

UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
CARRERA DE ESPECIALIZACIÓN EN SISTEMAS
EMBEBIDOS



MEMORIA DEL TRABAJO FINAL

**Entorno de programación educativo en
lenguaje Python para la EDU-CIAA-NXP**

Autor:
Ing. Ernesto Gigliotti

Director:
Esp. Ing. Eric Pernía (FIUBA)

Jurados:
Dr. Ing. Pablo Gomez (FIUBA)
Ing. Alejandro Permingeat (FIUBA)
Esp. Ing. Pablo Ridolfi (UTNFRBA, FIUBA)

*Este trabajo fue realizado en las Ciudad Autónoma de Buenos Aires, entre enero
de 2016 y diciembre de 2016.*

Resumen

En este trabajo se presenta la implementación de un conjunto de herramientas que permite a estudiantes de educación secundaria y universitaria, utilizando el lenguaje Python junto con la versión educativa de la Computadora Industrial Abierta Argentina (EDU-CIAA-NXP), aprender programación sobre sistemas embebidos. El conjunto de herramientas desarrollado consta de un firmware que permite a la EDU-CIAA-NXP ejecutar código Python, un software que permite al usuario escribir dicho código y grabarlo en la placa, ejemplos y documentación para facilitar el aprendizaje. Para asegurar la calidad del trabajo se utilizaron técnicas de ingeniería de software, planificación y gestión de proyectos.

Agradecimientos

Agradecimientos personales. [OPCIONAL]

No poner anti-agradecimientos (este trabajo fue posible a pesar de...)

Índice general

Resumen	III
1. Introducción General	1
1.1. Dificultades planteadas en la enseñanza de programación	1
1.2. Objetivo	2
1.3. Solución propuesta	2
2. Introducción Específica	3
2.1. La plataforma de Hardware	3
2.2. Utilización de MicroPython	4
2.3. Arquitectura del Firmware	5
2.4. Arquitectura del Software	6
2.5. Requerimientos	8
3. Diseño e Implementación	9
3.1. Diseño de Firmware	9
3.1.1. Punto de partida	9
3.1.2. Creación de bibliotecas Python: Módulos y Clases	10
3.1.3. Diseño de bibliotecas para el manejo de periféricos desde C	12
3.1.4. Diseño de bibliotecas para el manejo de periféricos desde el código Python	14
3.2. Diseño del Software	15
3.2.1. Punto de partida	15
3.2.2. Diseño del IDE	15
3.2.3. Envío del archivo a la placa	18
3.3. Documentación	18
3.3.1. Diseño de proyectos de ejemplo	18
3.3.2. Documentación de las bibliotecas implementadas	18
4. Ensayos y Resultados	19
4.1. Test unitarios para bibliotecas para el manejo de periféricos desde C	19
4.2. Test unitarios para bibliotecas para el manejo de periféricos desde Python	19
4.3. Test Unitarios para ventanas que componen el IDE	19
5. Conclusiones	21
5.1. Conclusiones generales	21
5.2. Próximos pasos	21
A. Creación de un módulo y clase Python desde código C	23
A.1. Creación de un módulo	23
A.2. Incorporación de una clase al módulo	24
A.3. Definición de un método de una clase	26

B. Tests funcionales del uso de periféricos desde código Python	29
C. Tests funcionales del uso del IDE	31
Bibliografía	33

Índice de figuras

2.1. Diagrama en bloques de la plataforma EDU-CIAA-NXP	3
2.2. Foto de una placa EDU-CIAA-NXP	4
2.3. Foto de una placa Pyboard	4
2.4. Diagrama en bloques de la arquitectura del Firmware	5
2.5. Diagrama en bloques de la arquitectura del Software del IDE	6
3.1. Conexión de la EDU-CIAA-NXP con la PC	9
3.2. Anidamiento de llamadas a funciones desde Python a C	11
3.3. Relación entre las clases Python y los archivos del Firmware	12
3.4. Diagrama de clases para manejo de periféricos	14
3.5. Procesador de texto EDILE v0.2	15
3.6. Botones adicionales del IDE.	16
3.7. Diagrama de clases de las funcionalidades agregadas al IDE.	16
3.8. Diagrama de secuencia que describe la comunicacion entre el IDE y el firmware.	18

Índice de Tablas

Dedicado a... [OPCIONAL]

Capítulo 1

Introducción General

En este capítulo se abordará la problemática en la enseñanza de la programación de sistemas embebidos y la solución propuesta por medio de este trabajo.

1.1. Dificultades planteadas en la enseñanza de programación

Un alumno de nivel secundario o universitario que comienza su formación en el ámbito de la informática o electrónica, y es instruido para adquirir conocimientos de programación sobre sistemas embebidos, generalmente debe enfrentarse a ciertos problemas a los que no debería enfrentarse en una etapa tan temprana de aprendizaje. Estos problemas generalmente tienen que ver con el lenguaje elegido para aprender a programar y la sintaxis del mismo, así como también a la complejidad del *IDE*¹ utilizado y a requerir conocimientos medianamente avanzados sobre arquitecturas de microcontroladores.

Los típicos problemas a los que un estudiante inicial debe enfrentarse según (Kaczmarczyk, East, Petrick y Herman, 2010) [5] son modelos de datos, referencias, punteros, tipos de variables, bucles y sentencias condicionales entre otros. Si a estos problemas se le suman los mencionados previamente, se crea una situación en donde el alumno no puede focalizarse en comprender y practicar algoritmia, y debe lidiar con problemas de sintaxis, compilación, configuración del entorno de desarrollo, y complicados conocimientos de arquitectura para configurar y utilizar los periféricos de un microcontrolador como por ejemplo una entrada o salida digital y en muchos casos no pudiendo acceder a la documentación adecuada o a ejemplos detallados.

(Brito y de Sá-Soares, 2013) [1] aseguran que existe una alta tasa de incomprensión en las materias de introducción a la programación e investigaciones demuestran que herramientas que permiten escribir un algoritmo en forma simple y probar su ejecución de inmediato como *Scratch*² dan resultados mucho más favorables (Resnick et al., 2009) [7].

¹IDE: Entorno de Desarrollo Integrado.

²Scratch: Lenguaje de programación basado en bloques.

1.2. Objetivo

El objetivo principal del proyecto es proveer un conjunto de herramientas que permitan a estudiantes de educación secundaria y universitaria aprender programación sobre sistemas embebidos, evitando los típicos problemas que surgen en los primeros acercamientos a estos temas, debido a la complejidad que imponen los lenguajes y plataformas utilizadas comúnmente (lenguajes de nivel medio o bajo como C o ASM y entornos de desarrollo difíciles de configurar para la mirada de un principiante). El proyecto busca desde su simplicidad (tanto a nivel programación como configuración y puesta en marcha del entorno de desarrollo) evitar al principiante frustraciones que desalienten el aprendizaje.

1.3. Solución propuesta

Para abordar estos inconvenientes, se plantea un conjunto de herramientas que abarcan cada uno de los aspectos mencionados y brindan una solución integral a los mismos, permitiendo al alumno focalizarse en los verdaderos problemas de cada una de las etapas de aprendizaje.

Para solucionar los problemas de sintaxis y compilación, se optó por el uso del lenguaje Python, cuya característica principal es su sintaxis clara y simple, con muy pocas reglas en lo que respecta a palabras reservadas del lenguaje y tipos de datos. Muchas universidades utilizan Python como lenguaje para enseñar programación debido a estas características[3]. Combinando la simplicidad del lenguaje Python con la popularidad de la plataforma de hardware EDU-CIAA-NXP se concluye que la creación de un conjunto de herramientas que permitan la utilización de dicho lenguaje sobre la plataforma, permitiendo el uso de los periféricos básicos de la misma, utilizando una sintaxis simple y clara, sin necesitar conocimientos avanzados de arquitecturas de microcontroladores para su utilización, es una justificación más que acertada para el desarrollo de este trabajo.

Para solucionar los conflictos de complejos entornos de desarrollo difíciles de configurar y utilizar, con muchos requerimientos de hardware en la PC donde se utilizarían, se desarrolló un IDE mediante el cual solo puede escribirse código Python en un archivo, y grabar el mismo en la EDU-CIAA-NXP mediante un botón, y en el cual toda la configuración requerida consta de indicar el puerto serie donde se encuentra conectada la placa. A su vez se trató de mantener lo más bajo posible los requerimientos necesarios para ejecutar el IDE, permitiendo el uso en viejas generaciones de PCs que todavía forman parte de las escuelas secundarias, las cuales no podrían ejecutar IDEs complejos como *Eclipse*³ o *Netbeans*⁴.

