ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Modelo analógico y digital en SystemC-AMS de la placa Arduino Mega 2560
(Analog & Digital SystemC-AMS model of an Arduino Mega 2560 board)

Para acceder al Título de

Graduado en
Ingeniería de Tecnologías de Telecomunicación

Autor: Daniel Lastra Lamarca

Octubre - 2015

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Daniel Lastra Lamarca
Director del TFG: Víctor Fernández Solórzano

Título: "Modelo analógico y digital en SystemC-AMS de la placa Arduino Mega 2560"
Title: "Analog & Digital SystemC-AMS model of an Arduino Mega 2560 board"

Presentado a examen el día:

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente: Fernández Solórzano, Víctor
Secretario: Posadas Cobo, Héctor
Vocal: Vía Rodríguez, Javier

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFG (sólo si es distinto del Secretario)
Vº Bº del Subdirector	Trabajo Fin de Grado Nº (a asignar por Secretaría)

Agradecimientos

A Víctor por ofrecerme la oportunidad de realizar este trabajo, así como toda la atención que me ha prestado durante el mismo.

A mi familia por apoyarme siempre que lo he necesitado.

A mis amigos y compañeros, especialmente a Sheila, por todos los buenos ratos que hemos pasado estos años.

Analog & Digital SystemC-AMS model of an Arduino Mega 2560 board

Palabras clave:

#Arduino #ATMEL #2560 #SystemC #SystemC-AMS #Simulador #Modelado #Vippe

1 - Introducción

1.1 - Contexto y motivación

Las nuevas aplicaciones que se empiezan a desarrollar hoy en día requieren componentes micro/nano electrónicos que interactúen estrechamente con su entorno en diferentes dominios físicos (óptico, mecánico, acústico…). Estos sistemas vienen siendo llamados sistemas ciberfísicos o sistemas multidominio. El reto principal consiste en especificar, dimensionar y verificar estos sistemas multidominio, para evitar errores innecesarios y rediseños que afecten a la calidad del producto y al tiempo de mercado.

Generalmente, el diseño y la verificación de estos sistemas multidominio suele ser un proceso iterativo, en el que cada parte correspondiente a un dominio físico se desarrolla de manera independiente y se combina junto al resto en una etapa final para crear el producto.

Errores como malfuncionamiento o incumplimiento de las especificaciones iniciales son muy comunes. Estos problemas desembocan en retrasos y cambios en los diseños, lo que lastra la introducción del producto al mercado. El principal problema es que falta una representación del sistema global, que incluya todos los dominios con los que interactúa, en las etapas tempranas del desarrollo.

En el mercado existe, además, un notable crecimiento en la demanda de sistemas que interactúan con el entorno en varios ámbitos profesionales (industria, medicina, consumo…). Durante el periodo 2011-2014, el crecimiento en la cuota de mercado de estos sistemas ha crecido un 19% en Europa.

Debido a todos los problemas existentes y a las oportunidades disponibles en el mercado, ha surgido una iniciativa europea que pretende desarrollar e implementar una metodología de diseño y unas herramientas unificadas para solucionar estos problemas de diseño y verificación de sistemas multidominio.

El trabajo aquí presentado se enmarca dentro de esta iniciativa, desarrollando un modelo virtual de un microcontrolador. Este modelo es exclusivamente electrónico, pero un gran número de sistemas multidominio cuentan con microcontroladores para gestionar los diversos procesos llevados a cabo al interactuar con el entorno.

Uno de los puntos clave de la iniciativa europea mencionada consiste en estandarizar una extensión del lenguaje SystemC, SystemC-AMS+MDVP (Multi-domain Virtual Prototypes), para el diseño de sistemas multidominio. En este proyecto se ha seguido esta recomendación, describiendo el microcontrolador en SystemC-AMS y habilitándolo para conectarse con cualquier otro sistema descrito en este mismo lenguaje.

1.2 - Objetivos

El objetivo global del proyecto consiste en la elaboración de un modelo virtual en SystemC-AMS de un microcontrolador para su simulación. En concreto, el microcontrolador a modelar es el ATmega 2560 que montan las placas Arduino Mega 2560. El modelo contará con las entradas y salidas, tanto digitales como analógicas, con que cuenta el microcontrolador, así como interrupciones.

Durante la simulación, el modelo será capaz de recibir instrucciones en código fuente de aplicación en lenguaje Arduino, ejecutarlas correctamente e interactuar temporalmente junto con cualquier otro modelo SystemC-AMS con el que se simule.

A modo de resumen, éstos son los objetivos a cumplir:

- Modelo en SystemC-AMS del microcontrolador ATmega 2560 montado sobre la placa Arduino Mega 2560. El modelo cuenta con entradas/salidas digitales y analógicas e interrupciones.

- Las principales funciones de la librería Arduino se adaptarán para funcionar durante la simulación, sin tener que realizar cambios (código de aplicación en lenguaje Arduino) en los ‘sketches’ del usuario.

1.2.1 - Detallado de objetivos

El Grupo de Ingeniería Microelectrónica de la UC dispone de un simulador de plataformas multiprocesadoras: Vippe. Dicho simulador (el cual se analizará en detalle más adelante), está preparado para emular la ejecución de aplicaciones en código C++. Los 'sketches' (código fuente de aplicación del usuario) Arduino, pese a estar basados en C++, tienen una serie de funciones específicas que habrá que adaptar para que Vippe las pueda 'entender' durante la simulación.

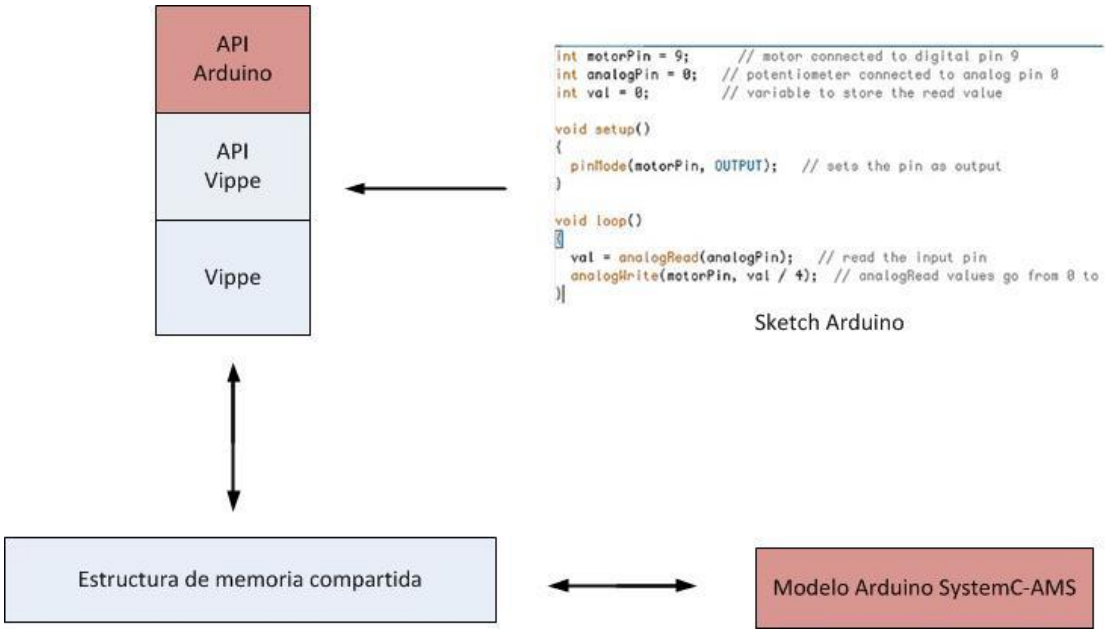

Figura 1 - Diagrama general del proyecto

En la Figura 1 se muestra un diagrama, muy general, del proyecto. Las cajas que aparecen en tono rojo son los elementos nuevos que se han desarrollado durante el mismo.

Durante la simulación, Vippe lee el código fuente contenido en el 'sketch' Arduino del usuario. Como no está preparado para ello, es necesario añadirle una pequeña API Arduino, creada a partir de las primitivas de la propia API de Vippe, para que interprete este código fuente correctamente.

A medida que Vippe lee estas instrucciones, emula su ejecución en el tiempo y, dicha emulación, se la comunica al modelo SystemC del Arduino a través de una estructura de memoria compartida. El modelo SystemC ejecuta lo que debe en función de la instrucción (escribir un bit en una salida digital, leer un valor de una entrada analógica…) y comunica a Vippe, a través de esta estructura también, que está listo para recibir otra orden.

1.3 - Organización del documento

El documento se encuentra dividido en varios capítulos, siendo éste el primero de ellos, como introducción.

El capítulo que se encuentra a continuación, el de conceptos previos, trata sobre diversas herramientas empleadas, como el simulador Vippe, analizando en detalle su funcionamiento y qué técnica de simulación emplea, o el lenguaje SystemC-AMS, describiendo sus características principales. También se describe en detalle la placa Arduino Mega 2560 y su microcontrolador.

El capítulo 3 contiene toda la información relativa a la adaptación de la librería Arduino de cara a la simulación con Vippe. Se muestran todos los ficheros fuente adaptados, junto con las funciones que contienen.

El capítulo 4 explica el funcionamiento del modelo SystemC-AMS. Se analiza en detalle los módulos que lo forman y las funciones que intervienen durante su ejecución. También se comenta cómo se realiza la sincronización con Vippe y la información generada tras la simulación.

Tras esto, en el capítulo 5, se ilustra un posible caso de uso, simulando el modelo Arduino junto con el modelo de un sistema microfluídico. Se comentan las pruebas realizadas y se muestra la información que se obtiene al terminar su simulación.

En los capítulos 6 y 7 se comentan las conclusiones que se han alcanzado y se recogen las principales fuentes de información consultadas, respectivamente.

2 - Conceptos previos

En este capítulo se analizarán las herramientas y lenguajes empleados en el proyecto: el simulador Vippe, Arduino como lenguaje y plataforma y SystemC y su extensión AMS.

2.1 - Vippe

Vippe (Virtual Parallel Platform for Performance Analysis) es una herramienta desarrollada por el Grupo de Ingeniería Microelectrónica (GIM) en la Universidad de Cantabria y se encuadra dentro de la simulación nativa.

2.1.1 - Simulación nativa

Las técnicas de simulación nativas han sido propuestas para generar plataformas virtuales al comienzo del proceso de diseño, reduciendo la dificultad de portar el diseño a otras plataformas. Esta técnica consiste en analizar el código fuente del usuario e instrumentarlo con código adicional durante la compilación. Este nuevo código provee de mucha información (tiempo de ejecución, número de accesos a memoria…) sobre la plataforma destino (‘target’) durante su ejecución en la plataforma anfitrión (‘host’).

Los simuladores basados en técnicas de simulación nativa cuentan con unos tiempos de simulación muy cortos, ofreciendo resultados lo suficientemente precisos. La creación de las plataformas virtuales se hace de manera rápida y sencilla, permitiendo explorar diversas alternativas, siendo ideal en etapas tempranas de desarrollo.

2.1.2 - Características

Vippe es capaz de simular, en una plataforma de usuario, cualquier código C++ a ser compilado y ejecutado sobre otra plataforma multiprocesadora genérica y, obtener como resultado, una estimación temporal y energética del código ejecutándose sobre dicha plataforma destino.

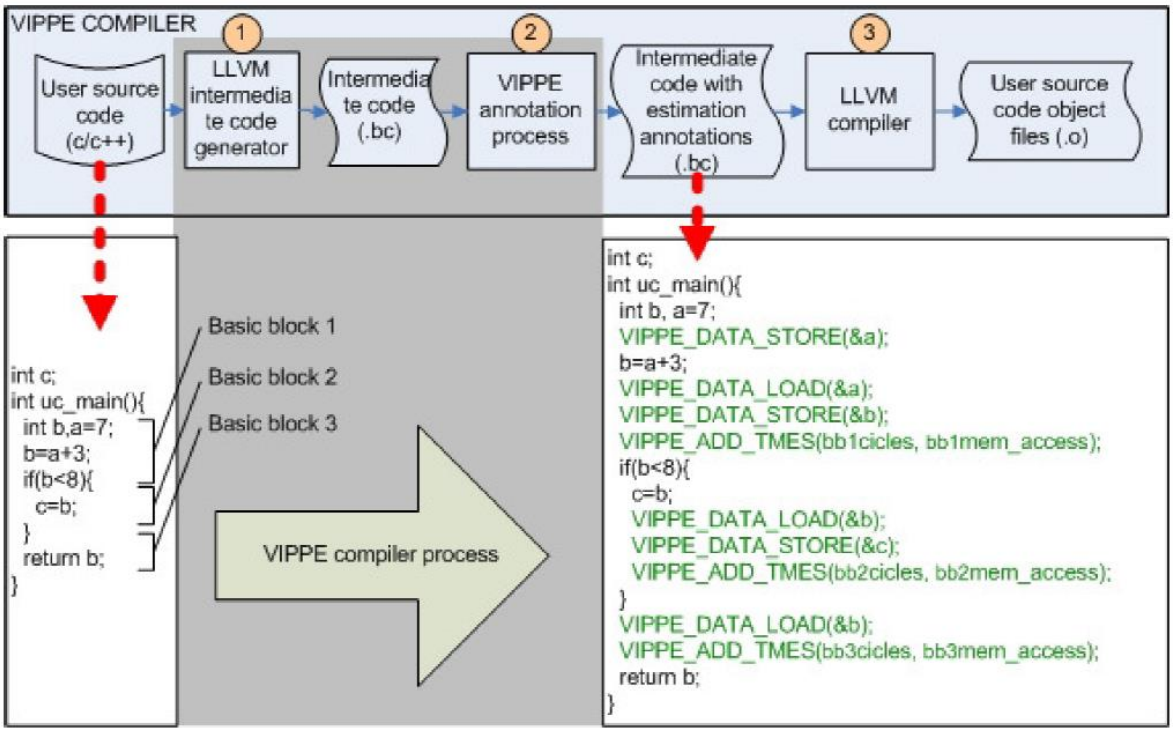

Figura 2 - Instrumentación del código

El código original se amplía introduciendo unas anotaciones, creando un código intermedio, como se ve en la Figura 2. Estas anotaciones se introducen entre bloques básicos del código. Se consideran bloques básicos a aquellas secciones del código que se ejecutan sin ninguna sentencia de control (condicionales y bucles). Al compilar y ejecutar el código instrumentado, se obtienen las estimaciones temporales y de consumo.

2.1.3 - API Vippe

Para utilizar los servicios que ofrece el 'kernel' de Vippe, se emplea su API, que consiste en un reducido número de primitivas. Este conjunto de primitivas constituye un modelo que hace posible implementar APIs de mayor nivel. En este proyecto se ha creado una API Arduino basada en la API de Vippe, que se analizará más adelante.

Dado que no se han empleado todas las primitivas con las que cuenta la API de Vippe, a continuación se detallarán solo las que sí se han usado en el proyecto.

2.1.3.1 - Manejo de Semáforos

2.1.3.1.1 - semaphore_wait

Esta función recibe un par de argumentos, el identificador del semáforo y un tiempo máximo de espera. Se encarga de bloquear el semáforo cuyo identificador sea el del argumento. Dependiendo del valor del tiempo máximo de espera, pueden suceder dos cosas:

- Si es igual a cero, la función no retornará hasta que consiga bloquear el semáforo o sea interrumpida por una señal.

- En cualquier otro caso, la función retornará si consigue bloquear el semáforo, si es interrumpida por una señal o si se alcanza el tiempo máximo de espera.

2.1.3.1.2 - semaphore_post

Desbloquea un semáforo con el identificador que recibe como parámetro.

2.1.3.2 - Manejo de Tiempos

2.1.3.2.1 - time_real_watch

Devuelve el tiempo de ejecución del sistema en nanosegundos, unidad empleada por Vippe.

2.1.3.2.2 - time_user_watch

Devuelve el tiempo de ejecución del ‘thread’ de usuario en nanosegundos, unidad empleada por Vippe.

2.1.3.3 - Funciones específicas para comunicación con SystemC

2.1.3.3.1 - vippe_IRQ

Notifica a Vippe qué función atiende una determinada interrupción y qué prioridad tiene. Si en el argumento del nombre de la función recibe un ‘NULL’, deshabilita la interrupción.

Vippe se encarga de administrar las interrupciones según se activan, dando paso siempre a la de mayor prioridad. En el caso en que se activen varias interrupciones de manera simultánea, Vippe da paso a la de mayor prioridad y desecha el resto. Si mientras se está atendiendo una interrupción, se activa otra de mayor prioridad, se pausa la rutina de atención actual y se atiende la nueva. Tras acabar, se reanuda la primera rutina.

2.1.3.3.2 - send_IRQ_to_Vippe

El programa SystemC notifica a Vippe que se ha producido una interrupción. Como parámetros, se envían el tiempo en que se ha producido y la prioridad de la interrupción.

2.1.3.3.3 - dig_write

Indica a Vippe que se va a realizar una operación de escritura sobre una dirección de memoria. Como parámetros recibe la propia dirección de memoria y el valor entero que se quiere escribir. La función se encarga de comunicar a un programa SystemC, a través de una estructura de memoria compartida, que se ha producido una escritura.

