RFC 8 – Device Access

Confidential, Draft
cpeg-rfc_8_device_access-1_00RC2

34 Pages

Abstract

The Device Access architecture provides a mechanism for device drivers to cooperate in automatically detecting, attaching and detaching hardware devices. It supports hot plugging and unplugging of devices and can be configured to download and install device drivers on demand.

0 Document Information

0.1 Table of Contents

All Page Within This Box

All Page Within This Box

0.2 Status

This document specifies ***Device Access*** for the Open Services Gateway Initiative, and requests discussion and suggestions for improvements. Distribution of this document is unlimited within OSGi.

0.3 Acknowledgement

0.4 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

0.5 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial draft	January 12, 2001	Tommy Bohlin, Gatespace AB. tommy@gatespace.com
2nd draft	February 20, 2001	Tommy Bohlin, Gatespace AB. tommy@gatespace.com
3rd draft	April 1, 2001	Tommy Bohlin, Gatespace AB. tommy@gatespace.com
4th draft (RC1)	April 7, 2001	Tommy Bohlin, Gatespace AB. tommy@gatespace.com
5th draft (RC2)	April 27, 2001	Tommy Bohlin, Gatespace AB. tommy@gatespace.com

All Page Within This Box

1 Introduction

This is a proposal for review by the Core Platform Expert Group. The document is a reworked version of the OSGi 1.0 Device Access Specification. It contains a number of changes adding new functionality. Chapter 8.1 contains a summary of the proposed changes.

The Device Access architecture provides a mechanism for automatically detecting, attaching and detaching hardware devices. It supports hot plugging and unplugging of devices and can be configured to download and install device drivers on demand.

The Device Access architecture deliberately does not prescribe any particular device or network technologies. Any mentioned technologies are by example only. It also does not discuss any device discovery methods. Rather it focuses on connecting components that may be supplied by different vendors, and emphasises standardised device interfaces (although no such interfaces are defined here).

The Device Access architecture is an optional component of OSGi.

2 Motivation and Rationale

A service gateway is a natural meeting point for services and devices potentially provided by many different vendors. In many situations it will be important for service gateways to adapt dynamically to changes in their environment.

Users will add and cancel service subscriptions and expect a timely response. Newly installed services must find and connect to their input and output devices.

Devices will come and go and newer technologies such as USB explicitly support plugging and unplugging devices at any time. Wireless technologies are even more dynamic.

This makes it very hard to provision all aspects of a service gateway, particularly those relating to hardware devices. When all details are factored in each domestic service gateway may be unique. To avoid a configuration nightmare it is important to split functions into replaceable components and use standard interfaces between them. The role of the device access architecture is to connect existing such components, and where appropriate to fetch and install new components.

3 Requirements

The OSGi architecture consists of a number of components (bundles) executing in a service framework. The target environments for this architecture have a number of specific characteristics, which the device access architecture must support:

- **Embedded devices**
  OSGi systems will run in embedded devices, meaning limited possibility for user interaction. Low cost systems also have limited resources.

- **Remote administration**
  OSGi platforms must support administration by a remote service provider.

- **Vendor neutrality**
  Different vendors will supply individual components. Each component should be well defined and replaceable.

- **Long running**
  OSGi systems will be running for extended periods (months, maybe years) without restarting, this requiring stable operation and stable resource consumption.

- **Dynamic update**
  Components should as far as possible be individually replaceable without affecting unrelated components. Component updates should in particular not require restarting the whole system. In the model presented here, the device manager and driver locator bundles can be updated without disrupting the operation of any connected devices.

Additionally, the device access architecture should itself fulfil the following criteria:

- **Hot-plugging**
  The device access architecture must support plugging and unplugging of devices at any time.

- **Automatic detection**
  The device access architecture should automatically detect plugged and unplugged devices whenever the underlying hardware allows it.

- **Legacy support**
  On the other end of the scale, the device access architecture must also cope with device technologies that do not support any automatic detection.

- **Dynamic driver loading**
  The device access architecture must be capable of loading new device drivers on demand with no prior device specific knowledge.

- **Multiple device representations**
  The device access architecture must allow a device to be accessed from multiple levels of abstraction.

- **Deep trees**
  The device access should allow connection of devices in a tree of mixed network technologies of arbitrary depth.

- **Topology independence**
  The device access architecture must clearly separate the interface of a device from where and how it is attached.

- **Complex devices**
  The device access architecture must support multifunction devices and devices that have multiple configurations.

4 Definitions

The functionality provided by the device access model is cleanly separated into different components:

- **Device manager**
  The device manager "runs the show" and is part of the base Framework installation. Its required functionality is specified in this document. There is room for implementation specific optimizations. The device manager is the only privileged bundle and requires AdminPermission in order to install and uninstall drivers.
- **Driver**
  Drivers handle specific devices, networks and discovery protocols. Device manufacturers and third parties provide these.
- **DriverLocator**
  DriverLocators handle all deployment specific knowledge of how and where to remotely fetch drivers.
- **DriverSelector**
  A DriverSelector provides a method to customize the driver attachment process.
- **Device**
  Standardized Device categories provide a consistent interface to devices and allow interoperability among drivers from different vendors.

4.1 Device Manager

The **device manager** is the motor in the device access system, it coordinates the relationships between drivers so that they can present multiple representations of the same device, and drives the process of downloading new drivers.

There is exactly one device manager.

A device manager acts on certain types of services registered in Framework based on their implementation of Device, Driver, DriverLocator or DriverSelector. As a clarification to this, the service registrations must also list the said interfaces. The device manager also recognizes services containing the service property DEVICE_CATEGORY.

