JSR80 API Specification

Dan Streetman
ddstreet@ieee.org
February 12, 2008

Contents

List of Tables

List of Figures

1 Preface

1.1 Introduction

JSR80 is the Java Specification Request concerned with communication with Universal Serial Bus (USB) devices. This document describes the API associated with this JSR.

2 USB Bus Topology

The structure of the USB bus is topological. The root of the USB physical bus is a physical Host Controller, which is contained in a computer system. Host Controllers are usually on-board or may be add-in PCI cards. They appear logically as hubs, so they are the root hub in their own topology tree. Connected to them are one or more USB devices, and since USB hubs are a type of USB device, this topology may continue down until there are no more connected devices. The non-hub devices form the leaves of the topology tree, while the hub-type devices form the branches of the topology tree. This forms the physical device topology.

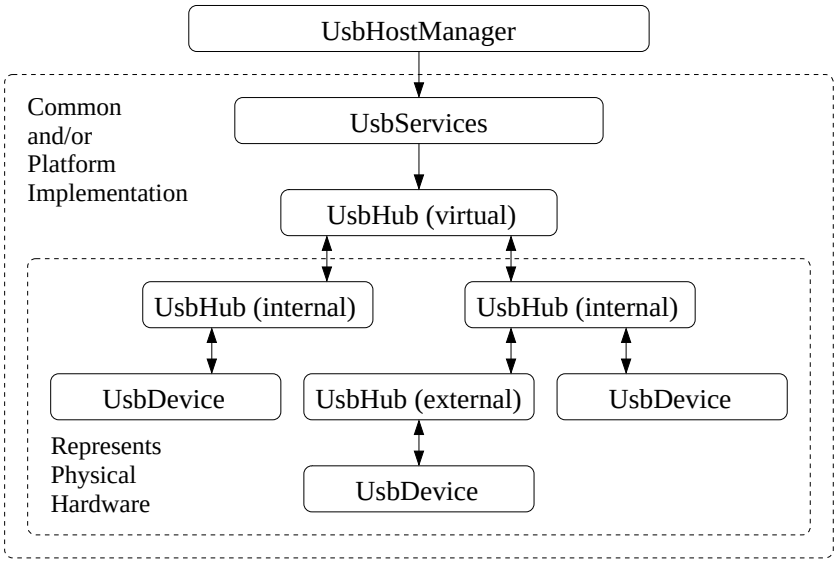

Figure 1: Logical Bus Topology

The javax.usb API closely matches this physical device topology, as shown in figure 1. However the root of the topology is slightly different. First, the entry point of javax.usb is the *UsbHostManager* class, shown at the top of the tree. This class instantiates the platform-specific instance of the *UsbServices* interface. From the *UsbServices* instance, the virtual root *UsbHub* is available. This hub is created by the implementation and does nothing except manage the actual physical devices. Each hub connected to the virtual root hub represents a physical Host Controller hub that is present in the system. Connected to those hubs are the real externally-connected physical devices connected to the system, including external hubs.

3 USB Device Hierarchy

The structure of a USB device is hierarchal, instead of topological. The general structure of a device is defined by the USB specification in Chapter 5. This structure is made up of different components as defined in the specification, and any specific device can organize those components as the device requires, within certain limitations which are also defined in the USB specification. All devices must have one or more configurations. As the name suggests, each configuration represents a different configuration of the device. Only one of the configurations may be active at a time. Each of those configurations must contain one or more interfaces. An interface represents a certain function of the device (or configuration). Any interface has at least one setting, and it may have alternate settings. Only one setting may be active at a time. Each setting may contain zero or more endpoints, and each of those endpoints contains a pipe. The device's pipes, which are used to communicate with the device, are the lowest level of the hierarchy.

