**1.1**

**Introducción a biblioteca sAPI:**

Comenzar a programar en la placa EDU-CIAA puede tornarse complicado en un principio, ya que si no se está familiarizado con RTOS, cargar, compilar y depurar un programa en osek-os resultara complejo al comienzo.

Para facilitar el manejo de la placa EDU-CIAA a aquellas personas que la utilizaran por primera vez, se puede hacer uso de una librería llamada sAPI, que los adentrara de una forma más amigable a la programación de la placa.

Esta biblioteca implementa una API simple para la programación de microcontroladores.

Una API es una interfaz de programación de aplicaciones, es el conjunto de [subrutinas](https://es.wikipedia.org/wiki/Subrutina), funciones y procedimientos (o [métodos](https://es.wikipedia.org/wiki/M%C3%A9todo_(inform%C3%A1tica)), en la [programación orientada a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos)) que ofrece cierta [biblioteca](https://es.wikipedia.org/wiki/Biblioteca_(programaci%C3%B3n)) para ser utilizado por otro [software](https://es.wikipedia.org/wiki/Software) como una capa de abstracción en la [programación](https://es.wikipedia.org/wiki/Programaci%C3%B3n), generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del software. Son usadas generalmente en las [bibliotecas](https://es.wikipedia.org/wiki/Biblioteca_(inform%C3%A1tica)) de programación.

La motivación para el desarrollo de la biblioteca sAPI surge de la necesidad de manejar los periféricos directamente desde una máquina virtual de Java para el desarrollo de Java sobre la CIAA y corresponde a la parte de bajo nivel de las clases de periféricos en Java que básicamente bindea a funciones escritas en C.

Luego se extendió la misma para facilitar el uso de la EDU-CIAA-NXP a personas no expertas en la arquitectura del LPC4337 facilitando el uso de esta plataforma.

La idea es tener periféricos abstractos y lo más genéricos posibles. Que sea bien independiente de la arquitectura y en lo posible que las funciones sean todas del tipo:

* moduloConfig();
* moduloRead();
* moduloWrite();

Utiliza nombres sencillos al estilo de la biblioteca Wiring, pero en lugar del concepto de pin utiliza el concepto periférico, para lograr una API unificada, sin importar el número de pines que tenga un cierto periférico.

**Módulos incluidos**

* Tipos de datos.
* Mapa de periféricos.
* Plataforma.
* Tick.
* Retardo.
* E/S Digital.
* E/S Analógica.
* Uart.

**Plataformas**

Actualmente disponible para las plataformas:

* EDU-CIAA-NXP (microcontrolador NXP LPC4337).
* CIAA-NXP (microcontrolador NXP LPC4337).

**Capas de software:**

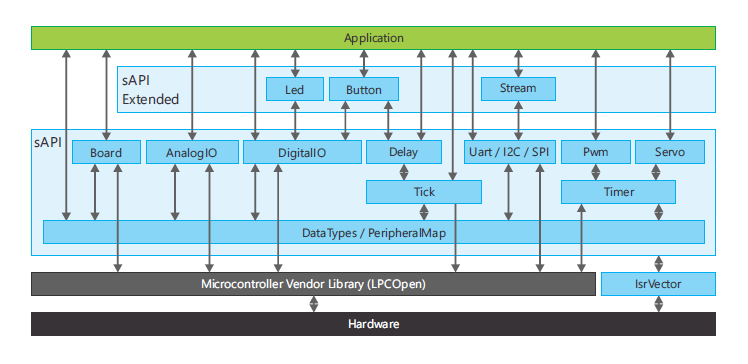


Figura 1.1: Modelo en capas y módulos.

**API de la biblioteca sAPI**

**Módulos:**

**sAPI\_Config**

Contiene configuraciones de la biblioteca.

**sAPI\_DataTypes**

Define las siguientes constantes:

Estados lógicos

**#define FALSE 0**

**#define TRUE !FALSE**

Estados funcionales

**#define ON 1**

**#define OFF 0**

Estados eléctricos

**#define HIGH 1**

**#define LOW 0**

Además define los tipos de datos:

* Booleano bool\_t
* Enteros sin signo uint8\_t, uint16\_t, uint32\_t, uint64\_t
* Enteros con signo int8\_t, int16\_t, int32\_t, int64\_t

**sAPI\_IsrVector**

Contiene la tabla de vectores de interrupción.

**sAPI\_Board**

Contiene la función de configuración para inicialización de la plataforma de hardware:

**void boardConfig( void );**

* Parámetros: **void**
* Retorna: **void**

**sAPI\_PeripheralMap**

Contiene el mapa de periféricos.

