CEN WORKSHOP AGREEMENT

CWA 16008-1

August 2009

ICS 35.240.40

English version

J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Part 1: Base Architecture - Programmer's Reference

This CEN Workshop Agreement has been drafted and approved by a Workshop of representatives of interested parties, the constitution of which is indicated in the foreword of this Workshop Agreement.

The formal process followed by the Workshop in the development of this Workshop Agreement has been endorsed by the National Members of CEN but neither the National Members of CEN nor the CEN Management Centre can be held accountable for the technical content of this CEN Workshop Agreement or possible conflicts with standards or legislation.

This CEN Workshop Agreement can in no way be held as being an official standard developed by CEN and its Members.

This CEN Workshop Agreement is publicly available as a reference document from the CEN Members National Standard Bodies.

CEN members are the national standards bodies of Austria, Belgium, Bulgaria, Cyprus, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland and United Kingdom.

EUROPEAN COMMITTEE FOR STANDARDIZATION
COMITÉ EUROPÉEN DE NORMALISATION
EUROPÄISCHES KOMITEE FÜR NORMUNG

Management Centre: Avenue Marnix 17, B-1000 Brussels

© 2009 CEN All rights of exploitation in any form and by any means reserved worldwide for CEN national Members.

Ref. No.:CWA 16008-1:2009 E

Contents

FOREWORD **4**

HISTORY **6**

1 SCOPE **7**

1.1 OVERVIEW 7
1.2 BASIC OPERATION PRINCIPLES 10
1.3 API SCOPE 11

2 GENERAL CONCEPTS **13**

2.1 OBJECT INSTANTIATION MODEL 13
2.2 BASIC USAGE SEQUENCE 13
2.3 RESERVING DEVICES FOR EXCLUSIVE USE 15
2.4 REMOTE DEVICE ACCESS 16
2.5 ASYNCHRONOUS DEVICE INPUT/OUTPUT AND EVENTS 16
2.6 NUMERIC IDENTIFIERS USED IN J/XFS 17
2.7 THREADS AND FLOW CONTROL 18
2.8 QUEUING 19
2.9 STARTUP & SHUTDOWN 20
2.10 USING COMPLEX DEVICES 21
2.11 FAILURE DETECTION AND REACTION 22
2.12 ENSURING DEVICE INDEPENDENCE 23
2.12.1 Device dependent mechanisms 23
2.12.2 Vendor specific functionality (directIO) 23
2.13 POWER MANAGEMENT 24
2.14 UPDATING FIRMWARE IN A DEVICE 25
2.15 NAMING CONVENTIONS 25
2.16 RETURN VALUES 26
2.17 SECURITY AND ENCRYPTION 26
2.18 HANDLING OF OPEN() ERRORS 27
2.19 THE ENUM PATTERN 30

3 MAIN J/XFS COMPONENTS **31**

3.1 J/XFS PACKAGES 31
3.2 JXFSDEVICEMANAGER 33
3.3 DEVICE CONTROL 38
3.3.1 Object model 38
3.3.2 IJxfsBaseControl 39
3.4 DEVICE SERVICE 48
3.4.1 Object model 48
3.4.2 IJxfsBaseService 49
3.4.3 Complex Devices 55
3.5 DEVICE COMMUNICATION 58

4 EXCEPTIONS AND EVENTS **60**

4.1 EXCEPTIONS 61
4.2 EVENTS 62
4.2.1 Event classes 63
4.2.2 Registering for Events and Event Delivery 66

5 SUPPORT CLASSES **68**

5.1 JXFSSERVER AND JXFSCONFIGURATION 68
5.2 JXFSDEVICEINFORMATION 69
5.3 TRACING AND ERROR LOGGING 71
5.3.1 Overview 71
5.3.2 JxfsLogger 73

5.3.3 Systems Management and Monitoring (e.g. SNMP) ... 77
5.4 J/XFS CONSTANT CODES ... 77
5.5 TEMPORARY DATA AND GENERIC CLASSES ... 83
5.5.1 JxfsType ... 83
5.5.2 JxfsBasicType ... 83
5.5.3 JxfsStatus ... 84
5.5.4 JxfsMediaStatus ... 87
5.5.5 JxfsThresholdStatus ... 91
5.6 REFERENCE IMPLEMENTATION FOR THE J/XFS ENUM PATTERN ... 93
5.7 PERSISTENT DATA ... 94
5.8 VERSION CONTROL ... 95
5.8.1 JxfsVersion ... 97

Foreword

This CWA contains the specifications that define the J/eXtensions for Financial Services (J/XFS) for the Java ™ Platform, as developed by the J/XFS Forum and endorsed by the CEN J/XFS Workshop. J/XFS provides an API for Java applications which need to access financial devices. It is hardware independent and, by using 100% pure Java, also operating system independent.

The CEN J/XFS Workshop gathers suppliers (among others the J/XFS Forum members), service providers as well as banks and other financial service companies. A list of companies participating in this Workshop and in support of this CWA is available from the CEN Secretariat, and at http://www.cen.eu/cenorm/sectors/sectors/isss/activity/jxfs_membership.asp. The specification was agreed upon by the J/XFS Workshop Meeting of 2009-05-6/9 in Brussels, and the final version was sent to CEN for publication on 2009-06-12.

The specification is continuously reviewed and commented in the CEN J/XFS Workshop. The information published in this CWA is furnished for informational purposes only. CEN makes no warranty expressed or implied, with respect to this document. Updates of the specification will be available from the CEN J/XFS Workshop public web pages pending their integration in a new version of the CWA (see http://www.cen.eu/cenorm/sectors/sectors/isss/activity/jxfs_cwas.asp).

The J/XFS specifications are now further developed in the CEN J/XFS Workshop. CEN Workshops are open to all interested parties offering to contribute. Parties interested in participating and parties wanting to submit questions and comments for the J/XFS specifications, please contact the J/XFS Workshop Secretariat hosted in CEN (jxfs-helpdesk@cen.eu).

Questions and comments can also be submitted to the members of the J/XFS Forum through the J/XFS Forum web-site http://www.jxfs.net.

This CWA is composed of the following parts:

- Part 1: J/eXtensions for Financial Services (J/XFS) for the Java Platform – Release 2009 - Base Architecture - Programmer's Reference
- Part 2: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Pin Keypad Device Class Interface - Programmer's Reference
- Part 3: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Magnetic Stripe & Chip Card Device Class Interface - Programmer's Reference
- Part 4: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Text Input/Output Device Class Interface - Programmer's Reference
- Part 5: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Cash Dispenser, Recycler and ATM Device Class Interface - Programmer's Reference
- Part 6: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Printer Device Class Interface - Programmer's Reference
- Part 7: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Alarm Device Class Interface - Programmer's Reference
- Part 8: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Sensors and Indicators Unit Device Class Interface - Programmer's Reference
- Part 9: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Depository Device Class Interface - Programmer's Reference
- Part 10: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Check Reader/Scanner Device Class Interface - Programmer's Reference (deprecated in favour of Part 13)
- Part 11: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Camera Device Class Interface - Programmer's Reference
- Part 12: J/eXtensions for Financial Services (J/XFS) for the Java Platform - Release 2009 - Vendor Dependant Mode Specification - Programmer's Reference
- Part 13: J/eXtensions for Financial Services (J/XFS) for the Java Platform –Scanner Device Class Interface - Programmer’s Reference (recommended replacement for Part 10)

Note: Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. The Java Trademark Guidelines are currently available on the web at http://www.sun.com All other trademarks are trademarks of their respective owners.

This CEN Workshop Agreement is publicly available as a reference document from the National Members of CEN : AENOR, AFNOR, ASRO, BDS, BSI, CSNI, CYS, DIN, DS, ELOT, EVS, IBN, IPQ, IST, LVS, LST, MSA, MSZT, NEN, NSAI, ON, PKN, SEE, SIS, SIST, SFS, SN, SNV, SUTN and UNI.

Comments or suggestions from the users of the CEN Workshop Agreement are welcome and should be addressed to the CEN Management Centre.

History

Major changes from CWA 14923-1:2004 document:
- Handling of errors during open sequence enhanced.
- Added *JxfsBasicType* to ease delivery event data that doesn't extend JxfsType.
- Fixes on Version Control handling.
- Added new Complex Device Service Protection mechanism.
- Added *JxfsEnum* to provide Enum types in new J/XFS proposals and device class interfaces.
- Device Manipulation handling added.
- Clarifications and enhancements on use of JxfsStatus, including new *getStatus(java.util.List)* method.
- Sample on how to initialize *JxfsDeviceManager* fixed.
- Added sentence about atomicity of query and write configuration data.
- *getThresholdState()* has been marked as deprecated.
- JXFS_E_ILLEGAL description has been enhanced.

Major changes from CWA 13937-1:2000 document:

- Updated JxfsThresholdStatus
  Deprecated the *getThresholdState()* method.
  Added new *isOK()* method.
  Added new *JXFS_S_BIN_UNSUPPORTED* state.
  Added new *isSupported()* method.
- Removed paragraph in section 4 allowing intermediate events to occur after operation complete events.
  Intermediate events cannot occur after an operation complete event.
- Improved JxfsMediaStatus class:
- Added state diagrams
- Added new "retracted" state
- The "not supported" state has been deprecated
- Added missing query methods, to provide consistency and orthogonality
- Added new helper method to calculate the transition between two media status
- Added *result* and *extendedResult* properties to the JxfsStatusEvent class, as in the JxfsOperationCompleteEvent class. Added a new constructor with the values for the new properties.
- Renamed the deprecated OperationCompleteEvent, IntermediateEvent and StatusEvent class names to
JxfsOperationCompleteEvent, JxfsIntermediateEvent and JxfsStatusEvent.
- The JxfsConst interface is now deprecated and replaced by IJxfsConst.
- The behaviour with sharing interdependent devices has changed. New support regarding claim/release for
interdependent devices has been added:
Added *generateUniqueTag(), claim(String tag, int timeout), isInterDependentDevice()* and *getInterDependentDevices()* methods
Method *claim(int timeout, int control_id)* of IJxfsBaseService has been deprecated.
- Updated *getDeviceFirmwareVersion()* method of IJxfsBaseControl: exception JXFS_E_NOHARDWARE
can be thrown.
- Changed description on errorCodeExtended on section 4.1.
- Changed description on getExtendedResult of JxfsOperationCompleteEvent: it is vendor denpendent, and
not just device type dependent.
- Changed description on getExtendedResult of JxfsStatusEvent: it is vendor denpendent.
- Added offset constant for VDM.
- At JxfsThresholdState, changed JXFS_S_BIN_UNSUPPORTED to JXFS_S_BIN_NOTSUPPORTED.
- Corrected description of getJxfsMajor() and getJxfsMinor() of JxfsVersion.
- Added deep copy support for JxfsType: new *copy()* method in JxfsType.

1 Scope

1.1 Overview

J/XFS defines a standardized interface to all common financial devices which can be used by *applications and applets*[1] written in the Java programming language. One of the reasons why these new banking applications are written in the Java language is that these programs are supposed to run on many different hardware platforms. One of the main obstacles in doing platform independent programming is accessing devices.
One of the main goals of this standard is to allow access to banking devices in a 100% pure Java way on both thin and thick clients, e.g. on a network computer as well as in a Linux, Windows, OS/2 or Unix workstation.

Another goal is to allow the remote access to devices on different machines. Additional efforts have to be done to find and access these devices. This is the main reason why central administration processes and an additional communication layer are also defined by this architecture.
If only local access to devices is needed, an implementation may omit this communication layer. No change is required to the Device Controls or Device Services. So, neither the application programmer nor the hardware manufacturer who programs a Device Service need be aware of whether or not a communication layer exists in the middle.

Due to the nature of network computers which are supported as clients, it is not possible to guarantee that local persistent storage possibilities exist on each client. Therefore, any configuration information must be kept on a central server. If local storage exists and no central configuration possibilities are required, all configuration information can also be kept on the local workstation.

The basic architecture of J/XFS is similar to the JavaPOS[2] architecture. It is event driven and asynchronous.
Three basic levels are defined in JavaPOS. For J/XFS this model is extended by a communication layer, which provides device communication that allows distribution of applications and devices within a network. So we have the following layers in J/XFS:

- Application or applet
- Device Control and Manager
- Device Communication
- Device Service

The Device Control API defines the way a Java application or applet can communicate with a specific device. Additionally, the Device Control layer contains the central Device Manager which organizes access and location of the services. It is the central coordinating instance in any Java VM which must access financial devices.
The Device Communication Layer is the layer which resolves the sharing of devices. It is invisible to the application. The only exception is that network errors are presented to the application. It must be able to cope with lost connections.
The Device Service is the layer supplied by the device manufacturer for use with J/XFS. It has a defined API which allows the Device Control and Device Communication layer to request device actions and translates them into the device specific commands which are then sent to the physical attached device. The way of connecting to the local device is not defined in this standard, it is rather left to the service provider. In the case of devices which attach through the serial or parallel ports the Java CommAPI may be used. Thus, the

[1] J/XFS is designed to be also usable by applets in a browser e.g. on a network computer. So, for the remainder of the document, ‘application‘ also always means ‘applet’.
[2] JavaPOS (Java point of sale) is an initiative for the retail industry with the goal of providing unified device access to POS devices. See http://www.javapos.com .

Device Service layer may not be 100% pure Java but the complete basic infrastructure of J/XFS is.

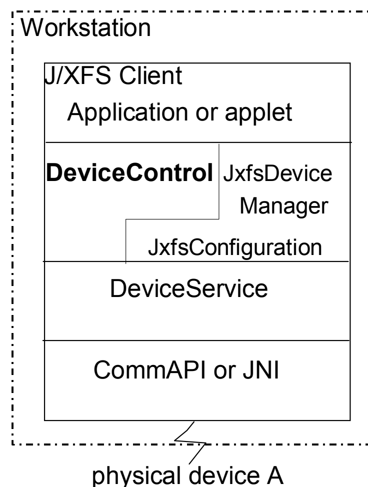
Application developers program against Device Control objects and the Device Manager which reside in the Device Control Layer. This is the usual interface between applications and J/XFS Devices. Device Control objects access the Device Manager to get access to an associated Device Service. And the Device Service objects finally provide the functionality to access the real device (i.e. they are like a device driver).

Java applications or applets run in a Java virtual machine (Java VM), possibly embedded in a WWW browser. Under some operating systems (i.e. JavaPC) only one JavaVM can run in the system (i.e. it allows multiple threads in the program but not multiple programs). There the J/XFS layers must run in the same process context as the application or applet. The bigger operating systems like OS/2, Windows or Unix can run multiple JavaVMs in parallel. This must also be possible with J/XFS.
Thus, the design of J/XFS must cope with the following scenarios:

1. A single JavaVM is present on a workstation and is running the application or applet (J/XFS Client) which accesses only local devices.
2. A single JavaVM is present on a workstation and is running the application or applet which also accesses one or more remote devices (which are physically attached to another workstation).
3. Multiple JavaVMs, each running an application or applet, run on the same workstation. As ports through which devices are connected (i.e. the com port) cannot be accessed in parallel only one of the running applications must control the local devices. For this scenario there is a distinction of whether these multiple applications really run in parallel or if they are started only one at any time.

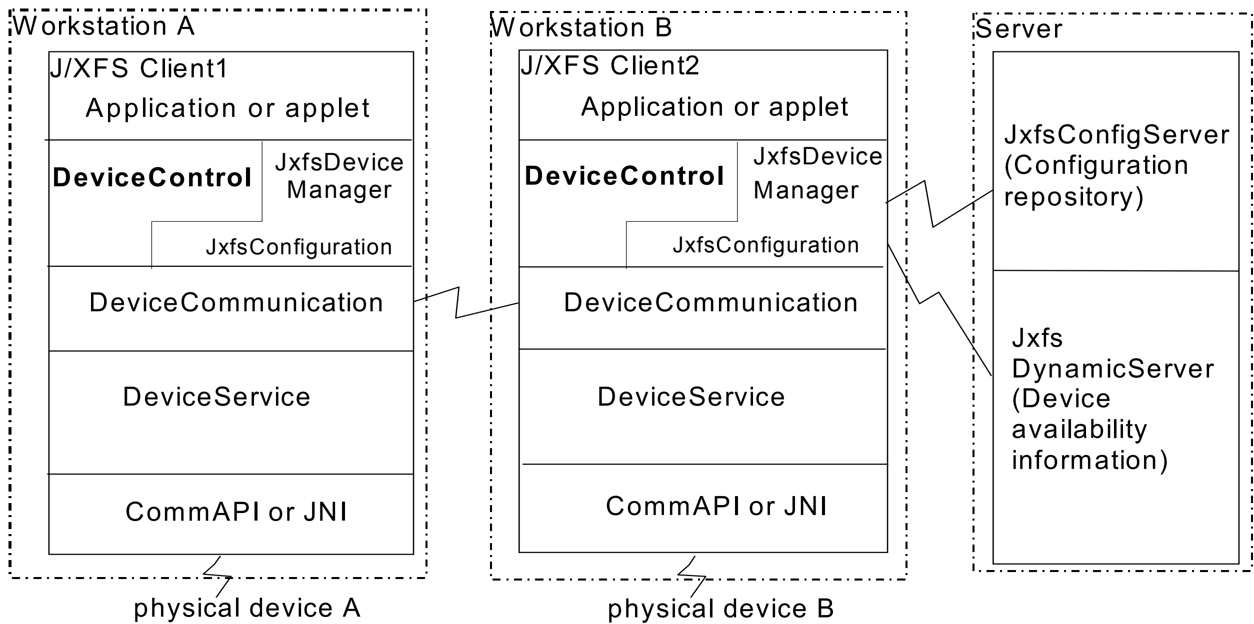
Only in the very simple first case it is possible to omit the communication layer. A device access from an application in one JavaVM to another one running on the same machine is similar to accessing a device on another machine as also interprocess communication is needed.

In the case **1** the following layers are present:



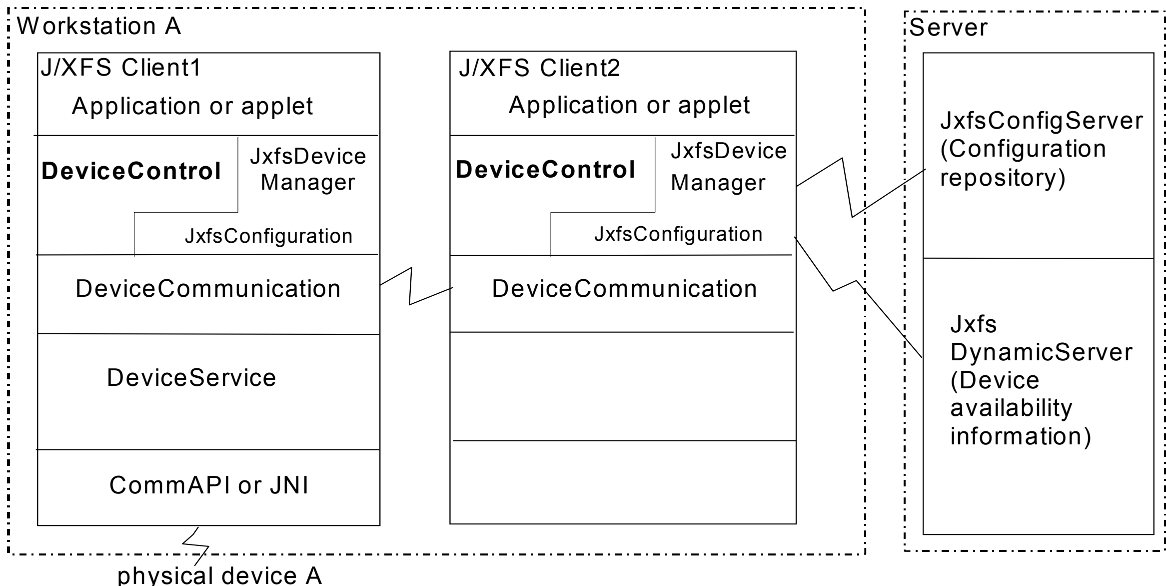
The application requests the device from the JxfsDeviceManager and accesses the device via the Device Control. The Device Control is directly connected to the Device Service for the device. The JxfsDeviceManager controls both objects. The whole configuration may be kept locally, either hard-coded in JxfsConfiguration object or somehow configurable on disk or in memory.

The next diagram illustrates the basic architecture for the case **2** (Remark: Both Client1 and Client2 access the server in the same way. The connections for Client1 are not shown here):



The clients have to start the J/XFS infrastructure upon startup. When the JxfsDeviceManager is instantiated it queries (via a JxfsConfiguration object) the central information repository relating to all J/XFS devices. This contains the configuration information for each workstation (the available devices, where they are attached, which service class handles them, whether a device can only be accessed on the local machine or also from a remote workstation, needed initialization information for each local device etc.). The means with which this information is stored is not within the scope of J/XFS. It might be a file, a database or an object repository. Only the access API is defined.
After successful initialization any locally connected devices which should be available to other workstations are registered with the JxfsDynamicServer. It thus contains up-to-date information on the availability of the devices. Any other workstation can query for a list of available remote devices there.

In the case **3** mentioned on the last page the “J/XFS Client1” and “J/XFS Client2” processes may run on the same workstation. If it is guaranteed that only one process runs at any time no problems arise. If they run in parallel, however, the following scenario must be used:



Why is it required that the J/XFS client2 has no local device access? Because it is impossible that multiple Device Services for the same physical device are started. And why

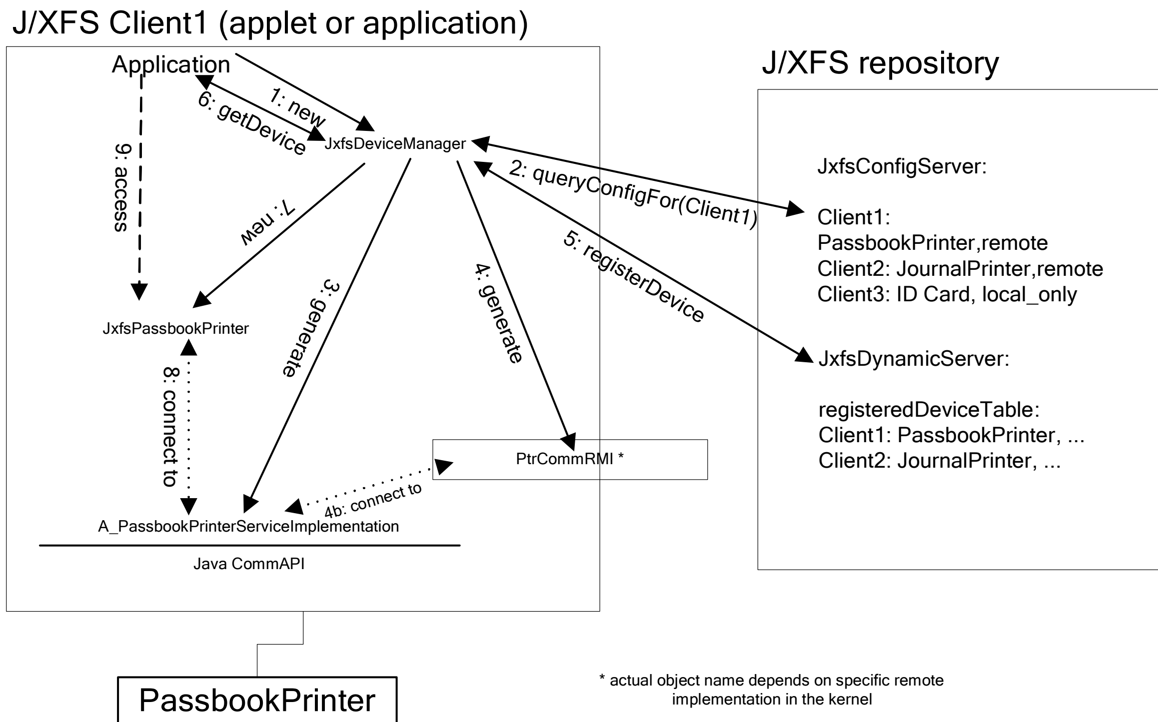
that? Because access to the device through the physical port it is connected to is restricted to 1 process only.
So, the configuration of J/XFS must cope with this fact. Several J/XFS clients running in parallel on one workstation must be differently configured. One must be clearly defined to be the J/XFS client which instantiates all the Device Services of the local devices and any other client running on this workstation must now access these devices as if they were remote.
Additionally, the first, controlling, J/XFS client must also be the first to start up and the last to shut down. If not, the others won't be able to access the devices any longer (because the Device Service is no longer available).

The correct way to configure such a system to use J/XFS would be to use a (likely very small) J/XFS client application which has the sole need to instantiate the Device Services needed to access all the local devices. It can be started automatically during the boot process of the workstation and will not terminate until the shutdown of the computer. The other applications are now free to access this device as if it were remote.

1.2 Basic operation principles

Let's now look in depth at the operation principles which define the way in which J/XFS uses financial devices.

First, we are giving a short **overview** about what happens if a device is accessed (a small distributed sample scenario). It is illustrated in the following graphic and used to describe the control flow in the following description.



On each client which participates in J/XFS device access, the application or applet generates a new JxfsDeviceManager object upon startup (1). It does the initialization of the J/XFS subsystem by first querying the repository for its configuration data (2). Then it instantiates all the Device Services for the locally connected devices (3). If configured for remote access also the objects responsible for enabling this remote connection are generated (4) and connected to their Device Service (4b). If all is successful, the now available Device Service is registered at the central directory where other J/XFS clients can query for accessible devices (5).
If the local application or applet now wants to access a J/XFS device, it has to ask the JxfsDeviceManager for the device. This happens via the *getDevice*() method (6). During this method the JxfsDeviceManager generates a Device Control (7), locates the Device

Service and connects it to the Control (8) and returns the Device Control to the application which can now start to use the device (9).

If the requested device were on a remote machine, then the JxfsDeviceManager would, in the *getDevice*() method, first ask the JxfsServer for the location of the requested device and, instead of connecting the generated Device Control to a local Device Service, use a remote object to establish a connection to the remote device. Note that the Device Control itself (and with it the application) would not notice any difference.

In order to minimize network traffic, the design of the Device Service API was done so that the granularity of the device access methods is as big as possible without impacting functionality.
If a Device Control is requested, the JxfsDeviceManager has to load the corresponding classes and connect the Device Control to a Device Communication or Device Service object. It also keeps track of whether a Service (or Communication) object already exists for the specified device. If *true*, it connects this to the Device Control, if not, it generates it.

There exists exactly one Device Service object for each physical device. If additional requests arrive, they are all routed to the same Device Service object. But there is no restriction on the number of Device Control objects accessing the device within J/XFS! Thus, it is the duty of the service object to synchronize concurrent commands to a device.

There are two **types of methods** within J/XFS: *Synchronous and asynchronous methods*. The former ones are always used for small functions which do not need to access the device, like e.g. querying the version of an object, querying the device status (which is held in the Device Service) or repository access. If synchronous methods fail they throw a JxfsException with additional information.
Any operation on a device is asynchronous. This means, that it immediately returns an identificationID for the requested operation to the application. The operation itself is queued and executed in the order it arrives at the Device Service (except if the device is claimed, see the later chapter “Reserving devices for exclusive use” for details). During execution on the device the Device Service sends one or more **events** to the application. *Intermediate Events* are sent to inform of intermediate steps during the execution of the command, and as soon as the operation is completed, one *JxfsOperationCompleteEvent* is sent. Both events also contain the identificationID to enable the application to uniquely identify the operation the event belongs to.
Additionally, if the device status changes, the Device Service has the possibility to send *JxfsStatusEvents* asynchronously to the application.
For each of the defined three event types the application can choose whether or not it wants to receive these events by calling the respective addListener method of the Device Control.

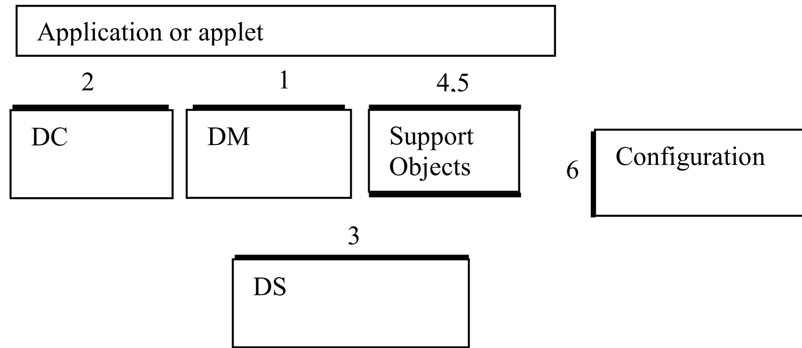
The above explanations are only meant to give a short overview of the design of J/XFS, they are all expanded in the following chapters.

1.3 API Scope

J/XFS defines the following APIs:

1. The JxfsDeviceManager API between the application or applet and the internal classes.
2. The Device Control API the application or applet uses to access a specific device, for all supported device types.
3. The Device Service API used by the Device Control to access a specific device type. This interface must be implemented by the manufacturer of a specific hardware device.
4. APIs for the additional supporting classes such as JxfsLogger, JxfsType etc.
5. All relevant Event objects and Error codes used by the Device Control and Device Service layer.
6. Definition of relevant data stored in the JxfsConfigServer and JxfsDynamicServer and their respective access methods.

The following graphic outlines the APIs:



2 General Concepts

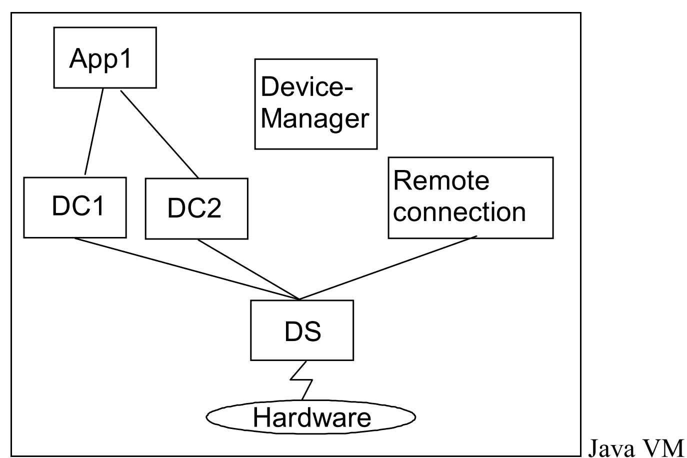
In the following section several of the key concepts in the J/XFS design are described in detail. This information provides the foundation for the API design found in the following chapters.
This is a very important part of this document as it serves to give a common understanding of how the properties, methods and events are to be used in a real J/XFS compliant application or applet.

