Proyecto de Grado

Interfaz USB genérica para comunicación con dispositivos electrónicos

Informe Final

A/C Andrés Aguirre, A/C Pablo Fernández y A/C Carlos Grossy.

Tutores: MSc Ing. Gonzalo Tejera y MSc Ing. Alexander Sklar

Instituto de Computación - Facultad de Ingeniería

Universidad de la República Oriental del Uruguay

14 de diciembre de 2007

Resumen

Debido a que en general los dispositivos electrónicos existentes (sensores, actuadores, displays, ADCs, DACs, etc.) utilizan múltiples interfaces y protocolos de comunicación ajenos al contexto de las computadoras personales, genera que la interacción no sea una tarea trivial y en general implique desarrollar soluciones particulares para lograr su integración.

Esta heterogeneidad motiva la búsqueda de un medio de comunicación existente en un PC, lo suficientemente versátil para satisfacer la mayoría de los requerimientos. Desde hace unos años la tecnología USB se ha convertido en un estándar, lo que ha llevado a una proliferación de dispositivos y un auge en su uso. La facilidad de uso, ancho de banda y funcionalidad Plug&Play son algunas de las características más atractivas para utilizar al USB como medio de comunicación.

La motivación de este proyecto se centra en lograr de una manera sencilla, la comunicación entre una PC y un conjunto de dispositivos electrónicos no necesariamente pensados para interactuar con una computadora. La solución está conformada por un hardware construido a medida basado en un microcontrolador, con firmware configurable vía software por medio de USB y conjunto de elementos de software que permiten la comunicación y manejo de los dispositivos desde las aplicaciones de usuario. El firmware desarrollado se caracteriza por su modularidad y extensibilidad, mientras que en el software del PC se destaca su orientación a objetos, soporte de múltiples instancias de hardware y de varias plataformas. Todos los elementos funcionan en conjunto de forma de dar una respuesta integral a la problemática permitiendo ocultar a los ojos de los usuarios toda la complejidad de la interacción con los dispositivos electrónicos.

Entre las principales contribuciones de este proyecto se destacan el desarrollo de un framework que permite acelerar la integración de dispositivos electrónicos con el PC por medio de la tecnología USB, pues sólo se necesita incorporar la funcionalidad específica para el manejo de los mismos. También permite un alto grado de reutilización de las funcionalidades específicas ya construidas, pues por medio de su composición se puede elaborar una nueva solución para el manejo de dispositivos más complejos o compuestos. A su vez el relevamiento del estado del arte de la tecnología USB y de los microcontroladores y drivers existente hoy en día que la soportan, requirió un gran esfuerzo y es considerado como un aporte de este proyecto.

Este proyecto habilita ampliar el horizonte de utilización por parte de un PC de elementos físicos así como el universo de usuarios que lo utilizan, logrando reducir la brecha de conocimientos necesarios para resolver las problemáticas de conectividad e interacción con dispositivos.

Palabras claves: USB, Microcontroladores, Framework de desarrollo, Controladoras de entrada / salida, Prototipado rápido, Programación embebida.

Agradecimientos

- Gonzalo Tejera y Alexander Sklar por la oportunidad de hacer el proyecto, apoyo e invaluables consejos durante el desarrollo del mismo.
- A nuestros compañeros de trabajo y superiores por su tolerancia, en particular a Cristina Lovesio.
- Texas Instruments, Microchip, Freescale, Philips por sus donaciones de muestras de semiconductores.
- Agradecimientos personales:
  - Carlos Grossy agradece a: Virginia López, madre y hermanos por su paciencia y apoyo durante este proyecto.
  - Rafael Fernández agradece a: familia y Claudia García por su paciencia y apoyo durante este proyecto y Noelia Pissani por su colaboración al conseguir fichas USB hardware.
  - Andrés Aguirre agradece a: Omar Aguirre, Fabiana Bianchi y Ana Dorelo por la paciencia y apoyo constante.

Índice general

Lista de algoritmos

Índice de figuras

Índice de cuadros

Parte I

Generalidades

Capítulo 1

Introducción

En los primeros meses del año 2006 las inquietudes personales de los miembros del grupo motivaron la gestación de la idea básica del proyecto que finalmente derivó en la presentación como propuesta del mismo. En un comienzo se contaba con muchas ideas muy ambiciosas pero con muy pocos conocimientos sobre la mayoría de las áreas involucradas. Más allá de las limitaciones de conocimientos y la amplitud de las áreas a investigar se pensó que era un buen desafío para realizar como proyecto de grado.

Gracias a los tutores que en su momento notaron la viabilidad de la idea y la apoyaron, se presentó una propuesta ante la Comisión de Proyectos que posteriormente fue aprobada. De inmediato se destino la totalidad del tiempo al aprendizaje de las tecnologías relacionadas para poder contar con un conocimiento básico con que realizar un relevamiento del estado del arte adecuado. Esto fue una tarea muy exigente y requirio más tiempo del planificado pero permitió seleccionar y plasmar de mejor manera todas las ideas generales que se tenían al comienzo del proyecto en un conjunto inicial de requerimientos.

1.1. Motivación

Durante muchos años la única forma de interactuar con dispositivos externos desde a un computador personal (Personal Computer) (PC) fueron los puertos seriales y paralelos, esto llevo a que se utilizaran ampliamente en multiplicidad de dispositivos. Sus principales carácteristicas son su simplicidad de manejo vía software y facilidad de inclusión en distintos dispositivos. En la última década han aparecido nuevos medios o canales de comunicación, tales como Bluetooth, Fidelidad Inalámbrica (Wireless Fidelity) (WiFi), FireWire, Bus Universal en Serie (Universal Serial Bus) (USB), etc. Estos permitieron mejorar las velocidades de comunicación y calidad de datos y lentamente tornaron a los puertos paralelos y seriales en obsoletos, hasta llegar al día de hoy en que se obtienen sólo de manera opcional en los nuevos PC.

Dentro de todas estas nuevas tecnologías que aparecieron la que tuvo mayor aceptación y difusión entre los usuarios fue el USB, debido a su facilidad y versatilidad de escenarios de uso. Esta simplicidad desde el punto de vista del usuario de los dispositivos USB tiene como contrapartida una mayor complejidad para los desarrolladores de software y hardware, lo cual es un gran problema al momento de interactuar con dispositivos electrónicos.

Frente a estas adversidades que presenta el entorno actual aparece la necesidad de buscar una forma de reducir el grado de complejidad y conocimientos necesarios para poder desarrollar software que utilice la tecnología USB. Al mismo tiempo, se requiere una plataforma base que incorpore componentes de hardware reutilizables y que en conjunto formen una solución genérica para la comunicación con dispositivos electrónicos diversos.

En el contexto de lo antedicho aparecen un conjunto de desafíos como son el desarrollo de controladores (drivers) y construcción de piezas de hardware que brinde una solución genérica reutilizable, desplazando el manejo de los casos particulares a componentes específicos. Esto proporciona el beneficio de evitar que cada vez que se necesite conectar un nuevo dispositivo, se comience desde cero. A su vez permite recuperar y potenciar la característica de facilidad de manejo que poseían los puertos seriales y paralelos, explotando todas las capacidades brindadas por USB.

Otra motivación importante que posee este proyecto es poder construir sistemas que se basen en la conjunción de dispositivos electrónicos simples con las capacidades de procesamiento, almacenamiento y comunicación de los PC actuales de forma de poder resolver una gran variedad de situaciones complejas y distintas, centrando principalmente la coordinación de las actividades en el PC.

1.2. ¿Qué es el proyecto?

Es una solución a la necesidad de comunicar de forma sencilla y genérica dispositivos electrónicos no necesariamente pensados para interactuar con un PC, la solución se basa en tres puntos: un componente de hardware, un medio de comunicación (USB), una arquitectura conformada por software, firmware, un pila de protocolos de comunicación y un driver USB genérico.

El componente de hardware es una interfaz o adaptador que permite comunicar los dispositivos externos con el PC, se busca que sea lo más abarcativo posible de forma de permitir la conexión de un gran número de dispositivos como son: sensores, actuadores, conversores analógico digital (Analog Digital Conversor) (ADC), conversores digital analógico (Digital Analog Conversor) (DAC), robots, etc. Además de utilizar interfaces seriales estándar, también se desea la facilidad de poder definir interfaces propias, para ello el hardware no tiene predefinido conectores para un puerto serial u otra interfaz sino que posee un único conector genérico configurable dinamicamente por software.

Como se mencionó en la sección 1.1, la selección de la tecnología USB como medio de comunicación se debe además de lo ya descripto, por su disponibilidad actual en todos los PC, sus características enchufar y listo (Plug and Play) y enchufado en caliente (Hot Plug) para la configuración y conexión (evitando el la necesidad de desarmar el PC para agregar un nuevo dispositivo).

La arquitectura es lo que permite integrar los componentes de hardware y la tecnología USB para lograr el comportamiento deseado. Una de sus cualidades más importante es que permite ocultar las complejidades de la comunicación USB y brindar al programador de aplicaciones una manera sencilla de interactuar con los dispositivos por medio de lenguajes de alto nivel. Para lograr ésto, es necesario contar con un protocolo que facilite y estandarice la comunicación, un driver USB que permita manejar el trafico de información en forma genérica, un firmware que habilite el procesamiento y toma de decisiones sobre el comportamiento de los dispositivos en el propio hardware, una interfaz de programación de aplicaciones (Application Programming Interface) (API) que brinde las funcionalidades mínimas para interactuar con los dispositivos y una biblioteca en lenguaje orientado a objetos que permita el rápido y fácil uso de la solución.

Se busca que la arquitectura sea modular y extensible de manera de permitir una administración sencilla de los componentes de firmware encargados del manejo de los dispositivos, logrando de ésta forma la fácil adecuación de la solución para el uso con nuevos dispositivos. Lo antedicho permite la reutilización de funcionalidades ya implementadas (dependiendo de las características de los dispositivos) en la construcción mediante composición de nuevas soluciones.

Todos estos atributos permiten que el usuario programador se concentre en la interacción con el dispositivo electrónico y no en la forma o medio de comunicación.

1.3. Que no es el proyecto

Durante el desarrollo de este proyecto fueron apareciendo un conjunto de conceptos erróneos sobre el alcance y características de uso de la solución, a continuación intentaremos detallar que elementos no forman parte de la solución construida.

- **No es un adaptador USB:** En general la solución se subestima pues se entiende que simplemente es una especie de adaptador USB a una interfaz específica como son la serial o paralela. Esto es totalmente erróneo, ya que si fuera un adaptador se perdería la generalidad y parte de las prestaciones USB además de no poder incorporar funcionalidades específicas para el manejo de los dispositivos en dicho hardware. Además se obligaría a que todo el procesamiento de la interacción con los dispositivos fuera desde el PC y reducido a un único tipo de interfaz, es decir no se podrían interactuar con un motor paso a paso serial y un termómetro de interfaz inter-circuitos integrados (Inter Integrated Circuit) (I^2C) al mismo tiempo.

- **No es un hub USB:** Este proyecto apunta a permitir la conexión de dispositivos que no poseen la interfaz USB en forma nativa, pues seria un contrasentido conectarlos por medio de nuestra solución y no en firma directa. Además se perdería todo el soporte de drivers estándar y aplicaciones de uso general que ya vienen con los sistemas operativos que soportan USB.

- **No es un dispositivo específico:** Una de las motivaciones de este proyecto es poder interactuar con la mayor cantidad posible de dispositivos electrónicos, los cuales poseen distintas necesidades de conexión e interfaces. Si se tomara partido por algunos tipos de dispositivos se limitaría el alcance del uso de la solución, lo que no es deseado. Además, tampoco se busca que la solución sea una especie de plataforma para la construcción de dispositivos USB específicos como pueden ser mouses, memorias o cámaras web pues los sistemas operativos no podrían reconocerlos como tales.

- **No es sólo hardware:** Lo primero que se atina a pensar cuando se ve la solución es que es una pieza de hardware que resuelve toda la problemática, pero en realidad la solución es más amplia pues incluye un firmware que ejecuta en el componente de hardware y un software que funciona como contraparte en el PC. Su accionar en forma conjunta es lo que permite dar respuesta a las necesidades planteadas.

- **No es necesario crear un driver para cada dispositivo:** La utilización de un driver USB genérico y de una arquitectura abierta y extensible que permite la incorporación de funcionalidad específica para el manejo de cada dispositivo por medio de la programación en alto nivel de su comportamiento, evita la creación de drivers particulares para cada uno de ellos.

1.4. Objetivos

El objetivo principal de este proyecto es poder interactuar con dispositivos electrónicos no necesariamente pensados para ser usados desde un PC por medio de una solución que conjuga: un hardware base reutilizable, una arquitectura de software modular y extensible y el medio de comunicación USB. A continuación se detallan un conjunto de objetivos más específicos que logran satisfacer el objetivo principal previamente descrito:

- Diseñar e implementar un hardware en forma de una placa base que resuelva la problemática de la comunicación USB con el PC y brinde una interfaz para conectar los dispositivo electrónicos.

- Diseñar e implementar una arquitectura de software modularizada y extensible que permita definir en el firmware de la placa base sus características físicas, los protocolos de comunicación y las funcionalidades específicas para la interacción con cada dispositivo. En el PC se cuente con una API para la interacción con el hardware y con una biblioteca de clases orientadas a objetos para la integración con aplicaciones de alto nivel.

- Construir un diseño que permita que la solución sea multiplataforma, priorizando su implementación para Windows y Linux.

- Diseñar y construir un driver genérico USB propio[1] para la plataforma Linux que corra en modo núcleo.

- Desarrollar un conjunto de utilitarios para la configuración de la placa base y carga de los módulos de manejo específico de dispositivos por medio del USB.

- Ocultar la complejidad USB sin perder las cualidades inherentes, la tecnología USB es más compleja que las soluciones serial o paralela, por lo tanto es vital poder ocultar la complejidad de esta interfaz para que el usuario se preocupe solamente de la problemática de la interacción con el dispositivo y no en la forma en que se comunica.

[1] Se entiende por driver genérico aquel que puede obtener la información propia del dispositivo, enviar y recibir datos.

- Construir un caso de uso (prototipo) relevante que permita mostrar las cualidades funcionales y técnicas de la solución implementada.

- La solución debe poseer un modo de funcionamiento extra (modo fachada)[2] que permita reutilizar los drivers y aplicaciones de usuario ya existentes en el PC por medio de la simulación del comportamiento de un dispositivo USB específico.

1.5. Organización del documento

Este documento está organizado en tres partes: *Generalidades*, *Arquitectura* y *Experimentos y Resultados*, además de los apéndices: *Glosario* y *Herramientas*. Esta organización pretende en primer lugar hacer una introducción al proyecto y a las tecnologías utilizadas, para luego centrarse en los distintos elementos y características de la arquitectura de software y finalmente presentar los productos construidos así como las conclusiones y trabajos a futuro que deja el proyecto.

En la parte *Generalidades* del documento se encuentran los capítulos:

- Capítulo 1 - ***Introducción:*** Se presenta como fue la gestación del proyecto, sus motivaciones, objetivos y alcance.

- Capítulo 2 - ***Resumen del Estado del Arte:*** Las áreas que involucran este proyecto son diversas pero las que se presentan en este capítulo son las más relevantes para sentar las bases de conceptos básicos y vocabulario. Comienza mostrando la tecnología USB por medio de su topología, características de comunicación (velocidades, tipos de transferencias, etc.), proceso de enumeración y jerarquía de dispositivos estándar. Luego se describen las características principales de los modelos de drivers de las plataformas Windows y Linux y presentan algunas de las opciones existentes actualmente. También se analizan algunas de las opciones de conectividad USB, contrastando sus características y prestaciones, para luego justificar la seleccionar de una de ellas en la implementación del hardware del proyecto. Por último se comentan algunos proyectos relacionados mostrando sus ventajas y falencias. Cabe destacar que las imagenes utilizadas en este capítulo son extraidas de los documentos, artículos y libros citados en la bibliografía del documento Estado del Arte [2].

En la parte *Arquitectura* del documento se encuentran los capítulos:

- Capítulo 3 - ***Introducción:*** Se presentan los elementos físicos de la solución así como los componentes de la arquitectura de software.

- Capítulo 4 - ***Arquitectura en el Baseboard:*** Se detallan los componentes electrónicos (microcontrolador, conectores, relojes, etc.) que constituyen el hardware base de la solución. Además se introducen los distintos elementos de firmware que permiten la comunicación y manejo de los dispositivos, esta presentación abarca no sólo la descripción a nivel de diseño de los componentes sino que también se explica brevemente como es el proceso de programación y configuración del firmware del hardware base.

- Capítulo 5 - ***Arquitectura en el PC:*** Se detallan los elementos que conforman la arquitectura de software de la solución en el PC, presentando sus principales características funcionales y diseño detallado. También se presenta el diseño de una propuesta de clases base para facilitar el uso de la solución desde un lenguaje orientado a objetos como es Java y por último se explica las características más importantes de un driver genérico implementado durante el proyecto.

- Capítulo 6 - ***Comunicación:*** En este capítulo se hace especial hincapié en la descripción del funcionamiento en conjunto de todos los elementos de la arquitectura de software, mostrando en detalle como es el flujo de ejecución de sus principales funcionalidades, además de presentar los protocolos de comunicación que define la solución.

[2] Este objetivo se eliminó posteriormente del alcance del proyecto, por más detalles ir a la sección 7.1.

- Capítulo 7 - ***Decisiones de diseño e Implementación:*** Durante el desarrollo de este proyecto fueron apareciendo distintas dificultades o alternativas al enfrentarse con ciertos temas y para los cuales se tuvo que tomar una postura, este capítulo muestra las decisiones de diseño más importantes que se tomaron justificando el porque de las mismas.

En la parte *Experimentos y Resultados* del documento se encuentran los capítulos:

- Capítulo 8 - ***Artefactos construidos:*** Este capítulo hace un repaso de los artefactos de software y hardware que se fueron construyendo durante el desarrollo del proyecto. Se describen entre otras cosas, como fue evolucionando el hardware base de la solución y los prototipos de dispositivos que se construyeron para su verificación.
- Capítulo 9 - ***Conclusiones y Trabajo a futuro:*** Como cierre del documento se presentan las conclusiones del proyecto por medio de las características más importantes de la solución construida y de los aportes que brinda la misma. Como forma de cierre se describen posibles trabajos a futuro de este proyecto tanto en el área de investigación como de mejoras e incorporaciones de funcionalidades a las ya existentes.

En los apéndices del documento se encuentran el glosario de los términos utilizados así como la descripción de las herramientas utilizadas para la documentación, notificación y seguimiento de errores, publicación de información del proyecto en la web, etc.

1.6. Etapas del proyecto

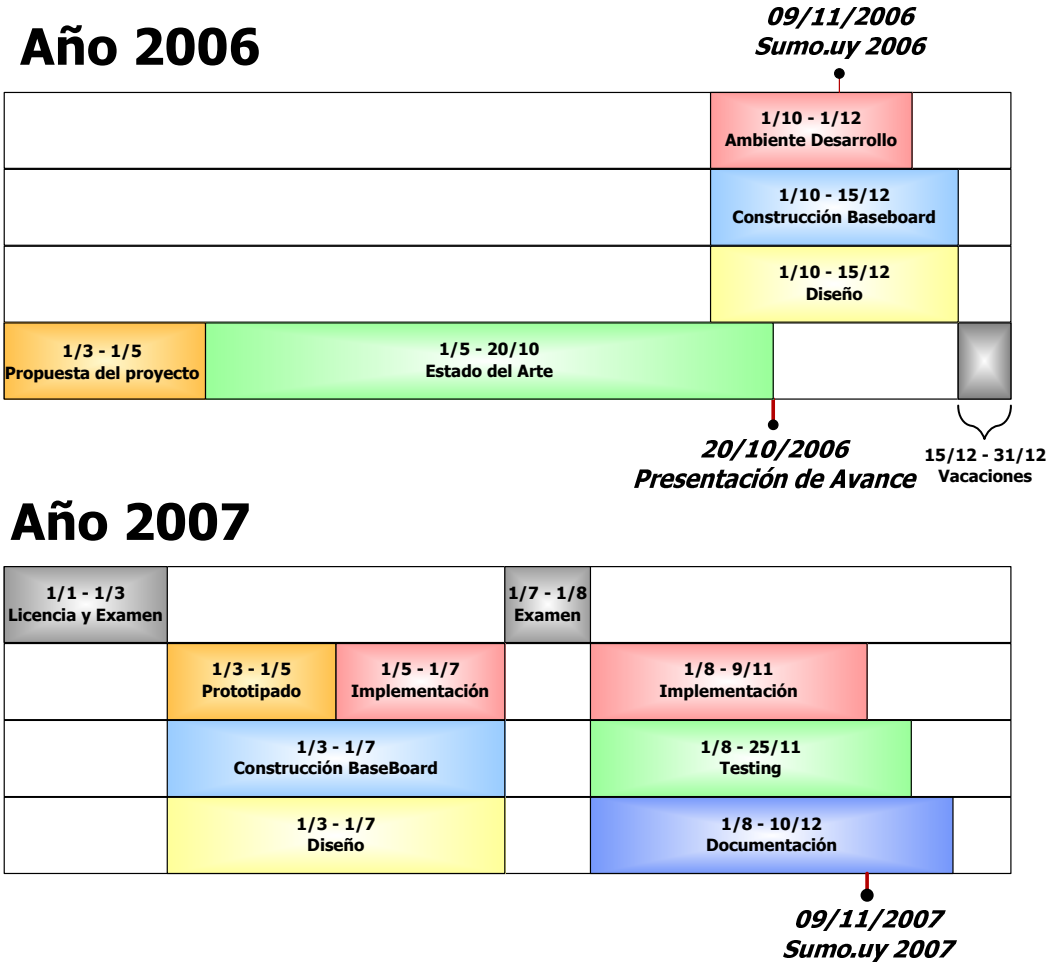


Figura 1.1: Línea de tiempo de las actividades del proyecto.

Capítulo 2

Resumen del estado del arte

Este capítulo tiene como objetivo introducir al lector en los conceptos más relevantes necesarios para comprender el vocabulario utilizado a lo largo del documento, para profundizar más en estos temas, se puede consultar el documento de estado del arte [2].

2.1. USB

En esta sección se brinda un resumen sobre los conceptos más importantes de la tecnología USB, los cuales sirven para poder comprender todos lo conceptos que se desarrollarán en los siguientes capítulos de este documento. Para profundizar más sobre estos conceptos se puede consultar la bibliografía relacionada [7, 5, 24, 15]

2.1.1. Características generales

El USB es un estándar [15] que define un bus utilizado para conectar periféricos a el PC. La principal característica que tiene es que la conexión es muy sencilla, ya que utiliza un único conector para conectar a través de un bus serie todos los dispositivos. En él se definen los conectores y los cables, una topología especial tipo estrella para conectar hasta 127 dispositivos y protocolos que permiten la detección y configuración automática de los dispositivos conectados. USB 1.0 soporta dos tasas de transferencia diferentes, una baja de 1,5 Mbps para la conexión de dispositivos lentos de bajo coste (joysticks, ratones) y otra alta de hasta 12 Mbps para la conexión de dispositivos que requieren un mayor ancho de banda (discos o CD-ROMs).

A mediados del año 2000 aparece la versión 2.0, que multiplica la velocidad del bus por un factor de 30 o 40, llegando a alcanzar una velocidad de 480 Mbps, con una diferencia de costo casi inapreciable. Es compatible con la versión anterior y utiliza los mismos cables y conectores, únicamente se necesitan nuevos concentradores (Hubs) que soporten la versión 2.0. Estos hubs son algo más complejos que los anteriores, ya que tienen que manejar el tráfico de datos de tres velocidades distintas sin ser excluyentes entre ellas. Cabe también destacar que USB 2.0 nunca llegará a reemplazar completamente a USB 1.0, ya que existen algunos tipos de dispositivos, como los dispositivos de interfaz humana (human interface devices) (HID) (teclados, ratones y joysticks)[17], que no requieren las altas velocidades que alcanza esta nueva versión y que únicamente encarecerían el dispositivo.

Anteriormente los periféricos se conectaban mapeados directamente en direcciones de entrada/salida (E/S), se les asignaba una dirección específica y en algunos casos un canal de acceso directo a memoria (direct memory access)(DMA). Esta situación conducía a tener conflictos en la asignación de estos recursos, puesto que siempre han estado bastante limitados en el ordenador. Además cada dispositivo tenía su propio puerto de conexión y utilizaba sus cables específicos, lo que daba lugar a un incremento de los costos. Debido a que a cada dispositivo se le tenían que asignar unos recursos específicos la detección del mismo debía hacerse a la hora de arrancar el sistema y nunca se podía incorporar un nuevo dispositivo cuando el sistema estaba en marcha. Los dos aspectos fundamentales que motivaron la realización de este estándar fueron la necesidad de configurar de forma sencilla los periféricos conectados al ordenador y la necesidad de aumentar el número de puertos disponibles.

Este estándar define una topología de conexión en estrella, tal como se muestra en la figura 2.1, por medio de la incorporación de varios hubs conectados en serie. Cada concentrador se conecta por un lado al ordenador, que contiene una o dos interfaces de este tipo en la placa base, o a otro hub y, por otro lado, se conecta a varios dispositivos o incluso a otro hub. De este modo pueden existir periféricos que vengan ya preparados con nuevos conectores USB para incorporar nuevos dispositivos, hasta un total de 127, todos ellos funcionando simultáneamente. Los hubs tienen la misión de ampliar el número de dispositivos que se pueden conectar al bus. Son concentradores cableados que permiten la conexión simultánea de múltiples dispositivos y lo más importante es que se pueden concatenar entre sí ampliando la cantidad de puertos disponibles para los periféricos. El hub detecta cuándo un periférico es conectado o desconectado a/de uno de sus puertos, notificándolo de inmediato al controlador de USB. También realiza funciones de acoplamiento de las velocidades de los dispositivos más lentos. Existe una gran variedad de dispositivos USB que se conectan todos al mismo bus. La característica más importante es que todos ellos utilizan el mismo tipo de cable y de conector y se conectan de la misma forma tan sencilla. El host decide qué dispositivo puede acceder al bus.

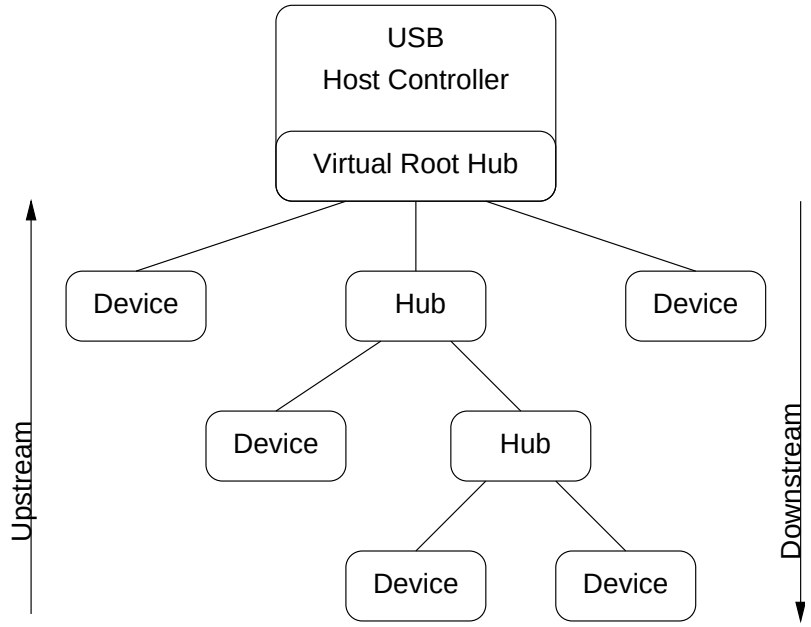


Figura 2.1: Topología tipo estrella

Desde el punto de vista lógico un dispositivo (comúnmente llamado función) posee un conjunto de endpoints, los que le permiten enviar y recibir datos. Un conjunto de endpoints forma parte de concepto de mayor abstracción llamado interfaz y es el que va a especificar cuales son las características de comunicación de un dispositivo.

2.1.2. Transferencias

USB está diseñado para manejar distintos tipos de periféricos con una gran variedad de requerimientos como lo son la frecuencia de transferencia, tiempo de respuesta y corrección de errores. El estándar define cuatro tipos de transferencias las que manejan diferentes necesidades. De esta manera los dispositivos pueden utilizar los tipos de transferencias que mejor se adecuen para sus propósitos.

Las transferencias de **control**, son las únicas que tienen funciones definidas por la especificación USB. Permiten al host leer información acerca del dispositivo, asignarle una dirección, seleccionar configuraciones y otras características. También pueden enviar pedidos específicos del vendedor. Todos los dispositivos USB deben soportar este tipo de transferencias.

Las transferencias **bulk** están pensadas para situaciones donde la latencia de la transferencias no es crítica, como enviar un archivo a una impresora, recibir datos de un scanner, o acceder

a un archivo en un disco. Para estas aplicaciones son llamativas las transferencias rápidas pero los datos pueden esperar si es necesario. Si el bus está muy ocupado las transferencias bulk son retardadas, pero si el bus está libre son muy rápidas. Sólo los dispositivos full y high speed pueden hacer transferencias bulk.

Las transferencias **interrupt** son para dispositivos que deben la atención del host o dispositivos periódicamente. Aparte de las transferencias de control, son la única forma de transferir datos para los dispositivos low-speed. Teclados y mouses utilizan este tipo de transferencias para enviar información.

Las transferencias **isochronous** tienen un tiempo de envío garantizado, pero no poseen control de errores. Son usadas para transmitir datos multimedia en aplicaciones de tiempo real. Es el único tipo de transferencia que no soporta retransmisión de datos recibidos con error. Sólo los dispositivos full y high speed pueden utilizarlas.

2.1.3. Descriptores

Los descriptores USB son estructuras de datos, o bloques de información con formato, que le permiten al host aprender acerca de un dispositivo. Cada descriptor contiene información acerca del dispositivo como un todo o un elemento del dispositivo como puede verse en la figura 2.2.

Todos los dispositivos USB deben responder a pedidos para los descriptores USB estándar. El dispositivo debe guardar información de los descriptores y responder a sus pedidos.

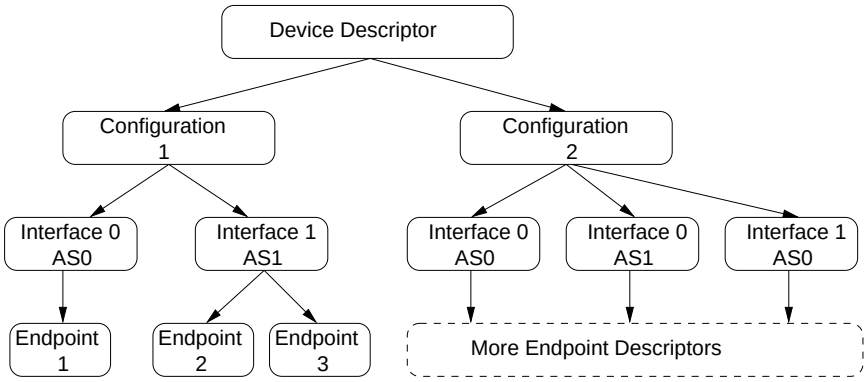


Figura 2.2: Jerarquía de Descriptores

2.1.4. Enumeración

Antes de que las aplicaciones puedan comunicarse con un dispositivo, el host necesita aprender acerca de los dispositivos y asignarle un driver. La enumeración es el intercambio de información que acompaña a estas tareas. El proceso incluye asignación de una dirección para el dispositivo, lectura de descriptores desde el dispositivo, asignación y carga de un driver, y selección de una configuración que especifique los requerimientos de consumo de energía, endpoint y otras características. El dispositivo luego está listo para transferir datos usando cualquiera de los endpoints asignados en su configuración.

2.1.5. Jerarquía de clases

Cuando un grupo de dispositivos o interfaces comparten muchos atributos o proveen o requieren de servicios similares, tienen sentido definir los atributos y servicios en una especificación de clase. Como es el caso de las clases HID que define dispositivos de interacción con una persona como lo son teclados, mouses, joysticks, etc o la clase de instrumentos musicales de interfaz digital (musical instrument digital interface) (MIDI) [16] que define instrumentos. Los sistemas operativos pueden proveer drivers para las clases en común, eliminando la necesidad de que los vendedores de dispositivos tengan que proveer los drivers para los dispositivos en esas clases.

Cuando un dispositivo en una clase soportada tiene una característica única o habilidades no incluidas en el driver, se puede proveer un driver de filtro para mantener las características agregadas y las habilidades, en lugar de escribir un driver completo. Una especificación de clase puede definir valores para los ítems en los descriptores estándar, como también descriptores class-specific.

2.1.6. Funcionamiento:

Los dispositivos tienen asociados canales lógicos unidireccionales, llamados pipes, que conectan al host controlador con una entidad lógica en el dispositivo llamada endpoint. Los datos son enviados en paquetes de largo variable. Típicamente estos paquetes son de 64, 128 o más bytes. Los endpoints y sus respectivos pipes, son numerados del 0 al 15 en cada dirección, por lo cual un dispositivo puede tener hasta 32 endpoints (16 de entrada y 16 de salida). La dirección se considera siempre desde el punto de vista del host controlador. Así un endpoint de salida será un canal que transmite datos desde el host controlador al dispositivo. Un endpoint solo puede tener una única dirección. El endpoint 0 (en ambas direcciones) está reservado para el control del bus.

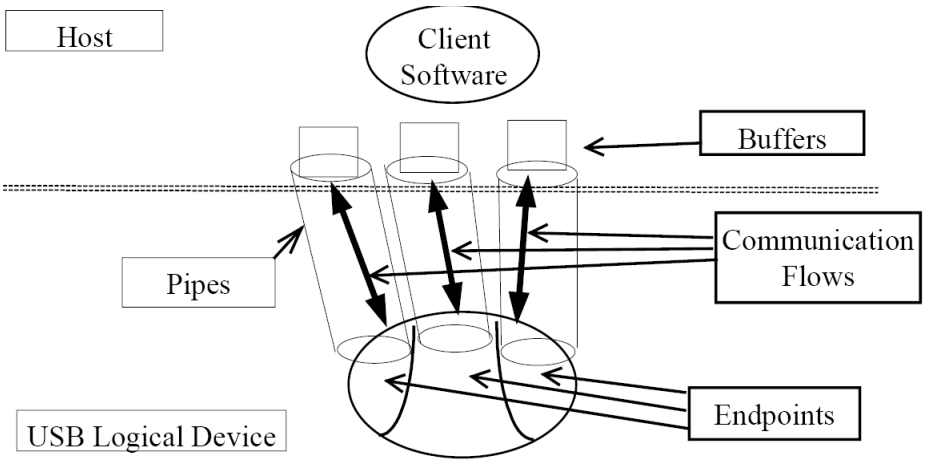


Figura 2.3: Comunicación virtual entre un dispositivo y el host

Cuando un dispositivo es conectado al bus USB, el host controlador le asigna una dirección única de siete bits (mediante el proceso de enumeración) que es utilizada luego en la comunicación para identificar el dispositivo. Luego, el host controlador consulta continuamente a los dispositivos para ver si tienen algo para enviar, de manera que ningún dispositivo puede enviar datos sin la solicitud previa explícita del host controlador. Para acceder a un endpoint se utiliza una configuración jerárquica de la siguiente manera: un dispositivo (llamado función) conectado al bus tiene un único descriptor de dispositivo, quien a su vez tiene uno (o varios) descriptores de configuración. Estos últimos guardan generalmente el estado del dispositivo (ej: activo, suspendida, ahorro de energía, etc). Cada descriptor de configuración tiene uno (o más) descriptores de interfaz. Y éstos últimos finalmente son los que contienen los endpoint, que a su vez pueden ser reutilizados entre varias interfaces (y distintas configuraciones) como muestra la figura 2.2.

2.2. Opciones de conectividad USB

Debido a que los objetivos de este proyecto incluyen el desarrollo de un hardware, se debieron relevar las soluciones de conectividad USB entre el hardware a construir y la PC.

Si bien hay muchos tipos de soluciones realizadas por un varios productores de semiconductores, luego de hacer un relevamiento se pudo realizar una categorización de ellas. Dentro de estas categorías, sólo se dejaron aquellas que no eran kits de desarrollo, y se muestran a continuación:

- Transceivers USB
- Conversores USB a serial o paralelo
- Controladores de periféricos
  - Externos
  - Embebido en un microcontrolador

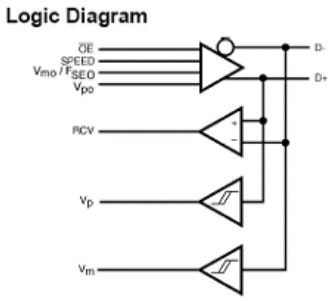
Se realizó una primera aproximación a todas las soluciones y luego se profundizó en aquellas que eran más apropiadas para los requerimientos de este proyecto, priorizándose aquellas en las cuales era posible conseguir muestras gratis.

A continuación se brinda un breve resumen de cada una las categorías relevadas, seleccionando para ello algunos representantes de cada una de ellas.

2.2.1. Transceivers USB

La principal responsabilidad de los transceptores (transceivers) USB es encapsular la capa física y realizar una interfaz con otros dispositivos programables. Esto sólo incluye una traducción de los voltajes que codifican la transmisión de información en dos señales D+ y D-, a un conjunto de señales para su posterior procesamiento de capas superiores realizadas por otros dispositivos. En las capas superiores se debe realizar un manejo de transacciones y endpoints, entre otros.

De esta forma estos son dispositivos muy simples, que a los sumo incorporan reguladores de voltaje, y detectores de conexión, lo que los hace muy baratos. Como representantes de esta categoría se seleccionaron el *USB1T20* de *Fairchild*, y el *Philips ISP110x*. En la figura 2.4 se muestra un diagrama lógico de las señales que traduce y la tabla de verdad de los valores lógicos que convierte.



Functional Truth Table

Input					I/O		Outputs			
Mode	V_{po}	V_{mo}/F_{SEO}	$\overline{OE}$	SUSPND	D+	D–	RCV	V_p	V_m	Result
0	0	0	0	0	0	1	0	0	1	Logic 0
0	0	1	0	0	0	0	U	0	0	$\overline{SE0}$
0	1	0	0	0	1	0	1	1	0	Logic 1
0	1	1	0	0	0	0	U	0	0	$\overline{SE0}$
1	0	0	0	0	0	0	U	0	0	$\overline{SE0}$
1	0	1	0	0	0	1	0	0	1	Logic 0
1	1	0	0	0	1	0	1	1	0	Logic 1
1	1	1	0	0	1	1	U	U	U	Illegal Code
X	X	X	1	0	Z	Z	U	U	U	D+/D– Hi-Z
X	X	X	1	1	Z	Z	U	U	U	D+/D– Hi-Z

X = Don't Care Z = 3-STATE U = Undefined State

Figura 2.4: Diagrama lógico y tabla de verdad del Transceiver USB

2.2.2. Conversores USB a interfaz serial o paralela

Otra de las posibilidades existentes para facilitar la conectividad entre el PC y un hardware vía USB es mediante el uso de conversores. La idea aquí, es que el conversor funciona como una caja negra, en donde uno de los extremos de la interfaz utilizada es USB y en el otro es serial o paralelo, según el conversor en cuestión. De esta forma, para el hardware que se desea conectar es transparente la forma en que los datos llegan al PC. Si se hace la analogía con conectividad en redes, podría pensarse que se crea un túnel entre el Host USB y el conversor por donde pasa la información de la interfaz del hardware externo. El conversor es visto en general por el PC como un puerto serial y así lo utilizan también las aplicaciones de usuario. El representante elegido en este caso es el *FT232BM* de *FTDI* y su diagrama en bloques se muestra en la figura 2.5. Los componentes más importantes de izquierda a derecha son: el transceiver USB, el motor de interfaz serial (serial interface engine)(SIE), el motor de protocolo USB y el transmisor-receptor asíncrono universal (universal asynchronous receiver-transmitter)(UART). Del transceiver ya se ha hablado con anterioridad, el SIE junto con el motor de protocolos USB tiene la responsabilidad de manejar transferencias en endpoints predeterminados por la interfaz

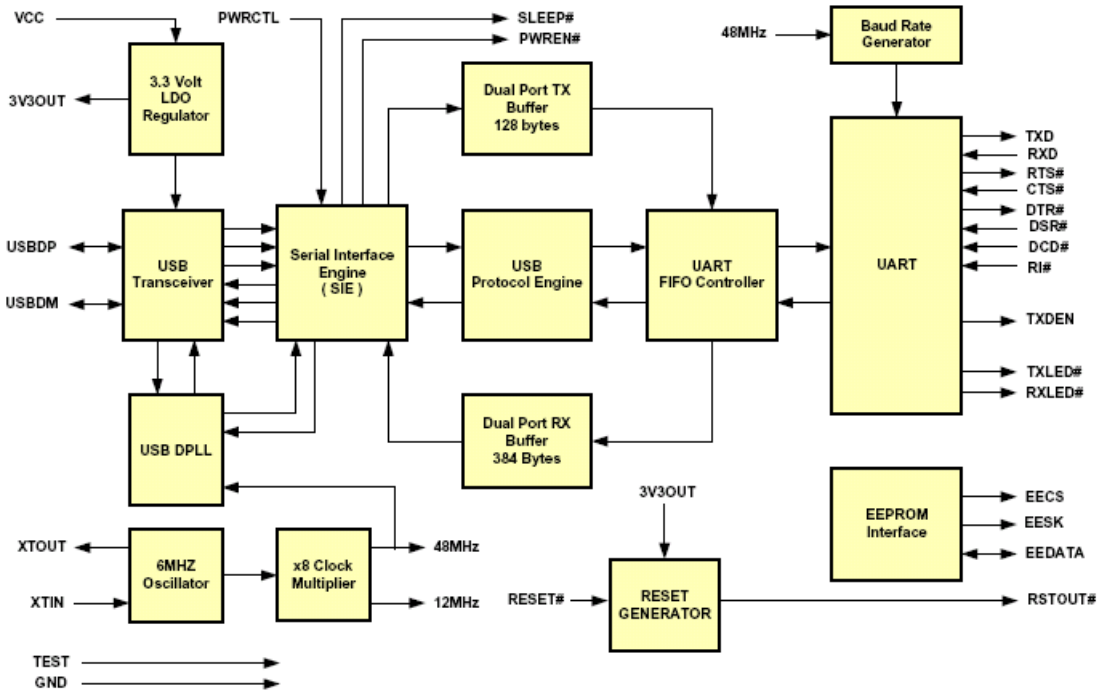


Figura 2.5: Diagrama en bloques del FT232

CDC. Luego el componente controlador UART FIFO, que con ayuda de los buffers de memoria compartida realizan las transferencias de datos con el Host USB. Este componente también sirve para setear la configuración del UART (velocidad, paridad, etc). Finalmente a la derecha de la figura se encuentra el UART que es donde se conecta el hardware vía una interfaz serial.

Otras características que tiene este conversor son:

- Cumple con el estándar USB 2.0 (Full speed)
- Velocidad de 300 a 3M bauds
- Manejo de handshaking y señales de modem.
- Bit Bang mode: Transforma las señales de control en puerto de E/S de 8 bits.
- Interfaz con EEPROM para caracterizar VID, PID, etc.
- Drivers de puerto COM virtual para Windows, MacOS y Linux.

2.2.3. Controladores de periféricos

Estos dispositivos incorporan un transceiver USB y la lógica para el manejo del protocolo. En estos es configurable además un número variable de endpoints y tipos de transferencias, así como también descriptores de dispositivo, interfaz, VID, PID, etc. Pueden encontrarse en dos modalidades: externos o embebidos en un microcontrolador. Teniendo en cuenta los requerimientos que se tenían para el proyecto de grado, resultaba más atractivo el uso de estas soluciones por sobre las estudiadas en las secciones 2.2.1 y 2.2.2. A continuación se describen algunas de sus principales características, pero un estudio más completo de todos los chips o microcontroladores que se comentan en esta sección se encuentra en el documento de Estado del Arte [2].

2.2.3.1. Controladores de periféricos externos

Estos dispositivos, como ya dijimos, manejan las comunicaciones USB al nivel de transacciones y endpoints y además es visto como otro periférico más para los microcontroladores o microprocesadores con los cuales interactúan. Vale la pena aclarar que estos dispositivos no son autónomos sino que deben interactuar con microcontroladores o microprocesadores para realizar parte de sus tareas, a diferencia de los conversores que se presentaron en la sección 2.2.2. Una de las ventajas más importantes que tiene, es el poco impacto que causa su aplicación en sistemas ya existentes, es decir, si se quiere agregar la funcionalidad a un dispositivo ya existente en un microcontrolador, sólo hay que agregar firmware necesario para el manejo del USB y no migrar todo a otra solución que utilice un controlador de periféricos embebido. En esta sección se tomó como representante el ISP1581 de Philips.