**DigitalIO Map**

**DIO0, DIO1, DIO2, DIO3, DIO4, DIO5, DIO6, DIO7,**

**DIO8, DOI9, DIO10, DIO11, DIO12, DIO13, DIO14, DIO15,**

**DIO16, DIO17, DIO18, DIO19, DIO20, DIO21, DIO22, DIO23,**

**DIO24, DIO25, DIO26, DIO27, DIO28, DIO29, DIO30, DIO31,**

**DIO32, DIO33, DIO34, DIO35,**

**TEC1, TEC2, TEC3, TEC4,**

**LED1, LED2, LED3, LEDR, LEDG, LEDB**

**AnalogIO Map**

AI0, AI1, AI2, AO

**Uart Map**

UART\_USB, UART\_232, UART\_485

**sAPI\_DigitalIO**

Manejo de Entradas y Salidas digitales.

**Configuración inicial y modo de una entrada o salida**

**bool\_t digitalConfig( int8\_t pin, int8\_t config);**

* Parámetros: **int8\_t pin, int8\_t config**
* Retorna: **bool\_t** TRUE si la configuración es correcta.

**Configuraciones:**

INITIALIZE

INPUT, INPUT\_PULLUP, INPUT\_PULLDOWN, INPUT\_REPEATER

OUTPUT

**Lectura de Entrada digital**

**bool\_t digitalRead( int8\_t pin );**

* Parámetros: **int8\_t pin**
* Retorna: **bool\_t** valor de la entrada digital.

**Escritura de Salida Digital**

**bool\_t digitalWrite( int8\_t pin, bool\_t value );**

* Parámetros: **int8\_t pin, bool\_t value**
* Retorna: **bool\_t** FALSE en caso de errores.

**sAPI\_Tick**

**Configuración de interrupción periódica**

**bool\_t tickConfig( tick\_t tickRateMSvalue );**

* Parámetros: **tick\_t tickRateMSvalue**
* Retorna: **bool\_t** FALSE en caso de errores.

Configura una interrupción periódica de temporizador cada tickRateMSvalue milisegundos para utilizar de base de tiempo del sistema. Una vez ejecutada esta función se dice que ocurre un tick del sistema cada tickRateMSvalue milisegundos.

La tasa de ticks en ms, tickRateMS, es un parámetro con rango de 1 a 50 ms.

**Leer la variable del conteo actual de ticks**

**tick\_t tickRead( void );**

* Parámetros: **void**
* Retorna: **tick\_t**

La variable del conteo actual de ticks se incrementa en 1 cada tickRateMSvalue milisegundos.

**Escribir la variable del conteo actual de ticks**

**void tickWrite( tick\_t ticks );**

* Parámetros: **tick\_t**
* Retorna: **void**

Se utiliza si se necesita cambiar el valor del contador, por ejemplo, para resetearlo.

**Función que se ejecuta en cada vez que ocurre un tick**

**void tickHook( void );**

* Parámetros: **void**
* Retorna: **void**

Esta función debe declararla el usuario en su programa. Para poder utilizarla, se debe configurar en **sAPI\_Config.h**

**#define SAPI\_USE\_TICK\_HOOK TRUE**

De esta manera se le da aviso a la biblioteca que se encargará de ejecutarla automáticamente a cada tick.

**sAPI\_Delay**

Para utilizar los retardos (con excepción del retardo inexacto) se debe configurar el Tick.

Todos los tiempos de parámetros están en milisegundos.

Define la constante #define INACCURATE\_TO\_MS 20400 y contiene las funciones:

**Retardo inexacto bloqueante**

**void delayInaccurate(tick\_t delay\_ms);**

* Parámetros: **tick\_t delay\_ms**
* Retorna: **void**

Utiliza un bloque for bloqueante que tiene una constante calculada a ojo (INACCURATE\_TO\_MS) para perder muchos ciclos de reloj y lograr hacer un retado.

**Retardo bloqueante**

**void delay (tick\_t time);**

* Parámetros: **tick\_t time**
* Retorna: **void**

Utiliza el conteo de ticks para determinar el tiempo transcurrido resultando en un retardo exacto. Es bloqueante pues se queda en un bucle while hasta que se cuentan los ticks necesarios para lograr el tiempo especificado.

**Retardo no bloqueante**

Este tipo de retardo permite realizar otras tareas mientras se ejecuta ya que simplemente se chequea si el tiempo de retardo se ha arribado en lugar de quedarse bloqueado esperando a que se complete el tiempo como en los casos anteriores.

Define el tipo de datos estructurado **delay\_t**

**Contiene las funciones:**

**void delayConfig( delay\_t \* delay, tick\_t duration );**

* Parámetros: **delay\_t \* delay, tick\_t duration**
* Retorna: **void**

**bool\_t delayRead( delay\_t \* delay );**

* Parámetros: **delay\_t \* delay**
* Retorna: **bool\_t** TRUE cuando el **delay** se cumplió, FALSE en caso contrario.

