Javax.usb TCK Test Suite

Version 1.0.4-IBMChanges

October 10, 2004

Dale Heeks
dheeks@us.ibm.com
Kevin Bell
khbell@us.ibm.com
Leslie Blair
lkblair@us.ibm.com

1 Summary of Changes

Changes resulting in document revisions will be summarized in this table in forward chronological sequence.

Version	Date	Change Description
0.01	11/27/2001	Original
0.02	01/05/2003	Modify test cases to more efficiently and exhaustively test javax.usb
1.0	07/01/2003	Modify tests to test updated javax.usb version 0.10.0
1.0.1	9/29/2003	Update tests with transfer info, expected results, and additional detail
1.0.2	12/06/2003	Minor updates from review.
1.0.3	01/07/2003	Minor updates post review
1.0.4	10/04/2004	Reflect changes made to BULKINT image

The comments received in the review meetings must be added at the end of this document under Document History section.

2 Introduction

The purpose of this document is to provide a specification for the Test Suite of the Technology Compatibility Kit (TCK) for the Java USB API Java Specification Request 80 (JSR080). The Java Specification Request is governed by the Java Community Process. This TCK provides a means to verify that any implementation of the Java USB API has been implemented according to the JSR080 technology specification.

For information on the Java Community Process, see http://www.jcp.org. The home page for JSR080 is http://javax-usb.org/.

3 Javax.usb TCK Hardware Requirements

Hardware required for running tests:

- Computer with two or more USB host controllers
- 4 USB hubs of any configuration
- 1 4-port USB hub
- Cypress EZ-USB FX Integrated Circuit Xcelerator Development Kit CY3671
- A low speed USB device
- A USB device with at least one non-default control pipe

The **Cypress Development Kit** is a USB Device which can be programmed to represent a USB with multiple configurations and endpoints.

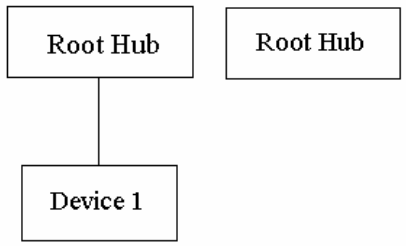
See Appendix A for the specification of the Cypress Development Kit function.

The EEPROM on the Cypress Development Board must be programmed with an image provided with the TCK. The image will be a B6 image which can be programmed into the EEPROM so that the development board can be used as a stand-alone USB device. Information on programming the EEPROM is provided with the Cypress Developmenrt Kit.

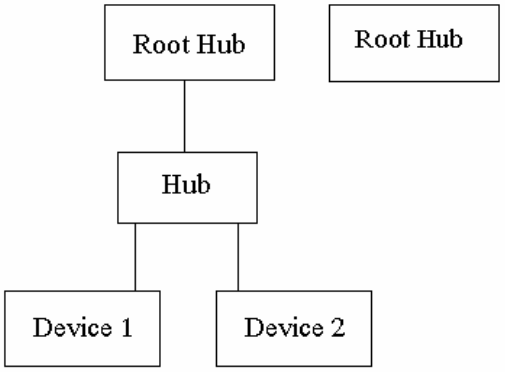
4 Hardware Configurations

The following configurations will be used for testing purposes:

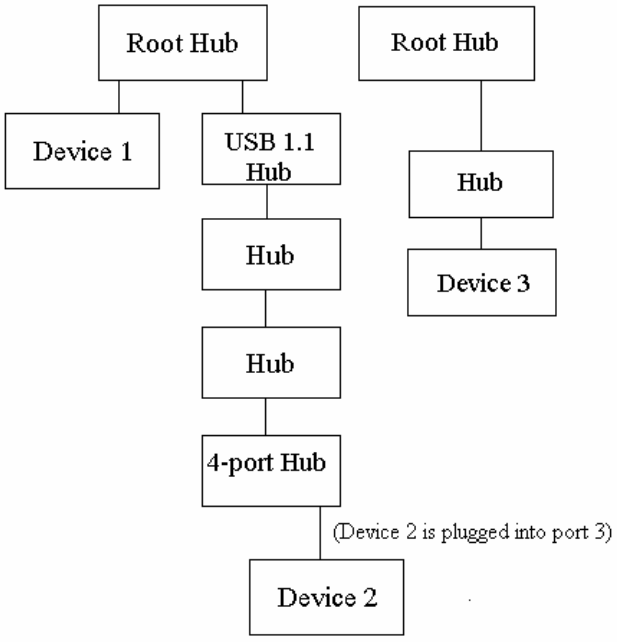
Configurations 1, 2, and 3 will be used for the Topology test. Configuration 3 will be used for all other testing.



Configuration 1



iguration 2



Configuration 3

Device 1 — Cypress Development Board
Device 2 — Any low speed USB device
Device 3 -– USB device with at least one non-default control pipe
Computer has 2 or more USB root hubs. No root hubs can have devices attached except as shown in the configurations.

5 Javax.usb TCK Software Requirements

The TCK test suite uses the JUnit testing frameworks as a harness for the test suite. The JUnit homepage can be found at www.junit.org. Junit 3.8.1 or later is required. The TCK will be tested using Junit 3.8.1.

The Ant build tool is required for building the TCK. The Apache Ant Project home page can be found at ant.apache.org. Ant 1.6.0 or later is required. The TCK will be tested using Ant 1.6.0.

A Java SDK is required to build the TCK and a Java JRE is required for running the TCK. The TCK will be tested with Java 1.4.1.

There must be no driver on the target operating system which will claim any interface on the Cypress Development Board.

There must be no driver on the target operating system which will claim any interface on the USB device with at least one non-default control pipe.

Documentation provided with the TCK will provide specific version information for the required software.

6 Javax.usb TCK Test Suite

Javax.usb TCK test suite consists of the following tests :

- ***API Signature Validation Tests***

  Signature *Test*

- ***Topology Tests***

  Topology *Test*

- ***I/O Tests***

  DefaultControlPipe *Test*
  Control I/O *Test*
  Bulk I/O *Test*
  Interrupt I/O *Test*
  Isochronous I/O *Test*
  Request *Test*

- ***Other Tests***

  IRP *Test*
  Constants *Test*
  Hot Plug *Test*

Each test is described in the following sections with these details:

- Goal of the test
- Technique used to perform the test
- Classes and methods tested

6.1 Signature Test

Goal :

This is the API signature test for the entire **javax.usb**.* package. This test verifies the existence of all expected public classes and interfaces. In those public classes and interfaces it verifies the existence, correct modifiers, arguments, and throws clauses of public constructors and methods. It also verifies the existence and correct modifiers of public fields.

Technique :

A signature file <javax.usb-1.0.sig> is bundled with the test suite. Reflection is used to verify that all classes and interfaces identified in the signature file exist and have an exact binary compatible match in the javax.usb implementation. The signature test will look for the javax.usb implementation in the classpath.

Classes and Methods Tested :

N/A

6.2 Topology Test

Goal :

This test verifies that the API can traverse a hierarchical device topology and that it properly reveals the correct pre-configured topology as defined by the test hardware setup. For the UsbDevice methods (and UsbConfig, etc. down to the UsbEndpoint) only the Cypress development board is used, other devices are present only to verify the accuracy of the topology tree.

Technique :

The list of classes and methods to be covered by this test is listed at the end of the test. Some of the methods and classes are identified explicitly in the test descriptions, but those that are not mentioned must also be called at least once in this test.

Get the virtual root UsbHub to which all Host Controller UsbHubs are attached.

Traverse the topology and verify it matches the defined hardware configuration. Verify the result of each accessor method against the expected value, ensure that specific values are compared only when the expected value cannot vary between correctly configured hardware setups. For methods where the values cannot be guaranteed, verify only that the method can be called without error and that any returned values match the return type expected.

For configurations which are not active, the device model (UsbInterfaces, UsbEndpoints, and UsbPipes) may still be accessed, but no functions of the configuration can be used. The configuration of the Cypress Development Board will be set as required for each test. All interfaces for each configuration default to alternate setting 0.

The computer required for the topology tests must have at least two root or more internal hubs. For each configuration, only devices shown in the configuration can be present. Additional root hubs not shown in the configuration can have no devices attached to them.

For each configuration, specifically test for the following:

- Configuration 1
    - Verify that there are two or more root hubs
    - Verify that there is a single device, the Cypress Development Board (Device 1), attached to one of the root hubs and that at no other root hub has devices attached.
- Configuration 2
    - Verify that there are two or more root hubs
    - Verify that there is a single hub attached to one of the root hubs
    - Verify that a the Cypress Development Board (Device 1) and a low speed device (Device 2) are attached to the single hub
- Configuration 3

- Verify that there are two or more root hubs
- Verify that a single hub is attached one root hub and that a single device with at least one non-default control pipe (Device 3) is attached to the hub
- Verify that the Cypress Development Board (Device 1) and a hub are attached to another root hub
- Verify that all other root hubs have no devices attached
- Verify that there are exactly four hubs between the root hub and the low speed device (Device 2) and that the hub to which the low speed device is attached has exactly four ports.
- Verify that the low speed device is in port 3
- For the Cypress Development Board, verify all methods of UsbDevice, UsbDeviceDescriptor, UsbDescriptor, UsbConfiguration, UsbConfigurationDescriptor, UsbInterface, UsbInterfaceDescriptor, UsbEndpoint, UsbEndpointDescriptor, and UsbPipe against expected values.

Classes and Methods Tested :

- UsbHostManager
  - getUsbServices() throws UsbException,SecurityException
  - getProperties() throws UsbException,SecurityException
- UsbServices
  - getRootUsbHub() throws UsbException, java.lang.SecurityException
  - getApiVersion()
  - getImpVersion()
  - getImpDescription()
- UsbPort
  - getPortNumber()
  - getUsbHub()
  - getUsbDevice()
  - isUsbDeviceAttached()
- UsbHub
  - getNumberOfPorts()
  - getUsbPorts()
  - getUsbPort(byte)
  - getAttachedUsbDevices()
  - isUsbRootHub()
- UsbDevice
  - getParentUsbPort()
  - isUsbHub()
  - getManufacturerString()
  - getSerialNumberString()
  - getProductString()
  - getSpeed()
  - getUsbConfigurations()
  - getUsbConfiguration(byte number)
  - containsUsbConfiguration(byte number)
  - getActiveUsbConfigurationNumber()
  - getActiveUsbConfiguration()
  - isConfigured()
  - getUsbDeviceDescriptor()
  - getUsbStringDescriptor(byte index)
  - getString(byte index)
- UsbDeviceDescriptor

    - bcdUsb()
    - bDeviceClass()
    - bDeviceSubClass()
    - bDeviceProtocol()
    - bMaxPacketSize()
    - idVendor()
    - idProduct()
    - bcdDevice()
    - iManufacturer()
    - iProduct()
    - iSerialNumber()
    - bNumConfigurations()

- UsbDescriptor
    - bLength()
    - bDescriptorType()

- UsbConfiguration
    - getUsbInterfaces()
    - getUsbInterface(byte number)
    - containsUsbInterface(byte number)
    - getUsbDevice()
    - getConfigurationDescriptor()
    - getConfigurationString()
    - isActive()

- UsbConfigurationDescriptor
    - wTotalLength()
    - bNumInterfaces()
    - bConfigurationValue()
    - iConfiguration()
    - bmAttributes()
    - bMaxPower()

- UsbInterface
    - getNumSettings()
    - getUsbEndpoints()
    - getUsbEndpoint(byte address)
    - containsUsbEndpoint(byte address)
    - getActiveSettingNumber()
    - getActiveSetting()
    - getSetting(byte)
    - containsSetting(byte)
    - getSettings()
    - getUsbConfiguration()
    - getUsbInterfaceDescriptor()
    - getInterfaceString()
    - isActive()

- UsbInterfaceDescriptor
    - bInterfaceNumber()
    - bAlternateSetting()
    - bNumEndpoints()
    - bInterfaceClass()
    - bInterfaceSubClass()
    - bInterfaceProtocol()
    - iInterface()

- UsbEndpoint
    - getDirection()
    - getType()
    - getUsbPipe()
    - getUsbInterface()
    - getUsbEndpointDescriptor()

- UsbEndpointDescriptor

    - bEndpointAddress()
    - bmAttributes()
    - wMaxPacketSize()
    - bInterval()

- UsbPipe
    - getUsbEndpoint()
    - addUsbPipeListener(UsbPipeListener listener)
    - removeUsbPipeListener(UsbPipeListener listener)

- UsbStringDescriptor
    - bString()
    - getString() throws java.io.UnsupportedEncodingException

- Version
    - getApiVersion()
    - getUsbVersion()
    - main(`java.lang.String[] args`)

6.3 DefaultControlPipe Test

Goal :

This test verifies that **control transfers** operations work successfully on the **Default Control Pipe** and that proper events are generated and proper exceptions are thrown in the operation.