A continuación se presentan las principales características relacionadas con la conectividad USB:

- Cumplen con el estándar USB 2.0 y soporta velocidades high y full.
- 7 Endpoints de entrada, 7 de salida. Soporta double buffer.
- Soporta todos los tipos de transferencias
- 8Kb de memoria FIFO integrada.
- Interfaces con un microcontrolador:
  - Interfaz de bus independiente para la mayoría de los microcontroladores/microprocesadores (12.5 MByte/s)
  - Interfaz DMA de alta velocidad (12.8 Mbyes/s)
  - Interfaz directa con periféricos ATA/ATAPI
- Conexión al bus USB controlada por software (SoftConnect tm)
- Data transceiver y regulador de voltaje de 3.3 V integrados

2.2.3.2. Controladores de periféricos embebidos

En este tipo de soluciones, se incorpora dentro del mismo microcontrolador el hardware necesario para conectarse directamente al Host USB. Brinda las mismas funcionales que el controlador de periféricos externo pero con algunas diferencias. Normalmente es utilizado como un periférico más y utiliza registros dedicados y un tipo de memoria especial, a veces llamada RAM de doble puerto (Dual Port RAM) para intercambiar información con el microcontrolador, además de un poseer una rama completa de interrupciones asociadas a los eventos USB. La comunicación en los casos relevados se maneja a nivel de endpoints y el manejo de transferencias es manejado por firmware con el soporte de hardware de este periférico especial, comúnmente conocido como SIE. Una desventaja que genera el hecho de que se utiliza un recurso embebido en un microcontrolador, es que se genera una dependencia con la arquitectura éste.

Esta opción ameritó tener un relevamiento más completo que los anteriores, debido a que fue evaluado como la opción más adecuada para los requerimientos del proyecto y que una decisión en favor de una u otra opción podía tener consecuencias importantes. Los dispositivos relevados fueron:

- TSUB3210 de Texas Instruments
- PIC18F4550 de Microchip
- AT90USB1287 de Atmel

En este caso se presenta el cuadro 2.6 que muestra un comparativo con los aspectos considerados relevantes a la hora de realizar una selección del dispositivo a utilizar para el proyecto. Las características detalladas de cada uno de estos microcontroladores se encuentran en el documento de Estado del Arte [2].

	TUSB3210	PIC18F4550	AT90USB1287
Arquitectura	CISC (8052)	Harvard RISC 75+8 inst	Harvard RISC 135 inst
Velocidad	12 Mhz	48 Mhz	16 Mhz
Package	TQFP 64	TQFP 44, QFN 44, DIP 40	TQFP 64, QFN 64
Memoria de programa	*6K ROM, 8K RAM (Firmware)	32Kb Flash autoprogramable por software	128Kb Flash autoprogramable por software
Memoria datos	768 bytes	2 Kb	8 Kb (hasta 64 KB externos)
USB 2.0 (full y low speed)	512 Bytes compartida, 3 endp IN, 3 OUT. transferencias interrupt y bulk	1024 Bytes compartida, hasta 32 endp con ping pong buffering, soporta todas las transferencias	832 bytes compartida, 6 endpoints con ping pong buffering, soporta todas las transferencias
Eeprom	no	256 bytes	4 Kbytes
Modo Bajo Consumo	Si	NanoPower, 3 modos Sleep	Si, 6 Modos Sleep
Pines de E/S	Hasta 36	Hasta 35	Hasta 48
Timers	3 de 16 bits	1 de 8 bits 3 de 16 bits	2 de 8 bits 2 de 16 bits
I2C	Master	Master/Slave	TWI* Master/Slave
SPI	No	Master/Slave	Master/Slave
USART	No	Si	Si
Canales PWM	No	Hasta 2 de 10 bits de resolucion	Hasta 6 de 2-16 bits de resolucion
A/D	No	13 canales 10 bits	8 canales 10 bits
Otros	Bootloader I2C o USB, niveles de prioridad en interrupciones, soporte multiproducto	Soporte bootloader, prioridad de interrupciones programables, multiplicador por hardware, 2 comparadores analógicos, Streaming Paralel Port. ICSP e ICD Bloqueo de secciones de mem.	Soporte bootloader, vector de interrupciones con prioridad fija, multiplicacion por hardware, mparadores analógicos,modos bajo consumo, USB OTG,Bloqueo de secciones de mem. JTAG.
Documentación	Poca, algunas notas de aplicación.	Mucha, recursos en la web, muchas notas de aplicación, framework USB	Poca, Framework USB, algunas notas de aplicación.
Entornos de desarrollo y compiladores	En general los de 8052, de 3eras partes, algunos gratuitos.	MPLAB, 3ras partes, varios compiladores	AVR Studio 4, 3ras partes

Figura 2.6: Cuadro comparativo de los microcontroladores

2.2.4. Decisiones tomadas

Luego de evaluar las posibilidades con que se cuenta para brindar conectividad USB al hardware que se desea construir, se decidió que la opción más adecuada para este proyecto era utilizar microcontroladores con controladores de periféricos USB embebidos. Esta elección se realizó teniendo en cuenta que los conversores USB a serial o paralelo no permiten el grado de generalidad en la configuración requerido ya que utilizan una interfaz de dispositivo USB fija que no puede modificarse, ni tampoco los tipos de transferencias que maneja. Además, también debemos considerar que las demás opciones no eran totalmente autónomas, ya que de todas formas precisaban un microcontrolador para el manejo del protocolo USB. También vale la pena resaltar que ya desde una primera instancia era atractivo el uso de un microcontrolador para lograr cumplir con requerimientos de facilidad de configuración y manejo de dispositivos electrónicos diversos ya que iban a simplificar notoriamente el hardware.

Realizada esta elección lo que restaba era seleccionar alguna de las opciones relevadas. Lo primero que se notó fue que el microcontrolador TSUB3210 brindaba pocas prestaciones en cuanto a periféricos que soportaba de manera embebida, así como pocos recursos y baja velocidad de USB en comparación con los otros dos. Eso motivó que la elección quedara entre el PIC18F4550 y el AT90USB1287 y para ello se tuvieron en cuenta los siguientes criterios:

- Aspectos Técnicos: el AT90USB1287 en general en sus recursos disponibles de hardware es superior al PIC18F4550.
- Documentación: Hay una mayor documentación y notas de aplicación disponible del PIC18F4550.
- Infraestructura y conocimientos previos:
  - Experiencia previa (materia Taller de firmware dictada en FING)
  - Conocimiento de la arquitectura y herramientas de desarrollo por los integrantes del equipo.
  - Hardware de programación/debugging disponible.
  - Kit de desarrollo PICDEM FS USB disponible (para experimentación previa mientras no se realizara el hardware).
- Disponibilidad y facilidad para soldar
  - PIC18F4550 disponible en la plaza uruguaya.
  - PIC18F4550 disponible en package DIP40.

En función de los puntos que consideramos relevantes, si bien técnicamente el AT90USB1287 es superior en muchos aspectos al PIC18F4550, hubo otros aspectos que también se tuvieron en

cuenta para la elección final, como ser la documentación, los conocimientos previos y disponibilidad en los que el PIC18F4550 era superior a la otra opción. Finalmente, teniendo en cuenta estos aspectos se tomó la decisión de usar el PIC18F4550 para la implementación del hardware en este proyecto de grado.

2.3. Drivers

En el contexto de este proyecto de grado, uno de las áreas de conocimiento que se estudiaron fueron los distintos tipos de drivers de las plataformas Windows y Linux, así como las herramientas de construcción y depuración de drivers. No todos los elementos estudiados se usaron posteriormente en el desarrollo del proyecto, solamente fueron aprovechados los conocimientos sobre el modelado de drivers en Linux y algunas herramientas de desarrollo como son LibUSB y MPUSBAPI Library de Microchip. Para un mayor detalle de todo lo relevado y estudiado (modelos de drivers y herramientas de desarrollo y depuración de drivers), leer el documento Estado del Arte [2]. En las siguientes subsecciones se presentaran las herramientas utilizadas, junto con algunas otras herramientas de desarrollo y depuración de carácter comercial que son muy usadas en la actualidad.

2.3.1. Herramientas de desarrollo

Con la aparición del USB se han multiplicado la cantidad de periféricos que se pueden conectar a un PC, esto es debido entre otras cosas a la facilidad de uso y configuración. Esta simplicidad se basa en una complejidad mayor a la hora de desarrollar un driver de un dispositivo, ya que se necesita tener conocimientos de como funciona la tecnología USB, así como de los detalles internos del funcionamiento del sistema operativo. Durante el proceso de relevamiento del estado del arte se estudiaron las siguientes herramientas de desarrollo de driver USB: Windows DDK, Linux DDK, WinDriver *y* KernelDriver, USBIO, RapidDriver, JCommUSB, MPUSBAPI Library, LibUSB, LibUSB-Win32, jUSB y JSR80. Por más detalle ver el documento del Estado del Arte [2]. A continuación se presenta la herramienta más usada a nivel comercial y las utilizadas en este proyecto.

2.3.1.1. Jungo Ltd. - WinDriver USB 8.02

WinDriver es un juego de herramientas de desarrollo que simplifica la creación de drivers monolíticos de dispositivos. Incluye un ambiente gráfico de desarrollo, API's, utilitarios de diagnóstico y depuración y ejemplos que permiten un rápido desarrollo de drivers de alto rendimiento.

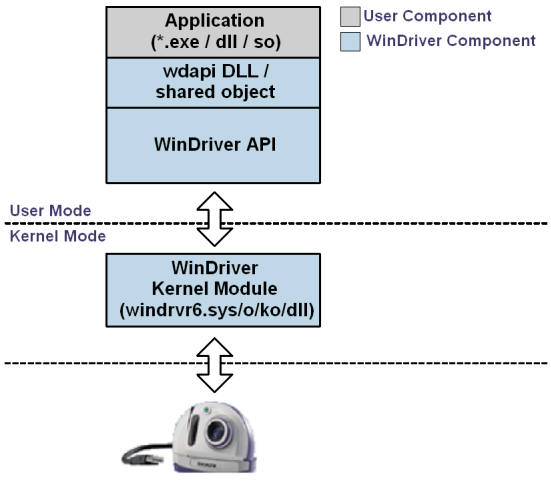


Figura 2.7: Arquitectura del WinDriver.

Características

- Soporta las especificaciones USB 1.x y 2.0.
- Se accede al hardware USB a través de una aplicación gráfica de usuario sin tener que escribir una sola línea de código.
- WinDriver posee un ayudante que genera un esqueleto del código del controlador personalizado para un dispositivo en particular.
- El esqueleto del código del controlador cumple con las características del WDM.
- Posee un ambiente gráfico para la depuración, que permite monitorear los eventos y datos relacionados al driver.
- Además de los dispositivos conectados directamente al PC, se pueden detectar dispositivos conectados por medio de un Hub USB.
- Se puede visualizar toda la información referente al dispositivo, como son los juegos de descriptores, número de serie, etc.
- Tipos de licencias y precios:
  - Node-Lock: `USD 3.000`
  - Floating: `USD 6.000`
  - 3-Pack: `USD 6.000`

Requerimientos

- Algunos de los sistemas operativos soportados: Windows 98, Windows ME, Windows NT, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Linux y Linux Embedded 2.4 - 2.6 (x86 32-bit y 64-bit; IA64; Power PC 32-bit).
- Compiladores: GCC, cualquier otro compilador ANSI C, Borland Delphi, Visual Basic 6.0, Visual C#.Net, Visual Basic.Net.
- Se necesita un espacio libre en disco duro de entre 26 y 34 MB dependiendo del sistema operativo.

2.3.1.2. Microchip Technology Inc. - MPUSBAPI Library 1.0

Es una API que provee acceso al puerto USB a aplicaciones que funcionan en sistemas operativos Windows. Forma parte del paquete gratuito *Microchip Full-Speed USB Solutions (MCHPFSUSB)*, que permite el desarrollo de aplicaciones de usuario y de firmwares para la comunicación con dispositivos USB, además como parte del paquete cuenta con un driver USB de propósito general (clase Custom USB). Sus principales características son:

- Soporta los estándares USB 1.x y 2.0.
- Soporta el uso de 32 endpoints.
- La API es una biblioteca de vinculación dinámica (DLL), para su fácil integración a los proyectos de desarrollo.
- Soporta los sistemas operativos: Windows 98SE, Windows ME, Windows 2000 y Windows XP.
- Es una herramienta gratuita.

2.3.1.3. LibUSB 0.1.12

Es un proyecto GNU-LGPL desarrollado para sistemas operativos del estilo Unix y tiene como meta la creación de una biblioteca que permita a aplicaciones de usuario poder comunicarse con dispositivos USB sin importar el sistema operativo. La versión estable de este proyecto es la 0.1.12 y fue diseñada para tener una fuerte analogía con las especificaciones USB, sin embargo su implementación es muy pobre debido a la existencia de mucho código rígido (hardcode) lo que trae como resultado que se pierdan algunas características del diseño. Las principales características de este producto son:

- Fue diseñado para soportar el estándar USB 1.x, aunque también puede soportar el 2.0.
- Los tipos de datos usan estructuras abstractas para mantener la portabilidad de representación de la información.
- Soportan todos los tipos de transferencia y todas las peticiones estándar de la especificación USB.
- Soporta los sistemas operativos: Linux, FreeBSD, NetBSD, OpenBSD, Darwin/MacOS X.
- Es una herramienta gratuita.

2.3.1.4. LibUSBWin32

Este proyecto es una migración del proyecto *LibUSB* a la plataforma Windows, esta biblioteca permite a las aplicaciones de usuario comunicarse con cualquier dispositivo USB en Windows de una forma genérica sin la necesidad de codificar ninguna línea de un controlador modo núcleo. Las principales características de este producto son:

- Puede ser usado como un controlador filtro o como un controlador base de un dispositivo simultaneamente.
- Las funcionalidades son 100 % compatibles con el proyecto LibUSB.
- Soporta transferencias del tipo control, bulk e interrupt y todas las peticiones estándar de la especificación USB.
- Soporta los sistemas operativos: Windows 98SE, Windows ME, Windows 2000 ,Windows XP.
- Es una herramienta gratuita.

2.3.2. Herramientas de depuración

Dentro del grupo de herramientas de depuración de drivers y dispositivos existen dos grandes clases: los analizadores software que permiten analizar los distintos eventos que suceden en el PC y los analizadores hardware que permiten analizar el trafico de información a nivel físico, pues se ubican entre el PC y el dispositivo a controlar. Durante el relevamiento del estado del arte se estudiaron las siguientes herramientas de depuración software y hardware: SourceUSB, USB Monitor, USBInfo, USB Monitor Pro, Advanced USB Port Monitor, Bus Hound, USB Tracker 110 *y* USB Explorer 200, Conquest USB *y* SBAE-30B. A pesar que se estudiaron, no fueron utilizadas en el desarrollo del proyecto, ya que no fue necesario llegar a tan bajo nivel en análisis en la comunicación. A continuación se presenta un ejemplar de herramienta de depuración software y hardware.

2.3.2.1. SourceQuest Inc.- SourceUSB 2.0

Es un analizador USB vía software que trabaja a nivel del host y que permite registrar las peticiones entrada/salida USB, eventos y invocaciones de funciones de bajo nivel entre los componentes de la pila de drivers USB. Su funcionamiento se basa en un drivers modo núcleo que se instala en el sistema operativo y coexiste con la pila de drivers USB de Windows y una aplicación de alto nivel para visualizar la información recolectada. Es un complemento ideal para un analizador vía hardware que registra las transacciones del canal pues brinda una visión desde la perspectiva del host.

Características

- Soporta los estándares USB 1.x y 2.0.
- No utiliza drivers filtro para capturar la información, esto le permite poder registrar todas las peticiones de entrada/salida que suceden durante la enumeración o remoción de un dispositivo.
- Captura de todas las IOCtrl internas de USB y IOCtrl modo usuario para los host controllers, hub y dispositivos HID.
- Se puede ver en detalle la composición (paquetes Setup, URB's, descriptores, etc.) y las distintas etapas (junto con sus estados) por las que pasan todas las peticiones USB de entrada/salida.
- Captura de todos los IRP de energía y Pnp y eventos remotos.
- Licencias y precios:
  - Single User: `USD 595`
  - 4-User: `USD 1.995`
  - 10-User: `USD 3.995`

Requerimientos

- Sistemas operativos soportados: Windows 2000, Windows XP, Windows Server 2003 y Windows Vista
- Hardware requerido:
  - Placa base Intel o compatible de 32-bit.
  - USB Host Controller (UHCI, OHCI, EHCI).

Type	#	Request	I/O	DO	Irp Request	Irp Status
USBINT	30	GET_DEVICE_HANDLE	--	'USBPDO-0'	0xffa3cc78	SUCCESS
USBINT	31	GET_DEVICE_HANDLE	--	'USBPDO-0'	0xffa3cc78	SUCCESS
URB	32	CLASS_OTHER - GetPortStatus	IN	'USBPDO-0'	0xffa3cc78	SUCCESS
URB	33	CLASS_OTHER - GetPortStatus	IN	'USBPDO-0'	0xffa3cc78	SUCCESS
URB	34	CLASS_OTHER - SetPortFeature	OUT	'USBPDO-0'	0xffa3cc78	SUCCESS
URB	35	CLASS_OTHER - SetPortFeature	OUT	'USBPDO-0'	0xffa3cc78	SUCCESS
URB	36	BULK_OR_INTERRUPT_TRANSFER	IN	'USBPDO-0'	0x81155e70	SUCCESS
URB	37	CLASS_OTHER - GetPortStatus	IN	'USBPDO-0'	0x81155e70	SUCCESS
URB	38	CLASS_OTHER - GetPortStatus	IN	'USBPDO-0'	0x81155e70	SUCCESS
URB	39	CLASS_OTHER - ClearPortFeature	OUT	'USBPDO-0'	0x81155e70	SUCCESS
URB	40	CLASS_OTHER - ClearPortFeature	OUT	'USBPDO-0'	0x81155e70	SUCCESS
URB	41	BULK_OR_INTERRUPT_TRANSFER	OUT	'USBPDO-0'	0x81155e70	SUCCESS
URB	42	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	43	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	44	CLASS_OTHER - GetPortStatus	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	45	CLASS_OTHER - GetPortStatus	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	46	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	47	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	48	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	49	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	50	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	51	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	52	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
URB	53	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa81b40	SUCCESS
PNP	54	START_DEVICE	--	'_HID00000000'	0xffa81b40	NOT_SUPPORTED
PNP	55	START_DEVICE	--	'USBPDO-4'	0xffa81b40	NOT_SUPPORTED
PNP	56	START_DEVICE	--	'USBPDO-4'	0xffa81b40	SUCCESS
URB	57	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-4'	0xffa201b8	SUCCESS
URB	58	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa201b8	SUCCESS
URB	59	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa201b8	SUCCESS
URB	60	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-4'	0xffa201b8	SUCCESS
URB	61	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-4'	0xffa201b8	SUCCESS
URB	62	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa201b8	SUCCESS
URB	63	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-0'	0xffa201b8	SUCCESS
URB	64	GET_DESCRIPTOR_FROM_DEVICE	IN	'USBPDO-4'	0xffa201b8	SUCCESS

Figura 2.8: Imagen de la información analizada.

2.3.2.2. ElliSys Sàrl - USB Explorer 200

El *USB Explorer 200* es un analizador de protocolo USB 2.0 de alto rendimiento, que ayuda a desarrollar de dispositivos USB de mejor calidad y en menos tiempo. Se encarga de monitorear los eventos USB y registrar el tráfico de información que intercambia por el cable USB, usualmente entre un PC y un dispositivo. Durante la captura del tráfico, se despliega en tiempo real en una estructura jerárquica y ordenada cronológicamente la información de las transacciones, paquetes, así como la decodificación de las peticiones estándar o de clases y los descriptores USB.

Características

- Análisis del tráfico a velocidades low, full y high.
- Análisis no intrusivo del tráfico y con una alta impedancia en la prueba.
- Alto grado de decodificación de las peticiones estándar y descriptores.
- Detección y medición de los estados del bus USB y protocolos de bajo nivel.
- Ediciones y Precio:
  - Standard: `USD 2.999`
  - Professional: `USD 5.999`

Figura 2.9: USB Explorer 200

Requerimientos

- Procesador Pentium III 600 MHz o superior.
- 512 MB de memoria RAM.
- Host controller USB 2.0.
- Windows 2000 y Windows XP.

2.3.3. Decisiones tomadas

Luego de relevado el estado del arte de las herramientas de desarrollo y depuración de drivers y junto con la decisión tomada de utilizar el PIC18F4550 como microcontrolador del hardware a desarrollar, se vió oportuno utilizar la *MPUSBAPI Library*. Principalmente se tomó esta decisión, buscando reutilizar el driver genérico USB que viene con la biblioteca de Microchip y centrar los esfuerzos de construcción de drivers modo kernel para la plataforma Linux.

Otra decisión tomada fue la elección de los productos LibUSB y LibUSBWin32 como opciones de driver modo usuario a utilizar en el proyecto, más allá de algunas limitaciones en la implementación (no soportan todos los tipos de transferencias USB) detectadas en el relevamiento, siguen siendo aceptables como opción.

Finalmente, se descartó el uso de herramientas de depuración de drivers vía software[1] y hardware, pues en el caso del primer tipo, las opciones relevadas sola funcionan en la plataforma Windows y no sirven para depurar un driver que ejecuta en Linux. Para el segundo tipo de herramientas de depuración el porque de su no utilización se debe estrictamente a sus costos económicos, los cuales escapan ampliamente del contexto del proyecto.

2.4. Proyectos relacionados

A continuación se presentarán algunos de los proyectos relacionados y sus principales características, del total de los que fueron relevados [1, 4, 6, 10, 12, 13, 14, 21, 30, 36]. Muchos de los proyectos relevados solamente se centralizaban en el hardware y daban un driver que no aprovechaba las transferencias USB, ya que por simplicidad reutilizaban drivers del sistema operativo asignados a clases especificas de dispositivos. El soporte de firmware que brindan es mínimo, dejando al usuario toda la tarea de desarrollo de este.

2.4.1. DevaSys - USB I2C / IO

Este proyecto [13] utiliza el microcontrolador CypressAN2131QC. Es un proyecto comercial y está fundamentalmente centrado en la comunicación con dispositivos del tipo I^2C.Algunas de sus características son:

- 20 bits I/O.
- Interfaz I^2C.
- Onboard 16KB I^2C EEPROM.
- Conector para hardware I^2C
- Bootloader
- Incluye API

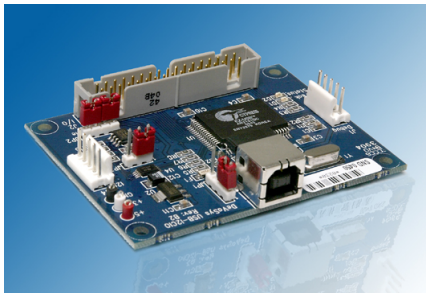


Figura 2.10: DevaSys

Puntos débiles:

- Usa transacciones de control para todas las comunicaciones, por tanto no utiliza transacciones Isochronous, Bulk, Interrupt.
- Utiliza USB low speed.
- No tiene módulos como PWM, serial, conversores A/D.
- Solo brinda driver para Windows.
- La API no ofrece una forma de comunicación genérica, solo está pensada para manejo de periféricos fijos (I^2C y E/S digital).
- No posee una arquitectura básica que permita agregar módulos programados por el usuario.

Puntos fuertes:

- Permite un diseño modular al poseer un conector para agregar los circuitos diseñados por los usuarios.
- Permite tener múltiples placas.
- Bastante memoria.

[1] Más allá de esta decisión, en los hechos se llegó a utilizar la herramienta de depuración SourceUSB, para detectar un problema en el driver genérico USB de la biblioteca de Microchip para el tipo de transferencia isochronous).

- La API está bien documentada.
- Permite la configuración de los pins de E/S.
- Se puede actualizar el firmware mediante el Bootloader.

2.4.2. Arduino

Es un proyecto [6] Software y Hardware libre, que ya tiene bastante tiempo el cual surgió como una placa que se conectaba por serial y luego se incorporó USB. Utiliza el microcontrolador Atmega8 de Atmel. Posee un lenguaje de programación propio llamado Wiring, el cual es un C reducido. Sus características principales son:

- 14 pins de E/S digital
- 6 pins de E/S analógica (A/D y PWM)
- Comunicación serial

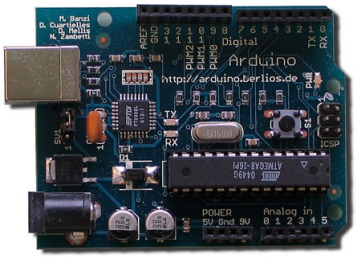


Figura 2.11: Arduino

Puntos débiles:

- Utiliza el conversor serial-USB FT232BL/TR [20] no aprovechando el ancho de banda, requerimientos de tiempo y arbitración de las transferencias USB. Utilizar el conversor tampoco permite reenumerarse como una clase de dispositivo conocido.
- No tiene una arquitectura básica donde el usuario pueda agregar dinámicamente módulos programados por él.

Puntos fuertes:

- Hay muchos ejemplos en la página
- Pueden configurarse los pins de E/S
- Incorpora Bootloader
- Posee PWM
- Hay drivers para Linux, Windows, MacOS (pero son los que trae el conversor serial - USB).
- Posee un modo de operación Stand Alone el cual le permite funcionar estando desconectado de la PC.
- Tiene un IDE propio.

2.4.3. Wiring

Las características principales de este proyecto [36] son:

- 43 pins de E/S digital
- 8 entradas analógicas
- 6 salidas PWM
- 2 puertos serial
- puertos I^2C.
- 8 pins para interrupciones externas
- 28 KB de memoria de programa flash

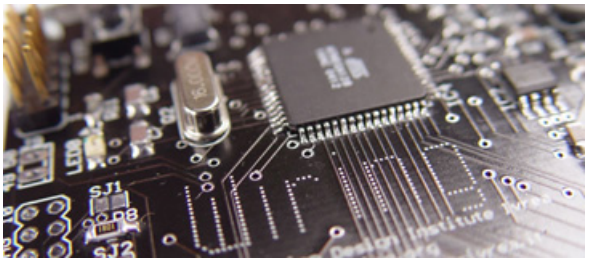


Figura 2.12: Wiring

Puntos débiles:

- Utiliza el conversor serial-USB FT232BL/TR
- Entonces no aprovecha el ancho de banda, requerimientos de tiempo y arbitración de las transferencias USB.
- Utilizar el conversor tampoco permite reenumerarse como una clase de dispositivo conocido.
- No tiene una arquitectura básica donde el usuario pueda agregar dinámicamente módulos programados por él.

Puntos Fuertes:

- Drivers para Linux Windows y MacOS (pero son los que proporciona el adaptador Serial - USB)
- Trae muchos pins para E/S
- Mucha memoria de programa
- Permite configurar los pins de E/S
- Posee gran cantidad de ejemplos en la página
- Hardware libre
- Pueden utilizarse los 8 pins analógicos como pins adicionales para E/S digital.
- Posee IDE propio

2.4.4. CUI

Una característica interesante de este proyecto [10] es que utiliza el microcontrolador PIC18F4550 de Microchip el cual fue seleccionado para realizar el *USB4all*

- 32 KB de memoria
- 5 puertos generales de E/S
- 13 entradas A/D

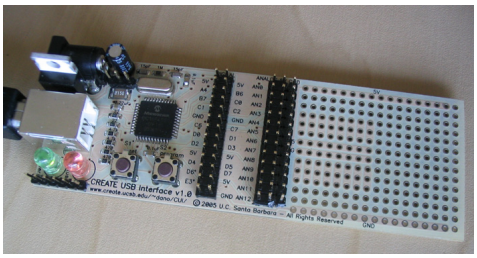


Figura 2.13: CUI

Puntos débiles:

- No proporciona software ni firmware, solo se brinda una placa con área de prototipado
- Tanto el driver como el bootloader son los que proporciona Microchip en su Framework ver Framework de Microchip.
- Pocos ejemplos y documentación inexistente.
- No da soporte para Linux
- Solo se puede usar enumerándose como clases conocidas por Linux.
- No tiene una arquitectura básica donde el usuario pueda agregar dinamicamente módulos programados por él.

Puntos fuertes:

- No utiliza conversores a USB
- Hardware libre
- Es actual (el proyecto está en funcionamiento aún y se usa como base de otros proyectos).

Parte II

Arquitectura

Capítulo 3

Introducción

La solución construida está compuesta por un PC, piezas de hardware y software especialmente diseñadas y los dispositivos con los que se quiere interactuar. En la figura 3.1 se muestra un diagrama de conexión física de los componentes, entre los que se destacan: el *Baseboard* y los *Adapterboards*.

El *Baseboard* es un dispositivo USB que funciona como controladora entrada/salida configurable y permite conectar dispositivos electrónicos que se comunican utilizando diversos protocolos e interfaces diferentes a la del USB. Para poder conectarse con el PC el *baseboard* cuenta con un conector USB y para la conexión con los dispositivos y/o *adapterboards* un puerto estándar de 40 pines llamado de aquí en más *U4APort* que agrupa un conjunto de señales configurables dinámicamente tales como señales analógicas, puertos digitales, interfaces seriales, etc.

El *Adapterboard* es una pieza de hardware que tiene como función principal encapsular toda la circuitería necesaria para la conexión de los dispositivos electrónicos al *baseboard*, para ello cuenta con un *U4APort* y los elementos necesarios para la conexión de los dispositivos específicos.

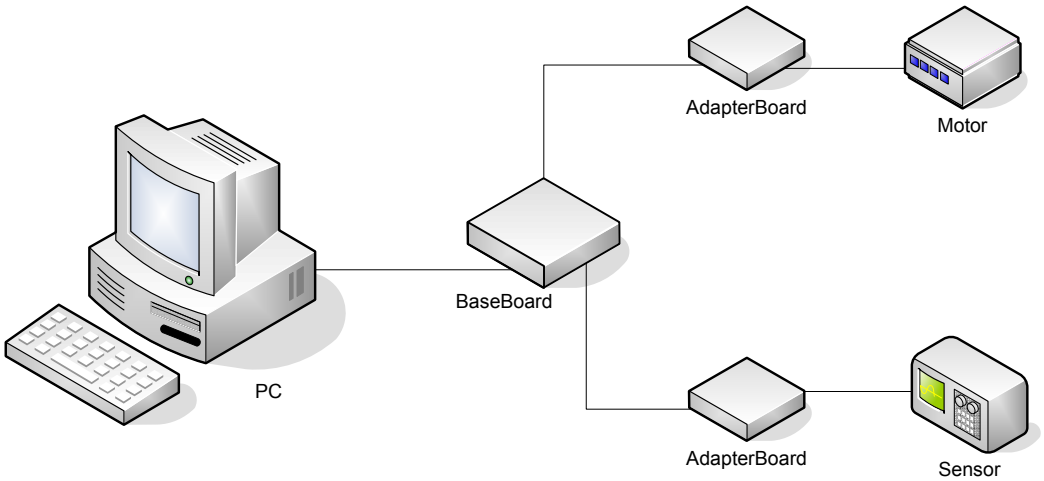


Figura 3.1: Componentes físicos del sistema USB4all.

La figura 3.1 muestra un PC con un *baseboard* conectado por medio de un cable USB, dos *adapterboards* conectados a un motor y a un sensor de temperatura y finalmente un cable plano multihilo (ribbon cable) que permite conectar los *adapterboards* al *baseboard*. La idea de fondo de la solución es enviar y recibir datos desde el PC por un único medio (USB) para que luego el *baseboard* transforme esa información mediante su procesamiento en señales abiertas que cumplen con los distintos protocolo e interfaces que requieren de los dispositivos. Vale destacar que toda la lógica de transformación de la información recae sobre el *baseboard* y no en los *adapterboards* que soló se encargan de cuestiones de conexión de los dispositivos (tratamiento de señales, circuiteria para el manejo de potencias, etc.). Para una justificación más en detalle de este concepto ver la sección 7.1.

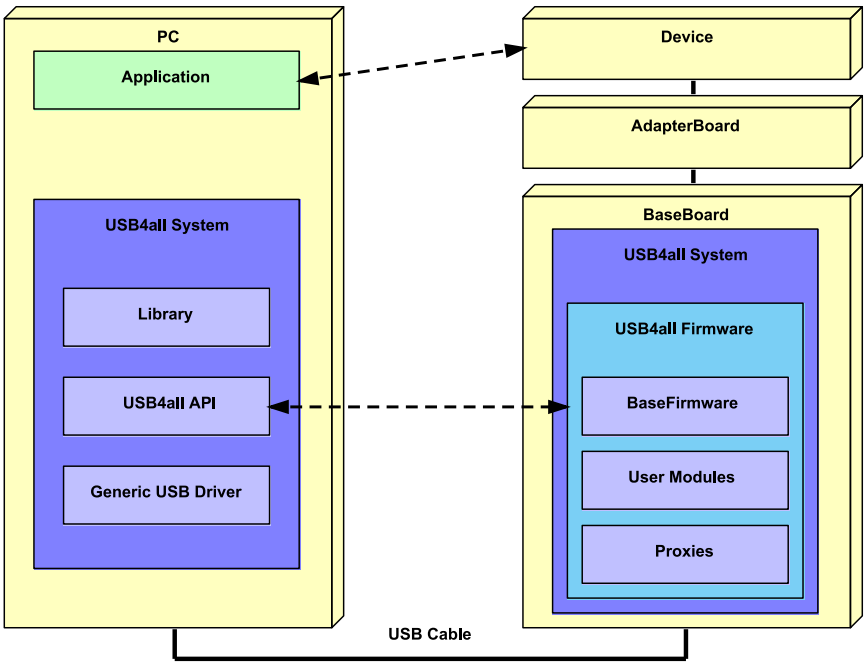


Figura 3.2: Componentes *USB4all System*.

Luego de haber presentado un panorama general de los elementos físicos de la solución, nos adentramos un poco más para presentar su arquitectura de software. La figura 3.2 es una vista interna de los elementos mencionados anteriormente, que permite visualizar los distintos componentes de software y firmware, que buscan satisfacer el objetivo principal de poder interactuar con dispositivos electrónicos desde un PC. A continuación se presentan brevemente los componentes que constituyen la arquitectura y en los capítulos siguientes se explicaran en detalle:

USB4all System Es el nombre del sistema construido y reparte sus funcionalidades entre el software del PC y el firmware del *baseboard*. Su función es brindar a las aplicaciones de usuario que corren en el PC servicios de alto nivel para la comunicación con los dispositivos electrónicos, similar a los provistos por los sistemas operativos para la gestión de archivos (abrir, cerrar, leer, etc.).

USB4all Firmware

Es la parte del sistema que reside en el *baseboard* y se encarga de transformar la información provenientes del PC (vía USB) a las señales eléctricas (vía *U4APORT*) necesarias para controlar los dispositivos deseados. Se divide en tres grandes componentes: *Base Firmware*, *Proxies* y *User Modules*.

Base Firmware:

Es el componente responsable de manejar el puerto USB del *baseboard*, implementar el protocolo de transporte y el de gestión de vínculos lógicos. Encapsula las funcionalidades básicas indispensables de la solución por medio de un conjunto de servicios que permiten la expansión del *USB4all firmware* mediante la incorporación de *user modules*. A su vez ofrece un entorno de ejecución concurrente que permite instanciar de forma dinámica varios *user modules*.

User Modules:

Son los componentes intercambiables del sistema que permiten encapsular la lógica de un dispositivo especifico y su protocolo de comunicación con las aplicaciones de usuario. Se acoplan al

base firmware de forma similar a plugins permitiendo expandir de esta manera las funcionalidades del *USB4all firmware.*

Proxies:

Es un representante en software de los módulos de hardware existentes en el microcontrolador y su principal función es encapsular y abstraer la complejidad del manejo del hardware de forma de brindar servicios de más alto nivel a los *user modules* para la comunicación con los dispositivos electrónicos, son la forma que tienen los *user modules* para acceder al hardware de manera compartida o exclusiva. Esta funcionalidad puede visualizarse como una capa de abstracción del hardware (hardware abstraction layer)(HAL), es decir, una capa que permite ocultar los detalles específicos de una plataforma, de forma de brindar servicios independientes del hardware sobre el que ésta se apoya.

USB4all API

Es el componente del sistema que reside en el PC y constituye la contraparte funcional del *base firmware.* Su misión es brindar a las aplicaciones de usuario una interfaz para la creación de vínculos lógicos con los *user modules* de forma de interactuar con los dispositivos electrónicos conectados a distintos *baseboards.* Este componente se desarrolló como una biblioteca de vinculación dinámica que soporta varias plataformas y drivers lo que permite ampliar el espectro de lenguajes de alto nivel que la utilizan así como ocultar la complejidad inherente de la tecnología USB.

USB4all Library

El componente *USB4all Library* se ubica por encima del *USB4all API* y brinda un manejo orientado a objetos sobre los servicios que brinda la *API.* Está formado por una jerarquía de clases base implementada en Java, que encapsulan los conceptos como *baseboards* y dispositivos. Esto permite extender fácilmente la solución para el manejo de nuevos dispositivos, pues sólo se debe implementar el protocolo de comunicación con el nuevo *user module* específico reutilizando todos los servicios de gestión ya implementados en las clases base.

Generic Driver:

Dado que el objetivo primordial es construir una solución genérica, no es factible la reutilización de drivers de clases USB definidas como HID, clase de dispositivos de comunicación (comunications device class) (CDC) [18], mass storage, etc., debido a que limita las clases de transferencias, la cantidad de endpoints a utilizar y la lógica de interacción con el dispositivo. Por lo tanto nuestra solución se basa en que los drivers que utiliza deben ser de la clase *Custom* definida por el estándar USB para poder utilizar todos los tipos de transferencias sin restricciones de la cantidad de endpoints y delegar la lógica de interacción con los dispositivos a las capas superiores de la arquitectura.

En los siguientes capítulos se estudiará en detalle cada uno de los componentes que conforman el *USB4all System* mostrado sus componentes internos, su funcionamiento y relación entre los componentes.

Capítulo 4

Arquitectura en el Baseboard

4.1. Baseboard Hardware

El *baseboard* es el hardware básico para permitir la conexión física entre el PC y los dispositivos electrónicos con los cuales se quiere interactuar. Su principal componente es el microcontrolador PIC18F4550 de Microchip encapsulado en el formato dos en línea (Dual In-line Package) (DIP), que se conecta al *baseboard* mediante un zócalo. El resto de los componentes proveen al microcontrolador de funciones básicas; ellos son: un oscilador principal y uno secundario opcional, componentes para el control de voltaje de alimentación, dos pulsadores y un diodo emisor de luz (Ligth Emitting Diode) (LED) para indicar que el *baseboard* está conectado al puerto USB. El *baseboard* cuenta con un conector USB tipo B para la conexión con el PC, un conector RJ11 para conectar un programador/debugger (por ejemplo MPLAB ICD2) y un conector de 40 pines llamado *U4APort* para conectar dispositivos electrónicos directamente con las entradas/salidas del microcontrolador, o módulos adaptadores (de aquí en más *adapterboards*) para cuando se precisa por ejemplo algún tipo de acondicionamiento de señal o manejo de dispositivos de mediana y alta potencia.

El *baseboard* constituye la piedra angular de una solución de hardware modular, pues contiene el hardware básico para la comunicación mediante USB con el PC y provee de un conector de interfaz sencilla para comunicarse con *adapterboards*, delegando en estos las funciones mínimas necesarias para la comunicación con dispositivos electrónicos específicos.

El *baseboard* posee varias cualidades, en donde se pueden destacar su pequeño tamaño, su robustez, y su simplicidad. Desde un principio se intentó minimizar su tamaño, ya que éste componente de hardware siempre está presente en todas las soluciones de conexión de dispositivos con el PC. Su tamaño de 70x60 mm (en su mayor parte ocupado por los conectores y el zócalo del microcontrolador) habilita la generación de dispositivos pequeños. La robustez está dada por el uso de conectores mecánicamente resistentes y soldados directamente al *baseboard* con longitudes mínimas de terminales de los componentes y la ausencia de cables o conexiones que se puedan doblar o quebrar con un uso normal. El circuito impreso es simple y utiliza sólo una faz de cobre, lo que facilita su fabricación y reduce los costos asociados.

Finalmente, la elección de componentes estándar y tecnología through-hole, que al contrario que la tecnología de montaje superficial (Surface Mount Technology) (SMT), permite cambiar fácilmente el microcontrolador en caso de que se dañe, además de facilitar el montaje del *baseboard*. Es de destacar que la mayoría de los elementos que conforman el *baseboard* con excepción del conector USB tipo B se encuentran en el mercado uruguayo a un costo razonable. Por más detalles en cuanto a su diagrama lógico y construcción leer el documento [3].

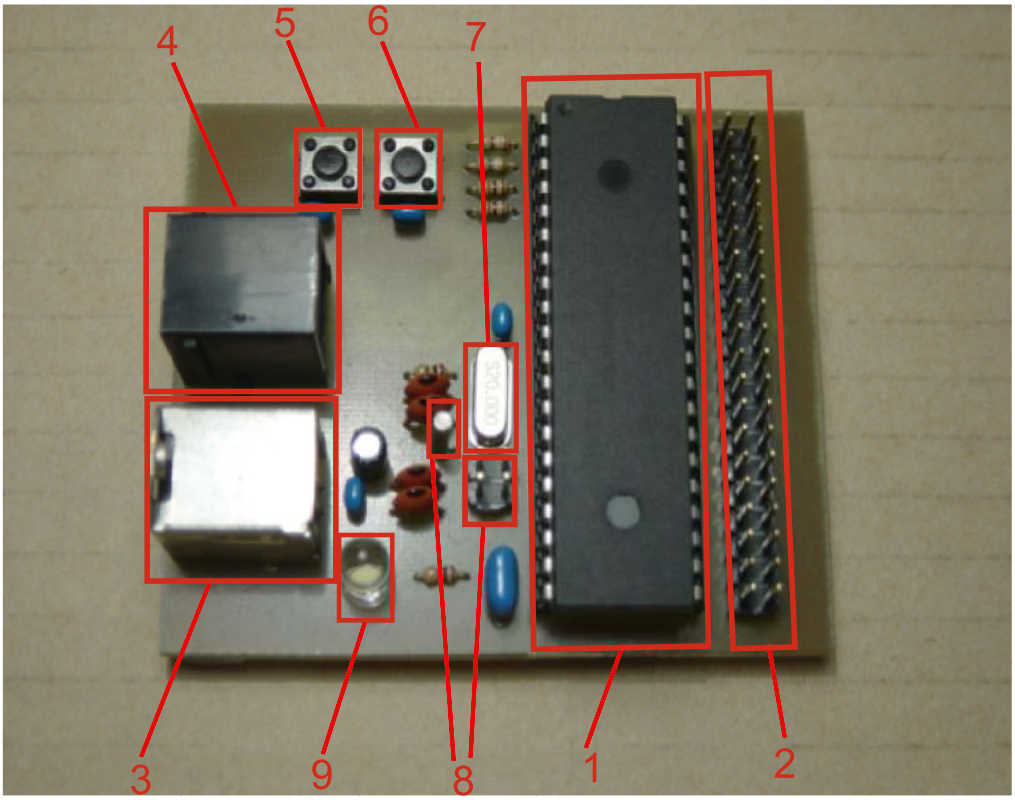


Figura 4.1: Identificación de los componentes del *baseboard*.

En la figura 4.1 muestra la distribución de los principales componentes del *baseboard*.

1. **Microcontrolador PIC18F4550**: Este microcontrolador es el responsable del manejo de toda la interacción entre el USB y los dispositivos finales o *adapterboards*. Por más detalles del microcontrolador ver el documento [3].

2. **U4APort**: Este conector se utiliza para la conexión con *adapterboards* que contienen o permiten interactuar con el hardware del dispositivo específico que se desea utilizar. Sus pines incluyen a casi todos los pines del microcontrolador, además de pines de alimentación de 5 voltios derivada del conector USB. Dentro de este conjunto de *adapterboards*, se encuentran los que simplemente unen los pines del conector del *baseboard* directamente con dispositivos de baja potencia (leds, otros chips, etc.) y los *adapterboard* que proveen cualquier tipo de acondicionamiento de señales, o controladores de mediana potencia para el funcionamiento del dispositivo (motores, relays, etc.).