**void delayWrite( delay\_t \* delay, tick\_t duration );**

* Parámetros: **delay\_t \* delay, tick\_t duration**
* Retorna: **void**

**Uso:**

Se utiliza declarando una variable de estructura del tipo delay\_t, por ejemplo:

**delay\_t myDelay;**

Luego, se configura inicialmente pasando como parámetro la variable recién declarada

**delayConfig( &myDelay, 500 );**

Se detecta con un bloque if si se cumplió el delay leyéndolo con

**delayRead( &myDelay );**

La primera vez que se ejecuta delayRead activa el mismo. delayRead devuelve TRUE cuando se completó y se vuelve a relanzar automáticamente.

Con **delayWrite( &myDelay, 1000 );** se puede cambiar la duración de un delay en tiempo de ejecución.

**Archivos que componen la biblioteca**

**src (.c):**

* sAPI\_AnalogIO.c
* sAPI\_Board.c
* sAPI\_Delay.c
* sAPI\_DigitalIO.c
* sAPI\_IsrVector.c
* sAPI\_Tick.c
* sAPI\_Uart.c

**inc (.h):**

* sAPI.h
* sAPI\_AnalogIO.h
* sAPI\_Board.h
* sAPI\_Con􀀏g.h
* sAPI\_DataTypes.h
* sAPI\_Delay.h
* sAPI\_DigitalIO.h
* sAPI\_IsrVector.h
* sAPI\_PeripheralMap.h
* sAPI\_Tick.h
* sAPI\_Uart.h

**Utilización de la biblioteca sAPI:**

EDU-CIAA-NXP

Para poder hacer uso de la biblioteca en la placa EDU-CIAA se puede descargar la versión más reciente de la misma desde:

<https://github.com/epernia/sAPI>

Luego de haber descargado la versión actual de sAPI se procede a copiar la carpeta sapi\_bm y pegar en C:/CIAA/Firmware/modules.

El paso siguiente es copiar el proyecto bare\_metal de C:/CIAA/Firmware/examples y renombrarlo a, por ejemplo “miProyectoConsAPI” y se debe modificar el makefile del mismo dentro de la carpeta mak y cambiar la última línea de:

**MODS += externals$(DS)drivers**

por:

**MODS += modules$(DS)sapi\_bm**

Para poder hacer uso de la biblioteca se debe colocar al inicio de cada archivo c:

**#include “sAPI.h”**

Y Ya no es necesario incluir chip.h ya que la propia biblioteca

sAPI lo incluye.

La sAPI también maneja el vector de interrupción, es por esto que es necesario eliminar el archivo vector.c que trae el ejemplo bare\_metal.

Por ultimo una vez que se quiera compilar y depurar un programa mediante el software eclipse se debe colocar sobre la carpeta firmware y mediante click derecho ubicarse en “Propiedades” -> “C/C++ build” -> “behaviour” y luego en la rama clean cambiar de “clean\_generate” a solamente “clean” como se puede apreciar en la siguiente figura:

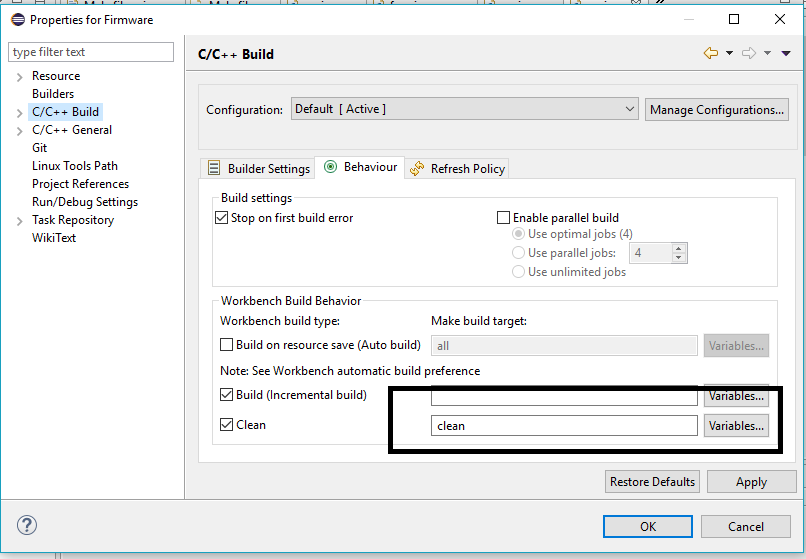


Figura 2.1: Cambio de “clean\_generate” a “clean”.