Technique :

A separate table defining the data transfers to be performed and expected results is provided for the Default Control Pipe. The transfers will be repeated once for synchronous submits and once for asynchronous submits.

The list of classes and methods to be covered by this test is listed at the end of the test. Some of the methods and classes are identified explicitly in the test descriptions, but those that are not mentioned must also be called at least once in this test.

Access Device

- Traverse topology looking for Cypress Development Board.
- Add usbDevice.addUsbDeviceListener(UsbDeviceListener)

Transfers

Transfer tables are defined in Appendix D.
First transfers will be performed with syncSumbit() repeat all transfers with asyncSubmit().
Perform each transfer 10 times.

For each transfer defined in the table

- Create the output buffer(s) of the specified size and fill it with the appropriate data
  - For IRP
    - Use usbDevice.createUsbControlIrp(bmRequestType, bRequest, wValue, wIndex)
    - Set data with usbControlIrp.setData(byte[]).
  - For IRP list
    - Create a list of IRPs following steps for IRP above, except use a single byte[] for any data required in the list of IRPs.
    - Set data in each IRP with usbControlIrp.setData(byte[], int, int).
- Create an input buffer(s) of the specified size
  - For IRPs
    - Use usbDevice.createUsbControlIrp(bmRequestType, bRequest, wValue, wIndex)
    - Perform usbControlIrp.setAcceptShortPacket(false) according to table.
  - For IRP List
    - Create a list of IRPs following steps for IRP above except use a single byte[] for the list of IRPs.

        - Set data in each IRP with usbControlIrp.setData(byte[], int, int).

Synchronous submissions

- usbDevice.syncSubmit(IRP)
    - Verify no exceptions on submit unless noted in transfer table.
    - Blocks until complete.
    - Verify usbControlIrp.isComplete() is true.
    - Verify usbControlIrp.isUsbException() according to transfer table.
        - If true, verify usbControlIrp.getUsbException() against expected exception. and
        - If false, verify usbControlIrp.getUsbException() is null and usbControlIrp.getActualLength() returns the expected length actually sent or received.
    - Verify usbControlIrp.getData() that data is not altered on transfer OUT and matches expected transformed data IN.
    - Verify usbControlIrp.getOffset() and usbControlIrp.getLength() are unchanged after submit.
    - Verify usbDeviceListener.dataEventOccurred or errorEventOccurred for each IRP. The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.
- usbDevice.syncSubmit(List of IRPs)
    - Verify no exceptions on submit unless noted in transfer table.
    - Blocks until all IRPs in list complete.
    - Perform all verifications listed under syncSubmit(IRP) for each IRP in list.

  Since syncSubmit() blocks until it is complete, the OUT buffer must be submitted before the IN buffer; otherwise, the syncSubmit() for the IN would never get a data event and complete.

  For syncSubmit(), all exceptions will be thrown on the submit as well as retrievable by getUsbException() and isUsbException().

Asynchronous submissions

- usbDevice.asyncSubmit(IRP)
    - Verify no exceptions on submit unless noted in transfer table.
    - Does not block until complete.
    - For OUT, call usbIrp.waitUntilComplete(5000) after submit.
    - For IN, call usbIrp.waitUntilComplete(5000) after associated OUT submit is complete.
    - Perform all verifications listed under syncSubmit(IRP) for IRP.
    - Verify usbDeviceListener.dataEventReceived or errorEventOccurred for each IRP. The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.
- usbDevice.asyncSubmit(List of IRPs)
    - Same as asyncSubmit(Irp) except for the following
    - Call usbIrp.waitUntilComplete(5000) for OUT IRPs, in order

- Call usbIrp.waitUntilComplete(5000) for IN IRPs, in order after OUTs complete.
- Perform all IRP and event verifications for each IRP in list.

For the default control pipe, the OUT buffers must be asyncSubmitted before the IN buffers. Call waitUntilComplete(5000) as listed above.
For asyncSubmit(), exceptions which occur on the submit will be thrown, but exceptions which occur after the submit has returned will not be thrown. However, all exceptions on the submit whether they are thrown or not are retrievable by getUsbException() and isUsbException().

Remove listener

- Remove usbDevice.removeUsbDeviceListener()

Error Conditions

- Null data buffer
  - Use usbDevice.createUsbControlIrp(…) and call usbIrp.setData() with a null data buffer. setData(..) should throw a java.lang.IllegalArgument Exception.

Classes and Methods Tested:

- UsbDevice
  - syncSubmit/asyncSubmit(UsbControlIrp irp) throws UsbException. IllegalArgumentException
  - syncSubmit/asynchSubmit(List list) throws UsbException,IllegalArgumentException
  - createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - addUsbDeviceListener(UsbDeviceListener listener)
  - removeUsbDeviceListener(UsbDeviceListener listener)
  - getUsbDeviceDescriptor()
- UsbDeviceDescriptor
  - bMaxPacketSize()
- UsbControlIrp
  - bmRequestType()
  - bRequest()
  - wValue()
  - wIndex()
  - getData()
  - setData(byte[] data)
  - setData(byte[] data, int offset, int length)
  - getLength()
  - waitUntilComplete(long timeout)
  - isComplete()
- UsbDeviceListener
  - errorEventOccured(UsbDeviceErrorEvent event)
  - dataEventOccured(UsbDeviceDataEvent event)
- UsbDeviceDataEvent extends UsbDeviceEvent
  - UsbControlIrp getUsbControlIrp()
  - byte[] getData()

- UsbDeviceErrorEvent extends UsbDeviceEvent
  - UsbException getUsbException()

- UsbDeviceEvent extends EventObject
  - UsbDevice getUsbDevice()

- UsbConst
  - ENDPOINT_DIRECTION_OUT
  - ENDPOINT_DIRECTION_IN
  - ENDPOINT_TYPE_CONTROL

6.4 Control I/O Test

Goal :

This test verifies pipes can be opened on a control endpoint other than endpoint 0. The programmable device used in this test suite supports only one control pipe, the default control pipe. The target device for this test will be a device with at least one non default control pipe as specified in the hardware configurations. Because the target device is not programmable, minimal communication verification will be performed with the non default control pipe endpoint.

Technique :

A separate table defining the data transfers to be performed and expected results is provided for the Control I/O Test. The transfers will be repeated once for synchronous submits and once for asynchronous submits.

The list of classes and methods to be covered by this test is listed at the end of the test. Some of the methods and classes are identified explicitly in the test descriptions, but those that are not mentioned must also be called at least once in this test.

Open Pipes

Traverse topology looking for a device with a control endpoint other than endpoint zero. The first control endpoint found will be used.

For the configuration/interface of the control endpoint, use Standard Requests to set the configuration and alternate setting as defined.

- Claim, usbInterface.claim(), the interface. Verify usbInterface.isClaimed() before (false) and after (true) claiming interface.
- Get the pipe, usbEndpoint.getUsbPipe() for the control endpoint. Verify usbPipe.isActive().
- Add usbPipe.addUsbPipeListener() for the pipe
- Open, usbPipe.open() the pipe. Verify usbPipe.isOpen() before (false) and after (true) opening pipe.

Transfers

Transfer tables are defined in Appendix E.
First transfers will be performed with syncSumbit() repeat all transfers with asyncSubmit().

Perform the set of transfers 10 times.

For each transfer defined in the table

- Create the buffer(s) of the specified size and fill it with the appropriate data
    - For IRP

        - Use usbPipe.createUsbControlIrp(bmRequestType, bRequest, wValue, wIndex)
        - Set data with usbControlIrp.setData(byte[]).
    - For IRP list
        - Create a list of IRPs following steps for IRP above, except use a single byte[] for any data required in the list of IRPs.
        - Set data in each IRP with usbControlIrp.setData(byte[], int, int).
- Create an input buffer(s) of the specified size
    - For IRPs
        - Use usbPipe.createUsbControlIrp(bmRequestType, bRequest, wValue, wIndex)
        - Perform usbControlIrp.setAcceptShortPacket(false) according to table.
    - For IRP List
        - Create a list of IRPs following steps for IRP above except use a single byte[] for the list of IRPs.

Synchronous Submissions

- syncSubmit(IRP)
    - Blocks until complete.
    - Verify usbIrp.isComplete() is true.
    - Since the capabilities of the endpoint are unknown, UsbException will most likely be returned. However, a non-error response might be received. The test verification will allow error or data response. For data event received, actualLengths and data contents on OUTs can be verified; however, IN actual Lengths and data contents will not be verified.
    - usbControlIrp.isUsbException() may be true or false
        - If true, verify usbControlIrp.getUsbException() is not null.
        - If false, verify usbControlIrp.getUsbException() is null and usbControlIrp.getActualLength() returns the expected value for OUT, but no check on actualLength for IN.
    - Verify usbControlIrp.getData() that data is not altered on transfer OUT, but no verification of data for transfer IN.
    - Verify usbControlIrp.getOffset() and usbControlIrp.getLength() are unchanged after submit.
    - Verify usbPipeListener.dataEventOccurred or errorEventOccurred for each IRP. The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits. (Note: Since the device under test is user selected, the expected response cannot be known. Verfication is limited to an event being received.)
- syncSubmit(List of IRPs)
    - Blocks until all IRPs in list complete.
    - Perform all verifications listed under syncSubmit(IRP) for each IRP in list.

  For syncSubmit(), all exceptions will be thrown on the submit as well as retrievable by getUsbException() and isUsbException().

Asynchronous Submissions

- asyncSubmit(IRP)
  - See notes about verification under syncSubmit(IRP)
  - Does not block until complete.
  - For OUT, call usbIrp.waitUntilComplete(5000) after submit.
  - For IN, call usbIrp.waitUntilComplete(5000) after submit.
  - Perform all verifications listed under syncSubmit(IRP) for IRP.
- asyncSubmit(List of IRPs)
  - same as asyncSubmit(IRP) except for the following
  - Call usbIrp.waitUntilComplete(5000) for OUT IRPs, in order
  - Call usbIrp.waitUntilComplete(5000) for IN IRPs, in order
  - Perform all verifications listed under syncSubmit(IRP) for each IRP in list.
- abortAllSubmissions()
  - Call usbPipe.abortAllSubmissions for the pipe. There will be no pending operations to check, but no exceptions should be thrown. Submit and verify an IRP to confirm pipe is still operational.

For asyncSubmit(), exceptions which occur on the submit will be thrown, but exceptions which occur after the submit has returned will not be thrown. However, all exceptions on the submit whether they are thrown or not are retrievable by getUsbException() and isUsbException().

Close Pipes

- Close, usbPipe.close(), the pipes. Verify usbPipe.isOpen() before (true) and after (false) opening pipes.
- Remove usbPipe.removeUsbPipeListener() for each pipe
- Release, usbInterface.release(), the interfaces. Verify usbInterface.isClaimed() before (true) and after (false) releasing interface.

Error Conditions

- Action against a closed pipe
  - Verify usbPipe.isOpen() is false. Verify calling usbPipe.abortAllSubmissions() throws UsbNotOpenException.
  - Verify usbPipe.isOpen() is false. Create an IRP and verify calling usbPipe.syncSubmit(IRP) thows UsbNotOpenException.

Classes and Methods Tested :

- UsbPipe
  - open() throws UsbException, UsbNotActiveException, USBNotClaimedException
  - close() throws UsbException, UsbNotOpenException
  - isActive()
  - isOpen()
  - getUsbEndpoint()
  - syncSubmit/asyncSubmit(UsbIrp irp) throws UsbException, UsbNotOpenException, IllegalArgumentException
  - syncSubmit/asyncSubmit(List list) throws UsbException, UsbNotOpenException, IllegalArgumentException
  - createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - addUsbPipeListener(UsbPipeListener listener)

    - removeUsbPipeListener(UsbPipeListener listener)

- UsbControlIrp
    - bmRequestType()
    - bRequest()
    - wValue()
    - wIndex()
    - getData()
    - setData(byte[] data)
    - setData(byte[] data, int offset, int length)
    - getLength()
    - waitUntilComplete()
    - waitUntilComplete(long timeout)
    - isComplete()

- UsbEndpoint
    - getUsbEndpointDescriptor()

- UsbEndpointDescriptor
    - wMaxPacketLength()

- UsbPipeListener
    - errorEventOccured(UsbPipeErrorEvent event)
    - dataEventOccured(UsbPipeDataEvent event)

- UsbPipeDataEvent extends UsbPipeEvent
    - UsbIrp getUsbIrp()
    - byte[] getData()

- UsbPipeErrorEvent extends UsbPipeEvent
    - UsbException getUsbException()

- UsbPipeEvent extends EventObject
    - UsbPipe getUsbPipe()

- UsbConst
    - ENDPOINT_DIRECTION_OUT
    - ENDPOINT_DIRECTION_IN
    - ENDPOINT_TYPE_CONTROL

6.5 Bulk, Interrupt, and Isochronous I/O Tests

Goal :

This test verifies IN and OUT pipes can be opened and closed, and verifies that **bulk, interrupt, and isochronous transfer** operations work successfully, proper events are generated and proper exceptions are thrown in the operation.

