JSR80 API Specification

Dan Streetman
ddstreet@ieee.org
ddstreet@us.ibm.com
January 11, 2003

Contents

List of Tables

List of Figures

1 Initialization

The entry point for javax.usb is the class *UsbHostManager* (all classes and interfaces are located in the `javax.usb` package unless otherwise noted). This class is final and cannot be instantiated; it is a completely static class. The only public method is *getUsbServices*, which starts up the javax.usb implementation and returns the *UsbServices* implementaion. To find the implementation class, the *UsbHostManager* attempts to load the properties file as defined by the field *UsbHostManager.JAVAX_USB_PROPERTIES_FILE*. It uses the *ClassLoader.getSystemResourceAsStream* method to find the properties file; this method essentially searches the classpath for the file. The properties contained in this file are loaded into a *Properties* instance using the method *Properties.load*, so the file should contain key-value pairs. The only required key-value pair is the key defined by *UsbHostManager.JAVAX_USB_USBSERVICES_PROPERTY*; the value must be the name of the class that implements *UsbServices*. The class must be instantiatable using *Class.newInstance*.

2 UsbServices

The *UsbServices* implementation is the entry point for accessing the javax.usb implementation. It contains informational methods *getApiVersion*, *getImpVersion*, and *getImpDescription*. The API version method returns the version number (String) of the API that the implementation supports; it corresponds to a version (String) returned by the API version class *Version*. The implementation version and description methods return information about the implementation itself.

The interface also allows for adding and removing a *UsbServicesListener* (all listener interfaces and event classes are located in the `javax.usb.event` package). When added, a listener will receive events when devices are connected or disconnected from the topology.

The most important method of *UsbServices* is *getRootUsbHub*. This method returns the virtual root *UsbHub* of the javax.usb topology. This gives access to all devices connected to the host machine.

3 Topology

The root *UsbHub* is the top level of the USB topology tree. It is a virtual device, meaning that it does not correspond to any physical hardware. The javax.usb implementation is responsible for creating and maintaining it. The next level in the topology contains *UsbHub*s which correspond to the physical hardware host controller(s) present in the host system. The next level consists of any devices attached directly to a hardware host controller, which may include external USB hubs (which are also devices). The levels below this correspond to devices connected to external hubs. Note that the USB specification imposes a maximum number of chained external devices to 5, but the API makes no such restriction; the implementation may or may not support more than 5 chained external devices. The implementation is of course required to support up to 5 chained external devices (which means a topology 7 levels deep including the virtual root hub and a hardware host controller hub, plus the 5 external hubs/device).

4 UsbHubs

*UsbHub*s are actually *UsbDevice*s with additional capabilites (methods). They contain *UsbPort*s, which in turn may contain an attached *UsbDevice*. The ports are numbered (starting with port number 1; there is no port number 0), and correspond to the physical ports located on either the hardware host controller or external hub. The exception of course is the virtual root hub, which does not correspond to any physical device at all, and contains enough ports to accomodate all hardware host controllers.

The topology may then be traversed using each hub's ports, which may contain a device (or not). Additionally, all *UsbDevce*s connected to a *UsbHub* may be accessed using the *getAttachedUsbDevices* method.

5 UsbDevice Structure

5.1 UsbDevice

When traversing the topology, it is important to be able to identify what device is of interest. This is possible by using the methods of *UsbDevice*. The most common way to examine a device is the fields contained in its *DeviceDescriptor*. This matches the device descriptor as defined in the USB specification, which contains many fields such as *idVendor* and *idProduct*. The *UsbDevice* also contains methods that allow adding and removing an *UsbDeviceListener*, which receives events related to the device. Each *UsbPipe* contains methods that allow adding or removing an *UsbPipeListener*, which receives events related to that pipe.

5.2 UsbConfig, UsbInterface, UsbEndpoint, and UsbPipe

The actual structure of a *UsbDevice* corresponds to the description from the USB specification. Each device contains one or more *UsbConfig*s. Only one of the configurations can be active at once, and it is possible that the device is in a not configured state, where none of the configurations are active (although this is not likely, as most operating system's USB subsystem configures the device automatically). Each *UsbConfig* contains a *ConfigDescriptor*, which matches the configuration descriptor as defined in the USB specification. Each configuration also contains one or more *UsbInterface*s. The interfaces may or may not have alternate settings, only one of which may be active at once. Each interface setting contains a *InterfaceDescriptor* as well as zero or more *UsbEndpoint*s. Each endpoint contains a *EndpointDescriptor* as well as a *UsbPipe*. The pipe is used for actual communication with the device.