2.1.3.3.4 - dig_read

Notifica a Vippe una operación de lectura sobre una dirección de memoria. La función se encarga de comunicar a un programa SystemC, a través de una estructura de memoria compartida, que se ha producido una lectura.

2.1.4 - Comunicación con SystemC

Las primitivas desarrolladas para la comunicación con SystemC (apartado 2.1.3.3) de la API de Arduino, interactúan con un programa SystemC a través de una estructura de memoria compartida, tal y como se muestra en la Figura 3.

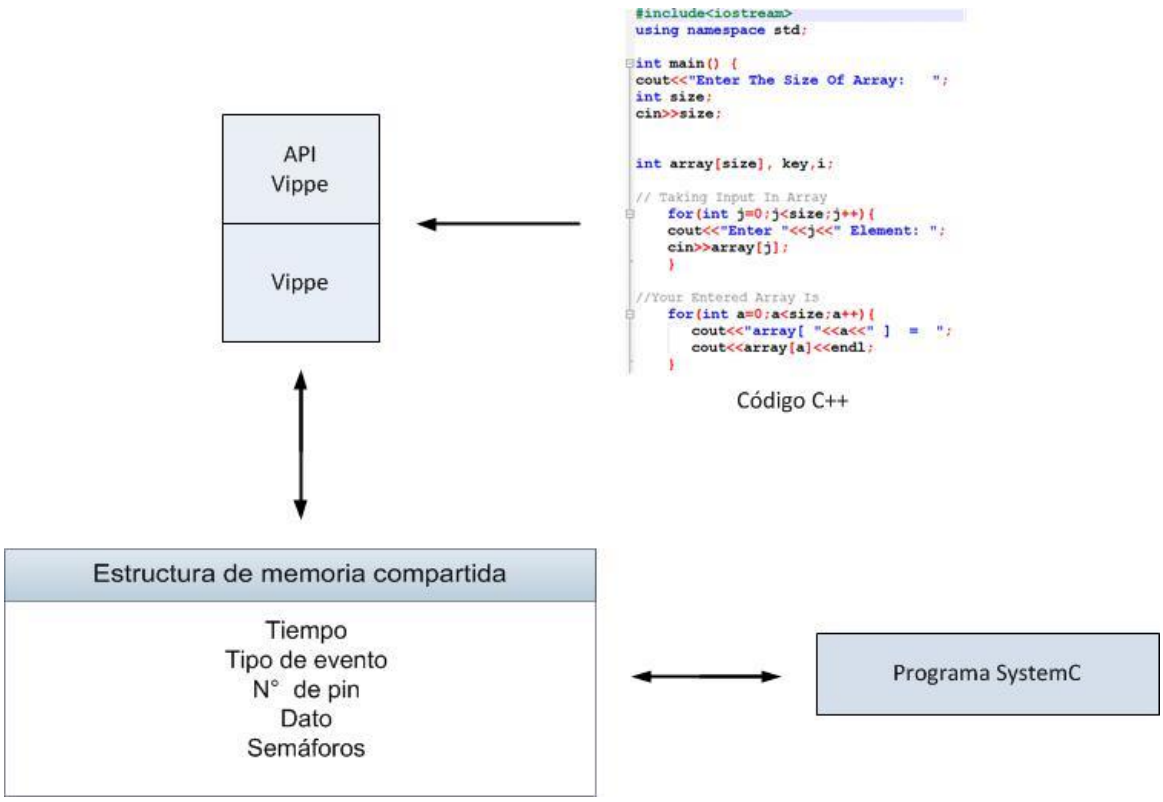

Figura 3 - Comunicación con SystemC

Dentro de la estructura se encuentran una serie de variables para controlar el ritmo de simulación y comunicar al programa SystemC los diferentes eventos que ocurren. A continuación, se detallan dichas variables:

- long long int *next_time*: Indica al programa SystemC hasta qué tiempo tiene que avanzar.

- int *type*: Indica qué es lo que ha ocurrido en el tiempo *next_time*. Si se recibe un '0', significa que no ha pasado nada en este tiempo, un '3' indica que se ha producido una lectura (*dig_read*) y un '4' una escritura (*dig_write*). Además, el programa SystemC manda un '5' a Vippe cuando ha acabado o, como parte del protocolo para notificar una interrupción, un número a partir de '6' cuando se ha producido una, en función de su prioridad.

- void **addr*: La dirección de memoria en la que se ha producido la lectura o la escritura. En este proyecto, se utilizarán las direcciones de memoria como identificadores de los números de pin de la placa.

- int *data*: El dato que se ha escrito (*dig_write*), o el que Vippe necesita leer (*dig_read*).

- sem_t *semaphorescv*: Semáforo que bloquea a Vippe mientras el programa SystemC se ejecuta.

- sem_t *semaphorevsc*: Semáforo que bloquea al programa SystemC mientras Vippe se ejecuta.

Este par de semáforos controlan el avance temporal del programa SystemC y evitan que se sobreescriban datos en la estructura.

2.1.5 - Definición de la plataforma

Vippe emplea un conjunto de archivos '.xml' donde el usuario define la plataforma que quiere simular. En estos ficheros se puede definir:

- La arquitectura de la plataforma: Se pueden definir el número de núcleos de la CPU, el tipo y tamaño de las memorias o de las cachés, el número de buses…entre otras cosas.

- Las conexiones o el mapeado de los elementos 'hardware'.

- El sistema operativo, si lo hubiese.

La plataforma empleada en este proyecto es bastante simple. No se define ningún sistema operativo, ya que no hay. Solo cuenta con un núcleo de procesado a una frecuencia de 16 MHz, con cachés de datos e instrucciones y conectado mediante un bus, a una memoria.

A modo de ejemplo, en la Figura 4 se muestra como queda el fichero que describe la arquitectura de la plataforma.

```
<Description xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" name="platform">
<HW_Platform>
    <HW_Components>
        <HW_Component category="dcache" name="DataCache" mem_size="8KB" associativity="1" blockSize="1" wordSize="1" writePolicy="writeBack"/>
        <HW_Component category="icache" name="InstructionCache" mem_size="256KB" associativity="1" blockSize="1" wordSize="1" writePolicy="writeBack"/>
        <HW_Component category="bus" name="Bus1" width="1B" bandWidth="100Mb/s" burstSize="8"/>
        <HW_Component category="memory" type="ROM" name="ROM" mem_size="4KB" mem_latency="1ns"/>
        <HW_Component category="processor" name="Processor" frequency="16MHz"/>
    </HW_Components>

    <HW_Architecture>
        <HW_Instance component="Processor" name="proc1">
            <HW_Instance component="DataCache" name="DataCache_proc1" />
            <HW_Instance component="InstructionCache" name="InstructionCache_proc1" />
        </HW_Instance>
        <HW_Instance component="ROM" name="rom1"/>
        <HW_Instance component="Bus1" name="bus1">
            <HW_Connection name="Port1_proc1" instance="proc1" channel="bus1"/>
            <HW_Connection name="Port2_rom1" instance="rom1" channel="bus1"/>
        </HW_Instance>
    </HW_Architecture>
</HW_Platform>
</Description>
```

Figura 4 - HWArquitecturePlatform.xml - Arquitectura

2.2 - Arduino

Arduino es una plataforma de prototipado ‘open-source’, basada en una placa con un microcontrolador. Las placas Arduino son capaces de leer y escribir puertos (luz en un sensor, presión sobre un botón…), activar un motor, encender un LED… todo ello definido en un conjunto de instrucciones programadas a través de su entorno de desarrollo.

Figura 5 - Arduino

El ‘hardware’ de las placas consiste normalmente en un microcontrolador AVR y puertos de entrada/salida. Se suelen emplear microcontroladores como el ATmega328, el ATmega2560, o el ATmega1280, entre otros, debido a su sencillez y bajo coste.

El ‘software’ consiste en un entorno de desarrollo que implementa el lenguaje Arduino y el cargador de arranque (‘bootloader’) que se ejecuta en la placa.

2.2.1 - Lenguaje Arduino

Todos los programas escritos para ejecutarse en una placa Arduino, se escriben con este lenguaje, que no es más que C++ junto con una librería Arduino. Gracias a esta librería, programar para la placa se convierte en una tarea sencilla, ya que se consigue un nivel de abstracción más alto. Por tanto, la sintaxis de los programas es la misma que en C++, y todos los operadores y estructuras de este lenguaje se mantienen.

Los programas Arduino reciben el nombre de 'sketches' y cuentan con una serie de particularidades:

- Un programa escrito en C++ comienza a ejecutarse en la función *main*. En Arduino la función *main* se encuentra dentro de la librería (el usuario no puede acceder a ella) y llama a un par de funciones, *setup* y *loop*, que edita el usuario.

- Es necesario incluir la cabecera de Arduino, *Arduino.h*, en los 'sketches'.

- Los 'sketches' tienen una extensión '.ino' en lugar de '.cpp'.

Debido a todas estas particularidades del lenguaje Arduino, se ha generado una nueva librería Arduino, habilitada para la simulación. Sobre esta nueva librería y su contenido se hablará en el capítulo 3.

2.2.2 - Arduino Mega 2560

De todo el catálogo Arduino, la placa empleada en este proyecto es la Arduino Mega 2560. Cuenta con un microcontrolador ATmega2560 que funciona a 16 MHz. La placa dispone de 54 entradas/salidas digitales, de las cuales 15 se pueden emplear como salidas PWM (Pulse Width Modulation) y 16 entradas analógicas. Además, tiene una conexión USB y 4 puertos UART.

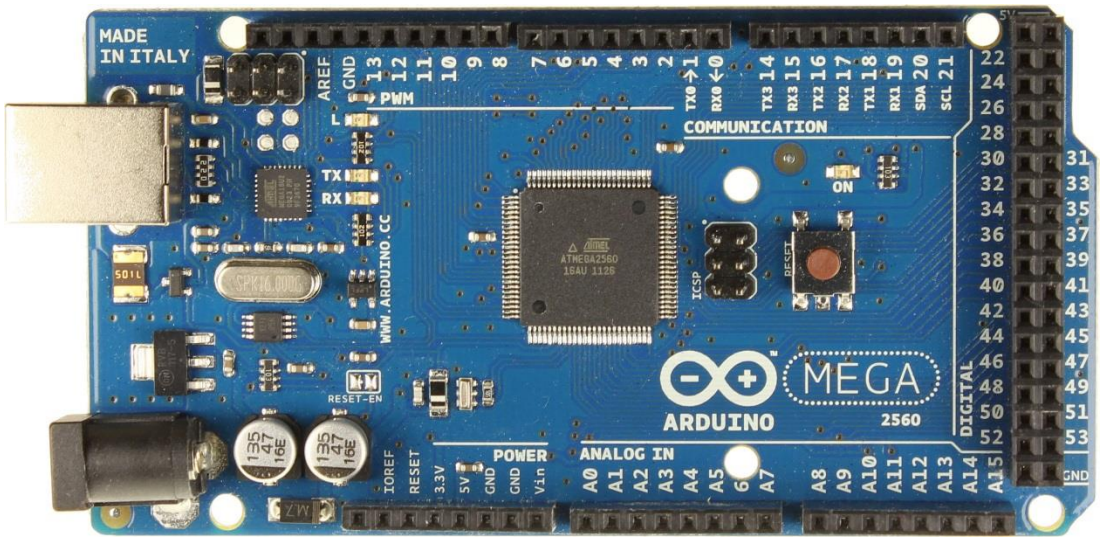

Figura 6 - Arduino Mega 2560

La dirección (entrada o salida) de los pines digitales así como la decisión de cuales son configurados como PWM se efectúa en el código 'sketch' del usuario.

Cada uno de los 54 pines digitales se puede emplear como entrada o salida usando las funciones *digitalWrite*, *digitalRead* y *pinMode* de la librería Arduino. Los pines operan a 5 V y cada uno de ellos recibe o provee 20 mA, pudiendo llegar a 40 mA como máximo.

Varios de estos pines, además, pueden tener funciones específicas como las siguientes:

- Interrupciones externas: Se pueden configurar los pines 2, 3, 21, 20, 19 y 18 (de mayor a menor prioridad) para activar una interrupción si se encuentran a nivel bajo, si hay un flanco de subida, si hay un flanco de bajada o si lo hay de cualquiera de los dos. Otras placas Arduino cuentan con un par más de opciones.

- PWM: Los pines 2-13 y 44-46 pueden generar una señal PWM empleando la función *analogWrite*. Las señales PWM son señales binarias que conmutan con un determinado periodo de tiempo entre ambos niveles. La frecuencia de esta señal es de 490 Hz. El pin sigue generando la misma señal hasta que se llama otra vez a *analogWrite*, *digitalRead* o *digitalWrite* sobre el mismo pin.

La placa cuenta con 16 entradas analógicas, cada una con una resolución de 10 bits (0-1023). Por defecto, el valor analógico de entrada se mide desde 0 V hasta 5 V, pero se puede alterar empleando la función *analogReference* y el pin AREF, dependiendo si se quiere usar una referencia interna o externa.

A continuación, en la Figura 7, se muestra como están mapeados los pines del microcontrolador ATmega 2560 en la placa Arduino Mega 2560.

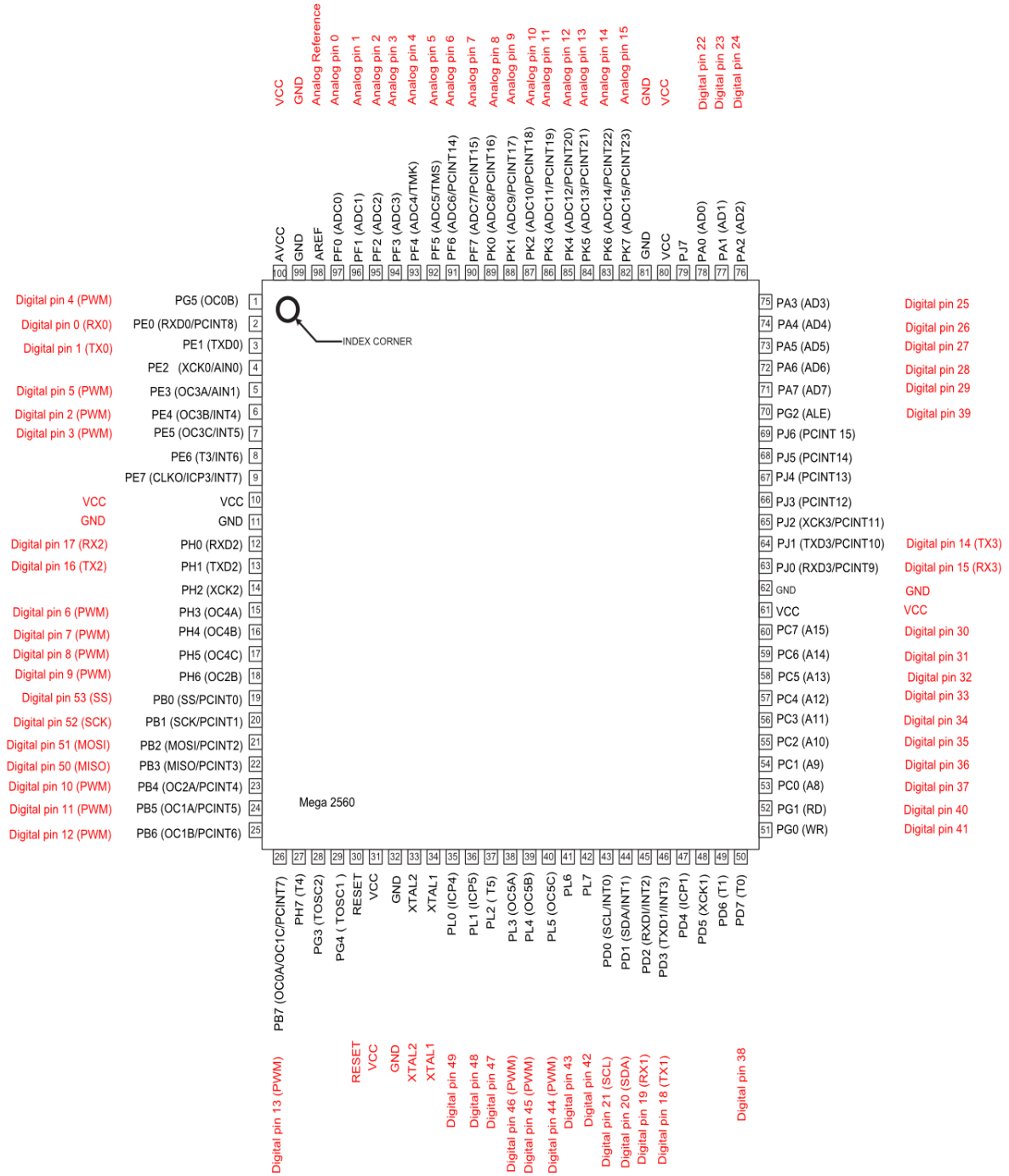

Figura 7 - Mapeo de pines ATmega 2560 en la placa Arduino

2.2.2.1 - Microcontrolador ATmega 2560

El ATmega 2560 es un microcontrolador de bajo consumo CMOS de 8 bits basado en la arquitecura RISC de AVR. Puede ejecutar grandes instrucciones en un solo ciclo de reloj, consiguiendo 'thoughputs' de 1 MIPS por MHz, permitiendo al diseñador optimizar los consumos de potencia en función de la velocidad de procesado. Funciona a 16 MHz y su rango de operación está entre 4.5 V y 5.5 V.