The device manager must track devices and drivers, and take action when they register or unregister. The device manager does not have to track driver locators and driver selectors. Registration and unregistration of these do not trigger any actions.

4.2 Driver

A **driver** is responsible for providing a representation of a physical device within the OSGi Framework. A driver does this by registering a Device service with the Framework (see below).

It can be convenient to think of drivers as belonging to one of two categories. The first category of drivers discovers physical devices using software outside of the OSGi Device Access architecture (e.g. through

notifications from a device driver in native code) and then register corresponding Device services. These are sometimes called *base drivers* since they provide the lowest-level representation of a physical device. An example of this category of driver would be a USB driver that discovers the presence of a USB device (e.g. a mouse) and registers it with the Framework as a generic USB device.

The second category of drivers provides a refined view of a physical device that is already represented by another Device service registered with the Framework. These are sometimes called *refining drivers*. They can be instructed by the device manager to attach to a Device. An example of this category would be a mouse driver that is attached to the generic USB representation of a mouse and then registers a device service that represents the physical device as a mouse. Refining drivers must register a service implementing org.osgi.service.device.Driver with the Framework. The registration must be performed synchronously within the bundle's start method. The Driver service must remain registered as long as the driver bundle is running. The device manager uses this service to control the driver.

The Driver service must have an associated property name DRIVER_ID whose value is a string identifying it. This string should start with the reversed form of the domain name of the company that implemented it (e.g. `com.acme`). The device manager uses this ID to prevent installing duplicate copies of a driver. Consequently, the DRIVER_ID should have the following properties:

- It must be independent of the location that the driver is obtained from.
- It must be independent of the DriverLocator service used to download it.
- It must be different from the ID of other drivers.
- It must be different for different revisions of the same driver.

In the case of base drivers, Devices are registered as a result of events outside the scope of the framework. In the case of refining drivers, Devices are registered as a result of events within the framework itself. Base drivers are no different from any other bundles, other than that they register device services. The pseudo base drivers discussed in 7.4 and 7.9 occupy a middle ground. They are base drivers in the sense that they don't register a Driver service, but refining in the sense that they attach to Devices.

Note: This document uses the term "Driver" (note the capital letter) to refer to a service implementing the Driver interface, while it uses the term "driver" to refer to the entity described immediately above. Similarly, it uses the term "Device service" to refer to a service representing a physical device, as defined in 4.3. The term "device" is used to describe the physical device itself.

It should be noted that any bundle may get and use a Device service without having to register a Driver service. The following functionality is offered to those bundles that do register a Driver service and play by the rules in this specification:

- On demand installation and uninstallation
- Exclusive access negotiated by the matching algorithm

4.3 Device

A **Device** is a Framework service that provides a software representation of some hardware. Devices are registered by base drivers and refining drivers, and must set the service property DEVICE_CATEGORY.

Device managers should also recognize as devices those services that include `org.osgi.service.device.Device` in the service registration, even if DEVICE_CATEGORY is not set.

In the OSGi 1.0 specification, devices were required to include the interface `org.osgi.service.device.Device`, and to set the service property DEVICE_CATEGORY. This was found

to be restrictive and in this document the requirement to include `org.osgi.service.device.Device` has been removed. Devices are still encouraged to implement `org.osgi.service.device.Device` for backwards compatibility, otherwise OSGi 1.0 conforming device managers will not recognize the devices as such.

The value of the property DEVICE_CATEGORY is an array of all device categories that the device belongs to, formatted as human readable strings. Devices may set the property DEVICE_DESCRIPTION, its value is a human readable string describing the device. If the device hardware contains a serial number the driver is encouraged to include it as the property DEVICE_SERIAL, formatted as a string. Different Devices representing the same physical hardware at different abstraction levels should set the same DEVICE_SERIAL thus simplifying identification. Devices should make every effort to set the property service.pid. It should be unique among all registered services. Even different abstraction levels of the same Device should use different pids. The pids should be reproducible, so that every time the same hardware is plugged in, the same pids are used.

Devices differ widely, some represent individual physical devices and others represent complete networks. There can even simultaneously exist several Devices representing the same physical device at different levels of abstraction. For example:

1. A USB network
2. A device attached on the USB network
3. The same device recognized as a USB to Ethernet bridge
4. A device discovered on the Ethernet using Salutation
5. The same device recognized as a simple printer
6. The same printer refined to a PostScript printer

This fine grained split into different devices allows the same set of drivers to be attached in different ways to cope with different scenarios.

4.4 DriverLocator

DriverLocators allow suitable drivers to be downloaded and installed on demand even when completely unknown devices are detected. DriverLocator services must implement org.osgi.service.device.DriverLocator. They contain all knowledge about how to remotely fetch drivers. DriverLocators do not require any privileges other than what is needed by their remote communication.

4.5 DriverSelector

DriverSelector services must implement `org.osgi.service.device.DriverSelector`. There should be at most one DriverSelector. In the event that multiple DriverSelectors are present, only the one returned by a call to `BundleContext.getServiceReference` will be used. If present, the DriverSelector is the final arbiter of which refining driver to attach to a Device, and whether to attach at all. If not present, the device manager uses a builtin selection algorithm.

4.6 Idle device

An idle Device does not have any driver attached to it. More precisely, it is a Device service (as defined in 4.3) not depended on by any bundle that is a refining driver.

Idle Devices receive special treatment by the device manager.

All Page Within This Box

4.7 Idle driver

An idle driver is a refining driver that is not currently attached to a device. More precisely, it is a bundle that has registered an org.osgi.service.device.Driver service and does not have a dependency on any Device service (as defined in 4.3).

Idle drivers loaded by the device manager are eligible for removal.