Por último se escribieron ejemplos con explicaciones detalladas que demuestran el uso de la placa para la construcción de proyectos típicos de escuelas técnicas secundarias, junto con la documentación detallada de todas las bibliotecas Python disponibles para manejar algunos periféricos de la EDU-CIAA-NXP, evitando así requerir conocimientos de microcontroladores para la configuración y uso de los mismos.

³Eclipse: Entorno de desarrollo integrado para múltiples lenguajes. url:<https://eclipse.org>

⁴Netbeans: Entorno de desarrollo integrado para múltiples lenguajes. url:<https://netbeans.org>

Capítulo 2

Introducción Específica

En este capítulo se detallará la plataforma de hardware utilizada, y se hará una introducción a la arquitectura del firmware y software.

2.1. La plataforma de Hardware

La Computadora Industrial Abierta Argentina (CIAA) es un proyecto que comenzó en el año 2013 gracias a las acciones conjuntas entre la ACSE¹ y CADIEEL², dando lugar a una plataforma de hardware que posee dos valiosas características: Ser de carácter industrial, es decir, pensada y diseñada para las exigencias de confiabilidad que la industria requiere, y ser abierta bajo licencia BSD, lo cual promueve su uso, modificación y redistribución.

Este trabajo se basó principalmente en la versión educativa de esta computadora, llamada “EDU-CIAA-NXP” la cual utiliza un microcontrolador NXP LPC 4337 JDB 144 (Dual-core Cortex-M4 + Cortex-M0).

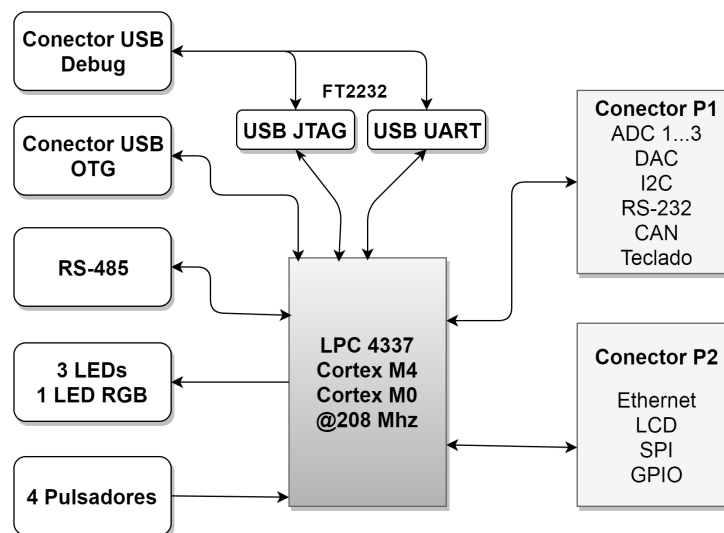


FIGURA 2.1: Diagrama en bloques de la plataforma EDU-CIAA-NXP

¹ACSE: Asociación Civil para la investigación, promoción y desarrollo de los Sistemas electrónicos Embebidos.

²CADIEEL: Cámara Argentina de Industrias Electrónicas, Electromecánicas y Luminotécnicas.

Como se observa en la figura 2.1 la placa tiene 4 pulsadores, 3 leds y un led RGB con los cuales es posible realizar muchos ejercicios, así como también una conexión USB mediante un chip FT2232 el cual brinda un puerto JTAG y una UART para conectar el microcontrolador con la PC. La placa también posee dos conectores P1 y P2 de 40 pines cada uno, en donde se encuentran conectados los periféricos del microcontrolador (GPIOs, UARTs, SPI, I2C, etc.)



FIGURA 2.2: Foto de una placa EDU-CIAA-NXP

El diseño fue pensado para proveer una plataforma de desarrollo moderna y económica basada en la CIAA que sirva a docentes y a estudiantes en los cursos de sistemas embebidos y lograr una amplia inserción en el sistema educativo argentino.

En la figura 2.2 se muestra una foto de una placa EDU-CIAA-NXP en donde se puede observar claramente los 4 pulsadores y los leds SMD que permiten realizar ejercicios sin requerir componentes adicionales.

2.2. Utilización de MicroPython

El proyecto MicroPython [9] es un desarrollo de firmware realizado por Damien George, el cual fue pensado para correr sobre la plataforma Pyboard, desarrollada por Jalttek Systems [6]. Este proyecto permite la ejecución de código Python y la utilización de los periféricos que posee la placa, desde dicho lenguaje.

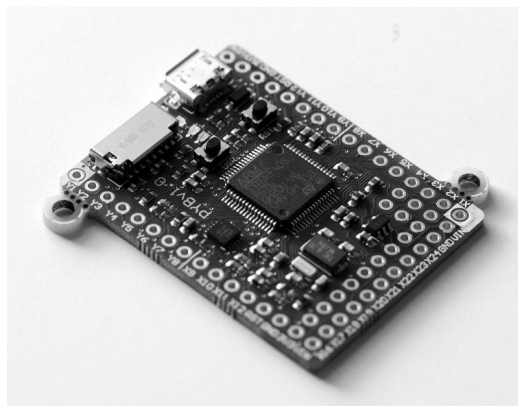


FIGURA 2.3: Foto de una placa Pyboard

Las características del microcontrolador que posee esta placa, son similares a las de la EDU-CIAA-NXP (Cortex-M4@168Mhz).

Para obtener la capacidad de ejecutar código Python en la EDU-CIAA-NXP, se adoptó el trabajo de Martin Ribelotta, quien realizó el port del proyecto MicroPython mencionado previamente, para esta plataforma [8]. Este port constaba de la inicialización del intérprete, el cual permitía ejecutar código Python, la inicialización y configuración de la *Garbage Collector*³ y la implementación de un filesystem FAT12 embebido en la memoria de programa del microcontrolador, de manera de poder escribir en forma permanente el código Python en la memoria y luego ser ejecutado desde allí.

El autor del trabajo presentado comenzó a desarrollar el soporte para algunos periféricos a mediados de 2015, ya que el port de Martin Ribelotta no contaba con la implementación para el manejo de dichos periféricos. Esto se tomó como punto de partida y se reescribió y mejoró en gran medida para lograr la calidad del firmware deseada al implementar técnicas de ingeniería de software.

2.3. Arquitectura del Firmware

A continuación se detallará la arquitectura del firmware implementado. En la misma se pueden apreciar los siguientes bloques:

- **EDU-CIAA-NXP** : Representa el hardware donde se ejecuta el firmware.
- **Intérprete uPython** : Representa el código del proyecto MicroPython, mediante el cual es posible interpretar y ejecutar el código Python escrito por el usuario.
- **Código Python programado por el usuario** : Representa el código que el usuario escribe en el IDE y graba en la placa para su posterior ejecución.

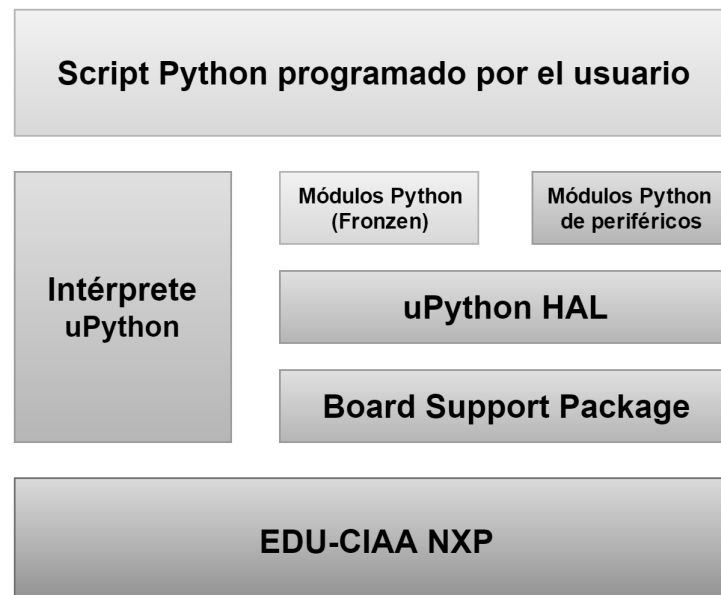


FIGURA 2.4: Diagrama en bloques de la arquitectura del Firmware

³Garbage Collector: Gestor automático de memoria dinámica.

Estos tres bloques mencionados y que pueden identificarse en la figura 2.4 representan el firmware del proyecto MicroPython creado por Martin Ribelotta que se tomó como punto de partida para el desarrollo de este trabajo.