2.1 Object instantiation model

In the J/XFS architecture there are a lot of classes involved. Some of them reside in local machines, some of them on remote machines. Some of them only occur once, others may have multiple instances.

Singleton objects are objects which are instantiated exactly once in a Java VM. They are the JxfsDeviceManager and the JxfsLogger.

The applications access a device through its Device Control object. A single application may access multiple DC-objects for the same device (perhaps in different parts of the program). If an application remotely wants to access a device it also gets a Device Control, and a remote connection to the DS is established. Thus, Device Controls for the devices can occur multiple times.
The Device Service for a specific device manages access to this device. It is instantiated *only once for each device*.
The following graphic gives a short overview about this. For the remote case the communication comes in via an object which looks just like another Device Control accessing the same Device Service object for the device.



If an application or applet wants to gain exclusive access to a device, it uses the *claim()* method on its Device Control. The basis is the control object, i.e. that if an application gets two control objects for the same device a *claim()* to the second one will fail, even though it is used by the same application unless the *claim()* uses the same tag value. So, an application can either use one control for all its activities with a device or use the claiming mechanism to synchronize its different program parts if necessary.
The synchronization itself is done in the Device Service layer. The Device Control only routes the requests to the DS and cannot make a decision on its own. The Device Service must decide whether or not a *claim()* is successful. The same applies for the method calls. Only the Device Service can decide whether or not a method invocation is executed or queued for execution.

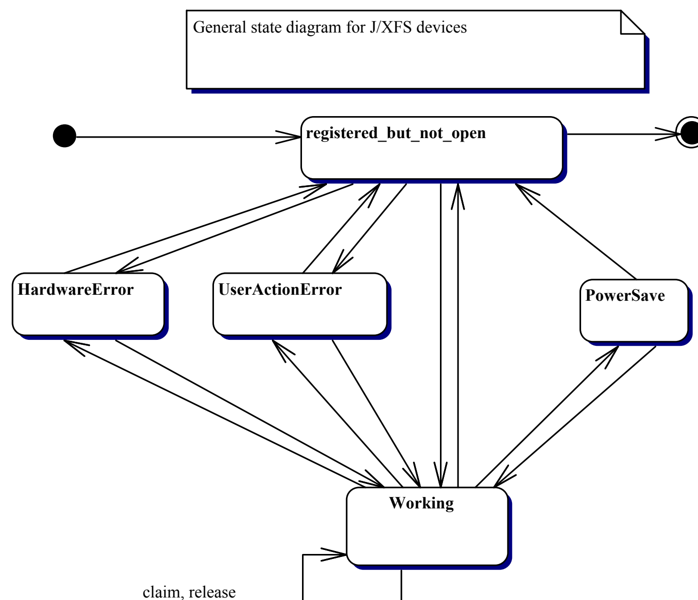
2.2 Basic usage sequence

The basic application usage scenario for all devices is defined by the following order:

1. The application or applet gets access to a new Device Control by calling the *getDevice*() method of the JxfsDeviceManager.
2. It registers for the events it is interested in by using the addXXXListener() methods of the DC (Note: Registering and deregistering for events is always possible, not only here).
3. It issues an *open*() call to the DC. The first open() received by a Device Service physically connects to the device.
4. It controls the device through the device specific functions. If it wants to exclusively use the device it can use the *claim()* and *release()* methods.
5. It closes the device with the method *close*(). It may now restart at number 3 or
6. It removes its listeners from the Device Control.
7. It stops using the Device Control and deregisters the Device Control by calling the *deregisterDevice*() method of the Device Control.

The first methods here is the *getDevice*() method offered by the Device Manager which enables any functionality of the Device Control. At this time, the infrastructure down to the specific device is erected and guaranteed to work. Before the *open* () method is called any other method call except for the addXXXListener() methods results in a JXFS_E_CLOSED exception. The reverse operation to *getDevice*() is the *deregisterDevice*() method of the Device Control after which the DC is no longer usable.
Then two bracket method pairs exists. One is the open-close pair which enables access to the other functions of the device. Any device specific methods can then be used by the accessing application or applet.
The other bracket is optional to reserve the device for exclusive use. It is the claim-release pair of methods.

All J/XFS devices have the states shown in the diagram.



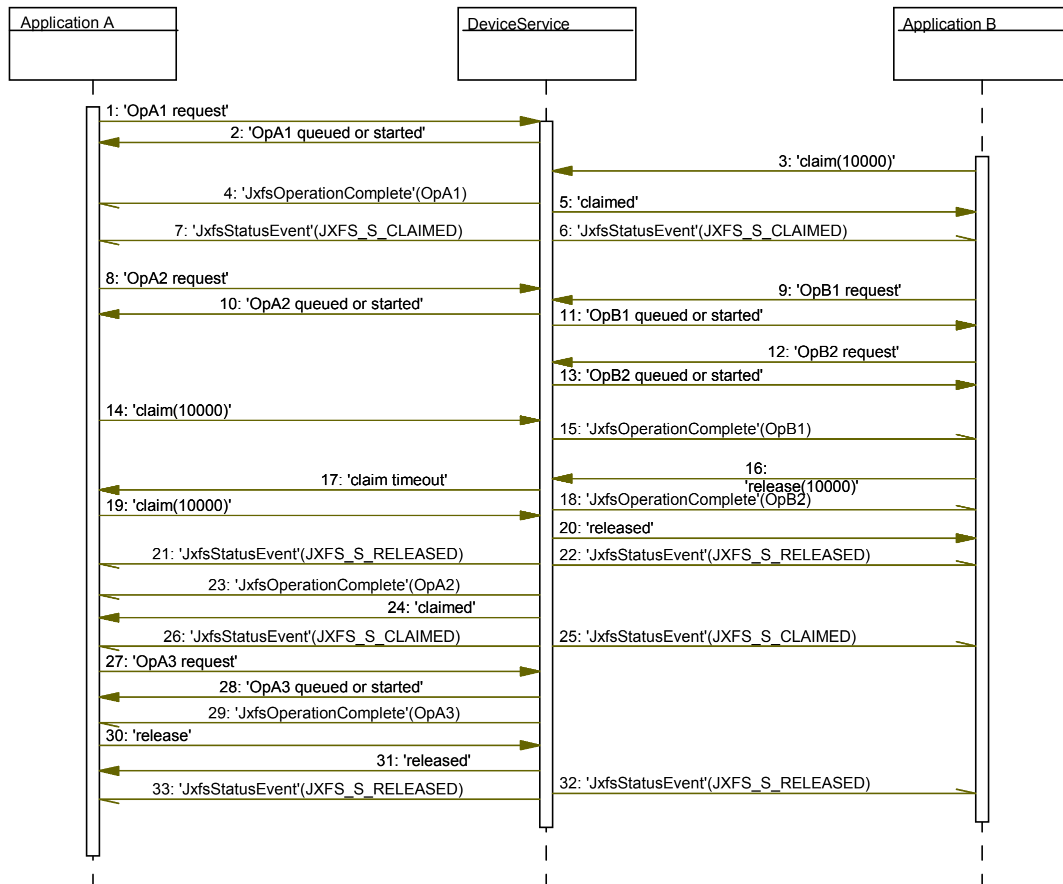
After the start (big dot) they are registered but not open, and an *open* call brings them to ‘Working’ (everything ready). During their operation devices may be claimed and released. Also, if an error occurs, they do only temporarily switch to one of the error states and after having it fixed they return to their previous state. It is also possible that the device switches to power save mode if it is not used for a longer period of time. It leaves this state if any command is issued.
If an error already exists like HardwareError (maybe the device is defective) or a UserActionError (paper missing etc) then the *open* succeeds anyway but the status reflects the error state.
Only a device that is not claimed can be closed and unregistered by the application.

Any device state change is reported by a *JxfsStatusEvent* to the application (except for the first change from ‘registered_but_not_open’ to ‘Working’ as this is only of interest to the DC calling the *open* () which is informed via a *JxfsOperationCompleteEvent*). For details see the description of the JxfsStatusEvent in the ‘Events’ chapter.

2.3 Reserving devices for exclusive use

If an application or applet accesses a device it is sometimes necessary to restrict access to this device for other applications or applets for a certain amount of time. This can be needed if multiple operations must be done consecutively and not interrupted by operations from other device controls. This is made possible by using a *claim()* method to lock the device and a corresponding *release()* method, which frees the device again.
There is, however, no method in J/XFS which requires the use of the claim/release bracket.

The following sequence diagram outlines the basic concept of claim and release (and also illustrates nicely the event send mechanism). Please note that the application does not directly access the Device Service as depicted here but of course issues any calls and receives any events via their Device Control object. The reason the Device Service is shown here is that it is the coordinating object.



If an application starts the claim request the Device Service queues this request and goes to a claim_pending state. It will not grant the claim until any operations still in its queue are finished (3,4,5). All requests are queued using a FIFO (first in first out) mechanism. After the claim was granted any connected DC is informed via a JxfsStatusEvent (6,7). From the moment the claim request arrives any incoming new requests from other application (8,9) are queued for execution after the release. Additional claim requests (14) have to wait for the release of the current claim.
The claim is only granted if a specified timeout has not expired before all other running or pending operations are completed (as happens in 14 - 17).
Likewise the release must be accompanied by a timeout value which defines how long to wait until pending operations are finished (18). Only after that the release returns (20) and the according JxfsStatusEvent is sent to every DC (21,22). If more claim requests are in the queue, the next one is granted (24). If not, every Device Control is free to directly access the device operations again.

2.4 Remote device access

Accessing a device which is locally connected to the workstation the application or applet is running on is simple and straightforward. J/XFS, however, also deals with device access to devices which are connected to a remote machine. For the application or applet this makes no difference to the local access. So, access to the devices can be shared among several J/XFS clients.
But also there is NO restriction of the number of Device Controls which are allowed to connect to a Device Service (other than system limits); especially it is not possible to restrict the access to a DS to only one DC.
The architecture is designed in such a way that it keeps the common local-access case simple but it is powerful enough that devices, no matter where they are attached, can be accessed identically by the application.
The JxfsDeviceManager knows via its JxfsServer of the existence and availability of the devices which are available on other machines and can offer them to the application accordingly.
How are the remote devices identified?
Every device has a unique name on the workstation it is connected to. This is sufficient to identify it locally. In the case of a remote device this name is augmented by the workstation name the device is connected to. Although an implementation of J/XFS may want to use its own format for the unique identification of a device we suggest the following format: "<devicename>@<hostname>" (i.e. printer2@workstation1.acme.com).
This is a readable format which also allows for simple separation of the two parts of that identifier.
This identifier is used by the application to request a specific device from the DeviceManager and also in the repository.

2.5 Asynchronous device input/output and events

As stated in chapter 1.2, Basic operation principles, there are two **types of methods** within J/XFS: *Synchronous and asynchronous methods*. The former ones are always used for small functions which do not need to access the device, like e.g. querying the version of an object, querying the device status (which is held in the Device Service) or repository access.
All properties on device controls and services are accessed synchronously from cached copies of the device's status. Although this means that the status returned may be inaccurate, applications should always assume that it is accurate.
Any operation on the device is asynchronous. This is *true* for both input and output data.
The usual flow is that the application issues a command by calling the appropriate method in the Device Control (i.e. ReadTracks for a magnetic stripe reader or PrintForm for a Printer). If the given parameters are not correct, immediately an exception is thrown; if they are valid the operation is issued asynchronously and an *identificationID* of type int is returned.
This identificationID can be used by the application to:

- Stop a running operation by giving the identificationID as a parameter to the cancel() command.
- Use it to identify the operation in the asynchronously returned *JxfsIntermediateEvent* and JxfsOperationCompleteEvent.

This identificationID is especially important if another asynchronous operation is called again (possibly with different parameters) before the first one is finished, i.e. has sent its JxfsOperationCompleteEvent. Then the identificationID can be used to distinguish between the two events.
If an error occurs during the operation, it is terminated and the JxfsOperationCompleteEvent is sent giving the corresponding error code.
The general rule in J/XFS regarding the use of asynchronous methods is:

> *Whenever the physical device has to be accessed to complete an operation an asynchronous method is used, whenever only a Device Service internal property is queried, a synchronous method is sufficient.*

So, e.g. querying the status of a device is synchronous because the status is (after the *open*) always known internally in the Device Service.

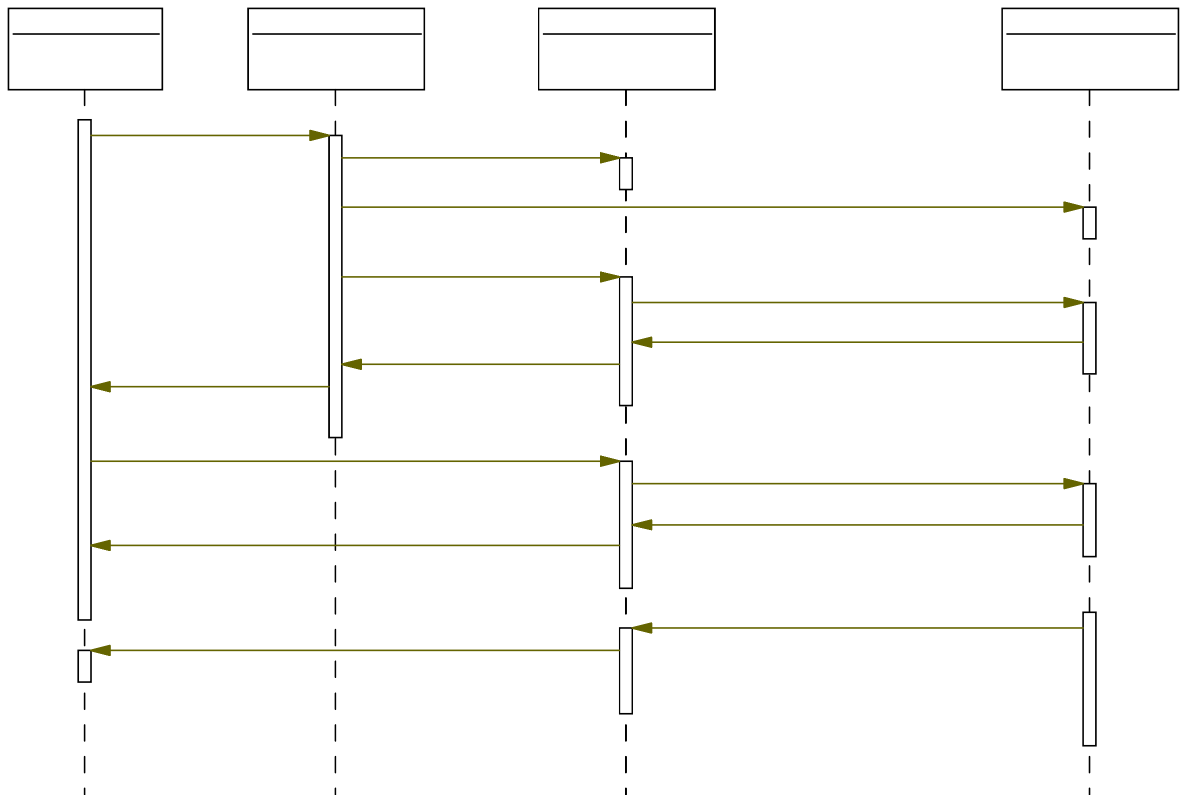
In some circumstances (e.g. for very small, quick operations) it may occur that an Event is returned to the application (via the call to one of the <eventType>occurred() methods) *before* the method call itself has returned, thus providing the application with an identificationID. Pitily this cannot be remedied by the J/XFS infrastructure itself as the same problem may occur because the application is too slow of informing its listener object of the arrived id.
Thus, the application must be prepared to either accept events for which the identificationID is not yet known and buffer them or prevent such a situation.
One simple and advisable way to prevent such a situation this is to simply declare the <eventType>occurred() method of the event listener instance "synchronized" and also synchronize the event provoking call on that instance. Thus, the event delivery is postponed until the method call has returned. See the Java Tutorial http://java.sun.com/tutorial for more details on Java Threading.

2.6 Numeric identifiers used in J/XFS

Identifier are used to identify Device Controls, operation types and asynchronous operation requests. These identifiers are of the *int* type. The following list shows the types of identifiers:

- **control_id**
  For each Device Service call the Device Control has to identify itself. This is done by adding a unique identifier as a parameter to any method call from DC to DS. The Device Control receives this *control_id* from the Device Service during registration. The control ids used by device services to identify controls are only unique within device services not between device services.
- **operationID**
  This identifier is included in each *JxfsOperationCompleteEvent* and *JxfsIntermediateEvent*. It identifies the type of operation it belongs to and is unique for each type of operation.
- **identificationID**
  This identifier is received by the application when issuing an asynchronous operation request. The *identificationID* is generated by the Device Service. The results of an asynchronous operation are sent via a JxfsOperationCompleteEvent containing this *identificationID*. So, the application can distinguish between results especially after issuing many operations in a short amount of time.

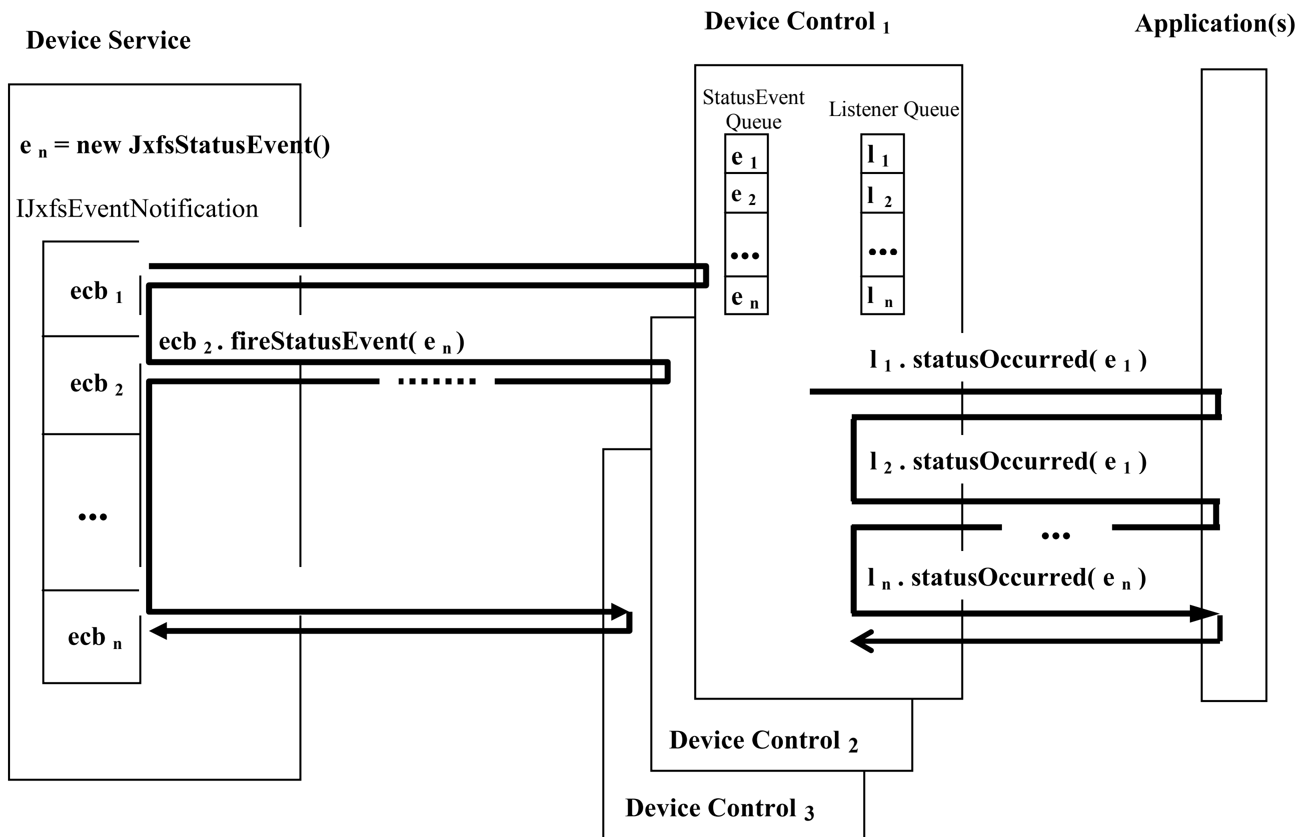
The following diagram shows how identifiers are used.



2.7 Threads and flow control

In the Java programming language using threads is common. In the case of J/XFS and applications using it, threads are used to decouple the device handling from the application logic, because the Device Service must be able to respond to asynchronous events coming from the attached devices. Furthermore there are some operations (like printing) which take a long time and it is unacceptable to lock the application while this takes place.
So, where are different threads likely used?
Each Device Service method might be called in a separate thread. It is very important to take this into account when programming a Device Service by programming in a thread-safe way.
Also, each Device Service implementation will have a number of distinct threads. At least one is normally used to wait for asynchronous messages from the device, another one is needed to wait for asynchronous requests from the application and yet another one is used for event delivery to the application (in order not to block the service). The events generated in the service are delivered to the control in the thread context of one of the service's threads if local.
The Device Control manages the events to be delivered to the application. It has **two separate queues**, one for the JxfsStatusEvent and the other one for JxfsOperationCompleteEvent and JxfsIntermediateEvent. Every new event generated by the Device Service is inserted into one of these queues by a DC thread. Then, using exactly one separate thread per queue, the Device Control delivers the events to the application by calling all connected event listeners (i.e. methods under application control).
An additional feature of the queues is that JxfsOperationCompleteEvents and JxfsIntermediateEvents are now delivered to the application in exactly the order in which they were generated by the Device Service.
For the application this means that it must be aware of a foreign thread context (apart from its own threading model) in its registered listener methods which will be executed in a different thread context. So, variables and routines called in the event processing method of the application should be synchronized to avoid unwanted interaction with the application's main thread.

The following diagram outlines the different threads involved in the delivery of events:



Each thick black arrow represents one thread. The Device Service generates an event and delivers it to every Device Control it is connected to (Shown here are three DC's). The threads only add the new event to the respective queue in the Device Control and immediately return to the DS. Now a separate thread running in the DC always picks the next available event in the queue and delivers it to all application objects which have registered to receive the Status events (Listener queue) via their listener methods, here: statusOccurred(). As a result the DS thread is always decoupled from any application level processing... a very desirable feature.
As explained above, a second identical queue and delivery thread exists for the JxfsOperationComplete and JxfsIntermediate events.

2.8 Queuing

There are several places in the J/XFS architecture where queuing takes place:

- As stated above, each Device Control has two queues where the events are inserted and each one is read by a single thread which delivers the events to the application.
- As all operations in J/XFS are asynchronous the Device Services internally have to administer a queue which collects the requests and works on them one after the other.
- If multiple claim requests arrive at the Device Service it must queue them until their respective timeouts occur. A claim will be granted to a Device Control if the Device Service is not already claimed and all pending operations which arrived at the Device Service earlier than the claim request finished execution. If a Device Control wants to give up its claim it must issue a release request. The release request will be queued like all other asynchronous operations and claim requests.

Queued Commands

The concern here is the scenario where an application sends commands to a device that is locked by another application - the commands are queued and actioned after the other

application has unlocked the device. The problem is that by the time any queued commands were actioned (after a release) then the calling application may:

- No longer wish the commands to be actioned.
- The device may no longer be in the state the application expected when it first sent the commands.
- An unexpected or invalid sequence of commands may be queued by several applications.

If an application developer is concerned about any of the above then the logical thing to do is to cancel the command when it receives the responses telling it that the command is queued. If an application does not contain this logic then it will potentially encounter unexpected device behavior.

Recommended Application Behaviour

There is no guarantee that commands will be actioned immediately if the device has not been previously locked by the calling application. Applications requiring guaranteed immediate access must lock the device before sending any commands.
Alternatively, in the circumstances where the application wants to send commands to a device without locking (see some scenarios described later), he can invoke a concurrent timer specifying the amount of time he is willing to wait for the command to be actioned - if this time elapses then he cancels the associated command.

Possible Scenarios

The following describes some example scenarios where the application may wish to access a device immediately without locking the device:

- If an application sends account information to the journal printer for auditing reasons, then it must be guaranteed that the data will be printed. In this scenario, you probably do not want to lock the device, because any thread may want to print an audit message in the journal.
- Accessing a supervisor panel (TTU) which is used by many co-operating application to display status and error information concurrently.

Summary

The responsibility in this area is with the application developer to add the 'management of the queuing complexity' at the application layer.

2.9 Startup & Shutdown

Prior to use of any device, the device environment must be initialized.
This is done by the first application or applet running in a specific VM on a system. It must issue a call to the Device Manager (DM.initialize(..)) with some initialization parameters. The Device Manager then initializes the local as well as the remote infrastructure for use by the applications. This is done by issuing the start() method for all locally attached devices.
In special cases this initialization could be done by defining the DM as a service to start up automatically, so an application or applet does not need to initialize it explicitly. But usually the application or applet must initialize the J/XFS infrastructure explicitly.
In the case of a separate workstation which only supplies one or more devices for use by applications running on remote computers, the Device Manager can be automatically started and configured to only support the given devices. Another possibility for this scenario would be to have a special miniature application running on this ‘device server’ which perhaps just shows the current status of the devices.

The *shutdown* of the controlling application or applet is the shutdown of the J/XFS infrastructure. Upon shutdown the application or applet should tell the Device Manager

about what is going to happen, so that it can shut down all the services it provides. This is achieved by the application or applet calling DM.shutdown() upon ending.
In this method the JxfsDeviceManager calls any instantiated Device Service and Device Communication objects and instructs them to shut down. They have to inform any of their peer Device Controls, which have to notice the status change and deactivate any device access.
There are additional methods in the Device Manager to control Service instantiation and destruction of local DS's: *start* and *stop.* Using stop the application can intentionally stop (or shut down) the specified DS only. It can then use start to try to re-start the stopped Device Service. If the device is accessed locally, a *getDevice*() call also implicitly issues a start command if it is not yet started (thus allowing an Activation-on-demand behavior).

For a more detailed description see the explanation of the shutdown() method in the chapter on the JxfsDeviceManager, the Device Control and the Device Service.

2.10 Using complex devices

It is common that a financial peripheral device consists of more than one subcomponent, e.g. a statement printer consists of the specialized printing engine and also contains an ID Card reader of some type and perhaps a small line display. Or, there is a device which consists of a magnetic card reading device (MSD) as well as a PINPad (PIN).
Usually these devices are accessed via the same connection (IO port), and additional dependencies may exist.
From the J/XFS point of view these devices are represented by a number of separate devices which are controlled separately. The device manufacturer writes just a single Device Service class which contains interfaces for both required J/XFS device types. If an application wishes to use MSD and PIN device mentioned above it must request a separate MSD Control and a PIN Control. The configuration of J/XFS then maps both devices to the same service class which is instantiated only once. The single Device Service class can thus easily synchronize access to the combined devices.
The basic idea behind this is that it should not be necessary to change the application when 2 separate attached devices are replaced by a single complex device.
If two or more devices are used on a call-by-call basis, i.e. without claiming them to lock the device, using a complex device is identical to using two separate devices.
If one device is claimed, then used and then released before the second device is claimed no problems arise either. The code would look like:

```
JxfsMSDControl c=myDeviceManager.getDevice("CardReader");
...work with c...
JxfsPINPadControl p=myDeviceManager.getDevice("PINPad");
...work with p...
```

The tricky part occurs when two (or more) combined devices are claimed simultaneously in order to work with them. A *claim()* of one of its subdevices would block all subdevices at once. Thus, a claim to the second subdevice of a complex device will fail as the device is already claimed:

```
JxfsMSDControl c=myDeviceManager.getDevice("CardReader");
c.open(this);
c.claim(JXFS_FOREVER);
JxfsPINPadControl p=myDeviceManager.getDevice("PINPad");
p.open(this);
p.claim(JXFS_FOREVER); // WORKS for separate but
                       // fails for complex devices!
...work with c and p...
c. release(); p.release();
c.close();  p.close(); ...
```

In order to avoid a potential deadlock, there is a form of the claim method which accepts a unique identifier:

boolean claim(String tag, int timeout) throws JxfsException;

If the device is already claimed by another device control, this method will block for the specified timeout –just as with the claim(int timeout) method– except if the tag parameter is the same as the one passed by the owner of the claim: in such a case, the claim will be granted.
Therefore, applications that need to claim two or more devices at a time, and which may be implemented by a single Device Service, can use this form of the claim method passing the

same tag value. A unique tag can be obtained by calling the *generateUniqueTag*() method of JxfsManager. For more information, see the description of the claim methods of IJxfsBaseControl and the *generateUniqueTag* method of JxfsManager.

2.11 Failure detection and reaction

As J/XFS provides for access to devices which are connected to another computer, the subject of failure detection and possible reactions to it is an important issue.

Currently several problem areas are known:
The first one is how to ensure that the central J/XFS dynamic device availability cache is kept up to date if a workstation breaks down.
For this release of the J/XFS Architecture the following approach is recommended: The dynamic server cache is updated if a connection request to a workstation failed or is detected inside the server process. As long as no device is used a possible *false* entry in this repository is considered bearable.
The second area of concern is the peer connection between devices. Two connectionFailure() methods as well as a predefined JxfsStatusEvent are provided by J/XFS which allow a communication layer to inform both the Device Service as well as the Device Control of a communication breakdown. Detection of such a communication failure without actual operations going on can be done via a heartbeat mechanism (such that the connections are checked regularly). During operations a failure is detected immediately.
A third problem area is to define how to react to breakdowns during an operation (i.e. has the command been sent, and if no JxfsOperationCompleteEvent arrives how to query the device if the operation was done?). As these are rare special circumstances J/XFS provides the logging mechanism separate from the event mechanism. In the above failure a system administrator could look up in the log to find out about the operation's state. The general assumption if a JxfsOperationCompleteEvent for an issued operation is missing must be that the operation was not completed.
The implementation of a heart beating mechanism to detect communication breakdowns between a device service in one Java VM and a device control in another Java VM is performed by device comm objects; whose implementation is vendor dependant.
If an operation is completed by a Device Service but there is a Network failure before the response can be transmitted to the Device Control, the response is discarded and an error is logged by the Device Service identifying that the operation took place but the response was discarded.

