



UNIVERSIDAD DE BUENOS AIRES

Facultad de Ingeniería

Carrera de Especialización en Sistemas Embebidos

Desarrollo de Firmware y Software para programar la CIAA en lenguaje JAVA con aplicación en entornos Industriales

Alumno: Ing. Eric Nicolás Pernia.

Director: MSc. Ing. Félix Gustavo E. Safar.

Buenos Aires, Argentina.

Presentación:
Noviembre de 2015

Desarrollo de Firmware y Software para programar la CIAA en lenguaje JAVA con aplicación en entornos Industriales por Ing. Eric Nicolás Pernia se distribuye bajo una **Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional**. Para ver una copia de esta licencia, visita <http://creativecommons.org/licenses/by-sa/4.0/>.



RESUMEN

El propósito de este Trabajo Final es la incorporación de nuevas tecnologías en ambientes industriales mediante el desarrollo de arquitecturas novedosas de sistemas embebidos. En particular, permitir crear aplicaciones *Real-Time* para entornos industriales, utilizando un lenguaje de programación orientado a objetos (en adelante POO), sobre la Computadora Industrial Abierta Argentina (CIAA). Además, se espera acercar a programadores informáticos a la disciplina de programación de sistemas embebidos, permitiéndoles aplicar técnicas avanzadas de programación.

Para llevarlo a cabo se ha escogido Java como lenguaje POO, y HVM¹, que es un entorno de ejecución de *Safety Critical Java*² (SCJ)[2], de código abierto, diseñado para plataformas embebidas de bajos recursos. Este trabajo consiste entonces, en la implementación y validación de un ambiente de Firmware y Software, basado en HVM, para programar las plataformas CIAA-NXP y EDU-CIAA-NXP en lenguaje Java SCJ.

Fundamentalmente, la implementación consiste en:

- La realización del *port* de la máquina virtual de HVM para que corra sobre el microcontrolador NXP LPC4337, que contienen las plataformas CIAA-NXP y EDU-CIAA-NXP, permitiendo la programación de aplicaciones Java.
- Un diseño e implementación de una biblioteca con API³ sencilla para permitir controlar el Hardware desde una aplicación Java, que funciona además, como HAL⁴.
- El *port* de la capa SCJ de la máquina virtual de HVM, para desarrollar aplicaciones Java SCJ.
- La integración manual del *port* para la CIAA al IDE de HVM y la descripción de los pasos necesarios para llevar a cabo un proyecto con HVM.

Para validar el desarrollo se presentan:

- Ejemplos de aplicaciones Java utilizando periféricos de la CIAA-NXP y EDU-CIAA-NXP mediante la biblioteca desarrollada.
- Un ejemplo de aplicación Java SCJ utilizando el concepto de Proceso SCJ para demostrar el funcionamiento del cambio de contexto.
- Otro ejemplo de aplicación Java SCJ que usa un Planificador SCJ.
- Una aplicación SCJ completa.

¹Siglas de *Hardware near Virtual Machine*, desarrollo de Stephan Erbs Korsholm, Dinamarca.

²La especificación *Safety Critical Java* es una extensión a la especificación RTSJ, una especificación de java para aplicaciones en tiempo real.

³*Application Programming Interface*, es decir, una interfaz de programación de aplicaciones.

⁴*Hardware Abstraction Layer*, significa: capa de abstracción de hardware.

En conclusión, se obtiene de este Trabajo Final un entorno de desarrollo para aplicaciones Java SCJ sobre las plataformas CIAA-NXP y EDU-CIAA-NXP, que además de ser software libre, cubre las necesidades planteadas, tanto al ofrecer programación orientada a objetos, así como funcionalidades de tiempo real para entornos industriales, sobre sistemas embebidos.

Agradecimientos

Índice general

1. INTRODUCCIÓN GENERAL	1
1.1. Marco temático: Programación Orientada a Objetos en Sistemas embebidos para aplicaciones industriales	1
1.1.1. Proyecto CIAA	1
1.1.2. Lenguajes de POO para sistemas embebidos	5
1.1.3. Especificaciones RTSJ y SCJ	7
1.1.4. Máquinas Virtuales de Java para aplicaciones de tiempo real	10
1.2. Justificación	11
1.3. Objetivo	11
2. DESARROLLO	13
2.1. HVM (<i>Hardware near Virtual Machine</i>)	13
2.1.1. Obtención de un IDE para desarrollar programas Java sobre HVM	13
2.1.2. Utilización de HVM	14
2.1.3. Consideraciones a tener en cuenta al utilizar HVM	17
2.1.4. Características de HVM	19
2.2. Port de HVM a una nueva plataforma de Hardware	21
2.2.1. Port de HVM para ejecutar Java	21
2.2.2. Port de HVM para ejecutar Java SCJ	22
2.3. Diseño de biblioteca para el manejo de periféricos desde Java	24
2.3.1. Modelo de la biblioteca Java	25
2.3.2. Mapeo de pines de las plataformas	27
2.3.3. Modelo de la biblioteca en C	28
2.4. Integración del desarrollo	29
3. IMPLEMENTACIÓN	31
3.1. Arquitectura del <i>port</i> de HVM para las plastaformas CIAA	31
3.2. Port básico de HVM al microcontrolador NXP LPC4337	32
3.3. Arquitectura del <i>port</i> de HVM para las plastaformas CIAA	36
3.4. Implementación de la biblioteca para manejo de periféricos	36
3.4.1. CIAA-NXP	37
3.4.2. EDU-CIAA	37
3.4.3. Java	37
3.5. Port de HVM SCJ al microcontrolador NXPLPC4337	37
3.5.1. Funciones para SCJ	37
3.6. Integración y uso de las herramientas desarrolladas	37
4. VALIDACIÓN Y RESULTADOS	39
4.1. Ejemplos de aplicaciones Java utilizando periféricos	39
4.2. Ejemplo de Procesos SCJ	40
4.3. Planificador SCJ	41

4.4. Ejemplo de aplicación SCJ completa	41
5. CONCLUSIONES Y TRABAJO A FUTURO	43
5.1. Conclusiones	43
5.2. Trabajo a futuro	44

Índice de figuras

1.1.	Plataforma CIAA-NXP	4
1.2.	Plataforma EDU-CIAA-NXP	5
1.3.	Concepto de misión SCJ	9
1.4.	Modelo de memoria SCJ	10
2.1.	Programa Hola Mundo con HVM	14
2.2.	Grado de dependencia del programa Hola Mundo con HVM	15
2.3.	Esquema de funcionamiento del IDE para trabajar con HVM sobre sistemas embebidos	16
2.4.	Ejemplo Hola mundo con HVM en Cygwin	17
2.5.	Cambiar un método a modo compilado	18
2.6.	Modelo de capas de HVM	21
2.7.	Estructura en caps del Firmware de la CIAA	25
2.8.	Modelo de Periférico	26
2.9.	Modelo de Dispositivo y Pin	27
2.10.	Mapeo de pines de la plataforma CIAA-NXP	27
2.11.	Mapeo de pines de la plataforma EDU-CIAA-NXP	28
3.1.	Arquitectura del port de HVM para las plastaformas CIAA	31
3.2.	Estructura de proyecto de firmware CIAA “Hola Mundo con HVM”	36

Índice de tablas

Capítulo 1

INTRODUCCIÓN GENERAL

1.1. Marco temático: Programación Orientada a Objetos en Sistemas embebidos para aplicaciones industriales

Con la creciente complejidad de las aplicaciones a realizar sobre sistemas embebidos en entornos industriales, y el aumento de las capacidades de memoria y procesamiento de los mismos, se desea poder aplicar técnicas avanzadas de diseño de software para realizar programas fácilmente mantenibles, extensibles y reconfigurables. El paradigma de Programación Orientada a Objetos (POO) cumple con estos requisitos. La aplicación de este paradigma abre las puertas a que programadores informáticos se acerquen a la disciplina de programación de sistemas embebidos, permitiéndoles aplicar técnicas avanzadas de programación. Un ejemplo donde es muy eficiente su utilización, es en la programación de sistemas donde existan recetas cambiantes para realizar el mismo proceso.

Para aplicaciones industriales, es requerimiento fundamental cumplir con especificaciones temporales. En consecuencia, se necesita un lenguaje POO que soporte de manejo de *threads real-time*. El ejemplo más ilustrativo de este requerimiento es el de una aplicación de control a lazo cerrado. En este dominio es necesario garantizar una tasa de muestreo periódica uniforme para poder aplicar la teoría de control automático, suficientemente rápida para que permita seguir la dinámica del sistema (f_{min} ¹) y suficientemente lenta para que permita calcular el algoritmo de control y actualizar la salida entre dos muestras (f_{max} ²). A lazo cerrado, finalmente, es el controlador quien impone la frecuencia del sistema, que corresponde a un parámetro de diseño a elegir en el rango entre f_{min} y f_{max} .

En la sección [1.1.1] se introduce el proyecto CIAA y sus distintas plataformas. Seguidamente, en la sección [1.1.2] se exponen las ventajas y desventajas de distintos lenguajes POO estudiados para la programación de sistemas embebidos, de los cuales se selecciona Java. Luego, en la sección [1.1.3] se introducen dos especificaciones de Java para aplicaciones de tiempo real. Finalmente, en la sección [1.1.4] se exponen distintas máquinas virtuales de Java para aplicaciones *real-time*, concluyendo en la elección de HVM.

1.1.1. Proyecto CIAA

El proyecto de la Computadora Industrial Abierta Argentina (CIAA) nació en 2013 como una iniciativa conjunta entre el sector académico y el industrial, representados por la ACSE³ y CADIEEL⁴, respectivamente.

¹Frecuencia mínima para asegurar la reconstrucción de la señal según el Teorema de muestreo de Nyquist.

²Frecuencia máxima impuesta por la duración del algoritmo de control.

³Asociación Civil para la investigación, promoción y desarrollo de los Sistemas electrónicos Embebidos. Sitio web: <http://www.sase.com.ar/asociacion-civil-sistemas-embebidos>

⁴Cámara Argentina de Industrias Electrónicas, Electromecánicas y Luminotécnicas. Sitio web: <http://www.cadieel.org.ar/>

Los objetivos del proyecto CIAA son:

- Impulsar el desarrollo tecnológico nacional, a partir de sumar valor agregado al trabajo y a los productos y servicios, mediante el uso de sistemas electrónicos, en el marco de la vinculación de las instituciones educativas y el sistema científico-tecnológico con la industria.
- Darle visibilidad positiva a la electrónica argentina.
- Generar cambios estructurales en la forma en la que se desarrollan y utilizan en nuestro país los conocimientos en el ámbito de la electrónica y de las instituciones y empresas que hacen uso de ella.

Todo esto en el marco de un trabajo libre, colaborativo y articulado entre industria y academia.

Con esta iniciativa, se han desarrollado en la actualidad varias plataformas de hardware y entornos de programación para utilizarlas.

Al momento de la presentación de este trabajo, existen dos versiones de la plataforma CIAA cuyo desarrollo ha sido completado:

- CIAA-NXP, basada en el microcontrolador NXP LPC4337, que ya se comercializa.
- CIAA-FSL, que utiliza, en cambio, el microcontrolador Freescale MK60FX512VLQ15, pero únicamente hay prototipos de esta plataforma.

Además, existe una versión educativa de bajo costo de la CIAA-NXP, nombrada EDU-CIAA-NXP, que ya se distribuyeron alrededor de 1000 unidades y ya hay otras 1000 reservadas en producción.

Debido a estas razones, el trabajo se enfoca en el desarrollo de herramientas para programar las dos plataformas basadas en el microcontrolador NXP LPC4337. Se introducen a continuación las características de las mismas.

Plataforma CIAA-NXP

La CIAA-NXP es la primera y única computadora del mundo que reúne dos cualidades:

- Ser **Industrial**, ya que su diseño está preparado para las exigencias de confiabilidad, temperatura, vibraciones, ruido electromagnético, tensiones, cortocircuitos, etc., que demandan los productos y procesos industriales.
- Ser **Abierta**, ya que toda la información sobre su diseño de hardware, firmware, software, etc. está libremente disponible en Internet bajo la Licencia BSD, para que cualquiera la utilice como quiera.

Esta plataforma se compone de:

- CPU: Microcontrolador NXP LPC 4337 JDB 144 (Dual-core Cortex-M4 + Cortex-M0 @ 204MHz).
- Debugger: USB-to-JTAG FT2232H. Soportado por OpenOCD.
- Memorias:
 - IS42S16400F - SDRAM. 64Mbit @ 143MHz.
 - S25FL032P0XMFI011 - Flash SPI. 32 Mbit, Quad I/O Fast read: 80 MHz.

- 24AA1025 - EEPROM I2C. 1 Mbit, 400 kHz. Almacenamiento de propósito general, datos de calibración del usuario, etc.
- 24AA025E48 - EEPROM I2C. 2 kbit, 400 kHz. Para implementación de MAC-Address o almacenamiento de propósito general.
- Entradas y salidas:
 - 8 entradas digitales opto-aisladas 24VDC.
 - 4 Entradas analógicas 0-10V/4-20mA.
 - 4 salidas Open-Drain 24VDC.
 - 4 Salidas con Relay DPDT.
 - 1 Salida analógica 0-10V/4-20mA.
- LV-GPIO:
 - 14 GPIOs.
 - I2C.
 - SPI.
 - 4 canales analógicos.
 - Aux. USB.
- Interfaces de comunicación:
 - Ethernet.
 - USB On-The-Go.
 - RS232.
 - RS485.
 - CAN.
- Múltiples fuentes de alimentación.

En la figura [1.1] se muestra una fotografía de la plataforma.

Plataforma EDU-CIAA-NXP

La plataforma EDU-CIAA-NXP es un desarrollo colaborativo, realizado por miembros de la Red Universitaria de Sistemas Embebidos (RUSE), en el marco del Proyecto CIAA. RUSE se compone de docentes pertenecientes a más de 60 Universidades a lo largo y a lo ancho del país.

Los propósitos de la plataforma son:

- Proveer una plataforma de desarrollo moderna, económica y de fabricación nacional basada en la CIAA-NXP, que sirva a docentes y a estudiantes en los cursos de sistemas embebidos.
- Lograr una amplia inserción en el sistema educativo argentino.
- Realizar un aporte eficaz al desarrollo de vocaciones tempranas en electrónica, computación e informática.
- Demostrar que las universidades argentinas son capaces de realizar un desarrollo colaborativo exitoso en el área de los sistemas embebidos, cumpliendo con requerimientos de tiempo y forma.



Figura 1.1: Plataforma CIAA-NXP.

Características de la EDU-CIAA-NXP:

- CPU: Microcontrolador NXP LPC 4337 JDB 144 (Dual-core Cortex-M4 + Cortex-M0 @ 204MHz).
- Debugger: USB-to-JTAG FT2232H. Soportado por OpenOCD.
- 2 puertos micro-USB (uno para aplicaciones y debug, otro OTG).
- 6 salidas digitales implementadas con leds (3 normales y uno RGB).
- 4 entradas digitales con pulsadores.
- 1 puerto de comunicaciones RS-485 con bornera.
- 2 conectores de expansión:
 - P0:
 - 3 entradas analógicas (ADC0 a ADC2).
 - 1 salida analógica (DAC0).
 - 1 conexión para un teclado de 3 x 4.
 - 12 pines genéricos de I/O.
 - P1:
 - 1 puerto Ethernet.
 - 1 puerto CAN.
 - 1 puerto SPI.
 - 1 puerto I2C.
 - 12 pines genéricos de I/O.

En la figura [1.2] se muestra una fotografía de esta plataforma.