Technique :

This same basic technique is used for Bulk, Interrupt, and Isochronous pipes.

The following tests will be repeated for the following pipes/submit types:
Bulk/synchronous
Bulk/asynchronous
Interrupt/synchronous
Interrupt/asynchronous
Isochronous/synchronous
Isochronous/asynchronous

A separate table defining the data transfers to be performed and expected results is provided for each type of pipe. The transfers for each pipe type will be repeated once for synchronous submits and once for asynchronous submits.

The list of classes and methods to be covered by this test is listed at the end of the test. Some of the methods and classes are identified explicitly in the test descriptions, but those that are not mentioned must also be called at least once in this test.

Open Pipes

Traverse topology looking for Cypress Development Board.

For each configuration/interface, use Standard Requests to set the configuration and alternate setting as defined.

- Claim, usbInterface.claim(), the interfaces. Verify usbInterface.isClaimed() before (false) and after (true) claiming interface.
- Get the pipe, usbEndpoint.getUsbPipe() for each endpoint defined in the transfer table. Verify usbPipe.isActive().
- Add usbPipe.addUsbPipeListener() for each pipe
- Open, usbPipe.open() the pipes. Verify usbPipe.isOpen() before (false) and after (true) opening pipes.

Transfers

Transfer tables are defined in Appendix B for BULK and INTERRUPT and Appendix C for Isochronous.
First transfers will be performed with syncSumbit() repeat all transfers with asyncSubmit().
Perform each transfer 10 times.

For each transfer defined in the table
    - Create the output buffer(s) of the specified size and fill it with the appropriate data
        - For byte, create byte array.
        - For IRP
            - Use usbPipe.createUsbIrp()
            - Set data with usbIrp.setData(byte[]).
        - For IRP list
            - Create a list of IRPs following steps for IRP above except use a single byte[] for the list of IRPs.
            - Set data in the byte[] with usbIrp.setData(byte[], int, int).
- Create an input buffer(s) of the specified size
    - For byte
        - create byte array
    - For IRPs
        - use usbPipe.CreateUsbIrp()
        - Verify usbIrp.getAcceptShortPacket() is true.
        - Perform usbIrp.setAcceptShortPacket(false) according to table.
    - For IRP List
        - Create a list of IRPs following steps for IRP above except use a single byte[] for the list of IRPs.
        - Set data in the byte[] with usbIrp.setData(byte[], int, int).

Synchronous Submissions

- syncSubmit(byte[])
    - Verify no exceptions on submit unless noted in transfer table.
    - Blocks until complete.
    - Verify return value matches number of bytes expected sent or received.
    - Verify data not altered after transfer OUT and matches expected transformed data IN.
    - Verify usbPipeListener.dataEventOccurred() or errorEventOccurred(). The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.
- syncSubmit(IRP)
    - Verify no exceptions on submit unless noted in transfer table.
    - Blocks until complete.
    - Verify usblIrp.isComplete() is true.
    - Verify usbIrp.isUsbException() according to transfer table.
        - If true, verify usbIrp.getUsbException() against expected exception.
        - If false, verify usbIrp.getUsbException() is null and usbIrp.getActualLength() returns the expected length actually sent or received.
    - Verify usbIrp.getData() that data is not altered on transfer OUT and matches expected transformed data IN.
    - Verify usbIrp.getOffset() and usbIrp.getLength() are unchanged after submit.
    - Verify usbPipeListener.dataEventOccurred or errorEventOccurred for each IRP. The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.

- syncSubmit(List of IRPs)
    - Verify no exceptions on submit unless noted in transfer table.
    - Blocks until all IRPs in list complete.
    - Perform all verifications listed under syncSubmit(IRP) for each IRP in list.

  For bulk pipes, since syncSubmits() block until they are complete, the OUT buffer must be submitted before the IN buffer; otherwise, the syncSubmit() for the IN would never get a data event and complete.

  For interrupt pipes, the IN buffers may need to be asyncSubmit() so that they will be available to receive data before the OUT is sent.

  For isochronous pipes, the IN buffers will need to be asyncSubmitted in an IRP list of many entries to ensure that an IRP is available when the OUT data is returned on the IN.

  For syncSubmit(), all exceptions will be thrown on the submit as well as retrievable by getUsbException() and isUsbException().

Asynchronous Submissions

- asyncSubmit(byte[])
    - Verify no exceptions on submit unless noted in transfer table.
    - Does not block until complete.
    - Return value is a UsbIrp.
    - For OUT, call usbIrp.waitUntilComplete(5000) after submit.
    - For IN, call usbIrp.waitUntilComplete(5000) after associated OUT submit is complete.
    - Perform all verifications listed under syncSubmit(IRP) for IRP.
    - Verify usbPipeListener.dataEventOccurred() or errorEventOccurred(). The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.
- asyncSubmit(IRP)
    - same as asyncSubmit(byte[])
- asyncSubmit(List of IRPs)
    - same as asyncSubmit(byte[]) except for the following
    - Call usbIrp.waitUntilComplete(5000) for OUT IRPs, in order
    - Call usbIrp.waitUntilComplete(5000) for IN IRPs, in order after OUTs complete.
    - Perform all verifications listed under syncSubmit(IRP) for each IRP.
- abortAllSubmissions()
    - asyncSubmit() an IRP list to an IN pipe.
    - syncSubmit() one OUT IRP that will cause the device to return data to fill in the first IRP in the IN list.
    - Call usbIrp.waitUntilComplete(5000) on the first IRP in the list.
    - Call usbPipe.abortAllSubmissions() for the IN pipe. All remaining IRPs should complete with error.
    - Verify usbIrp.isComplete() is true and usbIrp.isUsbException() is true for all remining IRPs in list. Verify usbIrp.getUsbException() returns UsbException.

- Verify the IN pipe is still in operation by issuing a submit on the IN pipe and submitting data on an OUT pipe causing a data event to be received on the IN pipe.
- Verify usbPipeListener.dataEventOccurred() or errorEventOccurred() for each IRP. The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.

For bulk, interrupt, and isochronous pipes, the IN buffers will need to be asyncSubmitted before the OUT buffers. Call waitUntilComplete(5000) as listed above.

For asyncSubmit(), exceptions which occur on the submit will be thrown, but exceptions which occur after the submit has returned will not be thrown. However, all exceptions on the submit whether they are thrown or not are retrievable by getUsbException() and isUsbException().

Close Pipes

- Close, usbPipe.close() the pipes. Verify usbPipe.isOpen() before (true) and after (false) opening pipes.
- Remove usbPipe.removeUsbPipeListener() for each pipe
- Release, usbInterface.release(), the interfaces. Verify usbInterface.isClaimed() before (true) and after (false) releaseing interface.

Error Conditions

- Null data buffer
  - Use usbPipe.createUsbIrp(), but do not call usbIrp.setData(). Submitting the IRP should throw a UsbException.
- Action against a closed pipe
  - Verify usbPipe.isOpen() is false. Verify calling usbPipe.abortAllSubmissions() throws UsbNotOpenException.
  - Verify usbPipe.isOpen() is false. Create an IRP and verify calling usbPipe.syncSubmit(IRP) thows UsbNotOpenException.
- Close a pipe with pending operations
  - Verify usbPipe.isOpen() is true. Create an IRP list and asyncSubmit() on an IN pipe.
  - Verify calling usbPipe.close() throws UsbException.
  - Call usbAbortAllSubmissions() and close pipe. Verify usbPipe.isOpen() is false.
- Open a pipe on an inactive interface
  - Verify usbPipe.open() called on a pipe in an inactive interface throws UsbNotActiveException.
- Open a pipe on an unclaimed interface
  - Verify usbPipe.open() called on an unclaimed interface throws UsbNotClaimedException
- Claim an already claimed interface (Not a pipe test, but convenient to do here. Does not need to be repeated in each pipe test.)
  - Verify usbInterface.claim() on an already claimed interface throws UsbClaimException.

- Claim an interface which is not active (Not a pipe test, but convenient to do here. Does not need to be repeated in each pipe test.)

Classes and Methods Tested :

- UsbPipe
  - open() throws UsbException, UsbNotActiveException, UsbNotClaimedException
  - close() throws UsbException, UsbNotOpenException
  - isActive()
  - isOpen()
  - getUsbEndpoint()
  - syncSubmit/asyncSubmit(byte[] data) throws UsbException, UsbNotOpenException
  - syncSubmit/asyncSubmit(UsbIrp irp) throws UsbException, UsbNotOpenException
  - syncSubmit/asyncSubmit(List list) throws UsbException, UsbNotOpenException
  - createUsbIrp()
  - addUsbPipeListener(UsbPipeListener listener)
  - removeUsbPipeListener(UsbPipeListener listener)
  - abortAllSubmissions() throws UsbNotOpenException

- UsbIrp
  - getData()
  - setData(byte[] data)
  - setData(byte[] data, int offset, int length)
  - getLength()
  - waitUntilComplete()
  - waitUntilComplete(long timeout)
  - isComplete()
  - getOffset();
  - getActualLength();
  - setOffset(int offset);
  - setLength(int length);
  - (skip—for implementation use only) setActualLength(int length);
  - isUsbException();
  - getUsbException();
  - setUsbException(UsbException usbException);
  - getAcceptShortPacket();
  - setAcceptShortPacket(boolean accept);
  - setComplete(boolean complete);
  - (skip—for implementation use only) complete();
  - waitUntilComplete();
  - waitUntilComplete(long timeout);

- UsbInterface
  - claim() throws UsbException,UsbNotActiveException
  - release() throws UsbException,UsbNotActiveException
  - isClaimed()
  - isActive()
  - getActiveSettingNumber() throws UsbNotActiveException
  - getActiveSetting() throws UsbNotActiveException

- UsbEndpoint
  - getUsbEndpointDescriptor()
  - getUsbPipe()

- UsbEndpointDescriptor
  - wMaxPacketSize()

- UsbPipeListener
  - errorEventOccured(UsbPipeErrorEvent event)
  - dataEventOccured(UsbPipeDataEvent event)

- UsbPipeDataEvent extends UsbPipeEvent
  - UsbIrp getUsbIrp()
  - byte[] getData()

- UsbPipeErrorEvent extends UsbPipeEvent
  - UsbException getUsbException()

- UsbPipeEvent extends EventObject

    - UsbPipeEvent(UsbPipe source)
    - UsbPipe getUsbPipe()

- UsbConst
    - ENDPOINT_DIRECTION_OUT
    - ENDPOINT_DIRECTION_IN
    - ENDPOINT_TYPE_BULK (for BULK)
    - ENDPOINT_TYPE_INTERRUPT (for INTERRUPT)
    - ENDPOINT_TYPE_ISOCHRONOUS (for ISOCHRONOUS)

- public class UsbShortPacketException extends UsbException
    - public UsbShortPacketException()
    - public UsbShortPacketException(String s)

- public class UsbStallException extends UsbException
    - public UsbStallException()
    - public UsbStallException(String s)

6.6 Request Test

Goal :

This test verifies the methods of the StandardRequest class by creating a Request object and verifying that the methods of StandardRequest send messages to the Default Control Pipe.

Technique :

The list of classes and methods to be covered by this test is listed at the end of the test. Some of the methods and classes are identified explicitly in the test descriptions, but those that are not mentioned must also be called at least once in this test.

Each request will be peformed twice, once using the methods where UsbDevice is not passed as a parameter (the UsbDevice that is passed in the constructor is used by default here), and once with the static methods that take UsbDevice in as a parameter.

Setup
Traverse the topology tree looking for the Cypress Development Board device.

Add usbDevice.addUsbDeviceListener(UsbDeviceListener).

Send Requests
Perform all Standard Device Requests as defined in Table 9-3 of the USB Specification 1.1. Ensure that each bmRequestType in this table is covered. Skip SET_ADDRESS to avoid causing the device to be unreachable.

For each request that returns a byte[ ], perform the request once with an array of minimum size and once with an array large enough to hold all of the data.

Verify setter methods where possible by following with a corresponding getter method.

Verify each request receives usbDeviceListener.dataEventOccurred. The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.

Error conditions
For each StandardRequest method, call the method with at least one invalid parameter.

Verify all methods throw proper exception if the value to create the Request is not valid.

Verify each request receives usbDeviceListener.errorEventOccurred. Verify any expected exceptions. The listener events occur after the submit is marked complete. Listener events occur in the same sequence as the associated submits.

