
Manual de compilación de módulos en C (usermod) para la tarjeta Ophyra

Versión 1.0

**Jonatan Emanuel Salinas Ávila
Carlos Daniel Hernández Miranda**

16 de Abril, 2021

Tabla de contenido

1. Prólogo: Motivación para la creación de módulos externos en C	2
2. Introducción	2
3. Estructura de un módulo externo en C	2
3.1 Contenido de una carpeta de un módulo externo en C estándar	3
3.2 Estructura interna ejemplo de un archivo *.c de un módulo.	4
3.3 Organización recomendada de las carpetas para trabajar con un proyecto con MicroPython usermod.	10
4. Dos métodos para compilar e integrar módulos al firmware de MicroPython.	11
4.1 Inclusión de los módulos en C mediante un comando en Cygwin.	11
4.2 Inclusión de los módulos en C modificando mpconfigboard.h	13
5. Referencias	14

1. Prólogo: Motivación para la creación de módulos externos en C

En ocasiones, es bastante útil desarrollar módulos externos en C (*MicroPython usermod* o también llamados *MicroPython external C modules*) con el objetivo de reemplazar librerías escritas en MicroPython para el firmware de una tarjeta de desarrollo determinada. Ciertamente, es posible implementar en C código de bajo nivel para realizar cierta funcionalidad en la tarjeta (como por ejemplo, intercambiar información entre el MCU y elementos en la tarjeta como sensores, memorias, etc.) e integrar este código en el firmware de MicroPython. Realizar esto tiene varias ventajas, pues la ejecución del código de dicha funcionalidad se lleva a cabo en menor tiempo comparado con aquel código escrito en MicroPython, y se reduce el tamaño del firmware de MicroPython a programar en el microcontrolador.

2. Introducción

En este documento, en primer lugar se describe la estructura de un módulo externo en C, que básicamente es una carpeta que posee dos tipos de archivos: archivo(s) de código fuente en C que implementa la funcionalidad del módulo; y un archivo `.mk`, para integrar este código en el firmware. Además, se presenta un ejemplo de un código fuente que incluye los elementos principales que conforman la estructura interna de un archivo en C que implementa la funcionalidad de un módulo.

Más adelante en el documento, se describen dos formas de incorporar los módulos escritos en C en el firmware de MicroPython para la tarjeta de desarrollo deseada. El primer método es utilizando un comando en Cygwin para compilar el firmware, especificando ciertos parámetros de los módulos a integrar. El otro método, tiene que ver con habilitar los macros de aquellos módulos que se desean integrar en el archivo `mpconfigboard.h`. Estos métodos se describen a detalle en sus respectivas secciones.

Para todos los ejemplos y explicaciones de este documento, se tomará como referencia la tarjeta de desarrollo Ophyra, diseñada y manufacturada por **Intesc Electrónica y Embebidos**.

3. Estructura de un módulo externo en C

3.1 Contenido de la carpeta de un módulo externo en C estándar

La siguiente imagen muestra la carpeta “`ophyra_mpu60`”, carpeta de un módulo en C que permite la funcionalidad del sensor MPU6050 en la tarjeta Ophyra:



Nombre	Fecha de modificación	Tipo	Tamaño
 micropython.mk	02/04/2021 05:01 p. m.	Archivo MK	1 KB
 ophyra_mpu60	02/04/2021 05:01 p. m.	Archivo C	14 KB

Imagen 1. Contenido de la carpeta “ophyra_mpu60”.

Básicamente, un módulo externo en C estándar para MicroPython es una carpeta que contiene lo siguiente:

- *.c / *.cpp / *.h : Código fuente del módulo.

En este archivo o archivos, se incluye el código necesario para implementar la funcionalidad deseada de bajo nivel. En este ejemplo, el código fuente del módulo corresponde al archivo “ophyra_mpu60.c”. Este archivo contiene código en C, funciones que permiten leer de los registros del sensor los valores de aceleración en el eje X, Y y Z; los valores del giroscopio en X, Y y Z; y el valor del registro de temperatura en grados centígrados del sensor.

Este tipo de archivos deben poseer una estructura específica, para que puedan ser compilados e integrados en el firmware de MicroPython. Más adelante en la sección 3.2, se da un ejemplo de un archivo como éstos, donde se describen a detalle las partes más importantes de un archivo de código fuente en C para un módulo usermod.

