Java y USB

En este documento se presentará la definición del lenguaje de programación Java, la definición del estándar USB, y la forma de lograr una comunicación o interacción entre ellos.

Jesús David Meneses Sánchez
Juan David Marín Martínez

30/05/2008

CONTENIDO

1 JAVA 4

1.1 Historia 4

1.2 Filosofía 7

1.2.1 Orientado a Objetos 7

1.2.2 Independencia de la plataforma 8

1.2.3 El recolector de basura 10

1.3 Entornos de funcionamiento 10

1.3.1 En dispositivos móviles y sistemas empotrados 11

1.3.2 En el navegador web 11

1.3.3 En sistemas de servidor 11

1.3.4 En aplicaciones de escritorio 12

1.3.5 Plataformas soportadas 12

1.4 Recursos 13

1.4.1 JRE 13

2 USB 15

2.1 Componentes Del Sistema De Bus Serie Universal USB 16

2.1.1 Controlador 16

2.1.2 Concentradores o Hubs 16

2.1.3 Periféricos 17

2.2 Características de Transmisión 18

2.3 La Topología del Bus 19

2.3.1 La Capa Física 21

2.3.2 La Capa Lógica 22

2.3.3 La relación "software del cliente-función" 23

2.4 El flujo de datos del bus USB 23

2.4.1 Endpoints y direcciones de dispositivo 24

2.4.2 Tuberías 24

2.4.3 Frames y microframes 26

2.4.4 Sync 26

2.4.5 Tipos de transferencias 26

2.5 Capa de protocolo 27
2.5.1 Formato de los campos 29
2.5.2 Codificación de datos 30
2.5.3 Relleno de bits 30
2.5.4 Formato de los paquetes 30
2.5.5 Transacciones 32
2.5.6 Split 34
2.5.7 El protocolo y las transferencias 36
2.6 La eléctrica 38
2.6.1 Identificación de la velocidad del dispositivo 38
2.7 La mecánica 39
2.7.1 Cable estándar de quita y pon 39
2.7.2 Cable fijo de velocidad alta y media 40
2.7.3 Cable fijo de velocidad baja 40
2.8 Estándar IEEE 1394 o FireWire. 41
3 COMUNICACIÓN USB A TRAVÉS DE JAVA 42
3.1 JSR-80 API 42
3.2 API jUSB 43

1 JAVA

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. 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, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

Las aplicaciones Java están típicamente compiladas en un bytecode, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el bytecode es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del bytecode por un procesador Java también es posible.

La implementación original y de referencia del compilador, la máquina virtual y las librerías de clases de Java fueron desarrolladas por Sun Microsystems en 1995. Desde entonces, Sun ha controlado las especificaciones, el desarrollo y evolución del lenguaje a través del Java Community Process, si bien otros han desarrollado también implementaciones alternativas de estas tecnologías de Sun, algunas incluso bajo licencias de software libre.

Entre noviembre de 2006 y mayo de 2007, Sun Microsystems liberó la mayor parte de sus tecnologías Java bajo la licencia GNU GPL, de acuerdo con las especificaciones del Java Community Process, de tal forma que prácticamente todo el Java de Sun es ahora software libre (aunque la biblioteca de clases de Sun que se requiere para ejecutar los programas Java todavía no es software libre).

1.1 Historia

La tecnología Java se creó como una herramienta de programación para ser usada en un proyecto de set-top-box en una pequeña operación denominada the Green Project en Sun Microsystems en el año 1991. El equipo (Green Team), compuesto por trece personas y dirigido por James Gosling, trabajó durante 18 meses en Sand Hill Road en Menlo Park en su desarrollo.

El lenguaje se denominó inicialmente Oak (por un roble que había fuera de la oficina de Gosling), luego pasó a denominarse Green tras descubrir que Oak era ya una marca comercial registrada para adaptadores de tarjetas gráficas y finalmente se renombró a Java.

Los objetivos de Gosling eran implementar una máquina virtual y un lenguaje con una estructura y sintaxis similar a C++. Entre junio y julio de 1994, tras una sesión maratónica de tres días entre John Gaga, James Gosling, Joy Naughton, Wayne Rosing y Eric Schmidt, el equipo reorientó la plataforma hacia la Web. Sintieron que la llegada del navegador web Mosaic, propiciaría que Internet se convirtiese en un medio interactivo, como el que pensaban era la televisión por cable.

Naughton creó entonces un prototipo de navegador, WebRunner, que más tarde sería conocido como HotJava.

En 1994, se les hizo una demostración de HotJava y la plataforma Java a los ejecutivos de Sun. Java 1.0a pudo descargarse por primera vez en 1994, pero hubo que esperar al 23 de mayo de 1995, durante las conferencias de SunWorld, a que vieran la luz pública Java y HotJava, el navegador Web. El acontecimiento fue anunciado por John Gage, el Director Científico de Sun Microsystems. El acto estuvo acompañado por una pequeña sorpresa adicional, el anuncio por parte de Marc Andreessen, Vicepresidente Ejecutivo de Netscape, que Java sería soportado en sus navegadores. El 9 de enero del año siguiente, 1996, Sun fundó el grupo empresarial JavaSoft para que se encargase del desarrollo tecnológico. Dos semanas más tarde la primera versión de Java fue publicada.

La promesa inicial de Gosling era Write Once, Run Anywhere (Escríbelo una vez, ejecútalo en cualquier lugar), proporcionando un lenguaje independiente de la plataforma y un entorno de ejecución (la JVM) ligero y gratuito para las plataformas más populares de forma que los binarios (bytecode) de las aplicaciones Java pudiesen ejecutarse en cualquier plataforma.

El entorno de ejecución era relativamente seguro y los principales navegadores web pronto incorporaron la posibilidad de ejecutar applets Java incrustadas en las páginas web.

Java ha experimentado numerosos cambios desde la versión primigenia, JDK 1.0, así como un enorme incremento en el número de clases y paquetes que componen la librería estándar.

Desde J2SE 1.4, la evolución del lenguaje ha sido regulada por el JCP (Java Community Process), que usa Java Specification Requests (JSRs) para proponer y especificar cambios en la plataforma Java. El lenguaje en sí mismo está especificado en la Java Language Specification (JLS), o Especificación del Lenguaje Java. Los cambios en los JLS son gestionados en JSR 901.

- JDK 1.0 (23 de enero de 1996) — Primer lanzamiento.
- JDK 1.1 (19 de febrero de 1997) — Principales adiciones incluidas. Una reestructuración intensiva del modelo de eventos AWT (Abstract Windowing Toolkit). Clases internas (inner classes). JavaBeans. JDBC (Java Database Connectivity), para la integración de bases de datos. RMI (Remote Method Invocation)
- J2SE 1.2 (8 de diciembre de 1998) — Nombre clave Playground. Esta y las siguientes versiones fueron recogidas bajo la denominación Java 2 y el nombre "J2SE" (Java 2 Platform, Standard Edition), reemplazó a JDK para distinguir la plataforma base de J2EE (Java 2 Platform, Enterprise Edition) y J2ME (Java 2 Platform, Micro Edition). Otras mejoras añadidas incluían: la palabra reservada (keyword) strictfp, reflexión en la programación la API gráfica (Swing) fue integrada en las clases básicas. La máquina virtual (JVM) de Sun fue equipada con

un compilador JIT (Just in Time) por primera vez. Java Plug-in. Java IDL, una implementación de IDL (Interfaz para Descripción de Lenguaje) para la interoperabilidad con CORBA. Colecciones (Collections)

- J2SE 1.3 (8 de mayo de 2000) — Nombre clave Kestrel. Cambios notables: la inclusión de la máquina virtual de HotSpot JVM (la JVM de HotSpot fue lanzada inicialmente en abril de 1999, para la JVM de J2SE 1.2). RMI fue cambiado para que se basara en CORBA. JavaSound. Se incluyó el Java Naming and Directory Interface (JNDI) en el paquete de librerías principales (anteriormente disponible como una extensión). Java Platform Debugger Architecture (JPDA).
- J2SE 1.4 (6 de febrero de 2002) — Nombre Clave Merlin. Este fue el primer lanzamiento de la plataforma Java desarrollado bajo el Proceso de la Comunidad Java como JSR 59. Los cambios más notables fueron: Palabra reservada assert (Especificado en JSR 41). Expresiones regulares modeladas al estilo de las expresiones regulares Perl. Encadenación de excepciones Permite a una excepción encapsular la excepción de bajo nivel original. Non-blocking NIO (New Input/Output) (Especificado en JSR 51.). Logging API (Specified in JSR 47). API I/O para la lectura y escritura de imágenes en formatos como JPEG o PNG. Parser XML integrado y procesador XSLT (JAXP) (Especificado en JSR 5 y JSR 63). Seguridad integrada y extensiones criptográficas (JCE, JSSE, JAAS). Java Web Start incluido (El primer lanzamiento ocurrió en Marzo de 2001 para J2SE 1.3) (Especificado en JSR 56)
- J2SE 5.0 (30 de septiembre de 2004) — Nombre clave: Tiger. (Originalmente numerado 1.5, esta notación aún es usada internamente) Desarrollado bajo JSR 176, Tiger añadió un número significativo de nuevas características comunicado de prensa. Plantillas (genéricos) — provee conversión de tipos (type safety) en tiempo de compilación para colecciones y elimina la necesidad de la mayoría de conversión de tipos (type casting). (Especificado por JSR 14). Metadatos — también llamados anotaciones, permite a estructuras del lenguaje como las clases o los métodos, ser etiquetados con datos adicionales, que puedan ser procesados posteriormente por utilidades de proceso de metadatos. (Especificado por JSR 175). Autoboxing/unboxing — Conversiones automáticas entre tipos primitivos (Como los int) y clases de envoltura primitivas (Como Integer). (Especificado por JSR 201). Bucle “for” mejorado.
- Java SE 6 (11 de diciembre de 2006) — Nombre clave Mustang. Estuvo en desarrollo bajo la JSR 270. En esta versión, Sun cambió el nombre "J2SE" por Java SE y eliminó el ".0" del número de versión. Los cambios más importantes introducidos en esta versión son Incluye un nuevo marco de trabajo y APIs que hacen posible la combinación de Java con lenguajes dinámicos como PHP, Python, Ruby y JavaScript. Incluye el motor Rhino, de Mozilla, una implementación de Javascript en Java. Incluye un cliente completo de Servicios Web y soporta las

últimas especificaciones para Servicios Web, como JAX-WS 2.0, JAXB 2.0, STAX y JAXP. Mejoras en la interfaz gráfica y en el rendimiento.

- Java SE 7 — Nombre clave Dolphin. En el año 2006 aún se encontraba en las primeras etapas de planificación. Se espera que su desarrollo dé comienzo en la primavera de 2006, y se estima su lanzamiento para 2008. Soporte para XML dentro del propio lenguaje. Un nuevo concepto de súper paquete. Soporte para closures. Introducción de anotaciones estándar para detectar fallos en el software.

Además de los cambios en el lenguaje, con el paso de los años se han efectuado muchos más cambios dramáticos en la librería de clases de Java (Java class library) que ha crecido de unos pocos cientos de clases en JDK 1.0 hasta más de tres mil en J2SE 5.0. APIs completamente nuevas, como Swing y Java2D, han sido introducidas y muchos de los métodos y clases originales de JDK 1.0 están obsoletos.

Entre noviembre de 2006 y mayo de 2007, Sun Microsystems liberó la mayor parte de sus tecnologías Java bajo la licencia GNU GPL, de acuerdo con las especificaciones del Java Community Process, de tal forma que prácticamente todo el Java de Sun es ahora software libre (aunque la biblioteca de clases de Sun que se requiere para ejecutar los programas Java todavía no es software libre).

1.2 Filosofía

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar la metodología de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería incluir por defecto soporte para trabajo en red.
4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

Para conseguir la ejecución de código remoto y el soporte de red, los programadores de Java a veces recurren a extensiones como CORBA (Common Object Request Broker Architecture), Internet Communications Engine u OSGi respectivamente.

1.2.1 Orientado a Objetos

La primera característica, orientado a objetos ("OO"), se refiere a un método de programación y al diseño del lenguaje. Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que use estén unidos a sus operaciones. Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Un objeto puede verse como un paquete que contiene el "comportamiento" (el código) y el "estado" (datos). El principio es separar aquello que cambia de las cosas que permanecen inalterables.