Classes and Methods Tested :

- StandardRequest
  - StandardRequest(UsbDevice)
  - clearFeature(byte recipient, short featureSelector, short target)
  - getConfiguration()
  - getDescriptor(byte type, byte index, short langid, byte[] data)
  - getInterface(short interfaceNumber)
  - getStatus(byte recipient, short target)
  - setAddress(short deviceAddress)
  - setConfiguration(short configurationValue)
  - setDescriptor(byte type, byte index, short langid, byte[] data)
  - setFeature(byte recipient, short featureSelector, short target)
  - setInterface(short interfaceNumber, short alternateSetting)
  - synchFrame(short endpointAddress)
  - clearFeature(UsbDevice usbDevice, byte recipient, short featureSelector, short target)
  - getConfiguration(UsbDevice usbDevice)
  - getDescriptor(UsbDevice usbDevice, byte type, byte index, short langid, byte[] data)
  - getInterface(UsbDevice usbDevice, short interfaceNumber)
  - getStatus(UsbDevice usbDevice, byte recipient, short target)
  - setAddress(UsbDevice usbDevice, short deviceAddress)
  - setConfiguration(UsbDevice usbDevice, short configurationValue)
  - setDescriptor(UsbDevice usbDevice, byte type, byte index, short langid, byte[] data)
  - setFeature(UsbDevice usbDevice, byte recipient, short featureSelector, short target)
  - setInterface(UsbDevice usbDevice, short interfaceNumber, short alternateSetting)
  - synchFrame(UsbDevice usbDevice, short EndpointAddress)

- UsbConst
  - REQUESTTYPE_RECIPIENT_DEVICE
  - REQUESTTYPE_RECIPIENT_INTERFACE
  - REQUESTTYPE_RECIPIENT_ENDPOINT
  - REQUEST_GET_STATUS
  - REQUEST_CLEAR_FEATURE
  - REQUEST_SET_FEATURE
  - REQUEST_SET_ADDRESS
  - REQUEST_GET_DESCRIPTOR
  - REQUEST_SET_DESCRIPTOR
  - REQUEST_GET_CONFIGURATION
  - REQUEST_SET_CONFIGURATION
  - REQUEST_GET_INTERFACE
  - REQUEST_SET_INTERFACE
  - REQUEST_SYNCH_FRAME
  - DESCRIPTOR_TYPE_DEVICE
  - DESCRIPTOR_TYPE_CONFIGURATION
  - DESCRIPTOR_TYPE_STRING
  - DESCRIPTOR_TYPE_INTERFACE
  - DESCRIPTOR_TYPE_ENDPOINT
  -
  - DESCRIPTOR_MIN_LENGTH
  - DESCRIPTOR_MIN_LENGTH_DEVICE
  - DESCRIPTOR_MIN_LENGTH_CONFIGURATION
  - DESCRIPTOR_MIN_LENGTH_INTERFACE
  - DESCRIPTOR_MIN_LENGTH_ENDPOINT
  - DESCRIPTOR_MIN_LENGTH_STRING

6.7 IRP Test

Goal :

This test verifies the creation of IRP's and Control IRP's from the UsbDevice, UsbPipe, UsbDefaultIrp, and UsbDefaultControlIrp classes.

Technique :

Create IRPs in each of the specified ways. Perform verifications on each IRP. For control IRPs, perform additional control IRP verifications.

IRPs to be created

- Create an IRP using usbPipe.createUsbIrp().
- Create a Control IRP using usbPipe. createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
- Create a Control IRP using usbDevice. createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
- Create an IRP using each of the constructors from DefaultUsbIrp:
  - DefaultUsbIrp()
  - DefaultUsbIrp(byte[] data)
  - DefaultUsbIrp(byte[] data, int offset, int length, boolean shortPacket)
- Create an IRP using each of the constructors from DefaultUsbControlIrp:
  - DefaultUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - DefaultUsbControlIrp(byte[] data, int offset, int length, boolean shortPacket, byte bmRequestType, byte bRequest, short wValue, short wIndex)

IRP Verifications

- IRP verifications after IRP is created

  Verifications after UsbIrp or UsbControlIrp is created:

  - getLength()
    - If length is specified when IRP is created, getLength() returns specified length.
    - If length not specified, but byte[] is specified when IRP is created; getLength() returns the size of the byte[].
    - Otherwise, getLength() returns 0.
  - getOffset()
    - If offset is specified when IRP is created, getOffset() returns specified offset.
  - getData()

        - If byte[] is specified when IRP is created, getData() returns the byte[] which was specified.
        - Otherwise, getData() returns an empty byte[]. It will never return null.
    - getAcceptShortPacket()
        - If shortPacket is specified as false when IRP is created, getAcceptShortPacket() returns false.
        - If shortPacket is specified as true when IRP is created, getAcceptShortPacket() returns true.
        - Otherwise, getAcceptShortPacket() returns true.
    - getActualLength() returns 0
    - isUsbException() returns false
    - getUsbException() returns null
    - isComplete() returns false

  Additional verifications after UsbControlIrp is created:
    - bmRequestType()
        - bmRequestType() returns the bmRequestType value that is specified when IRP is created
    - bRequest()
        - bRequest() returns the bRequest value that is specified when IRP is created
    - wValue()
        - wValue() returns the wValue that is specified when IRP is created
    - wIndex()
        - wIndex() returns the wIndex that is specified when IRP is created

- IRP verifications after using setter methods
    - setData(byte[])
        - getData() returns byte[]
        - getLength() returns length of byte[]
        - getOffset() returns 0
    - setData(byte[] data, int offset, int length)
        - getData() returns byte[]
        - getLength() returns length specified
        - getOffset() returns offset specified
    - setLength(int length)
        - getLength() returns length specified
        - getData() returns unchanged byte[]
        - getOffset()returns unchanged offset
    - setOffset(int offset)
        - getOffset() returns offset specified
        - getData() returns unchanged byte[]
        - getLength() returns unchanged length
    - setAcceptShortPacket(boolean accept)
        - If accept parameter is false, getAcceptShortPacket() returns false
        - If accept parameter is true, getAcceptShortPacket() returns true
    - complete()
        - isComplete() returns true

- setComplete(boolean complete)
    - If complete parameter is false, isComplete() returns false
    - If complete parameter is true, isComplete() returns true
- setActualLength(int length)
    - getActualLength() returns specified length
- setUsbException(UsbException usbException)
    - getUsbException() returns usbException which was set
    - isUsbException() returns true

- Verify waitUntilComplete() and waitUntilComplete(long timeout)

    - Verify waitUntilComplete() ends when complete() is called.
        - Create another thread and pass a reference to IRP to other thread which calls waitUntilComplete() and terminates after waitUntilComplete() returns.
        - In main routine, call complete(), wait a second and verify thread has terminated.

    - Verify waitUntilComplete(long timeout) ends when complete() is called within the timeout period.
        - Create another thread and pass a reference to IRP to other thread which calls waitUntilComplete(100000)(100 seconds) and terminates after waitUntilComplete(100000) returns.
        - In main routine, call complete(), wait a second, and verify thread has terminated.

    - Verify waitUntilComplete(long timeout) ends with when the timeout expires.
        - Create another thread and pass a reference to IRP to other thread which calls waitUntilComplete(5000)(5 seconds) and terminates after waitUntilComplete(5000) returns. Subtract time prior to calling waitUntilComplete(5000) from time after waitUntilComplete(5000) returns. Time should be less than 6 seconds allowing for extra time for call and return.
        - In main routine, do not call complete(), verify that thread has terminated after 6 seconds. Verify isComplete() returns false.

Classes and Methods Tested :

- UsbDevice
    - createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)

- UsbPipe
    - createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
    - createUsbIrp()

- UsbIrp
    - getData()
    - getOffset()
    - getLength()
    - getActualLength()
    - setData(byte[] data)
    - setData(byte[] data, int offset, int length)
    - setOffset(int offset)

- setLength(int length)
- setActualLength(int length)
- isUsbException()
- getUsbException()
- setUsbException(UsbException usbException)
- getAcceptShortPacket()
- setAcceptShortPacket(boolean accept)
- isComplete()
- complete()
- waitUntilComplete()
- waitUntilComplete(long timeout)

- DefaultUsbIrp
  - DefaultUsbIrp()
  - DefaultUsbIrp(byte[] data)
  - DefaultUsbIrp(byte[] data, int offset, int length, boolean shortPacket)
  - getData()
  - getOffset()
  - getLength()
  - getActualLength()
  - isUsbException()
  - getUsbException()
  - setUsbException(UsbException exception)
  - getAcceptShortPacket()
  - isComplete()

- UsbControlIrp
  - bmRequestType()
  - bRequest()
  - wValue()
  - wIndex()
  - getData()
  - getOffset()
  - getLength()
  - getActualLength()
  - setData(byte[] data)
  - setData(byte[] data, int offset, int length)
  - setOffset(int offset)
  - setLength(int length)
  - setActualLength(int length)
  - isUsbException()
  - getUsbException()
  - setUsbException(UsbException usbException)
  - getAcceptShortPacket()
  - setAcceptShortPacket(boolean accept)
  - isComplete()
  - setComplete(boolean complete)
  - complete()
  - waitUntilComplete()
  - waitUntilComplete(long timeout)

- DefaultUsbControlIrp
  - DefaultUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - DefaultUsbControlIrp(byte[] data, int offset, int length, boolean shortPacket, byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - bmRequestType()
  - bRequest()
  - wValue()
  - wIndex()
  - wLength()
  - getData()
  - getOffset()
  - getLength()
  - getActualLength()

- isUsbException()
- getUsbException()
- setUsbException(UsbException exception)
- getAcceptShortPacket()
- isComplete()

6.8 Constants Test

Goal :

This test verifies constants defined in the implementation against that defined in the USB specification. Additionally, public constants in UsbHostManager are verified.

Technique :

Verifies constants defined in the implementation by comparing them against that defined in the USB specification.

Classes and Methods Tested :

- UsbConst
  - HUB_CLASSCODE = (byte)0x09;
  - CONFIGURATION_POWERED_MASK = (byte)0x60;
  - CONFIGURATION_SELF_POWERED = (byte)0x40;
  - CONFIGURATION_REMOTE_WAKEUP = (byte)0x20;
  - ENDPOINT_NUMBER_MASK = (byte)0x0f;
  - ENDPOINT_DIRECTION_MASK = (byte)0x80;
  - ENDPOINT_DIRECTION_OUT = (byte)0x00;
  - ENDPOINT_DIRECTION_IN = (byte)0x80;
  - ENDPOINT_TYPE_MASK = (byte)0x03;
  - ENDPOINT_TYPE_CONTROL = (byte)0x00;
  - ENDPOINT_TYPE_ISOCHRONOUS = (byte)0x01;
  - ENDPOINT_TYPE_BULK = (byte)0x02;
  - ENDPOINT_TYPE_INTERRUPT = (byte)0x03;
  - ENDPOINT_SYNCHRONIZATION_TYPE_MASK = (byte)0x0c;
  - ENDPOINT_SYNCHRONIZATION_TYPE_NONE = (byte)0x00;
  - ENDPOINT_SYNCHRONIZATION_TYPE_ASYNCHRONOUS = (byte)0x04;
  - ENDPOINT_SYNCHRONIZATION_TYPE_ADAPTIVE = (byte)0x08;
  - ENDPOINT_SYNCHRONIZATION_TYPE_SYNCHRONOUS = (byte)0x0c;
  - ENDPOINT_USAGE_TYPE_MASK = (byte)0x30;
  - ENDPOINT_USAGE_TYPE_DATA = (byte)0x00;
  - ENDPOINT_USAGE_TYPE_FEEDBACK = (byte)0x10;
  - ENDPOINT_USAGE_TYPE_IMPLICIT_FEEDBACK_DATA = (byte)0x20;
  - ENDPOINT_USAGE_TYPE_RESERVED = (byte)0x30;
  - REQUESTTYPE_DIRECTION_MASK = (byte)0x80;
  - REQUESTTYPE_DIRECTION_IN = (byte)0x80;
  - REQUESTTYPE_DIRECTION_OUT = (byte)0x00;
  - REQUESTTYPE_TYPE_MASK = (byte)0x60;
  - REQUESTTYPE_TYPE_STANDARD = (byte)0x00;
  - REQUESTTYPE_TYPE_CLASS = (byte)0x20;
  - REQUESTTYPE_TYPE_VENDOR = (byte)0x40;
  - REQUESTTYPE_TYPE_RESERVED = (byte)0x60;
  - REQUESTTYPE_RECIPIENT_MASK = (byte)0x1f;
  - REQUESTTYPE_RECIPIENT_DEVICE = (byte)0x00;
  - REQUESTTYPE_RECIPIENT_INTERFACE = (byte)0x01;
  - REQUESTTYPE_RECIPIENT_ENDPOINT = (byte)0x02;
  - REQUESTTYPE_RECIPIENT_OTHER = (byte)0x03;
  - REQUEST_GET_STATUS = (byte)0x00;
  - REQUEST_CLEAR_FEATURE = (byte)0x01;
  - REQUEST_SET_FEATURE = (byte)0x03;
  - REQUEST_SET_ADDRESS = (byte)0x05;
  - REQUEST_GET_DESCRIPTOR = (byte)0x06;
  - REQUEST_SET_DESCRIPTOR = (byte)0x07;
  - REQUEST_GET_CONFIGURATION = (byte)0x08;

- REQUEST_SET_CONFIGURATION = (byte)0x09;
- REQUEST_GET_INTERFACE = (byte)0x0a;
- REQUEST_SET_INTERFACE = (byte)0x0b;
- REQUEST_SYNCH_FRAME = (byte)0x0c;
- DESCRIPTOR_TYPE_DEVICE = (byte)0x01;
- DESCRIPTOR_TYPE_CONFIGURATION = (byte)0x02;
- DESCRIPTOR_TYPE_STRING = (byte)0x03;
- DESCRIPTOR_TYPE_INTERFACE = (byte)0x04;
- DESCRIPTOR_TYPE_ENDPOINT = (byte)0x05;
- DESCRIPTOR_MIN_LENGTH = (byte)0x02;
- DESCRIPTOR_MIN_LENGTH_DEVICE = (byte)0x12;
- DESCRIPTOR_MIN_LENGTH_CONFIGURATION = (byte)0x09;
- DESCRIPTOR_MIN_LENGTH_INTERFACE = (byte)0x09;
- DESCRIPTOR_MIN_LENGTH_ENDPOINT = (byte)0x07;
- DESCRIPTOR_MIN_LENGTH_STRING = (byte)0x02;
- FEATURE_SELECTOR_ENDPOINT_HALT = (byte)0x00;
- FEATURE_SELECTOR_DEVICE_REMOTE_WAKEUP = (byte)0x01;

- UsbHostManager
  - public static final String JAVAX_USB_PROPERTIES_FILE = "javax.usb.properties"
  - public static final String JAVAX_USB_USBSERVICES_PROPERTY = "javax.usb.services"

6.9 Hot Plug Test

Goal :

This test verifies that devices can be attached and detached at runtime (hot plugging) and checks that proper UsbServices events are generated.