5 Detailed Operation

The device access system centers around the device manager, who is responsible for initiating all actions in response to the appearance and disappearance of Devices and Drivers. There is exactly one Device Manager. There is no interface to the device manager itself. To prevent racing conditions during framework startup, the device manager must monitor the appearance and disappearance of Devices and Drivers immediately when it is started, but the device manager must not begin refining Device services until the FrameworkEvent.STARTED event is received.

Implementation note: If the device manager is started after the framework was launched, for instance if the device manager bundle is updated, it will not receive the FrameworkEvent.STARTED event. This case is easily detected by checking the state of bundle 0 (the system bundle). If its state is ACTIVE, the framework has already been launched.

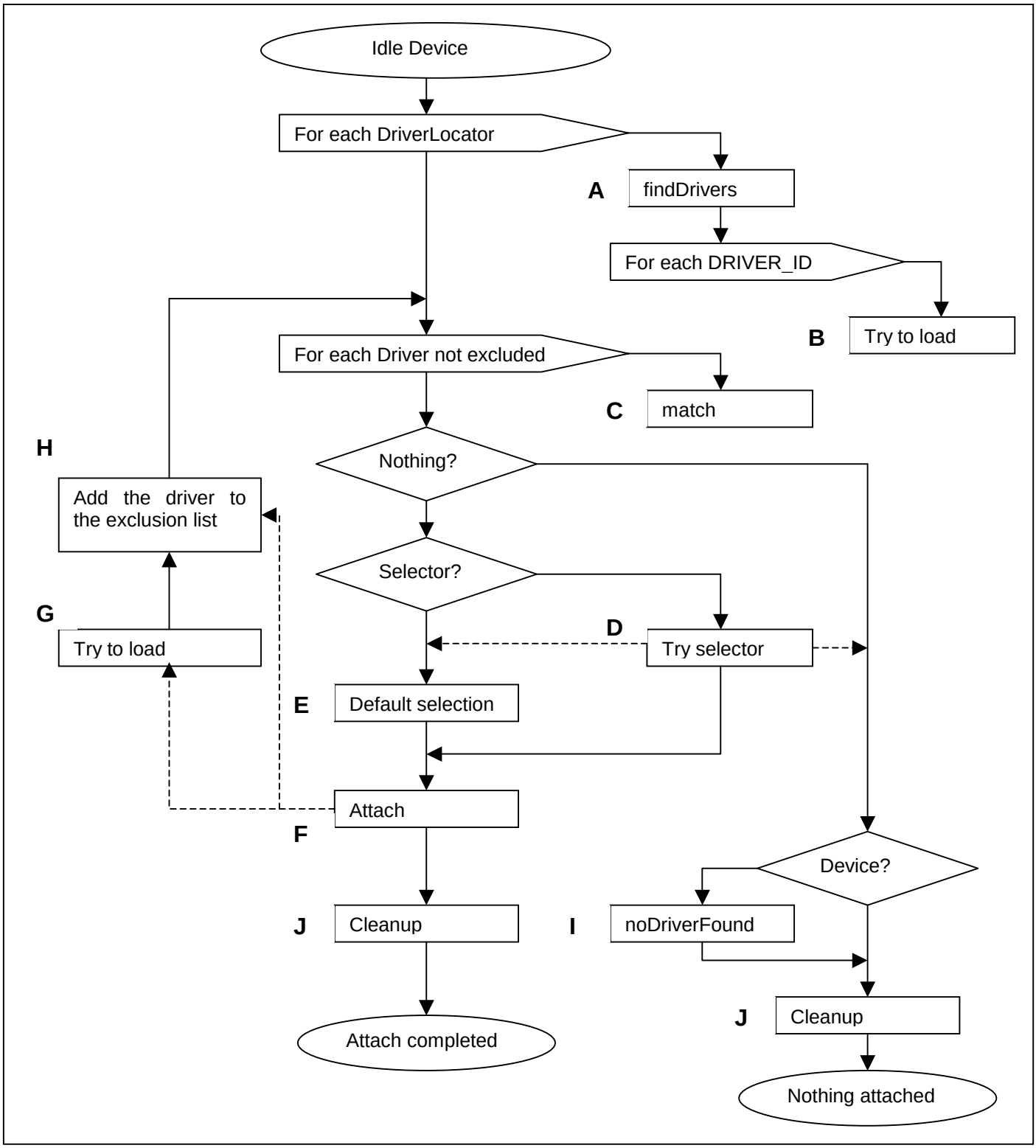
The actions undertaken by the device manager are:

- Calling the DriverLocator findDrivers and loadDriver methods
- Calling the Driver match and attach methods
- Calling the DriverSelector select method
- Calling the Device noDriverFound method
- Installing/starting drivers
- Stopping/uninstalling drivers

5.1 Driver attachment

The main role of the device manager is to attach refining drivers to idle devices. The following flow chart illustrates the attachment process:

All Page Within This Box



All Page Within This Box

Explanations to the illustration:

A
DriverLocator.findDrivers() is called for each registered DriverLocator passing the properties of the newfound Device. Each method call returns zero more DRIVER_ID values (identifiers of particular drivers). If a findDrivers() call throws an exception it is ignored and processing continues with the next DriverLocator. (See 5.3.2 for further guidance on handling exceptions)

B
For each found DRIVER_ID that does not correspond to an already registered Driver service, the device manager calls DriverLocator.loadDriver() to return an InputStream to the driver. If this succeeds, the device manager installs and starts the driver. Driver bundles are required to register their Driver service synchronously during bundle activation. If a loadDriver() call fails, that driver is ignored. Each call to loadDriver() is directed to one of the DriverLocators that mentioned the driver in step **A**.