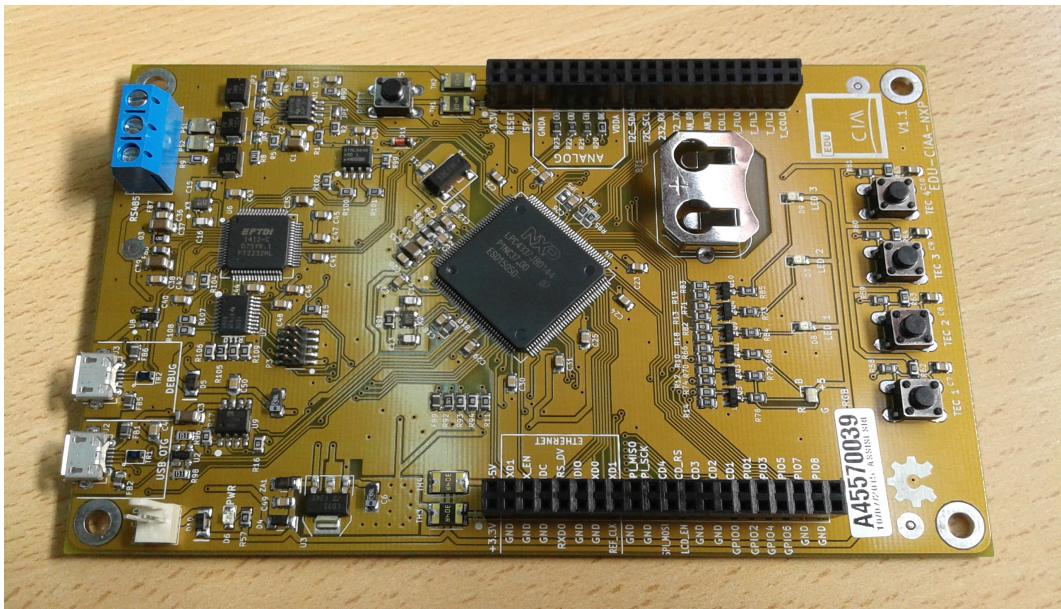


Figura 1.2: Plataforma EDU-CIAA-NXP.

Siendo el autor participante de este proyecto desde mediados de 2014, ocupando el rol de Responsable de Software-PLC mediante el aporte al proyecto CIAA de un IDE⁵ que permite programar esta plataforma con lenguajes de PLC industriales (IEC-661131-3), se desea agregar en esta oportunidad la posibilidad de programar a esta plataforma con un lenguaje de programación orientado a objetos mediante el desarrollo de un IDE para tal fin.

1.1.2. Lenguajes de POO para sistemas embebidos

En la actualidad existen muchos desarrollos de lenguajes de programación orientado a objetos de propósito general para sistemas embebidos. Puntualmente se han evaluado las siguientes alternativas:

- C++.
- Java.
- Python.

Lenguaje C++

El lenguaje de programación C++ se encuentra disponible para la mayoría de los sistemas embebidos del mercado. Básicamente, todo embebido que dispone de un compilador de C, trae además, un compilador de C++. En este lenguaje se pueden manejar las interrupciones de un microcontrolador a través de funciones en C embebidas en el código C++.

No tiene soporte de manejo de *threads* en el lenguaje, sino que debe programarse desde cero una aplicación que resuelva la concurrencia de procesos a bajo nivel.

Es un lenguaje estáticamente tipado, es decir, cada variable debe ser declarada con un tipo, esto implica una ventaja para el programador ya que pueden detectarse en tiempo de compilación muchos errores por incompatibilidad de tipos de datos.

⁵IDE4PLC. Sitio web: <http://proyecto-ciaa.com.ar/devwiki/doku.php?id=desarrollo:software-plc>

Si bien aplica los conceptos principales que debe tener un lenguaje orientado a objetos, el mismo es considerado obsoleto por ingenieros informáticos debido a que arrastra muchos conceptos de C que lo vuelven inseguro, como por ejemplo, permite manejar la memoria sin ninguna protección a través de punteros, una fuente habitual de errores. El manejo de memoria manual se extiende a la eliminación explícita de objetos, cuya responsabilidad recae sobre el programador. El tiempo que tarda la creación y destrucción de un objeto es variable, esto es una desventaja para aplicaciones de tiempo real. Tampoco incluye características modernas de lenguajes como, por ejemplo, bloques (*closures*), o sintaxis simplificada para recorrer colecciones.

Este lenguaje POO está disponible para utilizarse actualmente en las plataformas CIAA.

Lenguaje Python

Este lenguaje posee características modernas, entre ellas Garbage Collector, que es un proceso que se encarga de detectar que objetos en memoria no se utilizan y los borra automáticamente, liberando al programador de esta tarea. Sin embargo, al igual que el caso anterior es una desventaja para aplicaciones de tiempo real pues la duración de su ejecución no es determinista. No utiliza punteros, posee solamente referencias. A diferencia de C++, Python es interpretado en lugar de compilado.

Es un lenguaje dinámicamente tipado, es decir una variable puede cambiar su tipo de datos según lo que contenga en cada momento, constituyendo una ventaja aparente para el programador al escribir su programa, pero los errores de incompatibilidad de datos solo se darán en tiempo de ejecución, dando más responsabilidad al programador para la detección de errores. Si bien en desarrollos unipersonales esto no es determinante, no se recomienda para grandes proyectos donde existan muchos programadores distribuidos.

El lenguaje Python posee soporte para el manejo de procesos, pero no se han encontrado especificaciones de soporte de procesos *real-time*.

MicroPython es una implementación de un intérprete de lenguaje Python para sistemas embedidos. Durante el tiempo de realización de este trabajo, un grupo perteneciente al proyecto CIAA se portó este intérprete para poder ser utilizado sobre la plataforma EDU-CIAA-NXP. Sin embargo, el mismo no se recomienda para aplicaciones industriales.

Lenguaje Java

El lenguaje Java, uno de los lenguajes de programación más utilizados en la actualidad. Realiza un balance entre las mejores características de los dos anteriores y además agrega algunas propias.

Java tiene aspectos que lo hace más robusto y seguro, entre ellos, una especificación del lenguaje (JLS) que es independiente de cualquier implementación, y ayuda que existan diferentes implementaciones en muchas arquitecturas totalmente compatibles; todos los accesos al hardware son a través de la Máquina Virtual de Java (JVM), que no permite los accesos ilegales a zonas de memoria y ha sido diseñado para ser seguro para trabajar en red.

Para lograr la independencia de la máquina, Java posee la característica de ser un lenguaje compilado e interpretado. Todo programa en Java, se compila primero a un lenguaje similar a un *assembler* genérico basando en pila (*bytecodes*), que luego es interpretado por la JVM, dependiente de la plataforma.

La JVM es habitualmente un programa que corre sobre un sistema operativo, sin embargo, existen implementaciones de la JVM que corren directamente sobre el hardware (*bare-metal*) y procesadores capaces de ejecutar *bytecodes* de Java directamente (por ejemplo, el microcontrolador ARM926EJ-S). Si bien es interpretado al igual que Python, se disponen de muchas implementaciones de la JVM para distintas plataformas, no siendo este el caso de los intérpretes de Python.

Posee comprobación estricta de tipos, como C++. Manejo de memoria automático mediante Garbage Collector y utiliza referencias al igual que Python. Además, permite programación concurrente de forma estándar y existen varias especificaciones de Java para aplicaciones de tiempo real.

En consecuencia, por todas las razones expuestas, se elige Java como lenguaje POO para el presente trabajo. Se introducen a continuación las especificaciones de Java RTSJ y SCJ.

1.1.3. Especificaciones RTSJ y SCJ

En Java existen varias descripciones del lenguaje pensadas para la implementación *threads real-time*, mitigando los puntos de desventaja de Java para la programación de aplicaciones industriales. Una de ellas es la especificación RTSJ que contempla aplicaciones *Real-Time*, otra es *Predictable Java* (PJ), un subconjunto de RTSJ que agrega algunos conceptos. Esta última se ha utilizado como inspiración para SCJ, la cual agrega conceptos de sistemas críticos y seguridad funcional. Se describen a continuación las especificaciones RTSJ y SCJ.

Especificación RTSJ

La Especificación de Tiempo Real para Java (RTSJ), o JSR 1, indica cómo un sistema Java debería comportarse en un contexto de tiempo real. Fue desarrollada durante varios años por expertos de Java y de aplicaciones en tiempo real.

Está diseñada para extender naturalmente cualquiera de las plataformas de la familia Java (Java, Java SE, Java EE, Java Micro Edition, etc.), y tiene el requerimiento de que cualquier implementación debe pasar el *Test de Compatibilidad JSR 1* (TCK) y el TCK propio de la plataforma en la cual está basada.

RTSJ introduce varias características nuevas para soportar operaciones en tiempo real. Estas características incluyen nuevos tipos de *thread*, nuevos modelos de gestión de memoria, y nuevos *frameworks*.

Modela una aplicación de tiempo real como un conjunto de tareas, cada una de las cuales tiene una meta de tiempo opcional. Esta meta especifica cuando debe ser completada la tarea. Las tareas de tiempo real se pueden agrupar en varias categorías, basadas en cómo el desarrollador puede predecir su frecuencia y ejecución:

- **Periódicas:** tareas que se ejecutan repetitivamente a una frecuencia fija.
- **Esporádicas:** tareas que no se ejecutan en una frecuencia fija, pero que tienen una frecuencia máxima.
- **Aperiódicas:** tareas cuya frecuencia y ejecución no pueden predecirse.

RTSJ utiliza información de los tipos de tarea para asegurar que las tareas críticas no infrinjan sus metas temporales. Permite asociarle a cada tarea un *Handler* de Meta Incumplida, de manera que una tarea no se completa antes de su meta de tiempo, se invoca al *handler* asociado para poder tomar medidas al respecto.

Define la gestión de prioridades de los *threads* con al menos 28 niveles de prioridad. Para evitar la inversión de prioridades utiliza herencia de prioridades para su gestión.

Brinda diversas formas de reservar memoria para objetos. Los objetos pueden asignarse a un área de memoria específica. Estas áreas tienen diferentes características de *garbage collector* y límites de reserva. Se clasifican en:

- **Heap estándar.** Como cualquier máquina virtual, RTJS mantiene un *heap* con *garbage collector* para que sea utilizado por cualquier tipo de tarea (*real-time* o no).
- **Memoria inmortal.** Un área de memoria que no tiene un *garbage collector*, cuyo uso lo debe gestionar el programador.
- **Memoria de ámbito.** Sólo disponible para *threads* de tiempo real (RTT⁶ y NHRT⁷). Estas áreas de memoria están pensadas para objetos con un tiempo de vida conocido. Al igual que la anterior no posee *garbage collector*.

Especificación SCJ

La especificación *Safety-Critical Java*, (JSR-302), es un subconjunto de la especificación RTSJ, que además, define un conjunto de servicios diseñados para ser utilizados en aplicaciones que requieran un nivel de certificación de seguridad funcional. La especificación está dirigida a una amplia variedad de paradigmas de certificación muy exigentes, tales como los requisitos de seguridad crítica DO-178B, Nivel A.

La misma presenta un conjunto de clases Java que implementan soluciones *Safety-Critical* para el inicio de la aplicación, concurrencia, planificación, sincronización, entrada/salida, gestión de memoria, gestión de temporización, procesamiento de interrupciones, interfaces nativas y excepciones. Presenta un conjunto de *annotations* que pueden ser utilizadas para garantizar que la aplicación exhibe ciertas propiedades de seguridad funcional, mediante comprobación estática, para mejorar la certificación de aplicaciones construidas para ajustarse a esta especificación.

Para aumentar la portabilidad de las aplicaciones *Safety-Critical* entre distintas implementaciones de esta especificación, se enumera un conjunto mínimo de bibliotecas Java que deben ser proporcionados en una implementación conforme a la especificación.

Modelo de programación SCJ

En esta especificación solo se permiten ***Threads Real-Time*** a diferencia de RTSJ. Un programa SCJ se organiza en **Misiones**⁸. Una misión encapsula una funcionalidad específica, o una fase, en el tiempo de vida de del sistema en tiempo real como un conjunto de **entidades planificables**⁹. Por ejemplo, un sistema de control de vuelo puede estar compuesto de despegue, crucero y aterrizaje; pudiendo dedicarse a cada una una misión. Una entidad planificable maneja una funcionalidad específica y tiene parámetros de liberación que describen el modelo de liberación y alcance temporal, por ejemplo tiempo de liberación y *deadline*. El patrón de liberación es periódico o aperiódico.

El concepto de misión se representa en la figura [1.3] y contiene cinco fases:

- **Configuración:** donde se asignan en memoria los objetos de la misión. Esto se hace durante el arranque del sistema y no se considera de tiempo crítico.
- **Inicialización:** donde se realizan todas las asignaciones de objetos relacionados con la misión o de la totalidad de la aplicación. Esta fase no es de tiempo crítico.
- **Ejecución:** durante el cual se ejecuta toda la lógica de aplicación y entidades planificables se preparan para su ejecución de acuerdo con un planificador apropiativo. Esta fase es de tiempo crítico.

⁶RTT son las siglas de *Real-Time Thread*. Es la clase Java que implementa las tareas de tiempo real

⁷NHRT significa *No Heap Real-time Thread*. Es una subclase de RTT donde el *garbage collector* no actúa durante su ejecución. Destinada a tareas *hard real-time*

⁸*Missions* en su idioma original

⁹*schedulable entities* en su idioma original.

- **Limpieza:** se ingresa cuando termina la misión y se utiliza para completar la ejecución de todas las entidades planificables, así como la realización de funciones relacionadas con limpieza de memoria. Después de esta fase, la misma misión puede ser reiniciada, se selecciona una nueva, o bien, se ingresa en la fase de desmontaje. Esta fase no es de tiempo crítico.
- **Desmontaje:** es la fase final de la vida útil de la aplicación y se compone de la liberación de memoria de los objetos y otros recursos. Esta fase no es de tiempo crítico.

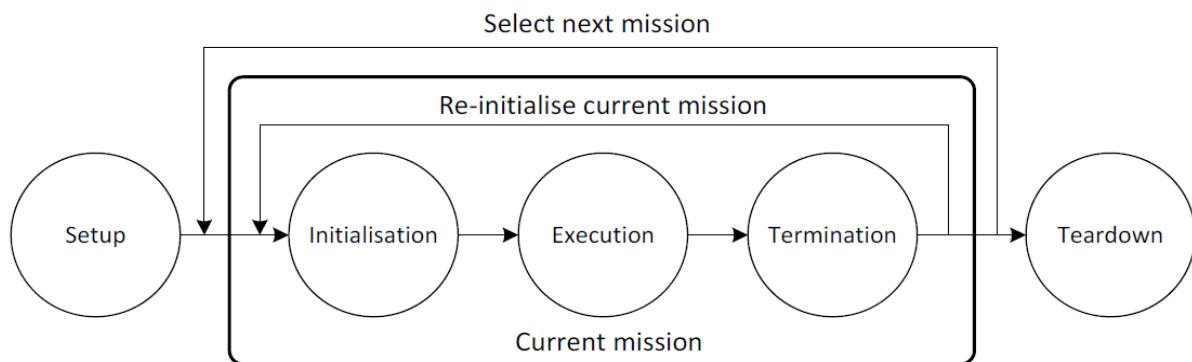


Figura 1.3: Concepto de misión SCJ.

Se utiliza un **secuenciador de misión** para regular el orden de los objetos de misión que puede ser personalizado para la aplicación.