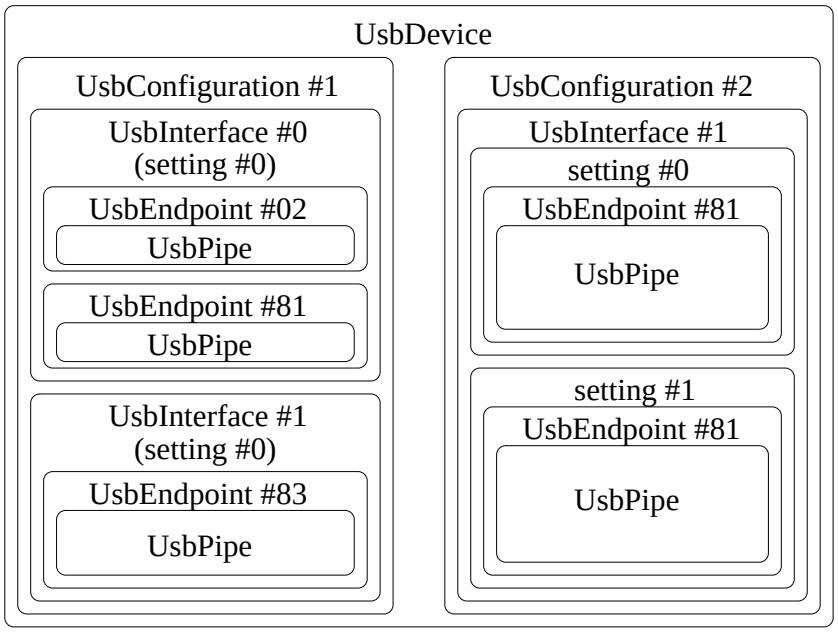

Figure 2: Logical Device Hierarchy

The javax.usb API again closely matches this logical hierarchy. Figure 2 shows the logical arrangement of an example device in javax.usb terms. A physical device is represented by an instance of the *UsbDevice* interface. In this example, there are two *UsbConfigurations*, which represent the device's configurations, inside the *UsbDevice*. Those *UsbConfiguration*s contain *UsbInterface*s; the *UsbInterface*s contain *UsbEndpoint*s; and each *UsbEndpoint* contains a *UsbPipe*. Each of those represents a logical component of the example device, i.e. the device's configurations, interfaces, endpoints, and pipes. To show how alternate settings are handled, inside *UsbConfiguration* number 2, the *UsbInterface* number 1 contains two settings, setting 0 and setting 1.

4 UsbDevice

The *UsbDevice* interface is the connection between the USB bus's topology and the device's logical hierarchy. In addition to navigational methods, the *UsbDevice* interface also has methods used for identification and communication. Figure 3 shows most of the methods provided by the *UsbDevice* interface.

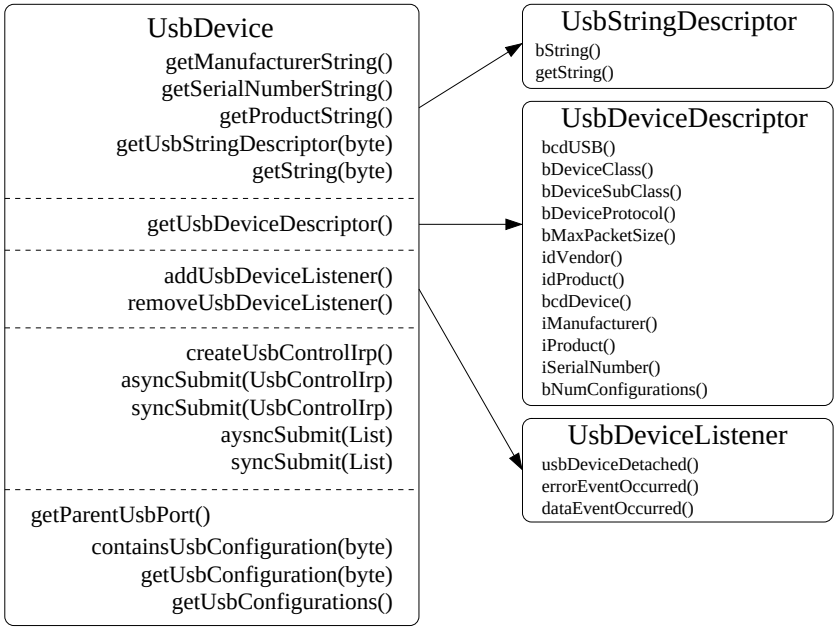

Figure 3: UsbDevice

The navigational methods connect to the *UsbPort* above this device in the bus topology, as well as all the *UsbConfiguration*s below this device in its device hierarchy. The *UsbConfiguration*s are available by specific configuration number or in a list of all configurations.

The identification methods provide the three strings defined in the USB specification, the Manufacturer, Product, and Serial Number. Also there is a method to get any of the device's string descriptors, by number. Finally there is a method to get the device's descriptor. The descriptor, which is represented by an instance of *UsbDeviceDescriptor*, contains methods to get all the values of a device descriptor as defined in the USB specification.

The communication methods provide access to the device's default control pipe. The default control pipe requires *UsbControlIrp*s, which contain meta-information besides the actual data buffer. The *UsbControlIrp*s may be submitted synchronously or asynchronously. The submission methods behave the same, except the synchronous submission method blocks until the submission is complete, essentially meaning all data has been transferred. The asynchronous submission method does not block, but returns as soon as possible, and the submission is completed in a seperate *Thread*. There are also methods that allow submission of a *List* of *UsbControlIrp*s.

Finally, there are methods that allow adding and removing a listener for *UsbDeviceEvent*s. After adding a *UsbDeviceListener*, that listener will get notified whenever a device event occurs. Device events

will get fired whenever and data is transferred, or if any error occurs while transferring data, or if the device is physically disconnected.

5 UsbConfiguration

A *UsbConfiguration* is the next interface in the device hierarchy. As shown in figure 4, the *UsbConfiguration* is much less complicated than the *UsbDevice*. It provides some identification methods and navigational methods, as well as a method for checking if this configuration is active or not.