3. **Conector USB tipo B**: Permite conectar el *baseboard* con el PC mediante un cable USB A-B estándar, igual al utilizado en impresoras, scanners, etc.

4. **Conector RJ11 para programador/debugger**: Permite conectar el *baseboard* a un programador/debugger, por ejemplo el MPLAB IDC2. Como programador se utiliza para la grabación del bootloader por única vez. También se utiliza si desea debuggear cualquier componente que integra el firmware del *baseboard*.

5. **Botón de Reset**: Pulsar este botón causa el reseteo del microcontrolador y por tanto el cierre de todas las comunicaciones de los programas de aplicación con el *baseboard*. Luego de ello y dependiendo del estado del botón de bootloader del *baseboard*, éste reinicia en modo normal donde interactúa mediante la *USB4all API* con aplicaciones en el PC, o en modo bootloader para una actualización del firmware.

6. **Botón de Bootloader**: Si se mantiene apretado el botón de bootloader y se pulsa el botón de reset, el *baseboard* inicia en modo bootloader. En este modo se puede actualizar

el firmware del *baseboard* mediante una aplicación dedicada a tal sentido en el PC vía USB. Si el botón de bootloader no es oprimido mientras se resetea, entonces el *baseboard* entra en modo normal.

7. **Cristal de 20 MHz**: Cristal utilizado para la generación de una señal de reloj principal para el microcontrolador.

8. **Cristal de reloj 32.768 kHz y jumper**: Al cerrar el jumper se establece el circuito que habilita el uso del oscilador de 32.768 kHz para su uso en aplicaciones relacionadas con reloj de tiempo real (Real Time Clock) (RTC).

9. **LED**: El led se enciende cuando el *baseboard* está conectado al PC mediante un cable USB.

Por más detalles de la constitución del baseboard, así como de su programación y actualización de firmware leer el documento[3]. Como se dijo anteriormente, el *baseboard* interactúa con uno (o varios) *adapterboard* que incluyen o permiten la conexión con los elementos electrónicos que se quiere utilizar desde el PC. Para ver en detalle algunos de los *adapterboards* construidos dirigirse a la sección 8.2.2.

4.2. Firmware

4.2.1. Introducción

El corazón del *baseboard* está constituido por el microcontrolador PIC18F4550 de Microchip y una de las claves para lograr las características deseadas en el marco de este proyecto de grado es un buen diseño de su firmware. La principal responsabilidad del firmware es la de permitir una interfaz configurable entre el USB y el(los) dispositivo(s) electrónicos conectado(s). Debido a que el objetivo final es facilitar la comunicación con dispositivos electrónicos diversos, el firmware debe tener la capacidad de proveer servicios genéricos de comunicación, así como permitir la conexión de un conjunto de dispositivos específicos. Las características de diseño y calidad que se pensaron para el firmware son las siguientes:

- **Modular**: Esto incluye identificar funcionalidades y responsabilidades específicas y encapsularlas en módulos. Algunos de estos módulos podrán ser intercambiables y para ello se van a precisar interfaces bien definidas.

- **Extensible**: Esta característica incluye el diseño de un núcleo genérico estable, al cual se le pueden ir realizando extensiones de manera sencilla y ordenada para el manejo de dispositivos o funcionalidades específicas.

- **Diseño basado en capas**: Para la comunicación con los dispositivos se utiliza una aproximación basada en capas en donde cada capa ofrece servicios a una capa superior y utiliza los servicios brindados por una capa inferior. Las capas tienen responsabilidades claras y utilizan una pila (stack) de protocolos para su implementación.

- **Configurable**: Se delega la configuración de la mayor cantidad posible de elementos al software que corre en el PC. La idea es mantener un firmware relativamente fijo y canalizar hacia el software del PC (que es más poderoso y más sencillo de realizar) las configuraciones específicas a utilizar en cada caso.

- **Facilidad de uso**: Esta característica significa eliminar (o minimizar) la necesidad de programación por los usuarios finales. Para el caso que se deba programar, el usuario debe concentrarse únicamente en la lógica para el manejo del dispositivo específico a utilizar. Para ello se dan las interfaces necesarias así como una metodología para la programación. El resto de las interacciones entre el USB y los controladores del PC queda oculto para el usuario.

- **Reusabilidad**: La clave es pensar en una arquitectura basada en componentes intercambiables. En función de realizar componentes o módulos para implementar extensiones al firmware de manera ordenada, y manteniendo cierta independencia entre estas, es posible el reuso de los componentes realizados por otros desarrolladores.

- **Independencia de plataforma**: El el diseño de la arquitectura se realiza abstrayéndose lo más posible del microcontrolador a utilizar. Esto facilita portar las implementaciones en otros microcontroladores.

Con esas metas en mente se logra la primera división del diseño del firmware del microcontrolador del *baseboard* (de aquí en más *USB4all firmware*) en componentes, tal como se observa en la figura 4.2.

A continuación se detalla cada uno de estos componentes:

- ***User Module***: Encapsula la lógica específica para el manejo de un determinado dispositivo o conjunto de dispositivos. Pueden utilizar servicios de *prox*ies, o si la interacción es muy simple, utilizar el hardware directamente (modo exclusivo), por ejemplo usar puertos de entrada/salida directamente con el dispositivo conectado. Estos módulos pueden verse como adaptadores específicos a un dispositivo y son los que en general deban ser implementados por los usuarios. Como ejemplos tenemos *user modules* encargados de la interacción con motores paso a paso, termómetros digitales, etc.

- ***Proxy***: Brinda servicios para facilitar el manejo del hardware del microcontrolador a los *user modules*. Permiten un uso sencillo y logran compartir recursos utilizados por varios *user modules*, por ejemplo permiten utilizar el bus I^2C por varios *user modules*. Éstos componentes están diseñados según el patrón de diseño Observer para poder notificar a los distintos *user modules* interesados el suceso de un evento.

- ***Base Firmware***: está es la porción del *USB4all firmware* responsable de toda la interacción con el PC mediante USB, la configuración dinámica del *baseboard* y de brindar la interfaz y los servicios necesarios a los *user modules*. Normalmente se van a instanciar varios *user modules* de manera simultánea, por lo que es responsabilidad del *base firmware* el manejo de un número variable de éstos. También tiene la responsabilidad de brindar interfaces específicas para facilitar el uso de interrupciones por parte de los *proxies*.

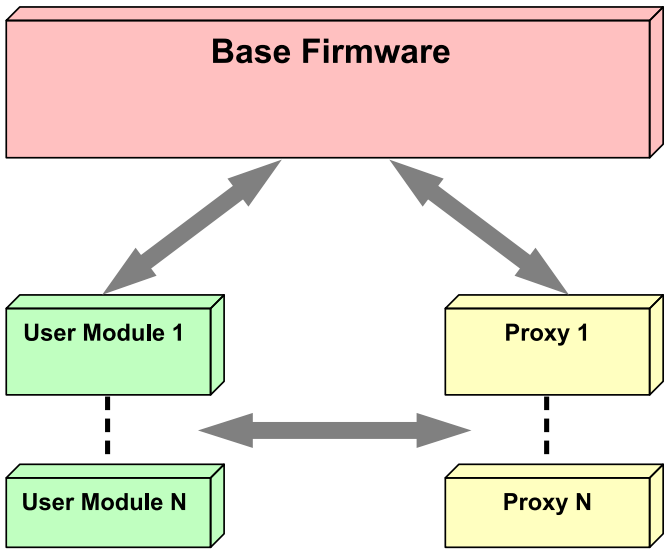


Figura 4.2: Componentes del *USB4all firmware*

En el cuadro 4.1 de la siguiente página, se detalla como el diseño del *USB4all firmware* en sus tres componentes básicos está vinculado con las características de diseño y calidad buscadas.

Característica	Descripción
Modularidad	Está presente en las tres componentes, si bien se puede aplicar directamente a los *user modules* y a los *proxies* debido a su relación uno a uno con un dispositivo o hardware especifico dentro del microcontrolador, la idea de modularidad también subyace en cada uno de los subcomponentes del *base firmware*.
Extensibilidad	Está vinculada en gran medida con los *user modules*, pues brinda una manera de que la solución puede extenderse para manejar una gran cantidad de dispositivos específicos.
Basado en capas	Está presente en los servicios de comunicación proporcionados por el *base firmware* a los *user modules*. Esto independiza el mecanismo de comunicación entre los *user modules* y su contraparte del lado del PC.
Configuración flexible	Está relacionada estrechamente con el componente *base firmware*, pues facilita la interacción con otros componentes en el software del PC para permitir configurar el *baseboard* estática y dinámicamente durante la ejecución de un programa de aplicación del PC.
Facilidad de uso	Está relacionada con la utilización de interfaces claras entre el *base firmware* y los *user modules*, en donde se delegan sólo algunas funciones específicas en los *user modules*, dejando el grueso de la comunicación encapsulado en el *base firmware*.
Reuso	Se basa fuertemente en que los *user modules* (y *proxies*) se relacionan con el *base firmware* mediante una interfaz que deben implementar. Existe una especie de contrato entre las responsabilidades que deben ser implementadas tanto en el *base firmware* como en los *user modules*. De esta manera si un desarrollador genera un *user module* para interactuar con un dispositivo, se vuelve reutilizable en diversas aplicaciones. A modo de ejemplo, luego de realizado un *user module* para manejar un motor paso a paso, éste puede utilizarse en un robot, en un plotter, etc.

Cuadro 4.1: Características del diseño del *USB4all firmware*.

En la figura 4.3 se muestra un ejemplo del *USB4all firmware,* para controlar un motor paso a paso y un sensor de temperatura. El *user module* encargado de interactuar con el motor se llama *stepper motor module* y para controlar el tiempo de encendido de cada bobina del motor, se utiliza el *timer proxy* y para controlar el sensor de temperatura, existe el *temperature sensor user module* que conoce el protocolo de comunicación serial del sensor particular.

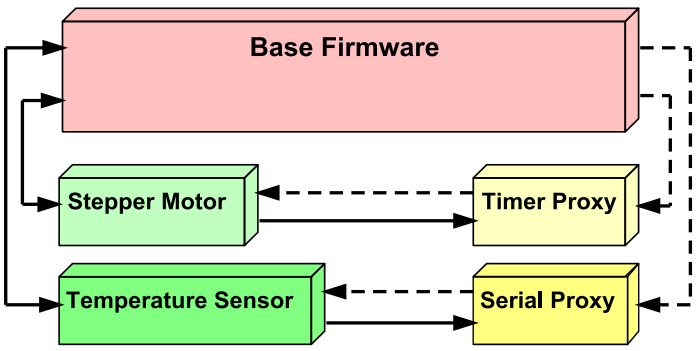


Figura 4.3: *Usb4all firmware* compuesto por *base firmware*, dos *proxies* y *user modules*

Es interesante notar que los *user modules* pueden acoplarse de manera muy sencilla con el *base firmware* y otros *proxies*. Esta forma de diseño, junto con algunas técnicas utilizadas, permiten la integración de varios *user modules* y *proxies* en el momento de generar el *USB4all firmware*. En la siguiente sección pasaremos a ver en detalle el *base firmware*.

4.2.2. Base Firmware

Como se explicó anteriormente, el diseño del *base firmware* persigue como objetivo encapsular la complejidad de la comunicación USB, el cual es alcanzado exportando interfaces claras, que permiten utilizar estos servicios por los usuarios. También se busca que la arquitectura sea extensible, brindándole al usuario mecanismos que le permitan extender el firmware con nuevas funcionalidades de manera muy sencilla, esto es resuelto agrupando en el *base firmware*, solamente lo que es fundamental para el funcionamiento elemental del *baseboard* y mediante la exportación de interfaces se dejan las funcionalidades especificas para que sean implementadas dentro de los *user modules*. Para lograrlo el *base firmware* provee un entorno en el cual pueden al mismo tiempo coexistir varios *user modules* y que el usuario logre tener la sensación de que todos ellos están ejecutando al mismo tiempo, sin la necesidad de modificar el *base firmware* cuando se desea agregar un nuevo módulo.

El *base firmware* está compuesto por dos grandes componentes, diseñados siguiendo un enfoque en capas como muestra la figura 4.4, con funcionalidades bien definidas en cada una de ellas. Cada capa está construida a partir de la inferior y tiene el propósito de ofrecer ciertos servicios a la capa superior. Estos servicios se logran implementar mediante el protocolo de comunicación que cada capa del *base firmware* mantiene con su contra parte en el PC, esto se explica en la sección 6.1. Se mantiene una comunicación virtual entre las capas equivalentes en el PC y el *USB4all firmware*, la cual es implementada mediante la utilización de las capas inferiores, para el intercambio de paquetes. Cada paquete está compuesto de un header con información para la capa en el otro extremo y la carga útil a intercambiar. A continuación se detallan las características de cada uno de los componentes del *base firmware*.

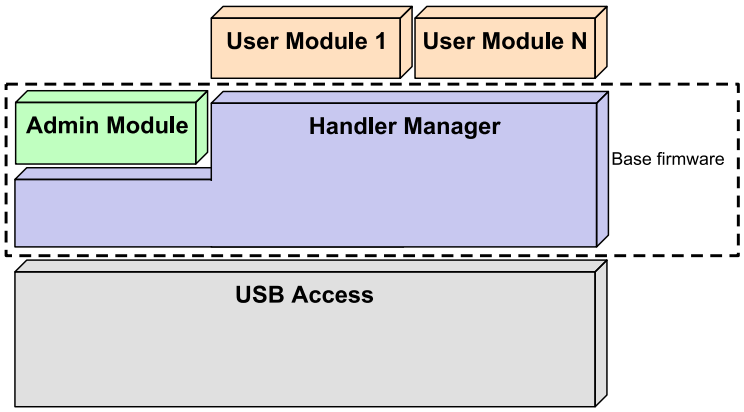


Figura 4.4: Diagrama de capas en firmware

4.2.2.1. Handler Manager

La capa del medio en la figura 4.4, a la cual denominamos *Handler Manager* es uno de los componentes fundamentales del *USB4all firmware*. Tiene como principal responsabilidad la gestión de los *handlers*, los cuales son identificadores que utiliza el componente *USB4all API* para diferenciar a qué *user module* enviar datos. La relación entre un *user module* y un *handler* es de uno a uno. A su vez un *handler* identifica de forma única a un número de endpoint, asignado al *user module* por el cual este va a comunicarse con el PC. Este módulo es responsable de almacenar una tabla encargada de mapear para cada *handler* que endpoint lo utiliza y a que *user module* pertenece.

Otra de las responsabilidades del *Handler Manager* es procesar los paquetes recibidos mediante la inspección de su header, determinar a que *user module* corresponden e invocar a la función de dicho módulo que se encarga de atender la llegada de nuevos datos, pasándole como parámetro la carga útil del paquete. Para poder lograr esta tarea se utiliza la tabla de mapeo *handler* - endpoint - *user module*.

Utiliza como servicios para realizar sus tareas, los expuestos por la capa de acceso al medio USB, para poder de esta manera recibir y enviar paquetes. La capa de acceso al medio USB está implementada en hardware por el microcontrolador y se accede a ella mediante la escritura y lectura de ciertas posiciones del espacio de memoria, que están mapeadas con los buffers de cada endpoint. Para simplificar esta tarea el *handler manager* expone buffers que se corresponden con los buffers de los endpoints, simplificando la lectura y escritura del endpoint.

Para que los *user modules* puedan comunicarse, deben registrarse en el *handler manager*. Esto se realiza mediante la operación `setHandlerReceiveFunction` en la que los *user modules* registran la función que va a ser encargada de manejar los datos recibidos para el *user module* que la registra. Luego el *handler manager* utiliza la referencia obtenida a la función registrada, como callback para ser invocada cada vez que se reciben datos por algún endpoint, dirigidos a un *user module* (identificado por su número de *handler*). La operación encargada de realizar dicha tarea es `USBRead`.

Operación	Descripción
`setHandlerReceiveFunction`	Recibe como argumentos un *handler* y un puntero a función, y registra a esa función como manejador de los datos que lleguen para ese handler.
`USBRead`	No recibe argumentos. Lee cada endpoint y si para uno de ellos hay nuevos datos se procesa el header extrayendo el *handler*, luego se invoca la función registrada como manejador de ese *handler*.
`USBWrite`	Recibe como argumentos un *handler* y un largo. Construye el header y lo coloca al comienzo de los datos ya ingresados a escribir en el mensaje a ser enviado.
`getSharedBuffer`	Recibe como argumento un *handler* y a partir de este obtiene el buffer asociado para escribir al endpoint correspondiente al *handler* pasado por argumento.

Cuadro 4.2: Interfaz del *Handler Manager*

La operación `USBWrite` es utilizada por los *user modules* para enviar datos hacia el PC. La última operación que exporta el *handler manager* es `getSharedBuffer`, la cual a partir de un *handler* pasado por parámetro devuelve el buffer correspondiente al endpoint que tiene asignado, esta operación es utilizada por los *user modules* para obtener buffers por donde pueden escribir datos por el USB.

4.2.2.2. Admin Module

Sobre la capa *Handler Manager* se encuentra la capa llamada *Admin Module*, esta capa se encarga de las tareas de gestión de los *user modules*, como ser apertura, cierre, listado de *user modules* almacenados en el *baseboard*, inicialización. Cuando se estudie el protocolo que mantiene esta capa, se verán en más detalle las tareas que lleva a cabo. Como vimos en la sección anterior el *handler manager* utiliza una tabla para mapear *handler* - endpoint - *user module,* la cual se utiliza para determinar a que *user module* va dirigido un paquete, el *admin module* le suministra la información necesaria al *handler manager* para que pueda poblar la tabla.

Como es común en las arquitecturas en capas, el protocolo de las capa superior viaja como payload en el paquete de la capa inferior, como veremos en forma más detallada en la sección 6.1, el paquete correspondiente al *admin module* viaja como payload del paquete correspondiente al Handler Manager.

El *Admin Module* al igual que los *user modules* debe registrarse en el *handler manager* especificando el *handler* en el cual atiende, ya que su estructura es similar a la de un *user module*, con la salvedad de que en lugar de registrarse al momento de ser abierto, el *admin module* se registra al momento de inicializarse el sistema. Se asigna el handler 0 para que atienda el *admin module*, este *handler* es reservado para el *admin module* y no puede ser utilizado por los *user modules*. La función registrada para atender los datos que llegan al *handler* 0 es la encargada

de implementar el protocolo correspondiente al *admin module*, el cual se explicará en la sección 6.1.

Al mismo nivel que el *admin module* se encuentran los *user modules*, cuyo protocolo no es fijo a diferencia del *admin module* y es compartido con su contraparte en el PC (la aplicación de usuario), en particular el *Admin Module* implementa una interfaz idéntica a los *user module* presentada en el cuadro 4.5, con la diferencia de que su protocolo es prefijado y dedicado a las tareas de gestión que se mencionaron. Sobre los *user modules*, no nos vamos a detener en esta sección, ya que no forman parte del *base firmware* y serán explicados en mayor detalle en la sección 4.2.3.

4.2.2.3. Main Loop

Algoritmo 1 Pseudocódigo de la operación `main`

```
main:
    initializeSystem()
    loop
        USBTasks()
        polling()
        USBRead()
    endloop
```

Debido a que utilizamos un microcontrolador con una aplicación dedicada sin ningún sistema operativo, se debe poseer de algún ciclo que permita que el microcontrolador continúe ejecutando siempre el *USB4all firmware*, esta es la tarea del `Mainloop` cuyo pseudocódigo puede verse en el algoritmo 1. Su interacción consta de inicializar el *baseboard* mediante la operación `initializeSystem`, luego le pasa el control al microcontrolador para que realice las tareas correspondientes al USB, luego ejecuta la funciones registradas por los módulos para hacer poll y por último lee los datos recibidos por los endpoints.

4.2.2.4. Dynamic Polling y Dynamic ISR

Los componentes *dynamic polling* y *dynamic ISR* implementan el patrón de diseño Observer para facilitar a lo *user modules* y/o *proxies* el manejo de la entrada/salida con los módulos de hardware del microcontrolador. Su responsabilidades son brindar un mecanismo por el cual los *user modules* y *proxies* registran y desregistran funciones que desean que se ejecuten repetidamente. Ambos persiguen los mismos objetivos, pero cada uno se especializa en mecanismos diferentes como son polling e interrupciones. Una funcionalidad que comparten es la de poder asignar tiempo de procesador a los *user modules* y *proxies* registrados en ellos, dándole la sensación al usuario de que todos ejecutan al mismo tiempo. Para ello el *dynamic polling* auspicia de despachador que utiliza la política round robin y el *dynamic ISR* implementa la rutina de atención de todas las interrupciones

El *dynamic polling* interactúa con los *user modules*, permitiéndoles registrar sus funciones, de forma de que en cada iteración del `Mainloop` sean invocados mediante la operación `polling` que es exportada por éste módulo. En el cuadro 4.3 pueden verse el conjunto de operaciones que definen la interfaz del *dynamic polling.*

Operación	Descripción
`addPollingFunction`	Recibe por parámetro una referencia a una función, la cual es registrada para ser ejecutada cada ves que se invoque a la operación `polling`.
`removePollingFunction`	Recibe por parámetro una referencia a una función, la cual es des registrada del mecanismo de polling.
`polling`	Al ser invocada, ejecuta todas las funciones registradas en el mecanismo de polling.

Cuadro 4.3: Interfaz del *Dynamic Polling*

Análogamente existe un mecanismo de registro similar, en el que en lugar de utilizarse polling se utiliza el método de interrupciones[1] y es implementado por el *dynamic ISR*. Éste se instala como manejador de interrupciones[2] y la operación `interruption` se encarga de notificar el suceso de una interrupción a los que se registraron. El orden de invocación se realiza según la política FIFO, esto permite implementar un mecanismo para el manejo de prioridades, dado que el primero que se registra es el primero en ser notificado. En el cuadro 4.4 puede verse el conjunto de operaciones que definen la interfaz del *dynamic ISR*.

Operación	Descripción
`addISRFunction`	Recibe por parámetro una referencia a una función, la cual es registrada para ser ejecutada cada ves que se invoque a la operación `interruption`.
`removeISRFunction`	Recibe por parámetro una referencia a una función, la cual es des registrada del mecanismo de interrupción.
`interruption`	Al ser invocada, ejecuta todas las funciones registradas en el mecanismo de interrupción.

Cuadro 4.4: Interfaz del *Dynamic ISR*

En caso de querer utilizar un recurso (módulo de hardware presente en el microcontrolador) por más de un *user module*, es necesario hacerlo utilizando el mecanismo de *proxies*. Esto se debe a que a los *user modules* desconocen la presencia de otros módulos registrados, por lo que se genera el problema de no saber cual de ellos debe blanquear la bandera de evento indicado. Uno de los usos de los *proxies* es asumir ésta responsabilidad de forma de centrar en un único componente toda la tarea, eliminando el problema previamente mencionado. Por más detalles sobre los *proxies*, ir a la sección 4.2.5.

Si por el contrario, se desea usar un recurso de hardware del microcontrolador en forma exclusiva por un *user module*, se puede utilizar directamente el mecanismo de *dynamic ISR* sin necesidad de los *proxies*.

4.2.2.5. Loader Module

También existe un módulo llamado *loader module* el cual brinda servicios al *admin module* que le permiten obtener referencias a las operaciones que los *user modules* implementan (pertenecientes a su interfaz) para poder ser anexados al *base firmware* y de esta manera el *admin module* puede acceder a las operaciones que los *user modules* necesitan exponer para que el *admin module* sea capaz de inicializarlos y cerrarlos. Los servicios implementados proporcionan mecanismos similares a los de *reflection*[27] en algunos lenguajes de programación. Este módulo se explicará en más detalle en la sección 4.2.3 en cuyo contexto será más evidente su motivación.

4.2.3. User Modules

Como mencionamos en la sección anterior, los *user modules* son los encargados de implementar la lógica particular encargada de resolver las funcionalidades que el usuario desea incorporar al *baseboard*, sin preocuparse de aspectos tales como la comunicación por el medio USB.

Cada *user module* se identifica por su nombre e implementa la interfaz del *user module* (ver cuadro 4.5), para que el *base firmware* sea capaz de anexar al *USB4all firmware* la lógica que implementa cada *user module*. Este conjunto de operaciones, es fundamental para que los *user modules* puedan colaborar con el *base firmware*, ésto forma parte de un contrato entre el *base firmware* y el *user module*, donde el *user module* tiene la obligación de implementar esas funciones y a cambio obtiene el beneficio de ser ejecutadas por el *base firmware* en los momentos estipulados.

[1] En el caso del microcontrolador PIC puede configurarse cada una de las banderas de hardware para que dispare una interrupción en el flujo de ejecución o no. El dynamic ISR se encarga de atender aquellas banderas que disparan interrupciones.

[2] Se entiende como manejador de interrupciones, a una rutina que además de la atención de la interrupción se encarga de salvar y recuperar el contexto (puntero de programa, registros de estado y trabajo).

Operación	Descripción	Momento en que se ejecuta
`init`	Inicializa el módulo de usuario y registra en el *handler manager* la operación encargada de manejar nuevos datos.	Al abrir el *user module*.
`Release`	Libera los recursos utilizados por el *user module*.	Al cerrar el *user module*.
`Received`	Operación encargada de manejar nuevos datos.	Cuando se envían datos al *handler* del *user module* que implementa dicha operación.
`ProcessIO`	Operación encargada de procesar entrada/salida mediante mecanismo de polling o interrupciones como fue descripto en la sección 4.2.2	Al ocurrir una interrupción o cada cierto tiempo, dependiendo del mecanismo de entrada/salida utilizado.

Cuadro 4.5: Interfaz de un *User Module*

Los *user modules* manejan los vínculos lógicos que le permiten comunicarse con las aplicaciones de usuario en PC mediante USB, dichos vínculos son una abstracción que brinda el *base firmware* de los endpoints que USB utiliza. Cada *user module* tiene un vínculo que utiliza un endpoint para enviar y otro recibir datos.

En la figura 4.5 podemos ver que el vínculo lógico del *user module 1* está constituido por el endpoint 1 IN que le permite enviar y el endpoint 2 OUT que le permite recibir. Los vínculos manejados por los *user modules* son consecuencia directa de la implementación del patrón de diseño Observer, donde los *user modules* juegan el rol de observadores de los datos que son recibidos por el *base firmware*, siendo el mecanismo de recepción del *user module* totalmente asincrónico para él. Por el contrario el mecanismo utilizado para enviar datos, es invocado directamente por el *user module*. Al permitirse tener un endpoint de entrada distinto al de salida, como se muestra en la figura 4.5 para el caso del *user module 1*, es posible tener un tipo de transferencia de recepción distinto al utilizado para el envió de datos.

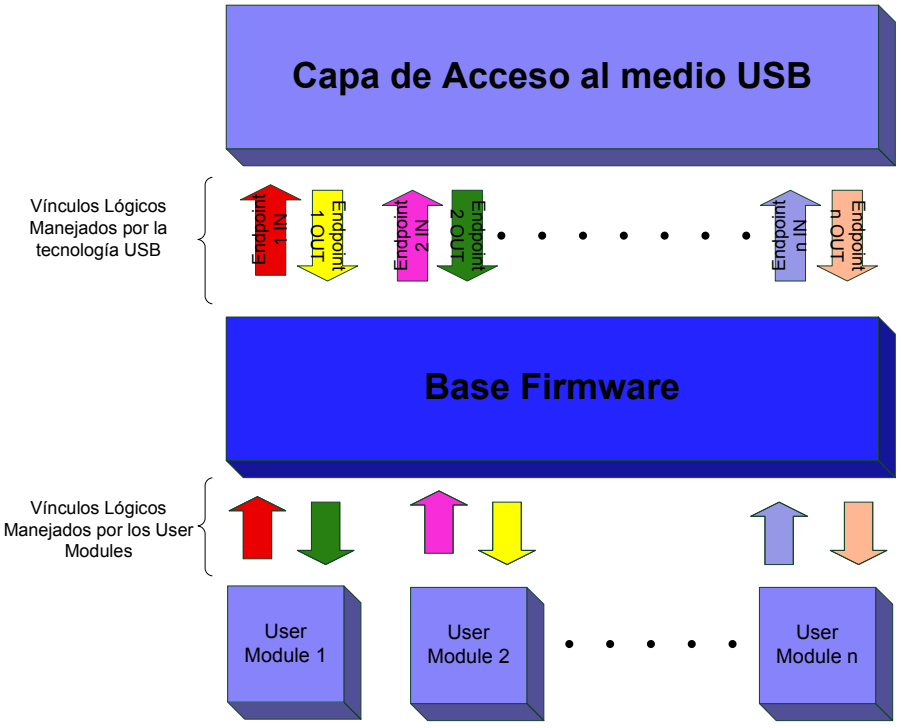


Figura 4.5: Relación entre los vínculos manejados por los *user modules* y los endpoints USB.

Para poder resolver lo que fue comentado en la sección 4.2.2 y evitar que el usuario deba realizar cambios al *base firmware* al agregar un *user module*, se busca eliminar las dependencias desde el *base firmware* hacia ellos. En particular las dependencias que se busca eliminar son los nombres de las operaciones utilizados en cada *user module*, ya que no es posible tener más de una operación con igual nombre en más de módulo[3]. Como solución a esta problemática se guardan y utilizan referencias a posiciones de memoria donde están definidas las operaciones en lugar de accederlas por medio de su nombre, más adelante en esta sección se explicará en detalle como se generan y almacenan en el *USB4all firmware* estas referencias.

Los *user modules* guardan referencias a las operaciones que el *base firmware* necesita en lugares fijos en memoria de programa (por más detalles ver sección 4.2.6), de esta manera el *admin module* puede encontrar las referencias en tiempo de ejecución y utilizándolas como callbacks puede acceder a las funcionalidades de los módulos sin depender de cada módulo particular. Las referencias a las operaciones `init` y `release` junto con el nombre del módulo deben almacenarse en una estructura especial en algún lugar determinado, para que puedan ser localizados en tiempo de ejecución y utilizados por el *admin module* como puede verse en la figura 4.6 donde se observa un ejemplo para el caso de un *user module* que implementa la lógica de control de un motor. Para llevar a cabo estas tareas existe un componente en el firmware llamado *loader module* que se encarga de manejar la estructura antes descrita y proporcionar operaciones para obtener las posiciones de cada operación a partir de su nombre.

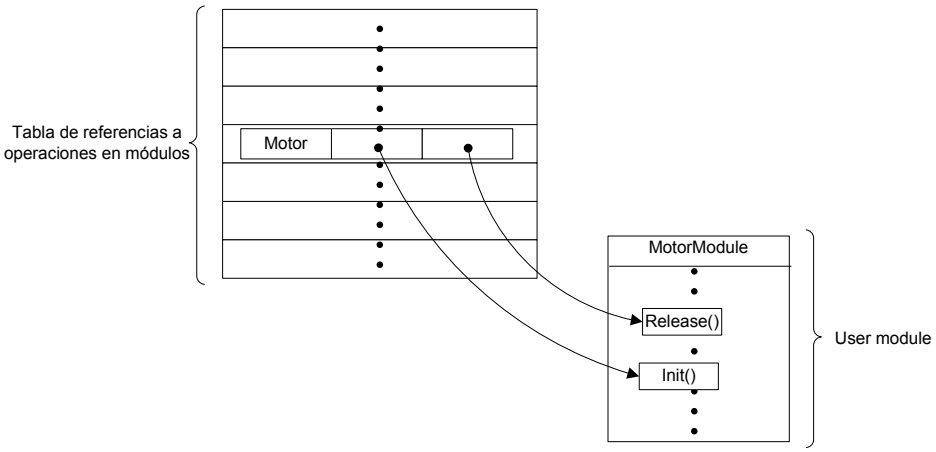


Figura 4.6: Estructura de un *user module*.

Es necesario contar con algún mecanismo que de manera sencilla permita almacenar la estructura que contiene las referencias a las operaciones exportadas por un módulo (que se necesitan como punto de inicio) y su nombre en un lugar predecible en memoria de programa. También es necesario que este mecanismo sea general para poder extenderlo a varios módulos, ya que al mismo tiempo puede haber varios cargados en memoria, para ello se utiliza un espacio de memoria continuo, donde de manera implícita se almacena una tabla, donde en cada fila de ésta se encuentra el nombre del *user module* y las referencias a las operaciones que expone en su interfaz y son necesarias como punto de inicio.

Se adecuó el linker script[4] utilizado, para poder implementar el espacio de memoria continuo mencionado, mediante una sección definida en el linker script, la cual nos permite almacenar en memoria de programa la estructura de mapeo. Las secciones especificadas en el linker script permiten definir un espacio y asignarle un nombre, luego se puede utilizar la directiva `pragma` [29] para almacenar variables o constantes en dicha sección, puede verse un pequeño ejemplo en el algoritmo 2. De esta manera, se soluciona el problema de encontrar las referencias a las operaciones que se precisan como punto de inicio del *user module* en lugares determinados.

[3] Esta restricción se debe a que si existierán más de un *user module* con operaciones de igual nombre, el linkeditor no podría resolver cual de las referencias debe utilizar.

[4] El linker script es utilizado por el linker, para describir como las secciones en el archivo origen son mapeadas en el archivo de salida.

Algoritmo 2 Mecanismo para almacenar las referencias en memoria de programa del módulo.

```
#pragma romdata user
uTab motorModuleTable = {&MotorInit,&MotorRelease,&userMotorConfigure,"motor"};
#pragma code
```

Donde `user` es el nombre de la sección y `uTab` es un estructura para almacenar las posiciones de memoria de las operaciones que exporta el módulo y su nombre.

La operación `init` como se ve en el algoritmo 3, se encarga de registrar en el *handler manager* la función `received` del *user module*, mediante la invocación de la operación `setHandlerReceiveFunction` con los parámetros: posición en memoria de la función y el *handler*, permitiendo de esta forma la recepción datos al *user module*. También se encarga de registrar la operación `ProcessIO` en el mecanismo de entrada/salida deseado (polling o interrupt), para que el *user module* realice entrada/salida con los módulos de hardware presentes en el microcontrolador. Para tal fín, el *user module* puede registrarse en el *dynamic polling* mediante la operación `addPollingFunction` (ver el cuadro 4.3) si desea utilizar el mecanismo de polling o en el *dynamic ISR* mediante la operación `addISRFunction` (ver el cuadro 4.4) si desea utilizar interrupciones.

Algoritmo 3 Pseudocódigo de la operación `init`

```
init(handler):
    guardo handler
    instalo manejador de recepción de datos en el handler manager
    instalo operación ProcessIO en mecanismo de polling o de interrupciones
    obtengo buffer para escribir en el endpoint asignado y guardo referencia a él
```

Otra responsabilidad de la operación `init` es guardar el *handler* asignado para el módulo, (el cual es pasado por parámetro) y obtener a partir de este, el buffer de memoria compartida correspondiente al endpoint asignado para escribir por USB, esto es resuelto mediante la operación `getSharedBuffer`. La operación `init` es invocada en el momento que desde el PC se pide que se cree una conexión lógica con un determinado *user module*. Determinar en que momento invocar las operaciones que el *user module* expone es una de las tareas del framework implementado por el *base firmware*, y el usuario no debe preocuparse de ello, sólo debe concentrarse en la lógica de las operaciones.

La operación `ProcessIO` será ejecutada automáticamente cada cierto tiempo (si se utiliza polling) o cuando ocurre una interrupción en el microcontrolador (si se utilizan interrupciones), además debe contener la lógica necesaria para atender los eventos generados por el mundo exterior y también puede utilizar los servicios que brinda el *handler manager* para comunicar a el PC la ocurrencia de dicho evento.

La operación `received` es la encargada de implementar el protocolo que mantiene el módulo con su contraparte en el PC y luego de ser registrada en el *handler manager*, es invocada cada vez que se reciban datos en el *baseboard* con destino el *handler* asociado al *user module* que la implementa.

La operación `release`, libera los recursos reservados por la operación `init` y es ejecutada por el *base firmware* al invocarse un pedido de cierre de la conexión lógica para dicho módulo desde el PC, puede verse su pseudocódigo en el algoritmo 4.

Algoritmo 4 Pseudocódigo de la operación `release`

```
release(handler):
    libera referencia al buffer de escritura del endpoint
    desinstalo manejador de recepción de datos
    desinstalo operación ProcessIO
```

Para enviar datos, un *user module* debe utilizar los servicios expuestos por el *handler m*anager mediante la operación `USBWrite`, previo a esto se debe escribir los datos a enviar en el buffer de memoria compartida en la posición correspondiente al *endpoint* del *user module*. Como el *handler manager* mantiene el mapeo entre los *user modules* y los endpoints, el *user module* no tiene porque preocuparse de cual es su endpoint, el *user module* sólo es responsable del *handler* y el *handler manager* se encarga de resolver a que endpoint corresponde.

4.2.4. Funcionamiento general

En la figura 4.7, se muestra un diagrama que resume el funcionamiento de lo explicado hasta el momento. El diseño del *USB4all firmware* está fuertemente influenciado por el patrón de diseño Observer para implementar el mecanismo de notificación de eventos. Con una línea punteada pueden verse las invocaciones correspondientes a la operación *notify* del mencionado patrón, las cuales son implementadas mediante callbacks.

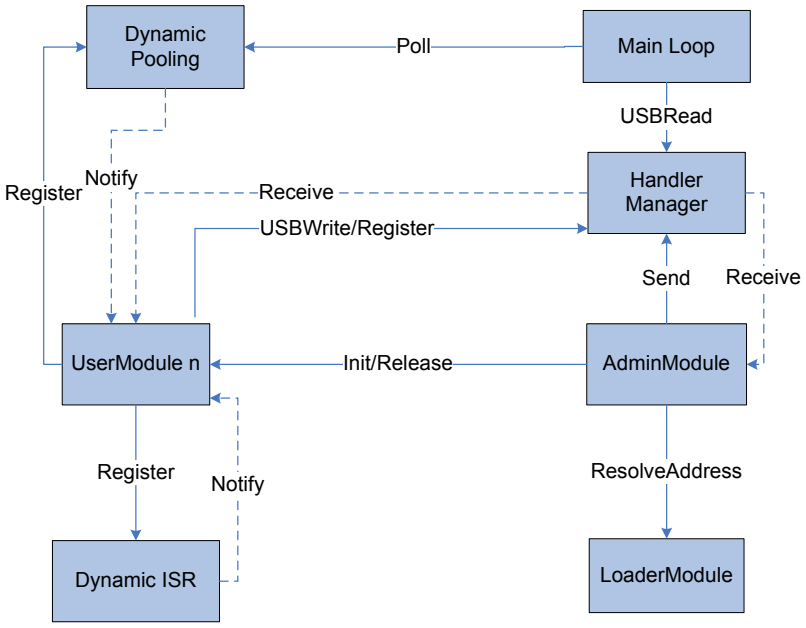


Figura 4.7: Componentes del *USB4all firmware*.

El funcionamiento del firmware comienza por el *main loop*, el cual se encarga de invocar a la operación `Poll` del componente *Dynamic Pooling* para que le asigne tiempo a cada *user module* o *proxies* registrado para utilizar el CPU (donde puede realizar E/S o lo que desee). Luego se ejecuta la operación `USBRead` del *Handler Manager* para que lea los datos que llegaron al *baseboard* y pueda notificar al *user module* destinatario de los mismos. Para esto, previamente el *user module* debe de haberse registrado. Si el destinatario de los datos es el *admin module*, usualmente va a ser necesario invocar operaciones de los *user modules*, como por ejemplo `init`, `release`. Para ello se utilizan los servicios del *loader module*, que resuelve las referencias a las operaciones del *user module*. Este mecanismo hace posible que el *base firmware* no este atado a los *user modules* cargados.

Normalmente los *user modules* o el *admin module* deben enviar datos hacia el PC, así como responder a los comandos recibidos. Para ello pueden escribir en el puerto USB utilizando la operación `USBWrite` que exporta el *Handler Manager*.

El otro mecanismo con que cuentan los *user modules* o *proxies* para responder a eventos, es mediante el uso de interrupciones. Para ello, pueden registrarse con el *Dynamic ISR* que los notificará al ocurrir alguna interrupción.

4.2.5. Proxies

Un *proxy* tiene la responsabilidad de brindar servicios para facilitar el manejo del hardware del microcontrolador a los *user modules*. Esto además de contribuir con el aislamiento de la lógica en los *user modules*, permite el uso de recursos de hardware compartidos. Vale la pena mencionar, que cuando se quiere compartir un recurso por más de un *user module*, no hay mucho que se pueda realizar para imponer restricciones en cuanto al manenejo del hardware. Esto trae aparejados problemas en la mutuo exclusión de los recursos, ya que en el microcontrolador utilizado no hay instrucciones privilegiadas a nivel de hardware que corran sólo en modo kernel.

De esta forma, aunque se pudiera hacer un sistema operativo, o algún componente que llevara el control de qué *user module* usa que recurso de hardware, no se puede impedir que ellos accedan directamente.

En los sistemas operativos actuales (de los PCs), hay componentes que se encargan del manejo centralizado de los recursos de hardware y son la única puerta de entrada para su acceso. De esta forma el sistema operativo es capaz de serializar el acceso a los recursos compartidos de manera adecuada. Para poder lograr tal control, se debe tener apoyo del lado del hardware y como se dijo no está disponible para el caso del microcontrolador utilizado.

Teniendo en cuenta este escenario, sólo se puede aspirar a dar algunos lineamientos y esperar a que el programador los siga. Ante este problema, lo que se suguiere es que haya un *proxy* que gestione los eventos que genera un determinado hardware (lo reconozca y apague las banderas adecuadas), ya que los *user modules* no saben de antemano el orden en que van a ser llamados luego de un `notify` en un *dynamic polling* o *dynamic ISR*. De esta manera todos los registrados al *proxy* son avisados del suceso del evento y no recae en ellos las responsabilidades de reconocimiento y apagado de las banderas.

Para la construcción de estos componentes, se utiliza el patrón de diseño Observer para notificar a todos los *user modules* registrados del suceso de un evento en el hardware. Dentro de esos servicios se encapsulan las funcionalidades para interactuar con módulos de hardware del microcontrolador en dos modalidades: mediante interrupciones y mediante polling, dando lugar a los *interrupt proxies* y *polling proxies* respectivamente. A continuación se muestra en detalle cada uno de ellos.

4.2.5.1. *Interrupt proxies*

Los *interrupt proxies* son particularmente útiles cuando deben notificarse eventos disparados mediante interrupciones, como suele suceder con módulos de timer, puertos seriales, etc. De esta manera puede implementarse un *proxy* para un Timer del microcontrolador, y permitir a varios *user modules* sean notificados del evento overflow del registro contador del Timer. Para la interacción con el resto del firmware, el *interrupt proxy* debe implementar la interfaz mostrada en el cuadro 4.6.

Operación	**Descripción**
`initFunctions`	Inicializa las funciones de callback.
`addFunction`	Recibe como parámetro una referencia a función. Agrega una función a la colección de callbacks a ser llamada al producirse una interrupción.
`removeFunction`	Recibe como parámetro una referencia a función. Remueve esa función de la colección de callbacks.
`config`	Recibe parámetros necesarios para configurar el *proxy*. Configura el hardware del microcontrolador según los parámetros recibidos.
`interrupt`	No recibe parámetros. Esta rutina es invocada desde el *DynamicISR* al producirse una interrupción.

Cuadro 4.6: Interfaz de los *Interrupt Proxies*.

La operación `initFunctions` es la encargada de inicializar la colección de callbacks y es llamada por única vez desde el *base firmware* (o luego de un reset).

El uso típico de los *proxies* por parte de los *user modules* es la aplicación del patrón de diseño Observer de la siguiente forma:

- La operación `config` para configuración del *proxy* es llevada a cabo por uno de los *user modules* con los parámetros adecuados.
- Al inicializarse, el *user module* registra una función de callback mediante la operación `addFunction` que debe ser ejecutada al ocurrir la interrupción procesada por el *proxy*.
- Al realizarse un release del *user module*, éste utiliza la operación `removeFunction`, ya que no está más interesado en ser notificado de la ocurrencia de una interrupción.

El *proxy* al ejecutar su operación `addFunction` verifica si es la primer función en registrarse, si es así, entonces el *proxy* registra la función de callback `interrupt`, en el *DynamicISR*. Luego, agrega la función pasada como parámetro en su colección de funciones de callbacks. Un procedimiento análogo sucede al invocarse la operación `removeFunction`, si es la última operación en removerse, entonces el *proxy* se desregistra del *dynamicISR*.

