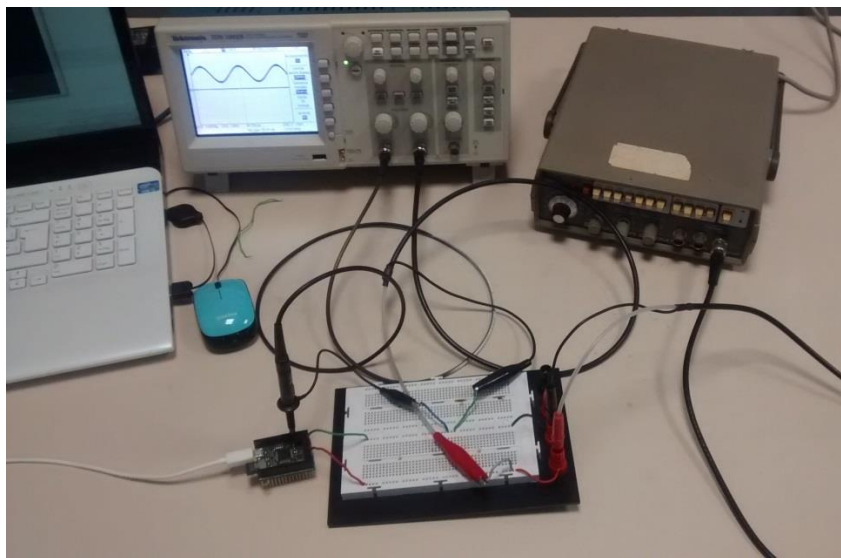


Fig. 7.7: Pyboard, osciloscopio y generador de señales conectados en una protoboard.

Vuelve a aparecer un problema a la hora de utilizar el intuitivo interruptor del temporizador. A base de prueba y error se ha comprobado que el problema radica en la transformación de la entrada (codificada en 12 bits) a la salida (codificada en 8 bits). Este paso se puede escribir de dos formas en Python. Se puede utilizar el código que se muestra en la Fig. 7.8.

```
valorSalida = int(valorEntrada*255/4095)
```

Fig. 7.8: Comando de cambio de valor de entrada a valor de salida poco eficiente.

O bien desplazando cuatro bits hacia la derecha como se observa en la Fig. 7.9.

```
valorSalida = valorEntrada >> 4
```

Fig. 7.9: Comando de cambio de valor de entrada a valor de salida eficiente.

El primero requiere asignación de memoria, lo que lo hace inviable para funcionar como la interrupción de un temporizador. Por lo tanto, se hubiera tenido que escribir el programa con la estructura de bucle infinito. Por consiguiente, no sería tan sencillo controlar la frecuencia de muestreo ya que no dependería del ciclo de un temporizador. Más adelante se plantean alternativas para que se pueda controlar la frecuencia de muestreo utilizando un bucle infinito.

El segundo ejemplo es perfectamente factible para el método *callback* en términos de memoria. Esto hace posible que pueda funcionar como un objeto independiente del hilo de ejecución principal, interrumpiéndolo a cada cambio de ciclo del temporizador.

Para visualizar el funcionamiento del programa se han hecho las conexiones entre el osciloscopio, el generador de señales y la Pyboard de modo que se puedan superponer la entrada y la salida. Así se puede mirar también el desfase entre las señales. Otro aspecto importante tiene relación con el teorema de muestreo de Nyquist en cuanto a que la velocidad de muestreo debe ser más grande que el doble de la frecuencia de la señal muestreada. De no ser así, se produce *aliasing* y el programa no funciona correctamente.

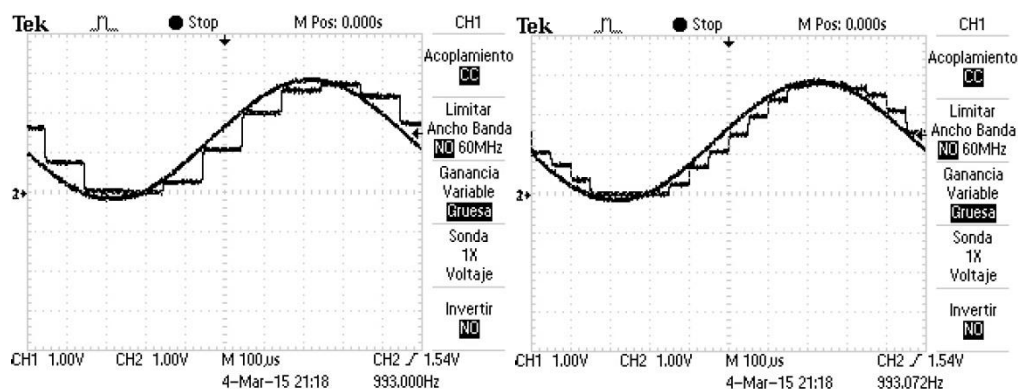


Fig. 7.10: Señales superpuestas (*adc_dac_v1*). Muestreo a 10khz y 20kHz respectivamente.

En la Fig. 7.10 se ilustran las señales de entrada y de salida (escalonada) para 10 kHz y 20 kHz de frecuencia de muestreo. Estas imágenes corresponden a la versión de programa escrito como objeto clase con atributos y métodos. Las siguientes líneas de código muestran cómo se ha escrito dicho objeto clase y cuáles son sus métodos y atributos.



```

class adc_dac:
    def __init__(self, sampleFreq, pinEntrada = pyb.Pin.board.X19,
pinSalida = 1):
        self.adc = pyb.ADC(pinEntrada)
        self.dac = pyb.DAC(pinSalida)
        self.timer = pyb.Timer(1)
        self.timer.init(freq = sampleFreq)

    def activar(self):
        self.timer.callback(self.readAndWrite)

    def desactivar(self):
        self.timer.callback(None)
        self.dac.write(0)

    def readAndWrite(self, timer):
        valor = self.adc.read()
        valor = valor >> 4
        self.dac.write(valor)

```

Fig. 7.11: Programa de puente “entrada-salida analógicas” con objeto clase (adc_dac_v1).

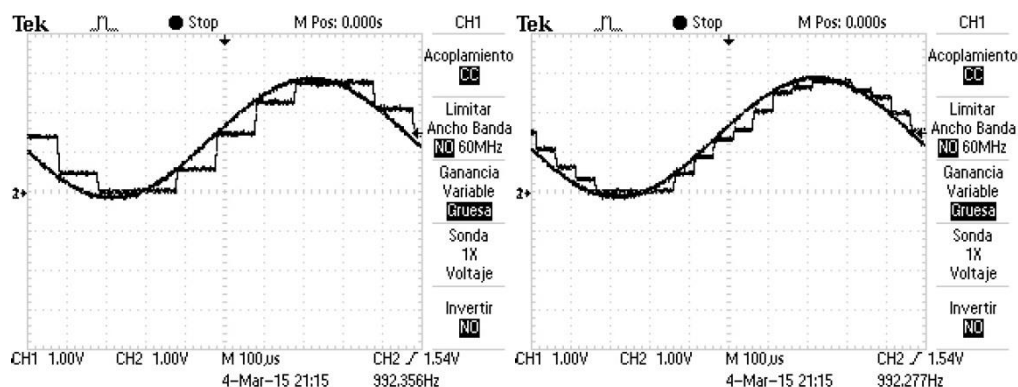


Fig. 7.12: Señales superpuestas (adc_dac_v2). Muestreo a 10kHz y 20kHz respectivamente.