- micropython.mk : Archivo Makefile para este módulo.

Típicamente, un archivo como estos tiene un aspecto similar al siguiente:

```

1  EXAMPLE_MOD_DIR := $(USERMOD_DIR)
2
3  # Add all C files to SRC_USERMOD.
4  SRC_USERMOD += $(EXAMPLE_MOD_DIR)/ophyra_mpu60.c
5
6  # We can add our module folder to include paths if needed
7  # This is not actually needed in this example.
8  CFLAGS_USERMOD += -I$(EXAMPLE_MOD_DIR)
9  CEXAMPLE_MOD_DIR := $(USERMOD_DIR)

```

Imagen 2. Contenido total del archivo micropython.mk dentro de la carpeta del módulo “ophyra_mpu60”.

Como se aprecia en la imagen, este tipo de archivos posee pocas líneas. Es necesario agregar aquí todos los archivos del código fuente del módulo que serán tomados en cuenta. En este ejemplo, hay solo un archivo .c llamado “ophyra_mpu60.c” que implementa la funcionalidad deseada, por ello, sólo este es incluido en la línea 4. Si hubiera más archivos de código para implementar la funcionalidad del módulo, se debe repetir la línea 4 para cada uno de estos archivos, indicando su nombre y su extensión. Por ejemplo, si el módulo necesita dos archivos en C, ophyra_mpu60.c y archivoAdicional.c, el archivo micropython.mk quedaría de la forma siguiente:

```

modules > ophyra_mpu60 > M micropython.mk
1  EXAMPLE_MOD_DIR := $(USERMOD_DIR)
2
3  # Add all C files to SRC_USERMOD.
4  SRC_USERMOD += $(EXAMPLE_MOD_DIR)/ophyra_mpu60.c
5  SRC_USERMOD += $(EXAMPLE_MOD_DIR)/archivoAdicional.c
6
7  # We can add our module folder to include paths if needed
8  # This is not actually needed in this example.
9  CFLAGS_USERMOD += -I$(EXAMPLE_MOD_DIR)
10 CEXAMPLE_MOD_DIR := $(USERMOD_DIR)

```

Imagen 3. Contenido del archivo `micropython.mk` si son dos archivos los que implementan la funcionalidad del módulo.

En esencia, solo esto es necesario modificar en este tipo de archivos.

3.2 Estructura interna ejemplo de un archivo `*.c` de un módulo.

A continuación se presenta un código simple de un módulo externo en C para MicroPython. Este código es “`ophyra_botones.c`”, y corresponde a un *MicroPython C usermod* que permite utilizar los cuatro botones de propósito general que posee la tarjeta Ophyra al frente. El archivo contiene la definición de una estructura y cuatro funciones, que retornan el estado del pin en donde está el botón especificado. Los comentarios dentro del código describen cada parte. Después de mostrar el código completo, se presentarán explicaciones adicionales parte por parte. El código es el siguiente:

```

#include "py/runtime.h"
#include "py/obj.h"
#include "ports/stm32/mphalport.h"          //Para el uso de la funcion mp_hal_pin_config

/*
    Estructura que será útil cuando se quiera crear en MicroPython un objeto "botón".
    Cuando "make_new" se ejecuta, se crea un apuntador a una estructura como ésta.
*/
typedef struct _buttons_class_obj_t{
    mp_obj_base_t base;
} buttons_class_obj_t;

const mp_obj_type_t buttons_class_type;

/*
    Función Print
    Esta función se ejecuta cuando el usuario escribe en MicroPython:
        print(sw())
*/
STATIC void buttons_class_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_t kind){
    (void)kind;
    mp_print_str(print, "buttons_class()");
}