SCJ presenta un modelo de memoria basado en el concepto de **ámbitos de memoria** de RTSJ, evitando el uso del *heap* con *garbage collector* para facilitar la verificación de los sistemas de SCJ. El modelo de memoria SCJ se muestra en la Figura [1.4] e introduce tres niveles de memorias, estos son:

- **Memoria Privada.** Se asocia a cada *handler* de eventos *real-time*. Esta memoria privada existe durante toda la duración del *handler* y se borra al finalizar.
- **Memoria Inmortal.** Es el área que perdura durante toda la vida útil del sistema quedando a cargo del programador.
- **Memoria de Misión.** Se asocia con cada misión del sistema y como tal, gestiona la memoria de todos los *handlers* de tiempo real de la misión, así como los objetos compartidos entre *handlers*. Cuando una misión completa su ejecución se borra su memoria asociada.

Niveles conformidad con la especificación SCJ

Existen 3 niveles conformidad con la especificación SCJ, dependiendo de las prestaciones ofrecidas:

- **Nivel 0.** Proporciona una ejecución cíclica (un único *thread*), sin *wait/notify*.
- **Nivel 1.** Provee una única **Misión** con múltiples **Objetos planificables**.
- **Nivel 2.** Ofrece **Misiones** anidadas (limitadas) con **ámbitos de memoria** anidados (limitados).

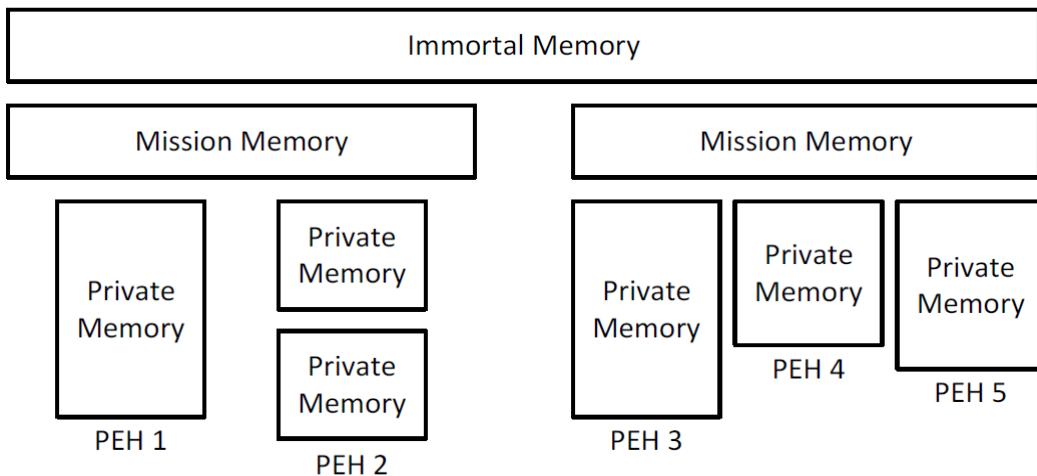


Figura 1.4: Modelo de memoria SCJ.

1.1.4. Máquinas Virtuales de Java para aplicaciones de tiempo real

Para poder utilizar Java sobre una plataforma de hardware en particular, se debe contar con una implementación de la JVM conforme a alguna de las especificaciones anteriores. Se exponen las distintas máquinas virtuales de Java que se consideraron y sus características:

- **JamaicaVM.** Sitio Web: <http://www.aicas.com/jamaica.html>.
- **FijiVM.** Sitio Web: <http://fiji-systems.com/>.
- **oSCJ.** Sitio Web: <https://www.cs.purdue.edu/sss/projects/oscj/>.
- **KESO VM.** Sitio Web: <https://www4.cs.fau.de/Research/KESO/>.
- **HVM.** Sitio Web: <http://icelab.dk/>.

JamaicaVM soporta la especificación RTSJ. Es un desarrollo de la empresa Aicas, planeada para aplicaciones *Hard Real-Time*, que posee un *garbage collector* determinístico (*fully preemptable*). Se encuentra en estado de certificación para su utilización en automóviles y aviones. Si bien es la JVM más prometedora, la misma es de código privado y por eso se descarta su utilización en este trabajo.

FijiVM soporta la especificación SCJ con muy buenas prestaciones, sin embargo al igual que JamaicaVM es un desarrollo de código privado.

Open Safety-Critical Java Implementation (oSCJ) es un desarrollo de la Universidad de Purdue, de código abierto, que implementa un conjunto restringido de la especificación SCJ, con foco en el nivel 0 de la misma. Posee un desarrollo de *Technology Compatibility Kit* (TCK) como es solicitado en SCJ, chequeo estático de *Annotations* SCJ y un conjunto de *benchmarks* SCJ. Su licencia es *New BSD*. La plataforma sobre la cual está desarrollada oSCJ es una FPGA Xilinx con un softcore LEON3 corriendo el sistema operativo de tiempo real RTEMS. Este desarrollo dista mucho del microcontrolador que se utiliza en el trabajo y no se ha encontrado documentación para portarlo a otra arquitectura.

KESO VM se desarrolla en Universidad Friedrich-Alexander, Alemania, con licencia *LGPL V3*. Está diseñada para correr sobre el sistema operativo de tiempo real OSEK, sobre las plataformas JOSEK, CiAO, Trampoline OS, Elektrobit ProOSEK y RTA-OSEK. En la web oficial existen

ejemplos sobre la arquitectura AVR de 8 bits de la compañía Atmel. Si bien posee soporte de *threads real-time*, no se basa en ninguna de las especificaciones de Java anteriores. Por otro lado, no se encontró documentación acerca de como llevar la misma a otra distribución de OSEK.

Hardware near Virtual Machine (**HVM**) comenzó como un desarrollo para la tesis doctoral de Stephan Erbs Korsholm (Icelabs). Es un entorno de ejecución de *Safety Critical Java* (SCJ) nivel 1 y 2, de código abierto diseñado para plataformas embebidas de bajos recursos.

HVM corre directamente sobre el hardware sin necesidad de un sistema operativo (*bare-metall*). Su diseño y excelente documentación (véase [\[\]](#)) facilita la portabilidad a nuevas arquitecturas.

Se compone de las siguientes partes:

- **Icecap tools.** Es un *plugin* que convierte al IDE Eclipse, en un IDE para la programación en lenguaje Java para HVM. Icecap tools genera código C a partir de la aplicación Java de usuario para correr sobre la máquina virtual de Java, HVM, así como los propios archivos que implementan a esta máquina virtual.
- **HVM SDK.** Es el *Software Development Kit* de HVM que incluye las clases que implementan SCJ.

En [\[3\]](#) se proveen *benckmarks* de HVM, KESO VM y FijiVM relativos a la aplicación de los mismos en lenguaje C.

Debido a estas características se elige HVM como la JVM a utilizar.

Cabe destacar, que durante el desarrollo del presente trabajo se ha entrado en contacto con Korsholm, vía correo electrónico, quien con excelente predisposición ha facilitado mucho la labor respondiendo todas las dudas. De esta manera, se ha logrado una cooperación entre los equipos de investigación de la Universidad Nacional de Quilmes e Icelabs, y se espera luego de la conclusión de este trabajo, contribuir al proyecto HVM con el aporte del *port* para la CIAA de HVM.

1.2. Justificación

El autor del presente Trabajo Final forma parte de un proyecto de investigación orientado por la práctica profesional perteneciente al departamento de Ciencia y Tecnología de la Universidad Nacional de Quilmes, titulado, “*Estrategias de desarrollo de sistemas embebidos en ambientes de automatización y control industrial. Un enfoque de programación con objetos y servicios web*”. Cuyo director es, además, el director de este trabajo, MSc. Ing. Félix Gustavo E. Safar y su co-director es Ing. Leonardo Ariel Gassman.

Este trabajo surge entonces como necesidad de obtener una herramienta para llevar a cabo los desarrollos en el marco de dicho proyecto.

1.3. Objetivo

El objetivo del Trabajo Final permitir la programación en lenguaje Java de la CIAA con aplicación en entornos industriales. El camino elegido para llevarlo a cabo es:

- Realizar el *port* de la máquina virtual de HVM para que corra sobre las plataformas CIAA-NXP y EDU-CIAA-NXP, permitiendo la programación de aplicaciones Java.
- Diseñar e implementar una API sencilla para permitir controlar periféricos del microcontrolador desde una aplicación Java.
- Llevar a cabo el *port* de la capa SCJ de la máquina virtual de HVM, para permitir desarrollar aplicaciones Java SCJ.

- La integración del *port* para la CIAA al IDE de HVM, para completar un IDE de Java SCJ sobre la CIAA.

Estas tareas conllevan a la obtención de un Entorno de Desarrollo Integrado (IDE) para programar en lenguaje Java las plataformas CIAA-NXP y EDU-CIAA-NXP.

Capítulo 2

DESARROLLO

En este capítulo se describe HVM (sección [2.1]) y como portarla a otra plataforma de hardware (sección [2.2]). Luego, se presenta el diseño de una biblioteca para el manejo de periféricos (sección [2.3]). Finalmente, se describe como se deben integrar todas estas partes para mejorar su utilización (sección [2.4]).

2.1. HVM (*Hardware near Virtual Machine*)

El propósito de HVM es habilitar la programación en lenguaje Java de dispositivos embebidos con pocos recursos. Los recursos mínimos necesarios en un microcontrolador son 10 kB de ROM y 512 bytes de RAM. Sin embargo, para ejecutar programas de tamaño razonables, se necesitan 32 kB de ROM y 2kB de RAM.

HVM funciona realizando una traducción de un programa de usuario escrito en lenguaje Java a un programa en lenguaje ANSI C, que incluye el código de dicho programa y el que implementa la máquina virtual.

El código ANSI C generado puede compilarse mediante un *cross compiler* para una plataforma en particular generando un ejecutable para luego descargarlo a la misma.

HVM está diseñada para ser independiente del hardware. Solamente una pequeña parte de la misma debe implementarse con código dependiente de la plataforma donde va a ejecutarse. Esta parte se encuentra bien definida de manera de facilitar la portabilidad entre diferentes microcontroladores.

Además, posibilita la integración con programas escritos previamente en lenguaje C (código *legacy*) para poder aprovechar cualquier biblioteca desarrollada. Esto es muy importante, dado que en la actualidad, casi la totalidad de las bibliotecas existentes para microcontroladores están hechas en este lenguaje.

2.1.1. Obtención de un IDE para desarrollar programas Java sobre HVM

En la sección [1.1.4] se adelantó que HVM se distribuye como un *plugin* de Eclipse para convertirlo en un IDE para desarrollar programas Java SCJ sobre HVM. Para su utilización se debe descargar:

- IDE Eclipse. En particular la distribución **Eclipse Automotive**, recomendada pues integra el desarrollo de aplicaciones Java y C.
- **Icecap tools**. Es el *plugin* de Eclipse de HVM.

- **HVM SDK.** El *Software Development Kit* que provee HVM.

Eclipse Automotive se descarga en <http://www.eclipse.org/downloads/packages/eclipse-ide-automotive-software-developers-includes-incubating-components/junosr2>.

Los otros dos se distribuyen como archivos *.jar* y pueden descargarse de <http://icelab.dk/download.html> sus respectivos nombres son **icecaptools_x.y.z.jar** e **icecapSDK.jar**.

Una vez que se descarga y descomprime Eclipse, se debe instalar sobre el mismo el *plugin icecaptools*, que es el encargado de compilar el programa Java para utilizarse sobre HVM. Para realizar programas SCJ debe incluirse **icecapSDK** como biblioteca (Jar externa) al proyecto de aplicación Java.

Finalmente se debe instalar un *toolchain* para la plataforma de hardware que integre un *cross compiler* de lenguaje C, y los programas necesarios para *debug* y descargar el ejecutable compilado. Este puede ser un conjunto de programas independiente o integrarse a Eclipse. De esta manera se completa el IDE para trabajar con HVM.

2.1.2. Utilización de HVM

Para realizar una aplicación Java para HVM se debe realizar un proyecto Java estándar, con el IDE Eclipse, que incluya **icecapSDK.jar** como biblioteca Jar externa.

Luego se realiza el programa Java de manera estándar escribiendo las Clases que lo componen. En la figura [2.1] se muestra una captura de una aplicación llamada *HolaMundoHVM.java* que simplemente imprime por consola este mensaje.

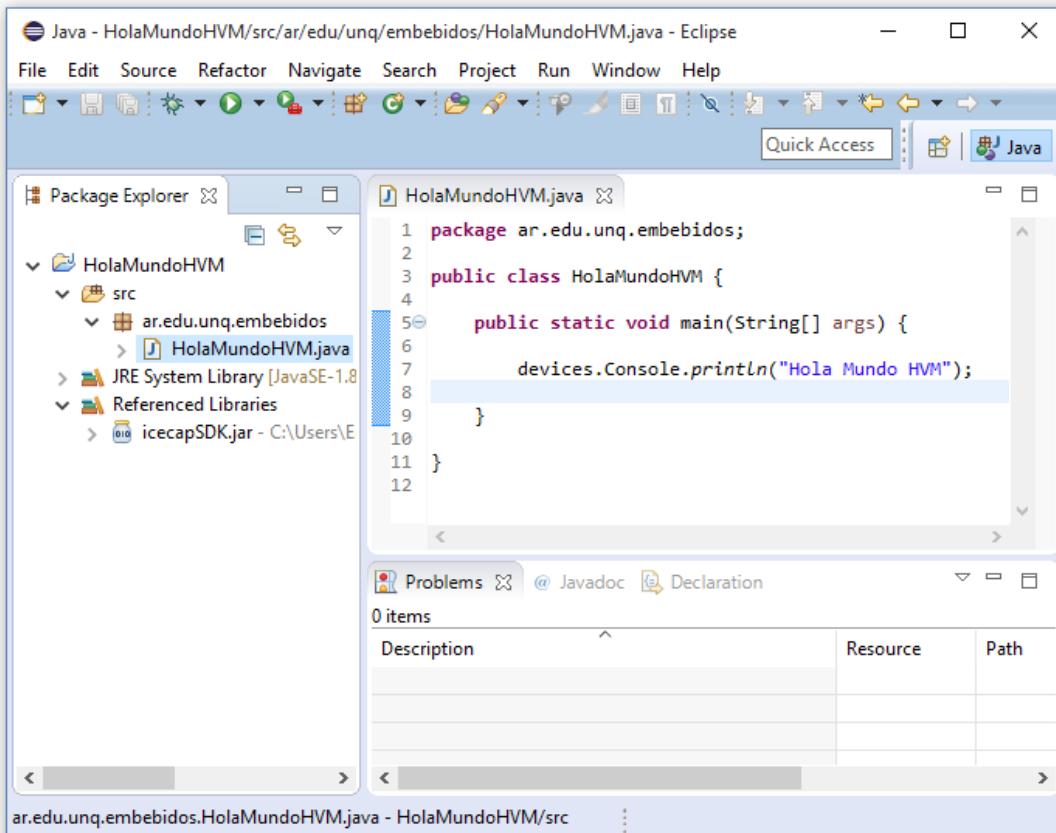


Figura 2.1: Programa Hola Mundo con HVM.

Una vez completada la aplicación Java se busca el método **main** de la clase principal donde comienza la aplicación¹ en el árbol de proyecto y se presiona el botón derecho del *mouse* sobre dicho método para abrir un menú contextual donde se selecciona la opción ***Icecap tools*** y luego ***Convert to Icecap Application***.

Esto lanza el proceso donde se genera el código C. El primer paso es generar el **grado de dependencia**² que es el conjunto de clases y métodos requeridos para la aplicación. En la figura [2.2] se ofrece el grado de dependencia del ejemplo (*Dependency Extent*) que se compone de 29 elementos. Esto se debe a todas las clases que necesita para el manejo de *Strings*, excepciones de Java, la propia clase del ejemplo, además de todas sus dependencias.