C
For each Driver except those on the exclusion list, call its Driver.match() method passing the ServiceReference to the new Device. Collect all successful matches, i.e. return values greater than Device.MATCH_NONE in a list of active matches. A match call that throws an exception is considered unsuccessful and is not added to the list.

D
If there is a DriverSelector service, the device manager calls the DriverSelector.select() method, passing the array of active Match objects. If the DriverSelector returns one of the drivers from the array, it is selected for attachment. If the DriverSelector returns DriverSelector.NONE, no driver is attached. If the DriverSelector throws an exception or returns an invalid result, the default selection algorithm is used. Only one DriverSelector is used even if there is more than one, see 4.5.

E
The default selection algorithm works as follows: If one Driver has a match value higher than all others, it is selected. Otherwise the active list is reduced to contain only the highest match value. If one of the remaining Drivers has a higher service ranking than all others, it is selected. Otherwise the active list is further reduced to contain only the highest ranking. Finally the selected Driver is the one with the lowest service id.

F
The selected driver's attach method is called. If it returns null, attach succeeded. If it returns a String it is interpreted as a referral to another Driver and processing continues at **G**. If an exception is thrown, the driver has failed and the algorithm proceeds to try another driver after excluding this one from further consideration at **H**.

G
The device manager attempts to load the referred driver in a manner similar to step **B**, the difference being that we do not know which DriverLocator to use. Therefore loadDriver must be called on each DriverLocator until one succeeds (or they all fail). If this succeeds, the device manager installs and starts the driver.

H
The referring driver is added to the exclusion list. Because each new referral adds an entry to the exclusion list, which in turn disqualifies another driver from further matching, the algorithm cannot loop. The list is maintained for the duration of this algorithm. The next time a new Device is processed, the list starts out empty.

All Page Within This Box

I
If no refining driver was attached, the Device is checked to see whether it implements org.osgi.service.device.Device. If so, the noDriverFound() method is called. Note that this may cause the Device to unregister and possibly new Device(s) to be registered in its place. Each new Device will restart the algorithm from the beginning.

J
Whether an attachment was successful or not, the algorithm may have installed a number of Drivers. The device manager is entitled to but not obliged to remove any drivers it has installed that remain idle.

5.2 Tracking services

5.2.1 Device manager startup

When the device manager is started, it must collect all idle devices and apply algorithm 5.1 to each idle device.

5.2.2 Device registration

When a new device is registered, algorithm 5.1 must be applied to that device.

5.2.3 Device unregistration

Device unregistration does not trigger any immediate device manager action. The normal service unregistering events provided by the framework take care of propagating the fact to any other affected drivers. The device manager may however take a device unregistration as a hint that drivers may have become idle and thus eligible for removal.

5.2.4 Driver registration

When a new driver is registered, algorithm 5.1 must be applied to each idle device. This gives the new driver a chance to compete with other drivers for attaching to idle devices. The optimizations outlined in 5.3.2 can provide significant shortcuts for this case.

This specification does not define a method for new Drivers to “steal” already attached Devices. This may be addressed in a future update.

5.2.5 Driver unregistration

When a Driver unregisters, algorithm 5.1 must be applied to each idle device. This gives others drivers a chance to take over the refinement of devices after the unregistering driver. The optimizations outlined in 5.3.2 can provide significant shortcuts for this case.

5.3 Notes

5.3.1 Simultaneous device and driver registration

Algorithm 5.1 may cause drivers to be installed, which according to 5.2.4 requires executing 5.1 recursively. Appearance of the new drivers should be noted but processing delayed until after completion of the algorithm. I.e. there should never be more than one simultaneous 5.1 in progress.

The following example illustrates this process:

1. A driver is registered.
2. Collect the set of all idle devices.
3. Apply algorithm 5.1 to each device in the set.
4. If no drivers were registered during the executions of 5.1 processing terminates.
5. Otherwise collect the set of all currently idle devices, and go to step 3.

5.3.2 Optimizations

Driver match values and referrals must be deterministic. I.e. repeated calls with arguments representing the same devices must return the same results. It is permissible for a device manager to cache match values and referrals. This can save considerable work. Short cuts in the above algorithms based on this assumption are allowed.

It is permissible for the device manager to delay loading a driver until it is needed.

The results of DriverLocator and DriverSelector method calls are not required to be deterministic, and may not be cached by the device manager.

Thrown exceptions may also not be cached. Exceptions are considered transient failures, and the device manager must always retry a method call even if it has thrown an exception on a previous invocation with the same arguments.

5.3.3 Driver cleanup

The device manager may remove drivers it has installed at any time provided they are idle. Doing so prevents unused drivers from accumulating over time. On the other hand, removing drivers too soon may cause unnecessary downloads and associated delays when drivers are needed again. Because of the diverse range of service gateway applications and their different characteristics this specification can provide no further guidance on driver reclamation.

Note: If a device manager implements driver reclamation, the algorithms are not guaranteed to terminate unless the device manager supports some level of optimization. For example: A new Device triggers the attachment algorithm. A driver recommended by a DriverLocator is loaded. It doesn't match, so the device remains idle. The device manager is eager to reclaim space and unloads the driver. The disappearance of the Driver causes the device manager to reattach idle Devices. Not keeping record of its previous activities, it tries to reattach the same Device which closes the loop.

On systems where the device manager implements driver reclamation, all refining drivers should be loaded through DriverLocators. This is to prevent the device manager from erroneously uninstalling preinstalled drivers that cannot later be reinstalled when needed. The device manager can be updated or restarted and according to

5.3.5, it cannot rely on previously stored information to determine which drivers were preinstalled and which were dynamically installed and thus eligible for removal.