La operación `interrupt` es el análogo al notify del patrón Observer y realiza las acciones que se muestran en el algoritmo 5.

Algoritmo 5 Pseudocódigo del la operación `interrupt`

```
interrupt():
    Si es la interrupción esperada:
        Para cada función de callback F registrada en la colección
            Invoca F
        Apaga las banderas para la interrupción procesada.
    Retorna
```

Otras operaciones pueden agregarse en caso de ser necesarias dadas las responsabilidades del *proxy*, por ejemplo, si se tratara de un *proxy* que envía datos por un puerto serial, se le agrega una operación de `send` que pueden ejecutar los *user modules* para enviar datos por el módulo de hardware del microcontrolador adecuado.

4.2.5.2. Polling proxies

Los *polling proxies* son útiles cuando se quiere esperar por un evento realizando polling sobre algún módulo de hardware. Deben implementar una interfaz similar a la de los *interrupt proxies*, pero la gran diferencia es que los *polling proxies* son llamados mediante la operación `polling` desde *dynamic polling* , en vez de ser llamada la operación `interrupt` desde *dynamicISR*. De esta forma periódicamente es ejecutada la operación *polling*, la que a su vez invoca a todos los *user modules* registrados para ese evento. El resto de las interacciones son análogas a las descriptas en *interrupt proxy*. Para la interacción con el resto del *base firmware*, el *polling proxy* debe implementar la interfaz mostrada en el cuadro 4.7.

Operación	Descripción
`initFunctions`	Inicializa las funciones de callback.
`addFunction`	Recibe como parámetro una referencia a función. Agrega una función a la colección de callbacks a ser llamada al producirse una interrupción.
`removeFunction`	Recibe como parámetro una referencia a función. Remueve esa función de la colección de callbacks.
`config`	Recibe parámetros necesarios para configurar el *proxy*. Configura el hardware del microcontrolador según los parámetros recibidos.
`polling`	No recibe parámetros. Esta rutina es invocada desde el *DynamicPolling* periódicamente.

Cuadro 4.7: Interfaz de los *Polling Proxies*.

4.2.6. Programación y configuración

Luego de haber definido los elementos que componen el *USB4all firmware*, se hace necesario presentar algunos detalles referentes a su implementación para explicar como se realiza la grabación del firmware dentro microcontrolador del *baseboard* y con que mecanismos se cuenta para configurar el *baseboard* como dispositivo USB.

Lo primero a considerar, es que se utiliza un bootloader para la grabación del *USB4all firmware* dentro del microcontrolador. El bootloader es una pequeña aplicación ubicada al principio de la memoria de programa, cuya finalidad es la de grabar un firmware dentro del microcontrolador que se transfiere desde el PC mediante USB. Este bootloader es grabado por única vez en el

microcontrolador utilizando un programador (por ejemplo MPLAB IDC2) mediante el conector RJ11 presente en el *baseboard*, esto se muestra en la figura 4.8.

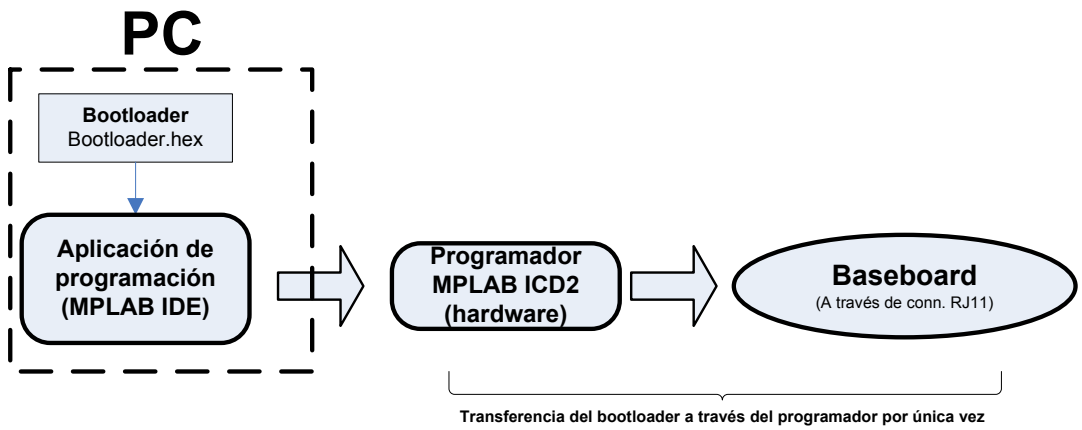


Figura 4.8: Programación del bootloader en el microcontrolador del *baseboard*

El *baseboard* de aquí en más podrá ser iniciada en dos modos, el modo bootloader y el modo normal. Para iniciarla en modo bootloader, se deberá mantener apretado el pulsador bootloader luego de soltar el pulsador de reset. De otra forma el *baseboard*, iniciará en modo Normal, adquiriendo el control *USB4all firmware*.

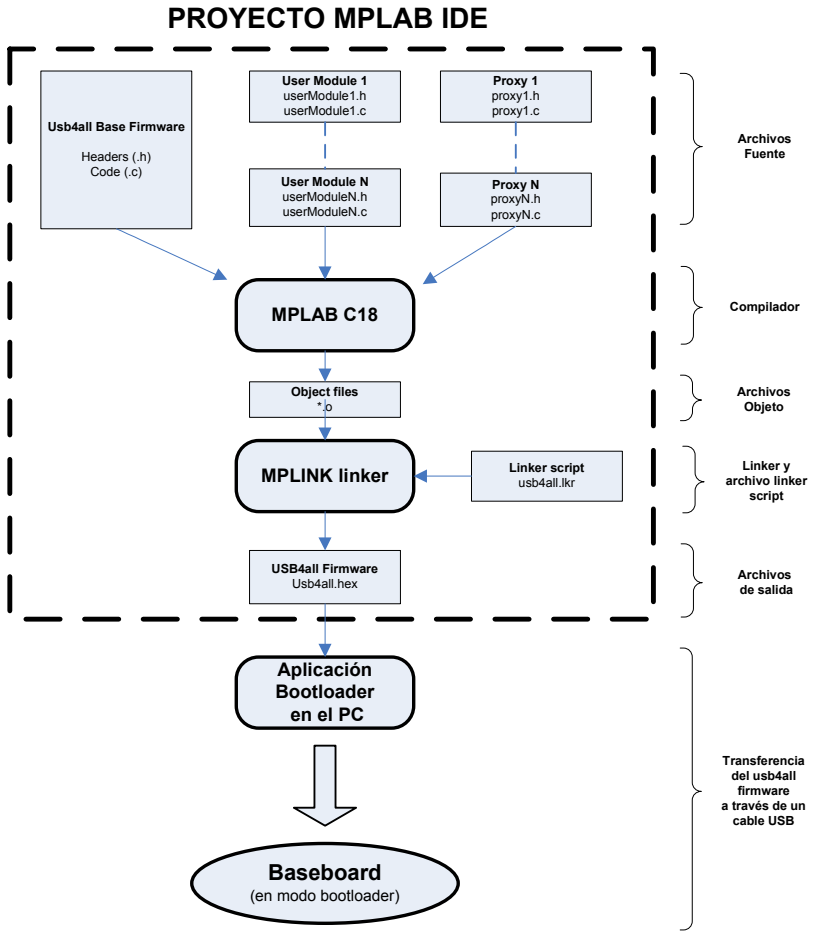


Figura 4.9: Proceso de desarrollo y deploy del *USB4all firmware*.

El *USB4all firmware*, está implementado en C, específicamente para el compilador MPLAB C18 de Microchip. Para generar éste firmware es necesario utilizar un proyecto que se brinda a manera de template. Dentro del proyecto se encuentran todos los fuentes del *base firmware* y se deben agregar los *proxies* y los *user modules* que sean necesarios para la solución a construir. El siguiente paso es compilar y linkeditar todos los fuentes, lo que da como resultado la generación de un archivo .hex, que describe una imagen de memoria de programa que debe ser grabada en el microcontrolador. Finalmente esta imagen es transferida al microcontrolador, utilizando el bootloader. El proceso se muestra en la figura 4.9.

4.2.6.1. Organización de la memoria de programa

Para la implementación de mecanismos de eliminación de dependencias del *base firmware* de los componentes y para la configuración de los descriptores USB se dividió la memoria de programa en regiones fijas. En la figura 4.10 se muestra el mapa de memoria de programa del microcontrolador.

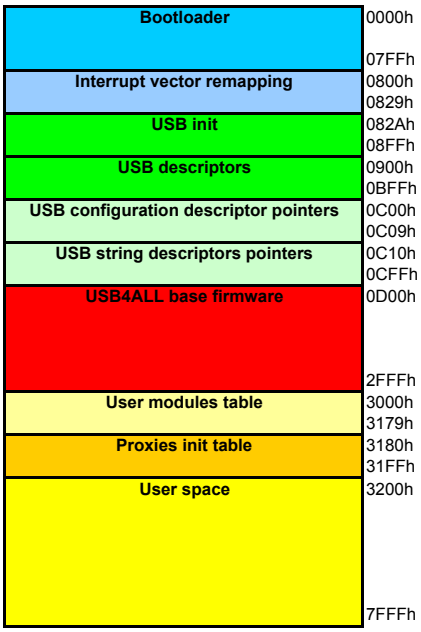


Figura 4.10: Mapa de memoria de programa del *baseboard*.

La primer área de memoria de programa es ocupada, como ya dijimos, por el código del bootloader. Esta es la única sección del mapa de memoria impuesta debido a las facilidades que otorga el hardware para este cometido, para la demás áreas fueron fijadas tratando de maximizar el espacio para *user modules* y *proxies*. A continuación se describen las demás áreas del *USB4all firmware:*

- **Interrupt vector remapping**: Es una pequeña área de memoria de programa en donde están mapeados los vectores de reset, e interrupciones de alta y baja prioridad. Aquí es donde las aplicaciones subidas por el bootloader ubican los saltos a sus funciones de main y manejadores de interrupciones, al igual que como lo harían si no estuviera el bootloader, pero sumando un offset de 0x0800. A modo de ejemplo, luego de un reset, el microcontrolador setea el contador de instrucciones en 0x0000 y comienza a ejecutar el código ubicado en dicha posición. Aquí el bootloader puede dependiendo del estado del botón de bootloader del *baseboard* iniciar en modo de ejecución Normal o Bootloader.
  En el modo Normal realiza un salto a la posición de memoria 0x0800, donde se encuentra un nuevo salto al main loop del *base firmware*. Eso mismo sucede con los vectores de interrupciones. Las direcciones de los nuevos vectores se hallan sumando un offset de 0x0800 a la direcciones predefinidas en el hardware del microcontrolador. En el **modo Bootloader**

el *baseboard* permanecerá esperando que se envíe el *USB4all firmware*, que será grabado dentro de la misma EEPROM del microcontrolador a partir de la dirección 0x0800.

- **USB Init**: Aquí se encuentra el código encargado de la inicialización de los recursos del microcontrolador destinados a la comunicación USB. Se inicializan los registros de la memoria compartida del microcontrolador, indicando tamaño y posición de los buffers de los endpoints, tipos de transferencia y dirección de la misma.
- **USB descriptors**: Aquí se encuentran almacenados todos los descriptores USB utilizados por el *baseboard*. Entre ellos se encuentran los descriptores del dispositivo, configuración, interfaces, endpoints y string. Están codificados según las especificaciones del capítulo 9 del estándar USB [15].
- **USB configuration descriptor pointers**: Son punteros hacia posiciones específicas del área *USB descriptors*. Son utilizados por el *base firmware* en el momento de la enumeración del dispositivo. Incluyen cantidad de interfaces, consumo de energía, y dentro de las interfaces: cantidad de endpoints, código de clase, de subclase, etc.
- **USB String descriptor pointers**: Son punteros hacia posiciones específicas del área *USB descriptors*. Son utilizados por el *base firmware* en el momento de la enumeración del dispositivo. Incluyen strings, que son descripciones, normalmente en ingles del tipo de producto USB. Además también son utilizados para contener un número de serie del producto.
- **USB4all base firmware**: En esta área se almacena todo el código correspondiente al *base firmware*.
- **User module table**: En esta área fija de memoria se ubica la tabla utilizada por el *base firmware* para encontrar las funciones de los *user modules*. Esta tabla está descripta en la sección 4.2.3.
- **Proxies init table**: Tabla con punteros al código de las operaciones `init` de los *proxies*. Esta tabla es utilizada por el *base firmware* para inicializar todos los *proxies*.
- **User Space**: En esta área de memoria de programa se almacena el código de los *user modules* y *proxies*.

Esta manera de organizar la memoria de programa tiene las siguientes ventajas:

1. **Provee de medios al *base firmware* para quedar desacoplado de las configuraciones USB y de la cantidad de *user modules* y *proxies***. Dado que el *USB4all firmware* define donde comienza cada área de memoria de programa, el *base firmware* puede localizar el código de las funciones o de los datos necesarios para la realización de su trabajo. Para el acceso a datos o a código fijo, su uso es directo, ya que conoce la dirección de inicio del área y su estructura. Para el caso donde la cantidad de elementos almacenados es variable, como es el caso de los *user modules* y *proxies*, se utilizan listas de referencias a las operaciones de cada uno de ellos. De esta manera, conociendo el inicio de la tabla y su estructura es posible recorrerla y ubicar las operaciones de cada uno.
2. **Permite la configuración de los recursos USB del *baseboard* en instancias posteriores**. Dado que el *base firmware* accede a las áreas de inicialización y descriptores USB en posiciones de memoria fijas, es posible modificar únicamente estas áreas específicas para cambiar los recursos y propiedades USB asignados al *baseboard* sin necesidad de modificar todo el *USB4all firmware*. En la sección 8.3 se muestra una aplicación que automatiza este mecanismo.

Capítulo 5

Arquitectura en el PC

5.1. Introducción

Como ya se mencionó en la introducción de la arquitectura de la solución (ver el capítulo 3), existen tres elementos en el PC: la biblioteca de clases Java, la *USB4all API* y los drivers USB genéricos. El primero de ellos permite el uso de la solución por parte de las aplicaciones de usuario, el segundo se encarga entre otras cosas de las funcionalidades de comunicación y gestión de los vínculos lógicos con los *user modules* y por último, el tercer elemento se encarga del manejo físico del puerto USB del PC.

La figura 5.1 permite visualizar las dependencias que se dan entre estos tres elementos. Además se observa que las aplicaciones de usuario pueden interactuar directamente con el *USB4all API* o por intermedio de la biblioteca de clases (*USB4all Library*), también se desprende que existen distintas opciones de drivers USB genéricos. Por un lado están los drivers que corren completamente en modo protegido dentro del sistema operativo y por otro, los llamados drivers modo usuario que exponen sus funcionalidades por medio de bibliotecas a los programas de alto nivel.

El resto de las secciones de este capítulo están destinadas a explicar en profundidad cada uno de los elementos que existen en el PC, comenzando por la *USB4all API*.

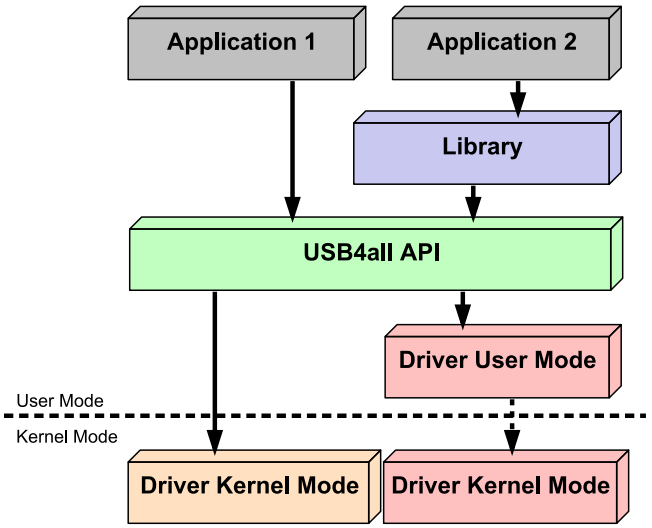


Figura 5.1: Diagrama de los elementos de la solución existentes en el PC.

5.2. USB4all API

5.2.1. Introducción

El pilar del lado del PC sobre el cual se edifica una parte importante de las funcionalidades de la arquitectura es el subsistema *USB4all API* (de aquí en más: *API*). Dentro de todas sus responsabilidades sobresale la de brindar a los programas de aplicación (alto nivel) una interfaz única y bien definida que les permita el manejo de el(los) dispositivo(s) electrónico(s) conectado(s) a el *baseboard* y por otro lado, la interacción con los drivers USB de las distintas plataformas.

Para cumplir con estas responsabilidades la *API* tiene la capacidad de establecer vínculos o canales lógicos con el *USB4all firmware* por medio del uso de la tecnología USB logrando encapsular y hacer transparente a los programas de aplicación toda la complejidad de la comunicación entre el PC y el *baseboard*. Las principales características de calidad y funcionalidad que se seleccionaron en el diseño de la *API* son las siguientes:

- **Modular:** Se busca que las funcionalidades que forman parte de la *API* estén organizadas en componentes con responsabilidades e interfaces claramente definidas.
- **Facilidad de uso:** Para reducir la dificultad al mínimo en el uso de la *API* por parte de los programas de aplicación, se ofrece una interfaz pública completa pero mínima en cuanto a la cantidad de primitivas que expone, comparable con las funcionalidades que ofrecen los sistemas operativos para el manejo de archivos a bajo nivel.
- **Diseño basado en capas:** Debido a sus responsabilidades con los programas de aplicaciones y los drivers USB se necesita tener un alto grado de desacoplamiento entre los componentes de la *API*, para ello se optó por un modelado en capas donde cada una tiene responsabilidades bien definidas. Además algunos de los componentes de la *API* mantienen una estrecha relación con el *USB4all firmware* debido a que implementan los protocolos de comunicación y como se vió en la sección 4.2.1, los componentes del *USB4all firmware* también están organizados en capas, lo que reafirma el enfoque tomado para el diseño de la *API*.
- **Portabilidad:** El diseño flexible permite el uso de la *API* en distintas plataformas (sistema operativos) con un impacto menor en cuanto a su implementación. Los escenarios de uso principales que se plantearon como objetivos en este proyecto de grado son las plataformas Windows y Linux.
- **Conexión orientada a user modules:** Las conexiones que se establecen entre el PC y el *baseboard* para la comunicación entre las aplicaciones de usuario y los *user modules* se instrumentan por medio de vínculos lógicos bidirecionales. Cada uno de estos vínculos lógicos están asociados a un único par de endpoints del *baseboard*, pero no en forma exclusiva, pues se busca optimizar la utilización de los recursos existentes en el *baseboard*. Además este enfoque de vínculos, busca aprovechar los distintos tipos de transferencias que posee la tecnología USB.
- **Soporte de múltiples *baseboards*:** Se busca que la *API* pueda trabajar concurrentemente con varios *baseboards* que estén conectadas a el PC de forma de permitir la escalabilidad de la cantidad de dispositivos electrónicos que se pueden controlar al mismo tiempo por un único programa de aplicación.
- **Registro de la comunicación:** Se considera importante registrar la información que se intercambia entre la *API* y el *USB4all firmware* de forma de brindar a los desarrolladores de programas de aplicación y del firmware una herramienta para la verificación de sus programas. Su implementación respeta la restricción de no afectar el desempeño del funcionamiento de la arquitectura y es por eso que el tipo de información que recolecta es reducido.

Como una primera aproximación al modelado de capas de la *API* se pueden distinguir tres grandes subsistemas en el diseño. La figura 5.2 ilustra estos tres subsistemas: *Public Interface*, *Logic* y *Connection to USB*.

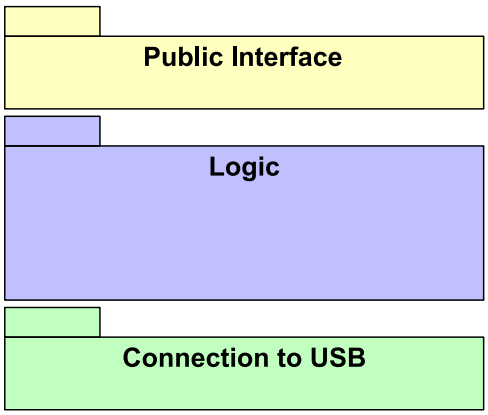


Figura 5.2: Subsistemas de la *API*.

El subsistema *Public Interface* expone un conjunto de primitivas a los programas de aplicación para que éstos puedan establecer conexiones con el *baseboard* y así poder comunicarse con los dispositivos electrónicos.

El subsistema *Logic* es el responsable de establecer y administrar los vínculos lógicos que se dan en la comunicación entre la *API* y el *USB4all firmware* y para ello implementa los protocolos de comunicación que están definidos en la arquitectura.

Y por último el subsistema *Connection to USB* es el responsable del manejo del puerto USB del PC de forma de ocultar todas las asimetrías de las distintas plataformas y drivers USB para lograr brindar al subsistema *Logic* una visión homogénea del medio USB.

Como lo muestra claramente la figura 5.2 estos subsistemas están diseñados siguiendo un modelo de capas en la cual la capa *Public Interface* es la que interactúa con los programas de aplicación recepcionando y devolviendo información, la capa *Logic* es quien tiene la capacidad establecer y administrar las conexiones, procesar los datos que se intercambian entre el *baseboard* para transformarlos en mensajes según lo establecidos en los protocolos de comunicación y por último la capa *Connection to USB* recibe y envía estos mensajes desde y hacia el *baseboard* por medio del puerto USB que maneja.

Ahora analizaremos en profundidad cada uno de estos subsistemas para poder describir sus componentes, características y funcionalidades particulares así como sus puntos de contacto con el *base firmware* y con las características de diseño anteriormente mencionadas.

5.2.2. Public Interface

El subsistema *Public Interface* está compuesto únicamente por el componente *u4aAPI*, el cual define el conjunto de primitivas que permiten la comunicación con el *baseboard*.

5.2.2.1. U4aAPI

Este componente está diseñado siguiendo el patrón Singleton [22] para poder brindar un único punto de acceso a la *API* y lograr por este medio un mayor control y orden en la interacción entre los programas de aplicación y el sistema.

Como se ve en la tabla 5.1 la comunicación con el *USB4all firmware* se resuelve básicamente por medio de cinco operaciones (las primeras en la tabla) muy simples e intuitivas, las cuales se asemejan mucho a las primitivas existentes en la mayoría de los sistemas operativos para el manejo de archivos a bajo nivel. Por medio de estas cualidades del componente *u4aAPI* se logra satisfacer la característica de calidad facilidad de uso, buscada en el diseño de la *API*.

Al igual que en el manejo de archivos donde la apertura da como resultado un identificador único que permite al sistema operativo identificarlo, la arquitectura define el concepto de *ModuleID* como un identificador único de cada uno de los vínculos lógicos que se establecen entre el PC y un *user module* de un *baseboard*, ésta referencia es obtenida como resultado de la operación `openDevice`.

Operación	Descripción
`openDevice`	Crear un vínculo o canal lógico con un *user module* existente en el *ubs4all firmware*.
`configDevice`	Se utiliza para enviar al *user module* una configuración inicial de las propiedades de los *proxies* que utiliza.
`sendData`	Envía información al *user module* que se indica como destinatario.
`receiveData`	Verifica si existe información enviada por el *user module* que se indica como emisor y la obtiene.
`closeDevice`	Destruye el vínculo lógico con un *user module* y libera los recursos que utilizaba el canal.
`qtyBaseBoards`	Devuelve la cantidad de baseboards conectados al PC.
`getBaseBoardSerial`	Dado un índice i, obtiene el número de serie del i-ésimo *baseboard* conectado a el PC.
`qtyUserModules`	Devuelve la cantidad de *user modules* presentes en un *baseboard*.
`getUserModuleName`	Dado un índice i, obtiene el nombre de i-ésimo *user module* presente en un *baseboard*.
`resetBaseBoard`	Indica al *base firmware* de un *baseboard* que debe cerrar todos los *user modules* activos.
`initAPI`	Inicializa la conexión con los *baseboards* conectados a el PC.
`apiVersion`	Obtiene el número de versión de la *API*.
`firmwareVersion`	Obtiene el número de versión del *base firmware* de un *baseboard*.

Cuadro 5.1: Interfaz del componente *u4aAPI*.

Este concepto es similar al *handler* que se explicó en la sección 4.2.2 pero distinto a la vez, pues los *moduleID* que se utilizan en la *API* son el resultado de la combinación de los números de serie y los *handler* que se utilizan en el *Base Firmware*.

Los números de serie son identificadores únicos de los *baseboards* y son utilizados para reconocerlas, permitiendo de esta forma que la *API* pueda trabajar con varios de ellos al mismo tiempo. Como ya se vió en la sección 2.1.3, los números de serie se almacenan en un descriptor básico del tipo String, opcional en el estándar USB y que la arquitectura lo convierte en obligatorio para su adecuado funcionamiento.

Para la correcta utilización de la operación `openDevice` es necesario obtener los números de series de los *baseboard* así como los nombres de los *user module* que ellos almacenan, para ello existen las operaciones `qtyBaseBoards`, `getBaseBoardSerial`, `qtyUserModules` y `getUserModuleName`. La primera operación permite consultar la cantidad de baseboards conectados al puerto USB del PC, la segunda permite obtener el número de serie del *baseboard* conectado en i-ésimo lugar. Las otras dos operaciones permiten consultar la cantidad de *user modules* existentes en un baseboard y obtener sus nombres según su orden de registro en el *base firmware*.

En la figura 5.3 se presenta un escenario hipotético de un PC con dos *baseboards* conectados por medio del puerto USB y dos dispositivos electrónicos (antena y display) conectados cada uno a un *baseboard* distinto, uno de ellos posee el número de serie 100 y tiene almacenados los *user module* de nombre Motor y Antena y el otro se identifica con el número de serie 200 y posee los *user module* de nombre Display y Termómetro.

Este ejemplo trata de explicar en forma visual como resuelve la *API* el problema de identificar unívocamente cada uno de los vínculos lógicos que se crean desde el PC cuando existen varios *baseboards* conectados. Como muestra el cuadro de texto de la figura 5.3, las invocaciones de las operaciones `qtyBaseBoards`, `getBaseBoardSerial`, `qtyUserModules` y `getUserModuleName` permiten obtener la información necesaria para poder crear las conexiones con los *baseboards* y las invocaciones de la operación `openDevice` de los *user module* de distintos *baseboards* tienen como resultado los *moduleID* 1 y 2.

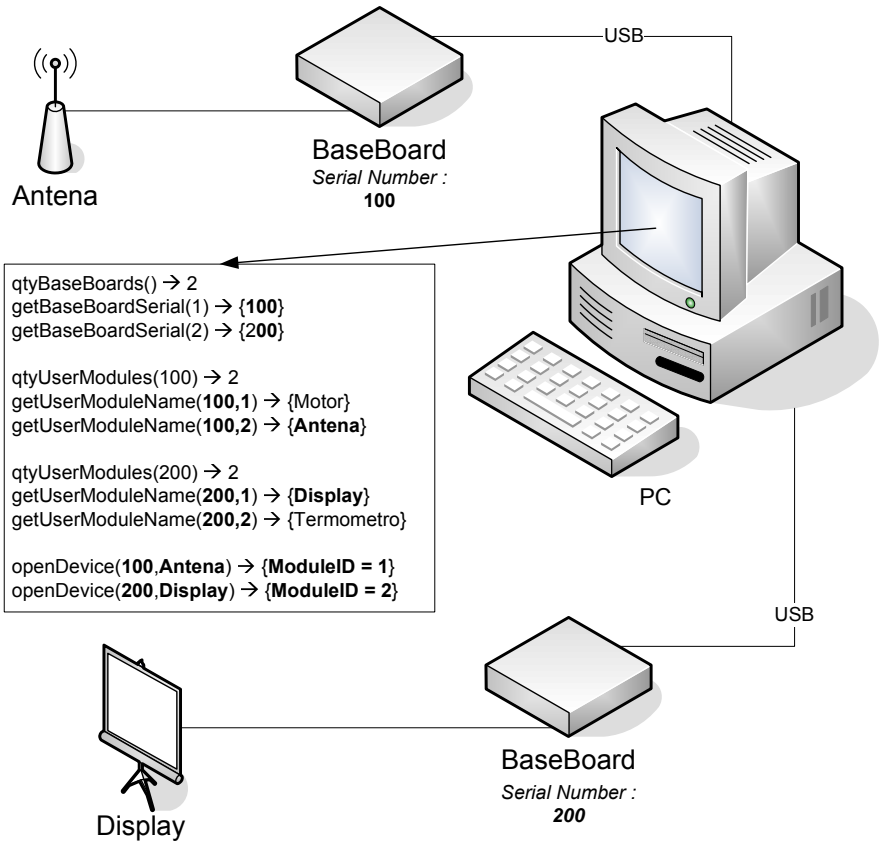


Figura 5.3: Escenario de uso con dos *baseboards*.

Operación	Parámetros de entrada		Parámetros de salida	
`openDevice`	N° de serie del *baseboard* Nombre del *user module* Tipo de envío Tipo de recepción	Entero Caracter Entero Entero	*moduleID*	Entero
`configDevice`	*moduleID* Configuración a enviar Largo del dato	Entero Caracter Entero	Éxito	Lógico
`sendData`	*moduleID* Dato a enviar Largo del dato Tiempo máximo de envío	Entero Caracter Entero Entero	Éxito	Lógico
`receiveData`	*moduleID*	Entero	Dato recibido Largo Éxito	Carácter Entero Lógico
`closeDevice`	*moduleID*	Entero	Éxito	Lógico
`qtyBaseBoards`			# *baseboards*	Entero
`getBaseBoardSerial`	Índice	Entero	N° de serie	Entero
`qtyUserModules`	N° de serie del *baseboard*	Entero	# *user modules*	Entero
`getUserModuleName`	N° de serie del *baseboard* Índice	Entero Entero	Nombre *user module* Largo	Carácter Entero
`resetBaseBoard`	N° de serie del *baseboard*	Entero		
`initAPI`				
`apiVersion`			Versión	Entero
`firmwareVersion`	N° de serie del *baseboard*	Entero	Versión	Entero

Cuadro 5.2: Detalle de las operaciones del componente *u4aAPI*.

La tabla 5.2 muestra los parámetros de entrada y de salida de las operaciones del componente *u4aAPI*, como se puede ver las operaciones `apiVersion` y `firmwareVersion` tienen caracter únicamente informativo sobre las versiones de los subsistemas que componen la arquitectura mientras que las operaciones `qtyBaseBoards`, `getBaseBoardSerial`, `qtyUserModules` y `getUserModuleName` permiten obtener información de los *baseboard* y de sus *user module*.

El resto de las operaciones con las excepciones de `resetBaseBoard` e `initAPI` (que explicaremos más adelante) son las destinadas a la comunicación y debido a eso tienen como parámetro de entrada el *moduleID* para poder indicar a la *API* que vínculo lógico está invocando la operación. Otra característica destacable que poseen las primitivas `sendData` y `receiveData` es la posibilidad de indicar por medio de uno de sus parámetros un tiempo máximo (timeout) para la ejecución de la operación. Por último están las operaciones `resetBaseBoard` e `initAPI`, la primera permite notificarle a un *baseboard* que debe cerrar todos sus *user modules* activos de forma de simular una acción de reset física y la segunda se utiliza para inicializar los componentes de la *API* previo a su uso para la comunicación.

En la siguiente sección estudiaremos a fondo el subsistema *Logic*, analizando sus componentes y la forma en que interactúan entre ellos y con los componentes de los subsistemas *Public Interface* y *Connection to USB*.

5.2.3. Logic

El subsistema *Logic* es el más importante de los tres subsistemas que componen la *API*, debido a que nuclea a la mayor parte de los componentes y además es responsable de establecer y administrar los vínculos lógicos que se establecen con los *baseboards* conectadas al PC, que constituye el objetivo principal de la *API*.

Sus componentes buscan satisfacer las características de diseño: modular y basado en capas, de forma de lograr un bajo acoplamiento y una alta cohesión de las funcionalidades, además de permitir un fácil entendimiento de como funcionan en conjunto.

Para alcanzar estos objetivos, el subsistema *logic* implementa los protocolos de comunicación que define la arquitectura por medio de dos componentes llamados *HandlerLayer* y *CommandLayer* que son las contrapartes en el PC de los componentes *handler manager* y *admin module* del *base firmware* del *baseboard* (ver secciones 4.2.2.1 y 4.2.2.2). Además se encuentra el componente *DescriptorLayer*, que tiene como responsabilidad la interacción con el subsistema *connection to USB* de forma de proveer a los componentes que implementan los protocolos de comunicación la información de los descriptores USB y *baseboards* de una forma simple, dejando la mayor complejidad a las capaz inferiores de la *API*.

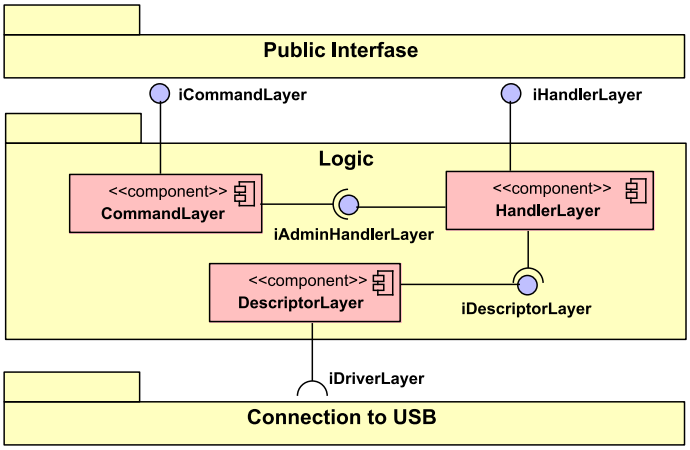


Figura 5.4: Componentes del subsistema *Logic*

Como se ve en la figura 5.4, los componentes *CommandLayer* y *HandlerLayer* son los encargados de ofrecer los servicios necesarios al subsistema *public interface* para que este pueda

cumplir con el servicio de comunicación que ofrece a los programas de aplicación y por otro lado el componente *descriptorlayer* (que no implementa los protocolos de comunicación) utiliza los servicios que ofrece el subsistema *connection to USB* para simplificar la información relacionada con la tecnología USB de forma de facilitar el funcionamiento de los componentes *commandlayer* y *handlerlayer*.

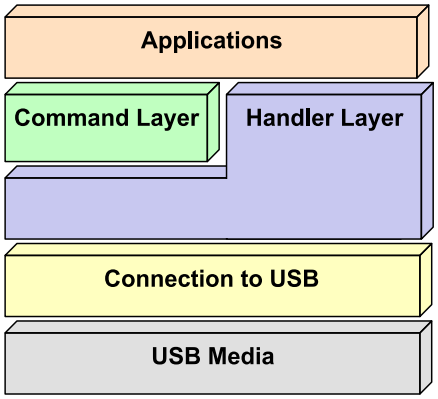


Figura 5.5: Diagrama de capas de la comunicación en el PC.

La figura 5.5 muestra en forma resumida los principales elementos que constituyen el modelado en capas de la comunicación con los *baseboards*. En primer lugar se encuentra la capa *Application* que corresponde a los programas de aplicación que corren en el PC y su nivel de comunicación es con los *user modules*, luego se observa los ya mencionados *commandlayer* y *handlerlayer* que se encargan del manejo de los mensajes por medio de los protocolos de comunicación que define la arquitectura (ver sección 6.1). La siguiente capa es el subsistema *connection to USB* que permite la estandarización del acceso a los drivers USB que se existen en las distintas plataformas y por último, se encuentra la capa *USB Media* que representa a los drivers y puertos USB de la PC. A continuación se analizan las características más importantes así como sus funcionalidades específicas de cada uno de los componentes que constituyen el subsistema *logic*.

5.2.3.1. HandlerLayer

El componente *handlerlayer* es la contraparte en el PC del componente *handler manager* que integra el *base firmware* y al igual que éste, es el responsable del envío y recepción de los mensajes que se intercambian los programas de aplicación y los *user modules*. Para cumplir con dicha tarea implementa el *handler protocol* definido en la arquitectura y se encarga del almacenamiento de los vínculos lógicos que se establecen con los *baseboards*.

El *handlerlayer* define las interfaces *iHandlerLayer* e *iAdminHandlerLayer* para ofrecer sus funcionalidades, la primera se expone al subsistema *public interface* (componente *u4aAPI*) que la utiliza para la recepción y envío de datos mientras que la segunda que extiende la interfaz *ihandlerlayer* se expone al componente *commandlayer* que la utiliza para enviar mensajes al *base firmware* como se estudiara en la próxima sección.

En la figura 5.6 se observa el conjunto de clases que constituyen al *handlerlayer*, así como son sus relaciones y quien de ellas implementa las interfaces que define el componente.

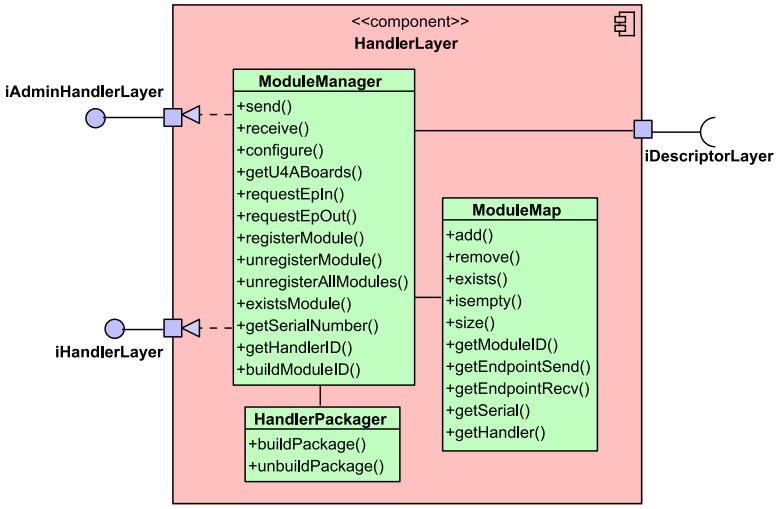


Figura 5.6: Clases del componente *HandlerLayer*.

A continuación para cada una de estas clases se explicarán sus características principales y se describirán sus operaciones.

ModuleManager La clase *ModuleManager* es la responsable de orquestar el funcionamiento de todo el componente y de esa forma permitir el tráfico de mensajes así como la persistencia de los vínculos lógicos. A causa de esa responsabilidad es la clase encargada de implementar las interfaces *ihandlerlayer* e *iadminhandlerlayer* para ofrecer a los componentes *u4aAPI* y *commandlayer* las funcionalidades que requieren para su funcionamiento.

Operación	Descripción
`send`	Envía al *user module* identificado por el *moduleID* pasado por parámetro un dato. Opcionalmente se puede indicar por parámetro un tiempo máximo de ejecución (timeout).
`receive`	Recibe un dato del *user module* identificado por el *moduleID* pasado por parámetro. Opcionalmente se puede indicar por parámetro un tiempo máximo de espera (timeout).
`configure`	Envía al *user module* identificado por el *moduleID* pasado por parámetro un dato para que pueda configurarse.

Cuadro 5.3: Operaciones de la interfaz *iHandlerLayer*

El cuadro 5.3, muestra las operaciones de la interfaz *ihandlerlayer* que define las operaciones `send` y `receive` que permiten el tráfico de mensajes y al igual que las operaciones homólogas del componente *u4aAPI*. También aceptan parámetros para indicar los tiempos máximos aceptados para el envío o recepción de mensajes y por último se encuentra la operación `configure` que permite enviar información a un *user module* para que este pueda configurarse.

El cuadro 5.4 en cambio, muestra las operaciones de la interfaz *iadminhandlerlayer*, que se pueden agrupar en tres grupos: las relacionadas con el componente *descriptorlayer* (`getU4ABoards`, `requestDscIn`, `requestDscOut`) pues consultan información de los descriptores USB y de los *baseboards*, las relacionadas con la clase *modulemap* (`registerModule`, `existsModule`, `unregisterModule`, `unregisterAllModule`, `getSerialNumber`, `getHandlerID`) pues permiten el registro y consulta de la información de los vínculos lógicos que se almacenan en ella y por último las operaciones propias de la clase *modulemanager* (`buildModuleID` y las operaciones `send`, `receive` y `configure` que se heredan de la interfaz *ihandlerlayer*).

Operación	Descripción
`getU4ABoards`	Obtiene la lista de números de serie de los *baseboards* conectados a el PC.
`requestEpIn`	Solicita un número de endpoint que utilice un tipo de transferencia particular que permita la recepción de datos de un *baseboard* dado.
`requestEpOut`	Solicita un número de endpoint que utilice un tipo de transferencia particular que permita el envío de datos a un *baseboard* dado.
`registerModule`	Registra un vínculo lógico.
`unregisterModule`	Desregistra un vínculo lógico.
`unregisterAllModule`	Desregistra todos los vínculo lógico registrados.
`existsModule`	Consulta si ya está registrado o no un vínculo lógico (*moduleID*).
`getSerialNumber`	Devuelve el número de serie asociado al vínculo lógico identificado por un *moduleID* pasado por parámetro.
`getHandlerID`	Devuelve el *handler* asociado al vínculo lógico identificado por un *moduleID* pasado por parámetro.
`buildModuleID`	Construye un *moduleID* utilizando un número de serie de un *baseboard* y un *handler* de un *user module*.

Cuadro 5.4: Operaciones de la interfaz *iAdminHandlerLayer*.

HandlerPackager La clase *HandlerPackager* se especializa en el conocimiento del *handler protocol* (ver sección 6.1.1) y debido a ésto ser encarga del empaquetado y desempaquetado de los mensajes que llegan al componente por medio de sus dos únicas operaciones `buildPackage` y `unbuildPackage` como se explica en la figura 5.5.

Operación	Descripción
`buildPackage`	Empaqueta según indica el *handler protocol* un conjunto de datos pasados como parámetro en un estructurado que define la arquitectura (hndPackage).
`unbuildPackage`	Dado un mensaje del *handler protocol* desempaqueta su contenido y lo devuelve en un estructurado que define la arquitectura (hndPackageResponse).

Cuadro 5.5: Interfaz de la clase *HandlerPackager*.

ModuleMap La clase *ModuleMap* implementa el tipo abstracto de datos (TAD) Map y se encarga de almacenar la información (*moduleID*, los endpoints USB asignados para enviar y recibir datos, el número de serie del *baseboard* y el *handler* asignado por el *base firmware*) de cada vínculo lógico que se crea en la *API*. La clave del Map son los valores *moduleID* y su implementación fue pensada para evitar la duplicación de claves de forma de garantizar que la información de cada vínculo lógico sea almacenada una única vez. Como muestra el cuadro 5.6 las operaciones de la clase son las típicas del TAD, con la salvedad de que no existe una única operación **get** para obtener el valor almacenado en el Map sino cuatro operaciones (`getEndpointSend`, `getEndpointRecv`, `getSerial` y `getHandler`) que permiten obtener por separado cada uno de los valores que el Map almacena asociado a cada *moduleID*.

Operación	Descripción
`add`	Crea una nueva entrada en el MAP utilizando el parámetro *moduleID* como clave y como valores asociado los números de endpoints asignados para la comunicación, el número de serie del *baseboard* y el *handler* del *user module*.
`remove`	Elimina del MAP la entrada con clave igual al valor *moduleID* pasado como parámetro.
`exists`	Consulta si el *moduleID* pasado como parámetro ya existe o no en el MAP.
`isEmpty`	Consulta si el MAP posee alguna entrada creada o si está vacío.
`size`	Devuelve la cantidad de entradas que almacena el MAP.
`getModuleID`	Devuelve el *moduleID* ubicado en la i-ésima posición del MAP.
`getEndpointSend`	Devuelve el número de endpoint para envío de datos asociado al *moduleID* pasado como parámetro.
`getEndpointRecv`	Devuelve el número de endpoint de recepción de datos asociado al *moduleID* pasado como parámetro.
`getSerial`	Devuelve el número de serie del *baseboard* asociado al *moduleID* pasado como parámetro.
`getHandler`	Devuelve el *handler* del *base firmware* asociado al *moduleID* pasado como parámetro.

Cuadro 5.6: Interfaz de la clase *ModuleMap*

5.2.3.2. CommandLayer

Otro de los componentes que forman parte del subsistema *logic* es el *commandlayer*, que es la contraparte en el PC del componente *admin module* del *base firmware*. Su responsabilidad más importante es la gestión de los vínculos lógicos que existen en la *API* y para ello implementa el *Admin Protocol* que es un protocolo fijo que define un conjunto de comandos orientados a la gestión como son **OPEN**, **CLOSE**, etc. Por más detalles de este protocolo ver la sección 6.1.2.

