UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

MAESTRÍA EN SISTEMAS EMBEBIDOS

MEMORIA DEL TRABAJO FINAL

sAPI (simpleAPI): diseño e implementación de una biblioteca para sistematizar la programación de sistemas embebidos

Autor:
Esp. Ing. Eric Nicolás Pernia

Director:
MSc. Ing. Félix Gustavo Emilio Safar (UNQ)

Jurados:
Dr. Ing. Pablo Martín Gómez (UBA)
Mg. Ing. Pablo Oscar Ridolfi (UTN FRBA)
Ing. Juan Manuel Cruz (FIUBA,UTN-FRBA)

Este trabajo fue realizado en las Ciudad Autónoma de Buenos Aires, entre marzo de 2018 y diciembre de 2018.

Resumen

En esta memoria se presenta el diseño de una biblioteca para la programación de sistemas embebidos portable entre plataformas de hardware y lenguajes de programación. Se realizó una implementación de referencia en lenguaje C para las plataformas del Proyecto CIAA. Se realizó un banco de pruebas de hardware, junto a la utilización de testeo unitario e integración contínua para la validación.

Además de la biblioteca, se creó una herramienta de código abierto para desarrolladores que automatiza la implemetación de bibliotecas a partir de un modelo que las describe. De esta manera se facilita la futura ampliación de la biblioteca y su implementación en otras plataformas de hardware.

Agradecimientos

Agradecimientos personales. **[OPCIONAL]**

No olvidarse de agradecer al tutor.

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

Índice general

Índice de figuras

Índice de Tablas

Dedicado a... [OPCIONAL]

Capítulo 1

Introducción General

En este capítulo se presenta el contexto y justificación de este trabajo final, la motivación para llevarlo a cabo, sus objetivos y alcance.

1.1. Contexto y justificación

En la actualidad existe una enorme variedad de plataformas de sistemas embebidos en el mercado, y si bien todas cuentan con dispositivos programables con características similares y periféricos compatibles, se observa que a la hora de programarlas todas son distintas. En consecuencia, el profesional desarrollador de sistemas embebidos debe invertir mucho tiempo en aprender a programar cada nueva plataforma.

Estas diferencias se deben a que los fabricantes de los dispositivos y empresas asociadas poseen sus respectivas arquitecturas de hardware (tanto en núcleos de procesamiento, como en periféricos). Cada periférico exhibe externamente la funcionalidad esperada, pero su configuración y uso es diferente. Si bien esto da a los fabricantes la posibilidad de diferenciarse frente a la competencia y a los desarrolladores elegir el dispositivo programable que más se adecúe a un proyecto, también causa que el programador deba conocer en detalle la arquitectura en particular del dispositivo a la hora de utilizarlo.

Además, estas empresas en su mayoría se limitan a ofrecer información de bajo nivel para programar el hardware directamente (escribiendo registros), o bien, mediante sus propias bibliotecas escritas generalmente en lenguaje C, que están diseñadas con una gran dependencia de la arquitectura de hardware subyacente, es decir, carecen de abstracción del hardware.

Por otro lado, se están portando cada vez más lenguajes de programación a las plataformas de sistemas embebidos, que antes se reservaban para las computadoras de propósito general. Esto se debe a que las nuevas plataformas poseen cada vez más poder de procesamiento y memoria, a causa de la complejidad creciente de las aplicaciones deseadas. En consecuencia se utilizan lenguajes de programación más modernos, en parte por su mayor capacidad de abstracción, y en parte para estandarizar el lenguaje de programación de una aplicación en la que intervienen sistemas embebidos y computadoras de propósito general, siendo un ejemplo típico una aplicación de IoT[1].

[1]Del inglés *Internet of Things*, es decir, Internet de las cosas, donde intervienen en una aplicación sistemas embebidos distribuidos, infraestructura de red y servidores.

Existen múltiples desarrollos de bibliotecas que logran una abstracción de hardware aceptable. Sin embargo, ninguna de estas se ha adoptado como estándar de facto en la industria y el alcance de las mismas se limita a cubrir plataformas de cierto ecosistema en el marco de una empresa o comunidad y soportan un único lenguaje de programación (en general C o C++).

Este trabajo final intenta contribuir a la resolución de esta problemática mediante la sistematización de la implementación de bibliotecas para sistemas embebidos. Para ello se presenta el diseño de una biblioteca para la programación de sistemas embebidos modelada independientemente del lenguaje de programación y arquitectura del hardware. Asimismo se realizaron herramientas de código abierto para desarrolladores que automatizan gran parte de la implemetación de la biblioteca en una nueva plataforma en base a una descripción de la misma, así como la definición de nuevos módulos de bibliotecas. De esta manera se facilita la futura ampliación de la biblioteca y su implementación en diferentes plataformas de hardware.

Utilizando estas herramientas se realizó una implementación de referencia en lenguaje C para las plataformas del Proyecto CIAA[2] []. Para su validación, se realizó un banco de pruebas de hardware, junto a la utilización de testeo unitario e integración continua.

1.2. Motivación

Este trabajo es parte de un grupo de iniciativas realizadas por el autor (docente e investigador en la Universidad Nacional de Quilmes), para colaborar en el marco del Proyecto CIAA [], con el objetivo de facilitar el desarrollo y la enseñanza de Sistemas Embebidos en Argentina.

El proyecto CIAA nació en 2013 como una iniciativa conjunta entre el sector académico y el industrial, representados por la ACSE [] y CADIEEL [] respectivamente. Consiste en una comunidad argentina de desarrolladores de herramientas de software y hardware abierto y usuarios que desde sus inicios intenta impulsar el desarrollo tecnológico nacional proveyendo herramientas abiertas de software y hardware, para mejorar la situación industrial, darle visibilidad a la electrónica y generar cambios estructurales en la forma en que se generan, comparten y utilizan los conocimientos en Argentina.

Entre los principales logros del proyecto se destacan:

- Se creó una comunidad de desarrolladores de software y hardware abierto con participación de profesionales, docentes Universitarios, alumnos y docentes de escuelas Secundarias a nivel nacional.
- Se desarrollaron múltiples diseños de referencia de plataformas de hardware para sistemas embebidos. Estas plataformas fueron diseñadas para diferentes casos de uso, entre ellos: educativo, industrial, sistema críticos y alta capacidad de cómputo.
- Se logró la colaboración de empresas nacionales para la fabricación y comercialización de la mayoría de estas plataformas.

[2]Siglas de Computadora Industrial Abierta Argentina.

- Se crearon bibliotecas, *frameworks* y entornos de programación para las plataformas, contando con múltiples lenguajes de programación.
- En colaboración con universidades, se insertó la plataforma educativa EDU-CIAA-NXP en las universidades con carreras de electrónica o afines, a lo largo y a lo ancho del país. De esta forma se logró modernizar las herramientas que utilizan los alumnos en su aprendizaje, alcanzando el estado del arte.
- Se han dictado múltiples cursos para docentes y alumnos de escuelas secundarias, docentes y alumnos universitarios y público interesado.

El autor de este trabajo final cumple actualmente el rol de coordinador general del proyecto CIAA. Tanto en el trabajo realizado para la implementación de bibliotecas para diferentes plataformas en el marco de este proyecto, como en el trabajo profesional, ha notado problemática de falta de estadarización en las bibliotecas de sistemas embebidos detallada en la sección 1.1.

Se considera de importancia estratégica para el proyecto CIAA que todas las plataformas de hardware diseñadas en el marco del mismo se programen utilizando la misma biblioteca para permitir a la comunidad de usuarios reducir tiempos de aprendizaje y desarrollo.

Como el lenguaje más utilizado en la actualidad para programación de sistemas embebidos basados en microcontroladores continúa siendo el lenguaje C, se decide realizar la implementación de la biblioteca en este lenguaje.

1.3. Objetivos y alcance

En esta sección se definen los objetivos y el alcance del presente trabajo final.

1.3.1. Objetivos

El objetivo de este proyecto es diseñar e implementar una biblioteca de software para la programación de sistemas embebidos basados en microcontroladores con las siguientes características:

- Estar modelada independientemente de los lenguajes de programación.
- Definir una interfaz de programación de aplicaciones (API) sencilla que abstraiga los modos de uso más comunes de los periféricos típicos que hallados en cualquier microcontrolador del mercado.
- Ser totalmente portable entre diferentes arquitecturas de hardware sobre dónde se ejecuta, manteniendo una API uniforme a lo largo de las mismas, cumpliendo la función de capa de abstracción de hardware.
- Deve ser útil tanto para enseñanza de programación de sistemas embebidos como para uso a nivel industrial.

La implementación de referencia se realizará en lenguaje C para las plataformas del Proyecto CIAA.

1.3.2. Alcance

Este Trabajo Final incluye realización de:

- Diseño de la biblioteca, archivos de descripción de la biblioteca mediante diferentes diagramas y código independiente del lenguaje de programación.
- Implementación en lenguaje C de la biblioteca para las plataformas de hardware:
  - CIAA-NXP.
  - EDU-CIAA-NXP.
  - CIAA-Z3R0.
  - PicoCIAA.
- Manual de instalación de las herramientas para utilizar la biblioteca con las plataformas de hardware citadas.
- Manual de referencia de la biblioteca.
- Ejemplos de utilización.

1.4. Metodología de trabajo

Para el desarrollo de este trabajo se elige utilizar las siguientes prácticas y herramientas:

- Sistema de control de versiones: utilización de repositorios *Git* [] en el sitio *Github* [].
- *Workflow* de desarrollo mediante *Fork* y *Pull Requests* de github. Se debe abrir un *issue* en *Github* por cada característica a diseñar/mejorar.
- Producción de código y su documentación: escribir la documentación a la par del código fuente.
- Definir un documento de estilo del código fuente.
- Documentación general en lenguaje *Markdown* [], que permite exportar el contenido a *html*, LaTeX, *pdf*, entre otros.
- Desarrollo de *tests* unitarios y de integración.

Capítulo 2

Introducción Específica

En este capítulo se explican los detalles para comprender las decisiones de diseño adoptadas. Se describen los trabajos previos que se tomaron como fundamento para el diseño de la biblioteca, se presentan las plataformas sobre las cuales se elige realizar la implementación de la misma y los requerimientos de este trabajo, junto a la planificación para su cumplimiento.

2.1. Antecedentes

Los primeros pasos para la idea de la realización de este trabajo surgen del trabajo final de la Carrera de Especialización en Sistemas Embebidos de la Universidad de Buenos Aires (CESE FIUBA) [], realizado por el autor, titulado "Desarrollo de Firmware y Software para programar la CIAA en lenguaje JAVA con aplicación en entornos Industriales"[] presentado en diciembre de 2015, donde como parte del trabajo se implementó la programación de plataformas de sistemas embebidos en lenguaje Java [].

En ese trabajo se realizó, entre otras cosas, la implementación de una biblioteca para acceder a los periféricos de un microcontrolador en lenguaje Java. Para llevarlo a cabo se realizó la definición de clases que modelan los periféricos GPIO, ADC y UART y se implementó el acceso al hardware como métodos nativos de Java, escritos en lenguaje C. Por este motivo se debió realizar una implementación de la biblioteca también lenguaje C. Se implementó para las plataformas del proyecto CIAA EDU-CIAA-NXP [] y CIAA-NXP []. Esta biblioteca se nombró sAPI, siglas de *simple API* en alusión a que provee una API sencilla para la programación de microcontroladores.

A partir mazo de 2016 se decide utilizar la biblioteca como base para la enseñanza de la programación de periféricos de microcontroladores. A lo largo de ese año se extendió la biblioteca de C para la plataforma EDU-CIAA-NXP de forma considerable para explicar la utilización los periféricos típicos de microcontroladores, logrando excelentes resultados en aprendizaje por parte de los alumnos tanto de niveles avanzados como quienes dan sus primeros pasos en el aprendizaje de programación de microcontroladores.

Además, la biblioteca sAPI en lenguaje C para la EDU-CIAA-NXP se puso a disposición de cualquier persona ya que se encuentra publicada de forma libre y gratuita por internet bajo una licencia BSD modificada [] en el sitio de github del autor [].

Finalmente, en diciembre de 2016 se decide utilizar la biblioteca sAPI realizada en lenguaje C como biblioteca estándar para las plataformas del proyecto CIAA, distribuyéndola como parte del *framework*[1] "Firmware v2" []. Este *framework* combinó la biblioteca sAPI con *framework* "Workspace"[], desarrollado por Pablo Ridolfi.

Esta plataforma ha sido adoptada por una gran cantidad de usuarios. Una prueba de esto es la encuesta realizada en octubre de 2018, titulada "Tecnologías usadas en los Trabajos Finales del Posgrado en Sistemas Embebidos: 2015-2018"[] donde en la sección 27, *"Uso de material del Proyecto CIAA en los trabajos finales"* se observa que alrededor de la mitad de los trabajos finales de la CESE/MSE utilizaron material generado en el marco del Proyecto CIAA. Y en particular, a partir de 2017 se produjo un cambio de tendencia y más del 60 % de los trabajos finales utilizaron material del Proyecto CIAA. Este cambio en la tendencia se observa que comienza en 2016 con la publicación de firmware v2 como se muestra en la gráfica de la figura 2.1, extraída de dicho artículo.

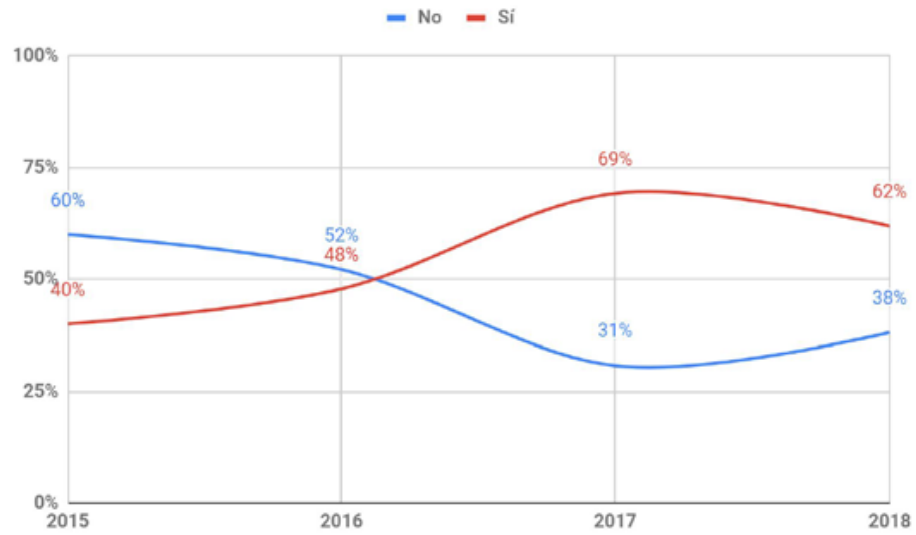

FIGURA 2.1: Evolución en el uso de material del Proyecto CIAA.

