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

Al Dr. Ing. Ariel Lutemberg, quien me otorgó la beca que me permitió cursar la carrera de especialización y hacer posible este trabajo.

A Martín Ribelotta, creador de la base sobre la que se desarrolló este trabajo y quien me ha ayudado desde el comienzo contestando absolutamente todas mis dudas.

A mis compañeros de la 3er cohorte, con quienes compartimos un año agotador pero lleno de nuevos conocimientos que nos permitieron formarnos en esta especialidad.

Por último a mi director Esp. Ing. Eric Pernía y a mis jurados Dr. Ing. Pablo Gomez, Ing. Alejandro Permingeat y Esp. Ing. Pablo Ridolfi quienes han dedicado mucho de su escaso tiempo en evaluar este trabajo.

Índice general

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción General

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

1.1. Dificultades planteadas en la enseñanza de programación de sistemas embebidos.

Un alumno de nivel secundario o universitario que comienza su formación en el ámbito de la informática o electrónica, y es instruído 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[9].

Los típicos problemas a los que un estudiante inicial de informática debe enfrentarse según (Kaczmarczyk, East, Petrick y Herman, 2010) [8] 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 (sintaxis, IDE y conocimientos de arquitectura de microprocesadores), 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) [11]. Este tipo de herramientas oculta las capas de bajo nivel requeridas para la utilización de un hardware específico, permitiendo a los alumnos de nivel inicial practicar programación con el incentivo extra de poder crear fácilmente dispositivos electrónicos que realicen determinadas acciones, sin caer

[1] IDE: Entorno de Desarrollo Integrado.

[2] Scratch: Lenguaje de programación basado en bloques.

en la necesidad de adquirir previamente conocimientos avanzados de sistemas embebidos.

1.2. Objetivo y alcance

El objetivo principal del presente trabajo 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.

Para eso 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[5]. 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 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.

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

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

Capítulo 2

Introducción Específica

En esta capítulo se detalla la plataforma de hardware utilizada, y se hace 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 la acción conjunta de *ACSE*[1] y *CADIEEL*[2], 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" cuyo diagrama en bloques se observa en la figura 2.1.

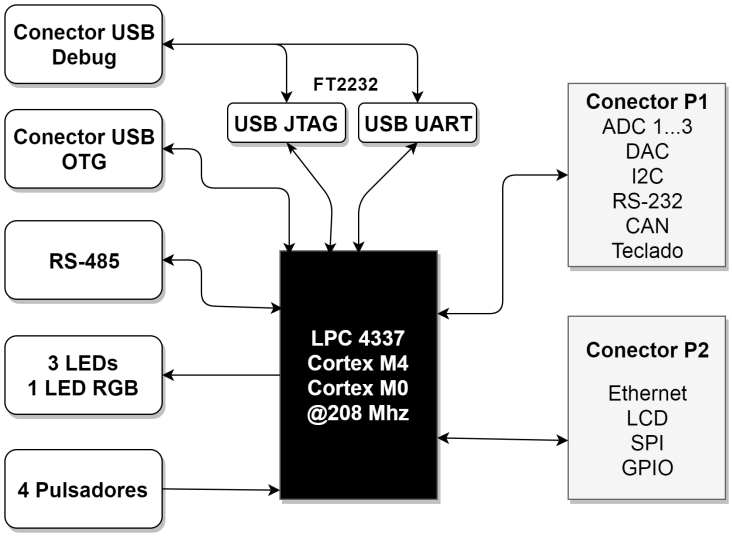

FIGURA 2.1: Diagrama en bloques de la plataforma EDU-CIAA-NXP

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

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

La EDU-CIAA-NXP utiliza un microcontrolador NXP LPC 4337 JDB 144 (Dual-core Cortex-M4 + Cortex-M0). 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, asi como tambien 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 micontrontrolador (GPIOs, UARTs, SPI, I2C, etc.). 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.

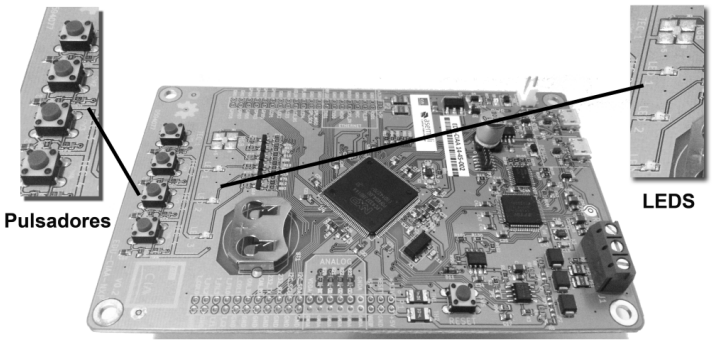

FIGURA 2.2: Foto de una placa EDU-CIAA-NXP

2.2. Utilización de MicroPython

El proyecto MicroPython [13] es un desarrollo de firmware realizado por Damien George, el cual fue pensado para correr sobre la plataforma Pyboard, desarrollada por Jaltek Systems [10]. Este proyecto permite la ejecución de código Python y la utilización de los periféricos que posee la placa, desde dicho lenguaje.

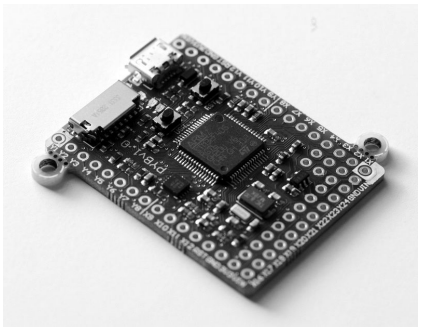

FIGURA 2.3: Foto de una placa Pyboard

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 [12]. 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 presente trabajo 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 detalla la arquitectura del firmware implementado en este trabajo por medio de la figura 2.4. 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.

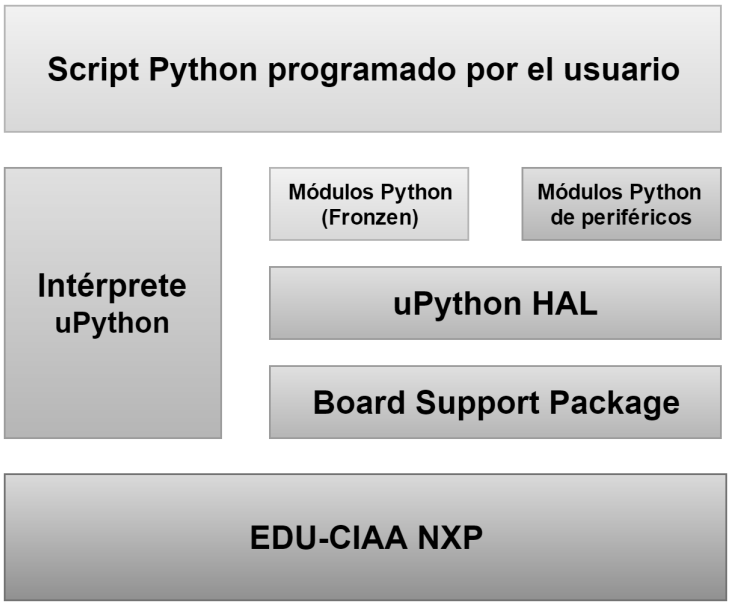

FIGURA 2.4: Diagrama en bloques de la arquitectura del Firmware

[3]Garbage Collector: Gestor automático de memoria dinámica.

Estos tres bloques mencionados 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 indicada en la figura 2.5. En la misma 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.