A continuación se detallan los bloques que se han desarrollado:

- **Board Support Package** : Aquí se encuentra la biblioteca que permite el acceso, uso e inicialización de los periféricos del microcontrolador.
- **uPython HAL** : Capa de abstracción del hardware. El proyecto MicroPython utiliza esta capa para abstraer funciones específicas del hardware y que el código que utilice los periféricos sea portable.
- **Módulos Python de periféricos** : Para que el usuario pueda utilizar los periféricos desde el código Python, se necesita escribir un código en lenguaje C que represente una clase Python, en dicho código se consigue ejecutar funciones de C al ejecutar funciones de Python, de esta manera se logra utilizar la capa uPython HAL desde el código Python escrito por el usuario.
- **Módulos Python (Frozen)** : También es posible escribir bibliotecas Python directamente en lenguaje Python, y que las mismas formen parte del firmware. Este trabajo incluyó una biblioteca escrita en Python que permite la ejecución de tests unitarios.

2.4. Arquitectura del Software

A continuación se detallará la arquitectura del entorno de desarrollo, en la cual se describe la utilización de una ventana principal en donde se puede editar el archivo de código Python, y mediante un mecanismo de plug-ins se indica la incorporación de ventanas específicas que tienen que ver con el proceso de grabar el código en la EDU-CIAA-NXP.

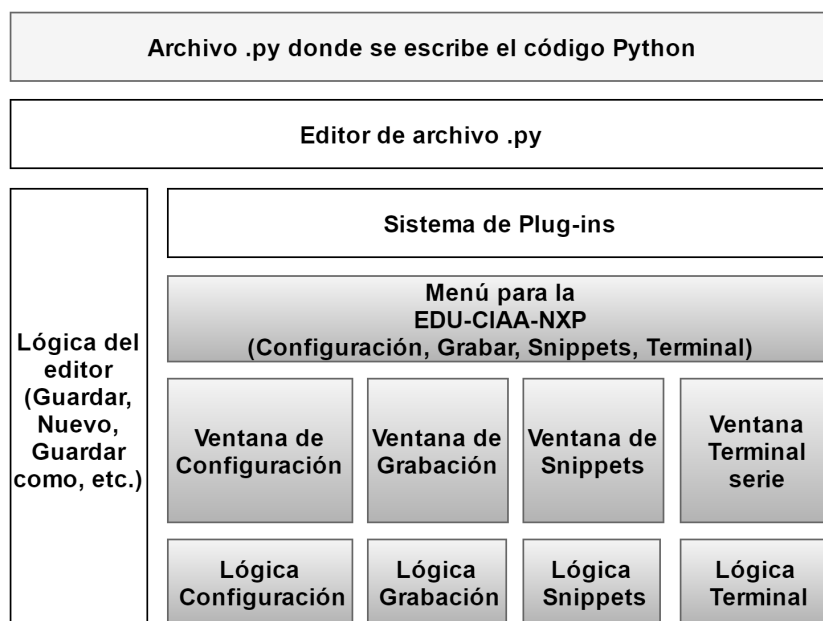


FIGURA 2.5: Diagrama en bloques de la arquitectura del Software del IDE

En la figura 2.5 se observan los siguientes módulos de software que componen el IDE:

- **Archivo .py** : Representa el archivo que se abre, edita y guarda usando el IDE. En este archivo se escribirá el código Python que se grabará y ejecutará en la placa.
- **Editor de archivo .py** : Representa la capa gráfica del programa en donde se escribe el código Python.
- **Lógica del editor** : Representa las funciones que permiten que la ventana principal donde se escribe el código funcione como un editor de texto.
- **Sistema de plug-ins** : La lógica del editor incorpora un sistema de plug-ins mediante el cual se agregan opciones al menú y permiten ejecutar ventanas adicionales a la principal.
- **Menú para la EDU-CIAA-NXP** : Aquí se encuentran las opciones que permiten interactuar con la placa y los ejemplos.

Dentro de las ventanas agregadas, se encuentran las que permiten la grabación del código en la placa, la ejecución de una terminal serial, mediante la cual es posible visualizar el standard output del código Python e ingresar datos por el standard input, una ventana con una lista de porciones de código de ejemplo (Snippets) y una ventana de configuración en donde se declara el puerto serie de la PC a utilizar. A continuación se detallan los módulos de software correspondientes a las ventanas mencionadas previamente:

- **Ventana de configuración** : Aquí se permite elegir el puerto serie de la PC en donde se conectó la placa.
- **Lógica configuración** : Representa las funciones que manejan la ventana de configuración.
- **Ventana de grabación** : Aquí se muestra el progreso del envío del archivo a la placa.
- **Lógica de grabación** : Representa las funciones que se encargan de enviar el archivo a la placa y mostrar el progreso en la ventana de grabación.
- **Ventana de Snippets** : Aquí se muestra una lista de porciones de código de ejemplo y mediante un botón se puede agregar dicha porción de código al archivo que se está escribiendo.
- **Lógica de Snippets** : Representa las funciones que manejan la interacción del usuario con la ventana de Snippets.
- **Ventana terminal serie** : Aquí se muestra el stdout del código de Python que se ejecuta en la placa y se capturan las teclas que presiona el usuario y se envían hacia el stdin del código Python que se ejecuta en la placa.
- **Lógica terminal serie** : Representa las funciones encargadas de la comunicación serie y representación en la ventana.

El desarrollo de este IDE se basó en el IDE del proyecto EDILE [2], y se agregaron las ventanas específicas mencionadas previamente.

2.5. Requerimientos

A continuación se detallan los requerimientos planteados para la realización de este trabajo.

Grupo de requerimientos referidos a las bibliotecas Python:

1. Manejo de los leds que dispone la placa.
2. Utilización de los pulsadores.
3. Manejo y configuración de los pines designados como GPIO.
4. Configuración y utilización de la UART.
5. Configuración y utilización de la interface RS485.
6. Lectura de las entradas ADC.
7. Salida DAC.
8. Utilización de la EEPROM interna.
9. Utilización de Timers.

Grupo de requerimientos referidos al entorno de desarrollo:

1. El software deberá ser multiplataforma (Windows/Linux/OSX).
2. No debe ser necesario recompilar el firmware de la placa para cambiar el código de python.
3. El programa de python se enviará por el COM virtual generado al conectar la placa a la PC.
4. El software deberá tener embebida una terminal serial, por donde se implementará la interfaz de salida y entrada estándar del programa de Python.
5. El software deberá tener porciones de código de ejemplo que puedan insertarse fácilmente junto con lo que el usuario programa (Snippets)
6. El software deberá tener links para acceder fácilmente a la documentación online y a los proyectos de ejemplo.

Grupo de requerimientos referidos a los proyectos de ejemplo:

1. Los proyectos de ejemplo serán divididos en tres categorías: Inicial, Intermedio y Avanzado.
2. Los proyectos de ejemplo consistirán en el código fuente, la explicación del mismo en forma detallada, y un esquemático de conexión con componentes externos en el caso que se requiera.
3. Los proyectos de ejemplo están basados en los proyectos típicos de electrónica que se realizan en escuelas secundarias.

Grupo de requerimientos generales:

1. El acceso a la información del proyecto será libre y gratuita.
2. Se publicará la documentación de las bibliotecas de Python disponibles para programar.

Capítulo 3

Diseño e Implementación

En este capítulo se detalla el desarrollo del firmware, software y documentación.

3.1. Diseño de Firmware

3.1.1. Punto de partida

Al tomar el port de MicroPython para la EDU-CIAA-NXP, era posible ejecutar un intérprete *REPL*¹ que provee el proyecto, el cual tiene como standard input (stdin) y standard output (stdout) el puerto serie que la placa tiene conectado a través del conversor USB, de modo que en la PC se crea un COM Virtual que permite a cualquier programa que emula una terminal por puerto serial, conectarse a dicho intérprete.

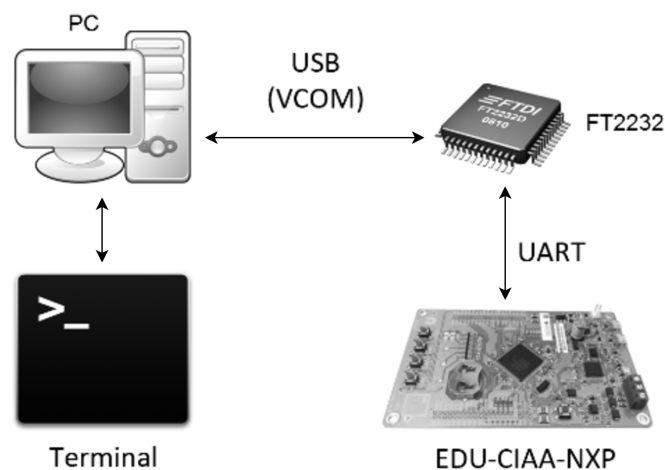


FIGURA 3.1: Conexión de la EDU-CIAA-NXP con la PC

En la figura 3.1 se observan los bloques que modelan la conexión de la placa con la PC. Mediante el circuito integrado FT2232 que posee la placa, se conecta por medio del bus USB, la PC a la EDU-CIAA-NXP, permitiendo visualizar uno de los módulos UART del microcontrolador como un puerto serie en la PC.