A principios de 2017, el autor llevó a cabo una profunda revisión y mejora de la biblioteca de C con el objetivo de extenderla a las demás plataformas del proyecto CIAA. Ese trabajo fue compilado en un artículo y publicado en el Congreso Argentino de Sistemas Embebidos (CASE) [] en agosto de 2017. Es un antecedente fundamental para la realización del presente trabajo final porque se realizó un estudio exhaustivo de las bibliotecas existentes en el mercado dando como resultado los siguientes puntos a considerar en el diseño de una biblioteca de C para la programación de sistemas embebidos:

- Extensión de la definición de la API.
- Dependencia del hardware.
- Nivel de abstracción.

[1] En el desarrollo de software, un *framework* es una estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que sirve de base para la organización y desarrollo de software []. En este caso concreto se compone de bibliotecas de código C y makefiles para su compilación permitiendo organizar proyectos de software en lenguaje C.

- Complejidad de aprendizaje y uso.
- Periféricos y modos soportados.
- Escalabilidad.

Parte de este rediseño se aplicó a la biblioteca sAPI en lenguaje C para la EDU-CIAA-NXP.

En 2017 el autor desarrolló una plataforma más económica que la EDU-CIAA-NXP, nombrada "CIAA-Z3R0" [], que la empresa Asembli [] sacó al mercado en noviembre. En diciembre el autor publicó la primera versión de la biblioteca sAPI en lenguaje C para esta plataforma dándole soporte a algunos periféricos de la misma.

Tomando la experiencia adquirida a lo largo de estos años, el autor presenta en esta memoria de trabajo final un diseño mejorado para la biblioteca, ampliando el mismo e independizándolo del lenguaje de programación. Asimismo, teniendo en cuenta las tareas repetitivas que requieren la implementación de una biblioteca para diferentes plataformas de sistemas embebidos se decide desarrollar herramientas para automatizar el proceso donde resulta posible.

2.2. Plataformas del Proyecto CIAA

Todos los desarrollos de hardware realizados en el marco del proyecto CIAA han sido publicados con licencia BSD de tres cláusulas con el espíritu de que sean utilizadas como base tanto para diseños abiertos como para diseños cerrados. Los mismos pueden descargarse del repositorio oficial del proyecto CIAA nombrado "CIAA Hardware" [].

Las plataformas sobre las cuales se decide realizar el presente trabajo son aquellas que han logrado pasar de la fase de prototipo y se pueden hallar como producto en el mercado. Cabe destacar que el proyecto CIAA no se beneficia económicamente de la venta de plataformas.

2.2.1. CIAA-NXP

La plataforma CIAA-NXP (figura 2.2) fue la primer paltaforma desarrollada en el marco del proyecto CIAA. Consiste en una computadora para uso industrial basada en el microcontrolador NXP LPC4337 JDB144 [] *dual-core* asimétrico, formado por un procesador Cortex-M4F y un Cortex-M0 (ambos de 32 bits), que corren con una frecuencia de sistema máxima de 204MHz; con 1 MB de memoria Flash y 136 KB de memoria SRAM.

Sus características destacables son:

- Interfaces de entrada/salida: 8 entradas digitales (opto-aisladas 24VDC), 4 entradas analógicas (0-10V/4-20mA), 4 salidas digitales Open-Drain (24VDC), 4 salidas digitales a relé DPDT y 1 salida analógica (0-10V/4-20mA).
- Interfaces de comunicación: 1 Ethernet, 2 USB On-The-Go, 1 RS232, 1 RS485, 1 CAN, 1 SPI, 1 I2C.

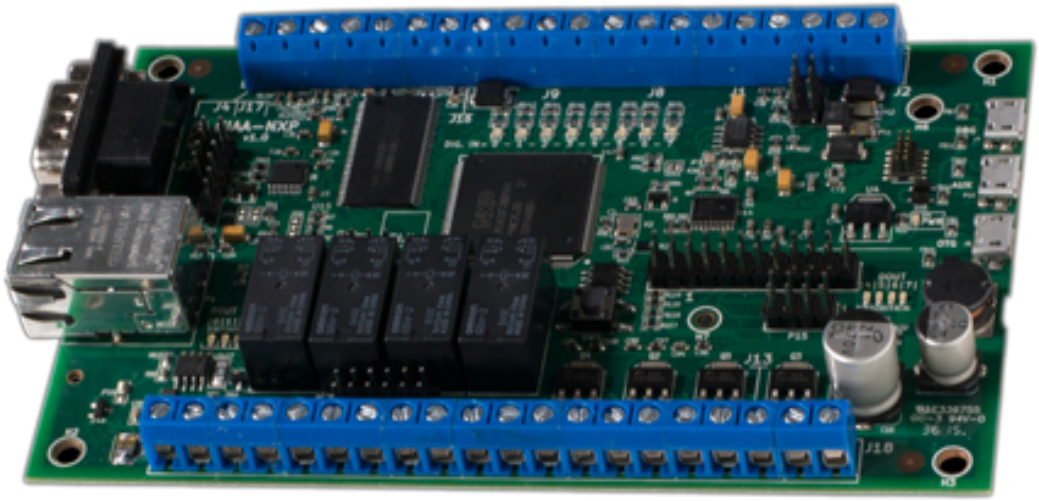

FIGURA 2.2: Plataforma CIAA-NXP.

- Uso de Linux: Posee memorias RAM y Flash externas que posibilitan ejecutar Linux sobre esta plataforma. Exite un *port* de Linux para la misma que puede hallarse en [].
- Incluye *debugger*: Esta plataforma incluye el circuito que permite depuración en tiempo real del programa que corre en la plataforma desde la PC. Se basa en el chip FTDI FT2232H [].

2.2.2. EDU-CIAA-NXP

En base al diseño de la CIAA-NXP los integrantes del proyecto CIAA realizan una versión educativa sin las interfaces y protecciones industriales, nombrada EDU-CIAA-NXP (figura 2.3).

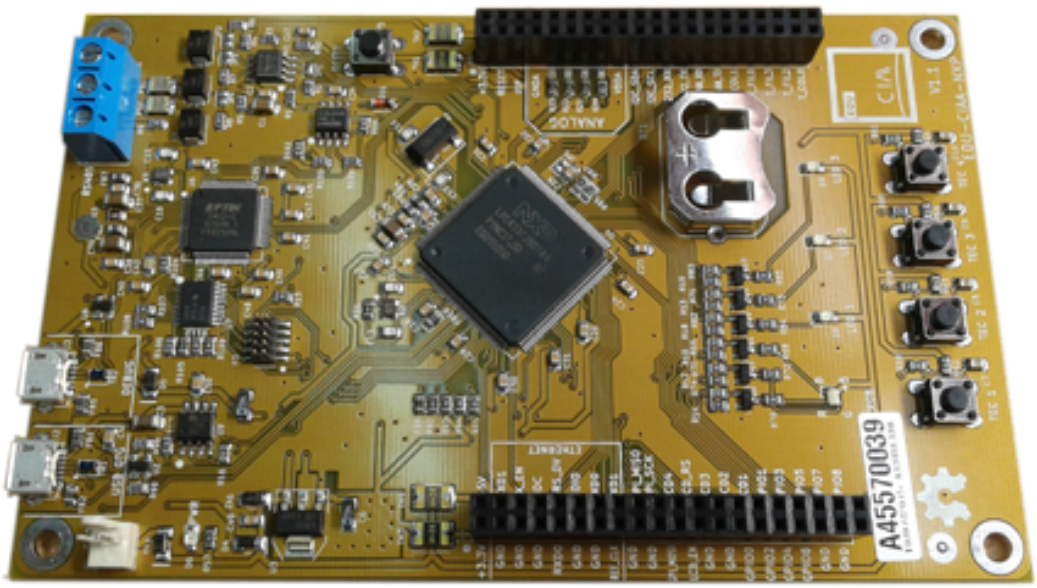

FIGURA 2.3: Plataforma EDU-CIAA-NXP.

Utiliza el mismo microcontrolador que la CIAA-NXP, circuito de depuración, interfaz RS-485 y posee 1 USB OTG.

Los pines no utilizados por las interfaces anteriores se disponen en los conectores P1 y P2. En ellos incluye todos los periféricos típicos que podemos encontrar en los microcontroladores disponibles en el mercado (GPIO, ADC, DAC, TIMER, UART, SPI, I2C, etc.). Además posee 1 LED RGB, 3 LEDs y 4 pulsadores.

2.2.3. PicoCIAA

La PicoCIAA [] (figura 2.4) es una placa en formato mini PCI Express, pensada para ser utilizada como un módulo de cómputo y/o adquisición en una plataforma mayor.

Se basa en el microcontrolador NXP LPC54102J512BD64 [], otro microcontrolador *dual-core* asimétrico formado por un procesador Cortex-M4F y un Cortex-M0+ (ambos de 32 bits), que corren con una frecuencia de sistema máxima de 100 MHz; con 512 KB de memoria Flash y 104 KB de SRAM .

FIGURA 2.4: Plataforma PicoCIAA.

Incluye circuito de depuración USB vía un microcontrolador NXP LPC11U35 [] programable.

Posee diversos puertos de comunicación, entre ellos USB, mini PCI Express, UART, SPI, I2C y soporte para PWM y entradas y salidas digitales de propósito general. Su tamaño reducido (51 x 30 mm) es ideal en Single Board Computers.

2.2.4. CIAA-Z3R0

La CIAA-Z3R0 se diseñó para ser la plataforma más económica del proyecto CIAA. Esta plataforma es ideal para aplicaciones de bajo consumo (como sensores de IoT) y proyectos de robótica educativa. Está diseñada para ser utilizada como componente en un diseño mayor debido a su tamaño (19.8 x 51.8 mm), soldada a través de su borde de agujeros para montaje *castellated*[], o bien, conectándola mediante tiras de pines.

Posee la mayoría de los periféricos que se encuentran en la EDU-CIAA-NXP pero utiliza un microcontrolador siete veces más económico que esta última, de la empresa Silicon Labs, modelo EFM32HG322F64 (QFP48) [] con núcleo ARM Cortex-M0+ a una frecuencia máxima de 25 MHz; 64 KB de memoria Flash y 8 KB de memoria SRAM, que es suficiente para que un alumno entre en el mundo de los microcontroladores modernos de 32 bits.

El *debugger* se debe comprar por separado, sin embargo, mediante un único dispositivo de depuración se pueden programar muchas plataformas CIAA-Z3R0.

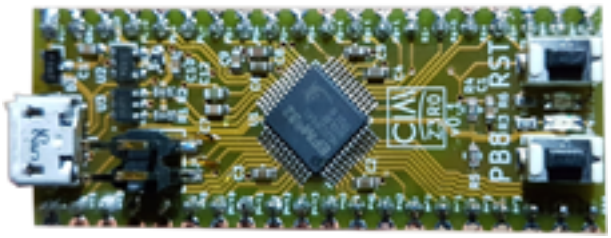

FIGURA 2.5: Plataforma CIAA-Z3R0.

2.2.5. Material provisto por los fabricantes

De las secciones anteriores, se advierte que las cuatro plataformas de hardware presentadas poseen núcleos de procesamiento diseñados por la empresa ARM [] los cuales son licenciados a diversos fabricantes.

ARM ofrece manuales acerca de sus núcleos de procesamiento y diversas bibliotecas para su programación.

Además, tres de estas paltaformas poseen microcontroladores fabricados por la empresa NXP (CIAA-NXP, EDU-CIAA-NXP con LPC4337 y PicoCIAA con LPC5410) y una por Silicon Labs (EFM32HG).

Tanto NXP como Silicon Labs ofrecen hojas de datos, manuales del sistema y notas de aplicación y bibliotecas para sus microcontroladores.

Se exponen a continuación las principales bibliotecas provistas por las empresas ARM, NXP y Silicon Labs para estos microcontroladores.

CMSIS de ARM

CMSIS son las siglas de *"Cortex Microcontroller Software Interface Standard"*, es decir, interfaz de software para microcontroladores Cortex. Mediante este conjunto de bibliotecas, la empresa ARM intenta estandarizar cómo se programan los microcontroladores que ofrecen las empresas proveedores de silicio licenciantes de sus núcleos de procesamiento Cortex-M. CMSIS se desarrolla públicamente en GitHub.

Las principales bibliotecas que posee están realizadas en su mayoría lenguaje C y son:

- CMSIS-CORE: define el arranque del sistema y acceso periféricos.
- CMSIS-RTOS: es una API de abtracción del RTOS[2] que permite capas de software coherentes con componentes de *middleware*[3] y bibliotecas de bajo nivel.
- CMSIS-DSP: es una colección de funciones de procesamiento de señales digitales, optimizada para núcleos de procesamiento Cortex-M.
- CMSIS-Driver: interfaces genéricas de periféricos para *middleware* y código de aplicación.

[2]Siglas en inglés de Sistema Operativo de Tiempo Real.

[3]*Middleware* son bibliotecas que asisten a una aplicación, por ejemplo, biblioteca para manejo de sistema de archivos y *stracks* de protocolos.