Frecuentemente, cambiar una estructura de datos implica un cambio en el código que opera sobre los mismos, o viceversa. Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software. El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos. Otra de las grandes promesas de la programación orientada a objetos es la creación de entidades más genéricas (objetos) que permitan la reutilización del software entre proyectos, una de las premisas fundamentales de la Ingeniería del Software. Un objeto genérico "cliente", por ejemplo, debería en teoría tener el mismo conjunto de comportamiento en diferentes proyectos, sobre todo cuando estos coinciden en cierta medida, algo que suele suceder en las grandes organizaciones. En este sentido, los objetos podrían verse como piezas reutilizables que pueden emplearse en múltiples proyectos distintos, posibilitando así a la industria del software a construir proyectos de envergadura empleando componentes ya existentes y de comprobada calidad; conduciendo esto finalmente a una reducción drástica del tiempo de desarrollo. Podemos usar como ejemplo de objeto el aluminio. Una vez definidos datos (peso, maleabilidad, etc.), y su "comportamiento" (soldar dos piezas, etc.), el objeto "aluminio" puede ser reutilizado en el campo de la construcción, del automóvil, de la aviación, etc.

La reutilización del software ha experimentado resultados dispares, encontrando dos dificultades principales: el diseño de objetos realmente genéricos es pobremente comprendido, y falta una metodología para la amplia comunicación de oportunidades de reutilización. Algunas comunidades de "código abierto" (open source) quieren ayudar en este problema dando medios a los desarrolladores para diseminar la información sobre el uso y versatilidad de objetos reutilizables y librerías de objetos.

1.2.2 Independencia de la plataforma

La segunda característica, la independencia de la plataforma, significa que programas escritos en el lenguaje Java pueden ejecutarse igualmente en cualquier tipo de hardware. Es lo que significa ser capaz de escribir un programa una vez y que pueda ejecutarse en cualquier dispositivo, tal como reza el axioma de Java, "'write once, run everywhere"'.

Para ello, se compila el código fuente escrito en lenguaje Java, para generar un código conocido como "bytecode" (específicamente Java bytecode) —instrucciones máquina simplificadas específicas de la plataforma Java. Esta pieza está "a medio camino" entre el código fuente y el código máquina que entiende el dispositivo destino. El bytecode es ejecutado entonces en la máquina virtual (VM), un programa escrito en código nativo de la plataforma destino (que es el que entiende su hardware), que interpreta y ejecuta el código. Además, se suministran librerías adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hilos o threads, la interfaz de red) de forma unificada. Se debe tener presente que, aunque hay una etapa explícita de compilación, el bytecode generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT (Just In Time).

Hay implementaciones del compilador de Java que convierten el código fuente directamente en código objeto nativo, como GCJ. Esto elimina la etapa intermedia donde se genera el bytecode, pero la salida de este tipo de compiladores sólo puede ejecutarse en un tipo de arquitectura.

La licencia sobre Java de Sun insiste que todas las implementaciones sean “compatibles”. Esto dio lugar a una disputa legal entre Microsoft y Sun, cuando éste último alegó que la implementación de Microsoft no daba soporte a las interfaces RMI y JNI además de haber añadido características ‘’dependientes’’ de su plataforma. Sun demandó a Microsoft y ganó por daños y perjuicios (unos 20 millones de dólares) así como una orden judicial forzando la acatación de la licencia de Sun. Como respuesta, Microsoft no ofrece Java con su versión de sistema operativo, y en recientes versiones de Windows, su navegador Internet Explorer no admite la ejecución de applets sin un conector (o plugin) aparte. Sin embargo, Sun y otras fuentes ofrecen versiones gratuitas para distintas versiones de Windows.

Las primeras implementaciones del lenguaje usaban una máquina virtual interpretada para conseguir la portabilidad. Sin embargo, el resultado eran programas que se ejecutaban comparativamente más lentos que aquellos escritos en C o C++. Esto hizo que Java se ganase una reputación de lento en rendimiento. Las implementaciones recientes de la JVM dan lugar a programas que se ejecutan considerablemente más rápido que las versiones antiguas, empleando diversas técnicas, aunque sigue siendo mucho mas lento que otros lenguajes.

La primera de estas técnicas es simplemente compilar directamente en código nativo como hacen los compiladores tradicionales, eliminando la etapa del bytecode. Esto da lugar a un gran rendimiento en la ejecución, pero tapa el camino a la portabilidad. Otra técnica, conocida como compilación JIT (Just In Time, o ‘’’compilación al vuelo’’’), convierte el bytecode a código nativo cuando se ejecuta la aplicación. Otras máquinas virtuales más sofisticadas usan una ‘’’recompilación dinámica’’’ en la que la VM es capaz de analizar el comportamiento del programa en ejecución y recompila y optimiza las partes críticas. La recompilación dinámica puede lograr mayor grado de optimización que la compilación tradicional (o estática), ya que puede basar su trabajo en el conocimiento que de primera mano tiene sobre el entorno de ejecución y el conjunto de clases cargadas en memoria. La compilación JIT y la recompilación dinámica permiten a los programas Java aprovechar la velocidad de ejecución del código nativo sin por ello perder la ventaja de la portabilidad.

La portabilidad es técnicamente difícil de lograr, y el éxito de Java en ese campo ha sido dispar. Aunque es de hecho posible escribir programas para la plataforma Java que actúen de forma correcta en múltiples plataformas de distinta arquitectura, el gran número de estas con pequeños errores o inconsistencias llevan a que a veces se parodie el eslogan de Sun, "Write once, run anywhere" como "Write once, debug everywhere" (o “Escríbelo una vez, ejecútalo en cualquier parte” por “Escríbelo una vez, depúralo en todas partes”)

El concepto de independencia de la plataforma de Java cuenta, sin embargo, con un gran éxito en las aplicaciones en el entorno del servidor, como los Servicios Web, los Servlets, los Java Beans, así como en sistemas empotrados basados en OSGi, usando entornos Java empotrados.

1.2.3 El recolector de basura

Un argumento en contra de lenguajes como C++ es que los programadores se encuentran con la carga añadida de tener que administrar la memoria solicitada dinámicamente de forma manual:

En C++, el desarrollador puede asignar memoria en una zona conocida como heap (montículo) para crear cualquier objeto, y posteriormente desalojar el espacio asignado cuando desea borrarlo. Un olvido a la hora de desalojar memoria previamente solicitada puede llevar a una fuga de memoria, ya que el sistema operativo seguirá pensando que esa zona de memoria está siendo usada por una aplicación cuando en realidad no es así. Así, un programa mal diseñado podría consumir una cantidad desproporcionada de memoria. Además, si una misma región de memoria es desalojada dos veces el programa puede volverse inestable y llevar a un eventual cuelgue. No obstante, se debe señalar que C++ también permite crear objetos en la pila de llamadas de una función o bloque, de forma que se libere la memoria (y se ejecute el destructor del objeto) de forma automática al finalizar la ejecución de la función o bloque.

En Java, este problema potencial es evitado en gran medida por el recolector automático de basura (o automatic garbage collector). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos. El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste (que, desde un punto de vista de bajo nivel es una dirección de memoria). Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aún así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y puede que más rápida que en C++.

La recolección de basura de Java es un proceso prácticamente invisible al desarrollador. Es decir, el programador no tiene conciencia de cuándo la recolección de basura tendrá lugar, ya que ésta no tiene necesariamente que guardar relación con las acciones que realiza el código fuente.

Debe tenerse en cuenta que la memoria es sólo uno de los muchos recursos que deben ser gestionados.

1.3 Entornos de funcionamiento

El diseño de Java, su robustez, el respaldo de la industria y su fácil portabilidad han hecho de Java uno de los lenguajes con un mayor crecimiento y amplitud de uso en distintos ámbitos de la industria de la informática.

1.3.1 En dispositivos móviles y sistemas empotrados

Desde la creación de la especificación J2ME (Java 2 Platform, Micro Edition), una versión del entorno de ejecución Java reducido y altamente optimizado, especialmente desarrollado para el mercado de dispositivos electrónicos de consumo se ha producido toda una revolución en lo que a la extensión de Java se refiere.

Es posible encontrar microprocesadores específicamente diseñados para ejecutar bytecode Java y software Java para tarjetas inteligentes (JavaCard), teléfonos móviles, buscapersonas, set-top-boxes, sintonizadores de TV y otros pequeños electrodomésticos.

El modelo de desarrollo de estas aplicaciones es muy semejante a las applets de los navegadores salvo que en este caso se denominan MIDlets.

Véase Sun Mobile Device Tecnology

1.3.2 En el navegador web

Desde la primera versión de java existe la posibilidad de desarrollar pequeñas aplicaciones (Applets) en Java que luego pueden ser incrustadas en una página HTML para que sean descargadas y ejecutadas por el navegador web. Estas mini-aplicaciones se ejecutan en una JVM que el navegador tiene configurada como extensión (plug-in) en un contexto de seguridad restringido configurable para impedir la ejecución local de código potencialmente malicioso.

El éxito de este tipo de aplicaciones (la visión del equipo de Gosling) no fue realmente el esperado debido a diversos factores, siendo quizás el más importante la lentitud y el reducido ancho de banda de las comunicaciones en aquel entonces que limitaba el tamaño de las applets que se incrustaban en el navegador. La aparición posterior de otras alternativas (aplicaciones web dinámicas de servidor) dejó un reducido ámbito de uso para esta tecnología, quedando hoy relegada fundamentalmente a componentes específicos para la intermediación desde una aplicación web dinámica de servidor con dispositivos ubicados en la máquina cliente donde se ejecuta el navegador.

Las applets Java no son las únicas tecnologías (aunque sí las primeras) de componentes complejos incrustados en el navegador. Otras tecnologías similares pueden ser: ActiveX de Microsoft, Flash, Java Web Start, etc.

1.3.3 En sistemas de servidor

En la parte del servidor, Java es más popular que nunca, desde la aparición de la especificación de Servlets y JSP (Java Server Pages).

Hasta entonces, las aplicaciones web dinámicas de servidor que existían se basaban fundamentalmente en componentes CGI y lenguajes interpretados. Ambos tenían diversos inconvenientes (fundamentalmente lentitud, elevada carga computacional o de memoria y propensión a errores por su interpretación dinámica).

Los servlets y las JSPs supusieron un importante avance ya que:

- El API de programación es muy sencilla, flexible y extensible.
- Los servlets no son procesos independientes (como los CGIs) y por tanto se ejecutan dentro del mismo proceso que la JVM mejorando notablemente el rendimiento y reduciendo la carga computacional y de memoria requeridas.
- Las JSPs son páginas que se compilan dinámicamente (o se pre-compilan previamente a su distribución) de modo que el código que se consigue una ventaja en rendimiento substancial frente a muchos lenguajes interpretados.

La especificación de Servlets y JSPs define un API de programación y los requisitos para un contenedor (servidor) dentro del cual se puedan desplegar estos componentes para formar aplicaciones web dinámicas completas. Hoy día existen multitud de contenedores (libres y comerciales) compatibles con estas especificaciones.

A partir de su expansión entre la comunidad de desarrolladores, estas tecnologías han dado paso a modelos de desarrollo mucho más elaborados con frameworks (pe Struts, Webwork) que se sobreponen sobre los servlets y las JSPs para conseguir un entorno de trabajo mucho más poderoso y segmentado en el que la especialización de roles sea posible (desarrolladores, diseñadores gráficos, ...) y se facilite la reutilización y robustez de código. A pesar de todo ello, las tecnologías que subyacen (Servlets y JSPs) son substancialmente las mismas.

Este modelo de trabajo se ha convertido en un estándar de-facto para el desarrollo de aplicaciones web dinámicas de servidor y otras tecnologías (pe. ASP) se han basado en él.

1.3.4 En aplicaciones de escritorio

Hoy en día existen multitud de aplicaciones gráficas de usuario basadas en Java. El entorno de ejecución Java (JRE) se ha convertido en un componente habitual en los PCs de usuario de los sistemas operativos más usados en el mundo. Además, muchas aplicaciones Java lo incluyen dentro del propio paquete de la aplicación de modo que su ejecución en cualquier PC.

En las primeras versiones de la plataforma Java existían importantes limitaciones en las APIs de desarrollo gráfico (AWT). Desde la aparición de la librería Swing la situación mejoró substancialmente y posteriormente con la aparición de librerías como SWT hacen que el desarrollo de aplicaciones de escritorio complejas y con gran dinamismo, usabilidad, etc. sea relativamente sencillo.

1.3.5 Plataformas soportadas

Una versión del entorno de ejecución Java JRE (Java Runtime Environment) está disponible en la mayoría de equipos de escritorio. Sin embargo, Microsoft no lo ha incluido por defecto en sus sistemas operativos. En el caso de Apple, éste incluye una versión propia del JRE en su sistema operativo, el Mac OS. También es un producto que por defecto aparece en la mayoría de las distribuciones de Linux. Debido a incompatibilidades entre distintas versiones del JRE, muchas aplicaciones prefieren instalar su propia copia del JRE antes que confiar su suerte a la aplicación

instalada por defecto. Los desarrolladores de applets de Java o bien deben insistir a los usuarios en la actualización del JRE, o bien desarrollar bajo una versión antigua de Java y verificar el correcto funcionamiento en las versiones posteriores.