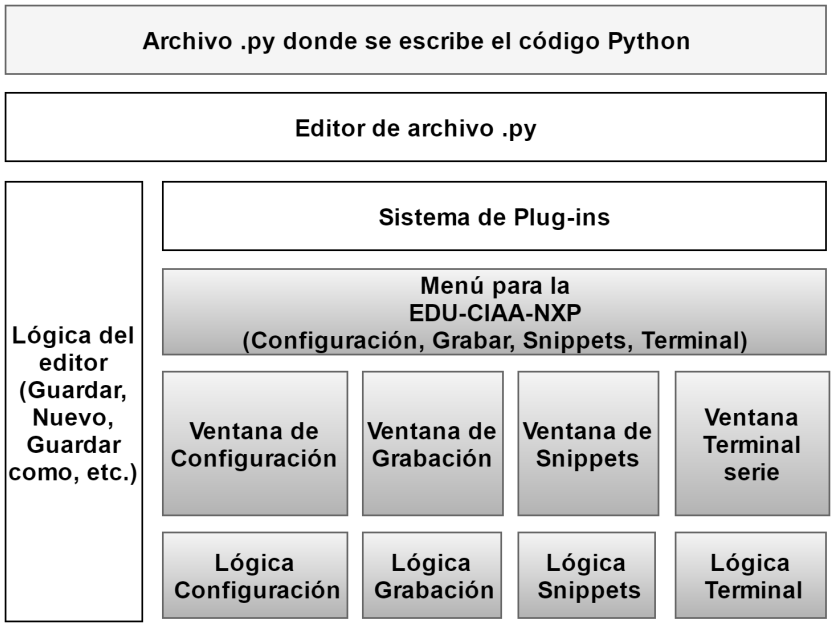

FIGURA 2.5: Diagrama en bloques de la arquitectura del Software del IDE

También 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 envian 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 desarrollado por Martín Ribelotta, 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.

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.

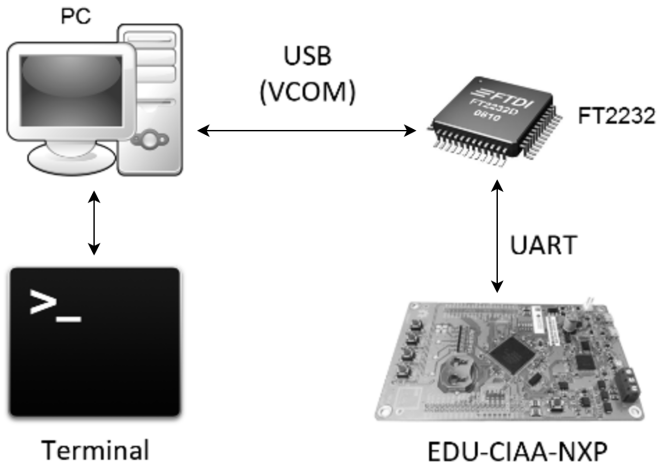

FIGURA 3.1: Conexión de la EDU-CIAA-NXP con la PC

[1]REPL: 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 como por ejemplo:

```
» 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 incluído 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.

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.

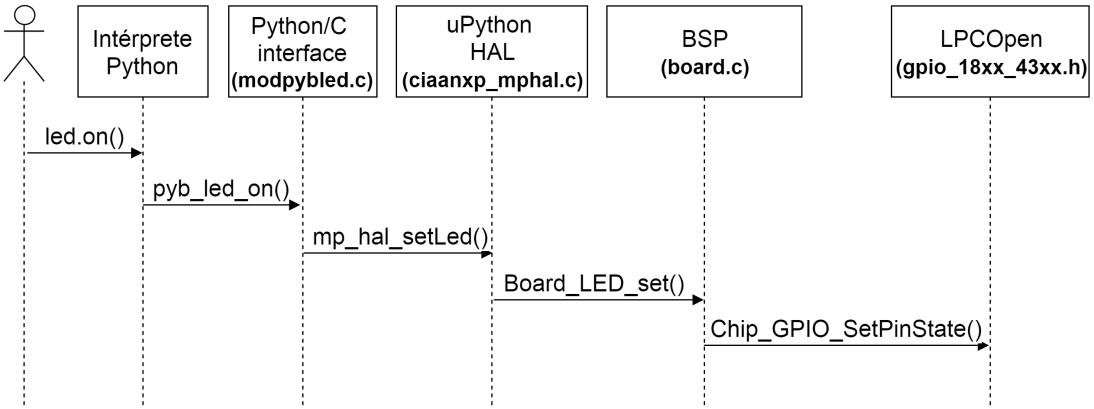

FIGURA 3.2: Anidamiento de llamadas a funciones desde Python a C

Al ejecutarse la llamada del método on() desde el objeto led, se 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. Cada método que pose el objeto led tendrá asociada una función dentro del archivo modpybled.c.

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.

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

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

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

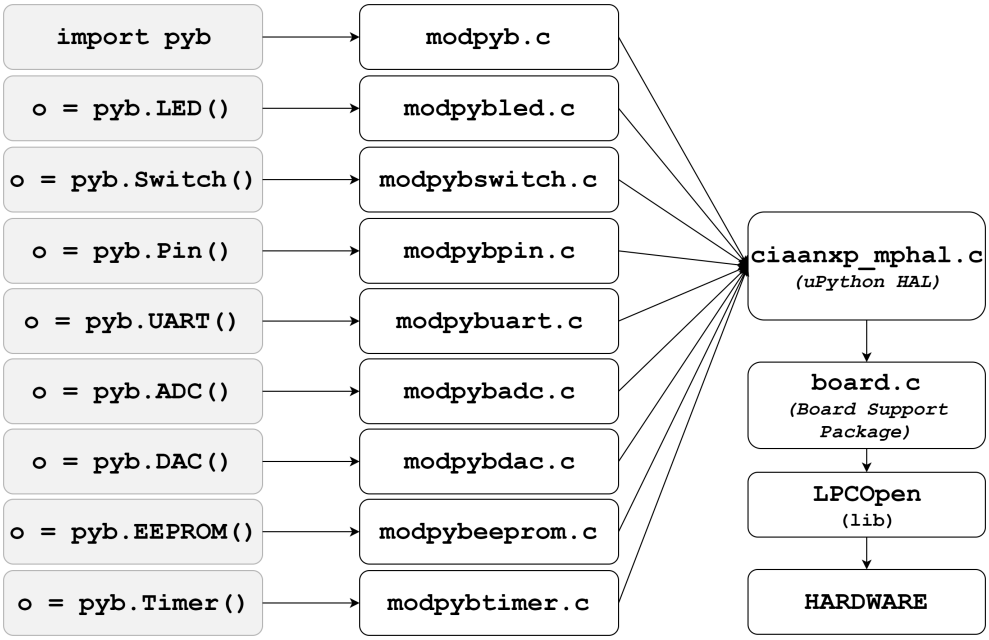

FIGURA 3.3: Relación entre las clases Python y los archivos del Firmware

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.

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.

```
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

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 (línea 11), 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 boquea 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.

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:

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

- 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.

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.