```

```

}

/*
    make_new: Constructor de la clase. Esta función se invoca
    cuando el usuario de MicroPython escribe:
        sw()
*/
STATIC mp_obj_t buttons_class_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_kw,
const mp_obj_t *args) {
    //Se crea un apuntador a una estructura buttons_class_obj_t
    buttons_class_obj_t *self = m_new_obj(buttons_class_obj_t);
    self->base.type = &buttons_class_type;

    //Configuración de los 4 pines donde están los botones en la Ophyra, como INPUT y PULL_UP
    mp_hal_pin_config(pin_C2, MP_HAL_PIN_MODE_INPUT, MP_HAL_PIN_PULL_UP, 0);
    mp_hal_pin_config(pin_D5, MP_HAL_PIN_MODE_INPUT, MP_HAL_PIN_PULL_UP, 0);
    mp_hal_pin_config(pin_D4, MP_HAL_PIN_MODE_INPUT, MP_HAL_PIN_PULL_UP, 0);
    mp_hal_pin_config(pin_D3, MP_HAL_PIN_MODE_INPUT, MP_HAL_PIN_PULL_UP, 0);

    //Se retorna el apuntador a la estructura que se ha creado:
    return MP_OBJ_FROM_PTR(self);
}

/*
    Estas 4 funciones retornan el estado actual de un pin específico,
    para saber si el botón ha sido presionado o no.
    Retorna 0 si el botón está presionado, y 1 si no lo está.
*/
STATIC mp_obj_t button0_pressed(mp_obj_t self_in) {
    return mp_obj_new_int(mp_hal_pin_read(pin_C2));
}

STATIC mp_obj_t button1_pressed(mp_obj_t self_in) {
    return mp_obj_new_int(mp_hal_pin_read(pin_D5));
}

STATIC mp_obj_t button2_pressed(mp_obj_t self_in) {
    return mp_obj_new_int(mp_hal_pin_read(pin_D4));
}

STATIC mp_obj_t button3_pressed(mp_obj_t self_in) {
    return mp_obj_new_int(mp_hal_pin_read(pin_D3));
};

//Se asocian las funciones arriba escritas con su correspondiente objeto de función para
MicroPython.
MP_DEFINE_CONST_FUN_OBJ_1(button0_pressed_obj, button0_pressed);
MP_DEFINE_CONST_FUN_OBJ_1(button1_pressed_obj, button1_pressed);
MP_DEFINE_CONST_FUN_OBJ_1(button2_pressed_obj, button2_pressed);
MP_DEFINE_CONST_FUN_OBJ_1(button3_pressed_obj, button3_pressed);

```

```

/*
    Se asocia el objeto de función de MicroPython con cierto string, que será el
    que se utilice en la programación en MicroPython. Ej: Si se escribe:
        sw().sw1()
    Internamente se llama al objeto de función button0_pressed_obj, que está
    asociado con la función button0_pressed, que retorna el valor al leer el pin
    donde se encuentra el botón 1.
*/
STATIC const mp_rom_map_elem_t buttons_class_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_sw1), MP_ROM_PTR(&button0_pressed_obj) },
    { MP_ROM_QSTR(MP_QSTR_sw2), MP_ROM_PTR(&button1_pressed_obj) },
    { MP_ROM_QSTR(MP_QSTR_sw3), MP_ROM_PTR(&button2_pressed_obj) },
    { MP_ROM_QSTR(MP_QSTR_sw4), MP_ROM_PTR(&button3_pressed_obj) },
    //Función que se va a          //Pointer al objeto de la función
    //invocar en Python          //que se va a invocar.
};

STATIC MP_DEFINE_CONST_DICT(buttons_class_locals_dict, buttons_class_locals_dict_table);

const mp_obj_type_t buttons_class_type = {
    { &mp_type_type },
    .name = MP_QSTR_ophyra_botones,          //Nombre del usercmmod
    .print = buttons_class_print,            //Funcion print asociada
    .make_new = buttons_class_make_new,      //Funcion make_new asociada
    .locals_dict = (mp_obj_dict_t*)&buttons_class_locals_dict, //Diccionario asociado
};

STATIC const mp_rom_map_elem_t ophyra_botones_globals_table[] = {
    //Nombre del archivo (User C module)
    { MP_ROM_QSTR(MP_QSTR__name__), MP_ROM_QSTR(MP_QSTR_ophyra_botones) },
    //Nombre de la clase          //Nombre del "tipo" asociado.
    { MP_ROM_QSTR(MP_QSTR_sw), MP_ROM_PTR(&buttons_class_type) },
};

STATIC MP_DEFINE_CONST_DICT(mp_module_ophyra_botones_globals, ophyra_botones_globals_table);

// Define module object.
const mp_obj_module_t ophyra_botones_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&mp_module_ophyra_botones_globals,
};

// Registro del módulo para hacerlo disponible para Python.
//          nombreArchivo, nombreArchivo_user_cmodule, MODULE_IDENTIFICADOR_ENABLED
MP_REGISTER_MODULE(MP_QSTR_ophyra_botones, ophyra_botones_user_cmodule,
MODULE_OPHYRA_BOTONES_ENABLED);