Además provee las siguientes herramientas:

- CMSIS-Pack: define la estructura de un paquete de software que contiene componentes de software. Los componentes del software son fácilmente seleccionables, y se resaltan las dependencias de otros paquetes.
- CMSIS-SVD: son archivos que habilitan vistas detalladas a los periféricos del dispositivo, que muestran el estado actual de cada registro, y aseguran que la vista del depurador coincida con la implementación real de los periféricos del dispositivo.
- CMSIS-DAP: una interfaz estandarizada para el puerto de acceso de depuración de Cortex (DAP) y es utilizada por muchos kits de de desarrollo, siendo compatible con varios dispositivos de hardware para depuración.
- CMSIS-NN: es una colección de núcleos de redes neuronales eficientes desarrollada para maximizar el rendimiento y minimizar la huella de memoria de las redes neuronales para núcleos Cortex-M.

Para los tres microcontroladores de este trabajo existe soporte completo para la capa Core, incluyendo el núcleo de procesamiento y los drivers de periféricos.

Mbed de ARM

Mbed es una plataforma de drivers y sistema operativo para prototipado rápido, enfocada en dispositivos IoT basados en mcirocontroladores ARM. Es un proyecto de código abierto, también disponible en github [], desarrollado colaborativamente por ARM y sus socios tecnológicos.

Provee abstracción del hardware pero se limita a plataformas del ecosistema ARM. A diferencia de CMSIS, en este caso se provee soporte para plataformas de hardware completas en lugar de solamente ocuparse del microcontrolador. De esta forma muchas empresas añaden soporte a mbed a sus kit de desarrollo con microcontroladores ARM Cortex-M.

Existen kits de desarrollo que contienen los microcontroladores LPC4337 y EFM32 HG322 con soporte de mbed, los cuales son LPCXpresso4337 [] y EFM32 USB-enabled Happy Gecko [] respectivamente. Para el LPC54102 no existe un kit de desarrollo soportado por mbed, pero existe el kit LPCXpresso54114 [] que posee un microcontrolador de la misma familia.

Las plataformas soportadas por Mbed se programan mediante un entorno de desarrollo on line. Las diferentes bibliotecas para microcontroladores provistas en el marco de mbed están escritas en su mayoría en lenguaje C++.

LPCOpen y MCUXpresso de NXP

NXP provee para su línea de microcontroladores LPC las bibliotecas LPCOpen [], que incluyen drivers para sus microcontroladores, bibliotecas *middelware* de terceros y programas de ejemplo.

Se puede descargar de forma gratuita de la web de NXP sin registrarse.

LPCOpen se compone de:

- Biblioteca de drivers, que se divide en dos capas: una capa de drivers de *chip* que contiene controladores optimizados para un dispositivo o familia específica, y una capa *board* que contiene funciones específicas de un dado kit de desarrollo.
- *Middleware*. Esta capa incluye: la biblioteca de objetos gráficos emWin, biblioteca de gráficos SWIM, *stack* de redes de código abierto LWIP y bibliotecas USB (*device* y *host*).
- Uso de LPCOpen junto a un RTOS: Incluye ejemplos para utilizar LPCOpen con FreeRTOS.
- Ejemplos: incluye un extenso conjunto de ejemplos diseñados para ilustrar cómo usar las funciones de la biblioteca del controlador central y el *middleware*.

Para la incialización del sistema utiliza la parte de CMSIS Core que inicializa cada uno de sus núcleos de procesamiento.

Si bien contiene ejemplos para comenzar a utilizar los microcontroladores LPC, sus drivers de periféricos están relacionados directamente con su arquitectura de hardware y proveen una muy baja abstracción de la misma. Además, tiene código duplicado de las bibliotecas de periféricos para cada núcleo de procesamiento de un mismo microcontrolador restándole mantenibilidad.

MCUXpresso de NXP es la evolución de LPCOpen luego de que NXP adquiera a Freescale, agregando soporte a las líneas de microcontroladores Kinetis [] e i.MX RT []. Requiere registrarse para tener acceso a la misma. Posee soporte únicamente para el microcontrolador LPC54102 de este trabajo.

Bibliotecas de Silicon Labs

Para el mcirocontrolador EFM32HG322 Silicon Labs ofrece las siguientes bibliotecas []:

- Drivers CMSIS-CORE para EFM32 Happy Gecko.
- Biblioteca de periféricos EMLIB, que provee soporte de bajo nivel para periféricos proporcionando una API unificada para todos los MCU y SoC EFM32, EZR32 y EFR32 de Silicon Labs.
- Biblioteca EMDRV, conjunto de drivers de alto rendimiento energético específicos para periféricos en chip EFM32, EZR32 y EFR32. Los controladores suelen estar basados en DMA y utilizan todas las funciones disponibles de bajo consumo. La API ofrece funciones síncronas y asíncronas para la mayoría de estos drivers. Además son totalmente reentrantes y basadas en callbacks.
- *Platform Middleware*: se compone de una biblioteca de sensado capacitivo (CSLIB), una biblioteca gráfica (GLIB, *stack* USB *device* para dispositivos Gecko y biblioteca de interfaz USBXpress.
- *Board Support Package*: El BSP proporciona una API para controladores de una cierta plataforma, incluyendo control de E/S para botones, LED y funcionalidades de *trace* los kits de desarrollo de EFM32, EZR32 y EFR32.

- Drivers para componentes de los kits de desarrollo: incluye pantallas, sensores y memorias.
- Programas de ejemplo.

2.3. Requerimientos

Los requerimientos se establecieron en base a los objetivos expuestos en la sección 1.3, reuniones con desarrolladores del Proyecto CIAA y el director del presente trabajo. Además se tuvieron en cuenta las opiniones de alumnos de grado de UNQ, posgrado de FIUBA y cursos CAPSE. Estos son:

1. Fecha de finalización: 19/11/2018.
2. Diseño de la biblioteca.
   a) Realizar un diseño independiente del hardware y lenguaje de programación, teniendo en cuenta los conceptos familiares al programador de Sistemas Embebidos.
   b) Debe estar modelada con objetos y contar con una descripción mediante diagramas UML.
   c) Debe modelar al menos: CORE, GPIO, ADC, DAC, TIMER, RTC, UART, SPI e I2C.
3. Implementación de la biblioteca.
   a) Utilizar un sistema de control de versiones con repositorios on line.
   b) Programar en lenguaje C la biblioteca para cada plataforma de hardware particular utilizando como plantilla los archivos generados.
   c) Desarrollar ejemplos de utilización para las diferentes plataformas.
4. Documentación y difusión.
   a) Confeccionar un manual de referencia de la API de la biblioteca.
   b) Desarrollar un tutorial de instalación de las herramientas para utilizar la biblioteca con las plataformas de hardware citadas.
   c) Publicación on line del código fuente.
   d) Informar a la comunidad del Proyecto CIAA y a la comunidad de programadores de Sistemas Embebidos.

2.4. Planificación

En la tabla 2.1 se resume la planificación del trabajo. En la misma se pueden observar cada una de las tareas planificadas, junto a su duración, fecha de inicio y fecha de finalización estimadas.

EDT	Nombre	Duración	Inicio	Fin
1	**Investigación preliminar.**	48horas	**03/05/2018**	**17/05/2018**
1.1	Investigar las bibliotecas para microcontroladores disponibles.	24horas	03/05/2018	10/05/2018
1.2	Investigar las bibliotecas de C y documentación de las plataformas de hardware.	24horas	10/05/2018	17/05/2018
2	**Diseño de la biblioteca independiente del hardware y lenguaje de programación.**	35.5días	**17/05/2018**	**16/08/2018**
2.1	**Modelar las propiedades y métodos de los periféricos:**	22días	**17/05/2018**	**12/07/2018**
2.1.1	GPIO.	16horas	17/05/2018	22/05/2018
2.1.2	ADC.	24horas	22/05/2018	29/05/2018
2.1.3	DAC.	22horas	31/05/2018	05/06/2018
2.1.4	TIMER.	40horas	07/06/2018	19/06/2018
2.1.5	RTC.	6horas	19/06/2018	21/06/2018
2.1.6	UART.	16horas	21/06/2018	26/06/2018
2.1.7	SPI.	24horas	26/06/2018	03/07/2018
2.1.8	I2C.	28horas	03/07/2018	12/07/2018
2.2	Modelar las propiedades y métodos del núcleo de procesamiento (CORE).	28horas	12/07/2018	21/07/2018
2.3	Modelar SoC.	16horas	21/07/2018	26/07/2018
2.4	Modelar Board.	16horas	26/07/2018	31/07/2018
2.5	Realización de diagramas UML.	12horas	31/07/2018	04/08/2018
2.6	Verificación del modelo.	8horas	04/08/2018	07/08/2018
2.7	Diseñar un archivo de descripción de la biblioteca.	24horas	07/08/2018	14/08/2018
2.8	Verificación del archivo de descripción.	4horas	14/08/2018	16/08/2018
3	**Implementación de la biblioteca.**	20.25días	**16/08/2018**	**06/10/2018**
3.1	**Implementar el archivo de descripción para las plataformas de hardware:**	2días	**16/08/2018**	**21/08/2018**
3.1.1	CIAA-NXP	4horas	16/08/2018	16/08/2018
3.1.2	EDU-CIAA-NXP	4horas	16/08/2018	18/08/2018
3.1.3	CIAA-Z3R0	4horas	18/08/2018	21/08/2018
3.1.4	PicoCIAA	4horas	21/08/2018	21/08/2018
3.2	Realizar un generador de archivos ".c" y ".h" en base a los archivos de descripción.	24horas	21/08/2018	28/08/2018
3.3	**Programar en lenguaje C la biblioteca para cada plataforma de hardware:**	14.25días	**28/08/2018**	**04/10/2018**
3.3.1	CIAA-NXP y EDU-CIAA-NXP.	40horas	28/08/2018	11/09/2018
3.3.2	CIAA-Z3R0.	36horas	11/09/2018	20/09/2018
3.3.3	PicoCIAA.	38horas	20/09/2018	04/10/2018
3.4	Verificación del funcionamiento de cada periférico en cada una de las plataformas.	8horas	04/10/2018	06/10/2018
4	**Validación.**	3.75días	**06/10/2018**	**16/10/2018**
4.1	Realización de ejemplos funcionales para cada perifércio.	30horas	06/10/2018	16/10/2018
5	**Procesos finales.**	10días	**16/10/2018**	**10/11/2018**
5.1	Redacción de memoria de trabajo final.	40horas	16/10/2018	30/10/2018
5.2	Confeccionar un manual de referencia de la API de la biblioteca.	12horas	30/10/2018	01/11/2018
5.3	Desarrollar un tutorial de instalación de las herramientas para utilizar la biblioteca.	16horas	01/11/2018	06/11/2018
5.4	Release on line del código fuente con documentación.	4horas	06/11/2018	08/11/2018
5.5	Informar a la comunidad del Proyecto CIAA y programadores de S.E.	1hora	08/11/2018	08/11/2018
5.6	Evaluar el cumplimiento de requerimientos.	3horas	08/11/2018	08/11/2018
5.7	Preparación de la presentación del proyecto.	4horas	08/11/2018	10/11/2018

TABLA 2.1: Planificación del Trabajo Final.

Capítulo 3

Diseño e implementación

En este capítulo se presenta el diseño de la biblioteca sAPI con un enfoque *top-down*. Se exponen los principios de diseño adoptados, la descripción de una aplicación donde se utiliza la biblioteca, la arquitectura general de la biblioteca, el modelo abstracto de un módulo de biblioteca, el diseño de los módulos de biblioteca de una plataforma de hardware genérica, su verificación y un diseño de archivo de texto para la descripción de una cierta plataforma concreta.

3.1. Descripción general

El sistema propuesto para la creación de bibliotecas de software para la programación de plataformas de hardware se ilustra en la figura 3.1.

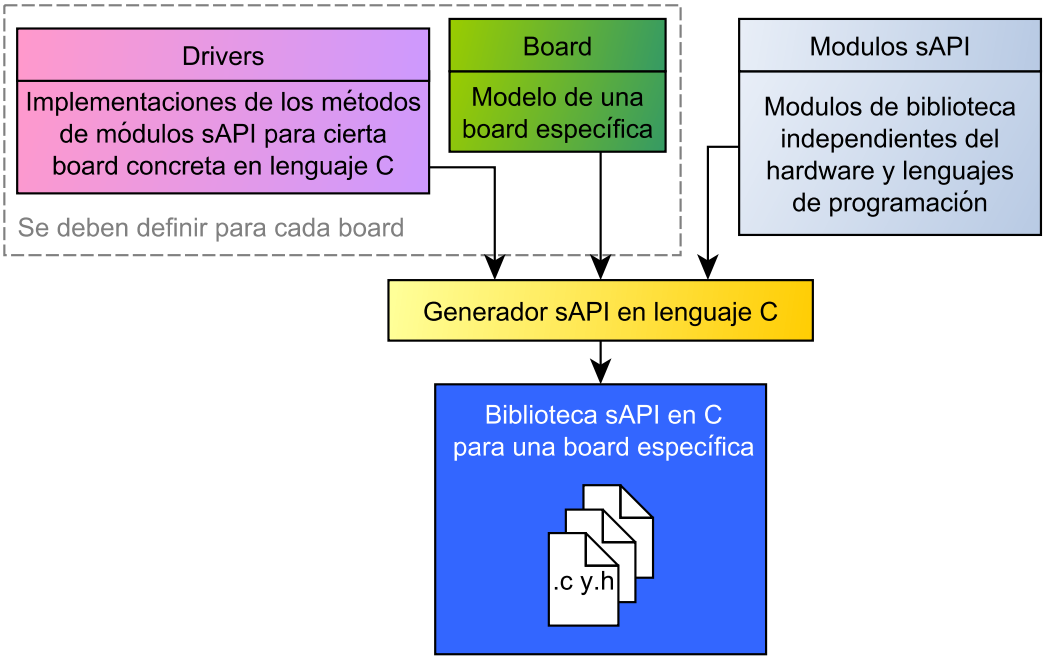

FIGURA 3.1: Diagrama del sistema diseñado.

Se diseñó un modelo de plataforma de hardware (en adelante *board* por sus siglas en inglés). Este modelo describe una plataforma de hardware completa y se debe instanciar para cada nueva plataforma de hardware.