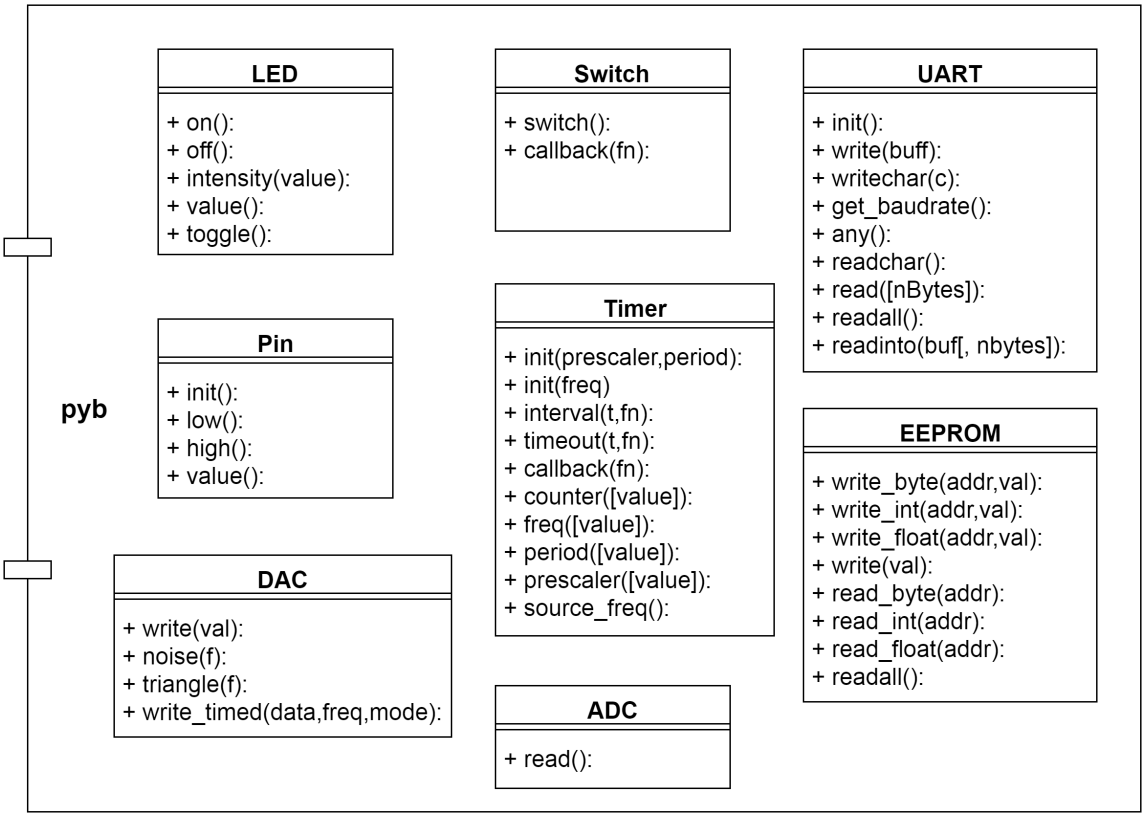

FIGURA 3.4: Diagrama de clases para manejo de periféricos

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 muestra en la figura 3.5. El mismo 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.

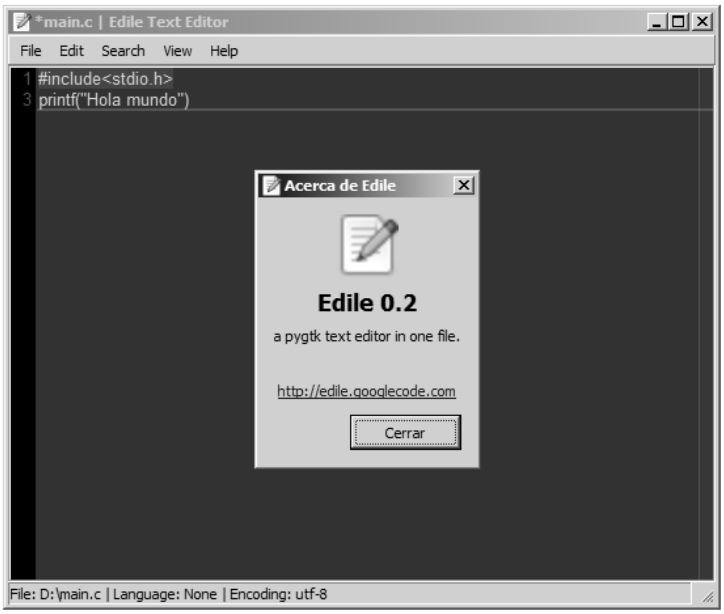

FIGURA 3.5: Procesador de texto EDILE v0.2

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 menúes que ejecuten ventanas adicionales que cumplan con cierta funcionalidad, de modo que el diseño de software que se realizó en el presente 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 así como también los botones que poseen las mismas funcionalidades que las opciones del menú.

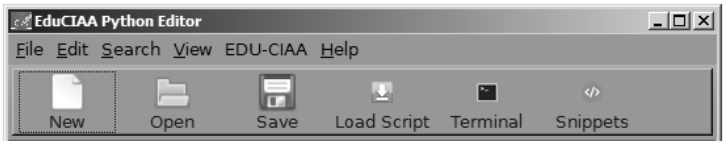

FIGURA 3.6: Botones adicionales del IDE.

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.

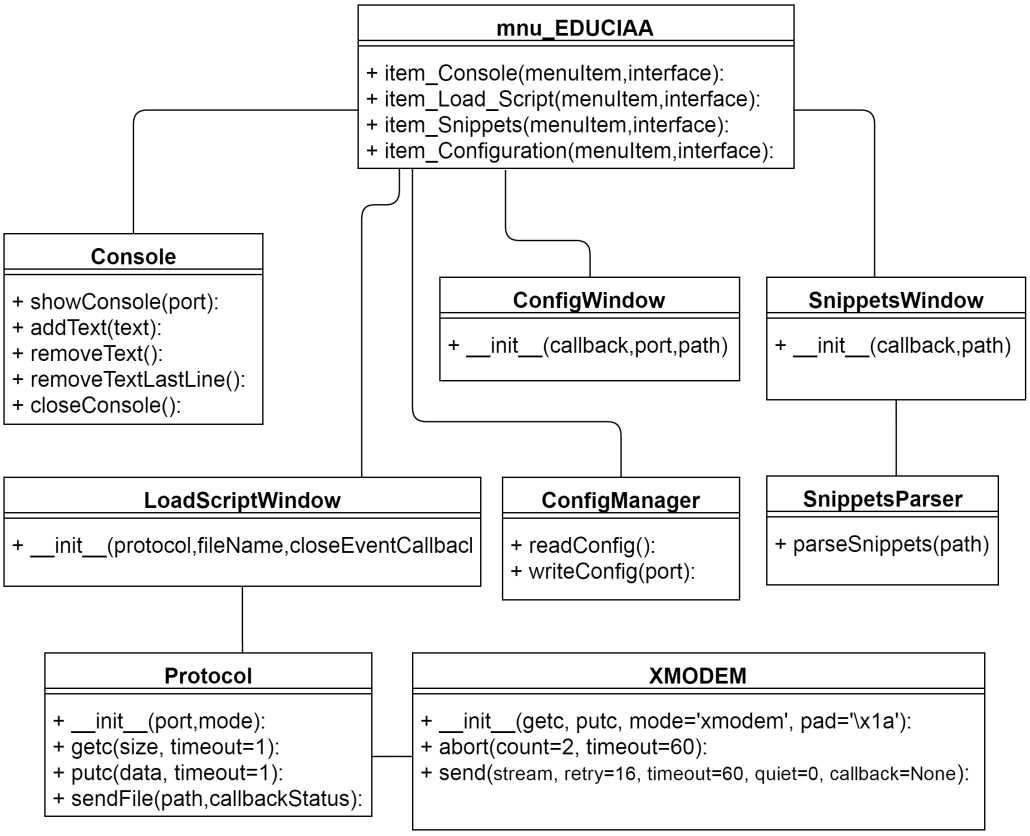

FIGURA 3.7: Diagrama de clases de las funcionalidades agregadas al IDE.

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.
- **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.

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

- **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[6] 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.)