1.4 Recursos

1.4.1 JRE

El JRE (Java Runtime Environment, o Entorno en Tiempo de Ejecución de Java) es el software necesario para ejecutar cualquier aplicación desarrollada para la plataforma Java. El usuario final usa el JRE como parte de paquetes software o plugins (o conectores) en un navegador Web. Sun ofrece también el SDK de Java 2, o JDK (Java Development Kit) en cuyo seno reside el JRE, e incluye herramientas como el compilador de Java, Javadoc para generar documentación o el depurador. Puede también obtenerse como un paquete independiente, y puede considerarse como el entorno necesario para ejecutar una aplicación Java, mientras que un desarrollador debe además contar con otras facilidades que ofrece el JDK.

Componentes [editar]

Bibliotecas de Java, que son el resultado de compilar el código fuente desarrollado por quien implementa la JRE, y que ofrecen apoyo para el desarrollo en Java. Algunos ejemplos de estas librerías son:

Las bibliotecas centrales, que incluyen:

- Una colección de bibliotecas para implementar estructuras de datos como listas, arrays, árboles y conjuntos.
- Bibliotecas para análisis de XML.
- Seguridad.

Bibliotecas de internacionalización y localización.

Bibliotecas de integración, que permiten la comunicación con sistemas externos. Estas librerías incluyen:

- La API para acceso a bases de datos JDBC (Java DataBase Conectivity).
- La interfaz JNDI (Java Naming and Directory Interface) para servicios de directorio.
- RMI (Remote Method Invocation) y CORBA para el desarrollo de aplicaciones distribuidas.

Bibliotecas para la interfaz de usuario, que incluyen:

- El conjunto de herramientas nativas AWT (Abstract Windowing Toolkit), que ofrece componentes GUI (Graphical User Interface), mecanismos para usarlos y manejar sus eventos asociados.

- Las Bibliotecas de Swing, construidas sobre AWT pero ofrecen implementaciones no nativas de los componentes de AWT.
- APIs para la captura, procesamiento y reproducción de audio.
- Una implementación dependiente de la plataforma en que se ejecuta de la máquina virtual de Java (JVM), que es la encargada de la ejecución del código de las librerías y las aplicaciones externas.
- Plugins o conectores que permiten ejecutar applets en los navegadores Web.
- Java Web Start, para la distribución de aplicaciones Java a través de Internet.
- Documentación y licencia.

APIs [editar]

Sun define tres plataformas en un intento por cubrir distintos entornos de aplicación. Así, ha distribuido muchas de sus APIs (Application Program Interface) de forma que pertenezcan a cada una de las plataformas:

- Java ME (Java Platform, Micro Edition) o J2ME — orientada a entornos de limitados recursos, como teléfonos móviles, PDAs (Personal Digital Assistant), etc.
- Java SE (Java Platform, Standard Edition) o J2SE — para entornos de gama media y estaciones de trabajo. Aquí se sitúa al usuario medio en un PC de escritorio.
- Java EE (Java Platform, Enterprise Edition) o J2EE — orientada a entornos distribuidos empresariales o de Internet.

Las clases en las APIs de Java se organizan en grupos disjuntos llamados paquetes. Cada paquete contiene un conjunto de interfaces, clases y excepciones relacionadas. La información sobre los paquetes que ofrece cada plataforma puede encontrarse en la documentación de ésta.

El conjunto de las APIs es controlado por Sun Microsystems junto con otras entidades o personas a través del programa JCP (Java Community Process). Las compañías o individuos participantes del JCP pueden influir de forma activa en el diseño y desarrollo de las APIs, algo que ha sido motivo de controversia.

En 2004, IBM y BEA apoyaron públicamente la idea de crear una implementación de código abierto (open source) de Java, algo a lo que Sun, a fecha de 2006, se ha negado.

2 USB

En un principio se tenía la interfaz serie y paralelo, pero era necesario unificar todos los conectores creando uno más sencillo y de mayores prestaciones. Así nació el USB (Universal Serial Bus) con una velocidad de 12Mb/seg. y como su evolución, USB 2.0, apodado USB de alta velocidad. Con velocidades en este momento de hasta 480 Mb/seg, es decir, 40 veces más rápido que las conexiones mediante cables USB 1.1.

El Universal Serial Bus sirve para conectar periféricos a una computadora. Fue creado en 1996 por siete empresas: IBM, Intel, Northern Telecom, Compaq, Microsoft, Digital Equipment Corporation y NEC.

USB es una nueva arquitectura de bus o un nuevo tipo de bus que forma parte de los avances plug-and-play y permite instalar periféricos sin tener que abrir el computador para instalarle el hardware, es decir, basta con conectar dicho periférico en la parte posterior del computador y listo.

Una característica importante es que permite a los dispositivos trabajar a velocidades mayores, en promedio a unos 12 Mbps, esto es más o menos de 3 a 5 veces más rápido que un dispositivo de puerto paralelo y de 20 a 40 veces más rápido que un dispositivo de puerto serial.

El estándar incluye la transmisión de energía eléctrica al dispositivo conectado. Algunos dispositivos requieren una potencia mínima, así que se pueden conectar varios sin necesitar fuente de alimentación extra. La gran mayoría de los concentradores incluyen fuentes de alimentación que brindan energía a los dispositivos conectados a ellos, pero algunos dispositivos consumen tanta energía que necesitan su propia fuente de alimentación. Los concentradores con fuente de alimentación pueden proporcionarle corriente eléctrica a otros dispositivos sin quitarle corriente al resto de la conexión (dentro de ciertos límites).

El diseño del USB tenía en mente eliminar la necesidad de adquirir tarjetas separadas para poner en los puertos bus ISA o PCI, y mejorar las capacidades plug-and-play permitiendo a esos dispositivos ser conectados o desconectados al sistema sin necesidad de reiniciar. Cuando se conecta un nuevo dispositivo, el servidor lo enumera y agrega el software necesario para que pueda funcionar.

El USB puede conectar los periféricos como mouse, teclados, escáneres, cámaras digitales, teléfonos celulares, reproductores multimedia, impresoras, discos duros externos, tarjetas de sonido, sistemas de adquisición de datos y componentes de red. Para dispositivos multimedia como escáneres y cámaras digitales, el USB se ha convertido en el método estándar de conexión. Para impresoras, el USB ha crecido tanto en popularidad que ha empezado a desplazar a los

puertos paralelos porque hace sencillo el poder agregar más de una impresora a un computador personal.

Esta arquitectura trabaja como interfaz para transmisión de datos y distribución de energía, que ha sido introducida en el mercado de los computadores y periféricos para mejorar las lentas interfaces serie (RS-232) y paralelo. Esta interfaz de 4 hilos, 12 Mbps y "plug and play", distribuye 5V para alimentación, transmite datos y está siendo adoptada rápidamente por la industria informática.

Es un bus basado en el paso de un testigo, semejante a otros buses como los de las redes locales en anillo con paso de testigo y las redes FDDI. El controlador USB distribuye testigos por el bus. El dispositivo cuya dirección coincide con la que porta el testigo responde aceptando o enviando datos al controlador. Este también gestiona la distribución de energía a los periféricos que lo requieran.

Emplea una topología de estrellas apiladas que permite el funcionamiento simultáneo de 127 dispositivos a la vez. En la raíz o vértice de las capas, está el controlador anfitrión o host que controla todo el tráfico que circula por el bus. Esta topología permite a muchos dispositivos conectarse a un único bus lógico sin que los dispositivos que se encuentran más abajo en la pirámide sufran retardo. A diferencia de otras arquitecturas, USB no es un bus de almacenamiento y envío, de forma que no se produce retardo en el envío de un paquete de datos hacia capas inferiores.

2.1 Componentes Del Sistema De Bus Serie Universal USB

2.1.1 Controlador

Reside dentro del computador y es responsable de las comunicaciones entre los periféricos USB y la CPU del computador. Es también responsable de la admisión de los periféricos dentro del bus, tanto si se detecta una conexión como una desconexión. Para cada periférico añadido, el controlador determina su tipo y le asigna una dirección lógica para utilizarla siempre en las comunicaciones con el mismo. Si se producen errores durante la conexión, el controlador lo comunica a la CPU, que, a su vez, lo transmite al usuario. Una vez se ha producido la conexión correctamente, el controlador asigna al periférico los recursos del sistema que éste precise para su funcionamiento.

El controlador también es responsable del control de flujo de datos entre el periférico y la CPU.

2.1.2 Concentradores o Hubs

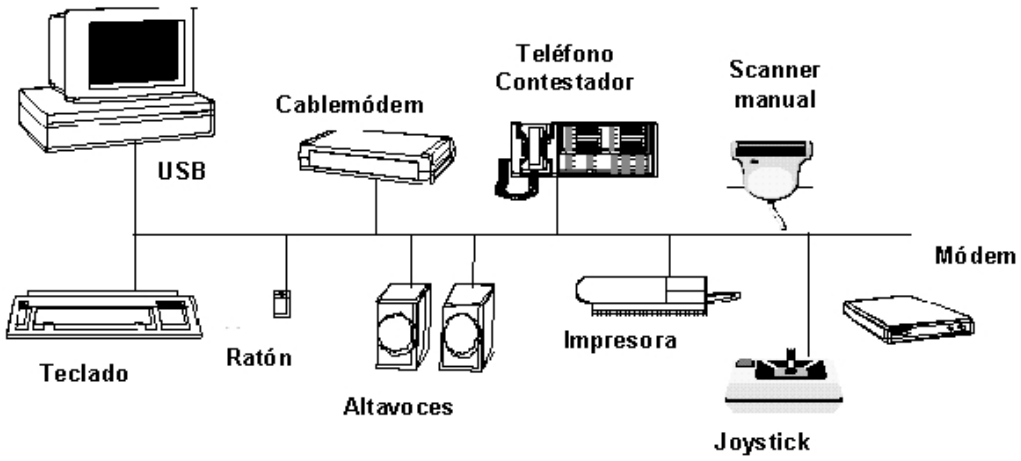
Son distribuidores inteligentes de datos y alimentación, y hacen posible la conexión a un único puerto USB de 127 dispositivos. De una forma selectiva reparten datos y alimentación hacia sus puertas descendentes y permiten la comunicación hacia su puerta de retorno o ascendente, un hub de 4 puertos, por ejemplo, acepta datos del computador para un periférico por su puerta de retorno o ascendente y los distribuye a las 4 puertas descendentes si fuera necesario.

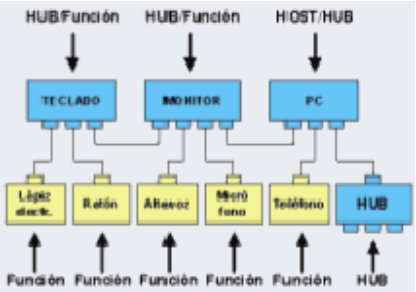
Además del controlador, el computador también contiene el concentrador raíz. Este es el primer concentrador de toda la cadena que permite a los datos y a la energía pasar a uno o dos conectores USB del PC, y de allí a los 127 periféricos que, como máximo, puede soportar el sistema. Esto es posible añadiendo concentradores adicionales. Por ejemplo, si el PC tiene una única puerta USB y a ella le conectamos un hub o concentrador de 4 puertas, el computador se queda sin más puertas disponibles. Sin embargo, el hub de 4 puertas permite realizar 4 conexiones descendentes conectando otro hub de 4 puertas a una de las 4 puertas del primero, habremos creado un total de 7 puertas a partir de una puerta del PC. De esta forma, es decir, añadiendo concentradores, el PC puede soportar hasta 127 periféricos USB.

2.1.3 Periféricos

USB soporta periféricos de baja y media velocidad. Empleando dos velocidades para la transmisión de datos de 1,5 y 12 Mbps se consigue una utilización más eficiente de sus recursos . Los periféricos de baja velocidad tales como teclados, ratones, joysticks, y otros periféricos para juegos, no requieren 12 Mbps. Empleando para ellos 1,5 Mbps, se puede dedicar más recursos del sistema a periféricos tales como monitores, impresoras, módems, scanner, equipos de audio. que precisan de velocidades más altas para transmitir mayor volumen de datos o datos cuya dependencia temporal es más estricta.

A continuación se muestra la interconexión de dispositivos mediante el puerto USB:





2.2 Características de Transmisión

Los dispositivos USB se clasifican en cuatro tipos según su velocidad de transferencia de datos:

Baja Velocidad (1.0): Bitrate de 1.5Mbit/s (192KB/s). Utilizado en su mayor parte por Dispositivos de Interfaz Humana (HID) como los teclados, los ratones y los joysticks.