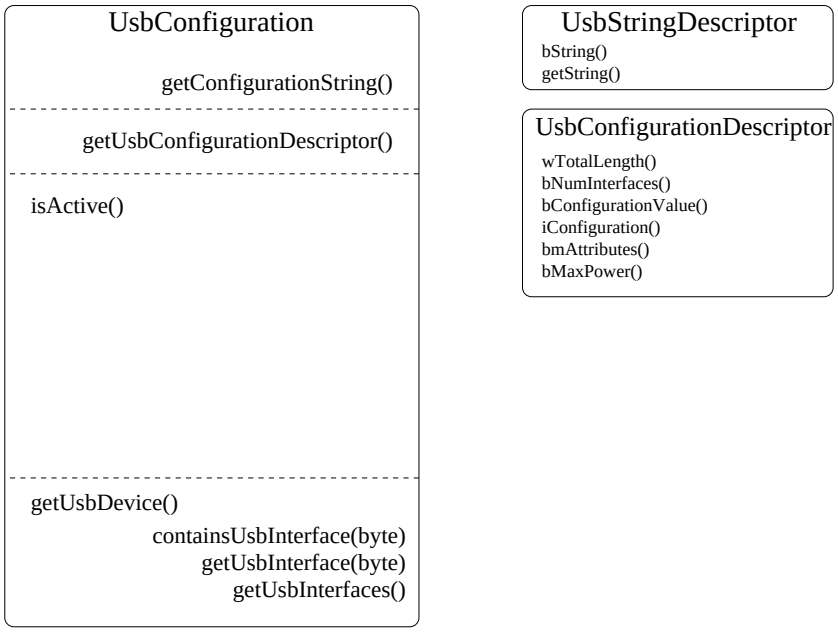

Figure 4: UsbConfiguration

The navigational methods are similar to the *UsbDevice*'s navigational methods, they allow access to the interfaces above and below in the device hierarchy. There is a method that leads to the *UsbDevice* instance above this in the device hierarchy. Also there are methods that lead to the *UsbInterface*s below this configuration in the device hierarchy. The interfaces are available by number or in a list of all available interfaces. If any interface has more than one alternate setting, and the configuration itself is active, then the active alternate setting of the interface is provided. If the configuration itself is not active, an implementation-dependent alternate setting is provided.

The identification methods include a method to get the string descriptor for this configuration, if one exists, and a method to get the configuration descriptor associated with this configuration. The *UsbConfigurationDescriptor* provides methods corresponding to the fields listed in the USB specification.

6 UsbInterface

Next in the device hierarchy is the *UsbInterface*. The *UsbInterface* is similar to those interfaces above it, as shown in figure 5; it has identification, navigation, and status methods. The navigational methods are more complicated however, because the interface may have alternate settings. Additionally, the status methods are more complicated.

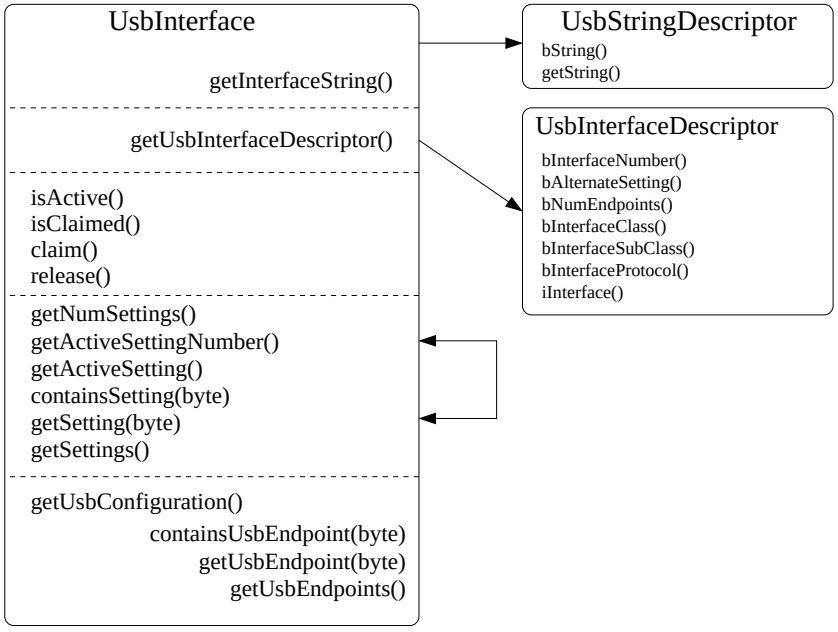

Figure 5: UsbInterface

The navigational methods allow access to the *UsbConfiguration* above this interface in the device hierarchy, as well as accessing all the endpoints below this interface in the device hierarchy. As usual, the *UsbEndpoint*s are available by number, more specifically endpoint address, as well as a list of all available endpoints. In addition to those navigational methods, there are methods that allow access to any alternate settings the interface may have.

The identification methods provide the string descriptor for this interface, if one exists, as well as the interface descriptor associated with this interface. The *UsbInterfaceDescriptor* contains methods corresponding to all the fields listed in the USB specification.

The status methods are slightly more complicated than other status methods. There is a method to check if this interface setting is active. This method is only true if the parent *UsbConfiguration* is active as well as this *UsbInterface* setting. Also, there are methods to allow claiming and releasing of the interface, and a method to check if the interface has been claimed. The claiming is used to aquire an exclusive lock on the interface and all the endpoints and their pipes under the interface. Before sending or receiving data to or from any of the pipes under this interface, the interface must be claimed.

7 UsbEndpoint

The *UsbEndpoint* is quite simple in comparison to other device hierarchy interfaces. As shown in figure 6, the only methods are navigational methods to access the *UsbInterface* above this, and the *UsbPipe* below. Also, there is a method to access the *UsbEndpointDescriptor*, and methods to get the direction and type of endpoint and its associated pipe.