El *commandlayer* define la interfaz *iCommandLayer* que expone al componente *u4aAPI* del subsistema *public interface* para que este pueda ofrecer a los programas de aplicación las funcionalidades de creación y destrucción de vínculos lógicos con los *user modules*.

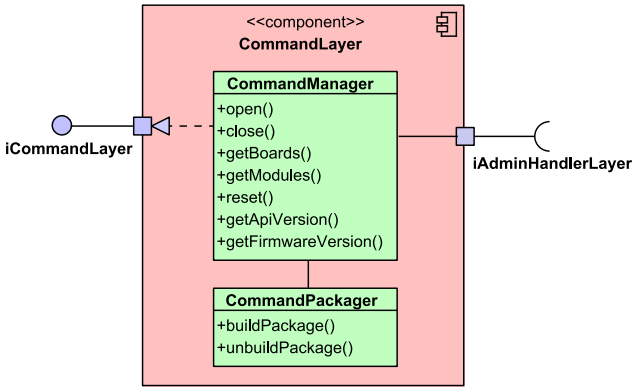


Figura 5.7: Clases del componente *CommandLayer*.

Como muestra la figura 5.7 el componente *commandlayer* está compuesto por la clase *CommandManager* que se encarga de implementar la lógica para la gestión de los vínculos lógicos y por la clase *CommandPackager* que se ocupa del empaquetado y desempaquetado de los mensajes que se intercambian entre el PC y el *baseboard* al establecerse o eliminarse las conexiones. A continuación se describen las operaciones y características principales de estas dos clases así como de la implementación de la interfaz *icommandlayer*.

CommandManager Esta clase es la encargada de la gestión del proceso de creación y destrucción de vínculos lógicos, para ello implementa la interfaz *icommandlayer* que expone al componente *u4aAPI* para que esta pueda ofrecer sus funcionalidades a las programas de aplicación. Además, utiliza la interfaz *iadminhandlerlayer* para poder realizar las tareas de enviar y recibir los mensajes que define el *admin protocol* y poder almacenar la información asociada a los vínculo lógicos que se establecen en la *API*.

Operación	Descripción
`open`	Crea un vínculo lógico con un *user module* de un *baseboard*. Recibe como parámetros el nombre del *user module*, los tipos de transferencias de envío y recepción de datos y el número de serie del *baseboard* y devuelve como identificador del vínculo un *moduleID*.
`close`	Destruye el vínculo lógico identificado por el *moduleID* pasado como parámetro y libera los recursos utilizados por la conexión.
`getBoards`	Obtiene los números de serie de los *baseboard* conectados a el PC.
`getModules`	Devuelve los nombres de los *user modules* existentes en el *baseboard* que se identifica por el número de serie pasado como parámetro.
`reset`	Indica al *baseboard* que se identifica por el número de serie pasado por parámetro que debe destruir todos los vínculos lógicos activos en él.
`getFirmwareVersion`	Devuelve el número de versión del subsistema *base firmware* del *baseboard* conectado a el PC que se identifica por el número de serie pasado como parámetro.
`getApiVersion`	Devuelve el número de versión del sistema *API*.

Cuadro 5.7: Operaciones de la interfaz *iCommandLayer*

Como muestra el cuadro 5.7 la interfaz *icommandlayer* define las operaciones `open` y `close` para la creación y destrucción de vínculos lógicos. Para poder utilizar correctamente la operación `open` es necesario obtener los números de serie de las *baseboard* conectadas al PC así como los nombres de los *user module* que ellas almacenan, para esto existen las operaciones `getBoards` y `getModules`. Por último está la operación `reset` que se encarga de destruir todos los vínculos lógicos existentes con un *baseboard* dado liberando todos los recursos involucrados de forma de simular vía software el resultado de reiniciar físicamente el microcontrolador del *baseboard*.

CommandPackager La clase *CommandPackager* se especializa en el conocimiento del *admin protocol* y debido a ésto se encarga del empaquetado y desempaquetado de los mensajes que llegan al componente por medio de sus dos únicas operaciones `buildPackage` y `unbuildPackage` como se explica en la figura 5.8.

Operación	Descripción
`buildPackage`	Empaqueta según indica el *admin protocol* un conjunto de datos pasados como parámetro en un estructurado que define la arquitectura (CommPackage).
`unbuildPackage`	Dado un mensaje del *admin protocol* desempaqueta su contenido y lo devuelve en un estructurado que la arquitectura (CommPackageResponse).

Cuadro 5.8: Interfaz de la clase *CommandPackager*.

5.2.3.3. DescriptorLayer

El último elemento que integra el subsistema *logic* es el componente *DescriptorLayer*. Sus principales obligaciones son: el procesamiento y almacenamiento de la información de los descriptores de los *baseboards* conectados a el PC, el almacenamiento de las vinculaciones de los números de serie de los *baseboards* con los identificadores de instancia[1] y por último ser responsable de la conexión entre los subsistemas *logic* y *connection to USB* de forma de permitir el envío y recepción de mensajes hacia el puerto USB.

El componente *descriptorlayer* define la interfaz *iDescriptorLayer* que expone al componente *handlerlayer* para permitirle el intercambio de mensajes con los *user modules* y la solicitud de los números de descriptores existentes en los *baseboards* que cumplen con un tipo de transferencia dado. Para poder brindar los servicios anteriormente mencionados utiliza la interfaz *iDriverLayer* perteneciente al subsistema *connection to USB* para poder enviar y recibir mensajes del puerto USB y obtener la información sin procesar de los descriptores USB (formato estándar [5]) de los *baseboards* conectados a los puertos USB del PC.

DescriptorManager Como muestra la figura 5.8 el componente *descriptorlayer* está compuesto únicamente por la clase *DescriptorManager* que implementa la interfaz *idescriptorlayer* para brindar los servicios de mensajería (envío y recepción) y consulta de los descriptores USB y números de serie de los *baseboards* al componente *handlerlayer*. Para ello utiliza las funcionalidades que se exponen en la interfaz *iDriverLayer* del subsistema *connection to USB* que le permiten comunicarse con el medio USB como veremos en la próxima sección. Para el almacenamiento de la información procesada de los descriptores USB de los *baseboards* así como de las vinculaciones de sus números de serie con los identificadores de bajo nivel (instancia) se utiliza una implementación genérica del TAD MAP.

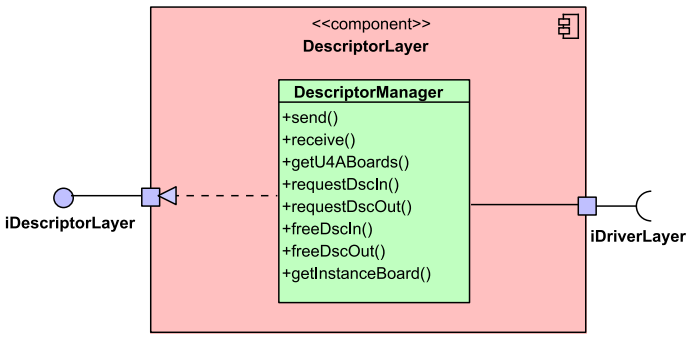


Figura 5.8: Clases del componente *DescriptorLayer*.

La operaciones `send` y `receive` ofician de pasarela para sus homólogas del componente *handlerlayer* en la comunicación con el subsistema *connection to USB*, su único aporte y causa de sus existencias es que permite aislar los componentes que implementan los protocolos de comunicación de los que manejan el medio USB de forma de lograr un mayor grado de generalidad. El resto de las operaciones que define la interfaz *idescriptorlayer* tienen el cometido de obtener información de los *baseboards* conectados al PC (`getU4ABoards` y `getInstanceBoard`) y de gestionar los endpoints activos de cada uno de ellos que participan en la comunicación entre los *user modules* y los programas de aplicación (`requestDscIn`, `requestDscOut`, `freeDscIn`, `freeDscOut`). En la próxima sección se analizará en detalle el conjunto de componentes que forman parte del último de los subsistema en que está dividido el *USB4all API*, el subsistema *Connection to USB*.

[1] Los identificadores de instancias son elementos utilizados en el subsistema *connection to USB* para poder reconocer los *baseboards* a bajo nivel.

Operación	Descripción
`send`	Envía un paquete de datos al subsistema *connection to USB* pasando como parámetros: el número de serie del *baseboard* así como el número de descriptor USB que debe utilizar para el envío.
`receive`	Recibe un paquete de datos del subsistema *connection to USB* pasando como parámetros: el número de serie del *baseboard* así como el número de descriptor USB que debe utilizar para la recepción.
`getU4ABoards`	Devuelve una lista con los números de serie de los *baseboards* conectados al PC.
`requestDscIn`	Solicita un número de descriptor USB de recepción del tipo de transferencia indicado y del *baseboard* que se identifica con el número de serie pasado por parámetro.
`requestDscOut`	Solicita un número de descriptor USB de envío del tipo de transferencia indicado y del *baseboard* que se identifica con el número de serie pasado por parámetro.
`freeDscIn`	Libera un número de descriptor USB de recepción previamente solicitado.
`freeDscOut`	Libera un número de descriptor USB de envío previamente solicitado.
`getInstanceBoard`	Devuelve el identificador de instancia que está asociado al número de serie pasado como parámetro.

Cuadro 5.9: Operaciones de la interfaz *iDescriptorLayer*

5.2.4. Connection to USB

El subsistema *Connection to USB* es responsable de la comunicación con los drivers USB genéricos para lograr el intercambio de información entre los programas de aplicación y los *user modules*. Por otro lado, también se encarga de interactuar con las distintas plataformas en donde ejecuta la *API* para detectar la cantidad de *baseboards* conectados al PC y obtener de cada un de ellos la información de sus descriptores.

Para cumplir con estas dos obligaciones el subsistema define los componentes *DriverLayer* y *PlatformLayer* como se observa en la figura 5.9, el primero se encarga del manejo del driver USB genérico y el segundo posee la lógica para interactuar con el sistema operativo con el fin de obtener los descriptores de los *baseboards*. Esta separación de responsabilidades permite generar una estructura lo más modular posible de forma de facilitar la extensibilidad de la solución. Para más detalles del porqué de esta separación de funcionalidades leer la sección 7.3.

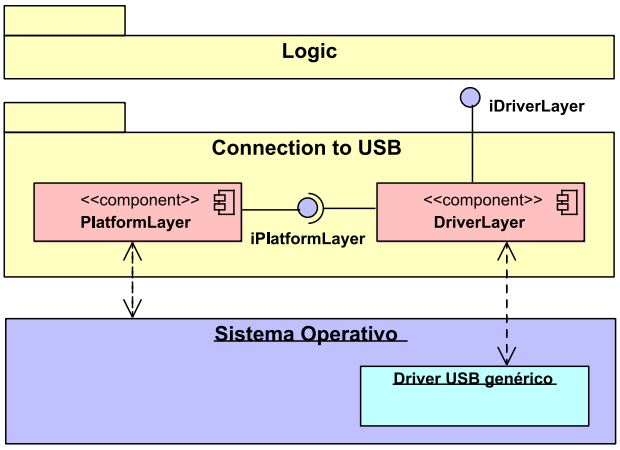


Figura 5.9: Componentes del subsistema *Connection to USB*.

En las siguientes secciones se describen las principales características y funcionalidades de los componentes *driverlayer* y *platformlayer*.

5.2.4.1. DriverLayer

El componente *driverlayer* es el encargado de la comunicación con las distintas implementaciones de drivers USB genérico que se pueden utilizar como parte de la solución. Además, es el responsable de interactuar con el componente *descriptorlayer* para la recepción y envío de paquetes que intercambian los programas de aplicación y los *user modules*. Para ésto último el componente *driverlayer*, define la interfaz *iDriverLayer* que expone al *descriptorlayer* para permitirle entre otras cosas, la creación y destrucción de las conexiones físicas con los endpoints del *baseboards* y el intercambio de paquetes. Una característica importante de éste componente es que fue diseñado para establecer conexiones físicas con endpoints en forma individual y no a nivel de todo el dispositivo. Ésto se debe principalmente, a que existen drivers genéricos que funcionan con un enfoque de manejo de todo el dispositivo y otros en los que es necesario instanciar la conexión para cada endpoint en forma independiente. La decisión de implementar éste componente con el enfoque de conexión a endpoints individuales no constituye una limitante, pues para los drivers que funcionan con el enfoque de manejo del todo el dispositivo, se puede implementar la lógica necesaria para simular las conexiones con los endpoints en forma individual.

Operación	Descripción
`getU4ABoards`	Devuelve una lista con los números de serie de los *baseboards* conectados al PC.
`qtyDsc`	Devuelve la cantidad de descriptores que posee un *baseboard* dado.
`getEndpointDsc`	Devuelve la información básica (número, tipo y tipo transferencia) del i-ésimo endpoint de un *baseboard*.
`openIn` `openOut`	Establece la conexión con un endpoint (entrante o saliente) del tipo de transferencia adecuado de un *baseboard* y devuelve un identificador único de la conexión.
`close`	Cierra la conexión con un endpoint de un *baseboard* por medio de su identificador.

`sendInt`
`sendCtrl`
`sendIso`
`sendBulk`
	Envía un paquete de datos a un *baseboard* a través de un endpoint (del tipo del transferencia adecuado) asociado al identificador de una conexión.

`receiveInt`
`receiveCtrl`
`receiveIso`
`receiveBulk`
	Recibe un paquete de datos de un *baseboard* a través de un endpoint (del tipo de transferencia adecuado) asociado al identificador de una conexión.

Cuadro 5.10: Operaciones de la interfaz *iDriverLayer*.

En el cuadro 5.10 se muestra el conjunto de operaciones que definen la interfaz *idriverlayer*, las operaciones `getU4ABoards`, `qtyDsc` y `getEndpointDsc` permiten obtener los números de serie de los *baseboards* conectados al PC así como la cantidad y la información de sus descriptores. Esta información es obtenida del componente *platformlayer* (como veremos en la próxima subsección) y es utilizada por el componente *descriptorlayer* para la creación de las conexiones con los endpoints de los distintos *baseboards* durante la etapa de inicialización de la *USB4all API*. Además, la interfaz define las operaciones `openIn`, `openOut`, `close` y los juegos de operaciones `send` y `receive` para cada uno de los tipos de transferencias USB. Estas operaciones son utilizadas por el *descriptorlayer* junto con la información de los *baseboards* (número de series y descriptores) para gestionar las conexiones físicas que permiten la comunicación entre el PC y los *baseboards*.

En el contexto de este proyecto se utilizaron distintas implementaciones de drivers genéricos como son: el facilitado por el fabricante Microchip, los drivers modo usuario LibUSb y LibUSB-Win32 [25] y el driver construido especialmente para la plataforma Linux y para cada una de ellos se implementó una clase particular en C++ que implementa la interfaz *idriverlayer*. Con este mecanismo de extensión de la API para el manejo de distintas implementaciones de drivers genéricos y sumado a las características del componente *platformlayer* (que veremos más ade-

lante), se logra satisfacer la característica de funcionalidad de portabilidad que era uno de los objetivos buscados en el diseño de la *API*.

5.2.4.2. PlatformLayer

Su misión es encapsular toda la lógica necesaria para la interacción con las distintas plataformas en que ejecuta la *API* con el fin de detectar los *baseboards* conectados al PC y obtener la información de sus descriptores. Este componente define la interfaz *iPlatformLayer* que es utilizada por el componente *driverlayer* para la obtención de la información de los *baseboards*. Como se ve en el cuadro 5.11 la interfaz define las operaciones: `findDevices` y `getDescriptors`. La primera se encarga de interactuar con el sistema operativo para detectar los *baseboard* y sus descriptores y generar una colección con dicha información (se utiliza un implementación genérica del TAD MAP) y la otra simplemente la devuelve. Ambas son invocadas únicamente en la inicialización del componente *driverlayer* pues la *API* no fue pensada para detectar la adhesión o remoción de *baseboards* luego de inicializada la misma.

Operación	Descripción
`findDevices`	Busca la existencia de *baseboard* conectados al PC y obtiene la información de sus descriptores.
`getDescriptors`	Devuelve una colección con los juegos de descriptores de cada uno de los *baseboards* detectados.

Cuadro 5.11: Operaciones de la interfaz *iPlatformLayer*.

Este componente es fundamental para poder satisfacer la característica de portabilidad buscada en el diseño de la *API*. Tiene como beneficios permitir la extensibilidad de la *API* por medio del desarrollo de clases en C++ que implementan la interfaz *iplatformlayer* para distintas plataformas. Además permite reutilizarlo cuando cambian las implementaciones del componente *driverlayer* y se mantiene la plataforma. En el marco de este proyecto se desarrollaron dos implementaciones del componente *platformlayer* para permitir el funcionamiento de la solución en las plataformas Linux y Windows.

5.2.5. Log

Una de las características seleccionadas para el diseño de la *API* fue el brindar como parte de sus funcionalidades un registro del tráfico de paquetes que se da entre las aplicaciones de usuario y los *baseboards* conectados al PC. Para ello se desarrolló un componente auxiliar llamado *Log*, que implementa el patrón de diseño Singleton y que permite ir almacenando en un archivo de texto de nombre *u4aapi.log* un detalle del tráfico de paquetes y sus contenidos en un formato legible. Su interfaz se limita a dos simples operaciones: `printLog` y `printLogLn` que permiten escribir en una línea un mensaje y ingresar un símbolo salto de línea en el archivo de texto. Como parte del formato que sigue el contenido del archivo de texto todas las líneas comienza con el día-hora en que se ingresa al archivo cada línea y a continuación el texto deseado.

Este componente no pertenece a ninguno de los subsistemas previamente descritos en esta sección, pues no participa de la función principal de la *API* que es la de comunicación, sino que realiza una tarea secundaria que permite la generación de una base de información histórica de la interacción con los dispositivos que facilita la depuración y detección de errores en la comunicación.

5.3. USB4all Library

La programación orientada a objetos es uno de los paradigmas de mayor aceptación en la actualidad, y consiste en utilizar objetos y sus interacciones para diseñar aplicaciones y programas de computadora. A fines de facilitar el uso de la solución *USB4all*, se brinda la posibilidad al usuario de poder utilizar librerías orientadas a objetos en donde se encapsulan las funcionalidades de la *USB4all API* de una forma más intuitiva y ordenada. Existen muchos lenguajes orientados a objetos, entre los que se destacan C++, Java, Python, VB.NET, C#.NET, etc. Algunas diferencias importantes a la hora de la implementación entre los lenguajes, son el manejo de la memoria (ej: garbage collector), la posibilidad de herencia múltiple y si el código generado es nativo o corre mediante una maquina virtual o con compilación JIT. De todas formas el diseño utilizado puede servir como modelo para portar a otros lenguajes las funcionalidades implementadas.

En el contexto de este proyecto se presenta una librería que permita un uso orientado a objetos y a la vez sirva de ejemplo de utilización de un lenguaje de programación distinto al que fue realizada la *USB4all API*. Una cualidad que también es tomada en cuenta es la posibilidad de utilización de esta librería tanto en entorno Windows como Linux, de manera que las aplicaciones construidas sobre ésta sean transparentes al sistema operativo utilizado. Considerando las alternativas existentes fue seleccionada la opción de Java, que cuenta con las ventajas de ser gratuito, multi-plataforma y de uso extendido. Como contraparte, éste lenguaje utiliza una maquina virtual para la ejecución de las aplicaciones, por lo que genera algunas dificultades para la programación en bajo nivel y para la utilización de librerías de vínculo dinámico (Dynamic Link Library) (DLL) como lo es la *USB4all API*. En otros lenguajes de programación como C++, la interacción con la *USB4all API* sería directa, pero en Java vale la pena ahondar un poco más en esta interacción. Java brinda un mecanismo para la ejecución de operaciones en librerías de vinculación dinámica llamado JNI el cual está poco documentado, hay pocos ejemplos disponibles y es necesario compilar ciertos componentes en lenguaje C o C++. En su lugar se decidió utilizar JNative que es un proyecto open source que simplifica el uso de librerías de vinculación dinámica y a su vez encapsula las diferencias de estas entre los distintos sistemas operativos utilizados. La otra decisión de diseño tomada, fue realizar en un componente separado, un envoltorio (Wrapper) de la *USB4all API* de manera de poder realizar aplicaciones que utilicen directamente los servicios de ésta. El componente *USB4allAPIWrapper* tiene como única responsabilidad el encapsular el llamado a las operaciones de la *USB4all API* a los clientes de la misma en Java. Todas las operaciones tienen los mismos nombres y argumentos que en la *API*, sólo difieren algunas de las firmas debido a que en Java no es posible utilizar parámetros de tipos de datos simples pasados por referencia (no son clases) de una función como argumentos de salida, como si lo es en C++.

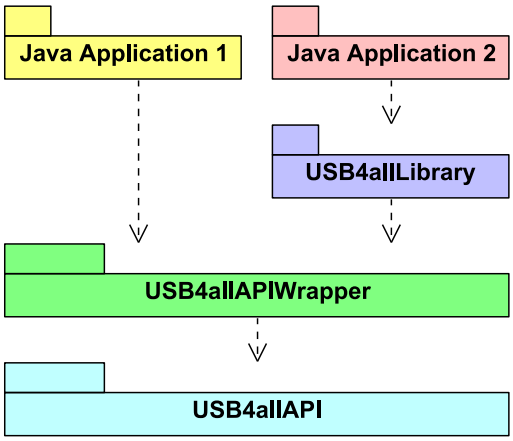


Figura 5.10: Dependencias para el uso de las librería orientada a objetos

En la figura 5.10 se muestra la interacción entre los distintos componentes y dependencias entre las aplicaciones que utilizan la *USB4all Library*. Aquí se muestra el componente indepen-

diente *USB4allAPIWrapper* sobre el que se apoyan tanto las aplicaciones Java que interactúan directamente con las operaciones de la *USB4all API* como la *USB4all Library*. Sobre el modelo de objetos de la *USB4all Library* se construyen aplicaciones Java que facilitan toda la interacción con los dispositivos electrónicos conectados al *baseboard*.

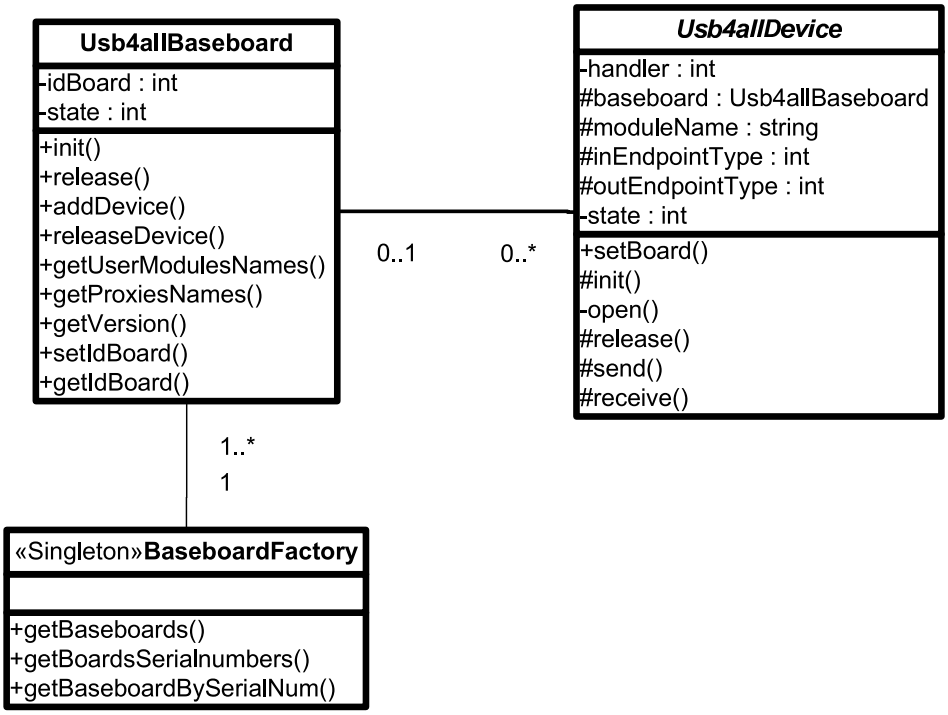


Figura 5.11: Diagrama de clases de la librería orientada a objetos

En la figura 5.11 se muestra en detalle las tres clases fundamentales, sobre las cuales es posible edificar aplicaciones de manera intuitiva, guiada y sencilla. Estas clases son: *BaseboardFactory*, *Usb4allBaseboard* y *USB4allDevice* y representan una abstracción de los conceptos fundamentales del *USB4all System*. La clase *USB4allBaseboard* es el representante en software de la pieza de hardware *baseboard* y encapsula las funcionalidades de inicialización, y obtención de información del mismo. Este objeto implementa una máquina de estados que fija las operaciones que pueden realizase en cada instante. Es muy simple y sólo posee dos estados que se ilustran en la figura 5.12 a la izquierda. Es importante destacar que sólo luego de la operación `init` el objeto en software queda ligado con un *baseboard* y recién en ese instante todas las operaciones subsiguientes en el objeto impactan el estado del hardware. Otra característica importante es que al efectuar alguna operación no permitida en la máquina de estados o cualquier otro error contemplado en la arquitectura (no existencia de *user module*, tipos transferencia inexistente, etc) se utiliza el mecanismo de lanzamiento de excepciones para notificar a los invocantes.

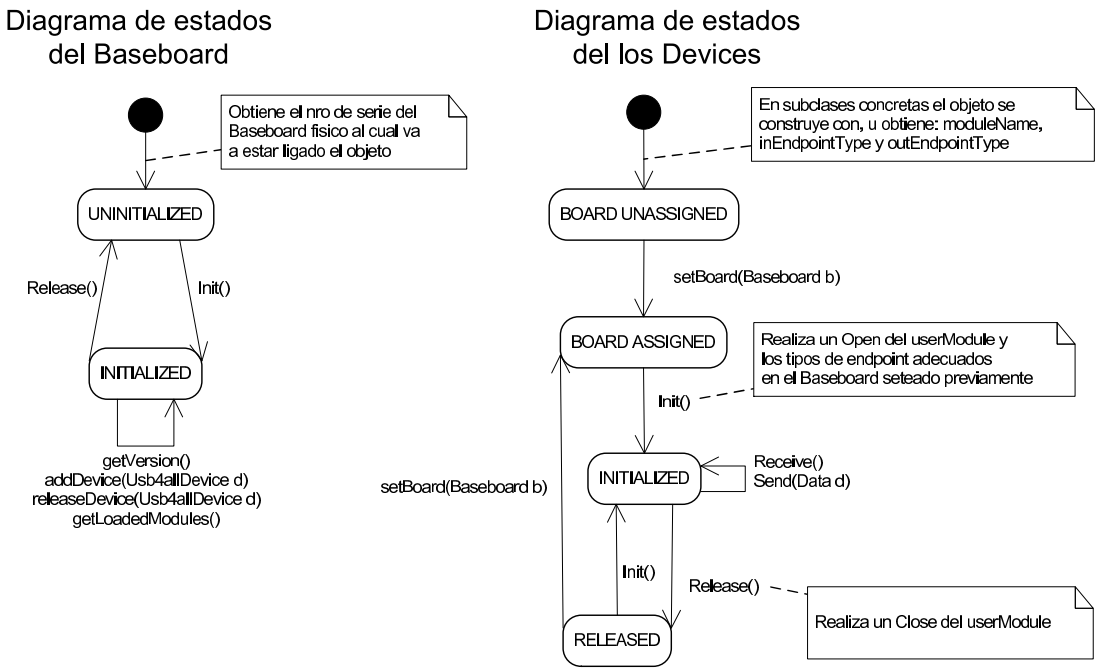


Figura 5.12: Diagrama de estados que ocurren dentro de las instancias de baseboard y device

A continuación se muestra un cuadro con una breve descripción de las operaciones de esta clase:

Operación	**Descripción**
`init`	Inicializa el *baseboard* y realiza el cierre de todos los *user modules* abiertos.
`release`	Libera los recursos utilizados, realiza el cierre de todos los *user modules* abiertos y desasocia todos los objetos *USB4allDevices*, devolviéndolos a su estado inicial.
`addDevice`	Asocia un device a este *USB4allBaseboard*.
`releaseDevice`	Desasocia un *USB4allDevice* del *USB4allBaseboard*.
`getUserModuleNames`	Obtiene una lista de los nombres de los *user modules* existentes en este *baseboard*.
`getProxiesNames`	Obtiene una lista de los nombres de los *proxies* existentes en este *baseboard*.
`getVersion`	Obtiene la versión del *base firmware* de este *baseboard*.
`setIdBoard`	Setea el número de serie del *baseboard* a utilizar.
`getIdBoard`	Obtiene el número de serie del *baseboard* a utilizar.

Cuadro 5.12: Operaciones de la clase *USB4allBaseboard*.

La otra clase importante es el *USB4allDevice* que es el representante del dispositivo conectado al *baseboard* y encapsula la interacción con el *user module* específico para esta tarea. Esta clase es abstracta, y las clases que la extienden son las que definen los datos particulares a utilizar, como lo son el tipo de transferencia y nombre del *user module* a utilizar, así como también las operaciones propias del dispositivo a interactuar. Aquí el trabajo del desarrollador se concentra principalmente en implementar el *user protocol*, utilizando para ello las operaciones `send` y `receive` de la clase padre. La clase *USB4allDevice* también implementa una máquina de estados la cual se ilustrada en la figura 5.12 en la parte derecha, de forma de ordenar su utilización y también utiliza el mecanismo de excepciones para notificar cualquier condición de error. En la tabla 5.13 se muestran las operaciones de esta clase junto con una breve descripción:

Operación	Descripción
`setBoard`	Setea el *USB4allBaseboard* al cual asociarse (con el baseboard en el cual se quiere comunicar).
`init`	Realiza la apertura del *user module* específico para este dispositivo.
`release`	Realiza el `cierre` del *user module*.
`send`	Envía datos al *user module*.
`receive`	Recibe datos del *user module*.

Cuadro 5.13: Operaciones de la clase abstracta *USB4allDevice*

Finalmente se encuentra la clase *BaseBoardFactory* que implementa el patrón de diseño Singleton y es utilizado para facilitar la instanciación e inicialización de los *USB4allBaseboards* y sus operaciones se muestran en la tabla 5.14.

Operación	Descripción
`getBaseboards`	Devuelve una colección de los *USB4allBaseboard* correspondientes a cada uno de los *baseboards* conectados al PC.
`getBoardsSerialNumbers`	Devuelve una colección de los números de serie de los *baseboards* conectados al PC.
`getBaseboardBySerialNum`	Devuelve una instancia del *USB4allBaseboard* correspondiente a un número de serie dado.

Cuadro 5.14: Operaciones de la clase abstracta *USB4allDevice*

Las tres clases anteriores constituyen la base de la *USB4allLibrary* sobre la cual se puede ir extendiendo la biblioteca. Como parte de este proyecto se crearon un conjunto de clases de dispositivos específicos: un sensor de temperatura, un motor paso a paso y un display 7 segmentos de forma de dar ejemplos concretos de implementación. En la figura 5.13 se muestra un diagrama de secuencia típico para el uso de un dispositivo desde la aplicación de usuario.

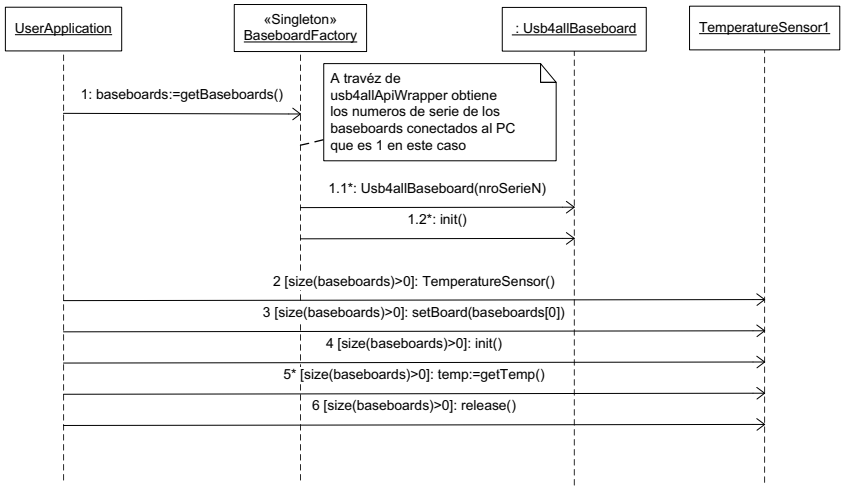


Figura 5.13: Diagrama de secuencia de uso simple de la library

La clase *TemperatureSensor* extiende la clase abstracta *USB4allDevice* y le agrega la función `getTemp` que devuelve la temperatura obtenida por un sensor de temperatura.

Para implementar esta función se utilizan los `send` y `receive` de la clase padre, y con ellos implementa el protocolo de comunicación con el *user module* correspondiente a dicho sensor en el *baseboard*.

Lo primero que debe realizar la aplicación de usuario, es obtener los *baseboards* conectados al PC, y para ello invoca la función `getBoards`. Esta operación está implementada en el *BaseboardFactory* e interactúa con la *USB4all API* (a través del *USB4allApiWrapper*) para obtener cada uno de los *baseboards* conectados, e instanciarlos adecuadamente para poder devolver una colección de objetos de la clase *USB4allBaseboards*. Luego con el número de secuencia de 2 y si la colección devuelta de *USB4allBaseboards* no es vacía, la aplicación crea un objeto de la clase *TemperatureSensor*. Se recuerda que en la clase *TemperatureSensor* están definidos los tipos de transferencia, así como el nombre del *user module* a utilizar por defecto. En los pasos 3 y 4 se envía el *USB4allBaseboard* en el cual se utilizará el *user module* correspondiente al sensor de temperatura, y se inicializa el mismo, mediante las operaciones `setBoard` e `init` respectivamente. A partir de este momento se encuentra todo preparado como para poder obtener las veces que sea necesaria la temperatura mediante la operación `getTemp`. Finalmente cuando no se desee utilizar más el sensor de temperatura, la aplicación invoca la operación `release` que es implementada en la clase abstracta *USB4allDevice* y realiza el cierre del *handler* asociado en la *USB4all API*.

5.4. Generic Driver USB

El driver desarrollado es genérico, esto implica que no está diseñado para ninguna de las clases de la jerarquía específica de dispositivos USB, sino que tiene como objetivo soportar todos los tipos de transferencias (Control, Bulk, Interrupt e Isochronous) así como el máximo número de endpoints (32) en forma abierta sin implementación de lógica para un dispositivo particular.

Dado que la *USB4all API* encapsula en el PC la implementación de los protocolo de comunicación que define la solución lleva a que el driver sólo sea responsable de la comunicación e intercambio de información de las características del dispositivo (número de serie, descriptores) así como del intercambio de datos. Esta característica de la solución permite reutilizar el código que implementan los protocolos de comunicación y evita tener que implementarlos directamente en el driver, ya que son dificiles de depurar (ejecutan en modo núcleo). Además otro de los beneficios que trae el no incluir la lógica de los protocolos en los drivers, es que se pueden sustituir por distintas implementaciones o incluso cambiar la plataforma que se utiliza con un impacto mínimo en la solución.

La implementación del proyecto comenzó utilizando el driver genérico que ofrece gratuitamente el fabricante del microcontrolador (Microchip) pues se ya se había utilizado en las pruebas del mismo durante el relevamiento del estado del arte. La utilización de este driver como parte de la solución no permite satisfacer totalmente los objetivos planteados, pues sólo funciona en la plataforma Windows y no en Linux, por lo tanto se tuvo que buscar alguna otra opción para hacer funcionar el dispositivo bajo éste sistema operativo.

Una de las opciones era enumerarse como alguna clase definida para la cual Linux implemente un driver, como ser HID o CDC, pero si se toma esta opción se termina atado a determinado tipo de transferencia y cantidad de endpoints (los definidos por las interfaces) lo cual es algo que se pretende evitar dado que uno de los principales objetivos del proyecto es que la solución debe ser lo más genérica posible. La siguiente opción fue utilizar LibUSB [25] la cual parecía una muy buena opción ya que contaba con implementaciones para diferentes sistemas operativos (Linux, FreeBSD, NetBSD, OpenBSD, Darwin, MacOS X, Windows). Se realizaron pruebas con LibUSB y se lograron buenos resultados pero sólo permitía utilizar tipos de transferencias de control y bulk y no transferencias interrupt e isochronous ni tampoco transferencias asincrónicas. Dado que la última versión estable es de hace varios años, nos hizo suponer que el proyecto había sido abandonado, por lo tanto se decidió investigar la posibilidad de realizar un driver propio el cual fue desarrollado como un módulo del kernel de Linux.

Para la realización del driver se tomaron las siguientes decisiones:

- **Un file (/dev/usb4all<instancia>) por *baseboard* conectado:** La otra opción manejada fue tener un file por endpoint el cual simplificaba las transferencias a diferentes endpoints pero dificulta el intercambio de descriptores, ya que no se cuenta con un único punto de acceso. Además, para el usuario siempre sería más difícil contar la cantidad de

files existentes que pedir a un único punto de acceso toda la información del dispositivo en estructuras de datos claras, así como simplificar la interacción del driver genérico con la *USB4all API*.

- **Encapsular lo elemental en modo Kernel:** Se coloca sólo lo elemental en el módulo de kernel y se deja para programar en modo usuario (en la *USB4all API*) las tareas que pueden resolverse a partir de un conjunto de primitivas básicas exportadas por el driver. De esta manera se puede extender con mayor facilidad y permite depurar con todas las ventajas que se cuentan en el modo usuario.

El driver se comunica con las aplicaciones que corren en modo usuario para el intercambio de información del dispositivo mediante llamadas IOCTL [9], las cuales implementan un conjunto de primitivas descriptas en la tabla 5.15.

Operación	**Descripción**
GET_DEVICE_DESCRIPTOR	Devuelve el descriptor de dispositivo.
GET_ENDPOINT_DESCRIPTOR	Devuelve el descriptor de endpoint.
GET_INTERFACE_DESCRIPTOR	Devuelve el descriptor de interfaz.
GET_CONFIGURATION_DESCRIPTOR	Devuelve el descriptor de configuración.
GET_STRING_DESCRIPTOR	Devuelve el string descriptor.
SET_DESC_INDEX	Setea un índice utilizado para obtener un descriptor determinado de los tipos que poseen F de uno por tipo.
SET_IN_ENDPOINT	Setea la dirección del endpoint in a utilizar.
SET_OUT_ENDPOINT	Setea la dirección del endpoint out a utilizar.
SET_TRANSFER_TYPE	Setea un tipo de transferencia a utilizar.
SET_TIMEOUT	Setea un tiempo de espera máximo para la llegada de los datos.

Cuadro 5.15: Interfaz para el intercambio de configuración entre el driver y la aplicación.

El conjunto de operaciones del tipo *get* permiten obtener los descriptores del dispositivo, de esta manera la *USB4all API* puede contar con toda la información necesaria para decidir que tipo de endpoint utilizar o la información de la instancia particular como por ejemplo el número de serie. Por medio de las operaciones de tipo *set* se cuenta con un mecanismo para poder configurar el endpoint a utilizar, tipo de transferencia y tiempo de espera (timeout).

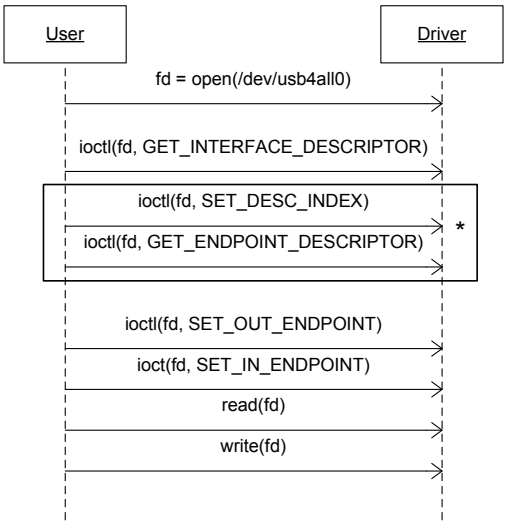


Figura 5.14: Ejemplo de apertura y envió de datos por un endpoint.

Se implementó siguiendo el framework para desarrollo de drivers USB que el kernel implementa [9, 19]. Como podemos ver en la figura 5.14 el primer paso para interactuar con el driver es realizar una llamada open sobre el file definido para el *baseboard*, luego se obtiene el descriptor de interfaz asociado al dispositivo *baseboard* con esa información se determina cuantos endpoints se poseen y se puede pasar a obtener sus descriptores como se muestra en el recuadro de la figura. Una vez obtenidos los descriptores de endpoints se cuenta con la información para poder setear que endpoint de entrada y salida se va a utilizar. Finalmente el driver se encuentra en un estado donde es posible comenzar a escribir y leer del endpoint seteado mediante las llamadas al sistema read y write. Existen variantes de este caso de uso donde se obtienen otros descriptores como el de device para determinar el número de serie del *baseboard* por ejemplo.

Capítulo 6

Comunicación

En este capítulo se explica en detalle los protocolos de comunicación que define al arquitectura de software y la interacción entre los componentes del *USB4all API* y *USB4all firmware*. Se busca la comprensión por parte del lector de como son los pasos que suceden internamente en la solución al momento de ejecutar las operaciones que brinda la *USB4all API* a las aplicaciones.

6.1. Protocolo de Comunicación

Se definió un stack de protocolos organizado según las capas que intervienen en la comunicación entre el *USB4all API* y el *USB4all firmware*, dichas capas pueden verse dividiendo a los componentes de cada subsistema mediante lineas horizontales en la figura 6.1. La capa inferior llamada *Medium access layer* implementa el acceso al medio utilizando el protocolo USB para comunicarse, esta capa implementa el medio por el cual se comunica físicamente el PC con el *USB4all*.

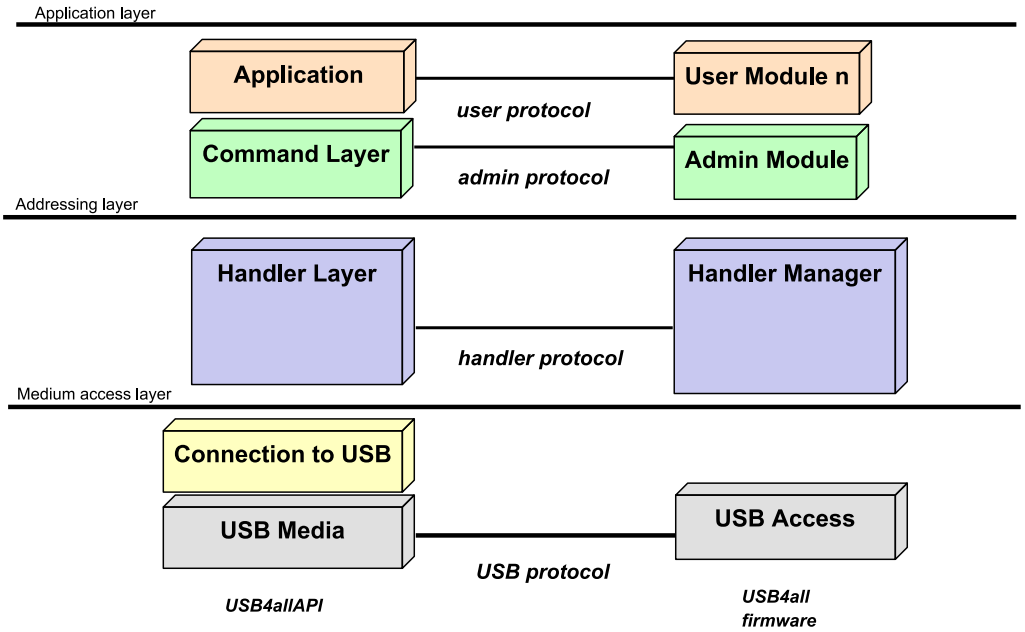


Figura 6.1: Stack de protocolos definido y capas

La capa *Addressing layer* se encarga de aspectos de direccionamiento de la información dirigida hacia los *user modules* permitiendo que la información enviada por una aplicación con destino un *user module* pueda llegar a éste. Esta tarea es de suma importancia ya que los *user modules* son los destinatarios finales de la información y como comentamos anteriormente en la sección

4.2.3, generalmente hay varios *user modules* simultaneamente ejecutando en el *baseboard*. Por lo tanto se debe contar con un mecanismo que permita encaminar correctamente a los paquetes que se envían desde el PC. El protocolo que implementa esta clase fue llamado *handler protocol* y es el que permite la comunicación entre el *handler layer* y el *handler manager,* su payload son paquetes definidos por uno de los protocolos de la capa *Application layer (user protocol* ó *admin protocol)* dependiendo cual sea el destinatario de la información. Esto es transparente para el protocolo, ya que el *admin module* fue implementado como un *user module*, por más información acerca de esta decisión puede consultarse la sección 7.2.