La Fig. 7.12 muestra la misma utilidad con las mismas frecuencias pero inicializado al reinicio de la Pyboard. Se puede ver que no hay diferencia notable de retraso escrito de una forma u otra. La Fig. 7.13 muestra los comandos correspondientes a la Fig. 7.12.

```

pinEntrada = pyb.Pin.board.X19
pinSalida = 1
timer = 1
sampleFreq = 10000

timer = pyb.Timer(timer)
timer.init(freq = sampleFreq)
adc = pyb.ADC(pinEntrada)
dac = pyb.DAC(pinSalida)
timer.callback(lambda t: dac.write(adc.read() >> 4))

```

Fig. 7.13: Programa de puente entrada-salida analógicas código directo (adc_dac_v2).

La frecuencia de 20 kHz es (al margen de pequeñas variaciones) la frecuencia máxima de muestreo. A partir de esta frecuencia, se bloquea la Pyboard. Esto se debe al hecho de que el programa tarda siempre un mínimo de tiempo en muestrear y puentear los valores de la entrada a la salida. Este tiempo es el periodo más pequeño que se pueda conseguir en la iteración y, por lo tanto, marca la frecuencia máxima de iteración.

En las figuras Fig. 7.10 y Fig. 7.12, se observa como los escalones van justo por detrás de la sinusoidal. El desfase es prácticamente inmensurable a esta escala para ambos casos. Se puede decir que la constitución del programa como objeto clase no es menos eficiente que escribir el código directamente como el cuerpo de un programa.

7.3. PUENTE VIRTUAL CON DESFASE

Siguiendo el ejemplo anterior, se escoge añadir un retraso en muestras entre la salida y la entrada. Para hacer esto se debe crear una lista de determinada longitud que sirva de cola circular, donde se acumulen las muestras de la entrada. Para generar el retraso se ha optado por llenar la cola inicialmente con valores nulos. Esto genera un retraso en muestras equivalente al número de ceros que se introduzcan en la cola.

La forma más intuitiva en Python para hacer esta cola sería utilizar la clase Queue. No obstante, esta clase no está implementada en Micro Python. Por lo tanto, la solución pasa por crear una lista vacía e ir añadiendo y sacando valores (Fig. 7.14). También se puede utilizar una cadena de bytes (bytearray).

```
lista = []  
lista += valorEntrada  
valorSalida = lista.pop(0)
```

Fig. 7.14: Cola circular.

El tratamiento de la cola impide que el programa se pueda escribir utilizando el interruptor de un temporizador, ya que requiere asignación de memoria. Por lo tanto, se ha decidido escribirlo como un bucle infinito sin controlar la frecuencia de muestreo, tal y como se muestra en la Fig. 7.15.



```
import pyb

adc = pyb.ADC(pyb.Pin.board.X19)
dac = pyb.DAC(1)
lista = []

for i in range(desfase):
    lista+=[0]

while True:
    lista+=[adc.read()>>4]
    dac.write(lista.pop(0))
```

Fig. 7.15: Programa de puente entre salida y entrada con desfase.

En el siguiente capítulo se indican algunas soluciones para poder controlar la frecuencia en procesos de este tipo.

7.4. CONCLUSIONES

Para concluir este apartado se tienen las siguientes consideraciones:

- Hasta la fecha, el método de interrupción que utilizan tanto los temporizadores como el pulsador de usuario, no puede asignar memoria. Esto supone un problema a la hora de iterar procesos mínimamente complejos a una frecuencia controlada.
- La frecuencia máxima de cambio de valor, para las funciones de lectura y escritura de valores analógicos, es de 44 kHz. Esta frecuencia está limitada por el propio conversor A/D y D/A.
- Si se quiere puentear la entrada con la salida de valores analógicos, la frecuencia máxima es de unos 20 kHz.

8. FRECUENCIA DE ITERACIÓN DE PROCESOS

Anteriormente, se han visto casos en los que el método de interrupción *callback* no era capaz de desempeñar determinados procesos debido a su incapacidad de asignar memoria. En este punto se explican dos alternativas que se han elaborado y comprobado.

8.1. ALTERNATIVA 1

Normalmente podría no interesar una gran precisión de frecuencia en los procesos que se quieran repetir periódicamente. En ese caso bastaría con hacer esperar al procesador un tiempo antes de repetir dicho proceso.

No obstante, es posible que la duración de un proceso cambie y provoque una gran variabilidad en la frecuencia de su iteración. Por ello, interesaría encontrar un método que mantenga la frecuencia de iteraciones sin importar el tiempo que tarde el proceso.

La forma más sencilla para hacer esto consiste en un contador de microsegundos que justo antes de empezar un proceso se ponga a cero y que mida el tiempo justo al acabar dicho proceso. Después de este proceso se ejecuta un comando de retardo *pyb.delay(n)*, donde “*n*” es el tiempo que le falta al temporizador para cambiar de ciclo. Cuando el retardo finaliza, todo el proceso vuelve a empezar.

Este último método funciona cuando no hay interrupciones, ya que la CPU es *monothread*. El retardo (*pyb.delay*) mantiene ocupada la CPU y si se interrumpe el hilo, hay que sumarle el tiempo que tarda dicha interrupción. Esto se ha comprobado utilizando el depurador de tiempo para comparar un retardo de 3 segundos sin interrupciones y otro retardo de 3 segundos con interrupciones. Como ejemplo se han programado 10 interrupciones por segundo que cambian el estado de un LED. La Fig. 8.1 muestra las diferencias en el tiempo que han tardado los procesos 1 y 2.



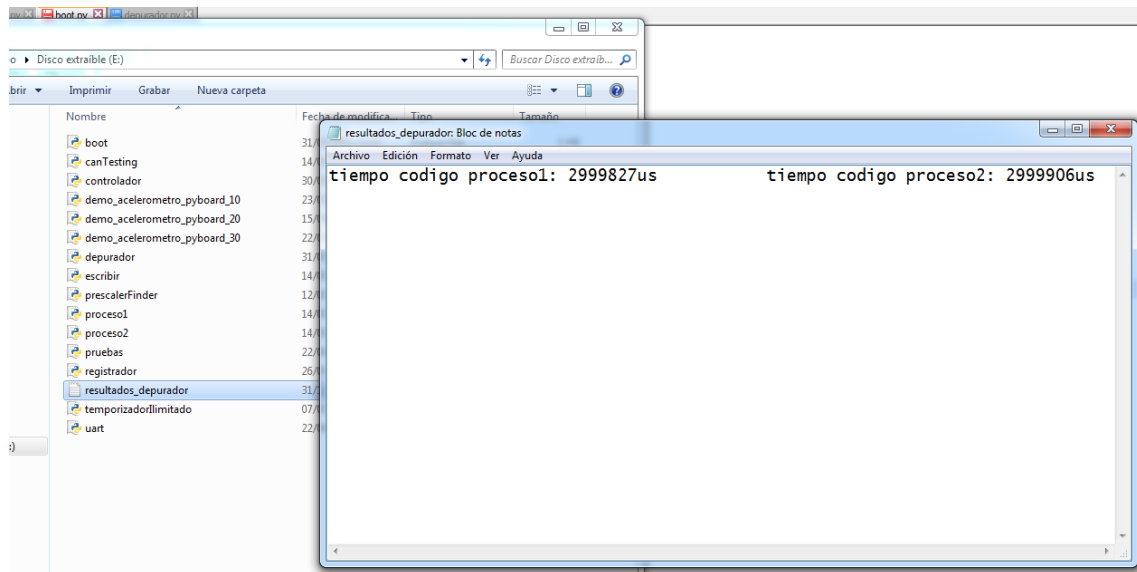


Fig. 8.1: Depurador para proceso iterativo con y sin interrupciones.

8.2. ALTERNATIVA 2

Este método impide que una interrupción pueda variar la frecuencia del proceso. Para ello es necesario un bucle de espera seguido por el proceso a iterar. Estos dos componentes se sitúan dentro de un bucle infinito.