Technique :

There are two possibilities here :

One is a test requiring user-interaction, where the user plugs in and removes devices, and the hands-off version, which makes use of Cypress board reset and binary upload to simulate device attach and detach function.

To make the TCK run completely hands-off, it is suggested that the Cypress board be used to test this API functionality by default.

Create an instance of UsbHostManager.

Get UsbServices from the UsbHostManager.

Traverse the topology tree to locate the Cypress Development Board.

Add a UsbServicesListener to receive events when the USB host has changes such as a new device is plugged in or unplugged.

Add a UsbDeviceListener to receive event when device is detached.

While transferring data through input and output endpoints (loop bulk data), unplug the device. (Vendor request to Cypress Board to renumerate will simulate unplug.)

Verify proper exceptions are thrown (without system failure) for all other I/O calls to a disconnected device.

Verify proper events are generated after disconnecting.

Cypress Board will be automatically reconfigured simulating a hotplug.

Verify proper events are generated after hot plugging.

Reconnect to bulk pipes and send and receive 10 transfers. Verify no errors on data transfers.

Classes and Methods Tested :

- UsbServices
  - addUsbServicesListener(UsbServicesListener)
  - removeUsbServicesListener(UsbServicesListener)

- UsbServicesListener
  - usbDeviceAttached(UsbServicesEvent)
  - usbDeviceDetached(UsbServicesEvent)

- UsbServicesEvent extends EventObject
  - UsbServicesEvent(UsbServices source, UsbDevice device)
  - UsbServices getUsbServices()
  - UsbDevice getUsbDevice()

- UsbDevice
  - addUsbDeviceListener(UsbDeviceListener)

- UsbDeviceListener
  - usbDeviceDetached(UsbDeviceEvent event)

- UsbDeviceErrorEvent extends UsbDeviceEvent
  - UsbException getUsbException()

- UsbDeviceEvent extends EventObject
  - UsbDevice getUsbDevice()

6.10 Interface Policy Test

Goal :

This test verifies that the non-default UsbInterfacePolicy is being used correctly.

Technique :

An implementation of the UsbInterfacePolicy interface will to be created. This implementation will allow the test to dynamically change the policy.

The UsbInterfacePolicy allows for an interface to potentially be force claimed. However, an implementation may or may not allow a force claiming of an interface. Since this test cannot know if the interface should have been successfully claimed, the test is limited to ensure that the implementation calls the usbInterfacePolicy.forceClaim() method when usbInterface.claim(UsbInterfacePolicy policy) is called. A tag will be needed that will flag whether the forceClaim method has been called.

Two keys will be created in the open and release methods. Key1 and key2. Key1 will set the policy to true, key2 will set the policy to false.

The interface policy test will use the above interface policy implementation to check the following:

Test the force claim policy with true and false, ensure the forceClaim method is called by verifying the forceClaim tag is set.

Test the open policy true and false with usbPipe.open(). If open policy is true, the test will expect a successful open; if false, the test will expect a policy denied exception to be thrown.

Test the open policy with key1 and key2 with usbPipe.open(key). If the key is set to key1, the pipe should open successfully; with key2, a policy denied exception should thrown.

Test the release policy true and false with usbInterface.release(). If the release policy is true, the interface should be released successfully; if false, the test will expect a policy denied exception to be thrown.

Test the release policy with key1 and key2, with usbInterface.release(key). It the key is set to key1, the interface should release successfully; with key2, a policy denied exception should be thrown.

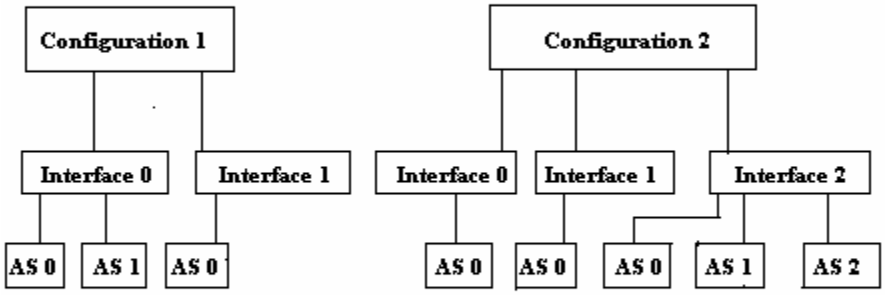
Test the release policy with the policy set to true but the interface not claimed, will expect a not claimed exception to be thrown

Test the release policy with key1 but with the interface not claimed, will expect a not claimed exception to be thrown

Classes and Methods Tested :

- UsbInterface
  - claim(UsbInterfacePolicy policy)
  - release (java.lang.Object key)
- UsbPipe
  - open (java.lang.Object key)
- UsbInterfacePolicy
  - forceClaim (UsbInterface usbInterface)
  - open (UsbPipe usbPipe, java.lang.Object key)
  - release (UsbInterface usbInterface, java.lang.Object key)

Appendix A – Programmable Device Configurations



Endpoints for Isochronous test are on Config 1, Interface0, Alternate Setting 0.
Endpoints for Bulk and Interrupt test are on Config 1, Interface 0, Alternate Setting 0.
All Endpoints in the tables are used for the Topology Test.
The tables also note which endpoints are available for each test.

The following tables show the descriptors.

Device Descriptors

Offset	Field	Description	Value
0	bLength	Length = 18 bytes	12H
1	bDescriptorType	Type = Device	01H
2	bcdUSB (L)	USB spec version 1.10 = 0110H	10H
3	bcdUSB (H)		01H
4	bDeviceClass	Vendor specific = FFH	FFH
5	bDeviceSubClass	Vendor specific = FFH	FFH
6	bDeviceProtocol	Vendor specific = FFH	FFH
7	bMaxPacketSize0	For EP0 = 64 bytes	40H
8	idVendor (L)	Cypress Semiconductor = 0547H	47H
9	idVendor (H)		05H
10	idProduct (L)	FF01H—not recognized by driver;	01H
11	idProduct (H)	Used for topology test	FFH
10*		1002H--bulk,int,iso, and dcp tests	02H
11*			10H
12	bcdDevice (L)	Device Release 1.0 = 0100H	00H
13	bcdDevice (H)		01H
14	iManufacturer	Manufacturer Index String = 1	01H
15	iProduct	Product Index String = 2	02H
16	iSerialNumber	Serial Number Index String = 3	03H
17	bNumConfigurations	Number of Configurations	02H

* The product id 1002H is the product id for the Cypress EZ-USB Sample Device

Configuration Descriptors

Offset	Field	Description	Configuration 1	Configuration 2
0	bLength	Length of Descriptor = 9 bytes	09H	09H
1	bDescriptorType	Type = Configuration	02H	02H
2	wTotalLength (L)	Total Length including Interface and Endpoint Descriptors (L)	3CH	7CH
3	wTotalLength (H)	Total Length (H)	01H	00H
4	bNumInterfaces	Number of Interfaces for this Config	02H	03H
5	bConfigurationValue	Configuration Value Used by Set_Configuration Request for this Config	01H	02H
6	iConfiguration	Index of String Descriptor for this Config	04H	05H
7	bmAttributes	Attributes – Bus Powered, Remote Wakeup	A0H	A0H
8	MaxPower	Maximum Power—100ma (ma/2)	32H	32H

Interface Descriptors

Length	Field – Description	Configuration 1			Configuration 2				
		Interface 0		Int 1	Int 0	Int 1	Interface 2		
		Alt Set 0	Alt Set 1	Alt Set 0	Alt Set 0	Alt Set 0	Alt Set 0	Alt Set 1	Alt Set 2
0	bLength -- Interface Descriptor= 9	09H	09H	09H	09H	09H	09H	09H	09H
1	bDescriptorType -- Interface = 4	04H	04H	04H	04H	04H	04H	04H	04H
2	InterfaceNumber – Zero based	00H	00H	01H	00H	01H	02H	02H	02H
3	AlternateSetting – Value	00H	01H	00H	00H	00H	00H	01H	02H
4	NumEndpoints	1EH	0AH	0H	0H	1H	2H	3H	4H
5	InterfaceClass – Vendor Specific = xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
6	InterfaceSubClass – Vendor Spec = xFF	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
7	InterfaceProtocol – Vendor Spec = xFF	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
8	InterfaceIndex – Index to String Desc	6	7	0	8	9	0AH	0BH	0CH

Endpoint Descriptors—("Test" column—all endpoints present in topology Image. Endpoints present in B-Bulk/Int or S-Iso/DCP tests as marked. Endpoint number key –TCIAE where T-- endpoint type; C—config #; I—interface #; A—alternate setting #; E—endpoint #

Diagram #	Test	Length	Type	Endpoint Address	Attributes	Max Packet Size	Interval
B1001I		7	5	0x81	2	64	0
B1001O		7	5	0x01	2	64	0
I1002I		7	5	0x82	3	8	0xFF
I1002O		7	5	0x02	3	8	0xFF
B1003I		7	5	0x83	2	64	0
B1003O		7	5	0x03	2	64	0
B1004I		7	5	0x84	2	8	0
B1004O		7	5	0x04	2	8	0
I1005I		7	5	0x85	3	16	0xFF
I1005O		7	5	0x05	3	16	0xFF
I1006I		7	5	0x86	3	32	0xFF
I1006O		7	5	0x06	3	32	0xFF
I1007I		7	5	0x87	3	64	0xFF
I1007O		7	5	0x07	3	64	0xFF
S1008I	S	7	5	0x88	1	16	1
S1008O	S	7	5	0x08	1	16	1
S1009I	S	7	5	0x89	1	16	1
S1009O	S	7	5	0x09	1	16	1
S10010I	S	7	5	0x8A	1	32	1
S10010O	S	7	5	0x0A	1	32	1
S10011I	S	7	5	0x8B	1	32	1
S10011O	S	7	5	0x0B	1	32	1
S10012I	S	7	5	0x8C	1	64	1
S10012O	S	7	5	0x0C	1	64	1
S10013I	S	7	5	0x8D	1	64	1
S10013O	S	7	5	0x0D	1	64	1
S10014I	S	7	5	0x8E	1	128	1
S10014O	S	7	5	0x0E	1	128	1
S10015I	S	7	5	0x8F	1	64	1
S10015O	S	7	5	0x0F	1	64	1
I1001I	B	7	5	0x81	3	64	0xFF
I1001O	B	7	5	0x01	3	64	0xFF
I1002I	B	7	5	0x82	3	8	0xFF
I1002O	B	7	5	0x02	3	8	0xFF
B1003I	B	7	5	0x83	2	64	0
B1003O	B	7	5	0x03	2	64	0
B1004I	B	7	5	0x84	2	8	0
B1004O	B	7	5	0x04	2	8	0
S1018I		7	5	0x88	1	1023	1
S1018O		7	5	0x08	1	1023	1
B2103I		7	5	0x83	2	64	0
I2205I		7	5	0x85	3	16	0xFF
I2205O		7	5	0x05	3	16	0xFF
S2218I		7	5	0x88	1	16	1
S2218O		7	5	0x08	1	16	1
S2219I		7	5	0x89	1	16	1
B2221I		7	5	0x81	2	64	0
B2223I		7	5	0x83	2	64	0
B2225I		7	5	0x85	2	16	0
B2227I		7	5	0x87	2	64	0

Data is passed between host and device by IO Request Packets (IRPs).
The device is programmed to support a max size of 200 bytes for bulk and interrupt endpoints and a max size of 1023 for isochronous endpoints.