Algunas de las características con las que cuenta el microcontrolador son las siguientes:

- Memoria 'Flash' de 256 Kbytes.
- EEPROM de 4 Kbytes.
- SRAM de 8 Kbytes.
- 86 líneas de entrada/salida de propósito general.
- 32 registros de propósito general.
- 12 canales PWM con 16 bits de resolución cada uno.
- Convertidor A/D con 16 canales y de 10 bits de resolución cada uno.

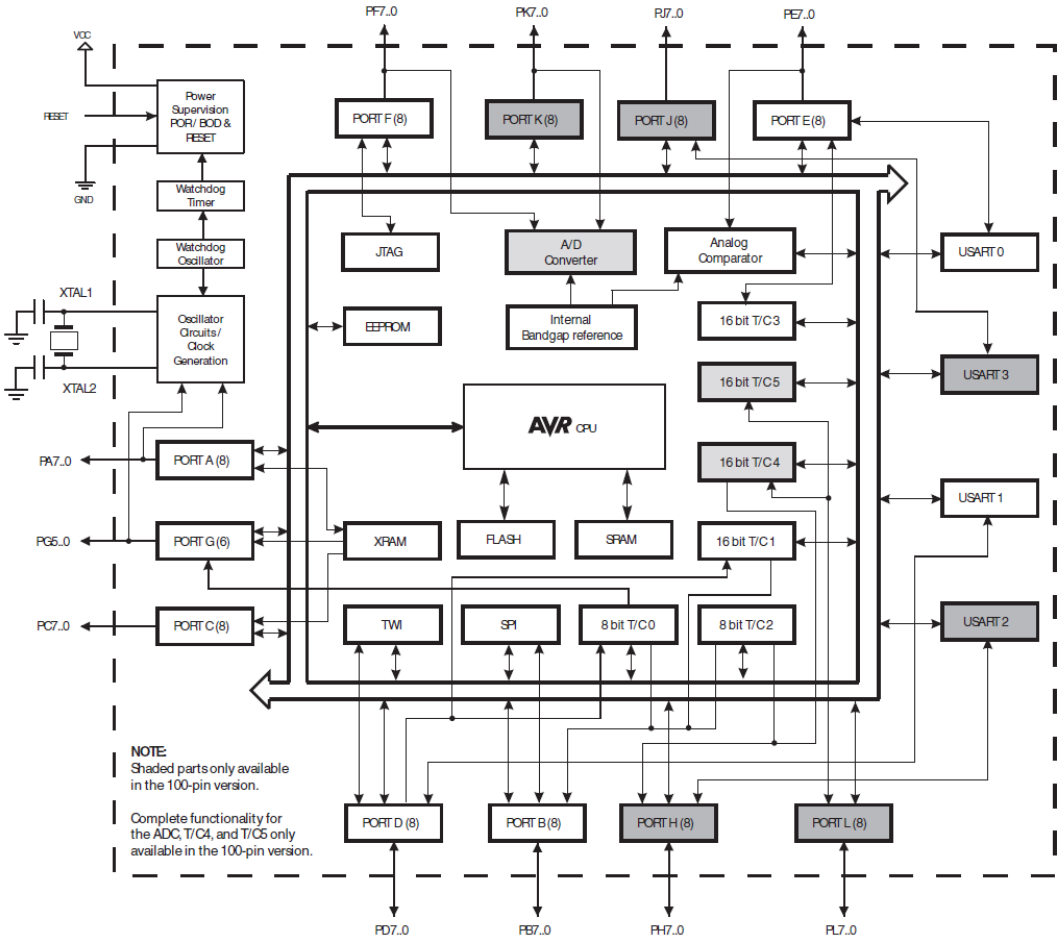

Figura 8 - Diagrama de bloques ATmega 2560

En el diagrama de bloques que se muestran en la Figura 8, se puede apreciar la estructura interna del microcontrolador. En el centro se encuentra la CPU (AVR CPU) del microcontrolador, conectada mediante un único bus a varios bloques. Estos bloques auxiliares van desde puertos de entrada/salida o de comunicaciones (USART, SPI, I2C) hasta módulos de memoria.

2.2.2.2 - AVR CPU

La función principal del núcleo de CPU es encargarse de la correcta ejecución del programa. La CPU debe contar con acceso a memorias, realizar cálculos, controlar periféricos y manejar interrupciones.

Para maximizar el rendimiento y el paralelismo, se emplea una arquitectura Harvard, que separa buses y memorias para el programa, por un lado, y para datos por otro. Las instrucciones de la memoria de programa se ejecutan con un único nivel de segmentación (‘pipelining’). Mientras se está ejecutando una instrucción, la siguiente se precarga desde la memoria de programa. Este proceso es el que permite ejecutar las instrucciones en un único ciclo de reloj. La memoria de programa está implementada como una memoria Flash reprogramable por el sistema.

El banco de registros de acceso rápido contiene 32 registros de 8 bits de propósito general con un tiempo de acceso de un ciclo de reloj. Esto permite que la ALU (Arithmetic Logic Unit) realice operaciones en un único ciclo de reloj. En la típica operación de la ALU, se extraen dos operadores del banco de registros, la operación se ejecuta y se almacena el resultado en el banco de registros –todo en un ciclo de reloj-.

La ALU soporta operaciones aritméticas y lógicas entre registros o bien, entre una constante y un registro. También puede ejecutar operaciones con un único registro. Después de una operación, se actualiza el registro de estado para mostrar información sobre el resultado.

El control del programa se maneja a través de saltos condicionales e incondicionales y llamadas a instrucciones. La mayoría de instrucciones de AVR tienen un formato de 16 bits.

La memoria ‘Flash’ de programa está dividida en dos secciones, la sección de arranque (‘boot’) y la de aplicación. Ambas secciones cuentan con un bit de bloqueo para protegerlas frente a lecturas/escrituras.

Durante las interrupciones y llamadas a subrutinas, la dirección de retorno (Program Counter) se almacena en el ‘stack’. El ‘stack’ se encuentra en la memoria

SRAM de datos y su tamaño se limita teniendo en cuenta el tamaño total de la memoria y el uso de la misma.

El espacio de memoria destinado a entrada/salida contiene 64 direcciones implementadas como registros de control, SPI...para controlar periféricos. Se puede acceder a este espacio de memoria de forma directa

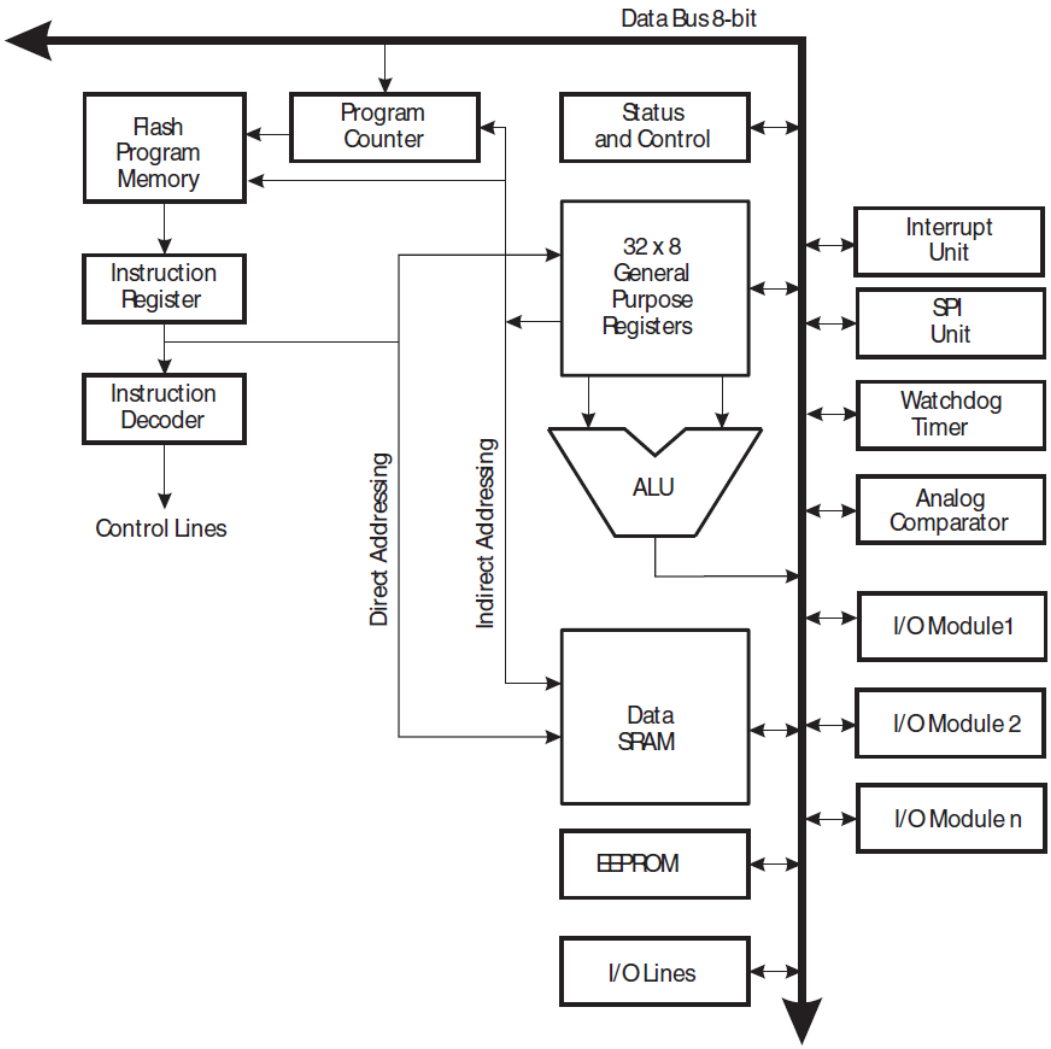

Figura 9 - Arquitectura AVR CPU

2.3 - SystemC

SystemC es un lenguaje de diseño de sistemas electrónicos HW/SW. Consiste en un conjunto de clases C++ y macros que hace posible una simulación de procesos concurrentes. Los objetos descritos en SystemC pueden comunicarse entre sí, durante una simulación de tiempo real, empleando señales de cualquier tipo disponible en C++, de los tipos ofrecidos por la librería SystemC o de aquellos definidos por el usuario.

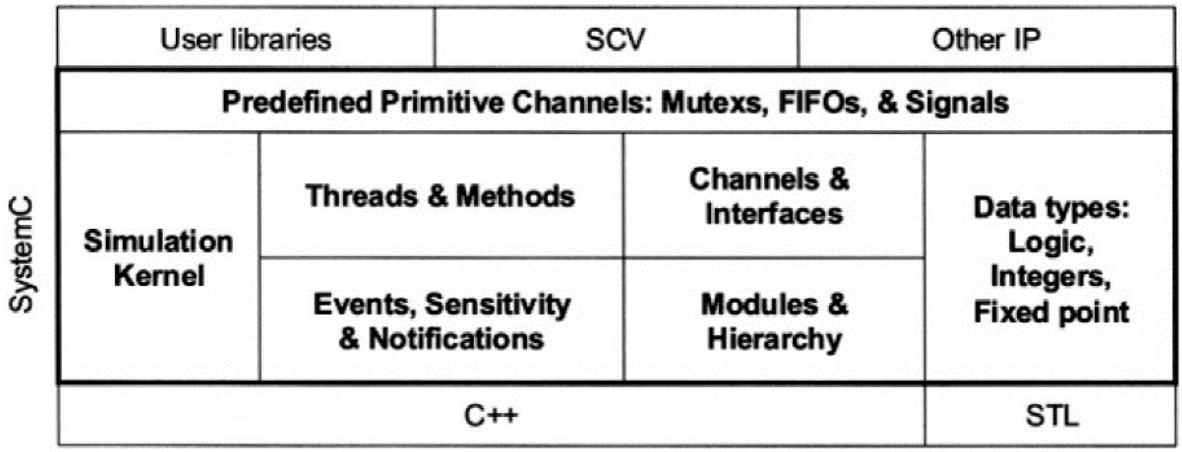

Figura 10 - Estructura del lenguaje SystemC

SystemC ofrece una mejora en la productividad debido a que permite diseñar tanto los componentes 'hardware' como 'software' que existirían en el diseño real, con un alto nivel de abstracción. Este nivel de abstracción permite al equipo de diseño, en las etapas más tempranas del proceso de diseño, comprender mejor las interacciones de todo el sistema y poder verificar el sistema antes y de manera más eficiente.

2.3.1 - Kernel de simulación

La simulación en SystemC tiene principalmente dos fases de operación: elaboración y ejecución. Una tercera fase, ocurre tras la ejecución y se podría caracterizar como post-procesado o 'cleanup'.

La fase de elaboración se encarga de ejecutar todo el código que va antes de la llamada a la función *sc_start*. Esta fase se caracteriza por la inicialización de estructuras de satos, establecer las conexiones y la preparación de la segunda fase, la ejecución.

La fase de ejecución pasa el control al ‘kernel’ de simulación de SystemC, que orquesta la ejecución de todos los procesos para crear una ilusión de concurrencia.

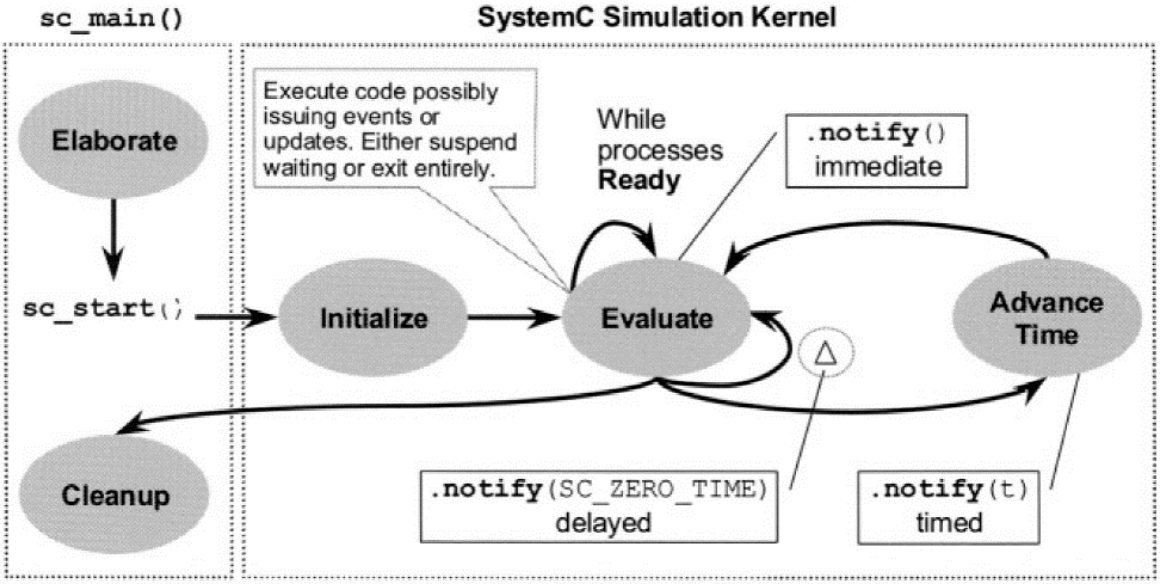

Figura 11 - Kernel de simulación de SystemC

La Figura 11 recuerda a los ‘kernels’ de simulación de Verilog y VHDL. Tras la llamada a la función *sc_start*, todos los procesos de simulación (salvo alguna excepción) se invocan aleatoriamente durante la inicialización.

Después de la inicialización, se ejecuta un proceso de simulación cuando ocurra alguno de los eventos a los que es sensible. Muchos procesos de simulación pueden activarse en el mismo instante en el tiempo del simulador. Debido a esto, se evalúan todos estos procesos, y se actualizan sus salidas. Una evaluación seguida de una actualización se conoce como ‘delta-cycle’. Si en ese instante no es necesario evaluar ningún otro proceso, la simulación avanza. Cuando ningún proceso de simulación necesita seguir ejecutándose, la simulación termina.

2.4 - SystemC-AMS

Las extensiones SystemC-AMS se han elaborado sobre el estándar IEEE de SystemC y definen construcciones adicionales para el lenguaje que introducen nuevas semánticas de ejecución y nuevas metodologías de modelado a nivel de sistemas, para diseñar y verificar sistemas de señales mixtos.