To further clarify things the following rules apply:

1. A network error is detected at the *Device Service* side. The Device Service itself is informed by the Communication layer via the connectionFailure() method. It must now do the following:
   * Break any claim held by DC in question
   * Cancel all pending IOs from this DC
   * Remove all EventListeners held by this DC
   * Retire the control _id
     (NEVER reuse it... DC may not know communication was lost)
   * Log error message
   * Refuse ANY subsequent requests with this Control ID
2. A network error is detected at the clients side in the *Device Control* either by an exception returned from a method call or via a connectionFailure() method being called by the communication layer.
   It can be expected that there is no use to re-try any calls because such retries should already have taken place in the communication layer.
   The following reaction is expected by the DC:
   * Generate and post a remote error status event to the application (This is the only place where the DC itself must generate an event!).
   * Try to log the error message
   * As the DS is not available any more it must perform internal "close" and "unregister" operations

* Go into ‘unregistered’ state, i.e. refuse ANY subsequent requests from the application.

3. A network error is detected at the *application* because a JxfsStatusEvent is received from the DC. It must now assume the following:
   * Any existing claim request on device is no longer valid.
   * All new requests to this Device Control will be rejected, including the getStatus() call.
   * All pending operations must be assumed cancelled; no separate OC Events will be received for these operations!
   * Actual device hardware is in unknown state.
   * Device Control will be auto-closed.
   * All EventListener threads will be auto-removed.
   * The Device Control will be auto-deregistered.

   How can it now go on? The application has to repeat the normal device sequence starting from DeviceManager::*getDevice*(). If communication is STILL down, this call will also fail.

2.12 Ensuring device independence

2.12.1 Device dependent mechanisms

One of the main goals of J/XFS is to allow an application to be independent of the make of the device it is using. If it uses e.g. a Magstripe reader it must not need to bother whether the physical device is from vendor A or vendor B.
Acknowledging the fact that differences in devices exists, however, and that there may be scenarios where an application *intentionally* wants to access device specific functionality, in turn giving up device independence.
The application must have checked the Device Service type before it uses device specific information.
And every DS is required to work even when the application does not analyze this field!

J/XFS provides the following access mechanisms to device specific things:

- extendedErrorCode in the JxfsException
  This allows a Device Service to deliver a specific return code which the application might analyse and take specific action.
- extendedResult in the JxfsOperationCompleteEvent and JxfsStatusEvent
  The same description as above applies here. This field gives the DS the possibility to report additional results for this specific operation complete or status change event.
- directIO method
  Via this method new functionality can be made available to the application. It is outlined in detail below.

2.12.2 Vendor specific functionality (directIO)

A vendor may want to add functionality to a device he is offering which is beyond the scope of J/XFS. If he wants to write a J/XFS compliant Device Service there should be a generic way to access this additional functionality for the banking application without sacrificing J/XFS compliance.
This is most easily achieved by using the pre-defined directIO method call.

Assume that there is no device type which allows Text in- and output in J/XFS. Then, as an example, assume a vendor of a banking printer has added an LCD panel to it where some information can be displayed to the customer.
When writing the J/XFS Device Service for its printer the vendor has to implement all the defined properties, methods and events for the printer device service. He decides that some default texts are displayed on his panel during these operations.
Then, he implements the directIO method to allow an application to control the LCD. In his document he states that to show something on the LCD the application can use a call to

directIO with a command-parameter of ACME_PRT_SHOW_LCD[3] and a data object (subclassed from JxfsType) which contains the message.
A banking application which knows of this printer can now use the directIO call to show specific messages on the printer's LCD. Other printers would simply ignore the given command by returning JXFS_E_NOT_SUPPORTED (which is the default behavior for this call). Via the getDeviceServiceDescription() method the application can find out whether or not this is the printer with the LCD panel.
If the new functionality is for data input, then a call to the directIO method would only activate the possibility of receiving data and return an operationID and later, when the data arrives, either a JxfsIntermediate or JxfsOperationComplete event (or both) are sent which the application can receive (if it has registered successfully beforehand). It contains the operationId so the application knows that these events belong to the previously issued directIO command.

2.13 Power Management

In order to save energy and therefore costs many electronic and electric devices support power save modes. These modes operate in two kind of ways. Either they are using operating modes that are consuming less energy or other resources and may work slower but respond at once at a new request or on the other hand they shut themselves or parts of themselves down to a power save mode where it takes some time to come back into business (certain response time).

If a financial device or the appropriate device service for this device is able to cope with the power save mode internally without performance impact to the application it is encouraged to do so.

But in some cases the application should be aware of the current power mode of a device to optimize the performance. Imagine a dispenser inside a ATM goes into a power save mode that takes 30 seconds to come back into operational mode. If now a transaction starts that involves the cash dispenser and the cash dispenser begins to wake up the moment it shall begin to work on bank notes then this will have a performance impact of 30 seconds to the transaction. But if the application begins to wake up the cash dispenser in the moment the customer enters his identification card, there is nearly no performance degradation for the transaction.

The way power management is achieved is highly device specific. J/XFS does not define a way to specify when and how to enter power save mode and the details on configuration of these features, this is left to the Device Service implementations.
There are, however, certain general methods which are defined here so the application is able to react. They are

- Allowing to query a device's power save capability – isPowerSaveModeSupported().
- Querying whether or not a device is in power save mode (and thus may take longer to finish with a following operation) – JxfsStatus, boolean property powerSave
- Actively waking the device up bringing it up to full functionality (Any requested operation to a device also implicitly wakes it up, this does not include any synchronous operation like e.g. obtaining the device status) – wakeUpFromPowerSave()
- Sending events whenever power save mode is entered or exited – JxfsStatusEvents JXFS_S_POWERSAVEON and JXFS_S_POWERSAVEOFF

For all these functions J/XFS has installed the mentioned mechanisms. Anything which serves these power management functions has 'Power save' in its name, thus making it easy to spot these things in the J/XFS architecture.
See the remaining chapters for details.

[3]The numeric value of these constants is left to the device service programmer. It must, however, have a value above JXFSDIRECTIO_OFFSET.

2.14 Updating Firmware in a device

Increasing numbers of modern banking peripherals allow a software-driven firmware update. As this is likely to become more and more of a commodity, it should be included in the J/XFS standard.
This is especially important as, via the remote access feature of J/XFS, it enables significant effort reductions for the service personnel as they need not be physically present to update the firmware.

Basically there are two different methods of updating the firmware: **automatic** and **manual**.
The automatic case occurs if the Device Service detects a need to update during normal startup (*open*) of a device. The exact behavior must be decided by the Device Service class. As no possibility exists to inform the user about the process, the automatic updating should not take longer than a couple of seconds, so the user will not think the device is non-responsive and perhaps turn it off during the update.
The manual update process is under control of the application (either a special J/XFS enabled program or part of a banking application) which could look like this:

The application has the possibility to
1. Check for availability of a new release of the firmware for a specific device in the repository.
2. Query the currently installed firmware name and release number from the Device Service.
3. Have a function to compare whether an update is needed or not (also provided by the Device Service).
4. Issue an update command to the Device Service.

It is the application's decision how the information is presented to the user and whether or not an automatic update takes place.
Functionality to enable the above mentioned requirements is embedded in the J/XFS infrastructure.

2.15 Naming conventions

The following rules are defined to ensure unique naming conventions within J/XFS. J/XFS itself and programs extending or implementing parts of it are requested to obey to these rules.
Interfaces:
- Interfaces start with IJxfs, e.g. IJxfsMSDControl.

Classes:
- Classes delivered as part of the standard have a leading “Jxfs” in their name, e.g. the JxfsDeviceManager, JxfsVersion.
- The Device Controls do not have a “Control” in their name, giving e.g. JxfsMSD as the implementation of the IJxfsMSDControl interface.

- Classes which implement a device in the service layer can have any name as long as it is specific enough that it does not 'use up' too much namespace. For instance, MagStripeReader.java would be a bad name for a specific Device Service. IBM477xMSD.java would be a correct naming.
- The subclasses of the JxfsOperationCompleteEvent are named OCyyyzzzzzEvent, where yyy is the three character device type identifier (PTR, CDR, etc) and zzzzz describes the event itself (1 or more (upper limit is implementation dependant) characters).

Constant definitions:

- The main constants interface file is called IJxfsConst.java. The previous JxfsConst.java class is now deprecated.
- The device type specific constants are named as their respective Control classnames, but with Const at the end, giving IJxfsMSDConst.java for the above example. The previous JxfsMSDConst.java (and Jxfs*DeviceName*Const.java in general), are now deprecated.
- The naming of all Identifiers must be JXFS_X_YYY_ZZZ where
  * JXFS is fixed,
  * the X identifies the use of the constant and may be one of these

JXFS_O_...	Constants used to identify an operation state (e.g. JXFS_O_PTR_EJECT). They are used to identify operation return codes in JxfsOperationCompleteEvents and also in JxfsIntermediateEvents.
JXFS_I_...	Constants used to identify the reason for a *JxfsIntermediateEvent*. If the same meaning is also used in the OC Events then the appropriate JXFS_O_... identifier should also be used in JxfsIntermediateEvents in order to avoid having multiple constants with the same meaning.
JXFS_S_...	Status change constant definitions.
JXFS_E_...	Constants to identify errors in Exceptions, JxfsOperationCompleteEvents and JxfsIntermediateEvents (if they report an error condition).

  * the YYY is the 3 digit device type identifier like PTR or MSD,
  * the ZZZ is the specific code (and may of course be 1 or more (upper limit is implementation dependant) characters).

For the basic definitions of the above constants see "J/XFS constant codes" at section 5.4. Any class or interface, especially those visible to the application (ex: Device Controls) should obey the JavaBeans naming convention.

2.16 Return values

Data which is returned to the user and which may be modified after it is created must be deep copied by the device service before it is returned. Any data returned by a method only reflects the state of the object at the time of the call.

Data that is sent from the user to the device service which can be modified after creation must be deep copied by the device service during the synchronous part of the method.

2.17 Security and Encryption

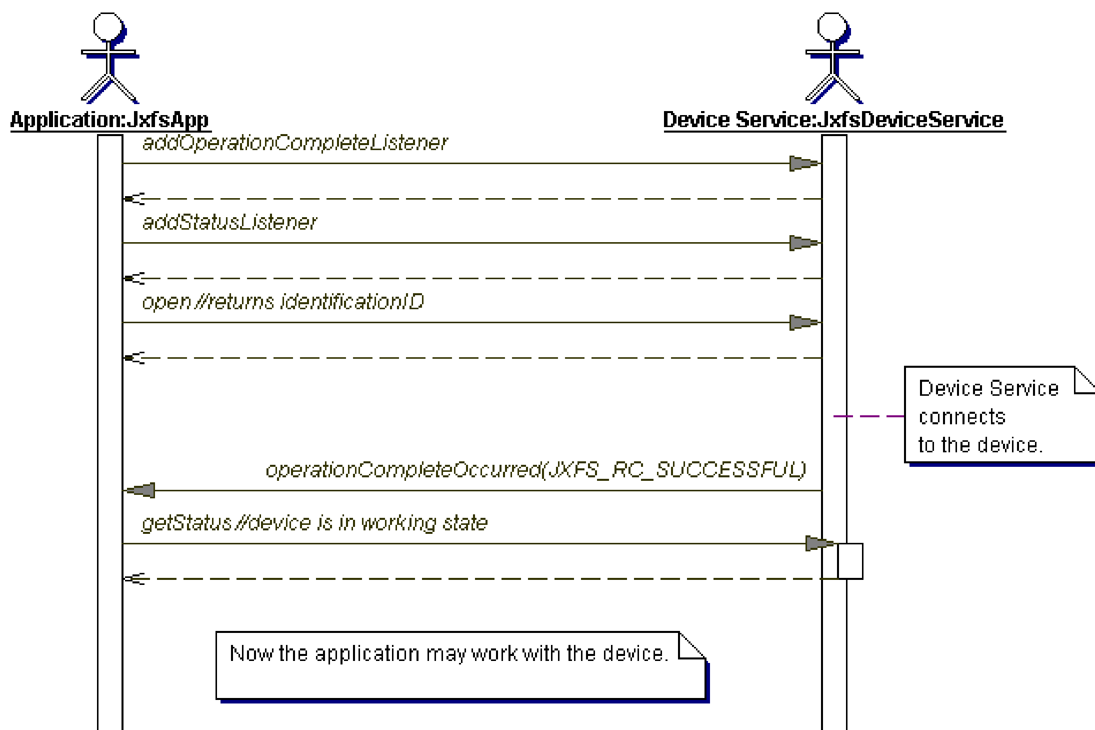
The access control for the device (i.e. the authorization to access a specific function in the J/XFS API) has to be controlled by the calling application and the network software. In the current CWA 12345 no support for login and user rights administration is supported.

Also, it may be desirable to encrypt all data which is send over the LAN between the workstations and the server, as well as between the peer-workstations sharing a device using J/XFS.
This is, however, also not a task defined in the CWA 12345. It is rather left to the TCP/IP installation and add-on security products to ensure that the data transfer is secure.
We assume that a solution to this is or will be available for use without the necessity to change the J/XFS structure. One possible option here would be to use RMI over SSL.

2.18 Handling of open() errors

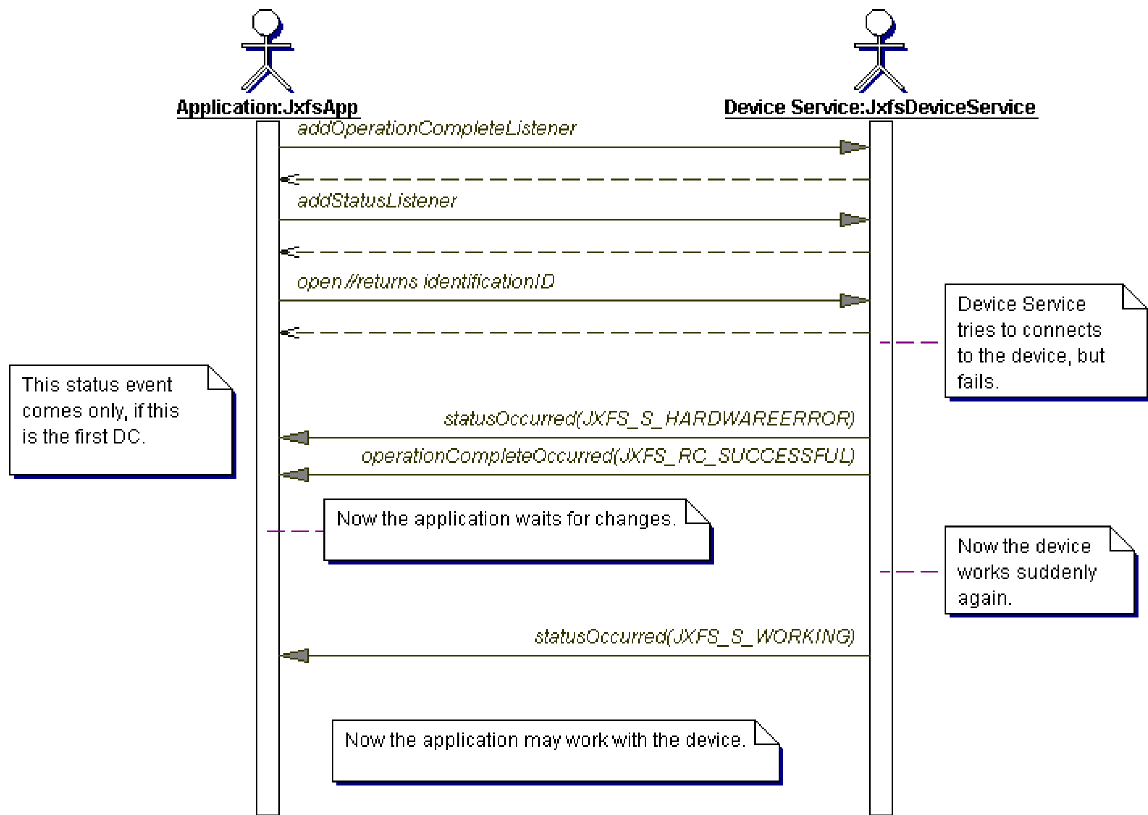
The J/XFS API is designed as a primarily asynchronous interface. Therefore an application should never be forced to poll. If an *open*() call fails due to some reason an application must not be responsible to continue calling *open*() until it may not fail. Therefore, an *open*() should only fail, if the error reason is so fatal that the device server, device and/or system cannot recover by itself. Examples could be the failure to load some necessary classes or the absence of a not hot-pluggable hardware interface.

Successful (normal) *open* call:



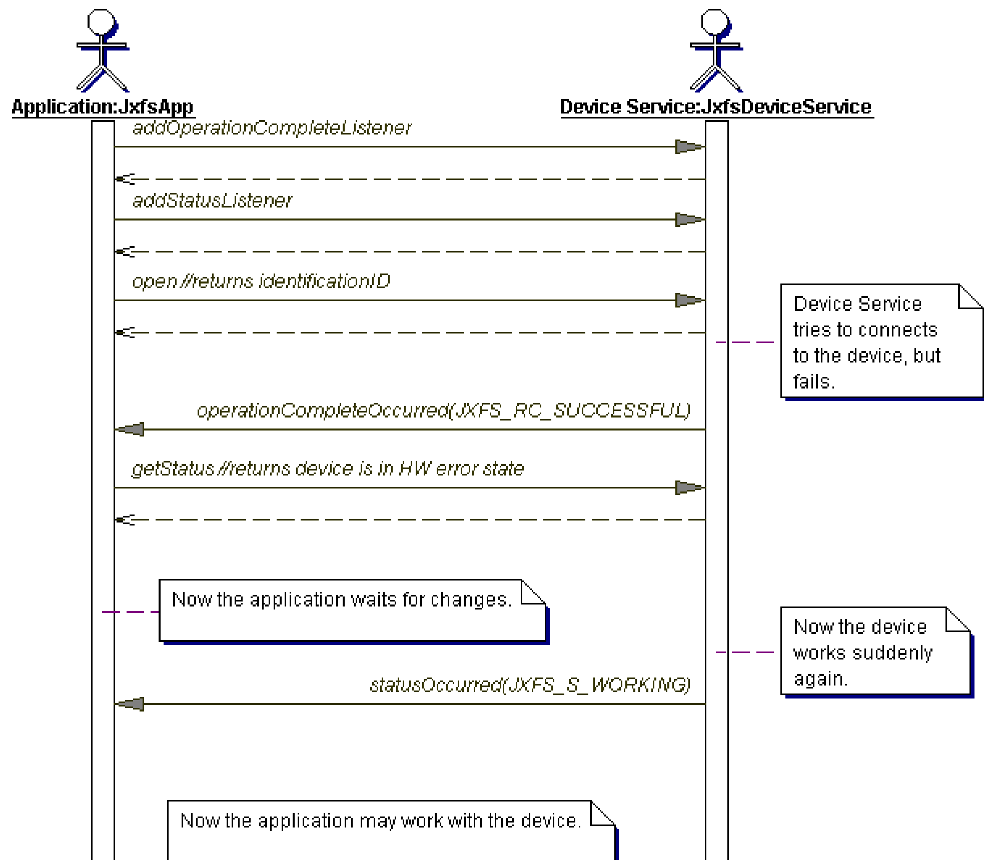
Simple errors like some communication problems with a device or a (temporary) disconnection of a communication line should result in a successful *open*() call. In such a case the device service shall go into a hardware or user error state before the *open*() call completes. Now the application may wait by listening to status events until the device is in a usable state again (JXFS_S_WORKING). This also implies that the device service is continuously working on solving the problem or at least checking for it's existence until either the problem is solved or the device is closed or the device service shut down.

open call with non-fatal error:

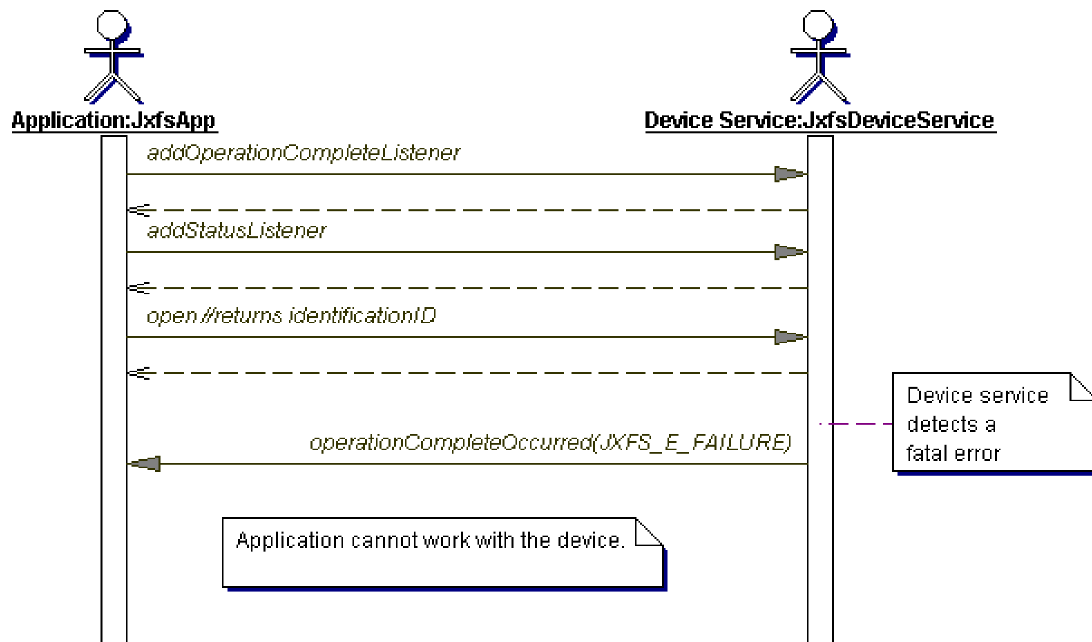


But this example assumes two things that an application should never rely on. This is that it is the first and only DC (only in this case a status event is defined as it changes from the "not open" into the "working" state) and that the status event will be delivered before the JxfsOperationCompleteEvent.

For a general handling an application always has to call getStatus() after a successful *open*() call to decide the next activity as shown in the following diagram.



open call with fatal error:



2.19 The Enum Pattern

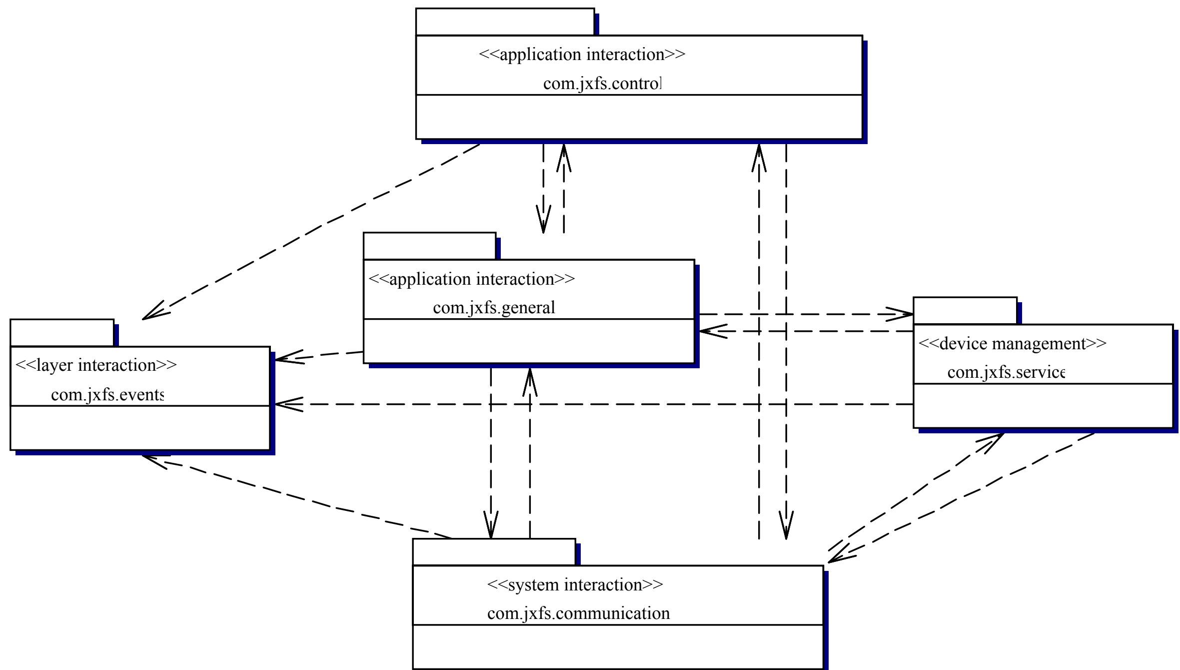
Newly defined Device Class Interfaces will handle constants using the Enum Pattern approach. J/XFS provides Enum classes for the constants defined in these Device Class Interfaces. These classes must cover the following requirements:

- They must run under the JVM version supported by J/XFS. In the future, when 1.5 version of the JVM will be adopted by J/XFS, the J2SE standard Enum implementation will be used by Device Class Interfaces instead of the J/XFS one. Applications and device services using this class should be implemented in a way which enables a smooth migration to the native Enum pattern implementation.
- They must provide proper serialization. Deserializing to a valid Enum value must be guaranteed.
- Besides, the constants using this pattern will be:
- Type safe. No other values than the ones explicitly created will be allowed. As the classes that extend JxfsEnum are final, they can’t be subclassed. This fact and the private access to the constructor make that the only objects that will be ever created are the static objects the class creates the first time is referenced.
- Printable. In order to create a suitable String representation of the constants, the toString method will be overwritten and will use the String parameter of the constructor.
- Comparable. The instances of this classes will be ordered using an internal ordinal number. The implementation of the compareTo method will make use of this ordinal to compare instances and will return the difference between the 2 ordinals. If this difference is 0 both instances are the same, otherwise the sign of the difference will tell which instance was created before.

3 Main J/XFS components

3.1 J/XFS packages

J/XFS consists of a number of packages. Each software layer of the architecture is separated into such a package. Thus, we have a **control**, a **communication** and a **service** package. As events and exceptions are used in all layers they are put into a separate package **events**. In addition to this, the generically used classes are put into a separate package named **general.**
Thus, the top-level package decomposition is like this:



The lines represent dependencies between the different packages. The most important concept here is that the Device Service layer does not depend on the Device Control layer as only then can the chore of transferring data over a network be invisible to the Device Service as well as to the application.
In addition to the packages shown above (which make up the basic J/XFS infrastructure) additional packages with the service implementations of the hardware manufacturer are used.

To prevent package collision between vendor specific packages the following naming convention should be used:
Companies should use their reverse internet domain name in their package names e.g. com.company.package. Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or project name after the company name e.g. com.company.region.package. The source for this convention is http://java.sun.com/docs/books/tutorial/java/interpack/createpkgs.html. The following convention should be used for jar file naming when delivering code for runtimes:
DS_<vendor | product>.jar
Optionally version information might be appended as well.

The following table gives an overview of which classes are put into which package:

com.jxfs	The base for all J/XFS classes and interfaces
com.jxfs.general	This package contains objects used in more than one other package. These are:

	JxfsDeviceManager, Jxfs{Local|Remote}DeviceInformation, JxfsConfiguration, JxfsServer, JxfsVersion, JxfsLogger, JxfsType, JxfsStatus, IJxfsConst, JxfsEnum.
com.jxfs.control	The device specific access classes for application access. Contains all the control interface files (IJxfs...Control..).
com.jxfs.control. <devicename>	One of these sub-packages exists per device type. Currently, these are ALM, CAM, CDR, CHK, DEP, MSD, PIN, PTR, SIU, SCN, TIO and VDM. They contain all the device type specific classes which are the Control classes, the IJxfsXXXConst and the JxfsXXXStatus classes.
com.jxfs.service	The device specific service classes. It only contains all the interface files (IJxfs....) implemented by the Device Services; the classes implementing them are vendor specific.
com.jxfs. communication	Anything concerned with network communication.
com.jxfs.events	The JxfsEvent and its subclasses, all the Listener interfaces which must be implemented by the application and the IJxfsEventNotification class as well as the JxfsException.
<vendor specific>	The implementations of the Device Service for a specific device type is not included into one of the jxfs packages but left in a vendor specific package. The JxfsDeviceManager can be configured to find those classes.

3.2 JxfsDeviceManager

The JxfsDeviceManager (DM) is a singleton object where device requests are routed to. There is exactly one DM in each Java VM.
Its main duties are

- Keep lists of devices / services / communication connections.
- Handle service instantiation and connect Device Controls, Device Communication and Device Services
- Query and write configuration data. Queries and writing operations are handled in an atomic way to avoid concurrency problems.
- Shield Device Controls and Device Services from using a specific set of Java APIs for configuration lookup and object creation (e.g. JSD and JSL) to gain flexibility.
- Make controls and services simpler and more straightforward to program.
- Communicate with a server to request device information on both local and remote devices (making it transparent for the device layers).
- Install any necessary classes so that other workstations can remotely access the devices.
- Register the devices which should be accessible by remote applications at the JxfsServer.

Except for the initialization and finalization phase most applications will not need to use the DM very often. Access to device specific functionality is solely available through the respective Device Controls.
The DM is used by the application for the first Device Control generation and for special purposes such as getting lists of available devices.
The detailed description of the DM's interface follows.

getReference

Syntax ***static JxfsDeviceManager getReference()***

Description This call returns a reference to the DM in a Java VM. There is exactly one DM in each Java VM.

initialize

Syntax ***void initialize(String configurationParameters) throws JxfsException***

Description This method must be called by the application to initialize the DM, e.g.

```
JxfsDeviceManager.getReference().initialize(

"RMI,client1,srv(2006),backupsrv(2007)");
```

The parameter must be provided to the DM by the application to inform the DM about any parameter it needs to successfully initialize itself. This is dependent on the implementation of the J/XFS infrastructure (and is defined there), but usually this should be similar to the outline given above.
This parameter is specific for EACH Java VM containing a J/XFS client application. This means that every application must read this configurationParameter from an administrator - changeable location.
The string in the sample above contains the unique name of this J/XFS client (client1), the communications method used to contact the server (RMI) as well as a list of hostname and ports where to find the server repository in a comma separated list.
As explained in the overview chapter more than one J/XFS client application can run on a single workstation. Every J/XFS client needs a unique identifier; the workstations hostname is not sufficient. The first element in the configurationParameters string is the configKey and is used to find the keys for this workstation in the repository.
If the initialization failed an exception is thrown which specifies the reason.