5.3 Default Control Pipe

Communication via the *UsbDevice* occurs on the Default Control Pipe (DCP). This pipe is a special pipe that must be present on all devices; this pipe is always usable, even if the device is in a not configured state. Other communication using the device's pipes, however, is only possible under certain conditions; the pipe must be located on a currently active interface setting, and that interface must be located on the device's currently active configuration. Before using the pipe, the interface it is located on must be *claim*ed, and the pipe must be *open*ed. Once the interface is claimed and the pipe is open, communication may begin on that pipe.

6 Communication

6.1 Submission

Communication with a device is accomplished through the submission methods located on the *UsbDevice* itself and on its *UsbPipe*s. The methods all accomplish the same goal, transferring data to or from a device. They perform this in different ways; the *syncSubmit* (synchronous) methods block until the communication is done, while the *asyncSubmit* (asynchronous) methods return immediately, or as fast as the implementation allows; in any case before the communication is done. For the pipe submission methods, the parameters vary; either a simple byte[], or a *UsbIrp* object, or a list of *UsbIrp* objects can be used. The simple byte[] is a buffer that contains the data transferred to (or from) the device. The irp is a more complicated object, which also contains a byte[], and additionally has fields such as offset and length, which can be used to specify that only a portion of the byte[] should be used. It also can indicate whether short packets (which are described in the USB specification) should be accepted or not. The list simply aggregates multiple irps and allows them all to be submitted at once. The implementation will ensure that no other submissions occur between each irp in the list, and if possible, it will optimize their submission. Particularly in the case of isochronous communication, list submission usually is better and/or faster than other submission methods.

6.2 Control-type Submission

Communication that occurs on a control-type pipe, which includes the DCP, requires additional meta-information about the communication itself. Therefore control pipes can only submit *ControlUsbIrp*s, which contain additional fields. It is not possible to use simple byte[] nor normal *UsbIrp*s on control-type *UsbPipe*s. Communication through the *UsbDevice*'s submission methods also requires *ControlUsbIrp*s, or a list of *ControlUsbIrp*s, since the DCP is a control-type pipe.

6.3 Direction

It is important to note that submissions are the only way to communicate with a device, and although listeners will receive events for all data transferred on a pipe (including the DCP), that data must be provided via submissions. Each pipe is unidirectional, and data may flow only one direction on it. If the pipe is an output pipe, the data located in the byte[] is sent to the device during submission; however if the pipe is an input pipe, the byte[] is filled up with data received from the device. Thus, if input is expected on an input pipe, one or more byte[]s (or irps) must be submitted, and only then will data be received from the pipe. Once all submissions for a pipe are done, no more data will be received on that pipe until more byte[]s (or irps) are submitted. If a constant flow of data is desired, multiple buffers should be submitted, and as each submission finishes, more buffers should be submitted. Using only a single buffer may result in undesirable delays, since the device may be able to produce data at a faster rate than each submission takes, especially if there are sudden bursts of data.

There is an exception to pipes being unidirectional, and that of course is control-type pipes. They are the exception to almost every rule about pipes. Control-type pipes are bidirectional, and the direction is determined on a per-submission basis by a bit in the meta-information. The USB specification explains in detail what is included in this meta-information, which is called a setup packet.

7 Security

Security is not yet fully addressed.

8 Utilities

The API contains several utility classes, all located in the `javax.usb.util` package, which make the API itself easier to use.

8.1 DefaultUsbIrp and DefaultControlUsbIrp

There is a default implementation of the *UsbIrp* interface, as well as a default implementation of the *ControlUsbIrp* interface. These may be used to create irps (and control-type irps) to use in submissions. Additionally, the *UsbPipe* itself contains a method to create *UsbIrp* and *ControlUsbIrp* objects which the implementation may prefer, and may require less overhead to process than either the default implementations or any other implementation. The *UsbDevice* also allows creation of *ControlUsbIrp*s. However, any implementation must be accepted by the implementation; it may not restrict the *UsbIrp* implementation nor the *ControlUsbIrp* implementation.

8.2 StandardRequest

The *StandardRequest* class provides a way to easily perform standard device requests. It contains methods which correspond to all standard device requests defined in the USB specification; however not all those requests may be possible, since some are intended for use only by the low-level USB subsystem driver(s).

8.3 Version

In order to determine what version of the API is in use, a *Version* class is provided in the base `javax.usb` package. It contains methods to determine the version of the API itself, as well as the version of the USB specification that the API supports. It also contains a *main* method so it can be called directly; this method simply prints out the version numbers.