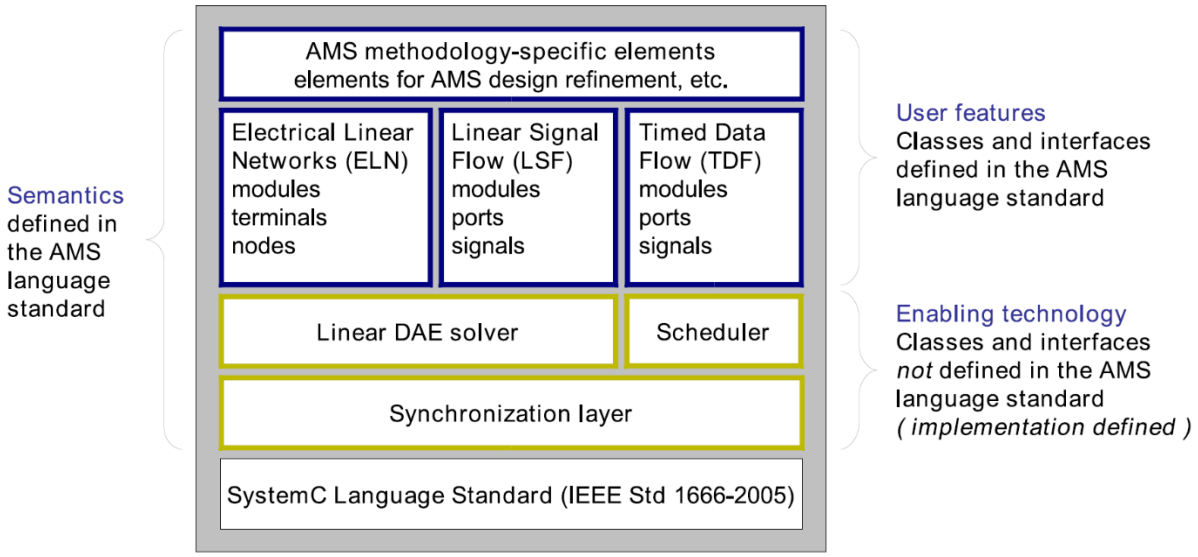

Figura 12 - Estructura del lenguaje SystemC-AMS

Las clases que proporciona la extensión AMS se pueden usar combinadas junto con una implementación compatible SystemC. Esta implementación se puede emplear para crear modelos AMS que validen y optimicen la arquitectura del sistema, que exploren varios algoritmos o que provean de un prototipo virtual operacional de un sistema completo, incluyendo la funcionalidad analógica.

Para apoyar todos estos tipos de usos, la extensión AMS define todos los formalismos necesarios para modelar un sistema a diferentes niveles de abstracción.

2.4.1 - Modelos de computación

Las extensiones AMS añaden nuevos métodos de computación para el modelado y simulación de sistemas a los ya existentes en SystemC. Los niveles de abstracción distinguen entre comportamiento con tiempo discreto o continuo y descripciones conservadoras o no conservadoras.

Los diferentes modelos de computación que se introducen son Timed Data Flow (TDF), Linear Signal Flow (LSF) y Electrical Linear Networks (ELN). A continuación se describirán brevemente todos ellos, y más adelante se hablará más en detalle sobre el TDF, que es el que se ha empleado en el proyecto.

2.4.1.1 - Linear Signal Flow

Soporta el modelado del comportamiento de señales en el tiempo continuo, empleando un conjunto de primitivas, como sumadores, multiplicadores o integradores. El modelo LSF se crea conectando estas primitivas a través de señales temporales de valor real, que pueden representar cualquier tipo de cantidad.

2.4.1.2 - Electrical Linear Networks

El modelado de redes eléctricas es posible gracias a unas primitivas predefinidas, como resistencias o condensadores, que se emplean como macro modelos para describir las relaciones en tiempo continuo entre tensiones y corrientes.

2.4.1.3 - Timed Data Flow

Las semánticas de ejecución basadas en TDF introducen un modelado de tiempo discreto y una simulación sin la sobrecarga de la planificación dinámica que impone el 'kernel' de SystemC. La simulación se acelera definiendo una planificación estática, que se computa antes de que empiece la simulación y que ejecuta las funciones de los módulos TDF acorde a la dirección del flujo de datos.

Las señales discretas, muestreadas, se propagan a través de los módulos TDF y pueden representar cualquier tipo C++.

La Figura 13 muestra la relación entre los modelos de computación (TDF, LSF y ELN) que la extensión AMS provee y el campo de modelado y abstracción en el que típicamente son usados.

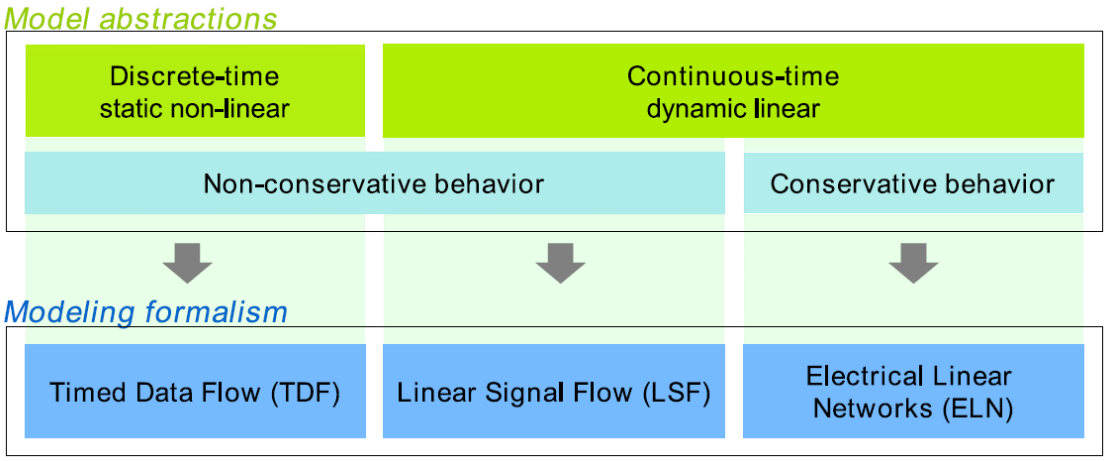

Figura 13 - Abstracción y modelos

3 - API Arduino

Los ‘sketches’ de Arduino incluyen la librería *Arduino.h*. Esta librería está definida en varios ficheros fuente en los que se encuentran codificadas las funciones que permiten la programación en lenguaje Arduino, frente a un algoritmo en C++ puro. Para el proyecto, son de especial interés las funciones relacionadas con la entrada/salida y las interrupciones pues son las que definen los momentos y modos de interacción de la placa con el entorno.

Vippe solo reconoce ficheros C++. Para habilitar cualquier ‘sketch’ Arduino de cara a la simulación, en dicho ‘sketch’ se sustituirá la librería *Arduino.h* por otra, *uc_Arduino.h*, en la que se hará uso de la API de Vippe en aquellas funciones que sea necesaria y que se detallarán después. Los ficheros fuente de la librería, modificados, tienen el prefijo ‘uc_’ en su nombre.

Los cuerpos de las funciones que contiene la librería se han modificado, pero su propósito es el mismo. Aquellas funciones relacionadas con la entrada/salida digital en un Arduino real, por ejemplo, ahora se encargan de que el modelo SystemC Arduino realice la misma funcionalidad en el mismo momento.

Para habilitar el ‘sketch’ de un usuario para la simulación en Vippe, aparte de incluir la línea *#include “uc_Arduino.h”*, en lugar de la normal, al principio de su código, debe renombrar la extensión del ‘sketch’ de ‘.ino’ a ‘.cpp’. El ‘sketch’ modificado ya puede ser compilado con Vippe e instrumentado para incluir anotaciones temporales y de rendimiento.

En la Figura 14 se muestran los ficheros fuente y las funciones que forman esta nueva librería y que se analizarán a continuación

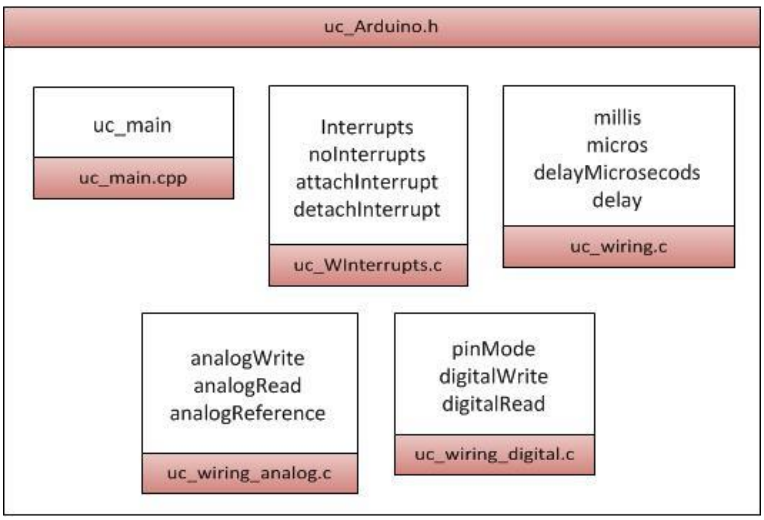

Figura 14 - API Arduino

3.1 - uc_Arduino.h

Se trata del fichero de cabecera que incluye cualquier ‘sketch’ Arduino. Se han introducido modificaciones para incluir todas las nuevas funciones habilitadas para simulación y las librerías de Vippe. Además se encuentran declaradas varias macros y funciones más simples.

3.2 - uc_main.cpp

Dentro se encuentra la función *uc_main*. Esta función es compilada por Vippe y actúa como una función *main* normal de C++. Se encarga de llamar, primero, a la función *setup*, y después en un bucle infinito a la función *loop,* simulando el funcionamiento de la placa Arduino.

La función *setup* es la primera en ejecutarse, como en cualquier ‘sketch’ y en ella se configuran todos los pines o se inicializan los valores que se van a emplear.

La función *loop* contiene el código que se encarga de controlar el comportamiento de la placa. Al encontrarse dentro de un bucle infinito, se repite tras acabar.

3.3 - uc_WInterrupts.c

Aquí se encuentran definidas todas las funciones que tienen que ver con las interrupciones. Además, una variable global (‘flag’) indica el estado en que se encuentran las interrupciones.

3.3.1 - interrupts

Se encarga de activar las interrupciones modificando un ‘flag’. Por defecto, siempre están activadas hasta que se llame a *noInterrupts*.

Dado que es el ejecutable SystemC el que manda interrupciones a Vippe, es necesario que este ejecutable conozca si están activadas o no. Para ello, se hace una lectura de un pin que no pertenece al microcontrolador (90) para notificárselo.

3.3.2 - noInterrupts

Es la función complementaria a la anterior, ya que desactiva las interrupciones hasta que se vuelva a llamar a *interrupts*. Se suele emplear en secciones críticas de código para evitar problemas. La notificación al ejecutable SystemC se hace de manera similar.

```
int interrupts_flag = 1;

void interrupts(void){
    interrupts_flag = 1;
    void* addr = (void*)90;
    dig_read(addr);
    return;
}

void noInterrupts(void){
    interrupts_flag = 0;
    void* addr = (void*)91;
    dig_read(addr);
    return;
}
```

Figura 15 - Interrupts & noInterrupts

3.3.3 - attachInterrupt

Habilita una de las interrupciones con las que cuenta la placa y le asigna un tipo de activación. Existen cuatro tipos:

- LOW: Activa la interrupción cuando el pin se encuentra a nivel bajo.
- FALLING: La activa cuando cambia a ‘1’.
- RISING: La activa cuando cambia a ‘0’.
- CHANGE: En cualquiera de los dos casos anteriores.

A la función le llega un número de interrupción. La prioridad de la misma, es mayor cuanto menor es el número (la interrupción 0 es la que tiene máxima prioridad).

Se comunica a Vippe, mediante la función de su API *vippe_IRQ*, qué función del código usuario atiende la interrupción.

```
void attachInterrupt(int interruptNum, void (*userFunc)(void), int mode){
    if(interruptNum < EXTERNAL_NUM_INTERRUPTS && interrupts_flag == 1){
        vippe_IRQ(interruptNum, userFunc);
        int val, pin;
        if(interruptNum == 0) pin = 2;
        else if(interruptNum == 1) pin = 3;
        else if(interruptNum == 5) pin = 18;
        else if(interruptNum == 4) pin = 19;
        else if(interruptNum == 3) pin = 20;
        else pin = 21;
        switch(mode){
            case 1: //CHANGE
                val = -6;
            break;
            case 2: //FALLING
                val = -4;
            break;
            case 3: //RISING
                val = -5;
            break;
            case 0x0: //LOW
                val = -3;
            break;
            }
        void* addr = (void*)pin;
        dig_write(addr, val);
    }
    return;
}
```

Figura 16 - attachInterrupt

Al ejecutable SystemC se le comunica, mediante la función *dig_write* de la API de Vippe, cuál es el pin que se corresponde con la interrupción en cuestión (a la interrupción 0 le corresponde el pin 2, por ejemplo). El ejecutable SystemC recibe una escritura en el pin de un valor negativo, dependiendo del tipo de interrupción. El uso de un valor negativo es una forma de usar la función *dig_write* para comunicar una opción de configuración cuando, usualmente, se usa para escribir un valor digital en un puerto de E/S.

3.3.4 - detachInterrupt

Al contrario que en la función anterior, se deshabilita la interrupción de un determinado pin.

El proceso es similar al visto anteriormente. Empleando la misma función de la API de Vippe, se desvincula la función que atendía la interrupción. El ejecutable SystemC recibe otra escritura para indicar que ese pin no funciona ya como interrupción.

```
void detachInterrupt(int interruptNum){
    if(interruptNum < EXTERNAL_NUM_INTERRUPTS && interrupts_flag == 1){
        int pin;
        vippe_IRQ(interruptNum, NULL);
        if(interruptNum == 0) pin = 2;
        else if(interruptNum == 1) pin = 3;
        else if(interruptNum == 5) pin = 18;
        else if(interruptNum == 4) pin = 19;
        else if(interruptNum == 3) pin = 20;
        else pin = 21;
        void* addr = (void*)pin;
        dig_write(addr, -7);
    }
    return;
}
```

Figura 17 - detachInterrupt

3.4 - uc_wiring.c

En este fichero se encuentran todas las funciones relacionadas con el tiempo

3.4.1 - millis

Esta función devuelve, al terminar de ejecutarse, el tiempo que lleva el programa ejecutándose en milisegundos. Para ello, en esta versión modificada se emplea la función *time_real_watch* de la API de Vippe, que devuelve el tiempo actual de simulación. Los tiempos que emplea Vippe, tienen una unidad de nanosegundos, por lo que hay que transformarlo a milisegundos.

3.4.2 - micros

De forma similar a *milis*, devuelve el tiempo que lleva el programa ejecutándose en microsegundos. Para ello, se emplea también la función *time_real_watch* de la API de Vippe. Los tiempos que emplea Vippe, tienen una unidad de nanosegundos, por lo que hay que transformarlo a microsegundos.

```
unsigned long millis(){

    unsigned long m;

    // Nanoseconds to Milliseconds:
    m = time_real_watch()/1000000;
    return m;
}

unsigned long micros(){

    unsigned long m;

    // Nanoseconds to Microseconds:
    m = time_real_watch()/1000;
    return m;
}
```

Figura 18 - millis & micros

3.4.3 - delayMicroseconds

Esta función se encarga de pausar la ejecución durante un determinado tiempo en microsegundos. Es una de las funciones más usadas en los 'sketches' Arduino.

El tiempo que recibe como argumento, hay que pasarlo a nanosegundos, unidad temporal que emplea Vippe. Se emplea de nuevo la función *time_real_watch* para conocer el tiempo actual de ejecución. Se suman el tiempo actual y el tiempo a esperar para obtener el instante de tiempo en el que hay que dejar de esperar.

Mediante un bucle, se compara si el tiempo actual, es mayor o igual que el instante en el que hay que abandonar la función. En caso contrario, se emplea otra función de Vippe, *uc_add_times*, para que el tiempo de simulación avance 1 microsegundo (‘slice’).

```
void delayMicroseconds(unsigned int us){

    // Convert time to ns (VIPPE units)
    long long int wait_time = (long long int)(us * 1e3);
    long long int actualtime=time_real_watch();
    long long int nexttime=actualtime+wait_time;
    while(nexttime > time_real_watch()){
      uc_add_times(1000,0,0,0);
    }
    return;
}
```

Figura 19 - delayMicroseconds

3.4.4 - delay

Esta función es prácticamente idéntica a la anterior, ya que se encarga de pausar la ejecución durante un determinado tiempo en milisegundos.

La única diferencia, es que el tiempo recibido como argumento se multiplica por una constante mayor para pasarlo a nanosegundos.

```
void delay(unsigned long ms){

    // Convert time to ns (VIPPE units)
    long long int wait_time = (long long int)(ms * 1e6);
    long long int actualtime=time_real_watch();
    long long int nexttime=actualtime+wait_time;
    while(nexttime > time_real_watch()){
      uc_add_times(1000,0,0,0);
    }
    return;
}
```

Figura 20 - delay

3.5 - uc_wiring_digital.c

En este fichero se encuentran todas las funciones relacionadas con la entrada/salida digital, así como la configuración de los pines.

3.5.1 - pinMode

Se encarga de configurar un pin como entrada o salida. Como argumentos recibe el número de pin y el modo, que puede ser 'OUTPUT' o 'INPUT'. Dependiendo del caso, se emplea la función *dig_write* de la API de Vippe, para escribir un valor negativo (-1 o -2) sobre el pin y que SystemC sepa cómo está configurado.

```
void pinMode(int pin, int mode){

    void* addr = (void*)pin;
    if (mode == INPUT)
    {
        dig_write(addr, -1);
    } else if (mode == OUTPUT) {
        dig_write(addr, -2);
    }
    return;
}
```

Figura 21 - pinMode

3.5.2 - digitalWrite

Esta función se encarga de escribir un valor binario en un pin digital configurado como salida por *pinMode*. Para ello, simplemente, se transforma el número de pin en dirección de memoria y se manda, junto con el valor, a SystemC con la función *dig_write* de Vippe.

```
void digitalWrite(int pin, int val){

    int vippe_val = val;
    void* addr = (void*)pin;
    dig_write(addr, vippe_val);
    return;
}
```

Figura 22 - digitalWrite

3.5.3 - digitalRead

Lee el valor de uno de los pines en SystemC y lo devuelve. Para ello, se transforma el argumento en una dirección de memoria y se llama a la función *dig_read* de la API de Vippe. Éste se encargará de pedírselo a SystemC.

```
int digitalRead(int pin){

    void* addr = (void*)pin;
    int val = dig_read(addr);
    return val;
}
```

Figura 23 - digitalRead

3.6 - uc_wiring_analog.c

En este fichero se encuentran todas las funciones relacionadas con la entrada/salida analógica.