Endpoint 0 has a max buffer size of 255 bytes.

The first byte of the IRP will be the desired manipulation of the IRP (see table below). The second byte of the IRP is the high order byte of the length of the IRP, and the third byte is the low order byte of the IRP.

Data Manipulation*

1. Pass through
2. Invert every bit starting with the second bit
3. Invert every other bit (invert odd bits assuming bit numbering begins at 0)

The device can buffer up to three IRPs for bulk and interrupt endpoints. IRPS for bulk and interrupt endpoints can span multiple packets.

Isochronous IRPs are not buffered in the device, but are immediately transformed and moved to the IN endpoint. The device supports isochronous IRPs of only a single packet.

For endpoint0 non-standard data the bmRequestType is for vendor specific command.
The other fields of the control irp are as follows:
bRequest request--transfer data 0xB0
wValueL -- out=0x00 data in= 0x80
wValueH
wIndexL --index in array to write to or read from
wIndexH
wLengthL --length of data to be written or read
wLengthH

String Descriptors -- descriptors are put in descriptor list in unicode

Index	String Bytes
0x01	Manufacturer
0x02	JSR80 TCK Device 1 (for topology test) JSR80 TCK Device 2 (for bulk/interrupt test) JSR80 TCK Device 3 (for isochronous test)
0x03	SN123456
0x04	Config 1
0x05	Config 2
0x06	C1 I0 AS0
0x07	C1 I0 AS1
0x08	C2 I0 AS0
0x09	C2 I1 AS0
0x0A	C2 I2 AS0
0x0B	C2 I2 AS1
0x0C	C2 I2 AS2
0x0D	0x30DE, 0x30F3, 0x30A5,0x0032,0x0020,0x0049,0x0032,0x0020,0x0041,0x0053,0x0032 Note: This string index is not referenced by any of the descriptors. This string is added to test Unicode characters that do not have 0x00 in the first byte. The characters starting with 0x30 come from the Katakana code chart.

Appendix B – BULK AND INTERRUPT TRANSFERS

BULK and INTERRUPT I/O TRANSFERS (sync/asyncSubmit(byte[]))								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(byte[])								
Buffers Multiples of maxPacketSize (1, 2, more than 2 packets)								
	(1) passthrough	(8)	(8)		8	8	8	
	(2) invert bits	(8)	(8)		16	16	16	
	(3) invert alt bits	(8)	(8)		24	24	24	
Buffers not Multiples of maxPacketSize (1, 2, more than 2 packets)								
	(1) passthrough	(64)	(64)		17	17	17	
	(2) invert bits	(64)	(64)		66	66	66	
	(3) invert alt bits	(64)	(64)		129	129	129	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3)Programmable device requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

BULK and INTERRUPT I/O TRANSFERS (sync/asyncSubmit(IRP))								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(IRP)								
Buffers Multiples of maxPacketSize (1, 2, more than 2 packets)								
	(2) invert bits	(8)	(8)		8	8	8	
	(3)invert alt bits	(8)	(8)		16	16	16	
	(1) passthrough	(8)	(8)		64	64	64	
Buffers not Multiples of maxPacketSize (1, 2, more than 2 packets)								
	(2) invert bits	(64)	(64)		25	25	25	
	(3)invert alt bits	(64)	(64)		72	72	72	
	(1)passthrough	(64)	(64)		130	130	130	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

BULK and INTERRUPT I/O TRANSFERS (sync/asyncSubmit(List of IRPs))								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size Offset,length	IN buffer data size Offset,length	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(List of IRPs)								
Buffers Multiples of maxPacketSize (1, 2, more than 2 packets) All buffers in IRPs in list come from a single byte[ ] (see Offset,Length)								
	(2) invert bits	(8)	(8)		0,8	0,8	8	
	(3)invert alt bits	(8)	(8)		8,16	8,16	16	
	(1) passthrough	(8)	(8)		24,24	24,24	24	
Buffers not Multiples of maxPacketSize (1, 2, more than 2 packets)								
	(2) invert bits	(64)	(64)		0,12	0,12	12	
	(3)invert alt bits	(64)	(64)		12,75	12,75	75	
	(1)passthrough	(64)	(64)		87,130	87,130	130	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte

3rd byte of OUT data is output buffer length low byte

BULK and INTERRUPT I/O TRANSFERS (error conditions)								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
syncSubmit(IRP) on OUT asyncSubmit(IRP) on IN								
Short packet error								
	(1)passthrough	(64)	(64)	NO	20	30		UsbShortPacketException on IN usbIrp.getUsbException().
								ClearFeature(ENDPOINT_HALT) for this endpoint via DefaultControlPipe
	(1)passthrough	(64)	(64)	YES	20	30	20	
	(1) passthrough	(64)	(64)	NO	25	25	25	Ok

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

Appendix C – ISOCHRONOUS TRANSFERS

ISOCHRONOUS I/O TRANSFERS (sync/asyncSubmit(byte[]))								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(byte[])								
Buffers of maxPacketSize (each transfer is 1 packet)								
	(2) invert bits	(32)	(32)		32	32	32	
	(3) invert alt bits	(64)	(64)		64	64	64	
	(1)passthrough	(128)	(128)		128	128	128	
Buffers less than maxPacketSize (each transfer is 1 packet)								
	(1) passthrough	(32)	(32)		30	30	30	
	(2) invert bits	(64)	(64)		25	25	25	
	(3) invert alt bits	(128)	(128)		127	127	127	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type

2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

ISOCHRONOUS I/O TRANSFERS (sync/asyncSubmit(IRP))								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(IRP)								
Buffers of maxPacketSize (each transfer is 1 packet)								
	(2) invert bits	(32)	(32)		32	32	32	
	(3)invert alt bits	(64)	(64)		64	64	64	
	(1) passthrough	(128)	(128)		128	128	128	
Buffers less than maxPacketSize (each transfer is 1 packet)								
	(2) invert bits	(32)	(32)		25	25	25	
	(3)invert alt bits	(64)	(64)		60	60	60	
	(1)passthrough	(128)	(128)		15	15	15	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte

3rd byte of OUT data is output buffer length low byte

ISOCHRONOUS I/O TRANSFERS (sync/asyncSubmit(List of IRPs))								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size Offset,length	IN buffer data size Offset,length	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(List of IRPs)								
Buffers of maxPacketSize (each transfer is 1 packet) All buffers in IRPs in list come from a single byte[ ] (see Offset,Length)								
	(2) invert bits	(32)	(32)		0,32	0,32	32	
	(3)invert alt bits	(32)	(32)		32,32	0,32	32	
	(1) passthrough	(32)	(32)		64,32	0,32	32	
Buffers less than maxPacketSize (each transfer is 1 packet)								
	(2) invert bits	(16)	(16)		0,12	0,15	12	
	(3)invert alt bits	(16)	(16)		12,15	0,15	15	
	(1)passthrough	(16)	(16)		27,10	0,15	10	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type

ISOCHRONOUS I/O TRANSFERS (error conditions)								
Device				Client				
C/I/AS(*1)	Transform(*2)	EPout(maxPacketSize)	EPin (maxPacketSize)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
syncSubmit(IRP) on OUT asyncSubmit(IRP) on IN								
Short packet error								
	See Bulk/Interrupt table for short packet tests							

2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

Appendix D – DEFAULT CONTROL PIPE TRANSFERS

Default Control Pipe I/O TRANSFERS (sync/asyncSubmit(IRP)) maxPacketSize for Endpoint 0 is 64 bytes								
Device				Client				
bmRequestType	bRequest 0xB0 – Transfer Data	wValue 0x00 – OUT 0x80--IN	wIndex (where to read or write to in device array for EP0)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size (wLength)	IN buffer data size (wLength)	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(IRP)								
Buffers Multiples of maxPacketSize (1, 2, more than 2 packets)								
REQUESTTYPE_TYPE_VENDOR + ENDPOINT_DIRECTION_IN(or_OUT) + REQUESTTYPE_RECIPIENT_DEVICE			0		64	64	64	
"			0		128	128	128	
"			0		192	192	192	
Buffers not Multiples of maxPacketSize (1, 2, more than 2 packets)								
"			0		50	50	50	
"			0		120	120	120	
"			0		160	160	160	

For DCP on EP0 all transfers are passthrough.
(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte

3rd byte of OUT data is output buffer length low byte

Default Control Pipe I/O TRANSFERS (sync/asyncSubmit(IRP)) maxPacketSize for Endpoint 0 is 64 bytes								
Device				Client				
bmRequestType	bRequest	wValue	wIndex	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT buffer data size (wLength)	IN buffer data size (wLength)	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(IRP)								
Selected Standard Requests with no data and with data								
REQUESTTYPE_TYPE_STANDARD + ENDPOINT_DIRECTION_OUT + REQUESTTYPE_RECIPIENT_ENDPOINT	(0x01) REQUEST_CLEAR_FEATURE	0x00 (no constant defined ENDPOINT_HALT)	Endpoint number		0			
REQUESTTYPE_TYPE_STANDARD + ENDPOINT_DIRECTION_IN + REQUESTTYPE_RECIPIENT_DEVICE	(0x06) REQUEST_GET_DESCRIPTOR	High (x03) DESCRIPTOR_TYPE_STRING Low(x01)- index which points to "Manufacturer" in unicode	0			64	26	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

Default Control Pipe I/O transfers (sync/asyncSubmit(List of IRPs)) maxPacketSize for Endpoint 0 is 64 bytes								
Device				Client				
bmRequestType	bRequest 0xB0 – Transfer Data	wValue 0x00—OUT 0x80--IN	wIndex (where to read or write to in device array for EP0)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT(*3) buffer data size (wLength) Offset,length	IN buffer data size (wLength) Offset,length	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(List of IRPs)								
Buffers Multiples of maxPacketSize (1, 2, more than 2 packets) All buffers in IRPs in list come from a single byte[ ] (see Offset,Length)								
REQUESTTYPE_TYPE_VENDOR + ENDPOINT_DIRECTION_IN(or_OUT) + REQUESTTYPE_RECIPIENT_DEVICE			0		0,64	0,64	64	
"			0		64,128	64,128	128	
"			0		192,192	192,192	192	
"			0		0,30	0,30	30	
"			0		30,60	30,60	60	
"			0		90,90	90,90	90	

(*1) C/I/AS—Configuration/Interface/Alternate Setting
(*2) see Programmable Device Configurations for key to transformations
(*3) Programmable Device Requirements--
1st byte of OUT data is transform type
2nd byte of OUT data is output buffer length high byte
3rd byte of OUT data is output buffer length low byte

Default Control Pipe I/O TRANSFERS (error conditions)								
Device				Client				
bmRequestType	bRequest 0xB0—Transfer Data	wValue 0x00—OUT 0x80--IN	wIndex (where to read or write to in device array EP0)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
syncSubmit(IRP) on OUT asyncSubmit(IRP) on IN								
Short packet error								
REQUESTTYPE_TYPE _VENDOR + ENDPOINT_ DIRECTION_IN(or_OUT) + REQUESTTYPE_RECIPIENT _DEVICE			0	NO	20	32		UsbShortPacketException on IN usbIrp.getUsbException(). Default control pipe will clear on its own (protocol stall).
"			0	YES	20	32	20	Ok
Invalid Device Command								
REQUESTTYPE_TYPE _STANDARD + ENDPOINT_ DIRECTION_OUT + REQUESTTYPE_RECIPIENT _DEVICE	20	0	0					UsbStallException on OUT. Default control pipe will clear on its own (protocol stall)

Appendix E -- Control I/O Transfers

Control Pipe I/O TRANSFERS (sync/asyncSubmit(IRP))								
Device				Client				
bmRequestType	bRequest (transform)	wValue (length)	wIndex (0)	acceptShortPackets yes-set short packets ok no-set short packets not ok empty-don't set short packets; default used	OUT buffer data size	IN buffer data size	Expected Actual Length of data for IN buffer if no error	Expected Error and Recovery Actions
sync/asyncSubmit(IRP)								
It is not expected that the device will recognize any of these transfers; however, expected results are uncertain. Test will pass for error or data events received.								
REQUESTTYPE_TYPE_STANDARD + ENDPOINT_DIRECTION_OUT + REQUESTTYPE_RECIPIENT_ENDPOINT	(0x01) REQUEST_CLEAR_FEATURE	0x00 (no constant defined ENDPOINT_HALT)	Endpoint number		0			
REQUESTTYPE_TYPE_STANDARD + ENDPOINT_DIRECTION_IN + REQUESTTYPE_RECIPIENT_DEVICE	(0x06) REQUEST_GET_DESCRIPTOR	High (x03) DESCRIPTOR_TYPE_STRING Low(x01)- index which points to "Manufacturer 1"	0			64	14	
sync/asyncSubmit (IRP List)								
Resubmit transfers as in sync/asyncSubmit(IRP), but as LIST.								