Luego tenemos la capa *Application layer*, la cual define dos protocolos, el *user protocol* que sirve para comunicar la aplicación con los *user modules*, siendo un protocolo abierto[1] para que el usuario defina como va a ser la interacción entre la aplicación y el *user module* que controla un dispositivo o característica del hardware. El *admin protocol* en cambio tiene como finalidad realizar la gestión de los *user modules*. Cualquiera de estos dos protocolos es el que viaja como payload del protocolo definido por el *handler protocol*, como puede verse en la figura 6.2.

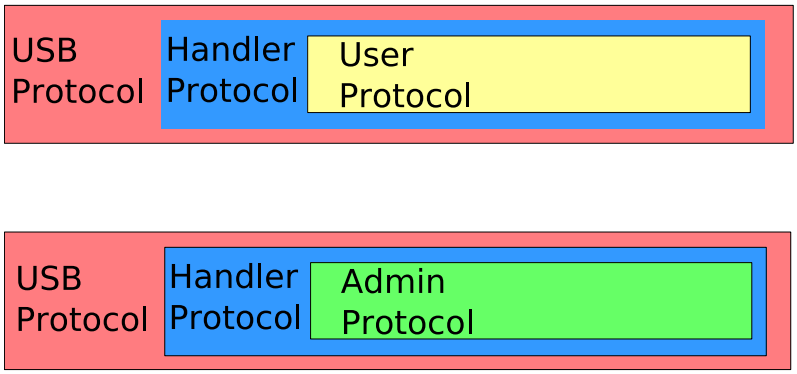


Figura 6.2: Paquetes utilizados por el stack de protocolos

A continuación se detallan en profundidad cada uno de los protocolos nombrados y los tipos de paquetes que intercambian.

6.1.1. Handler Protocol

El *handler protocol* comunica las capas *handler layer* en el *USB4all API* y *handler manager* en el *USB4all firmware,* su objetivo es brindar mecanismos que permiten comunicar una aplicación con un *user module.* Habilita enviar datos y configurar un *user module.* Utiliza el concepto de *handler number* para poder identificar a que *user module* se está enviando los datos. En la figura 6.3, puede verse el paquete definido para el intercambio de datos, su tamaño es de 64 bytes como máximo debido a la decisión en la implementación tomada en este proyecto, por más detalle ir a la sección 7.1.

El primer campo que define es el *opType*, el cual determina que tipo de operación se quiere ejecutar, los posibles valores que pueden tomarse son `send` y `config`. La responsabilidad del comando `send` es que el *user module* reciba el payload del paquete. El comando `config` se utiliza para enviar información al *user module* que éste espera como parámetros de configuración. El segundo campo *handler number* identifica al *user module* destinatario del paquete. El tercer campo *pLength*, específica el largo del paquete enviado, éste incluye todos los campos del paquete en el cálculo. El cuarto campo *reserved,* es reservado para uso futuro. Por último el campo *payload* encapsula el paquete correspondiente al *admin protocol* o al *user protocol.*

[1] Se entiende por protocolo abierto en este contexto, a un protocolo que a priori no impone ninguna estructura y que el usuario es libre de implementarla para comunicar las aplicaciones de usuario y los *user modules.*

Figura 6.3: Formato de paquete para intercambio de datos utilizado por el *handler protocol*.

6.1.2. Admin Protocol

El *admin protocol* es utilizado para tareas de gestión de los *user modules*, las cuales permiten abrir, cerrar, y listar los módulos existentes en el *USB4all firmware*. Este protocolo es intercambiado por las capas *command layer* y *admin module* y pertenece a la capa *application layer* de la jerarquía de capas definidas para la comunicación.

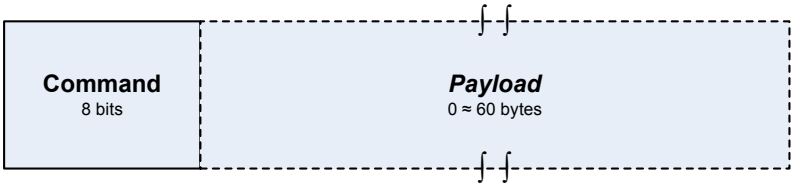


Figura 6.4: Formato de paquete para el intercambio de datos utilizado por el *admin protocol*

Como se observa en la figura 6.4, los paquetes intercambiados constan de un campo *command* de un byte y un *payload* variable de como máximo 60 bytes, a continuación se describe cada uno de los posibles comandos que pueden especificarse y sus correspondientes argumentos que viajan en el payload del paquete.

6.1.2.1. OPEN

Un pedido de open para un *user module* se específica mediante un paquete como el que se ilustra en la figura 6.5, el cual es definido por el *admin protocol*. Dicho paquete está compuesto por el identificador del comando OPEN en el primer campo, luego se encuentra el campo `inEP` que especifica el endpoint que va a utilizar la *API* para recibir los datos y el campo `outEP` para el endpoint que va a utilizar para enviar datos. Por último se encuentra el campo `moduleName` que especifica el nombre del *user module* que se desea abrir.

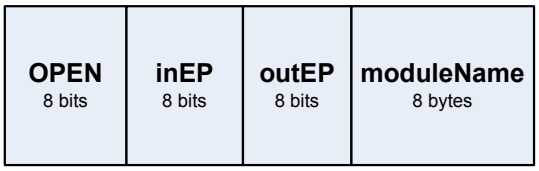


Figura 6.5: Paquete de pedido del comando `open`

Una respuesta de OPEN para un *user module* se realiza mediante un paquete como el que se ilustra en la figura 6.6, el cual es especificado por el *admin protocol*. Dicho paquete está compuesto por el identificador del comando OPEN en el primer campo y el *handler number* asignado al *user module* en el segundo campo, en caso de que el *user module* no exista o ya esté abierto se devuelve la constante `ERROR`.

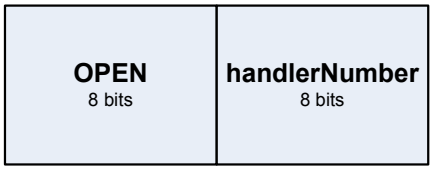


Figura 6.6: Paquete de respuesta del comando `open`

6.1.2.2. CLOSE

Cuando se quiere que el *base firmware* cierre un determinado *user module* se envía un paquete de pedido de CLOSE como se ilustra en la figura 6.7, el cual es especificado por el *admin protocol*. Dicho paquete está compuesto por el identificador del comando CLOSE en el primer campo y el *handler number* del *user module* a cerrar en el segundo.

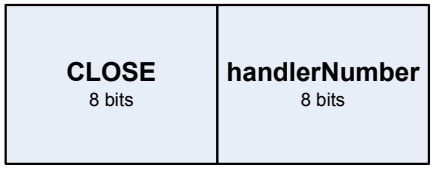


Figura 6.7: Paquete de pedido del comando `close`

La respuesta a este comando se especifica con un paquete de respuesta de CLOSE como el que se muestra en la figura 6.8, el cual está compuesto por el identificador del comando CLOSE en el primer campo y una constante que representa el resultado del comando en el segundo campo. La constante toma el valor `ACK` en caso de éxito o `NACK` en caso de que el *user module* no exista o ya este cerrado.

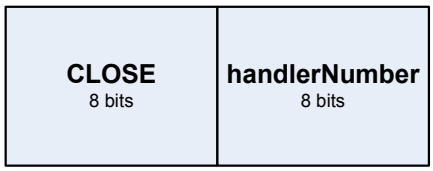


Figura 6.8: Paquete de respuesta del comando `close`

6.1.2.3. GET_USER_MODULES_SIZE

Su objetivo es obtener la cantidad de *user modules* cargados en un *basboard*, junto con el GET_USER_MODULES_LINE implementan el mecanismo por el cual el usuario puede obtener los nombres de los *user modules* cargados en el *USB4all firmware* de un *baseboard* en particular. Esto es muy útil ya que a partir de esta información es que se puede invocar al comando OPEN.

Para hacer el pedido se envía un paquete como el que muestra la figura 6.9, el cual especifica simplemente el deseo de conocer cuantos *user modules* hay cargados mediante el campo identificador del comando.

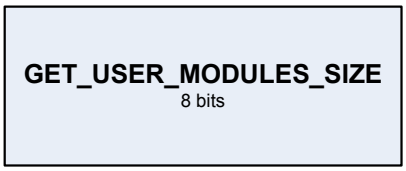


Figura 6.9: Paquete de pedido del comando `GET_USER_MODULES_SIZE`

La respuesta contiene solamente un campo con el identificador del comando como es usual y otro con la cantidad de *user modules* cargados en el *USB4all firmware.*

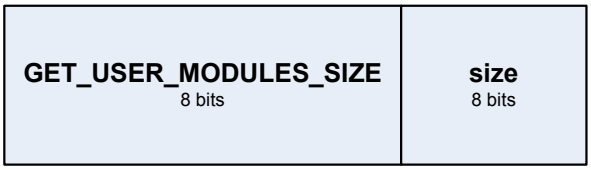


Figura 6.10: Paquete de respuesta del comando `GET_USER_MODULES_SIZE`

6.1.2.4. GET_USER_MODULES_LINE

Una vez obtenido la cantidad de *user modules* cargados en el *USB4all firmware*, se está en condiciones de poder invocar a este comando para determinar el nombre de cada *user module* cargado. El funcionamiento es el siguiente: para realizar el pedido se envía un paquete conteniendo el identificador del comando seguido de un campo que especifica el número de fila en la tabla comentada en la figura 4.6 como puede verse en la figura 6.11. El número obtenido mediante el comando anterior es útil para determinar cuantas son las filas que posee la estructura que almacena la información de un *user module* e ir pidiendo de a una sin excederse en la cantidad que posee la tabla.

Se decidió hacerlo de esta manera, en lugar de contar con una sola operación que devuelva la totalidad de los nombres presentes, para lograr simplificar la programación en el *USB4all firmware* debido a que es más sencillo de implementar en la *API* a partir de estas dos primitivas.

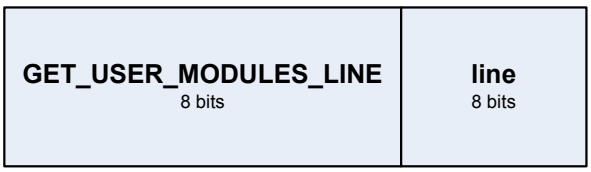


Figura 6.11: Paquete de pedido del comando `GET_USER_MODULES_LINE`

La respuesta, como muestra la figura 6.12, consta del campo identificador del comando y otro campo con un string de ocho caracteres conteniendo el nombre del *user module* correspondiente al identificador pasado.

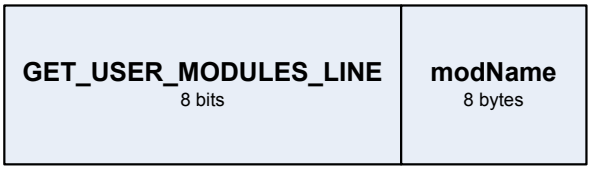


Figura 6.12: Paquete de respuesta del comando `GET_USER_MODULES_LINE`

6.1.2.5. CLOSE_ALL

Este comando simplemente sirve para que el *USB4all firmware* cierre todos los *user modules* que estan abiertos. Es útil para liberar los recursos al momento de terminar de trabajar con la *baseboard* o el dispositivo a controlar. El paquete es muy sencillo y tanto el pedido como la de respuesta constan sólo del identificador del comando.

6.1.2.6. GET_VERSION

Este comando es utilizado para obtener el número de versión del *USB4all firmware,* lo cual fue útil a la hora de realizar el testing debido a que dicho número de versión se corresponde con un tag en el sistema de control de versiones (concurrent versions system)(CVS), lo que permite tener un mejor seguimiento del firmware una vez grabado en el microcontrolador.

El paquete que se intercambia para una solicitud, consta solamente del identificador del comando correspondiente y el de respuesta además del identificador tiene un campo con el número de versión.

6.2. Interacción USB4all API ⇔ USB4all Firmware

La *USB4all API* interactúa con el *USB4all firmware* para poder brindarle al usuario el servicio de enviar y recibir datos hacia o desde un *user module* (como vemos en la figura 6.13). Esto es de suma importancia ya que abstrae al usuario de la tecnología utilizada como medio físico (en este caso USB), permitiéndole concentrarse en la interacción con el dispositivo a controlar. Todo esto se realiza en un entorno donde dinámicamente se pueden ir instanciando *user modules* que van a estar activos en simultáneo.

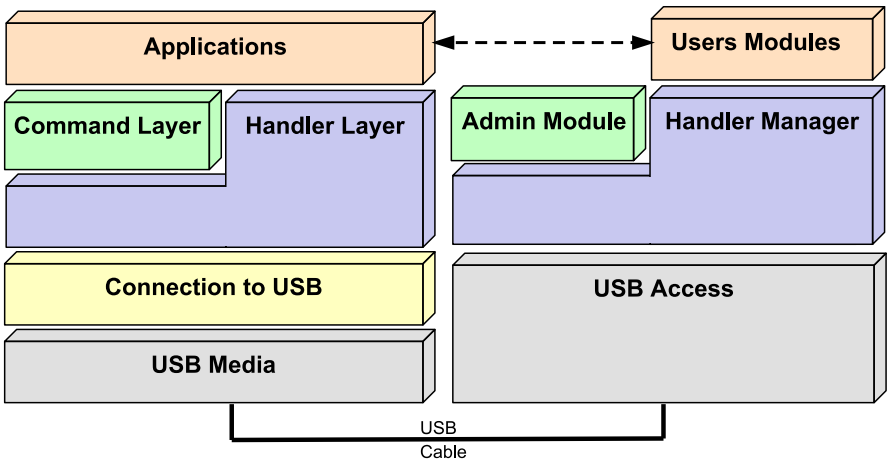


Figura 6.13: Interacción API - Firmware

El permitir que los *user modules* se instancien dinámicamente nos brinda la posibilidad de gestionar más eficientemente los recursos del microcontrolador del *baseboard*, ya que si un *user module* no es abierto, éste no utiliza tiempo de CPU ni los recursos que tenga asignados, salvo la memoria de programa.

La interfaz que brinda la *API* al usuario es similar a la que brindan los sistemas operativos para el manejo de archivos, la aplicación comienza creando los vínculos lógicos con los *user modules* (OPEN) que luego le van a permitir enviar y recibir datos. Los mensajes de OPEN son delegados a la capa *command layer* en el PC la cual es la encargada de generar los paquetes necesarios para la capa *admin module* en el *USB4all firmware*, lo mismo ocurre con los demás mensajes que esta capa maneja, como son el CLOSE y de listado de *user modules* presentes en los

baseboards. En el caso de que la aplicación envie datos a un *user module*, estos mensajes van a ser delegados a la capa *handler layer* en la *API*, donde se van a generar los paquetes necesarios para su contraparte en el USB4all firmware (*handler manager*), estos paquetes van a contener la información necesaria para identificar a que *user module* es dirigido el paquete. Cabe destacar que los paquetes dirigidos al *admin module* también son direccionados por el *handler manager* al igual que lo hace para los paquetes destinados a los *user modules*. Por eso, es que podemos pensar a la capa *admin module* como la encargada de administrar la tabla de direccionamiento de *user modules*, que es utilizada por la capa *handler manager* para poder enviar los datos al *user module* adecuado.

A continuación se presentan los diagramas de secuencia de las operaciones *openDevice*, *closeDevice*, *sendData* y *receiveData* que expone la *USB4all API* a las aplicaciones de usuario, para poder explicar en forma detallada la sucesión de pasos que se realizan durante la invocación de estas operaciones, de forma de brindar una visión más adecuada del funcionamiento interno de toda la solución *USB4ll*.

6.2.1. Operación `openDevice`

El diagrama de secuencia de la figura 6.14, muestra la cadena (simplificada) de invocaciones y pasos que se suceden cuando una aplicación de usuario invoca la función `openDevice` del *u4aAPI* para abrir un vínculo lógico con un *user module*. Esta invocación se propaga hasta llegar al *commandlayer*, el cual, en primera instancia consulta al *handlerlayer* (a su vez, éste consulta al *descriptorlayer*) para la obtención de los números de endpoints a utilizar para el envío y recepción de datos (los tipos de transferencias son parámetros de la operación `open` del *commandlayer*). Luego se construye un paquete del *admin protocol* y se invoca la operación `send` del *handlerlayer*, éste construye un paquete del *handler protocol* y carga como destinatario al *admin module* (*handler* 0). El paquete construido es pasado al *descriptorlayer*, el cual invoca la operación `send` apropiada para el tipo de transferencia que utiliza el *admin module*. Finalmente el *driverlayer* envía el paquete al driver genérico USB que este utilizando y éste lo envía por el bus USB. Luego del envío del paquete, el flujo de ejecución vuelve al *commandlayer* para invocar la operación `receive` del *handlerlayer*. Esta invocación se propaga hasta llegar al *driverlayer* en la que dependiendo del timeout indicado como parámetro se queda esperando la respuesta del éxito de la apertura.

Mientras tanto en el *baseboard*, el *main loop* invoca periódicamente la operación `USBRead` del *handler manager*, para verificar si llegaron nuevos mensajes. Cuando llega el mensaje de apertura del *user module* enviado desde el PC, el *handler manager* lo procesa para obtener su payload, verifica quién es el *user module* destinatario (en éste caso el *admin module*) e invoca su operación `receive`. El *admin module* procede a obtener el payload del paquete, verifica si el nombre del *user module* indicado en el mismo está registrado en el *base firmware* (`getUserTableDirection` y `existsTableEntry`). Luego pide al *loader module* la dirección de memoria (`getModuleInitDirection`) de la operación `init` del módulo, pide al *handler manager* que realice el alta del *user module* en la tabla de módulos activos (`newHandlerTableEntry`), obteniendo como resultado el *handler* del mismo. Finalmente invoca a la operación `init` del *user module*, la cual almacena el *handler* que lo identifica, le indica al *handler manager* la función del módulo que debe llamar, para que el *user module* pueda recepcionar los mensajes enviados desde el PC (`getHandlerReceiveFunction`) y finalmente, obtiene la dirección del buffer de envío hacia el PC (`getSharedBuffer`).

Al volver al *admin module*, se crea un paquete del *admin protocol* con la respuesta de la apertura y se envía al *handler manager* (`USBWrite`), éste construye el paquete del *handler protocol* y lo envía al bus USB. Al llegar el mensaje al PC, el driver genérico USB devuelve el mensaje al *driverlayer* que esperaba la respuesta del *baseboard*. Una vez que el flujo de ejecución vuelve al *handlerlayer*, este obtiene el payload del paquete y lo devuelve al *commandlayer* que determina si la operación de apertura del *user module* fue exitosa o no. Si fue exitosa, el mensaje recibido posee el *handler* asignado por el *handler manager* al *user module* abierto. Finalmente, el *commandlayer* registra toda la información relacionada al *user module* (`registerModule`) en el *handlerlayer*, construye el *moduleID* y lo devuelve a la aplicación de usuario.

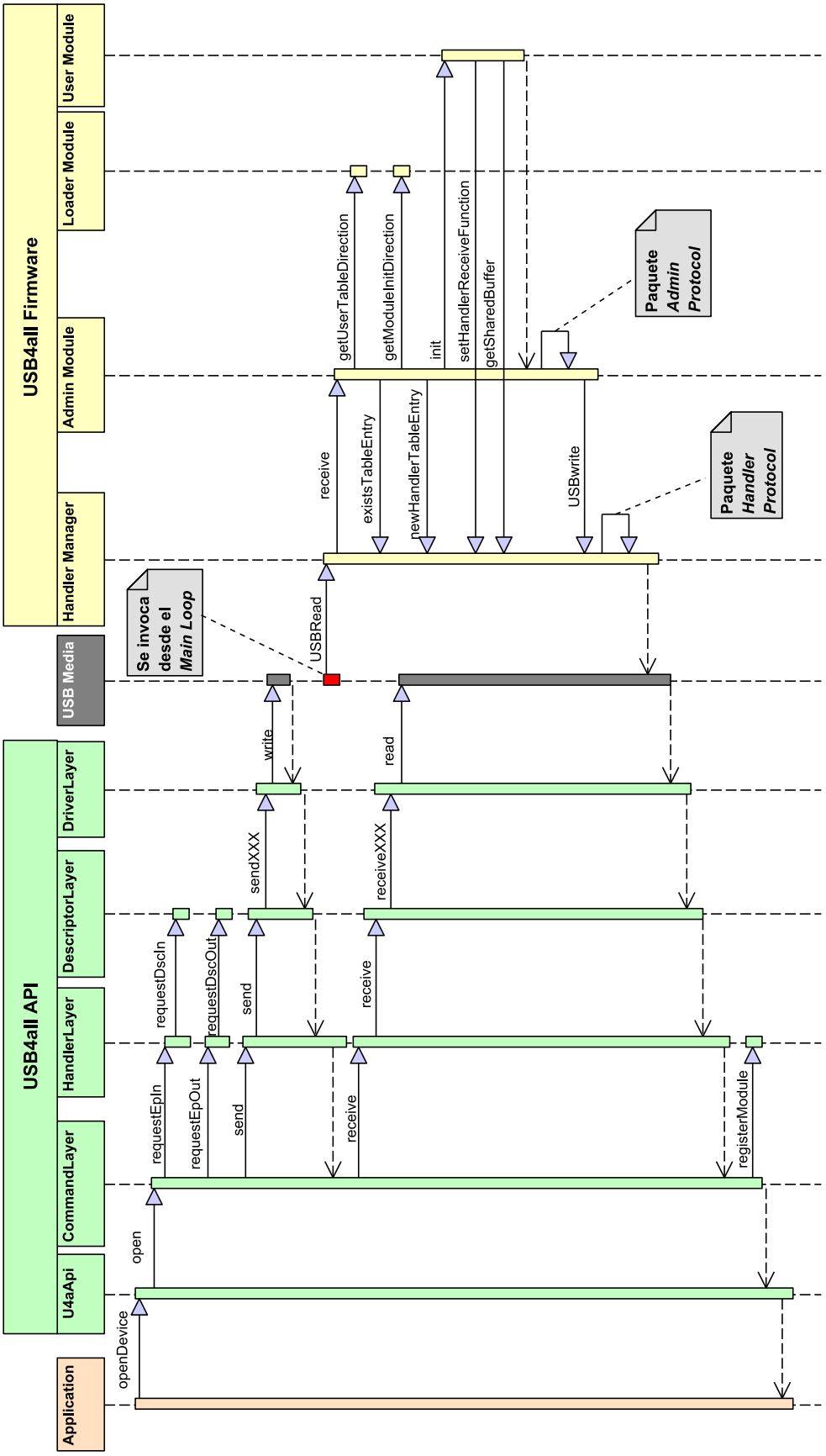


Figura 6.14: Diagrama de secuencia de la operación *openDevice*

6.2.2. Operaciones `sendData` y `receiveData`

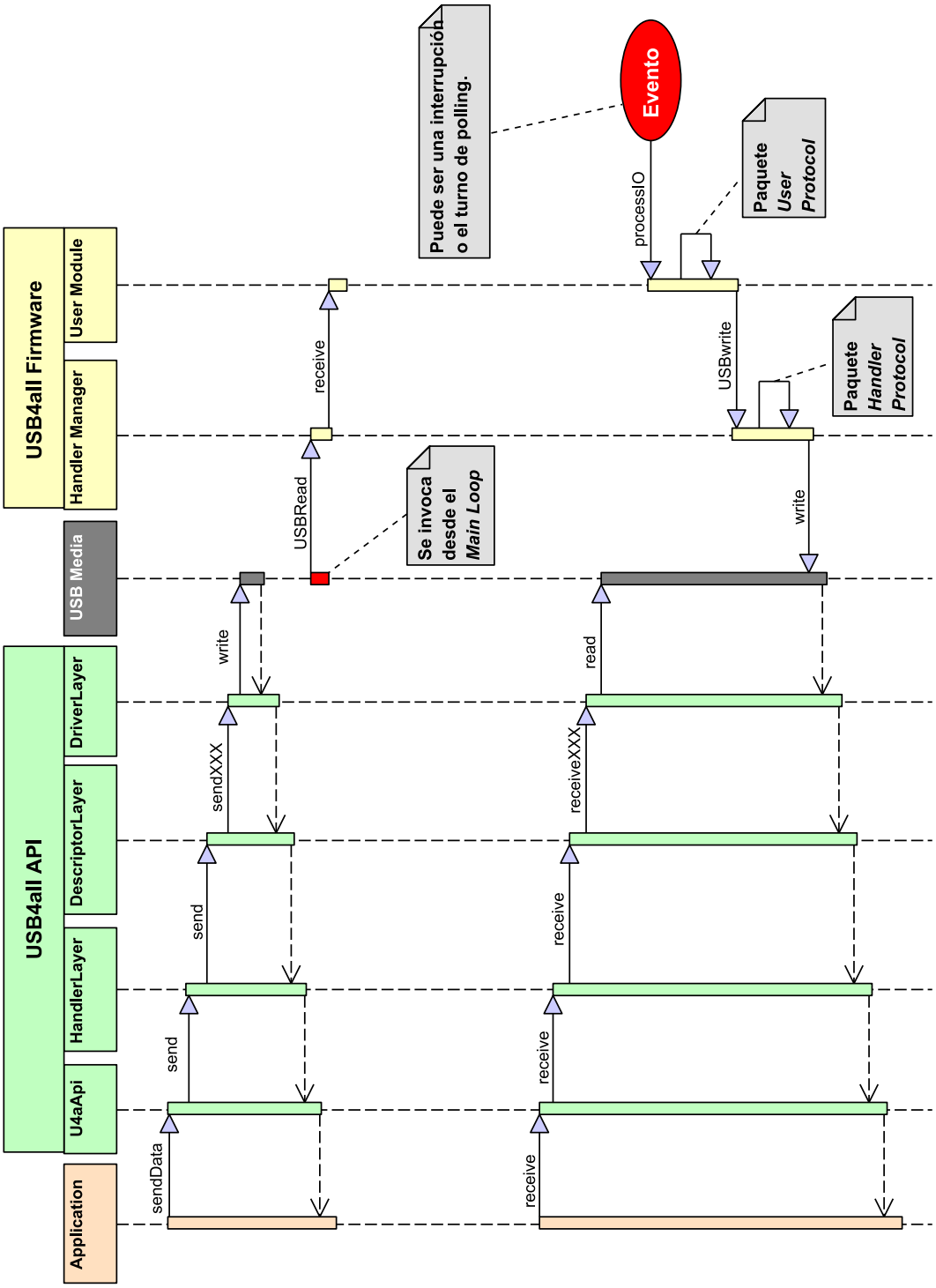


Figura 6.15: Diagrama de secuencia de la operación *sendData* y *receiveData*

El diagrama de secuencia de la figura 6.15, muestra la cadena (simplificada) de invocaciones y pasos que se suceden cuando una aplicación de usuario invoca la funciones `sendData` y `receiveData` del *u4aAPI* para enviar y recibir información desde y hacia los *user module*. Primero se observa la invocación de la operación `sendData` por parte del programa de aplicación, esta invocación se propaga hasta el *handlerlayer* donde se construye el paquete del *handler protocol*,

luego es enviado al *descriptorlayer* (`send`), de ahí al *driverlayer* y finalmente el driver genérico USB, que lo envía por el bus USB. Luego de realizado esto, se devuelve la respuesta de envío exitoso o no al programa de aplicación. Algún momento después el *main loop* del *base firmware* ejecuta la función `USBRead` del *handler manager* y encuentra un mensaje nuevo. Obtiene el payload y el *handler* del paquete para identificar el *user module* destinatario y una vez identificado invoca la operación `receive` (que corresponde a la función que registró el *user module* en el momento de la apertura) y termina.

Más abajo en el diagrama de secuencia, aparece la invocación de la primitiva `receiveData` del *u4aAPI*, pues se quiere recibir un mensaje proveniente del *baseboard*. Esta invocación atraviesa los distintos componentes de la *USB4all API* hasta llegar al *driverlayer* donde queda esperando (según el timeout indicado como parámetro) hasta que llegue un mensaje. En algún momento después, sucede un evento (polling o interrupción) en un dispositivo o en el *baseboard* mismo, que genera la invocación de la operación `processIO` de un *user module*. Éste ejecuta su lógica específica, genera un paquete del *user protocol* con la información que el módulo desea enviar al PC y se lo pasa *handler manager* (`USBWrite`) que construye un paquete del *handler protocol* y lo envía al bus USB. Cuando el driver genérico USB detecta que llegó el mensaje se lo pasa al *driverlayer*, que se había quedado esperando algún mensaje del *baseboard*. Luego ese mensaje pasa el *handlerlayer*, donde se obtiene el payload con la información enviada por el *baseboard* y finalmente se devuelve dicho payload al programa de aplicación.

6.2.3. Operación `closeDevice`

El diagrama de secuencia de la figura 6.16, muestra la cadena (simplificada) de invocaciones y pasos que se suceden cuando una aplicación de usuario invoca la función `closeDevice` del *u4aAPI* para cerrar un vínculo lógico con un *user module*. Esta invocación se propaga hasta llegar al *commandlayer*, el cual, en primera instancia consulta al *handlerlayer* si el *user module* que se quiere cerrar se está usando (`existsModule`). Luego construye un paquete del *admin protocol* e invoca la operación `send` del *handlerlayer*, éste construye un paquete del *handler protocol* y carga como destinatario al *admin module* (*handler* 0). El paquete construido es pasado al *descriptorlayer*, el cual invoca la operación de `send` apropiada para el tipo de transferencia que utiliza el *admin module*. Finalmente el *driverlayer* envía el paquete al driver genérico USB que este utilizando y éste lo envía el bus USB. Luego del envío del paquete, el flujo de ejecución vuelve al *commandlayer* para invocar la operación `receive` del *handlerlayer*. Esta invocación se propaga hasta llegar al *driverlayer* en la que dependiendo del timeout indicado como parámetro se queda esperando la respuesta del éxito del cierre.

Mientras tanto en el *baseboard*, el *main loop* invoca periódicamente la operación `USBRead` del *handler manager*, para verificar si llegaron nuevos mensajes. Cuando llega el mensaje de cierre del *user module* enviado desde el PC, el *handler manager* lo procesa para obtener su payload, verifica quién es el *user module* destinatario (en éste caso el *admin module*) e invoca su operación `receive`. El *admin module* procede a obtener el payload del paquete, le pide al *handler manager* que desregistre al *user module* identificado por el *handler* que viene en el payload de la tabla de módulos activos (`removeHandlerTableEntry`). Luego pide al *loader module* la dirección de memoria (`getModuleReleaseDirection`) de la operación `release` del módulo y finalmente la invoca.

Al volver al *admin module*, se crea un paquete del *admin protocol* con la respuesta del cierre y se envía al *handler manager* (`USBWrite`), éste construye el paquete del *handler protocol* y lo envía al bus USB. Al llegar el mensaje al PC, el driver genérico USB devuelve el mensaje al *driverlayer* que esperaba la respuesta del *baseboard*. Una vez que el flujo de ejecución vuelve al *handlerlayer*, este obtiene el payload del paquete y lo devuelve al *commandlayer* que determina si la operación de cierre del *user module* fue exitosa o no. Si fue exitosa, el *commandlayer* desregistra toda la información relacionada al *user module* (`unregisterModule`) en el *handlerlayer* y devuelve el resultado de la operación a la aplicación.

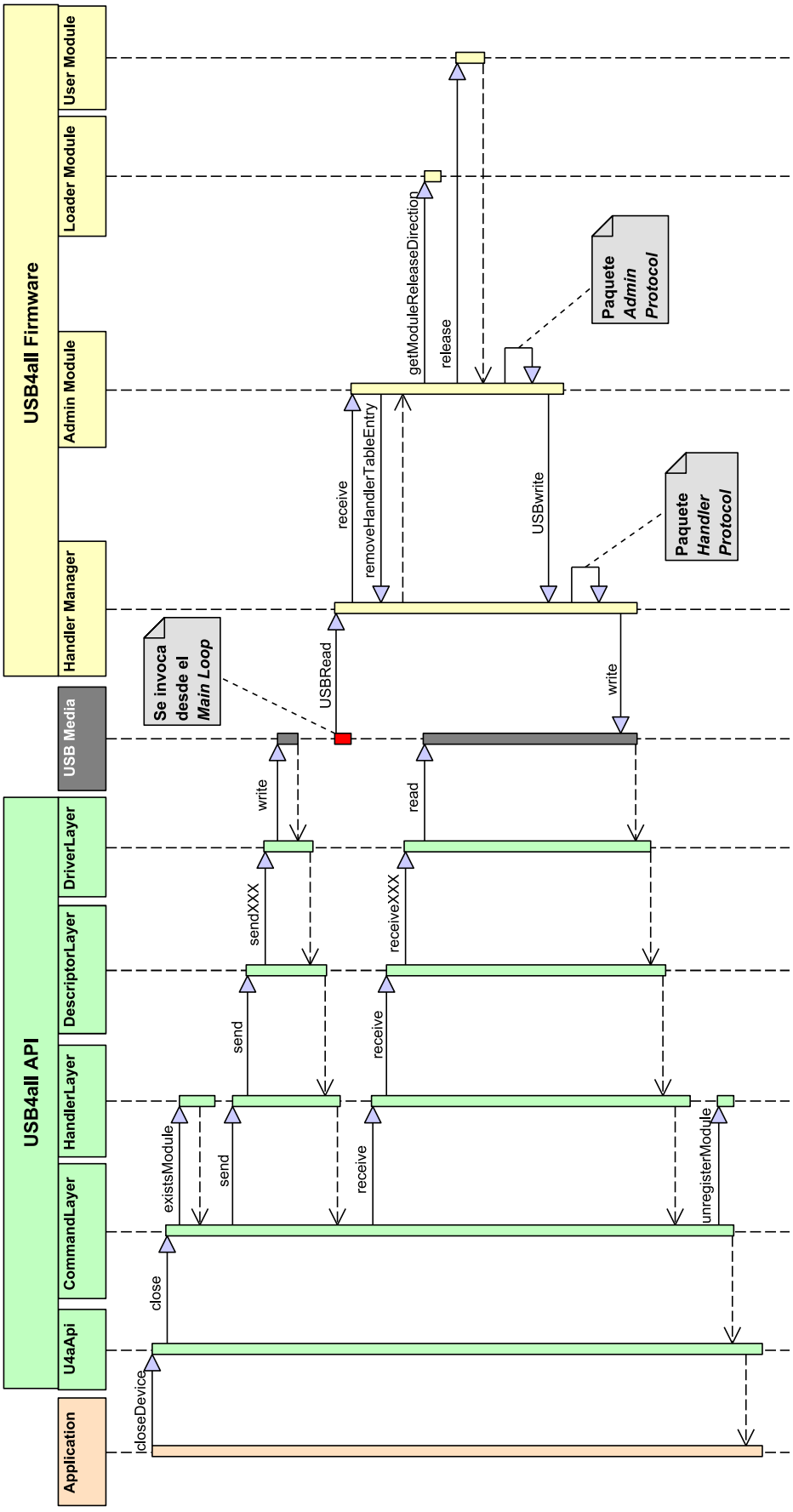


Figura 6.16: Diagrama de secuencia de la operación *closeDevice*

Capítulo 7

Decisiones de Diseño e Implementación

Introducción

El diseño y la construcción de los distintos elementos de la solución fué fruto de un proceso de desarrollo, que busco en todo momento seguir los objetivos planteados en el proyecto, trantando de respetar los distintos puntos de vista y aportes de forma de alcanzar un producto final que represente los intereses de todos los integrantes del equipo. Algunos temas fueron motivo de discusión ya que no se lograron congenear las distintas visiones y fue necesario debatir en profundidad para lograr llegar a una decisión final sobre el tema. Esta sección intenta describir algunos de estos temas polémicos, mostrando las distintas alternativas existentes y como fué que se tomaron las decisiones, las misma se presentan divididas en tres grupos: *Decisiones generales* que hacen referencia a temas que afectaban a toda la solución y finalmente *Decisiones en el firmware* y *Decisiones en el PC.*

7.1. Decisiones Generales

Modo de uso Fachada

Este modo de uso de la solución era uno de los objetivos planteados en este proyecto, la idea de fondo de su funcionamiento era poder simular el comportamiento de un dispositivo USB clasificado dentro del estándar, de forma de aprovechar los drivers específicos y todas las aplicaciones ya existentes en el PC. Un ejemplo de este modo fachada es la utilización de un conjunto de sensores de presión de forma tal que desde el PC se los reconozca como una batería MIDI, por ejemplo el golpear un sensor sería reconocido en él como un golpe de platillo.

Luego de comenzado el diseño e implementación inicial del proyecto se reconsideró su inclusión dentro del alcance del mismo, pues se observó que a pesar de que técnicamente era viable e interesante su aplicación, existían varias opciones para su realización pero demasiado costosas en cuanto al tiempo. Además no se lograba llegar a una arquitectura que permitiera unificar las dos modalidades de uso en forma armónica sino que la realización de una iba en contra sentido de las funcionalidades de la otra, como si fueran dos proyectos disjuntos. Teniendo en cuenta lo antes mencionado y que esta modalidad de uso era un objetivo secundario se decidió sacarla del alcance del proyecto y dejarla como trabajo a futuro (ver sección 9.2 para más detalles).

Adapterboard sin lógica programable

Una de las decisiones tomadas en el diseño de toda la solución fue que los *adapterboard* no contuvieran lógica programable ya que esta funcionalidad recae sobre el *baseboard* donde ejecutan los *user modules* que son los encargados de encapsular el conocimiento para la interacción con los dispositivos o en su defecto en las aplicaciones de usuario que corren en el PC. Básicamente un *adapterboard* es considerado en nuestra arquitectura como un conjunto de componentes de hardware encargados de la conexión entre los *baseboards* y los dispositivos electrónicos que

se quieren manejar. La expresión mínima que puede tener un *adapterboard* es una placa que contenga un *U4APort* y una salida que consista en la selección de un conjunto de señales para conectar al dispositivo. El escenario típico de uso es cuando existe una incompatibilidad entre las características de las señales del *baseboard* y del dispositivo a utilizar, por ejemplo diferentes voltajes, diferentes potencias, etc.

Tamaño máximo de paquetes

El microcontrolador utilizado en este proyecto soporta velocidades de trasmisión Low y Full, lo implica según el estándar USB una restricción en los tamaños máximos de datos que se pueden trasmitir para los distintos tipos de transferencias. Frente a esta limitante se pensaron distintas alternativas como definir transacciones de paquetes, de forma de lograr superar esta cota y brindar a los ojos de los usuarios la flexibilidad (sólo limitada por los recursos de memoria disponibles en el *baseboard*) en el tamaño de la información que se puede enviar o recibir. Estas ideas lentamente se fueron descartando pues introducían un grado de complejidad en el manejo de la memoria del microcontrolador para el almacenamiento temporal de los paquetes en tráfico y generaba una ineficiencia importante del uso de la memoria frente aquellos casos en que no se necesitaba transaccionar paquetes.

Todo esto llevó a tomar la decisión de definir un único tamaño máximo de paquete de 64 bytes, que cumple en el mayor grado posible con los topes máximos establecidos en el estándar USB para los cuatro tipos de transferencias, logrando evitar de esta forma introducir en los protocolos de comunicación: *handler protocol* y *admin protocol* y en lógica del *base firmware* los mecanismos necesarios para la transaccionalidad de paquetes.

Cabe destacar que a nivel del *user protocol*, la fragmentación de la información a enviar o recibir queda abierta a que los desarrolladores de *user modules* y de aplicaciones de usuario puedan definir transacciones de mensajes como parte de su interacción mutua, pues conocen de antemano los volúmenes de memoría requeridos.

7.2. Decisiones en el Firmware

Carga y descarga de user modules a demanda

Esta idea surge frente a la realidad de la programación del firmware de los microcontroladores, en los que se estila recompilar todos los códigos fuentes cada vez que se realiza un cambio en los mismos. Para nuestra arquitectura esta práctica atenta contra la extensibilidad, modularidad y amigabilidad de uso, pues implica la modificación del código fuente del *base firmware* (diseñado para ser un núcleo estable en la solución) cada vez que se quiera agregar o eliminar un *user module*.

La primera aproximación a una solución definitiva fue la eliminación de las referencias del *base firmware* a los *user modules* por medio de la indirección en la referencia de sus ubicaciones en la memoria de programa del microcontrolador (por más detalle ver la sección 4.2.3), lo que resulta una técnica interesante dado permite evitar la recompilación del *base firmware* y dejar abierto la cantidad de *user modules* que se quieren agregar al *USB4all firmware*.

El siguiente paso para lograr una independencia total entre los *user modules* y el resto de los componentes del *USB4all firmware* implicaba agregar un grado de indirección en las referencias que utilizan los *user modules* hacia el resto de los componentes, esto se dejó de lado no tanto por su aporte técnico sino por su beneficio minimal de funcionalidad y considerando que el *base firmware* es fijo no valía la pena incurrir en la implementación de esta característica.

Por otro lado, más allá de poder resolver el tema de las dependencias mutuas entre los componentes del *USB4all firmware* existe el problema de donde ubicar cada *user module* en la memoria de programa del microcontrolador y la resolución de la localización de sus variables en memoria de datos. Todo esto fue debidamente investigado y se llegó a la conclusión de que es viable pero se decidió acotar su alcance, optándose por cargar un conjunto de *user modules* al mismo tiempo en una única transacción como si fueran un bloque. Esto no implica una restricción fuerte en cuanto a la funcionalidad de poder cargar y descargar *user modules* para ser usados en distintos escenarios, simplemente se renuncia a que la carga sea dinámica.

Optimización de performance del *handler manager*

Como se explicó en la sección 4.2.2.1 el *handler manager* es el encargado de la recepción y envío de los datos provenientes desde el PC por medio de la lectura y escritura de los distintos endpoints definidos en el *baseboard*. A continuación explicaremos en concreto las optimizaciones realizadas y sus justificaciones:

- **Optimización en la recepción:** Intuitivamente tendría sentido sólo realizar el escrutiño de aquellos endpoints que están siendo usados por algún vinculo lógico de un *user module,* pero a la hora de la implementación el costo de consultar y almacenar en una estructura de datos que endpoint está activo en cada momento resulta mayor al que simplemente se tendría si se consultan todos. Esto tiene un sentido práctico ya que la tecnología USB permite la existencia de como máximo dieciséis endpoints y normalmente en el *baseboard* se definen en el entorno de cuatro.

- **Optimización en el envío:** El *handler manager* brinda un servicio a los *user modules* que les permite obtener un buffer de memoria de datos para que carguen en él la información que desean enviar al el PC (es el payload del *handler protocol*). Lo normal sería que ese buffer fuera local al *handler manager* para respetar el ocultamiento de información entre capas pero por cuestiones de performance y optimización en el uso de memoria se decidió brindar la referencia de la posición en la memoria compartida del endpoint asignado al *user module* para el envío de datos.

Admin module implementado como user module

Como parte del proceso de diseño de la arquitectura de software se observó que el componente *admin module* tenía muchos puntos de contacto con los *user modules* como ser que a ambos se les asocia un *handler* para identificarlos y coexisten en la misma capa (*application layer*) en la estructura en capas de la comunicación. Estos factores llevaron a tomar la decisión de implementar el *admin module* como un *user module* con la particularidad de que no necesita la apertura en forma explícita desde el PC sino que se instancia durante el proceso de inicialización del *base firmware.*

Esta decisión tiene como ventaja la reutilización de todo el mecanismo de direccionamiento de paquetes que implementa el *handler manager*, además se habilita la posibilidad de actualizar el componente en forma independiente del resto del *base firmware.* Este enfoque se inspira en la escencia del diseño de micro-kernels [31], en donde se mantiene dentro del núcleo del sistema lo mínimo indispensable y se traslada a módulos toda la funcionalidad que puede ser actualiza o mejorada en forma independiente del mismo.