3.6.1 - analogWrite

Esta función se emplea cuando se quiere escribir un valor analógico (0-255) en un pin. Este valor sirve para definir el porcentaje de tiempo que la señal PWM se encuentra en valor 1 respecto al periodo de dicha señal.

Lo primero que hace es comprobar que los valores de los argumentos son correctos. El número del pin debe ser número válido (los pines que pueden funcionar como analógicos son 2-13 y 44-46) y el valor a escribir debe estar dentro del rango, en caso contrario se sesga.

A continuación, se indica la escritura a SystemC con la función *dig_write* de Vippe. Dado que esta función también se usa para escrituras digitales y que los pines que funcionan como analógicos también pueden funcionar como digitales, es necesario hacer algo para detectar de qué tipo de escritura se trata.

Se añade un 'offset' de 100 al número de pin que se envía a SystemC, de modo que cualquier pin con un número mayor que 100 que reciba, lo considerará analógico.

```
void analogWrite(int pin, int val){

    //Digital Pins -- PWM
    if((pin >=2 && pin <= 13) || (pin >=44 && pin <= 46)){
        if (val < 0) val = 0;
        if (val > 255) val = 255;
        // Analog offset
        int ana_pin = pin + 100;
        void *addr = (void*)ana_pin;
        dig_write(addr ,val);
    }
    return;
}
```

Figura 24 - analogWrite

3.6.2 - analogRead

Se emplea cuando se quiere leer un valor analógico (0-1023) en un pin. Al igual que antes, se comprueba que el número de pin del argumento sea válido.

Tras esto, se indica la lectura a SystemC con la función *dig_read* de Vippe, apareciendo el mismo problema de antes. Esta función también se usa para lecturas digitales y estos pines pueden funcionar como digitales.

Como el problema es el mismo, se emplea la misma solución. Se añade un 'offset' de 100 al número de pin que se envía a SystemC, de modo que cualquier pin con un número mayor que 100 que reciba, lo considerará analógico.

```
int analogRead(int pin){

    int val = 0;

    if(pin >= 0 && pin <= 15){
        int ana_pin = pin + 100;
        void *addr = (void*)ana_pin;
        val = dig_read(addr);
    }
    return val;
}
```

Figura 25 - analogRead

3.6.3 - analogReference

Configura el voltaje de referencia para la entrada analógica (especifica el voltaje máximo). Puede recibir los siguientes argumentos como opciones:

- DEFAULT: Fija la referencia en 5 V. Es el valor predeterminado.
- INTERNAL1v1: Se emplean 1.1 V como referencia.
- INTERNAL2V56: Se emplean 2.56 V como referencia.
- EXTERNAL: El valor de tensión viene fijado por el pin AREF (de 0 a 5 V) que configura el usuario.

Cada una de estas opciones se corresponde con un número entero (0-3) definido en *uc_Arduino.h*. Con la función *dig_write*, se escribe sobre una dirección ficticia (200), que no pertenece al microcontrolador, para que SystemC reconozca que se trata de un cambio en la referencia analógica.

```
void analogReference(int mode){

    int val = mode;
    void *addr = (void*)200;
    if(val <= 3 || val >= 0)
        dig_write(addr, val);
    return;
}
```

Figura 26 - analogReference

4 - Modelo Arduino en SystemC-AMS

4.1 - Módulos del modelo SystemC

El modelo SystemC está compuesto por un único módulo (*arduino-ams*). Dentro del módulo *arduino-ams* se encuentran otros dos sub-módulos, uno llamado *Sim_arduino* que modela en SystemC todo el comportamiento digital del micro y un convertidor A/D (parte SystemC-AMS), llamado *ad_converter*, que se encarga de convertir las entradas analógicas. En la Figura 27 aparecen ambos sub-módulos, así como las funciones o procesos que los forman y que se detallarán más adelante.

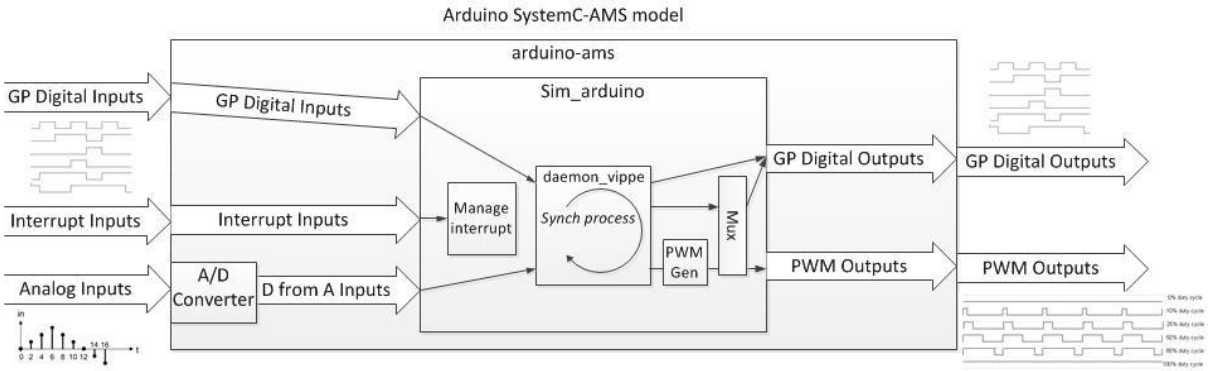

Figura 27 - Modelo Arduino SystemC-AMS

El módulo *arduino-ams*, puede conectarse con aquello que se quiera simular, ya sea un mero banco de pruebas, o un modelo SystemC-AMS de un sistema. Esta conexión se realiza en el fichero *main.cpp*. Posteriormente se mostrará un ejemplo.

A continuación, se explicarán todas las funciones y procesos que forman los dos sub-módulos y que aparecen en la Figura 27, comenzando por *Sim_arduino* y después con *ad_converter*. También se analizará más en detalle el modelo TDF de SystemC-AMS, que es el que se ha empleado en este convertidor.

4.1.1 - Sim_arduino

Se trata del módulo principal del programa, ya que se encarga de gestionar su eje temporal, cambiar el valor de los pines en cada instante de tiempo, controlar la gestión de interrupciones y de acceder a la memoria compartida.

```
extern char * shmemnumber;

SC_MODULE(Sim_arduino){

    // Arduino Mega 2560 Pinout

    // Digital
    sc_inout_resolved digital_pin[N_DIG_PINS];

    // Analog - DE
    sc_in <sc_uint<10> > analog_pin[N_ANA_PINS];

    // A/D Configuration
    sc_out <double> ad_config;

    // PWM Pins Config | 2-11 & 44-46
    int m_mux_id;
    int pwm_value[N_PWM_PINS]; // PWM value
    sc_event mux_event[N_PWM_PINS]; // Digital/PWM Write Event
    sc_logic pwm_output[N_PWM_PINS]; // PWM output
    int mux_mode[N_PWM_PINS]; // 0-DigWr, 1-PWM, 2-DigRd
    int m_mux_dig_val; // Digital Write value store


    // Interrupt Pins Config | 2, 3, 18, 19, 20, 21
    int interrupt_control[N_INT_PINS];
    bool interrupt_enable;
    sc_logic interrupt_old[N_INT_PINS];

    void *pwm_gen(void *pin);
    void mux();
    void send_IRQ_to_Vippe(sc_time time, int proc_id, int IRQ_number);
    void manage_interrupt();
    void daemon_vippe();

    SC_CTOR(Sim_arduino){
        m_mux_id = 0;
        for(int i = 0; i < N_PWM_PINS; i++){
            SC_THREAD(mux);
        }
    SC_THREAD(daemon_vippe);
    SC_METHOD(manage_interrupt);
            dont_initialize();
            sensitive << digital_pin[2] << digital_pin[3] << digital_pin[18]
            << digital_pin[19] << digital_pin[20] << digital_pin[21];
  }
};
```

Figura 28 - Sim_arduino

Este sub-módulo es el más complejo de todo el proyecto. Contiene numerosas funciones y variables para controlar el comportamiento del modelo Arduino a lo largo de la simulación. Las funciones se explican a continuación.

4.1.1.1 - manage_interrupt

Se trata de una función implementada como SC_METHOD, que se ejecuta cada vez que cambia de valor cualquiera de los pines que puedan funcionar como interrupción. Independientemente de si funcionan como interrupción o no, siempre almacena el valor al que cambian, para poder comparar más adelante.

Es capaz de enviar a Vippe varias interrupciones que hayan ocurrido en el mismo instante de tiempo.

```
void Sim_arduino::manage_interrupt(){

    int i, pin;
    for(i = 0; i < N_INT_PINS; i++){
        if(i == 0) pin = 2;
        else if(i == 1) pin = 3;
        else if(i == 2) pin = 21;
        else if(i == 3) pin = 20;
        else if(i == 4) pin = 19;
        else pin = 18;
        if(interrupt_control[i] != -1 && interrupt_enable == 1){
            switch(interrupt_control[i]){
            case -3: // LOW
                if(digital_pin[pin].read() == 0 || interrupt_old[i] == 0){
                    send_IRQ_to_Vippe(sc_time_stamp(), 0, i);
                    cout << "Interrupt | Type: LOW, pin: " << pin << ", t: " << sc_time_stamp() << endl;
                }
            break;
            case -4: // FALLING
                if(digital_pin[pin].read() == 1 && interrupt_old[i] == 0){
                    send_IRQ_to_Vippe(sc_time_stamp(), 0, i);
                    cout << "Interrupt | Type: FALLING, pin: " << pin << ", t: " << sc_time_stamp() << endl;
                }
            break;
            case -5: // RISING
                if(digital_pin[pin].read() == 0 && interrupt_old[i] == 1){
                    send_IRQ_to_Vippe(sc_time_stamp(), 0, i);
                    cout << "Interrupt | Type: RISING, pin: " << pin << ", t: " << sc_time_stamp() << endl;
                }
            break;
            case -6: // CHANGE
                if(digital_pin[pin].read() != interrupt_old[i]){
                    send_IRQ_to_Vippe(sc_time_stamp(), 0, i);
                    cout << "Interrupt | Type: CHANGE, pin: " << pin << ", t: " << sc_time_stamp() << endl;
                }
            break;
            default:
                cout << "Interrupt | Error" << endl;
            break;
            }
        }
        interrupt_old[i] = digital_pin[pin].read(); // Old value <- Actual value
    }
}
```

Figura 29 - manage_interrupt

4.1.1.2 - send_IRQ_to_Vippe

Contiene el protocolo necesario para notificar a Vippe que se ha producido una interrupción. Vippe almacena todas las interrupciones que tiene pendientes y las va atendiendo conforme a su prioridad.

```
void Sim_arduino::send_IRQ_to_Vippe(sc_time time, int proc_id, int IRQ_number){

    //cout << "IRQ | Sent to Vippe" << systemc_conn->next_time << endl;
    systemc_conn->next_time = (long long int) time.value()/1000;
    systemc_conn->data = proc_id;
    systemc_conn->type = 6 + IRQ_number;
    //cout << "IRQ | POST done" << endl;
    sem_post(&systemc_conn->semaphorescv);
    //cout << "IRQ | Wait done" << endl;
    sem_wait(&systemc_conn->semaphorevsc);
}
```

Figura 30 - send_IRQ_to_Vippe

4.1.1.3 - pwm_gen

Esta función no se encuentra en el constructor del módulo ya que es un 'thread' que se llama dinámicamente si es necesario. En el Arduino Mega 2560 hay varios pines digitales que pueden funcionar como tal o como pin PWM.

La señal PWM de cada pin, debe cambiar a una determinada frecuencia y 'duty cycle', que puede ser diferente en cada uno. Por esta razón, se ha optado por crear un thread dinámicamente cada vez que se produce un *analogWrite* sobre un pin.

Como ya se ha visto anteriormente, si al estar un pin generando la señal PWM, se produce una lectura o escritura digital, el pin pararía de generar la señal y volvería a ser un pin digital normal. De modo similar, si se produce otro *analogWrite*, cambiaría el 'duty cycle'.

La función se ha implementado de forma que si ocurre lo primero, acabe su ejecución, y si ocurre lo segundo, simplemente cambie sus variables para adecuarse al nuevo valor.

En un principio existía un problema con esta función, que también se encargaba de escribir a los pines directamente. Si se estaba generando una señal PWM en un determinado pin y acto seguido llegaba un *digitalWrite* (atendido por *Sim_arduino*), el pin tomaba un valor inválido, ya que se intentaba escribir desde dos sitios diferentes.

Para solucionarlo, la escritura y lectura de estos pines la gestiona otra función que actúa como multiplexor. *pwm_gen* y *Sim_arduino* (función de la que se hablará más adelante) activan un evento para que el multiplexor atienda una escritura o una lectura cuando sea necesario.

```
void *Sim_arduino::pwm_gen(void *pin){

    int num = (int)pin;
    int pin_pwm = num;
    if(pin_pwm >= 0 && pin_pwm <= 11)
        pin_pwm += 2;
    else if(pin_pwm >= 12 && pin_pwm <= 14)
        pin_pwm += 32;
    else
        cout << "PWM | Error" << endl;
    int pwm = pwm_value[num];
    double pwm_period_high = 1000 * (double)(pwm) / (255 * 490);
    double pwm_period_low = 1000 * (double)(255 - pwm) / (255 * 490);
    while (1){
        if(pwm_value[num] == -1)
            break;
        else if (pwm != pwm_value[num]){ // Changes PWM if AnagWr
            pwm = pwm_value[num];
            pwm_period_high = 1000 * (double)(pwm) / (255 * 490);
            pwm_period_low = 1000 * (double)(255 - pwm) / (255 * 490);
        }
        pwm_output[num] = SC_LOGIC_1;
        mux_event[num].notify();
        wait(pwm_period_high, SC_MS);
        if(pwm_value[num] == -1)
            break;
        else if (pwm != pwm_value[num]){ // Changes PWM if AnagWr
            pwm = pwm_value[num];
            pwm_period_high = 1000 * (double)(pwm) / (255 * 490);
            pwm_period_low = 1000 * (double)(255 - pwm) / (255 * 490);
        }
        pwm_output[num] = SC_LOGIC_0;
        mux_event[num].notify();
        wait(pwm_period_low, SC_MS);
    }
}
```

Figura 31 - pwm_gen

4.1.1.4 - mux

Existe un SC_THREAD que actúa como multiplexor para cada uno de los pines que pueda funcionar como PWM desde el principio de la simulación. El 'thread' se queda esperando hasta que le notifican un evento. En ese momento, comprueba qué tiene que hacer (lectura/escritura digital, PWM) y lo hace.

Los pines PWM son los únicos que se escriben/leen en esta función, para el resto se hace desde la función *daemon_vippe*.

```
void Sim_arduino::mux(){

    int num = m_mux_id;
    m_mux_id += 1;
    int pin_mux = num;

    if(pin_mux >= 0 && pin_mux <= 11)
        pin_mux += 2;
    else if(pin_mux >= 12 && pin_mux <= 14)
        pin_mux += 32;
    while(1){
        wait(mux_event[num]);
        switch(mux_mode[num]){
            case 0: // DigitalWr
                if(m_mux_dig_val == 0)
                    digital_pin[pin_mux].write(SC_LOGIC_0);
                else
                    digital_pin[pin_mux].write(SC_LOGIC_1);
            break;
            case 1: // AnalogWr
                digital_pin[pin_mux].write(pwm_output[num]);
            break;
            case 2: // DigitalRd
                digital_pin[pin_mux].write(SC_LOGIC_Z);
            break;
            default:
                cout << "Mux | Error" << endl;
            break;
        }
    }
}
```

Figura 32 - mux

4.1.1.5 - daemon_vippe

Es la función principal del módulo. Se implementa como un SC_THREAD en el constructor, ya que cuenta con paradas.

En esta función se crea el puntero *systemc_conn* a la estructura de memoria compartida empleando el identificador *shmemnumber* que Vippe pasa al programa como argumento cuando lo ejecuta. Este identificador creado por Vippe, es un número aleatorio, por lo que cambia en cada ejecución para evitar problemas.

Pese a que ya se han mencionado, a continuación se muestran las variables que forman esta estructura compartida:

- long long int *next_time*: Indica al programa SystemC hasta qué tiempo tiene que avanzar.

- int *type*: Indica qué es lo que ha ocurrido en el tiempo *next_time*. Si se recibe un '0', significa que no ha pasado nada en este tiempo, un '3' indica que se ha producido una lectura (*dig_read*) y un '4' una escritura (*dig_write*). Además el programa SystemC manda un '5' a Vippe cuando ha acabado o, como parte del protocolo para notificar una interrupción, un número a partir de '6' cuando se ha producido una, en función de su prioridad.