5.3.4 Driver update

A Driver service must remain registered as long as the driver bundle is active. Thus a Driver service should only be unregistered if the driver bundle is stopping which may prelude being uninstalled or updated. It is difficult to determine whether the driver is being updated when the unregister event for the Driver service is received. In order to facilitate this distinction, the device manager should wait a short period of time after the unregistration for one of the following events:

- A BundleEvent.UNINSTALLED event for the driver bundle
- A ServiceEvent.REGISTERED event for another Driver service registered by the driver bundle

If the driver bundle is uninstalled or if neither of the above events are received within the allotted time period, the driver should be assumed to be inactive. The appropriate waiting period will be implementation dependent and will vary for different installations. As a general rule it should be long enough to allow a driver to be stopped, updated, and restarted under normal conditions, and short enough not to cause unnecessary delays in reattaching devices.

5.3.5 Device manager persistence

The device manager may persistently store information, but must be able to cold start without any persistent information and still be able to manage an existing connection state, satisfying all of the requirements above.

5.3.6 Driver permissions

Before a Driver bundle provided by a DriverLocator and installed by the device manager can be started, it may require a set of permissions. The device manager itself should be free from any knowledge of policies and should not actively set bundle permissions. If permissions must be set, it is up to management bundles to listen to synchronous bundle events and set the appropriate permissions.

6 Device categories

In order to ensure exchangeability among drivers for similar devices, it is essential for OSGi to define a number of device categories. All Device services in a defined category must conform to the approved interface. This RFC does not set out to define any specific device categories, only to lay out guidelines for how to define them.

A device category definition comprises the following elements:

- A interface or class that any device belonging to this category must implement. OSGi may define its own device interfaces (or classes) or leverage existing ones. When registering a device belonging to this category with the framework, the specified class names must always include the name of this interface (or class).
- A set of service registration properties (and their data types), each of which must be declared as either MANDATORY or OPTIONAL for this device category.
- A range of match values specific to this device category.

The following is an incomplete example of a device category:

6.1 Device Interface Example

```
interface foo.bar.USBDevice {
  int MATCH_CLASS_MAKE_MODEL_REV_SERIAL = 6;
  int MATCH_CLASS_MAKE_MODEL_REV = 5;
  int MATCH_CLASS_MAKE_MODEL = 4;
  int MATCH_CLASS_MAKE = 3;
  int MATCH_CLASS = 2;
  int MATCH_GENERIC = 1;

  void sendPacket(Pipe pipe, Data data);
  Data receivePacket(Pipe pipe);
  …
}
```

Devices in this category must implement the interface `foo.bar.USBDevice`.

All Page Within This Box

6.2 Service Properties Example

Devices properties for this category are:

```
DEVICE_CATEGORY       mandatory = “USBDevice”
foo.bar.usb.class     mandatory
foo.bar.usb.make      mandatory
foo.bar.usb.model     mandatory
foo.bar.usb.revision  optional
foo.bar.usb.serial    optional
```

6.3 Match Range Example

Drivers for Devices of this fictitious device category must return one of the above match codes or `Device.MATCH_NONE`. The exact rules for the match codes must be defined:

- `MATCH_CLASS_MAKE_MODEL_REV_SERIAL`
  The driver must match the hardware device down to the exact serial number
- `MATCH_CLASS_MAKE_MODEL_REV`
  The driver matches the hardware down to the revision but not serial number
- `MATCH_CLASS_MAKE_MODEL`
  The driver matches the hardware model but not revision
- `MATCH_CLASS_MAKE`
  The driver is for another product by this manufacturer but can do something useful with the hardware
- `MATCH_CLASS`
  The driver is for products of this general class
- `MATCH_GENERIC`
  The driver is for USB devices in general and is marginally useful
- `MATCH_NONE`
  The driver is useless for this hardware

All Page Within This Box

7 Example Drivers

The following examples illustrate some useful driver “design patterns” supported by the device access model.

7.1 Base driver with discovery

A USB driver that registers a generic USBDevice service for each discovered physical device. USB defines a tightly integrated discovery method, devices are individually addressed and there is no provision for broadcasting messages. Therefore there is no reason to expose the USB network itself, instead the driver registers individual devices as they are discovered.

7.2 Base driver without discovery

Most serial devices do not support detection, and it is often not even possible to electrically detect whether a device is attached to a serial port (there is a plug and play standard for serial ports, but few actual devices support it). Therefore, although the device access architecture expects each driver to perform discovery a serial port driver cannot really do this on its own. The solution is for the driver to get outside help, either by presenting a user interface or by allowing a configuration manager to control it. In response to information received from outside, the driver can register and unregister serial Devices with appropriate properties. To the device manager and the other drivers it appears as though the serial port driver is actually discovering devices on its own.

It is quite possible for the driver to combine automatic discovery of plug and play compliant devices with manual configuration when non-compliant devices are plugged in.

7.3 Network driver

An IP (Internet Protocol) capable network such as Ethernet supports individually adressable devices, allows broadcasts but does not define an intrinsic discovery protocol. It could make sense to expose the entire network as a Device.

7.4 Pure discovery driver

UPnP, Salutation and JINI are all based on IP and contain their own discovery protocols. They can each run on any medium capable of transporting IP packets. It makes very much sense to make these into separate drivers. That way a single UPnP driver is sufficient to support Ethernet, IEEE 802.11, IEEE 1394, Token ring, and others.

It is also possible to concurrently run UPnP, Salutation and JINI on the same network without conflicting. Each one discovers its own type of devices. The device manager will however only attach one driver to an IPDevice. To overcome this, the discovery bundles can be written as pseudo base drivers, drivers that look like base drivers in that they don’t register a Driver service and consequently are not attached. Instead they actively listen and get Devices by themselves.