Por otra parte, se desarrollaron módulos de biblioteca (en adelante módulos sAPI) independientes del hardware y lenguajes de programación. Estos definen propiedades y métodos describen las propiedades y métodos de cada uno de los periféricos de un cierto SoC, definiendo entonces, la API de la biblioteca.

Además, se debe proveer la implementación de cada uno de los métodos de los módulos sAPI (en adelante *drivers*). Estos *drivers* deben estar escritos en un cierto lenguaje de programación y son dependientes de la arquitectura del hardware.

También, se diseñó un generador de biblioteca sAPI. Este generador se debe definir para cada lenguaje de programación. Para este trabajo final se desarrolló solamente el generador de lenguaje C, sin embargo, el mismo se realizó de forma modular para que pueda ser fácilmente adaptado a otros lenguajes.

Mediante el generador de C, utilizando una instancia de *board* y *drivers* específicos, junto con los módulos sAPI, se genera entonces una biblioteca sAPI en lenguaje C para una plataforma particular.

En las siguientes secciones de detalla el diseño e implementación de cada una de las entidades descriptas. Las mismas se modelaron utilizaron los conceptos del paradigma de la programación orientada a objetos, que son especialmente útiles para describir módulos de software que encapsulan funcionalidad. Estos se describen mediante diagramas UML de clases de forma independiente del lenguaje de programación.

3.2. Modelo de plataforma de hardware

En la figura 3.2 se muestra el diagrama que describe una *board* y las partes que la componen.

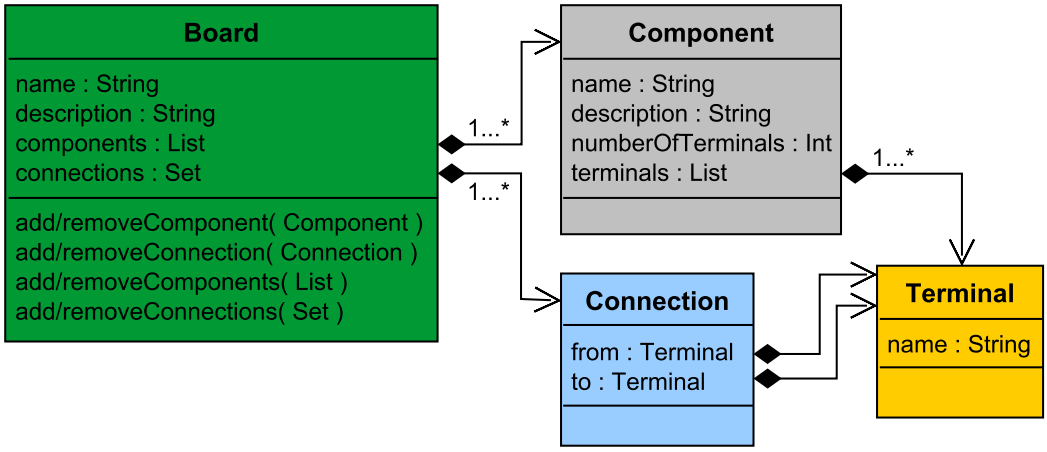

FIGURA 3.2: Diagrama de clases de *board*.

Estas partes son:

- Componente: representa los componentes dentro de cierta placa, por ejemplo, circuitos integrados, botones, leds, conectores, etc.
- Terminal: describe un terminal de conexión que posee cada componente.
- Conexión: modela el mapa de conexiones entre componentes. Cada conexión se compone de los dos terminales conectados.

3.2.1. Componente

Un componente incluye nombre, documentación, una lista de terminales de conexión y la cantidad de terminales. En la figura 3.3 se expone el diagrama de clases. Existen los siguientes tipos de componentes:

- *SystemOnChip*: modela un sistema completo en un chip.
- Componente con driver: modela a componentes que necesitan un driver, por ejemplo, LEDs, botones, sensores y memorias montados en la placa.
- Conector: representa los conectores físicos de la placa, como ser, tira de pines, borneras, etc. Su importancia en el modelo radica en la descripción de a qué terminales del SoC se conectan cada uno.

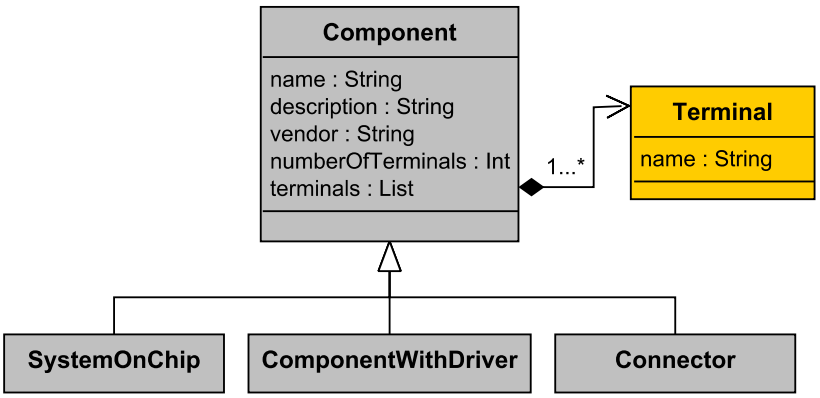

FIGURA 3.3: Diagrama de clases de componente.

3.2.2. Modelo de SoC

Un SoC describe un sistema completo dentro de un chip. Este término se utiliza para describir tanto microcontroladores (que incluyen núcleos de procesamiento, memorias y diversos periféricos), como sistemas que incluyen además lógica programable (FPGA) o módulos analógicos de radiofrecuencia complejos como ser Bluetooth y Wi-Fi.

En la figura 3.4 se muestra el modelo de SoC, el cual se compone de núcleos de procesamiento (*Core*), periféricos (*Peripheral*) y memorias (*Memory*).

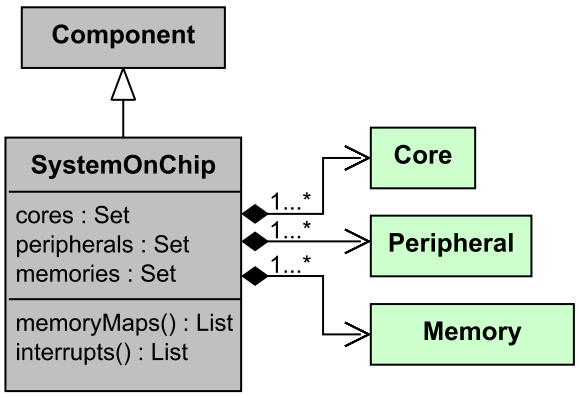

FIGURA 3.4: Diagrama de clases de SoC.

3.2.3. Modelo de IP core

Para modelar núcleos de procesamiento, periféricos y memorias se utiliza el concepto de *IP core*. Un núcleo de propiedad intelectual es un bloque de lógica o datos reutilizable, para definir un diseño de hardware de forma abstracta. Normalmente se utilizan en electrónica como componentes en un diseño de hardware ya sea en una FPGA o ASIC. Define una interfaz de señales y comportamiento interno.

En la figura 3.5 se muestra el modelo de *IPCore* y sus relaciones.

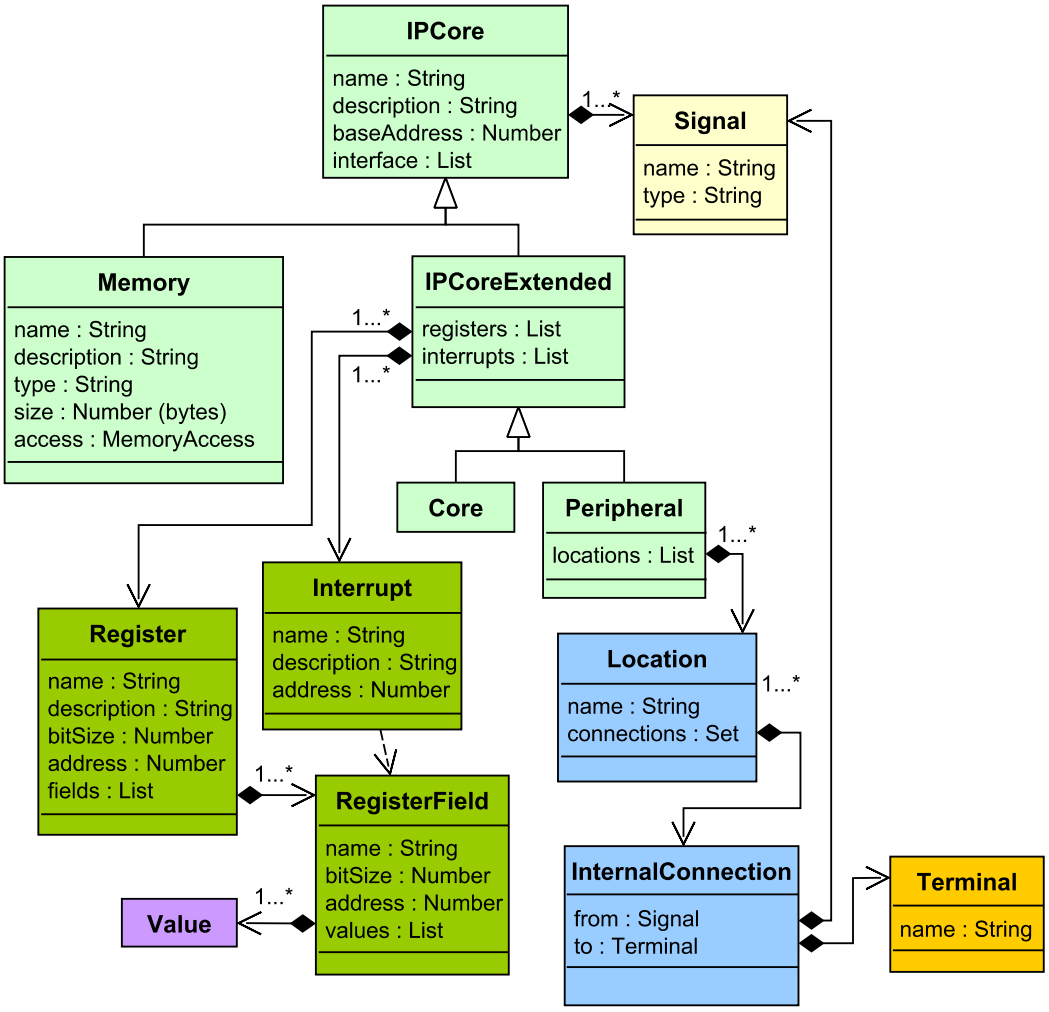

FIGURA 3.5: Diagrama de clases de *IPcore*.

Un *IPCore* define una dirección base, que se utiliza para localizarlo en el mapa de memoria. También define una interfaz que se compone de una lista ordenada de señales.

Las clases *Memory* e *IPCoreExtended* heredan de la clase *IPCore* agregándole funcionalidades particulares.

Memory modela una memoria interna del SoC, por ejemplo, RAM o Flash. Tiene las propiedades: tipo de memoria, tamaño en bytes y tipo de acceso permitido (lectura, lectura/escritura).

IPCoreExtended contiene registros e interrupciones. Los registros se modelan tanto para fines de documentación, como para poder realizar *tests* sobre valores de los

mismos. Las interrupciones son necesarias para conocer cuales tenemos disponibles físicamente y a qué evento responden. De la clase *IPCoreExtended* heredan *Core* y *Peripheral*. La primera modela un núcleo de procesamiento y la segunda un periférico.

Peripheral contiene una lista ordenada de *Locations* (ubicaciones). Una ubicación define un conjunto de conexiones entre señales del periférico y terminales del SoC. Esto permite modelar las posibles configuraciones de pines de un dado periférico.

3.2.4. Archivos para la descripción de una plataforma de hardware

Para definir una plataforma de hardware se utilizan archivos de texto en formato *JSON* [], de esta manera se puede definir cada una de las entidades presentadas por separado y referenciarlas.

Como ejemplo, se expone a continuación los archivos propuestos para una pequeña plataforma de hardware ficticia nombrada "fake-board-v1" (se ilustra en la figura 3.6).

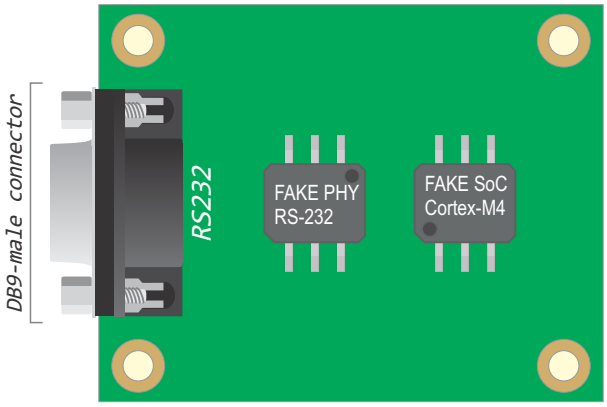

FIGURA 3.6: Fake-board-v1.

Archivo de descripción de *board* nombrado *fakeBoard.json*:

```
{
  "type": "Board",
  "name": "fake-board-v1",
  "components": [
    {
      "type": "@/vendor1/soc/fakeSoc.json",
      "name": "SOC0"
    },
    {
      "type": "@/vendor2/phy/fakePhy232.json",
      "name": "PHY0"
    },
    {
      "name": "RS232",
      "type": "@/vendor3/connector/fakeDb9Male.json"
    }
  ],
```

```
  "connections": [
    [ "#/SOC0/P1", "#/PHY0/RX_LVL" ],
    [ "#/SOC0/P2", "#/PHY0/TX_LVL" ],
    [ "#/SOC0/P3", "#/PHY0/GND" ],
    [ "#/PHY0/RX_232", "#/RS232/P2" ],
    [ "#/PHY0/TX_232", "#/RS232/P3" ],
    [ "#/PHY0/GND", "#/RS232/P5" ]
  ]
}
```

"@" indica una referencia a un archivo externo con el path incluido, mientras que "#" indica una referencia a un terminal de un componente.