- void **addr*: La dirección de memoria en la que se ha producido la lectura o la escritura. En este proyecto, todas las direcciones de memoria equivaldrán a números de pines de la placa.

- int *data*: El dato que se ha escrito (*dig_write*), o el que Vippe necesita leer (*dig_read*).

- sem_t *semaphorescv*: Semáforo que bloquea a Vippe mientras el programa SystemC se ejecuta.

- sem_t *semaphorevsc*: Semáforo que bloquea al programa SystemC mientras Vippe se ejecuta.

La función consiste en un bucle que durará lo mismo que el tiempo de simulación definido en el fichero *main.cpp* del programa. Este bucle ejecuta una iteración cada vez que Vippe desbloquea el semáforo, haciendo avanzar a SystemC hasta un tiempo determinado. En este tiempo, SystemC comprueba qué es lo que ha pasado en el campo *type* de la estructura. Dependiendo de su valor, la función atiende una lectura, una escritura, o no hace nada.

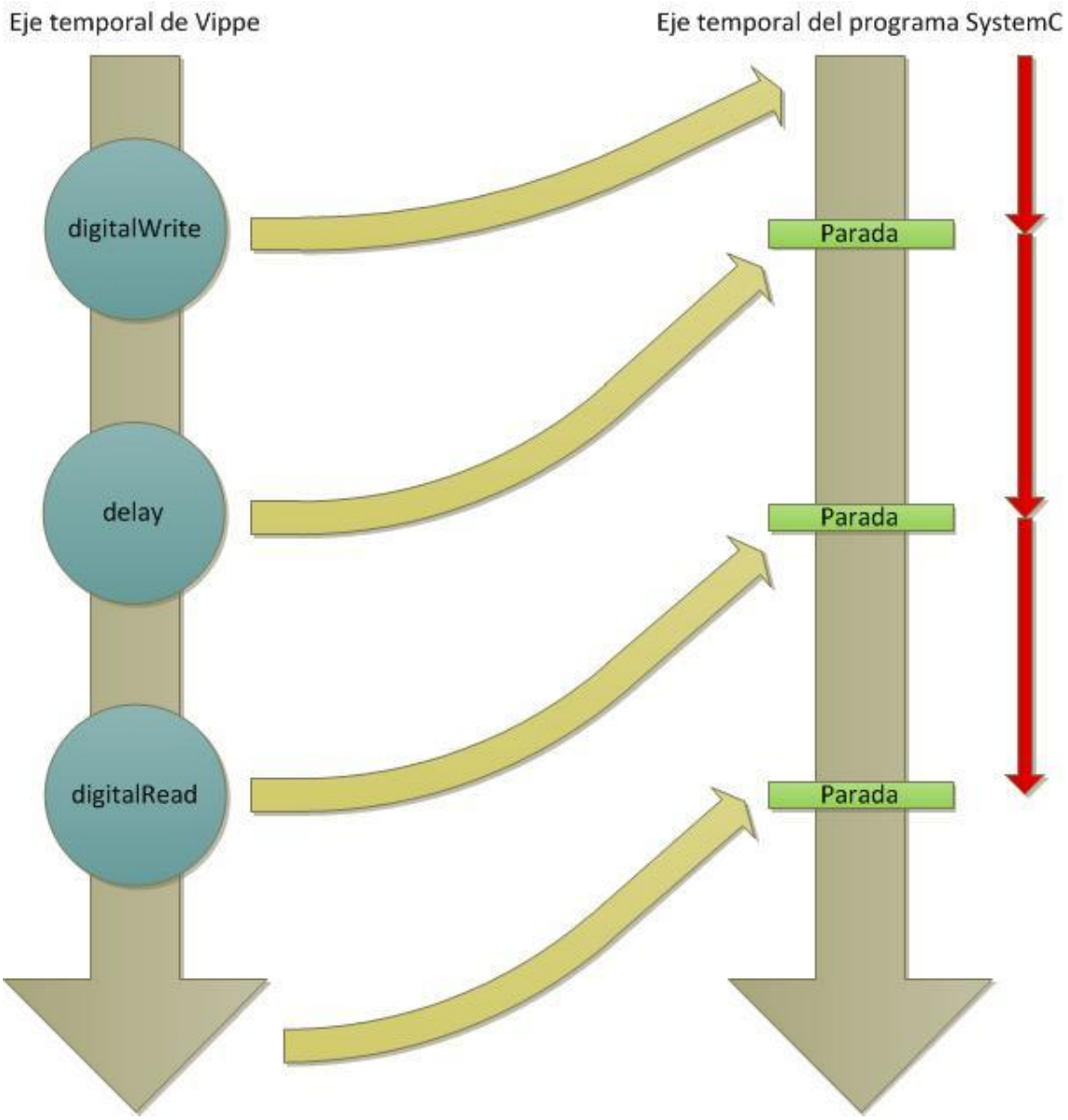

Figura 33 - Ejes temporales

En la Figura 33 se muestra de forma simplificada el avance de los ejes temporales de Vippe y el programa SystemC, controlado por *daemon_vippe*. Vippe, conforme va leyendo instrucciones del 'sketch' Arduino (círculos azules), rellena la estructura y desbloquea al programa SystemC para que avance su eje temporal y haga lo que sea necesario.

Este avance temporal del programa SystemC equivale a una iteración de la función *daemon_vippe* (flecha roja), en la que se evalúa el contenido de la estructura.

Si lo que ha llegado es una lectura, puede tratarse de una lectura digital, una analógica, o de la activación/desactivación de las interrupciones.

En los dos primeros casos, se hacen una serie de operaciones para, por ejemplo, comprobar si el pin puede funcionar o está funcionando como PWM, si es digital, o para quitar el offset del pin si es analógico. Finalmente se almacena el valor del pin en el campo *data* de la estructura compartida.

Si lo que ha llegado es una escritura, puede tratarse de una escritura digital o PWM, un cambio de pin a entrada o un nuevo valor de referencia para los pines analógicos. De manera análoga a la lectura, se hacen varias comprobaciones y finalmente se almacena el valor del campo *data*.

Una vez ha llegado al final de la iteración, se desbloquea el semáforo para que Vippe continúe, parando al programa SystemC (rectángulo verde). Vippe volverá a introducir otros datos en la estructura compartida y se repetirá todo el proceso hasta que finalice la simulación.

```
void Sim_arduino::daemon_vippe(){

    systemc_conn=(systemc_conn_struct *)create_sh_mem(sizeof(systemc_conn_struct),
    INT_TO_SHARED_MEM_CREATE_SYSTEMC, shmemnumber);

   int pin;
   int data;

   // Default analog reference
   ad_config.write(5);

    // Interrupts initially detached but enabled
    interrupt_enable = 1;
    for(int i = 0; i < N_INT_PINS; i++)
        interrupt_control[i] = -1;

    while(true){
    sem_wait(&systemc_conn->semaphorevsc);
      pin = (int)systemc_conn->addr;
      data = systemc_conn->data;

    switch(systemc_conn->type){
        case 0:
            nexttime=sc_time(systemc_conn->next_time, SC_NS);
            wait(nexttime-sc_time_stamp());
            cout << "SystemC advances to " << nexttime << endl;
            systemc_conn->type=5;
            sem_post(&systemc_conn->semaphorescv);
        break;
        case 3:
            // Read - Digital/Analog
        nexttime=sc_time(systemc_conn->next_time, SC_NS);
        wait(nexttime-sc_time_stamp());
        if(pin < N_DIG_PINS){ //Digital
            cout << "Digital Read | Pin: " << pin << ", value: "
            << digital_pin[pin].read() << ", t: " << sc_time_stamp() << endl;
                if(pin >= 2 && pin <= 13){ // Check if stopping PWM is necessary
                // Removes offset & checks if pin is generating a PWM signal
                    if(pwm_value[pin - 2] != -1){
                        pwm_value[pin - 2] = -1;
                        mux_mode[pin - 2] = 2;
                        mux_event[pin - 2].notify();
                    }
                }
                else if(pin >= 44 && pin <= 46){ // Check if stopping PWM is necessary
                // Removes offset & checks if pin is generating a PWM signal
                    if(pwm_value[pin - 32] != -1){
                        pwm_value[pin - 32] = -1;
                        mux_mode[pin - 32] = 2;
                        mux_event[pin - 32].notify();
                    }
                }
                if(digital_pin[pin].read() == 1)
                    systemc_conn->data = 1;
                else
                    systemc_conn->data = 0;
        }
        else if(pin == 90){ // Enable Interrupts
            interrupt_enable = 1;
            cout << "Interrupts enabled" << endl;
        }
        else if(pin == 91){ // Disable Interrupts
            interrupt_enable = 0;
            cout << "Interrupts disabled" << endl;
        }
```

```
        else if(pin >= ANA_OFFSET){ // Analog
            pin -= 100;
            cout << "Analog Read | Pin: " << pin << ", value: "
            << analog_pin[pin].read() << ", t: " << sc_time_stamp() << endl;
            systemc_conn->data = analog_pin[pin].read();
        }
        systemc_conn->type=5;
    sem_post(&systemc_conn->semaphorescv);
    break;
    case 4:
        // Write - Digital Bit/PWM | PinMode
    nexttime=sc_time(systemc_conn->next_time, SC_NS);
    wait(nexttime-sc_time_stamp());
    if(data == -1){ // PinMode IN
        if(pin >= 2 && pin <= 13){ // Check if stopping PWM is necessary
        // Removes offset & checks if pin is generating a PWM signal
                if(pwm_value[pin - 2] != -1){
                    pwm_value[pin - 2] = -1;
                    mux_mode[pin - 2] = 2;
                    mux_event[pin - 2].notify();
                }
            }
            else if(pin >= 44 && pin <= 46){ // Check if stopping PWM is necessary
            // Removes offset & checks if pin is generating a PWM signal
                if(pwm_value[pin - 32] != -1){
                    pwm_value[pin - 32] = -1;
                    mux_mode[pin - 32] = 2;
                    mux_event[pin - 32].notify();
                }
            }
            else
            digital_pin[pin].write(SC_LOGIC_Z);
        cout << "PinMode | Input pin: " << pin
        << ", t: " << sc_time_stamp() << endl;
        }
        else if(data == -2) // PinMode OUT | Nothing to be done
            cout << "PinMode | Output pin: " << pin
            << ", t: " << sc_time_stamp() << endl;
        else if(data <= -3 && data >= -6){ // Attach Interrupt
            if(pin == 2) interrupt control[0] = data;
            else if(pin == 3) interrupt_control[1] = data;
            else if(pin == 18) interrupt_control[5] = data;
            else if(pin == 19) interrupt_control[4] = data;
            else if(pin == 20) interrupt_control[3] = data;
            else interrupt_control[2] = data;
            cout << "Attach Interrupt | Pin: " << pin
            << ", t: " << sc_time_stamp() << endl;
        }
        else if(data == -7){ // Detach Interrupt
            if(pin == 2) interrupt_control[0] = -1;
            else if(pin == 3) interrupt_control[1] = -1;
            else if(pin == 18) interrupt_control[5] = -1;
            else if(pin == 19) interrupt_control[4] = -1;
            else if(pin == 20) interrupt_control[3] = -1;
            else interrupt_control[2] = -1;
            cout << "Detach Interrupt | Pin: " << pin
            << ", t: " << sc_time_stamp() << endl;
        }
        else if(pin < N_DIG_PINS){ // Digital Write
            if(pin >= 2 && pin <= 13){ // Check if PWM/Interrupt pin
                pwm_value[pin - 2] = -1; // Stops PWM Thread
                mux_mode[pin - 2] = 0;
                m_mux_dig_val = systemc_conn->data;
                mux_event[pin - 2].notify();
            }
            else if(pin >= 44 && pin <= 46){ // Check if PWM pin
                pwm_value[pin - 32] = -1; // Stops PWM Thread
                mux_mode[pin - 32] = 0;
                m_mux_dig_val = systemc_conn->data;
                mux_event[pin - 32].notify();
            }
```

```
                else{
                    if(systemc_conn->data == 0)
                        digital_pin[pin].write(SC_LOGIC_0);
                    else
                        digital_pin[pin].write(SC_LOGIC_1);
                }
                cout << "Digital Write | Pin: " << pin << ", value: "
                << data << ", t: " << sc_time_stamp() << endl;
            }
            else if(pin >= ANA_OFFSET){ // PWM Write - 490 Hz
                pin -= 100;
                if(pin >= 2 && pin <= 13){ // Pins 2-13
                    pin -= 2;
                        pwm_value[pin] = data;
                        if(mux_mode[pin] != 1){ // Checks if already in PWM mode
                            mux_mode[pin] = 1;
                            sc_spawn(sc_bind(&Sim_arduino::pwm_gen, this, (void *)pin));
                        }
                }
                else if(pin >= 44 && pin <= 46){ // Pins 44-46
                    pin -= 32;
                    pwm_value[pin] = data;
                        if(mux_mode[pin] != 1){ // Checks if already in PWM mode
                            mux_mode[pin] = 1;
                            sc_spawn(sc_bind(&Sim_arduino::pwm_gen, this, (void *)pin));
                        }
                }
                else if(pin == 100){ // Analog Reference
                    switch(data){
                        case 0:
                            cout << "Analog Reference | Internal 5V" << endl;
                            ad_config.write(5);
                        break;
                        case 1:
                            cout << "Analog Reference | Internal 1.1V" << endl;
                            ad_config.write(1.1);
                        break;
                        case 2:
                            cout << "Analog Reference | Internal 2.56V" << endl;
                            ad_config.write(2.56);
                        break;
                        default:
                            cout << "Analog Reference | External" << endl;
                            ad_config.write(-1);
                        break;
                    }
                }
                else{
                    cout << "Analog Write | Error" << endl;
                }
            }
            systemc_conn->type=5;
            sem_post(&systemc_conn->semaphorescv);
            break;
            default:
                cout << "Vippe communication | Error" << endl;
            break;
            }
        }
}
```

Figura 34 - daemon_vippe

4.1.2 - ad_converter

Se trata de un módulo SystemC-AMS conectado a las entradas analógicas de *arduino_ams*. Se encarga de convertir estas entradas a señales que se puedan emplear en SystemC.

ad_converter es un módulo sencillo que solo contiene dos funciones, *set_attributes* y *processing*. Sus puertos de salida (*out_de*) son de un tipo especial que convierte los datos de un dominio a otro (AMS TDF -> SystemC Discrete Event).

```
SCA_TDF_MODULE(ad_converter){

    sca_tdf::sca_in<double>  in_tdf[N_ANA_PINS]; // TDF port
    sca_tdf::sca_in<double>  extern_ref; // External analog reference
    sca_tdf::sca_de::sca_in<double> in_max_range; // Max range config.
    sca_tdf::sc_out<sc_dt::sc_uint<10> > out_de[N_ANA_PINS]; // TDF to DE domain port

    double in_range_min, in_range_max;
    double scaleFactor;
    void processing();
    void set_attributes();

    SCA_CTOR(ad_converter){
        in_range_min = 0;
        in_range_max = 5;
        scaleFactor = (pow(2, 10) - 1) / (in_range_max - in_range_min);
        cout << "A/D Converter | Min: " << in_range_min << ", max: " << in_range_max
        << ", scale: " << scaleFactor << endl;
    }
};
```

Figura 35 - Módulo ad_converter

De los tres modelos de computación vistos en un apartado anterior, el que se ha empleado para crear este módulo es el TDF, por lo que antes de analizar el módulo, se profundizará un poco más en el modelo TDF.

4.1.2.1 - TDF en detalle

El modelo Timed Data Flow está basado en otro modelo de computación, muy conocido, llamado Synchronous Data Flow (SDF). Se diferencian principalmente en que SDF es un modelo sin tiempos, mientras que TDF modela tiempos discretos, en los que muestrea señales. Estas señales se vinculan a puntos discretos en el tiempo y llevan valores discretos o continuos como amplitudes.

4.1.2.1.1 - Fundamentos del modelo

La Figura 36 muestra los principios básicos en que se basa el modelo TDF. No representa exactamente la implementación de este modelo en el proyecto, pero las diferencias son mínimas y se comentarán más adelante.

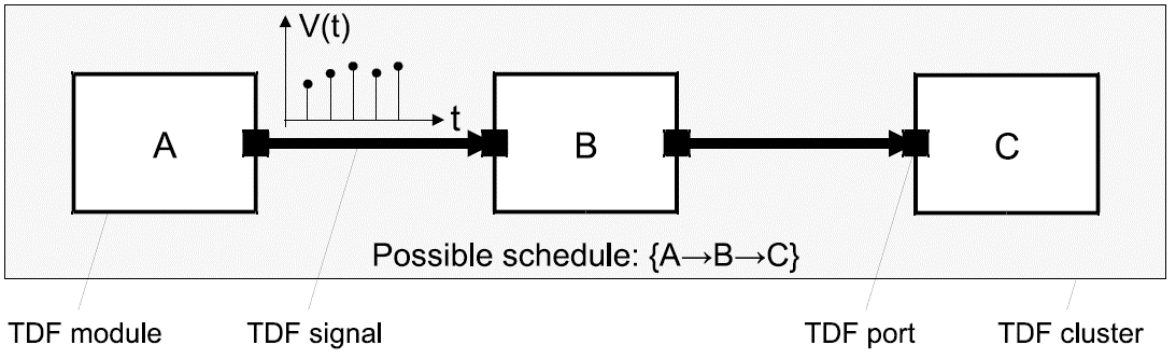

Figura 36 - 'Cluster' de módulos TDF

En la Figura 36 se pueden observar tres módulos TDF comunicados entre sí y llamados A, B y C. Un modelo TDF se compone por una serie de módulos TDF conectados, formando un 'cluster' TDF. Cada módulo TDF puede contar con numerosas entradas y salidas en forma de puertos TDF. Un módulo que contenga solo puertos de salida recibe el nombre de fuente ('source') y si solo tiene puertos de entrada, sumidero ('sink'). Para conectar módulos entre sí se emplean señales TDF.