```

Ahora bien, se deben resaltar ciertas partes importantes del código. Es importante, en cada módulo, al menos agregar estos dos headers: `py/obj.h`, donde todas las constantes y macros de MicroPython están definidos, y `py/runtime.h`, que contiene la declaración del

intérprete. También se agrega `mphalport.h` para utilizar la función que permitirá configurar los pines más adelante.

```
#include "py/runtime.h"
#include "py/obj.h"
#include "ports/stm32/mphalport.h"           //Para el uso de la funcion mp_hal_pin_config
```

En este ejemplo, se define una estructura simple, con solo un campo llamado “base” de tipo `mp_obj_base_t`. No todos los módulos deben llevar forzosamente la declaración de una estructura. Es útil cuando se desea que un módulo incluya la implementación de una clase; es decir, cuando en MicroPython se desea instanciar objetos de una clase implementada en este módulo.

```
/*
    Estructura que será útil cuando se quiera crear en MicroPython un objeto "botón".
    Cuando "make_new" se ejecuta, se crea un apuntador a una estructura como ésta.
*/
typedef struct _buttons_class_obj_t{
    mp_obj_base_t base;
} buttons_class_obj_t;
```

Esta estructura puede poseer más campos, dependiendo del objeto de MicroPython con el que esta estructura esté asociada. En C no hay objetos, el objeto es el que se crea en MicroPython. En la siguiente línea, se declara una constante de tipo `mp_obj_type_t`, llamada `buttons_class_type`:

```
const mp_obj_type_t buttons_class_type;
```

Más adelante en el código se define su contenido. La función `buttons_class_print`, se ejecuta cuando se intenta “imprimir un objeto” desde el código de MicroPython. Cuando en MicroPython un objeto de cierta clase es creado, y ese objeto se pasa como parámetro a la función `print()`, se imprime en la consola de MicroPython el texto dentro de esta función en C, en este caso, se imprimiría “`buttons_class()`”. Ésta función es el equivalente al método “`__str__`” de una clase en Python:

```
/*
    Función Print
    Esta función se ejecuta cuando el usuario escribe en MicroPython:
        print(sw())
*/
STATIC void buttons_class_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_t kind){
    (void)kind;
    mp_print_str(print, "buttons_class()");
}
```


La función `make_new` es de las más importantes en el código. Esta función es un constructor de clase, es el equivalente al método “`__init__`” de una clase en Python. En el momento que deseamos crear un objeto de la clase para la cual estamos creando este módulo, se invoca a `make_new`, creando así el objeto en MicroPython, pero internamente creando un apuntador a una estructura en C definida en la parte de arriba. Nótese que la localidad de memoria donde está alojada la constante `buttons_class_type` declarada anteriormente se asigna al campo `base` de la estructura:

```
/*
    make_new: Constructor de la clase. Esta función se invoca
    cuando el usuario de MicroPython escribe:
        sw()
*/
STATIC mp_obj_t buttons_class_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_kw,
const mp_obj_t *args) {
    //Se crea un apuntador a una estructura buttons_class_obj_t
    buttons_class_obj_t *self = m_new_obj(buttons_class_obj_t);
    self->base.type = &buttons_class_type;
```

En el caso particular de este código, se usa la función `mp_hal_pin_config` para configurar los pines del microcontrolador en donde están los botones de la tarjeta:

```
mp_hal_pin_config(pin_C2, MP_HAL_PIN_MODE_INPUT, MP_HAL_PIN_PULL_UP, 0);
```

También, particularmente para este módulo, a continuación se encuentran las funciones necesarias para utilizar los botones. En MicroPython, cuando se crea un objeto botón (`sw()`), ese objeto tiene las 4 funciones asociadas del código. Estas funciones deben ser estáticas y de tipo `mp_obj_t` y son como la siguiente:

```
/*
    Estas 4 funciones retornan el estado actual de un pin específico,
    para saber si el botón ha sido presionado o no.
    Retorna 0 si el botón está presionado, y 1 si no lo está.
*/
STATIC mp_obj_t button0_pressed(mp_obj_t self_in) {
    return mp_obj_new_int(mp_hal_pin_read(pin_C2));
}
```

El parámetro `mp_obj_t self_in` hace referencia al apuntador que apunta al objeto de MicroPython (internamente a la estructura inicializada en `make_new`) que está llamando a esa función. Cuando se invoca esta función a través de su nombre definido para Python (`sw1`) este parámetro no se le pasa a la función, simplemente MicroPython lo toma en automático.