¹REPL: Read Eval Print Loop. Mecanismo que toma una expresión escrita por el usuario, la evalúa y ejecuta, devolviendo el resultado al usuario.

Esto permite ejecutar programas que no tengan interacción con el resto del hardware, por ejemplo, se puede escribir:

```
» print("Hola mundo")
```

Una vez ingresado el código, el interprete lo evalúa y ejecuta, y el texto "Hola mundo" se transmite por el stdout, el cual está ligado a la UART conectada al conversor serie-USB, por lo que el texto "Hola mundo" se muestra por la terminal de la PC.

De la misma forma se puede utilizar el sdtin con sentencias como:

```
» n = input("Ingrese un numero")
```

También se pueden realizar operaciones matemáticas sin inconvenientes:

```
» a = 27
» b = 3
» c = a + b
» print(c)
```

Este último ejemplo imprime por la terminal el valor 30.

Pero para interactuar con los periféricos como los pulsadores y los leds, solo existía una versión preliminar de algunas bibliotecas Python creadas previamente por el autor de este trabajo, las cuales se tomaron como punto de partida para el desarrollo profesional de las mismas verificando su correcto funcionamiento por medio de test unitarios y funcionales los cuales son explicados en el capítulo 4.

3.1.2. Creación de bibliotecas Python: Módulos y Clases

Si bien es posible definir funciones sueltas sin contexto, en Python también es posible la declaración de clases, al ser un lenguaje multiparadigma, el mismo contempla la Programación Orientada a Objetos (POO), mediante la cual se modelaron los periféricos del microcontrolador.

En Python, las clases se agrupan dentro de módulos. Un módulo Python es un conjunto de clases, para poder utilizar una clase que se encuentra dentro de un módulo, deberemos incluir dicho módulo en nuestro programa, mediante la sentencia `import`, por ejemplo:

```
» import pyb
```

El módulo `pyb` tiene dentro definidas las clases que representan los periféricos del microcontrolador. Una vez incluido el módulo, podremos utilizar las clases definidas dentro de él para crear objetos. Por ejemplo, para la creación de un objeto que representa el led 1 de la EDU-CIAA-NXP, se escribe:

```
» import pyb
» led = pyb.LED(1)
» led.on()
```

En este ejemplo se utiliza la clase `LED` para crear el objeto `led`, la misma recibe como argumento de su constructor, el número de led que el objeto manejará, en este caso, el led 1.

Para poder manejar un periférico del microcontrolador (GPIOs, UART, etc.) es necesario que al ejecutar una o más funciones de código Python, se ejecute una o más funciones de código C, en la cual se coloca el código necesario para la utilización del periférico en cuestión.

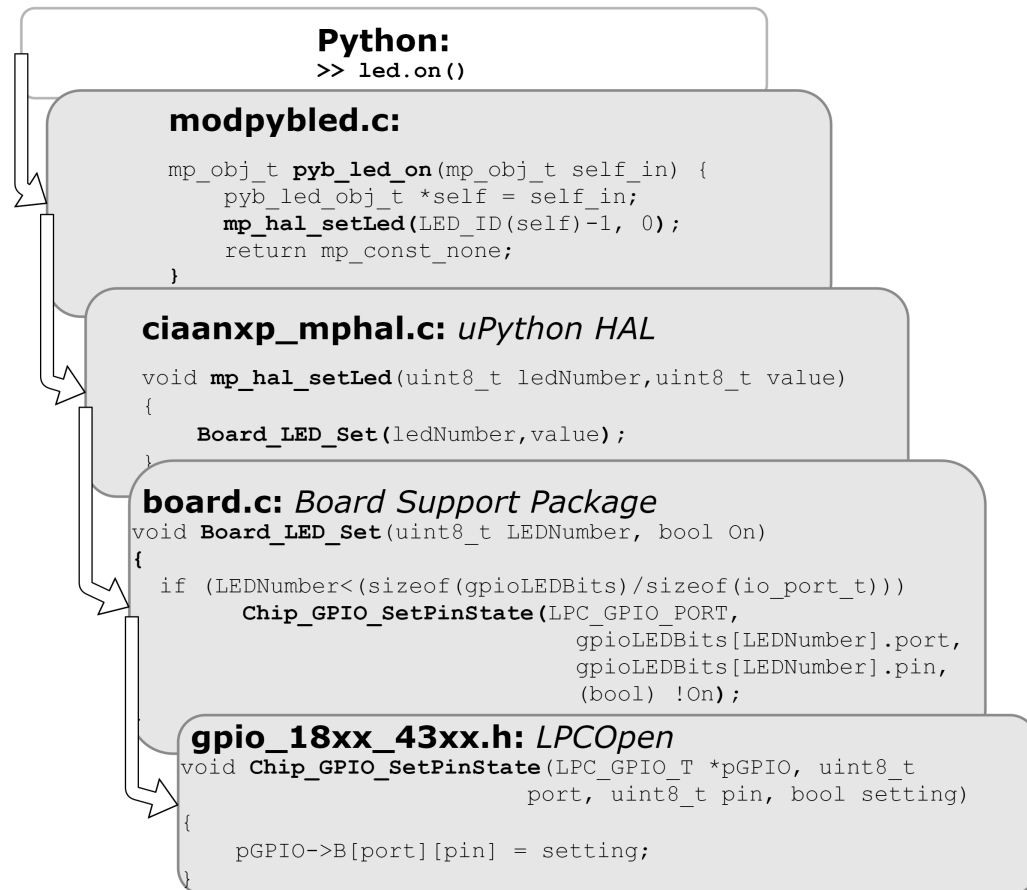


FIGURA 3.2: Anidamiento de llamadas a funciones desde Python a C

En la figura 3.2 se muestra un diagrama donde se aprecia el *trace*² de las llamadas a función que se ejecutan cuando el usuario ejecuta desde código Python el método on() de un objeto LED.

En esta figura se observa la llamada del método on() desde el objeto led, esto produce la ejecución de la función pyb_led_on() la cual se encuentra definida en el archivo modpybled.c, en este archivo, están declaradas todas las funciones que se mapean a métodos del objeto LED. Estos archivos modpybXXX.c representan la clase de un periférico (en este ejemplo modpybled.c representa la clase LED) y utilizan las funciones de la capa de abstracción de hardware (uPython HAL) definidas en ciaanxp_mphal.c para acceder y utilizar los periféricos. Dentro de la capa uPython HAL, se utilizan las funciones definidas en la capa Board Support Package (archivo board.c) en donde se utilizan las funciones de la biblioteca *LPCOpen*³ para el manejo de periféricos.

²Tracing: Log que muestra información acerca de la ejecución de un programa junto con las llamadas a funciones.

³LPCOpen: Biblioteca desarrollada por NXP la cual provee macros y funciones para acceder a los registros del microcontrolador desde lenguaje C.

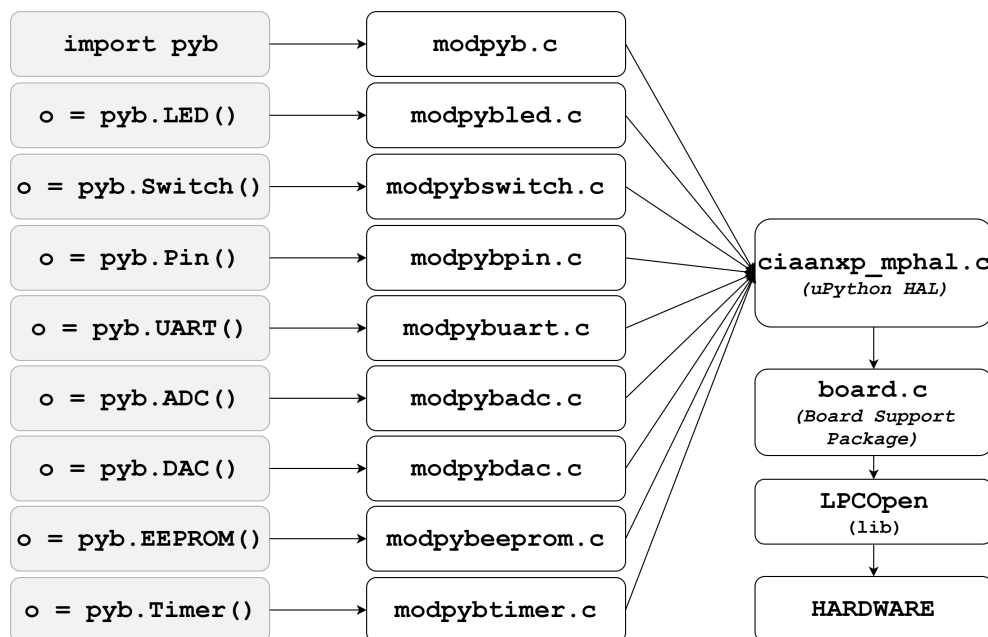


FIGURA 3.3: Relación entre las clases Python y los archivos del Firmware

En la figura 3.3 se observan los archivos implicados para poder brindar desde el código Python una interfaz para el uso de cada periférico. En ella se observa que existe un archivo `modpybXXX.c` para definir cada clase Python. Dentro de cada archivo se definen las funciones de C que se ejecutarán al invocar los métodos desde Python. Estas funciones de C utilizan la capa uPython HAL para interactuar con los periféricos del microcontrolador. Para una explicación detallada sobre el desarrollo de una clase Python desde código C, referirse al apéndice A

3.1.3. Diseño de bibliotecas para el manejo de periféricos desde C

En el archivo `board.c` y `board.h` se definió la capa Board Support Package (BSP), mediante la cual se accede a los periféricos del microcontrolador, en este archivo se definieron las funciones para inicializar y utilizar dichos periféricos, requiriendo la menor cantidad de datos de inicialización posibles y abstrayendo en gran medida la arquitectura del microprocesador.