Es preciso definir una serie de términos que se utilizarán para explicar el sistema que se ha desarrollado. Se utiliza la Fig. 8.2 para definir los conceptos:

- Banda de bloqueo (rojo): es la parte inicial del periodo de un temporizador en la que se cumple la condición del bucle de espera.
- Banda de paso (gris): es la parte final del periodo del temporizador en la que no se cumple la condición del bucle de espera.
- Tiempo de proceso (verde): Es el tiempo (variable) que tarda el proceso que se itera.
- Tiempo de espera (azul): es el tiempo en el que se cumple la condición de bloqueo y el puntero está atrapado en el bucle de espera.
- Inicio de ciclo (rosa): es el instante en el que el proceso acaba y empieza una nueva iteración del bucle infinito.
- Tiempo de ciclo: corresponde al tiempo de bloqueo más el tiempo de paso.

Para detectar si funciona correctamente, es decir, a frecuencia constante, se ha decidido crear un temporizador de 17 minutos (*micros*) en el que cada bit del periodo corresponde a un microsegundo. La consulta a este temporizador se hace justo después de acabar el tiempo de espera. El valor que dé `micros.counter()` se compara con el mismo valor de la iteración anterior (*counterMicros*). Esta comparación debe dar un número igual al periodo en microsegundos de la iteración del programa y debe ser el mismo en cada iteración (*periodIter*). Para comprobar que estos valores son los adecuados, se puede utilizar el comando `print()` para visualizarlos en el terminal (PuTTY). Estas modificaciones se pueden observar en la Fig. 8.4.

También se puede imprimir en pantalla el valor del estado del temporizador *timer*. Este valor debe ser siempre el mismo, ya que corresponde al mismo bit (o el mismo estado) del temporizador que marca la frecuencia de las iteraciones (*timer*).

```
timer = pyb.Timer(1)
timer.init(prescaler = 930, period = 20000)
micros = pyb.Timer(2, prescaler = 83, period = 0x3fffffff)
micros.counter(0)

while True:
    while timer.counter() < 19000:      #bucle de espera
        pass                          #bucle de espera
    periodIter = micros.counter() - counterMicros
    counterMicros = micros.counter()
    pyb.LED(2).toggle()
    pyb.delay(lista[n-1])
    n = n%10+1
    print(timer.counter(), ' ', counterMicros, ' ', periodIter)
```

Fig. 8.4: Código de comprobación para el programa de iteración a frecuencia constante.

Uno de los posibles errores es que el programa se salte alguna iteración y se solape con la siguiente, esperando 2 tiempos de ciclo. También es posible que varias iteraciones ocurran en un mismo tiempo de paso. Por eso es imprescindible calibrar bien los valores de *prescaler* y *period* del temporizador *timer*, además de la condición de paso del bucle de espera.

Para hacer esto hay que saber cuánto tiempo puede tardar el proceso principal y, si éste tiene variabilidad, conocer los tiempos máximos y mínimos. Obviamente, si se configura para una frecuencia demasiado grande, el programa no funcionará adecuadamente. Utilizando el depurador, explicado en el punto 6 de este documento, se puede calcular el tiempo que tardan los comandos que componen el proceso principal para configurar los valores del temporizador *timer*.

```

counterMicros = micros.counter()
periodIter = 0
n=0

timer = pyb.Timer(1)
timer.init(prescaler = 930,period =20000)

def proceso1():
    global periodIter, counterMicros, timer, n
    periodIter = micros.counter()-counterMicros
    counterMicros = micros.counter()
    print(timer.counter(), ' ', counterMicros, ' ', periodIter)
    pyb.LED(2).toggle()
    n = n%10+1

def proceso2():
    global periodIter, counterMicros, timer, n
    pass

```

Fig. 8.5: Código para la comprobación del tiempo que dura el proceso principal.

La Fig. 8.5 muestra cómo utilizar el programa depurador para calcular el tiempo de proceso. Antes de llamar a los procesos hay que crear las variables con las que trabajan y nombrarlas como variables globales dentro de las dos funciones (*proceso1* y *proceso2*). El temporizador *micros* ya existe en el depurador, por lo que no es necesario crearlo por segunda vez. Nótese que el código anterior a las funciones no corresponde al programa entero del depurador. Únicamente muestra los procesos y las variables que deben inicializarse.

El “*proceso1*” tarda 112µs a lo que se le deben sumar 100ms que será el máximo valor de la variabilidad dada por la lista *listaRetardos*. La frecuencia máxima que permite este periodo es de unos 10Hz. Por precaución se escoge utilizar una frecuencia de unos 9Hz con valores de *prescaler* = 930 y *period* = 20000.



Figura 8.2.1.1

	A	B	C	D	E	F
7						
8		INFO				
9		clock (MHz)	84	84	168	168
10		periodo máximo	262143	1073741823	262143	1073741823
11		temporizadores	3,4,6,7,12,13,14	5,2	1,8,9,10,11	-
12		period	20000	1073741823	20000	10000
13		prescaler	91	2147483647	930	40000
14		freq (s-1)	913043,4783	0,03911554813	180451,1278	4199,895003
15		periodo ciclo (s)	0,0219047619	27450511989	0,1108333333	2,381011905
16		frec ciclo	45,65217391	0	9,022556391	0,4199895003
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						
31						
32						
33						
34						
35						
36						

Fig. 8.6: Captura de pantalla de la ejecución del programa y valor del programa.

La Fig. 8.6 muestra la hoja de cálculo para establecer los valores del temporizador *timer* y el terminal con los valores descritos anteriormente. La tercera columna muestra el valor de la variable *periodIter*. Se puede ver como cada cierto número de iteraciones aparece un valor de periodo muy pequeño (145). Este valor implica que en esa iteración el proceso principal se itera dos veces. Este defecto ocurre por no estar bien calibrada la condición de paso del bucle de espera. En el ejemplo anterior (Fig. 8.4), esta condición está ajustada a un valor de 19000. Si se substituye este valor por 19900, funciona correctamente.

8.3. CONCLUSIONES

Se puede solucionar el problema de los temporizadores con las alternativas que se han propuesto en este apartado. No obstante, estos ejemplos solo sirven para iterar un proceso a la vez, mientras que el método de interrupción *callback* puede funcionar para varios temporizadores simultáneamente.

9. BUS CAN

Antes de utilizar el bus CAN en la Pyboard se ha tenido que estudiar su funcionamiento utilizando un manual explicativo de la empresa Bosch [5] y una presentación de diapositivas de la universidad de Múrcia [6].

9.1. MONTAJE CON TRANSCEPTORES

La Pyboard tiene incorporadas 2 conexiones de bus CAN. No tiene incorporados, sin embargo, los transceptores. Para probar el funcionamiento del bus CAN se ha optado por comprar los transceptores de 3,3V (teniendo en cuenta la tensión de la Pyboard). Se han soldado a una placa de tamaño reducido con 8 pines o conectores para cada una de las patas del transceptor.

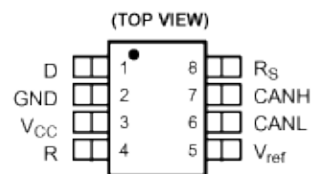
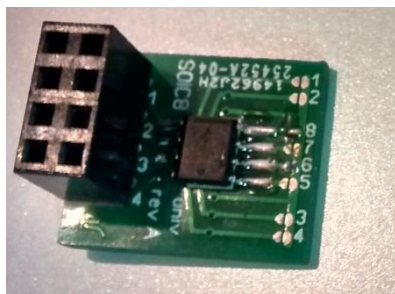


Fig. 9.1: *Izquierda: transceptor con placa y acoplador. Derecha: Esquema del transceptor.*

Las conexiones se hacen de los transceptores a los respectivos pines de bus CAN de la Pyboard. Y de transceptor a transceptor con los dos cables trenzados del bus can (CAN_HIGH y CAN_LOW). Las características de los transceptores se han obtenido de su *datasheet* [7].