Exceptions

JXFS_E_REMOTE	Communication error during remote call.
JXFS_E_PARAMETER_INVALID	Error in the parameter: There was a problem connecting to the specified server(s).
JXFS_E_NOEXIST	Error in the parameter: The server was found

	but the identification (configKey) given is unknown.
JXFS_E_ILLEGAL	The DM is already initialized, i.e. this call was already done.

getDeviceList

Syntax *Vector getDeviceList(int level)*

Description Returns a Vector of JxfsDeviceInformation objects which represent the information about devices available to this J/XFS client. Level specifies in more detail which devices are reported and can be one of the following constants (Also defined in the DeviceManager):

JXFS_LEVEL_ACTIVE	Active devices in the same process as the application (application local). A device is active if its Device Service is successfully started.
JXFS_LEVEL_ CONFIGURED	Configured devices in the same process as the application (not necessarily active).
JXFS_LEVEL_ WORKSTATION	The same as JXFS_LEVEL_ACTIVE plus all active devices in other processes on this workstation
JXFS_LEVEL_ALL	JXFS_LEVEL_WORKSTATION and active devices on any other workstation as well.

getDeviceListFor

Syntax *Vector getDeviceListFor(Class control_classname, int level)*

Description Returns a list of JxfsDeviceInformation objects which represent the available devices of the given type (identified by the given control classname, e.g. JxfsTIOControl). The level is the same as in the above call.

getDevice

Syntax *IJxfsBaseControl getDevice(String logical_name) throws JxfsException;*

Description With this method the application requests a device. The logical_name given here can be queried from a JxfsDeviceInformation object with the method getDeviceName().
The returned Device Control object has a connection to its Device Service, but the device is not opened yet. If the application stops using the device it must be closed (if it has opened it) and then deregistered with *deregisterDevice*().

Exceptions

JXFS_E_NOSERVICE	devicename/service class unknown or not found
JXFS_E_NOEXIST	logical_name unknown
JXFS_E_FAILURE	service class failed to initialize.

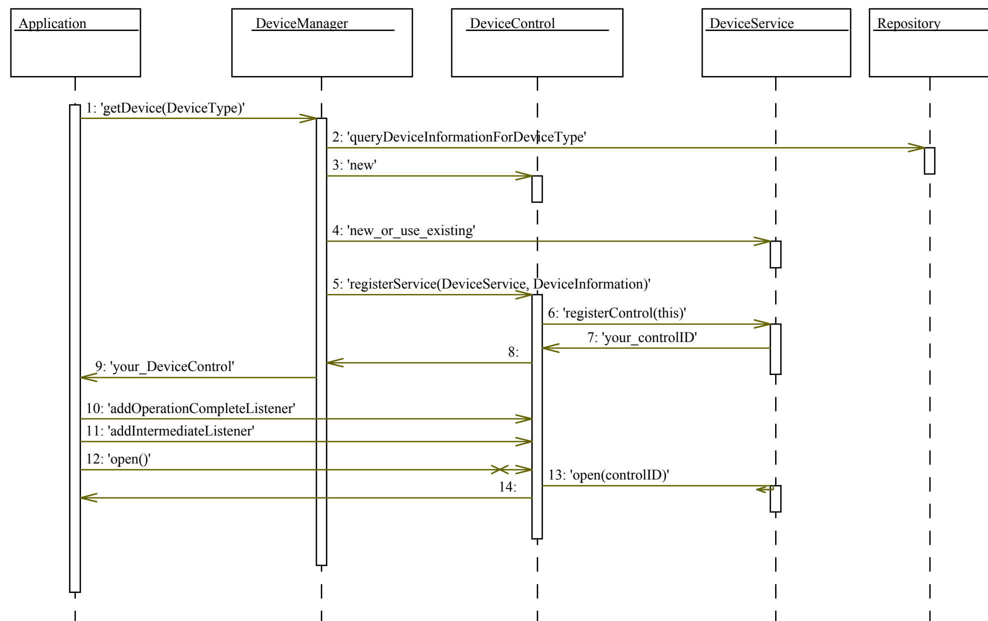
The application sample code to check for a returned class would be:

```
IJxfsBaseControl b=JxfsDeviceManager.getReference().getDevice("MSD1");
// Sanity check: Did I really get a JxfsMagStripe object?

if (b instanceOf JxfsMagStripe)) {
       JxfsMagStripe msr = (JxfsMagStripe)b;
       // do something with the device
  }
```

If the application wants to use a device, it must call the Device Manager's getDevice() method. If no error is thrown then this returns a valid reference to a Device Control of the requested type.

What happens internally during such a request? First, the DM checks if the requested device is attached locally. If so, it connects the corresponding DS (or implicitly starts it if it is not yet started), generates a new DC and returns it to the user.

The usual scenario is depicted in the graphic (where also a following *open* is shown):



getDevice

Syntax ***IJxfsBaseControl getDevice(Class control_classname) throws JxfsException;***

Description Here, the application requests a device of type control_classname without specifying a concrete name. The DM should return a Device Control for the default device of this type for this J/XFS client or - if no default is configured - to the first such device found. If the application stops using the device it must be released and then deregistered with *deregisterDevice*().

Exceptions

JXFS_E_NOSERVICE	devicename/service class unknown or not found
JXFS_E_NOEXIST	logical_name unknown
JXFS_E_FAILURE	service class failed to initialize.

The application sample code to use this would be:

```
JxfsMSD msr=(JxfsMSD)JxfsDeviceManager.getReference()
                     .getDevice(JxfsMSD);
// do something with the device
```

getValueForKey

Syntax *Serializable getValueForKey(String key) throws JxfsException*

Description This method allows an arbitrary object to be retrieved under the given key from the repository. It must be either a basic Java data type (String, int, etc.) or a subclass of JxfsType.
If the key is not found in the repository an exception with JXFS_E_NOEXIST is thrown.
Any String value may be used as a key name.

setValueForKey

Syntax ***void setValueForKey(String key, Serializable value) throws JxfsException***

Description Saves the given object persistently in the repository using the given key. If the key does not exist, it is created, if it exists, the value is replaced.
To remove a key from the repository, use this method and specify null as the value parameter.
An exception JXFS_E_ILLEGAL is thrown if the key specified is not allowed. This can e.g. happen if a read-only key with the same name exists which cannot be overwritten. Any String value may be used as a key name.

addKeyValueChangeListener

Syntax ***void addKeyValueChangeListener(IJxfsKeyValueChangeListener l, String key) throws JxfsException***

Description If the application or a Device Control or Device Service want to be informed about changes that happen to the value of a certain key, they must use this method to indicate where the change information should be delivered and what key it is interested in.
They have to implement the *IJxfsKeyValueChangeListener* interface. This contains only the method
void keyValueChangeOccurred(String key, Serializable value);
It is called by the Device Manager after registering. Also, the second parameter provides the new value for the key.

removeKeyValueChangeListener

Syntax ***void removeKeyValueChangeListener(KeyValueChangeListener l) throws JxfsException***

Description Remove the given KeyValueChangeListener object from the listening list.

getDeviceManagerVersion

Syntax ***JxfsVersion getDeviceManagerVersion()***

Description Return the version object for this Device Manager.

addStatusListener
removeStatusListener

Syntax ***boolean addStatusListener(IJxfsStatusListener l)***
boolean removeStatusListener(IJxfsStatusListener l)

Description With these methods the application can register as a listener to receive the JxfsStatusEvents from the DeviceManager (Returning *true* if the listener was successfully added or removed).
The DeviceManager informs of general things which are happening in the J/XFS infrastructure using Status events.
Currently, JxfsStatusEvents with the following Ids are defined:

JXFS_S_SHUTDOWN	A shutdown was received by the DeviceManager (see below)
JXFS_S_ REMOTEFAILURE	Communication is broken down.
JXFS_S_SERVICE_ STOPPED	A running Device Service was stopped. In the details parameter the logical name of the DS is given.
JXFS_S_SERVICE_ STARTED	A stopped Device Service was started. In the details parameter the logical name of the DS is given.

start

Syntax ***void start(String logical_name) throws JxfsException;***

Description Start the Device Service identified by the given name. The logical_name given here can be queried from a DeviceInformation object with the method getDeviceName().
If the device is already started this method immediately returns.
A start is called by the DeviceManager during initialization phase and is also implicitly done during a *getDevice* call on the local machine. So, this method is reserved for administration purposes.
This method only works for devices attached locally to this DM (i.e. where the DS is running in the same JavaVM as the DM).

Exceptions JXFS_E_NOSERVICE devicename/service class unknown or not found

JXFS_E_NOEXIST	logical_name unknown
JXFS_E_FAILURE	service class failed to initialize.

stop

Syntax ***void stop(String logical_name) throws JxfsException;***

Description Stop the device with the given name. The logical_name given here can be queried from a DeviceInformation object with the method getDeviceName(). The Device Manager does the following: First, the DeviceManager removes any entries for this device from the central J/XFS server. Then it calls the shutdown method of all the local Device Service objects. The Device Services in turn shut down the physical device, write any remaining persistent data into the repository and send a *JxfsStatusEvent* (JXFS_S_SHUTDOWN) to any remaining connected Device Controls, which have to notice the status change and deactivate any device access (See the Device Service method description for details).
This method fails if the given name is unknown by throwing a JxfsException with code JXFS_E_NOEXIST or if the DS was not started.
This method is reserved for administration purposes; the DM uses this method to shut down any DS during a shutdown.

shutdown

Syntax ***void shutdown();***

Description Prepare the shutdown of the J/XFS infrastructure. The Device Manager calls the above stop() method for all local devices. Finally it deactivates the logger by calling its shutdown() method, disables itself and returns control to the application.

generateUniqueTag

Syntax ***String generateUniqueTag() throws JxfsException;***

Description This method generates a unique tag on each call across all workstations managed by the Device Manager. The format and mechanism for generating this tag will be dependent on each implementation of the Device Manager and **no assumptions on format should be made by application developers**. However, one possible format that may be used by Device Manager implementers to generate the unique tag is the following

<workstation_id>_<timestamp>_<counter>

workstation_id: a string that uniquely identifies the workstation which contains the requesting application
timestamp: a string representation of the time at which this request was received by the Device Manager
counter: a counter which is incremented on each call to generateUniqueTag received by the Device Manager. This counter may be initialized to 1 on first call after the Device Manager is started.
This helper method is useful when used in conjunction with the form of the claim method which accepts a tag parameter, when accessing interdependent devices. For more information, see the claim method description of the IJxfsBaseControl interface.

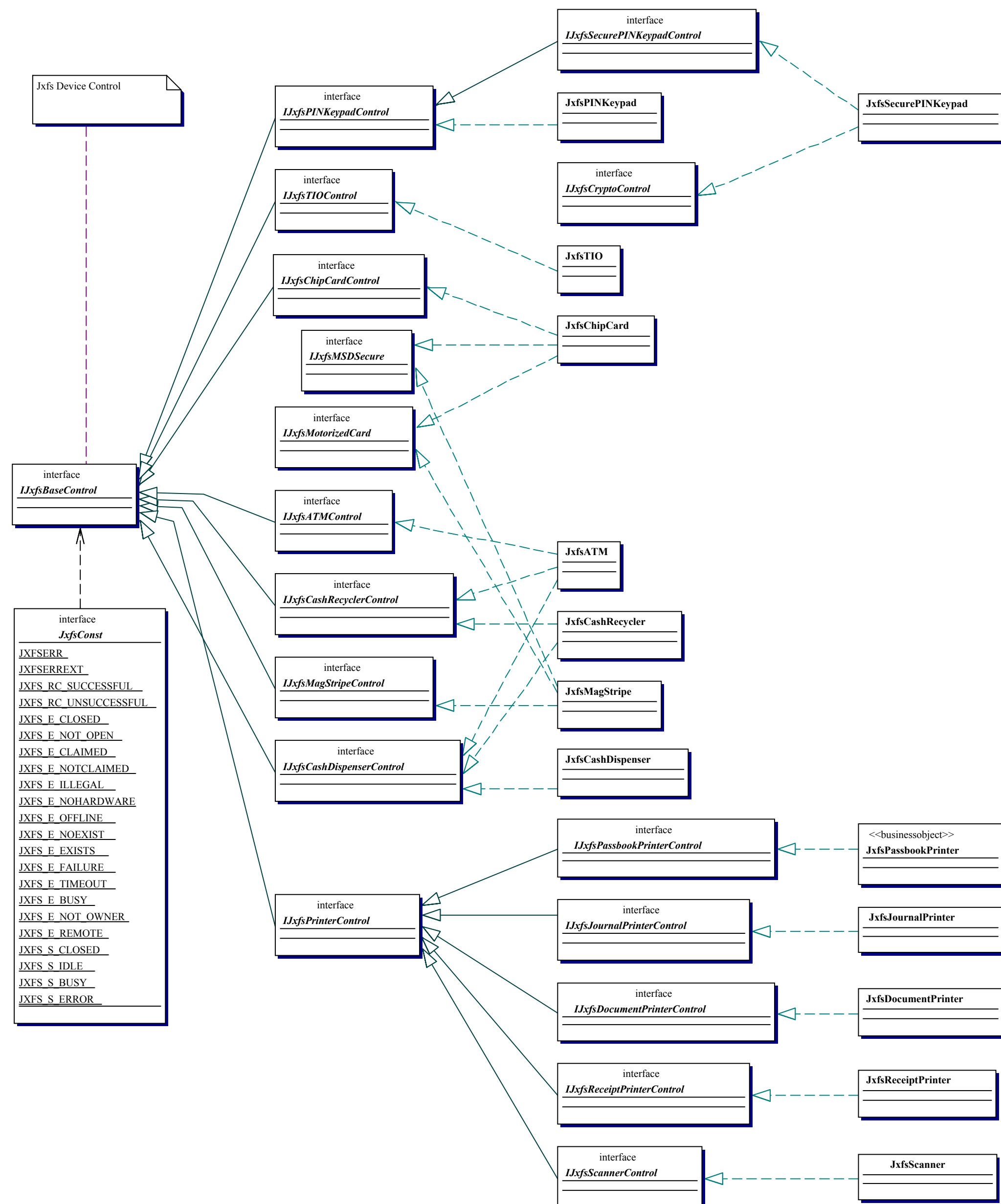
3.3 Device Control

The interfaces and methods in the Device Control are the tools the application uses to gain access to a financial device. It consists of the interface hierarchy for the supported devices and their implementation.
It is recommended that the finalizer of a Device Control should check if the Device Control has still: a) claimed b) opened and c) registered the Device Service. If needed and possible the Device Control finalizer should a) release b) close and c) deregister the control.

3.3.1 Object model

Depicted here are only the device classes, i.e. the interfaces and classes corresponding to the different device types which are supported.
As displayed in the picture below there is a common base interface for all the controls, IJxfsBaseControl. The different types of devices are reflected by having different subinterfaces, e.g. IJxfsPrinterControl. If the devices of this type have more subtypes, then an additional layer of interfaces is provided (i.e. IJxfsPassbookPrinterControl, IJxfsJournalPrinterControl etc.).

In the following only the IJxfsBaseControl interface is described in detail; all the subclasses of this interface are described in the respective device class documentation.



There are a number of classes missing from the class diagram as the diagram is meant to be illustrative only and not comprehensive.

3.3.2 IJxfsBaseControl

All methods in the IJxfsBaseControl are synchronous, unless it is explicitly stated otherwise.
There is no general timeout control available for asynchronous methods. Unless a method specifically makes it possible for the user to specify a timeout value, a timeout will not occur. All asynchronous methods can be cancelled.
Any method which is not listed as being able to throw the JXFS_E_CLOSED exception can be executed before the *open* method.
The JXFS_E_REMOTE exceptions will only ever be thrown by device comm objects (i.e. the communications sub-system) and never directly by a device service or control.

Public methods

The methods all Device Controls must support and which define the basic device behavior are:

open

Syntax *int open() throws JxfsException;*

Description This method must be the first method an application calls in a newly generated Device Control in order to use all other functions. Exceptions are the methods addXXXListener, getStatus()and getStatus(java.util.List). A call to any other method will throw a JxfsException with code JXFS_CLOSED.
This is the first time the device is physically accessed. It is asynchronous[4] and returns an identificationID. After the open has completed a JxfsOperationCompleteEvent with operationID = JXFS_O_OPEN and the given identificationID is sent to the application. The result is either JXFS_RC_SUCCESSFUL or one of the error codes.
After the open operation has been issued (but even before the OC Event has arrived) any other method is callable. Operation requests are queued for execution. If the open request fails, all queued requests will not be executed but responded with an operation complete event with an error code of JXFS_E_CLOSED.
But, of course the correct behavior for an application is rather to wait for the OCEvent of the open() and only then start using the device.
The open() must only fail for severe, unrecoverable errors (these kind of errors are implementation dependent). Minor defects should be noticed by the DS but the open() should succeed. For details on this please also see the device specifications detailing more on the correct open() behaviour.
Even if the open() fails, Status events are generated to inform the application that the Device status has changed. The application may then re-try to open the device.

Exceptions

JXFS_E_UNREGISTERED	Device is not registered at the Device Manager. It must either have been instantiated using new(), which is not allowed, or has already been deregistered at the Device Manager which disables this control completely.
JXFS_E_OPEN	Device is already opened. The application should not rely on JXFS_E_OPEN being thrown at once if the device is open as it depends on the internal scheduling mechanism of the Device Service if the Device Service will accept an open call when the device is open, but a *close* is already pending. Therefore the application should also be prepared to accept the JXFS_E_OPEN code as an OC event code and not only as an exception.
JXFS_E_REMOTE	Communication error during remote call.

close

Syntax *int close() throws JxfsException;*

Description Finishes the usage of the device by the application. If this is the last connected Device Control to issue close, this method disables further use of the device and releases any resources currently in use. All properties are reset to their initial default state.
The device must be released before close() is called.
This method is asynchronous and returns an identificationID. After that it returns a JxfsOperationCompleteEvent with operationID = JXFS_O_CLOSE. the given identificationID and a result (most likely JXFS_RC_SUCCESSFUL).

Exceptions

JXFS_E_UNREGISTERED	Device is not registered at the DM.
JXFS_E_CLOSED	Device has not been opened yet. The application should not rely on JXFS_E_CLOSED being thrown at once if the device is closed as it

[4] The reason that open and close are defined as asynchronous methods is that they also access the device. The generic J/XFS rule for device access requires asynchronous behavior (see section 2.5).

	depends on the internal scheduling mechanism of the Device Service if the Device Service will accept a close call when the device is closed, but an *open* is already pending. Therefore the application should also be prepared to accept the JXFS_E_CLOSED code as an OC event code and not only as an exception.
JXFS_E_REMOTE	Communication error during remote call.

claim

Syntax ***boolean claim(int timeout) throws JxfsException;***

Description This method attempts for the time specified by timeout (specified in milliseconds) to gain exclusive access to the device. This method returns control to the application when the claim is granted or when the timeout expires.
The Device Control will generate a unique tag (via the Device Manager *generateUniqueTag* method) for use with the Device Service claim method. This will ensure that only this Device Control can claim the Device Service, if successful.
This behaviour means that when this call is made by an application, no other Device Control in this application, or any other application, can share access to the device[5].
A claim is granted if no other Device Control has claimed the device and only after all pending operations are finished. All the operation requests are queued. As soon as a claim request is granted at the Device Service any operation requests from the DC holding the claim are the only ones which are processed. Operations from other DC's are queued until after the release is done.
If timeout is equal to JXFS_FOREVER (-1) then the operation waits as long as needed for the device to become available.
The return value is equal to *true* if claim() succeeds. The return value is equal to FALSE if claim() has timed out.
An application should release the claimed device as soon as possible.

Exceptions

JXFS_E_UNREGISTERED	Device is not registered at the DM.
JXFS_E_CLAIMED	Device is already claimed by caller.
JXFS_E_CLOSED	The *open* call has not been issued yet.
JXFS_E_REMOTE	Communication error during remote call.
JXFS_E_PARAMETER_INVALID	Invalid value for timeout parameter.

claim

Syntax ***boolean claim(String tag, int timeout) throws JxfsException;***

Description This method attempts for the time specified by timeout (specified in milliseconds) to gain exclusive access to the device. This method returns control to the application when the claim is granted or when the timeout expires.
If successful, the tag parameter will be used to allow subsequent claims from other Device Controls within the same application to also claim the device. It is the application's responsibility to create a sufficiently unique tag, typically by using the Device Manager *generateUniqueTag* method to create a unique tag.
A claim is granted if the device is not claimed by a Device Control with a different tag (or no tag) and only after all pending operations are finished. All the operation requests are queued. As soon as a claim request is granted at the Device Service any operation requests from the Device Control(s) holding the claim are the only ones which are processed. Operations from other DCs are queued until after the release is done.
If timeout is equal to JXFS_FOREVER (-1) then the operation waits as long as needed for the device to become available.

---

[5] Note that if another Device Control requests a claim on a device which is interdependent on this one, it will fail, as the tag generated for this subsequent claim will be different. In such cases, the claim method which accepts a tag should be used. See the claim(String tag, int timeout) method description for more information.

The return value is equal to *true* if claim() succeeds. The return value is equal to FALSE if claim() has timed out.
An application should release the claimed device as soon as possible.

Exceptions

JXFS_E_UNREGISTERED	Device is not registered at the DM.
JXFS_E_CLAIMED	Device is already claimed by caller.
JXFS_E_CLOSED	The *open* call has not been issued yet.
JXFS_E_REMOTE	Communication error during remote call.
JXFS_E_PARAMETER_INVALID	Invalid value for timeout parameter.
JXFS_E_NOT_SUPPORTED	This method is not supported.

release

Syntax ***boolean release(int timeout) throws JxfsException;***

Description Removes exclusive access to the device for this Device Control.
This will wait for all running asynchronous operations from the claiming DC to finish, but only up to “timeout” milliseconds. Then the queued operations from other DCs are executed. If another *claim()* requests arrives at execution position it will be granted. This method returns *true* if the release was successful, and FALSE if device operations are still pending after “timeout” milliseconds. In that case the release was NOT done and the DC has to re-issue this command (possibly after canceling an operation which ‘hangs’).
If other Device Controls have a valid claim when this call is made then release only removes this Device Control from having a claim against the device. All other current claims remain in effect. In this circumstance this call will wait for all running asynchronous operations from this Device Control to finish, but only up to “timeout” milliseconds.

Exceptions

JXFS_E_UNREGISTERED	Device is not registered at the DM.
JXFS_E_NOTCLAIMED	Device was not claimed by caller.
JXFS_E_CLOSED	The *open* call has not been issued yet.
JXFS_E_REMOTE	Communication error during remote call.
JXFS_E_PARAMETER_INVALID	Invalid value for timeout parameter.

cancel

Syntax ***void cancel(int identificationID) throws JxfsException;***

Description This method attempts to stop the operation specified by the identificationID. If it can do so, a JxfsOperationCompleteEvent is sent which indicates that the operation was cancelled. If this method is called after an operation has completed but before the operation complete event has been returned to the caller, then no operation will take place and no exception will be thrown; eventually the operation complete event will still be returned.
The identificationID is returned by the Device Control to the application by any asynchronous operation request. If an invalid Id is presented here, or the operation with this Id has already finished, no action takes place.

Exceptions

JXFS_E_UNREGISTERED	Device is not registered at the DM.
JXFS_E_CLOSED	The *open* call has not been issued yet.
JXFS_E_REMOTE	Communication error during remote call.

getStatus

Syntax ***JxfsStatus getStatus() throws JxfsException;***

Description This method returns a JxfsStatus object which contains the current status of the J/XFS device. This method may be called before the control has been opened. In this case whether the status returned is accurate or fake is implementation dependant.
The following is deprecated in favour of the status selector approach:
Every device can return a device specific JxfsStatus object that extends the JxfsStatus (e.g. JxfsPrinterStatus etc.). For detailed information see the separate

chapter on the JxfsStatus object.Usually, it is not a JxfsStatus object returned here but one of its subclasses, depending on which device type is queried.

Exceptions JXFS_E_UNREGISTERED — Device is not registered at the DM.
JXFS_E_REMOTE — Communication error during remote call.

getStatus

Syntax ***java.util.Map getStatus(java.util.List) throws JxfsException;***

Description This method returns different combinations of status information in one call. The status information to be returned is defined by the List of Status Selector Enums. Each of these Enums represent the status object to be returned. To retrieve the whole status information in one call an empty list should be passed. The returned java.util.map contains the Status Selector Enum as keys and the correspondant status object as value.
For detailed information see separate chapter on *JxfsStatus*.

Exceptions JXFS_E_UNREGISTERED — Device is not registered at the DM.
JXFS_E_REMOTE — Communication error during remote call.

addIntermediateListener
addOperationCompleteListener
addStatusListener

Syntax ***boolean addIntermediateListener(IJxfsIntermediateListener l);***
boolean addOperationCompleteListener(IJxfsOperationCompleteListener l);
boolean addStatusListener(IJxfsStatusListener l);

Description These methods are used by the application to register as a listener to receive the given type of events. They return *true* if the listener was registered successfully.

removeIntermediateListener
removeOperationCompleteListener
removeStatusListener

Syntax ***boolean removeIntermediateListener(IJxfsIntermediateListener l);***
boolean removeOperationCompleteListener(IJxfsOperationCompleteListener l);
boolean removeStatusListener(IJxfsStatusListener l);

Description These methods are used by the application to deregister itself as a listener of the given type of events. They return *true* if the listener was removed.

getDeviceName

Syntax ***String getDeviceName();***

Description Get the unique device name for this device (Type and distinction between similar devices, e.g. port name). Used to identify the device. This is the logical name key given in the configuration repository.

getDeviceControlVersion

Syntax ***JxfsVersion getDeviceControlVersion();***

Description Return the version object of this Device Control. See the chapter on versioning for a detailed explanation.

getDeviceServiceVersion

Syntax ***JxfsVersion getDeviceServiceVersion() throws JxfsException;***

Description Return the version object of this Device Service. See the chapter on versioning for detailed explanation.

Exceptions JXFS_E_UNREGISTERED — Device is not registered at the DM.

JXFS_E_REMOTE	Communication error during remote call.

getPhysicalDeviceDescription

Syntax	***String getPhysicalDeviceDescription() throws JxfsException;***	
Description	The physical device description, e.g. “Acme Magstripe Reader Model 36 subtype 5 (c) 1997 Acme corp.”	
Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device has not been opened yet
	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_REMOTE	Communication error during remote call

getPhysicalDeviceName

Syntax	***String getPhysicalDeviceName() throws JxfsException;***	
Description	The physical device’s name, e.g. “Acme MSD 36/5”	
Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device has not been opened yet
	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_REMOTE	Communication error during remote call

updateFirmware

Syntax	***boolean updateFirmware() throws JxfsException;***	
Description	Asynchronous function to trigger a firmware update. Returns *true* if the update process could be started. Delivers a JxfsOperationCompleteEvent with operationID = JXFS_O_UPDATEFIRMWARE and a result when finished. The identificationID is not used because only one update can be active at any time.	
Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device has not been opened yet
	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_FIRMWARE	Nothing to update / available firmware does not match.
	JXFS_E_NOT_SUPPORTED	Operation not supported by this device.
	JXFS_E_REMOTE	Communication error during remote call

getFirmwareStatus

Syntax	***int getFirmwareStatus() throws JxfsException;***	
Description	Checks the firmware in the device against the one found in the repository and return:	
Result	OK_NEWER	Firmware in repository is newer than firmware in device.
	OK_OLDER	Firmware in repository is older (!) than firmware in device. Update possible (but not recommended).
	OK_EQUAL	Firmware in the repository is equal to the firmware in the device. Update possible.
	OK_OTHER[6]	Firmware in repository has a different functionality, but an update is possible.
	NO_SOURCE	Update not possible, no firmware found in repository.
	NO_MATCH	Update not possible, firmware in repository not correct for this device.

---

[6]What's the difference between version and functionality?
It could be the same firmware, but another version, i.e. it is the firmware for a chip card reader with German ZKA standards, but a newer version. Or it could be a firmware with other functionality, i.e. the firmware for a French chip card shall be loaded, but the chip card reader currently contains the firmware for the German ZKA standard.

	NO_SUPPORT	No firmware update possibility with this device.
Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device is closed.
	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_REMOTE	Communication error during remote call.

getDeviceFirmwareVersion

Syntax ***JxfsVersion getDeviceFirmwareVersion() throws JxfsException;***

Description Returns a JxfsVersion object informing about the loaded version of the firmware in the device.

Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device has not been opened yet
	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_NOT_SUPPORTED	Operation not supported by this device.
	JXFS_E_REMOTE	Communication error during remote call

getRepositoryFirmwareVersion

Syntax ***JxfsVersion getRepositoryFirmwareVersion() throws JxfsException;***

Description Returns a JxfsVersion object informing about the available version of the firmware in the repository.

Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device has not been opened yet
	JXFS_E_NOT_SUPPORTED	Operation not supported by this device.
	JXFS_E_REMOTE	Communication error during remote call

isPowerSaveModeSupported

Syntax ***boolean isPowerSaveModeSupported() throws JxfsException***

Description Returns *true* if the attached device is capable of going to and returning from a power save mode.

Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device has not been opened yet
	JXFS_E_REMOTE	Communication error during remote call

wakeUpFromPowerSave

Syntax ***int wakeUpFromPowerSave() throws JxfsException;***

Description This method can be used by the application to actively request that the device becomes active again. It initiates the wakeup (if needed) and returns immediately. The int that is returned specifies the average time in seconds needed to get back to an active state (or –1 if n/a). If the device is powered up again a JxfsStatusEvent with JXFS_S_POWERSAVEOFF is sent.