Esta capa utiliza la biblioteca `LPCOpen`, la cual es de más bajo nivel pero permite un fácil acceso a los registros del microcontrolador, proveyendo funciones y macros para resolver este problema.

Las funciones definidas en `board.c` tienen el formato:

```
Board_NombrePeriferico_NombreFuncion()
```

Algunos ejemplos de las funciones que pueden encontrarse en este archivo son:

```
void Board_LED_Set(uint8_t LEDNumber, bool On);
bool Board_LED_Test(uint8_t LEDNumber);
void Board_LED_Toggle(uint8_t LEDNumber);
```

Todos los periféricos tienen su función `Init`:

```
Board_NombrePeriferico_Init()
```

La cual se encarga de inicializar dicho periférico. Existe también en el archivo, definida una función principal que se encarga de llamar a todas las funciones “Init” y es llamada “Board_Init()”

Como se indica en la figura 3.2, las funciones definidas en esta capa son ejecutadas por las funciones definidas en la capa uPython HAL, las cuales se definen en el archivo `ciaanxp_mphal.c` y `ciaanxp_mphal.h`.

La capa uPython HAL, es prácticamente transparente, ya que en la mayoría de los casos no agrega funcionalidades, y simplemente invoca a las funciones de la capa BSP. Sin embargo en algunos casos en donde las funciones definidas en BSP no cubren las características necesarias para ser utilizadas desde Python, se han agregado las características en esta capa.

```

1 uint32_t mp_hal_rs232_write(uint8_t const * const buffer ,
2                               uint32_t size , uint32_t delay)
3 {
4     if (delay==0)
5         return Board_UART_Write(LPC_USART3,buffer , size) ;
6
7     uint32_t i=0;
8     while (size >0)
9     {
10         Board_UART_Write(LPC_USART3,&(buffer[i]) ,1) ;
11         mp_hal_milli_delay (delay) ;
12         size --;
13         i ++;
14     }
15     return i ;
16 }
```

ALGORITMO 3.1: Función de envío por la UART de la capa uPython HAL

También se han agregado porciones de código para validación de rangos o para agregar argumentos, como se muestra en el algoritmo 3.1 en donde se define la función para enviar por el puerto serie. En este caso se observa que la función de la capa uPython HAL tiene un argumento que indica un retardo (delay) entre cada byte que se envía, funcionalidad que no existe en la capa BSP, ya que solo se dispone de la función “Board_UART_Write()” la cual recibe el buffer a transmitir. En esta función se evalúa el valor del argumento delay, y en el caso de que sea mayor a cero, se harán envíos de 1 byte y por cada envío se ejecutará la función “mp_hal_milli_delay()” la cual bloquea el código por el tiempo especificado.

Las funciones de la capa uPython HAL tienen la forma:

```
mp_hal_nombrePeriferico_nombreFuncion()
```

Tanto la biblioteca BSP como la capa uPython HAL pueden utilizarse de forma aislada en un proyecto *baremetal*⁴ para el microcontrolador LPC4337, proveyendo un acceso simple a los periféricos del mismo.

⁴Baremetal: Método de programación de bajo nivel para un hardware específico utilizando herramientas básicas y sin un sistema operativo.

3.1.4. Diseño de bibliotecas para el manejo de periféricos desde el código Python

Se pretendió modelar cada periférico mediante una clase, si bien era posible definir funciones pertenecientes al módulo `pyb`, sin definir clases, por ejemplo:

```
» pyb.setLed(1, 0)
```

Se decidió respetar el modelo orientado a objetos por las siguientes razones:

- El proyecto MicroPython original utiliza este modelo para acceder a los periféricos.
- Ayuda al entendimiento de programación orientada a objetos (POO), en donde la premisa más básica es representar objetos del mundo real, en este caso un objeto `Switch` representará un pulsador físico en la placa, generando un ejemplo muy claro del concepto.
- Para el uso básico y explicaciones de algoritmo, no es necesario tener los conocimientos de POO, simplemente se trata al objeto como una variable, y no se generan grandes trabas en el momento del aprendizaje.

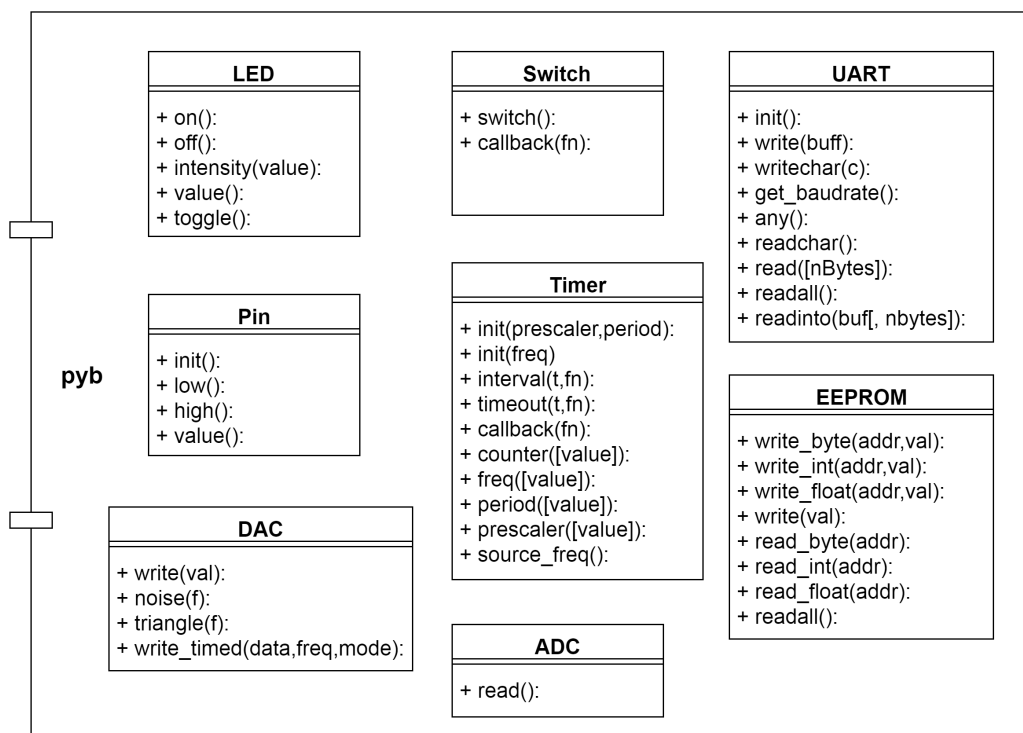


FIGURA 3.4: Diagrama de clases para manejo de periféricos

En la figura 3.4 se observa el diagrama de clases definido para el módulo `pyb`, en donde se encuentran las clases que representan los periféricos del microcontrolador junto con los métodos desarrollados.

3.2. Diseño del Software

3.2.1. Punto de partida

Para el desarrollo del IDE se tomó como base el proyecto EDILE [2] el cual se encuentra escrito en lenguaje Python y proporciona un procesador de texto con coloreado de palabras clave según el lenguaje seleccionado, el cual se detecta automáticamente por la extensión del archivo que se abre.