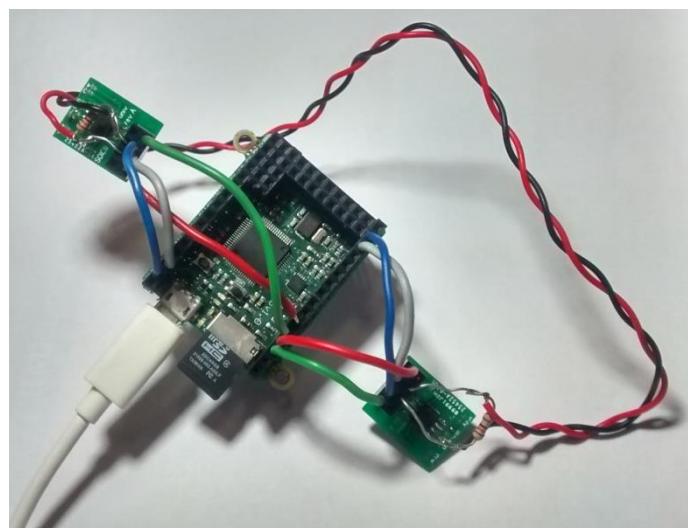


Fig. 9.2: Conexión entre los buses CAN1 y CAN2.

La clase objeto `pyb.CAN` tiene varios modos y atributos que se deben caracterizar a la hora de construir un objeto de este tipo. El atributo `mode` nos permite definir qué tipo de funcionamiento tendrá respecto a los filtros de direcciones. Además uno de estos modos (`CAN.LOOPBACK`) permite enviar y recibir sobre sí mismo con el simple motivo de hacer pruebas sin disponer de un bus CAN real.

La primera prueba que se ha hecho ha sido con el modo “`CAN.LOOPBACK`” y ha funcionado correctamente. El puerto CAN ha enviado y recibido el mismo mensaje correctamente escribiendo las instrucciones desde el terminal PuTTY. La Fig. 9.3 muestra el código para esta prueba.

```
>>> from pyb import CAN
>>> can = pyb.CAN(1)
>>> can.send('message',123)
>>> can.recv(0)
b'message'
```

Fig. 9.3: Comandos de creación de objeto CAN, envío y recepción de datos.

En los siguientes experimentos se ha tratado de enviar y recibir de un puerto CAN al otro mediante el uso del terminal y de los transceptores pero no ha sido posible. Para hacer esto se han utilizado los comandos tal y como se muestran en la Fig. 9.4.

```
>>> from pyb import CAN
>>> can1 = pyb.CAN(1)
>>> can1.init(CAN.NORMAL, prescaler = 16, sjw = 1,bs1 = 14, bs2 = 6)
>>> can2 = pyb.CAN(2)
>>> can1.send('message',123)
>>> can2.recv(0)
```

Fig. 9.4: Comandos de comunicación entre CAN1 y CAN2.

La configuración dada en la inicialización de cada bus de la Fig. 9.4 establece una velocidad de comunicación (*baudrate*) de 125000 bits/s. Los parámetros que se definen son comunes en el protocolo de bus CAN. Se ha elaborado un pequeño programa en una hoja de cálculo para definir el valor de *baudrate* en función de dichos parámetros. La diferencia entre ambos puertos es que el bus CAN1 tiene un reloj periférico de 42MHz mientras que el CAN2 lo tiene de 84MHz.

El resultado de los comandos utilizados en la Fig. 9.4 varía según el caso. Siendo a veces la generación de un error (`OSError: 116`) al recibir y en otros casos al enviar. Después de estas pruebas entre CAN1 y CAN2 se ha decidido empezar de nuevo con el modo `CAN.LOOPBACK` surgiendo esta vez el mismo error que en el modo normal.

Gracias al foro de Micro Python se ha descubierto que el error “*OSError 116*” es un error de *timeout*. Significa que no hay nada para recibir o que no hay ningún dispositivo al que enviar información. En uno de los casos en los que se ha programado el objeto CAN1 para enviar información, se han podido ver los pulsos del mensaje en el osciloscopio. Sin embargo el CAN2 no ha sido capaz de recibir la información. En los intentos posteriores no ha habido más pulsos ya que el error se ha generado no en la recepción sino tras el envío de información.

Llegados a este punto se ha verificado el montaje con un multímetro. El problema no parece ser el montaje, aunque sí que podría ser el transceptor.

9.2. PRUEBAS CON KVASER CANKING

Como alternativa se ha decidido utilizar un convertidor comercial de bus CAN a USB, y el programa “Kvaser CanKing”, para analizar la transferencia de datos desde el PC.

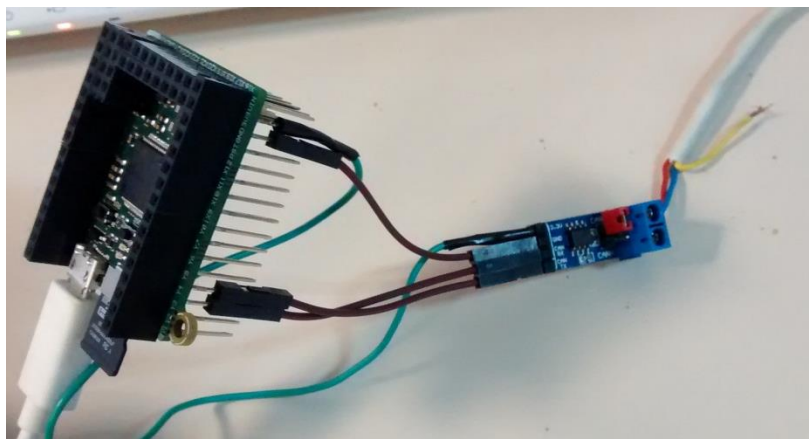


Fig. 9.5: Conexión de bus CAN entre placa y puerto serial del PC.

Los resultados indican que el CAN1 envía información que la computadora recibe satisfactoriamente. No obstante, el PC no es capaz de enviar datos a la placa sin generar ésta el mismo error *OSError 116*. La comunicación se ha establecido, como en los casos anteriores, a 125kbit/s.