Velocidad Completa (1.1): Bitrate de 12Mbit/s (1.5MB/s). Esta fue la más rápida antes de que se especificara la USB 2.0 y muchos dispositivos fabricados en la actualidad trabajan a esta velocidad. Estos dispositivos, dividen el ancho de banda de la conexión USB entre ellos basados en un algoritmo FIFO.

Alta Velocidad (2.0): Bitrate de 480Mbit/s (60MB/s).

Súper Velocidad (3.0) Actualmente en fase experimental. Bitrate de 4.8Gbit/s (600MB/s). Esta especificación será lanzada a mediados de 2008 por la compañía Intel, de acuerdo a información recabada de Internet. Las velocidades de los buses serán 10 veces más rápidas que la de USB 2.0 debido a la inclusión de un enlace de fibra óptica que trabaja con los conectores tradicionales de cobre. Se espera que los productos fabricados con esta tecnología lleguen al consumidor en 2009 o 2010.

Las señales del USB son transmitidas en un cable par trenzado con impedancia de 90Ω ±15% llamados D+ y D-. Éstos, colectivamente utilizan señalización diferencial en half dúplex para combatir los efectos del ruido electromagnético en enlaces largos. D+ y D- usualmente operan en conjunto y no son conexiones simplex. Los niveles de transmisión de la señal varían de 0 a 0.3V para bajos (ceros) y de 2.8 a 3.6V para altos (unos) en las versiones 1.0 y 1.1, y en ±400mV en Alta Velocidad (2.0). En las primeras versiones, los alambres de los cables no están conectados a tierra, pero en el modo de alta velocidad se tiene una terminación de 45 Ω a tierra o un diferencial de 90 Ω para acoplar la impedancia del cable.

Pin	Nombre	Color del Cable	Descripción
1	VCC	Rojo	+5V

2	D−	Blanco	Data −
3	D+	Verde	Data +
4	GND	Negro	Tierra

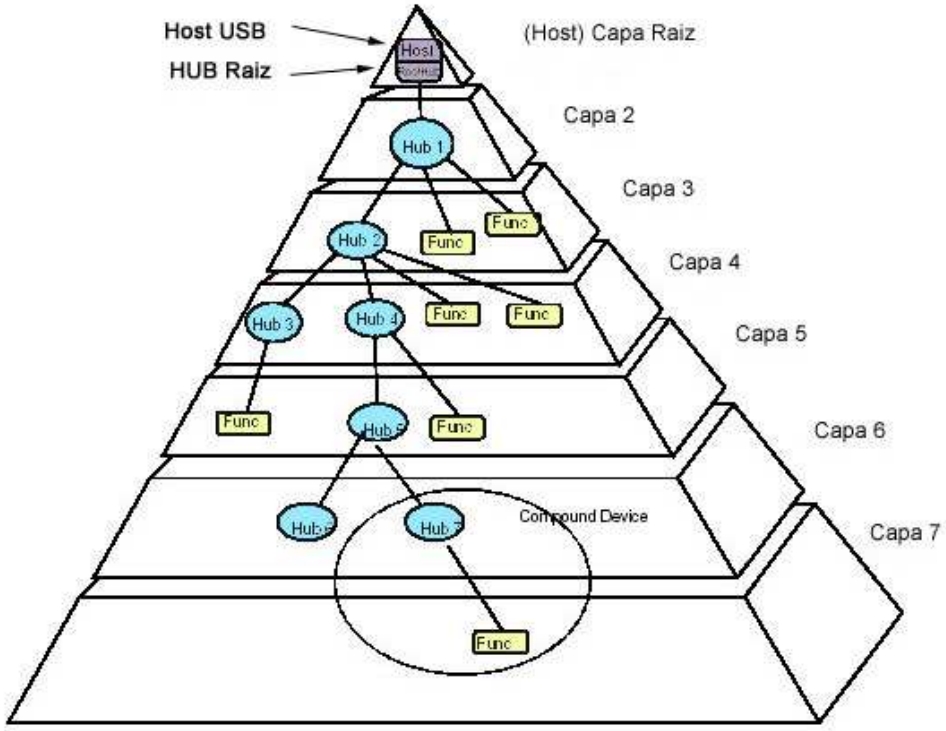
Algunos aspectos básicos en el funcionamiento del USB son:

- La topología
- El flujo de datos
- La capa de protocolo
- La eléctrica
- La mecánica

2.3 La Topología del Bus

El Universal Serial Bus conecta los dispositivos USB con el host USB. La interconexión física USB es una topología de estrellas apiladas donde un hub es el centro de cada estrella. Cada segmento de cable es una conexión punto-a-punto entre el host y los hubs o función, o un hub conectado a otro hub o función.

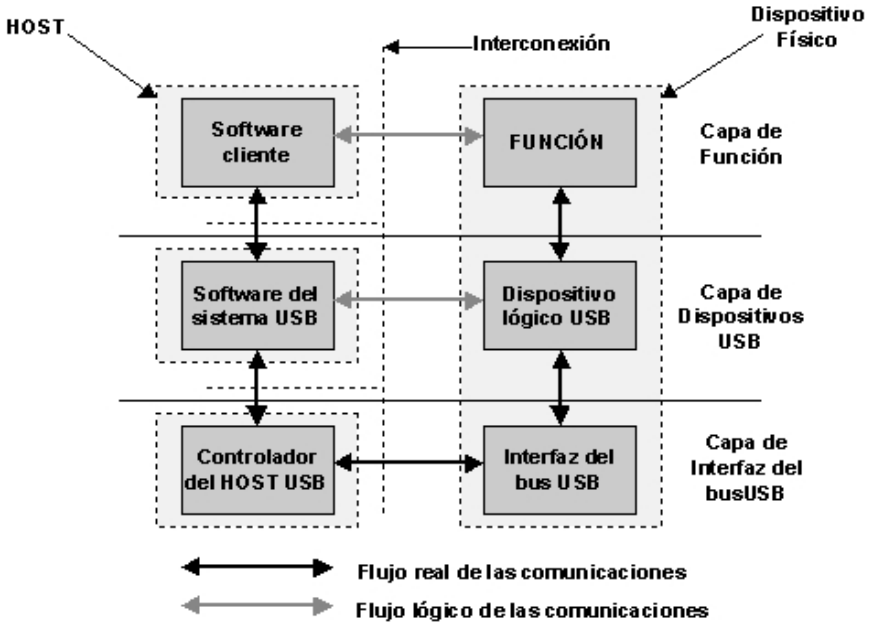
El número máximo de dispositivos que puede conectar USB es de 127, pero debido a las constantes de tiempo permitidas para los tiempos de propagación del hub y el cable, el número máximo de capas permitido es de siete incluida la capa raíz, con un máximo de longitud de cable entre el hub y el dispositivo de 5 metros.



Estructura de Capas del bus USB

La topología del bus USB se puede dividir en tres partes:

- **La capa física:** Como están conectados los elementos físicamente
- **La capa lógica**: Los roles y las responsabilidades de los elementos USB
- **La relación software del cliente- función**: Como se ven mutuamente el software del cliente y los interfaces de las funciones relacionadas



2.3.1 La Capa Física

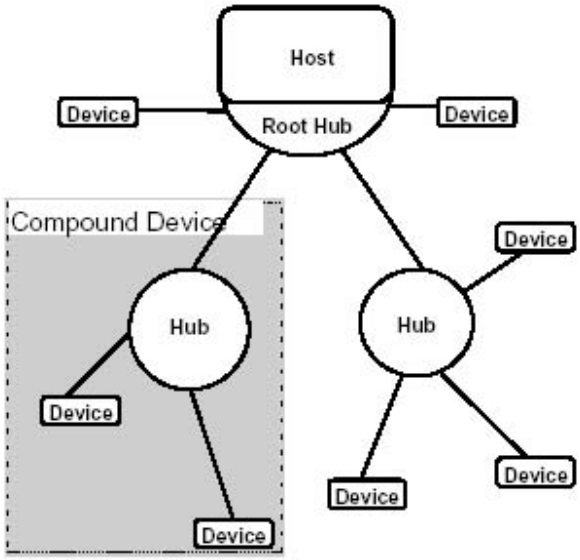
La arquitectura física del USB se centra en las piezas de plástico y de metal con las que el usuario debe tratar para construir un entorno USB. Cada entorno físico USB esta compuesto por cinco tipos de componentes:

- El host
- El controlador del host
- Los enlaces
- Los dispositivos
- Los hubs

Los dispositivos están conectados físicamente al host a través de una topología en estrella, como se muestra en la siguiente figura. Los puntos de acople están provistos de una clase de dispositivos USB llamados hubs, los cuales tienen puntos de acople adicionales llamados puertos. Estos hubs se conectan a otros dispositivos a través de enlaces que son cables de cuatro hilos.

El host proporciona uno o mas puntos de acople a través del hub raíz. Para prevenir los acoples circulares, se impone una estructura ordenada por capas en la topología de estrella y como resultado se obtiene una configuración al estilo de un árbol.

Todas las comunicaciones físicas son iniciadas por el host. Esto quiere decir que cada milisegundo, o en cada ventana de tiempo que el bus lo permita, el host preguntará por nuevos dispositivos en el bus USB. Además el host inicia todas las transacciones físicas y soporta todas las transferencias de datos sobre la capa física



USB Physical Bus Topology

2.3.2 La Capa Lógica

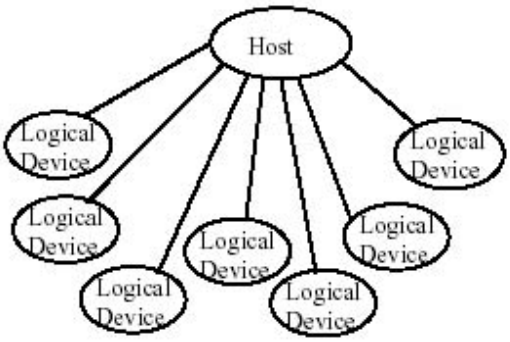
El punto de vista lógico presenta capas y abstracciones que son relevantes para los distintos diseñadores e implementadores. La arquitectura lógica describe como unir el hardware del dispositivo USB a un driver del dispositivo en el host para que tenga el comportamiento que el usuario final desea.

La vista lógica de esta conexión es la mostrada en el esquema siguiente. En el podemos ver como el host proporciona conexión al dispositivo, donde esta conexión es a través de un simple enlace USB. La mayoría de los demás buses como PCI, ISA, etc. proporcionan múltiples conexiones a los dispositivos y los drivers lo manipulan mediante algunas combinaciones de estas conexiones (I/O y direcciones de memoria, interrupciones y canales DMA).

Físicamente el USB tiene sólo un cable simple de bus que es compartido por todos los dispositivos del bus. Sin embargo, desde el punto de vista lógico cada dispositivo tiene su propia conexión punto a punto al host.

Si lo miramos desde el punto de vista lógico, los hubs son también dispositivos pero no se muestran en la figura para la simplificación del esquema.

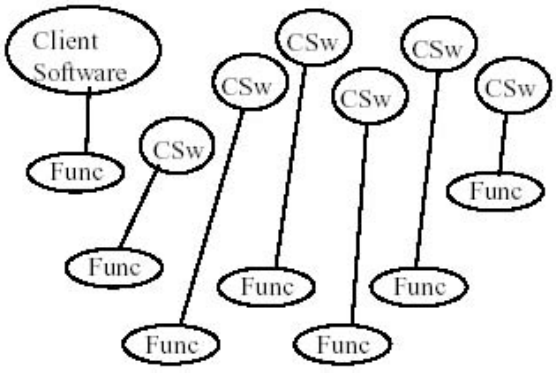
Aunque la mayoría de las actividades de los hosts o de los dispositivos lógicos usan esta perspectiva lógica, el host mantiene el conocimiento de la topología física para dar soporte al proceso de desconexión de los hubs. Cuando se quita un hub, todos los dispositivos conectados a él son quitados también de la vista lógica de la topología.



2.3.3 La relación "software del cliente-función"

A pesar de que la topología física y lógica del USB refleja la naturaleza de compartición del bus, la manipulación del interfaz de una función USB por parte del software del cliente se presenta con una vista distinta.

El software del cliente para las funciones USB debe usar el interfaz de programación software USB para manipular sus funciones en contraposición de las que son manipuladas directamente a través de la memoria o los accesos I/O como pasa con otros buses (PCI,EISA,PCMCIA,...). Durante esta operación, el software del cliente debería ser independiente a otros dispositivos que puedan conectarse al USB.