Las siguientes líneas, que todo módulo en C debe tener, convierte a nuestras funciones escritas en C en objetos de función para MicroPython. Cuando se trabaja MicroPython

usermod, cada función en C debe ser convertida a un “objeto de función” para que el intérprete de MicroPython pueda ejecutarla. En estas líneas, esa conversión ocurre:

```
MP_DEFINE_CONST_FUN_OBJ_1(button0_pressed_obj, button0_pressed);
MP_DEFINE_CONST_FUN_OBJ_1(button1_pressed_obj, button1_pressed);
MP_DEFINE_CONST_FUN_OBJ_1(button2_pressed_obj, button2_pressed);
MP_DEFINE_CONST_FUN_OBJ_1(button3_pressed_obj, button3_pressed);
```

El número resaltado con azul siempre debe indicar el número de argumentos que necesita la función en C. En este caso, ese argumento es `mp_obj_t self_in`. En el siguiente código, se crea una matriz o tabla de pares en donde define el string que corresponderá a cada función de la clase y que se escribirá en MicroPython para invocarla, como se describe en el comentario:

```
/*
    Se asocia el objeto de función de MicroPython con cierto string, que será el
    que se utilice en la programación en MicroPython. Ej: Si se escribe:
        sw().sw1()
    Internamente se llama al objeto de función button0_pressed_obj, que está
    asociado con la función button0_pressed, que retorna el valor al leer el pin
    donde se encuentra el botón 1.
*/
STATIC const mp_rom_map_elem_t buttons_class_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_sw1), MP_ROM_PTR(&button0_pressed_obj) },
    { MP_ROM_QSTR(MP_QSTR_sw2), MP_ROM_PTR(&button1_pressed_obj) },
    { MP_ROM_QSTR(MP_QSTR_sw3), MP_ROM_PTR(&button2_pressed_obj) },
    { MP_ROM_QSTR(MP_QSTR_sw4), MP_ROM_PTR(&button3_pressed_obj) },
    //Función que se va a          //Pointer al objeto de la función
    //invocar en Python          //que se va a invocar.
};
```

Ésta matriz o tabla debe ser convertida a un diccionario, de manera que MicroPython pueda interpretarlo. Esto se hace con el macro siguiente:

```
STATIC MP_DEFINE_CONST_DICT(buttons_class_locals_dict, buttons_class_locals_dict_table);
```

Ahora, se define la constante `buttons_class_type` de tipo `mp_obj_type_t`, que fue declarada arriba en el código. Aquí se vinculan ciertos parámetros del código para la clase botones: el nombre de la clase, cuál es su función print, cuál es su función make_new asociada, y el diccionario local asociado:

```
const mp_obj_type_t buttons_class_type = {
    { &mp_type_type },
    .name = MP_QSTR_ophyra_botones,          //Nombre de la clase
    .print = buttons_class_print,            //Funcion print asociada
    .make_new = buttons_class_make_new,      //Funcion make_new asociada
    .locals_dict = (mp_obj_dict_t*)&buttons_class_locals_dict, //Diccionario asociado
};
```

Después, otra matriz de pares se debe definir también. Ésta contendrá dos pares: se asocia el string `__name__` con el nombre del módulo; y se asocia el nombre de la clase (string que se usará en el código de MicroPython para crear un nuevo objeto de dicha clase) con el apuntador a la constante `buttons_class_type` definida previamente.

```
STATIC const mp_rom_map_elem_t ophyra_botones_globals_table[] = {
    //Nombre del archivo (User C module)
    { MP_ROM_QSTR(MP_QSTR__name__), MP_ROM_QSTR(MP_QSTR_ophyra_botones) },
    //Nombre de la clase          //Pointer a la constante mp_obj_type_t
    //arriba definida
    { MP_ROM_QSTR(MP_QSTR_sw), MP_ROM_PTR(&buttons_class_type) },
};
```

Esta matriz o tabla de pares también debe ser convertida a un diccionario (de acuerdo a la documentación de MicroPython) con el macro siguiente:

```
STATIC MP_DEFINE_CONST_DICT(mp_module_ophyra_botones_globals, ophyra_botones_globals_table);
```