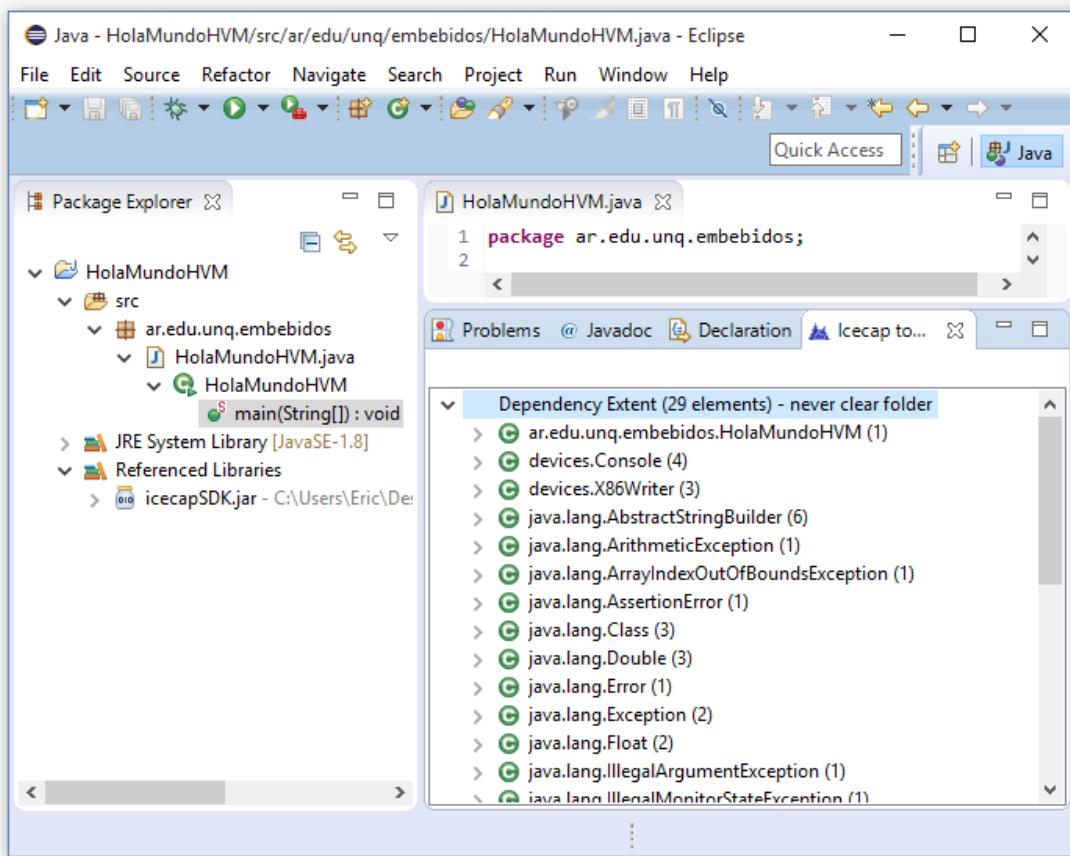


Figura 2.2: Grado de dependencia del programa Hola Mundo con HVM.

Luego se debe presionar el botón derecho del *mouse* sobre ***Dependency Extent*** y mediante la opción ***Set output folder*** y se elije la carpeta donde se guardarán los archivos C generados por HVM. Una vez elegida la carpeta de salida se vuelve a ejecutar ***Convert to Icecap Application*** para generar los archivos C.

Finalmente se utiliza el compilador de la plataforma y se descarga a la misma. Un resumen de todo el proceso se muestra en la figura [2.3].

¹En Java cada Clase puede tener un método **main** pero en el proyecto debe especificarse cual es la que se utiliza como punto de inicio a la aplicación

²En el manual de referencia de HVM [2] se llama *Dependency Extent* en su idioma original.

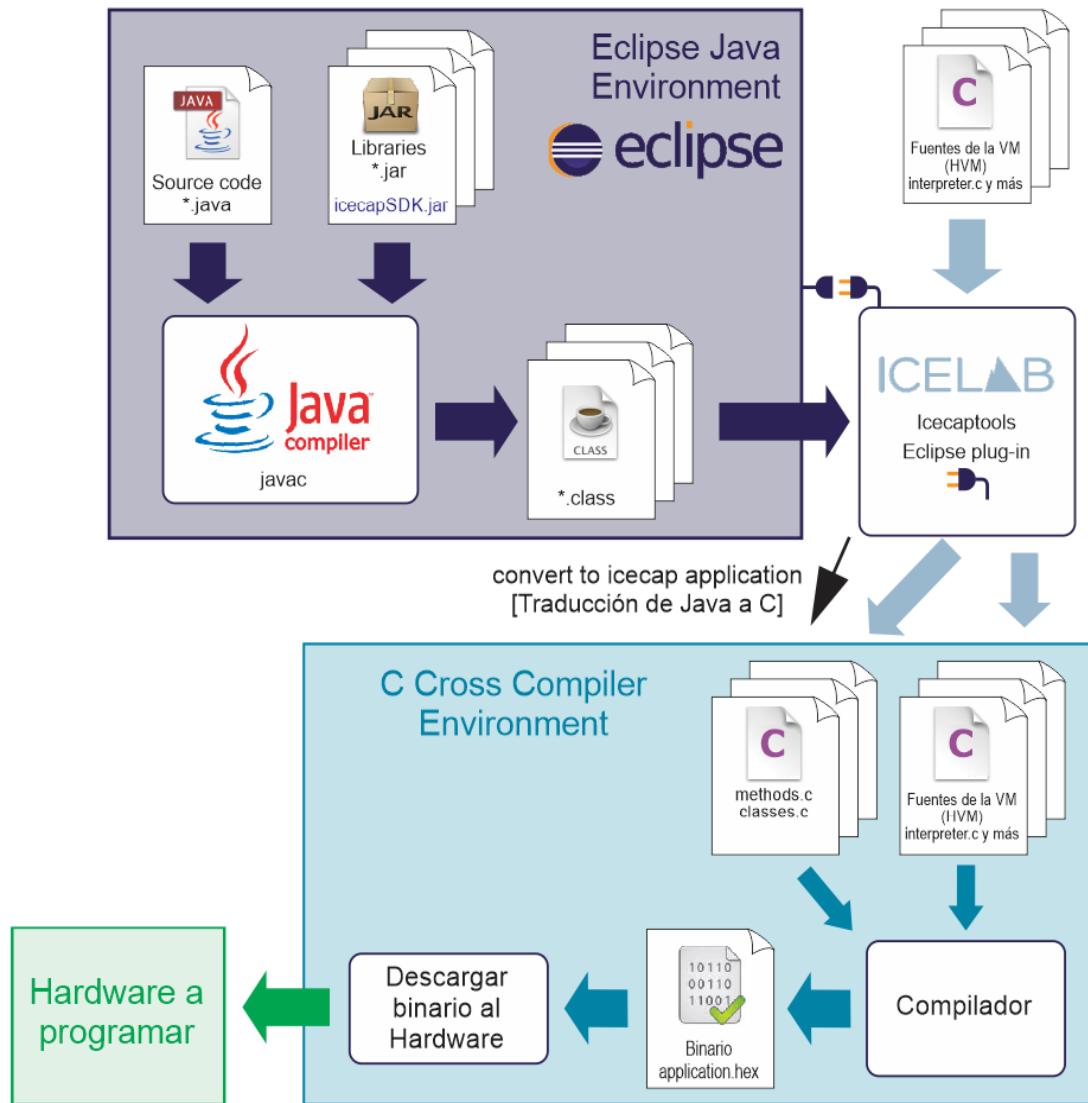


Figura 2.3: Esquema de funcionamiento del IDE para trabajar con HVM sobre sistemas embebidos.

Compilación y ejecución del ejemplo en PC x86 (Posix)

Con los archivos C generados se puede utilizar la **Terminal** de Linux, o **Cygwin** en Windows, para compilar el programa con **gcc** para su posterior ejecución en la PC mediante el comando:

```
gcc -Wall -g -O0 -DPC32 -DPRINTFSUPPORT -DJAVA_HEAP_SIZE=6500 -DJAVA_STACK_SIZE=1024
classes.c icecapvm.c methodinterpreter.c methods.c gc.c natives_allOS.c natives_i86.c
allocation_point.c print.c
```

Esto genera un ejecutable llamado **a.exe**. Para probarlo, se ejecuta mediante el comando **./a.exe** que produce la salida en pantalla:

Hola Mundo HVM

En la figura [2.4] se muestra la ejecución del comando de compilación y del archivo **a.exe** en **Cygwin**.

```

/cygdrive/c/hvm_C_output
Eric@Lenovo-PC ~
$ cd /cygdrive/c/hvm_C_output
Eric@Lenovo-PC /cygdrive/c/hvm_C_output
$ gcc -Wall -g -O0 -DPC32 -DPRINTFSUPPORT -DJAVA_HEAP_SIZE=650
0 -DJAVA_STACK_SIZE=1024 classes.c icecapvm.c methodinterpreter
r.c methods.c gc.c natives_allOS.c natives_i86.c allocation_po
int.c print.c
Eric@Lenovo-PC /cygdrive/c/hvm_C_output
$ ./a.exe
Hola Mundo HVM

```

Figura 2.4: Ejemplo Hola mundo con HVM en Cygwin.

2.1.3. Consideraciones a tener en cuenta al utilizar HVM

Archivos C generados por HVM

La salida del proceso de traducción de Java a C produce un conjunto de archivos fuente en lenguaje C. Algunos de estos dependen de la aplicación a traducir y otros son fijos. Los archivos dependientes de la aplicación son:

- **methods.c, methods.h** - Contiene la implementación en lenguaje C de todos los métodos de Java incluidos en el grado de dependencia.
- **classes.c, classes.h** - Contiene la implementación en lenguaje C de todas las clases de Java incluidas en el grado de dependencia y sus relaciones jerárquicas.

mientras que los archivos fijos los archivos fijos:

- **ostypes.h** - Tipos de datos dependientes de la arquitectura.
- **types.h** - Tipos de datos de HVM, por ejemplo, para definir objetos.
- **icecapvm.c** - La función main que inicia HVM.
- **methodinterpreter.c, methodinterpreter.h** - Funciones del intérprete de HVM para *bytecodes* de Java.
- **gc.c, gc.h, allocation point.c, allocation point.h** - Funciones para el manejo de memoria de HVM.
- **print.c** - Funciones nativas para imprimir mensajes simples.
- **natives_allOS.c** - Funciones que implementan una Java SDK rudimentaria, con lo mínimo indispensable para que funcione.
- **natives_XX.c** - Funciones de HVM específicas de la arquitectura del microcontrolador.
- **XX_interrupt.s** - Funciones en *assembler* para implementar un cambio de contexto si se utilizan hilos de ejecución (*threading*).

- **native_scj.c** - Funciones para correr aplicaciones SCJ sobre plataformas POSIX.

Una explicación detallada de cada uno de los archivos generados puede encontrarse en [2].

Grado de dependencia

Si el grado de dependencia es mayor a lo que puede manejar HVM puede ocurrir error por **pérdida de dependencias**³. Cuando sucede se informa por consola en qué línea ocurrió. En programas embebidos pueden utilizarse muchas de las **java.util.*** sin que se produzca esta pérdida.

Nótese que en el ejemplo realizado, no se utiliza la típica línea de Java **System.out.println("Hola Mundo HVM")**; para imprimir por consola. En caso de utilizarla causará una pérdida de dependencias porque el analizador de HVM no puede manejar la excesiva cantidad de dependencias generadas. Para subsanarlo se utiliza **devices.Console.println("Hola Mundo HVM")**; que genera un número mucho menor de dependencias fácilmente controlables por HVM.

Métodos compilados y métodos interpretados

Icecapools permite elegir a nivel de métodos cuales serán compilados y cuales interpretados. Por defecto son todos interpretados. Para pasar un método a compilado se debe presionar el botón derecho del *mouse* sobre el método y seleccionar la opción **Toggle Compilation** (figura [2.5]). Una vez pasado a compilado la esfera verde que indica el método se pasa a color violeta.

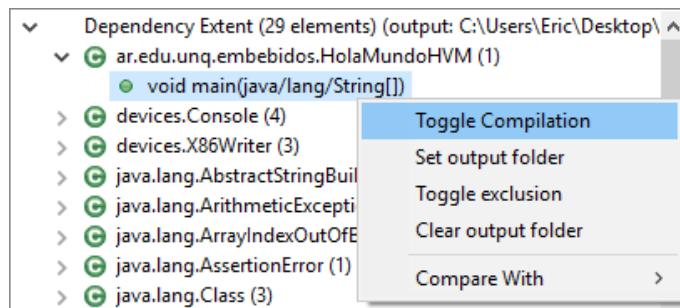


Figura 2.5: Cambiar un método a modo compilado.

Los compilados son traducidos a funciones en lenguaje C, mientras que los interpretados se traducen como un vector de *bytecodes* constante en lenguaje C para ser interpretados mediante el intérprete de HVM.

Cada método de ejecución tiene sus ventajas: Los métodos interpretados ocupan menos espacio en ROM pero son más lentos al ejecutarse y los métodos compilados requieren un poco más de ROM pero la ejecución es significativamente más rápida.

Acerca del comando de compilación

Optimizaciones:

- **-O0** produce un ejecutable optimizado para *debug*.
- **-Os** produce un ejecutable con optimización en tamaño.
- **-O3** produce un ejecutable con optimización en velocidad de ejecución.

³En inglés *Dependency Leak*.

Definiciones para compilación condicional de HVM:

- **-DPC32** define para la plataforma objetivo. Esto selecciona los tipos que usa HVM en `ostypes.h`. Los posibles son DPC64, DPC32, WIN32, CR16C, V850ES, SAM7S256, AVR.
- **-DPRINTFSUPPORT** habilita el uso de la función `printf` si la plataforma tiene una biblioteca `libc`.
- **-DJAVA HEAP SIZE=6500** define el tamaño del *heap* de Java. El tamaño por defecto es de 64 kB. Este ejemplo utiliza 6500 bytes para el heap.
- **-DJAVA STACK SIZE=1024** se utiliza para definir el tamaño de la pila principal en celdas de 4 bytes. Esto significa que en este ejemplo se utilizan 4 kB. Para Windows o Linux no puede ser menor a 4 kB. Para sistemas embebidos puede ser tan chico como 256 bytes, pero debe ajustarse según la aplicación.

2.1.4. Características de HVM

Manejo de memoria

HVM no soporta el *Garbage Collector* estándar de Java. Para la gestión de la asignación y liberación de recursos de memoria utiliza el modelo de ámbitos de memoria de SCJ.

Para los programas no SCJ esto significa que los datos se asignan consecutivamente en áreas de memoria y se liberan fragmentos de un área completa a la vez.

El uso de áreas de memoria requiere más cuidado que el uso del *Garbage Collector* estándar. En este esquema el manejo de memoria es responsabilidad del programador y pueden ocurrir todos los problemas habituales que suceden cuando se utiliza la gestión de la memoria explícita (por ejemplo, punteros colgantes [4]). El concepto de área de memoria para Java es una violación de la seguridad de Java que el desarrollador HVM debe comprender y utilizar con cuidado.

En la práctica la mayoría de las aplicaciones para sistemas embebidos (en especial los de bajos recursos) tienen requisitos muy sencillos de asignación de memoria. En tales escenarios no se requiere un *Garbage Collector* completo y el concepto de área de memoria es suficiente.