All Page Within This Box

An example of a refining discovery driver is 1-Wire discovery. There is only one discovery method used in 1-Wire, and it could be handled directly in the 1-Wire network base drivers. However, the discovery protocol is not so tightly integrated and can easily be shared among all base drivers. In this case it makes sense to separate the discovery bit even though it is in a sense part of the network.

7.5 Refining driver

The majority of specific device drivers fall into this category. An example is a printer driver. It refines a parallel port device into a printer.

Regarding the interface of the registered printer service, there are conflicting goals. On the one hand it is desirable to make the printer interface as comprehensive as possible allowing applications to extract the most out of the printer. On the other hand the printer interface should be generic so the parallel printer can be replaced by a USB printer from another vendor without affecting any applications.

One way to reconcile these goals is to make the service implement a simple generic Printer interface, as well as a ParallelPrinter interface adding methods common to parallel port printers, and a HP710CPrinter interface adding methods for this particular printer model. This way it is up to each application to decide whether to go for generality or features.

7.6 Composite driver

Complex devices can often be decomposed into several parts. For example a USB speaker containing software accessible buttons can be registered by its driver as two separate Devices, an AudioDevice and a ButtonDevice. This approach can strongly reduce the number of interfaces needed as well as enhance compatibility among devices. After all, once it is decomposed, a speaker is a speaker whether it has buttons or not.

7.7 Matching driver

The matching algorithm in **Error! Reference source not found.** mentions referring drivers. At first glance it may seem strange, but is actually a powerful concept. Vendor X can for example implement one driver that specializes in recognizing all devices produced by them. Because the driver doesn't actually contain any driving code, only sufficient logic to recognize the devices, the driver can be small yet identify a large product line. This can drastically reduce the amount of downloading and matching needed to find the correct driver.

This is even more powerful if the matching drivers themselves are fetched dynamically. Vendors A-Z manufacture hundreds of devices each and have a driver for each one. They also have a matching driver each. A DriverLocator is installed whose findDriver() method decodes the manufacturer from the service properties and returns the DRIVER_ID to that manufacturer's matching driver.

When a new device from manufacturer P is plugged in, the DriverLocator points out P's matching driver. The device manager loads it, matches and attaches. Attach returns the DRIVER_ID to the real driver. The device manager loads it, matches and attaches. This attach succeeds. The correct driver was found among thousands of other drivers in only two steps.

7.8 Bridging driver

The most obvious stacking order of networked Devices places a network at the bottom, individual generic devices above that, then one or more refined Device levels on top. It could turn out that such a device is a bridge into another network. There are for example USB to Ethernet bridges that allow connection to an Ethernet through a

USB device. In this case the top level of the “USB part” of the Device stack would be an EthernetDevice. But the same EthernetDevice can also be the bottom layer of an “Ethernet part” of the Device stack. A few layers up there could be a bridge into yet another network.

There is no limit to the stacking depth of Devices, and the same drivers could in fact appear at different levels in the same Device stack. The graph of drivers/Devices roughly mirrors the hardware connections.

7.9 Multiplexing driver

In some cases it is desirable to collect a number of devices into one. As an example it could be nice if all attached keyboards and mice acted the same. To spare each application from having to get all mice and concatenating the inputs, a multiplexing mouse driver can provide that service. It registers one Mouse itself regardless of how many other mice it finds. The mouse multiplexer can be written either as a refining driver, or as a pseudo base driver that gets and listens to mice on its own without involving the device manager.

7.10 Pure consuming driver

It is possible for a driver to attach to devices without itself registering a refined version. One could for example decide to handle all serial ports through javax.comm. When a USB serial port is plugged in, a series of drivers are attached and a Device stack forms, resulting in a SerialPortDevice. On top of this a driver could attach to the SerialPortDevice and register a new serial port in the javax.comm.* registry instead. This effectively exports the device from Framework into another environment.

Another example could be a driver that picks up certain types of Devices and makes them available outside of the Framework through the UPnP protocol.

8 Status

8.1 Changes since OSGi 1.0

The OSGi 1.0 specification contained a device access specification but no corresponding RFC. This RFC is very close to the 1.0 device access specification, but contains a number of changes. Below is a summary of the changes.

.

8.1.1 Match disambiguation

According to the 1.0 specification, if the matching process results in a tie, the device manager is free to choose any one of the highest bidders. The choice need not be consistent if faced with the same situation again. This kind of randomness may surprise a user and also makes automated testing difficult. To avoid this, the selection now also involves the service ranking and service ids.

8.1.2 Constants interface

The Constants interface has been added. It contains constants for standard service property keys.

8.1.3 DriverSelector

A DriverSelector interface has been added. It allows complete customization of the selection process.

The device manager by default eagerly attaches all the drivers it can with no regard to whether the resulting refined devices are wanted or not. This could lead to installing and starting many unnecessary drivers. The DriverSelector also provides a way to limit the eagerness.

8.1.4 Match interface

The Match interface has been added. It is used in the second argument to DriverSelector.select().

8.1.5 Tracking drivers

The 1.0 specification prescribed that device managers only should track the appearance of Devices. This causes a problem if resident drivers are started in the “wrong” order. If a low level driver registers a device before the higher level driver is started, the device manager could conclude that there is no refining driver available, and leave the device unattached.

This has been improved in two ways. First, the device manager does not start until the framework is launched. Second, the device manager now listens to Driver registrations and unregistrations.

8.1.6 Free format devices

The 1.0 specification defines devices as services that implement Device. This is unfortunately intrusive in that it rules out existing interfaces that were not designed with this in mind. This has been improved be recognizing any service registered with the DEVICE_CATEGORY property as a Device.