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.

```
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

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 trasferencia 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).

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

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

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 reponderá 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.

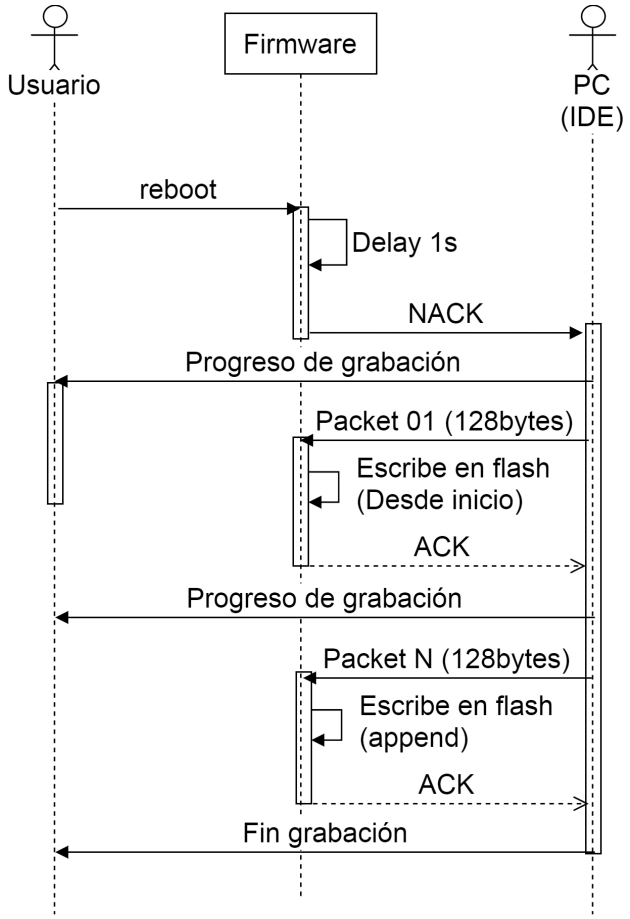

FIGURA 3.8: Diagrama de secuencia que describe la comunicación entre el IDE y el firmware.

Este proceso continúa hasta que el IDE envía todas las tramas que corresponden al contenido del archivo que se esta 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

Como se indica en el capítulo 1, el desarrollo del firmware y del software no es suficiente para que un proyecto educativo tenga el impacto deseado, sino que debe estar acompañado por la documentación y los ejemplos adecuados. Esta es la razón por la que el IDE posee una ventana de Snippets con porciones de código de ejemplo, sin embargo esto no es suficiente, ya que no existe explicación alguna de dichas porciones de código. Además de los Snippets, se agregaron proyectos de ejemplo y documentación de las clases y métodos desarrollados para el manejo de los periféricos.

3.3.1. Proyectos de ejemplo

Para la publicación de ejemplos completos, se creó un repositorio público en GitHub[4] en donde pueden descargarse diferentes tipos de proyectos con una explicación detallada del código que se implementó. Los ejemplos se dividen en tres categorías:

- **Inicial**: cubren conceptos básicos de programación y del lenguaje Python.
- **Intermedio**: proyectos que requieren conocimientos intermedios de programación y básicos de electrónica.
- **Avanzado**: proyectos que requieren conocimientos avanzados de programación y electrónica..

Cada proyecto es acompañado por un archivo README.md en donde se aclaran las funciones y sentencias utilizadas, y se explica el funcionamiento del código paso por paso.

La idea es que la comunidad que utilice esta plataforma para desarrollar trabajos, en escuelas secundarias o universidades, aporte los trabajos implementados al repositorio, para que otros estudiantes puedan tomarlos como base para nuevos proyectos o simplemente para aprender de lo desarrollado.

3.3.2. Documentación de las bibliotecas implementadas

Para acompañar a los proyectos de ejemplo y dar una base sólida del uso y soporte de los periféricos, se escribió una documentación detallada de cada clase con todos sus métodos, escribiendo también uno o más ejemplos del uso de algunos métodos en particular.

Esta documentación se escribió en la página del proyecto CIAA, en la sección Python[3], la cual también es de acceso público, cumpliendo así con uno de los requerimientos mencionados en el capítulo 2. El link a esta página se encuentra accesible desde la ventana de Snippets del IDE.

Capítulo 4

Ensayos y Resultados

En este capítulo se explica el desarrollo e implementación de tests unitarios y funcionales para el firmware y el software.

4.1. Tests unitarios para bibliotecas para el manejo de periféricos desde C

Para asegurar el correcto funcionamiento de las bibliotecas para el manejo de periféricos, se implementaron tests unitarios para la capa uPython HAL utilizando la técnica *TAD*[1]. Mediante estos tests se probó el uso de las funciones de dicha capa cubriendo la complejidad ciclomática de cada una de ellas.

Teniendo como principal objetivo un consumo reducido de memoria de datos y de programa para la implementación de los tests (debido a que se ejecutan en la propia placa sin utilizar *mocks*[2]), no se utilizó ninguna biblioteca existente, sino que se desarrolló una pequeña biblioteca mayormente compuesta por *macros*[3], que permitió ejecutar los tests unitarios de una manera simple y controlada.

En el algotimo 4.1 se muestra la función utest_startTest la cual se encarga de ejecutar un test unitario. La misma recibe los argumentos que se detallan a continuación:

- **fncTest**: una función donde se escribirá el test unitario.
- **fncBefore**: una función que en el caso de existir, se ejecutará antes de fncTest (puede utilizarse para inicializaciones antes de la ejecución del test).
- **testName**: un texto con el nombre del test, el cual se imprime antes de ejecutar las funciones (línea 8).

El control de errores se observa en la línea 7 en donde se pone a cero la variable global utest_flagTestError, luego de la ejecución de las funciones pasadas como argumento, en la línea 13 se evalúa el estado de dicha variable, la cual se pudo cargar con el valor 1 en la función del test si hubo un error. En dicho caso se imprime un mensaje de error (línea 15) indicando mediante otras variables globales el nombre del archivo y el número de línea donde se produjo el error en el test.

[1] TAD:Test-After Development. Técnica en la cual los tests se escriben luego de escribir el código.

[2] Mock:Porción de código simple que simula otra porción de código más compleja.

[3] Macro:Fragmento de código con un nombre asociado.El precompilador reemplaza el nombre por el código.

```
1  void utest_startTest(void(*fncTest)(void),
2                       void(*fncBefore)(void),
3                       char* testName)
4  {
5    if(fncTest!=0)
6    {
7      utest_flagTestError=0;
8      utest_print1("TESTING: %s\r\n",testName);
9      if(fncBefore!=0)
10       fncBefore();
11     utest_totalTestsCounter++;
12     fncTest();
13     if(utest_flagTestError==1)
14     {
15       utest_print2("TEST FAILED. FILE: %s LINE: %d\r\n",
16                    utest_fileTestError,utest_lineTestError);
17     }
18     else
19     {
20       utest_okTestsCounter++;
21     }
22   }
23 }
```

ALGORITMO 4.1: Función que ejecuta un test unitario incluida en el archivo utest.c del firmware.

Las variables globales mencionadas previamente se cargan en la ejecución de la función de test, para ello en el archivo utest.h del firmware, se definieron una serie de macros que permiten ejecutar las comparaciones de valores (asserts) del test unitario.

En el algoritmo 4.2 se observa una de las macros definidas la cual es utilizada para comparar dos números enteros. En la línea 4 se realiza la comparación de los valores y en el caso de que no sean iguales se imprime un mensaje de error (línea 6) y se cargan las variables globales que son analizadas en la función utest_startTest.