Appendix F – Javax.usb API

(*)-classes and methods marked with * are not covered in the Test Suite

package javax.usb

- *class UsbBabbleException extends UsbException
  - UsbBabbleException()
  - UsbBabbleException(String s)

- *class UsbBitStuffException extends UsbException
  - UsbBitStuffException()
  - UsbBitStuffException(String s

- *class UsbClaimException extends UsbException
  - UsbClaimException()
  - UsbClaimException(String s)

- interface UsbConfiguration
  - boolean isActive();
  - List getUsbInterfaces();
  - UsbInterface getUsbInterface(byte number);
  - boolean containsUsbInterface(byte number);
  - UsbDevice getUsbDevice();
  - UsbConfigurationDescriptor getUsbConfigurationDescriptor();

- String getConfigurationString() throws UsbException,UnsupportedEncodingException;

- interface UsbConfigurationDescriptor extends UsbDescriptor
  - short wTotalLength();
  - byte bNumInterfaces();
  - byte bConfigurationValue();
  - byte iConfiguration();
  - byte bmAttributes();
  - byte bMaxPower();

- interface UsbConst
  - static final byte HUB_CLASSCODE = (byte)0x09;
  - static final Object DEVICE_SPEED_UNKNOWN = new Object();
  - static final Object DEVICE_SPEED_LOW = new Object();
  - static final Object DEVICE_SPEED_FULL = new Object();
  - static final byte CONFIGURATION_POWERED_MASK = (byte)0x60;
  - static final byte CONFIGURATION_SELF_POWERED = (byte)0x40;
  - static final byte CONFIGURATION_REMOTE_WAKEUP = (byte)0x20;
  - static final byte ENDPOINT_NUMBER_MASK = (byte)0x0f;
  - static final byte ENDPOINT_DIRECTION_MASK = (byte)0x80;
  - static final byte ENDPOINT_DIRECTION_OUT = (byte)0x00;
  - static final byte ENDPOINT_DIRECTION_IN = (byte)0x80;
  - static final byte ENDPOINT_TYPE_MASK = (byte)0x03;
  - static final byte ENDPOINT_TYPE_CONTROL = (byte)0x00;
  - static final byte ENDPOINT_TYPE_ISOCHRONOUS = (byte)0x01;
  - static final byte ENDPOINT_TYPE_BULK = (byte)0x02;
  - static final byte ENDPOINT_TYPE_INTERRUPT = (byte)0x03;
  - static final byte ENDPOINT_SYNCHRONIZATION_TYPE_MASK = (byte)0x0c;

- static final byte ENDPOINT_SYNCHRONIZATION_TYPE_NONE = (byte)0x00;
- static final byte ENDPOINT_SYNCHRONIZATION_TYPE_ASYNCHRONOUS = (byte)0x04;
- static final byte ENDPOINT_SYNCHRONIZATION_TYPE_ADAPTIVE = (byte)0x08;
- static final byte ENDPOINT_SYNCHRONIZATION_TYPE_SYNCHRONOUS = (byte)0x0c;
- static final byte ENDPOINT_USAGE_TYPE_MASK = (byte)0x30;
- static final byte ENDPOINT_USAGE_TYPE_DATA = (byte)0x00;
- static final byte ENDPOINT_USAGE_TYPE_FEEDBACK = (byte)0x10;
- static final byte ENDPOINT_USAGE_TYPE_IMPLICIT_FEEDBACK_DATA = (byte)0x20;
- static final byte ENDPOINT_USAGE_TYPE_RESERVED = (byte)0x30;
- static final byte REQUESTTYPE_DIRECTION_MASK = (byte)0x80;
- static final byte REQUESTTYPE_DIRECTION_IN = (byte)0x80;
- static final byte REQUESTTYPE_DIRECTION_OUT = (byte)0x00;
- static final byte REQUESTTYPE_TYPE_MASK = (byte)0x60;
- static final byte REQUESTTYPE_TYPE_STANDARD = (byte)0x00;
- static final byte REQUESTTYPE_TYPE_CLASS = (byte)0x20;
- static final byte REQUESTTYPE_TYPE_VENDOR = (byte)0x40;
- static final byte REQUESTTYPE_TYPE_RESERVED = (byte)0x60;
- static final byte REQUESTTYPE_RECIPIENT_MASK = (byte)0x1f;
- static final byte REQUESTTYPE_RECIPIENT_DEVICE = (byte)0x00;
- static final byte REQUESTTYPE_RECIPIENT_INTERFACE = (byte)0x01;
- static final byte REQUESTTYPE_RECIPIENT_ENDPOINT = (byte)0x02;
- static final byte REQUESTTYPE_RECIPIENT_OTHER = (byte)0x03;
- static final byte REQUEST_GET_STATUS = (byte)0x00;
- static final byte REQUEST_CLEAR_FEATURE = (byte)0x01;
- static final byte REQUEST_SET_FEATURE = (byte)0x03;
- static final byte REQUEST_SET_ADDRESS = (byte)0x05;
- static final byte REQUEST_GET_DESCRIPTOR = (byte)0x06;
- static final byte REQUEST_SET_DESCRIPTOR = (byte)0x07;
- static final byte REQUEST_GET_CONFIGURATION = (byte)0x08;
- static final byte REQUEST_SET_CONFIGURATION = (byte)0x09;

    - static final byte REQUEST_GET_INTERFACE = (byte)0x0a;
    - static final byte REQUEST_SET_INTERFACE = (byte)0x0b;
    - static final byte REQUEST_SYNCH_FRAME = (byte)0x0c;
    - static final byte FEATURE_SELECTOR_DEVICE_REMOTE_WAKEUP = (byte)0x01;
    - static final byte FEATURE_SELECTOR_ENDPOINT_HALT = (byte)0x00;
    - static final byte DESCRIPTOR_TYPE_DEVICE = (byte)0x01;
    - static final byte DESCRIPTOR_TYPE_CONFIGURATION = (byte)0x02;
    - static final byte DESCRIPTOR_TYPE_STRING = (byte)0x03;
    - static final byte DESCRIPTOR_TYPE_INTERFACE = (byte)0x04;
    - static final byte DESCRIPTOR_TYPE_ENDPOINT = (byte)0x05;
    - static final byte DESCRIPTOR_MIN_LENGTH = (byte)0x02;
    - static final byte DESCRIPTOR_MIN_LENGTH_DEVICE = (byte)0x12;
    - static final byte DESCRIPTOR_MIN_LENGTH_CONFIGURATION = (byte)0x09;
    - static final byte DESCRIPTOR_MIN_LENGTH_INTERFACE = (byte)0x09;
    - static final byte DESCRIPTOR_MIN_LENGTH_ENDPOINT = (byte)0x07;
    - static final byte DESCRIPTOR_MIN_LENGTH_STRING = (byte)0x02;

- interface UsbControlIrp extends UsbIrp
    - byte bmRequestType();
    - byte bRequest();
    - short wValue();
    - short wIndex();

- *class UsbCRC extends UsbException
    - UsbCRCException()
    - UsbCRCException(String s)

- interface UsbDescriptor
  - byte bLength();
  - byte bDescriptorType();

- interface UsbDevice
  - UsbPort getParentUsbPort();
  - boolean isUsbHub();
  - String getManufacturerString() throws UsbException,UnsupportedEncodingException;
  - String getSerialNumberString() throws UsbException,UnsupportedEncodingException;
  - String getProductString() throws UsbException,UnsupportedEncodingException;
  - Object getSpeed();
  - List getUsbConfigurations();
  - UsbConfiguration getUsbConfiguration(byte number);
  - boolean containsUsbConfiguration(byte number);
  - byte getActiveUsbConfigurationNumber();
  - UsbConfiguration getActiveUsbConfiguration();
  - boolean isConfigured();
  - UsbDeviceDescriptor getUsbDeviceDescriptor();
  - UsbStringDescriptor getUsbStringDescriptor(byte index) throws UsbException;
  - String getString(byte index) throws UsbException,UnsupportedEncodingException;
  - void syncSubmit(UsbControlIrp irp) throws UsbException,IllegalArgumentException;
  - void asyncSubmit(UsbControlIrp irp) throws UsbException,IllegalArgumentException;
  - void syncSubmit(List list) throws UsbException,IllegalArgumentException;
  - void asyncSubmit(List list) throws UsbException,IllegalArgumentException;
  - UsbControlIrp createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex);
  - void addUsbDeviceListener(UsbDeviceListener listener);
  - void removeUsbDeviceListener(UsbDeviceListener listener);

- interface UsbDeviceDescriptor extends UsbDescriptor
    - short bcdUSB();
    - byte bDeviceClass();
    - byte bDeviceSubClass();
    - byte bDeviceProtocol();
    - byte bMaxPacketSize0();
    - short idVendor();
    - short idProduct();
    - short bcdDevice();
    - byte iManufacturer();
    - byte iProduct();
    - byte iSerialNumber();
    - byte bNumConfigurations();

- interface UsbEndpoint
    - UsbInterface getUsbInterface();
    - UsbEndpointDescriptor getUsbEndpointDescriptor();
    - byte getDirection();
    - byte getType();
    - UsbPipe getUsbPipe();

- interface UsbEndpointDescriptor extends UsbDescriptor
    - byte bEndpointAddress();
    - byte bmAttributes();
    - short wMaxPacketSize();
    - byte bInterval();

- class UsbException extends Exception
  - UsbException()
  - UsbException(String s)

- final class UsbHostManager
  - static synchronized UsbServices getUsbServices() throws UsbException,SecurityException
  - static synchronized Properties getProperties() throws UsbException,SecurityException
  - static final String JAVAX_USB_PROPERTIES_FILE = "javax.usb.properties";
  - static final String JAVAX_USB_USBSERVICES_PROPERTY = "javax.usb.services";

- interface UsbHub extends UsbDevice
  - byte getNumberOfPorts();
  - List getUsbPorts();
  - UsbPort getUsbPort(byte number);
  - List getAttachedUsbDevices();
  - boolean isRootUsbHub();

- interface UsbInterface
  - void claim() throws UsbClaimException,UsbException,UsbNotActiveException;
  - *void claim(UsbInterface policy) throws UsbClaimException,UsbException,UsbNotActiveException;
  - void release() throws UsbPolicyDenied,UsbClaimException,UsbException,UsbNotActiveException;
  - *void release(Object key) throws UsbPolicyDenied,UsbClaimException,UsbException,UsbNotActiveException;
  - boolean isClaimed();
  - boolean isActive();
  - int getNumSettings();
  - byte getActiveSettingNumber() throws UsbNotActiveException;
  - UsbInterface getActiveSetting() throws UsbNotActiveException;

    - UsbInterface getSetting(byte number);
    - boolean containsSetting(byte number);
    - List getSettings();
    - List getUsbEndpoints();
    - UsbEndpoint getUsbEndpoint(byte address);
    - boolean containsUsbEndpoint(byte address);
    - UsbConfiguration getUsbConfiguration();
    - UsbInterfaceDescriptor getUsbInterfaceDescriptor();
    - String getInterfaceString() throws UsbException,UnsupportedEncodingException;

- interface UsbInterfaceDescriptor extends UsbDescriptor
    - byte bInterfaceNumber();
    - byte bAlternateSetting();
    - byte bNumEndpoints();
    - byte bInterfaceClass();
    - byte bInterfaceSubClass();
    - byte bInterfaceProtocol();
    - byte iInterface();

- *interface UsbInterfacePolicy
    - *boolean release(UsbInterface usbInterface, Object key);
    - *boolean open(UsbPipe usbPipe, Object key);
    - *boolean forceClaim(UsbInterface usbInterface);

- interface UsbIrp
    - byte[] getData();
    - int getOffset();

    - int getLength();
    - int getActualLength();
    - void setData(byte[] data);
    - void setData(byte[] data, int offset, int length);
    - void setOffset(int offset);
    - void setLength(int length);
    - void setActualLength(int length);
    - boolean isUsbException();
    - UsbException getUsbException();
    - void setUsbException(UsbException usbException);
    - boolean getAcceptShortPacket();
    - void setAcceptShortPacket(boolean accept);
    - boolean isComplete();
    - void setComplete(boolean complete);
    - void complete();
    - void waitUntilComplete();
    - void waitUntilComplete(long timeout);

- *class UsbNativeClaimException extends UsbClaimException
    - UsbNativeClaimException()
    - UsbNativeClaimException(String s)

- class UsbNotActiveException extends RuntimeException
    - UsbNotActiveException()
    - UsbNotActiveException(String s)