Acceso al hardware desde Java

HVM provee cuatro formas de acceso al hardware:

- **Variables Nativas.** Sirven para mapear variables de C a variables estáticas de Clase en Java. Sólo pueden ser accedidas dentro de métodos compilados.
- **Objetos Hardware.** Es una abstracción que permite acceder a registros del microcontrolador mapeados en memoria para manipularlos desde el programa en lenguaje Java. De esta forma se puede crear una biblioteca completa dependiente del microcontrolador que maneje un periférico a nivel de registros directamente en Java.
- **Métodos Nativos.** Desde Java pueden utilizarse métodos nativos escritos en lenguaje C y pasar parámetros de cualquier tipo. De esta manera permite ejecutar código *legacy* dando la posibilidad de utilizar bibliotecas completas realizadas en C desde Java. Para conectar un método con una función en lenguaje C, deben respetarse ciertas convenciones de nombres y de pasaje de parámetros en las funciones realizadas para que puedan ser asociadas.

- **Manejadores de interrupciones⁴**. HVM SDK contiene clases en infraestructura para el manejo de interrupciones desde Java. En particular, la clase `vm InterruptDispatcher` define como registrar *interrupt handlers*.

Se elige métodos nativos como alternativa para proveer al programa de usuario en lenguaje Java el acceso a los periféricos básicos del microcontrolador. Existen además *stacks*, *file systems* y muchas otras bibliotecas que se querrán aprovechar.

Reflexión⁵

HVM SDK contiene una API de *Reflection* limitada que la utiliza la infraestructura de SCJ. Permite escanear información estática de clases, referencias a objetos creados y vectores; instanciar Clases en *run time* e invocación reflexiva de métodos.

Depuración a Nivel de Java

El *plugin* de Eclipse de HVM soporta depuración a nivel de Java mediante Eclipse. Las sesiones de depuración (*debug*) se inician lanzando la aplicación en modo *debug* mediante las configuraciones de *Launcher* (Lanzador) de HVM. Soporta las siguientes características de depuración:

- Agregar y eliminar *breakpoints*. Por el momento uno por método.
- *Step over* y *step into*.
- Inspeccionar los valores de las variables locales de tipos de datos básicos. Momentáneamente no permite objetos ni *Arrays*.
- Inspección de la pila de ejecución
- Depuración únicamente sobre métodos interpretados.

La aplicación que se ejecuta de forma remota es controlada por el depurador mediante un enlace de comunicación. Cuando la aplicación es ejecutada en modo depuración se establece una comunicación con el depurador con el fin de actualizar *breakpoints*, informar que se alcanzó un *breakpoints* y enviar los datos de variables y pila al depurador. Para las plataformas POSIX el enlace de comunicación es sobre es TCP/IP, para plataformas AVR es sobre comunicación serie (UART).

El depurador se implementa en Eclipse utilizando el *Eclipse debugging framework* (`org.eclipse.debug.*`). Del lado cliente la conexión de depuración se lleva a cabo principalmente en el archivo *natives_allOS.c* para todas las plataformas y en *natives_XXX.c* las funciones específicas de la plataforma.

⁴Del inglés *Interrupt Handlers*.

⁵Del inglés *Reflection*.

2.2. Port de HVM a una nueva plataforma de Hardware

HVM genera código ANSI C independiente de la plataforma. Únicamente una pequeña parte de la infraestructura es dependiente de la plataforma. En la figura [2.6] se da ofrece esquema en capas de HVM.

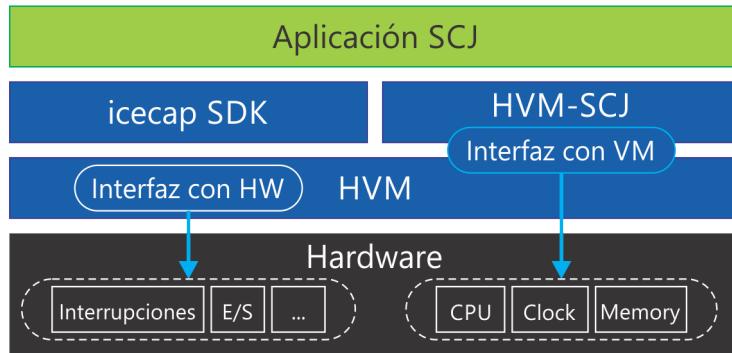


Figura 2.6: Modelo de capas de HVM.

Para portar HVM a una nueva plataforma se deben realizar las siguientes tareas:

- Obtener un entorno de desarrollo de C para la plataforma y probar su correcto funcionamiento.
- Construir un comando de compilación para compilar y linkear los archivos generados de HVM.
- Agregar una definición para la plataforma en el archivo **ostypes.h**.
- Definir las funciones específicas de la plataforma. HVM aísla estas funciones en dos archivos, **natives_XX.c** y **XX_interrupts.s**.
- Completar el comando de compilación y probar el funcionamiento.
- Implementar el acceso a periféricos de la plataforma.

Según el manual de referencia de HVM [2], la plataforma más chica donde se ha ejecutado HVM es el Arduino UNO (basado en AVR Atmega328 de Armel). Posee un microcontrolador de 8 bits con 32 kB de memoria ROM y 2 kB de memoria RAM. En esta configuración existe lugar para programas Java no triviales que controlan periféricos mediante el uso **variables nativas** u **objetos hardware**. Es posible también soportar cambio de procesos utilizando el concepto de **Procesos SCJ**, sin embargo, para tener SCJ completo se requiere más recursos. El microcontrolador más pequeño donde se probó un programa SCJ completo es el AVR ATMega1280, con 128 kB de ROM y 8 kB de RAM. Este microcontrolador puede encontrarse en la plataforma Arduino Mega. Actualmente existen implementaciones de HVM para las arquitecturas AVR (8 bits), CR16c (16 bits), ARM7 (32 bits), Intel de 32 bits e Intel 64 bits.

El port de HVM a una nueva plataforma puede implementarse en dos partes, una para lograr ejecutar programas Java simples, y otra para dar soporte de aplicaciones SCJ. En las siguientes secciones se describe que funciones deben realizarse para lograr ambos objetivos.

2.2.1. Port de HVM para ejecutar Java

Como se ha adelantado, se requiere agregar una **definición para la plataforma** en el archivo **ostypes.h**. Esto se debe a que HVM utiliza su propia definición de tipos de datos básicos y de puntero para independizarse de la arquitectura donde se ejecuta. Es por esto que la definición

consiste en indicar correctamente los tipos de datos básicos, de punteros a memoria de programa de HVM, así como algunas macros para la plataforma. Como ambas plataformas CIAA donde se porta HVM contienen el mismo microcontrolador. La definición de plataforma para la CIAA se nombra **LPC4337_TYPES_FOR_HVM** en la implementación.

En el archivo **natives_XX.c** deben definirse las siguientes funciones específicas de la plataforma:

- **void init_compiler_specifics(void)**. Esta función se llama al comienzo de la función *main*. Se utiliza en algunas plataformas para copiar datos inicializados en los segmentos correctos. Si esto no es necesario, se puede dejar vacía. Sólo se llama una única vez.
- **int32* get_java_stack_base(int16 size)**. Esta función se llama antes de entrar en la máquina virtual. Debe devolver un puntero a un área de memoria RAM que se utilizará como la pila de Java.
- **void initNatives(void)**. Esta función se llama antes de iniciar la máquina virtual. Si la máquina virtual se reinicia, la misma vuelve a ser llamada. Se puede dejar vacía.
- **void mark_error(void), void mark_success(void)**. Sólo las utiliza el sistema de pruebas de regresión. Si el mismo no es utilizado se puede dejar vacío.
- **void writeByteToIO(pointer address, unsigned short offset, unsigned char lsb)**. Esta función se utiliza para la implementación de objetos hardware. Se debe implementar para escribir lsb a la dirección + *offset*. El *offset* es en bits. En la mayoría de las arquitecturas esto puede implementarse muy fácilmente como una referencia normal de puntero. En otras arquitecturas, en cambio, se deben ejecutar instrucciones de propósito especial para la lectura y escritura de registros I/O. Existen varias otras funciones similares *read/writeXXToIO* para otros tipos de datos.
- **init_memory_lock, lock_memory, unlock_memory**. Deben ser implementadas en caso de que puedan producirse interrupciones mientras se asigna memoria utilizando *new*. Estas funciones se utilizan para realizar un *mutex* alrededor de la asignación de memoria. Se pueden dejar vacías para programas que no utilizan las interrupciones o si ninguno de los *handlers* de interrupciones asignan memoria.
- **void sendbyte(unsigned char byte)**. Imprime un byte. Se utiliza para imprimir los mensajes de la consola. Se puede dejar vacía. Si la plataforma posee una UART disponible, se puede utilizar para la impresión.

Este archivo para la plataforma CIAA se nombra **LPC4337_natives.c** en la implementación.

Existen funciones adicionales para implementar *debug* directo desde un programa Java las cuales no se tratan en este trabajo.

2.2.2. Port de HVM para ejecutar Java SCJ

Para dar soporte a aplicaciones basadas en la especificación SCJ sobre HVM, debe construirse una HAL⁶. Esta HAL debe tener primitivas para planificación de tareas expropiativas⁷, manejo de memoria, acceso a dispositivos mediante *Hardware Objects*, manejo de interrupciones de primer nivel, un programa monitor y *real-time clock*. Esto se conoce generalmente como la construcción de un *micro kernel*. En el caso particular de la HAL de Java SCJ para HVM se encuentra implementada en su mayoría en lenguaje Java. Esto da 2 beneficios importantes:

⁶Siglas de *Hardware Abstraction Layer*.

⁷Una posible traducción al castellano de *preemptive*, otra posible es apropiativa.

1. **Probabilidad.** Todas las partes de la HAL de Java escritas en Java pueden ejecutarse sobre HVM.
2. **Especialización de programa.** En este contexto se refiere a incluir únicamente las partes utilizadas de la HAL de Java, excluyendo las partes no utilizadas del ejecutable final. La adaptación de la HAL para una aplicación en particular no requiere ningún esfuerzo para las partes de la HAL escritas en Java.

Dejando una pequeña parte que debe implementarse en lenguaje C. Esta parte corresponde a las siguientes funciones.

Archivo **natives_XX.c**:

- **void start_system_tick(void), void stop_system_tick(void).** Estas funciones inician y terminan un temporizador que actualiza la variable global **systemTick** utilizada por el planificador.
- **int16 n_vm_RealtimeClock_awaitNextTick(int32 *sp).** Debe bloquear hasta que se actualice la variable global **systemTick**.
- **int16 n_vm_RealtimeClock_getNativeResolution(int32 *sp).** Debe retornar la resolución del temporizador que se inició en la función **void start_system_tick (void)**. Debe retornar en número de nano segundos entre dos *ticks* del sistema con tipo de datos *uint32*.
- **int16 n_vm_RealtimeClock_getNativeTime(int32 *sp).** Devuelve el tiempo actual del reloj de tiempo real como un objeto **AbsoluteTime** con mili segundos y nano segundos.

Este archivo para la plataforma CIAA se nombra **LPC4337_natives_SCJ.c** en la implementación. **Nótese** que el archivo **natives_XX.c** se ha separado en dos archivos en la implementación para mejorar la modularización.

Luego, en el archivo **XX_interrupt.s** deben realizarse tres funciones en *assembler* necesarias para implementar los **procesos SCJ** y el cambio de *threads* (cambio de contexto).

La función **_yield**. Debe guardar todos los registros en la pila, guardar el puntero a pila en la variable global **stackPointer** (declarada en **natives_allOS.c**) y llama a la función **transfer** (definida también en **natives_allOS.c**). Cuando termina la ejecución de **transfer** debe guardar el valor de la variable global **stackPointer** al puntero a pila, restaurar todos los registros (en orden inverso) y retornar.

Cuando se llama a la función **pointer* get_stack_pointer(void)** debe retornar el valor del puntero a pila. Los pasos para llevarlo a cabo son:

1. Mover el valor del puntero a pila al registro utilizado por el compilador para el valor de retorno de funciones.
2. El valor actual del puntero a pila es el valor del *frame* actual. Se debe ajustar el valor de retorno ya que se requiere el valor del *frame* que llamó a esta función.
3. Devolver este último valor de retorno.

La función **set_stack_pointer(void)** Debe establecer el valor de la variable global **stackPointer** en el puntero a pila y retornar a la función invocante. Concretamente:

1. Mover el valor de retorno de la pila a algún registro.
2. Mover el valor de la variable global **stackPointer** al registro puntero a pila.

3. Mover el valor de retorno guardado en 1 a la pila.
4. Retornar.

Por lo general (en la mayoría de las arquitecturas) al llamar a una función se inserta en la pila la dirección de retorno. Esta es la dirección donde debe continuar al ejecución cuando termine de ejecutar la función que ha llamado. Luego cuando se retorna de la función, se saca de la pila la dirección de retorno y se realiza un salto a dicha dirección. Esto provoca que la ejecución continúe en la dirección correcta al terminar de ejecutar la función. En la función `set_stack_pointer` se utiliza una nueva pila, sin embargo, se necesita de todas formas retornar la dirección de donde se ha llamado a la función `set_stack_pointer`. Como se establece un nuevo puntero a pila, la dirección de retorno correcta no se encuentra en esta nueva pila. Por esto, es necesario que en el paso 1 se mueva la dirección de retorno a la nueva pila para poder retornar correctamente a donde se ejecutó `set_stack_pointer`.

En la implementación este archivo se nombra **LPC4337_interrupt.s**

2.3. Diseño de biblioteca para el manejo de periféricos desde Java

Proveer el manejo de periféricos de la CIAA directamente desde Java es requisito fundamental para que tenga sentido la adopción de este lenguaje.

En Java, esto se traduce a la necesidad de diseñar una biblioteca Java para el manejo de periféricos. Una biblioteca de Java se compone de un conjunto de Clases Java empaquetadas en un archivo de extensión **.jar**. El alcance de esta biblioteca para el presente trabajo es permitir la utilización de los siguientes periféricos:

- Entradas y salidas digitales.
- Entradas y salidas analógicas.
- Periférico de comunicación serie (UART).

Para la construcción de esta biblioteca se utilizarán métodos nativos escritos en lenguaje C. En consecuencia, se necesita obtener una implementación de bibliotecas de C para estos periféricos y luego programar los métodos nativos que formarán parte de las Clases Java que componen la biblioteca.

Al comienzo se analizó la estructura del Firmware de la CIAA (que está realizado en lenguaje C) para intentar compatibilizar HVM con el mismo, con el interés de reutilizar el manejo de periféricos mediante **Posix** y el protocolo **Modbus** para HVM. En la figura [2.7] se ilustra la estructura en capas (simplificada) del Firmware de la CIAA.

Se descubrió que los módulos de interés dependen de **FreeOSEK**, el sistema operativo de tiempo real, que provee manejo de tareas mediante un planificador apropiativo, alarmas, eventos, recursos y *handlers* de interrupción. HVM realiza las mismas tareas que OSEK y está diseñada para correr directamente sobre el hardware, entonces, no es posible lograr fácilmente una convivencia entre ambos en una misma aplicación. Para realizarlo, una estrategia sería reformar HVM para que corra sobre OSEK. Otra posibilidad es reescribir las partes de los módulos Posix y Modbus que dependen de FreeOSEK para que utilicen servicios de HVM. Ambas tareas demandan bastante trabajo.

Por otro lado Posix utiliza una filosofía donde todos los periféricos se acceden como un *stream* de bytes (concepto de todo es un archivo) mediante las funciones *open*, *close*, *read*, *write* e *ioctl*. Si bien es una abstracción muy eficiente, resulta poco natural para programadores ajenos a los sistemas Unix.