8.2 Unresolved problems

8.2.1 Unreliable driver update detection

It is hard to reliably detect if a driver is being updated, stopped or uninstalled. The procedure suggested in 5.3.4 involves a timer.

8.3 Future Enhancements

8.3.1 Dynamic installation of pseudo base drivers

The device manager currently does not distinguish between pseudo base drivers and other base drivers. In particular, base drivers cannot be dynamically installed and uninstalled. It would be useful to support dynamic installation of pseudo base drivers in response to discovery of new network types.

8.3.2 Notification of new drivers from DriverLocator

The updated device manager algorithm now reacts to new drivers. But it does not itself update the drivers and there is no notification support in DriverLocator.

8.3.3 Support for stealing devices

There is no support for “stealing” a Device and giving it to a better driver. Should a detach method be added to Driver to handle this

8.3.4 Device locking

There is a difference between attaching to a lower level Device and actually using it. Many Devices should support exclusive access in their APIs. This should be discussed. The issue is not specific to devices and should probably be addressed in a wider context.

8.3.5 Named match values

For logging and debugging purposes it should be possible to provide textual descriptions of match values.

9 API Specification

9.1 org.osgi.service.device Interface Constants

public interface **Constants**

This interface defines standard names for property *keys* associated with Device and Driver services.

The *values* associated with these keys are of type `java.lang.String`, unless otherwise stated.

Field Summary

static java.lang.String	**`DEVICE_CATEGORY`** Property (named "DEVICE_CATEGORY") containing a human readable description of the device categories implemented by a device.
static java.lang.String	**`DEVICE_SERIAL`** Property (named "DEVICE_SERIAL") specifying a device's serial number.
static java.lang.String	**`DRIVER_ID`** Property (named "DRIVER_ID") identifying a driver.

Field Detail

9.1.1 DRIVER_ID

public static final java.lang.String **`DRIVER_ID`**

This property (named "DRIVER_ID") identifies a driver.

A `DRIVER_ID` should start with the reversed domain name of the company that implemented the driver (e.g., `com.acme`), and must meet the following requirements:

- It must be independent of the location from where it is obtained.
- It must be independent of the DriverLocator service that downloaded it.
- It must be unique.
- It must be different for different revisions of the same driver.

This property is mandatory, i.e., every `Driver` service must be registered with it.

All Page Within This Box

9.1.2 DEVICE_CATEGORY

public static final java.lang.String **DEVICE_CATEGORY**
This property (named "DEVICE_CATEGORY") contains a human readable description of the device categories implemented by a device.

Services registered with this property will be treated as devices and discovered by the device manager.

9.1.3 DEVICE_SERIAL

public static final java.lang.String **DEVICE_SERIAL**
This property (named "DEVICE_SERIAL") specifies a device's serial number.

9.2 org.osgi.service.device
Interface Device

---

public interface **Device**

Interface for identifying device services.

A service must implement this interface or use the Constants.Constants.DEVICE_CATEGORY registration property to indicate that it is a device. The device manager will discover a service if it is registered as implementing this interface or has the DEVICE_CATEGORY property set.

Device services implementing this interface give the device manager the opportunity to indicate to the device that no drivers were found that could (further) refine it. In this case, the device manager calls the noDriverFound() method on the device.

Specialized device implementations will extend this interface by adding methods appropriate to their device category to it.

Field Summary

static int	**MATCH_NONE** Return value from Driver.Driver.match(org.osgi.framework.ServiceReference), indicating that the driver cannot refine the device presented to it by the device manager.

Method Summary

void	**noDriverFound**() Indicates to this Device that the device manager has failed to attach any drivers to it.

Field Detail

All Page Within This Box

9.2.1 MATCH_NONE

public static final int **MATCH_NONE** = 0

Return value from Driver.Driver.match(org.osgi.framework.ServiceReference), indicating that the driver cannot refine the device presented to it by the device manager.

Method Detail

9.2.2 noDriverFound

public void **noDriverFound**()

Indicates to this Device that the device manager has failed to attach any drivers to it.

If this Device can be configured differently, the driver that registered this Device may unregister it and register a different device service instead.

9.3 org.osgi.service.device Interface Driver

public interface **Driver**

Each driver that wishes to attach to device services provided by other drivers must register a `Driver` service. For each newly discovered Device, the device manager enters a bidding phase. The `Driver` whose match(org.osgi.framework.ServiceReference) method bids the highest for a particular device will be instructed by the device manager to attach to the device.

Method Summary

java.lang.String	**`attach`**(org.osgi.framework.ServiceReference reference) Attaches this driver to the device represented by the given ServiceReference.
int	**`match`**(org.osgi.framework.ServiceReference reference) Checks whether this driver can be attached to the device represented by the given ServiceReference, and returns a value indicating how well this driver can support the given device, or Device.Device.MATCH_NONE if it cannot support the given device at all.

Method Detail

9.3.1 match

public int **match**(org.osgi.framework.ServiceReference reference)
throws java.lang.Exception

All Page Within This Box

Checks whether this driver can be attached to the device represented by the given ServiceReference, and returns a value indicating how well this driver can support the given device, or Device.Device.MATCH_NONE if it cannot support the given device at all.

The return value must be one of the possible match values defined in the device category definition for the given device, or `Device.MATCH_NONE` if the category of the device is not recognized.

In order to make its decision, this driver may simply examine the properties associated with the given device, or may get the referenced service object (representing the actual physical device) to talk to it, as long as it ungets the service and returns the physical device to a normal state before this method returns.

A driver must always return the same match code whenever it is presented with the same device.

The device manager calls the match method during the matching process.