Archivo de descripción de *SoC* nombrado *fakeSoc.json*:

```
{
  "type": "SystemOnChip",
  "name": "fake-board-v1",
  "cores": [
    {
      "type": "@/arch/arm/armv7m.json",
      "name": "CORTEXM4"
    }
  ],
  "peripherals": [
    {
      "type": "@/vendor1/ipcore/uart.json",
      "name": "UART0"
    }
  ],
  "memories": [
    {
      "type": "@/vendor1/mem/sram8kb.json",
      "name": "RAM"
    },
    {
      "type": "@/vendor1/mem/flash32kb.json",
      "name": "FLASH"
    }
  ],
  "terminals": [ "P1", "P2", "P3" ]
}
```

Archivo de descripción de un chip de capa física para RS-232 nombrado *fakePhy232.json*:

```
{
  "type": "ComponentWithDriver",
  "name": "phy-rs232",
  "terminals": [
    "RX_LVL", "TX_LVL", "GND",
    "RX_232", "TX_232"
  ],
```

```
  "drivers": [
    {
      "lang":"c",
      "files": [
        "@/driver/c/vendor2/phy/fakePhy232.h",
        "@/driver/c/vendor2/phy/fakePhy232.c"
      ]
    }
  ]
}
```

Archivo de descripción de un conector *DB-9* nombrado *fakeDb9Male.json*:

```
{
  "type": "Connector",
  "name": "DB9-male",
  "terminals": [
    "P1", "P2", "P3", "P4", "P5", "P6", "P7", "P8", "P9"
  ]
}
```

Cabe destacar que el ejemplo presentado es válido solamente para describir la estructura de los archivos y filosofía de diseño adoptada, sin embargo, los archivos de implementación poseen más propiedades y niveles de anidamiento en sus entidades.

3.3. Módulos de biblioteca sAPI

Se utilizan las siguientes premisas para guiar el diseño de los módulos de la biblioteca sAPI:

- Utilización de nombres sencillos para facilitar el aprendizaje y uso.
- Mantener baja la extensión de la definición de la API.
- Dar soporte a la programación de núcleos de procesamiento y periféricos, utilizando los modos más comunes de funcionamiento de los mismos, con un nivel de abstracción suficiente para independizarse del hardware, pero manteniendo la identidad y conceptos de cada cada periférico.
- Debe especificar las dependencias entre módulos de la biblioteca, y el uso de recursos físicos en cada implementación particular.
- La definición de los módulos se realiza independiente del lenguaje de programación.

Para el diseño de la API se llevó a cabo un estudio exhaustivo sobre arquitecturas de núcleos de procesamiento y periféricos, así como de las bibliotecas disponibles. Entre las bibliotecas relevadas se destacan:

- MicroPython libraries [].
- STM32 HAL [].
- RUST Embedded HAL [].
- Mbed OS drivers API [].

- ChibiOS/HAL [].
- CMSIS [].
- LPCOpen [].
- MCUXpresso [].
- EMLIB [].

Se realizó un rediseño completo de la API, mejorando y apliando muchas características, pero provocando un quiebre en la compatibilidad con versiones anteriores de sAPI.

3.3.1. Definiciones de abstractas de sAPI

Para la creación de módulos de sAPI independientes del lenguaje de programación se definen tipos de datos, constantes y modos de acceso de parámetros de funciones abstractos, cuya implementación luego dependerá de lenguaje particular. Además, cada módulo agrega sus propias definiciones.

Constantes

Los nombres de las constantes se escriben en mayúsculas y con guiones bajos para separar palabras múltiples.

- Estados lógicos: *TRUE* y *FALSE*.
- Estados funcionales: *ON* y *OFF*.
- Estados eléctricos: *HIGH* y *LOW*.
- Estados de habilitación: *ENABLE* y *DISABLE*.
- Valor nulo: *NULL*.

Tipos de datos

Los tipos de datos se indican con el sufijo *_t* para que se distinga a simple vista los tipos de datos de las variables o parámetros formales de los métodos.

Tipos de datos primitivos:

- Booleano: *bool_t*.
- Entero: *int_t*.
- Flotante: *float_t*.
- Byte: *byte_t*.

Tipo de datos para manejo de cadenas de caracteres:

- *String_t*.

Arreglos:

- *Array_t*. Permite guardar elementos ordenados. Longitud fija.

- *List_t*. Permite guardar elementos ordenados. Longitud variable.
- *Set_t*. Permite guardar elementos sin orden. Longitud variable.
- *ByteArray_t*. Arreglo de bytes.

Tipos de datos para temporización:

- *Time_t*, representa una duración de tiempo en días horas, minutos, segundos y milisegundos.
- *TimeOfDay_t*, representa un valor de tiempo absoluto en horas, minutos, segundos y milisegundos. El rango es de $00:00:00{,}000$ a $23:59:59{,}999$.
- *Date_t*, representa una fecha (año, mes y día).
- *DateAndTime_t*, que contiene ambos valores anteriores.

Tipo de datos para manejo de eventos:

- *Callback*: *Callback_t*. Representa a una función a llamar en respuesta a un evento.

Modificadores

Tanto para las variables como parámetros formales se definen los siguientes modificadores:

- *in*: la variable o parámetro que se precede con el modificador *in* sólo puede ser leída.
- *out*: la variable o parámetro que se precede con el modificador *out* es de solo escritura.
- *inOut*: la variable o parámetro que se precede con el modificador *inOut* puede ser tanto leída como escrita.
- *const*: la variable o parámetro que se precede con el modificador *const* no se puede modificar.

En caso de no especificar un modificador el comportamiento por defecto es que las variables se comportan como *inOut*, mientras que los parámetros formales de métodos se comportan como *in* dentro del mismo.

3.3.2. Módulo de sAPI

Cada módulo de biblioteca sAPI define:

- Nombre del módulo.
- Descripción. Se utiliza para la generación de documentación.
- Versión del módulo. Se utiliza versionado semántico [].
- Autor del módulo. Se debe completar nombres, apellidos y dirección de correo electrónico del autor.
- Licencia. Se puede completar con el texto completo de la licencia del módulo o un enlace a la misma. Los módulos sAPI utilizan la licencia de código abierto *BSD-3clause* [].

- Dependencias. Una lista de los módulos de sAPI de los cuales depende el módulo actual. Se puede además exigir que la dependencia sea a una versión específica, a partir de una versión mínima o hasta cierta versión máxima.

Además, define un conjunto de tipos de datos, constantes, propiedades y métodos.

En los módulos de sAPI se encuentran las siguientes categorías de propiedades (o atributos):

- Configuración: corresponde a las propiedades para la configuración de modos de funcionamiento de un módulo. Por ejemplo, *power*, *clockSource*, etc.
- Valor: un cierto valor que se puede leer o escribir en un módulo.
- Eventos: representa los eventos que controla el módulo.

Todos los módulos poseen los siguientes métodos:

- *init* (inicialización): es un método para inicializar el módulo. Permite establecer múltiples parámetros de configuración.
- *read* (leer): permite leer el valor característico del módulo.
- *write* (escribir): permite escribir un valor característico del módulo.
- *deInit* (restablecer configuración): es un método para restablecer el módulo a la configuración por defecto.
- *interrupt*: es un método para habilitar/deshabilitar interrupciones del módulo.
- *getters y setters* de cada propiedad.

Se destaca que en una implementación de la biblioteca para lenguajes no orientados a objetos, en lugar de métodos se tiene funciones y por lo tanto, se deberá agregar en cada una el parámetro *this* que corresponde a una referencia al objeto sobre el cual se ejecuta el método. Por ejemplo, en un lenguaje con objetos como *JavaScript* tendremos,

- `uart0.init( 9600 )`
  `<objeto>.<método>( <parámetros> )`

mientras que en un lenguaje sin objetos como *C*,

- `uartInit( UART0, 9600 )`
  `<función>( <objeto>, <parámetros> )`

donde además, por problemas de espacio de nombres, se debe agregar un prefijo con el nombre del periférico.

3.3.3. Módulo GPIO

El módulo GPIO modela tanto un único pin de entrada/salida de propósito general (pin), así como a un conjunto de pines (puerto).

Tipos de datos

- *Pin_t*, modela un pin.
- *Port_t*, representa un puerto.
- *PortValue_t*, representa el valor de todos los pines del puerto.

La cantidad de pines que forman un puerto depende de la arquitectura.

Propiedades de pin

- *mode*, del tipo *PinMode_t*, cuyos valores posibles son: *DISABLE* (deshabilidato), *INPUT* (pin configurado como entrada) y *OUTPUT* u *OUTPUT_PUSHPULL* (pin configurado como salida, modo *push-pull*), *OUTPUT_OPEN_DRAIN* (pin como salida, en modo drenador abierto).
- *pull*, del tipo *PinPull_t* cuyos valores posibles son: *NONE* (modo por defecto, pin flotante), *PULL_UP* (con resistencia de *pull-up*), *PULL_DOWN* (con resistencia de *pull-down*), *PULL_BOTH* (con ambas resistencias).
- *value*, del tipo *bool_t*. Representa el valor a escribir o leer de un pin.

Métodos de pin

Los métodos básicos de configuración y uso de un pin son:

- `init( PintInit_t init )`
- `read() : bool_t`
- `write( bool_t value )`
- `deInit()`

El parámetro *init* se debe completar con un valor posible de *mode*, y opcionalmente, un valor de *pull* separados con |, por ejemplo:

- `init( INPUT )`
- `init( INPUT | PULL_UP )`

Contiene además el método *toggle()* intercambia el valor del pin si el mismo está configurado como salida.:

- `toggle()`

Para establecer una interrupción en un pin ante cierto evento define los métodos:

- `eventCallbackSet( PinEvent_t evt, Callback_t func )`
- `eventCallbackClear( PinEvent_t evt )`
- `interrupt( bool_t enable ) : PinStatus_t`

eventCallbackSet() establece el evento ante el cual se interrumpe, junto a una función de *callback* que se ejecutará ante su ocurrencia. *eventCallbackClear()* permite remover el *callback*. Los posibles eventos son: *HIGH_LEVEL* (interrupción por nivel alto), *LOW_LEVEL* (interrupción por nivel bajo), *RISING_EDGE* (interrupción por flanco ascendente), *FALLING_EDGE* (interrupción por flanco descendente) y

BOTH_EDGES (interrupción por flanco ascendente y descendente); todos del tipo *PinEvent_t*.

interrupt() permite habilitar o deshabilitar la interrupción de hardware del pin ante la ocurrencia del evento. Según la plataforma particular existe la posibilidad de que soporten interrupción algunos pines fijos, o que exista una cierta cantidad de interrupciones de pin y se pueda elegir a qué pines se le asignan, es por eso que el método *interrupt()* retorna un valor del tipo *PinStatus_t*. Este tipo de datos define los valores *STATUS_OK*, si pudo configurar la interrupción, *ERROR_NOT_SUPPORT*, si el pin no soporta capacidad de interrupción, y *ERROR_NO_MORE_AVAILABLE*, si ya se configuraron todas las interrupciones de hardware disponibles.

Propiedades y métodos de puerto

Puerto define la propiedad *value*, del tipo *PortValue_t*. Representa el valor a escribir o leer en un puerto.

Para la configuración y uso de un puerto se definen los métodos:

- `init( List_t config )`
- `read() : PortValue_t`
- `write( PortValue_t value )`
- `deInit()`

El método *init()* de puerto requiere una lista de configuraciones. Los métodos *read()* y *write()* permiten leer y escribir el puerto completo.

Se agregan además los métodos:

- `pins(): List_t`
- `set( PortValue_t mask )`
- `reset( PortValue_t mask )`

El método *pins()*, devuelve una lista de pines que componen el puerto. Los métodos *set()* y *reset()* permiten escribir el puerto afectado por una máscara de bits que se le pasa como parámetro.

3.3.4. Módulo ADC

El módulo ADC modela un periférico Conversor Analógico-Digital. Este periférico contiene varios canales de conversión multiplexados los cuales se pueden configurar individualmente como entradas analógicas.

Propiedades de ADC

- *conversionRate*, del tipo *int_t*, representa la tasa de conversión en Hertz. Los valores posibles dependen de cada plataforma. Se pueden utilizar alternativamente los valores genéricos: *LOW*, *MEDIUM*, *HIGH* y *VERY_HIGH*.
- *conversionMode*, del tipo *AdcConvMode_t*, con valores: *SINGLE* (activado por software desde el programa de usuario), *CONTINUOUS* (conversion periódica disparada por hardware a tasa *conversionRate*).

- *voltageReferece*, del tipo *AdcVRef_t* cuyos valores posibles son: *VCC*, *INTERNAL* y *EXTERNAL*.
- *resolution*, del tipo *AdcRes_t*, cuyos valores también dependen de la plataforma pero se pueden utilizar en su lugar los valores genéricos.
- *channelsMode*, del tipo *AdcChannelsMode_t*, con valores: *SIGLE* y *DIFFERENTIAL*.
- *location*, del tipo *Location_t*. Son las posibles ubicaciones del periférico con respecto a pines físicos del SoC. Si bien dependen de la arquitectura, se definen los valores genéricos *LOCATION0* a *LOCATION7* para cada plataforma.

Métodos de ADC

Inicialización y restablecimiento de un periférico ADC:

- `init()`
- `init( int_t conversionRate, AdcInit_t init )`
- `deInit()`

Los valores posibles del parámetro *init* se forman con valores de:

- `conversionMode | voltageReferece | resolution | channelsMode | location`

Todos los valores de configuración del método *init()* son opcionales y los que no se aplican se inicializan por defecto con:

- `conversionRate = VERY_HIGH`
- `voltageReferece = VCC`
- `conversionMode = SINGLE`
- `resolution = VERY_HIGH`
- `channelMode = SINGLE`
- `loacation = LOCATION0`

Habilitación/deshabilitación de un canal particular del ADC (configuración de una entrada analógica):

- `channel( AdcChannel_t channel, bool_t enable )`

Para conocer los canales disponibles de un ADC existe el método *channelsGet()* que devuelve la lista de canales del ADC. Esta lista se reduce a la mitad si el modo *channelsMode* es *DIFFERENTIAL)*

- `channelsGet() : List_t`

Si el ADC tiene configurado *conversionMode* = *CONTINUOUS* muestrea automáticamente los canales habilitados. El método*startConversion()* comienza el muestreo automático, para parar el muestreo se utiliza *stopConversion()*. Si en cambio *conversionMode* = *SINGLE*, la lectura se debe lanzar mediante el método *startConversion()* que recibe un canal como parámetro. Una conversión se puede abortar con *stopConversion()*.