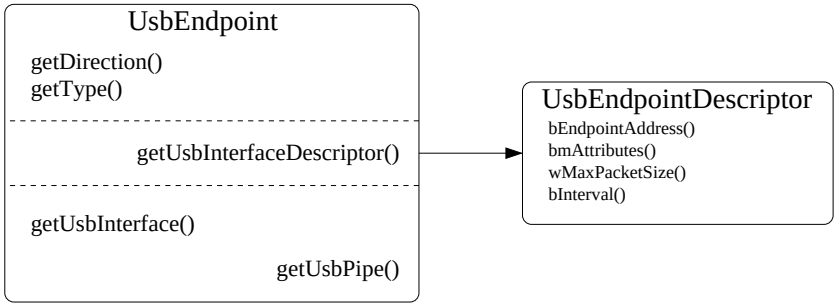

Figure 6: UsbEndpoint

8 UsbPipe

The *UsbPipe* is the lowest interface in the device hierarchy. As shown in figure 7, it has navigational and status methods, but most of its methods are communication methods. The navigation method allows access to the *UsbEndpoint* that this pipe is associated with, and there is a method to determine if this pipe is active or not. A pipe is active if its parent interface setting is active.

In addition to the method to determine if the pipe is active, there are methods that allow opening and closing of the pipe, and a method to determine if the pipe is open. The pipe must be opened before it can be used for communication. Also, the parent interface must be claimed before the pipe can be opened.

The communication methods are similar to the communication methods from the *UsbDevice* interface, which allow communication on the default control pipe. However, this pipe is not neccesarily a control-type pipe, and so other objects may be used for the communication. There are synchronous and asynchronous methods, which behave the same as the methods from the *UsbDevice* interface; the synchronous methods block until complete, while the asynchronous methods return immediately and perform processing in a background thread. If the pipe is a control-type pipe, the object used for communication must be a *UsbControlIrp*, just as is used with the default control pipe. However for non-control-type pipes, a *UsbIrp* can be used as well as a simple byte[]. The byte[] is the simplest method of communication. It simply transfers all the contained data to the device, if the pipe direction is host-to-device, or fills up part or all of the buffer with data from the device, if the pipe direction is device-to-host. The *UsbIrp* is similar, it contains a byte[] data buffer also, but there are other methods to limit the amount of transferred data, or start the transfer at a certain offset into the buffer, or cause an error if any short packets are detected.

The *UsbPipe* also has methods that allow a *UsbPipeListener* to be added or removed from the pipe. After a *UsbPipeListener* has been added, it will receive a *UsbPipeEvent* when any data is transferred to or from the device on this pipe, or when any error occurs while transferring data. After removing the listener, it will not receive any more events.

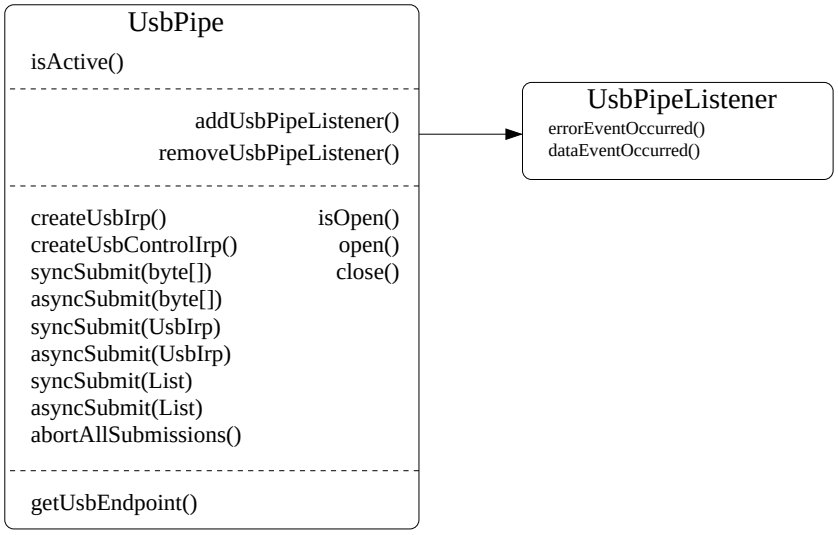

Figure 7: UsbPipe

9 UsbControlIrp and UsbIrp

The interfaces used to actually communicate with the device are shown in figure 8. The *UsbControlIrp* interface is actually an extension of the *UsbIrp* interface, adding methods that represent fields in the special control-type header required for control communication. Otherwise, the *UsbControlIrp* is identical to a normal *UsbIrp*. A *UsbIrp* has methods that can be divided into four different categories; the data buffer, status, short packets, and the *UsbException*.