Parameters:
reference - the ServiceReference of the device to match
Returns:
value indicating how well this driver can support the given device, or Device.MATCH_NONE if it cannot support the device at all
Throws:
java.lang.Exception - if the driver cannot examine the device

9.3.2 attach

public java.lang.String **attach**(org.osgi.framework.ServiceReference reference)
throws java.lang.Exception

Attaches this driver to the device represented by the given ServiceReference.

A return value of `null` indicates that this driver has successfully attached to the given device. If this driver is unable to attach to the given device, but knows of a more suitable driver, it must return the ID of that driver. This allows for the implementation of referring drivers whose only purpose is to refer to other drivers capable of handling a given device.

After having attached to the device, this driver may register the device as a new service exposing driver-specific functionality.

The device manager calls this method.

Parameters:
reference - the ServiceReference of the device to attach to
Returns:
`null` if this driver has successfully attached to the given device, or the ID of a more suitable driver
Throws:
java.lang.Exception - if the driver cannot attach to the given device and does not know of a more suitable driver

9.4 org.osgi.service.device Interface DriverLocator

public interface **DriverLocator**

A DriverLocator can find and load device driver bundles given a property set. A unique ID represents each driver.

All Page Within This Box

DriverLocator services provide the mechanism for dynamically downloading new device driver bundles into an OSGi device. They are supplied by OSGi providers and encapsulate all provider-specific details related to the location and acquisition of device driver bundles.

Method Summary

java.lang.String[]	**findDrivers**(java.util.Dictionary props) Returns an array of driver IDs of drivers capable of attaching to a device with the given properties.
java.io.InputStream	**loadDriver**(java.lang.String id) Get an InputStream from which the driver bundle providing a driver with the giving ID can be installed.

Method Detail

9.4.1 findDrivers

public java.lang.String[] **findDrivers**(java.util.Dictionary props)

Returns an array of driver IDs of drivers capable of attaching to a device with the given properties.

The property keys in the specified Dictionary are case-insensitive.

Parameters:
props - the properties of the device for which a driver is sought
Returns:
array of driver IDs of drivers capable of attaching to a device with the given properties, or null if this DriverLocator does not know of any such drivers

9.4.2 loadDriver

public java.io.InputStream **loadDriver**(java.lang.String id)
throws java.io.IOException

Get an InputStream from which the driver bundle providing a driver with the giving ID can be installed.
Parameters:
id - the ID of the driver that needs to be installed.
Returns:
the InputStream from which the driver bundle can be installed
Throws:
java.io.IOException - the input stream for the bundle cannot be created

9.5 org.osgi.service.device
Interface DriverSelector

public interface **DriverSelector**

All Page Within This Box

When the Device Manager detects a new device, it calls all registered drivers to determine if anyone matches the device. If at least one driver matches, the Device Manager must choose one. If there is a DriverSelector service registered in the framework, the Device Manager will ask it to make the selection. If there is no DriverSelector, or if it returns an invalid result, or throws an exception, the device manager uses the default selection strategy.

Field Summary

static int	**SELECT_NONE** Return value from DriverSelector.select, if no driver should be attached to the device.

Method Summary

int	**select**(org.osgi.framework.ServiceReference reference, Match[] matches) Select one of the matching drivers.

Field Detail

9.5.1 SELECT_NONE

public static final int **SELECT_NONE** = -1

Return value from DriverSelector.select, if no driver should be attached to the device.

Method Detail

9.5.2 select

public int **select**(org.osgi.framework.ServiceReference reference,
Match[] matches)

Select one of the matching drivers. The device manager calls this method if there is at least one driver bidding for a device. Only drivers that have responded with nonzero (not Device.MATCH_NONE) match values will be included in the list.

Parameters:
reference - the ServiceReference of the device
matches - the array of all non-zero matches
Returns:
index into the array of Matches, or SELECT_NONE if no driver should be attached

9.6 org.osgi.service.device Interface Match

org.osgi.service.device.Match

public interface **Match**

All Page Within This Box

Instances of Match are used in the DriverSelector.select() method to identify drivers matching a device.

Method Summary	
org.osgi.framework.ServiceReference	**getDriver()** ServiceReference to a driver.
int	**getMatchValue()** The match value returned by this driver.

Method Detail

9.6.1 getDriver

public org.osgi.framework.ServiceReference **getDriver()**

Returns:
Service Reference to this driver

9.6.2 getMatchValue

public int **getMatchValue()**

Returns:
The match value returned by this driver

10 Security Considerations

10.1 Device manager

The device manager requires AdminPermission to install and uninstall drivers. This gives it full framework permissions, so it must be trusted.

10.2 DriverLocator

A bad `DriverLocator` could feed a bad input stream to the device manager, thus causing it to install a malevolent bundle. The `DriverLocator` interface should be protected by `ServicePermissions` and allow only genuine driver locators to register as such.

10.3 DriverSelector

A bad `DriverSelector` could fool the device manager into selecting the wrong driver or disable driver attachment altogether. The `DriverSelector` interface should be protected by `ServicePermissions` and allow only genuine driver selectors to register as such.

10.4 Driver

The Driver interface should be protected by ServicePermissions and allow only the device manager to get Drivers. This prevents drivers from being fooled by a rogue device manager.

Ill behaved drivers could attach to devices without being told to. Drivers could engage in unfair bidding. Dynamic installation of drivers should be done with care.

If a driver unregisters it's Driver service without being stopped it could cause a delay in the device manager, see 5.3.4.

All Page Within This Box

Document Support

10.5 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

10.6 Author's Address

Name	Tommy Bohlin
Company	Gatespace AB
Address	Stora Badhusgatan 18-20 SE-411 21 Göteborg, Sweden
Voice	+46 31 7439815
e-mail	tommy@gatespace.com

10.7 Acronyms and Abbreviations

10.8 End of Document

All Page Within This Box