7.3. Decisiones en el PC

USB4all API usa el patrón de diseño Singleton

Durante la tarea de construcción del diseño de alto nivel de la arquitectura de software de este proyecto, se detectó la necesidad de que el subsistema *USB4all API* tuviera ciertas características que permitieran un uso sencillo y ordenado por parte de las aplicaciones de usuario que interactúan con los dispositivos. La idea de utilizar el patrón Singleton [22] en la implementación de la interfaz pública de la misma, genera la ventaja inherente de que exista una única instancia en memoria de la *USB4all API.*

Esto tiene como beneficios el evitar la existencia de múltiples instancias de la misma en las aplicaciones de usuario que llevarían a la necesidad de tener mecanismos de sincronización y control para evitar la apertura simultánea del mismo *user module* o el acceso a los datos que llegan desde el medio USB en forma errónea, es decir, recibir en una instancia del *USB4all API* datos de un *user modules* no registrado en ella.

Separación del manejo del driver de la obtención de los descriptores del baseboard

La *USB4all API* basa su funcionamiento en la obtención de la información de los descriptores de cada uno de los *baseboards* conectados al PC, de forma de identificar sus números de series y cantidades y tipos de endpoints. Esta información es utilizada por la *API* para identificar un *baseboard* en particular o para saber si existen determinados endpoint que manejan los tipos de transferencias de recepción y envió requeridos al establecerse los vínculos lógicos con los *user module*.

Las opciones de drivers que se utilizaron en este proyecto para las distintas plataformas presentan distintas funcionalidades, algunas soluciones sólo brindan las prestaciones para enviar y recibir datos y otras además permiten obtener la información de los descriptores de los dispositivos USB. Esta diferencia entre las distintas implementaciones de los drivers genéricos junto con la posibilidad de obtener la información de los descriptores del sistema operativo directamente, llevó a tomar la decisión de aislar la lógica necesaria para la obtención de los descriptores de los *baseboards* de la lógica de envío y recepción de datos, logrando así no depender de la opción de driver seleccionada.

Además, esta decisión tiene como beneficio el encapsular en un único componente de la *API* toda la lógica de obtención de descriptores de los *baseboard* para cada plataforma en que se quiere ejecutar la solución, logrando independizar a la *API* del uso de drivers que soporten la obtención de la información de los descriptores y permitiendo reutilizar la lógica de obtención de descriptores entre los distintos drivers de una plataforma.

Un file por *baseboard* conectado

En los sistemas Unix todos los componentes de hardware se mapean a un descriptor de archivo por lo tanto se estudiaron dos opciones para la interacción entre el driver genérico que se construyo y las aplicaciones de usuario. Una de ellas era tener un descriptor de archivo para cada endpoint que existe en el *baseboard*, esta opción simplifica el uso del driver pues elimina la necesidad de configurar el tipo de transferencia, dirección del endpoint, etc. previo al envió o recepción de datos pero tiene como desventaja el aumento en la complejidad del manejo del conjunto endpoints pues no existe único punto de acceso.

La otra opción es tener un único descriptor de archivo para todo el *baseboard* y por medio de él poder obtener la información de los endpoint disponibles utilizando previo al envío o recepción de datos una etapa de configuración del endpoint que se desea manipular. Debido a la mayor simplicidad en la implementación se optó por esta última opción más allá del esfuerzo de configuración previo a cada comunicación con el driver aunque al ser reducida la cantidad de las mismas se logró encapsular en funcionalidades claramente definidas.

Parte III

Experimentos y Resultados

Capítulo 8

Artefactos construidos

8.1. Introducción

En el marco de este proyecto fue necesario la construcción de un conjunto de artefactos de software, hardware y firmware, de los cuales algunos de ellos son parte estructural de la arquitectura, como es el caso del *baseboard*, en el cual se presenta una descripción de su proceso de evolución destacando las principales características de cada etapa. Además se construyeron un conjunto de aplicaciones de usuario, *adapterboards, user modules, proxies* y dispositivos que permitieron mostrar las cualidades técnicas así como la viabilidad del uso de la solución en escenarios reales y de prototipados rápidos. Además se hace una descripción de un utilitario que se entrega en forma conjunta con la solución y que permite la configuración de las propiedades del *baseboard*.

8.2. Hardware

Uno de las requisitos ineludibles para la conexión con dispositivos electrónicos externos al PC es el uso de un hardware mediante el cual interactuar con dichos dispositivos. En el contexto de este proyecto, si bien se podía haber optado por usar un kit de desarrollo y adaptarlo a tales fines, se consideró desde un primer momento el desafío del diseño y la construcción de un hardware a medida como parte del los objetivos. Esta decisión, si bien aumenta la complejidad del proyecto pues involucra áreas de conocimiento que no están abarcadas en la carrera de computación, permite por un lado disminuir los costos y el tamaño del hardware, ya que sólo se utilizan los elementos adecuados para tal fin y no los elementos extra que normalmente contienen los kits de desarrollo. Otra ventaja obtenida en el diseño construido, es la forma modular y constructiva lo que contribuye al reuso y extensibilidad de las capacidades del mismo.

Para el desarrollo y construcción de los prototipos se intentó utilizar siempre que fue posible muestras comerciales gratuitas, principalmente de microcontroladores y sensores, tanto para disminuir los costos, como por la dificultad de conseguir algunas de estas piezas en la plaza de semiconductores uruguaya. En algunas ocasiones también fue necesario reciclar componentes de partes de hardware en desuso, como es el caso del conector USB.

Para la construcción de circuitos impresos en general se utilizó la técnica que consiste en el planchado de una transparencia (realizada en impresora láser) en una chapa laminada FR10, luego de lo cual, se sumerge la chapa en percloruro férrico para eliminar el cobre sobrante. Excepcionalmente para una versión ya depurada de el *baseboard* se recurrió a el encargo de fabricación del mismo a la empresa ENEKA SA. Si bien se realizaron más prototipos de los que se describen en esta sección sólo se presentaran los más relevantes.

8.2.1. Evolución del Baseboard

Como ya se explicó es el componente de hardware que siempre está presente para interactuar con dispositivos electrónicos desde el PC. Esta es la parte de hardware que sufrió más cambios a lo largo del proyecto, y debido a ello se desea comentar algunas instancias en su proceso de desarrollo. Hubo cinco versiones en su proceso de desarrollo y a continuación de detallan las motivaciones, características y finalidades que tuvo cada versión:

- **Versión 0.2**: En esta primera versión el trabajo se concentró principalmente en lograr un diseño del circuito impreso pequeño y compacto, logrando una distribución de los componentes principales adecuada para que sea cómodo el acceso físico a los conectores RJ11, USB y *U4Aport*, así como también facilidad de uso de los pulsadores. Era importante minimizar la longitud de algunas pistas de cobre, como también eliminar el uso de cables para realizar saltos entre pistas cuando no era posible rutearlas en una sola faz de cobre, pero ésto no fue prioritario. A pesar de que la motivación de la creación de este prototipo no perseguía que fuera funcional (no era necesario soldar los componentes), igual se testeo la técnica de planchado de transparencia para obtener experiencia. En la parte superior izquierda de la figura 8.1 se muestra una imagen de la placa con sus componentes principales.
- **Versión 0.4**: Luego de haber ubicado los componentes principales, la meta fue acercarse a un prototipo funcional del mismo, para ello en la versión 0.4 se mejoró el ruteo y ancho de las pistas, debido a que se notó en la version anterior a veces quedaban en contacto algunas de ellas luego del proceso de planchado. Otra característica que también se mejoró fue el aumento del ancho de las pistas alrededor de los agujeros, pues en algunos casos, luego de perforados en la placa, quedaba muy poco cobre para soldar el componente de manera robusta. Otra requerimiento que se estimó necesario, fue dejar espacio en las 4 esquinas como para realizar agujeros que permitieran afirmar la placa a un gabinete, para ello se modificó la posición de la ficha RJ11 y USB para que estuvieran más juntas y hacia el medio de la placa. En esta versión tampoco fueron soldados los componentes. En la imagen superior derecha de la figura 8.1 se ve la cara posterior del *baseboard* pudiéndose observar el diagramado de las pistas.

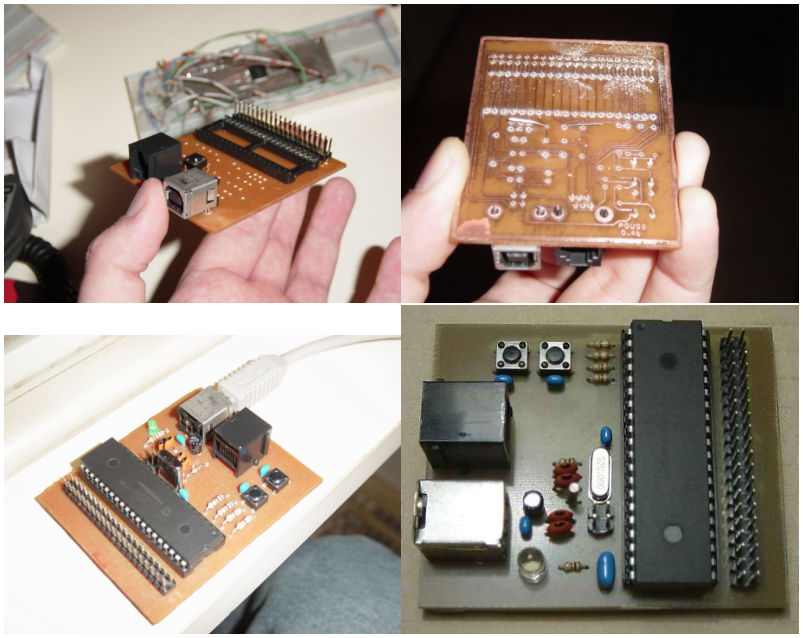


Figura 8.1: Evolución del *Baseboard*

- **Versión 0.6**: Esta versión fue la primera en la que se planteó alcanzar un prototipo funcional. Se agregó alrededor de la placa un borde de cobre que sirve de tierra, se soldaron todos los componentes y se testeó. Al conectarse al PC, se notó que tenía un funcionamiento errático, así como también se sucedían errores esporádicos en su programación (errores de checksum al grabar el firmware). Luego de revisado el circuito se observó la omisión de un componente (capacitor para la regulación de voltaje USB). Al agregar de manera provisoria el mismo, resulto en un funcionamiento correcto y ausencia de errores aleatorios, logrando satisfacer el objetivo planteado. Esta versión se muestra en la parte inferior izquierda de la figura 8.1.

- **Versión 0.8**: Se agregó el componente omitido en la versión anterior, se ensancharon las pistas de transmición de voltaje de alimentación (Vcc y Gnd) en todo el circuito. También se eliminaron algunas pistas de conexión entre el microcontrolador y el *U4Aport* para evitar problemas de programación en caso de que haya algún dispositivo conectado en el momento de utilizar el programador ICD2 para grabar el bootloader en el microcontrolador. Para tener más espacio, una esquina se sustituyeron las resistencias de 1/4 por 1/8 de watt y se juntaron un poco más los pulsadores de bootloader y reset. También se minimizó la longitud del trazo de las pistas de datos de USB para disminuir interferencias de dichas señales al llegar al microcontrolador.
- **Versión 1.0**: Esta es la versión final del *baseboard* y su objetivo principal era permitir que sea fácilmente replicable. Para atender esta necesidad se debían generar archivos en uno de los formatos estándar para la fabricación automática de circuitos impresos (GERBER), de manera de que se pueda encargar a un tercero su fabricación. Si bien se trata de un estándar, cada proveedor requiere algunas características y nombres de archivos particulares, en nuestro caso se optó realizarlos (utilizando el servicio PLAKA-circuitos impresos) en ENEKA SA. Finalmente se logró obtener un *baseboard* de manera satisfactoria utilizando esta forma de construcción. El resultado final, luego de soldados todos los componentes se muestra en la parte inferior derecha de la figura 8.1.

Como resultado de este proceso evolutivo por el cual fue transitando el *baseboard* se logró obtener un hardware con un diseño adecuado y construcción reproducible del *baseboard*, completando uno de los objetivos del proyecto. En la siguiente sección se muestran los distintos *adapterboards* y dispositivos que se fueron construyendo durante el desarrollo del proyecto.

8.2.2. Adapterboards

8.2.2.1. USB4all-protoboard

Este *adapterboard* cumple una función simple pero muy útil a la hora de prototipar dispositivos (o *adapterboards* complejos que requieren más hardware), facilitando la selección de las señales provenientes del *baseboard* (*U4Aport*) mediante cables finos para posteriormente conectarlos a un protoboard. Como ya se dijo anteriormente, este *adapterboard* es muy sencillo, ya que sólo cumple la función de seleccionar algunas señales del *U4Aport* para su posterior uso. En la figura 8.2 se muestra este *adapterboard*, como vemos consiste en un conector que por medio de un cable un cable chato de 40 pines se conecta al *baseboard* y por otro lado un par de zócalos torneados SIL de 20 pines que permiten encastrar los cables finos para llevar las señales necesarias a un protoboard. La disposición de estos zócalos permiten obtener facilmente las señales provenientes del microcontrolador, pues están ordenadas de la misma manera que éste. También se cuenta con un par zócalos de cinco pines a los costados para obtener las lineas de alimentación de tierra y 5 volts provenientes del conector USB del PC.

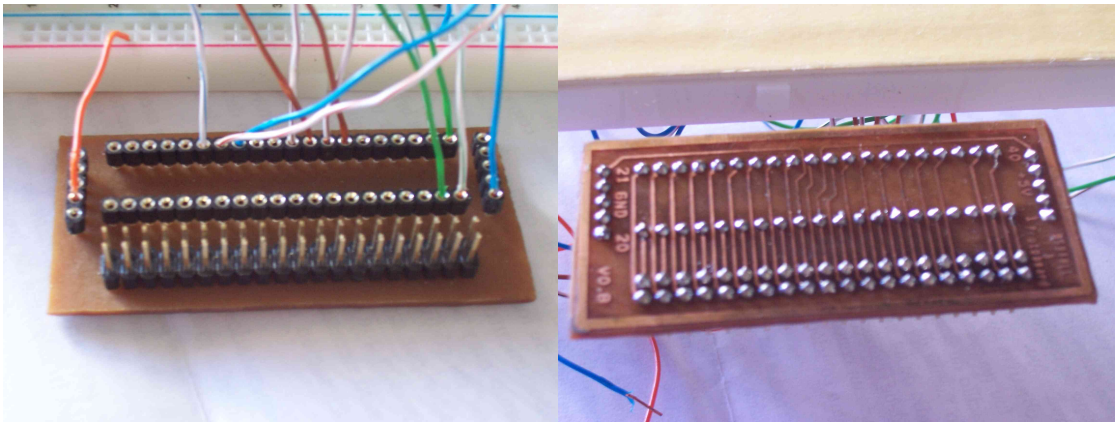


Figura 8.2: Imagenes del USB4all-Protoboard *adapterboard*.

8.2.2.2. Stepper motor adapterboard

Como no es posible conectar directamente las señales de salida del *baseboard* para manejar las bobinas del motor paso a paso unipolar, es necesario un *adapterboard*. Este está compuesto por cuatro mosfets de potencia, junto con algunas resistencias. Este *adapterboard* sólo se prototipó en un protoboard, utilizando el USB4all-protoboard. Como tampoco es suficiente la potencia brindada por el USB, se necesita una fuente externa de energía para poder hacer funcionar el motor. En la figura 8.3 se muestra una foto del mismo.

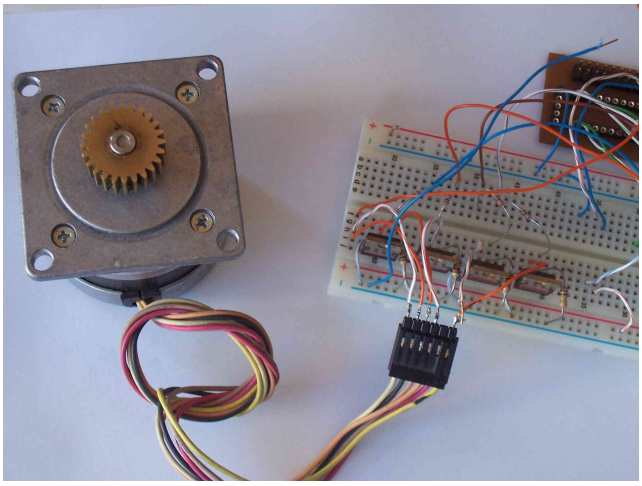


Figura 8.3: Imagen del stepper motor *adapterboard*

8.2.3. Dispositivos

8.2.3.1. Motor paso a paso (unipolar)

Utilizando el *adapterboard* mostrado en la sección 8.2.2.2 se puede conectar un motor paso a paso unipolar. Este dispositivo es muy simple y con la lógica de *user modules* se puede controlar mediante comandos tales como `mover(pasos,sentido)`, `setVelocidad(velocidad)`, `stop()`. Se realizaron dos *user modules*, el primero llamado *motor* para un testeo de la lógica del manejo del motor paso a paso pero que utilizaba un mecanismo de polling para medir el tiempo, y luego otro llamado motorISR que utiliza un *interrupt proxy* que controla los tiempos de encendido de las bobinas mediante el mecanismo de interrupciones En la figura 8.4 se muesta una imagen del *baseboard*, USB4all-protoboard, stepper motor *adapterboard*, motor y fuente externa.

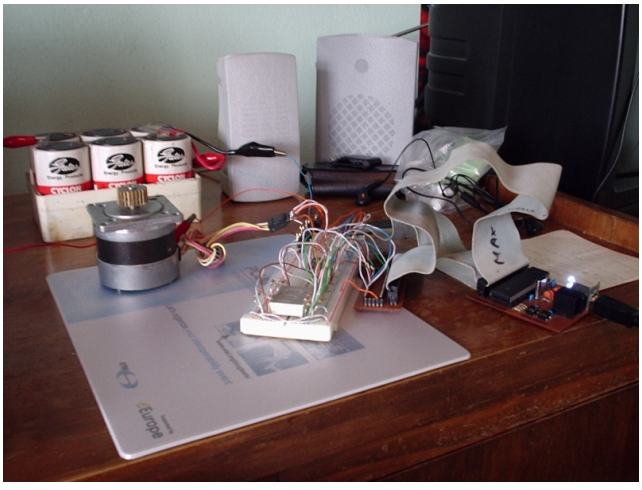


Figura 8.4: Dispositivo motor paso a paso

8.2.3.2. Sensor de Temperatura

Este dispositivo está implementado mediante un sensor de temperatura de Texas Instrument TMP101 que utiliza la interfaz Serial Peripheral Interface (SPI) obtenido como una muestra gratuita. Hubo que realizar un conversor de SMT a DIP para poder utilizarlo en el protoboard. También utilizó USB4all-protoboard para realizar las conexiones de las señales utilizadas. En la imagen 8.5 se puede observar éste dispositivo.

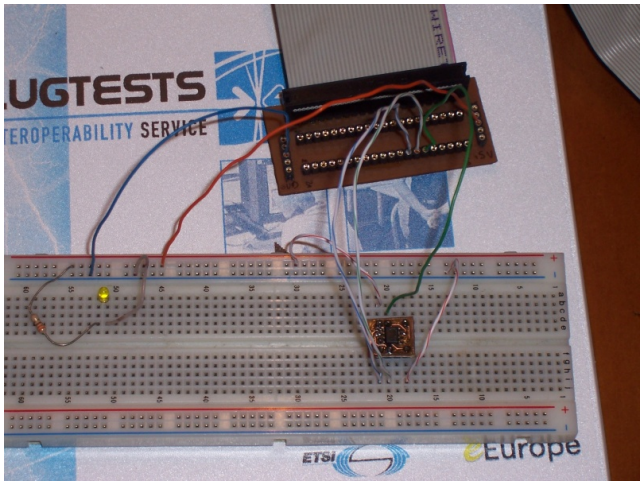


Figura 8.5: Dispositivo sensor de temperatura.

8.2.3.3. Display

Este dispositivo está implementado mediante un display de siete segmentos, y una resistencia de 330 Ohm para cada uno de ellos. El *user module* correspondiente para su manejo permite, a partir de un número recibido desde el PC transformarlo en las señales de cada segmento del display necesarias para la construcción visual del número. En la imagen 8.6 se puede observar éste dispositivo.

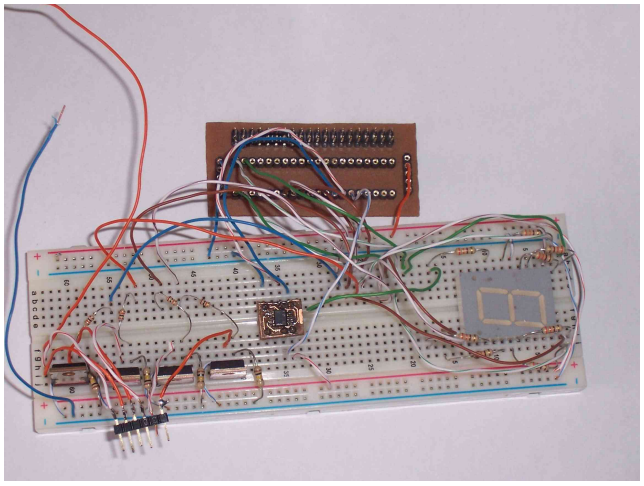


Figura 8.6: Dispositivo display 7 segmentos

8.2.3.4. USB4bot

La construcción del USB4bot se realizó en el marco de un prototipado rápido para la 4ta competencia de sumo robótico [32] organizada por el grupo MINA del Instituto de computación (InCo). Para ello se implementó un *user module* que controla un radiocontrol para facilitar el enlace inalámbrico con el robot. El *adapterboard* utilizado consta de dos DAC que permiten convertir señales digitales en analógicas para generar distintos niveles de voltajes que logran simular el movimiento físico de los controles de dirección del radiocontrol, que a la postre se convierten en distintas velocidades en los motores del robot.

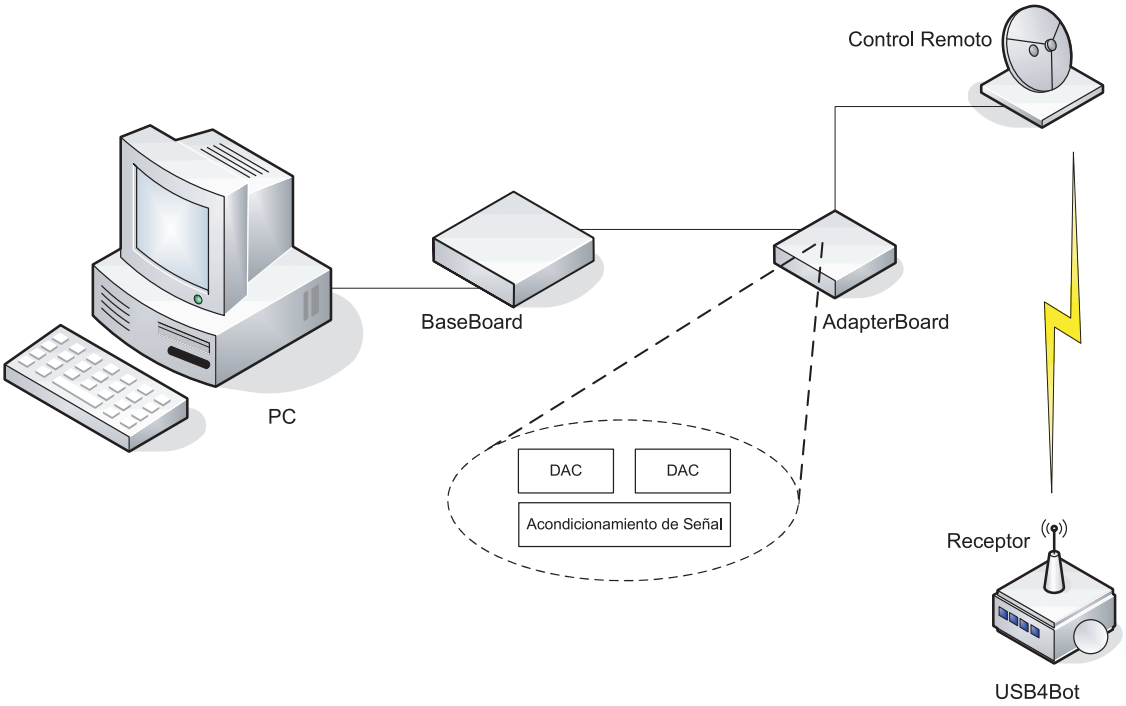


Figura 8.7: Diagrama de funcionamiento USB4bot.

Este es un caso típico de un dispositivo que no fue pensado para interactuar con un PC y mucho menos con la tecnología USB. En la figura 8.7 se muestra de manera esquemática todos los componentes que interactúan para poder controlar el robot mediante el PC. Se contaba con la inteligencia del robot de la edición anterior del campeonato realizada en Java (jugador BOTija), por lo que también se utilizaron todas las capas de software construidas dentro del PC, el driver USB genérico construido para Linux y la *USB4all Library*. Si nos centramos en lo que refiere sólo a la comunicación entre el PC y el radiocontrol los elementos o artefactos que debieron ser construidos o integrados junto a sus tiempos asociados a su elaboración y testing se muestran a continuación:

- Creación del *user module* USB4bot: 2 horas/hombre.
- Extención del componente USB4allDevice correspondiente a la *USB4all Library* (USB4bot): 15 minutos/hombre.
- Integración BOTija ⇔ *USB4all Library*: Esta integración resultó ser muy sencilla, ya que sólo hubo que cambiar algunas líneas: 15 minutos/hombre.
- *Adapterboard*: la mayor parte del tiempo fue insumida en la construcción de los DAC. Hubo que modificar el radiocontrol sacando hacia el exterior las señales provenientes de los potenciometros de los controles de dirección para poder "inyectarle" señales a fin de simular el movimiento de los mismos. Esto llevó un tiempo aproximado de 10 horas/hombre.

Teniendo en cuenta estos tiempos, que son significativamente menores que el tiempo de construcción y calibración del robot, se considera un tiempo más que apropiado para la integración con un dispositivo no pensado para interactuar con un PC.

En la figura 8.8 se muestra el testing del prototipo de un DAC conectado a un un voltímetro y leds para visualizar el estado de cada uno de los bits de entrada al DAC y verificar su correcto funcionamiento. Además, se visualizan parte de los componentes de hardware del robot, así como una de las luchas en Campeonato Uruguayo de Sumo Robótico edición 2007 [32].

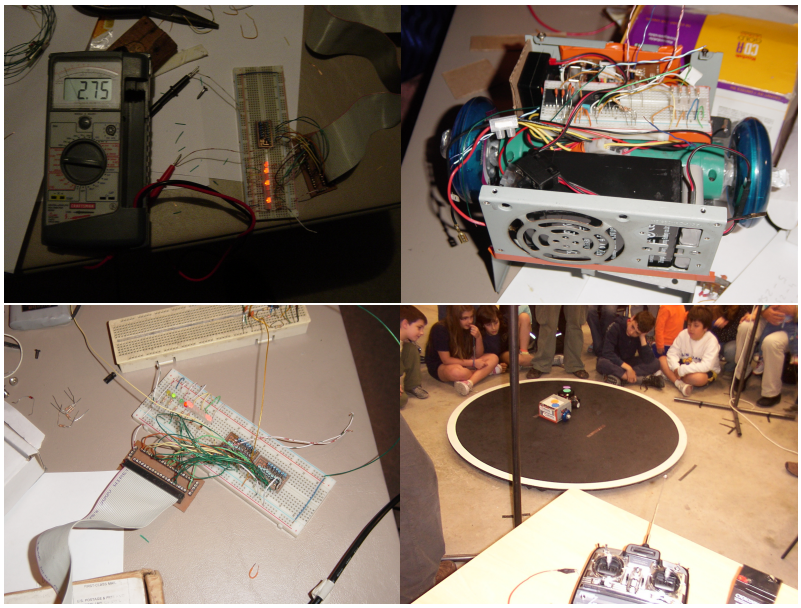


Figura 8.8: Imagenes de la construcción y verificación del USB4bot.

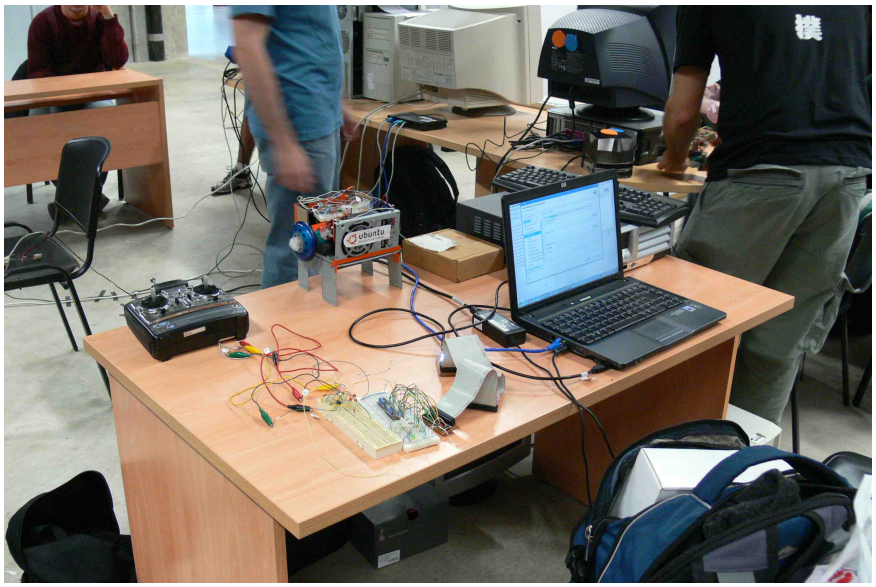


Figura 8.9: Evento Sumo.uy 2007

8.3. Software

Además de todo el software y firmware que componen la arquitectura, también se realizaron diversas aplicaciones de usuario, las cuales utilizan en algunos casos la *USB4all Library* y en otros directamente la *USB4all API* para verificar su equivalencia de uso. También se construyó un programa utilitario que permite la configuración del *baseboard* en forma amigable. A continuación se describe las características de esta herramienta, llamada *USB4all baseboard utility*.

8.3.1. USB4all baseboard utility

Esta herramienta es la encargada de dos tareas principales: por un lado la configuración de los descriptores USB y por otro, la tarea de carga de *user modules* y *proxies* al *baseboard*. Esta aplicación fue desarrollada en Java, lo que permite su uso en varias plataformas.

En lo que refiere a la configuración de los descriptores, este utilitario permite definir toda la información que el dispositivo USB intercambia con el PC en el proceso de enumeración. En el caso del *baseboard* en particular, son de vital importancia la definición tanto de la cantidad y tipo de endpoints que éste contiene, como también el número de serie con el cual se va a identificar. Es importante destacar lo poderosa que es esta funcionalidad y lo sencillo que es utilizarla, ya que este tipo de configuración normalmente está restringida a su implementación mediante código de firmware fijo.

Dado que uno de los objetivos iniciales fue brindar facilidad de uso, inclusive a usuarios con muy poco conocimiento, fue necesario invertir un tiempo importante para la implementación de un mecanismo muy sencillo de usar. El camino elegido para realizar esta funcionalidad fue la de generación de código fuente de manera automática así como su posterior grabación en el microcontrolador por medio de un *user module* dedicado a tales fines, el *Bootloader module*. Debido a la interacción que debe realizarse con un proyecto de MPLAB, compilador y linkeditor, además de haber sido necesario un conocimiento en detalle de el formato de los archivos .HEX, hacen de esta aproximación una solución compleja de implementar, pero permite un uso muy sencillo.

Si bien están implementados todos los mecanismos necesarios para esta tarea, no se llegó a la meta de brindar una interfaz gráfica de modo que el usuario pueda pueda interactuar de manera más amigable. Provisoriamente se brinda un ejemplo de código fuente que puede ser modificado a los efectos de configurar un *baseboard* con los descriptores deseados.

La otra tarea que lleva a cabo este utilitario es la carga de *user modules* y *proxies* al *baseboard*. Para realizar esto, sólo es necesario seleccionar de un conjunto de *user modules* y *proxies*, los cuales van a ser cargados en la memoria de programa del microcontrolador. Como ellos ocupan memoria de programa en espacios reservados según el mapa de memoria presentado en la sección 4.2.6 pueden grabarse de manera independiente al resto del firmware. Para la realización de este mecanismo también se necesitó una interacción con un proyecto MPLAB, compilador y linkeditor para poder luego grabar el conjunto de elementos seleccionados.

Una aclaración importante es que debido a la dependencia que se agregan en las dos tareas principales de este módulo con el MPLAB, compilador y linkeditor MPLAB C18 limitan el uso de este utilitario a la plataforma Windows.

Capítulo 9

Conclusiones y Trabajo a Futuro

9.1. Conclusiones

El desarrollo de este trabajo planteo una variedad de desafíos como ser la construcción de los baseboards y la programación de su microcontrolador entre otros, que mediante un importante esfuerzo de investigación y experimentación se pudieron superar.

Una característica particular de este proyecto fue el proceso de maduración que sufrieron sus objetivos en las primeras etapas de trabajo, pues se comenzó con un conjunto inicial muy general y luego del esfuerzo importante de investigación del estado del arte de las distintas temáticas involucradas (desconocidas para el grupo) se logró depurarlos a un conjunto de objetivos más concretos pero igual de ambiciosos a los propuestos inicialmente.

Esto tuvo como consecuencia que la fase de investigación del estado del arte y definición del alcance del proyecto se extendiera más de lo deseable y junto con una dificultad mayor a la esperada en el aprendizaje de las técnicas de construcción del hardware y programación de su firmware impactaron en forma negativa en el tiempo de finalización del proyecto. De todas formas se contaron con algunos elementos que permitieron mitigar en parte la duración del proyecto, como fueron, el curso Taller de Firmware dictado por la Facultad de Ingeniería y la placa PICDEM FSUSB de Microchip proporcionada por el tutor que se utilizaron en la experimentación realizada en las primeras etapas de desarrollo.

La utilización durante todo el proyecto de herramientas como Wiki [33] para la documentación y actas de reuniones, Mantis [28] para el reporte y seguimiento de defectos y fallas y CVS [11] para el control de cambios de los códigos fuentes y documentación, permitieron una buena coordinación del trabajo entre los integrantes del equipo y se evaluó como positivo.

Otro aspecto relevante de destacar del proceso de implementación fue la estrategia de aislar los desarrollos del *USB4all firmware*, drivers y *USB4allApi* por medio del uso de algunos prototipos descartables que permitieron reducir los posibles factores de riesgo, lograr un mayor grado de paralelismo en la programación y facilitar la integración a posterior.

Características de la solución construida

Al final de este trabajo se pudieron cumplir con casi la totalidad de los objetivos propuestos para este proyecto destacando las siguientes características de la solución.

- **Integral**: La solución está compuesta por hardware y software que permite ocultar toda la complejidad existente entre las aplicaciones de usuario y los dispositivos electrónicos. Del estudio del estado del arte se vió que en general, las soluciones existentes se dividen en las que sólo contemplan el hardware (algunas incluyendo un soporte mínimo para la PC) y las que se centran en la problemática de los drivers del PC y no en las dos áreas al mismo tiempo en forma conjunta.

- **Dispositivo genérico**: El *baseboard* se clasifica en el estándar USB como un dispositivo Custom y no en otra clase específica, que limitaría a un subconjunto los dispositivos a conectar. Esta característica genera que los sistemas operativos no puedan asignar un driver por defecto como lo harían frente a clases de dispositivos específicos como son HID, CDC, Audio, etc. Para solucionar esta dificultad se utiliza único driver USB genérico que

implementa todos los tipos de transferencias de forma de no necesitar un driver especifico para cada dispositivo en particular.

- **Protocolo abierto y user modules inteligentes**: La solución brinda un protocolo de comunicación entre las aplicaciones de usuario y los *user modules* sin restricciones en cuanto a su contenido permitiendo el uso de los mismos a demanda. Si además sumamos que los *user modules* tienen la libertad de implementar con mayor o menor grado de inteligencia el manejo de los dispositivos, obtenemos una forma de solucionar la interacción con dispositivos cuyas restricciones de tiempo no pueden ser procesados desde las aplicaciones de usuario en el PC.

- **Constructivo**: Brinda la posibilidad de componer las funcionalidades de manejo de dispositivos que brindan los *user modules* en las aplicaciones de usuario del PC de forma de resolver problemas de mayor tamaño y complejidad. En otras palabras, la aplicación de usuario del PC se encarga de manipular a cada uno de los dispositivos electrónicos participantes de la solución de forma de lograr un accionar en conjunto. Esto redunda en multiplicidad de beneficios, entre los que se destacan: la reutilización de *user modules* en distintos escenarios de uso, la delegación de la complejidad de la coordinación (lógica) entre ellos hacia la aplicación de usuario donde se cuenta con mayor cantidad recursos, facilidad de programación y nivel de abstracción evitando la programación directa del firmware del microcontrolador.

- **Multi-instancia de baseboards**: La capacidad de manejar dispositivos electrónicos no está limitada a un solo baseboard por PC, sino que la solución fue diseñada para soportar varias instancias de baseboards en forma conjunta. En otras palabras, si las posibilidades de conexión de dispositivos en un baseboard se ven excedidas frente al problema a resolver, simplemente se puede conectar otro baseboard al PC y conectar los dispositivos faltantes sin que este tengo un impacto importante en la aplicación de usuario. Esta característica hace que las soluciones sean escalables en el tiempo, ya que al aumentar las necesidades de manejo de dispositivos de una solución ya existente simplemente se deben agregar nuevos baseboards para el manejo de los dispositivos adicionales.

- **Multi-plataforma**: El diseño fue pensado para soportar cualquier plataforma debido a que su núcleo solo utiliza funciones estándar de C++ [8] y se encapsulan las funcionalidades de interacción con los sistemas operativos y drivers en forma estructurada para cada plataforma específica. Dentro del alcance de este proyecto solo se implementó para funcionar en las plataformas Linux y Windows.

- **Multi-lenguaje de programación para aplicaciones**: Las funcionalidades básicas del sistema en el PC (*USB4all API*) están disponibles como una biblioteca de vinculación dinámica (.dll en Windows o .so en Linux), lo que permite que sea utilizada por cualquier lenguaje de alto nivel que soporte este tipo de bibliotecas.

- **Orientación a objetos**: Para facilitar el uso de la solución así como ampliar el espectro de usuarios, se propuso a manera de ejemplo una jerarquía de clases que logra abstraer los conceptos de baseboard y dispositivos en clases base, las que habilitan la especialización para dispositivos particulares por parte de los usuarios. Esto genera el beneficio de utilizar las características de los lenguajes orientados a objetos sin perder las funcionalidades que se brindan a nivel *USB4all API*.

- **No uso de conversores USB-Serial**: A diferencia de otras soluciones que hacen uso de conversores USB-Serial se opto por no utilizarlo debido a que esto sacrifica gran parte de las prestaciones que ofrece la tecnología USB como son los tipos de transferencias, cantidad de endpoints, tamaños de buffers y velocidades de trasmisión entre otras. Esta elección nos permite tener un mayor grado de granularidad y flexibilidad para la asignación de los recursos USB a los *user modules*, así como evitar los posibles cuellos de botella en la comunicación que introduce utilizar dicho conversor.

- **Costos económicos**: La casi totalidad de los costos de la solución se centran en los elementos que conforman el hardware del *baseboard*, los cuales rondan en el mercado uruguayo para una unidad un valor de $700 pesos (USD 32) el microcontrolador y los componentes

electrónicos y alrededor de $500 (USD 23) la fabricación del circuito impreso en ENEKA. Estos precios convierten al producto en una opción que compite con otras soluciones tanto open source como propietarias.

- **Open Source Software y Hardware**: La totalidad de los fuentes del software y firmware se encuentra a disposición de la comunidad bajo la licencia GNU-GPL [23] y para el hardware se presentan los esquemáticos, el diseño del PCB incluyendo formato Gerber y toda la documentación de construcción asociada a fin de que sea fácilmente reproducible.

Aportes de la solución

La solución construida permite resolver algunas falencias detectadas luego de relevar el estado del arte, así como cumplir satisfactoriamente con el objetivo primario de este proyecto de grado además de introducir innovaciones propias. A continuación se muestra un resumen de los aportes más significativos de la solución:

- **Extensión del dominio de acción del PC y dispositivos:** Esta característica puede verse desde dos ópticas distintas, la del PC y la de los dispositivos electrónicos. Desde el PC se logra expandir las capacidades de interacción con el mundo físico que lo rodea, permitiendo aumentar los recursos disponibles con que normalmente cuenta. Esta posibilidad habilita plantear nuevos escenarios de uso así como resolver necesidades ya existentes. Desde la perspectiva de los dispositivos electrónicos, la interacción con el PC les permite contar con los recursos de procesamiento, almacenamiento y comunicación del mismo logrando expandir sus capacidades de uso.
  A modo de ejemplo, si pensamos en un dispositivo sencillo como un sensor de temperatura que normalmente sólo funciona como un termómetro indicando la temperatura en un display, al incorporarle las capacidades de un PC se puede llevar una estadística de temperaturas por períodos de tiempo, publicar la temperatura en una pagina web en tiempo real, etc. Esto es útil en los escenarios donde ya se cuenta con un PC y se le agrega la capacidad de interacción con dispositivos electrónicos, también es aplicable cuando se intenta dar uso a PCs antiguas (Pentium I o superiores) como recurso dedicado, y finalmente para la creación de sistemas embebidos en los cuales con más frecuencia se utilizan placas madre de PC en formato reducido con los sistemas operativos habituales.
  Resumiendo, el sistema USB4all permite obtener lo mejor de ambos mundos.

- **Desarrollo guiado y amigable:** Como parte de la solución se presenta una metodología de uso e implementación que permite un desarrollo guiado y amigable para la perspectiva de un programador. Para acelerar los tiempos de aprendizaje en el uso del sistema, se proveen de templates o esqueletos de los *user modules* y proxies en el firmware y una jerarquía de clases base en Java de las cuales se pueden extender para la realización de dispositivos específicos. Estos templates son ejemplos que definen los contratos necesarios que deben ser respetados para la integración con las demás partes del sistema que logran abstraer al usuario de toda la complejidad inherente a la tecnología USB y de los vínculos lógicos entre las aplicaciones de usuario y los *user modules*.
  Otra característica relacionada con lo amigable de la solución, es que a diferencia de otros productos relevados en el estado del arte que proveen una solución sólo para el hardware o sólo para el software, USB4all elimina por completo los problemas de integración entre el hardware y software, ya que brinda una respuesta integral a ambas problemáticas.
  Por último el *USB4all system* cuenta con un utilitario con interfaz gráfica que permite la definición de los descriptores USB, la generación de su código fuente, compilación y actualización del baseboard, sin la utilización de un programador (vía USB). Esta información es utilizada por el baseboard al momento de enumerarse como dispositivo USB en el PC, para diferenciarse de otros baseboard, así como para establecer los vínculos lógicos entre las aplicaciones de usuario y los *user modules*.

- **Perfiles de usuarios**: El sistema *USB4all* contempla a distintas clases de usuarios separándolos en básicos, avanzados y especialistas según su nivel de conocimiento de la solución. Los usuarios básicos son aquellos que sólo desarrollan aplicaciones de usuario utilizando la jerarquía de clases, los *user modules*, los adapterboard, los proxies y los dispositivos electrónicos ya existentes para lograr manejarlos en forma coordinada. Luego están los usuarios

avanzados que poseen un mayor grado de conocimiento de la arquitectura y dispositivos hardware permitiéndoles además del desarrollo de aplicaciones de usuario la construcción de *user modules* siguiendo los templates, construcción de adapterboards y el desarrollo de nuevas clases que extiendan la jerarquía de dispositivos. Por último están los usuarios especialistas que además de todo lo descrito anteriormente poseen un conocimiento detallado de la arquitectura *USB4all* (microcontrolador, plataforma, drivers, USB, etc.) que los habilita a construir proxies, extender el uso de la API a otras plataformas o drivers y poder hacer cambios funcionales en los propios componentes del sistema.

- **Fomenta la colaboración entre usuarios:** Debido a que solución es de carácter libre (open source software y hardware) y los *user modules*, *proxies* y clases de la jerarquía son autocontenidos e intercambiables, se favorece el reuso de los mismo frente a distintos escenarios y se habilita la colaboración entre usuarios. Esto último tiene como ventaja que los usuarios básicos (la mayoría) se beneficien de los aportes de usuarios con mayor nivel de conocimiento que compartan sus aportes de forma de crear una comunidad.

- **Apoyo a tiempo real:** La facilidad de incorporar funcionalidad específica por medio de los *user modules* y el manejo el mecanismo de atención de interrupciones del *dynamicISR,* permiten mejorar las condiciones de atención de eventos que requieren respuestas en tiempo real debido a que se evita el tiempo de latencia en la comunicación que existiría si la lógica se encontrara en el PC. El nivel de complejidad de las respuestas a estos eventos determina si pueden ser implementadas en los *user modules* o necesariamente deben residir en el PC, para este último escenario la tecnología USB ofrece la alternativa del tipo de transferencia Interrupt el cual permite consultar frecuentemente (cada 100 ms por ejemplo) si los dispositivos poseen información para enviar al PC.
  Estas dos características dejan abierto al usuario la posibilidad de construir soluciones que combinen la atención de eventos directamente desde los *user modules* o desde el PC según las necesidades de los problemas.