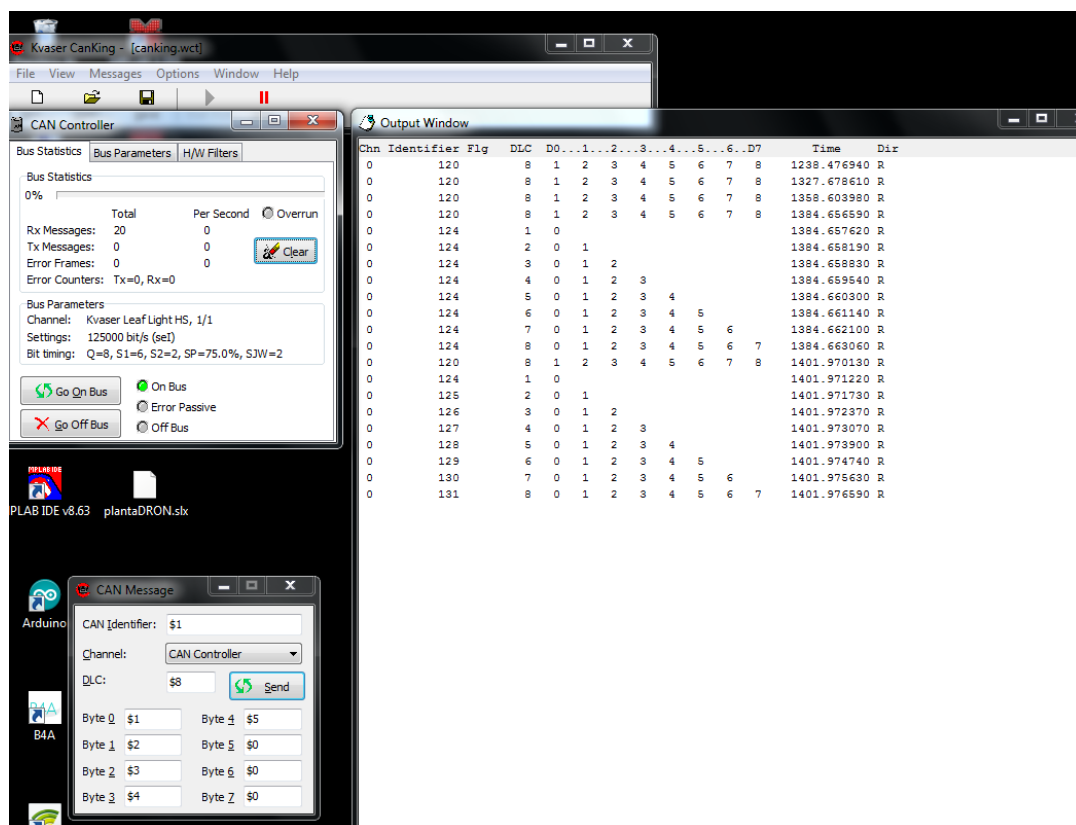


Fig. 9.6: Interfaz Kvaser CanKing.

En la Fig. 9.6 se muestra el programa utilizado y su interfaz. El “output window” muestra los datos recibidos desde la placa de Micro Python. Se envían hasta un máximo de 8 bytes por trama, cambiando los identificadores según la serie. El último conjunto de mensajes (con forma de rampa) corresponde al siguiente código de la Fig. 9.7.

```
c = bytearray()
for j in range(8):
    c.append(j)
    can.send(c, 124+j)
```

Fig. 9.7: Programa de ejemplo de envío de datos por el bus CAN.

Se ha comprobado cómo el uso del terminal PuTTY puede influir en que funcione correctamente o no el programa. En varias ocasiones se han intentado enviar datos desde el bus CAN. Si el terminal PuTTY ha estado conectado no se han podido enviar mensajes mientras que cerrando el terminal PuTTY si se han podido enviar. Sin embargo, esto no ocurre siempre por lo que ésta no es la causa principal del problema.

Finalmente se ha probado otra placa del mismo modelo para cerciorarse de que la Pyboard utilizada no es defectuosa o tiene algún componente estropeado. Los resultados han sido los mismos. Después de preguntar en el foro oficial de Micro Python, y dadas las

contestaciones de varios usuarios iniciados, se ha concluido que existe algún problema con el firmware. El tema en cuestión debatido en el foro de Micro Python se puede encontrar en el siguiente enlace:

<http://forum.Micro Python.org/viewtopic.php?f=2&t=619>.

9.3. CONCLUSIONES

La clase CAN no funciona correctamente hasta la fecha. Tanto el CAN1 como el CAN2 pueden enviar datos, pero no pueden recibirlos.



10. APLICACIÓN ACELERÓMETRO

En este apartado se presenta el uso del acelerómetro de la Pyboard en una aplicación de carácter didáctico. Uno de los usos más comerciales que se dan últimamente a los acelerómetros es el desarrollo de aplicaciones y juegos en teléfonos móvil. También hay otras aplicaciones en la industria y en la robótica. Los drones son un buen ejemplo dónde podría ser útil. Convendría conocer el alcance del acelerómetro antes de pensar aplicaciones que uno mismo pudiera llevar a cabo con la Pyboard.

Uno de los factores a tener en cuenta es la precisión, la resolución y la velocidad máxima de transmisión de datos si se utiliza con una plataforma externa. Para visualizar estos factores se ha optado por crear un programa que se ejecute en el PC y que utilice un puerto serie para recibir los datos del acelerómetro de la placa. Por lo tanto tendremos dos partes del programa, una que se ejecuta en la Pyboard y la otra en el PC.

Para crear el programa en el PC se utiliza el lenguaje Python. Éste tiene una librería llamada PySerial, que permite conectarse a un puerto serie. Además, también se puede utilizar la librería de Pygame para escribir un programa rápidamente y con una interfaz gráfica atractiva.

La placa tiene varios medios de comunicación por los que se podría conectar al PC. Uno de ellos, el más directo de utilizar, es el puerto micro USB. Todos los protocolos de comunicación vía USB están agrupados en la clase USB_VCP de la Pyboard. Otro es el puerto UART. No obstante, se requiere un adaptador de UART-USB exterior para conectarse al PC.

En cualquier caso, la única función del programa que se localiza en la Pyboard es la de enviar datos constantemente a una frecuencia controlada.

10.1. DEMO USB

Este punto trata sobre el programa que envía datos desde la Pyboard vía USB. El primer problema que se ha encontrado es que no se pueden establecer dos objetos controlando el mismo puerto serie a la vez desde el PC. Por lo que para probar la librería USB no se puede abrir el terminal PuTTY. Esto conlleva que no se puedan depurar fácilmente los programas que se ejecuten.

Para enviar datos de la placa al ordenador a modo de prueba, se ha escrito un pequeño programa en la Pyboard que consiste en un bucle infinito con un retardo de 10 segundos entre un envío y el siguiente (Fig. 10.1).


```
serial = pyb.USB_VCP()

a = bytearray(2)
while True:
    a[0], a[1] = pyb.Accel().x(), pyb.Accel().y()
    a[0] += 40
    a[1] += 40
    serial.send(a)
```

Fig. 10.3: Programa de envío de datos para la demostración del acelerómetro.

Otro aspecto a tener en cuenta es la acumulación de mensajes en el PC. Esto se da cuando la frecuencia de lectura es menor a la frecuencia de emisión. El programa que ejecuta la demostración en el PC alcanza unas 60 iteraciones por segundo. Éste es un parámetro que depende de la velocidad del computador.

Haciendo uso del depurador explicado en el apartado 6 se obtiene un tiempo de 184.174µs para el proceso de enviar datos. Siendo éste el periodo, obtenemos una frecuencia de unos 5,4 iteraciones/segundo. Esta es muy inferior a los 60fps (*frames per second*, o iteraciones/segundo) del programa del PC. Por lo tanto, no se acumularán mensajes en el buffer del ordenador.

10.2. DEMO UART

En este caso se ha utilizado un adaptador UART-USB comercial para conectar la Pyboard al PC. La ventaja de utilizar este medio es que se pueden depurar los procesos de la Pyboard desde el terminal PuTTY a través del puerto USB. No obstante, se ha comprobado que en la utilización del terminal PuTTY se genera un mensaje de error al recibir datos del PC a la Pyboard, pero no así de la Pyboard al PC.