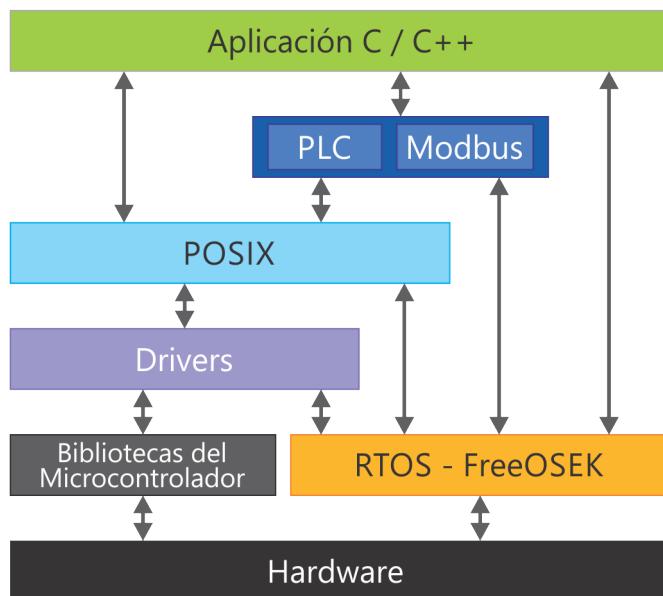


Figura 2.7: Estructura en caps del Firmware de la CIAA.

En los siguientes apartados se presentan las distintas características del diseño propuesto para esta biblioteca.

2.3.1. Modelo de la biblioteca Java

Para el modelado de la biblioteca de Java se persiguen los siguientes objetivos:

- Abstraer y simplificar la configuración del Hardware.
- De uso fácil para programadores informáticos y gente familiarizada con otras plataformas de hardware.
- Independiente del sistema operativo (en este caso HVM).

Se enfoca la misma a los conceptos que manejan los programadores en entornos industriales. Allí el dispositivo de uso más masivo es el PLC⁸ donde los conceptos que se manejan son los de dispositivos de entrada, salida y comunicaciones.

Se propone modelar el concepto de **Dispositivo** que se compone de **Periféricos**. Los periféricos se clasifican en:

- **DigitalIO**. Modela una entrada o salida cuyo valor lo representa un booleano.
- **AnalogIO**. Modela una entrada o salida cuyo valor lo representa un número entero.
- **Uart**. Modela el periférico de comunicación serie, UART.

Como en las plataformas de hardware existe más de un periférico de cada tipo, se debe poder identificar a cada uno en particular.

Cada periférico utiliza uno o más de pines del microcontrolador. En la plataforma CIAA-NXP esta relación es fija por diseño y no tiene sentido configurar cada pin para otra funcionalidad que no sea la asociada a su hardware de salida. En la EDU-CIAA-NXP, en cambio, esto es distinto porque

⁸Siglas de Controlador Lógico Programable.

se dispone de los pines directos que salen del microcontrolador pudiéndose utilizar para cualquier propósito. Un pin puede ser compartido por varios periféricos. Es por esto que se agrega al modelo el concepto de **Pin** de un periférico, que sirve para mantener la relación física del periférico con el microcontrolador. El Pin también sirve para determinar si un pin está en uso por un periférico cuando se intente usar dos periféricos distintos que utilicen los mismos pines físicos.

Los métodos públicos para el acceso a un periférico son:

- **read()**. Método para leer el periférico.
- **write()**. Método para escribir el periférico.
- **config()**. Método para configurar el periférico.

En la figura [2.8] se ofrece el modelo de Periférico (se han simplificado en el mismo tipos de datos).

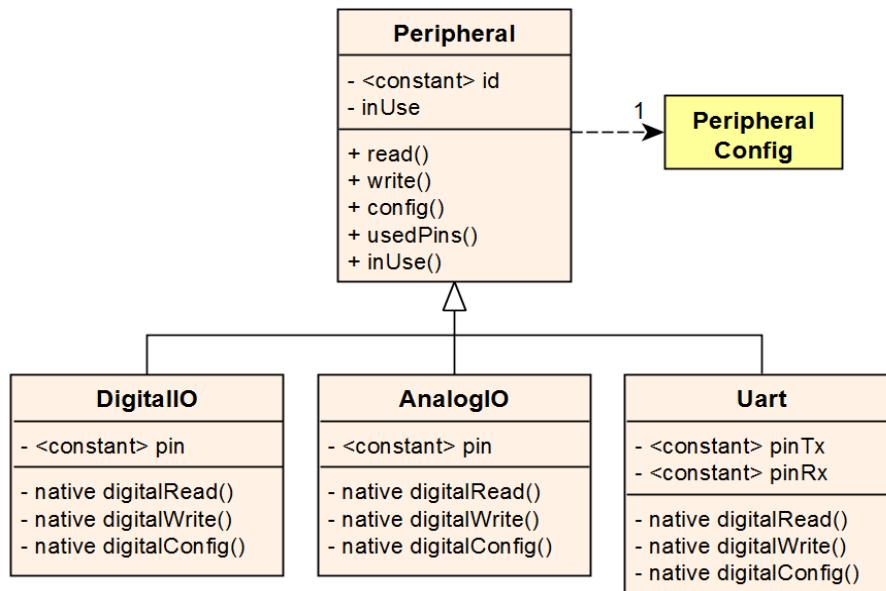


Figura 2.8: Modelo de Periférico.

Una clase **Peripheral** define los métodos propuestos para el acceso, que luego son redefinidos en cada una de las subclases las cuales contienen la implementación concreta de cada tipo de periférico. Esta implementación incluye los métodos nativos que se deben implementar en C. En el método constructor del objeto Periférico se deben establecer un identificador de dispositivo constante (id) y los pines físicos que usará el dispositivo (también definidos como constantes).

Existe una jerarquía de clases similar para **PeripheralConfig**, que se utilizan como parámetros para la configuración de los periféricos.

El diagrama de como se articula un periférico con las otras clases se muestra en la figura [2.9]. Un **Device** contiene una colección de pines y una de periféricos. Contiene métodos para añadir tanto periféricos como pines, y consultar si está en uso un pin. Contiene métodos estáticos para la construcción de instancias para cada plataforma (EDU-CIAA-NXP y CIAA-NXP) con todos sus periféricos y pines construidos.

El **Pin** contiene varios atributos constantes, que se asignan en el método constructor, uno corresponde a un id único de pin y los otros son indicaciones de que tipo de periféricos soporta. También tiene un atributo que referencia al periférico que tenga tomado al pin.

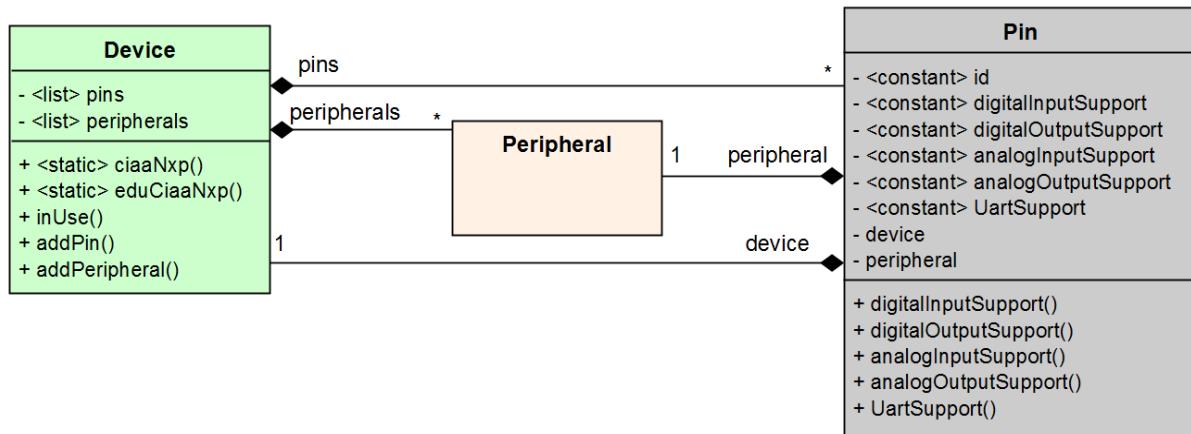


Figura 2.9: Modelo de Dispositivo y Pin.

2.3.2. Mapeo de pines de las plataformas

Con el diseño propuesto se debe mapear los pines de las plataformas. En este se deciden los números de pines, sus identificadores y se asignan las posibles funciones. En la figura [2.10] se muestra el mapeo de pines de la CIAA-NXP, mientras que en la figura [2.11] se ilustra el de la EDU-CIAA-NXP.

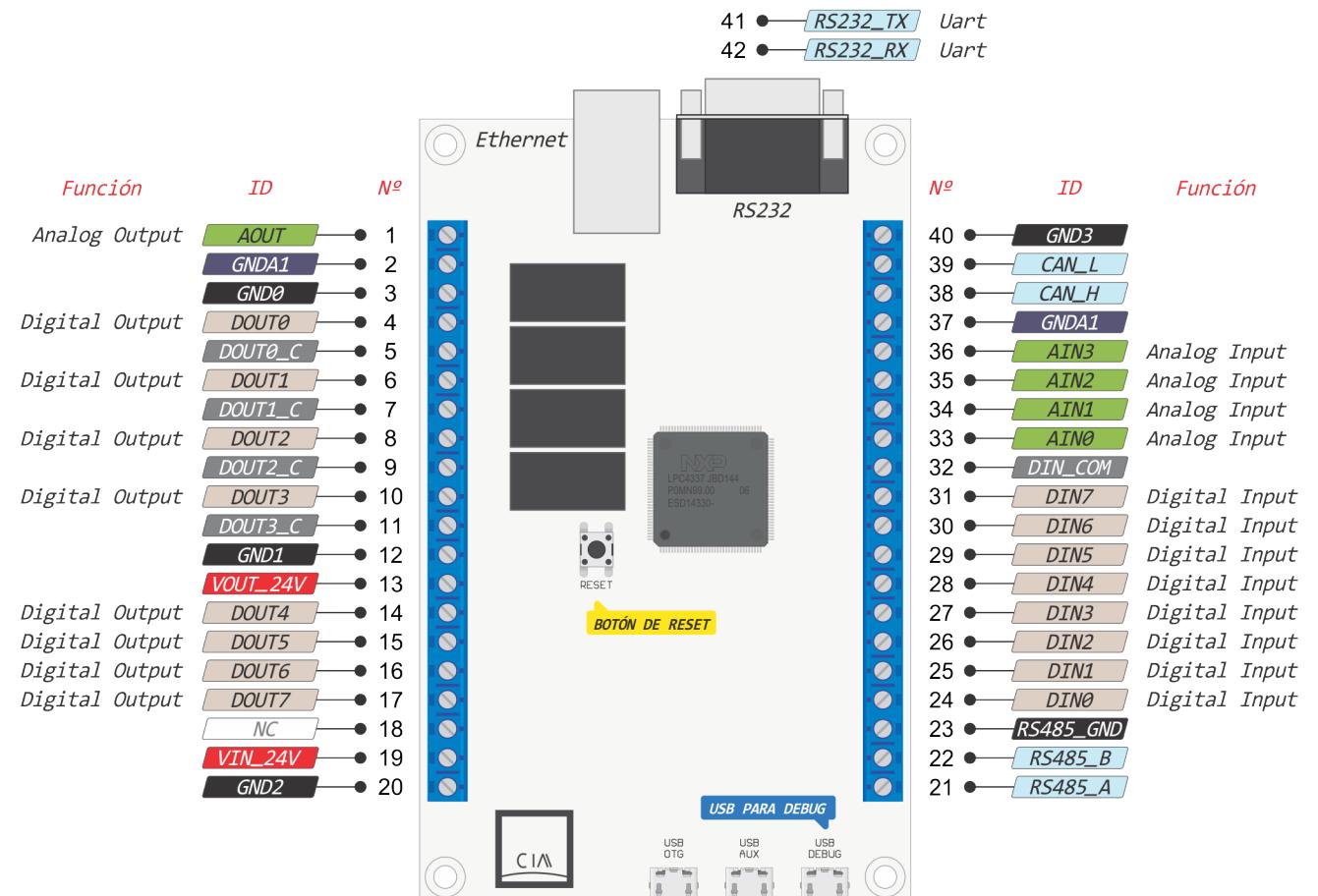


Figura 2.10: Mapeo de pines de la plataforma CIAA-NXP.

Debido a que los pines de la EDU-CIAA-NXP tienen muchas funcionalidades posibles se resumieron como DI (*Digital Input*), DO (*Digital Output*), AI (*Analog Input*), AO (*Analog Output*) y U (Uart).

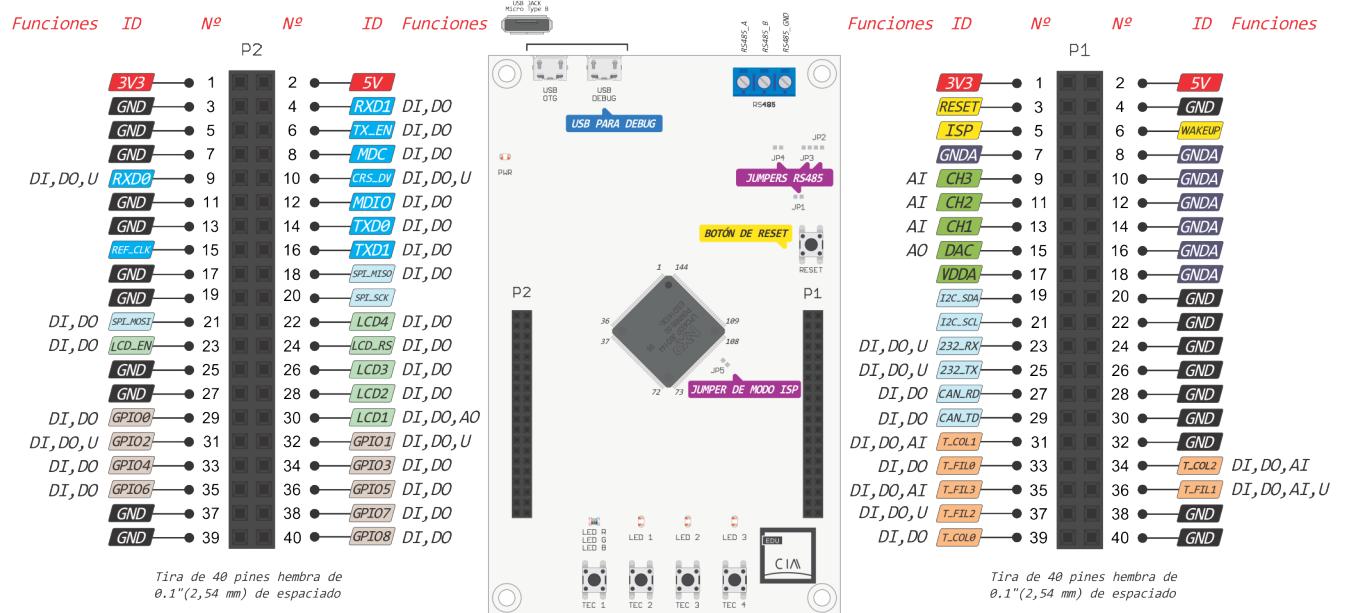


Figura 2.11: Mapeo de pines de la plataforma EDU-CIAA-NXP.

2.3.3. Modelo de la biblioteca en C

Debido a que readaptar el Firmware de la CIAA era una tarea muy ardua se optó por crear una biblioteca en C que de el soporte necesario para los métodos nativos.