- **Driver USB genérico para Linux:** Se implementó un modulo de kernel que auspicia de driver USB del *baseboard* donde soló se colocaron los elementos necesarios para la obtención de los descriptores USB y de los mecanismos para lograr el envío y recepción de datos. Cabe destacar que este driver no sólo funciona con esta solución sino que puede ser usado para manejar cualquier tipo de dispositivo USB.

- **Prototipos rápidos:** La conjunción de características tales como: la facilidad de incorporar *user modules*, la existencia de una jerarquía de clases Java que modela los dispositivos electrónicos, la posibilidad de configurar los descriptores USB y la reutilización del *baseboard* para distintos usos, permiten brindar una base inicial para el desarrollo de nuevas soluciones que disminuye los tiempos de construcción, integración y verificación a la vez que facilita su elaboración pues el desarrollador se centra únicamente en el comportamiento del dispositivo electrónico y no en como es la comunicación desde el PC.

9.2. Trabajo a Futuro

La solución construida durante este proyecto de grado alcanzó la mayoría de los objetivos planteados inicialmente, durante el proceso se fueron incorporando nuevas características a las ya planificadas y también fueron necesarios ajustes en su alcance. A continuación se presentan los posibles trabajos a futuro separados en tres grupos: objetivos eliminados del alcance, oportunidades de mejora e investigación.

Objetivos eliminados del alcance

- **Actualización de User Module vía USB en forma independiente:** La idea principal consiste en tener un mecanismo de carga y descarga de *user modules* similar al proporcionado por el sistema operativo Linux. Con esto se obtiene el beneficio de no ser necesario recompilar todo los fuentes para agregar, eliminar o modificar un *user module*, también se proporciona un mecanismo que permite comenzar con un entorno mínimo e ir extendiéndolo según las necesidades del usuario. Otra ventaja es que se elimina la necesidad del

uso del entorno de desarrollo para extender el firmware, con lo cual se facilita el uso para los usuarios básicos, este mecanismo también permite compartir de forma muy sencilla *user modules* entre usuarios similar al mecanismo de plugins. Actualmente se cuenta con la facilidad por medio de un utilitario (solo disponible para plataforma Windows) de cargar todo el conjunto de *user modules* sobrescribiendo los contenidos previamente existentes en el *USB4all Firmware*, por más información referirse a la sección 8.3.1. Un problema relacionado con este tema es resolver la fragmentación que se genera al cargar y descargar módulos dinámicamente.

- **Aplicación Bootloader en PC integrada a utilitario desarrollado:** Actualmente se utiliza la aplicación de Microchip (sólo para Windows) para interactuar con el bootloader que se encuentra en el firmware, dado que se tuvo que investigar el formato del HEX para la aplicación utilitaria desarrollada, la idea es reutilizar el conocimiento para integrarla a la aplicación existente. De esta forma se elimina la dependencia de aplicaciones de terceros y se cuenta con una aplicación integrada la cual es multi-plataforma. Para lograr esto se debería modificar el *USB4all API* para que interactúe con los *baseboards* en modo bootloader.

- **Transferencias isócronas (no soportada por los drivers existentes):** LibUSB actualmente no soporta transferencias isócronas, Microchip según la documentación de su driver las soporta pero le fueron encontrados bugs, pero como es propietario no se cuenta con los fuentes para solucionarlo. Esta fue una de las motivaciones para el desarrollo del driver propio para Linux, pero por temas de tiempo se dejo para trabajo futuro brindar soporte a ese tipo de transferencia en nuestro driver.

- **Modo de uso Fachada:** Como ya se explicó en la sección 7.3 del capítulo Decisiones de diseño e implementación, éste modo de uso de la solución se eliminó del alcance del proyecto, de todas formas sigue siendo un escenario interesante para la creación de un proyecto paralelo. Se propone como trabajo a futuro la investigación de distintas alternativas que logren satisfacer adecuadamente los requerimientos de esta modalidad de uso.

Oportunidades de mejora

- **Mejorar la robustez**: Actualmente el manejo de errores implementado a nivel del *USB4all API* es muy simple, sólo informa si se produjo un error o no, se debería contar con una clasificación de los errores de forma de notificar adecuadamente a las aplicaciones de usuario cual es el problema.
  Frente a un conjunto de situaciones anormales como son la ausencia de baseboards conectados al PC o la remoción de ellos durante la ejecución de una aplicación de usuario no existen mecanismos adecuados para su manejo. Se debería poder ampliar las funcionalidades del sistema de forma de poder detectar los cambios (adición o remoción) de los baseboards conectados al PC y actuar en consecuencia.
  Se debería incorporar a las funcionalidades del sistema *USB4all* la posibilidad de recuperación de los vínculos lógicos ante fallas en el PC o en los baseboards por medio de la sincronización de la información.

- **Mejoras al Scheduler:** El actual mecanismo que utiliza el *base firmware* para asignar tiempo de CPU del microcontrolador a aquellos *user modules* que lo requieren utiliza una política de round robin sin cotas de tiempo (se aspira a un accionar cooperativo entre los *user modules* y el *base firmware*). Se cree conveniente la incorporación de una técnica de manejo de prioridades que permita que algunos *user modules* pueden acceder con mayor frecuencia a tiempos de CPU que otros. Otra mejora interesante, sería la implementación de un mecanismo de expropiación del CPU que utiliza un *user module* de forma de evitar la inanición del resto de los *user modules*.

- **Modo Stand Alone:** La solución se diseñó para la interacción con dispositivos electrónicos desde un PC, esto implica que exista una conexión física permanente con el mismo para su funcionamiento. Durante el desarrollo de este proyecto se vió con interés el empleo en escenarios de funcionamiento independientes del PC (desconectado). Por ejemplo, para implementar una red de sensores que adquieran datos meteorológicos, los almacenen de

forma temporal para posteriormente transferirlos al PC vía USB para su procesamiento. Para esto sería necesario realizar algunos cambios en la circuitería del *baseboard* para resolver el conflicto que presenta la doble alimentación de energía desde el USB y una fuente externa. Además se debería cambiar el *base firmware* para que sea capaz de detectar si está conectado a un PC o no y mediante un nuevo módulo encapsular toda la apertura (open) de los *user modules* existentes en el *baseboard*.

- **Asincronismo:** La actual implementación de la solución fue pensada para un escenario sincrónico donde las aplicaciones de usuario tienen el protagonismo en la comunicación con los *baseboards* (formato comando/respuesta), de todas formas se puede simular un comportamiento asincrónico a expensas de no compartir los endpoints entre los *user modules* y realizar un `receiveData` bloqueante en un hilo de proceso separado para cada vínculo lógico abierto.
  La inclusión de un mecanismo de asincronismo implicaría que el *USB4all API* tuviera un hilo de proceso por cada endpoint de entrada de forma de bloquearse a la espera de la recepción de datos desde el *baseboard*, como se ve en la figura 9.1. Cuando llega un dato debería analizarlo para identificar a que aplicación corresponde entregarlo. El beneficio principal de esta mejora sería poder compartir el uso de un endpoint determinado por varias aplicaciones. Para ello, la *USB4all API* debería implementar un mecanismo de mutua exclusión para la notificación del uso del recurso compartido (endpoint). Cabe señalar que la actual implementación no posee ningún tipo de espera activa.

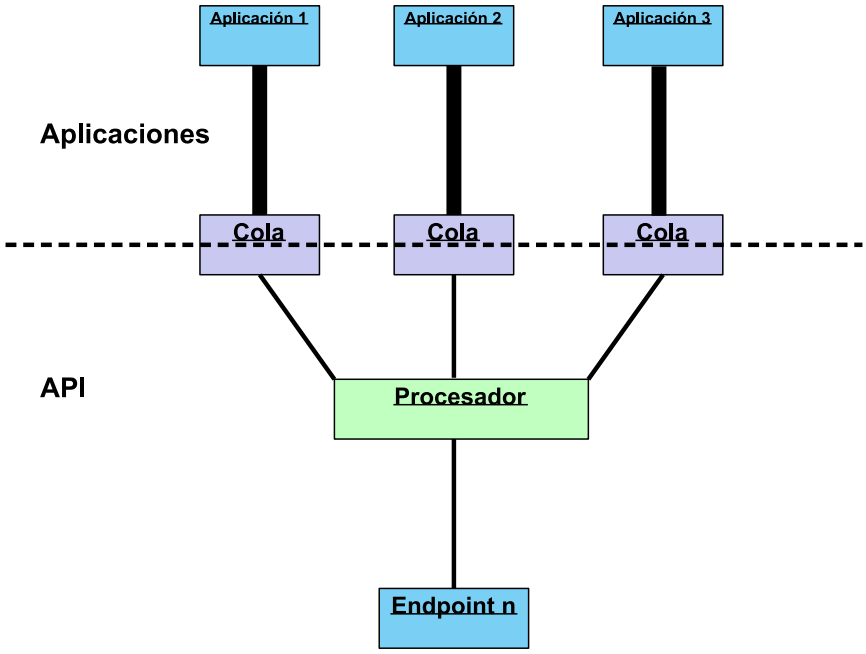


Figura 9.1: Idea de asincronismo de futuro

- ***USB4all baseboard utility* en Linux:** El utilitario *USB4all baseboard utility* es dependiente de los proyectos MPLAB y el compilador MPLAB C18, por lo que actualmente esta aplicación no puede utilizarse desde la plataforma Linux. Se investigó y en algunos foros en Internet se ha comentado de instalaciones exitosas de MPLAB en Linux mediante WINE [34]. Como trabajo a futuro se propone experimentar y efectuar los cambios necesarios en la aplicación *USB4all baseboard utility* de manera de lograr su funcionamiento también en este sistema operativo.

Investigación

- **Wireless USB**: Durante el desarrollo de este proyecto de grado se liberó el estándar Wireless USB 1.0 [35] que implementa la comunicación inalámbrica entre los dispositivos

USB y la PC. Este nuevo estándar abre las puertas a una nueva dimensión de conectividad USB a la vez que introduce nuevo desafíos, los cuales son atractivos para la investigación y/o experimentación. En lo referente al producto *USB4all* se ve con interés el estudio del impacto que puede tener el uso de *Wireless USB Hub (F5U302)* de Belkin que permitiría prescindir del cable USB que conecta el PC con los *baseboards*. Esto sería el primer paso en el estudio de la temática a la espera de que los fabricantes de microcontroladores ofrezcan versiones compatibles con el nuevo estándar.

Figura 9.2: Wireless USB Hub (F5U302) de Belkin

- **Portabilidad a Mac OS, Solaris y otros sistemas operativos:** En el contexto de este proyecto se seleccionaron las plataformas Windows y Linux para su implementación, aunque el diseño de la arquitectura contempla la utilización en cualquier plataforma que soporte la tecnología USB. Por este motivo es desafiante la migración a otros sistemas operativos de los cuales los miembros de este proyecto no tienen experiencia aún como son MacOS, Solaris, etc.

Bibliografía

[1] Active Wire, http://www.activewireinc.com/, último acceso diciembre 2007.
(Citado en la página 36.)

[2] AGUIRRE, Andrés; FERNÁNDEZ, Rafael; GROSSY, Carlos; Estado del Arte; UDELAR - FING, 2007.
(Citado en las páginas 20, 23, 28, 29 y 31.)

[3] AGUIRRE, Andrés; FERNÁNDEZ, Rafael; GROSSY, Carlos; Manual de Usuario USB4All; UDELAR - FING, 2007.
(Citado en las páginas 47, 48 y 49.)

[4] AID, http://www.student.ocad.on.ca/~aid_dev/, último acceso diciembre 2007.
(Citado en la página 36.)

[5] ANDERSON, Don; USB System Architecture (USB 2.0); Second edition, 2001.
(Citado en las páginas 23 y 76.)

[6] Arduino, http://www.arduino.cc/en/, último acceso noviembre del 2007.
(Citado en las páginas 36 y 37.)

[7] AXELSON, Jan; USB Complete, Everithing You Need to Develop Custom USB Peripherals; Third Edition, 2005.
(Citado en la página 23.)

[8] C and C++ Standards, http://home.att.net/~jackklein/c/standards.html, último acceso 26 de noviembre del 2007.
(Citado en la página 114.)

[9] CORBET, Jonathan; RUBINI, Alessandro; KROAH-HARTMAN, Greg. Linux device drivers. 3rd Edition, O' Reilly Media, 2005.
(Citado en las páginas 85 y 86.)

[10] CUI, http://www.create.ucsb.edu/~dano/CUI/, último acceso noviembre del 2007.
(Citado en las páginas 36 y 39.)

[11] CVS, Concurrent Versions System, http://www.nongnu.org/cvs, último acceso 26 de noviembre del 2007.
(Citado en las páginas 113 y 133.)

[12] Data Translation, http://www.datx.com/products/dataacquisition/usb/dt9835.asp, último acceso diciembre 2007.
(Citado en la página 36.)

[13] DevaSys - USB I2C, http://www.devasys.com/usbi2cio.htm, último acceso noviembre del 2007.
(Citado en la página 36.)

[14] Elexol, http://www.elexol.com/IO_Modules/, último acceso diciembre 2007.
(Citado en la página 36.)

[15] Estándar USB 2.0, http://www.usb.org/developers, último acceso 26 de noviembre del 2007.
(Citado en las páginas 23 y 64.)

[16] Estándar USB para la clase MIDI, http://www.usb.org/developers/devclass_docs/, último acceso 26 de noviembre del 2007.
(Citado en la página 25.)

[17] Estándar USB para la clase HID, http://www.usb.org/developers/devclass_docs/, último acceso 26 de noviembre del 2007.
(Citado en la página 23.)

[18] Estándar USB para la clase CDC, http://www.usb.org/developers/devclass_docs/, último acceso 26 de noviembre del 2007.
(Citado en la página 45.)

[19] FLIEGL, Detlef; Programming Guide for Linux USB Device Drivers, Departamento de Informática, Universidad Técnica de Munich, 2000.
(Citado en la página 86.)

[20] FT232BL/TR, http://www.ftdichip.com/Products/FT232BM.htm, último acceso noviembre del 2007.
(Citado en la página 37.)

[21] Gainer, http://gainer.cc/, último acceso diciembre 2007.
(Citado en la página 36.)

[22] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John Design Patterns: elements of reusable object-oriented software 1st Edition, Addison-Wesley Professional Computing Series, 1994.
(Citado en las páginas 67 y 101.)

[23] GNU-GPL, The GNU General Public License, http://www.gnu.org/licenses, último acceso 26 de noviembre del 2007.
(Citado en la página 115.)

[24] HYDE, John;USB Design by Example, A Practical Guide to Building I/O Devices, 1999.
(Citado en la página 23.)

[25] LIBUSB, http://libusb.sourceforge.net, último acceso en julio del 2007.
(Citado en las páginas 78 y 84.)

[26] Lyx, http://www.lyx.org, último acceso 26 de noviembre del 2007.
(Citado en la página 133.)

[27] MANDANY; ISLAM; KOUGIOURIS: CAMPBELL, Reification and Reflection in C++ An Operating Systems Perspective.
(Citado en la página 55.)

[28] MANTIS de USB4all, http://usb4all.dyndns.org/mantis/, último acceso 26 de noviembre del 2007.
(Citado en las páginas 113 y 134.)

[29] Microchip Technology Inc.; MPLAB C18 C Compiler User's Guide, 2005.
(Citado en la página 57.)

[30] Morphy Planning's USB-IO, http://www.fexx.org/usbio/index-en.html, último acceso diciembre 2007.
(Citado en la página 36.)

[31] SILBERSCHATZ, Abraham; LABS, Bell; GALVIN, Peter B., Sistemas Operativos 5° Edición, Addison-Wesley Longman Inc., 1999.
(Citado en la página 101.)

[32] Sumo.uy, http://www.fing.edu.uy/inco/eventos/sumo.uy/, último acceso 26 de noviembre del 2007.
(Citado en las páginas 110 y 111.)

[33] WIKI, Portada - USB4ALL http://usb4all.dyndns.org/wiki/ ,último acceso 26 de noviembre del 2007.
(Citado en las páginas 113 y 134.)

[34] WINE, http://www.winehq.org/, último acceso diciembre 2007.
(Citado en la página 118.)

[35] Wireless Universal Serial Bus Specification 1.0, http://www.usb.org/developers/wusb/docs/, último acceso 26 de noviembre del 2007.
(Citado en la página 118.)

[36] Wiring, http://wiring.org.co/ioboard/index.html, último acceso noviembre del 2007.
(Citado en las páginas 36 y 38.)

Apéndice A

Glosario

En esta sección definiremos algunos términos básicos para el entendimiento del material contenido en éste informe.

ACK: Abreviatura de ***Ack****nowledgement* (Asentimiento positivo), en comunicaciones entre computadores, es un mensaje que se envía para confirmar que un mensaje o un conjunto de mensajes han llegado. Si el terminal de destino tiene capacidad para detectar errores, el significado de ACK es "ha llegado y además ha llegado correctamente".

ADC: Acrónimo de ***A****nalog* ***D****igital* ***C****onversor* (Conversor Analógico-Digital), es un circuito integrado electrónico que convierte señales continuas a números digitales discretos.

ANSI_C: Nombre con que se conoce vulgarmente la versión del lenguaje C estandarizado por el Instituto Nacional Estadounidense de Estándares (sigla en inglés ANSI) en el año 1989.

API: Sigla de ***A****pplication* ***P****rogramming* ***I****nterface* (Interfaz de Programación de Aplicaciones), es el conjunto de funciones y procedimientos que ofrece cierta librería para ser utilizado por otro software como una capa de abstracción. Una buena API hace más fácil desarrollar un programa mediante el suministro de todos los elementos de construcción.

ATA: Acrónimo de ***A****dvanced* ***T****echnology* ***A****ttachment* (Tecnología Avanzada de Fijación), es una interfaz estándar para conectar dispositivos de almacenamiento como discos duros y unidades de CD-ROM dentro de los computadores personales.

ATAPI: Acrónimo de ***A****dvanced* ***T****echnology* ***A****ttachment* ***P****acket* ***I****nterface* (Interfaz de Paquetes de Tecnología Avanzada de Fijación), es una extensión de la interfaz ATA y permite conectar medio removibles.

B: Lenguaje de programación creado en 1969 por Ken Thompson con contribuciones de Dennis Ritchie en los Laboratorios Bell, predecesor del lenguaje de programación C.

Bit: Acrónimo de ***B****inary* ***Digit*** (Dígito Binario), es un dígito del sistema de numeración binario.

Bluetooth: Es el nombre común de la especificación industrial IEEE 802.15.1, que define un estándar global de comunicación inalámbrica que posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia segura, globalmente y sin licencia de corto rango.

Boot-Loader: Es un firmware que permite la rápida descarga de programas en los microcontroladores. En el caso de los PIC, el boot-loader permite descargar programas directamente desde el PC sin necesidad de utilizar ningún tipo de grabador.

BSD: Acrónimo de Berkeley Software Distribution (Distribución de Software Berkeley) y se utiliza para identificar un sistema operativo derivado del sistema Unix nacido a partir de los aportes realizados a este sistema por la Universidad de California en Berkeley.

Byte: Unidad básica de almacenamiento de información que hace referencia a una secuencia de ocho bits contiguos. También se utiliza el termino octeto como sinónimo preciso de la cantidad de bits que representa.

ByteCode: Es un código intermedio más abstracto que el código máquina.

C: Lenguaje de programación creado en 1972 por Ken Thompson y Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B.

C++: Lenguaje de programación, diseñado a mediados de los años 1980, por Bjarne Stroustrup, como extensión del lenguaje de programación C. Se puede decir que C++ es un lenguaje que abarca tres paradigmas de la programación: la programación estructurada, la programación genérica y la programación orientada a objetos.

Callback: (Retro llamada) Es un código ejecutable que es pasado como argumento a otro código. Esto permite a una capa de software de más bajo nivel invocar una rutina (o función) definida en una capa de más alto nivel.

CDC: Acrónimo de ***C**omunications **D**evice **C**lass* (Clase de Dispositivo de Comunicaciones), es una de las clases que define el estándar USB y permite clasificar a los dispositivos según su comportamiento esperado.

Chip: Sinónimo de circuito integrado, es una pastilla muy delgada en la que se encuentran una enorme cantidad (del orden de miles o millones) de dispositivos microelectrónicos interconectados, principalmente diodos y transistores, además de componentes pasivos como resistencias o condensadores.

COM: Acrónimo de ***C**omponent **O**bject **M**odel* (Modelo de Objetos de Componentes), es una plataforma de Microsoft para componentes de software introducida en 1993. Permite la comunicación entre procesos y la creación dinámica de objetos en cualquier lenguaje de programación que soporte dicha tecnología.

CPU: Sigla de ***C**entral **P**rocessing **U**nit* (Unidad Central de Procesos), es el componente en una computadora digital que interpreta las instrucciones y procesa los datos contenidos en los programas de computadora. Es la parte que constituye el cerebro de cualquier computadora, es el encargado de realizar y dirigir todas las funciones.

CVS: Acrónimo de ***C**oncurrent **V**ersrions **S**ystem* (Sistema de Control de Versiones), es una aplicación informática que mantiene el registro de todo el trabajo y de los cambios los elementos (código fuente principalmente) que forman un proyecto (programa) y permite que distintas personas (desarrolladores) colaboren entre sí.

DAC: Sigla de ***D**igital **A**nalog **C**onversor* (Conversor digital-analógico), es un dispositivo para convertir datos digitales en señales de corriente o de tensión analógica.

DarwinBSD: Es un sistema que subyace en MacOS, cuya primera versión fue liberada en 2001 para funcionar en ordenadores Macintosh.

DIL: Véase DIP.

DIP: Acrónimo de ***D**ual **I**n-line **P**ackage*, es un tipo de circuito integrado alojado en una carcasa rectangular y de dos filas paralelas de pines de conexión eléctrica.

DLL: Acrónimo de ***D**ynamic **L**ink **L**ibrary* (Biblioteca de Vinculación Dinámica), por vinculación dinámica se entiende que las subrutinas de la biblioteca son cargadas en un programa de aplicación en tiempo de ejecución.

DMA: Acrónimo de ***D**irect **M**emory **A**ccess* (Acceso Directo a Memoria), es una característica de los computadores modernos, que permite a ciertos componentes de hardware dentro del computador, a acceder al sistema de memoria para lectura y/o escritura en forma independiente al CPU.

Driver: (Controlador) Programa que permite al sistema operativo interactuar con un dispositivo, haciendo una abstracción del hardware y proporcionando una interfaz -posiblemente estandarizada- para usarlo.

E/S: Abreviatura ***E**ntrada/ **S**alida*, es la colección de interfaces que usan las distintas unidades funcionales de un sistema de procesamiento de información para comunicarse con otras, o las señales enviadas a través de esas interfaces.

EEPROM: Acrónimo de ***E****lectrically* ***E****rasable* ***P****rogrammable* ***R****ead* ***O****nly* ***M****emory* (Memoria de Sólo Lectura Programable y Borrable Eléctricamente), es un tipo de memoria ROM que puede ser programado, borrado y reprogramado eléctricamente. Sólo puede ser borrada y reprogramada entre 100.000 y 1.000.000 de veces.

FIFO: Acrónimo de ***F****irst* ***I****n,* ***F****irst* ***O****ut* (Primero en Entrar, Primero en Salir), es un método utilizado en estructuras de datos, contabilidad de costes y teoría de colas. Guarda la analogía con las personas que esperan en una cola y van siendo atendidas en el orden en que llegan.

FireWire: Véase IEEE 1394.

Firmware: Es un bloque de instrucciones de programa para propósitos específicos, grabado en una memoria tipo ROM, que establece la lógica de más bajo nivel que controla los circuitos electrónicos de un dispositivo.

FreeBSD: Es un sistema operativo libre basado en los sistemas BSD para ordenadores personales basado en los CPU de arquitectura Intel.

Free_Software: (Software Libre) Es la denominación del software , que una vez obtenido, puede ser usado, copiado, estudiado, modificado, mejorado y redistribuido libremente de forma de beneficiar a toda la comunidad.

FSF: Acrónimo de ***F****ree* ***S****oftware* ***F****oundation* (Fundación para el Software Libre), es una organización creada en octubre de 1985 por Richard Stallman y otros entusiastas del Software Libre con el propósito de difundir este movimiento. Una de las principales funciones de la FSF es dar cobertura legal, económica y logística al Proyecto GNU.

GNU: Acrónimo recursivo de ***G****NU is* ***N****ot* ***U****nix* (GNU No es Unix), es un sistema operativo de computadores compuesto enteramente por software libre. Se lo conoce también como Proyecto GNU y fue iniciado por Richard Stallman en 1983.

GPL: Acrónimo de ***G****eneral* ***P****ublic* ***L****icense* (Licencia Pública General), es una licencia creada por la FSF a mediados de los años 80, y está orientada principalmente a proteger la libre distribución, modificación y uso de software.

HAL: Acrónimo de ***H****ardware* ***A****bstraction* ***L****ayer* (Capa de Abstracción de Hardware), es un elemento del sistema operativo que funciona como interfaz entre el software y el hardware del sistema, proveyendo una plataforma de hardware consistente sobre la cual correr las aplicaciones.

HEX: Su nombre formal es *Intel HEX* y corresponde a un formato de archivo para transmitir información binaria de aplicaciones programadas para microcontroladores, ROMs o cualquier otro tipo de chip. Éste formato es uno de los más viejos disponibles para este propósito y se usa desde los años 1970. El formato es un archivo de texto, donde cada línea contiene valores hexadecimales que codifican una secuencia de datos y sus corrimientos o direcciones absolutas.

HID: Acrónimo de ***H****uman* ***I****nterface* ***D****evices* (Dispositivo de Interfaz Humana), es un tipo de dispositivo de un computador, que interactúa directamente con seres humanos, tomando y/o devolviendo información.

Hot_Plug: (Enchufado en caliente) Véase Hot Swap.

Hot_Swap: (Sustitución en caliente) Hace referencia a la capacidad de algunos componentes hardware para realizar su instalación o sustitución sin necesidad de detener o alterar la operación normal de la computadora donde se alojan.

I/O: Abreviatura de ***I****nput/* ***O****utput*, véase E/S.

I2C: Acrónimo de ***I****nter-****I****ntegrated* ***C****ircuit* (Inter Circuitos Integrados), es un bus de comunicaciones serie diseñado por Phillips en el año 1992.

IDE: Acrónimo de ***I**ntegrated **D**evelopment **E**nvironment* (Entorno integrado de desarrollo), es un programa compuesto por un conjunto de herramientas (un editor de código, un compilador, un depurador y un constructor de interfaz gráfica) para brindar un marco de trabajo amigable a un programador.

IEEE_1394: Conocido como FireWire por Apple Inc. y como i.Link por Sony, es un estándar multiplataforma para entrada/salida de datos en serie a gran velocidad. Suele utilizarse para la interconexión de dispositivos digitales como cámaras digitales y videocámaras a computadoras.

IEEE_802: Es un comité y grupo de estudio de estándares perteneciente al Instituto de Ingenieros Eléctricos y Electrónicos (IEEE), que actúa sobre Redes de Ordenadores, concretamente y según su propia definición sobre redes de área local y redes de área metropolitana.

IRP: Acrónimo de ***I**/O **R**equest **P**acket* (Paquete de Petición E/S), es una estructura de modo kernel que es usada por WDM para comunicarse internamente y con el sistema operativo.

IOCTL: Acrónimo de ***I**nput/**O**utput **C**ontrol* (Control entrada/salida), la llamada de sistema ioctl en los sistemas basados en Unix, permite a una aplicación controlar o comunicarse con un driver de dispositivo fuera de los usuales read/write de datos.

ISR: Acrónimo de ***I**nterrupt **S**ervice **R**outine* (Rutina de Servicio de Interrupción), es una rutina (callback) en un sistema operativo o driver de dispositivo encargada de atender una interrupción del hardware. Su ejecución es disparada por la recepción de la interrupción.

Java: Lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 1990. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel como punteros.

JIT: Sigla de ***J**ust **I**n **T**ime* (Compilación en tiempo de ejecución), es una técnica para mejorar el rendimiento de sistemas de programación que compilan a bytecode, consistente en traducir el bytecode a código máquina nativo en tiempo de ejecución.

JNI: Sigla de ***J**ava **N**ative **I**nterface*, es un framework de programación que permite que un programa escrito en Java ejecutado en la máquina virtual Java pueda interactuar con programas escritos en otros lenguajes como C, C++ y ensamblador.

Jumper: Elemento para interconectar dos terminales de manera temporal sin tener que efectuar una operación que requiera herramienta adicional, dicha unión de terminales cierran el circuito eléctrico del que forma parte.

kB: (kilobyte) Es una unidad de información o de almacenamiento informático equivalente a mil bytes o 1.024 bytes, dependiendo el contexto. Puede ser abreviado como: K, KB, Kbytes.

Kernel: (Núcleo) Es la parte fundamental de un sistema operativo. Es el software encargado de gestionar los recursos del sistema.

kHz: (kilohertz) Unidad de frecuencia equivalente a mil ciclos por segundo.

LED: Sigla de ***L**ight **E**mitting **D**iode* (Diodo emisor de luz), es un dispositivo semiconductor (diodo) que emite luz cuasi-monocromática, es decir, con un espectro muy angosto, cuando se polariza de forma directa y es atravesado por una corriente eléctrica.

LGPL: Acrónimo de ***L**esser **G**eneral **P**ublic **L**icense* (Licencia Pública General Menor), es una licencia que aplica generalmente a bibliotecas y permite a programas privativos (no libres) utilizar las bibliotecas como parte de sus trabajos. Esta licencia mantiene términos más laxos que la GPL en que necesariamente todas las partes deben ser Software Libre.

Linux: Es la denominación de un sistema operativo tipo Unix (creado en 1992) y el nombre de un kernel (creado por Linus Torvalds en 1991). Es uno de los paradigmas más prominentes del Software Libre y del desarrollo del código abierto.

MacOS: Acrónimo de ***M**acintosh **O**perating **S**ystem* (Sistema Operativo de Macintosh), es el nombre del primer sistema operativo de Apple Computer para las computadoras Macintosh.

MB: (megabyte) Es una unidad de información o de almacenamiento informático equivalente a 10^6bytes o 2^{20} bytes dependiendo el contexto. Puede ser abreviado como Mbyte.

Mb: (megabit) Es una unidad de información o de almacenamiento informático equivalente un millón de bits. Puede ser abreviado como Mbit.

Mbps: (megabit per second) Es una unidad tasa de transferencia de datos equivalente a un millón de bits por segundo. Puede ser abreviado como Mbit/s o mbps).

MHz: (megahertz) Unidad de frecuencia que equivale a un millón de ciclos por segundo.

MIDI: Acrónimo de ***M**usical **I**nstrument **D**igital **I**nterface* (Interfaz Digital de Instrumentos Musicales). Es un protocolo estándar que permite a los computadores, sintetizadores, secuenciadores y otros dispositivos musicales electrónicos, comunicarse entre si para la generación de sonidos.

MPLAB: Es un editor IDE gratuito, destinado a productos de la marca Microchip. Este editor es modular, permite seleccionar los distintos microprocesadores soportados, además de permitir la grabación de estos circuitos integrados directamente al programador.

MPLAB_C18: Es un compilador C compatible con ANSI C y completo para la familia PIC18 de PICmicro de 8-bits.

MPLAB_ICD2: Depurador (con la característica de depuración directa en el circuito) y programador del firmware de los microcontroladores de la empresa Microchip.

NACK: Acrónimo de ***N**egative **Ack**nowledgement* (Asentimiento negativo), en comunicaciones entre computadoras, es un mensaje que se envía para informar de que en la recepción o procesamiento de los datos ha habido un error.

NetBSD: Es un sistema operativo del tipo Unix basado en los sistemas BSD, de distribución libre y de códigos fuentes abiertos.

Ohm: (Ohmnio) Es la unidad de resistencia eléctrica en el Sistema Internacional de Unidades. Su nombre se deriva del apellido del físico alemán Georg Simon Ohm, autor de la Ley de Ohm.

OpenBSD: Es un sistema operativo libre tipo Unix, multiplataforma, basado en los sistemas BSD. Es un descendiente de NetBSD, con un foco especial en la seguridad y la criptografía.

PC: Sigla de ***P**ersonal **C**omputer* (Computadora Personal), término genérico utilizado para referirse a microcomputadores que se basan en un microcontrolador Intel o compatible.

PCB: Sigla de ***P**rinted **C**ircuit **B**oard* (Circuito Impreso), es un medio para sostener mecánicamente y conectar eléctricamente componentes electrónicos, a través de rutas o pistas de material conductor, grabados desde hojas de cobre laminadas sobre un sustrato no conductor.

PIC: Son una familia de microcontroladores tipo RISC fabricados por Microchip Technology Inc. y derivados del PIC1650, originalmente desarrollado por la división de microelectrónica de General Instruments. El nombre actual no es un acrónimo, en realidad el nombre completo es PICmicro.

PID: Acrónimo de ***P**roduct **ID*** (Identificador de Producto), es un identificador único asignado por el USB-IF a los productos fabricados por las empresas registradas.

Plug_and_Play: (Enchufar y listo) Se refiere a la tecnología que permite a un dispositivo informático ser conectado a una computadora sin tener que configurar el mismo.

Plug-In: (Enchufar en) Pequeño programa que añade alguna función a otro programa, habitualmente de mayor tamaño. Un programa puede tener uno o más conectores.

Polling: (Escrutiño) Refiere a la toma de muestras en forma activa del estado de un dispositivo externo, por medio de un programa sincrónico. Este concepto se usa normalmente asociados a escrutinios de E/S.

Proxy: (Intermediario, Apoderado) En su forma más general, es una clase que funciona como interfaz de otra, logrando ocultar su verdadera identidad.

PWM: Sigla de ***P**ulse **W**idth **M**odulation* (Modulación en el ancho del pulso). Forma de codificar un valor análogo en una onda digital, de forma que el ciclo de trabajo está en relación con el valor a codificar.

Python: Es un lenguaje de programación (interpretado) que soporta múltiples paradigmas de programación (funcional, orientado a objetos y imperativo) y fue creado por Guido van Rossum en el año 1990.

QFN: Acrónimo de ***Q**uad **F**lat Package **N**o Lead* (Encapsulado Cuadrado Plano Sin Conectores), es un encapsulado de circuito integrado para montaje superficial donde los conectores de componentes no se extienden fuera del encapsulado.

RAM: Sigla de ***R**andom **A**ccess **M**emory* (Memoria de acceso aleatorio), es una memoria de semiconductores en la que se puede escribir o leer información. Es volátil, es decir, pierde su contenido al desconectar la energía eléctrica.

Ribbon_Cable: (Cable Cinta) También conocido como cable plano multihilo, es un cable con muchos cables conductores que corren en forma paralela unos con otros sobre el mismo plano. Como resultado, el cable es ancho y chato en vez de redondo. Su nombre proviene de la semejanza de estos cables con un pedazo de cinta. Son normalmente utilizados en periféricos internos de un computador personal.

ROM: Sigla de ***R**ead **O**nly **M**emory* (Memoria de sólo lectura), es una memoria de semiconductores que puede leerse pero no modificarse. La ROM suele almacenar la configuración del sistema o el programa de arranque de la computadora.

RTC: Acrónimo de ***R**eal **T**ime **C**lock* (Reloj de Tiempo Real), es un reloj un computador (la mayoría de las veces en forma de circuito integrado) que permite mantener la pista del tiempo actual.

SIE: Acrónimo de ***S**erial **I**nterface **E**ngine* (Motor de Interfaz Serial), es un circuito integrado que normalmente se encarga de la recepción y envío de datos de las transacciones y trabaja en conjunto con el transceiver USB y los endpoints. El SIE no interpreta o utiliza los datos, sólo envía los datos que están disponibles para hacerlo y almacena los datos recibidos.

SIL: Acrónimo de ***S**ingle **I**n-**L**ine Package*, es un tipo de circuito integrado alojado en una carcasa que posee una única fila de pines de conexión eléctrica.

SMT: Sigla de ***S**urface **M**ount **T**echnology* (Tecnología de montaje superficial), es un método para la construcción de circuitos electrónicos en el que los componentes se montan directamente sobre la superficie de los circuito impreso.

Solaris: Es un sistema operativo desarrollado por Sun Microsystems. Es un sistema certificado como una versión de Unix. Aunque Solaris en sí mismo aún es software propietario, la parte principal del sistema operativo se ha liberado como Software Libre.

SPI: Sigla de ***S**erial **P**eripherical **I**nterface* (Bus de Interfaz de Periféricos Serie), es un estándar de comunicaciones, usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos.

TAD: Acrónimo de ***T**ipo **A**bstracto de **D**atos* (Abstract Data Type), es una especificación de un conjunto de datos y un conjunto de operaciones que pueden ser realizadas sobre los datos.

Through-Hole_Technology: (Tecnología A Través de Orificio) Se refiere al esquema utilizado para el montaje de componentes electrónicos, que implica la inserción de los pines de los componentes en orificios perforados en el circuito electrónico impreso y el soldado al circuito en el lado opuesto.

TimeOut: (Tiempo vencido) Hace referencia a la conclusión intencional de una tarea incompleta luego de superado un tiempo límite estipulado para su conclusión en forma normal.

Transceivers: (Transceptores) Es un dispositivo que realiza funciones tanto de transmición como de recepción, utilizando componentes de circuito comunes para ambas funciones.

UART: Acrónimo de ***U****niversal* ***A****synchronous* ***R****eceiver* ***T****ransmitter* (Transmisor Receptor Asíncrono Universal), es un circuito integrado (o parte de uno) usado para comunicaciones seriales en computadores que forma parte de los puertos seriales de computadores personales o periféricos.

Unix: Es un sistema operativo portable, multitarea y multiusuario; desarrollado en principio por un grupo de empleados de los laboratorios Bell de AT&T, entre los que figuran Ken Thompson, Dennis Ritchie y Douglas McIlroy.

URB: Acrónimo de ***U****SB* ***R****equest* ***B****lock* (Petición en Bloque USB), son estructuras utilizadas a nivel de los sistemas operativos, que permiten la configuración de los dispositivos y transmición de datos. Estas estructuras son enviadas por medio de un IRP a la capas inferiores de la pila de drivers.

USB: Sigla de ***U****niversal* ***S****erial* ***B****us* (Bus Universal en Serie), es una interfaz que provee un estándar de bus serie para conectar dispositivos a un ordenador personal. Fue creado en 1996 por IBM, Intel, Northern Telecom, Compaq, Microsoft, Digital Equipment Corporation y NEC.

USB_Hub: (Concentrador USB) Es un dispositivo que permite tener varios puertos USB a partir de uno sólo.

VID: Acrónimo de ***V****endor* ***ID*** (Identificador de Vendedor), es un identificador único asignado por el USB-IF a las empresas (previo registro) que quieren fabricar dispositivos USB.

WiFi: Acrónimo de ***Wi****reless* ***Fi****delity*. Es un conjunto de estándares para redes inalámbricas basados en las especificaciones IEEE 802.11.

WINE: Acrónimo recursivo de ***W****ine* ***I****s* ***N****ot an* ***E****mulator* (Wine no es un emulador), es una reimplementación de la API de Windows (en sus versiones de 16-bits y 32-bits) para sistemas operativos basados en Unix bajo plataformas Intel.

Windows: (Microsoft Windows) Es un sistema operativo con interfaz gráfica para computadoras personales propiedad de la empresa Microsoft.

Wireless_USB: (USB Inalámbrico) Es un protocolo de comunicación inalámbrico por radio con gran ancho de banda que combina la sencillez de uso de USB con la versatilidad de las redes inalámbricas. En mayo del 2006 se aprobó la especificación del nuevo estándar que se abrevia como W-USB o WUSB.

Wrapper: (Envoltura) Es una clase que se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

Apéndice B

Herramientas de la Gestión del Proyecto

B.1. LyX

Lyx [26] es un programa gráfico multiplataforma creado por Matthias Ettrich que permite la edición de texto usando LaTeX, por lo que «hereda» todas sus capacidades (notación científica, edición de ecuaciones, creación de índices, etcétera). Se trata de un procesador de textos en el que el usuario no necesita pensar en el formato final de su trabajo, sino sólo en el contenido y su estructura (WYSIWYM)(Lo Que Ve Es Lo Que Quieres Decir, por sus siglas en Inglés), por lo que puede ser utilizado para editar documentos grandes (libros) o con formato riguroso (tesis, artículos para revistas científicas), con facilidad. La decisión de utilizar esta herramienta se debió más allá de todas las características previamente descritas, a que su formato es texto y no binario, lo que permite el trabajo en forma concurrente sobre el documento, así como la utilización de herramientas de control de versión como es CVS.

B.2. CVS

CVS [11] utiliza una arquitectura cliente-servidor: un servidor guarda la(s) versión(es) actual(es) del proyecto y su historial. Los clientes se conectan al servidor para sacar una copia completa del proyecto. Esto se hace para que eventualmente puedan trabajar con esa copia y más tarde ingresar sus cambios con comandos GNU.

Típicamente, cliente y servidor se conectan utilizando Internet, pero con el sistema CVS el cliente y servidor pueden estar en la misma máquina. El sistema CVS tiene la tarea de mantener el registro de la historia de las versiones del programa de un proyecto solamente con desarrolladores locales. Originalmente, el servidor utilizaba un sistema operativo similar a Unix, aunque en la actualidad existen versiones de CVS en otros sistemas operativos, incluido Windows. Los clientes CVS pueden funcionar en cualquiera de los sistemas operativos más difundidos.

Varios clientes pueden sacar copias del proyecto al mismo tiempo. Posteriormente, cuando actualizan sus modificaciones, el servidor trata de acoplar las diferentes versiones. Si esto falla, por ejemplo debido a que dos clientes tratan de cambiar la misma línea en un archivo en particular, entonces el servidor deniega la segunda actualización e informa al cliente sobre el conflicto, que el usuario deberá resolver manualmente. Si la operación de ingreso tiene éxito, entonces los números de versión de todos los archivos implicados se incrementan automáticamente, y el servidor CVS almacena información sobre la actualización, que incluye una descripción suministrada por el usuario, la fecha y el nombre del autor y sus archivos log.

En el contexto de este proyecto fue prioritario la utilización de este tipo de herramienta, de forma de permitir a cada integrante del grupo trabajar en sus casas optimizando los tiempos sin perder el control del código generado.

B.3. Mantis

Mantis es una aplicación web que permite el reporte de errores y su posterior seguimiento. Está escrito en PHP y requiere una base de datos (MySQL, MS SQL, PostgreSQL son soportados) y un servidor web. Mantis puede ser instalado sobre las plataformas Windows, Mac OS, OS/2 y una variedad de sistemas operativos Unix. Casi cualquier navegador web debería ser capaz de funcionar como cliente. La interfaz de usuario utiliza un código de colores para distintas cuestiones que facilitan al usuario reconocer de una sola mirada los estados de varios ítems. También posee una gran cantidad de opciones de ordenamiento y filtrado para facilitar la ubicación de un problema en particular.

Durante la etapa de implementación y verificación fue utilizado para ir reportando errores [28] y asignando responsables de investigarlos y corregirlos, logrando de esta forma centralizar la información de los errores y no dejando lugar para olvidos.

B.4. Wiki

Un wiki es un sitio web colaborativo que puede ser editado por varios usuarios. Los usuarios de una wiki pueden así crear, modificar, borrar el contenido de una página web, de forma interactiva, fácil y rápida; dichas facilidades hacen de la wiki una herramienta efectiva para la escritura colaborativa.

La tecnología wiki permite que páginas web alojadas en un servidor público (las páginas wiki) sean escritas de forma colaborativa a través de un navegador web, utilizando una notación sencilla para dar formato, crear enlaces, etc, conservando un historial de cambios que permite recuperar fácilmente cualquier estado anterior de la página. Cuando alguien edita una página wiki, sus cambios aparecen inmediatamente en la web, sin pasar por ningún tipo de revisión previa.

En el contexto de este proyecto se utilizo un wiki propia [33], que oficio de sitio web del proyecto y facilito toda la gestión de publicación de documentación, páginas web de sitios de interés para el proyecto. Además, fue muy utilizado para registrar de una forma ágil y de rápido acceso las planificaciones y actas de las reuniones que se fueron realizando durante todo el proceso.