Exceptions	JXFS_E_UNREGISTERED	Device is not registered at the DM.
	JXFS_E_CLOSED	Device has not been opened yet
	JXFS_E_REMOTE	Communication error during remote call

directIO

Syntax ***int directIO(int command, JxfsType serializable) throws JxfsException;***

Description This method gives an application the means to access device specific functions which only apply to a specific physical device. The application can check for the availability of the special hardware e.g. via the getDeviceServiceDescription() method.

As the device may reside on another machine, the subclass of JxfsType containing the data must be serializable.

The service itself can either synchronously work on the command and return immediately or work asynchronously and notify the application via the usual intermediate and operation complete events (with special codes) during and after

completion.
The default behavior of any services not having additional commands is to throw a JXFS_E_NOT_SUPPORTED exception.
For an exact description of how to use this method see the explanation in the chapter on 'Vendor specific functionality'.

Exceptions

JXFS_E_UNREGISTERED	Device is not registered at the DM.
JXFS_E_CLOSED	Device has not been opened yet
JXFS_E_NOHARDWARE	Device is not connected to the workstation.
JXFS_E_CLAIMED	This method is not available at this time because the device is currently claimed for exclusive access by another control.
JXFS_E_NOT_SUPPORTED	Operation not supported by this device.
JXFS_E_REMOTE	Communication error during remote call

deregisterDevice

Syntax ***void deregisterDevice() throws JxfsException;***

Description This method must be used by the application to inform the Device Control that will no longer be used.
This allows the DC to remove the connection to the Device Service and free up any allocated resources.
Any method of the DC which needs to access the Device Service from now on only returns a JXFS_E_UNREGISTERED exception.

Exceptions

JXFS_E_OPEN	Device is still open.
JXFS_E_REMOTE	Communication error during remote call

isInterDependentDevice

Syntax ***boolean isInterDependentDevice() throws JxfsException;***

Description This method informs the application whether the associated Device Control represents a device which is part of an inter-dependent device.
A device is inter-dependent if either of the following circumstances apply:

- If two or more devices share the same physical component. For example a journal & passbook printer may share the same print head.
- If two or more logical devices are mapped onto one physical device. For example, a printer that serves as both as document & passbook printer.

This method returns *true* if the device is inter-dependent, FALSE otherwise.

getInterDependentDevices

Syntax ***Vector getInterDependentDevices() throws JxfsException;***

Description This method returns a vector of strings which contains the names of all devices that make up this interdependent device. The name of each device returned is the logical name as described in the DeviceInformation object *getDeviceName* method.
Note that *getInterDependentDevices* returns the names of all devices, including the name of the device associated with the calling/called Device Control.

Exceptions

JXFS_E_ILLEGAL	Device is not part of an interdependent device.

Internal methods

There are additional methods necessary to successfully connect the Device Control to its Device Service and the Device Manager. They are only for J/XFS internal use and are only briefly outlined here. These methods must not be called by applications although this is not enforced.

registerService

Syntax ***void registerService(IJxfsBaseService myService, JxfsDeviceInformation di)***

throws JxfsException;

Description This method is used by the Device Manager to initialize the Device Control by providing a reference to the service object itself. The Device Manager also provides the appropriate Device Information for the DC's use.

connectionFailure

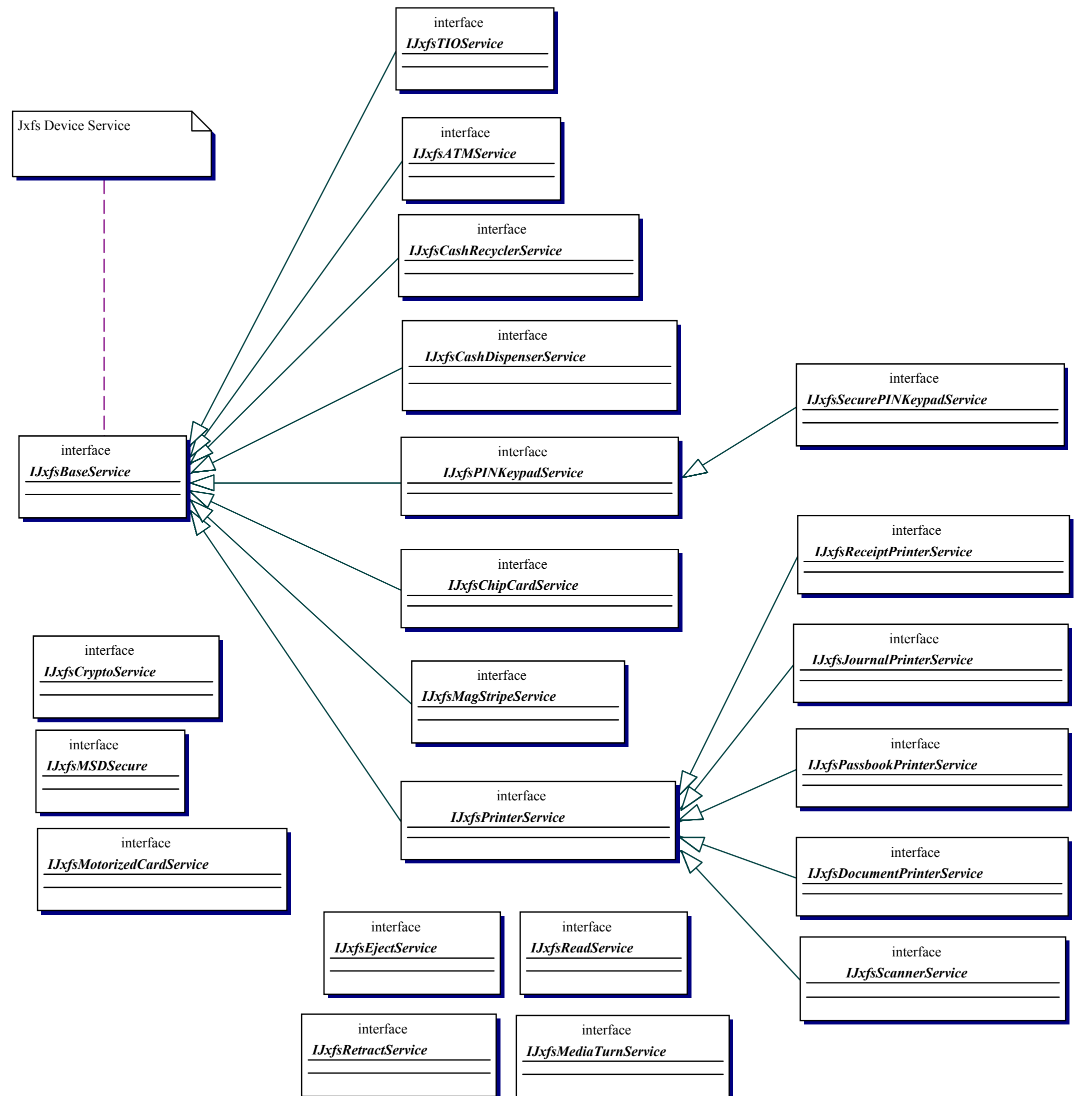
Syntax ***void connectionFailure();***

Description This method is called by any communication layer to inform the Device Control that the connection to the DS is broken.
The Device Control now has to approve this fact by using its deregisterService() functionality and must send the application a JxfsStatusEvent with JXFS_S_REMOTEFAILURE.
The exact details of the connection failure are written to the logger by the communication layer.

3.4 Device Service

In the Device Service layer the interfaces for the hardware vendor's (HV) device drivers are defined. In order to be compliant an HV must implement the interface.
This can be done in 100% pure Java. The use of 100% pure Java is not mandatory. Note that the CommAPI is not yet available for all platforms, and there are some services that may use other interfaces to access the device. One alternative could be JNI, the Java Native Interface. For each Device Control class there is a corresponding Device Service interface which has to be implemented in order to access the physical hardware.

3.4.1 Object model



The diagram depicts the inheritance tree of the device service interface classes. Because the diagram is meant to be illustrative only and not comprehensive there are a number of classes missing from the class diagram. Also shown are some of the available supplemental interfaces to add special functionality (e.g. the support for motorized card interface for the MagStripe and ChipCard devices). These are the ones which do not inherit from the IJxfsBaseService interface.
All the methods in the control interfaces are reflected in the methods of the service classes.
The Device Service class must synchronize access to it from multiple Device Controls and guarantee that after a successful claim any operation the Device Control that got the claim does is sequential and precedes any other arriving requests.
In order to support control identification for event delivery more easily, an additional parameter (int control_id) is passed into every method as the last parameter. The Device

Control gets this identifier from the Device Service after registering there. If the device is claimed by a control the service class can thus lock out any other accessing control.
In the case of complex devices (which are devices that implement more than one J/XFS device type in a single service), an additional *claim()* will only succeed if it is claimed with the same tag value as the one used when claiming the device control which now holds the claim. See the definition of claim below and in the Device Control description.

3.4.2 IJxfsBaseService

All methods in the IJxfsBaseService interface are synchronous, unless it is explicitly stated otherwise.
There is no general timeout control available for asynchronous methods. Unless a method specifically makes it possible for the user to specify a timeout value, a timeout will not occur. All asynchronous methods can be cancelled.
Any method which is not listed as being able to throw the JXFS_E_CLOSED exception can be executed before the *open* method.
The JXFS_E_REMOTE exceptions will only ever be thrown by device comm objects (i.e. the communications sub-system) and never directly by a device service or control.
If a device service is called with any method before the initialize method, or after the initialize method fails, then the device service should return JXFS_E_FAILURE.

The methods all services must support and which define the basic device behavior are:

open

Syntax ***int open(int control_id) throws JxfsException;***

Description This method must be the first method a control (identified by control_id) calls in a newly generated Device Control in order to use all other functions. Exceptions are the addXXXListener methods, getStatus() and getStatus(java.util.List). The device service is required to obtain and cache device status before an open is issued; although precisely when this occurs is an implementation detail. A call to another methods throws a JxfsException with code JXFS_CLOSED.
This is the first time the device is physically accessed. It is asynchronous[7] and returns an identificationID. After the open completed a JxfsOperationCompleteEvent with operationID = JXFS_O_OPEN and the given identificationID is sent to the application. The result is either JXFS_RC_SUCCESSFUL or one of the error codes.
After the open operation has been issued (but even before the OC Event has arrived) any other method is callable. Operation requests are queued for execution, and if the open fails, they are discarded.
But, of course the correct behavior for an application is rather to wait for the OCEvent of the open() and only then start using the device.
If the OC event returns success the device is connected to the workstation and device status is correct.
The open() must only fail for severe, unrecoverable errors (this kind of errors is implementation dependent). Minor defects should be noticed by the DS but the open() should succeed. For details on this please also see the device specifications detailing more on the correct open() behaviour.
Even if the open() fails, Status events are generated to inform the application that the Device status has changed. The application may then re-try to open the device.

Exceptions

JXFS_E_OPEN	Device is already opened.
JXFS_E_REMOTE	Communication error during remote call.

[7] The reason that open and close are defined as asynchronous methods is that they also access the device. The generic J/XFS rule for device access requires asynchronous behavior (see section 1.2, “Basic Operation Principles”).

close

Syntax	***int close(int control_id) throws JxfsException;***
Description	This method closes the device for the DC's usage. If no other control is using it (i.e. there is no other Control that has issued an *open*() call), then the device is also physically closed (i.e. shutdown or deactivated). This method is asynchronous and returns an identificationID. After that it returns a JxfsOperationCompleteEvent with operationID = JXFS_O_CLOSE. the given identificationID and a result (most likely JXFS_RC_SUCCESSFUL).
Exceptions	JXFS_E_CLAIMED — Device is still claimed. JXFS_E_CLOSED — Device is already closed. JXFS_E_REMOTE — Communication error during remote call.

claim

Syntax	***boolean claim(int timeout, int control_id) throws JxfsException;***
Description	Tries to claim the device for exclusive use. See the explanation on claim in the Device Control chapter as well as the section on 'Reserving devices for exclusive use'. Claim() returns *true*, if performed successfully or FALSE if not. This method is now deprecated and will never be invoked by the infrastructure. It is replaced by the claim method below.
Exceptions	JXFS_E_CLAIMED — Device is still claimed. JXFS_E_CLOSED — Device is already closed. JXFS_E_REMOTE — Communication error during remote call. JXFS_E_PARAMETER_INVALID — Invalid value for timeout parameter.

claim

Syntax	***boolean claim(String tag, int timeout, int control_id) throws JxfsException;***
Description	Tries to claim the device for exclusive use. See the explanation on claim in the Device Control chapter as well as the section on 'Reserving devices for exclusive use'. Claim() returns *true*, if performed successfully or FALSE if not. If the timeout parameter is set to 0 then *true* will be returned only if the device is currently unclaimed. If a timeout value less than –1 (JXFS_FOREVER) is specified the results are implementation dependant.
Exceptions	JXFS_E_CLAIMED — Device is still claimed. JXFS_E_CLOSED — Device is already closed. JXFS_E_REMOTE — Communication error during remote call. JXFS_E_PARAMETER_INVALID — Invalid value for timeout parameter. JXFS_E_NOT_SUPPORTED — This method is not supported.

release

Syntax	***boolean release(int timeout, int control_id) throws JxfsException;***
Description	Assuming that the calling Device Control has a current claim (Device Service keeps track of all Device Controls with a claim), this call removes access to the device for the calling Device Control. If this is the last 'claiming' Device Control to release its claim then exclusive access is removed. Other Device Controls (using another unique string tag) may now successfully claim the device. It also causes the queue of waiting claim() requests to be checked and will result in the longest waiting request to be granted. The control_id identifies the control. This method returns to the application when the operation is complete. If timeout occurs, e.g. an operation is still pending, FALSE is returned.
Exceptions	JXFS_E_NOTCLAIMED — Device was not claimed by caller. JXFS_E_CLOSED — Device is already closed. JXFS_E_REMOTE — Communication error during remote call. JXFS_E_PARAMETER_INVALID — Invalid value for timeout parameter.

cancel

Syntax	***void cancel(int identificationID, int control_id) throws JxfsException;***
Description	This method attempts to restore the device and its service back to the conditions before the operation was called, that has to be cancelled. An attempt will be made to stop the operation specified by the identificationID and to cancel any corresponding events that have not yet been reported to registered listeners. This method will try its best to cancel the specified operation. Even if there is no corresponding operation for the identificationID or the operation can not be cancelled, no exception will be thrown. If cancel() ends with success a JxfsOperationCompleteEvent will be sent. The control_id identifies the control.
Exceptions	JXFS_E_NOHARDWARE Device is not connected to the workstation. JXFS_E_CLOSED Device is already closed. JXFS_E_REMOTE Communication error during remote call.

getStatus

Syntax	***JxfsStatus getStatus(int control_id) throws JxfsException;***
Description	This method returns a JxfsStatus object that reports the current status of the J/XFS device. When this method is called before an open, whether the status returned is accurate or fake is implementation dependant. Every device usually returns a device specific JxfsStatus object that extends the JxfsStatus (e.g. JxfsPrinterStatus etc.).
Exceptions	JXFS_E_REMOTE Communication error during remote call.

getDeviceServiceVersion

Syntax	***JxfsVersion getDeviceServiceVersion(int control_id) throws JxfsException;***
Description	Returns the version information object of the Device Service.
Exceptions	JXFS_E_REMOTE Communication error during remote call.

getPhysicalDeviceDescription

Syntax	***String getPhysicalDeviceDescription(int control_id) throws JxfsException;***
Description	The physical device description, e.g., “Acme Magstripe Reader Model 36 subtype 5 (c) 1997 Acme corp.”
Exceptions	JXFS_E_NOHARDWARE Device is not connected to the workstation. JXFS_E_CLOSED Device is already closed. JXFS_E_REMOTE Communication error during remote call.

getPhysicalDeviceName

Syntax	***String getPhysicalDeviceName(int control_id) throws JxfsException;***
Description	The physical device’s name, e.g., “Acme MSD 36/5”
Exceptions	JXFS_E_NOHARDWARE Device is not connected to the workstation. JXFS_E_CLOSED Device is already closed. JXFS_E_REMOTE Communication error during remote call.

updateFirmware

Syntax	***boolean updateFirmware(int control_id) throws JxfsException;***
Description	Asynchronous function to trigger a firmware update. Returns *true* if the update process could be started. Delivers a JxfsOperationCompleteEvent with operationID = JXFS_O_UPDATEFIRMWARE and a result when finished. The identificationID is not used because only one update can be active at any time.
Exceptions	JXFS_E_NOHARDWARE Device is not connected to the workstation. JXFS_E_CLOSED Device is already closed.

JXFS_E_FIRMWARE	Nothing to update / available firmware does not match.
JXFS_E_NOT_SUPPORTED	Operation not supported by this device.
JXFS_E_REMOTE	Communication error during remote call.

getFirmwareStatus

Syntax	***int getFirmwareStatus(int control_id) throws JxfsException;***	
Description	Checks the firmware in the device against the one found in the repository.	
Result	OK_NEWER	Firmware in repository is newer than firmware in device.
	OK_OLDER	Firmware in repository is older (!) than firmware in device. Update possible (but not recommended).
	OK_EQUAL	Firmware in the repository is equal to the firmware in the device. Update possible.
	OK_OTHER[8]	Firmware in repository has a different functionality, but an update is possible.
	NO_SOURCE	Update not possible, no firmware found in repository.
	NO_MATCH	Update not possible, firmware in repository not correct for this device.
	NO_SUPPORT	No firmware update possibility with this device.
Exceptions	JXFS_E_CLOSED	Device is closed.
	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_REMOTE	Communication error during remote call.

getDeviceFirmwareVersion
getRepositoryFirmwareVersion

Syntax	***JxfsVersion getDeviceFirmwareVersion(int control_id) throws JxfsException; JxfsVersion getRepositoryFirmwareVersion(int control_id) throws JxfsException;***	
Description	Return JxfsVersion objects informing about the loaded and available Versions of the firmware in the device. If the operation is not supported the according exception is thrown.	
Exceptions	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_NOT_SUPPORTED	Operation not supported by this device.
	JXFS_E_CLOSED	Device is already closed.
	JXFS_E_REMOTE	Communication error during remote call.

isPowerSaveModeSupported

Syntax	***boolean isPowerSaveModeSupported(int control_id) throws JxfsException***	
Description	Returns *true* if the attached device is capable of going to and returning from a power save mode.	
Exceptions	JXFS_E_CLOSED	Device is already closed.
	JXFS_E_REMOTE	Communication error during remote call.

wakeUpFromPowerSave

Syntax ***int wakeUpFromPowerSave(int control_id) throws JxfsException***

[8] What's the difference between version and functionality?
It could be the same firmware, but another version, i.e. it is the firmware for a chip card reader with German ZKA standards, but a newer version. Or it could be a firmware with other functionality, i.e. the firmware for a French chip card shall be loaded, but the chip card reader currently contains the firmware for the German ZKA standard.

Description	This method can be used by the application to actively request that the device becomes active again. It initiates the wakeup (if needed) and returns immediately. The int that is returned specifies the average time in seconds needed to get back to an active state (or –1 if n/a). If the device is powered up again a JxfsStatusEvent with JXFS_S_POWERSAVEOFF is sent.	
Exceptions	JXFS_E_CLOSED	Device is already closed.
	JXFS_E_REMOTE	Communication error during remote call.

directIO

Syntax	***int directIO(int command, JxfsType serializable, int control_id) throws JxfsException;***	
Description	This method gives an application the means to access device specific functions which only apply to a specific physical device. For an exact description about the functionality see the description of the directIO in the Device Control chapter and the explanation in 'Vendor specific functionality'. The default behavior of any services not having additional commands is to totally ignore this method by throwing a JxfsException with the exception code JXFS_E_NOTSUPPORTED.	
Exceptions	JXFS_E_NOHARDWARE	Device is not connected to the workstation.
	JXFS_E_CLOSED	Device is closed.
	JXFS_E_CLAIMED	This method is not available at this time because the device is currently claimed for exclusive access by another control.
	JXFS_E_NOT_SUPPORTED	Operation not supported by this device.
	JXFS_E_REMOTE	Communication error during remote call.

There are a small number of additional methods which are used by the Device Manager to initialize the Device Service:

initialize

Syntax ***void initialize(JxfsLocalDeviceInformation your_info) throws JxfsException***

Description This method is used by the Device Manager to deliver the detailed device information to the service.

Exceptions JXFS_E_PARAMETER_INVALID The given parameter is invalid.

registerControl

Syntax ***int registerControl(String device_control_type, IJxfsEventNotification callbacks_implementing_control) throws JxfsException;***

Description This method must be the first method that is called by the Device Control in order to register for events and identify the control during all method calls. If a method on the service is called before the registerControl method is called, or after the deregisterControl method is called, then a JXFS_E_UNREGISTERED exception should be thrown. The Device Service must keep a list of all connected controls to deliver events and check the state of the registered controls (has opened the service, has claimed the service). As long as any useful information can be retrieved from the Device Service it should be allowed to start. So, e.g. even if no hardware device is attached the service should be started, but the status should then be JXFS_S_HARDWAREERROR, as it still may be desirable to issue some other options. In this case, during the following *open*() a JXFS_E_NOHARDWARE is returned. The return value is an unique id identifying the control registering for this service.

The EventCallback object given here may also be a Device Communication object and not the Device Control itself. The device_control_type parameter given here is the name of the Device Control interface class (i.e. "IJxfsALMControl") and allows a Device Service which implements a complex device to identify which 'part' of the device the DC wants to access[9].

The returned value is greater or equal than JXFS_VALID_CONTROLID = 2.

Exceptions

JXFS_E_PARAMETER_INVALID	A parameter was null or otherwise invalid.
JXFS_E_REMOTE	Communication error during the method call.
JXFS_E_EXISTS	Specified Control is already registered.

deregisterControl

Syntax ***void deregisterControl(int control_id) throws JxfsException;***

Description This method must be the last method a Device Control object (identified by control_id) calls in a device service to deregister for events.

Exceptions

JXFS_E_OPEN	Device is still opened.
JXFS_E_ REMOTE	Communication error during remote call.

connectionFailure

Syntax ***void connectionFailure(int control_id);***

Description This method is called by any communication layer to inform the Device Service that the connection to the Device Control is broken.

The Device Service now has to approve this fact by, if applicable, releasing, closing and deregistering on behalf of the Device Control.

The exact details of the connection failure are logged to the logger by the communication layer.

[9] We cannot use a direct reference to the Device Control class here because of the possibility of network transfers which hide the originating type of the class from the DS.

shutdown

Syntax ***void shutdown() throws JxfsException;***

Description This method is used by the Device Manager to deactivate a Device Service. It should be implemented by the DS in a way to guarantee that it always succeeds (shouldn't get stuck).
The DS ends the current job (if not possible it terminates it) and throws away all the pending jobs (without sending OC Events). Then it shuts down the physical device and writes any remaining persistent data into the repository.
Finally, it sends a JxfsStatusEvent with the status JXFS_S_SHUTDOWN to all registered Device Controls. They have to delete their reference to the Device Service and disable themselves (i.e. always return JXFS_E_UNREGISTERED out of any operation from now on). This event is then propagated to the application by the DC's.

Exceptions JXFS_E_REMOTE Communication error. Probably not all remote Device Controls could be informed.

isInterdependentDevice

Syntax ***boolean isInterdependentDevice() throws JxfsException;***

Description This method informs the Device Control whether the associated device represents a device which is part of an inter-dependent device. The Device Service returns *true* if the device is inter-dependent, FALSE otherwise.

Exceptions JXFS_E_OPEN Device is still opened.
JXFS_E_ REMOTE Communication error during remote call.

getInterdependentDevices

Syntax ***Vector getInterDependentDevices() throws JxfsException;***

Description On receiving this call, the Device Service builds and returns a vector of strings which contains the names of all component devices that make up this interdependent device. The name of each device returned is the logical_name as described in the DeviceInformation object getDeviceName method.

Exceptions JXFS_E_ILLEGAL Device is not part of an Interdependent device.

3.4.3 Complex Devices

The advocated way to implement complex devices is to have a single device service object that will create a delegated object for each sub-component. By doing this it is possible to have interface version protection. The *IJxfsComplexDeviceService* interface, defined in the *com.jxfs.service* package, enables the device manager to query a complex device service for a specific sub-component, and register it with the device control. The following methods are specified by this interface:

initialize

Syntax ***void initialize(JxfsLocalDeviceInformation[] your_infos) throws JxfsException***

Description This method is used by the Device Manager to supply the detailed device information to the complex device service. The your_infos array contains detailed information objects for all sub-components of the complex device service.

Exceptions JXFS_E_PARAMETER_INVALID The given parameter is invalid.

getSubComponent

Syntax ***IJxfsBaseService getSubComponent(String logical_name) throws JxfsException***

Description This method is used by the Device Manager to retrieve the delegate object which represents a specific sub-component of the complex device service. The

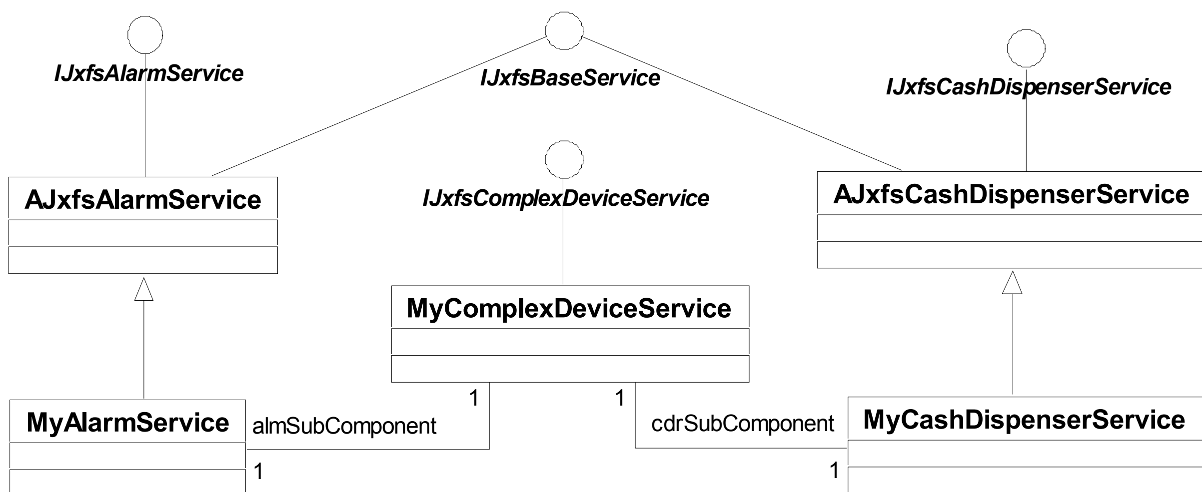
	logical_name parameter is the logical name of the device (e.g. "MSD1"). This name corresponds with the logical_name parameter in the *getDevice*() method of the IJxfsDeviceManager interface.	
Exceptions	JXFS_E_PARAMETER_INVALID	The given parameter is invalid. There is no sub-component with the given logical name.

A complex device service is required to implement the *IJxfsComplexDeviceService* interface only.

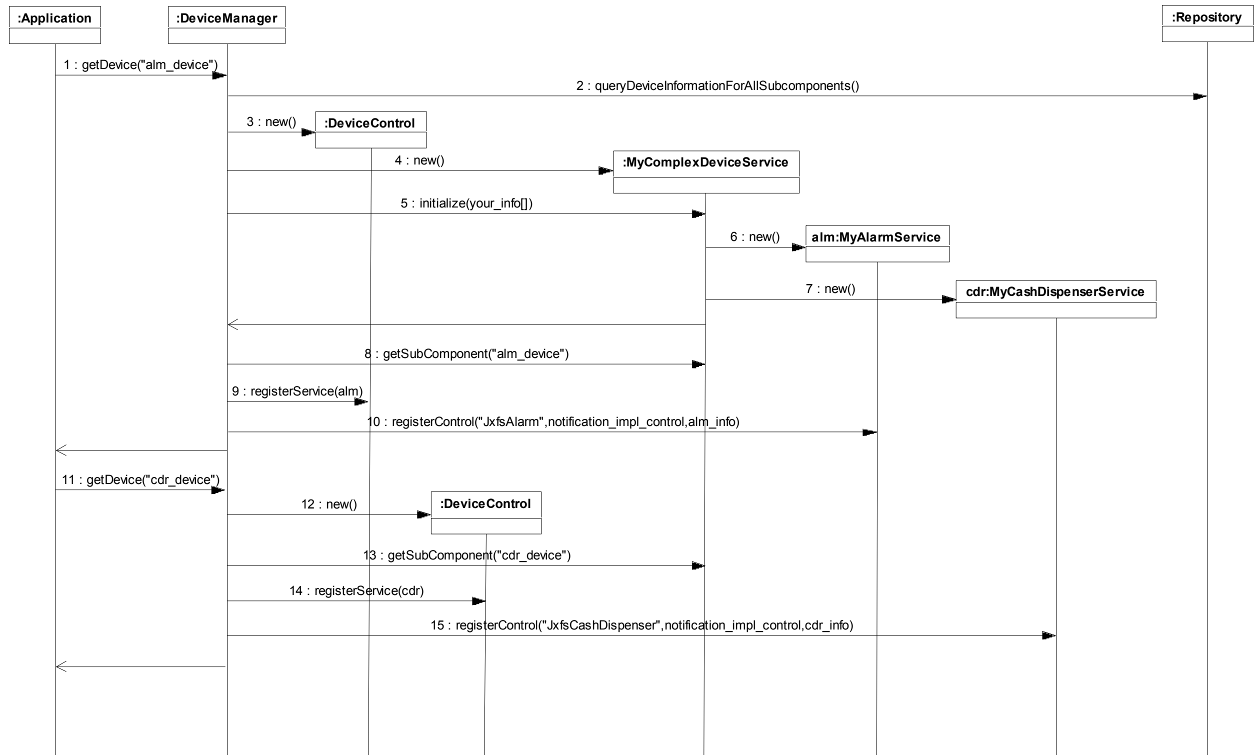
The following diagram illustrates this with a complex device that includes a cash dispenser and an alarm device. A cash dispenser device service class *MyCashDispenserService* is defined to extend the abstract *AJxfsCashDispenserService* class which implements the *IJxfsCashDispenserService* interface and is provided for the purpose of the interface version protection. Similarly, a *MyAlarmService* class extends the *AJxfsAlarmService* class, which protects the *IJxfsAlarmService* interface.