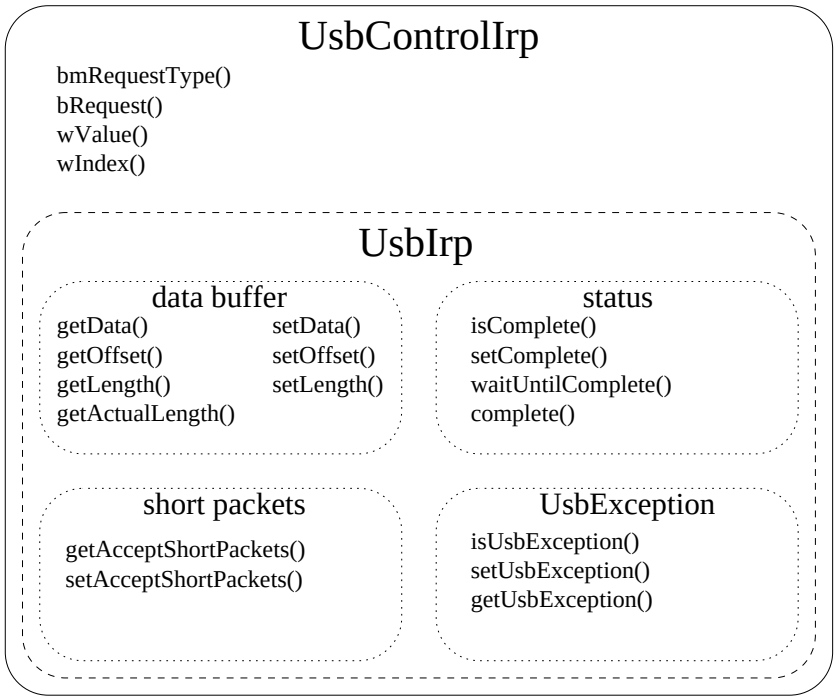

Figure 8: UsbControlIrp and UsbIrp

First, the data buffer is simply a byte[] but there are additional methods that are used to modify the transfer of data. There are methods to set and get the offset, which indicates what offset into the data buffer the implementation should use when transferring data. If the offset is zero, data will be transferred starting at the beginning of the byte[], if the offset is above zero, data starting at the offset into the byte[] will be used when communicating with the device. There are also methods to set and get the length of data to transfer with the device. Lastly, there is a method to get the actual amount of data transferred with the device. For an input-direction pipe, this value may be less than the length of data that was requested from the device; for an output-direction pipe, this should be the full amount of data to transfer.

The status methods deal with the complete status of the *UsbIrp*. There is a method to check if the *UsbIrp* is complete or not. There are also methods to wait until the *UsbIrp* is complete; these will block until the *UsbIrp* changes status to complete. There is a method used to set the *UsbIrp* complete or not complete, and finally there is a method that is used by the implementation to set the *UsbIrp* as complete after finishing processing. This method will also call the method to set the status of the *UsbIrp*

as complete, and cause any *Thread*s waiting for the *UsbIrp* to complete to return from the methods they are blocking in.

The methods related to short packets are used to set and get the policy that should be used when handling short packets. The policy can be set to either accept or reject short packets. Short packets will happen if the device transfers less data than the host was expecting. Normally this will happen only if the pipe direction is device-to-host, and the data buffer provided is larger than the amount of data that the device has to send to the host during a specific communication. If short packets are accepted, and a short packet occurs, the communication will complete successfully and the actual length of transferred data will be less than the size of the provided data buffer. If short packets are not accepted, and a short packet occurs, the *UsbIrp* will complete with an error.

Finally, there are methods used to manage any UsbException that may occur during communication on the pipe. There is a method to check if an *UsbException* has occurred during communication on the pipe, as well as methods to get and set any *UsbException* that occurs.

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, 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 data buffers 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 data buffers 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.

10 Default Control Pipe

The Default Control Pipe is used for much of the communication with a device. It is a special pipe that is required to be present on all devices, and it is always available for communication. It is used for all the configuration-type communication, as well as communication to get identification information from the device.

In figure 9, an example of communication using the Default Control Pipe is shown. First, the user has to obtain an instance of a *UsbControlIrp*. The user can use their own implementation of *UsbControlIrp*, or they can use the implemenation provided by the *UsbDevice* implementation, as shown in the figure. In the example, the *UsbControlIrp* is created by the *UsbDevice*, and the extra control-type methods are set up during creation. Next, it should be given a data buffer, containing data to transfer to the device or a buffer to fill with data from the device, depending on the direction of communication. This is the minimum amount of configuration of the *UsbControlIrp* required.

To actually perform the communication on the Default Control Pipe, the *UsbControlIrp* is submitted using, in this example, the *syncSubmit* method. The implementation of this method will pass the buffer down to the platform's low-level USB subsystem, which will perform the actual transfer of data. After the data transfer is complete, the *UsbDevice* implementation will set either the actual length of transferred data or the *UsbException* on the *UsbControlIrp*, and then *complete* the *UsbControlIrp*. This is the simplest communication example, which does not show the use of data offset or length, and assumes no listeners are being used.