Cada periférico se modela con una estructura de configuración, y las siguientes funciones asociadas:

DigitalIO:

```
1 bool_t digitalConfig( int32_t pinID, int32_t mode )
2 bool_t digitalRead( int32_t pinID )
3 bool_t digitalWrite( int32_t pinID, bool_t value )
```

AnalogIO:

```
1 |     bool_t analogConfig( int32_t pinID, int32_t mode )
2 |     int32_t analogRead( int32_t pinID )
3 |     bool_t analogWrite( int32_t pinID, int32_t value )
```

UART:

```
1 bool_t uartConfig( int32_t uartID, int32_t baudRate )
2 bool_t uartAdvancedConfig( int32_t uartID, uartConfig_t * uartConfig )
3 uint8_t uartRead( int32_t uartID )
4 bool_t uartWrite( int32_t uartID, uint8_t byte )
```

Se diseña la misma para permitir también su uso *bare-metal*.

2.4. Integración del desarrollo

Tanto el *port* para la CIAA como la biblioteca de periféricos son necesarios para el funcionamiento de HVM. Contando con estas partes ya se pueden realizar programas con HVM. Sin embargo, esto requiere de varias tareas manuales, entre ellas:

- Reemplazo manual de parte de los archivos generados de HVM por los nuevos archivos que dan soporte a la CIAA.
- Creación de un proyecto de Firmware de la CIAA.
- Integración al makefile de dicho proyecto de los archivos que conforman HVM.

Para llevar a cabo la automatización de estas tareas manuales es necesario modificar el *plugin* de Eclipse *Icecap tools*. Esta tarea excede el alcance del trabajo. La misma está siendo llevada a cabo en colaboración con el Ing. Leonardo Gassman.

Capítulo 3

IMPLEMENTACIÓN

3.1. Arquitectura del *port* de HVM para las plastaformas CIAA

Para la implementación de este trabajo se utiliza una arquitectura en capas con responsabilidades e interfaces definidas. El esquema general se presenta en la figura [3.1].

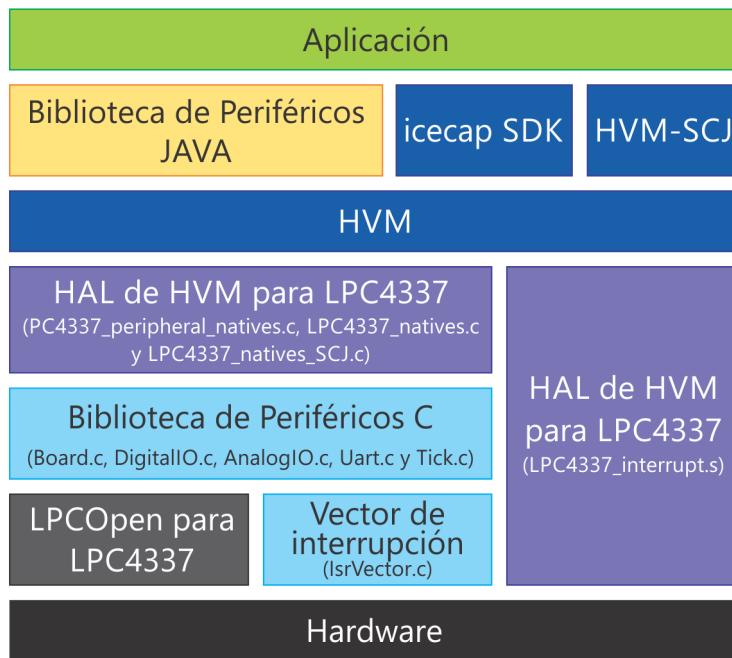


Figura 3.1: Arquitectura del port de HVM para las plastaformas CIAA.

Las capas preexistentes son **icecap SDK**, **HVM**, **HVM SCJ** que corresponden a las distintas partes de que conforman HVM; y **LPCOpen**, la biblioteca provista por NXP para el microcontrolador LPC4337. Las capas desarrolladas se describen a continuación.

Vector de interrupción

Esta capa se compone de un único archivo *IsrVector.c* que contiene el vector de interrupción. Para definir una nueva interrupción se debe agregar el nombre de la función correspondiente al *handler* de la misma y luego definir dicha función. El único *handler* utilizado es el de interrupción de **SysTick**.

Biblioteca de Periféricos C

La capa Biblioteca de Periféricos C, implementa la biblioteca de periféricos a bajo nivel, está diseñada para abstraer el hardware y presentar una API sencilla para el usuario. Se detalla en el apartado [3.4]. Utiliza la biblioteca LPCOpen y el vector de interrupción.

Capas que implementan la HAL de HVM para LPC4337

La HAL de HVM a bajo nivel se implementa en dos capas, una que solo puede acceder a la Biblioteca de Periféricos C, compuesta por los archivos,

- **LPC4337_peripheral_natives.c.** Esta capa simplemente implementa las funciones que respetan la firma de los métodos nativos y llaman a sus respectivas funciones de la Biblioteca de Periféricos C.
- **LPC4337_natives.c.** Contiene las funciones necesarias para el funcionamiento básico de HVM, se detalla en la sección [3.2].
- **LPC4337_natives_SCJ.c.** Resuelve las funciones de temporización para el **Planificador SCJ**. Utiliza el módulo *Tick.c* de la capa Biblioteca de Periféricos C. En la sección [3.5] se comunica su implementación.

y otra capa que accede directamente al hardware, implementada en *assembler* (archivo *LPC4337_interrupt.s*). Está última que resuelve el cambio de *threads* guardando el contexto y es la base del funcionamiento del módulo **Proceso SCJ**.

Biblioteca de Periféricos Java

Corresponde a la implementación de la biblioteca de periféricos en Java que finalmente dispondrá el usuario de HVM. Sus métodos nativos se encuentran implementados en *LPC4337_peripheral_natives.c*.

3.2. Port básico de HVM al microcontrolador NXP LPC4337

Para realizar el *port* de HVM se deben realizar una serie de tareas descriptas en la sección [2.2], en esta sección se informa como han sido llevado a cabo el subconjunto de las mismas necesario para la primer etapa del *port*, que consiste en poder ejecutar programas Java (no SCJ) en la CIAA.

Obtención de un entorno de desarrollo de C para la plataforma

Para esta tarea se utilizaron las herramientas libres, provistas por el Proyecto CIAA, para el desarrollo en C sobre sus plataformas de hardware. Se utilizó el instalador sencillo para Windows **CIAA-IDE-Suite 1.2.2** disponible para su descarga en: <https://github.com/ciaa/Software-IDE/releases/tag/v1.2.2>. Este paquete de software incluye:

- **Consola cygwin** (incluye paquetes: PERL, PHP, gcc x86, arm-none-ebi-gcc y ruby). Esta consola permite compilar un programa en C, generando un ejecutable para el microcontrolador LPC4337.
- **openOCD 0.9 x86/x64**. Se utiliza tanto para la descarga del ejecutable al microcontrolador, como para depuración mediante comandos GDB.
- **FTDI drivers libusb 1.0**. Se debe instalar por única vez el *driver* según la versión del sistema operativo. Reemplaza el driver que windows instala por defecto para el chip FTDI que contiene el *debugger* de las plataformas.

- **Eclipse (Luna)**. IDE para desarrollo de programas en lenguaje C para la CIAA. Desde el mismo se pueden gestionar proyectos, y permite la compilación y *debug* en un entorno gráfico (utiliza cygwin por debajo).
- **Firmware 0.6.1**. Firmware para utilizar las plataformas CIAA.
- **IDE4PLC v1.0.2**. Software-PLC. No utilizado en este trabajo final.
- **Uninstaller**. Desinstalador del paquete.

Para probar su correcto funcionamiento, se debe crear un proyecto de CIAA Firmware. Pueden realizarse dos tipos de proyectos, los que utilizan **OSEK** (el RTOS¹ de la CIAA, presentado en la sección [2.3]) y los proyectos **bare-metal** (programas que corren directo sobre el hardware sin la gestión de un Sistema Operativo). Como no se utiliza OSEK, se realizó un proyecto *bare-metal*, este consiste en la utilización de una estructura definida de carpetas donde existe una carpeta principal que contiene el nombre del proyecto y con sub-carpetas:

- **inc**. Aquí deben colocarse los *headers* del proyecto (archivos *.h*).
- **src**. En esta carpeta se alojan los archivos fuentes en del proyecto (de extensión *.c*).
- **mak**. Contiene el *makefile* del proyecto (archivo Makefile). En el mismo debe indicarse los módulos de Firmware de la CIAA que se desean agregar al proyecto así como los archivos a incluir en la compilación.

Para realizarlo, se parte del proyecto plantilla llamado *bare-metal*, que se incluye entre los ejemplos provistos (ubicados en *Firmware/examples*). Esta plantilla contiene un archivo que define la función *main*, e incluye a la biblioteca del microcontrolador *LPCOpen*, y otros archivos donde se define la tabla de vectores de interrupción y algunos *handlers* por defecto de las mismas.

Se crea un pequeño programa que hace destellar un *led* (*Blinky*) de la plataforma EDU-CIAA-NXP. Luego se modifica el archivo *makefile.mine* (ubicado en la raíz de *Firmware*) con los parámetros de plataforma a utilizar (*BOARD*) y la ruta dle proyecto creado (*PROJECT_PATH*). Finalmente, mediante **cygwin**, se compila y descarga a la plataforma EDU-CIAA-NXP.

NOTA: Se cuenta con la plataforma EDU-CIAA-NXP desde el comienzo del trabajo. En el caso de la CIAA-NXP se obtuvo casi a la finalización del mismo. Sin embargo, esto sólo influye en la definición de la biblioteca de periféricos, para todas las demás tareas, el uso de cualquiera de las dos plataformas es indistinto.

Construcción de un comando de compilación para compilar y *linkear* los archivos generados por HVM

En el apartado [2.1.2] se introduce como realizar un proyecto básico con HVM. Partiendo del comando de compilación presentado, se observa cuales son los archivos que compila, así como las definiciones necesarias.

Se crea un nuevo proyecto *bare-metal* y se ingresan los datos necesarios al *makefile* obteniéndose:

```

1 # Project Name: based on Project Path and used to define OSEK
2   configuration file
3 PROJECT_NAME = $(lastword $(subst $(_DS), , $(PROJECT_PATH)))
4
5 # Project path
6 # Defined $(PROJECT_PATH) in makefile.mine

```

¹Siglas de Sistema Operativo de Tiempo Real.

```

6  #HVM files
7
8
9 vpath classes.c ..
10 vpath icecapvm.c ..
11 vpath methodinterpreter.c ..
12 vpath methods.c ..
13 vpath icecapvm.c ..
14 vpath gc.c ..
15 vpath natives_allOS.c ..
16 vpath allocation_point.c ..
17 vpath print.c ..
18 vpath $(PROJECT_PATH)$(DS)src
19
20 HVM_PATH = $(PROJECT_PATH)$(DS)hvm
21
22 HVM_SRC_FILES = $(HVM_PATH)$(DS)classes.c \
23 $(HVM_PATH)$(DS)icecapvm.c \
24 $(HVM_PATH)$(DS)methodinterpreter.c \
25 $(HVM_PATH)$(DS)methods.c \
26 $(HVM_PATH)$(DS)gc.c \
27 $(HVM_PATH)$(DS)natives_allOS.c \
28 $(HVM_PATH)$(DS)allocation_point.c \
29 $(HVM_PATH)$(DS)print.c
30
31 CFLAGS += -DJAVA_HEAP_SIZE=$(JAVA_HEAP_SIZE) \
32 -DJAVA_STACK_SIZE=$(JAVA_STACK_SIZE) $(ADDITIONAL_FLAGS)
33
34 # source path
35 $(PROJECT_NAME)_SRC_PATH += $(PROJECT_PATH)$(DS)src
36
37 # include path
38 INC_FILES += $(PROJECT_PATH)$(DS)inc
39
40 # library source files
41 SRC_FILES += $(wildcard $($PROJECT_NAME)_SRC_PATH)$(DS)*.c \
42 $(HVM_SRC_FILES)
43
44 # Modules needed for this example
45 MODS ?= externals$(DS)drivers \
        modules$(DS)sapi
46

```

Nótese que los archivos generados por HVM deben estar en una carpeta llamada **hvm** dentro del proyecto.

Definición de la plataforma en el archivo `ostypes.h`

Para la definición de la plataforma se investigó la arquitectura del microcontrolador LPC4337 y su compilador *arm-none-eabi-gcc* para definir los distintos tipos de datos, resultando en:

```

1 #if defined(CIAA_NXP_TYPES_FOR_HVM)
2     #undef PACKED
3     #define PACKED
4     #define DATAMEM __attribute__ ((section (.data)))
5     #define PROGMEM
6     #define RANGE
7     #define pgm_read_byte(x) *((unsigned char*)x)

```

```

8 #define pgm_read_word(x) *((unsigned short*)x)
9 #define pgm_read_pointer(x, typeofx) *((typeofx)x)
10 #define pgm_read_dword(x) pgm_read_pointer(x, uint32*)
11 typedef int int32;
12 typedef unsigned int uint32;
13 typedef unsigned int pointer;
14 #endif

```

Definición de las funciones específicas de la plataforma. Archivo **LPC4337_natives.c**

Como ha sido adelantado, este archivo implementa la parte de la HAL de HVM para que funcionen los programas básicos de Java sobre esta máquina virtual.

- **void init_compiler_specifics(void).** Esta función inicializa la plataforma CIAA. Primero se inicializa el *clock* del sistema. Luego la **UART 2**. En ambas plataformas CIAA, la UART 2 del microcontrolador LPC4337 se encuentra conectada al chip FTDI del *debugger* y puede accederse vía USB en el mismo conector que se utiliza para depuración de la plataforma. De esta manera, es ideal para su utilización como salida de consola de HVM.
- **Pila de Java.** En este archivo se define la pila de Java como un vector cuyo tamaño será definido en el *makefile.mine*, esto es: **int32 java_stack[JAVA_STACK_SIZE];**
- **int32* get_java_stack_base(int16 size).** Devuelve un puntero a la pila de Java definida por el vector anterior: **return (int32*) java_stack;**
- **void initNatives(void).** En esta función se inicializa el resto de los periféricos de la biblioteca de C incluyendo el **SysTick** para interrumpir cada 10ms. Este periférico lo utilizan las funciones SCJ para manejar el tiempo del sistema.
- **void mark_error(void), void mark_success(void).** Como no se utiliza el sistema de pruebas de regresión, en esta primera iteración se dejan vacías.
- **read/writeXXToIO.** No se utilizan objetos hardware para tratar con los registros debido a la implementación de la biblioteca de C para periféricos. En consecuencia no se requieren.
- **init_memory_lock, lock_memory, unlock_memory.** Para las pruebas iniciales se dejan vacías pues el único *handler* de interrupción que hay es el de *SysTick* que no asigna memoria.
- **void sendbyte(unsigned char byte).** Imprime el *byte* recibido como parámetro a través de la **UART 2**.

Comprobación inicial de funcionamiento

Una vez completado el *port* básico se realiza una prueba inicial del sistema. Para llevarla a cabo, se utiliza el programa “*Hola Mundo HVM*” dado en [2.1.2]. Luego se crea un nuevo proyecto y dentro una carpeta nombrada **hvm**, se copian los archivos generados por HVM. El archivo **LPC4337_natives.c** se copia a la carpeta **src** del proyecto. En la figura [3.2] se muestra la estructura de carpetas resultante.

Para poder ser compilado debe modificarse, además, el archivo *makefile.mine*. Además de definir **BOARD** y **PROJECT_PATH** se debe agregar,

- La cantidad de memoria para el *Heap* de Java: **JAVA_HEAP_SIZE ?= 16384**.
- El tamaño de la Pila de Java: **JAVA_STACK_SIZE ?= 2048**.