The complex device service *MyComplexDeviceService* is now able to instantiate one of each of these classes, which will represent the sub-components of the complex device. When the device manager queries the complex device for the "cash_dispenser" service subcomponent, it will return a reference to the *MyCashDispenserService* object, and likewise for the "alarm" service subcomponent.

The coordination between the two subcomponent instances of *MyAlarmService* and *MyCashDispenserService* classes is accomplished in a vendor-specific way. One possibility is to use the common complex device service object of the *MyComplexDeviceService* class.



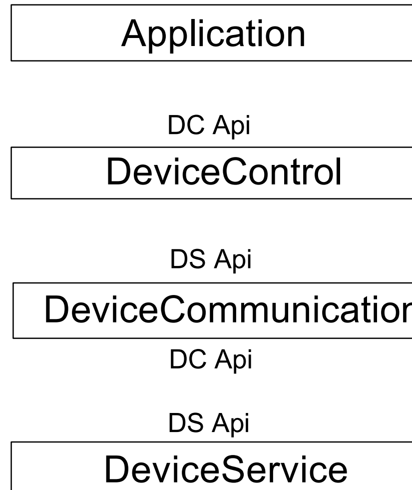
The following sequence diagram specifies how device manager creates and initializes a complex device service, retrieves the required delegate objects and finally associates them with their corresponding device controls as results of *getDevice*() calls for it's sub-components.



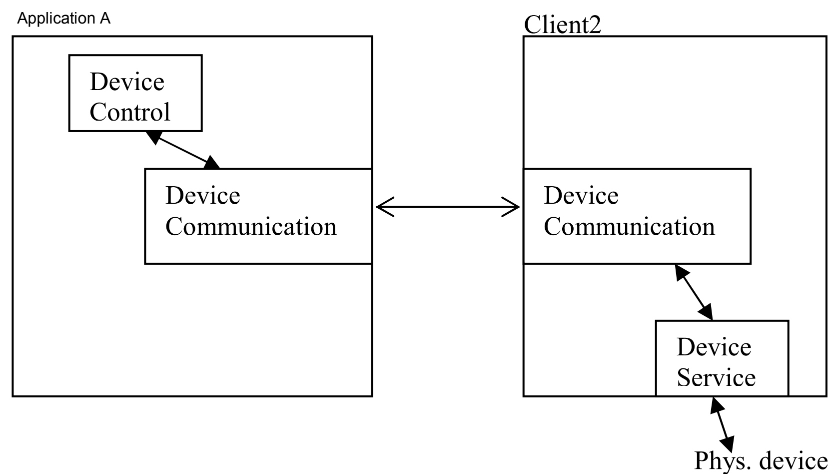
In order to provide the backward compatibility with complex device services conform to the older version of the J/XFS specification, the device manager should check whether the instantiated complex device service implements the *IJxfsComplexDeviceService* interface. If yes, the complex device service should be handled as described in this section. If not, the instantiated complex device service object is used directly, without being queried for it's subcomponents. Note that in the latter case the interface version protection can be provided for only one of device services represented by the complex device service.

3.5 Device Communication

The Device Communication package implements the peer-to-peer transportation layer to enable device sharing.
This layer provides a Device Service-like API to the top (i.e. to the Device Control) and a Device Control-like API to the bottom (i.e. to the local Device Service). Thus, it serves as an additional indirection layer to hide the network communication for the Control and Service objects.



It implements the Device Service interface for the Device Controls to use. For the Device Service it implements the same interface as the Device Controls do -- which is mainly the IJxfsEventNotification - interface.
For the simpler case of only local device access, the Device Communication layer may be omitted and the JxfsDeviceManager is reduced to an interface to a registry (JSD, a file or other available storage).



The above chart tries to give a short sketch of how the communication classes enable the sharing of devices across a network.

To the Device Control the Device Communication looks exactly like a Device Service, and to the Device Service on the other side of the connection the Device Communication looks exactly like a Device Control.

The existence of a network communication layer must not be known to the application. The J/XFS architecture, however, has some features to enable this communication layer.

Probably the most important feature is that all operations of the devices are designed to be independent of each other. This means that to issue an operation any relevant parameters are usually given in the same method call, there is no requirement that for a single operation multiple method calls are necessary. This minimizes the effort for error handling which needs to be done in a communication layer as well as in the application.

4 Exceptions and Events

J/XFS has several means to deliver information to the application: Return codes, exceptions or events.
Return codes are only used if very simple return information is presented to the application, generally if only one parameter is needed. As in most object-oriented designs, this parameter should not be misused to deliver information about errors (e.g. returning a String and defining that if it contains "ERROR" then an error occurred would be extremely bad practice).
In such a situation an **Exception** should be used. If for example a parameter of a method call is illegal (and this is detected very early in the method call) then a JxfsException with error code JXFS_E_PARAMETER_INVALID should be thrown.
Events are used by the asynchronous methods and can be sent at any time. The following duties are assigned to the events:

1. Notify the application of intermediate results during the running operation (e.g. sending the single keystrokes from a keyboard).
2. Notify the application of asynchronous operation completion (i.e. track read). This may be successful completion as well as abortion of the operation due to an error.
3. Inform the application of status changes (e.g. busy, offline) and special conditions (e.g. threshold values reached such as paper low)

To satisfy the above duties the following categories of events exist:

1. The *JxfsIntermediateEvent* (I) is sent whenever a meaningful intermediate result is available for the running asynchronous operation.
2. The *JxfsOperationCompleteEvent* (OC) is sent whenever an asynchronous operation is completed. The return code depends on whether the operation was successful, partially successful or a failure.
3. The *JxfsStatusEvent* (S) is sent whenever device status changes.

The application itself can decide whether or not it is interested in these event messages since it must specifically register to receive the events. It can register for each of the different event types.

As outlined in the chapter on object instantiation, multiple Device Controls can be connected to the same Device Service. The following rules apply regarding which event is sent to which Device Control:

1. The JxfsIntermediateEvent and JxfsOperationCompleteEvent are sent ONLY to the Device Control which has started the currently running operation.
2. Usually the JxfsStatusEvent is sent to all the connected Device Controls to inform the DCs of the generic Status changes a device goes through.

As a *common guideline* on when to use Exceptions vs. Events it can be said that, as Exceptions are a more direct way to inform the application of some error condition they are used in preference to Events.
But, as in J/XFS all the methods involving the device are asynchronous, there is, apart from instant parameter checking etc., no other way to inform the application than via events.

Please also note that currently there is no such feature as returning some interaction values back to the Device Service (such as continue / retry / cancel). As banking devices are very sensitive to the completion of an operation, an operation which encounters for example an out of paper condition is cancelled and must be re-issued by the application (after action like refilling the device with paper was taken).

There is intentionally no such event like an ErrorEvent. Any information the application needs in the case of an error is the error code. This is delivered via the appropriate JxfsOperationCompleteEvent. The details regarding the error are only of interest for a supervisor application which can gather this information from the Logger. See the chapter on “Tracing and error logging” on section 5.3 for details.

4.1 Exceptions

The standard exception within J/XFS is the JxfsException. It is thrown wherever an exception is needed and contains the following parameters (all except errorCode may be empty:

Parameter	Type	Meaning
errorCode	int	The error code. One of the defined JXFS_E_... codes.
errorCodeExtended	int	An extended error code. This can be a system or vendor dependent error code.
description	String	Textual description of the error.
origException	Exception	The original exception which was caught and replaced by this JxfsException. A standard sample would be that a RemoteException is caught and a new JxfsException with error code JXFS_E_REMOTE is created. Then the original exception is put into it – if the application wants to analyze the exception further it can get it from here.

Every method may return the following error codes even if not explicitly specified for that method either as an exception code or as an error codes from an asynchronous request:

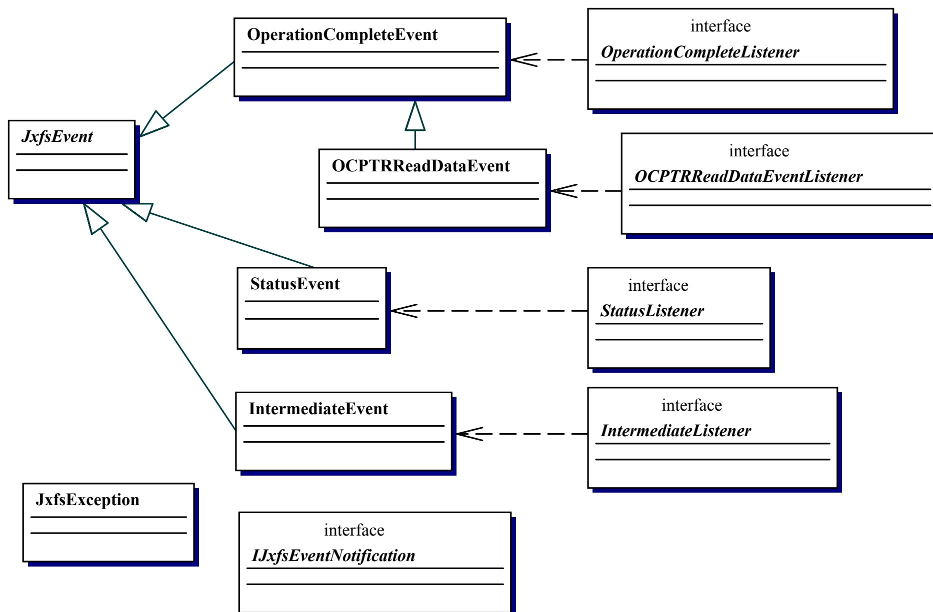
- JXFS_E_OPEN
- JXFS_E_CLOSE
- JXFS_E_DISABLED
- JXFS_E_ILLEGAL
- JXFS_E_NOHARDWARE
- JXFS_E_OFFLINE
- JXFS_E_NO_EXIST
- JXFS_E_EXISTS
- JXFS_E_FAILURE
- JXFS_E_TIMEOUT
- JXFS_E_BUSY
- JXFS_E_PARAMETER_INVALID
- JXFS_E_REMOTE
- JXFS_E_IO
- JXFS_E_CANCELLED
- JXFS_E_NOT_SUPPORTED
- JXFS_E_FIRMWARE
- JXFS_E_SYSTEM
- JXFS_E_UNREGISTERED
- JXFS_E_HARDWAREERROR

The code for JxfsException is shown below:

```
//////////////////////////////////////////////////////////////////////
//
// JxfsException
//
//   Exception class used to report all J/XFS errors.
//
//////////////////////////////////////////////////////////////////////
package com.jxfs.events;

public class JxfsException extends java.lang.Exception
{
   /* Some utility constructors to allow some parameters
      to be omitted. If description is not directly given
      it is filled with errorCode and errorCodeExtended. */
    public JxfsException(int errorCode)
    {
        this(errorCode, 0, "" + errorCode, null);
```

```
        }
        public JxfsException(int errorCode, int errorCodeExtended)
        {
            this(errorCode, errorCodeExtended,
                 "" + errorCode + ", " + errorCodeExtended, null);
        }
        public JxfsException(int errorCode, String description)
        {
            this(errorCode, 0, description, null);
        }
        public JxfsException(int errorCode, int errorCodeExtended,
            String description)
        {
            this(errorCode, errorCodeExtended, description, null);
        }
        public JxfsException(int errorCode, String description,
            Exception origException)
        {
            this(errorCode, 0, description, origException);
        }
        /* main constructor with all parameters */
        public JxfsException(int errorCode, int errorCodeExtended,
            String description, Exception origException)
        {
            super(description);
            this.errorCode = errorCode;
            this.errorCodeExtended = errorCodeExtended;
            this.origException = origException;
        }
        public int getErrorCode()
        {
            return errorCode;
        }
        public int getErrorCodeExtended()
        {
            return errorCodeExtended;
        }
        public Exception getOrigException()
        {
           return origException;
        }
        protected int errorCode;
        protected int errorCodeExtended;
        private Exception origException;
    }
```

4.2 Events



4.2.1 Event classes

All possible events are categorized and belong to one of the event classes outlined below. The common base class for any J/XFS event is the JxfsEvent.

JxfsEvent

This event class contains the generic variables for all the events. It extends java.util.EventObject. It is not used on its own but serves as a base class for the other J/XFS event types.
In the following table all relevant methods are outlined

Method	Return	Meaning
getWhen()	Date	Contains the date the event was first created. If the event comes from a remote machine then the date will be the time the event was created on the remote machine.
setSource(Object source)	void	Sets the source of this event. This is filled with the Device Service object (as the events are generated here), but is replaced by the Device Control object before it is passed to the application.
getSource()	IJxfsBaseService or IJxfsBaseControl	Returns the source of the event. (Inherited from the EventObject class).

The real working events are the following:

JxfsIntermediateEvent

An Intermediate Event is sent if intermediate results of an operation have to be sent to the application. This can either be to inform the application of some conditions specific to the operation (e.g. if a cash dispense command has to be delayed for a couple of minutes) or to deliver intermediate data (e.g. the keystrokes pressed by the user).

It is sent only to the Device Control which started the operation.

Method	Returns	Meaning
JxfsIntermediateEvent (Object source, int operationID, int identificationID, int reason)	-	Constructor for this Event. the ‘data’ variable is set to null.
JxfsIntermediateEvent (Object source, int operationID, int identificationID, int reason, Serializable object)	-	Constructor for this Event with complete parameters
getOperationID()	int	The id number for the operation type. One of the constant definitions showing which type of operation the event is related to.
getIdentificationID()	int	The id which was given by the operation method to the application, -1 if not used.
getReason()	int	Specifies what the reason for this event was (e.g. JXFS_I_CDR_DISPENSE_DELAYED).
getData()	Serializable	Contains the optionally added data for the application or null. If it is not a Java base data type then the object stored here should be a subclass of JxfsType.

Interface class:IJxfsIntermediateListener
Listener method: intermediateOccurred(JxfsIntermediateEvent e)

JxfsOperationCompleteEvent

A JxfsOperationCompleteEvent is always sent when a previously started operation terminates. It is sent only to the Device Control which started the operation.
It can either just inform if a successful completion of an output command (e.g. “printed”), returns the data of a requested input operation or gives information if the operation failed, perhaps returning a partial dataset.
The identificationID identifies the operation. The operationID identifies the type of the operation. The result code contains the result of the operation, the optional data object can deliver additional data.

The exact parameters of the event are defined in the Device Control definition of every device type.

Method	Returns	Meaning
JxfsOperationCompleteEvent (Object source, int operationID, int identificationID, int result)	-	Constructor for this Event. Variables data and extendedResult left empty.
JxfsOperationCompleteEvent (Object source, int operationID, int identificationID, int result, int extendedResult)	-	Constructor for this Event. Only data is not given here.
JxfsOperationCompleteEvent (Object source, int operationID, int identificationID, int result, Serializable data)	-	Constructor for this Event. Without the extendedResult.
JxfsOperationCompleteEvent (Object source, int operationID, int identificationID, int result, int extendedResult, Serializable data)	-	Constructor for this Event. Fills all parameters.
getOperationID()	int	The operationID which states what kind of operation is complete.
getIdentificationID()	int	The id which was given by the operation method to the application, -1 if not used.
getResult()	int	Specifies the operation result. It is JXFS_RC_SUCCESSFUL if everything was fine.
getExtendedResult()	int	An additional int specifying a result. This is a device specific value! Its usage by the application must only occur if the specific device service is determined. Also, by using it application will now be vendor dependent. Returns –1 if not used.
getData()	Serializable	Contains optionally added data for the application. If it is not a Java base data type then the object stored here should be a subclass of

		JxfsType.

Interface class: IJxfsOperationCompleteListener
Listener method: operationCompleteOccurred(JxfsOperationCompleteEvent e)

JxfsStatusEvent

A Status Event is sent if the status of the device has changed. The reason for this may either be a change due to an operation (such as “paper low”), or it may occur due to user interaction (such as “device made offline”).
This type of event is sent to ALL the connected Device Controls.

Method	Returns	Meaning
JxfsStatusEvent (Object source, int status)	-	Constructor for this Event.
JxfsStatusEvent (Object source, int status, JxfsType details)	-	Constructor for this Event – all parameters except result and extendedResult
JxfsStatusEvent (Object source, int status, JxfsType details, int result, int extendedResult)	-	Constructor for this Event – complete parameters
getStatus()	int	The status the affected device has changed to. See the list below for details on when these events are sent. Each specific Device type also adds some more status codes.
getDetails()	**JxfsType** or one of its subclasses	Indicates the detailed status conditions of the device. This is filled if the given status does not give the exact and complete status change information. An example here would be (for the printer) that a status of JXFS_S_PTR_TONER would be sent indicating something with the toner changed. The application could then query this details object for the exact condition (is it empty or low or ...). If a generic status change is reported (see section on JxfsStatus) then this field is empty (==null).
getResult()	int	Specifies additional information on the reason that originated the event. Notice that a status event may be issued even if the status itself did not change, but the result field did. Returns -1 if not used.
getExtendedResult()	int	Specifies extended device specific information. Its usage by the application must only occur if the specific device service is determined. Also, by using it application will now be vendor dependent. Notice that a status event may be issued even if the status itself did not change, but the extended result did. Returns -1 if not used.

Interface class: IJxfsStatusListener
Listener method: statusOccurred(JxfsStatusEvent e)

All the connected Device Controls (and thus the application or applet) are informed of certain changes of the device status. They receive a JxfsStatusEvent with the corresponding status and with a null details object in the following cases:

status	Meaning
JXFS_S_CLAIMED	Sent if the device was claimed.
JXFS_S_RELEASED	Sent if the device was just released by a Device Control which had claimed the device.
JXFS_S_HARDWAREERROR	Sent if a hardware error was detected by the Device Service.
JXFS_S_USERACTIONERROR	Sent if an error was detected which is resolvable by user intervention. If a more specific JxfsStatusEvent is generated regarding the error (e.g. a TONER_OUT) then NO additional event with this id is sent.
JXFS_S_WORKING	Sent if an error has been fixed to indicate that the device is working again.
JXFS_S_SHUTDOWN	The device service has completed its shutdown and is not usable any more.
JXFS_S_REMOTEFAILURE	The communication between DS and DC is broken; device is not accessible any more.
JXFS_S_POWERSAVEON	Device is gone into power save mode.
JXFS_S_POWERSAVEOFF	Device has returned from power save mode.
JXFS_S_MANIPULATION	The device service detected a manipulation.
JXFS_S_NO_MANIPULATION	The manipulation state has ended and the device is normal again. Some manipulation sensors require vendor specific functionality to reset the manipulation state. Therefore not all devices that join the manipulation state can also leave that state automatically.

4.2.2 Registering for Events and Event Delivery

To make an application ready for receiving events of a specific type, it must implement the corresponding listener interface by defining the included listener method.
All events have applicable event data attached to them and can be explicitly requested by the application by using the addXXXListener Methods in the Device Control classes (where XXX depicts the EventType). An application registering for 2 event types would look like this (the try-catch expressions are not included):

```
public class Sample implements IJxfsStatusListener,
                               IJxfsOperationCompleteListener
{
  public Sample()
  {
    JxfsDeviceManager mgr=JxfsDeviceManager.getReference();
    JxfsMagStripe mag=mgr.getDevice("myMagStripe");

mag.addOperationCompleteListener((IJxfsOperationCompleteListener)this);
    mag.addStatusListener((IJxfsStatusListener)this);
    ...
  }
  public void operationCompleteOccurred(JxfsOperationCompleteEvent e)
  {
    if(e.getResult()==IJxfsConst.JXFS_RC_SUCCESSFUL)
    {
       ...
    }
    ...
  }
  public void statusOccurred(JxfsStatusEvent e)
  {
     ...
  }
```

```
} // Sample.java
```

The JxfsOperationCompleteEvent received in the above method will be the “receipt” of the requested operation for the application. It is received only by this application.
In contrast to that, the JxfsStatusEvent will be sent to any application which has a valid Device Control.
The events generated in the Device Service are delivered to the control in a different thread context. The Device Control has to catch them and store them in a event queue; returning quickly to the Device Service. In another thread (one per event type) it now starts to deliver the events to the application. It uses only a single thread to do it. Thus flow control is simplified for the application. For details see “Threads and flow control” on page 18.

The possible error codes reported by these events are

- One of the global codes defined in IJxfsConst.java
- Special codes for specific devices (IJxfsXYZConst.java). Each Device Control can optionally have such an additional constants file.

Any exceptions from the Device Communication layer (i.e. RemoteException) are logged and a new JxfsException with error code JXFS_E_REMOTE is generated.

The Device Service gets references to objects which implement the following IJxfsEventNotification interface. In these objects the corresponding fireXXXEvent methods are invoked to deliver the events.
In cases where the event could not be delivered a JxfsException is thrown.

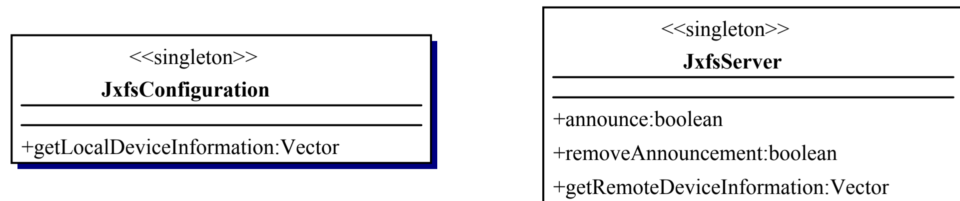
```
/////////////////////////////////////////////////////////////////////
//
// IJxfsEventNotification
//
//   Interface defining callback methods in the Device
//   Control that are callable by a Device Service.
//
/////////////////////////////////////////////////////////////////////
package events;

public interface IJxfsEventNotification
{
  public void fireIntermediateEvent(JxfsIntermediateEvent e)
               throws JxfsException;
  public void fireOperationCompleteEvent(JxfsOperationCompleteEvent e)
                throws JxfsException;
  public void fireStatusEvent(JxfsStatusEvent e)
              throws JxfsException;
}
```

5 Support Classes

The following section discusses the number of additional classes provided to generally support the defined infrastructure of J/XFS. These are both internal classes not visible to the application or applet as well as support classes which are used to present data to the application.

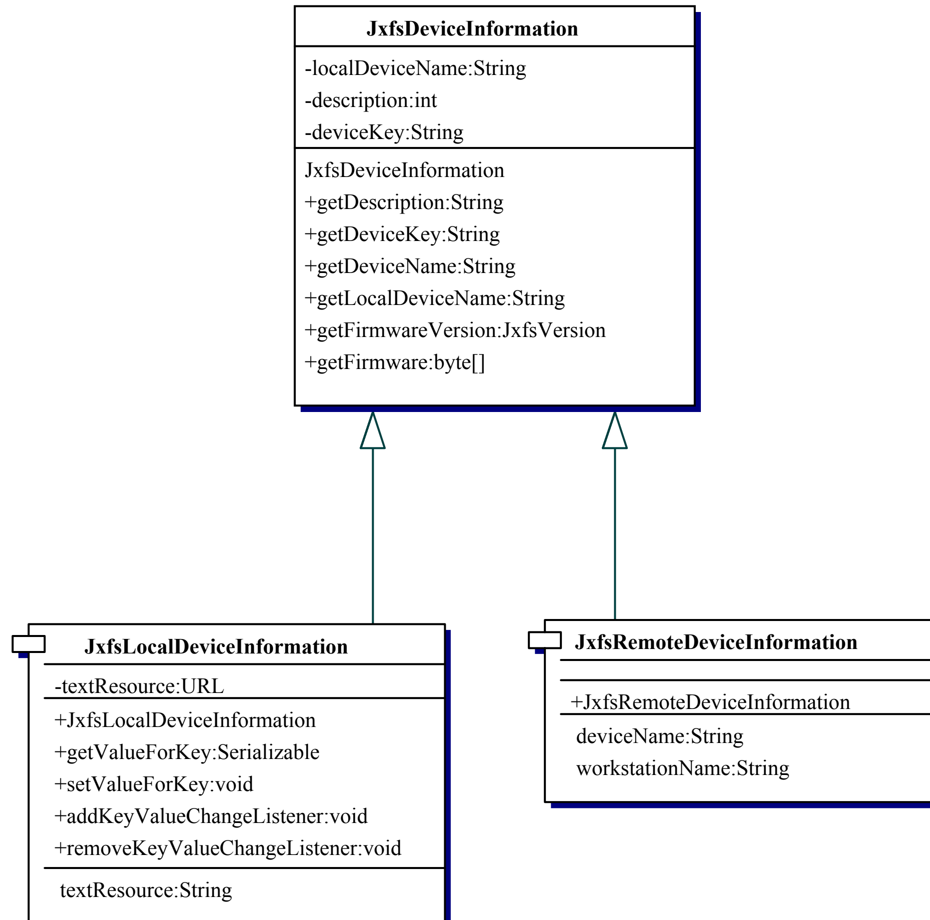
5.1 JxfsServer and JxfsConfiguration

The Device Manager must access a central data storage to be able to retrieve the configuration information for the local machine (JxfsConfiguration). This information is later used to initialize service objects for the locally attached devices. Additionally, the Device Manager and the Device Services retrieve any information they need from that repository, e.g. the default devices for each J/XFS client, their workstation name and port number.
The Device Manager uses this information to start up the remote access infrastructure so that the devices can be accessed by other J/XFS clients.
After successful initialization it must register these devices with a dynamic centralized cache (JxfsServer) so that other J/XFS clients know of the availability of its devices.
The DeviceManager is the only one to access these repositories directly; the services and controls do it via the DeviceInformation objects which are discussed in the next chapter.
The following graphic illustrates a sample interface to these repositories.



As this is a DM internal interface which is neither used by the application nor the controls and services it is not within the scope of this standard to define the exact interface; this is rather hidden in each specific DM implementation.
The important point is that two different kinds of server tasks are needed by the J/XFS infrastructure; one for the static configuration and one for dynamic availability information.
The question about how the information is loaded into the repository and how it can be changed there is not generically solvable and thus not explained here. The forum is aware of the fact that initializing, loading and administering the repository is a key feature of any program implementing J/XFS.

5.2 JxfsDeviceInformation



The basic configuration data describing the device is stored in the JxfsDeviceInformation object or in its subclasses JxfsLocalDeviceInformation and JxfsRemoteDeviceInformation. The objects of this classes are a kind of information container hiding configuration structure from the Controls and Services layer; the means how this information is stored / gathered can be changed by different implementations without affecting Device Service and Device Controls.
They can query these objects for generic information about the control like it's name, a description of the device etc.

During DM initialization all information about locally attached devices is gathered by requesting the JxfsLocalDeviceInformation objects from the repository. Their device information objects are then given by the DM to the Device Services and Device Controls at initialization time.
If there are devices which should be remotely accessible the DM generates the corresponding JxfsRemoteDeviceInformation and announces this at the J/XFS Server. The JxfsRemoteDeviceInformation also includes the connection information where the device can be found and the names of the remote objects, but this is implementation specific.

Currently the following standard properties (all read-only) are defined in the *JxfsDeviceInformation*:
localDeviceName
The unique device name for this device in this J/XFS client. It is freely assignable during the installation.
Note that this is a LOGICAL device name. Every device on a workstation must have a unique name.
deviceName

This property only exists by its getDeviceName() method. It returns the localDeviceName, but in the case of a remote device this name is augmented by the workstation name the device is connected to. Although an implementation of J/XFS may want to use its own format for the unique identification of a device we strongly suggest to use the format "<devicename>@<hostname>", i.e. printer2@workstation1.sample.domain.com to specify this. See chapter "Remote device access" at section 2.4.

description

A String with the clear text description what this device is and where it can be found. May be used to present in a list to the user. Sample: "Passbookprinter 2nd floor (Mr. Millers office)".

deviceKey

Unique identifier for the device in the repository, e.g. the deviceName without blanks. Like the localDeviceName this must be unique for all devices connected to a workstation.

The *JxfsLocalDeviceInformation* also contains

textResource

An URL (uniform resource locator, the Internet – way of specifying resource, e.g. "http://www.acme.com/support/printers") which identifies a location where the DS can find a file. This allows the Device Service to gain access to a file in a device specific format which can contains any language dependant strings the DS wants to use as parameters, e.g. in error messages.

The *JxfsRemoteDeviceInformation* also contains:

workstationName

The (unique) name of the workstation the device is connected to, usually the TCP/IP hostname.

The three DeviceInformation classes must implement Serializable as they are retrieved from the repository and thus must be streamable over the network.

Almost every Device Service has a need to store some additional device specific configuration data. This must also be put into the repository. As each Device Service has a reference to its local configuration data this object also supports reading / writing of arbitrary data.

The following methods are used for this:

getValueForKey

Syntax ***Serializable getValueForKey(String key) throws JxfsException***

Description This method allows an arbitrary persistent object to be retrieved from the repository under the given key. If the key is not found in the repository an exception with JXFS_E_NOEXIST is thrown.

setValueForKey

Syntax ***void setValueForKey(String key, Serializable value) throws JxfsException***

Description Saves the persistent object under the given name. If the key does not exist, it is created, if it exists, the value is replaced. The object must be a subclass of Serializable. To remove a key from the repository, use this method and specify null as the value parameter.
An exception JXFS_E_ILLEGAL is thrown if the key specified is not allowed. This can e.g. happen if a read-only key with the same name exists which cannot be overwritten.

Also, there is a provision so that the Device Service can register itself here to be informed if one of its key entries is changed on the server. It needs to implement the IJxfsKeyValueChangeListener interface as described in the chapter on the JxfsDeviceManager and use the following method of a LocalDeviceInformation object to register:

addKeyValueChangeListener

Syntax ***void addKeyValueChangeListener(IJxfsKeyValueChangeListener listener, String key) throws JxfsException***

Description This method registers the listener to be informed when the value for the key "key" in the repository changes. Throws a JXFS_E_PARAMETER_INVALID exception if one of the parameters is null.

removeKeyValueChangeListener

Syntax ***void removeKeyValueChangeListener(IJxfsKeyValueChangeListener listener) throws JxfsException***

Description Removes the given listener. Throws a JXFS_E_PARAMETER_INVALID exception if one of the parameters is null. If the listener is not known, a JXFS_E_FAILURE is thrown.