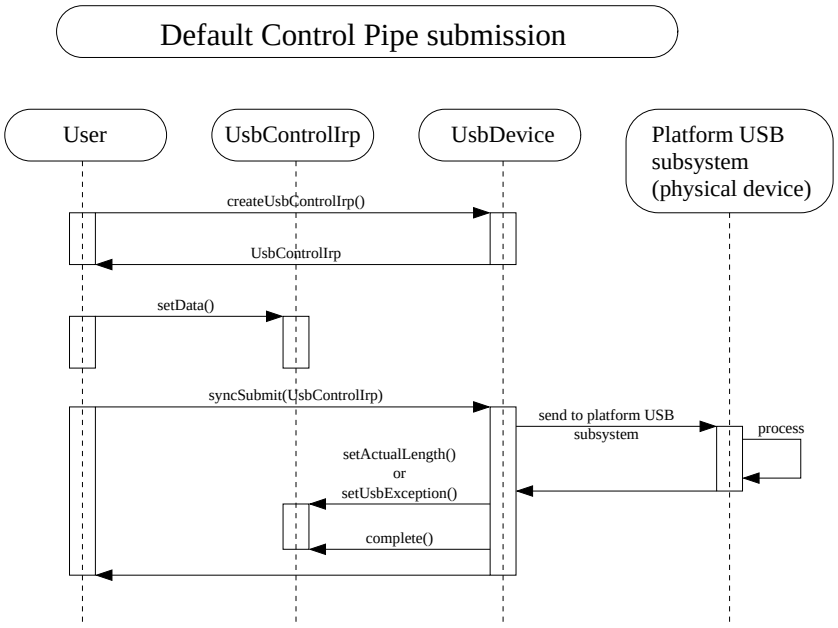

Figure 9: Default Control Pipe

11 UsbPipe submission, synchronous

For most communication on a device, the Default Control Pipe is not used. Instead, normal unidirectional pipes are used to transfer data. Figure 10 shows an example of communication using one of the normal *UsbPipe*s. There are more steps to take before actually submitting the *UsbIrp* than when using the Default Control Pipe.

First, before using any of the *UsbPipe*s contained in a *UsbInterface* setting, the *UsbInterface* must be claimed. Then, the *UsbPipe* must be opened. If either of these steps fails, no communication will be possible on that pipe; the problem that prevented the claim or the open must be corrected first. If the claim and the open succeed, the pipe is ready for communication. The interface setting may be left claimed and the pipe may be left open as long as the interface and its pipes are being used.

To actually communicate on the pipe, data buffers must be provided to the pipe. In this example, a *UsbIrp* is used. The *UsbIrp* implementation may be provided by the caller, but the example shows using the *UsbPipe* to create a *UsbIrp* instance. After creating the *UsbIrp*, at least the data must be set. Additionally, the other fields of the *UsbIrp* may be set, including the offset, length, and short packet policy of the *UsbIrp*. Those fields are optional, and their defaults should be acceptable for the majority of communication. Once the data buffer and any optional fields are set, the *UsbIrp* must be provided to the *UsbPipe*; in the example, the *syncSubmit* method is used. Similar to the Default Control Pipe example, the implementation passes the data buffer to the platform USB subsystem, which performs the actual communication with the device. After the communication is complete, the implementation either sets the actual length of the data transferred, or it sets the appropriate *UsbException* on the *UsbIrp*. It then calls the *UsbIrp*'s *complete* method to indicate the *UsbIrp* is completed. While it is not shown in the example, if any *UsbPipeListener*s had been added to the *UsbPipe*, they would receive either a *UsbPipeDataEvent* or a *UsbPipeErrorEvent*, depending on whether the communication succeeded or not.

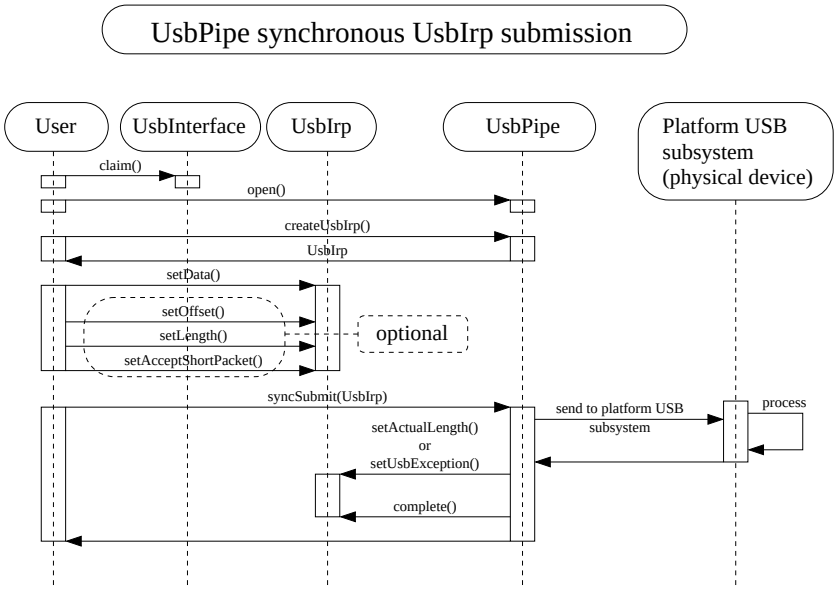

Figure 10: UsbPipe synchronous submission

12 UsbPipe submission, asynchronous

Asynchronous submission on a pipe is very similar to synchronous submission. As shown in figure 11, the initial setup is the same; the *UsbInterface* setting must be claimed, and the *UsbPipe* must be opened. The *UsbIrp* should be created in the same way, and the data buffer set as well as any of the optional fields such as offset or length. This example also shows how to use a *UsbPipeListener*. Before submitting the *UsbIrp*, the listener should be added to the *UsbPipe*. The listener only needs to be added once to the pipe. Next, the *UsbIrp* is submitted to the pipe using the *asyncSubmit* method. The submitting *Thread* should return immediately, while the *UsbIrp* is processed in the background by the implementation. While the implementation is processing the *UsbIrp*, the calling *Thread* can perform other actions. Eventually, it will want to know if the *UsbIrp* is complete. The calling *Thread* can then call the *waitUntilComplete* which will block until the *UsbIrp* is complete.