Cada módulo TDF cuenta con un método C++ que computa una determinada función matemática. El resultado de esta función depende de las entradas y de los estados internos del módulo. El comportamiento de un 'cluster' TDF, por tanto, se define como la composición de todas las funciones de todos los módulos que intervienen en un determinado instante de tiempo, en el orden adecuado.

Una determinada función se procesa solo cuando hay suficientes muestras disponibles en los puertos de entrada. Si este es el caso, el módulo TDF las lee y la función los usa para calcular uno o varios resultados, que se escriben al puerto de salida apropiado.

El número de muestras que se leen o escriben en uno de los puertos se fija durante la simulación, pero el número de muestras leídas y escritas no tiene porqué ser el mismo. El tiempo fijado entre dos muestras se llama 'time step'.

En nuestro caso particular, existen unas pequeñas diferencias con respecto a la imagen superior.

Se podría considerar que el módulo B es nuestro *ad_converter*, el módulo A es aquel que se está simulando y el módulo C no es un módulo TDF, sino un módulo normal de SystemC, *Sim_arduino*.

La conexión entre *ad_converter* y *Sim_arduino* es un poco particular, ya que se emplea un puerto especial que transforma los datos de un dominio a otro. La última diferencia es que no hay una única señal uniendo todos los módulos, sino tantas como puertos analógicos tiene la placa.

4.1.2.1.2 - Atributos de los módulos y puertos TDF

El modelo TDF permite definir los atributos de cada módulo y puerto TDF. Es posible:

- Asignar un 'time step' particular a cada módulo. En la imagen inferior, el módulo A muestrea cada 20 µs.

- Dar un determinado 'time step' a cierto puerto de un módulo perteneciente a un 'cluster'. El puerto del módulo B que se puede observar en la imagen inferior, muestrea cada 10 µs.

- Introducir un cierto ratio en un puerto de un módulo perteneciente a un 'cluster'. El ratio determina cuantas muestras se cogen en un determinado instante de tiempo. El módulo B lee 2 muestras del puerto cada vez que se activa.

- Dar un 'delay' temporal a un puerto de un módulo perteneciente a un 'cluster'. El 'delay' temporal hace que se escriban las muestras pertenecientes a tantos 'time step' anteriores como indique el 'delay'. El módulo C, cada vez que se activa, escribe en el puerto la muestra del anterior 'time step'.

- Permite asignar un 'offset' temporal a un determinado puerto de un módulo perteneciente a un 'cluster'. Este 'offset' solo se puede asignar a puertos especiales que se conectan a módulos pertenecientes al dominio de eventos discretos (SystemC). El módulo D cuenta con un offset de 1 µs.

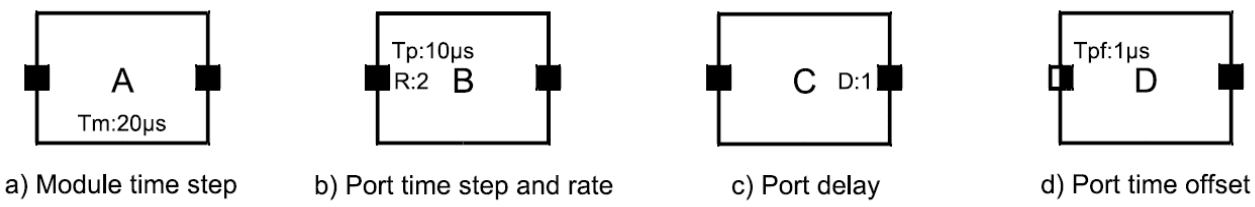

Figura 37 - Atributos de los puertos y módulos TDF

4.1.2.2 - set_attributes

Todos los atributos de cualquier módulo TDF de SystemC-AMS se definen dentro de esta función. Esta función se llama durante la fase de elaboración, al comienzo de la ejecución del programa.

De lo único que se encarga la función es de definir un tiempo de muestreo a los puertos de entrada, de acuerdo a la información extraída del manual del microcontrolador. A todos estos puertos se les otorga un 'time step' de 26 ms.

```
void ad_converter::set_attributes(){

    for(int i = 0; i < N_ANA_PINS; i++){
        in_tdf[i].set_timestep(0.26,SC_MS); // The sampling time of the A/D Converter
    }
}
```

Figura 38 - set_attributes

4.1.2.3 - processing

Esta función se ejecuta cada vez que el módulo se activa. Es necesaria en cualquier módulo TDF, ya que define el comportamiento de éste.

La tarea principal del conversor A/D es convertir los valores de tensión analógicos que tiene a la entrada en valores digitales discretos.

Lo primero que hace es comprobar qué tipo de referencia se ha fijado. Si la referencia es externa, coge del pin de referencia externa el nuevo valor máximo y si está fijado en interna, un puerto que conecta con *Sim_arduino* le dice qué valor tomar. En todos los casos, se actualiza la escala de valores.

Tras realizar esta comprobación, comienza a muestrear todos los valores posibles. Si alguno de estos valores excede el rango, ya sea por arriba o por abajo, lo sesga al valor máximo o mínimo, según corresponda.

El puerto de salida es de un tipo especial, ya que se encarga de llevar estos valores del dominio TDF al de eventos discretos de SystemC.

```
void ad_converter::processing(){

    double val;

    if(in_max_range == -1){ // External reference
        if(in_range_max != extern_ref){
            in_range_max = extern_ref;
            scaleFactor = (pow(2, 10) - 1) / (in_range_max - in_range_min);
            }
    }
    else if(in_max_range != in_range_max){ // Internal reference
        in_range_max = in_max_range;
        scaleFactor = (pow(2, 10) - 1) / (in_range_max - in_range_min);
    }

    for(int i = 0; i < N_ANA_PINS; i++){
        if(in_tdf[i].read() > in_range_max)
            val = pow(2, 10) - 1; // Clip if
        else if(in_tdf[i].read() < in_range_min)
            val = 0;
        else
            val = (in_tdf[i].read() - in_range_min) * scaleFactor; // Scale
        //cout << "A/D Converter value: " << val << ", pin: " << i << endl;
        out_de[i].write(static_cast<sc_dt::sc_uint<10> >(val));
    }
}
```

Figura 39 - processing

4.2 - arduino_pin_def.h

En esta cabecera se encuentran definidos los números de pines digitales y analógicos con que cuenta la placa, así como qué número de ellos se pueden emplear como salidas PWM o como interrupciones.

También se define el ‘offset’ que se emplea para las funciones analógicas, número que debe coincidir con el que se emplea en la librería Arduino modificada.

4.3 - Adaptar el modelo a otras placas Arduino

Si bien este modelo está pensado para la placa Arduino Mega 2560, con una serie de cambios sería posible adaptarlo fácilmente a cualquier otra placa Arduino.

Los números definidos en la cabecera *arduino_pin_def.h* habría que cambiarlos para la placa en cuestión, ya sea aumentándolos o disminuyéndolos, dependiendo de sus prestaciones. El 'offset' que se emplea para las funciones analógicas es lo suficientemente alto como para ser empleado en cualquier otra placa Arduino, por lo que no sería necesario alterarlo.

En la función *manage_interrupt* simplemente habría que cambiar los pines que pueden funcionar como interrupción, que difieren en cada placa, así como su número.

Por último, en la función *daemon_vippe* habría que hacer lo mismo con los pines que pueden funcionar como PWM, cambiando una serie de condicionales.

Los microcontroladores que montan las placas, pese a ser de la familia ATmega, tendrán ciertas diferencias que deberán verse reflejadas en la definición de la plataforma de Vippe.

4.4 - Funcionamiento

Durante este cuarto capítulo se han analizado en detalle todos los elementos que intervienen en la simulación: el simulador Vippe, la librería Arduino modificada y el modelo Arduino en SystemC-AMS. También se han mencionado varias de las interacciones que existen entre cada uno de estos elementos.

En la Figura 40 se muestra un esquema completo de cómo se relacionan todos los elementos que conforman este proyecto.

En la parte superior de la figura se encuentra en modelo Arduino en SystemC-AMS, compuesto, como ya se ha visto, de un par de sub-módulos que se encargan de la sincronización con el simulador, de generar las señales PWM, muestrear las entradas analógicas...Las entradas y las salidas del modelo (a la izquierda y a la derecha, respectivamente) se comunican con algún otro sistema o prototipo.

Vippe hace de intermediario entre el sketch Arduino del usuario (el programa que debe ejecutar la placa) y el modelo Arduino. Además, la arquitectura 'hardware' del microcontrolador se encuentra descrita en Vippe. A medida que Vippe lee instrucciones del 'sketch', se las comunica al modelo Arduino a través de una estructura de memoria compartida ya mencionada anteriormente.

Por último, la librería modificada de Arduino, permite a Vippe reconocer las funciones clave Arduino, instrumentarlas y comunicar al modelo qué debe hacer.

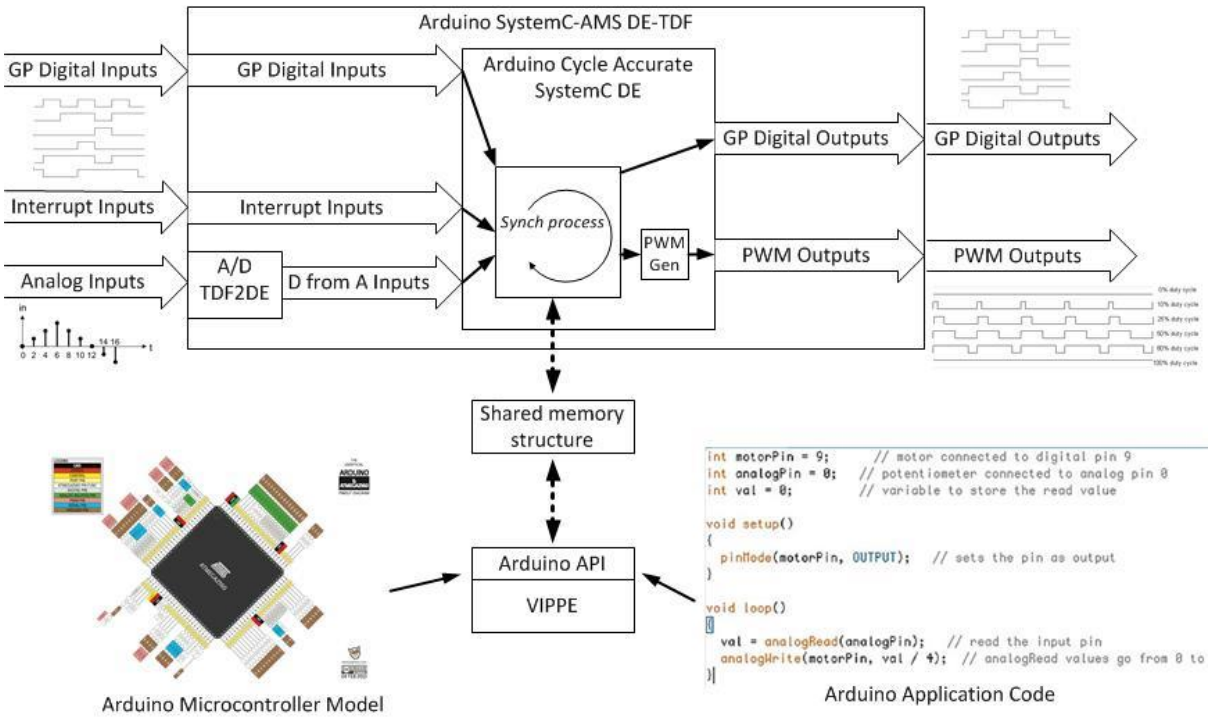

Figura 40 - Esquema del funcionamiento

4.5 - Simulación del modelo. Información generada

En este apartado, se comenta la información reportada al usuario al simular el modelo.

Por un lado, al conectar varios módulos en un fichero *main.cpp*, SystemC permite generar un archivo '.vwf' a partir de las señales conectoras o de los propios puertos de estos módulos. La función *sc_create_vcd_trace_file* permite la creación de estos ficheros y *sc_trace* les añade las trazas que considere el usuario. En la Figura 41 se ilustra un posible ejemplo.

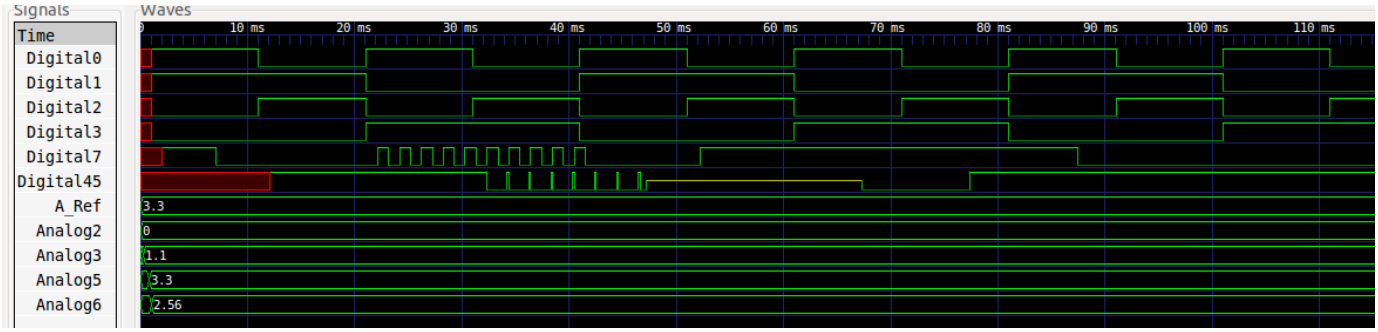

Figura 41 - Formas de onda generadas

En este caso, no se han empleado todos los puertos del modelo. Se han usado seis pines digitales, cuatro de ellos funcionando de manera normal y los dos últimos alternando entre escritura binaria y PWM.

El pin de referencia analógico lo ha fijado el usuario a 3.3 V, los valores leídos en los cuatro pines analógicos mostrados estarán entre 0 V y ese valor.

Gracias a esta representación, el usuario puede conocer cómo se han comportado todos los módulos que intervinieran durante la simulación y corregir posibles errores de diseño o de comunicación.

Por su parte, Vippe muestra toda la información que se ve en la Figura 42 al terminar la simulación. Notifica el tiempo total de simulación y hace un desglose entre elemento de memoria, el bus de datos y los procesadores (en este caso solo uno).

Se muestra el número de instrucciones ejecutadas por el procesador, el tiempo de actividad e inactividad y la energía consumida, desglosada en dos partes, la estática y la dinámica.

En los elementos de memoria y el bus de datos aparece simplemente el número de accesos y la energía consumida.

Se calcula el consumo total de energía de todos estos elementos y a continuación aparecen todos los procesos ejecutados. En este ejemplo, se han ejecutado dos, el proceso principal y una interrupción al final.

Para cada proceso, se informa de las instrucciones ejecutadas y su equivalente en ciclos de reloj, información sobre los accesos a cache y tiempos de comienzo y final.

Toda esta información generada por Vippe, puede ser de interés para el usuario, pero lo realmente interesante de este proyecto, es la capacidad de interacción del modelo SystemC generado con otros módulos descritos en el mismo lenguaje.

```
Simulation end 172727854 ns

Process Elements:
         PE 0:
         Idle time:      0 ns
         CPU use:        100.000000 %
         Instructions executed:  212446 instructions
         Energy:         172727854 (static) + 1808222658 (dynamic) = 1980950512
nJ

Channel Elements:
         Channel 0:
         Accesses:       0 accesses
         Energy:         172727854 (static) + 0 (dynamic) = 172727854 nJ

Memory Elements:
         Memory 0:
         Accesses:       0 accesses
         Energy:         172727854 (static) + 0 (dynamic) = 172727854 nJ

Total system energy:     2326406220 nJ

Process:
         Process 0:
                 Thread 0 of process 0:
                 Instructions:                   212446 instructions
                 Cycles:                         2763662 cycles
                 Instruction cache misses:       0 misses
                 Data cache hits:                107366 hits
                 Data cache misses:              0 misses
                 Data cache write backs:         0 writes
                 Start time:                     0 ns
                 End time:                       172727854 ns

                 Energy:                         212446 ns

         Process 1:
                 Thread 0 of process 1:
                 Instructions:                   0 instructions
                 Cycles:                         1 cycles
                 Instruction cache misses:       0 misses
                 Data cache hits:                0 hits
                 Data cache misses:              0 misses
                 Data cache write backs:         0 writes
                 Start time:                     120000000 ns
                 End time:                       120000062 ns

                 Energy:                         0 ns
```

Figura 42 - Resultados de Vippe

5 - Caso de uso

Para comprobar la utilidad y el buen funcionamiento del modelo, éste se va a conectar a un sistema microfluídico, descrito en SystemC-AMS. Esta descripción forma parte del trabajo desarrollado conjuntamente entre el Grupo de Ingeniería Microelectrónica, la Universidad de Zaragoza y la empresa AlphaSIP SL dentro del proyecto CATRENE HI-INCEPTION.