For the special case of querying the firmware level of the device, each DeviceInformation object also provides the following methods:

getFirmwareVersion

Syntax ***JxfsVersion getFirmwareVersion();***

Description This method can be used to gather the version information of a new firmware present in the repository for this device. It is null if no new firmware is present.

getFirmware

Syntax ***JxfsVersion getFirmwareVersion();***

Description This call returns the actual byte codes of the firmware from the repository. It can be used by the Device Service to update the device if requested.

The advantage of providing these special methods versus a generic getValueForKey() access is that the same keys are used by all vendors to access the firmware.

5.3 Tracing and error logging

Within the J/XFS architecture all components have the possibility to write traces and to do error logging via a standard interface.
The interface may also be used by the application and is provided by a *JxfsLogger* object.

5.3.1 Overview

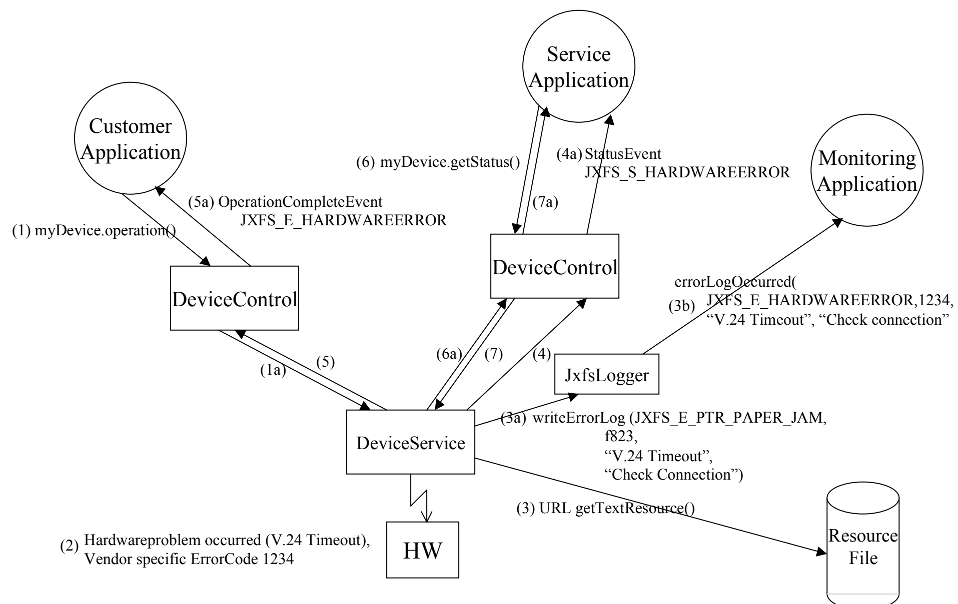
Tracing

Tracing is used to track the running of the various components: To do this, trace points are implemented in the programs. When they are activated they provide the logger object with information about internal states and events.
The trace points can be defined in different levels (trace point for function entry and function exit, trace point for tracing configuration entries, trace point for tracing debugging information, etc.). The activation of the different trace points can be component specific during runtime, e.g. if a developer is interested in the function entry and function exit points only, he has to activate the appropriate trace point for this specific component. This mechanism to activate different trace points during runtime is defined by the logger object.
The trace is therefore primarily a mechanism to analyze the behavior of the application or software modules and is mainly used by developers or field engineers.
A sufficient number of trace points can be set for every component; each of these points being unique in the system. Every trace point can be activated externally and without the relevant module being involved.

Error logging

In contrast to traces, error logging is used for the continuous logging of error or warning events from the components. Error logging is always active. Whenever a component calls the logger object's method to do error logging, an error log entry is produced by the logger object.
When collecting and recording trace and error data the standard logger object separates the two types of data from one another so that subsequent components which are, for example, only interested in error data, do not have to filter a mass of trace entries to isolate this error data. Via additional software modules these collected errors may be transferred to a central system (i.e. SNMP Event Reporting, see below).

This error logging facility serves as the central point where error information from the devices is gathered. It is therefore important that all devices make extensive use of this facility.
In the graphic below we have a small scenario of which messages and events would be triggered by a hardware error. It also shows how a service application might register for the Device and query its status after an error has occurred. Please note that the textual strings are in a language-specific format. Every DS has the duty to create this form of the information. In order to allow for a multi-language installation the DS can use the getTextResource() utility function from the JxfsLocalDeviceInformation object to receive the textual representation of an error. See “JxfsDeviceInformation” on section 5.2 for further details.



Explanation of the flow chart:

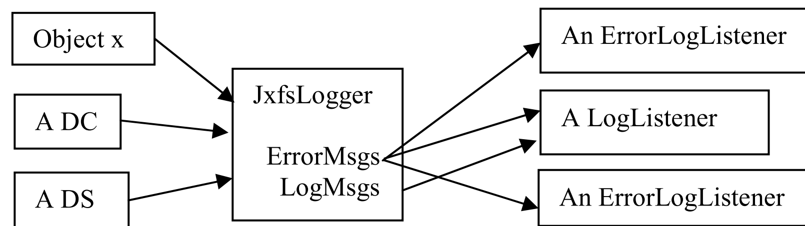
(1) - (1a)	An operation is sent to the Device Service
(2)	During performing the operation a hardware error occurs.
(3) - (3b)	The Device Service reads it resources and finds the corresponding error text and hint. The Device Service writes an error log to the JxfsLogger, the JxfsLogger sends an event to the registered listener with the errorCode, the extendedErrorCode, the error message, an error hint and a string containing an URL (uniform resource locator) where more information can be found.
(4) - (4a)	The Device Service sends a JxfsStatusEvent to the registered listener
(5) - (5a)	The Device Service sends the JxfsOperationCompleteEvent with the result JXFS_E_HARDWAREERROR to the application that wanted to perform the operation.
(6) - (6a)	The Service application tries to get some more detailed status information and performs a status query.

(7) - (7a) The JxfsStatusEvent is sent to a Service application.

5.3.2 JxfsLogger

The JxfsLogger is a single separate object which exists in every Java VM. Every object of course has access to its local JxfsLogger only. It can report errors and write informational and trace messages to the log.
The final logging of all messages is not done by the JxfsLogger itself. Rather a listener interface exists which can be implemented by other objects (under application control). The JxfsLogger sends out the messages to any object which has registered to be a listener. This is a very flexible mechanism using the Java event notification scheme, which makes it easy to distribute the available information.
There is also no restriction on who is using the JxfsLogger. It is especially not restricted to transfer only J/XFS related messages. Basically, even the application might use the JxfsLogger to report what's going on to an independent instance.

It is relatively easy to write a listener which simply writes every line to a file. This would of course only be available for an application or a signed applet on a system with a harddisk. An even more elaborate listener class could open a network connection to a supervising workstation, register e.g. with a SNMP instance and deliver its contents there. In the listener it can also be decided what to do with the delivered messages and filter them.
Basically, any number of objects can report to the JxfsLogger, which in turn informs one or more ErrorLogListener and LogListeners. The architecture looks like this:



The objects (or module groups of objects) which want to send messages to the JxfsLogger must first initialize the JxfsLogger for their use. They have to send it a short textual identifier and a longer description suitable for display. The short identifier must be used for all subsequent write calls. This is used by the JxfsLogger and its listeners to identify which class or module has generated the message and also enables them to use filters on a module bases.
Then they can use the 2 different writing methods, one for reporting error conditions and the other one to write trace log entries. The log entries are delivered with a level indicator which can be analyzed by a LogListener.

Every message given to the JxfsLogger is automatically augmented with a timestamp (a Date object) and a description of the issuing thread where the line was generated, so it is not needed to include such information by the issuing object.

The ErrorLogListener interface servers for instances which are only interested in the error messages which are reported. It is defined like this:

```
/**
 * IJxfsErrorLogListener.java
*/
public interface IJxfsErrorLogListener
{
  public boolean initialize(String parameters);
  public void errorLogOccurred(String source_identification,
                               String origin,
                               long errorCode,
                               long extendedErrorCode,
                               String message,
                               String hintText,
                               String help_url,
                               String curr_thread,
```

```
                              Date timestamp);
  public void shutdown();
  public String getDescription();
}
```

The first method is used to initialize the object. This is needed to have an initialization mechanism independent of the specific implementation.
The errorLogOccurred method is called by the JxfsLogger for every error message which is reported. The parameters in the errorLogOccurred method are explained below in the writeErrorLog method of the Logger, with the only difference, that the first parameter here is a stringized representation of the originating object.
The shutdown method is called by the JxfsLogger to inform the registered logger that after this method the system will shut down. This call must always return and allows the logger to do some cleanup work.
And finally, the getDescription() method should return a short (max. 80 chararcters) human-readable description of this LogListener, e.g. "FileLogger logging to c:\logfile". In multi-language environments this String should be internationalized.

A LogListener interface is also provided which also contains the error interface but augments it by supplying the log messages, too.

```
/**
 * IJxfsLogListener.java
 */
public interface IJxfsLogListener extends IJxfsErrorLogListener
{
  public void logOccurred(String source_identification,
                          String origin,
                          int level,
                          String log_message,
                          String curr_thread,
                          Date timestamp);
} // LogListener
```

As can be seen in above interface description, the idea is that LogListener inherit from the ErrorLogListener, i.e. they are receiving both error and normal logging messages. The ErrorLogListener only receive the error messages.

JxfsId
In order to provide information about a registered Logger, a helper class JxfsId is defined, which only contains information about a LogListener. It contains a description (as returned by the corresponding LogListener) as well as an integer id which is set to a unique value from the JxfsLogger in order to clearly identify this LogListener. Instances of class JxfsID are unique within the VM where they are created. If the parameterless constructor is used for this type the values of the ID and description are undefined.
Remark: This class may also be used to identify objects in other occasions, as its definition is quite generically applicable. The only prerequisite is that an integer is sufficient to uniquely identify the instance and a string-based description is available.
JxfsId has the following methods:

```
JxfsId(); // the constructor
JxfsId(int id, String description); // second constructor
void setId(int id); // Setter for id
int getId(); // Getter for id
void setDescription(String description); // Setter for description
String getDescription(); // Getter for description
String toString(); // returns the id + the description in one String
Object clone(); // returns a clone of this object
```

Now, the main class JxfsLogger has the following public methods:

getReference

Syntax	***static JxfsLogger getReference()***
Description	Returns the reference to the JxfsLogger-Object. Must be used to access this singleton.

registerModule

Syntax ***boolean registerModule(String origin, String description)***

Description Must be used by each object before the first line is reported to identify itself in both a short and a long form. The short form should uniquely identify this object (i.e. “DevMgr”), and the description should be suitable for display in a supervisor application, e.g. “Acme Passbook printer Device Service, Version 1.2”)
This method returns *false* if the given origin already exists as registered.

deregisterModule

Syntax ***boolean deregisterModule(String origin)***

Description If the object doesn’t use the Logger any more it should deregister using this call. If should only do this if it has successfully registered before.
This method returns *false* if the origin wasn’t found in the registered list.

writeErrorLog

Syntax ***boolean writeErrorLog(Object issuer, String origin, long errorCode, long extendedErrorCode, String message, String hintText, String help_url)***

Description Use this method to issue an error message. The issuer is the sending object itself. The origin is a short string with the module identification, it should have been announced to the logger by a previous registerModule() call.
The errorCode is the generic errorCode for this error, extendedErrorCode is a vendor specific, more detailed, code. The message is the error message itself in a language-specific form, the hintText gives some hints regarding a way to solve the error. Both message and hintText should not be long explanations but rather short strings (1-3 lines), and the help_url gives the originator of the message a way to announce where more detailed information regarding this error can be found.
When not applicable, extendedErrorCode may be specified as 0, and hintText and help_url may be left blank (“”). The other parameters are mandatory.
If some internal error occurred, *false* is returned.
Using an unregistered origin here is not recommended, but the message is logged anyway.

writeLog

Syntax ***boolean writeLog(Object issuer, String origin, int level, String message)***

Description Use this method to issue a log message. The issuer is the sending object itself. The origin is a short string with the module identification, it should have been announced to the logger by a previous registerModule() call.
The level identifier is an integer. It is left to the programmer to define the exact semantics of this integer.
The message itself should be clearly readable, and may also be language specific using the mechanism as outlined in the writeErrorLog method description.
To use an unregistered origin here is not recommended, but the message is logged anyway.

isLogActive

Syntax ***boolean isLogActive(String origin, int level)***

Description If logging is used there are potentially very many log entries. It is advisable to prevent generation of many log messages which are only thrown away afterwards. Also, creating the message to be logged may be time-consuming. For maximum system performance, before issuing a writeLog() call the issuer can check if that message should be logged at all. It does so by calling this method. If True is returned, the logging is desired. So, a typical usage is

```
if (JxfsLogger.getReference().isLogActive("DM",5)
  {
    JxfsLogger.getReference().writeLog("DM",5,0,
              "Cannot load class"+classToInstantiate,"","");
  }
```

The logActive state may dynamically change during runtime. A user of the logger should not issue this only once during startup but before every call.
How is this logging activated and deactivated? This is considered to be a detail of a J/XFS implementation and is thus not described here.

addErrorLogListener
addLogListener

Syntax ***JxfsId addErrorLogListener(IJxfsErrorLogListener listener),***
JxfsId addLogListener(IJxfsLogListener listener)

Description Any object implementing the required interfaces can register with these methods to receive either only the error messages or both error and log messages. Any registered LogListener will also receive all error messages.
A registration of the same object to both methods returns an error.
If the listener could not be added, a null value is returned.

removeErrorLogListener
removeLogListener

Syntax ***boolean removeErrorLogListener(int listenerId),***
boolean removeLogListener(int listenerId)

Description Use this method to deregister interest in the messages. The parameter can be queried from the JxfsId object corresponding to this Listener (returned from the following method)

getErrorLogListeners

Syntax ***Vector getErrorLogListeners()***

Description Return a Vector containing objects of type JxfsId for all registered ErrorLogListeners.

getLogListeners

Syntax ***Vector getLogListeners()***

Description Return a Vector containing objects of type JxfsId for all registered LogListeners.

getErrorLogListener

Syntax ***IJxfsErrorLogListener getErrorLogListener(int listenerId)***

Description Return the reference to the ErrorLogListener identified by the listenerId.

getLogListener

Syntax ***IJxfsErrorLogListener getLogListener(int listenerId)***

Description Return the reference to the LogListener identified by the listenerId.

Additionally, there are some available methods only to be used internally although these methods are declared public and the internal use only is not enforced.

shutdown

Syntax ***void shutdown()***

Description This method should only be used by the DeviceManager. It prepares for system shutdown: The JxfsLogger now tells all connected listeners to shutdown. After

this call has completed the JxfsLogger is in its original startup state again.

As stated above, the ‘level’ integer specified in the trace log messages is not predefined. This standard, however, proposes to adhere to the following rules:

- Ids from 1 to 9 should be used to describe the workflow as outlined below;
- Ids 10 to 99 are for generic trace points
- Ids starting with 100 are for additional custom trace points.

Value	Meaning
1	Report with this id that an operation request was made and interesting parameters to it.
2	Report with this id if an operation was completed, i.e. an OC Event is sent and its values.
3	A device property has changed.
4	The device status has changed (includes device ready, device closed and shutdown).
5 .. 9	reserved, not used in this version
10	Method entry
11	Method exit
80-99	reserved for J/XFS internal use
100	JXFS_LOG_USEROFFSET, start of first custom trace point.

5.3.3 Systems Management and Monitoring (e.g. SNMP)

In order to centrally manage hard- and software installations in a banking environment it is desirable to be able to centrally supervise the workstation and connected hardware, as well as to have a means that the complex peripheral devices can use to post a message to a central administrative supervisor workstation.

J/XFS contains a generic logger to which arbitrary listening objects can register. This can be any specific logger objects (e.g. a Tivoli SNMP client object) or even the application itself. It can be expected that when the J/XFS device framework spreads, the systems management vendors will implement respective loggers for their infrastructure. Any program implementing the J/XFS infrastructure should usually also provide for a set of simple LogListeners.

5.4 J/XFS constant codes

The currently available constant definitions can be seen in the following code snippet. Each device type can have its own additional constants file, and the codes used in it should be in the range from 3000 (`JXFSDEVICE_OFFSET`) to 30000 in order to avoid overlap of the standardized codes. The return codes defined in the directIO statement should start from 30000 as outlined below.

```
package com.jxfs.general;

/**Constant definitions concerning all devices.*/
public interface IJxfsConst {
  //######################################################################
  //#### General constants
  //######################################################################

  /** General constant:
   *  Any error start at number...
   */
  public static final int JXFSERR = 1000;

  /** General constant:
   *  Any extended Error starts at number ...
   */
  public static final int JXFSERREXT = 2000;
```

```
/** General constant:
 *  basic operation id code OPEN
 */
public static final int JXFS_O_OPEN = 900;

/** General constant:
 *  basic operation id code CLOSE
 */
public static final int JXFS_O_CLOSE = 901;

/** General constant:
 *  basic operation id code JXFS_O_UPDATEFIRMWARE
 */
public static final int JXFS_O_UPDATEFIRMWARE = 902;

/** General constant:
 * Firmware in repository is newer than current firmware.
 */
public static final int OK_NEWER = 903;

/** General constant:
 * Current firmware is newer but update possible.
 */
public static final int OK_OLDER = 904;

/** General constant:
 * Different firmware functionality but update possible.
 */
public static final int OK_OTHER = 905;

/** General constant:
 * No firmware found in the repository.
 */
public static final int NO_SOURCE = 906;

/** General constant:
 * Firmware in the repository is not correct for this device.
 */
public static final int NO_MATCH = 907;

/** General constant:
 * Firmware update is not possible with this device.
 */
public static final int NO_SUPPORT = 908;

/**
 * Specifies active (successfully started) local devices.
 */
public static final int JXFS_LEVEL_ACTIVE = 909;

/**
 * Specifies local devices.
 */
public static final int JXFS_LEVEL_CONFIGURED = 910;

/**
 * Specifies active (successfully started) devices on this
 * workstation.
 */
public static final int JXFS_LEVEL_WORKSTATION = 911;

/**
 * Specifies active (successfully started) devices.
 */
public static final int JXFS_LEVEL_ALL = 912;

/** General constant:
 * Firmware in the repository is equal to the firmware in the device.
 * Update possible.
 */
public static final int OK_EQUAL = 913;

//####################################################################
//#### Offset constant
//####################################################################

/** Offset constant:
 *  Any code defined by a specific device operation should start from
```

```
 *  this offset in order not to mix up with J/XFS definitions.
 */
public static final int JXFSDEVICE_OFFSET = 3000;

/** Offset constant of known device types:
 * PTR
 */
public static final int JXFS_PTR_OFFSET = JXFSDEVICE_OFFSET + 0000;

/** Offset constant of known device types:
 *  MSD
 */
public static final int JXFS_MSD_OFFSET = JXFSDEVICE_OFFSET + 1000;

/** Offset constant of known device types:
 *   PIN
 */
public static final int JXFS_PIN_OFFSET = JXFSDEVICE_OFFSET + 2000;

/** Offset constant of known device types:
 *  PINIso
 */
public static final int JXFS_ISO_OFFSET = JXFSDEVICE_OFFSET + 2500;

/** Offset constant of known device types:
 *  CDR
 */
public static final int JXFS_CDR_OFFSET = JXFSDEVICE_OFFSET + 3000;

/** Offset constant of known device types:
 *  ALM
 */
public static final int JXFS_ALM_OFFSET = JXFSDEVICE_OFFSET + 4000;

/** Offset constant of known device types:
 *  TIO
 */
public static final int JXFS_TIO_OFFSET = JXFSDEVICE_OFFSET + 5000;

/** Offset constant of known device types:
 *  CHK
 */
public static final int JXFS_CHK_OFFSET = JXFSDEVICE_OFFSET + 6000;

/** Offset constant of known device types:
 *  SIU
 */
public static final int JXFS_SIU_OFFSET = JXFSDEVICE_OFFSET + 7000;

/** Offset constant of known device types:
 *  DEP
 */
public static final int JXFS_DEP_OFFSET = JXFSDEVICE_OFFSET + 8000;

/** Offset constant of known device types:
 *  CAM
 */
public static final int JXFS_CAM_OFFSET = JXFSDEVICE_OFFSET + 9000;

/** Offset constant of known device types:
 *  VDM
 */
public static final int JXFS_VDM_OFFSET = JXFSDEVICE_OFFSET + 10000;

/** Offset constant:
 *  Any rc or code defined by a specific device for a direct-IO
 *  operation should start from this offset in order not to mix
 *  up with J/XFS definitions.
 */
public static final int JXFSDIRECTIO_OFFSET = 30000;

/**
 * Offset constant for trace points / trace levels of device services or
 * applications.
 */
public static final int JXFS_LOG_USEROFFSET = 100;
```

```
////////////////////////////////////////////////////////////////////
// Return Codes from calls which deliver an immediate result
////////////////////////////////////////////////////////////////////

/** Return code from calls which deliver an immediate result:
 *  Standard return for successful calls
 */
public static final int JXFS_RC_SUCCESSFUL = 0;

/** Return codes from calls which deliver an immediate result:
 *  Unspecified unsuccessful return
 */
public static final int JXFS_RC_UNSUCCESSFUL = 1;

////////////////////////////////////////////////////////////////////
// Exception code
////////////////////////////////////////////////////////////////////

/** Exception code:
 *  Device Control is not registered at the service
 */
public static final int JXFS_E_UNREGISTERED = 1 + JXFSERR;

/** Exception code:
 *  Device still closed, function not yet available
 */
public static final int JXFS_E_CLOSED = 2 + JXFSERR;

/** Exception code:
 *  Device still already or still opened
 */
public static final int JXFS_E_OPEN = 3 + JXFSERR;

/** Exception code:
 *  Device is already or still claimed by this Device Control
 */
public static final int JXFS_E_CLAIMED = 4 + JXFSERR;

/** Exception code:
 *  Device is not claimed
 */
public static final int JXFS_E_NOTCLAIMED = 5 + JXFSERR;

/** Exception code:
 *  Requested Service not available
 */
public static final int JXFS_E_NOSERVICE = 6 + JXFSERR;

/** Exception code:
 *  Requested communications object not available, i.e
 *  the device is not remotely accessible
 */
public static final int JXFS_E_NOTREMOTE = 7 + JXFSERR;

/** Exception code:
 *  Requested Control not available
 */
public static final int JXFS_E_NOCONTROL = 8 + JXFSERR;

/** Exception code:
 *  Device is disabled
 */
public static final int JXFS_E_DISABLED = 9 + JXFSERR;

/** Exception code:
 *  Illegal request. Not allowed at this time or never allowed.
 *  This value is usually the result of a problem in the application code.
 */
public static final int JXFS_E_ILLEGAL = 10 + JXFSERR;

/** Exception code:
 *  The device hardware could not be found or is not connected
 */
public static final int JXFS_E_NOHARDWARE = 11 + JXFSERR;

/** Exception code:
 *  The device is switched offline
 */
```

```
public static final int JXFS_E_OFFLINE = 12 + JXFSERR;

/** Exception code:
 *  The requested item (device or key) does not exit
 */
public static final int JXFS_E_NOEXIST = 13 + JXFSERR;

/** Exception code:
 *  Object already exists
 */
public static final int JXFS_E_EXISTS = 14 + JXFSERR;

/** Exception code:
 *  The operation failed or device service is not initialized.
 */
public static final int JXFS_E_FAILURE = 15 + JXFSERR;

/** Exception code:
 *  A timeout occurred before completion
 */
public static final int JXFS_E_TIMEOUT = 16 + JXFSERR;

/** Exception code:
 *  Operation not possible, device is already busy
 */
public static final int JXFS_E_BUSY = 17 + JXFSERR;

/** Exception code:
 *  One of the parameters given was invalid. Further information
 *  may be found in extendedErrorCode
 */
public static final int JXFS_E_PARAMETER_INVALID = 18 + JXFSERR;

/** Exception code:
 *  Errors during a remote operation
 */
public static final int JXFS_E_REMOTE = 19 + JXFSERR;

/** Exception code:
 *  Errors during an input or output operation
 */
public static final int JXFS_E_IO = 20 + JXFSERR;

/** Exception code:
 *  The operation was cancelled by the application via cancel()
 */
public static final int JXFS_E_CANCELLED = 21 + JXFSERR;

/** Exception code:
 *  The operation is not supported by this object
 */
public static final int JXFS_E_NOT_SUPPORTED = 22 + JXFSERR;

/** Exception code:
 *  Error during firmware update or no runnable firmware in device
 */
public static final int JXFS_E_FIRMWARE = 23 + JXFSERR;

/** Exception code:
 *  Unpredictable system error/exeception occurred that is not
 *  device related.
 */
public static final int JXFS_E_SYSTEM = 24 + JXFSERR;

/** Exception code:
 *  Error occurred that is device related.
 */
public static final int JXFS_E_HARDWAREERROR = 25 + JXFSERR;

/////////////////////////////////////////////////////////////////////
// Status constants
/////////////////////////////////////////////////////////////////////
/** Status constant:
 *  control released
 */
public static final int JXFS_S_RELEASED = 1;
```

```
/** Status constant:
 *  control claimed
 */
public static final int JXFS_S_CLAIMED = 2;

/** Status constant:
 *  hardware error occured
 */
public static final int JXFS_S_HARDWAREERROR = 3;

/** Status constant:
 *  user action error occured
 */
public static final int JXFS_S_USERACTIONERROR = 4;

/** Status constant:
 *  action still working
 */
public static final int JXFS_S_WORKING = 5;

/** Status constant:
 *  shutdown
 */
public static final int JXFS_S_SHUTDOWN = 6;

/** Status constant:
 *  power save mode on
 */
public static final int JXFS_S_POWERSAVEON = 7;

/** Status constant:
 *  power save mode off
 */
public static final int JXFS_S_POWERSAVEOFF = 8;

/**
 * A running device service was stopped.
 */
public static final int JXFS_S_SERVICE_STOPPED = 9;

/**
 * A stopped device service was started.
 */
public static final int JXFS_S_SERVICE_STARTED = 10;

/**
 * Communication is broken.
 */
public static final int JXFS_S_REMOTEFAILURE = 11;

/**
 * Threshold status has changed.
 */
public static final int JXFS_S_BIN_STATUS = 12;

/**
 * Media status has changed.
 */
public static final int JXFS_S_MEDIA_STATUS = 13;

/**
 * Device detected a manipulation.
 */
public static final int JXFS_S_MANIPULATION = 14;

/**
 * Manipulation state has ended.
 */
public static final int JXFS_S_NO_MANIPULATION = 15;


////////////////////////////////////////////////////////////////////
// General Constants
////////////////////////////////////////////////////////////////////
/** General constant:
 */
public static final int JXFS_FOREVER = -1;
```

```
  /** General constant:
   */
  public static final int JXFS_ALL = -1;

  /**
   * Specifies the first valid value for a controlId.
   */
  public final static int JXFS_VALID_CONTROLID=2;
}
```

5.5 Temporary data and generic classes

5.5.1 JxfsType

This is a class any J/XFS class which contains data should inherit from. This can be the data objects delivered within some events or any other complex object which serves as an input or output parameter for Device Control methods.
This is needed to ensure the streamability of any data class used in J/XFS because any data object might be streamed over a network connection and must be stored in the repository in its binary format.
Java base data types like String, int, etc. are streamable anyway and thus may be used as parameters without putting them into special wrapper classes.

Summary

Implements : Serializable **Extends : Object**

Method	Returns	Meaning
copy()	*JxfsType*	Constructs a deep copy of the object and returns a reference to that copy.

Methods

copy

Syntax ***JxfsType copy();***

Description This method constructs deep copies of this object and all objects directly or indirectly referenced by it. A reference to the copy is returned. It is guarantied that all modifications of the copy won't affect the original object.

In the following chapters a number of generic data classes are defined. They can be used as base classes for device specific extensions or right away as return values for device specific things.

5.5.2 JxfsBasicType

This class can be used in cases where just a simple serializable data element is required, in order to avoid the proliferation of JxfsType subclasses.

Summary

Implements : Serializable **Extends : JxfsType**

Property	Type	Access	Initialized by
value	Serializable	R	Device Service Classes

value Property R

Type	***Serializable***
Initial Value	Depends on event and Device Service
Description	Holds a serializable object. Its class will vary depending on the Device Service and the place where this object is used.

5.5.3 JxfsStatus

The JxfsStatus object delivers status information for J/XFS.
Each Device Service has such an object. A copy of this is returned from the getStatus() method.

Implements : Cloneable **Extends : JxfsType**

Property	Type	Access	Initialized by
open	boolean	RW	Device Service
claimPending	boolean	RW	Device Service
claimed	boolean	RW	Device Service
busy	boolean	RW	Device Service
hardwareError	boolean	RW	Device Service
userActionError	boolean	RW	Device Service
powerSave	boolean	RW	Device Service
manipulated	boolean	RW	Device Service

Method	Returns	Meaning
JxfsStatus()	-	Constructs a new status. Any property is *false*.
set*Property*	*void*	Set the corresponding property, i.e. *void setBusy(boolean setTo)*.
isOpen	boolean	Returns *true* if the device is opened, *false* if not.
isClaimPending	boolean	The device has received a claim request which is not yet granted.
isClaimed	boolean	Returns *true* if the device is claimed, *false* if not.
isBusy	boolean	Is set if an operation is running.
isHardwareError	boolean	A hardware error is a device error which can only be fixed by service personnel.
isUserActionError	boolean	If an error condition can be fixed by user action (e.g. supplying more paper) this is *true*. Even if this is *false* an error may be present, namely the above hardware error.
isWorking	boolean	If neither a hardware nor user action error is present and the device is opened it is assumed to be working, i.e. this method returns *true*.
isPowerSave	boolean	If the device is in power save mode this returns *true*.
isManipulated	boolean	If the device is manipulated.