Once the platform USB subsystem has finished transferring the data, it will return it to the implementation. The implementation handles it normally, by setting either the actual length of data transferred, or the *UsbException* that occurred, and finally completing the *UsbIrp*. Next the *UsbPipeListener* will receive a *UsbPipeDataEvent* or *UsbPipeErrorEvent*, depending on whether the submission was successful or not.

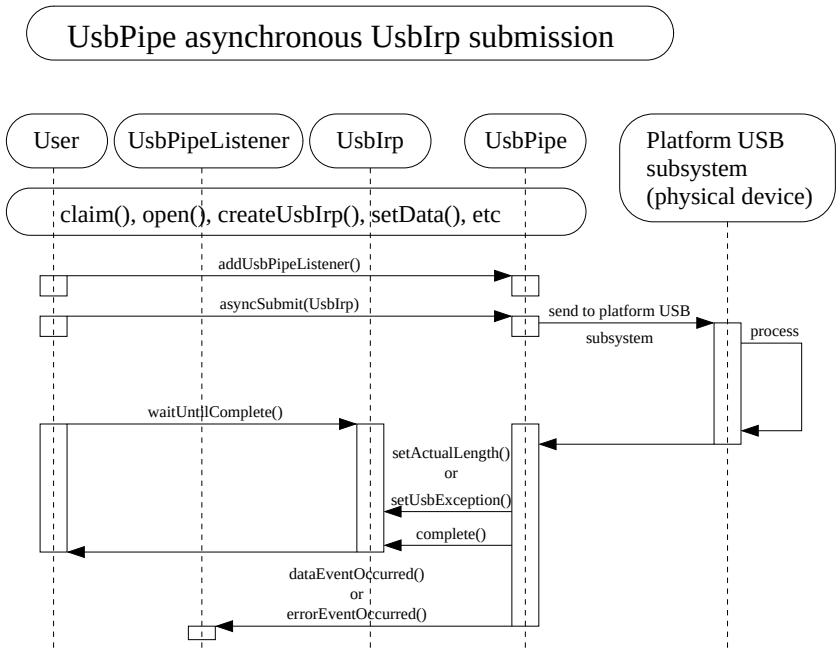

Figure 11: UsbPipe asynchronous submission

13 UsbPipe submission, byte[]

Submissions using *UsbIrp*s are the most flexible type of communication. Submissions using byte arrays are much more simple, however. In figure 12, both synchronous and asynchronous submission using a byte[] are shown. The setup steps, such as claiming the interface and opening the pipe, are the same. After these initial steps, in the first example using asynchronous submission, the byte[] is submitted to the *UsbPipe* using the *asyncSubmit* method. The implementation then creates a *UsbIrp* to represent the provided data buffer, and this *UsbIrp* is returned to the caller. Then processing continues the same as asynchronous submission of a *UsbIrp*. In the synchronous example, the data buffer is passed to the *UsbPipe* using the *syncSubmit* method, which blocks while the implementation handles transferring the data. After the data has finished transferring, the implementation either returns the actual length of data transferred, or it throws a *UsbException* representing the error that occurred during data transfer. In both cases, after the data transfer is complete or an error occurs, any listeners that have been added to the pipe will receive a *UsbPipeDataEvent* or *UsbPipeErrorEvent*.

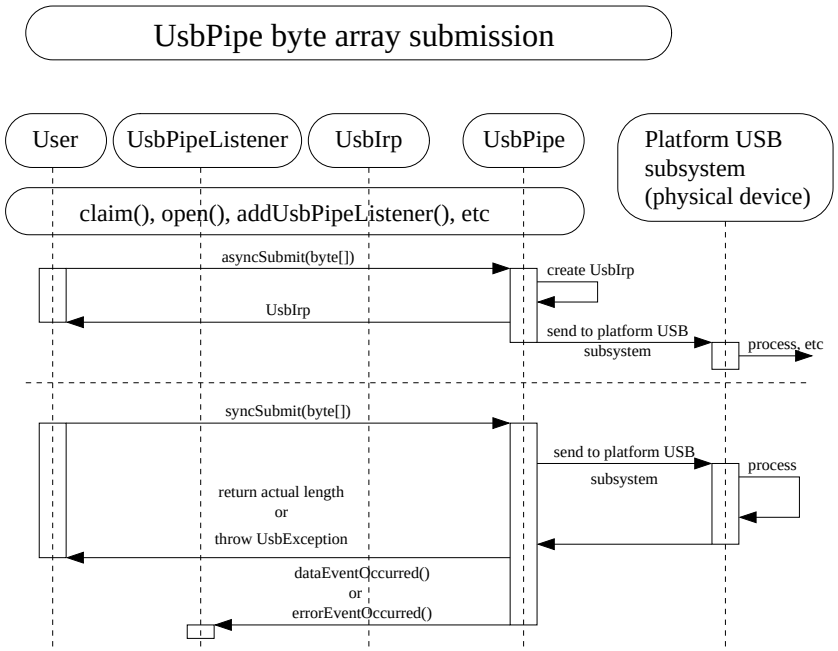

Figure 12: UsbPipe byte array submissions

14 Hotplugging

An important feature of the USB bus is the ability for USB devices to be connected and disconnected while the system is running. This is called hotplugging, and system software must be able to handle hotplugging a device. Hotplugging is fully supported by the javax.usb subsystem. Figure 13 shows how the user can get notified of hotplugging events. First, the user can add a *UsbServicesListener* to the *UsbServices* instance. When a physical device is connected to the system, the *UsbServicesListener* receives a *UsbServicesEvent* indicating that a device has been connected to the system. If the user is interested in this newly connected device, they can add a *UsbDeviceListener* to the *UsbDevice*. Then, when the device is disconnected, both the *UsbServicesListener* and the *UsbDeviceListener* will be notified with a *UsbServicesEvent* and a *UsbDeviceEvent*, respectively.