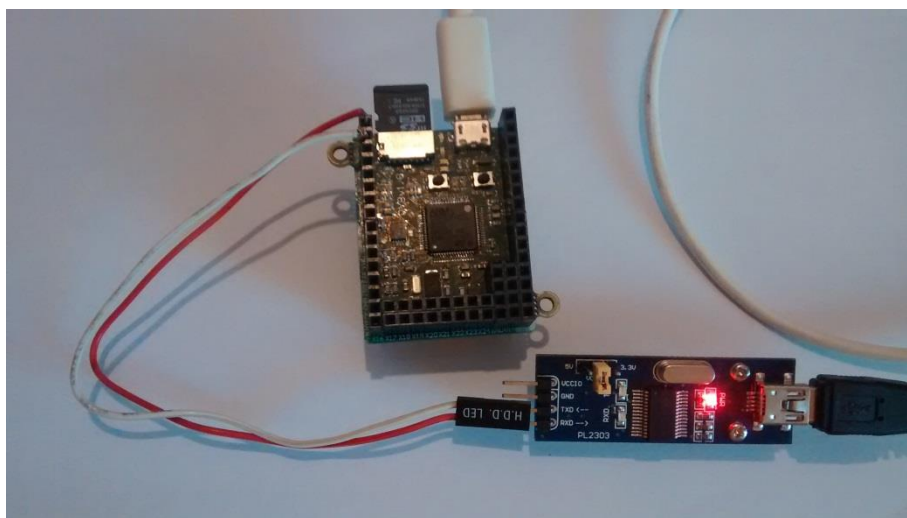


Fig. 10.4: Montaje de la Pyboard con adaptador UART-USB

El sistema basado en la UART funciona correctamente si se escribe el programa directamente en la memoria SD o la memoria flash y se ejecuta mediante un hard reset.

El mecanismo es exactamente el mismo que se utiliza con el USB. Sin embargo, en este caso se debe definir la velocidad de la señal y el número de bits por mensaje. La velocidad de transmisión es también del mismo orden. Se ha comprobado que la velocidad de transmisión de bits con la UART no modifica la frecuencia final de envío de datos. Esto se ha hecho calculando la frecuencia para una velocidad de transmisión de 9600 y de 115.200 bits/s. Ésta es una conclusión lógica ya que, de todo el tiempo que tarda entre un envío y el siguiente, solo una pequeña parte corresponde al envío de bits en sí.

La ventaja es, por lo tanto, la posibilidad de utilizar el terminal como depurador para algunos casos, aunque éste pueda dar problemas.

10.3. DEMO PC

Como ya se ha mencionado, el programa encargado de recibir información en el PC utiliza las librerías Pygame y Pyserial. Este programa consiste principalmente en un bucle infinito en el que cada iteración lee el buffer de entrada para tener los datos del acelerómetro. La demo consiste en una bola, sobre una superficie, que se mueve de acuerdo a la aceleración dada por el acelerómetro vista desde un plano superior. Por lo tanto, la bola no es más que la imagen de un círculo.



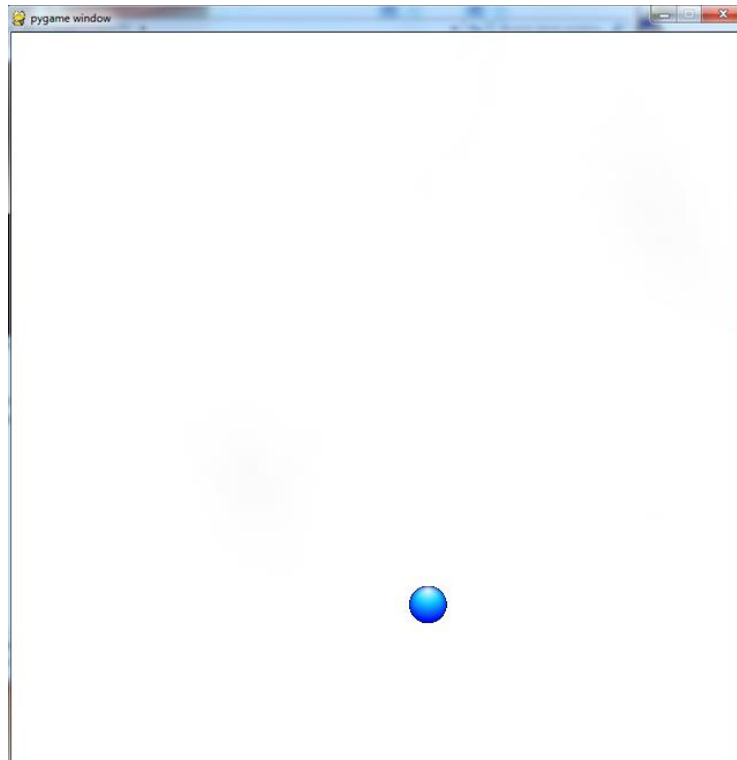


Fig. 10.5: Captura de pantalla de la demostración con el acelerómetro.

La primera versión que se ha hecho de esta demo consiste en sumar la velocidad a la posición de la bola en cada “frame”. Dicha velocidad es una propiedad de la bola que en este caso se ha igualado a la aceleración dada por el acelerómetro.

En la segunda versión se ha intentado equiparar el movimiento de la bola al movimiento real que se produciría si ésta estuviera sobre el plano XY de la Pyboard.

Para conseguir este movimiento más “realista” se ha integrado la aceleración para obtener las ecuaciones de velocidad y posición conociendo la aceleración (Fig. 10.6).

$$v = v_0 + a \cdot t$$

$$x = x_0 + v_0 \cdot t + \frac{a}{2} \cdot t^2$$

Fig. 10.6: Ecuaciones de la cinemática de una partícula.

Dado que el tiempo no es una variable fácilmente cuantificable y que la aceleración y la velocidad varían en el tiempo, las ecuaciones que se utilizan en el programa de estructura iterativa son las que se muestran en la Fig. 10.7.

$$v_k = v_{k-1} + a_k$$

$$x_k = x_{k-1} + v_{k-1} + \frac{a_k}{2}$$

Fig. 10.7: Ecuaciones discretas de la cinemática de una partícula.

Si en el “input” del programa se tiene un vector de dos componentes, éste pasa a ser la nueva aceleración de la bola. Si el “input” no contiene información significa que la Pyboard no ha tenido tiempo para enviarla y, por lo tanto, el buffer queda vacío.

Recuérdese que la frecuencia de iteración del programa del PC es superior a la de la Pyboard. En ese caso se da la orden de mantener la aceleración de la bola en el valor anterior hasta que haya un nuevo cambio de información. De acuerdo con la velocidad de transmisión de datos calculada en el punto anterior se obtendrán unas 5 muestras por segundo.

Esto último implica un cierto retardo en la bola al cambiar de dirección y magnitud de velocidad. Para pequeñas velocidades el retardo no es muy notable. Sí lo es para altas velocidades. También se puede hablar de precisión en el sentido de que estando la placa quieta, sus valores oscilan ligeramente. Esto ya se había comprobado al utilizar el modo HID de la Pyboard para controlar el cursor del mouse del ordenador.

10.4. CONCLUSIONES

Se ha visto como la comunicación de datos del acelerómetro, tanto vía USB como vía UART, tienen un máximo de frecuencia de unos 5 Hz.

Se ha comprobado que el método de captura de datos del acelerómetro puede dar errores puntuales, provocando el fallo del programa en el PC.



11. COMPILAR FUNCIONES EN C

El objetivo que se persigue en este capítulo es encontrar una manera de programar funciones en C que puedan ser utilizadas desde Micro Python. Esto puede ser útil cuando un programa escrito en Python es demasiado lento. Para estos casos existe la posibilidad de escribir en lenguaje ensamblador. No obstante, éste puede ser muy tedioso. Por eso se plantea C como la alternativa a Python y ensamblador en línea.

La primera idea que se investigó en esta línea fue el uso de CPython. En las diapositivas de presentación de Micro Python [8] se aclara que no es posible.

11.1. MICRO-CTYPES

En Python existe el módulo Ctypes, que permite utilizar funciones de librerías escritas en C y gestionar el paso de variables por valor o por referencia.

Micro Python tiene su homólogo uCtypes. En las librerías de la página oficial de Micro Python (<http://docs.Micro Python.org/en/latest/library/uctypes.html>) se listan los métodos de éste módulo pero con escasa y complicada explicación. Además, se requerirían librerías específicas para este microcontrolador. La falta de información hasta la fecha sobre este tema implica que no se haya profundizado más en este aspecto.