Se prueba nuestro modelo en un entorno multidominio, el modelo Arduino se encarga del control electrónico del prototipo microfluídico y éste de las mezclas y reacciones químicas entre líquidos.

5.1 - Prototipo microfluídico

El prototipo real cuenta con una serie de micro-conductos y micro-cámaras (conductos y cámaras, en adelante) que contienen los líquidos. Los líquidos se introducen en el sistema a través de unos tanques y dentro de las cámaras se realizan todas las mezclas y reacciones químicas. Todo ello se encuentra interconectado mediante conductos.

Existen válvulas y encaminadores que cortan, modifican el caudal o cambian de dirección en que se propagan los líquidos. Así se consigue separar fluidos en caso de que sea necesario o controlar cuales sufren reacciones o alteraciones. Las cámaras cuentan con sensores que permiten extraer información sobre la mezcla, para analizarla posteriormente.

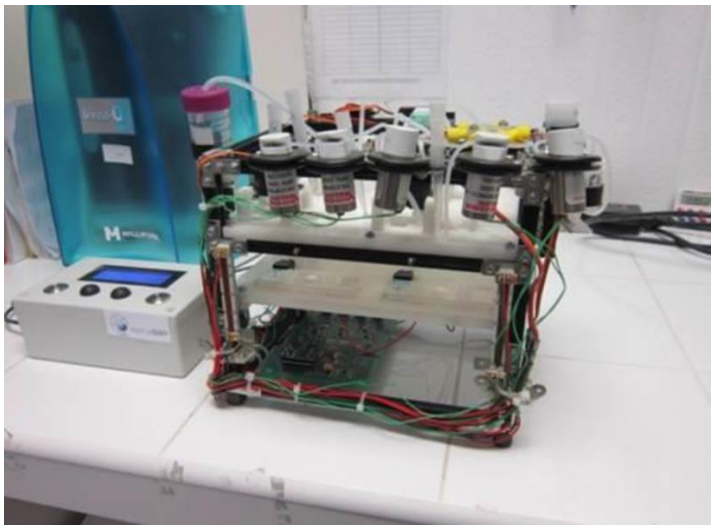

Figura 43 - Prototipo microfluídico

El modelo SystemC-AMS del prototipo empleado en las pruebas, pese a integrar todos estos componentes, es más sencillo que el real. Cada uno de estos componentes, cuenta con dos sub-módulos en SystemC-AMS, uno escrito en ELN (Electrical Linear Network) y otro en TDF (Timed Data Flow).

La parte ELN de cada componente se encarga de calcular las variaciones de presión de dicho componente teniendo en cuenta la totalidad del sistema y aplicando un modelo resistivo como el propuesto por Poiseuille. La parte TDF calcula la resistencia en función de la viscosidad, geometría y posición de los líquidos a lo largo del tiempo.

5.2 - Simulación multidominio

El modelo SystemC-AMS a simular es más simple que el que se puede observar en la Figura 43. Cuenta solo con tres entradas digitales, dos para controlar la abertura de unos tanques y otra para controlar un encaminador. El modelo cuenta también con una cámara donde los tanques vacían su contenido.

Este modelo microfluídico era probado inicialmente en un 'testbench', conectado junto con un generador de señales. El objetivo de esta simulación multidominio es conectar el modelo Arduino (sustituyendo el generador de señales) junto con el prototipo microfluídico, replicar el comportamiento del 'testbench' y finalmente comparar los resultados de esta simulación y del 'testbench'.

En la Figura 44 se puede observar el modelo Arduino conectado junto con el modelo del prototipo microfluídico.

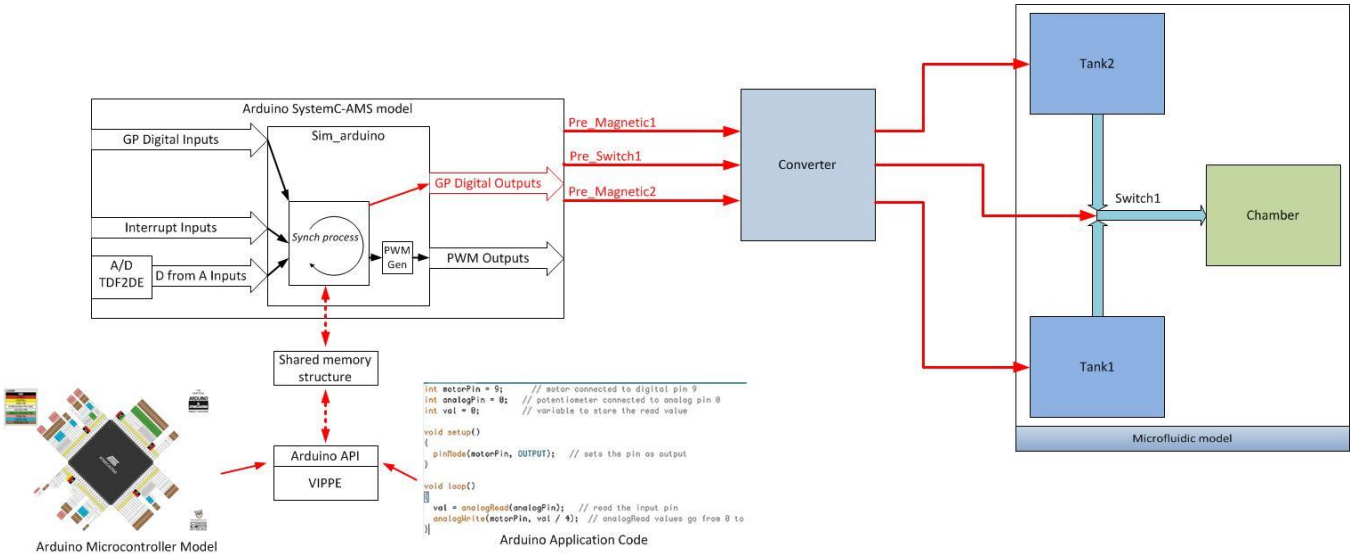

Figura 44 - Conexión de ambos modelos

Las tres señales digitales que el modelo Arduino manda hacia el modelo microfluídico son *Pre_Magnetic1* y *Pre_magnetic2* para los tanques y *Pre_Switch1* para el encaminador. Antes de llegar al otro modelo, existe un módulo que se encarga de realizar una conversión de tipos entre los puertos de los dos modelos.

```
SCA_TDF_MODULE(port_converter){

    sca_tdf::sca_out<int> fluidic_out[3];
    sca_tdf::sc_in<sc_dt::sc_logic > dig_in[3];

    void processing();

    SCA_CTOR(port_converter){
    }
};
```

```
void port_converter::processing(){

    for(int i = 0; i < 3; i++){
        if(dig_in[i].read() == 0)
            fluidic_out[i].write(0);
        else
            fluidic_out[i].write(1);
    }
}
```

Figura 45 - port_converter

En la Figura 45 se pueden observar los tipos de ambos puertos y la conversión que consiste en un condicional para cada una de las señales. La conexión de ambos modelos junto con el módulo conversor se realiza en un fichero *main.cpp.*

Conectados ambos modelos, es necesario extraer el patrón que sigue el generador de señales del 'testbench' del modelo microfluídico, para replicarlo mediante un 'sketch'. El patrón consiste en una sucesión de cambios de nivel lógico en las tres señales digitales cada 5 segundos. El 'sketch' generado consiste en varias escrituras digitales con esperas entre medias, como se ve en la Figura 46. Las formas de onda replicadas con este 'sketch', idénticas a las creadas por el generador del 'testbench', aparecen en la Figura 47.

```
#include "uc_Arduino.h"

const byte Tank1 = 0;
const byte Tank2 = 1;
const byte Switch1 = 2;

void setup(){

    pinMode(Tank1, OUTPUT);
    pinMode(Tank2, OUTPUT);
    pinMode(Switch1, OUTPUT);

    return;
}

void loop(){

    digitalWrite(Tank1, 1);
    digitalWrite(Tank2, 0);
    digitalWrite(Switch1, 1);
    delay(5000);
    digitalWrite(Tank1, 0);
    digitalWrite(Tank2, 1);
    digitalWrite(Switch1, 1);
    delay(5000);
    digitalWrite(Tank1, 1);
    digitalWrite(Tank2, 0);
    digitalWrite(Switch1, 0);
    delay(5000);
    digitalWrite(Tank1, 0);
    digitalWrite(Tank2, 1);
    digitalWrite(Switch1, 0);
    delay(5000);
    digitalWrite(Tank1, 1);
    digitalWrite(Tank2, 0);
    digitalWrite(Switch1, 1);
    digitalWrite(Tank1, 0);
    digitalWrite(Tank2, 1);
    digitalWrite(Switch1, 0);
    delay(2000);

    return;
}
```

Figura 46 - 'Sketch' de la simulación

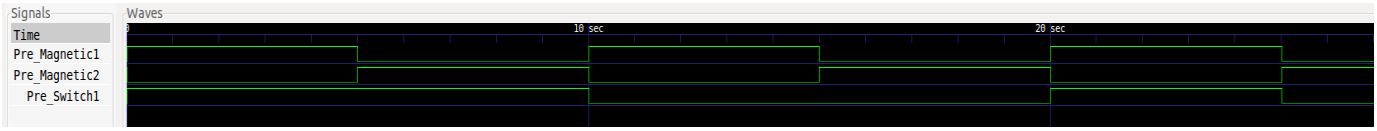

Figura 47 - Formas de onda

Al realizar la simulación con Vippe y el modelo Arduino conectado, los resultados temporales obtenidos son idénticos a los obtenidos al ejecutar el 'testbench' del prototipo.

```
TANK2 L:698.621m r:0.000675m Time:6802 ms
============================
    WATER2
  Volume: 7.49107e-07 mÂ³  Presion: 1127.69   Resistor: 8.58686e+
12
============================

PIPE-2 L:0.02316 m  r:0.000233083 m  Time:6802 ms
============================================================
========================
    WATER2: 0.00523381 m   Magnetics: 0| AIR: 0.0179262 m
     Magnetics: 0|
============================================================
========================
  Pressure drop:2.62939 Pa  Flow:1.31327e-10 mÂ³/s
Speed:0.000769452 m/s  Drag:2.00217e+10

TANK1 L:698.621m r:0.000675m Time:6802 ms
============================
     WATER1
  Volume: 7.38722e-07 mÂ³  Presion: 14237.3   Resistor: 8.58686e+
12
============================

PIPE-1 L:0.02316 m  r:0.000233083 m  Time:6802 ms
============================================================
========================
    WATER1: 0.02316 m      Magnetics: 1.77333e+08|
============================================================
========================
  Pressure drop:33.1968 Pa  Flow:1.65804e-09 mÂ³/s
Speed:0.00971454 m/s  Drag:2.00217e+10

PIPE-3 L:0.00234 m  r:0.000764327 m  Time:6802 ms
============================================================
========================
    WATER1: 0.00234 m      Magnetics: 1.77333e+08|
============================================================
========================
  Pressure drop:0.0313058 Pa  Flow:1.78945e-09 mÂ³/s
Speed:0.000975012 m/s  Drag:1.74947e+07

CHAMBER-1 L:0.01 m  r:0.00103998 m  Time:6802 ms
============================================================
========================
    WATER1: 0.0010633 m    Magnetics: 1.77333e+08 millions of
particles/l | AIR: 0.0089367 m  Magnetics: 0 millions of
particles/l |
============================================================
========================
  Pressure drop:0.0390326 Pa  Flow:1.78944e-09 mÂ³/s
Speed:0.000526646 m/s  Magnetic contents:0 millions of
particles/l  Drag:2.18127e+07
```

```
TANK2 L:698.621m r:0.000675m Time:6802 ms
============================
    WATER2
  Volume: 7.49107e-07 mÂ³  Presion: 1127.69   Resistor:
8.58686e+12
============================

PIPE-2 L:0.02316 m  r:0.000233083 m  Time:6802 ms
============================================================
========================
    WATER2: 0.00523381 m   Magnetics: 0| AIR: 0.0179262 m
     Magnetics: 0|
============================================================
========================
  Pressure drop:2.62939 Pa  Flow:1.31327e-10 mÂ³/s
Speed:0.000769452 m/s  Drag:2.00217e+10

TANK1 L:698.621m r:0.000675m Time:6802 ms
============================
     WATER1
  Volume: 7.38722e-07 mÂ³  Presion: 14237.3   Resistor:
8.58686e+12
============================

PIPE-1 L:0.02316 m  r:0.000233083 m  Time:6802 ms
============================================================
========================
    WATER1: 0.02316 m      Magnetics: 1.77333e+08|
============================================================
========================
  Pressure drop:33.1968 Pa  Flow:1.65804e-09 mÂ³/s
Speed:0.00971454 m/s  Drag:2.00217e+10

PIPE-3 L:0.00234 m  r:0.000764327 m  Time:6802 ms
============================================================
========================
    WATER1: 0.00234 m      Magnetics: 1.77333e+08|
============================================================
========================
  Pressure drop:0.0313058 Pa  Flow:1.78945e-09 mÂ³/s
Speed:0.000975012 m/s  Drag:1.74947e+07

CHAMBER-1 L:0.01 m  r:0.00103998 m  Time:6802 ms
============================================================
========================
    WATER1: 0.0010633 m    Magnetics: 1.77333e+08 millions of
particles/l | AIR: 0.0089367 m  Magnetics: 0 millions of
particles/l |
============================================================
========================
  Pressure drop:0.0390326 Pa  Flow:1.78944e-09 mÂ³/s
Speed:0.000526646 m/s  Magnetic contents:0 millions of
particles/l  Drag:2.18127e+07
```

Figura 48 - Comparación de resultados

Todos los componentes imprimen en pantalla su estado a medida que avanza el tiempo de simulación. Indican cambios de presión, volumen del líquido, flujo y velocidad del líquido, cantidad de aire…Todos estos reportes, junto con el tiempo en que se producen, son idénticos en ambas simulaciones, lo que indica que el modelo Arduino se comporta correctamente.

En la Figura 48 aparecen, como ejemplo, los resultados en un determinado instante de tiempo, a la izquierda los obtenidos con el 'testbench' y a la derecha con el modelo Arduino conectado, idénticos en ambos casos.

6 - Conclusiones

El modelo Arduino generado cumple con todos los requisitos que se fijaron en un primer momento y es válido para el análisis y la simulación de cualquier sistema que cuente con la placa en cuestión. Además, con una serie de mínimos cambios, podría adaptarse el modelo a cualquier otra placa Arduino.

A continuación, se resumen las principales características del proyecto, cumpliendo con todos los objetivos marcados:

- Modelo en SystemC-AMS de la placa Arduino Mega 2560. El modelo cuenta con entradas/salidas digitales e interrupciones. La extensión AMS permite introducir también entradas analógicas.

- SystemC favorece la interconexión entre diversos sistemas, lo cual es idóneo para este tipo de proyectos. El usuario conoce en todo momento de la simulación qué valores tienen las entradas y las salidas de su modelo y del modelo Arduino.

- Las principales funciones de la librería Arduino se han modificado para funcionar durante la simulación con Vippe, sin tener que realizar cambios profundos en los ‘sketches’ del usuario.

- Al ser Vippe una herramienta desarrollada por el Grupo de Ingeniería Microelectrónica de la Universidad de Cantabria, todas las futuras mejoras que esta herramienta pueda recibir en el futuro, mejorarán el rendimiento de este proyecto.

- La simulación con Vippe es rápida y lo suficientemente precisa como para obtener estimaciones temporales y de consumo relevantes en etapas tempranas del desarrollo de sistemas.

- Como línea futura, cabría destacar la posibilidad de simular el modelo junto con un sistema que contara con realimentación digital/analógica, de tal forma que el ‘sketch’ Arduino pudiera variar su ejecución en función de los valores recibidos.

7 - Referencias consultadas

Las fuentes (libros, artículos, páginas web…) que se han consultado para elaborar el proyecto son las siguientes:

- Black C. & Donovan J. (2004). *"SystemC From The Ground Up"*
- Díaz L., González E., Villar E., Sánchez P. (2014). *"VIPPE: Native simulation and performance analysis framework for multi-processing embedded systems"*
- Fernández V., Mena A., Ben Aoun C., Pêcheux F., Fernández L. J. "Virtual prototyping of pressure driven microfluidic systems with SystemC-AMS extensions" http://www.sciencedirect.com/science/article/pii/S0141933115001064
- ATMEL ATmega 2560 Datasheet
- Manual de referencia del modelo SystemC-AMS del prototipo microfluídico
- Estándar de los lenguajes SystemC y SystemC-AMS. Manual de referencia de SystemC-AMS: http://accellera.org/downloads/standards/systemc
- Referencia sobre la placa Arduino Mega 2560: https://www.arduino.cc/en/Main/arduinoBoardMega2560
- Referencia sobre el lenguaje Arduino: https://www.arduino.cc/en/Reference/HomePage