En todo módulo debe también existir una constante de tipo `mp_obj_module_t`, en este caso llamada `ophyra_botones_user_cmodule`. Aquí, se asocia el “diccionario global” con el módulo:

```
// Define module object.
const mp_obj_module_t ophyra_botones_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&mp_module_ophyra_botones_globals,
};
```

Finalmente, el módulo debe ser registrado para ser visible para MicroPython. Esto se hace con la línea:

```
// Registro del módulo para hacerlo disponible para Python.
//          nombreArchivo, nombreArchivo_user_cmodule, MODULE_IDENTIFICADOR_ENABLED
MP_REGISTER_MODULE(MP_QSTR_ophyra_botones, ophyra_botones_user_cmodule,
MODULE_OPHYRA_BOTONES_ENABLED);
```

El tercer argumento tiene especial relevancia, ya que este será el nombre del macro que debe ser habilitado o deshabilitado en el archivo `mpconfigport.h`, para incluir este módulo en la compilación del firmware (ver sección 4.2).

En la siguiente página web, es posible profundizar en cada parte de un módulo estándar en C para MicroPython:

https://micropython-usermod.readthedocs.io/en/latest/usermods_05.html#compiling-our-module

3.3 Organización recomendada de las carpetas para trabajar con un proyecto con MicroPython usermod.

Es importante mencionar que es recomendable mantener las carpetas de cada uno de los módulos dentro en una sola carpeta (puede ser llamada *modules*), donde ésta carpeta esté al mismo nivel que la carpeta del código fuente de MicroPython, como se muestra en el siguiente diagrama:

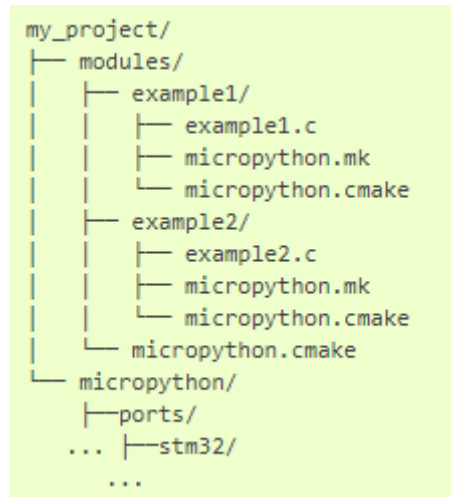


Imagen 4. Estructura típica de un proyecto con módulos externos en C y MicroPython.

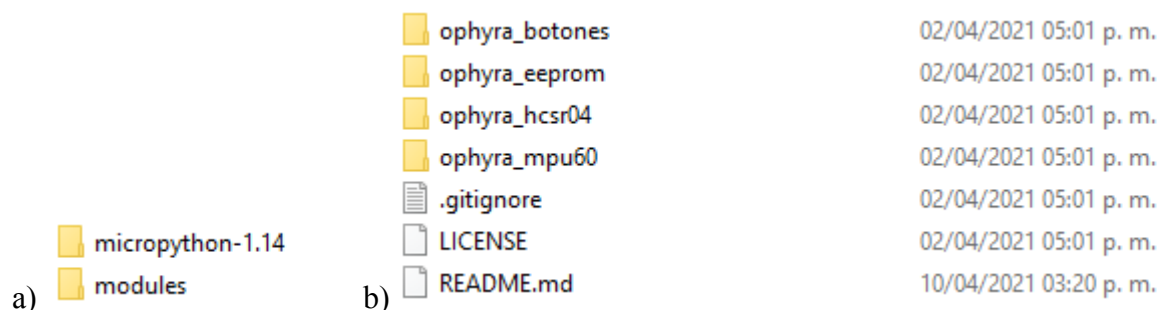


Imagen 5. a) Contenido de la carpeta del proyecto. b) Contenido de la carpeta "modules".

4. Dos métodos para compilar e integrar módulos al firmware de MicroPython.

En la siguiente sección se explican dos métodos para realizar la compilación de un módulo nuevo e integrarlo al firmware de Micropython, para que éste pueda ser después programado en la tarjeta de desarrollo deseada.