11.2. MODIFICACIÓN DEL FIRMWARE

Una opción para añadir nuevas funciones y módulos a Micro Python es compilar el código fuente con las nuevas funciones que se requieran, escritas en C.

11.2.1. Compilación del código fuente

El primera paso es descargar el código del portal GitHub [9] de Micro Python donde se encuentran también ejemplos, e incluso el texto del tutorial para Sphinx.

Las librerías necesarias para la compilación están disponibles para diferentes sistemas operativos. Se ha optado por utilizar Linux, ya que es el sistema operativo utilizado por los que acumulan más experiencia en Micro Python.

Los comandos, que se han seguido para preparar Linux para la compilación, se muestran en la Fig. 11.1. Las dos primeras líneas instalan las librerías necesarias mientras que la tercera arranca la compilación. Es posible que la segunda línea no sea necesaria, dependiendo de la versión de Linux. En este caso (trabajando con Linux Mint 17) sí que ha sido necesaria ya que de lo contrario se generan diferentes errores en la compilación.

Para que funcione es necesario que el comando **make** se escriba en el directorio */stmhal/* de la carpeta donde se aloja el código de Micro Python. En esa carpeta está alojado el archivo *Makefile* que es el archivo de inicio de la compilación.

```
sudo apt-get install gcc-arm-none-eabi
sudo apt-get install libnewlib-arm-none-eabi
make
```

Fig. 11.1: Comandos de preparación de Linux y compilación del código fuente

Una vez acabada la compilación, se generan varios archivos. El archivo compilado se aloja automáticamente en una carpeta con nombre "build-PYBV10". Éste se encuentra en extensión ".hex" y ".dfu". La versión necesaria para "flashear" la Pyboard es la segunda.

11.2.2. Incorporación de una función propia al firmware

En este apartado se han seguido varios caminos erráticos por falta de conocimiento básico sobre el tema. También cabe destacar la falta de información y que ésta, está dirigida a usuarios con un nivel muy avanzado.

Se ha empezado con la familiarización de los archivos que componen el código fuente. Buscando información, en algunos casos, sobre el lenguaje C para poder entender mejor el funcionamiento.

Se han seguido ejemplos, como la creación de la clase LED, por todos los archivos del código fuente. Hay archivos que declaran objetos que luego se utilizarán para construir la clase LED. Hay archivos que simplemente le dan un nombre a las clases que sean utilizadas por Micro Python. Después de divagar por los archivos se ha conseguido concluir, gracias al Wiki de Micro Python [10], en lo que puede servir de introducción a la incorporación de clases en el módulo PYB.

Se han creado los archivos *goc.c* y *goc.h* dentro del directorio */stmhal/*. En el archivo *goc.c* se han escrito las líneas de la Fig. 11.2.

```
#include "py/runtime.h"

const mp_obj_type_t pyb_goc_type = {
    {&mp_type_type }
    .name = MPQSTR_GOC
};
```

Fig. 11.2: Declaración de objeto GOC en goc.c

Es necesario añadir el archivo *goc.c* al *stmhal/Makefile* para que lo incorpore en la compilación (Fig. 11.3).



```
100
101 SRC_C = \
102     goc.c \
103     main.c \
104     system_stm32f4xx.c \
```

Fig. 11.3: Adición de goc.c al archivo Makefile.

También es necesario darle un nombre en el entorno de Micro Python. Esto se ha hecho añadiendo `Q(GOC)` al archivo `stmhal/qstrdefsport.h` como muestra la Fig. 11.4.

```
728 Q(WWDG_CR)
729 Q(WWDG_SR)
730
731 Q(GOC)
```

C++ source file

Fig. 11.4: Asignación del nombre GOC.

Una vez hecho esto, se ha compilado como se explica en el apartado anterior y se ha instalado el nuevo software en la Pyboard. Se ha comprobado que lo único que permite hacer es llamar a la función `pyb.GOC` sin poder crear una instancia haciendo `pyb.GOC()` sobre una variable. Al escribir `pyb.GOC` en Micro Python nos retorna `<class 'GOC'>`.

11.3.CONCLUSIONES

Dada la dureza de la información que se puede encontrar online sobre programación en C para Micro Python, lo que se ha conseguido no es gran cosa. La falta de conocimiento sobre el funcionamiento de microcontroladores y del lenguaje en C representa un hándicap importante. No obstante, las herramientas existen y pueden ser muy útiles para un usuario experimentado en el futuro.

CONCLUSIONES

Micro Python ofrece algunas de las ventajas que tiene Python frente a lenguajes de más bajo nivel. Permite la creación rápida y sencilla de programas a costa de reducir la eficiencia y velocidad en algunos de ellos, sobre todo los más complejos. En estos casos cuenta con ensamblador en línea útil para procesos en los que la velocidad es crítica.

El intérprete de Micro Python se hace viable gracias a la alta velocidad de reloj (168 MHz) de la Pyboard. Por lo tanto, Python, en su versión reconstruida y adaptada, se convierte en un lenguaje prometedor para microcontroladores, capaz de desarrollar infinidad de aplicaciones y utilidades.

Los usuarios con mucha experiencia en microcontroladores y programación, pueden involucrarse fácilmente en el desarrollo de Micro Python, ya que el código fuente es de fácil acceso. Un usuario de este tipo podría también personalizar su propio firmware añadiendo clases en C compiladas con relativa poca dificultad, que podrían ser llamadas desde Micro Python. De este modo se podrían solventar también los problemas que pudiera haber al utilizar el intérprete.

Los usuarios con conocimientos únicamente de Python, tienen la oportunidad de experimentar, introducirse y crear aplicaciones con gran facilidad.

Por lo tanto, los límites de la *Micro Python board v1.0* están marcados, no solo por el hardware, sino también por la experiencia y el conocimiento que se tenga en microcontroladores y lenguajes de programación de bajo nivel.

La comparación que se puede hacer frente a otro tipo de productos funcionando con lenguajes de más bajo nivel, es que Micro Python combina al mismo tiempo la programación en alto y bajo nivel, ofreciendo las ventajas propias de cada caso.

El aspecto negativo, y ciertamente importante, es que Micro Python está todavía en fase de desarrollo y se pueden encontrar defectos en el software dejando ciertas funcionalidades inoperativas o con deficiencias tales como la recepción de mensajes por bus CAN o el defecto que se ha visto en el registrado iterativo de datos del acelerómetro.

AGRADECIMIENTOS

Agradezco este trabajo especialmente a Manuel Moreno Eguilaz por haber mostrado disponibilidad y por mostrarse siempre positivo en cualquier coyuntura. También muestro mi agradecimiento a Josep Vilaplana por mostrarse considerado e ilustrarme en conceptos que escapaban de mi entendimiento. Por último quiero dedicar este trabajo a mi padre por darme ánimos y carácter en los momentos difíciles.