2.4 El flujo de datos del bus USB

Un dispositivo USB desde un punto de vista lógico hay que entenderlo como una serie de endpoints, a su vez los endpoints se agrupan en conjuntos que dan lugar a interfaces, las cuales permiten controlar la función del dispositivo.

Como ya se ha visto la comunicación entre el host y un dispositivo físico USB se puede dividir en tres niveles o capas. En el nivel mas bajo el controlador de host USB se comunica con la interfaz del bus utilizando el cable USB, mientras que en un nivel superior el software USB del sistema se comunica con el dispositivo lógico utilizando la red de control por defecto. En lo que al nivel de función se refiere, el software cliente establece la comunicación con las interfaces de la función a través de los enlaces asociados a endpoints.

2.4.1 Endpoints y direcciones de dispositivo

Cada dispositivo USB está compuesto por una colección de endpoints independientes, y una dirección única asignada por el sistema en tiempo de conexión de forma dinámica. A su vez cada endpoint dispone de un identificador único dentro del dispositivo al que pertenece, a este identificador se le conoce como número de endpoint y viene asignado de fábrica. Cada endpoint tiene una determinada orientación de flujo de datos. La combinación de dirección, número de endpoint y orientación, permite referenciar cada endpoint de forma inequívoca. Cada endpoint es por si solo una conexión simple que soporta un flujo de datos en una única dirección, bien de entrada o bien de salida.

Cada endpoint se caracteriza por:

- Frecuencia de acceso al bus requerida
- Ancho de banda requerido
- Número de endpoint
- Tratamiento de errores requerido
- Máximo tamaño de paquete que el endpoint puede enviar o recibir
- Tipo de transferencia para el endpoint
- La orientación en la que se transmiten los datos

Existen dos endpoints especiales que todos los dispositivos deben tener, los endpoints con número 0 de entrada y de salida, que deben de implementar un método de control por defecto al que se le asocia la tubería de control por defecto. Estos endpoints están siempre accesibles mientras que el resto no lo estarán hasta que no hayan sido configurados por el host.

2.4.2 Tuberías

Una tubería USB es una asociación entre uno o dos endpoints en un dispositivo, y el software en el host. Las tuberías permiten mover datos entre software en el host, a través de un buffer, y un endpoint en un dispositivo. Hay dos tipos de tuberías:

Stream: Los datos se mueven a través de la tubería sin una estructura definida.

Mensaje: Los datos se mueven a través de la tubería utilizando una estructura USB.

Además una tubería se caracteriza por:

Demanda de acceso al bus y uso del ancho de banda

Un tipo de transferencia.

Las características asociadas a los endpoints

Como ya se ha comentado, la tubería que esta formada por dos endpoints con número cero se denomina tubería de control por defecto. Esta tubería está siempre disponible una vez se ha conectado el dispositivo y ha recibido un reseteo del bus. El resto de tuberías aparecen después que se configure el dispositivo. La tubería de control por defecto es utilizada por el software USB del sistema para obtener la identificación y requisitos de configuración del dispositivo, y para configurar al dispositivo.

El software cliente normalmente realiza peticiones para transmitir datos a una tubería vía IRPs y entonces, o bien espera, o bien es notificado de que se ha completado la petición. El software cliente puede causar que una tubería devuelva todas las IRPs pendientes. El cliente es notificado de que una IRP se ha completado cuando todas las transacciones del bus que tiene asociadas se han completado correctamente, o bien porque se han producido errores.

Una IRP puede necesitar de varias tandas para mover los datos del cliente al bus. La cantidad de datos en cada tanda será el tamaño máximo de un paquete excepto el último paquete de datos que contendrá los datos que faltan. De modo que un paquete recibido por el cliente que no consiga llenar el buffer de datos de la IRP puede interpretarse de diferentes modos en función de las expectativas del cliente, si esperaba recibir una cantidad variable de datos considerará que se trata del último paquete de datos, sirviendo pues como delimitador de final de datos, mientras que si esperaba una cantidad específica de datos lo tomará como un error.

2.4.2.1 Tuberias Streams

No necesita que los datos se transmitan con una cierta estructura. Las tuberías stream son siempre unidireccionales y los datos se transmiten de forma secuencial: "first in, first out". Están pensadas para interactuar con un único cliente, por lo que no se mantiene ninguna política de sincronización entre múltiples clientes en caso de que así sea. Un stream siempre se asocia a un único endpoint en una determinada orientación.

2.4.2.2 Tuberias Mensajes

Las tuberías mensajes hay una interacción de la tubería con el endpoint consta de tres fases. Primero se realiza una petición desde el host al dispositivo, después se transmiten los datos en la dirección apropiada, finalmente un tiempo después se pasa a la fase estado. Para poder llevar a cabo este paradigma es necesario que los datos se transmitan siguiendo una determinada estructura.

Las tuberías de mensajes permiten la comunicación en ambos sentidos, de hecho la tubería de control por defecto es una tubería de mensajes. El software USB del sistema se encarga de que múltiples peticiones no se envíen a la tubería de mensajes concurrentemente. Un dispositivo ha

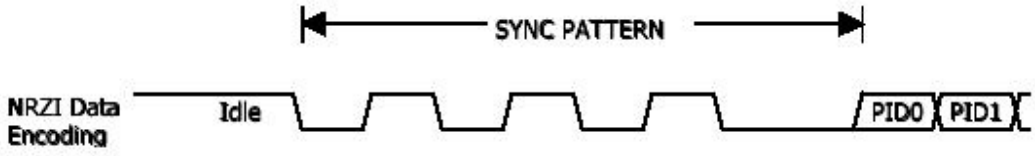
de dar únicamente servicio a una petición de mensaje en cada instante por cada tubería de mensajes.

2.4.3 Frames y microframes

USB establece una unidad de tiempo base equivalente a 1 milisegundo denominada frame y aplicable a buses de velocidad media o baja, en alta velocidad se trabaja con microframes, que equivalen a 125 microsegundos. Los microframes no son más que un mecanismo del bus USB para controlar el acceso a este, en función del tipo de transferencia que se realice. En un microframe se pueden realizar diversas transacciones de datos.

2.4.4 Sync

Teniendo en cuenta que K y J representan respectivamente nivel bajo y nivel alto, el patrón de señal Sync emitido, con los datos codificados, es de 3 pares KJ seguidos de 2 K para el caso de velocidad media y baja. Para velocidad alta es una secuencia de 15 pares KJ seguidos de 2 K. A continuación el caso de velocidad media y baja:



El patrón de señal Sync siempre precede al envío de cualquier paquete, teniendo como objetivo que el emisor y el receptor se sincronicen y se preparen para emitir y recibir datos respectivamente.
Si partimos de que el estado de reposo de la señal es J, podemos interpretar Sync como una secuencia impar de "0's" y un "1" que se inserta antes de los datos.

2.4.4.1 EOP ("End Of Packet")

A todo paquete le sigue EOP, cuya finalidad es indicar el final del paquete. En el caso de velocidad media y baja el EOP consiste en que, después del último bit de datos en el cual la señal estará o bien en estado J, o bien en estado K, se pasa al estado SEO durante el periodo que se corresponde con el ocupado por dos bits, finalmente se transita al estado J que se mantiene durante 1 bit. Esta última transición indica el final del paquete.

En el caso de la velocidad alta se utilizan bits de relleno erróneos, que no están en el lugar correcto, para indicar el EOP. Concretamente, el EOP sin aplicar codificación consisitiría en añadir al final de los datos la secuencia 0111 1111.

2.4.5 Tipos de transferencias

La interpretación de los datos que se transmitan a través de las tuberías, independientemente de que se haga siguiendo o no una estructura USB definida, corre a cargo del dispositivo y del software cliente. No obstante, USB proporciona cuatro tipos de transferencia de datos sobre las tuberías para optimizar la utilización del bus en función del tipo de servicio que ofrece la función.

Estos cuatro tipos son:

- Transferencias de control
- Transferencias isócronas
- Transferencias de interrupción
- Transferencias de bultos

2.5 Capa de protocolo

La forma en la que las secuencias de bits se transmiten en USB es la siguiente; primero se transmite el bit menos significativo, después el siguiente menos significativo y así hasta llegar al bit más significativo. Cuando se transmite una secuencia de bytes se realiza en formato "LITTLE-ENDIAN", es decir del byte menos significativo al byte más significativo.

En la transmisión se envían y reciben paquetes de datos, cada paquete de datos viene precedido por un campo Sync y acaba con el delimitador EOP, todo esto se envía codificado además de los bits de relleno insertados. En este punto cuando se habla de datos se refiere a los paquetes sin el campo Sync ni el delimitador EOP, y sin codificación ni bits de relleno.

El primer campo de todo paquete de datos es el campo PID. El PID indica el tipo de paquete y por lo tanto, el formato del paquete y el tipo de detección de errores aplicado al paquete. En función de su PID podemos agrupar los diferentes tipos de paquetes en cuatro clases:

Tipo PID	Nombre PID	PID	Descripción
Token	OUT	0001B	Dirección + número de endpoint en una transacción host a función.
	IN	1001B	Dirección + número de endpoint en una transacción función a host.
	SOF	0101B	Indicador de inicio de frame (Start Of Frame) y número de frame.
	SETUP	1101B	Dirección + número de endpoint en una transacción host a función para realizar un Setup de una tubería de control.
Data	DATA0	0011B	PID de paquete de datos par.
	DATA1	1011B	PID de paquete de datos impar.
	DATA2	0111B	PID de paquete de datos de alta velocidad, elevado ancho de banda en una transferencia isócrona en un microframe.
	MDATA	1111B	PID de paquete de datos de alta velocidad para split y elevado ancho de

Tipo PID	Nombre PID	PID	Descripción
			banda en una transferencia isócrona.
Handshake	ACK	0010B	El receptor acepta el paquete de datos libre de errores.
	NAK	1010B	El dispositivo receptor no puede aceptar los datos o el dispositivo emisor no puede enviar los datos.
	STALL	1110B	Endpoint sin servicio o una petición de control sobre una tubería no está soportado.
	NYET	0110B	Aún no se ha recibido una respuesta del receptor.
Special	PRE	1100B	(Token) Habilita trafico de bajada por el bus a dispositivos de velocidad baja.
	ERR	1100B	(Handshake) Error de transferencia Split.
	SPLIT	1000B	(Token) Transferencia de alta velocidad Split.
	PING	0100B	(Token) Control de flujo sobre endpoints de tipo control o bulk.
	Reservado	0000B	PID reservado.

A continuación el formato de los campos y los paquetes, además de una vista general de como funciona el protocolo:

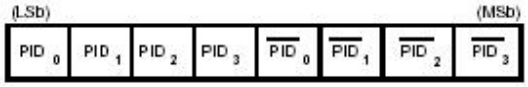
- El formato de los campos
- El formato de los paquetes
- Transacciones
- Split
- El protocolo y las transferencias

2.5.1 Formato de los campos

Los paquetes se dividen en campos, a continuación el formato de los diferentes campos.

2.5.1.1 Campo identificador de paquete (PID)

Es el primer campo que aparece en todo paquete. El PID indica el tipo de paquete, y por tanto el formato del paquete y el tipo de detección de error aplicado a este. Se utilizan cuatro bits para la codificación del PID, sin embargo el campo PID son ocho bits, que son los cuatro del PID seguidos del complemento a 1 de esos cuatro bits. Estos últimos cuatro bits sirven de confirmación del PID. Si se recibe un paquete en el que los cuatro últimos bits no son el complemento a 1 del PID, o el PID es desconocido, se considera que el paquete está corrupto y es ignorado por el receptor.



2.5.1.2 Campo dirección

Este campo indica la función, a través de la dirección, que envía o es receptora del paquete de datos. Se utilizan siete bits, de lo cual se deduce que hay un máximo de 128 direcciones.

2.5.1.3 Campo endpoint

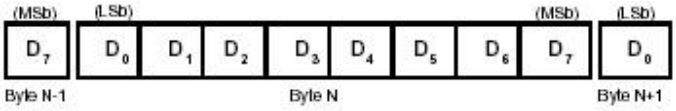
Se compone de cuatro bits e indica el número de "enpoint" al que se quiere acceder dentro de una función, como es lógico este campo siempre sigue al campo dirección.

2.5.1.4 Campo número de frame

Es un campo de 11 bits que es incrementado por el host cada (micro)frame en una unidad. El máximo valor que puede alcanzar es el 7FFH, si se vuelve a incrementar pasa a cero.

2.5.1.5 Campo de datos

Los campos de datos pueden variar de 0 a 1024 bytes. En el dibujo se muestra el formato para múltiples bytes.