4.1 Inclusión de los módulos en C mediante un comando en Cygwin.

Para incluir cierto módulo en C en el firmware de Micropython para una tarjeta de desarrollo con este método, es necesario hacer lo siguiente:

1. En el comando `make` para compilar, es necesario agregar la bandera `USER_C_MODULES` apuntando a la carpeta donde están los módulos que se desean agregar, por ejemplo, a nuestra carpeta `modules`.
2. También en el comando `make`, es necesario habilitar los macros de los módulos a agregar mediante la bandera `CFLAGS_EXTRA`. Para esto, en el archivo de código fuente de cada módulo es útil buscar la línea donde aparece `MP_REGISTER_MODULE`. El tercer argumento, es el nombre del macro correspondiente al módulo, que debe ser puesto a 1 usando la bandera `CFLAGS_EXTRA`. Por ejemplo:

```
349 // Registro del modulo para hacerlo disponible para Python.
350 //          nombreArchivo, nombreArchivo_user_cmodule, MODULE_IDENTIFICADOR_ENABLED
351 MP_REGISTER_MODULE(MP_QSTR_ophyra_mpu60, ophyra_mpu60_user_cmodule, MODULE_OPHYRA_MPU60_ENABLED);
```

Imagen 6. Línea de código de `MP_REGISTER_MODULE`.

En este caso, el módulo es habilitado añadiendo

`CFLAGS_EXTRA=-DMODULE_OPHYRA_MPU60_ENABLED=1` al comando `make`.

En resumen, un ejemplo de comando `make` que puede construir el firmware de Micropython agregando los módulos externos de C, es el siguiente:

```
make BOARD=OPHYRA USER_C_MODULES=../../../../../modules
CFLAGS_EXTRA="-DMODULE_OPHYRA_BOTONES_ENABLED=1
-DMODULE_OPHYRA_MPU60_ENABLED=1
-DMODULE_OPHYRA_EEPROM_ENABLED=1
-DMODULE_OPHYRA_HCSR04_ENABLED=1" all
```

Donde:

- Color verde: Tarjeta para la cual se está construyendo el firmware.
- Color azul: Path hacia la carpeta donde se buscarán las carpetas de los módulos a integrar en el firmware.
- Color amarillo: Habilitación de los macros de los módulos deseados para integrarlos al firmware.
- Color morado: La palabra `all` es parte del comando, y simplemente le dice a la herramienta `make` que construya el *target* u objetivo “all” en el archivo Makefile, que en el caso de MicroPython es el `firmware.hex`.

Por ejemplo, para compilar el firmware de Micropython para la tarjeta Ophyra, navegamos en Cygwin hacia la carpeta `/ports/stm32` en el folder del código fuente de Micropython, y escribimos en Cygwin el comando (se sugiere escribirlo una vez y después copiarlo y pegarlo para volver a compilar).

```

ASUS@LAPTOP-ASUS-VIVOB00K-JONA /cygdrive/d/Jona/Intesc/Carpeta_Prueba/micropython-1.14/ports/stm32
$ make BOARD=OPHYRA USER_C_MODULES=../../modules CFLAGS_EXTRA="-DMODULE_OPHYRA_BOTONES_ENABLED=1 -DMODULE_OPHYRA_MPU60_ENABLED=1 -DMODULE_OPHYRA_EEPROM_ENABLED=1 -DMODULE_OPHYRA_HCSR04_ENABLED=1" all

```

Imagen 7. Ejemplo del comando para construir el firmware de Micropython para la Ophyra, incluyendo los módulos externos en C.

Los módulos incluidos se mostrarán a continuación de haber pulsado Enter:

```

ASUS@LAPTOP-ASUS-VIVOB00K-JONA /cygdrive/d/Jona/Intesc/Carpeta_Prueba/micropython-1.14/ports/stm32
$ make BOARD=OPHYRA USER_C_MODULES=../../modules CFLAGS_EXTRA="-DMODULE_OPHYRA_BOTONES_ENABLED=1 -DMODULE_OPHYRA_MPU60_ENABLED=1 -DMODULE_OPHYRA_EEPROM_ENABLED=1 -DMODULE_OPHYRA_HCSR04_ENABLED=1" all
Use make V=1 or set BUILD_VERBOSE in your environment to increase build verbosity.
Including User C Module from ../../modules/ophyra_botones
Including User C Module from ../../modules/ophyra_eeprom
Including User C Module from ../../modules/ophyra_hcsr04
Including User C Module from ../../modules/ophyra_mpu60

```

Imagen 8. Se muestra "Including User C Module from" que indica que se está agregando dicho módulo al firmware.