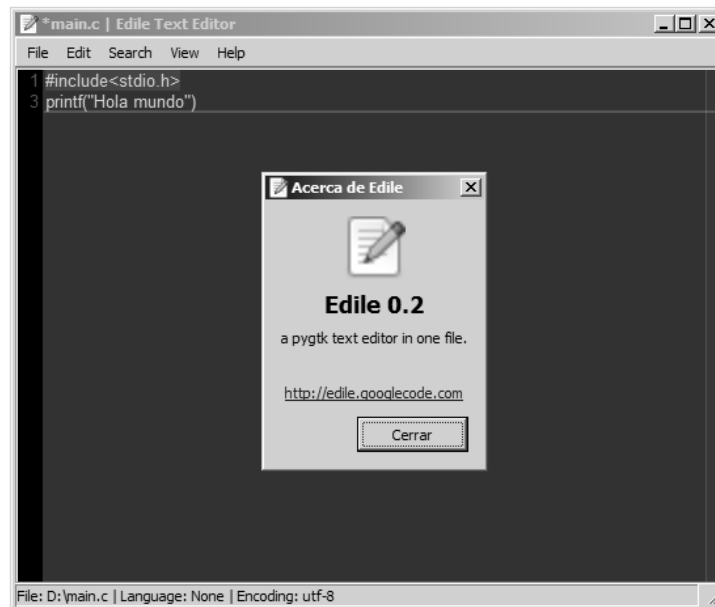


FIGURA 3.5: Procesador de texto EDILE v0.2

Debido a que se encuentra programado en Python es muy simple ejecutar el programa en diferentes sistemas operativos, de modo que utilizando este proyecto se puede cumplir con uno de los requerimientos planteados el cual decía que el software debe ser multiplataforma.

Este procesador de texto posee un sistema de plug-ins mediante el cual se simplifica la incorporación de menús que ejecuten ventanas adicionales que cumplan con cierta funcionalidad, de modo que el diseño de software que se realizó en este trabajo fue la incorporación de dichas ventanas y funcionalidades al editor, junto con modificaciones para que solo trabaje con archivos de código Python (.py) y algunas correcciones en el sistema de plug-ins.

Junto con las modificaciones mencionadas, se escribieron test unitarios para las ventanas y clases de lógica desarrolladas, asegurando al igual que en el firmware, la calidad del software requerida para un trabajo de especialización.

3.2.2. Diseño del IDE

Se modificó el código del proyecto EDILE y se incorporó una barra de botones junto con una nueva opción de menú, llamada “EDU-CIAA”.

La incorporación de este nuevo menú fue posible gracias al sistema de plug-ins con el que cuenta en editor base, en la figura 3.6 se aprecia el menú agregado

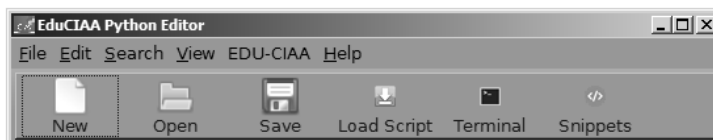


FIGURA 3.6: Botones adicionales del IDE.

así como también los botones que poseen las mismas funcionalidades que las opciones del menú.

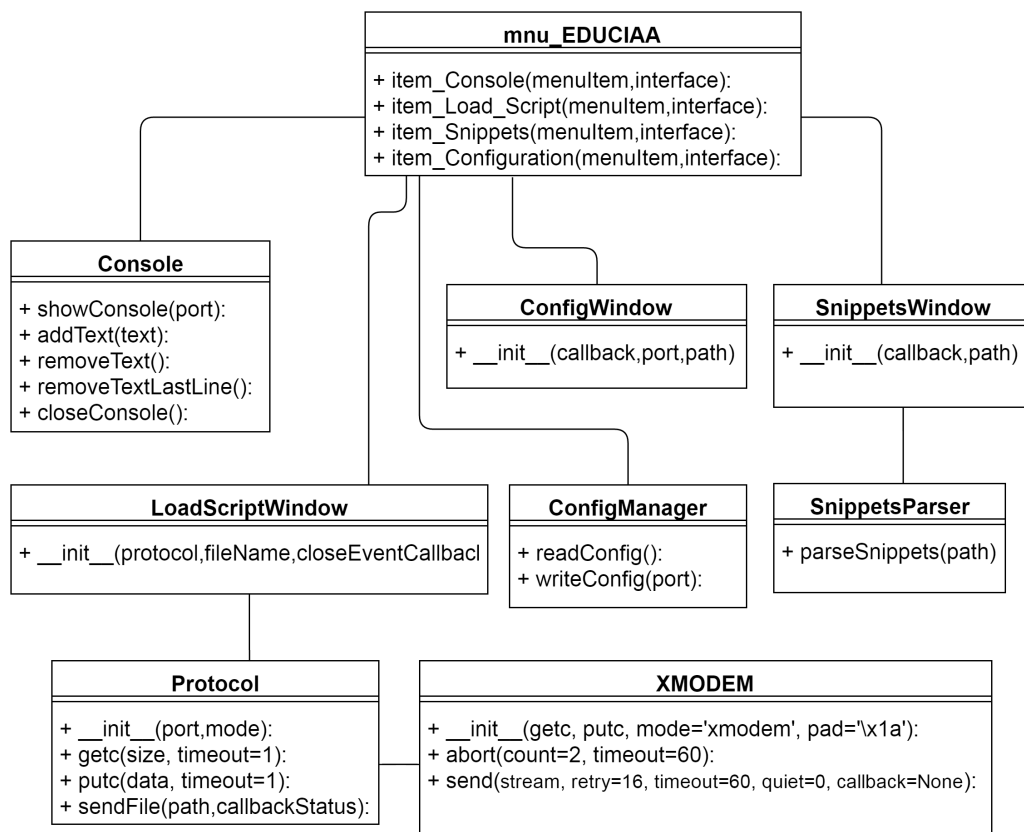


FIGURA 3.7: Diagrama de clases de las funcionalidades agregadas al IDE.

En la figura 3.7 se observan las clases desarrolladas. La clase que funciona como un plug-in del editor, es la llamada `mnu_EDUCIAA`, en ella se ejecutan métodos según qué opción de menú eligió el usuario (o que botón presionó). Esta clase hace las veces de *controller*⁵ y dispara las diferentes ventanas según la opción seleccionada. A continuación se detallan las clases creadas:

- **mnu_EDUCIAA** : Recibe los eventos de click sobre los ítems del menú o los botones y lanza las diferentes ventanas (Terminal serie, Configuración, Snippets y Grabación).
- **Console** : Ventana que se conecta por el puerto serial configurado y muestra en pantalla los caracteres recibidos, de la misma forma captura las teclas que el usuario presiona y las envía.

⁵controller: Componente del patrón de diseño MVC el cual se encarga de recibir eventos y realizar acciones.

- **ConfigWindow** : Ventana que permite al usuario configurar el puerto serie mediante el cual se comunica el IDE con la EDU-CIAA-NXP.
- **SnippetsWindow** : Ventana que muestra una lista de porciones de código de ejemplo, y que por medio de un botón permite agregarlas al texto que el usuario está escribiendo.
- **LoadScriptWindow** : Ventana que se encarga del envío del archivo a la placa utilizando el protocolo xmodem y de mostrar el progreso.
- **ConfigManager** : Esta clase se encarga de escribir y leer en un archivo la configuración del IDE (es decir el puerto serie que seleccionó el usuario).
- **SnippetsParser** : Todos los ejemplos que se muestran en la ventana de Snippets, se leen de un archivo XML. Esta clase se encarga de leer dicho archivo y devolver la información para que la ventana la pueda mostrar.
- **Protocol** : Recibe como dato el puerto serie y la ruta al archivo a transmitir, y se encarga de utilizar la biblioteca XMODEM para enviar el archivo hacia la placa.
- **XMODEM** : Biblioteca que implementa el protocolo *xmodem*⁶ y permite el envío del archivo a la placa.

El diseño de las ventanas y sus contenidos fueron creados con el programa Glade[4] utilizado para crear ventanas para la biblioteca gráfica pyGTK.

La lógica de cada ventana se encuentra implementada dentro de cada una de las clases, las cuales utilizan la biblioteca pyGTK para construir la interfaz gráfica por medio de un archivo XML (extensión .glade) donde se define el formato de la ventana y los componentes que posee dentro (Botones, etc.)

```

1      try :
2          builder = gtk.Builder()
3          builder.add_from_file(basePath+"/LoadScriptWindow.glade")
4      except Exception,e:
5          print(e)
6          return
7
8      self.buttonClose = builder.get_object("btnClose")

```

ALGORITMO 3.2: Porción de código que muestra la creación de la ventana a partir del archivo glade

En el algoritmo 3.2 se muestra la creación del objeto builder el cual obtiene los datos de la ventana gráfica del archivo LoadScriptWindow.glade, archivo creado con el programa Glade. Luego en la línea 8 se observa cómo mediante el método `get_objet()` se obtiene una referencia de un objeto Button definido para dicha ventana y que tiene asignado el nombre "btnClose". De esta manera se obtienen las referencias de los objetos definidos en el archivo XML y que componen la ventana, permitiendo la interacción con los mismos desde el código.