- `startConversion( AdcChannel_t channel )`
- `stopConversion()`

Existen 2 formas de leer los valores convertidos del ADC, por encuesta, o por interrupción. En el primer caso se debe chequear si finalizó la conversión con el método *status()*, que retorna un un valor del tipo *AdcStatus_t*. Si se realizó la conversión correctamente se puede leer el valor convertido con el método *read()*.

- `status() : AdcStatus_t`
- `read() : AdcValue_t`

Los valores posibles de *AdcStatus_t* son:

- `READY`
- `BUSY`
- `CONVERSION_COMPLETE`
- `ERROR`
- `ERROR_TIMEOUT`

Para leer un canal por interrupción se debe primero establecer una función de *callback* al evento *CONVERSION_COMPLETE*, otra al evento *ERROR* y luego activar la interrupción del periférico. Esto se realiza con los métodos:

- `eventCallbackSet( ChannelEvent_t evt, Callback_t func )`
- `eventCallbackClear( ChannelEvent_t evt )`
- `interrupt( bool_t enable )`

Los eventos del ADC son del tipo *AdcEvent_t*. Al completar la conversión se ejecutará una de las dos funciones.

Métodos de Canal de ADC

Como alternativa, existen métodos que aplican a cierto canal de ADC y proveen un mayor nivel de abstracción:

- `read() : AdcValue_t`
- `readSync( AdcValue_t data, Time_t timeout ) : AdcStatus_t`
- `readAsync( Callback_t sucess, Callback_t error )`

Tanto *read()* como *readSync()* realizan una lectura bloqueante, en consecuencia, es necesario establecer un tiempo de *time out* para que no bloquee indefinidamente. *read()* en fija automáticamente el *timeout* al doble del *conversionRate*, siempre retorna un valor, que a veces puede ser inválido. *readSync()* devuelve el estado de conversión, para que el usuario chequee si valor convertido es válido, en ese caso lo carga en la variable *data*, que recibe como parámetro. Si se establece en 0 el valor de *timeout* en el método con sufijo *Sync* entonces el tiempo de `timeout` será considerado infinito. Lo mismo sucede con todos los métodos que cuenten con dicho sufijo que se describen que se describen en esta sección.

readAsync() realiza una lectura no bloqueante. Recibe como parámetros dos funciones de *callback*, una que lanza en caso de conversión exitosa (si esto ocurre le pasa como parámetro el valor convertido) y otra en caso de error (le pasa el estado de conversión).

Este módulo no define método *write()*.

3.3.5. Módulo DAC

El módulo DAC modela un periférico Conversor Digital-Analógico. Este periférico comparte muchas características con el ADC, contando también con uno o más canales de conversión, los cuales se pueden configurar individualmente como salidas analógicas.

Propiedades de DAC

- *conversionRate*, del tipo *int_t*, representa la tasa de conversión en Hertz.
- *conversionMode*, del tipo *DacConvMode_t*.
- *voltageReferece*, del tipo *DacVRef_t*.
- *resolution*, del tipo *DacRes_t*.
- *location*, del tipo *DaLocation_t*.

Métodos de DAC

Inicialización y restablecimiento de un periférico DAC:

- `init()`
- `init( int_t conversionRate, DacInit_t init )`
- `deInit()`

Los valores posibles del parámetro *init* se forman con valores de:

- `conversionMode | voltageReferece | resolution | location`

Todos los valores de inicialización son opcionales y si no se aplican se aplican utiliza por defecto:

- `conversionRate = VERY_HIGH`
- `voltageReferece = VCC`
- `conversionMode = SINGLE`
- `resolution = VERY_HIGH`
- `loacation = LOCATION0`

El resto de la API también es muy similar a la del ADC, con la diferencia que define métodos para escribir el DAC en lugar de métodos para leer (no define el método *read()*):

- `channel( DacChannel_t channel, bool_t enable )`
- `channelsGet() : List_t`
- `write( DacValue_t value )`
- `startConversion( DacChannel_t channel )`
- `stopConversion()`
- `status() : DacStatus_t`
- `eventCallbackSet( ChannelEvent_t evt, Callback_t func )`
- `eventCallbackClear( ChannelEvent_t evt )`
- `interrupt( bool_t enable )`

Los estados y eventos son los mismos que los presentados para ADC, aunque del tipo *DacStatus_t* y *DacEvent_t*, respectivamente.

Métodos de Canal de DAC

- `write( DacValue_t value )`
- `writeSync( DacValue_t value, Time_t timeout )`
  `: DacStatus_t`
- `writeAsync( DacValue_t value,`
  `Callback_t sucess,`
  `Callback_t error`
  `)`

Se agrega el conjunto de métodos para la generación de señales:

- `waveGenSet( Array_t samples, Time_t periodicity )`
- `waveGenStart()`
- `waveGenStop()`

3.3.6. Módulo UART

El módulo UART modela un periférico de comunicación serie transmisor/receptor asincrónico universal.

Propiedades de UART

- *baudRate*, del tipo *int_t*, tasa de baudios. Depende de la arquitectura pero en general se permiten: 1200, 2400, 4800, 9600, 19200, 38400, 57600 o 115200.
- *dataBits*, del tipo *UartDataBits_t*, que puede tomar los valores *DATABITS5*, *DATABITS6*, *DATABITS7*, *DATABITS8* o *DATABITS9*.
- *parity*, del tipo *UartParity_t*, con valores posibles *NONE*, *EVEN* o *ODD*.
- *stopBits*, del tipo *UartStopBits_t*, con valores *STOPBITS1*, *STOPBITS2* o *STOPBITS1_5*.
- *flowControl*, del tipo *UartFlowCtrl_t*, cuyos valores posibles son *NONE*, *CTS*, *RTS* o *RTS_CTS*.
- *enableTransmitter*, del tipo *bool_t*, se utiliza para habilitar/deshabilitar el transmisor de la UART.
- *enableReceiver*, del tipo *bool_t*, permite habilitar/deshabilitar el receptor de la UART.
- *location*, del tipo *Location_t*.
- *transmitValue*, del tipo *byte_t*. Representa el valor a transmitir.
- *receiveValue*, del tipo *byte_t*. Representa el último valor recibido.

Métodos de UART

Inicialización y restablecimiento de un periférico UART:

- `init()`
- `init( int_t baudRate )`
- `init( int_t baudRate, UartInit init )`
- `deInit()`

Los valores posibles del parámetro *init* se forman con valores de:

- `dataBits | parity | stopBits | flowControl | enableTransmitter | enableReceiver | location`

Todos los valores del método *init()* son opcionales. En caso de no inicializarlos los valores por defecto son:

- `baudRate = 9600`
- `dataBits = DATABITS8`
- `parity = NONE`
- `stopBits = STOPBITS1`
- `flowControl = NONE`
- `loacation = LOCATION0`
- `enableTransmitter = TRUE`
- `enableReceiver = TRUE`

Se puede transmitir o recibir un byte usando el módulo UART tanto por encuesta como por interrupción mediante los métodos:

- `status() : UartStatus_t`
- `send( const byte_t value )`
- `receive() : byte_t`
- `eventCallbackSet( UartEvent_t evt, Callback_t func )`
- `eventCallbackClear( UartEvent_t evt )`
- `interrupt( bool_t enable )`

Los posibles valores de *UartStatus_t* son:

- `TRANSMITTER_READY`
- `TRANSMIT_COMPLETE`
- `RECEIVE_COMPLETE`
- `ERROR_TRANSMIT_DATA_OVERRUN`
- `ERROR_RECEIVE_FRAMER`
- `ERROR_RECEIVE_DATA_OVERRUN`
- `ERROR_RECEIVE_PARITY`
- `ERROR_TIMEOUT`

Eventos (tipo *UartEvent_t*):

- `TRANSMITTER_READY`
- `TRANSMIT_COMPLETE`
- `RECEIVE_COMPLETE`
- `ERROR_TRANSMIT`
- `ERROR_RECEIVE`

Existe además el método *sendBreak()* que envía una *break condition* al bus. Esto significa que mantiene la línea de transmisión en nivel bajo durante un tiempo mayor al requerido para enviar un caracter.

- `sendBreak()`

Métodos de UART de alto nivel

Métodos de lectura de byte:

- `readByte() : byte_t`
- `readByteSync( inOut byte_t value, Time_t timout ) : UartStatus_t`
- `readByteAsync( Callback_t sucess, Callback_t error )`

Métodos de escritura de byte:

- `writeByte( const byte_t value )`
- `writeByteSync( const byte_t value, Time_t timout ) : UartStatus_t`
- `writeByteAsync( const byte_t value, Callback_t sucess, Callback_t error )`

Métodos de lectura de *String*:

- `readString( inOut String_t data, inOut size_t dataSize, String_t terminator )`
- `readStringSync( inOut String_t data, inOut size_t dataSize, String_t terminator, Time_t timout ) : UartStatus_t`
- `readStringAsync( inOut String_t data, inOut size_t dataSize, String_t terminator, Callback_t sucess, Callback_t error )`

Métodos de escritura de *String*:

- `writeString( const String_t data inOut size_t dataSize )`
- `writeStringSync( const String_t data, inOut size_t dataSize, Time_t timeout ) : UartStatus_t`
- `writeStringAsync( const String_t value, inOut size_t dataSize, Callback_t sucess, Callback_t error )`

Métodos de lectura de arreglo de bytes:

- `readByteArray( inOut ByteArray_t data, inOut size_t dataSize )`
- `readByteArraySync( inOut ByteArray_t data, inOut size_t dataSize, Time_t timeout ) : UartStatus_t`
- `readByteArrayAsync( inOut ByteArray_t data, inOut size_t dataSize, Callback_t sucess, Callback_t error )`

Métodos de escritura de arreglo de bytes:

- `writeByteArray( const ByteArray_t data, inOut size_t dataSize )`
- `writeByteArraySync( const ByteArray_t data, inOut size_t dataSize, Time_t timeout ) : UartStatus_t`
- `writeByteArrayAsync( const ByteArray_t data, inOut size_t dataSize, Callback_t sucess, Callback_t error )`

En todos los métodos de lectura se le pasa como primer parámetro la variable donde escribirá los datos recibidos. En la lectura de *String* se le debe pasar un parámetro `terminator` que es un `String` que indica como debe ser el fin de cadena a buscar. Cuando se lee un *String* o *Byte Array* se almacena la cantidad de datos leídos en la variable *dataSize* que se le pasa como parámetro.

3.3.7. Módulo SPI

Modela un bus de serie para interfaz con periféricos (bus SPI). El periférico SPI proporciona comunicación serial sincrónica *full duplex* entre dispositivos maestros y esclavos. Se usa comúnmente para la comunicación con periféricos externos como memorias flash, sensores, relojes en tiempo real (RTC), etc.

Propiedades de SPI

- *mode*, del tipo *SpiMode_t*, que representa el modo de funcionamiento del periférico SPI, con valores *MASTER* o *SLAVE*.
- *clockFrequency*, del tipo *int_t*, es la frecuencia de reloj (en Hz) del BUS SPI (solo tiene sentido en modo *MASTER*).
- *clockPolarity*, del tipo *SpiClockParity_t*, define el nivel cuando se considera activo el clock, con valores *POLARITY_LOW* o *POLARITY_HIGH*.

- *clockPhase*, del tipo *SpiClockPhase_t*, establece en que flanco de reloj se muestrean los datos, cuyos valores posibles son *PHASE_EDGE1* o *PHASE_EDGE2*.
- *dataBits*, del tipo *SpiDataBits_t*, que puede tomar los valores *DATABITS8* o *DATABITS16*.
- *dataOrder*, del tipo *SpiDataOrder_t*, representa el orden con el que se mueven los bits en el registro de desplazamiento, es decir si se mueven primero el bit menos significativo o el más significativo: *MSB_FIRST* o *LSB_FIRST*.
- *location*, del tipo *Location_t*.
- *transmitValue*, del tipo *ByteArray_t*. Representa el valor a transmitir.
- *receiveValue*, del tipo *ByteArray_t*. Representa el último valor recibido.