- class UsbNotClaimedException extends RuntimeException

    - UsbNotClaimedException()
    - UsbNotClaimedException(String s)

- class UsbNotOpenException extends RuntimeException
    - UsbNotOpenException()
    - UsbNotOpenException(String s)

- *class UsbPIDException extends UsbException
    - UsbPIDException()
    - UsbPIDException(String s)

- interface UsbPipe
    - void open() throws UsbDenied,UsbException,UsbNotActiveException,UsbNotClaimedException;
    - *void open(Object key) throws UsbPolicyDenied, UsbException, UsbNotActiveException, UsbNotClaimedException;
    - void close() throws UsbException,UsbNotOpenException;
    - boolean isActive();
    - boolean isOpen();
    - UsbEndpoint getUsbEndpoint();
    - int syncSubmit(byte[] data) throws UsbException,UsbNotOpenException,IllegalArgumentException;
    - UsbIrp asyncSubmit(byte[] data) throws UsbException,UsbNotOpenException,IllegalArgumentException;
    - void syncSubmit(UsbIrp irp) throws UsbException,UsbNotOpenException,IllegalArgumentException;
    - void asyncSubmit(UsbIrp irp) throws UsbException,UsbNotOpenException,IllegalArgumentException;
    - void syncSubmit(List list) throws UsbException,UsbNotOpenException,IllegalArgumentException;
    - void asyncSubmit(List list) throws UsbException,UsbNotOpenException,IllegalArgumentException;
    - void abortAllSubmissions() throws UsbNotOpenException;
    - UsbIrp createUsbIrp();

    - UsbControlIrp createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex);
    - void addUsbPipeListener(UsbPipeListener listener);
    - void removeUsbPipeListener(UsbPipeListener listener);

- *class UsbPlatformException extends UsbException
    - UsbPlatformException()
    - UsbPlatformException(String s)
    - UsbPlatformException(int e)
    - UsbPlatformException(Exception pE)
    - UsbPlatformException(String s, int e)
    - UsbPlatformException(String s, Exception pE)
    - UsbPlatformException(int e, Exception pE)
    - UsbPlatformException(String s, int e, Exception pE)
    - Exception getPlatformException()
    - int getErrorCode()

- *class UsbDenied extends UsbException
    - UsbPolicyDenied()
    - UsbPolicyDenied(String s)

- interface UsbPort
    - byte getPortNumber();
    - UsbHub getUsbHub();
    - UsbDevice getUsbDevice();
    - boolean isUsbDeviceAttached();

- interface UsbServices
  - UsbHub getRootUsbHub() throws UsbException,SecurityException;
  - void addUsbServicesListener(UsbServicesListener listener);
  - void removeUsbServicesListener(UsbServicesListener listener);
  - String getApiVersion();
  - String getImpVersion();
  - String getImpDescription();

- class UsbShortPacketException extends UsbException
  - UsbShortPacketException()
  - UsbShortPacketException(String s)

- class UsbStallException extends UsbException
  - UsbStallException()
  - UsbStallException(String s)

- interface UsbStringDescriptor extends UsbDescriptor
  - byte[] bString();
  - String getString() throws UnsupportedEncodingException;

- class DefaultUsbControlIrp extends DefaultUsbIrp implements UsbControlIrp
  - DefaultUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - DefaultUsbControlIrp(byte[] data, int offset, int length, boolean shortPacket, byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - byte bmRequestType()
  - byte bRequest()

    - short wValue()
    - short wIndex()
    - short wLength()

- class DefaultUsbIrp implements UsbIrp
    - DefaultUsbIrp()
    - DefaultUsbIrp(byte[] data)
    - DefaultUsbIrp(byte[] data, int offset, int length, boolean shortPacket)
    - byte[] getData()
    - int getOffset()
    - int getLength()
    - int getActualLength()
    - void setData(byte[] d, int o, int l) throws IllegalArgumentException
    - void setData(byte[] d) throws IllegalArgumentException
    - void setOffset(int o) throws IllegalArgumentException
    - void setLength(int l) throws IllegalArgumentException
    - void setActualLength(int l) throws IllegalArgumentException
    - boolean isUsbException()
    - UsbException getUsbException()
    - void setUsbException(UsbException exception)
    - boolean getAcceptShortPacket()
    - void setAcceptShortPacket(boolean accept)
    - boolean isComplete()
    - void setComplete(boolean b)
    - void complete()
    - void waitUntilComplete()
    - void waitUntilComplete(long timeout)

- class StandardRequest
  - StandardRequest(UsbDevice usbDevice)
  - void clearFeature(byte recipient, short featureSelector, short target) throws UsbException,IllegalArgumentException
  - byte getConfiguration() throws UsbException
  - int getDescriptor(byte type, byte index, short langid, byte[] data) throws UsbException
  - byte getInterface(short interfaceNumber) throws UsbException
  - short getStatus(byte recipient, short target) throws UsbException,IllegalArgumentException
  - void setAddress(short deviceAddress) throws UsbException
  - void setConfiguration(short configurationValue) throws UsbException
  - int setDescriptor(byte type, byte index, short langid, byte[] data) throws UsbException
  - void setFeature(byte recipient, short featureSelector, short target) throws UsbException,IllegalArgumentException
  - void setInterface(short interfaceNumber, short alternateSetting) throws UsbException
  - short synchFrame(short endpointAddress) throws UsbException
  - static void clearFeature(UsbDevice usbDevice, byte recipient, short featureSelector, short target) throws UsbException,IllegalArgumentException
  - static byte getConfiguration(UsbDevice usbDevice) throws UsbException
  - static int getDescriptor(UsbDevice usbDevice, byte type, byte index, short langid, byte[] data) throws UsbException
  - static byte getInterface(UsbDevice usbDevice, short interfaceNumber) throws UsbException
  - static short getStatus(UsbDevice usbDevice, byte recipient, short target) throws UsbException,IllegalArgumentException
  - static void setAddress(UsbDevice usbDevice, short deviceAddress) throws UsbException
  - static void setConfiguration(UsbDevice usbDevice, short configurationValue) throws UsbException
  - static int setDescriptor(UsbDevice usbDevice, byte type, byte index, short langid, byte[] data) throws UsbException
  - static void setFeature(UsbDevice usbDevice, byte recipient, short featureSelector, short target) throws UsbException,IllegalArgumentException
  - static void setInterface(UsbDevice usbDevice, short interfaceNumber, short alternateSetting) throws UsbException
  - static short synchFrame(UsbDevice usbDevice, short endpointAddress) throws UsbException

package javax.usb.event

- class UsbDeviceDataEvent extends UsbDeviceEvent
  - UsbDeviceDataEvent(UsbDevice source, UsbControlIrp irp, byte[] data, int offset, int length) throws IllegalArgumentException
  - UsbControlIrp getUsbControlIrp()
  - byte[] getData()

- class UsbDeviceErrorEvent extends UsbDeviceEvent
  - UsbDeviceErrorEvent(UsbDevice source, UsbException uE)
  - UsbException getUsbException()

- class UsbDeviceEvent extends EventObject
  - UsbDeviceEvent(UsbDevice source)
  - UsbDevice getUsbDevice()

- interface UsbDeviceListener extends EventListener
  - void usbDeviceDetached(UsbDeviceEvent event);
  - void errorEventOccurred(UsbDeviceErrorEvent event);
  - void dataEventOccurred(UsbDeviceDataEvent event);

- class UsbPipeDataEvent extends UsbPipeEvent
  - UsbPipeDataEvent(UsbPipe source, UsbIrp irp, byte[] data, int offset, int length) throws IllegalArgumentException
  - UsbIrp getUsbIrp()

    - byte[] getData()

- class UsbPipeErrorEvent extends UsbPipeEvent
    - UsbPipeErrorEvent(UsbPipe source, UsbException uE)
    - UsbException getUsbException()

- class UsbPipeEvent extends EventObject
    - UsbPipeEvent(UsbPipe source)
    - UsbPipe getUsbPipe()

- interface UsbPipeListener extends EventListener
    - void errorEventOccurred(UsbPipeErrorEvent event);
    - void dataEventOccurred(UsbPipeDataEvent event);

- class UsbServicesEvent extends EventObject
    - UsbServicesEvent(UsbServices source, UsbDevice device)
    - UsbServices getUsbServices()
    - UsbDevice getUsbDevice()

- interface UsbServicesListener extends EventListener
    - void usbDeviceAttached(UsbServicesEvent event);
    - void usbDeviceDetached(UsbServicesEvent event);

package javax.usb.util

- *class UsbUtil
  - static short unsignedShort(byte b)
  - static int unsignedInt(byte b)
  - static int unsignedInt(short s)
  - static long unsignedLong(byte b)
  - static long unsignedLong(short s)
  - static long unsignedLong(int i)
  - static short toShort(byte msb, byte lsb)
  - static int toInt(byte byte3, byte byte2, byte byte1, byte byte0)
  - static long toLong(byte byte7, byte byte6, byte byte5, byte byte4, byte byte3, byte byte2, byte byte1, byte byte0)
  - static int toInt(short mss, short lss)
  - static long toLong(short short3, short short2, short short1, short short0)
  - static long toLong(int msi, int lsi)
  - static String toHexString(byte b)
  - static String toHexString(short s)
  - static String toHexString(int i)
  - static String toHexString(long l)
  - static String toHexString(long l, char c, int min, int max)
  - static String toHexString(String delimiter, byte[] array, int length)
  - static String toHexString(String delimiter, short[] array, int length)
  - static String toHexString(String delimiter, int[] array, int length)
  - static String toHexString(String delimiter, long[] array, int length)
  - static String toHexString(String delimiter, byte[] array)
  - static String toHexString(String delimiter, short[] array)
  - static String toHexString(String delimiter, int[] array)
  - static String toHexString(String delimiter, long[] array)
  - static String getSpeedString(Object object)
  - static UsbDevice synchronizedUsbDevice(UsbDevice usbDevice)

- static UsbPipe synchronizedUsbPipe(UsbPipe usbPipe)

- static class SynchronizedUsbDevice implements UsbDevice
  - SynchronizedUsbDevice(UsbDevice usbDevice)
  - UsbPort getParentUsbPort()
  - boolean isUsbHub()
  - String getManufacturerString() throws UsbException,UnsupportedEncodingException
  - String getSerialNumberString() throws UsbException,UnsupportedEncodingException
  - String getProductString() throws UsbException,UnsupportedEncodingException
  - Object getSpeed()
  - List getUsbConfigurations()
  - UsbConfiguration getUsbConfiguration(byte number)
  - boolean containsUsbConfiguration(byte number)
  - byte getActiveUsbConfigurationNumber()
  - UsbConfiguration getActiveUsbConfiguration()
  - boolean isConfigured()
  - UsbDeviceDescriptor getUsbDeviceDescriptor()
  - UsbStringDescriptor getUsbStringDescriptor(byte index) throws UsbException
  - String getString(byte index) throws UsbException,UnsupportedEncodingException
  - void syncSubmit(UsbControlIrp irp) throws UsbException
  - void asyncSubmit(UsbControlIrp irp) throws UsbException
  - void syncSubmit(List list) throws UsbException
  - void asyncSubmit(List list) throws UsbException
  - UsbControlIrp createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
  - void addUsbDeviceListener(UsbDeviceListener listener)
  - void removeUsbDeviceListener(UsbDeviceListener listener)
  - UsbDevice usbDevice = null;

- static class SynchronizedUsbPipe implements UsbPipe
  - SynchronizedUsbPipe(UsbPipe usbPipe)

    - void open() throws UsbPolicyDenied,UsbException,UsbNotActiveException,UsbNotClaimedException
    - void open(Object o) throws UsbPolicyDenied,UsbException,UsbNotActiveException,UsbNotClaimedException
    - void close() throws UsbException,UsbNotOpenException
    - boolean isActive()
    - boolean isOpen()
    - UsbEndpoint getUsbEndpoint()
    - int syncSubmit(byte[] data) throws UsbException,UsbNotOpenException
    - UsbIrp asyncSubmit(byte[] data) throws UsbException,UsbNotOpenException
    - void syncSubmit(UsbIrp irp) throws UsbException,UsbNotOpenException
    - void asyncSubmit(UsbIrp irp) throws UsbException,UsbNotOpenException
    - void syncSubmit(List list) throws UsbException,UsbNotOpenException
    - void asyncSubmit(List list) throws UsbException,UsbNotOpenException
    - void abortAllSubmissions() throws UsbNotOpenException
    - UsbIrp createUsbIrp()
    - UsbControlIrp createUsbControlIrp(byte bmRequestType, byte bRequest, short wValue, short wIndex)
    - void addUsbPipeListener(UsbPipeListener listener)
    - void removeUsbPipeListener(UsbPipeListener listener)
    - UsbPipe usbPipe = null;

- class Version
    - static void main(String[] args)
    - static String getApiVersion()
    - static String getUsbVersion()