⁶xmodem: Protocolo simple de transferencia de datos por medio de paquetes de 128bytes creado en 1977.

3.2.3. Envío del archivo a la placa

Para el envío del archivo a la placa, se utilizó el protocolo xmodem debido a su sencillez, este protocolo permite la transferencia de datos en paquetes de 128 bytes o 1kbytes, por cada paquete transferido el receptor envía un acuse de recibo (ACK) o un aviso de que hubo un error (NACK).

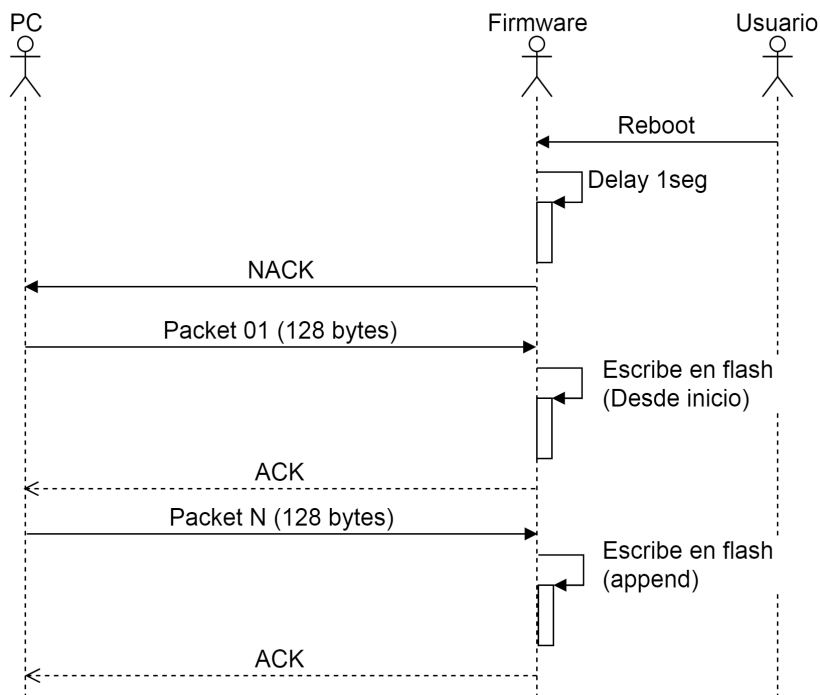


FIGURA 3.8: Diagrama de secuencia que describe la comunicación entre el IDE y el firmware.

En la figura 3.8 se observa el diagrama de secuencia correspondiente a la transferencia de datos entre el firmware desarrollado y la lógica de la ventana LoadScriptWindow. En él se observa que al reiniciar la placa mediante su pulsador de reset, el firmware envía una trama NACK por el puerto serie, en el caso de que el IDE se encuentre en el modo de inicializar una grabación (porque el usuario abrió la ventana de grabación) se recibirá dicha trama y se repondrá con el primer paquete de datos de 128bytes, al cual el firmware que recibe dicha trama responderá con ACK si pudo grabarlo en la memoria de programa del microcontrolador o NACK en caso de error.

Este proceso continúa hasta que el IDE envía todas las tramas que corresponden al contenido del archivo que se está enviando. El firmware irá agregando los datos recibidos al final de la memoria donde se grabaron los datos de la trama anterior.

3.3. Documentación

3.3.1. Diseño de proyectos de ejemplo

3.3.2. Documentación de las bibliotecas implementadas

Capítulo 4

Ensayos y Resultados

- 4.1. Test unitarios para bibliotecas para el manejo de periféricos desde C
- 4.2. Test unitarios para bibliotecas para el manejo de periféricos desde Python
- 4.3. Test Unitarios para ventanas que componen el IDE

Capítulo 5

Conclusiones

5.1. Conclusiones generales

5.2. Próximos pasos

Apéndice A

Creación de un módulo y clase Python desde código C

A.1. Creación de un módulo

Para poder definir un módulo personalizado, y luego definir clases dentro de él, se debe crear un archivo de código C en el cual se define una struct que representará el módulo, declarando todas las clases que estarán dentro de él. Por ejemplo el archivo `modpyb.c` se utilizó para declarar el módulo “pyb” el cual contiene todas las clases para el manejo de los periféricos.

```
1 STATIC const mp_map_elem_t pyb_module_globals_table[] = {
2     { MP_OBJ_NEW_QSTR(MP_QSTR__name__), MP_OBJ_NEW_QSTR(MP_QSTR_pyb) },
3     { MP_OBJ_NEW_QSTR(MP_QSTR_delay), (mp_obj_t)&pyb_delay_obj },
4     { MP_OBJ_NEW_QSTR(MP_QSTR_LED), (mp_obj_t)&pyb_led_type },
5     //...
```

ALGORITMO A.1: Estructura donde se definen los atributos funciones y clases del módulo.

En la porción de código del algoritmo A.1 se observan 3 ítems, cada ítem posee dos campos, el primero es un texto que representa el nombre de un atributo, función o clase que posee el módulo:

- `MP_OBJ_NEW_QSTR(MP_QSTR__name__)`: Atributo `__name__` del módulo.
- `MP_OBJ_NEW_QSTR(MP_QSTR_delay)`: Funcion delay del módulo
- `MP_OBJ_NEW_QSTR(MP_QSTR_LED)`: Clase LED del módulo

y el segundo es el valor, que puede ser otro texto, o un puntero a una struct del tipo `mp_obj_t` que representará una función o una clase:

- `MP_OBJ_NEW_QSTR(MP_QSTR_pyb)`: String “pyb” definido en `qstrdefsport.h`
- `(mp_obj_t)&pyb_delay_obj`: Puntero a struct que representa la función delay.
- `(mp_obj_t)&pyb_led_type`: Puntero a struct que representa la clase LED.

El texto “pyb” así como todos los textos que se utilizan desde el código Python, se deben definir en el archivo `qstrdefsport.h` con la macro “Q()” y para ser referenciados desde código C, se utiliza la macro “MP_OBJ_NEW_QSTR()”

```

1 STATIC mp_obj_t pyb_delay(mp_obj_t ms_in) {
2     mp_int_t ms = mp_obj_get_int(ms_in);
3     if (ms >= 0) {
4         mp_hal_milli_delay(ms);
5     }
6     return mp_const_none;
7 }
8 STATIC MP_DEFINE_CONST_FUN_OBJ_1(pyb_delay_obj, pyb_delay);

```

ALGORITMO A.2: Definición de la función de C que se ejecuta al invocar la función delay desde Python.

La función “delay” se encuentra definida dentro del mismo archivo modpyb.c y puede observarse en el algoritmo A.2. Primero se define la función “pyb_delay”, y luego mediante la macro “MP_DEFINE_CONST_FUN_OBJ_1()” se define la struct “pyb_delay_obj”, la cual se utilizó para declarar en el array de ítems del módulo mencionado previamente.

La clase LED no se encuentra definida dentro del archivo sino que la misma se crea en un archivo aparte, llamado modpybled.c

```

1 extern const struct _mp_obj_module_t pyb_module;
2
3 #define MICROPY_PORT_BUILTIN_MODULES \
4     { MP_OBJ_NEW_QSTR(MP_QSTR_pyb), (mp_obj_t)&pyb_module }, \
5     // ...

```

ALGORITMO A.3: Inclusión de el módulo pyb a la lista de módulos disponibles.

Para agregar el módulo “pyb” al intérprete, se debe editar el archivo mpconfigport.h y agregar las líneas indicadas en el algoritmo A.3. De esta manera el intérprete cargará el módulo “pyb” cuando se ejecute:

```
>>> import pyb
```

junto con todas sus funciones y clases, como la función “delay” o la clase LED.

A.2. Incorporación de una clase al módulo

A continuación se pasa a definir la clase LED, declarada dentro del módulo “pyb”. Para ello se crea el archivo modpybled.c en donde se define la struct “pyb_led_type” (la que se utilizó en el archivo modpyb.c).

```

1 const mp_obj_type_t pyb_led_type = {
2     { &mp_type_type },
3     .name = MP_QSTR_LED,
4     .print = pyb_led_print,
5     .make_new = pyb_led_make_new,
6     .locals_dict = (mp_obj_t)&pyb_led_locals_dict,
7 };

```

ALGORITMO A.4: Estructura que define la clase LED.