This table reflects current use of JxfsStatus object in all Device Class Interfaces:

PIN	getStatus returns a JxfsStatus object. No specific status objects.
MSD	getStatus returns a JxfsStatus object. Additional mediaStatus object.
TIO	getStatus returns a JxfsStatus subclassed object with a few properties.
CDR	getStatus returns a JxfsStatus subclassed object with lots of new properties.
PTR	getStatus returns a JxfsStatus object. Additional ptrStatus property contains a new subclassed JxfsStatus object with additional properties.
ALM	getStatus returns a JxfsStatus object. No specific status objects.
SIU	getStatus returns a JxfsStatus subclassed object with lots of new properties.
DEP	getStatus returns a JxfsStatus subclassed object with lots of new properties. Additional mediaStatus object.
CHK	getStatus returns a JxfsStatus object. Additional mediaStatus, inkStatus and lampStatus objects.
CAM	getStatus returns a JxfsStatus object. Additional getCameraStatus asynchronous method returns a JxfsStatus object for a specific focus Type.
VDM	getStatus returns a JxfsStatus object. Additional vdmStatus property containing new properties but not extending JxfsStatus class.
SCN	getStatus returns a JxfsStatus object. No specific status objects.

The contents of this object reflects the Device Service status at the time when the object was returned.

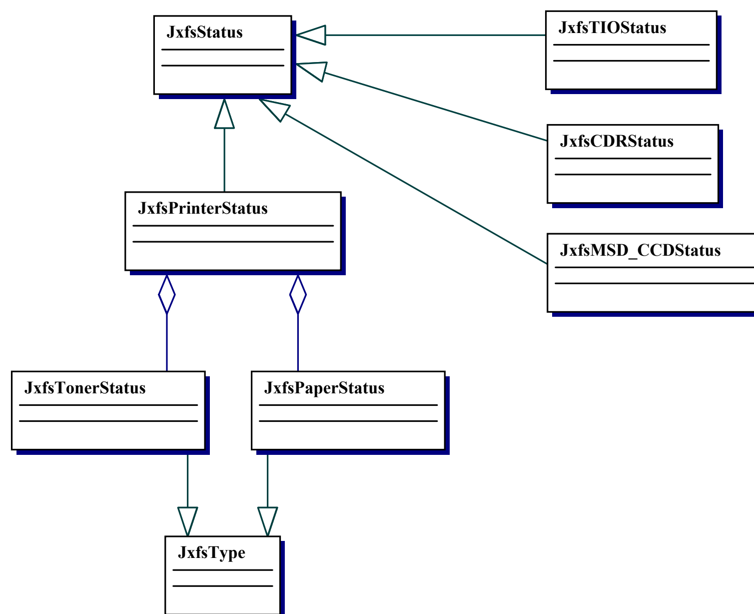
The following information is deprecated in favour of the Status Selector Approach:

- ✓ The object returned is at least of type JxfsStatus, and may be a subclass of it. Each device type which has additional status makes a subclass of it and adds the corresponding set and query methods to this object. A specific get method is provided by the Device Class Interface in order to get the subclassed status object containing full status information.

To get information about the fill level of the retain bin in a motorized MSD the application has to issue the following calls:

```
JxfsThresholdStatus retain=myMSD.getStatus().getRetainBinStatus();
if (retain.isFull()) ....
```

This relationship of JxfsStatus, its subclasses and their aggregates is outlined in the following graphic:



For future specs the additional status won't extend JxfsStatus to prevent redundant status eventing. Additional status will be retrieved using the *getStatus(java.util.List)* method.

5.5.3.1 Usage and Interpretation of Manipulation Information

As many fraudulent attempts result in a device going into a hardware error state this approach opens for some attractive status combinations of the HW error and manipulation property of the JxfsStatus class:

hardware error	**manipulation**	**comment**
false	*false*	No manipulation, no problem
false	true	A manipulation was detected, but the device can still be operated. In this state an application should not use this device any more for security relevant operations. An example can be that the card reader device service has detected that the serial number of the device has changed what may indicate an unauthorizated device exchange. Another example is a card reader that detected a skimming device. In this case the application may inhibit all transaction, where magnetic cards are involved, but may still allow the usage of chip cards. Because of the specific rule in CWA14923-3 (Appendix A: Card Reader Fraud Behaviour) an older application may rely on a pending hardware error indicating a manipulation. The issuance of a hardware error in such a situation was only a temporary measure until there is a clear solution in a new CWA (as specified here).
true	*false*	The device cannot be operated.
true	true	The device service detects a manipulation plus the device cannot be operated any more. An example is if a card is trapped in a card reader with a lebanese loop and cannot be moved any more and the device service can detect that this is the result of a manipulation.

Not in all situations it is possible to distinguish a hardware error from a manipulation without ambiguity. So in very seldom situations a possible manipulation can be reported as hardware error without manipulation. It is the responsibility of the device service to assure that these cases can be minimized. An application must be sure that a manipulation has occurred only if the manipulation flag indicates this. An example can be a lebanese loop in a card reader that resulted in a card jam. Often the device does not know if the jam occurred from a manipulation or another problem. In such a case the information provided by the device service must be a compromise between probability and impact of an assumed manipulation. In any case it must be assured that an application is not harmed, if this state is deliberately assumed as not manipulated ("*false* positive case")..

There are currently many vendor specific ways how manipulation can be detected at different devices. Hardware vendors constantly develop new and more efficient ways to detect attacks. As these ways differ in many aspects it is not possible to standardize more than the information that a manipulation has been detected. Even the way how to reset the manipulation detection cannot be standardized. Some detection mechanisms automatically detect if an attack has ended while others have to be reset manually by technicians.

5.5.4 JxfsMediaStatus

This class defines a generic API to query the status of a media in a device. It is e.g. used by the printer to represent its paper position or by the motorized MagStripe device class to represent the card location.

This object is received by the application either by calling getStatus() and querying the device specific status object[10] or by analyzing the details object in the JxfsStatusEvent. The state given by the JxfsMediaStatus object reflects the state of the device at the time of its sending, i.e. the device state may already have changed again. It is always only filled by the Device Service.

If the application wants to check if the mediaState checking is supported a method inside the containing JxfsStatus or the device capabilities must be queried.

Summary

Implements : Cloneable **Extends : JxfsType**

Property	Type	Access	Initialized by
mediaState	int	RW	Device Service classes

Method	Return	Meaning
JxfsMediaStatus(int state)	void	Constructs a new object, media state is set accordingly.
setMediaState(int state)	void	Set the mediaState property to the given value. No sanity checking is done.
getMediaState()	int	Return the mediaState property.
isEjected()	boolean	Media is in the entry/exit slot of the device.
isJammed()	boolean	Media is jammed in the device.
isPresent()	boolean	Media is inserted in the device.
isUnknown()	boolean	State of the media cannot be determined with the device in its current state.
isRetracted()	boolean	Media is in the retain bin
isNoMedia()	boolean	There is no media in the device
getTransition(JxfsMediaStatus previousState)	int	Calculates the transition that triggered the state change between the previous state and the current one.
toString()	String	Returns a short textual representation of the contents of this object.

Properties

mediaState Property R

Type *int*
Initial Value JXFS_S_MEDIA_UNKNOWN
Description Specifies the current state of the media. Depending on device capability, *mediaState* will be set to one of the following values:

Value	Meaning
JXFS_S_MEDIA_NOT_PRESENT = 0	Media is not present in the device.

[10] The method call would be similar to *myPrinterDeviceControl.getStatus().getPaperPosition()*. See section JxfsStatus for explanation.

JXFS_S_MEDIA_EJECTED=1	Media is at the entry/exit slot of the device.
JXFS_S_MEDIA_JAMMED=2	Media in the device is jammed.
JXFS_S_MEDIA_PRESENT=4	Media is inserted in the device.
JXFS_S_MEDIA_UNKNOWN=8	State of the media cannot be determined.
JXFS_S_MEDIA_RETRACTED=32	Media is retracted into the retain bin.
JXFS_S_MEDIA_NOTSUPPORTED =16	Media is not supported by this device. (deprecated)

Event If the value of this property changes, the Device Service will send all registered IJxfsStatusListeners a JxfsStatusEvent with the following value and the corresponding mediaState object. This usually overrides the generation of a generic UserActionError event.

Value	**Meaning**
JXFS_S_MEDIA_STATUS	*mediaState* changed.

The setting of the property should only be done by the respective device service (using the setMediaState() method).

Methods

Rather than analysing the mediaState integer, the application should use the following methods query the status. As the contents of the queried status object does not change after it has been received they are always available and do always return the same value. They also do not generate any events or take any parameters.

isEjected Method

Syntax ***boolean isEjected()***

Description Returns *true* if media is in the entry/exit slot of the device (the value of the *mediaState* property is JXFS_S_MEDIA_EJECTED).

isJammed Method

Syntax ***boolean isJammed()***

Description Returns *true* if media is jammed in the device (the value of the *mediaState* property is JXFS_S_MEDIA_JAMMED).

isPresent Method

Syntax ***boolean isPresent()***

Description Returns *true* if media is inserted in the device (the value of the *mediaState* property is JXFS_S_MEDIA_PRESENT).

isUnknown Method

Syntax ***boolean isUnknown()***

Description Returns *true* if the status of the media cannot be determined with the device in its current state (the value of the *mediaState* property is JXFS_S_MEDIA_UNKNOWN).

isRetracted Method

Syntax ***boolean isRetracted()***

Description	Returns *true* if media is in the retract bin (the value of the *mediaState* property is JXFS_S_MEDIA_RETRACTED). ***Notice***: this media state value is a special temporary condition. See the state diagram below and its related text for an explanation.

isNoMedia Method

Syntax	***boolean isNoMedia()***
Description	Returns *true* if no media is present in the device (the value of the *mediaState* property is JXFS_S_MEDIA_NOT_PRESENT).

getTransition Method

Syntax	***int getTransition(JxfsMediaStatus previousState)***
Description	Calculates the transition that triggered the state change between the previous state and the current one. This is a helping method that can be used to determine what user action or application request originated the change to the current media status. The returned value will be one of the following:

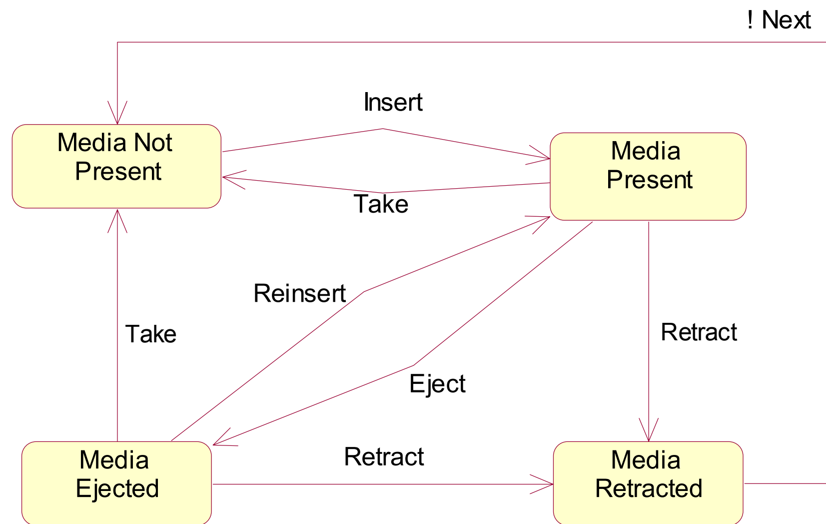
Value	**Meaning**
JXFS_S_MTR_INSERT = 0	Media is inserted by the user
JXFS_S_MTR_TAKE = 1	Media is taken by the user
JXFS_S_MTR_EJECT = 2	Media is ejected, and presented to the user at the exit slot
JXFS_S_MTR_REINSERT = 3	Media is reinserted from the exit slot back to the unit
JXFS_S_MTR_RETRACT = 4	Media is retracted to the retain bin
JXFS_S_MTR_NEXT = 5	This transition always takes place after a retract transition. See explanation below.
JXFS_S_MTR_JAM = 6	Media is jammed from any state
JXFS_S_MTR_UNKNOWN = 7	Media status has gone into an unknown state, from any state
JXFS_S_MTR_INVALID = -1	The transition is invalid, and not contemplated in the state diagram below. This value will be returned when an invalid value of the previousState parameter is given.

toString Method

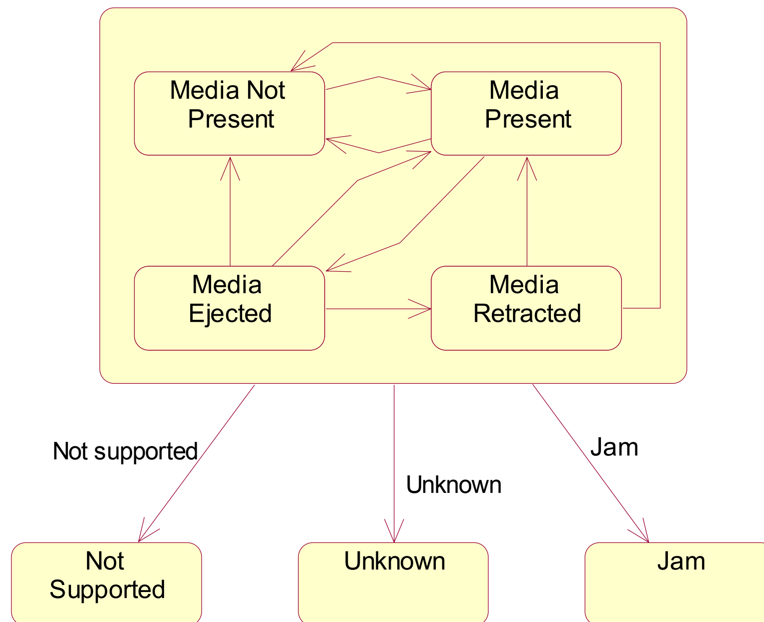
Syntax	***String toString()***
Description	Returns a small textual representation of this object. This is an identifier, the int and a short code of the state (in capitals if *true*, in small caps otherwise). Sample: “MediaStatus(6-eJpu)@4AC2F, with the hex number being the hash code of the object.

State diagram

The following state diagram shows the possible states that a given medium can be at, along with the possible transitions between its states. Only the normal (non-problem) conditions are shown first, then a separate diagram is provided to depict the media status where there is some problem condition, such as “jam” or “unknown”.
States and transitions are labeled after the constants defined above.



The following state diagram shows the possible problem states and their transitions from any of the above states:



Media Retracted State

The "Media Retracted" state is a special state: it has been placed in order to help applications know that a given medium was just successfully retracted. After a media status event with a "media retracted" state is sent, another media status event follows immediately, with the "media not present" state, indicating that the reader is now ready to accept new media.
In that sense, the "media retracted" state is not a real state by itself, but a temporary state between the "media present" or "media ejected" states and the "media not present" state. This new state ensures backward compatibility with device services and applications that are not aware of it. An application programmed under the previous version of the J/XFS specifications, would ignore the new state. And a device service programmed under the previous version of the specifications, will simply fire a "media not present" event directly after a retract command from the application.

5.5.5 JxfsThresholdStatus

This class defines a generic API to query and detect several threshold values. This can be e.g. the paper supply present in a printer (to detect if it's low or empty) or the retain bin of a card reader (to detect if its full or almost full).
This object is received by the application either by calling getStatus() and querying the device specific status object or via certain JxfsStatusEvents. The state given by an object of this type reflects the state of the device at the time of its sending, i.e. the state may already have changed again. It is always only filled by the Device Service.

If the application wants to check if the threshold checking is supported a method inside the applicable JxfsStatus or the device capabilities must be queried.

Summary

Implements : Cloneable **Extends : JxfsType**

Property	Type	Access	Initialized by
thresholdState	int	RW	Device Service classes

Method	Return	Meaning
JxfsThresholdStatus(int state)	void	Constructs a new object, threshold state is set accordingly.
setThresholdState(int state)	void	Sets the property. No sanity checking is done.
getThresholdState()	int	Returns the property. *This method is deprecated since CWA 14923:2004.*
isFull()	boolean	see property description.
isHigh()	boolean	see property description.
isLow()	boolean	see property description.
isEmpty()	boolean	see property description.
isUnknown()	boolean	see property description.
isSupported()	Boolean	See property description
isOK()	boolean	see property description
toString()	String	Returns a short textual representation of the contents of this object.

Properties

thresholdState Property RW

Type ***int***
Initial Value n/a
Description Returns the property. *This method is deprecated since CWA 14923:2004.*
Event If the value of this property changes, the Device Service will send to all registered IJxfsStatusListeners a JxfsStatusEvent with the following value and the corresponding JxfsThresholdStatus object. This usually overrides the generation of a generic UserActionError event.

Value	Meaning
JXFS_S_BIN_STATUS	*thresholdStatus* changed.

The setting of the property should only be done by the respective device service (using the setThresholdState() method).

Methods

isFull Method

Syntax ***boolean isFull()***

Description Returns *true* if the bin is full (the value of the *thresholdState* property is JXFS_S_BIN_FULL).

Parameter **None**

Event No additional events are generated.

isHigh Method

Syntax ***boolean isHigh()***

Description Returns *true* if the bin is high (the value of the *thresholdState* property is JXFS_S_BIN_HIGH).

Parameter **None**

Event No additional events are generated.

isLow Method

Syntax ***boolean isLow()***

Description Returns *true* if the bin is low (the value of the *thresholdState* property is JXFS_S_BIN_LOW).

Parameter **None**

Event No additional events are generated.

isEmpty Method

Syntax ***boolean isEmpty()***

Description Returns *true* if the bin is empty (the value of the *thresholdState* property is JXFS_S_BIN_EMPTY).

Parameter **None**

Event No additional events are generated.

isUnknown Method

Syntax ***boolean isUnknown()***

Description Returns *true* if the status of the bin cannot be determined with the device in its current state (the value of the *thresholdState* property is JXFS_S_BIN_UNKNOWN).

Parameter **None**

Event No additional events are generated.

isSupported Method

Syntax ***boolean isSupported()***

Description Returns *true* if the device supports bin threshold status capabilities. (the value of the *thresholdState* property is **not** JXFS_S_BIN_UNSUPPORTED).

Parameter **None**

Event No additional events are generated.

isOK Method

Syntax ***boolean isOK()***

Description Returns *true* only if the value of the property is exactly JXFS_S_BIN_OK. Therefore, this does not mean that if isOK() returns *false*, there is necessarily a problem. The bin can be just high or low, for example, but functioning perfectly.

Parameter **None**

Event No additional events are generated.

toString Method

Syntax ***String toString()***

Description Returns a small textual representation of this object. This is an identifier, the int and a short code of the stati (in capitals if *true*, in small caps otherwise). Sample: “Threshold(3-FHleu)”.

5.6 Reference Implementation for the J/XFS Enum Pattern

The following code defines an abstract class that meets all the requirements for the J/XFS Enum Pattern. An Enum pattern containing a list of constants can be easily defined extending this abstract class.

```
package com.jxfs.general;

import java.lang.reflect.*;

public abstract class JxfsEnum extends JxfsType{

    private transient String _fieldName;

    private void writeObject( java.io.ObjectOutputStream out) throws
            java.io.IOException {

        Class clazz = getClass();
        Field [] f = clazz.getDeclaredFields();

        for( int i = 0; i < f.length; i++) {
            try{
                int mod = f[ i].getModifiers();

                if( Modifier.isStatic( mod) &&
                    Modifier.isFinal(  mod)  &&
                    Modifier.isPublic( mod)) {

                        if( this == f[ i].get( null)) {
                            String fName=f[ i].getName();
                            out.writeObject( fName);
                        }

                }
            } catch( IllegalAccessException ex){
                throw new java.io.IOException( ex.getMessage());
            }
        }
    }

    private void readObject( java.io.ObjectInputStream in) throws
            java.io.IOException {

        try{
            _fieldName = ( String)in.readObject();
        }catch( ClassNotFoundException ex){
            throw new java.io.IOException( ex.getMessage());
        }
    }

    public Object readResolve() throws java.io.ObjectStreamException {

       try{
           Class clazz = getClass();
           Field f = clazz.getField( _fieldName);
           return f.get( null);
      } catch( Exception ex){
           ex.printStackTrace();
           throw new java.io.InvalidObjectException( "Failed to resolve
          object");
      }
  }
```

```
}
```

An example of use of this Enum Pattern definition. Please, note that the use of camel case is intentional. It is due to the fact that these enum contants were defined before *"The Java Language Specification, Third Edition"* was actually published to cover enum usage and it is based on the original approach on Enum classes in Java prior to Java 5 introduction.

```
public final class JxfsScnStyleEnum extends JxfsEnum{

          public static final JxfsSCNStyleEnum standard   = new
          JxfsSCNStyleEnum( "standard");
          public static final JxfsSCNStyleEnum bold      = new
          JxfsSCNStyleEnum( "bold");
          public static final JxfsSCNStyleEnum compressed = new
          JxfsSCNStyleEnum( "compressed");
          public static final JxfsSCNStyleEnum underline  = new
          JxfsSCNStyleEnum( "underline");
          public static final JxfsSCNStyleEnum italics    = new
          JxfsSCNStyleEnum( "italics");

          private    final       String Id;
          private final    int    Ordinal;
          private static int      UpperBound = 0;

          private JxfsSCNStyleEnum( String myId){
              this.Id = myId;
              this.Ordinal= UpperBound++;
          }

          public String toString(){
              return this.Id;
          }

          public int compareTo ( JxfsSCNStyleEnum enum){
              return (Ordinal-enum.Ordinal);
          }
}
```

5.7 Persistent data

In any of the mentioned software layers of J/XFS it may be necessary to store data persistently. A Device Service might want to keep track of certain counters or other values which should be safely stored to be available the next time the system is booted. Also, the application might want to store some data persistently (e.g. the number of times a specific function was used or similar things).
The currently available standard Java programming environment does not provide a common way to store such information across all required platforms.
Thus, a generic means is defined within J/XFS to safely store / retrieve arbitrary data from a central location. By doing so, we reach the goal of being able to use different available infrastructure with all existing Device Service implementations and also prevent that every part of a J/XFS installation uses a different means to store its data.

J/XFS does provide only a basic access methods to such data as it is felt that this is sufficient for J/XFS needs. This means that the following features are NOT found in the J/XFS repository access interface:

- Elaborate, transaction-based access and rollback mechanisms.
- Mechanisms to enumerate and cycle through key sets.
- Temporary data storage (this can be done by each DS itself). The persistent data store offered here really writes any keys persistently to a disk.

To sum it up, J/XFS

- provides a simple interface for the Device Services to query the standard settings (like e.g. querying the local port the device is connected to) via the DeviceInformation objects,
- provides a dictionary with a flexible key - value access interface for persistent storage of arbitrary data,
- provides encapsulated access methods to the whole configuration information the Device Manager needs to successfully set up the J/XFS services.

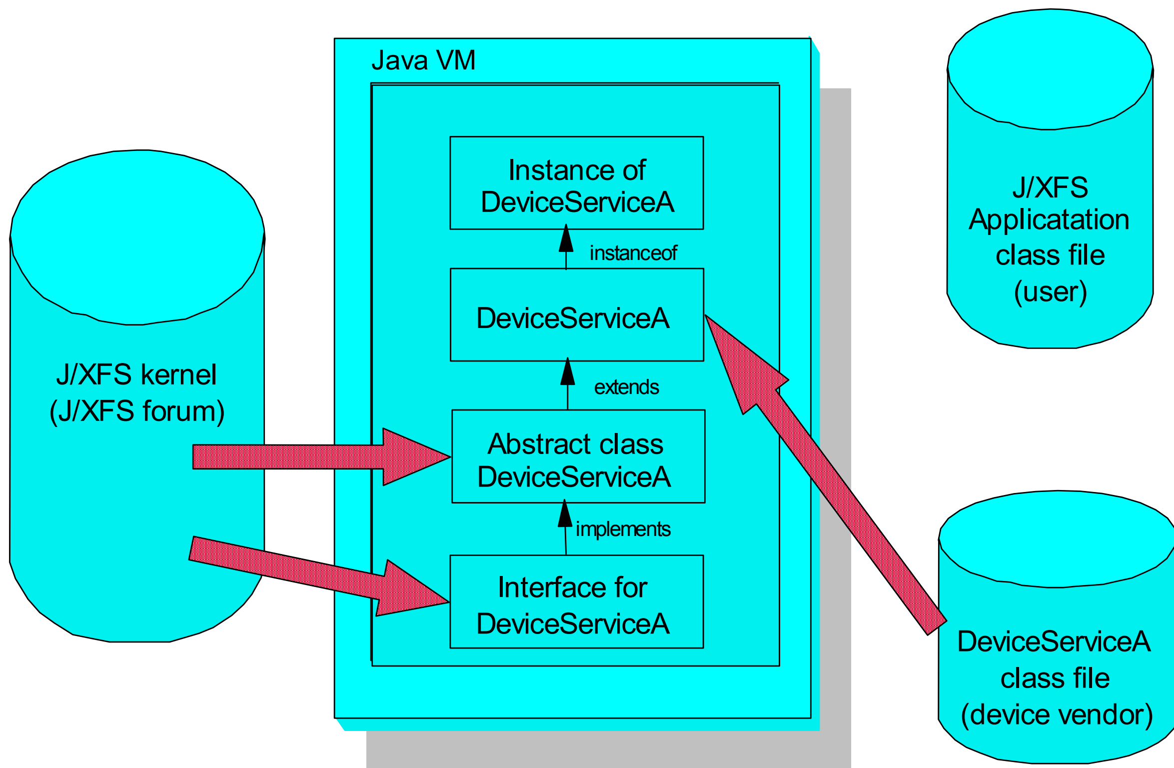
The place where the information is stored is not known to the application, it is usually a server based repository, but for certain restricted implementations it may also be the local hard disk or a local repository. In a specific implementation there may even be a storage hierarchy involved!

5.8 Version control

New versions of J/XFS should be backward compatible with previous releases. Device Services of a newer version can be successfully used in older implementations (e.g. an AcmeService of Version 1.2 can be used by a Device Control of Version 1.1).
If possible new versions of the control layer should be able to handle old version of the service classes by returning a JXFS_E_NOT_SUPPORTED from non-supported new functions if the service is not able to handle the requests.

Each Device Control and Device Service as well as the JxfsDeviceManager has a method to return its version. This is returned via a JxfsVersion object (see below).
In addition to the version numbers some description calls must be implemented, which return a string giving a description and copyright. The control layer collects these properties during startup, e.g. “ACME Magnetic Stripe Reader Device Service 1.03 (c)1999 Acme corp.”
If a class is loaded, its version number can be checked by the calling class through getXXXVersion() calls against its own version. Generally, to be usable, the major numbers must match, but the minor number may be different. It is the duty of the calling class (either a Device Control or the Device Manager) to decide whether or not the class is usable.
Build numbers should be incremented for bug fixes and minor changes and should not be needed for decision purposes.

Every time the J/XFS standard is extended by addition of methods to device control interfaces and device service interfaces, device controls and device services must implement these new methods to be usable with kernel implementations of the new standard.
Old services, which do not implement these methods are not usable with new kernel versions until the vendor of the service provides an up-to-date service implementation. To bridge this time gap between new kernel release and updated service release, kernel implementations offer abstract service classes for each device type which can be extended by device service implementations. These abstract base classes implement their corresponding device service interfaces and their implementation of new methods throw JxfsException with error code JXFS_E_NOT_SUPPORTED.
Old device service implementations extending these abstract classes inherit the new methods and can so continued to be used with new kernels. Once the vendor provides the new functionality in updated device service implementations, the service classes override the inherited methods and perform their intended function instead of throwing JxfsException with error code JXFS_E_NOT_SUPPORTED.



The names of the abstract base classes derived from the device specific interface types to bridge the gap between different version of device services are:

- AJxfsAlarmService – The abstract base class for the alarm device.
- AJxfsATMService – The abstract base class for the automatic teller machine.
- AJxfsCashDispenserService – The abstract base class for the cash dispenser.
- AJxfsCashRecyclerService – The abstract base class for the cash recycler.
- AJxfsChipCardService – The abstract base class for the chip card device.
- AJxfsMagStripeService – The abstract base class for the magnetic stripe device.
- AJxfsPINKeypadService – The abstract base class for the PIN keypad.
- AJxfsSecurePINKeypadService – The abstract base class for the secure PIN keypad.
- AJxfsDocumentPrinterService – The abstract base class for the document printer.
- AJxfsJournalPrinterService – The abstract base class for the journal printer.
- AJxfsPassbookPrinterService – The abstract base class for the passbook printer.
- AJxfsReceiptPrinterService – The abstract base class for the receipt printer.
- AJxfsBarcodeScannerService – The abstract base class for the barcode scanner.
- AJxfsChequeScannerService – The abstract base class for the cheque scanner.
- AJxfsImageScannerService – The abstract base class for the image scanner.
- AJxfsTIOService – The abstract base class for the text input/output device.
- AJxfsSiuService – The abstract base class for the sensors and indicators device.
- AJxfsCheckReaderService – The abstract base class for the check reader.
- AJxfsCAMService - The abstract base class for the camera device.
- AJxfsDepositoryService - The abstract base class for the depository device.

The protection provided by these abstract base classes is available to complex devices which have to support multiple device specific interfaces if delegation is used. See the Complex Devices subsection.

5.8.1 JxfsVersion

The .JxfsVersion object delivers version information for J/XFS. It also contains a description of the generating object. Each Device Control and Device Service has such an object which is returned in the getDeviceControlVersion() and getDeviceServiceVersion() methods. The (only) constructor for this class is **JxfsVersion(int vendorMajor, int vendorMinor, int vendorBuild, int jxfsMajor, int jxfsMinor, String description).** Usually, this is a static object within each class which needs to deliver versioning information of itself.
Additionally, the following methods can be used to query the object:

getVendorMajor

Syntax ***public int getVendorMajor()***
Description Return the major release number of the vendor’s implementation.

getVendorMinor

Syntax ***public int getVendorMinor()***
Description Return the minor release number of the vendor's implementation. The minor release number will be counted as fractions from thousand. As an example the version 1.02 will have a minor number of 20.

getVendorBuild

Syntax ***public int getVendorBuild()***
Description Return the build number of the vendor’s implementation.

getJxfsMajor

Syntax ***public int getJxfsMajor()***
Description Return the major release number of the implemented J/XFS standard.

getJxfsMinor

Syntax ***public int getJxfsMinor()***
Description Return the minor release number of the implemented J/XFS standard. The minor release number will be counted as fractions from thousand. As an example the version 1.02 will have a minor number of 20.

getDescription

Syntax ***public int getDescription()***
Description Return a more detailed description about this object which should also be suitable to be printed out. So, the format should be similar to “ACME Magnetic Stripe Reader Device Service 1.03 (c)1999 Acme corp.”