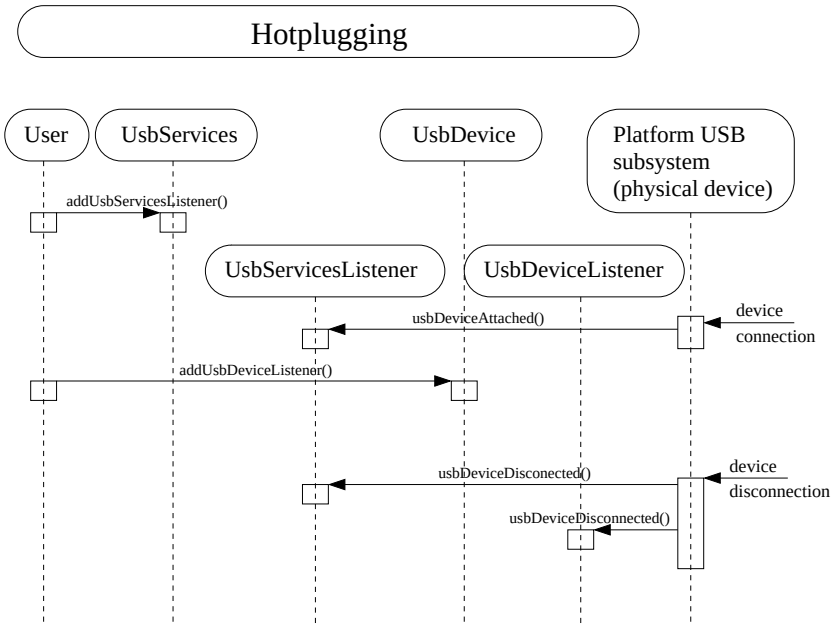

Figure 13: Hotplugging

15 Exceptions

The javax.usb subsystem uses many *Exception*s to represent some of the errors that can occur on the USB bus. Figure 14 shows the *Exception*s used. The base *Exception* is *UsbException*, which is extended by many more specific *Exception*s such as *UsbBabbleException*, *UsbCRCException*, and others. Also, there are some *RuntimeException*s such as *UsbNotActiveException* and *UsbNotClaimedException*.

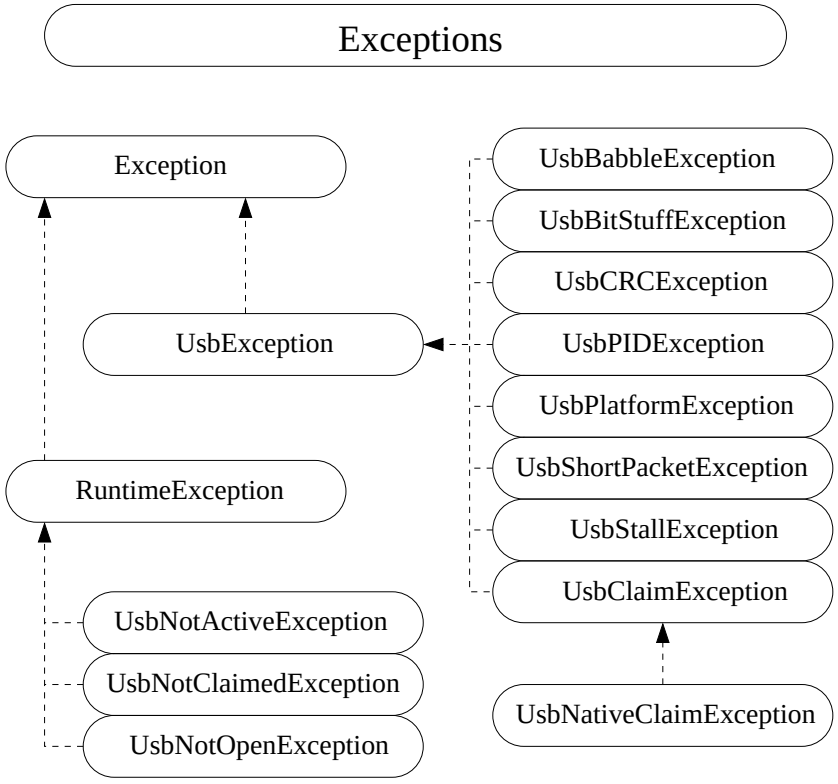

Figure 14: Exceptions

16 Security

Security is not yet fully addressed.

17 Utilities

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

17.1 DefaultUsbIrp and DefaultUsbControlIrp

There is a default implementation of the *UsbIrp* interface, as well as a default implementation of the *UsbControlIrp* 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 *UsbControlIrp* 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 *UsbControlIrp*s. However, any implementation must be accepted by the implementation; it may not restrict the *UsbIrp* implementation nor the *UsbControlIrp* implementation.

17.2 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).

17.3 Version

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.