4.2 Inclusión de los módulos en C modificando *mpconfigboard.h*

Otro método para incorporar módulos en C en el firmware de MicroPython para una tarjeta de desarrollo, es modificando el archivo *mpconfigboard.h*. Este archivo se encuentra en *ports/stm32/boards/OPHYRA*. Aquí, se deben agregar las banderas de los módulos a incorporar, de la siguiente forma:

```

1  // #define OPHYRA
2  #define MICROPY_HW_BOARD_NAME      "Ophyra"
3  #define MICROPY_HW_MCU_NAME        "STM32F407VG"
4
5  #define MICROPY_HW_HAS_SWITCH      (1)
6  #define MICROPY_HW_HAS_FLASH      (1)
7  #define MICROPY_HW_HAS_MMA7660    (0)
8  #define MICROPY_HW_HAS_LIS3DSH    (0)
9  #define MICROPY_HW_HAS_LCD        (0)
10 #define MICROPY_HW_ENABLE_RNG      (1)
11 #define MICROPY_HW_ENABLE_RTC      (1)
12 #define MICROPY_HW_ENABLE_SERVO    (1)
13 #define MICROPY_HW_ENABLE_DAC      (1)
14 #define MICROPY_HW_ENABLE_USB      (1)
15 #define MICROPY_HW_ENABLE_SDCARD   (1)
16 #define MODULE_OPHYRA_MPU60_ENABLED (1)
17 #define MODULE_OPHYRA_EEPROM_ENABLED (1)
18 #define MODULE_OPHYRA_BOTONES_ENABLED (1)
19 #define MODULE_OPHYRA_HCSR04_ENABLED (1)
20 #define MODULE_OPHYRA_TFTDISP_ENABLED (1)
21 // HSE is 8MHz
22 #define MICROPY_HW_CLK_PLLM (8)
23 #define MICROPY_HW_CLK_PLLN (336)
24 #define MICROPY_HW_CLK_PLLP (RCC_PLLP_DIV2)

```

Imagen 9. Parte de arriba del archivo `mpconfigboard.h`. Se observa que en las líneas 16-20, se agregan los macros o banderas que especifican qué módulos se tomarán en cuenta para su integración en la compilación del firmware de MicroPython.

El comando `make` ahora ya no contiene la bandera `CFLAGS_EXTRA`, de modo que el firmware se construiría con un comando como el siguiente:

```
make BOARD=OPHYRA USER_C_MODULES=../../modules -j8
```

Imagen 10. Ejemplo de comando para construir el firmware con este método.

Como nota adicional, es importante recalcar que si se desean agregar los módulos externos en C para sustituir las librerías hechas en Python (tanto con un método como con el otro), es necesario quitar estas últimas del archivo `manifest.py`, ubicado en `ports/stm32/boards`, con el fin de que en el momento de la compilación no surja ningún conflicto.

Es posible encontrar información más detallada visitando los sitios con la documentación oficial de MicroPython sobre este tema, y otros sitios no oficiales pero que sin duda son de ayuda. Dichos sitios se encuentran en la sección de Referencias.

5. Referencias

George, D; Sokolovsky, P. (2021). *MicroPython external C modules*. Micropython 1.15 documentation. Recuperado de:

<https://docs.micropython.org/en/latest/develop/cmodules.html>

George, D; Sokolovsky, P. (2021). *Implementing a Module*. Micropython 1.15 documentation. Recuperado de:

<https://docs.micropython.org/en/latest/develop/library.html>

Vörös, Z. (2020). *User modules in micropython*. Welcome to micropython-usermod's documentation! Recuperado de:

https://micropython-usermod.readthedocs.io/en/latest/usermods_03.html?highlight=make#user-modules-in-micropython

Vörös, Z. (2020). *Micropython usermod documentation. Release 1.622*. Recuperado de:

<https://readthedocs.org/projects/micropython-usermod/downloads/pdf/latest/>

Abby. (2020). *Extend the C module for MicroPython!* DEV Community. Recuperado de:

<https://dev.to/abby06/extend-the-c-module-for-micropython-1inc>

Robson, O. (2019). *MicroPython C Stub Generator*. Recuperado de:

<https://mpy-c-gen.oliverrobson.tech/>