2.5.1.6 Cyclic Redundancy Checks (CRC)

El CRC se usa para proteger todos los campos no PID de los paquetes de tipo token y de datos. El CRC siempre es el último campo y se genera a partir del resto de campos del paquete, a excepción del campo PID. El receptor al recibir el paquete comprueba si se ha generado de acuerdo a los

campos del paquete, si no es así, se considera que alguno o mas de un campo están corruptos, en ese caso se ignora el paquete.

El CRC utilizado detecta todos lo errores de un bit o de dos bits. El campo de CRC es de cinco bits para los paquetes de tipo IN, SETUP, OUT, PING y SPLIT. El polinomio generador es el siguiente:

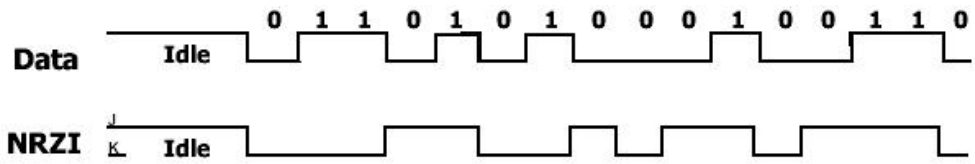
$$G(X) = X^5 + X^2 + 1$$

En el caso de los paquetes de datos se utiliza un campo CRC de 16 bits, el polinomio generador es el siguiente:

$$G(X) = X^{16} + X^{15} + X^2 + 1$$

2.5.2 Codificación de datos

El USB utiliza la codificación NRZI para la transmisión de paquetes. En esta codificación los "0" se representan con un cambio en el nivel, y por el contrario los "1" se representan con un no cambio en el nivel. De modo que las cadenas de cero producen transiciones consecutivas en la señal, mientras que cadenas de unos produce largos periodos sin cambios en la señal. A continuación un ejemplo:



2.5.3 Relleno de bits

Debido a que cadenas de unos pueden producir largos periodos en los que la señal no cambia dando lugar a problemas de sincronización, se introducen los bits de relleno. Cada 6 bits consecutivos a "1" se inserta un bit a "0" para forzar un cambio, de esta forma el receptor puede volverse a sincronizar. El relleno bits empieza con el patrón de señal Sync. El "1" que finaliza el patrón de señal Sync es el primer uno en la posible primera secuencia de seis unos.

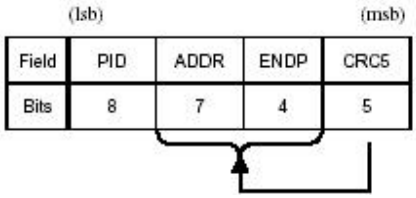
En las señales a velocidad media o baja, el relleno de bits se utiliza a lo largo de todo el paquete sin excepción. De modo que un paquete con siete unos consecutivos será considerado un error y por lo tanto ignorado.

En el caso de la velocidad alta se aplica el relleno de bits a lo largo del paquete, con la excepción de los bits intencionados de error usados en EOP a velocidad alta.

2.5.4 Formato de los paquetes

2.5.4.1 Paquetes de tipo token

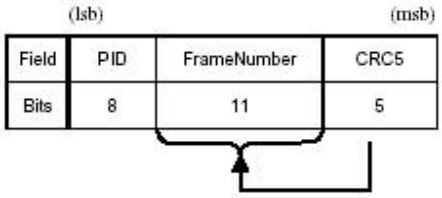
Un token está compuesto por un PID que indica si es de tipo IN, OUT o SETUP. El paquete especial de tipo PING también tiene la misma estructura que token. Después del campo PID viene seguido de un campo dirección y un campo endpoint, por último hay un campo CRC de 5 bits.



En los paquetes OUT y SETUP esos campos identifican al endpoint que va a recibir el paquete datos que va a continuación. En los paquetes IN indican el endpoint que debe transmitir un paquete de datos. En el caso de los paquetes PING hacen referencia al endpoint que debe responder con un paquete "handshake".

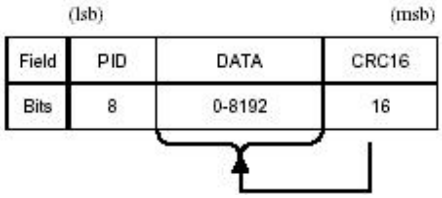
2.5.4.2 Paquete inicio de frame (SOF)

Estos paquetes son generados por el host cada un milisegundo en buses de velocidad media y cada 125 microsegundos para velocidad alta. Este paquete está compuesto por un campo número de frame y un campo de CRC de 5 bits.



2.5.4.3 Paquete de datos

Este paquete está compuesto por cero o más bytes de datos seguido de un campo de CRC de 16 bits. Existen cuatro tipos de paquetes de datos: DATA0, DATA1, DATA2 y MDATA. El número máximo de bytes de datos en velocidad baja es de ocho bytes, en media de 1023 bytes y en alta de 1024 bytes. El número de bytes de datos ha de ser entero.



2.5.4.4 Paquetes "Handshake"

Los paquetes "handshake", en castellano apretón de manos, se utilizan para saber el estado de una transferencia de datos, indicar la correcta recepción de datos, aceptar o rechazar comandos, control de flujo, y condiciones de parada. El único campo que contiene un paquete de este tipo es el campo PID.

(lsb) (msb)

Field	PID
Bits	8

Existen cuatro paquetes de tipo "handshake" además de uno especial:

ACK: Indica que el paquete de datos ha sido recibido y decodificado correctamente. ACK sólo es devuelto por el host en las transferencias IN y por una función en las transferencias OUT, SETUP o PING.

NAK: Indica que una función no puede aceptar datos del host (OUT) o que no puede transmitir datos al host (IN). También puede enviarlo una función durante algunas fases de transferencias IN, OUT o PING. Por último se puede utilizar en el control de flujo indicando disponibilidad. EL host nunca puede enviar este paquete.

STALL: Puede ser devuelto por una función en transacciones que intervienen paquetes de tipo IN, OUT o PING. Indica que una función es incapaz de transmitir o enviar datos, o que una petición a una tubería control no está soportada. El host no puede enviar bajo ninguna condición paquetes STALL.

NYET: Sólo disponible en alta velocidad es devuelto como respuesta bajo dos circunstancias. Como parte del protocolo PING, o como respuesta de un hub a una transacción Split indicando que la transacción de velocidad media o baja aún no ha terminado, o que el hub no está aún habilitado para realizar la transacción.

ERR: De nuevo sólo disponible en alta velocidad y de nuevo formando parte del protocolo PING, permite a un hub de alta velocidad indicar que se ha producido un error en un bus de media o baja velocidad.

2.5.5 Transacciones

Los paquetes de tipo token tiene como objetivo indicar el incio de una transacción de datos de una determina forma.

2.5.5.1 Transacción IN

La transacción empieza con el envío de un paquete de tipo IN por parte del host a un determinado endpoint en una función. Un endpoint cuando recibe un paquete de tipo IN debe comportarse como muestra la siguiente tabla.

Token recibido corrupto	Estado de endpoint	La función puede transmitir los datos	Acción
Si	--	--	No responde
No	Deshabilitado	--	Envía STALL
No	Habilitado	No	Envía NAK
No	Habilitado	Si	Envía el paquete de datos

La respuesta que del host al recibir un paquete de datos en la transacción se puede observar en la siguiente tabla:

Paquete de datos corrupto	El host puede aceptar los datos	Respuesta del host
Si	--	Descarta el paquete, no responde
No	No	Descarta el paquete, no responde
No	Si	Envía ACK

2.5.5.1.1 Sincronización mediante conmutación de bits

USB ofrece un mecanismo que garantiza la sincronización entre el emisor y el receptor de datos a lo largo de múltiples transacciones. La idea es garantizar que los dos sean conscientes de que los handshakes han sido recibidos correctamente. Para ello se utilizan los dos PID DATA0 y DATA1 que sólo varían en un bit. De manera que inicialmente tanto el emisor y el receptor están sincronizados en la secuencia de bits. De modo que, el que recibe datos conmuta su bit cuando puede aceptar datos y ha recibido un paquete de datos libre de errores. Por su parte el que envía conmuta su bit cuando recibe un ACK, tal que si el último paquete de datos tenía PID DATA0 el siguiente tendrá DATA1, y viceversa, además si todo ha salido bien, el PID del paquete coincidirá con la secuencia de bits del receptor.

2.5.5.2 Transacción OUT

El host envía un paquete OUT a un determinado endpoint y seguidamente envía el paquete de datos. Suponiendo que el endpoint ha decodificado correctamente el token, en caso contrario se ignora el token y lo datos, espera a recibir un paquete de datos. El comportamiento del endpoint cuando recibe el paquete de datos es el siguiente.

Paquete de datos corrupto	Estado del receptor	Coincidencia de la secuencia de bits	La función puede aceptar los datos	Acción
Si	--	--	--	No responde
No	Deshabilitado	--	--	Envía STALL
No	Habilitado	No	--	Envía ACK
No	Habilitado	Si	Si	Envía ACK
No	Habilitado	Si	No	Envía NAK

2.5.5.3 Transacción SETUP

La transacción SETUP es una transacción especial que tiene las mismas fases que una transacción OUT, que sólo se puede utilizar con endpoints de tipo control y cuya finalidad es la de indicar el inicio de la fase setup.

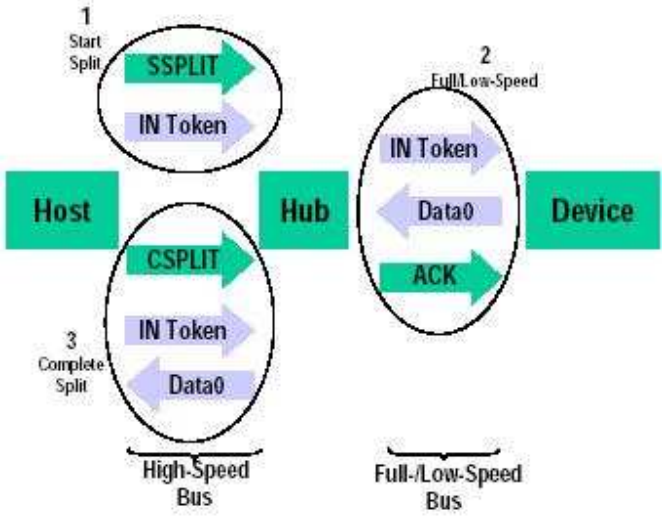
2.5.6 Split

El token especial SPLIT es utilizado para poder soportar transacciones en varias partes entre un hub que trabaja a velocidad alta con dispositivos de velocidad media o baja. Se definen nuevas transacciones que utilizan el token SPLIT, en concreto dos transacciones con sus respectivos paquetes: "start-split transaction (SSPLIT)" y "complete-split transaction (CSPLIT)".

2.5.6.1 Transacciones Split

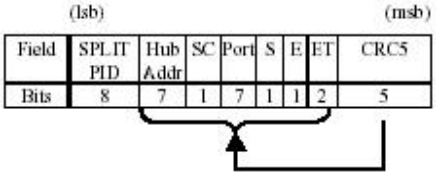
La transacción split es utilizada solamente por el host y un hub cuando se quiere comunicar con un dispositivo de velocidad media o baja conectado con el hub. El host puede iniciar una transacción por partes a través del paquete SSPLIT y puede completarla, recibiendo las respuestas de la función de velocidad baja o media, a través del paquete CSPLIT. Esto permite al host realizar otras transacciones a velocidad alta sin tener que esperar a que acabe la transacción.

A continuación un ejemplo de una transacción por partes para establecer una transacción de tipo IN.



2.5.6.2 *Formato del paquete SPLIT*

Campos del paquete SSPLIT y CSPLIT:



El campo "Hub Addr" es un campo dirección que indica la dirección del hub. El bit SC indica el tipo de paquete, si SC=0 se trata de un SSPLIT, y si SC=1 de un CSPLIT. El campo "puerto del hub destino" hace referencia al puerto del hub al que está destinado esta transacción por partes, el formato de este campo es idéntico al de un campo dirección. El significado de los bits S y E dependen del tipo de transferencia de datos que se esté utilizando, y puede hacer referencia a como se relaciona los tamaños de los paquetes de datos de alta y media velocidad, o hacer referencia a la velocidad de la transacción. El bit E en los paquetes CSPLIT nunca se utiliza, de hecho se llama bit U. Por último el campo de dos bits ET indica el tipo de transferencia de datos, es decir, el tipo de endpoint. En la siguiente tabla se puede observar el funcionamiento de ET.