Métodos de SPI

Inicialización y restablecimiento de un periférico SPI:

- `init()`
- `init( SpiInit_t init )`
- `init( SpiInit_t init, int_t clockFrequency )`
- `deInit()`

Los valores posibles del parámetro *init* se forman con valores de:

- `mode | clockPolarity | clockPhase | dataOrder | dataBits | location`

Son todos opcionales, con valores por defecto:

- `mode = MASTER`
- `clockFrequency = 100000`
- `clockPolarity = POLARITY_HIGH`
- `clockPhase = PHASE_EDGE1`
- `dataBits = DATABITS8`
- `dataOrder MSB_FIRST`
- `location = LOCATION0`

En caso de seleccionar modo *MASTER* se debe configurar la frecuencia de reloj y al menos un pin como selector de esclavo. La frecuencia de reloj puede configurarse usando el método *init()* de 2 parámetros, o bien, mediante:

- `clockFrequencySet( int_t clockFrequency )`

Utilizando *initSSPin()* se configura un pin como selector de esclavo:

- `initSSPin( Pin_t pin )`

Luego para elegir que esclavo habilitar o deshabilitar se utiliza:

- `slaveSelect( Pin_t pin, bool_t enable )`

Permite realizar envío, recepción o transferencia de datos (que envía y recibe simultáneamente). Estas operaciones pueden realizarse por encuesta o interrupción con los siguientes métodos:

- `status() : SpiStatus_t`
- `send( const ByteArray_t data )`
- `receive( inOut ByteArray_t receive )`
- `transfer( const ByteArray_t send,`
  `inOut ByteArray_t receive`
  `) : SpiStatus_t`
- `eventCallbackSet( SpiEvent_t evt, Callback_t func )`
- `eventCallbackClear( SpiEvent_t evt )`
- `interrupt( bool_t enable )`

Los posibles valores de *SpiStatus_t* son:

- `READY`
- `BUSY`
- `TRANSFER_COMPLETE`
- `ERROR`
- `ERROR_TIMEOUT`

Eventos (tipo *SpiEvent_t*):

- `TRANSFER_COMPLETE`
- `TRANSFER_ERROR`

Métodos de SPI de alto nivel

Lectura de arreglo de bytes:

- `readByteArray( inOut ByteArray_t data,`
  `inOut size_t dataSize`
  `)`
- `readByteArraySync( inOut ByteArray_t data,`
  `inOut size_t dataSize,`
  `Time_t timeout`
  `) : SpiStatus_t`
- `readByteArrayAsync( inOut ByteArray_t data,`
  `inOut size_t dataSize,`
  `Callback_t sucess,`
  `Callback_t error`
  `)`

Escritura de arreglo de bytes:

- `writeByteArray( const ByteArray_t data,`
  `inOut size_t dataSize`
  `)`
- `writeByteArraySync( const ByteArray_t data,`
  `inOut size_t dataSize,`
  `Time_t timeout`
  `) : SpiStatus_t`
- `writeByteArrayAsync( const ByteArray_t data,`
  `inOut size_t dataSize,`
  `Callback_t sucess,`
  `Callback_t error`
  `)`

Transferencia de arreglo de bytes:

- `transferByteArray( const ByteArray_t send, inOut size_t sendSize, inOut ByteArray_t receive, inOut size_t dataSize )`
- `transferByteArraySync( const ByteArray_t send, inOut size_t sendSize, inOut ByteArray_t receive, inOut size_t dataSize, Time_t timeout ) : SpiStatus_t`
- `transferByteArrayAsync( const ByteArray_t send, inOut size_t sendSize, inOut ByteArray_t receive, Callback_t sucess, Callback_t error )`

3.3.8. Módulo I2C

El módulo I2C modela un periférico para comunicaciones serie entre circuitos integrados[1]. Al igual que SPI, es un bus sincrónico y permite conectarse a memorias, sensores y otros periféricos externos. El bus I2C requiere únicamente dos terminales para su funcionamiento, uno para la señal de reloj y otro para el envío y recepción de datos. Cada dispositivo conectado al bus dispone de una dirección distinta que se utiliza para su acceso. De esta forma, evita la necesidad de agregar una línea de conexión extra por cada periférico conectado al bus como sucede en el bus SPI.

Propiedades de I2C

- *mode*, del tipo *I2cMode_t*, que representa el modo de funcionamiento del periférico I2C, al igual que SPI perimite los valores *MASTER* o *SLAVE*.
- *addressMode*, del tipo *I2cAddrMode_t*, representa el modo de direccionamiento del bus, con posibles valores: *ADDR_MODE7* o *ADDR_MODE10* (7 o 10 bits de dirección).
- *clockDuty*, del tipo *I2cClockDuty_t*, que establece la relación del tiempo en alto y tiempo en bajo del reloj, con valores dependientes de la arquitectura. Se definen los posibles valores: *CLK_DUTY_SYMMETRIC*, *CLK_DUTY_ASYMMETRIC* o *CLK_DUTY_FAST*.
- *generateAck*, del tipo *bool_t*. Habilita o deshabilita el envío de ACK, de no estar habilitado (*true*) el periférico I2C estará virtualmente desconectado del bus.
- *acceptGlobalCalls*, del tipo *bool_t*. En modo *SLAVE* permite responder a solicitudes a la dirección $0x00$.

[1] En inglés *Inter-Integrated Circuit*, cuyas siglas son IIC, o I^2C.

- *address*, del tipo *int_t*, es la dirección del periférico I2C cuando se utiliza en modo esclavo.
- *clockFrequency*, del tipo *int_t*, es la frecuencia de reloj (en Hz) del BUS I2C (solo tiene sentido en modo *MASTER*). Alternativamente pueden utilizarse los valores *CLK_STD_MODE* (100 kHz), *CLK_FAST_MODE* (400 kHz) y *CLK_FAST_MODE_PLUS* (1 MHz).
- *location*, del tipo *Location_t*.
- *value*, del tipo *byte_t*. Representa el registro de transferencia.

Métodos de I2C

Inicialización y restablecimiento de un periférico I2C:

- `init()`
- `init( I2cInit_t config )`
- `init( I2cInit_t config, int_t freqOrAddr )`
- `deInit()`

Los valores posibles del parámetro *config* se forman con valores de:

- `mode | addressMode | clockDuty | generateAck | acceptGlobalCalls | location`

El parámetro *freqOrAddr* permite establecer la frecuencia de reloj si se configura en modo *MASTER*, o la dirección del dispositivo en modo *SLAVE*. Alternativamente se pueden utilizar los métodos:

- `clockFrequencySet( int_t clockFrequency )`
- `addressSet( int_t address )`

Todos los valores de configuración son opcionales, con valores por defecto:

- `mode = MASTER`
- `addressMode = ADDR_MODE7`
- `clockDuty = CLK_DUTY_SYMMETRIC`
- `generateAck = TRUE`
- `acceptGlobalCalls = FALSE`
- `location = LOCATION0`
- `clockFrequency = CLK_STD_MODE`
- `address = 0xEC`

Permite realizar envío o recepción de datos en ambos modos. Asimismo, estas operaciones pueden realizarse por encuesta o interrupción, se utilizan los siguientes métodos:

- `status() : I2cStatus_t`
- `sendStart()`
- `sendStop()`
- `sendCommandTo( I2cCommand_t, int_t slaveAddress )`
- `busReset()`
- `holdClock( bool_t enable )`
- `sendData( const byte_t data )`
- `startReception( I2cAck_t ackOrNack )`

- `eventCallbackSet( I2cEvent_t evt, Callback_t func )`
- `eventCallbackClear( I2cEvent_t evt )`
- `interrupt( bool_t enable )`

En modo *MASTER* se utilizan los métodos *sendStart()*, que realiza el envío de una condición de *start*) al bus, *sendStop()*, envía una condición de *stop*, *sendCommandTo()* que envía un comando de lectura/escritura a la dirección de un esclavo) y *busReset()* la cual genera una secuencia para volver el bus al estado *READY*, con las líneas *SCL* y *SDA* en estado alto.

Para *I2cCommand_t* existen los valores:

- `READ`
- `WRITE`

En modo *SLAVE* se utilza *holdClock()* para causar un estiramiento de reloj en el bus.

Los siguientes métodos son válidos en ambos modos:

- `status() : I2cStatus_t`
- `sendData( const byte_t data )`
- `startReception( I2cAck_t ackOrNack )`

Los posibles valores de *I2cStatus_t* son:

- `READY`
- `BUSY`
- `TRANSFER_COMPLETE`
- `SLAVE_NAK`
- `ERROR_NACK_RECEIVED`
- `ERROR_BUS`
- `ERROR_ARB_LOST`
- `ERROR`
- `ERROR_TIMEOUT`
- `REQUEST_TO_SEND`
- `REQUEST_TO_RECEIVE`

startReception() comienza una recepción de un byte en el bus, el parámetro del tipo *I2cAck_t* define si envía un *acknowledgement*, o no, al finalizar la recepción, con valores:

- `ACK`
- `NACK`

Eventos de I2C (tipo *I2cEvent_t*)::

- `TRANSFER_COMPLETE`
- `REQUEST_TO_SEND`
- `REQUEST_TO_RECEIVE`
- `ERROR`

Métodos de I2C de alto nivel

Transferencia de arreglo de bytes (sólo en modo *MASTER*):

- `transferByteArray( int_t slaveAddress, const ByteArray_t send, inOut size_t sendSize, bool_t sendRepeatedStart, inOut ByteArray_t receive, inOut size_t dataSize )`
- `transferByteArraySync( int_t slaveAddress, const ByteArray_t send, inOut size_t sendSize, bool_t sendWriteStop, inOut ByteArray_t receive, inOut size_t dataSize Time_t timeout ) : I2cStatus_t`
- `transferByteArrayAsync( int_t slaveAddress, const ByteArray_t send, inOut size_t sendSize, bool_t sendWriteStop, inOut ByteArray_t receive, inOut ByteArray_t receive Callback_t sucess, Callback_t error )`

Si *sendRepeatedStart* es verdadero se genera una condición de *REPEATED_START* y vuelve a enviar la dirección del esclavo, luego del último byte a escribir y antes de comenzar a recibir.

3.3.9. Módulo RTC

El módulo RTC modela un periférico Reloj de Tiempo Real. Este periférico permite mantener la hora y fecha en un sistema.

Propiedades de RTC

- *dateAndTime*, contiene la fecha/hora actual, del tipo *DateAndTime_t*.
- *alarm*, establece la fecha/hora para un evento de alarma, del tipo *DateAndTime_t*.
- *interval*, del tipo *Time_t*, establece el intervalo de tiempo para un evento periódico.

Métodos de RTC

Inicialización y restablecimiento de un periférico RTC:

- `init( DateAndTime_t absTime )`
- `deInit()`

El parámetro *absTime* establece la fecha/hora actual.

La lectura del RTC se realiza con el método *read()*, que retorna la fecha/hora actual. Para cambiar la fecha/hora se utiliza el método *write()*:

- `read() : DateAndTime_t`
- `write( DateAndTime_t absTime )`

Para establecer una interrupción en un RTC ante cierto evento define los métodos:

- `eventCallbackSet( RtcEvent_t evt, Callback_t func )`
- `eventCallbackClear( RtcEvent_t evt )`
- `interrupt( bool_t enable )`

Los eventos posibles son: *ALARM*, que genera una única interrupción cuando se cumple cierto valor absoluto de fecha/hora, y *PERIODIC*, que interrumpe de forma periódica cada cierto lapso de tiempo.

3.3.10. Módulo TIMER

Este módulo modela un periférico contador/temporizador. Este periférico se utiliza para medición de tiempo, generación de retardos, conteo de eventos, generación de señales, etc. Es el periférico que más variedad de modos tiene según la arquitectura. En consecuencia, la documentación de implementación de la biblioteca para cada plataforma, debe indicar qué modos, propiedades, métodos y eventos soporta, de forma clara.

Propiedades de TIMER

Propiedad *mode*, del tipo *TimerMode_t*. Permite establecer el modo de funcionamiento del contador/temporizador, existen los siguientes modos:

- *STOP*. Contador del temporizador detenido.
- *OVERFLOW*. Contar hasta desborde.
- *COMPARE_OUTPUT*. Contar hasta alcanzar un valor de comparación.
- *INPUT_CAPTURE*. Almacenar valor del contador ante eventos de entrada de captura.
- *PWM*. Genera una, o varias salidas digitales moduladas por ancho de pulso, contiene dos sub-modos (del tipo *TimerPwmMode_t*):
  - *FAST_PWM* (se puede establecer también con el valor *EDGE_PWM*). Este modo de PWM alcanza las mayores frecuencias.
  - *PHASE_CORRECT_PWM* (o también *CENTER_ALIGNED_PWM*). Permite generar PWM con frecuencias de la mitad del valor máximo del modo *fast*, pero tiene la ventaja que mantiene la señal generada, siendo

un comportamiento deseado al controlar motores de corriente continua. *PHASE_CORRECT = CENTER_ALIGNED_PWM*

Propiedades del reloj de entrada:

- *clockSource*, del tipo *TimerClockSource_t*. Establece la fuente de reloj del temporizador, la cual se divide por el pre-escalador para obtener el reloj del temporizador. Con valores: *INTERNAL_CPU_CLK*, *INTERNAL_CLK0*, *INTERNAL_CLK01*, *INTERNAL_CLK2*, *EXTERNAL_CLK0*, *EXTERNAL_CLK1*, *EXTERNAL_CLK2* o *EXTERNAL_CLK2*, que se deben definir para cada arquitectura.
- *prescale*, del tipo *TimerPresc_t*. Permite establecer el valor de pre-escalador del reloj del *timer*, es decir, el valor por el cual se dividirá, los valores que puede tomar son: *PRESC_DIV1*, *PRESC_DIV2*, *PRESC_DIV4*, *PRESC_DIV8*, *PRESC_DIV16*, *PRESC_DIV32*, *PRESC_DIV64*, *PRESC_DIV128*, *PRESC_DIV256*, *PRESC_DIV512* o *PRESC_DIV1024*.

Propiedades del registro de conteo:

- *counter*, del tipo *TimerCounter_t*. Representa el registro de conteo del temporizador.
- *counterSize*, del tipo *TimerCounterSize_t*. Define el tamaño del registro de conteo, en general podemos encontrar registros de 8, 16, 24 o 32 bits (valores *COUNTER_BIT8*, *COUNTER_BIT16*, *COUNTER_BIT24*, *COUNTER_BIT32*).
- *counterMode*, del tipo *TimerCounterMode_t*. Establece el modo de contep del *timer*, con valores *COUNT_UP*, *COUNT_DOWN* y *COUNT_UP_DOWN*.

Propiedades que configuran el comportamiento del contador ante eventos (todos del tipo *TimerCounterBehavior_t*):

- *onOverflowEvent* evento de desborde del vlaor de conteo. Permite usar los valores: *RESET_COUNTER*, *INVERT_COUNTER* o *STOP_COUNTER*.
- *onCompareMatchEvent0, ..., onCompareMatchEventN*, evento de comparación exitosa con valor de comparación. Además de los valores ateriores agrega *DO_NOTHING*.
- *onCaptureEvent0, ..., onCaptureEventN*, evento de entrada de captura. Permite los valores anteriores con el agregado de *START_COUNTER* y*RELOAD_AND_START_COUNTER*.

Propiedades de captura y comparación:

- *captureInputPin0, ..., captureInputPinN*, del tipo *TimerCaptureInputPin_t*, representan los pines de entrada de captura. Un cambio de estado en éstos define si ocurre un evento de entrada de captura según la configuración del temporizador.
- *captueValue0, ..., captueValueN*, del tipo *TimerCounter_t*, se utilizan para guardar los valores de conteo del timer cuando ocurre un evento de captura.
- *compareValue0, ..., compareValueN*, del tipo *TimerCounter_t*, en estos se setean los valores que se utilizarán para comparar con el contador del timer.