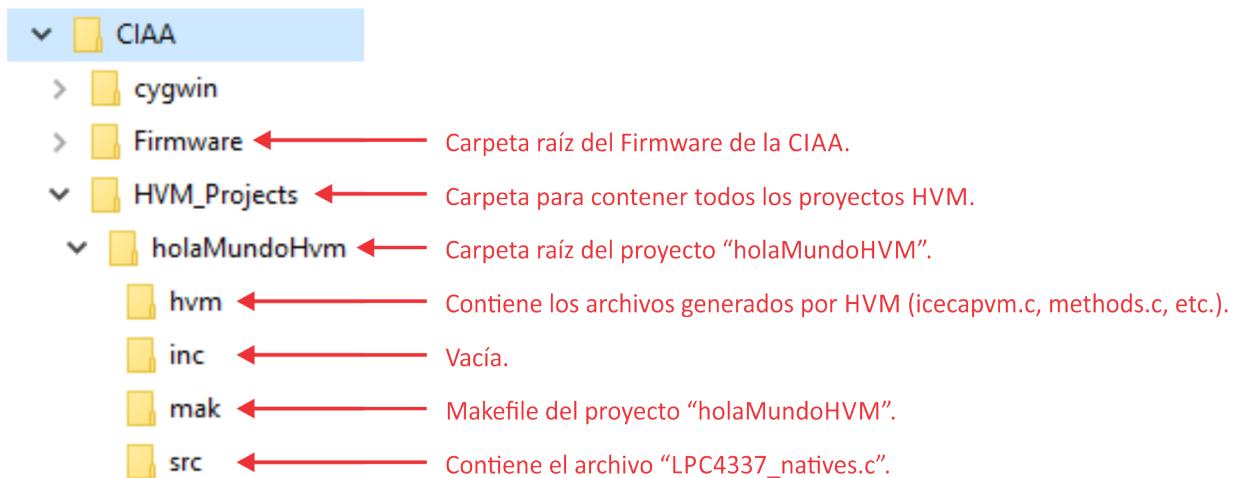


Figura 3.2: Estructura de proyecto de firmware CIAA “Hola Mundo con HVM”.

- Algunos *Flags* adicionales, siendo el más importante **-DLPC4337_TYPES_FOR_HVM** que representa la arquitectura para la cual se va a compilar HVM.

obteniéndose:

```

1 BOARD      ?=      edu_ciaa_nxp
2 PROJECT_PATH ?=  ..$(DS)HVM_Projects$(DS)holaMundoHVM
3
4 JAVA_HEAP_SIZE ?= 16384
5 JAVA_STACK_SIZE ?= 2048
6 ADDITIONAL_FLAGS ?= -g -Os -DLPC4337_TYPES_FOR_HVM -I ..

```

Se compila con **cygwin** y se descarga a la plataforma. Después se conecta a una Terminal Serie, se ingresan los parámetros de conexión y finalmente se resetea la EDU-CIAA-NXP placa para recibir el mensaje “Hola Mundo HVM” en la Terminal.

3.3. Implementación de la biblioteca para manejo de periféricos

3.4. Biblioteca para manejo de periféricos (C)

Módulo **sAPI**: Está compuesta por los módulos:

```

sAPI.h
sAPI_Board.c, sAPI_Board.h
sAPI_DataTypes.h
sAPI_Delay.c, sAPI_Delay.h
sAPI_DigitalIO.c, sAPI_DigitalIO.h
sAPI_IsrVector.c, sAPI_IsrVector.h
sAPI_Tick.c, sAPI_Tick.h

```

Este actualiza una variable global y luego ejecuta una función cuya dirección está contenida en una variable global para permitir engancharse a la interrupción de tick.

3.5. Biblioteca para manejo de periféricos (Java)

Está formada por los módulos Device.Java, Pin.Java, Peripheral.Java, DigitalIO.Java, AnalogIO.Java, Uart.Java, Delay.Java, Button.Java y Led.Java.

3.5.1. CIAA-NXP

3.5.2. EDU-CIAA

3.5.3. Java

3.6. Port de HVM SCJ al microcontrolador NXPLPC4337

Definir las funciones específicas de la plataforma

LPC4337_natives_SCJ.c

LPC4337_interrupt.s

3.6.1. Funciones para SCJ

3.7. Integración y uso de las herramientas desarrolladas

Capítulo 4

VALIDACIÓN Y RESULTADOS

En las siguientes secciones se exponen distintas aplicaciones que prueban el funcionamiento del desarrollo. Estas son:

- Ejemplos de aplicaciones Java utilizando periféricos de la EDU-CIAA-NXP mediante la biblioteca desarrollada (sección [4.1]).
- Un ejemplo de aplicación Java SCJ utilizando el concepto de Proceso SCJ para demostrar el funcionamiento del cambio de contexto (sección [4.2]).
- Otro ejemplo de aplicación Java SCJ utilizando el concepto de Planificador SCJ (sección [4.3]).
- Una aplicación SCJ completa (sección [4.4]).

Finalmente se exponen las características del IDE desarrollado (sección [??]).

4.1. Ejemplos de aplicaciones Java utilizando periféricos

```
1 package ar.edu.unq.embebidos;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         // Instanciación de los objetos Led
7         Led ledR = new Led(0);
8         Led led1 = new Led(4);
9
10        // Instanciación de los objetos Pulsador
11        Pulsador tec1 = new Pulsador(0);
12        Pulsador tec2 = new Pulsador(1);
13
14        // Instanciación del objetos Uart
15        Uart serialPort = new Uart();
16
17
18        // Se configura la UART2
19        serialPort.config();
20
21        // Envío un carácter tipo prompt
22        serialPort.write('>');
23    }
}
```

```
24         while (true) {
25
26             if ( tec1.isPulsado() ) {
27                 ledR.on();
28             }else {
29                 ledR.off();
30             }
31
32             if ( tec2.isPulsado() ) {
33                 led1.on();
34             }else {
35                 led1.off();
36             }
37
38             // Recibo un caracter
39             serialPort.read();
40
41             // Si es distinto de Null lo envío
42             if( serialPort.getRxData() != 0 ){
43                 // Envio un caracter
44                 serialPort.write( serialPort.getRxData() );
45             }
46
47         }
48     }
49 }
```

4.2. Ejemplo de Procesos SCJ

La clase *Process* implementa el concepto de proceso. Se puede utilizar para iniciar la ejecución concurrente del método *run* de una instancia de *Runnable* con una pila suministrada en la forma de un vector de enteros en Java.

El siguiente ejemplo implementa un mecanismo de conmutación de co-rutina simple. Se utiliza para demostrar el correcto funcionamiento de las funciones nativas que implementan los procesos SCJ.

```
1 // CODIGOoooooooooooooooooooo  
2
```

Para cambiar de un proceso a otro, el proceso en ejecución llama *TransferTo* con el siguiente proceso a ejecutar como parámetro.

La llamada a *TransferTo* entra al espacio de código nativo, y mediante una secuencia corta de instrucciones en *assembler* inserta en la pila el estado actual de la CPU, el puntero de pila se almacena en el objeto proceso que se está ejecutando actualmente, se recupera un nuevo puntero de pila del próximo objeto proceso y se saca de la pila de este el nuevo estado.

El efecto que se consigue es que cuando *TransferTo* retorna, no retorna al proceso que lo llama, sino al siguiente proceso en el punto en que llamó a *TransferTo* previamente.

4.3. Planificador SCJ

HVM permite implementar planificadores¹ directamente en Java. El planificador debe buscar el siguiente proceso a ejecutar y realizar una transferencia a dicho proceso. El planificador es llamado en los puntos de reprogramación²

El intérprete llama periódicamente a una función nativa *yieldToScheduler*. Esta función (implementada en *natives_allOS.c*) chequea una variable global llamada *systemTick* para no reprogramar las tareas muy seguido. La variable *systemTick* debe ser actualizada a intervalos razonables.

HVM SDK contiene tres tipos de planificadores:

1. *PriorityScheduler*
2. *CyclicScheduler*
3. Un planificador que simula el planificador del paquete estándar *Java Thread*.

En el siguiente ejemplo se prueba el funcionamiento de uno de ellos para comprobar el correcto funcionamiento de las funciones nativas que manejan la temporización a través del SysTick.

```

1 // CODIGOoooooooooooooooooooooooo
2

```

4.4. Ejemplo de aplicación SCJ completa

Finalmente se expone un ejemplo de un programa SCJ completo para demostrar el correcto funcionamiento del sistema.

```

1 // CODIGOoooooooooooooooooooo
2

```

La aplicación SCJ está compuesta por un *mission sequencer*, una *mission* y un único *handler* de eventos periódicos. Los *handler* son objetos planificables³, en este caso planificados por el *PriorityScheduler* SCJ Nivel 1.

El período del evento que lanza el *handler* es de un segundo. Luego de ejecutarse cinco veces solicita un *mission termination* para finalizar la misión.

Se requieren los siguientes recursos de memoria:

- *Immortal memory area*. Todas las aplicaciones SCJ utilizan memoria inmortal.
- El área de memoria privada del Secuenciador de misión y una pila de ejecución. Un secuenciador de misión SCJ es también un *event handler*, entonces necesita una pila de ejecución.
- La memoria de misión.
- El área de memoria privada del *handler* y su pila de ejecución.
- La pila de ejecución del *priority scheduler*. Este planificador se invoca en los puntos de reprogramación. Corre utilizando su propia pila de ejecución aunque no es un verdadero *handler*.

¹Schedulers en inglés.

²Reschedule points en inglés.

³En inglés schedulable objects.

La cantidad de memoria correcta para cada una de las áreas es difícil de predecir. Para elegirlos correctamente primero se utilizan valores conservadores y luego se puede realizar un *Memory Profiling* para identificar cuanta memoria se está utilizando verdaderamente.

Capítulo 5

CONCLUSIONES Y TRABAJO A FUTURO

5.1. Conclusiones

En el presente Trabajo Final se ha logrado obtener un entorno de desarrollo para aplicaciones Java SCJ sobre las plataformas CIAA-NXP y EDU-CIAA-NXP, que además de ser software libre, cubre las necesidades planteadas, tanto al ofrecer programación orientada a objetos, así como funcionalidades de tiempo real para entornos industriales, sobre sistemas embebidos.

Como subproducto, se obtiene además una biblioteca con una API sencilla para el manejo de periféricos de las plataformas CIAA-NXP y EDU-CIAA-NXP, que puede utilizarse en aplicaciones Java, o directamente en lenguaje C, debido a su diseño como módulo de Firmware de la CIAA. Se ha tenido especial cuidado en el diseño de esta biblioteca para que la misma sea lo más genérica posible logrando que además se comporte como una HAL.

El desarrollo de este Trabajo Final demandó la articulación de conocimientos adquiridos a lo largo de la Carrera de Especialización en Sistemas Embebidos, en particular, las asignaturas:

- **Arquitectura de microprocesadores.** Se utilizaron de esta asignatura los conocimientos adquiridos sobre la arquitectura ARM Cortex M necesarios para implementar en lenguaje *assembler* las funciones que realizan el cambio de contexto, necesarias para que funcione el concepto de Proceso SCJ.
- **Programación de microprocesadores.** De esta asignatura se aprovecha la experiencia sobre lenguaje C para microcontroladores de 32 bits y el manejo de sus periféricos. Fue de especial importancia, debido a que, a excepción de unas pocas, todas las funciones para portar HVM a la CIAA debían realizarse en lenguaje C. Este lenguaje también se utilizó en la creación de la API para el manejo de periféricos.
- **Ingeniería de software en sistemas embebidos.** Se aplican de la misma las metodologías de trabajo, provenientes de la ingeniería de software, que aportan calidad y eficiencia al desarrollo. En particular, diseño iterativo y manejo repositorios de software, diseño modular y en capas.
- **Gestión de Proyectos en Ingeniería.** Durante esta se desarrolló el Plan de Proyecto del Trabajo Final, permitiendo desde un principio tener una clara planificación del trabajo a realizar.
- **Sistemas Operativos de Propósito General.** Se aprovechan los conocimientos adquiridos sobre Linux para probar las herramientas desarrolladas sobre este sistema operativo.

- **Sistemas Operativos de Tiempo Real (I y II).** De estas asignaturas se aplica el conocimiento obtenido sobre planificadores de tareas expropiativos y la manera en que trabajan. Esto ha sido muy importante para la realización de este Trabajo Final. También la creación de módulos de Firmware para la CIAA.
- **Desarrollo de Sistemas Embebidos en Android.** La plataforma Andoid es un claro caso de éxito de la aplicación de un lenguaje POO en sistemas embebidos (aunque no sea para aplicaciones industriales). Estas aplicaciones se realizan en lenguaje Java. Si bien se contaba con experiencia en programación orientada a objetos en otros lenguajes, esta asignatura fue para el autor el primer acercamiento a dicho lenguaje. En consecuencia, mucho de lo aprendido colaboró en la decisión de llevar a cabo este trabajo.
- **Diseño de Sistemas Críticos.** Los conceptos aprendidos en esta asignatura contribuyeron a comprender, inmediatamente, las importantes implicancias de poder programar aplicaciones SCJ en sistemas embebidos, que provee HVM, para aplicaciones industriales.

Además, se han adquirido aprendizajes en las temáticas:

- Programación de aplicaciones en lenguaje Java.
- Especificaciones de Java, entre ellas, RTSJ y SCJ.
- Programación de aplicaciones SCJ.
- Experiencia en implementación de cambio de contexto, procesos y planificadores.
- Máquinas Virtuales de Java para sistemas embebidos y su funcionamiento interno.
- Desarrollo de una biblioteca Java para manejo de periféricos en sistemas embebidos. Conexión con bibliotecas nativas en lenguaje C.

Por lo tanto, se llega a la conclusión que los objetivos planteados al comenzar el trabajo han sido alcanzados satisfactoriamente, y además, se han adquirido conocimientos muy importantes para la formación profesional del autor.

5.2. Trabajo a futuro

Como labor a futuro, pueden realizarse las siguientes tareas:

- Programar los *Launchers* para las plataformas CIAA y EDU-CIAA-NXP para permitir *debuggear* directamente en Java.
- Completar una versión del IDE lista para instalar.
- Integrar el *port* de la CIAA al repositorio oficial de HVM.
- Investigar HVM-TP, el nuevo desarrollo de Stephan Erbs Korsholm. Es una máquina virtual de Java *Time Predictable* y Portable para sistemas embebidos *Hard Real-Time*, que toma como base a HVM, extendiéndola y mejorando sus capacidades.

REFERENCIA BIBLIOGRÁFICA

1. The Open Group (2013). *Safety-Critical Java Technology Specification JSR-302*. Version 0.94, 25-06-2013. Oracle. On line, última consulta 12-10-2015 http://download.oracle.com/otn-pub/jcp/safety_critical-0_94-edr2-spec/scj-EDR2.pdf.
2. Stephan E. Korsholm (2014). *The HVM Reference Manual*. Icelabs, Dinamarca. On line, última consulta 05-10-2015 en <http://icelab.dk/resources/HVMRef.pdf>.
3. Stephan E. Korsholm (2014) Y ALGUNO MAS. *TITULO PAPER*. Icelabs, Dinamarca. On line, última consulta 09-10-2015 en http://icelab.dk/resources/SCJ_JTRES12.pdf.
4. Wikipedia, the free encyclopedia. *Dangling pointer*. Wikipedia, the free encyclopedia. On line, última consulta 14-10-2015 en https://en.wikipedia.org/wiki/Dangling_pointer.