```
1
2  #define utest_assertEqualsInt(A,B)
3  {
4    if(A!=B)
5    {
6      utest_print2("assert equals failed '%d' != '%d'\r\n",A,B);
7      utest_flagTestError=1;
8      utest_lineTestError = __LINE__;
9      utest_fileTestError = __FILE__;
10     return;
11   }
12 }
```

ALGORITMO 4.2: Ejemplo de una macro assert incluida en el archivo utest.h del firmware.

En el algoritmo 4.3 puede apreciarse el uso de la macro utest_assertEqualsInt la cual recibe el valor esperado (-1) y el valor devuelto por la función de la capa uPython HAL que escribe en la EEPROM. Esta función es uno de los tests unitarios escritos para el manejo de la EEPROM del microcontrolador. La macro realiza la comparación, y en caso de error, imprime el mensaje y carga las variables globales necesarias para que se puedan imprimir los datos del error. La sentencia return hace que se termine la ejecución de la función del test.

```
void testEEPROM2(void)
{
  int32_t r = mp_hal_writeByteEEPROM(0x8000,0x55); // invalid address
  utest_assertEqualsInt(-1,(int)r);
}
```

ALGORITMO 4.3: Ejemplo de un test unitario para la EEPROM usando una dirección inválida.

Para ejecutar la función testEEPROM2 así como el resto de las funciones de test para todos los periféricos, se definió el archivo mainTest.c, en el cual se ejecutan las llamadas a la función utest_startTest mencionada en el algoritmo 4.1 y en donde se colocaron líneas de código como la que se observa en el algoritmo 4.4 en una función llamada startTesting.

```
  utest_startTest(testEEPROM1,0,"EEPROM: write/read byte Test");
  utest_startTest(testEEPROM2,0,"EEPROM: write invalid address Test");
  utest_startTest(testEEPROM3,0,"EEPROM: read invalid address Test");
```

ALGORITMO 4.4: Ejemplo de ejecución de funciones de test en archivo mainTest.c.

Esta función startTesting se ejecuta en el main principal del firmware en el caso de que se compile indicando que se quieren ejecutar los tests (ver sección 4.3).

La estructura de archivos referidos a los tests se muestra en la figura 4.1. Los archivos llamados testsXXX.c poseen las funciones de cada test unitario referido a cada periferico, las cuales utilizan las macros definidas en utest.h para realizar las comparaciones (asserts). Todos estos archivos se encuentran dentro de una carpeta llamada testing, y los mismos no se incluyen en el firmware excepto que se compile el mismo para ejecutar los tests.

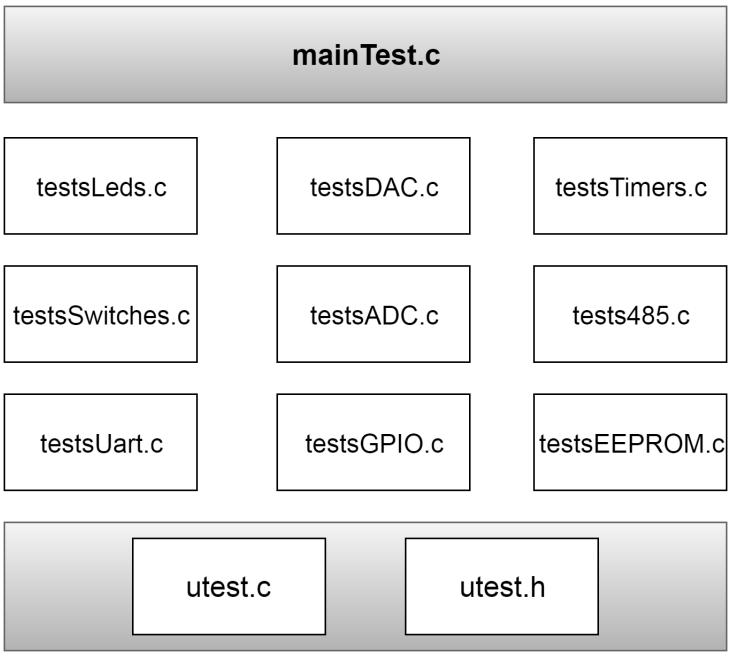

FIGURA 4.1: Estructura de archivos para la ejecución de tests unitarios de la capa uPython HAL.

4.2. Test unitarios para bibliotecas para el manejo de periféricos desde Python

Para asegurar el correcto funcionamiento de las bibliotecas para el manejo de periféricos desde Python, se implementaron tests unitarios para todas las clases Python desarrolladas. Mediante estos tests se pretendió probar el uso de los metodos de dicha capa cubriendo la complejidad ciclomática de cada una de ellos.

Una vez más teniendo como principal objetivo un consumo reducido de memoria de datos y de programa para la implementación de los tests, no se utilizó ninguna biblioteca de tests unitarios para Python, sino que se escribió una que posea lo mínimo indispensable para la ejecución de los tests, en un módulo llamado unittest.

Este módulo forma parte de las bibliotecas Python que pueden escribirse directamente en Python y no en C (bloque Frozen, ver sección 2.3), e incorporarse al firmware y al conjunto de módulos que el usuario puede utilizar para programar Python.

En esta biblioteca se definió la clase TestCase, una parte de ella puede observarse en el algoritmo 4.5 en donde se muestra la definición de la clase y de los métodos:

- **setUp**: método que se ejecutará antes del método de test.
- **tearDown**: método que se ejecutará despues del método de test.
- **assertEqual**: método assert encargado de comparar dos valores por igual. En el caso de no ser iguales, se lanza una excepción que será capturada por el método que ejecuta los tests.

```
1  class TestCase (object):
2      testCounter=0
3      testOKCounter=0
4
5      def setUp(self):
6          pass
7
8      def tearDown(self):
9          pass
10
11     def assertEqual(self,v1,v2,m):
12         if v1 != v2:
13             raise TestException(m)
```

ALGORITMO 4.5: Clase TestCase utilizada para crear tests unitarios para Python.

La clase posee muchos más métodos assertXXX para poder comparar todo tipo de datos.

Para escribir un test, se debe crear una clase que herede de TestCase, en donde se deben definir los métodos test_X siendo X un número comenzando de 1.

En el algoritmo 4.6 se muestra un ejemplo de una clase TestEEPROM la cual hereda de TestCase y posee definido los métodos de los tests, en el ejemplo se muestra solo el método test_1, el cual hace uso del método assertEqual, esto es posible debido a que la clase hereda de TestCase donde se encuentra definido el metodo.