En el algoritmo A.4 se define la clase LED declarando el campo “make_new” el cual debe ser cargado con una función que hará las veces de constructor, es decir, al crear desde el código Python un objeto del tipo LED se ejecutará la función “pyb_led_make_new()” que está definida en este archivo.

```

1 STATIC const mp_map_elem_t pyb_led_locals_dict_table[] = {
2     { MP_OBJ_NEW_QSTR(MP_QSTR_on), (mp_obj_t)&pyb_led_on_obj },
3     { MP_OBJ_NEW_QSTR(MP_QSTR_off), (mp_obj_t)&pyb_led_off_obj },
4     { MP_OBJ_NEW_QSTR(MP_QSTR_toggle), (mp_obj_t)&pyb_led_toggle_obj },
5     { MP_OBJ_NEW_QSTR(MP_QSTR_intensity), (mp_obj_t)&
      pyb_led_intensity_obj },
6     { MP_OBJ_NEW_QSTR(MP_QSTR_value), (mp_obj_t)&pyb_led_value_obj },
7 };
8 STATIC MP_DEFINE_CONST_DICT(pyb_led_locals_dict,
      pyb_led_locals_dict_table);

```

ALGORITMO A.5: Definición de métodos de la clase LED.

El campo “locals_dict” se carga con el array “pyb_led_locals_dict” en donde están declarados todos los métodos y atributos de esta clase como lo muestra el algoritmo A.5

En esta definición puede verse que se repite el mismo formato que para definir las funciones y clases del módulo “pyb”, los ítems poseen dos campos, el primero es el nombre de los métodos y el segundo los punteros a una struct que representa las funciones definidas en este mismo archivo.

TABLA A.1: Relación entre métodos Python y funciones C definidas en modpybled.c para la clase LED

Método Python	Estructura C	Función C
LED	-	pyb_led_make_new
on	pyb_led_on_obj	pyb_led_on
off	pyb_led_off_obj	pyb_led_off
toggle	pyb_led_toggle_obj	pyb_led_toggle
intensity	pyb_led_intensity_obj	pyb_led_intensity
value	pyb_led_value_obj	pyb_led_value

Tomando las definiciones en la struct del algoritmo A.5 puede construirse la tabla A.1, en donde se resume que para cada método existe una struct que representa a una función de C así como también la función.

Con estas definiciones se puede observar que al crear un objeto LED desde el código Python:

```

>>> import pyb
>>> led = pyb.LED()

```

se ejecutará la función “pyb_make_new()” y al ejecutarse el método “on()”:

```

>>> led.on()

```

se ejecutará la función “pyb_led_on()” a la cual hace referencia la struct “pyb_led_on_obj”.

```

1 mp_obj_t pyb_led_on(mp_obj_t self_in) {
2     // ...
3 }
4 STATIC MP_DEFINE_CONST_FUN_OBJ_1(pyb_led_on_obj, pyb_led_on);

```

ALGORITMO A.6: Función que se ejecuta al invocar el método on().

Dentro de esta función (algoritmo A.6) es donde ejecutaremos las funciones de la biblioteca uPython HAL para el manejo de los leds.

A.3. Definición de un método de una clase

A continuación se explicará cómo definir la función que hará las veces de constructor de la clase. Para ello se comienza por definir el tipo de dato `pyb_led_obj_t` que representará un objeto LED como una struct de C.

```

1 typedef struct _pyb_led_obj_t {
2     mp_obj_base_t base;
3 } pyb_led_obj_t;
4
5 STATIC const pyb_led_obj_t pyb_led_obj[] = {
6     {&pyb_led_type},
7     {&pyb_led_type},
8     {&pyb_led_type},
9     {&pyb_led_type},
10    {&pyb_led_type},
11    {&pyb_led_type},
12 };
13
14 #define NUM_LED MP_ARRAY_SIZE(pyb_led_obj)
15 #define LED_ID(obj) ((obj) - &pyb_led_obj[0] + 1)

```

ALGORITMO A.7: Definición de un array de objetos LED para C.

Luego se define un array de 6 variables de este tipo, ya que la clase LED podrá crear 6 objetos leds diferentes, porque la placa posee 6 leds. Como se muestra en el algoritmo A.7 dentro de la struct del tipo `pyb_led_obj_t` se debe definir un campo `mp_obj_base_t` y luego todos los campos adicionales que se requieran, en este caso no se utiliza ninguno. Cada ítem del array (cada “objeto” LED) se carga con los valores definidos en el algoritmo A.4.

```

1 STATIC mp_obj_t pyb_led_make_new(mp_obj_t type_in, mp_uint_t n_args,
2                                 mp_uint_t n_kw, const mp_obj_t *args) {
3
4     mp_arg_check_num(n_args, n_kw, 1, 1, false);
5
6     mp_int_t led_id = mp_obj_get_int(args[0]);
7
8     if (!(1 <= led_id && led_id <= NUM_LED)) {
9         nlr_raise(mp_obj_new_exception_msg_varg(&mp_type_ValueError,
10                                                "LED %d does not exist", led_id));
11     }
12
13     return (mp_obj_t)&pyb_led_obj[led_id - 1];
14 }

```

ALGORITMO A.8: Definición de la función constructor.

En el algoritmo A.8 se muestra el código de la función que se ejecuta al invocarse el constructor de la clase LED. En la línea 4 se realiza un chequeo de los argumentos recibidos, los cuales deben ser 1 (el número de led). En la línea 6 se lee el valor del argumento y se almacena en la variable `led_id`. Con este valor, se chequea que el número de led sea válido (línea 8). En el caso de ser válido se devuelve el puntero del ítem del array de variables del tipo `pyb_led_obj_t` que corresponde a la posición del número de led recibido como argumento (línea 13), de lo contrario se lanza una excepción indicando que el número de led no es correcto.

Esta variable devuelta en el constructor, es la que recibirán todas las funciones que se ejecuten al invocar los métodos del objeto Python, junto con sus argumentos si los tienen.


```
1 mp_obj_t pyb_led_on(mp_obj_t self_in) {  
2     pyb_led_obj_t *self = self_in;  
3     mp_hal_setLed(LED_ID(self)-1, 0);  
4     return mp_const_none;  
5 }
```

ALGORITMO A.9: Definición de la función que se ejecuta al invocar el método on().

En el algoritmo A.9 se muestra el código que se ejecuta al invocarse el método “on()” de un objeto led. Como se mencionó previamente, se recibe como argumento el puntero del ítem del array que devolvió la función que actúa como constructor, es decir la referencia del ítem que representa al objeto led desde C. Llamamos a este puntero `self_in` y mediante la macro `LED_ID` (definida en el algoritmo A.7) se calcula el número de led que corresponde, para luego ejecutar la función de la capa uPython HAL que se encarga de cambiar dicho led de estado.

Apéndice B

Tests funcionales del uso de periféricos desde código Python

Apéndice C

Tests funcionales del uso del IDE

Bibliografía

- [1] M. de Sá-Soares. Brito. «Assessment frequency in introductory computer programming disciplines». En: *Computers in Human Behavior* (2014), págs. 623-628. ISSN: 07475632. URL: <http://fulltext.study/preview/pdf/350717.pdf>.
- [2] EDILE project. *EDILE Python Editor*. Disponible: 2016-10-20. URL: <https://code.google.com/archive/p/edile/>.
- [3] Mauricio Dávila. Marisa Panizzi. Darío Rodríguez. Ramón García-Martínez. «Propuesta de Protocolo de Análisis de Dispositivos de Enseñanza de Programación». En: *Proceedings XI Congreso de Tecnología en Educación y Educación en Tecnología* (jun. de 2016), págs. 428-437. URL: <http://sistemas.unla.edu.ar/sistemas/gisi/papers/TEYET-2016-MD-MP-DR-RGM-TeE.pdf>.
- [4] Glade. *A User Interface Designer*. Disponible: 2016-10-20. URL: <https://glade.gnome.org>.
- [5] Lisa C. Kaczmarczyk. J. Philip East. Elizabeth R. Petrick. Geoffrey L. Herman. «Identifying Student Misconceptions of Programming». En: *SIGCSE '10 Proceedings of the 41st ACM technical symposium on Computer science education*. (2010), págs. 107-111. URL: <http://publish.illinois.edu/glherman/files/2016/03/2010-SIGCSE-Programming-Misconceptions.pdf>.
- [6] Jaltek Systems. *Jaltek Systems Web Page*. Disponible: 2016-10-20. URL: <http://www.jaltek.com>.
- [7] Resnick M. Maloney J. Monroy-Hernández A. Rusk N. Eastmond E. Brennan K. Kafai. «Scratch: programming for all.» En: *Communications of the ACM* (2009), págs. 60-67. URL: <http://cacm.acm.org/magazines/2009/11/48421-scratch-programming-for-all/fulltext>.
- [8] Martin Ribelotta. *Port de MicroPython para la EDU-CIAA-NXP*. Disponible: 2016-10-20. URL: <https://github.com/martinribelotta/micropython>.
- [9] MicroPython. *MicroPython - Python for microcontrollers*. Disponible: 2016-10-20. URL: <https://micropython.org>.