BIBLIOGRAFÍA

REFERENCIAS BIBLIOGRÁFICAS

- [1] DAMIEN GEORGE, *Micro Python*, 2014. <http://Micro Python.org/>. Fecha último acceso: 5 de Agosto de 2015. (Página oficial de Micro Python. Tutorial y documentación).
- [2] STMICROELECTRONICS, *Página Web Oficial*, 2015. <http://www.st.com/web/en/home.html>. Fecha acceso: 27 de Agosto de 2015 (Descarga Dfuse. Descarga de Manual).
- [3] Powered by phpBB® Forum Software © phpBB Limited. Style We_universal created by INVENTEA. <http://forum.Micro Python.org/>. Fecha último acceso: 5 de Agosto de 2015. (Foro oficial de Micro Python).
- [4] STMicroelectronics, *RM0008 Reference manual*. http://www.st.com/web/en/resource/technical/document/reference_manual/CD00171190.pdf. Fecha acceso: 15 Junio 2015. (Propiedades temporizadores, páginas consultadas: 293, 361, 419 y 461).
- [5] F. HARTWICH, ARMIN BASSEMIR, *The Configuration of the CAN Bit Timing*. 4 de Noviembre de 1999. http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/cia99paper.pdf. Fecha acceso: 20 Abril 2015. (Funcionamiento del bus CAN).
- [6] UM, Universidad de Murcia. *Introducción al bus CAN*. 26 de Noviembre de 2009. <http://ocw.um.es/ingenierias/sistemas-embedidos/material-de-clase-1/ssee-da-t03-02.pdf>. Fecha acceso: 14 Mayo 2015. (Funcionamiento del bus CAN).
- [7] TEXAS INSTRUMENTS, *SN65HVD23x 3.3-V CAN Bus Transceivers (Rev. N)*. Marzo 2001, http://www.ti.com/product/SN65HVD232/datasheet/pin_configuration_and_functions. Fecha acceso: 8 Mayo 2015. (Conexiones transceivers).
- [8] DAMIEN GEORGE. *Micro Python: Shrinking Python down to run on a microcontroller*. 20 de Setiembre de 2014. <http://Micro Python.org/static/resources/pyconuk14->

[mpinternals.pdf](#). Fecha consulta: 29 de Junio de 2015. (Pag. 5. Uso de CPython o PyPy).

[9] DAMIEN GEORGE, *Micro Python Source*. <https://github.com/MicroPython/MicroPython> Fecha acceso: 30 Julio 2015. (Descarga de código fuente para compilar).

[10] PETER HINCH, *Micro Python Wiki*, Agosto de 2015. <http://wiki.MicroPython.org/Home>. Fecha acceso: 25 de Julio de 2015. (Apartado consultado: *Creating pyb modules*).



ANEXO

A1. CÓDIGO: BUCLE EN ENSAMBLADOR.

```
@Micro Python.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_BSRR])

    # 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)
```

A2. CÓDIGO: FIND_PRESCALER_PERIOD

```
import pyb

def p(n): #n es el número id del temporizador
    period = 200
    #prescaler = 200
    k = 9
    kFactor = 10
    tim = pyb.Timer(n)
    p = True

    #bucle para encontrar el número de cifras que tiene el prescaler
    máximo
    while p:
        try:
            tim.init(period = period, prescaler = int(k))
            #tim.init(period = int(k), prescaler = prescaler)
            k+=9*kFactor
            kFactor*=10
        except OverflowError:
            p = False
        print(k)
    # p es un numero format per 9's (ej: 9999)
    p = k/2

    #bucle para encontrar el número aproximado de (k) teniendo en
    cuenta que puede haber un defecto a causa de los decimales
    while p>=1:
        try:
            tim.init(period = period, prescaler = int(k))
            #tim.init(period = int(k), prescaler = prescaler)
            k+=p
            print('>',int(k))
        except OverflowError:
            k-=p
            print('<',int(k))
        p = p/2
    k=int(k)

    #pequeño bucle para llegar al valor real, sin decimales
    p = True
    while p:
        try:
            tim.init(period = period, prescaler = k)
            #tim.init(period = int(k), prescaler = prescaler)
            p = False
        except OverflowError:
            k -=1
    return k
```



A3. CÓDIGO: FIND_FREQ

```
import pyb

tim = pyb.Timer(1)
tim.init(freq = 20000)
a=0

def led():
    global a
    if a>20000:
        pyb.LED(2).toggle()
        a=0
    else:
        a+=1
tim.callback(lambda t:led())
```

A4. CÓDIGO: DEPURADOR

```
import pyb

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

def procesol():
    pass

def proceso2():
    pass

pyb.LED(4).on()

startMicros = micros.counter()
procesol()
endMicros = micros.counter()
tiempoProcesol = endMicros-startMicros

startMicros = micros.counter()
proceso2()
endMicros = micros.counter()
tiempoProceso2 = endMicros-startMicros

text = open('/sd/resultados_depurador.txt','w')
text.write('tiempo codigo procesol: ' + str(tiempoProcesol) + 'us'+
'+ 'tiempo codigo proceso2: ' +str(tiempoProceso2)+'us')
text.close()

pyb.LED(4).off()
```

A5. CÓDIGO: DEMO ACELERÓMETRO PC

```
import serial, string, pygame, sys
from pygame.locals import *
from math import *

pygame.display.set_caption('Accel Game')
pantalla = pygame.display.set_mode((800,800))

ser = serial.Serial()
ser.port = 'COM3'
ser.open()
ser.timeout = 0

sprites = pygame.sprite.Group()
bola1 = pygame.sprite.Sprite()
bola1.image = pygame.image.load('images/bola.png').convert_alpha()
bola1.rect = bola1.image.get_rect()
bola1.rect[0] = 380
bola1.rect[1] = 380
sprites.add(bola1)

fondo = pygame.Surface((800,800))
fondo.fill((0,0,0))
fpsTime = pygame.time.Clock()
fps = 50

velocidad = [0,0]
aceleracion = [0,0]

def input():
    for event in pygame.event.get():
        #SALIR
        if event.type == QUIT:
            quitGame()
        if event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                quitGame()

def updateScreen():
    pantalla.blit(fondo,(0,0))
    a = accel()
    aceleracion[0] = a[0]
    aceleracion[1] = a[1]
    if bola1.rect[0]+velocidad[1]>760:
        bola1.rect[0] = 760
        velocidad[1] = 0
        aceleracion[1] = 0
    elif bola1.rect[0]+velocidad[1]<0:
        bola1.rect[0] = 0
        velocidad[1] = 0
        aceleracion[1] = 0
    else:
        bola1.rect[0] += velocidad[1] + aceleracion[1]/2
```



```
if bola1.rect[1]+velocidad[0]>760:
    bola1.rect[1] = 759
    velocidad[0] = 0
    aceleracion[0] = 0
elif bola1.rect[1]+velocidad[1]<0:
    bola1.rect[1] = 0
    velocidad[0] = 0
    aceleracion[0] = 0
else:
    bola1.rect[1] += velocidad[0] + aceleracion[0]/2

velocidad[0]+=aceleracion[0]
velocidad[1]+=aceleracion[1]
sprites.draw(pantalla)
pygame.display.update()
fpsTime.tick(fps)

def quitGame():
    pygame.quit()
    sys.exit()

def accel():
    r=ser.read(2)
    if len(r)==2:
        return [-(r[0]-40)/10,-(r[1]-40)/10]
    else:
        return [aceleracion[0],aceleracion[1]]

def main():
    while True:
        input()
        updateScreen()

if __name__=='__main__':
    main()
```

A6. PRESUPUESTO

En el cálculo del presupuesto se tienen en cuenta tanto las horas dedicadas en la búsqueda de información, como las horas dedicadas en el aprendizaje y experimentación. También se tienen en cuenta las horas dedicadas a la confección de la memoria y el tutorial, además del material utilizado.

El cómputo total de horas dedicadas asciende a 630 horas. El precio final de recursos humanos, teniendo en cuenta un precio de 40€/hora, asciende a 25200€.

Los materiales adquiridos específicamente para el proyecto se desglosan como sigue:

- Micro Python board → 28£ (equivalencia actual = 39,85€).
- 8 x conectores Arduino macho-hembra (8pin) (ref. A000084) → 9,29€
- 1 x Condensadores cerámicos SMD 100uds (ref. CSY100KB) → 2,35€
- 2 x SN65HVD23x 3,3V CAN Bus Transceivers → 1,48 €
- 2 x Aplomb board SOIC 8 adapter → 0.68 €

El total del precio de materiales asciende a 53,65 €.

El total del presupuesto del proyecto por lo tanto asciende a **25253,65 €**.