```
1 from unittest import TestCase
2 import pyb
3
4 class TestEEPROM (TestCase):
5
6     def test_1(self):
7         eeprom = pyb.EEPROM()
8         eeprom.write_byte(0x0000,0x55)
9         val = eeprom.read_byte(0x0000)
10        self.assertEqual(0x55,val,"EEPROM addr 0x0000")
```

ALGORITMO 4.6: Clase que hereda de TestCase donde se definen los métodos de test para la EEPROM.

En la figura 4.2 el diagrama de clases del módulo unittest muestra la clase TestCase con todos sus métodos assertXXX y cómo las clases donde se definen los test (TestXXX) deben heredar de TestCase e implementar los métodos test_X como lo hace TestEEPROM.

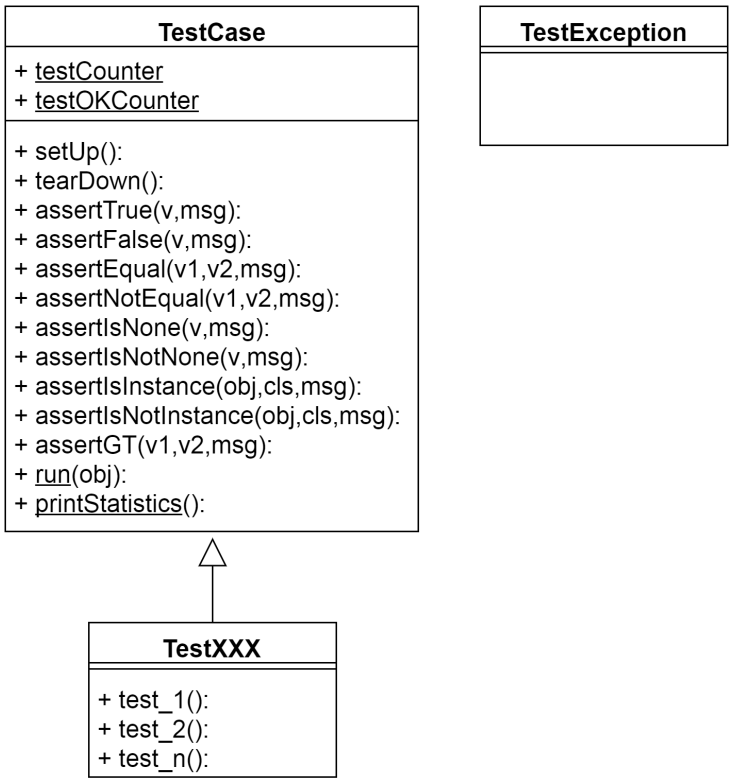

FIGURA 4.2: Diagrama de clases del módulo unittest.

Para ejecutar los tests, se utiliza el método estático run(), el cual debe recibir un objeto creado a partir de una clase que herede de TestCase y que posea los métodos test_X.

El método estático run(), encargado de ejecutar los tests unitarios, puede apreciarse en el algoritmo 4.7, en donde se observa que el mismo recibe un objeto y mediante el método hasattr (línea 6) se consulta si el mismo posee el método llamado nName (variable que contiene un string con el formato test_X). En el caso de que el objeto posea el método, se obtiene dicho método mediante getattr (línea 7) y dentro de un bloque try-except se procede a la ejecución del mismo (línea 12) ejecutando previamente el método setUp y posteriormente el método tearDown.

```
@staticmethod
def run(o):
    i=1
    while True:
        mName = "test_"+str(i)
        if hasattr(o,mName):
            m = getattr(o,mName)
            print("Testing "+mName+" ... ")
            TestCase.testCounter+=1
            try:
                o.setUp()
                m()
                o.tearDown()
                TestCase.testOKCounter+=1
            except Exception as e:
                print("ASSERT ERROR:"+str(e))
        else:
            break
        i+=1
```

ALGORITMO 4.7: Método que ejecuta los métodos de test en la biblioteca unittest.py implementada.

Si dentro de la ejecución del método m() un assert falla, se lanzará una excepción, y el bloque try-except la capturará y se imprimirá el mensaje de error. En el algoritmo 4.8 se observa la definición de una clase MainTest la cual posee un método run y dentro del mismo se colocaron todas las llamadas al método run de la clase TestCase, pasando como argumento un objeto de cada una de las clases que hereda de TestCase, las cuales se listan a continuación:

- **TestLeds**
- **TestSwitches**
- **TestUart**
- **TestEEPROM**
- **TestDAC**
- **TestADC**
- **TestGPIO**
- **TestRS485**
- **TestTimers**

```
class MainTest (object):
    def run(self):
        print("Running python tests")

        print("LEDs Tests")
        TestCase.run(TestLeds())
        print("_____________________")

        print("Switches Tests")
        TestCase.run(TestSwitches())
        print("_____________________")
        #...
```

ALGORITMO 4.8: Clase MainTest desde donde se ejecutan todos los tests Python.

4.3. Ejecución de los tests sobre la placa

Para llevar a cabo la ejecución de los tests unitarios sobre la placa, se modificó el archivo Makefile agregando la regla «test». Como se indica en el algoritmo 4.9 la regla test incluye en la compilación la definición de la macro TESTING (línea 1) y la definición de una variable de entorno MP_INCLUDE_TESTS (línea 2).

```
1 test: CFLAGS += -DTESTING
2 test: export MP_INCLUDE_TESTS = true
3 test: all
```

ALGORITMO 4.9: Regla test en Makefile.

La defínción TESTING se utiliza en el archivo main.c como se observa en el algoritmo 4.10 para incluir todos los archivos mencionados en las secciones anteriores, e incluir las llamadas a la función startTesting (línea 10), para ejecutar los tests unitarios para la capa uPython HAL, y la función do_str (línea 12) para ejecutar una porción de código Python que crea un objeto MainTest y ejecuta el método run de dicha clase, iniciando así la ejecución de todos los test unitarios escritos en Python.

```
1 #ifdef TESTING
2     #include "testing/mainTest.c"
3     #include "testing/pythonTest.c" // do_str function
4 #endif
5 //...
6 int main(int argc, char **argv) {
7   //...
8   #ifdef TESTING
9       // Run C tests
10      startTesting();
11      // Run Python tests
12      do_str("import testing_MainTest\r\n
13              m=testing_MainTest.MainTest()\r\n
14              m.run()\r\n\0",MP_PARSE_FILE_INPUT);
15      return 0;
16  #endif
17  //...
18 }
```

ALGORITMO 4.10: Inclusión de los archivos de test en el main.

Para compilar el firmware habilitando la ejecución de los tests, simplemente se ejecuta:

```
» make test
```

La variable de entorno MP_INCLUDE_TESTS sirve para que el script que genera código C a partir del código Python escrito como Frozen, incluya las clases de test escritas en Python, de lo contrario estas clases no se incluyen en el firmware, para minimizar el tamaño de la memoria requerida.

4.3.1. Requerimientos para la ejecución de los tests.

Los tests desarrollados no fueron pensados para probar el hardware sino el firmware, esto quiere decir que es un requerimiento necesario que los periféricos del

microcontrolador funcionen correctamente en su interfaz física (pines, señales eléctricas, etc.) Los tests unitarios que tienen relación con el hardware pueden dividirse en tres tipos[7] detallados a continuación:

- **Tests de hardware automáticos**: los tests se ejecutan de forma automática desde su inicio hasta su fin y arrojan los resultados obtenidos.
- **Tests de hardware automáticos parciales**: requieren intervencion manual durante el proceso.
- **Tests de hardware automáticos con instrumentación externa**: requieren un instrumento externo que se conecte al hardware a testear.

Los test desarrollados en este trabajo pueden catalogarse como parciales y con instrumentación externa, ya que se requiere intervención de una persona en ciertas etapas (por ejemplo para presionar un pulsador) y también se requiere la conexión de un dispositivo para recibir y reenviar las tramas RS-485 generadas en los tests.

En la tabla 4.1 se detallan las conexiones de hardware requeridas para la ejecución de los tests, las comunicaciones seriales (las dos UARTs) requieren que se reciba lo mismo que se transmite, por lo que en el caso de la UART a nivel 3.3V se soluciona conectando ambos pines entre sí, pero en el caso de la interfaz RS-485 se requiere conectar el bus a un dispositivo externo (una PC con entrada RS-485 por ejemplo) que reciba y envíe lo recibido. Para probar las GPIOs tanto en su configuración como entradas así como su configuración como salidas, el problema se resuelve conectando dos GPIOs entre sí. Se recuerda que el propósito del test no es probar el funcionamiento del hardware, por lo que con solo utilizar dos pines se pueden completar los tests requeridos. Por último se conectan las entradas analógicas a un valor de 0V.

TABLA 4.1: Conexiones de hardware requeridas para ejecutar los tests.

PIN	Conectar a
232_RX	232_TX (con una R de 100ohm en serie)
GPIO8	GPIO7 (con una R de 100ohm en serie)
CH1	GNDA
CH2	GNDA
CH3	GNDA
RS485	Terminal con eco.9600-8N1

Se realizazon 58 tests para la capa uPython HAL y 69 tests para la capa de Python que utiliza el usuario.

4.4. Tests Unitarios para ventanas que componen el IDE

Para la creación de tests unitarios que prueben las ventanas del IDE y el proceso de grabación del código Python en la placa, se utilizó la biblioteca standard de Python llamada unittest la cual posee la clase TestCase. Como el IDE se ejecuta en una PC, no se utilizó la versión de unittest desarrollada para uPython (ver sección 4.2) sino que se optó por el uso de la original.

En la figura 4.3 puede observarse el diagrama de clases que involucra a la clase TestCase como clase padre de todas las clases de test que se implementaron.A diferencia de la biblioteca unittest para uPython, aquí los nombres de los métodos deben comenzar con "test_".

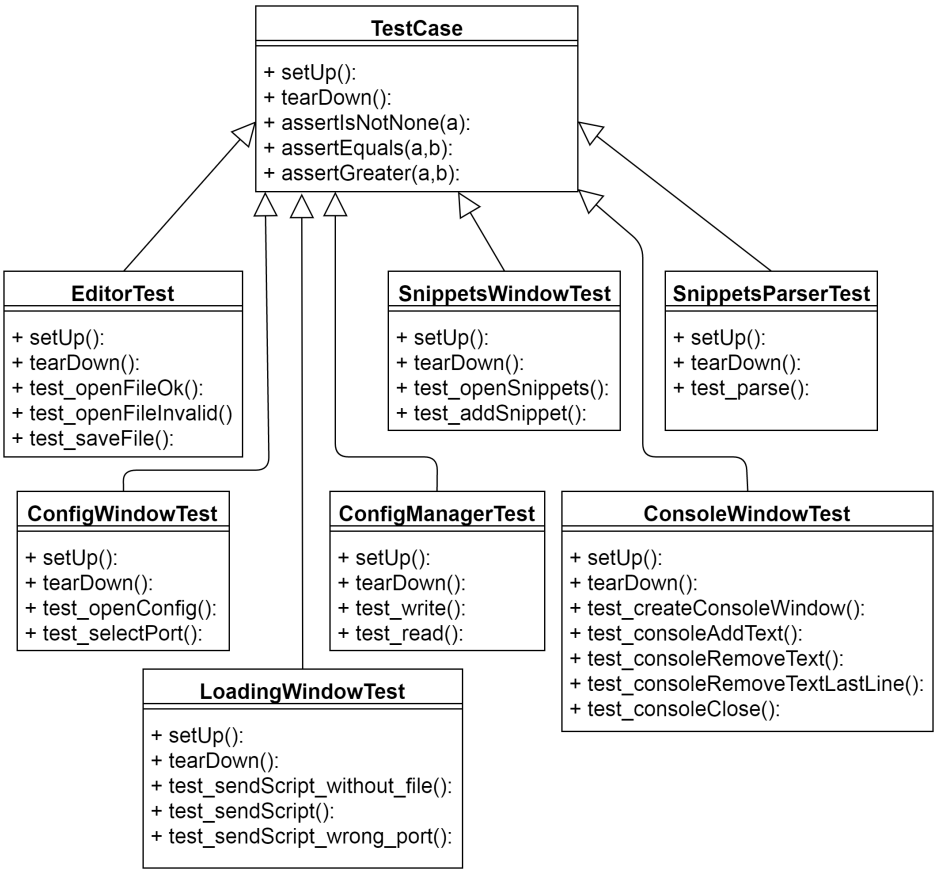

FIGURA 4.3: Diagrama de clases de tests unitarios para el IDE

A continuación se detallan los test implementados:

1. **EditorTest**: se chequean las funcionalidades básicas de la ventana del IDE (abrir y guardar un archivo, etc.).
2. **SnippetsWindowTest y SnippetsParserTest**: comprueban la correcta lectura del archivo XML con las porciones de código de ejemplo y su inserción en el código.
3. **ConfigWindowTest y ConfigManagerTest**: comprueban el correcto manejo de los datos de configuración (el nombre del puerto serial seleccionado).
4. **ConsoleWindowTest**: aquí se definen los tests para las operaciones de texto que realiza la consola, como agregar y quitar un texto de la misma.
5. **LoadingWindowTest**: comprueba el envío del archivo con el código Python.

Los tests escritos son generalmente de las ventanas, aunque también existen algunos de clases de más bajo nivel como la encargada de leer el archivo XML con los snippets, o la encargada de leer y guardar la configuración, de esta manera al ejecutar los tests se crean múltiples instancias de las ventanas del IDE y el resultado de los tests se emite por la consola.

4.5. Tests funcionales

Para complementar los test unitarios implementados, se realizaron una serie de test funcionales que permitieron comprobar el correcto funcionamiento de las clases Python que puede usar el usuario para programar código Python en la EDU-CIAA-NXP y manejar los periféricos de la misma, así como también se escribieron test funcionales para chequear el comportamiento IDE una vez integradas las ventanas que se incorporaron al mismo.

Para escribir estos tests, primero se listaron los requerimientos asignándoles un nombre con la forma R-XX, por ejemplo R-01 o R-02, luego se escribieron los tests funcionales, cada uno cubrió al menos un requerimiento, y se nombraron con la forma TF-XX por ejemplo TF-01 o TF-02. Cada test consiste en una lista de prerequisitos y una serie de pasos que el usuario debe seguir y que le indican qué hacer, y cual es el resultado esperado para cada paso. Al final del test se aclara qué requisitos cubre el test escrito.

Por último se dibujó la matriz de trazabilidad, la cual consiste en una tabla con los nombres de los requerimientos en las columnas y los nombres de los tests en las filas y en donde se indica qué test cubre qué requerimiento mediante una X.

4.5.1. Tests funcionales para el IDE

Para escribir estos tests, primero se listaron 8 requerimientos y luego se escribieron 7 tests funcionales, por ejemplo a continuación se detalla el test TF-01:

Test funcional TF-01: Ventana de Configuración del puerto serie.

Pre-condiciones:

- Tener una placa conectada a la PC
- Tener el IDE instalado

Pasos:

1. Abrir el IDE. Resultado: deberá aparecer un splash y luego la ventana del editor.
2. Seleccionar el menú “EDU-CIAA” ->Configuration. Resultado: se lanzará una nueva ventana con la configuración del puerto serie.
3. Seleccionar del combo el puerto serie correspondiente a la placa. Resultado: al desplegar el combo deberá aparecer al menos una opción.
4. Presionar OK. Resultado: la ventana deberá cerrarse.

Requerimientos testeados:R-07

4.5.2. Tests funcionales para clases Python de manejo de periféricos

Para escribir estos tests, primero se listaron 8 requerimientos y luego se escribieron 8 tests funcionales, por ejemplo a continuación se detalla el test TF-08:

Test Funcional T-08. Módulo de Python pyb.EEPROM.

Pasos:

1. Abrir el IDE.
2. Conectar la EDU-CIAA-NXP y configurar el Puerto Serie en el IDE.
3. Escribir un programa que cree un objeto EEPROM.
4. Escribir en la dirección 0x0000 el valor 0x27.
5. Ejecutar el programa para que se escriba el valor en la EEPROM.
6. Comentar la línea de la escritura.
7. Leer el valor en la dirección 0x0000 e imprimirlo en hexadecimal.
8. Volver a ejecutar el programa, debería aparecer el valor 0x27.

Código a escribir:

```
import pyb
eeprom = pyb.EEPROM()
#eeprom.write_byte(0x0000,0x27)
val = eeprom.read_byte(0x0000)
print(hex(val))
```

Requerimientos testeados:R-08

Capítulo 5

Conclusiones

En este capítulo se detallan las conclusiones del presente trabajo y los pasos a seguir.

5.1. Conclusiones generales

En el presente trabajo se ha logrado implementar un conjunto de herramientas que permiten a estudiantes de educación secundaria y universitaria aprender programación sobre sistemas embebidos. Esto fue posible gracias a la decisión de utilizar el lenguaje de programación Python y la plataforma de hardware EDU-CIAA-NXP, proveyendo al alumno de un entorno de trabajo simple y bien documentado eliminando las trabas mencionadas en el capítulo 1 que dificultan el aprendizaje en el ámbito de la informática y los sistemas embebidos.

Como prueba del éxito de este trabajo, puede mencionarse el dictado de una clase en el curso *Paquete Tecnológico del Proyecto CIAA*[1] en el marco de los Cursos Abiertos de Programación de Sistemas Embebidos, en la cual los alumnos utilizaron una versión preliminar de este trabajo y manejando los leds y pulsadores de la placa pudieron realizar un gran número de prácticas.

5.2. Próximos pasos

Como se menciona en la sección 2.2 este trabajo partió de la utilización de una capa de soporte de hardware desarrollada en 2015 la cual se reescribió y mejoró utilizando técnicas de ingeniería de software (las cuales no implementaba) para los periféricos mencionados en los requerimientos de este trabajo (sección 2.5). Existe otro grupo de periféricos para los cuales se debe hacer este mismo trabajo de refactoring e implementación de tests, los cuales se detallan a continuación:

1. Interrupciones.
2. PWM.
3. Keyboard y LCD (Poncho UI).
4. SPI e I2C (modo master).
5. RTC.

[1] URL: http://www.proyecto-ciaa.com.ar/devwiki/doku.php?id=educacion:cursos:cursos_programacion_ciaa

También existen bibliotecas programadas exclusivamente en Python como el soporte de *Modbus*[2] y operaciones con fecha y hora las cuales tampoco fueron implementadas siguiendo requerimientos ni validadas mediante ningún tipo de test. Y por último existen periféricos para los que no hay soporte y se debería comenzar de cero, como USB, CAN, Ethernet, los modos slave de los buses I2C y SPI y el core del cortex M0.

Con respecto al IDE, existe una nueva versión beta la cual posee autocompletado y una ventana con tips de ayuda, ambas características no han sido desarrolladas con requerimientos ni validadas de ninguna manera.

También existe una versión preliminar de un emulador de la placa EDU-CIAA-NXP, creado por el autor de esta trabajo, el cual soporta ser lanzado desde otro programa (por ejemplo el IDE desarrollado) y simula el script de Python programado, por el momento solo soporta el uso de leds y pulsadores, y no simula el resto de los periféricos, pero la continuación de este proyecto a una versión estable garantiza una versión del conjunto de herramientas desarrollado la cual puede correr en una PC en su totalidad, eliminando la necesidad de que exista una placa por cada alumno o grupo de alumnos y bajando de esta forma los costos requeridos para utilizar esta herramienta en ambientes educativos.

Con respecto a la ayuda y documentación, el autor de este trabajo tiene la idea de desarrollar video-tutoriales en donde se muestre la ventana del IDE mientras se programa y se explica paso por paso lo programado, con ayuda de gráficos u otras herramientas en el caso de requerirse. Estos videos también podrían estar divididos en diferentes niveles de complejidad como se hizo con el repositorio de ejemplos.

Por último existe la idea de generar una release del firmware en su versión binaria, para poder ser programada en la placa sin necesidad de instalar en la PC las herramientas de compilación, esto se logra puenteando el jumper JP5 de la placa y ejecutando programas como *lpc21isp*[3] o *FlashMagic*[4] los cuales permiten grabar el microcontrolador por medio de un puerto serial. Si bien es verdad que en el caso de una entidad educativa solo el profesor podría instalarse las herramientas de compilación y grabar las placas con el firmware para los alumnos, es una buena idea disponer de una manera simple de grabar la EDU-CIAA-NXP para que principiantes y autodidactas que han adquirido la placa puedan comenzar a programar sin trabas.

[2]Modbus:Modbus un protocolo de comunicaciones situado en el nivel 7 del Modelo OSI, basado en la arquitectura maestro/esclavo (RTU) o cliente/servidor (TCP/IP), diseñado en 1979.

[3]lpc21isp:https://sourceforge.net/projects/lpc21isp

[4]Flashmagic:http://www.flashmagictool.com

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()"

```
STATIC mp_obj_t pyb_delay(mp_obj_t ms_in) {
    mp_int_t ms = mp_obj_get_int(ms_in);
    if (ms >= 0) {
        mp_hal_milli_delay(ms);
    }
    return mp_const_none;
}
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

```
extern const struct _mp_obj_module_t pyb_module;

#define MICROPY_PORT_BUILTIN_MODULES \
    { MP_OBJ_NEW_QSTR(MP_QSTR_pyb), (mp_obj_t)&pyb_module }, \
    //...
```

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).

```
const mp_obj_type_t pyb_led_type = {
    { &mp_type_type },
    .name = MP_QSTR_LED,
    .print = pyb_led_print,
    .make_new = pyb_led_make_new,
    .locals_dict = (mp_obj_t)&pyb_led_locals_dict,
};
```

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.

```
STATIC const mp_map_elem_t pyb_led_locals_dict_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR_on), (mp_obj_t)&pyb_led_on_obj },
    { MP_OBJ_NEW_QSTR(MP_QSTR_off), (mp_obj_t)&pyb_led_off_obj },
    { MP_OBJ_NEW_QSTR(MP_QSTR_toggle), (mp_obj_t)&pyb_led_toggle_obj },
    { MP_OBJ_NEW_QSTR(MP_QSTR_intensity), (mp_obj_t)&
    pyb_led_intensity_obj },
    { MP_OBJ_NEW_QSTR(MP_QSTR_value), (mp_obj_t)&pyb_led_value_obj },
};
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 algortimo 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".

```
mp_obj_t pyb_led_on(mp_obj_t self_in) {
    //...
}
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 item 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.

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] Ernesto Gigliotti. *Documentación de bibliotecas Python para la EDU-CIAA-NXP.* Disponible: 2016-10-20. URL: http://www.proyecto-ciaa.com.ar/devwiki/doku.php?id=desarrollo:edu-ciaa:edu-ciaa-nxp:python:bibliotecas.

[4] Ernesto Gigliotti. *Proyectos de ejemplo utilizando Micropython en la EDU-CIAA-NXP.* Disponible: 2016-10-20. URL: https://github.com/ernesto-g/educiaa-micropython-demos.

[5] 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.

[6] Glade. *A User Interface Designer.* Disponible: 2016-10-20. URL: https://glade.gnome.org.

[7] James W. Grenning. «Test-Driven Development for embedded C». En: (2011), págs. 116-119.

[8] 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.

[9] Wayne Wolf. Fellow. IEEE. y Jan Madsen. «Embedded Systems Education for the Future.» En: *PROCEEDINGS OF THE IEEE, VOL. 88, NO. 1* (2000). ISSN: 1558-2256. URL: http://home.iitj.ac.in/~saurabh.heda/Papers/Scheduling/Embedded%20system%20design%20for%20future%20-1999.pdf.

[10] Jaltek Systems. *Jaltek Systems Web Page*. Disponible: 2016-10-20. URL: http://www.jaltek.com.

[11] 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.

[12] Martin Ribelotta. *Port de MicroPython para la EDU-CIAA-NXP*. Disponible: 2016-10-20. URL: https://github.com/martinribelotta/micropython.
[13] MicroPython. *MicroPython - Python for microcontrollers*. Disponible: 2016-10-20. URL: https://micropython.org.