ET	Tipo de endpoint
00	Control
01	Isócrono
10	"Bulk"

ET	Tipo de endpoint
11	Interrupción

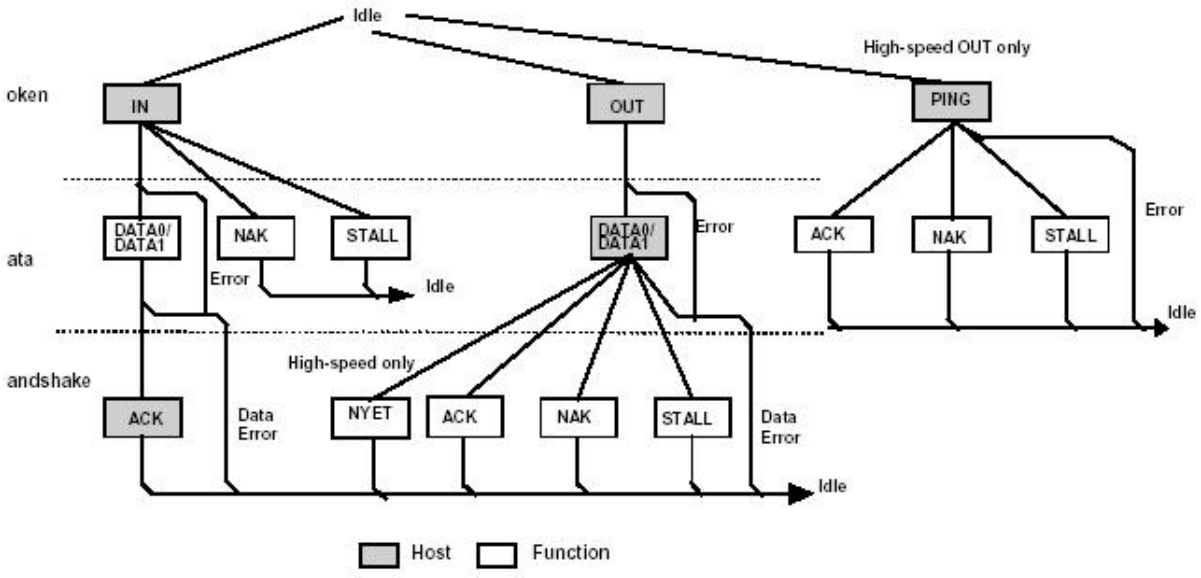
2.5.7 El protocolo y las transferencias

En este apartado se muestra de forma general las transacciones que se realizan, y las particularidades de cada tipo de transferencia. De forma general toda función en principio está esperando a que el host le envíe un paquete, si este paquete es de tipo token entonces cambia el estado de la función iniciándose una transacción. También se debe de tener en cuenta, que cuando o bien el host, o bien una función se encuentran en un estado que no es el de reposo, aparecen los timeouts como otra causa de error.

2.5.7.1 Transferencias de bulto ("Bulk")

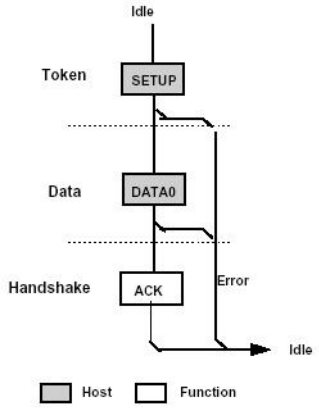
En las transferencias de bulto se puede hablar en parte de transacciones de bulto, pues la idea es enviar datos de paquete en paquete sin que tenga que haber un flujo de datos continuo. La diferencia reside en que si hablamos de transferencia, no se puede considerar que haya terminado hasta que el host haya conseguido enviar los datos, o bien que después de varios intentos fallidos de un error.

Cada transacción de bulto se puede dividir en tres fases: token, datos y *"handshake"*, si bien gracias al token PING pueden haber transacciones de dos fases: token y *"handshake"*. A continuación un esquema de una transacción de bultos.



2.5.7.2 Transferencias de control

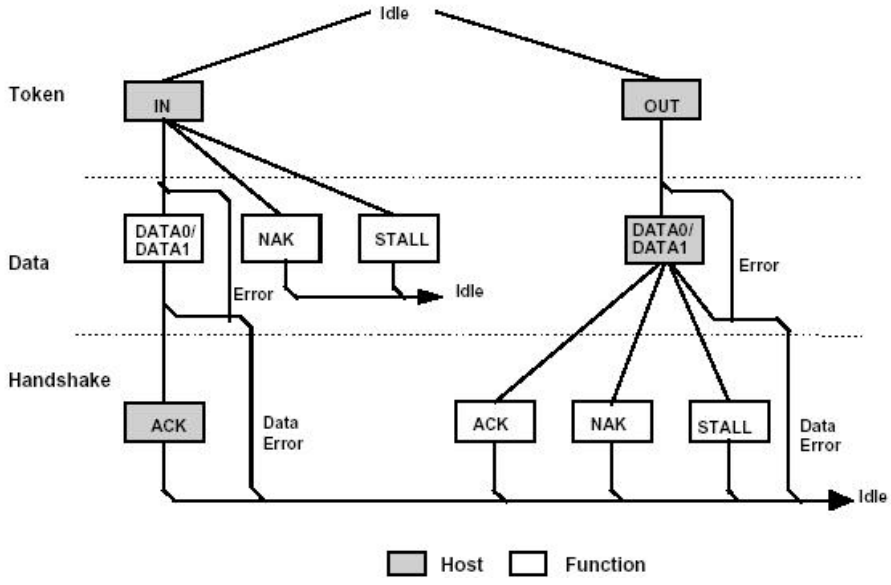
Estas transferencias constan de tres fases: transacción setup, fase de datos y transacción de estado. La transacción siempre la inicia el host, y sirve para enviar información de control para indicar al endpoint que se quiere realizar. El siguiente esquema representa una transacción setup.



A continuación se inicia la fase de transferencia de datos, que consiste en una secuencia de transacciones de datos siempre en la misma dirección alternando DATA0 y DATA1. Esta fase no es obligatoria, la dirección se establece en la fase setup. Finalmente tiene lugar una transacción estado, esta transacción es idéntica a una transacción de bultos. La dirección de esta transacción es siempre la contraria a la de la fase de transferencia de datos, y si esta no exisitiese, la dirección es del endpoint al host.

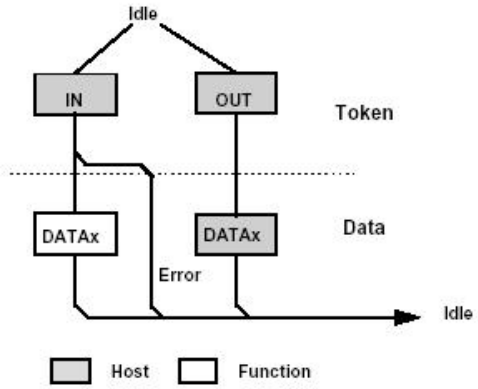
2.5.7.3 Transferencias de interrupción

Las transferencias de interrupción son solamente transacciones de tipo IN y OUT. Desde el punto de vista de las transacciones es muy similar a una transferencia de bultos. A continuación el esquema de una transacción de interrupción.



2.5.7.4 *Transferencias isócronas*

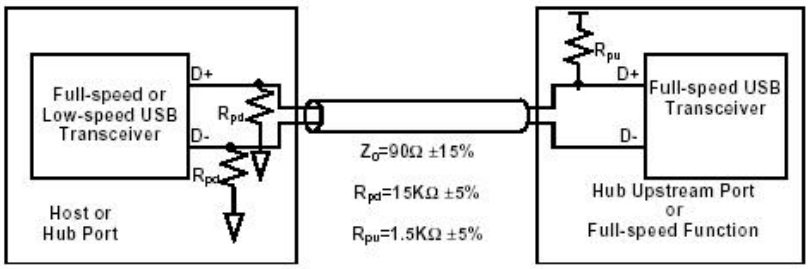
Una transferencia isócrona se plantea como una secuencia de transacciones muy sencillas para enviar o recibir datos. Estas transacciones no utilizan *"handshakes"* y por lo tanto no se reenvían paquetes, ya que el objetivo de la transferencia es simular un flujo constante de datos. A continuación un esquema de una transacción.

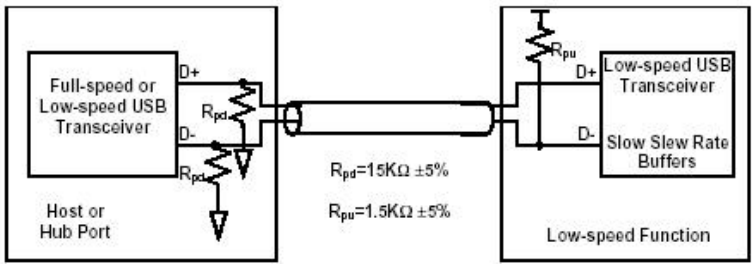


2.6 La eléctrica

2.6.1 Identificación de la velocidad del dispositivo

Para poder iniciar cualquier tipo de transacción cuando se conecta el dispositivo al host, es necesario que este conozca la velocidad a la que trabaja. Con esa finalidad existe un mecanismo a nivel eléctrico. La diferencia entre los dispositivos de velocidad media y los de velocidad baja, es que en velocidad media tiene una resistencia conectada al D+, en velocidad baja la misma resistencia se encuentra en D- y no en D+ como se puede observar en la siguiente figura.

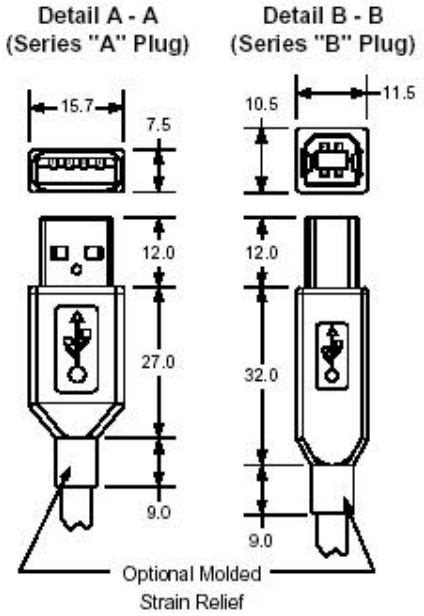




De forma que después del reset el estado de reposo de la línea es diferente si se trata de baja o media velocidad. En el caso de dispositivos de alta velocidad lo que se hace es que en un principio se conecta como un dispositivos de velocidad media y más tarde a través de un protocolo se pasa a velocidad alta.

2.7 La mecánica

Como ya se ha visto la topología física USB consiste en la conexión del puerto de bajada de un hub o host, con el puerto de subida de algún otro dispositivo o hub. Para facilitar la conexión de dispositivos de cara el usuario, USB utiliza dos tipos de conectores totalmente diferentes, los conectores de Serie A y los conectores de serie B. Los conectores de serie A permiten la conexión directa de dispositivos USB con el host o con el puerto de bajada de un host, y es obligatorio que estén presentes en todos los dispositivos y hubs USB. Los conectores de Serie B no son obligatorios y sirven para conectar un cable USB con el puerto de subida de un dispositivo, permitiendo por parte de los fabricantes de dispositivos la utilización de cables estándar USB.



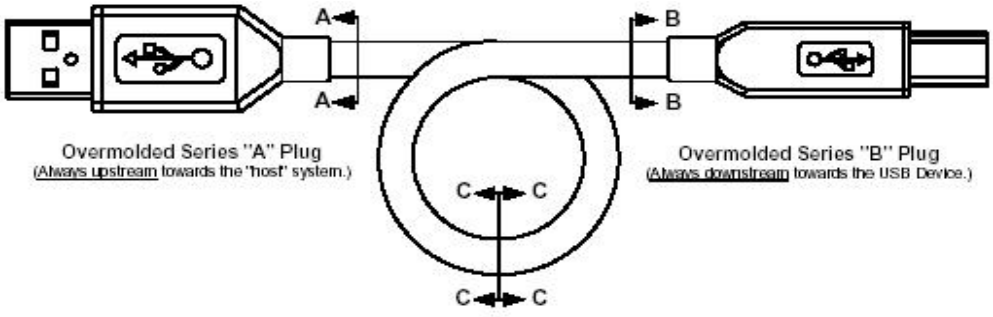
El cable USB consiste de cuatro conductores, dos conductores de potencia y dos de señal, D+ y D-. Los cables de media y alta velocidad están compuestos por un par trenzado de señal, además de GND (Tierra) y Vbus.

Existen tres tipos de cables USB: cable estándar de quita y pon, cable fijo de media y alta velocidad, y cable fijo de baja velocidad.