- *compareOutputPin0, ..., compareOutputPinN*, del tipo *TimeCompareOutputPin_t*, representan los pines de salida de comparación. En base a las configuraciones se pueden generar señales digitales en los mismos ante eventos de comparación o desborde.

Propiedades de PWM:

- *inverted*, del tipo *bool_t*, establece si en modo PWM la salida es invertida.
- *pwmMode*, del tipo TimerPwmMode_t.

Propiedad *location*, del tipo *Location_t*.

Propiedad de cada pin de entrada de captura del temporizador

- *onCaptureEvent0, ..., onCaptureEventN*, del tipo *TimerCaptureInputPinEvent_t*. Configura ante qué eventos del pin de entrada se captura el valor de conteo del temporizador, con valores posibles *RISING_EDGE*, *FALLING_EDGE* o *BOTH_EDGES*.

Propiedad de cada pin de salida de comparación del temporizador

- *onCompareMatchEvent0, ..., onCompareMatchEventN*, del tipo *TimerCompareOutputPinEvent_t*. Configura el comportamiento del pin de salida ante eventos de comparación, con valores posibles *DO_NOTHING*, *TOGGLE_OUTPUT*, *SET_OUTPUT* o *CLEAR_OUTPUT*.

Métodos de TIMER

Inicialización y restablecimiento de un periférico TIMER:

- `init()`
- `init( TimerInit_t config )`
- `deInit()`

Los valores posibles del parámetro *config* se forman con valores de:

- `mode | clockSource | prescale | location`

Todos los valores de configuración son opcionales, con valores por defecto:

- `mode = STOP`
- `clockSource = INTERNAL_CPU_CLK`
- `prescale = PRESC_DIV1`
- `location = LOCATION0`

Métodos para utilizar el temporizador por encuesta o interrupción:

- `isOverflow() : bool_t`
- `isCompareMatch( int_t n ) : bool_t`
- `isCapture( int_t n ) : bool_t`
- `eventCallbackSet( TimerEvent_t evt, Callback_t func )`
- `eventCallbackClear( TimerEvent_t evt )`
- `interrupt( bool_t enable )`

Los posibles eventos del temporizador (valores de *TimerEvent_t*) son:

- `OVERFLOW`
- `COMPARE_MATCH_0, ..., COMPARE_MATCH_N`
- `INPUT_CAPTURE_0, ..., INPUT_CAPTURE_N`
- `ERROR`

Métodos de TIMER de alto nivel

Generar retardo bloqueante:

- `delaySync( Time_t duration )`

Generar retardo no bloqueante:

- `delayAsync( Time_t duration, Callback_t func )`

Modo *ticker*, para generar interrupciones periódicas (base de tiempo):

- `initTicker() // Por defecto: 1 ms`
- `initTicker( Time_t periodicity )`
- `tickerCallbackSet( Callback_t func )`
- `tickerCallbackClear()`
- `tickCounterGet() : Tick_t`
- `tickCounterSet( Tick_t value )`

Tick_t, representa un valor de conteo de un lapso de tiempo adimencional (en ticks).

Modo *PWM*:

- `initPWM() // Por defecto: Fast PWM, 1 kHz, no inv (ni).`
- `initPWM( int_t frequency ) // Por defecto: Fast PWM, ni.`
- `initPWM( int_t frequency, TimerPwmMode_t pwmMode ) // ni.`
- ```
  initPWM( int_t frequency,
           TimerPwmMode_t pwmMode,
           bool_t inverted
         )
  ```

En este modo se definen pines de salida PWM del tipo *PwmPin_t* que tienen los métodos:

- `enable( bool_t enable )`
- `dutyCycleSet( TimerPwmDuty_t value )`
- `dutyCycleGet() : TimerPwmDuty_t`

El valor del tipo *TimerPwmDuty_t*, permite establecer el porcentaje de ciclo de trabajo del PWM.

Modo de temporizador *input capture*:

- ```
  initInputCapture( int_t frequency,
                    TimerCaptureInputPin_t pin,
                    TimerCaptureInputPinEvent_t captureOn
                  )
  ```

- `inputCaptureCallbackSet( TimerCaptureInputPin_t pin, Callback_t func )`
- `inputCaptureCallbackClear( TimerCaptureInputPin_t pin )`

Modo de temporizador *output compare*:

- `initOutputCompare( int_t frequency, int_t compareNumber, TimerCounter_t compareValue )`
- `initOutputCompare( int_t frequency, int_t compareNumber, TimerCounter_t compareValue, TimerCompareOutputPinEvent_t onMatch )`
- `outputCompareCallbackSet( int_t compareNumber, Callback_t func )`
- `outputCompareCallbackClear( int_t compareNumber )`

3.3.11. Módulo CORE

Este módulo modela un núcleo de procesamiento.

Propiedades de CORE

- *clockSource,* del tipo *CoreClockSource_t*. Permite elegir la fuente de reloj.
- *clockFrequency,* del tipo *int_t*. Permite elegir la frecuencia del reloj.

Métodos de CORE

Métodos de inicialización:

- `init()`
- `init( int_t clockFrequency )`
- `init( int_t clockFrequency, CoreClockSource_t clockSource )`

Métodos para control de interrupciones:

- `interrupt( bool_t enable )`
- `peripheralInterrupts( bool_t enable )`
- `enterCritical()`
- `leaveCritical()`

Métodos de control de reseteo:

- `reset()`
- `resetCause() : CoreResetCauseEvent_t`
- `onReset( CoreResetCauseEvent_t evt, Callback_t func )`

Los posibles valores de *CoreResetCauseEvent_t* son:

- `HARD_RESET`
- `SOFT_RESET`
- `DEEPSLEEP_RESET`
- `WDT_RESET`

Métodos de control de modos bajo consumo:

- `init()`
- `sleepUntil( CoreWakeupEvent_t evt )`
- `wakeupReason() : CoreWakeupEvent_t`
- `onWakeup( CoreWakeupEvent_t evt, Callback_t func )`
- `deepSleep()`

Los posibles valores de *CoreWakeupEvent_t* son:

- `INTERRUPT_WAKEUP`
- `PIN_WAKEUP`
- `RTC_WAKEUP`

3.3.12. Módulos SoC y Board

SoC modela el chip completo, mientras que *Board* define la plataforma. Estos módulos contienen todos los mapas de periféricos, que en la implemetación se extraen del modelo de *board*.

Métodos de SOC

- `init()`
- `uniqueId()`

Métodos de BOARD

- `init()`

3.4. Verificación del modelo

Para la verificación del modelo se llevó a cabo una revisión por pares, esto es, se envió la descripción de la API a colegas sometiéndola a sus revisiones.

El proceso de revisión por pares se llevó a cabo de forma iterativa e incremental, es decir, se sostuvieron reuniones iniciales, luego se envió una versión inicial, se discutieron posibles mejoras y cambios, se llevaron a cabo los cambios y se volvieron a someter a revisión. Este proceso se llevó a cabo al menos dos veces por cada módulo con dos revisores distintos.

3.5. Implementación del código C dependiende del hardware

En la implementación para las plataformas del proyecto CIAA, del código dependiente del hardware, que forma parte de la biblioteca sAPI, se utilizaron las bibliotecas de drivers provistas por los fabricantes. En particular, para los microcontroladores de la empresa NXP se utilizó LPCOpen, en la version 3.02 para el LPC4337 y versión 3.04 para el LPC54102. Para el microcontrolador EFM32HG322 de Siicon Labs se utilizó la biblioteca EMLIB versión 5.1.2 (todas en sus últimas versiones al momento de la realización de este trabajo).

Por otra parte, se reutilizó parte del código desarrollado en versiones anteriores de la biblioteca sAPI [] para las plataformas EDU-CIAA-NXP y CIAA-Z3R0 y parte del código de PicoAPI [] para la implementación en la plataforma PicoCIAA.

Una aplicación de embebidos típica que utiliza la biblioteca sAPI, se puede combinar con un sistema operativo de tiempo real, *stracks* y *middelware* resultando una arquitectura de capas de software típica en aplicaciones de sistemas embebidos como se ilustra en la figura 3.7.

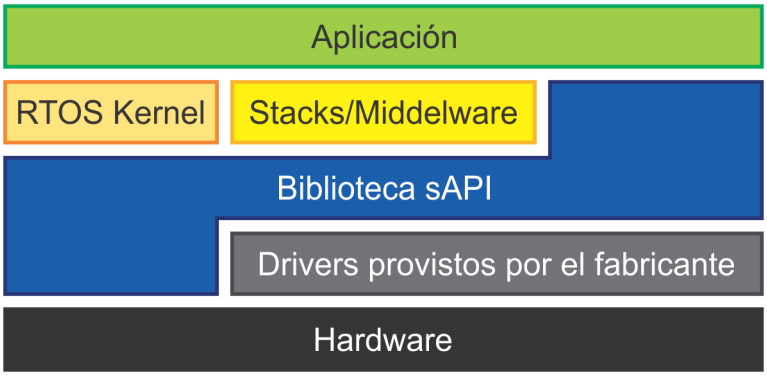

FIGURA 3.7: Arquitectura de una aplicación que utiliza la biblioteca.

3.6. Generador de sAPI

El generador de sAPI toma como entrada el una instancia de *board* y *drivers* específicos, junto con los *módulos sAPI*, y genera como resultado una estructura de archivos y carpetas con la biblioteca sAPI en lenguaje C para una plataforma particular. También permite generación de documentación a partir del mismo modelo.

Para su diseño se estudió una forma sistematizada de generar un módulo de código en lenguaje C. Un módulo de biblioteca en lenguaje C se compone de al menos dos archivos, uno con extensión .h (*header*) que contiene declaraciones y se utiliza para incluir el código del módulo en otro archivo; y otro de extensión .c (*c code*) que contiene las implementaciones.

Para la implementación de los módulos de biblioteca sAPI se utilizan tres archivos por módulo, un archivo *module.h* que contiene la siguiente estructura:

```
Evitar inclusión multiple{
```

```
    Inclusiones de dependencias de funciones publicas
    CPlusPlus (para cuando se usa este módulo desde C++) {
      Macros de definición de constantes publicas.
      Macros "estilo función" publicas.
      Definiciones de tipos de datos públicos.
      Prototipos de funciones publicas.
    }
}
```

Un archivo *module_private.h* con:

```
Inclusiones de dependencias de funciones privadas.
Macros de definición de constantes privadas.
Macros "estilo función" privadas.
Definiciones de tipos de datos privados.
Prototipos de funciones privadas.
```

Y un archivo *module.c* con:

```
Definiciones de variables globales públicas externas (extern).
Definiciones de variables globales públicas.
Definiciones de variables globales privadas (static).
Implementaciones de funciones públicas.
Implementaciones de funciones privadas.
```

En los módulos de sAPI no se utilizan variables globales públicas y variables públicas externas para mantenerlos desacoplados.

Con esta estructura se definen las clases para formar un módulo de biblioteca en lenguaje C.

Además, se definen las clases para generar en lenguaje C todos los tipos de datos, constantes y modificadores de acceso y objetos definidos en los módulos de biblioteca sAPI.

A partir de todas estas clases se formo el generador de código en lenguaje C y documentación.

Para la implementación tanto del del generador de biblioteca sAPI como de las clases que definen el modelo se utilizó el lenguaje *Javascrip* [] y el entorno de ejecución *NodeJS* []. El motivo principal de esta decisión es permitir en un futuro realizar la definición de plataformas y generación de código desde una plataforma web. Estas herramientas permiten generar aplicaciones de escritorio, web o móviles con mínimos esfuerzos.

De esta forma para utilizar el generador se debe primero definir los archivos dependientes de la plataforma y luego ejecutar el generador desde una terminal con NodeJS como se esquematiza en figura 3.8), que resume el producto de este trabajo final.

Tomado los campos de descripción que incluye cada entidad del modelo genera la documentación en lenguaje *Markdown*.

FIGURA 3.8: Generador de la biblioteca sAPI.

Capítulo 4

Ensayos y Resultados

Pruebas funcionales del hardware: La idea de esta sección es explicar cómo se hicieron los ensayos, qué resultados se obtuvieron y analizarlos.

4.1. Ejemplos de utilización

4.2. Testeo Unitario

4.3. Banco de pruebas de hardware

4.4. Integración continua

4.5. Utilización de la biblioteca para la enseñanza de programación de Sistemas Embebidos

4.6. Características de la implementación de sAPI en lenguaje C

Un usuario de la biblioteca sAPI puede elegir el nivel de abstracción deseado a la hora de programar su aplicación. De menor a mayor abstracción tendremos programas:

1. Dependientes del chip: utilizando la API genérica y los nombres de los pines y periféricos del *chip*, definidos por el fabricante (opcionalmente con el agregado de funciones específicas de periféricos de cierto *chip*).
2. Dependientes de la placa: mediante la API genérica y los nombres de la serigrafía de la placa, incluyendo pines y periféricos del chip, así como otros componentes de la placa (por ejemplo, otros chips, conectores, puertos de comunicación).
3. Portables entre placas compatibles: usando la API genérica y los nombres genéricos para todos.
4. Totalmente portables: Mediante la API genérica y es responsabilidad del usuario agregar un archivo de nombres, por ejemplo, remap.h donde elija los nombres de los nombres genéricos para todo lo que va a usar.

VER SI PONGO ALGUNOS EJEMPLOS

4.7. Documentación y difusión

4.7.1. Manual de referencia de la API

4.7.2. Tutoriales de instalación y uso

4.8. Difusión y difusión

Difusión a la comunidad del Proyecto CIAA y Embebidos32

Capítulo 5

Conclusiones

5.1. Trabajo realizado

La idea de esta sección es resaltar cuáles son los principales aportes del trabajo realizado y cómo se podría continuar. Debe ser especialmente breve y concisa. Es buena idea usar un listado para enumerar los logros obtenidos.

5.2. Próximos pasos

Acá se indica cómo se podría continuar el trabajo más adelante.