2.7.1 Cable estándar de quita y pon

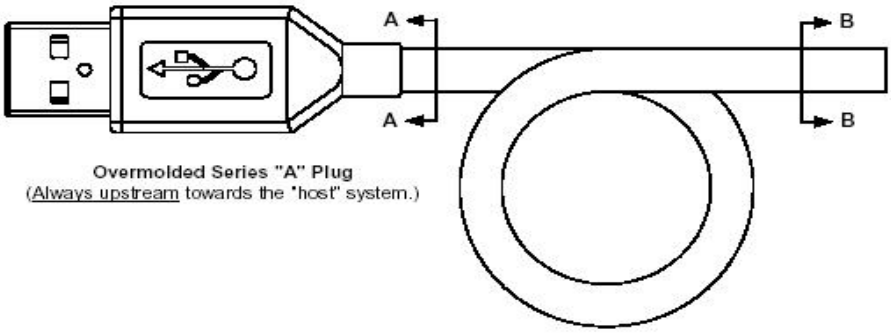
Se trata de un cable de velocidad alta y media, con un conector macho de Serie A en un extremo y un conector macho de Serie B en el otro extremo. Esto permite a los fabricantes de dispositivos fabricarlos sin cable y al usuario le facilita la sustitución del cable en caso de que se estropee. Es un requisito que los dispositivos que utilicen este cable sean de velocidad alta o media, el correcto

funcionamiento del cable con dispositivos de velocidad baja no está garantizado porque podría superar la longitud máxima del cable de velocidad baja.



2.7.2 Cable fijo de velocidad alta y media

Con la denominación de fijo nos referimos a los cables que son proporcionados por el fabricante del dispositivo fijos a este, o bien sin ser fijos, con un conector especifico del fabricante. Es obligatorio que en un extremo tenga un conector macho de Serie A. Dado que lo suministra el fabricante, puede ser utilizado por dispositivos tanto de velocidad alta y media, como de velocidad baja. En el caso de que se utilice para un dispositivo de velocidad baja, además de poder ser utilizado con dispositivos de velocidad media y alta, deberá cumplir con todos los requisitos propios de la velocidad baja.



2.7.3 Cable fijo de velocidad baja

Al igual que el cable fijo de alta y media velocidad tiene un conector macho de Serie A en un extremo, mientras que el otro depende del fabricante. La diferencia es que este tipo de cables sólo funciona con dispositivos de velocidad baja.

Contacto	Señal	Cable
1	VBUS	Rojo
2	Datos-	Blanco

3	Datos+	Verde
4	GND	Negro

2.8 Estándar IEEE 1394 o FireWire.

Apple y Sony inventaron el FireWire a mediados de los 90 y lo desarrollaron hasta convertirlo en el estándar multiplataforma IEEE 1394. FireWire es una tecnología para la entrada/salida de datos en serie a alta velocidad y la conexión de dispositivos digitales como videocámaras o cámaras fotográficas digitales que ha sido ampliamente adoptado por fabricantes de periféricos digitales como Sony, Canon, JVC y Kodak.

FireWire es uno de los estándares de periféricos más rápidos que se han desarrollado, característica que lo hace ideal para su uso con periféricos del sector multimedia (como cámaras de vídeo) y otros dispositivos de alta velocidad como, por ejemplo, lo último en unidades de disco duro e impresoras. Se ha convertido en la interfaz preferida de los sectores de audio y vídeo digital, ya que reúne numerosas ventajas, entre las que se encuentran la elevada velocidad, la flexibilidad de la conexión y la capacidad de conectar un máximo de 63 dispositivos. Además de cámaras y equipo de vídeo digital, la amplia gama de productos FireWire comprende reproductores de vídeo digital, sistemas domésticos para el ocio, sintetizadores de música, escáneres y unidades de disco duro.

Con un ancho de banda 30 veces mayor que el conocido estándar de periféricos USB 1.1, el FireWire 400 se ha convertido en el estándar más respetado para la transferencia de datos a alta velocidad. Apple fue el primer fabricante de ordenadores que incluyó FireWire en toda su gama de productos. Una vez más, Apple ha vuelto a subir las apuestas duplicando la velocidad de transferencia con su implementación del estándar IEEE 1394b o FireWire 800.

La velocidad sobresaliente del FireWire 800 frente al USB 2.0 convierte al primero en un medio mucho más adecuado para aplicaciones que necesitan mucho ancho de banda, como las de gráficos y vídeo, que a menudo consumen cientos o incluso miles de megabytes de datos por archivo.

Algunas de las características más importantes del FireWire son:

Flexibles opciones de conexión. Admite un máximo de 63 dispositivos con cables de hasta 4,25 metros. Distribución en el momento. Fundamental para aplicaciones de audio y vídeo, donde un fotograma que se retrasa o pierde la sincronización arruina un trabajo. Alimentación por el bus. Mientras el USB 2.0 permite la alimentación de dispositivos sencillos que consumen un máximo de 2,5 W, como un ratón, los dispositivos FireWire pueden proporcionar o consumir hasta 45 W, más que suficiente para discos duros de alto rendimiento y baterías de carga rápida.

Es conectable/desconectable en uso. Lo que significa que no se necesita desactivar un dispositivo para conectarlo o desconectarlo y que no es necesario reiniciar el ordenador. Funciona tanto con Mac como con PC. Lo que garantiza la compatibilidad con una larga lista de productos con FireWire a precios razonables. [App]

3 COMUNICACIÓN USB A TRAVÉS DE JAVA

3.1 JSR-80 API

El proyecto JSR-80 fue creado por Dan Streetman a IBM en 1999. En 2001, el proyecto fue aceptado como candidato para llegar a ser un estándar extendido del lenguaje Java a través de la Solicitud de Especificación Java (JSR). El proyecto que ahora se llama JSR-80 y ha sido asignado oficialmente el paquete Java javax.usb. El proyecto está bajo la licencia Common Public License y es desarrollado utilizando la Java Community Process. El objetivo de este proyecto es desarrollar una interfaz USB para la plataforma Java que permitirá el acceso pleno al sistema USB para cualquier aplicación Java o componentes middleware. El JSR-80 API ofrece pleno apoyo a la transferencia de los cuatro tipos que se definen por la especificación USB. Actualmente, la implementación de la API para Linux trabaja en las más recientes distribuciones de GNU / Linux con soporte kernel 2.4, tales como Red Hat 7.2 y 9.0.

El proyecto JSR-80 incluye tres paquetes: javax-usb (API javax.usb), javax-usb-ri (la parte común del sistema operativo independiente de la implementación de referencia), y javax-usb-ri-linux (la implementación de referencia para la plataforma Linux, que conecta la implementació de referencia común a la pila USB de Linux USB). Las tres partes son necesarias para lograr un completo funcionamiento de API java.usb en la plataforma Linux.

Aunque la dependencia del sistema operativo de la implementación de la JSR-80 API varía de un sistema operativo a otro, el programador de Java debe entender sólo el paquete javax.usb para iniciar el desarrollo de aplicaciones. En el siguiente recuadro se enumeran las interfaces y clases en javax.usb con la que un programador Java debe estar familiarizado:

Interfaz	**Descripción**
UsbConfiguration	Representa la configuración de un dispositivo USB
UsbConfigurationDescriptor	Interfaz para un descriptor de configuración USB
UsbDevice	Interfaz para un dispositivo USB
UsbDeviceDescriptor	Interfaz para un descriptor de dispositivo USB
UsbEndpoint	Interfaz para un End Point USB
UsbEndpointDescriptor	Interfaz para un descriptor de End Point USB
UsbHub	Interfaz para un Hub USB

UsbInterface	Interfaz para una Interfaz USB
UsbInterfaceDescriptor	Interfaz para un descriptor de interfaz USB
UsbPipe	Interfaz para una Pipe USB
UsbPort	Interfaz para un Puerto USB
UsbServices	Interfaz para una implementación javax.usb
Clases	**Descripción**
UsbHostManager	Punto de entrada para javax.usb

El procedimiento normal para acceder a un dispositivo USB con el JSR-80 API es el siguiente:

1. Arrancar obteniendo el apropiado UsbServices de la UsbHostManager.

2. Acceda a la raíz a través del centro UsbServices. El concentrador raíz está considerado como una UsbHub en la solicitud.

3. Obtenga una lista de los UsbDevices que están conectados a la raíz central. Recorra a través de todos los centros de nivel inferior para encontrar el apropiado UsbDevice.

4. Interactuar con el UsbDevice directamente con un mensaje de control (UsbControlIrp), o reclamar una UsbInterface de la apropiada UsbConfiguration del UsbDevice y realizar I / O con el UsbEndpoint disponible en la UsbInterface.

5. Si un UsbEndpoint se utiliza para realizar I / O, abra la UsbPipe asociada a él. Puede presentarse ya sea sincrónica o asincrónicamente tanto las fases de upstream data (desde el dispositivo USB al computador central) y downstream data (desde el ordenador host para el dispositivo USB) a través de la UsbPipe.

6. Cerrar la UsbPipe y libere la apropiada UsbInterface cuando la aplicación ya no necesita tener acceso al UsbDevice.

3.2 API jUSB

El proyecto JUSB fue creado por Mojo Jojo y David Brownell en junio de 2000. Su objetivo era proporcionar un conjunto de software libre de Java API para acceder a los dispositivos USB en plataformas Linux. La API se distribuye para que pueda ser usado con propiedad como proyectos de software libre. La API proporciona múltiple acceso a múltiples dispositivos físicos USB, y soporta tanto dispositivos propios como dispositivos remotos. Los dispositivos con múltiples interfaces pueden acceder a múltiples aplicaciones (o controladores de dispositivos) al mismo tiempo, con cada aplicación (o controlador de dispositivo) seleccionar una interfaz diferente. La API soporta el control de las transferencias, las grandes transferencias, e interrumpir transferencias; las transferencias síncronas no son compatibles debido a que estas se utilizan para los medios de comunicación de datos (tales como audio y video) que ya están bien apoyados por la API JMF. Actualmente, la API trabaja sobre distribuciones GNU / Linux, ya sea con el kernel Linux 2,4 o en kernel 2.2.18. Por lo tanto, las distribuciones más recientes cuentan con el apoyo, por ejemplo, la API que trabaja sobre Red Hat 7,2 y 9,0 sin ningún tipo de parches o actualizaciones.

La API jUSB incluye los siguientes paquetes:

•usb.core: Este paquete es la parte central de la API jUSB. Permite que las aplicaciones Java puedan acceder a los dispositivos USB remotos desde los hots USB.

•usb.linux: Este paquete contiene una aplicación de Linux usb.core.Host de un objeto, apoyo autosuficiente, y otras clases de apalancamiento para el soporte de USB sobre linux. Esta aplicación accede a los dispositivos USB a través del dispositivo USB virtual del sistema de archivos (usbdevfs).

•usb.windows: Este paquete tiene una aplicación de Windows de un objeto usb.core.Host, apoyo autosuficiente, y otras clases de Windows aprovechando el soporte para USB. Esta aplicación se encuentra todavía en una etapa muy temprana.

•usb.remote: Este paquete es una versión remota de la API usb.core. Incluye un proxy RMI y una aplicación demonio (daemon), que permiten mediante una aplicación Java acceder a los dispositivos USB en un computador remoto.

•usb.util: Este paquete proporciona algunos servicios públicos útiles para descargar el firmware para dispositivos USB, volcar el contenido del sistema USB en XML, y convertir un dispositivo USB con sólo una entrada / salida en un socket.

•usb.devices: Este paquete opcional recoge el código Java para acceder a una variedad de dispositivos USB con la API jUSB, incluyendo cámaras digitales Kodak y Rio 500 MP3. Estas APIs están especialmente escritas para simplificar el proceso de acceso a los dispositivos USB designados y no puede ser usado para acceder a otros dispositivos. La API se basa en los usb.core API, para que funcione en cualquier sistema operativo que cuente con el apoyo jUSB.

•usb.view: Este paquete opcional proporciona un simple árbol USB navegador. Es un muy buen programa de ejemplo que ilustra el uso de la API jUSB.

Aunque el objeto de la aplicación usb.core.Host varía de un sistema operativo a otro, el programador de Java debe entender sólo el paquete de usb.core para iniciar el desarrollo de aplicaciones con la API jUSB.

El procedimiento normal para acceder a un dispositivo USB con la API jUSB es el siguiente:

1. Bootstrap obteniendo el USB host de la HostFactory.

2. Acceda al bus USB del host, entonces se accede al concentrador raíz USB (que es un dispositivo USB) del bus.

3. Obtener el número de puertos USB disponibles en el centro, y recorrer a través de todos los puertos para encontrar el dispositivo adecuado.

4. Acceso el dispositivo USB que se adjunta a un puerto en particular. Un dispositivo se puede acceder directamente desde el equipo con su identificador de puertos.

5. Interactuar con el dispositivo directamente mediante mensajes de control.